An Advanced User's Guide to Stat-JR version 1.0.1

Programming and Documentation by

William J. Browne, Christopher M.J. Charlton, Danius T. Michaelides*, Richard M.A. Parker, Bruce Cameron, Camille Szmaragd, Huanjia Yang*, Zhengzheng Zhang, Harvey Goldstein, Kelvyn Jones, George Leckie and Luc Moreau*

Centre for Multilevel Modelling,

University of Bristol.

*Electronics and Computer Science,

University of Southampton.

March 2014

An Advanced User's Guide to Stat-JR version 1.0.1

ISBN: To be confirmed

Printed in the United Kingdom

Contents

1. About Stat-JR 1

1.1 Stat-JR: software for scaling statistical heights. 1

1.2 About the Advanced User's Guide 2

2 Installation instructions 3

3 A simple regression template example 4

3.1 Running a first template 4

3.2 Opening the bonnet and looking at the code 8

3.2.1 Inputs 10

3.2.2 Model 11

3.2.3 Latex 13

3.2.4 Some points to note 14

3.3 Writing your own first template 15

4 Running templates with the eStat engine 15

4.1 Algebra and Code Generation 15

4.2 The algebraic software system 22

5 Including Interoperability 25

5.1 eStat.py 26

5.2 Regression2.py 27

5.3 WinBUGS and Winbugsscript.py 27

5.4 MLwiN 31

5.5 R 36

5.6 Other packages 42

6 Input, Data manipulation and output templates 43

6.1 Generate template (generate.py) 43

6.2 Recode template (recode.py) 47

6.3 AverageAndCorrelation template 49

6.4 XYPlot template 52

7 Single level models of all flavours – A logistic regression example 54

7.1 Inputs 56

7.2 Engines 57

7.3 Model 58

7.4 Latex 59

8 Including categorical predictors 60

9 Multilevel models 64

9.1 2LevelMod template 64

9.2 NLevelMod template 68

10 Using the Preccode method 74

10.1 The 1LevelProbitRegression template 74

10.3 preccode and deviancecode attributes 77

11 Multilevel models with Random slopes and the inclusion of Wishart priors 80

11.1 An example with random slopes 80

11.2 Preccode for NLevelRS 84

12 Improving mixing (1LevelBlock and 1LevelOrthogParam) 87

12.1 Rats example 87

12.2 The 1LevelBlock template 89

12.3 The 1LevelOrthogParam template 91

12.4 Multivariate Normal response models 95

12.5 The preccode function for this template 98

13 Out of sample predictions 103

13.1 The 1LevelOutSampPred template – using the zxfd trick 104

14 Solutions to the exercises 106

References 107

Acknowledgements

The Stat-JR software is very much a team effort and is the result of work funded under three ESRC grants: the LEMMA 2 and LEMMA 3 programme nodes (Grant: RES-576-25-0003 & Grant:RES-576-25-0032) as part of the National Centre for Research Methods programme, and the e-STAT node (Grant: RES-149-25-1084) as part of the Digital Social Research programme. We are therefore grateful to the ESRC for financial support to allow us to produce this software.

All nodes have many staff that, for brevity, we have not included in the list on the cover. We acknowledge therefore the contributions of:

Fiona Steele, Rebecca Pillinger, Paul Clarke, Mark Lyons-Amos, Liz Washbrook, Sophie Pollard, Robert French, Nikki Hicks, Mary Takahama and Hilary Browne from the LEMMA nodes at the Centre for Multilevel Modelling.

David De Roure, Tao Guan, Alex Fraser, Toni Price, Mac McDonald, Ian Plewis, Mark Tranmer, Pierre Walthery, Paul Lambert, Emma Housley, Kristina Lupton and Antonina Timofejeva from the e-STAT node.

A final acknowledgement to Jon Rasbash who was instrumental in the concept and initial work of this project. We miss you and hope that the finished product is worthy of your initials.

WJB November 2013.

1. About Stat-JR

1.1 Stat-JR: software for scaling statistical heights.

The use of statistical modelling by researchers in all disciplines is growing in prominence. There is an increase in the availability and complexity of data sources, and an increase in the sophistication of statistical methods that can be used. For the novice practitioner of statistical modelling it can seem like you are stuck at the bottom of a mountain, and current statistical software allows you to progress slowly up certain specific paths depending on the software used. Our aim in the Stat-JR package is to assist practitioners in making their initial steps up the mountain, but also to cater for more advanced practitioners who have already journeyed high up the path, but want to assist their novice colleagues in making their ascent as well.

One issue with complex statistical modelling is that using the latest techniques can involve having to learn new pieces of software. This is a little like taking a particular path up a mountain with one piece of software, spotting a nearby area of interest on the mountainside (e.g. a different type of statistical model), and then having to descend again and take another path, with another piece of software, all the way up again to eventually get there, when ideally you'd just jump across! In Stat-JR we aim to circumvent this problem via our interoperability features so that the same user interface can sit on top of several software packages thus removing the need to learn multiple packages. To aid understanding, the interface will allow the curious user to look at the syntax files for each package to learn directly how each package fits their specific problem.

To complete the picture, the final group of users to be targeted by Stat-JR is the statistical algorithm writers. These individuals are experts at creating new algorithms for fitting new models, or better algorithms for existing models, and can be viewed as sitting high on the peaks with limited links to the applied researchers who might benefit from their expertise. Stat-JR will build links by incorporating tools to allow this group to connect their algorithmic code to the interface through template-writing, and hence allow it to be exposed to practitioners. They can also share their code with other algorithm developers, and compare their algorithms with other algorithms for the same problem. A template is a pre-specified form that has to be completed for each task: some run models, others plot graphs, or provide summary statistics; we supply a number of commonly used templates and advanced users can use their own – see the Advanced User's Guide. It is the use of templates that allows a building block, modular approach to analysis and model specification.

At the outset it is worth stressing that there a number of other features of the software that should persuade you to adopt it, in addition to interoperability. The first is flexibility – it is possible to fit a very large and growing number of different types of model. Second, we have paid particular attention to speed of estimation and therefore in comparison tests, we have found that the package compares well with alternatives. Third it is possible to embed the software's templates inside an e-book which is exceedingly helpful for training and learning, and also for replication. Fourth, it provides a very powerful, yet easy to use environment for accessing state-of-the-art Markov Chain Monte Carlo procedures for calculating model estimates and functions of model estimates, via eStat

engine. The eStat engine is a newly-developed estimation engine with the advantage of being transparent in that all the algebra, and even the program code, is available for inspection.
While this is a beginner's guide – it is a beginner's guide to the software. We presume that you have a good understanding of statistical models which can be gained from for example the LEMMA online course (http://www.bristol.ac.uk/cmm/learning/online-course/index.html) . It also pre-supposes familiarity with MCMC estimation and Bayesian modelling – the early chapters of Browne (2012) available at http://www.bristol.ac.uk/cmm/software/mlwin/download/2-26/mcmc-web.pdf provide a practical introduction to this material.

Many of the ideas within the Stat-JR system were the brainchild of Jon Rasbash (hence the "JR" in Stat-JR). Sadly, Jon died suddenly just as we began developing the system, and so we dedicate this software to his memory. We hope that you enjoy using Stat-JR and are inspired to become part of the Stat-JR community: either through the creation of your own templates that can be shared with others, or simply by providing feedback on existing templates.

Happy Modelling,

The Stat-JR team.

1.2 About the Advanced User's Guide

The Advanced Guide is meant to complement the Beginners guide and we recommend that users read that guide first to get an idea of how the Stat-JR software works. A major component of the Stat-JR package is the use of (often user written) templates. Templates are pieces of computer code (written in the Python language) that perform a specific task. Many of the templates are used to fit a specific family of statistical models although there are other templates that perform data input, data manipulation and data and graphical output.

In this document it is our aim to give users who intend to write their own templates, or more generally are interested in how the Stat-JR system works, more details about how to write templates and to some degree how the system fits together. We will do this by showing the code for several of the templates we have written and giving a detailed explanation of what each function and even in places each line of code does.

An initial question posed by potential template writers has been what language are templates written in and when told 'Python' then ask whether we are providing an introductory chapter on this language. We are not specifically writing an introductory chapter on Python (good books include Hetland (2005) and Lutz and Ascher (2004)) as it has a vast language and we will mainly be interested in specific aspects of the language, some of which are non-standard and specific to Stat-JR. In fact many of the functions that make up a template in Stat-JR are designed to create text blocks in other languages, for example C++, WinBUGS or any of the other macro languages associated with the software packages supported via inter-operability. This is not to say that reading up on Python is without merit and certainly Python experts will find writing templates initially easier than others (though more because of their programming skills than their Python skills per se).

Our advice is therefore to work through this guide first and try the exercises and have a Python book as a backstop for when you are stuck writing your own templates. We will now give instructions into how to install all the software needed to run Stat-JR before moving on to our first example template.

2 Installation instructions

Stat-JR has a dedicated website for requests for a copy of the software and which contains instructions for installation. This is currently located at http://www.bristol.ac.uk/cmm/software/statjr/index.html

To run the software:

Stat-JR runs in a web browser; whilst it will work in most web browsers we suggest not using Internet Explorer, although it is hoped support for more browsers will be added in future. To start Stat-JR, select the *Stat-JR TREE* link from the *Centre for Multilevel Modelling* suite on the start up menu; this should bring up a (Chrome) web browser.

When you open *TREE*, this action starts a Command prompt window in the background to which commands are printed out. This window is useful for viewing what the system is doing: for example, on the machine on which I have run *TREE* I can see the following commands:

```
WARNING:root:Failed to load package GenStat_model (GenStat not found)
WARNING:root:Failed to load package Minitab_model (Minitab not found)
WARNING:root:Failed to load package Minitab_script (Minitab not found)
WARNING:root:Failed to load package SABRE (Sabre not found)
INFO:root:Trying to locate and open default web browser
http://0.0.0.0:51886/
```

The most important command when starting up is the line:

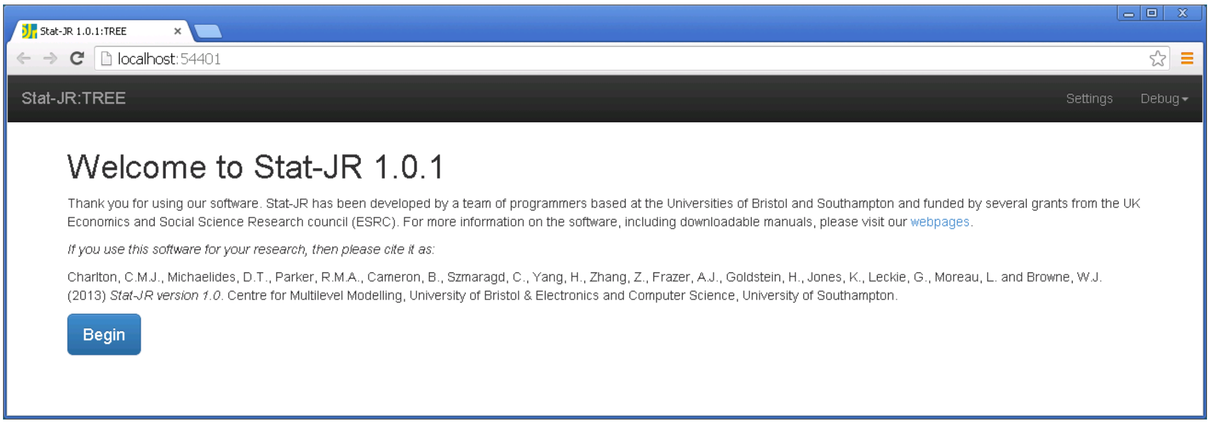
```
http://0.0.0.0:51886/
```

Note that the number 51886 is specific to this run of the program and will be different on your machine. This line only appears when the program has performed all its initial set-up routines. This may take a while, particularly the first time you use the program. You should then be able to view the start page of the TREE interface in your browser (note: not all browser packages are supported: see note above); if you can't, then try refreshing the browser window, or typing **localhost:51886** into the address bar (substituting the number you see for 51886). TREE is short for *Template Reading and Execution Environment* and is an interface into Stat-JR that allows the user to look at a single template and dataset at a time. There are also an eBook interface (called DEEP) and a Python command line interface but in this manual we stick with the TREE interface

3 A simple regression template example

3.1 Running a first template

We will firstly consider a very simple template that is included in the core model templates distributed with Stat-JR which has the title *Regression1*. This template is used in the Beginners guide and perhaps before looking at the code it would be good to run the template again in the TREE interface to see what it does. To do this start up the Stat-JR package as directed in section 2. If you refresh the screen you should be greeted by the following display:



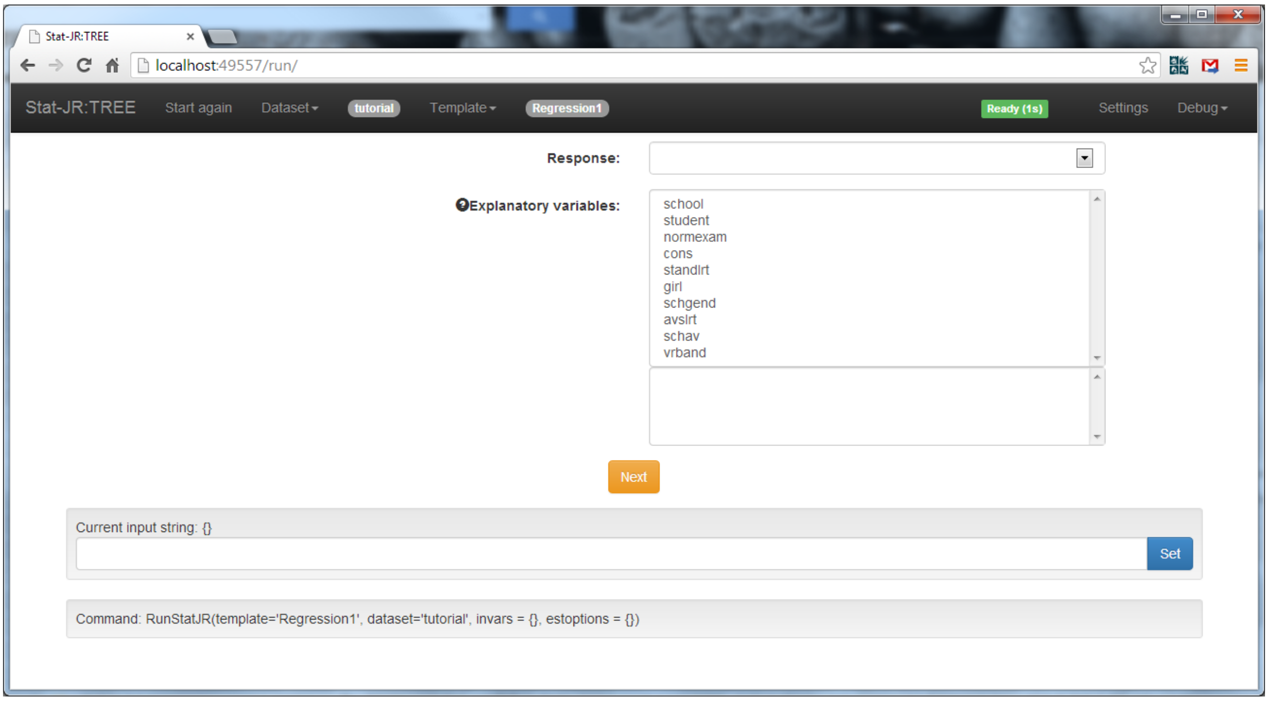
Welcome to Stat-JR 1.0.1

Thank you for using our software. Stat-JR has been developed by a team of programmers based at the Universities of Bristol and Southampton and funded by several grants from the UK Economics and Social Science Research council (ESRC). For more information on the software, including downloadable manuals, please visit our webpages.

If you use this software for your research, then please cite it as:

Charlton, C.M.J., Michaelides, D.T., Parker, R.M.A., Cameron, B., Szmaragd, C., Yang, H., Zhang, Z., Frazer, A.J., Goldstein, H., Jones, K., Leckie, G., Moreau, L. and Browne, W.J. (2013) *Stat-JR version 1.0.* Centre for Multilevel Modelling, University of Bristol & Electronics and Computer Science, University of Southampton.

If you click on **Begin** then the software will move from this welcome page to the general working page. Here you will see that the template *Regression1* is the default template on start up and the default dataset is called *tutorial* [see the beginner's guide for more information on the dataset]. The screen will look as shown below:



Response:

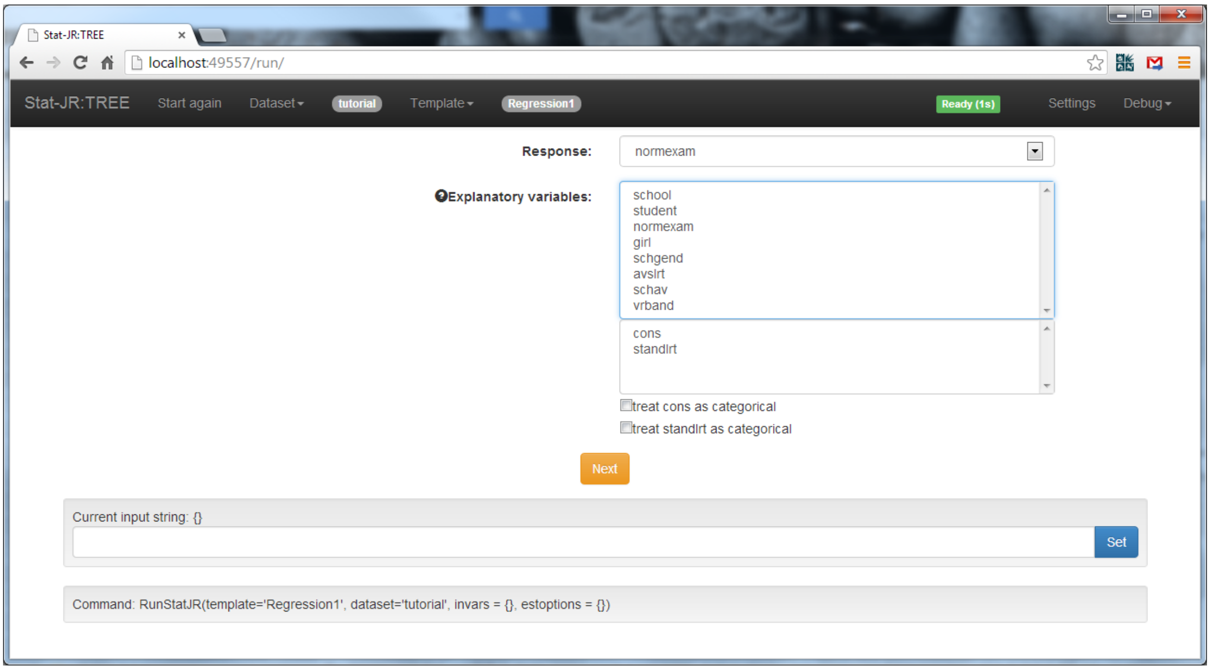
Explanatory variables:

school
student
normexam
cons
standlrt
girl
schgend
avslrt
schav
vrband

Current input string: {}

Command: RunStatJR(template='Regression1', dataset='tutorial', invars = {}, estoptions = {})

Here you will see a **main pane** which is looking for inputs for a response (a single select list) and explanatory variables (a multiple select list). We will here select *normexam* as the response and *cons* and *standlrt* as the explanatory variables as shown below:



Response: normexam

Explanatory variables:

school
student
normexam
girl
schgend
avslrt
schav
vrband

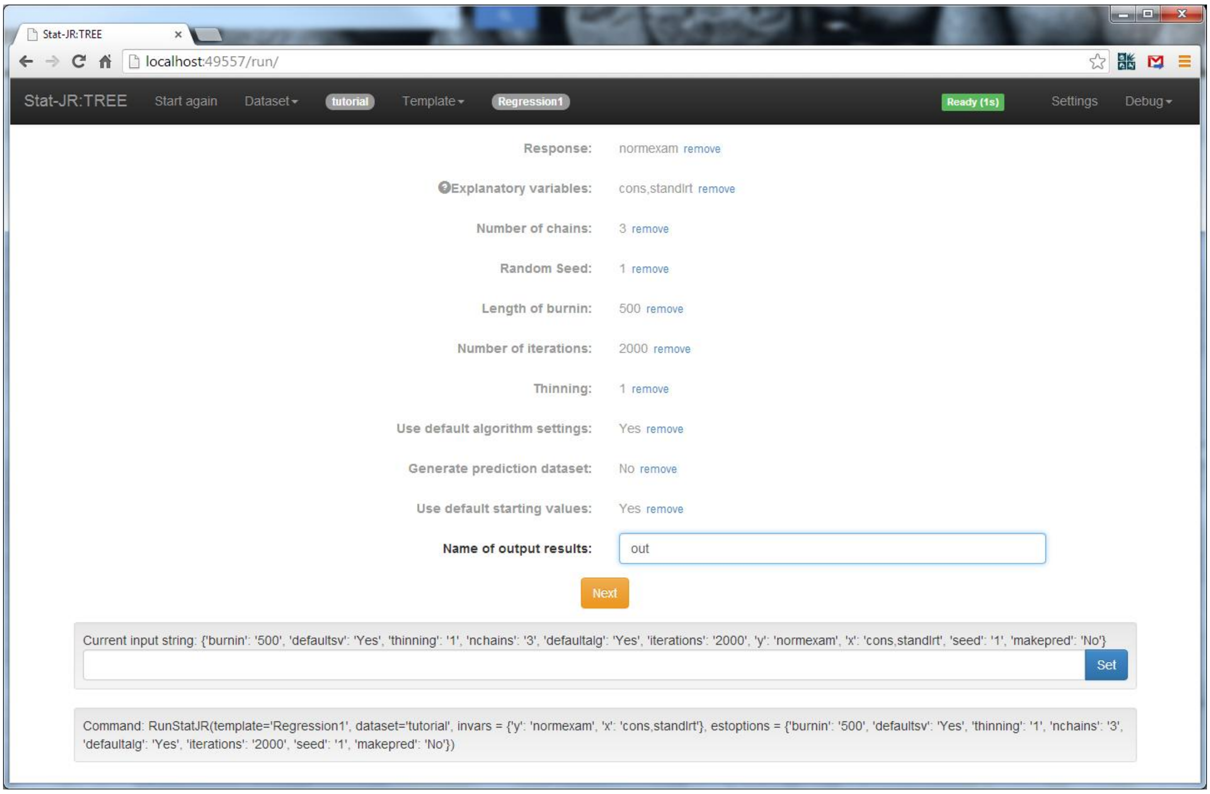
cons
standlrt

□treat cons as categorical
□treat standlrt as categorical

Current input string: {}

Command: RunStatJR(template='Regression1', dataset='tutorial', invars = {}, estoptions = {})

Next we click on the **Next** button and fill in the input boxes that appear as follows:



Response: normexam remove

Explanatory variables: cons,standlrt remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

Use default algorithm settings: Yes remove

Generate prediction dataset: No remove

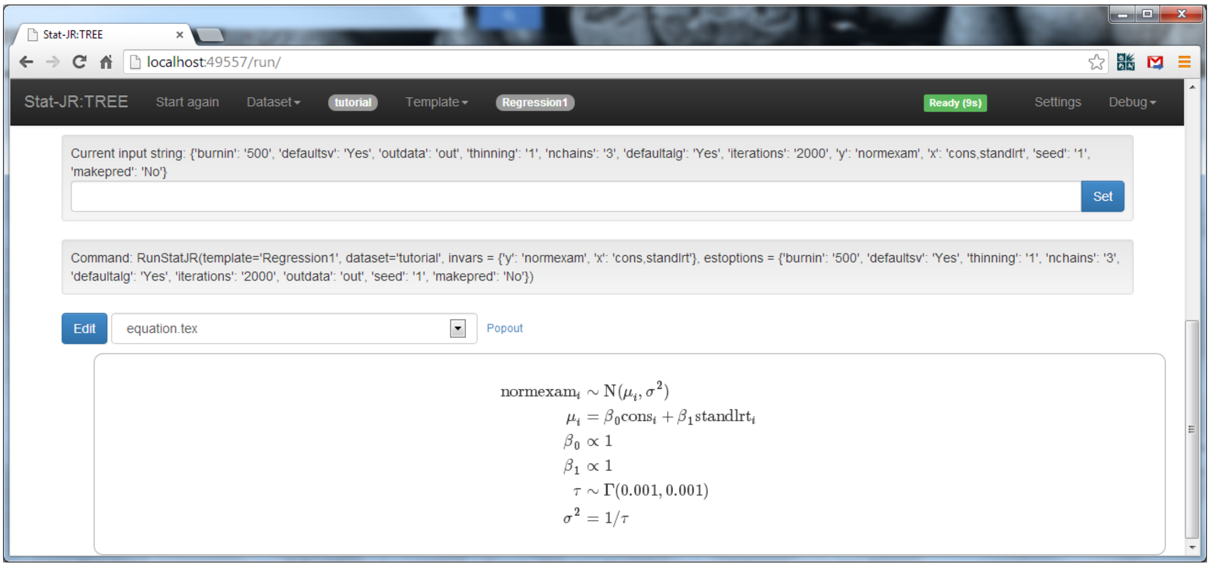
Use default starting values: Yes remove

Name of output results: out

Current input string: {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1', 'makepred': 'No'}

Command: RunStatJR(template='Regression1', dataset='tutorial', invars = {'y': 'normexam', 'x': 'cons,standlrt'}, estoptions = {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'seed': '1', 'makepred': 'No'})

Note that when all boxes on the screen are filled in, clicking the **Next** button will show further inputs if there are any. Here we have given a name of the object/dataset where the results are to be stored. The other inputs are for the MCMC estimation methods we are using. When you click **Next** again the software will perform some procedures in the background and after a short while the screen will expand to include a lower pane with an accompanying pull down list in which various objects generated by Stat-JR can be selected and displayed thus:

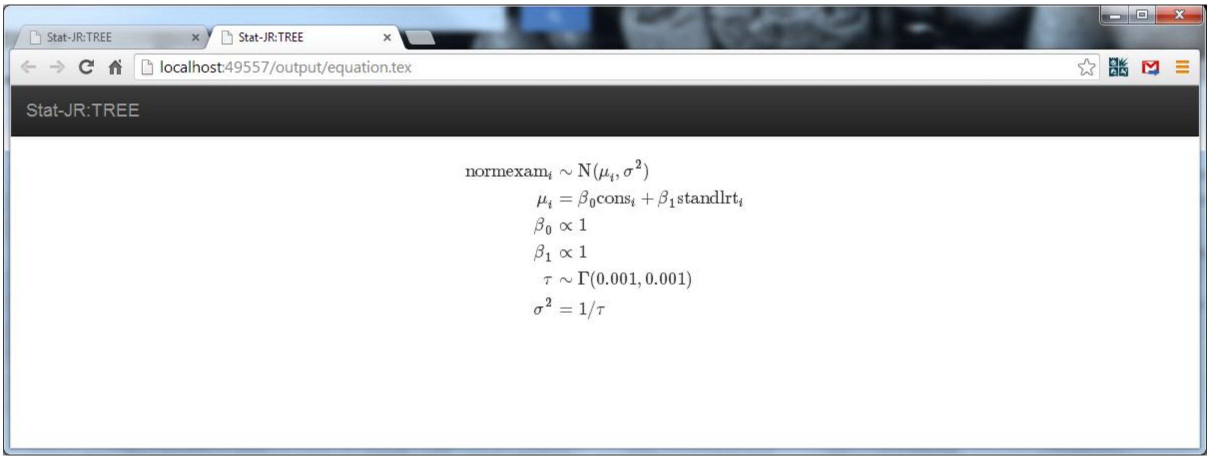


Current input string: {'burnin': '500', 'defaultsv': 'Yes', 'outdata': 'out', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1', 'makepred': 'No'}

Command: RunStatJR(template='Regression1', dataset='tutorial', invars = {'y': 'normexam', 'x': 'cons,standlrt'}, estoptions = {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'outdata': 'out', 'seed': '1', 'makepred': 'No'})

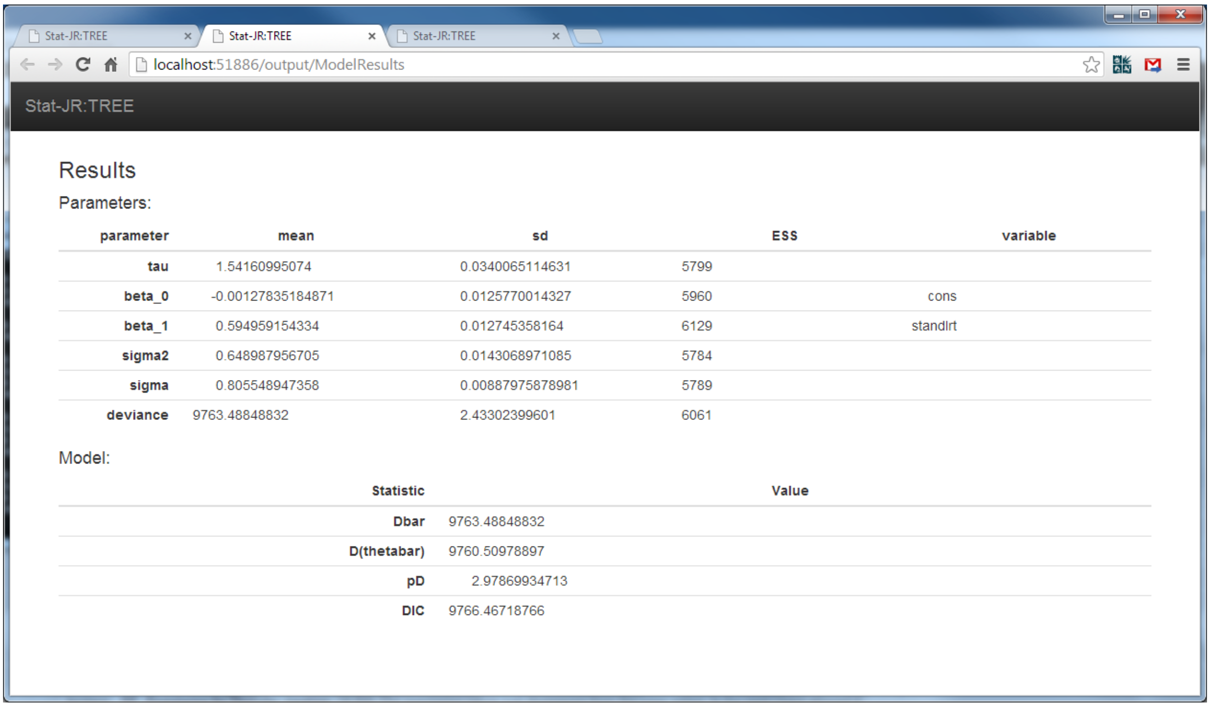
$$\begin{aligned} \mathrm{normexam}_i &\sim \mathrm{N}(\mu_i, \sigma^2) \\ \mu_i &= \beta_0 \mathrm{cons}_i + \beta_1 \mathrm{standlrt}_i \\ \beta_0 &\propto 1 \\ \beta_1 &\propto 1 \\ \tau &\sim \Gamma(0.001, 0.001) \\ \sigma^2 &= 1/\tau \end{aligned}$$

The pull down list contains several objects including the model code which looks a bit like WinBUGS code which the system uses to create code that will fit the model. Currently a nicely formatted mathematical description of the model (in LaTeX code) is shown in the pane and named ***equation.tex***. Objects can be displayed in their own tab by clicking on the word *Popout* to the right of the pull down list:



$$\begin{aligned} \mathrm{normexam}_i &\sim \mathrm{N}(\mu_i, \sigma^2) \\ \mu_i &= \beta_0 \mathrm{cons}_i + \beta_1 \mathrm{standlrt}_i \\ \beta_0 &\propto 1 \\ \beta_1 &\propto 1 \\ \tau &\sim \Gamma(0.001, 0.001) \\ \sigma^2 &= 1/\tau \end{aligned}$$

Back in the first Stat-JR tab there is a green **Run** button above the output pane. If we click this button then after a short while the model will run. There is a counter in the bar at the top of the tab which indicates when Stat-JR is working (coloured blue) or ready (coloured green) and how long the execution took. The first time a model is run the code will need compiling which will take a while. When the execution has finished we will have more objects to choose from in the pull down list including the results (**ModelResults)** which we show popped out below:



Results

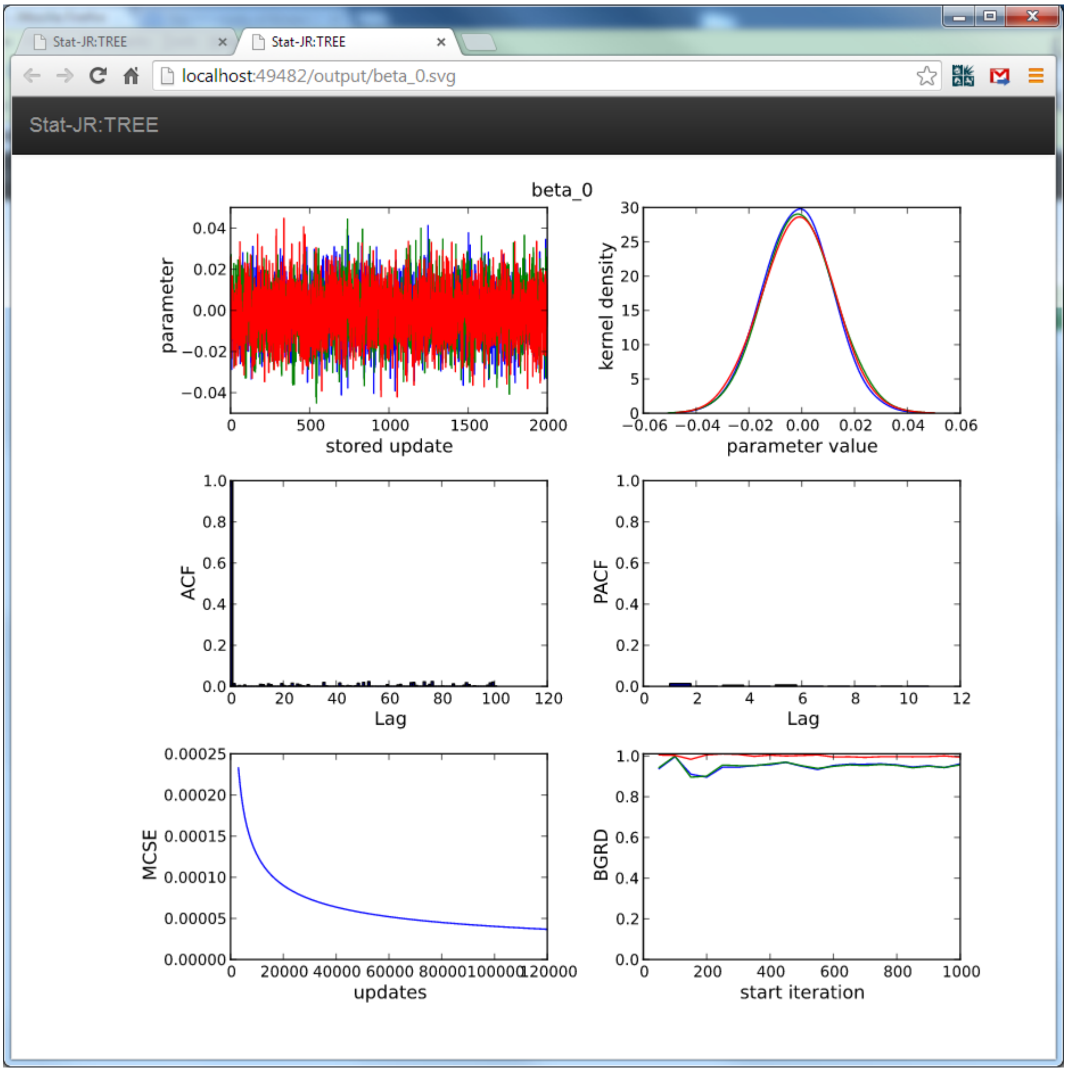
Parameters:

parameter	mean	sd	ESS	variable
tau	1.54160995074	0.0340065114631	5799	
beta_0	-0.00127835184871	0.0125770014327	5960	cons
beta_1	0.594959154334	0.012745358164	6129	standlrt
sigma2	0.648987956705	0.0143068971085	5784	
sigma	0.805548947358	0.00887975878981	5789	
deviance	9763.48848832	2.43302399601	6061	

Model:

Statistic	Value
Dbar	9763.48848832
D(thetabar)	9760.50978897
pD	2.97869934713
DIC	9766.46718766

This screen contains summary statistics for five parameters (in fact sigma, sigma2 and tau are all functions of each other). We can also look at diagnostic plots for the parameters e.g. **beta_0.svg**:



The purpose of this document is not to go into details about what these figures mean – interested readers can look at the accompanying Beginner's guide for such information. Instead we want to teach you here how to write a similar template yourself.

3.2 Opening the bonnet and looking at the code

The operations that we have here performed in fitting our first model are shared between the user written template *Regression1* and other code that is generic to all templates and which we will discuss in more detail later.

So our next stage is to look at the source python file for Regression1. All templates are stored in the *templates* subdirectory under the base directory and have the extension *.py* and so if we open *Regression1.py* (in Wordpad/Notepad and not Python) we will see the following:

```
# Copyright (c) 2013, University of Bristol and University of
Southampton.

from EStat.Templating import Template

class Regression1(Template):
    'A model template for fitting 1 level Normal multiple regression
model in eStat only.'

    __version__ = '1.0.0'

    tags = [ 'Model', '1-Level', 'Normal' ]
    engines = ['eStat']

    inputs = '''
y = DataVector('Response: ', help= 'a.k.a. <em>Y</em>, Outcome
variable, Dependent variable, etc.')
x = DataMatrix('Explanatory variables: ', allow_cat = True, help=
"<p style='text-align:left'>A.k.a. X, Predictor variables,
Independent variables, etc.</p><p style='text-
align:left'><strong>Note:</strong> if you wish to include an
<strong>intercept</strong> then you need to add it (e.g. a constant
of ones) as one of the explanatory variables.</p><p style='text-
align:left'>Once you've selected a variable, you have the
opportunity to indicate whether it's categorical or not; if
categorical, dummy variables will be added to the model on your
behalf.</p>")
beta = ParamVector(parents=[x], as_scalar=True)
tau = ParamScalar()
sigma = ParamScalar(modelled = False)
sigma2 = ParamScalar(modelled = False)
deviance = ParamScalar(modelled = False)
'''

    model = '''
model{
    for (i in 1:length(${y})) {
        ${y}[i] ~ dnorm(mu[i], tau)
        mu[i] <- ${mmult(x, 'beta', 'i')}
    }

    # Priors
    % for i in range(0, x.ncols()):
    beta_${i} ~ dflat()
    % endfor
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    sigma <- 1 / sqrt(tau)
}
'''
    latex = r'''
\begin{aligned}
 \mbox{${y}}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & =
  ${mmulttex(x, r'\beta', 'i')} \\
%for i in range(0, len(x)):
\beta_${i} & \propto 1 \\
%endfor
\tau & \sim \Gamma (0.001,0.001) \\
\sigma^2 & = 1 / \tau
\end{aligned}
'''
```

We will now describe in some detail what this code does. The first line (after the initial copyright statement) here is simply importing information needed by the template and is generic to many templates. We then have a class statement which defines a class *Regression1* which is a subclass of a generic *Template* class. There is then a sentence known as a descriptor that describes what the template does. For those unfamiliar with the terminology we are using think of a class as being a definition of a type of object, for example we might have a class of rectangles where each rectangle might be described by two attributes, length and width. Then an instance of the class which we might call Dave will have these values instantiated e.g. Dave's length is 3 and width is 1.
We might think of the subclass of rectangles the squares which again have the two attributes length and width. We could state that class Square (Rectangle): in which case we know that as squares are a subclass of rectangles they have a length and width but we would now redefine the attribute width within the squares definition to equal length.
This terminology is what is used in what is called object orientated programming.
In the definition here five attributes (tags, engines, inputs, model, and latex) are then defined as being parts of a *Regression1* class although there will be other attributes that are generic to the template class and are defined elsewhere.
Briefly:
The *__version__* attribute identifies which version of this file this is. It is useful for debugging as when a bug is fixed and a new version created we update the version number.
The *tags* attribute identifies the template as belonging to the tag groups 'Model', '1-Level', and 'Normal' and this is used in the web interface to decide which templates to show in specific template lists.
The *engines* attribute identifies which estimation and or graphical engines can be used with this template (in this case just the built-in 'eStat' estimation engine) which is used by Stat-JR to decide which estimation options to offer. This attribute is how, along with additional attributes, we allow Stat-JR to interoperate with other software.
The *inputs* attribute is a text string (hence the starting and ending ''') which consists of a list of the inputs in this template.
The *model* attribute is a text string that will produce the model code we saw in the web interface for this template.
The *latex* attribute is a text string that will produce a piece of LaTeX code which is converted into the nice mathematics we saw in the web interface. We will next look at the last three attributes in more detail.

3.2.1 Inputs

When this template has been selected in the web interface it will firstly have its inputs interrogated and start creating an instance of a model object. Stat-JR has a list of object types that can be thought of as the building blocks of a model object. Statements like

```
y = DataVector('Response: ', help= 'a.k.a. <em>Y</em>, Outcome
variable, Dependent variable, etc.')
```

can be thought of as defining the components that make up a model object , so here we are building a model object that contains a data vector called *y*. The text in the brackets is used by the web interface as a piece of text to place on the screen alongside the appropriate input device (in the case of a data vector a single select list) and the second *help* string is help text that appears if you hover over the input in the browser.

Stat-JR distinguishes between Data objects which require user inputs and Parameters (Param) which just need to be declared. This template therefore has 7 components (2 pieces of data and 5 parameters) that make up the model. The DataMatrix declaration for *x* will correspond to the multiple-select list that we saw when running the template. Here we see that this declaration takes a couple of arguments:

```
x = DataMatrix('Explanatory variables: ', allow_cat = True, help=
"<p style='text-align:left'>A.k.a. X, Predictor variables,
Independent variables, etc.</p><p style='text-
align:left'><strong>Note:</strong> if you wish to include an
<strong>intercept</strong> then you need to add it (e.g. a constant
of ones) as one of the explanatory variables.</p><p style='text-
align:left'>Once you've selected a variable, you have the
opportunity to indicate whether it's categorical or not; if
categorical, dummy variables will be added to the model on your
behalf.</p>")
```

In this version of Stat-JR we have implemented code to deal with categorical predictor variables and so the allow_cat argument tells Stat-JR that elements of x might be treated as categorical variables. The *help* argument contains a rather long text string that will appear on the screen if we hover over x with the mouse. Note that to save space when we display code for templates later in this manual we remove the help text.

3.2.2 Model

The *model* attribute gives a definition (as a text string) of an instance of a model set up using this template. The definition is in a language that very much resembles the language used by the WinBUGS package to specify models (with some minor differences) and will be used in Stat-JR to create code to run the model using the eStat engine. The definition can be shown on the screen in the objects pane under the label *model.txt* so you can for example see the definition for the model we fitted to the *tutorial* dataset earlier by selecting this object from the pull down list. As the text is specific to the inputs given, the definition is a text string containing some quantities that depend on inputs. These are integrated into the text string via the $ symbol for substitutions, through conditional and looping computation achieved via % commands and through the calling of external functions. The model code for this template uses all three devices, and so we will here go through stage by stage the instance of model shown in the earlier screen shots.

We start with the raw code:

```
    model = '''
model{
    for (i in 1:length(${y})) {
        ${y}[i] ~ dnorm(mu[i], tau)
        mu[i] <- ${mmult(x, 'beta', 'i')}
    }

    # Priors
    % for i in range(0, x.ncols()):
    beta_${i} ~ dflat()
    % endfor
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    sigma <- 1 / sqrt(tau)
}
'''
```

Now we can substitute normexam for ${y} as this is the column we chose for *y* thus:

```
model = '''
model{
    for (i in 1:length(normexam)) {
        normexam[i] ~ dnorm(mu[i], tau)
        mu[i] <- ${mmult(x, 'beta', 'i')}
    }

    # Priors
    % for i in range(0, x.ncols()):
    beta${i} ~ dflat()
    % endfor
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    sigma <- 1 / sqrt(tau)
}
        '''
```

Next we can evaluate the *for* loop with, in our example *x* having 2 columns:

```
model = '''
model{
    for (i in 1:length(normexam)) {
        normexam[i] ~ dnorm(mu[i], tau)
        mu[i] <- ${mmult(x, 'beta', 'i')}
    }

    # Priors
    beta0 ~ dflat()
    beta1 ~ dflat()
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    sigma <- 1 / sqrt(tau)
}
        '''
```

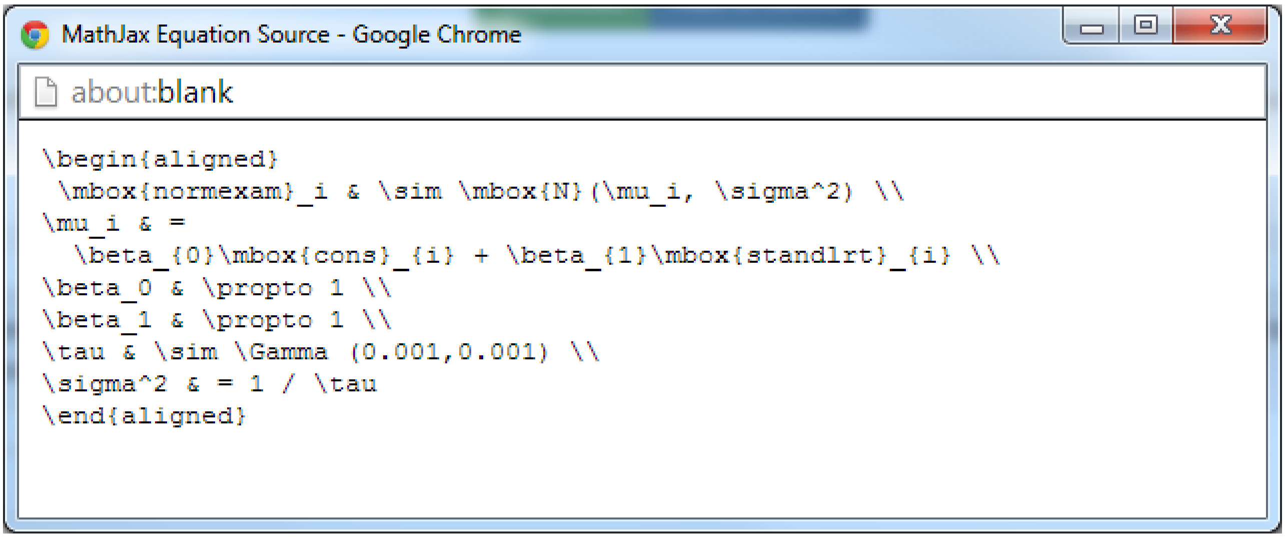
Finally the function *mmult* is a function written separately and is used to create the products of the *x* variables and their associated *beta*s with appropriate indexing. When run we get:

```
model = '''
model{
    for (i in 1:length(normexam)) {
        normexam[i] ~ dnorm(mu[i], tau)
        mu[i] <- cons[i]*beta0 + standlrt[i]*beta1
    }

    # Priors
    beta0 ~ dflat()
    beta1 ~ dflat()
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    sigma <- 1 / sqrt(tau)
}
        '''
```

This is identical to the code we see under **model.txt** in the TREE interface and is one way of displaying the model we wish to fit. Another way is to write the model in mathematical form using the LaTeX language and this can also be shown in the web output in the pull down list under **equation.tex** as we saw earlier. Basically we are using a program called *MathJax* which will display LaTeX code in a nice format embedded within a webpage. The attribute that is used for creating this code is latex.

3.2.3 Latex

If you select *equation.tex* in the pull down list click for the bottom pane then the equations will appear in LaTeX format. If you then right click in the pane and select *Show Maths as TeX commands* option you will get a window popping up that shows the LaTeX source:



```
\begin{aligned}
 \mbox{normexam}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & =
  \beta_{0}\mbox{cons}_{i} + \beta_{1}\mbox{standlrt}_{i} \\
\beta_0 & \propto 1 \\
\beta_1 & \propto 1 \\
\tau & \sim \Gamma (0.001,0.001) \\
\sigma^2 & = 1 / \tau
\end{aligned}
```

This code is created via the *latex* function and we will now look at how we get from *latex* to this source for our example. The generic code is as follows:

```
latex = r'''
\begin{aligned}
 \mbox{${y}}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & =
  ${mmulttex(x, r'\beta', 'i')} \\
%for i in range(0, len(x)):
\beta_${i} & \propto 1 \\
%endfor
\tau & \sim \Gamma (0.001,0.001) \\
\sigma^2 & = 1 / \tau
\end{aligned}
'''
```

We have three steps as with the *model* function, firstly we will substitute *normexam* for ${y}

```
latex = r'''
\begin{aligned}
 \mbox{normexam}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & =
  ${mmulttex(x, r'\beta', 'i')} \\
%for i in range(0, len(x)):
\beta_${i} & \propto 1 \\
%endfor
\tau & \sim \Gamma (0.001,0.001) \\
\sigma^2 & = 1 / \tau
\end{aligned}
'''
```

Next we can evaluate the for loop with, in our example x having 2 columns:

```
latex = r'''
\begin{aligned}
 \mbox{normexam}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & =
  ${mmulttex(x, r'\beta', 'i')} \\
\beta_0 & \propto 1 \\
\beta_1 & \propto 1 \\
\tau & \sim \Gamma (0.001,0.001) \\
\sigma^2 & = 1 / \tau
\end{aligned}
'''
```

and finally we have the step to expand a function – this time called mmulttex :

```
latex = r'''
\begin{aligned}
 \mbox{normexam}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & =
  \beta_0\mbox{cons}_{i} + \beta_{1}\mbox{standlrt}_{i}
\beta_0 & \propto 1 \\
\beta_1 & \propto 1 \\
\tau & \sim \Gamma (0.001,0.001) \\
\sigma^2 & = 1 / \tau
\end{aligned}
'''
```

3.2.4 Some points to note

You will notice that the string object created in *latex* has an **r** before the ‘’’ and that similarly there is an **r** inside the *mmulttex* function call before the ‘. Basically the triple quotes are used in place of quotes to allow the use of single quotes within the actual expression. The **r** is used to let the computer know that the expression in the quotes is a raw string and so for example although the \ character is often used as a control character, in a raw string it will be treated simply as a \ and passed through to the LaTeX reading software. This avoids the use of lots of double \ for each \. One debugging tip is that lines often finish with a double slash to denote a new line in LaTeX. It is important to add a space after the double slash in the text file as otherwise it will be concatenated onto the next line.

Some of you will know LaTeX and so the code in the source window will be familiar. It is however not essential to write a *latex* function for your own templates as the code is purely decorative. We will not give a crash course on LaTeX here but essentially the *aligned* environment is used to write a set of mathematical equations with the & sign denoting the place where the lines are lined up horizontally and the double slashes denoting new lines. LaTeX uses the \ preceding terms to denote special characters e.g. \beta gives a Greek lowercase beta. The aligned environment is for mathematics and so if we wish to write words in normal font we enclose them in a \mbox. With this basic knowledge you should be able to compare the source code and the maths it produces and thus see what each of the special characters is.

3.3 Writing your own first template

We haven't at this stage explained how the *model* function is used to create code to fit the model. This is done by the Stat-JR system's eStat engine using generic code that is common to all templates and which we will discuss a bit more later. It is enough for now to realise that to write some basic templates simply requires writing code similar to that seen here and the Stat-JR system will do the rest of the hard work for you. We will now test your understanding by getting you to construct your own first template:

Exercise 1

It is best when starting writing templates to start from a template that works and modify it to confirm you understand what is going on. You will therefore now take the *Regression1* template and construct a template for an even simpler model – a simple linear regression. To do this in the template directory copy the file *Regression1.py* to *LinReg.py*. It is also sensible to change the classname in the template.

For a linear regression we want a template with two inputs y and x – only this time x is a vector rather than a matrix i.e. there is only one predictor plus a constant. Try changing the text to ask specifically for a Y variable and an X variable for the inputs. You will need to change *inputs* a little.

Try also then simplifying the *model* and *latex* functions – you should be able to get away without needing the *mmult/mmulttex* functions.

In fact mu[i] should be something like alpha + beta*x[i], though if you use alpha and beta they will both need declaring as ParamScalar's in the *inputs* function.

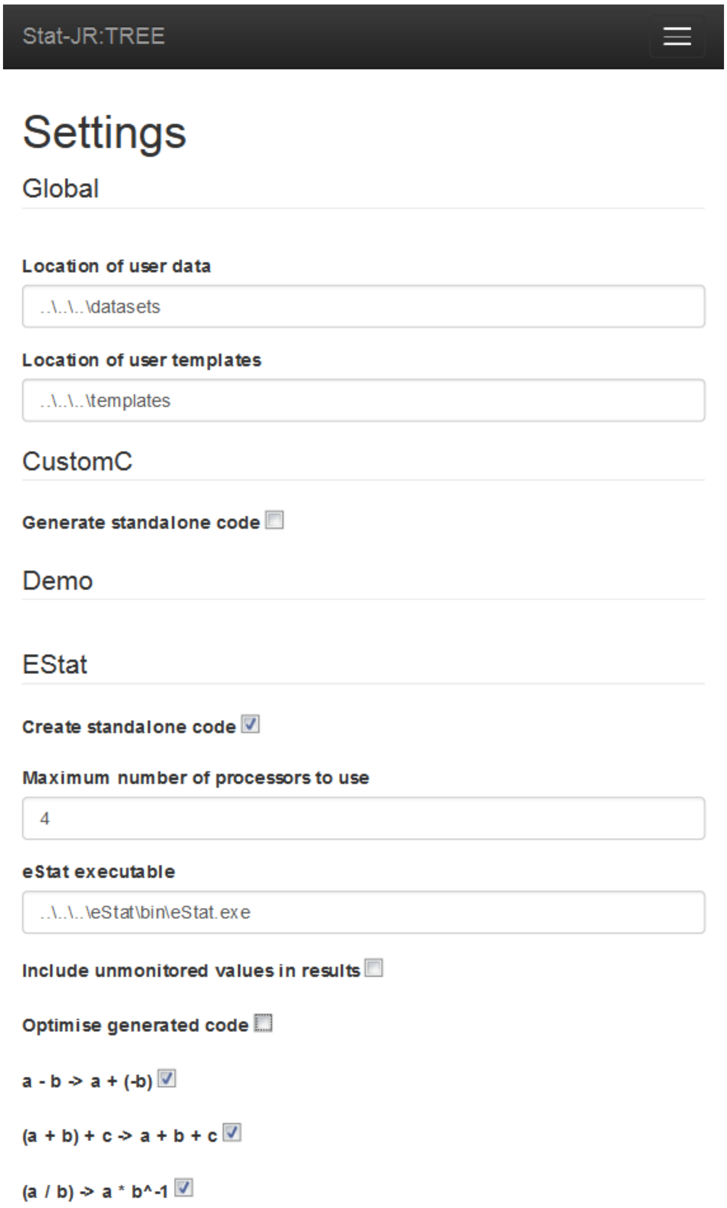
When you think you have the template correctly written save it and reload templates in Stat-JR (via the *Debug* menu) and test it out. If it is saved in the templates directory it will be automatically picked up. It should give similar results to *Regression1* for the example shown earlier.

4 Running templates with the eStat engine

4.1 Algebra and Code Generation

In section 3 we have seen the code required to create a template that fits a simple model using the built-in eStat estimation engine. We have however hidden away many of the details. In this section we will expose a few more details, including a little section on the algebra system. Let us start by returning to the same example and show a few more screens that we have not yet exposed.

We will begin however by switching a few of the settings so that we can easier see what is going on. To do this look at the black bar at the top of the screen and you will see the word *Settings* . Click on **Settings** and you will be greeted by a settings screen (part of which is shown below) where you will need to change the inputs to look as follows:



Settings

Global

Location of user data

..\..\..\datasets

Location of user templates

..\..\..\templates

CustomC

Generate standalone code □

Demo

EStat

Create standalone code ☑

Maximum number of processors to use

4

eStat executable

..\..\..\eStat\bin\eStat.exe

Include unmonitored values in results □

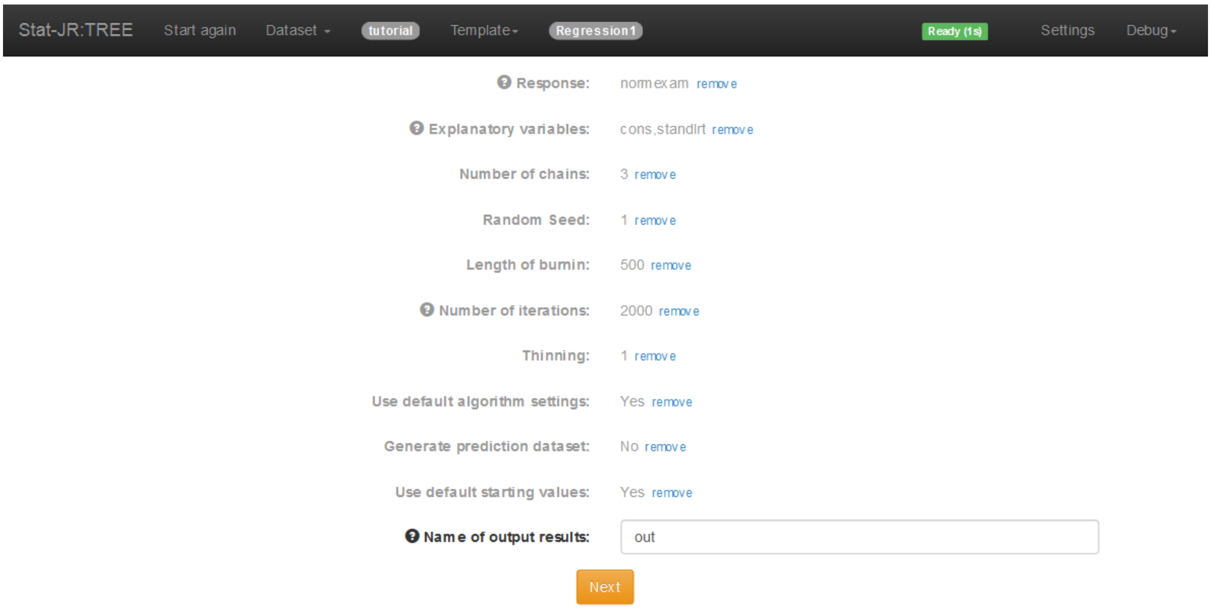
Optimise generated code □

a - b -> a + (-b) ☑

(a + b) + c -> a + b + c ☑

(a / b) -> a * b^-1 ☑

Here we have switched on *standalone code* and also switched off *optimisation*. The Settings screen contains the pathnames to all third party software one might use with Stat-JR and specific eStat settings which we have modified here. Scroll to the bottom of the screen and click on the **Set** button when you have made the changes which will take you back to the welcome screen. Now click Begin as before and using **Regression1** as the Template and **tutorial** as the dataset set up the inputs as follows:



Response: normexam remove

Explanatory variables: cons,standlrt remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

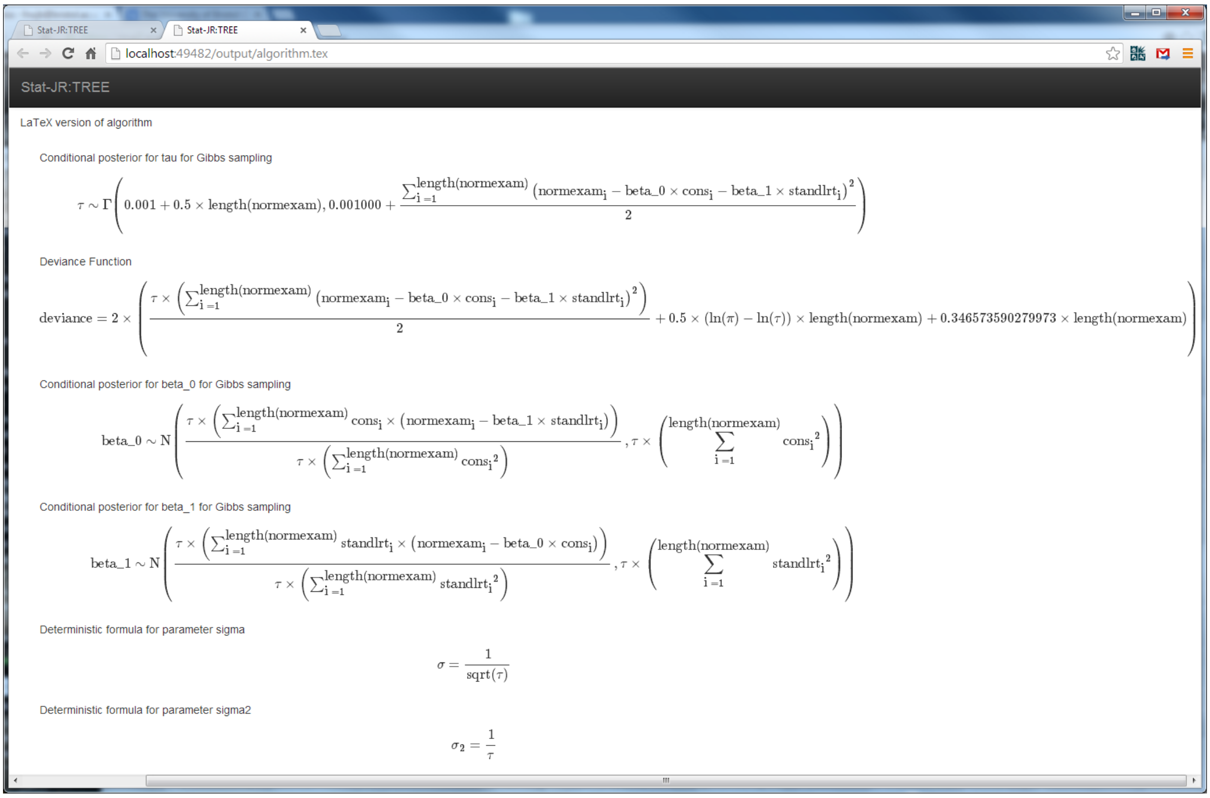
Use default algorithm settings: Yes remove

Generate prediction dataset: No remove

Use default starting values: Yes remove

Name of output results: out

Now clicking on **Next** we can choose other options from the pull down list so firstly choose **algorithm.tex** from the list and pop it out into a new tab.



LaTeX version of algorithm

Conditional posterior for tau for Gibbs sampling

$$\tau \sim \Gamma\left(0.001 + 0.5 \times \mathrm{length(normexam)}, 0.001000 + \frac{\sum_{\mathrm{i}=1}^{\mathrm{length(normexam)}} \left(\mathrm{normexam_i} - \mathrm{beta\_0} \times \mathrm{cons_i} - \mathrm{beta\_1} \times \mathrm{standlrt_i}\right)^2}{2}\right)$$

Deviance Function

$$\mathrm{deviance} = 2 \times \left(\frac{\tau \times \left(\sum_{\mathrm{i}=1}^{\mathrm{length(normexam)}} \left(\mathrm{normexam_i} - \mathrm{beta\_0} \times \mathrm{cons_i} - \mathrm{beta\_1} \times \mathrm{standlrt_i}\right)^2\right)}{2} + 0.5 \times (\ln(\pi) - \ln(\tau)) \times \mathrm{length(normexam)} + 0.346573590279973 \times \mathrm{length(normexam)}\right)$$

Conditional posterior for beta_0 for Gibbs sampling

$$\mathrm{beta\_0} \sim \mathrm{N}\left(\frac{\tau \times \left(\sum_{\mathrm{i}=1}^{\mathrm{length(normexam)}} \mathrm{cons_i} \times \left(\mathrm{normexam_i} - \mathrm{beta\_1} \times \mathrm{standlrt_i}\right)\right)}{\tau \times \left(\sum_{\mathrm{i}=1}^{\mathrm{length(normexam)}} \mathrm{cons_i}^2\right)}, \tau \times \left(\sum_{\mathrm{i}=1}^{\mathrm{length(normexam)}} \mathrm{cons_i}^2\right)\right)$$

Conditional posterior for beta_1 for Gibbs sampling

$$\mathrm{beta\_1} \sim \mathrm{N}\left(\frac{\tau \times \left(\sum_{\mathrm{i}=1}^{\mathrm{length(normexam)}} \mathrm{standlrt_i} \times \left(\mathrm{normexam_i} - \mathrm{beta\_0} \times \mathrm{cons_i}\right)\right)}{\tau \times \left(\sum_{\mathrm{i}=1}^{\mathrm{length(normexam)}} \mathrm{standlrt_i}^2\right)}, \tau \times \left(\sum_{\mathrm{i}=1}^{\mathrm{length(normexam)}} \mathrm{standlrt_i}^2\right)\right)$$

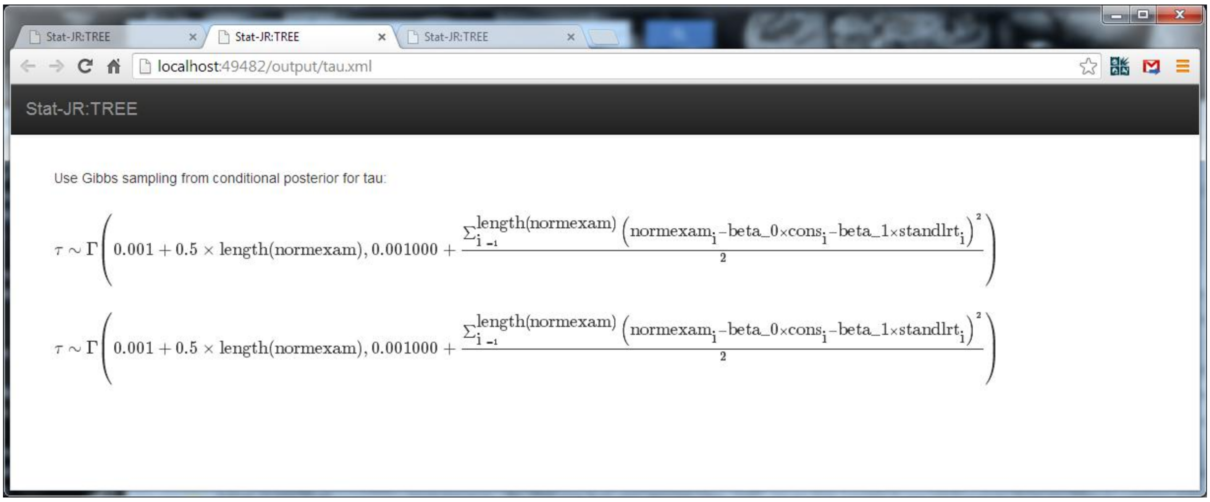
Deterministic formula for parameter sigma

$$\sigma = \frac{1}{\mathrm{sqrt}(\tau)}$$

Deterministic formula for parameter sigma2

$$\sigma_2 = \frac{1}{\tau}$$

Basically this window shows a nicely presented result of what is returned from the algebra system when it is given the model description constructed by the **model** method. We will look at the algebra system in a little more detail later on, but for now you will see that three of the parameters (*beta0, beta1* and *tau*) have posterior distributions that require sampling from a conditional distribution using a method called Gibbs sampling whilst two (*sigma* and *sigma2*) are simply calculated as deterministic functions of the other parameters. Finally a formula for the deviance function is also returned. In fact the algebra system returns a series of files (in xml format), one for each parameter and we can also view these (in nicely presented form) for example **tau.xml**

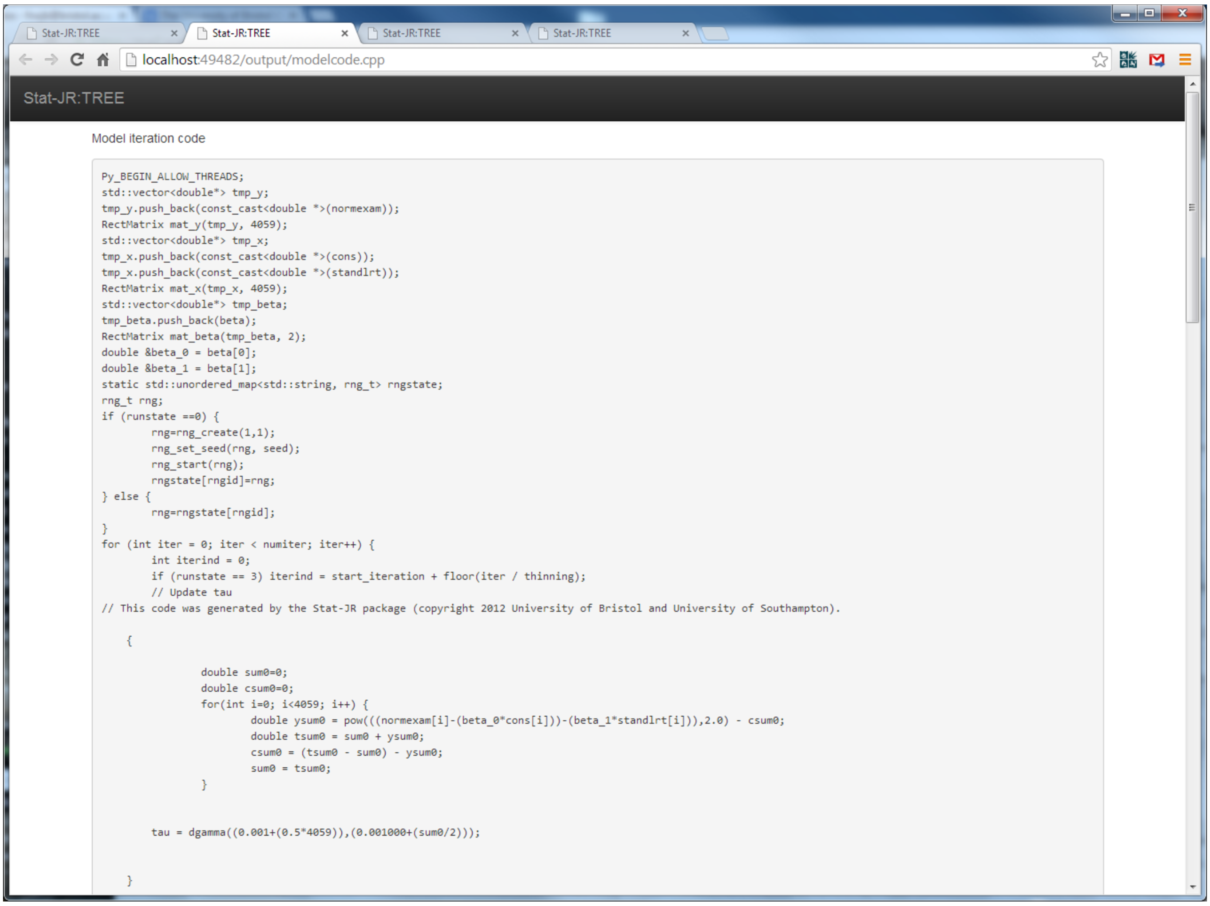


Use Gibbs sampling from conditional posterior for tau:

$$\tau \sim \Gamma\left(0.001 + 0.5 \times \mathrm{length(normexam)}, 0.001000 + \frac{\Sigma_{i=1}^{\mathrm{length(normexam)}} \left(\mathrm{normexam}_i - \mathrm{beta\_0} \times \mathrm{cons}_i - \mathrm{beta\_1} \times \mathrm{standlrt}_i\right)^2}{2}\right)$$

$$\tau \sim \Gamma\left(0.001 + 0.5 \times \mathrm{length(normexam)}, 0.001000 + \frac{\Sigma_{i=1}^{\mathrm{length(normexam)}} \left(\mathrm{normexam}_i - \mathrm{beta\_0} \times \mathrm{cons}_i - \mathrm{beta\_1} \times \mathrm{standlrt}_i\right)^2}{2}\right)$$

Here we get the same line repeated twice as the second line shows the posterior after optimisation (which here we have switched off). Stat-JR takes these files and converts each of them into the C++ programming language so if we look at the file **modelcode.cpp** we will see the actual C++ code constructed below (note we will not go into detail as to how this is achieved):



Model iteration code

```
Py_BEGIN_ALLOW_THREADS;
std::vector<double*> tmp_y;
tmp_y.push_back(const_cast<double *>(normexam));
RectMatrix mat_y(tmp_y, 4059);
std::vector<double*> tmp_x;
tmp_x.push_back(const_cast<double *>(cons));
tmp_x.push_back(const_cast<double *>(standlrt));
RectMatrix mat_x(tmp_x, 4059);
std::vector<double*> tmp_beta;
tmp_beta.push_back(beta);
RectMatrix mat_beta(tmp_beta, 2);
double &beta_0 = beta[0];
double &beta_1 = beta[1];
static std::unordered_map<std::string, rng_t> rngstate;
rng_t rng;
if (runstate ==0) {
        rng=rng_create(1,1);
        rng_set_seed(rng, seed);
        rng_start(rng);
        rngstate[rngid]=rng;
} else {
        rng=rngstate[rngid];
}
for (int iter = 0; iter < numiter; iter++) {
        int iterind = 0;
        if (runstate == 3) iterind = start_iteration + floor(iter / thinning);
        // Update tau
// This code was generated by the Stat-JR package (copyright 2012 University of Bristol and University of Southampton).

    {

                double sum0=0;
                double csum0=0;
                for(int i=0; i<4059; i++) {
                        double ysum0 = pow(((normexam[i]-(beta_0*cons[i]))-(beta_1*standlrt[i])),2.0) - csum0;
                        double tsum0 = sum0 + ysum0;
                        csum0 = (tsum0 - sum0) - ysum0;
                        sum0 = tsum0;
                }


        tau = dgamma((0.001+(0.5*4059)),(0.001000+(sum0/2)));


    }
```

Here after some code that is required for passing the variables back and fore from Python to C++ we see the step for tau. This is similar to that given in the algebra. One difference is that the length of normexam has its value (4059) substituted in. The code also uses a technique called Kahan summation and so what would have been the line

```
sum0 = pow(((normexam[i]-(beta0*cons[i]))-(beta1*standlrt[i])),2);
```

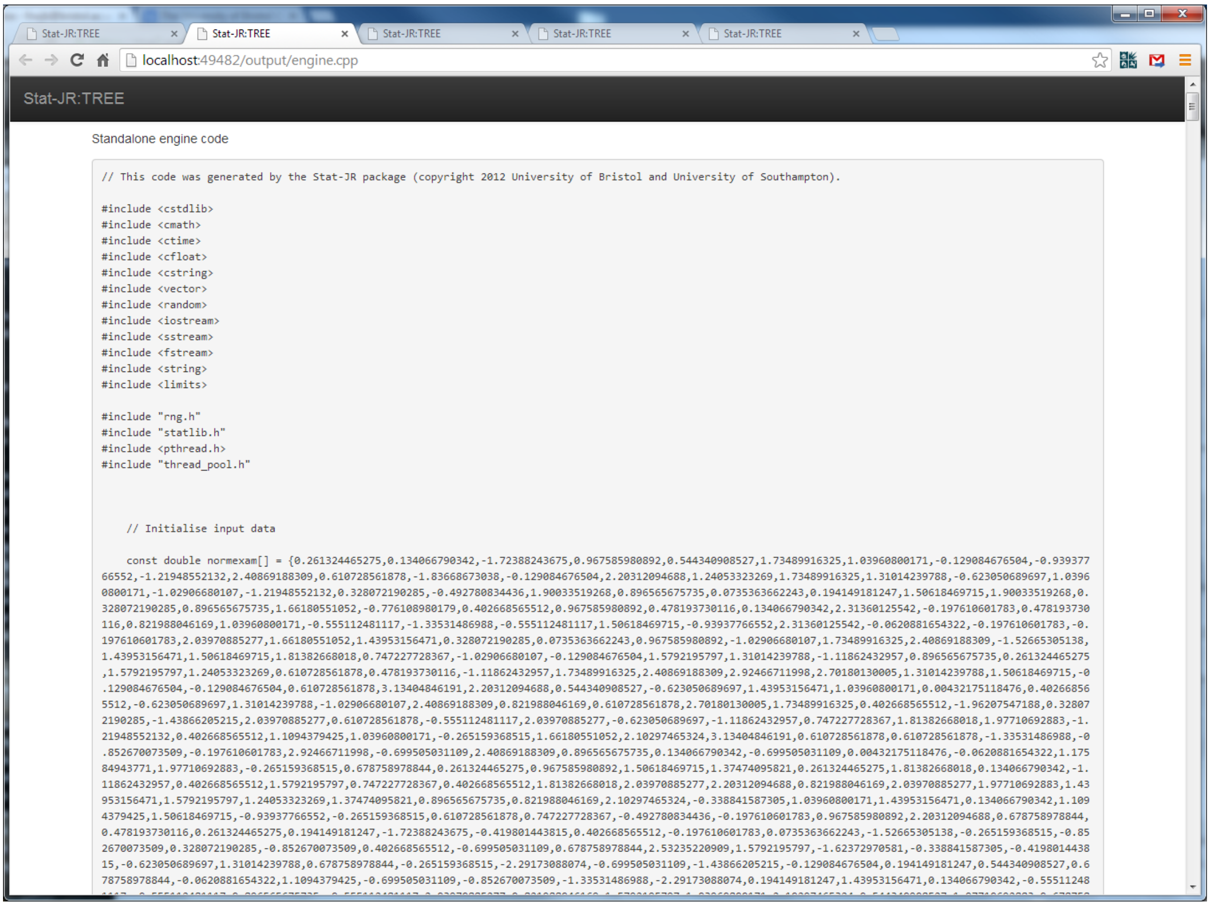
is expanded to the following

```
double ysum0 = pow(((normexam[i]-(beta0*cons[i]))-(beta1*standlrt[i])),2) - 
csum0;
                        double tsum0 = sum0 + ysum0;
                        csum0 = (tsum0 - sum0) - ysum0;
                        sum0 = tsum0;
```

to deal with potential rounding issues.

If you scroll down you will see similar code to perform the steps for the other parameters and the deviance. There are further C++ files which contain supporting routines (*supportcode.cpp* – note this used to contain random number generators but they are now included via a library instead), perform the DIC calculation (*dic.cpp*) and set up proposal distributions via adaptation when using Metropolis Hastings sampling but not in this example (*adapt.cpp*).

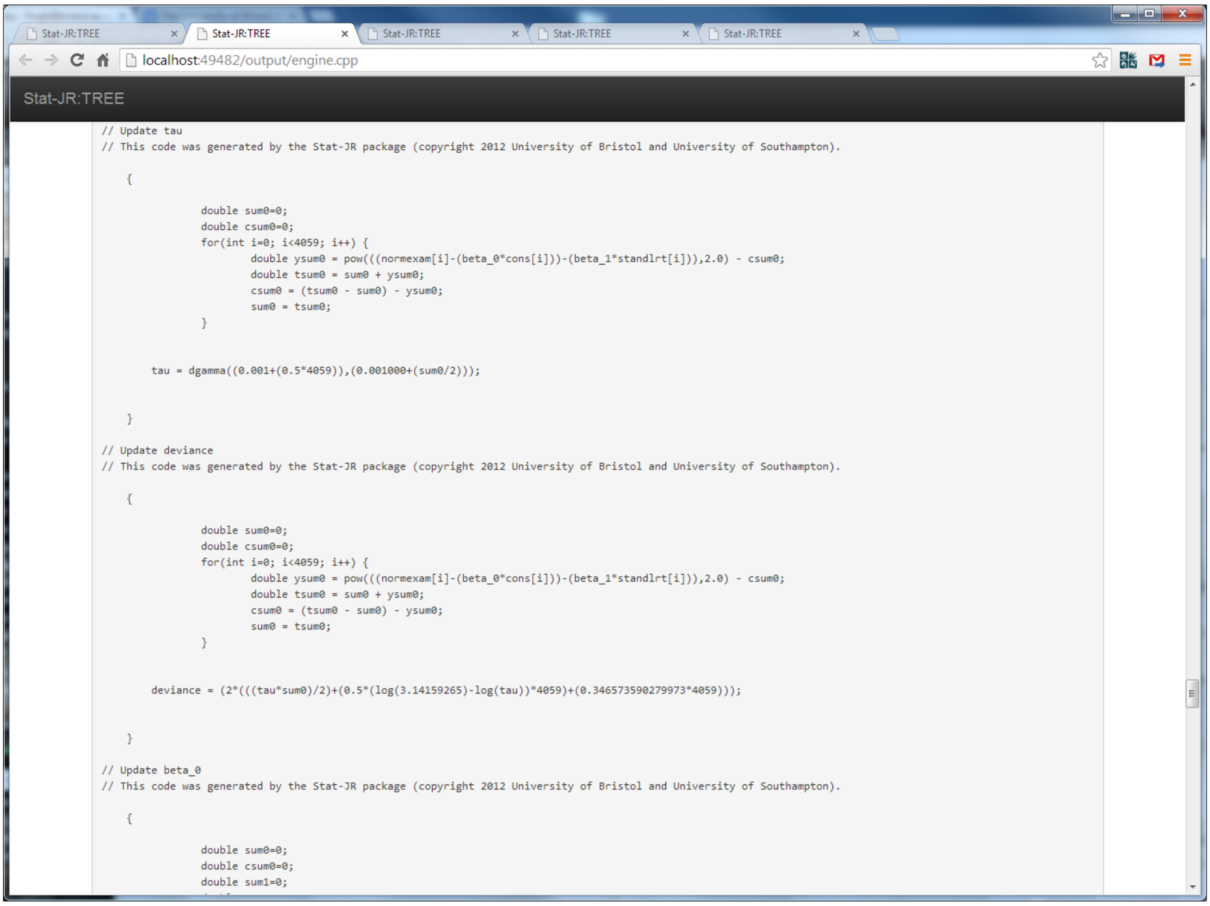
When run in the usual way i.e. without switching settings to *run as standalone* each of these pieces of C code is compiled separately and Python code within Stat-JR pieces everything together. If as we have done we choose *run as standalone* and now click on **Run** then the software does as it suggests and creates standalone C++ files. In the current version of Stat-JR we have included parallel processing and so only one standalone file is constructed, engine.cpp which contains the starting values for all three chains. If we look at **engine.cpp** in a new tab we see the following:



Standalone engine code

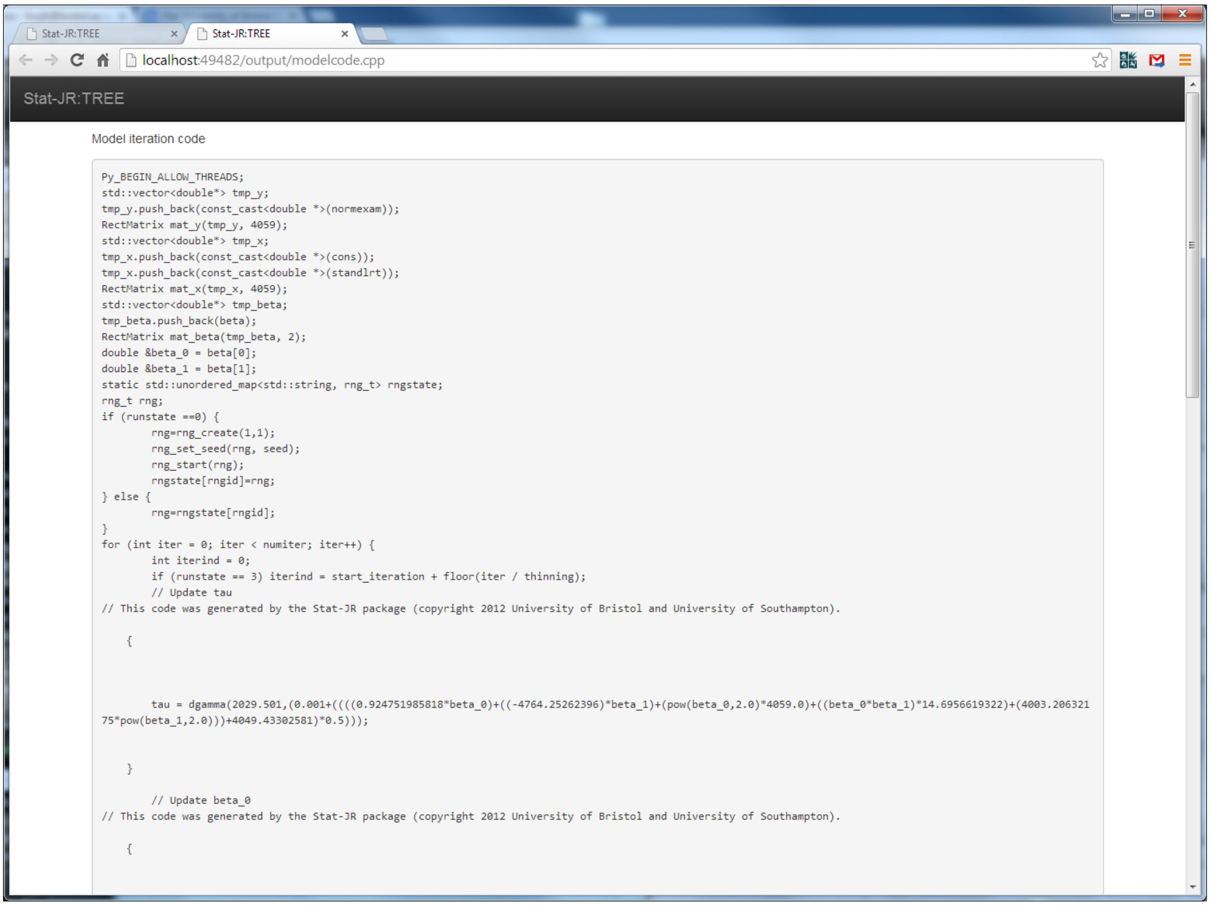
```
// This code was generated by the Stat-JR package (copyright 2012 University of Bristol and University of Southampton).

#include <cstdlib>
#include <cmath>
#include <ctime>
#include <cfloat>
#include <cstring>
#include <vector>
#include <random>
#include <iostream>
#include <sstream>
#include <fstream>
#include <string>
#include <limits>

#include "rng.h"
#include "statlib.h"
#include <pthread.h>
#include "thread_pool.h"



    // Initialise input data

    const double normexam[] = {0.261324465275,0.134066790342,-1.72388243675,0.967585980892,0.544340908527,1.73489916325,1.03960800171,-0.129084676504,-0.939377
66552,-1.21948552132,2.40869188309,0.610728561878,-1.83668673038,-0.129084676504,2.20312094688,1.24053323269,1.73489916325,1.31014239788,-0.623050689697,1.0396
0800171,-1.02906680107,-1.21948552132,0.328072190285,-0.492780834436,1.90033519268,0.896565675735,0.0735363662243,0.194149181247,1.50618469715,1.90033519268,0.
328072190285,0.896565675735,1.66180551052,-0.776108980179,0.402668565512,0.967585980892,0.478193730116,0.134066790342,2.31360125542,-0.197610601783,0.478193730
116,0.821988046169,1.03960800171,-0.555112481117,-1.33531486988,-0.555112481117,1.50618469715,-0.93937766552,2.31360125542,-0.0620881654322,-0.197610601783,-0.
197610601783,2.03970885277,1.66180551052,1.43953156471,0.328072190285,0.0735363662243,0.967585980892,-1.02906680107,1.73489916325,2.40869188309,-1.52665305138,
1.43953156471,1.50618469715,1.81382668018,0.747227728367,-1.02906680107,-0.129084676504,1.5792195797,1.31014239788,-1.11862432957,0.896565675735,0.261324465275
,1.5792195797,1.24053323269,0.610728561878,0.478193730116,-1.11862432957,1.73489916325,2.40869188309,2.92466711998,2.70180130005,1.31014239788,1.50618469715,-0
.129084676504,-0.129084676504,0.610728561878,3.13404846191,2.20312094688,0.544340908527,-0.623050689697,1.43953156471,1.03960800171,0.00432175118476,0.40266856
5512,-0.623050689697,1.31014239788,-1.02906680107,2.40869188309,0.821988046169,0.610728561878,2.70180130005,1.73489916325,0.402668565512,-1.96207547188,0.32807
2190285,-1.43866205215,2.03970885277,0.610728561878,-0.555112481117,2.03970885277,-0.623050689697,-1.11862432957,0.747227728367,1.81382668018,1.97710692883,-1.
21948552132,0.402668565512,1.1094379425,1.03960800171,-0.265159368515,1.66180551052,2.10297465324,3.13404846191,0.610728561878,0.610728561878,-1.33531486988,-0
.852670073509,-0.197610601783,2.92466711998,-0.699505031109,2.40869188309,0.896565675735,0.134066790342,-0.699505031109,0.00432175118476,-0.0620881654322,1.175
84943771,1.97710692883,-0.265159368515,0.678758978844,0.261324465275,0.967585980892,1.50618469715,1.37474095821,0.261324465275,1.81382668018,0.134066790342,-1.
11862432957,0.402668565512,1.5792195797,0.747227728367,0.402668565512,1.81382668018,2.03970885277,2.20312094688,0.821988046169,2.03970885277,1.97710692883,1.43
953156471,1.5792195797,1.24053323269,1.37474095821,0.896565675735,0.821988046169,2.10297465324,-0.338841587305,1.03960800171,1.43953156471,0.134066790342,1.109
4379425,1.50618469715,-0.93937766552,-0.265159368515,0.610728561878,0.747227728367,-0.492780834436,-0.197610601783,0.967585980892,2.20312094688,0.678758978844,
0.478193730116,0.261324465275,0.194149181247,-1.72388243675,-0.419801443815,0.402668565512,-0.197610601783,0.0735363662243,-1.52665305138,-0.265159368515,-0.85
2670073509,0.328072190285,-0.852670073509,0.402668565512,-0.699505031109,0.678758978844,2.53235220909,1.5792195797,-1.62372970581,-0.338841587305,-0.4198014438
15,-0.623050689697,1.31014239788,0.678758978844,-0.265159368515,-2.29173088074,-0.699505031109,-1.43866205215,-0.129084676504,0.194149181247,0.544340908527,0.6
78758978844,-0.0620881654322,1.1094379425,-0.699505031109,-0.852670073509,-1.33531486988,-2.29173088074,0.194149181247,1.43953156471,0.134066790342,-0.55511248
```

This code contains everything and if you scroll down to near the bottom you will find the code to update the parameters:



```
// Update tau
// This code was generated by the Stat-JR package (copyright 2012 University of Bristol and University of Southampton).

    {

                double sum0=0;
                double csum0=0;
                for(int i=0; i<4059; i++) {
                        double ysum0 = pow(((normexam[i]-(beta_0*cons[i]))-(beta_1*standlrt[i])),2.0) - csum0;
                        double tsum0 = sum0 + ysum0;
                        csum0 = (tsum0 - sum0) - ysum0;
                        sum0 = tsum0;
                }


        tau = dgamma((0.001+(0.5*4059)),(0.001000+(sum0/2)));


    }

// Update deviance
// This code was generated by the Stat-JR package (copyright 2012 University of Bristol and University of Southampton).

    {

                double sum0=0;
                double csum0=0;
                for(int i=0; i<4059; i++) {
                        double ysum0 = pow(((normexam[i]-(beta_0*cons[i]))-(beta_1*standlrt[i])),2.0) - csum0;
                        double tsum0 = sum0 + ysum0;
                        csum0 = (tsum0 - sum0) - ysum0;
                        sum0 = tsum0;
                }


        deviance = (2*(((tau*sum0)/2)+(0.5*(log(3.14159265)-log(tau))*4059)+(0.346573590279973*4059)));


    }

// Update beta_0
// This code was generated by the Stat-JR package (copyright 2012 University of Bristol and University of Southampton).

    {

                double sum0=0;
                double csum0=0;
                double sum1=0;
```

If you view the rest of this C++ code in detail you will see that there is a chunk at the top that is common to all models but the rest of the code is mostly model specific. If you return to the **Settings** screen and switch back on *optimisation* and switch off *standalone code* and press **Set** then repeating the model setup you can fit the model and view the code in **modelcode.cpp** :



Model iteration code

```
Py_BEGIN_ALLOW_THREADS;
std::vector<double*> tmp_y;
tmp_y.push_back(const_cast<double *>(normexam));
RectMatrix mat_y(tmp_y, 4059);
std::vector<double*> tmp_x;
tmp_x.push_back(const_cast<double *>(cons));
tmp_x.push_back(const_cast<double *>(standlrt));
RectMatrix mat_x(tmp_x, 4059);
std::vector<double*> tmp_beta;
tmp_beta.push_back(beta);
RectMatrix mat_beta(tmp_beta, 2);
double &beta_0 = beta[0];
double &beta_1 = beta[1];
static std::unordered_map<std::string, rng_t> rngstate;
rng_t rng;
if (runstate ==0) {
        rng=rng_create(1,1);
        rng_set_seed(rng, seed);
        rng_start(rng);
        rngstate[rngid]=rng;
} else {
        rng=rngstate[rngid];
}
for (int iter = 0; iter < numiter; iter++) {
        int iterind = 0;
        if (runstate == 3) iterind = start_iteration + floor(iter / thinning);
        // Update tau
// This code was generated by the Stat-JR package (copyright 2012 University of Bristol and University of Southampton).

    {



        tau = dgamma(2029.501,(0.001+((((0.924751985818*beta_0)+((-4764.25262396)*beta_1)+(pow(beta_0,2.0)*4059.0)+((beta_0*beta_1)*14.6956619322)+(4003.206321
75*pow(beta_1,2.0)))+4049.43302581)*0.5)));


    }

        // Update beta_0
// This code was generated by the Stat-JR package (copyright 2012 University of Bristol and University of Southampton).

    {
```

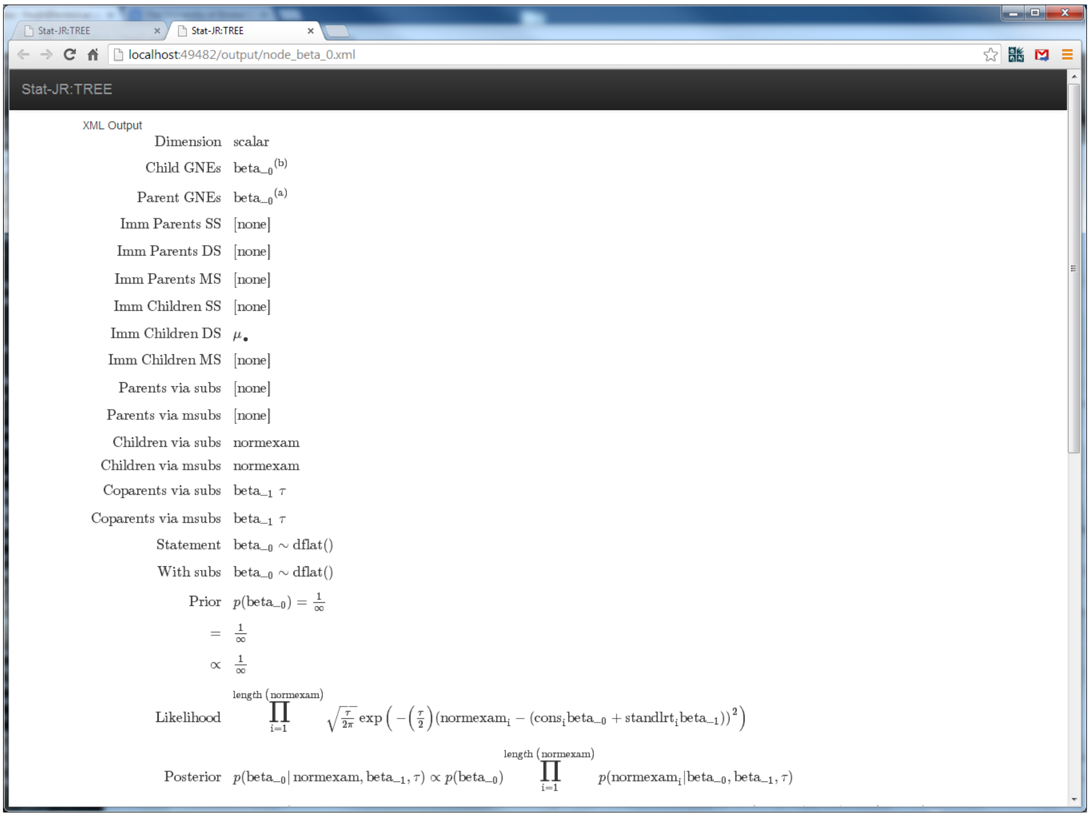
Here the code is much harder to link to the algebra system as the data has been included into the model steps and any constants have thus been evaluated. You might like to compare the code for the tau step and see if you can spot the links, for example 2029.501 is 4059/2 + 0.001. Our advice is that if you are interested in understanding the C++ code and the algorithm generally then it is probably easier to switch off optimisation whereas if you want the code to run faster then switch it on. We will revisit the C++ code in later sections when we introduce the use of the *preccode* method.

We will next look at how the algebra system converts the model statements into a set of steps in more detail.

4.2 The algebraic software system

The algebra system that we have developed for the Stat-JR system (with main developer Bruce Cameron) will take a WinBUGS like model file and produce output xml format files for each parameter. This will consist of their full conditional posterior distribution either as a known distribution with formula or as an unknown distribution function. In version 1.0 of the software we have integrated the algebra system with the main software and so we can view some of the intermediate files that show how the algebra system works. If you have run the model that we saw above with the *Regression1* template then the intermediate algebra system steps are included as xml format files.

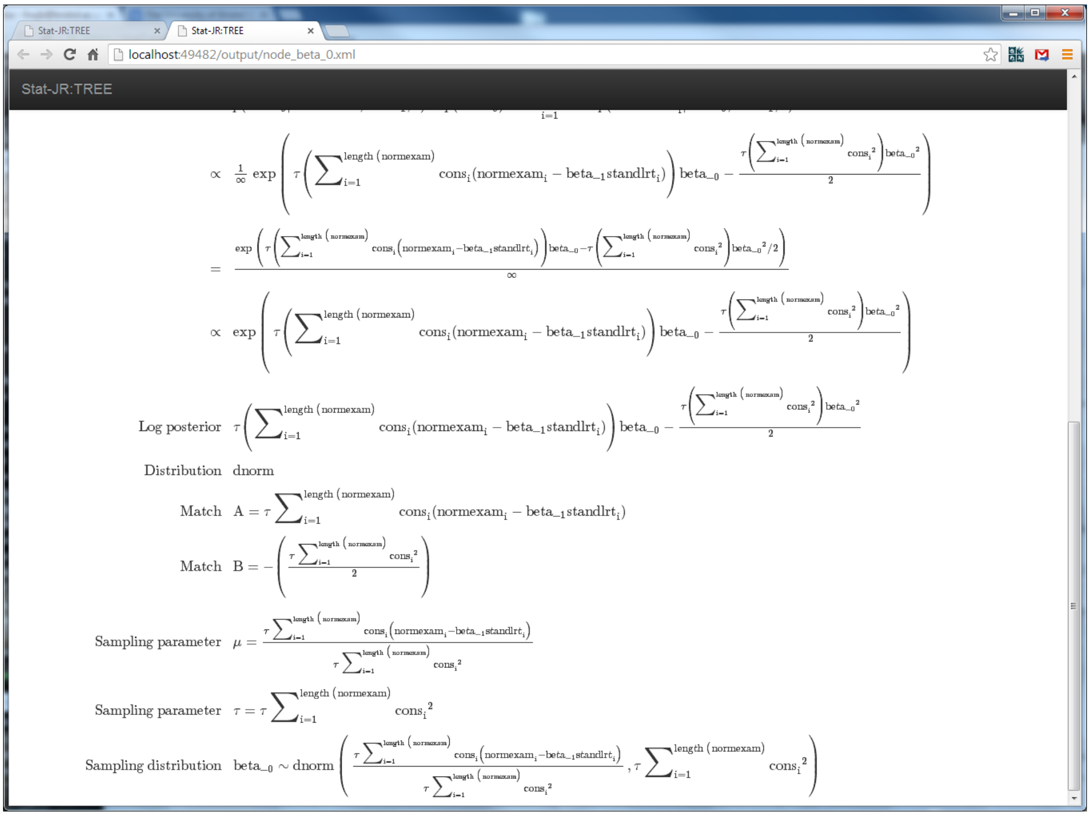
If we select *node_beta_0.xml* from the pull down list and pop it out we get the following:



XML Output

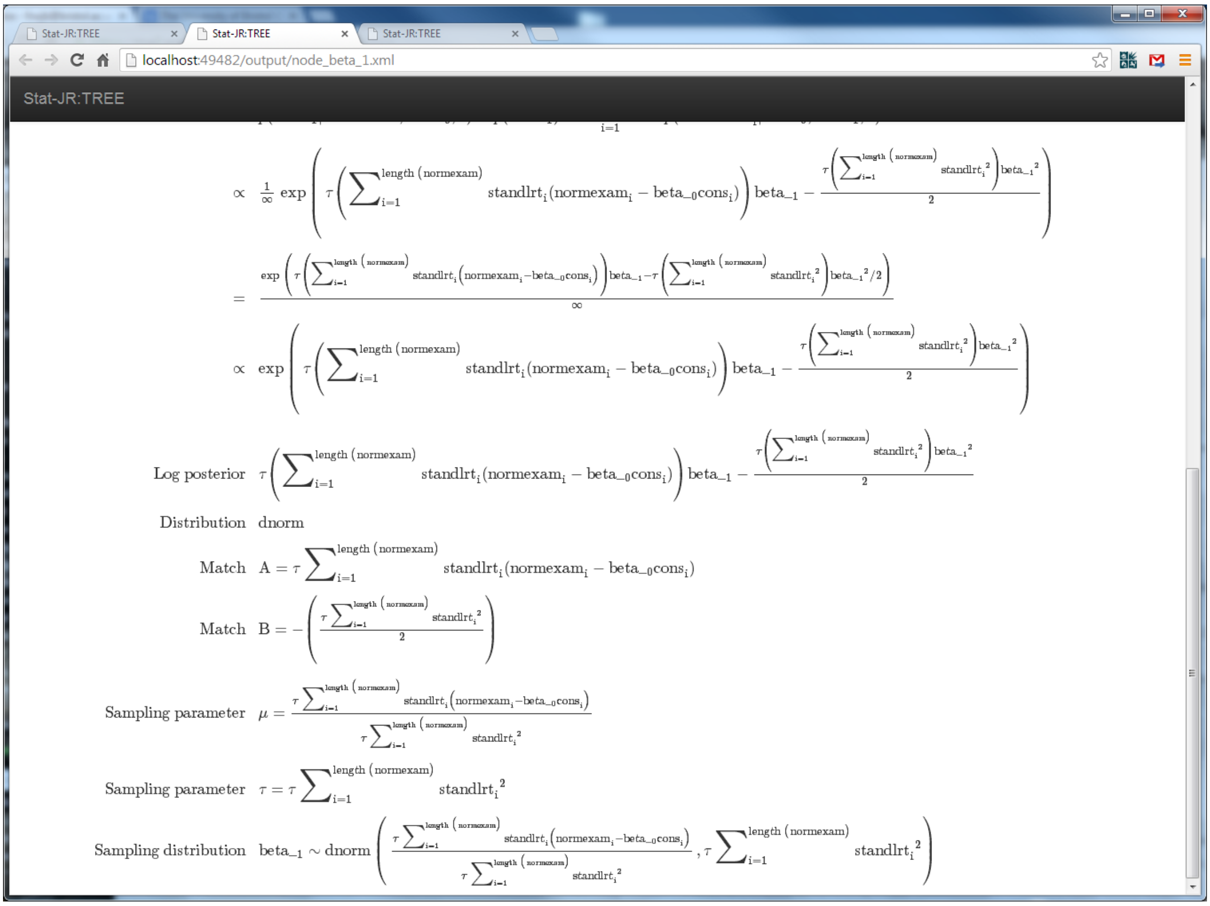
Dimension	scalar
Child GNEs	$\text{beta\_}_0{}^{(b)}$
Parent GNEs	$\text{beta\_}_0{}^{(a)}$
Imm Parents SS	[none]
Imm Parents DS	[none]
Imm Parents MS	[none]
Imm Children SS	[none]
Imm Children DS	$\mu_\bullet$
Imm Children MS	[none]
Parents via subs	[none]
Parents via msubs	[none]
Children via subs	normexam
Children via msubs	normexam
Coparents via subs	$\text{beta\_}_1\ \tau$
Coparents via msubs	$\text{beta\_}_1\ \tau$
Statement	$\text{beta\_}_0 \sim \text{dflat}()$
With subs	$\text{beta\_}_0 \sim \text{dflat}()$
Prior	$p(\text{beta\_}_0) = \frac{1}{\infty}$
$=$	$\frac{1}{\infty}$
$\propto$	$\frac{1}{\infty}$
Likelihood	$\prod_{i=1}^{\text{length}(\text{normexam})} \sqrt{\frac{\tau}{2\pi}} \exp\left(-\left(\frac{\tau}{2}\right)(\text{normexam}_i - (\text{cons}_i\text{beta\_}_0 + \text{standlrt}_i\text{beta\_}_1))^2\right)$
Posterior	$p(\text{beta\_}_0 \mid \text{normexam}, \text{beta\_}_1, \tau) \propto p(\text{beta\_}_0) \prod_{i=1}^{\text{length}(\text{normexam})} p(\text{normexam}_i \mid \text{beta\_}_0, \text{beta\_}_1, \tau)$

Here we see some algebraic processing and if we scroll to the bottom of the window we get the remainder:



$\propto$	$\frac{1}{\infty} \exp\left(\tau\left(\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i(\text{normexam}_i - \text{beta\_}_1\text{standlrt}_i)\right)\text{beta\_}_0 - \frac{\tau\left(\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i{}^2\right)\text{beta\_}_0{}^2}{2}\right)$
$=$	$\frac{\exp\left(\tau\left(\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i\left(\text{normexam}_i - \text{beta\_}_1\text{standlrt}_i\right)\right)\text{beta\_}_0 - \tau\left(\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i{}^2\right)\text{beta\_}_0{}^2/2\right)}{\infty}$
$\propto$	$\exp\left(\tau\left(\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i(\text{normexam}_i - \text{beta\_}_1\text{standlrt}_i)\right)\text{beta\_}_0 - \frac{\tau\left(\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i{}^2\right)\text{beta\_}_0{}^2}{2}\right)$
Log posterior	$\tau\left(\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i(\text{normexam}_i - \text{beta\_}_1\text{standlrt}_i)\right)\text{beta\_}_0 - \frac{\tau\left(\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i{}^2\right)\text{beta\_}_0{}^2}{2}$
Distribution	dnorm
Match	$\text{A} = \tau\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i(\text{normexam}_i - \text{beta\_}_1\text{standlrt}_i)$
Match	$\text{B} = -\left(\frac{\tau\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i{}^2}{2}\right)$
Sampling parameter	$\mu = \frac{\tau\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i\left(\text{normexam}_i - \text{beta\_}_1\text{standlrt}_i\right)}{\tau\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i{}^2}$
Sampling parameter	$\tau = \tau\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i{}^2$
Sampling distribution	$\text{beta\_}_0 \sim \text{dnorm}\left(\frac{\tau\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i\left(\text{normexam}_i - \text{beta\_}_1\text{standlrt}_i\right)}{\tau\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i{}^2}, \tau\sum_{i=1}^{\text{length}(\text{normexam})} \text{cons}_i{}^2\right)$

The program decides which lines in the model specification involve the parameter *beta0*. It then finds the prior and likelihood parts of the model for this parameter before merging them together to find the posterior and log posterior as a product of distributions. It then attempts to match the distribution to known statistical distributions and here spots that the posterior for *beta0* is a normal distribution. Finally it gives the conditional posterior distribution in terms of other objects in the model. We can view *beta1* via *node_beta_1.xml* and see similarly a Normal posterior:



$$\propto \frac{1}{\infty}\exp\left(\tau\left(\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i(\mathrm{normexam}_i-\mathrm{beta\_}_0\mathrm{cons}_i)\right)\mathrm{beta\_}_1-\frac{\tau\left(\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i{}^2\right)\mathrm{beta\_}_1{}^2}{2}\right)$$

$$=\frac{\exp\left(\tau\left(\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i\left(\mathrm{normexam}_i-\mathrm{beta\_}_0\mathrm{cons}_i\right)\right)\mathrm{beta\_}_1-\tau\left(\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i{}^2\right)\mathrm{beta\_}_1{}^2/2\right)}{\infty}$$

$$\propto \exp\left(\tau\left(\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i(\mathrm{normexam}_i-\mathrm{beta\_}_0\mathrm{cons}_i)\right)\mathrm{beta\_}_1-\frac{\tau\left(\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i{}^2\right)\mathrm{beta\_}_1{}^2}{2}\right)$$

Log posterior $\tau\left(\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i(\mathrm{normexam}_i-\mathrm{beta\_}_0\mathrm{cons}_i)\right)\mathrm{beta\_}_1-\frac{\tau\left(\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i{}^2\right)\mathrm{beta\_}_1{}^2}{2}$

Distribution dnorm

Match $\mathrm{A}=\tau\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i(\mathrm{normexam}_i-\mathrm{beta\_}_0\mathrm{cons}_i)$

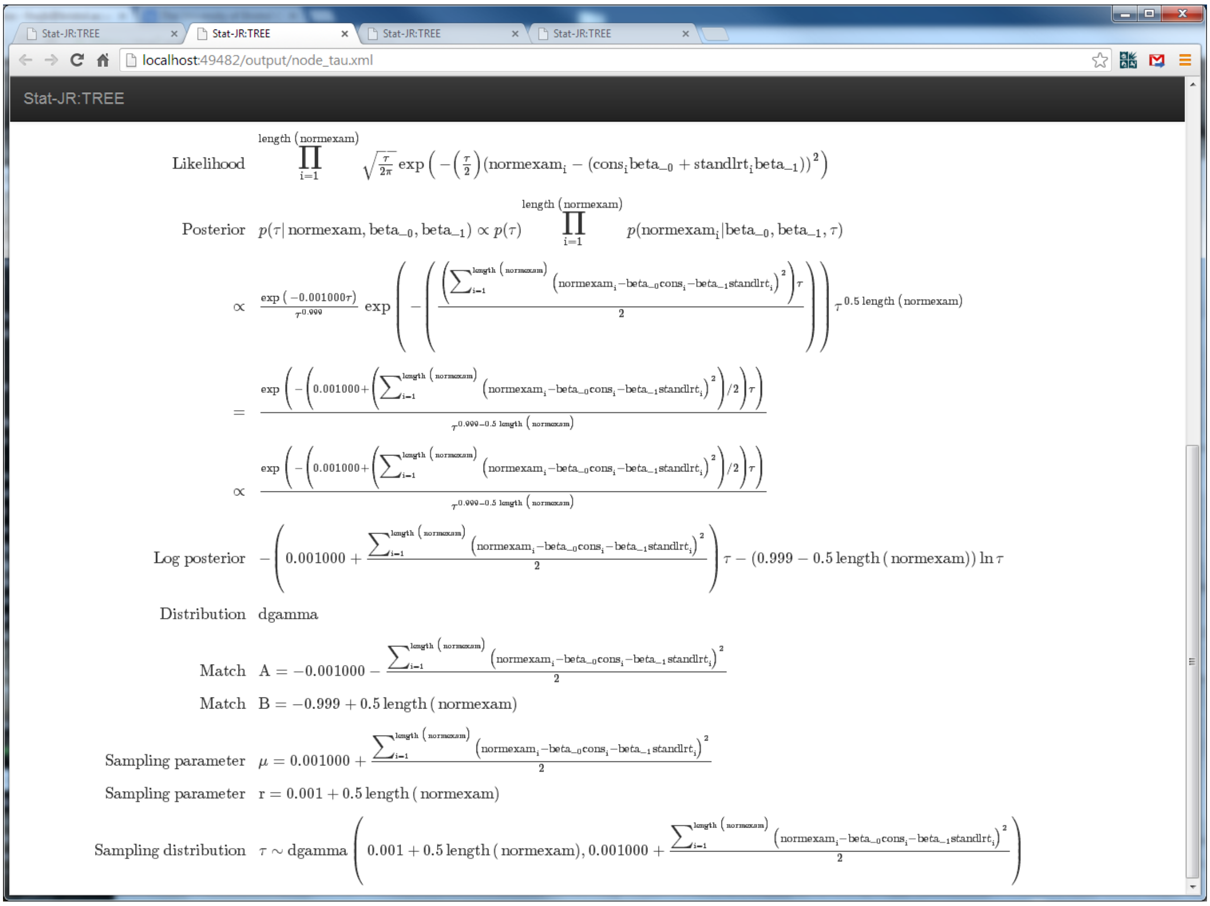
Match $\mathrm{B}=-\left(\frac{\tau\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i{}^2}{2}\right)$

Sampling parameter $\mu=\frac{\tau\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i\left(\mathrm{normexam}_i-\mathrm{beta\_}_0\mathrm{cons}_i\right)}{\tau\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i{}^2}$

Sampling parameter $\tau=\tau\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i{}^2$

Sampling distribution $\mathrm{beta\_}_1\sim\mathrm{dnorm}\left(\frac{\tau\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i\left(\mathrm{normexam}_i-\mathrm{beta\_}_0\mathrm{cons}_i\right)}{\tau\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i{}^2},\tau\sum_{i=1}^{\mathrm{length}\,(\mathrm{normexam})}\mathrm{standlrt}_i{}^2\right)$

Finally *tau* the precision has a Gamma posterior distribution with calculations shown in *node_tau.xml*:



$$\text{Likelihood}\quad \prod_{\text{i}=1}^{\text{length}\,(\text{normexam})} \sqrt{\frac{\tau}{2\pi}}\exp\left(-\left(\frac{\tau}{2}\right)(\text{normexam}_{\text{i}}-(\text{cons}_{\text{i}}\text{beta\_}_{0}+\text{standlrt}_{\text{i}}\text{beta\_}_{1}))^{2}\right)$$

$$\text{Posterior}\quad p(\tau|\,\text{normexam},\text{beta\_}_{0},\text{beta\_}_{1})\propto p(\tau)\prod_{\text{i}=1}^{\text{length}\,(\text{normexam})} p(\text{normexam}_{\text{i}}|\text{beta\_}_{0},\text{beta\_}_{1},\tau)$$

$$\propto\ \frac{\exp\,(-0.001000\tau)}{\tau^{0.999}}\exp\left(-\left(\frac{\left(\sum_{\text{i}=1}^{\text{length}\,(\text{normexam})}\left(\text{normexam}_{\text{i}}-\text{beta\_}_{0}\text{cons}_{\text{i}}-\text{beta\_}_{1}\text{standlrt}_{\text{i}}\right)^{2}\right)\tau}{2}\right)\right)\tau^{0.5\,\text{length}\,(\text{normexam})}$$

$$=\ \frac{\exp\left(-\left(0.001000+\left(\sum_{\text{i}=1}^{\text{length}\,(\text{normexam})}\left(\text{normexam}_{\text{i}}-\text{beta\_}_{0}\text{cons}_{\text{i}}-\text{beta\_}_{1}\text{standlrt}_{\text{i}}\right)^{2}\right)/2\right)\tau\right)}{\tau^{0.999-0.5\,\text{length}\,(\text{normexam})}}$$

$$\propto\ \frac{\exp\left(-\left(0.001000+\left(\sum_{\text{i}=1}^{\text{length}\,(\text{normexam})}\left(\text{normexam}_{\text{i}}-\text{beta\_}_{0}\text{cons}_{\text{i}}-\text{beta\_}_{1}\text{standlrt}_{\text{i}}\right)^{2}\right)/2\right)\tau\right)}{\tau^{0.999-0.5\,\text{length}\,(\text{normexam})}}$$

$$\text{Log posterior}\quad -\left(0.001000+\frac{\sum_{\text{i}=1}^{\text{length}\,(\text{normexam})}\left(\text{normexam}_{\text{i}}-\text{beta\_}_{0}\text{cons}_{\text{i}}-\text{beta\_}_{1}\text{standlrt}_{\text{i}}\right)^{2}}{2}\right)\tau-(0.999-0.5\,\text{length}\,(\text{normexam}))\ln\tau$$

$$\text{Distribution}\quad \text{dgamma}$$

$$\text{Match}\quad \text{A}=-0.001000-\frac{\sum_{\text{i}=1}^{\text{length}\,(\text{normexam})}\left(\text{normexam}_{\text{i}}-\text{beta\_}_{0}\text{cons}_{\text{i}}-\text{beta\_}_{1}\text{standlrt}_{\text{i}}\right)^{2}}{2}$$

$$\text{Match}\quad \text{B}=-0.999+0.5\,\text{length}\,(\text{normexam})$$

$$\text{Sampling parameter}\quad \mu=0.001000+\frac{\sum_{\text{i}=1}^{\text{length}\,(\text{normexam})}\left(\text{normexam}_{\text{i}}-\text{beta\_}_{0}\text{cons}_{\text{i}}-\text{beta\_}_{1}\text{standlrt}_{\text{i}}\right)^{2}}{2}$$

$$\text{Sampling parameter}\quad \text{r}=0.001+0.5\,\text{length}\,(\text{normexam})$$

$$\text{Sampling distribution}\quad \tau\sim\text{dgamma}\left(0.001+0.5\,\text{length}\,(\text{normexam}),0.001000+\frac{\sum_{\text{i}=1}^{\text{length}\,(\text{normexam})}\left(\text{normexam}_{\text{i}}-\text{beta\_}_{0}\text{cons}_{\text{i}}-\text{beta\_}_{1}\text{standlrt}_{\text{i}}\right)^{2}}{2}\right)$$

The algebraic processing software then saves these three final distributions in XML file format (*beta_0.xml*, *beta_1.xml* and *tau.xml*) so that they can be read in later when we create code to fit the model. These files are then used to generate the C++ code to fit the model via a code generator as explained earlier.

5 Including Interoperability

We have now seen that the Stat-JR package has its own new algebra system and estimation engine, known as eStat as illustrated in the last section. Another aspect of the package is its ability to interface with other software packages and in particular (but not exclusively) their estimation engines. This feature doesn’t however come for free and translator methods that are often template specific need writing to achieve interoperability. The work here can be broken down into generic work that is built into the software and includes interfacing with the external software and managing the output received, and other work such as construction of data and script files for the external package that may be template specific and thus written by the template writer or generic as well. In this section we will describe the (generic) Python code that is written to support interoperability and found in the *packages* subdirectory of Stat-JR. We will return to the regression modelling template and take a look at how we can include interoperability via an adapted template (*Regression2.py*). We will here describe work on three of the software packages that have been considered for interoperability, namely WinBUGS, MLwiN, and R but first we will delve a little further into the workings of the eStat engine and look at the file *eStat.py*.

5.1 eStat.py

When running a model in Stat-JR with a specific estimation engine an object is constructed of a unique class related to that engine. These objects are what pull together inputs and data, perform the estimation and store the results. The files for the various engines are found in the *packages* subdirectory which also contains equivalent files for use with templates not related to model estimation. The object of type eSTAT is defined in the file *eStat.py* and you will see if you access this code that it is rather long and complicated. We will not here try and go through everything as this would only be useful for the most expert Python coders. There are however some commonalities across engines and so we will give very brief indications of what certain methods do in the template. Note that only some of these are externally referenced (Methodinput, init, run and runmore) whilst others are called internally as they do parts of the work of the externally referenced attributes:

- The *MethodInput* method is present in each engine and contains the engine's specific estimation method inputs that we saw when running the *Regression1* template earlier (Number of chains, Random Seed etc.).
- The *init* method is what is called after the estimation method inputs have been answered by the user and the *Next* button is pressed. It calls lots of other methods to perform the various tasks here including getting the algorithm from the algebra system and constructing the code for running the model.
- The *applydata* method is used with eStat to construct starting values for parameters in the model
- The *compilemodel* method is used to call the algebra system and get back algebraic steps for each parameter.
- The *calcconsts* method is what is run with eStat when optimisation is switched on to pull out terms in the algebra that are purely data and evaluate them.
- The *run* method is used to run the current model with the prescribed estimation settings
- and is called when the **Run** button is pressed.
- The *runmore* method is used when the **More** button has been pressed for further iterations.
- The *genCPP* method is used to generate the C++ code for the standalone engine.
- The *runCPP* method is used to run the estimation algorithm when standalone C++ code is selected.
- The *saveresults* method brings together the (potentially multi-chain) output and constructs the *ModelResults* and *output* chains objects.
- The *dic* method constructs code if required to calculate the DIC diagnostic for the model.

With regard engine classes in general we would expect to find a *MethodInput* method, an *init* method, a *run* method and often a *saveresults* method but also some engine specific methods. The *MethodInput* method always contains any additional engine specific inputs that are displayed on the screen. The *init* method contains the Python code to be run upon pressing the **Next** button prior to running and the *run* method contains the Python code to be run after pressing the **Run** button. The *saveresults* method, where present, is usually called from the *run* method. If the estimation method allows more iterations (typically packages using MCMC estimation) then there will be a *runmore* method that is called after pressing the **More** button. We will now look at a second template that contains further interoperability.

5.2 Regression2.py

In this section we will consider a second template – *Regression2* that extends the first template by including the option to fit the same model in a variety of packages. If you look at the code in the Python file you will see that this template has identical code for the attributes defined in *Regression1* but in addition has methods to allow the user to call other programs. We will begin however by looking at the *engines* attribute:

```
engines = ['eStat', 'WinBUGS', 'OpenBUGS', 'JAGS' , 'MLwiN_MCMC',
'MLwiN_IGLS', 'R_MCMCglmm', 'R_glm', 'Stata_model', 'SPSS_model',
'SAS_model', 'Minitab_model', 'SABRE', 'MATLAB_script',
'Octave_script','GenStat_model', 'gretl_model', 'Python_PyMC']
```

Here we see that this template offers very many software packages to be used. For several packages there is simply one engine whereas for MLwiN and R which we will see later there are two engines as these packages have both classical (ML) and Bayesian (MCMC) engines built-in. If you scroll down the file you will see additional attributes:

`mlnscript`, `rscript`, `statascript`, `spssscript`, `sasscript`, `minitabscript`, `sabrescript`, `matlabscript`, `genstatscript`, `gretlscript`, and `pymcscript`.

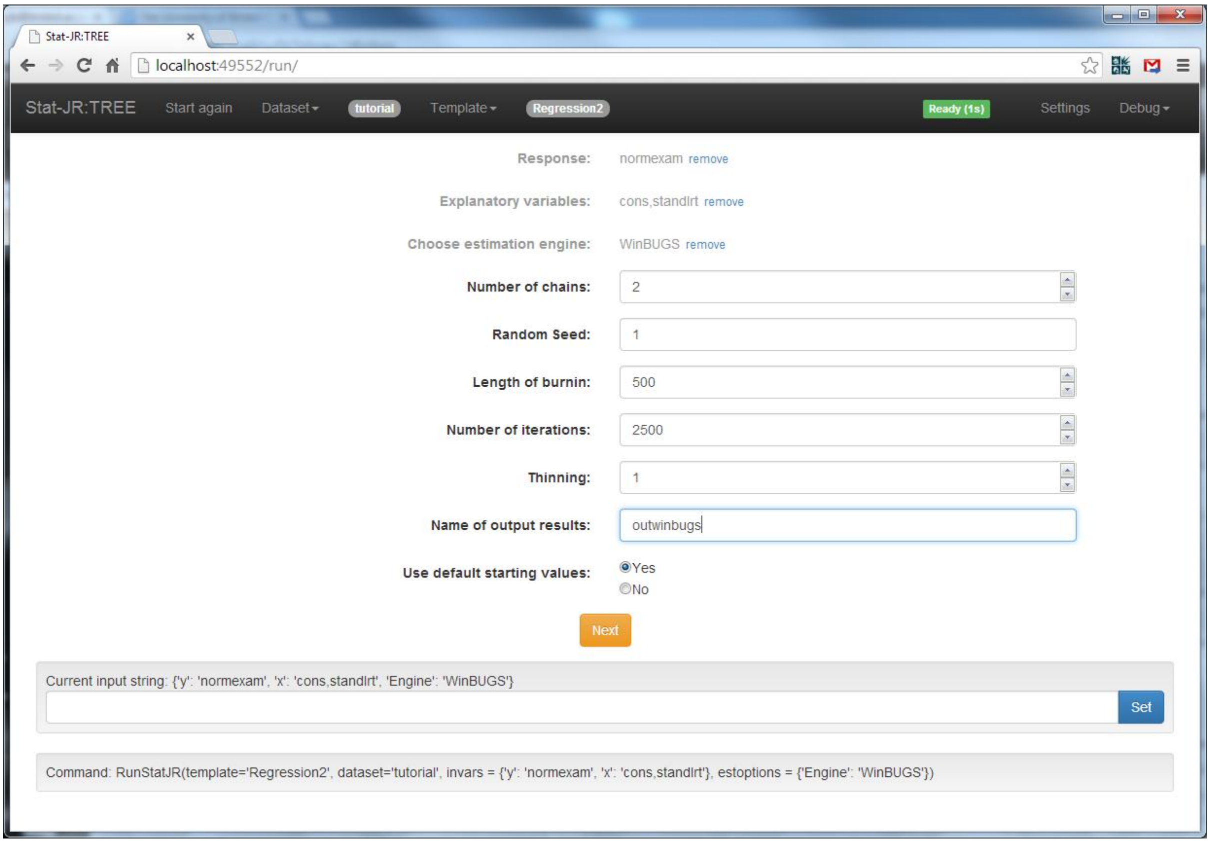
Each of these methods is used to produce scripts or parts of scripts for the specific package and the main point to take home here is that although some template specific coding is required, the code required is generally short functions (and in the case of WinBUGS, OpenBUGS and JAGS non-existent) and so the bulk of the work, at least for this template is done by the generic code within the package files for the engines.

5.3 WinBUGS and Winbugsscript.py

We will begin by looking at the WinBUGS package (Lunn et al., 2000) as the model code we have been creating for the Stat-JR engine has many similarities with WinBUGS code. We will begin by running the template and viewing the output. It should be noted that in order to run the WinBUGS engine Stat-JR needs to be able to find it. Stat-JR has a file of settings, *settings.cfg* which it will have placed in a .statjr directory under your *Users* directory. This file contains amongst other things directory names for each package. For example on my machine I have:

```
[WinBUGS]
executable=../../../WinBUGS14/WinBUGS14.exe
```

If you wish to use this option you need to either install WinBUGS in this directory or change these paths to point to WinBUGS on your machine. This can either be done by editing the file or by going to the *Settings* screen we looked at earlier and changing the file there. If you edit the file itself you will need to restart the *TREE* program or reload packages so that it uses these settings and then select *Regression2.py* from the *template* choices and *tutorial* for the *dataset*. Next select the following inputs:



Response: normexam remove

Explanatory variables: cons,standlrt remove

Choose estimation engine: WinBUGS remove

Number of chains: 2

Random Seed: 1

Length of burnin: 500

Number of iterations: 2500

Thinning: 1

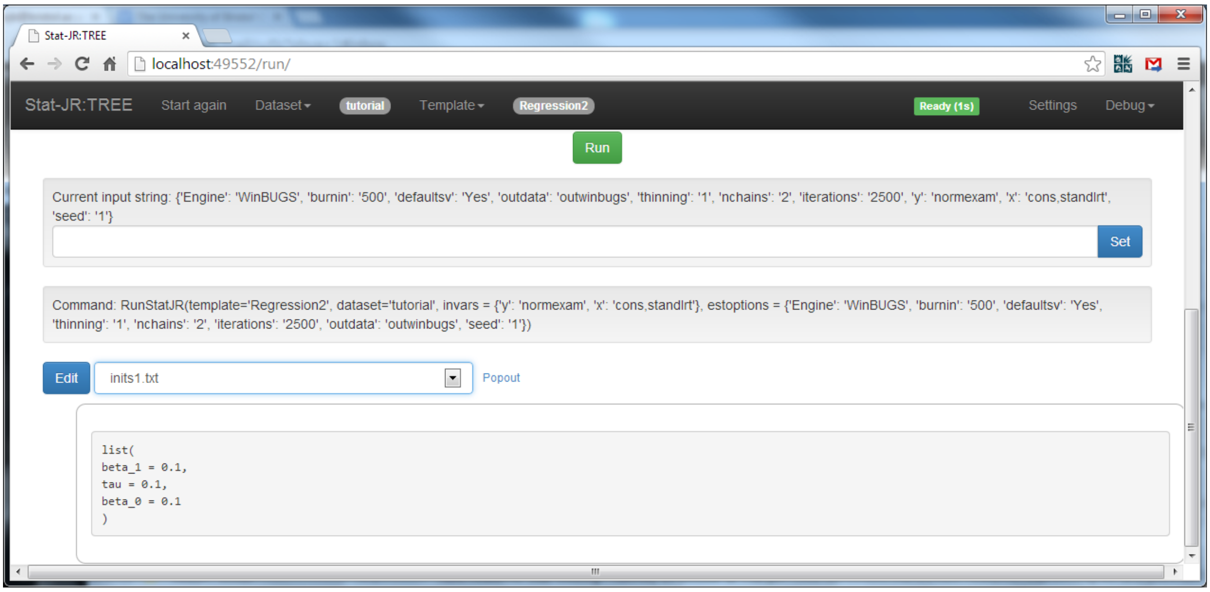
Name of output results: outwinbugs

Use default starting values: ● Yes ○ No

Current input string: {'y': 'normexam', 'x': 'cons,standlrt', 'Engine': 'WinBUGS'}

Command: RunStatJR(template='Regression2', dataset='tutorial', invars = {'y': 'normexam', 'x': 'cons,standlrt'}, estoptions = {'Engine': 'WinBUGS'})

Clicking on **Next** will cause Stat-JR to construct all the files it needs to fit the model in WinBUGS and these can be found under the pull down list. There will be Model code (*model.txt*) and the mathematical representation (*equation.txt*) as we saw for *Regression1* with the eStat engine apart from that this model code has been modified slightly to be in line with standard WinBUGS code, in this case *length(normexam)* has been replaced with *4059* in code (Note that this template also supports the eStat estimation engine). We can look at the other input files required by WinBUGS for example here is the file containing initial values for chain 1:

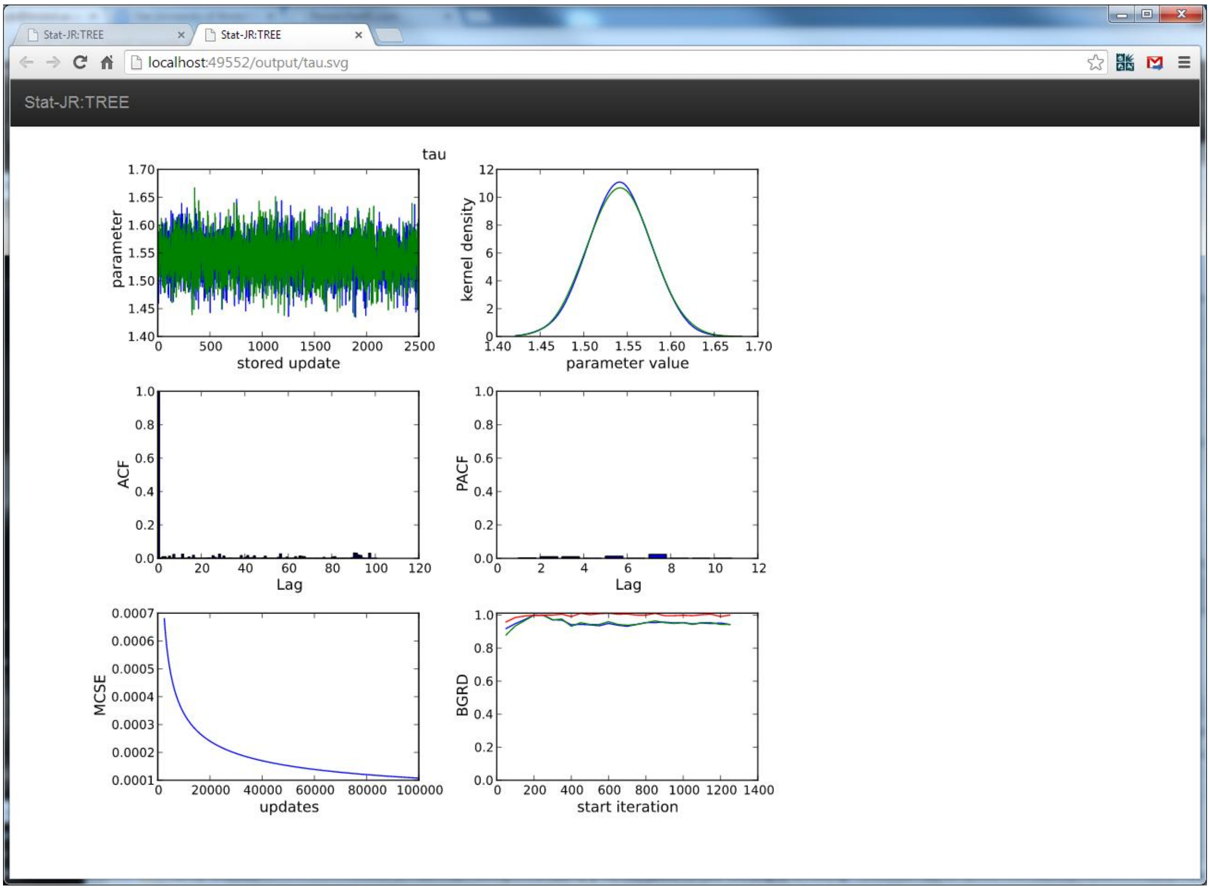


Current input string: {'Engine': 'WinBUGS', 'burnin': '500', 'defaultsv': 'Yes', 'outdata': 'outwinbugs', 'thinning': '1', 'nchains': '2', 'iterations': '2500', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1'}

Command: RunStatJR(template='Regression2', dataset='tutorial', invars = {'y': 'normexam', 'x': 'cons,standlrt'}, estoptions = {'Engine': 'WinBUGS', 'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '2', 'iterations': '2500', 'outdata': 'outwinbugs', 'seed': '1'})

```
list(
beta_1 = 0.1,
tau = 0.1,
beta_0 = 0.1
)
```

There is also a data file and a script file. If we next click on **Run** you will see a WinBUGS window appear on your toolbar and in the background whilst WinBUGS is fitting the model. When it finishes it will disappear and the list of files in the pull down list will lengthen so for example if we select *tau.svg* and pop it out we will get the following output in the browser:



As we chose 2 chains you will also observe a green and blue output for both the chains and kernel density plots. If you look at *ModelResults* you will notice that we get results for each parameter (including the deviance and some reordering of the output). We now need to see how the connection to WinBUGS was achieved. Interestingly for the *Regression2* template you will not find any additional code to run WinBUGS within the template itself apart from putting WinBUGS in the engines list. This means that all the code is generic and not template specific and will be found in the *WinBUGS.py* file within the packages directory.

As mentioned in the last section these engine files give class definitions for classes that will perform the interoperability work for specific packages and the file WinBUGS.py gives the class definition for the *WinBUGSScript* class. This class has as expected *MethodInput*, *init* and *run* methods and as WinBUGS supports running further MCMC iterations there is also a *runmore* method. The *MethodInput* method is fairly self explanatory and contains the various additional estimation method inputs required by WinBUGS along with some code to allow the user to specify their own starting values if they wish. The *init* method is split into two main parts written in two methods:

PrepareWBugsInputs which is used to create, in turn, the three files that are needed to fit a model in WinBUGS, namely, the data, initial values and model files; *WriteScript* which creates the script file that WinBUGS uses to perform the model fitting and extraction of results etc.

The *PrepareWBugsInputs* method has code that will construct the model and data files for BUGS as well as a series of initial value files. In many simple model scenarios including the regression model we have considered, the WinBUGS model file will aside from simple substitutions be identical to the input file for the eStat algebra system and so the chunk of code dealing with model construction is fairly simple and simply involves copying the *model.txt* file and making the substitutions.

If you are interested in writing templates that either only use WinBUGS or are for models where the code for WinBUGS and eStat diverges then it is possible to write methods within the template to construct the required model, data and inits files. These methods are named *bugsmodel*, *bugsdata* and *bugsinits* respectively. The *bugsmodel* method should resemble to some degree the *model* method used for eStat whilst the *bugsdata* and *bugsinits* methods will generally consist of commands to pull out or construct the various objects that make up the data and initial values. For example in the *1LevelMVNormal* template there are the lines like

`data['M'] = M` and `data['R'] = Rmat` which tells Stat-JR that there are two data objects that need adding to the WinBUGS output file. Stat-JR can evaluate that M is an integer and R is a matrix from how they have been constructed in Python and within *PrepareWBugsInputs* is code to then write these out correctly into the data file. For initial values there is a similar construction with the *data[]* construction replaced with an *inits[]* construction. Note that if you wish to write your own *bugsdata* and/or *bugsinit* methods that all data and/or parameters requiring initial values must be defined in the method.

For examples of template with their own WinBUGS functions you might look at *1LevelMVNormal* or *CapRecap.* The template *1LevelMVNormal* fits a multivariate response model and as discussed later in the manual the eStat engine has an unusual way of fitting such models and so for WinBUGS we have files for *bugsmodel* and *bugsdata* to create these files in a more standard use of the WinBUGS language. We do not have a *bugsinit* methods as the generic code works OK for creating the initial values here. The template *CapRecap* fits a capture recapture model which involves multinomial distributions where again WinBUGS and eStat diverge. Here there is code to create all three required files however the reason for the *bugsinit* attribute is primarily because we want to have a specific pattern of starting values.

The *WriteScript* function is totally generic as it creates the script file to be run in WinBUGS and this at least at present is consistent across templates.

The *run* function which is run when the **Run** button is pressed is fairly short:

```
def run(self):
        self.eng.run('script.txt')

        try:
            self.saveresults()
        except:
            logging.error('There was a problem running the model')
```

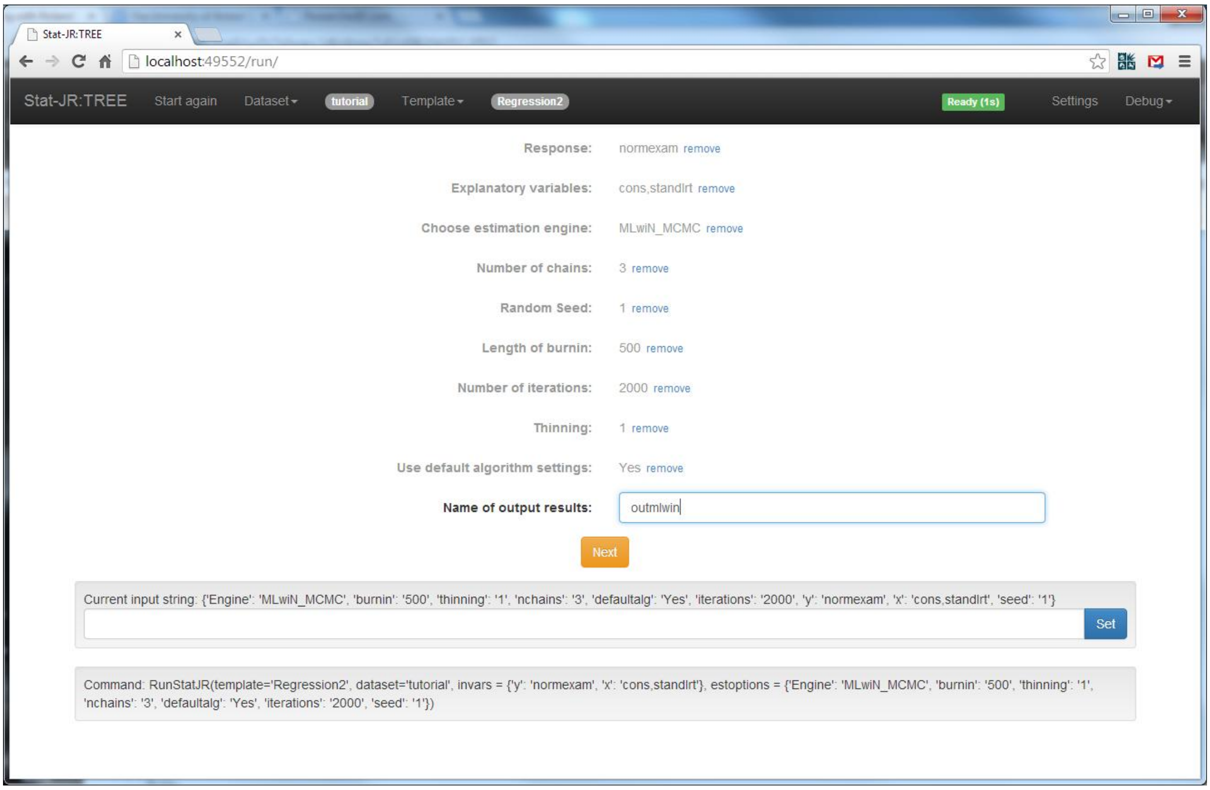
The command `self.eng.run('script.txt')` actually runs WinBUGS. The last command `self.saveresults()` both extracts the numbers from the text files returned from WinBUGS and constructs the *ModelResults* object that can be viewed. We have limited this section to a broad description of the purposes of specific functions used in the interoperability and how an advanced user if required, might write their own methods for their template. Stat-JR also supports OpenBUGS (Lunn et al., 2009) and JAGS (Plummer, 2003) which are in terms of input files similar to WinBUGS. There are differences in their script files and so the files *OpenBUGS.py* and *JAGS.py* have similar but slightly differing code to account for this. JAGS also has a slightly different format for data and initial values files which *JAGS.py* takes care of. If you are writing your own *bugsmodel*, *bugsdata* and *bugsinit* methods then these will also be used to create the model code, data and initial values in OpenBUGS and JAGS so you will not need to repeat the work. We next look at MLwiN.

5.4 MLwiN

MLwiN (Rasbash et al. 2009) is another package with MCMC functionality but which can also fit multilevel models using classical statistical methods. In Stat-JR, for the *Regression2* template, we offer the option of fitting models in MLwiN using either approach. Having seen how WinBUGS links into Stat-JR we will now show the similarities and differences in how MLwiN links in. The first observation is that MLwiN doesn't use a model description language like Stat-JR or WinBUGS. It is also more restrictive in terms of which models it can fit which means that it will not be available for all templates but many of the templates we have written thus far fit models that MLwiN can also fit. Although MLwiN has a GUI user interface which is typically how users will use it, it also has a macro language and it is this language that we have to make use of when writing interoperability code for Stat-JR. So as with WinBUGS we need to tell Stat-JR where to find MLwiN and this is found in the *settings.cfg* file, for example:

```
[MLwiN]
executable=../../../MLwiN v2.30/x64/mlnscript.exe
```

Let us demonstrate using MLwiN and MCMC for the *tutorial* dataset and *Regression2* template. Here select the *template* and *dataset* and next choose inputs as follows:



Response: normexam remove

Explanatory variables: cons,standlrt remove

Choose estimation engine: MLwiN_MCMC remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

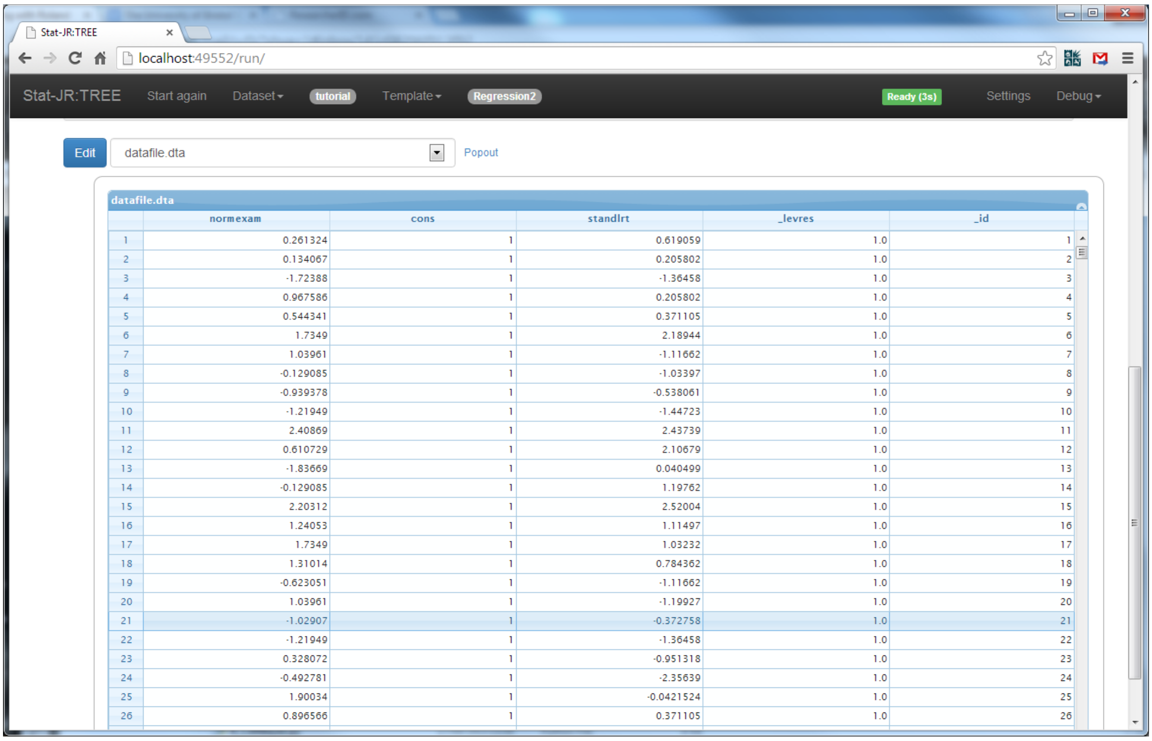
Use default algorithm settings: Yes remove

Name of output results: outmlwin

```
Current input string: {'Engine': 'MLwiN_MCMC', 'burnin': '500', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1'}
```

```
Command: RunStatJR(template='Regression2', dataset='tutorial', invars = {'y': 'normexam', 'x': 'cons,standlrt'}, estoptions = {'Engine': 'MLwiN_MCMC', 'burnin': '500', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'seed': '1'})
```

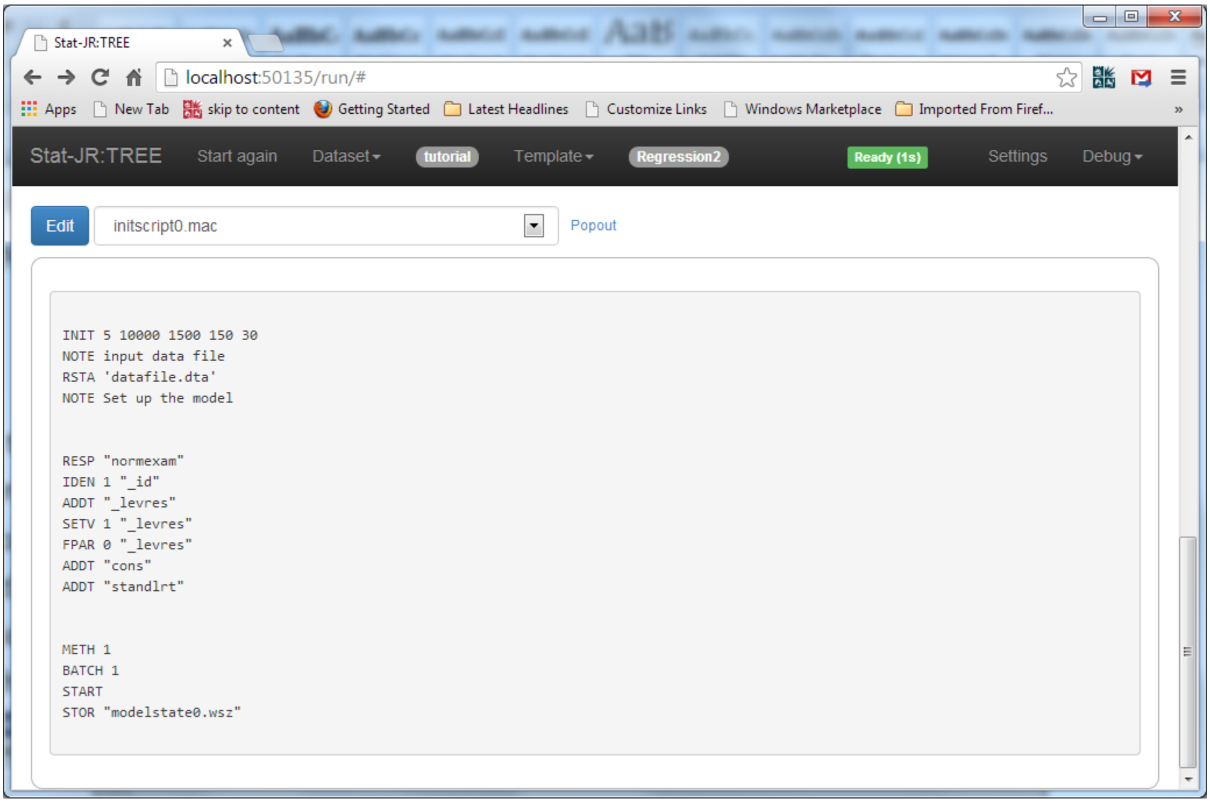
The first thing to note is that the two approaches for MLwiN have their own engine name, and we will see later their own python files in the package directory. A further thing to note is that MLwiN normally only offers single chains for MCMC. However if you run it from Stat-JR you can get the illusion of multiple chains as Stat-JR will run MLwiN several times (in parallel), once for each chain. Currently each chain has the same initial values but different random number seed but in the future we hope to allow different starting values as well. Clicking on the **Next** button we see the following:



datafile.dta

	normexam	cons	standlrt	_levres	_id
1	0.261324	1	0.619059	1.0	1
2	0.134067	1	0.205802	1.0	2
3	-1.72388	1	-1.36458	1.0	3
4	0.967586	1	0.205802	1.0	4
5	0.544341	1	0.371105	1.0	5
6	1.7349	1	2.18944	1.0	6
7	1.03961	1	-1.11662	1.0	7
8	-0.129085	1	-1.03397	1.0	8
9	-0.939378	1	-0.538061	1.0	9
10	-1.21949	1	-1.44723	1.0	10
11	2.40869	1	2.43739	1.0	11
12	0.610729	1	2.10679	1.0	12
13	-1.83669	1	0.040499	1.0	13
14	-0.129085	1	1.19762	1.0	14
15	2.20312	1	2.52004	1.0	15
16	1.24053	1	1.11497	1.0	16
17	1.7349	1	1.03232	1.0	17
18	1.31014	1	0.784362	1.0	18
19	-0.623051	1	-1.11662	1.0	19
20	1.03961	1	-1.19927	1.0	20
21	-1.02907	1	-0.372758	1.0	21
22	-1.21949	1	-1.36458	1.0	22
23	0.328072	1	-0.951318	1.0	23
24	-0.492781	1	-2.35639	1.0	24
25	1.90034	1	-0.0421524	1.0	25
26	0.896566	1	0.371105	1.0	26

Here we see the dataset file which contains only the columns in the data that are involved in the modelling in MLwiN. The list of objects has been populated by many MLwiN script files and we will look at one of these now so select *initscript0.mac* and the screen will look as follows:


Stat-JR:TREE
localhost:50135/run/#
Apps New Tab skip to content Getting Started Latest Headlines Customize Links Windows Marketplace Imported From Firef...
Stat-JR:TREE Start again Dataset tutorial Template Regression2 Ready (1s) Settings Debug
Edit
initscript0.mac
Popout

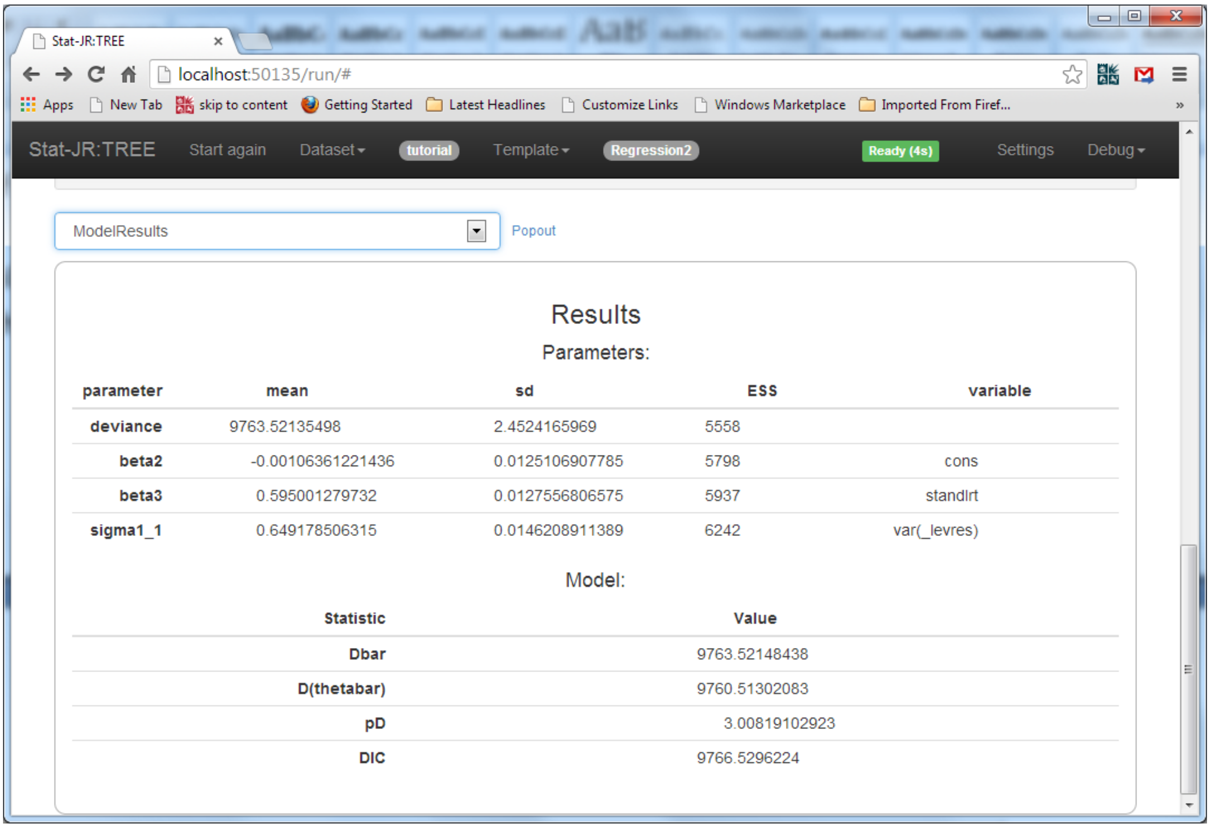
```
INIT 5 10000 1500 150 30
NOTE input data file
RSTA 'datafile.dta'
NOTE Set up the model


RESP "normexam"
IDEN 1 "_id"
ADDT "_levres"
SETV 1 "_levres"
FPAR 0 "_levres"
ADDT "cons"
ADDT "standlrt"


METH 1
BATCH 1
START
STOR "modelstate0.wsz"
```

This is the script that is run first in MLwiN and sets up the required model and runs it (using IGLS first to generate initial values). If we look in the file *Regression2.py* we can find the attribute *mlnscript* which is as follows:

```
mlnscript = '''
RESP "${y}"
IDEN 1 "_id"
ADDT "_levres"
SETV 1 "_levres"
FPAR 0 "_levres"
% for i in range(len(x)):
ADDT "${x[i]}"
% endfor
'''
```

You should be able to see that, for our inputs, if we were to expand out this Python code we would get the second section of code that appears within *initscript0.mac*. This code essentially sets up the model in MLwiN ready to be fitted. The package file *MLwin_MCMC.py* will therefore take the code that appears in *mlnscript* in the template and place it in the *initscript* macros (note there is one *initscript* for each chain). Stat-JR will also construct 3 further macros for each chain, *burninscript*, *runscript* and *resultsscript.* As MLwiN works by building up the model (as performed in *initscript*) these further scripts perform the burnin iterations, main run iterations and results extraction respectively and are generic. They only depend on the estimation method inputs i.e. length of burnin, number of iterations, thinning etc. and each chain has a different random seed set.

Clicking on **Run** will fire off the three instances of MLwiN and bring back the output as follows (after changing the output list to show **ModelResults**):



Results

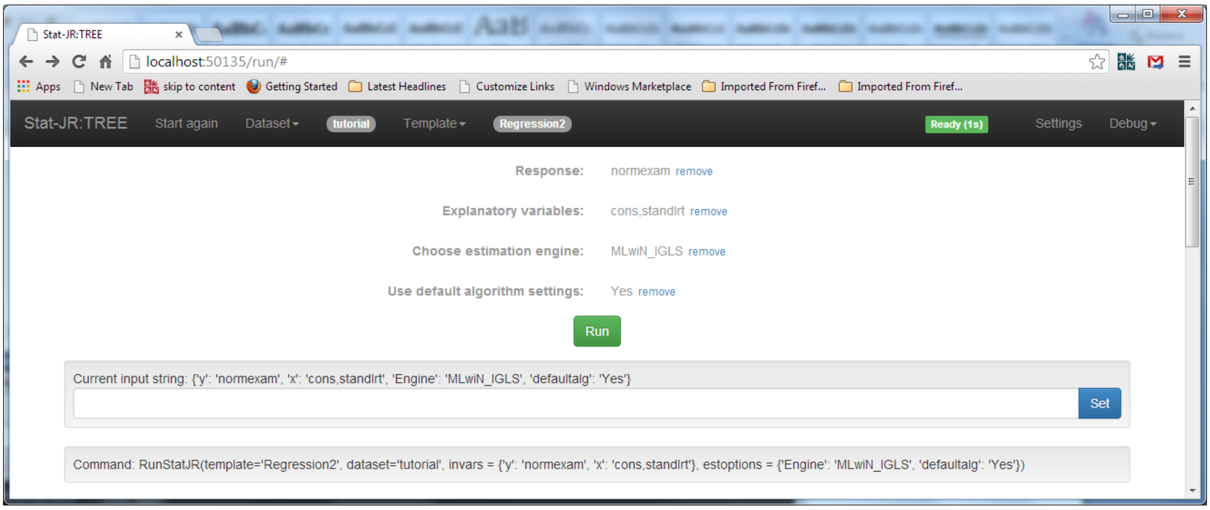
Parameters:

parameter	mean	sd	ESS	variable
deviance	9763.52135498	2.4524165969	5558	
beta2	-0.00106361221436	0.0125106907785	5798	cons
beta3	0.595001279732	0.0127556806575	5937	standlrt
sigma1_1	0.649178506315	0.0146208911389	6242	var(_levres)

Model:

Statistic	Value
Dbar	9763.52148438
D(thetabar)	9760.51302083
pD	3.00819102923
DIC	9766.5296224

Apart from the speed of estimation (which is much quicker than Stat-JR and WinBUGS) the results are very similar. Note that we have some slightly different numbering with MLwiN. MLwiN requires being told that there is a constant variance at level 1 and to do this we create a constant column (of 1s) named *_levres* which is made random at level 1 to represent this constant variance. The fixed effects are then numbered from 2 rather than 0. We could as an alternative run the model not using MCMC by clicking on **Choose Estimation Engine** and choosing the following and clicking **Next**:



Response: normexam

Explanatory variables: cons,standlrt

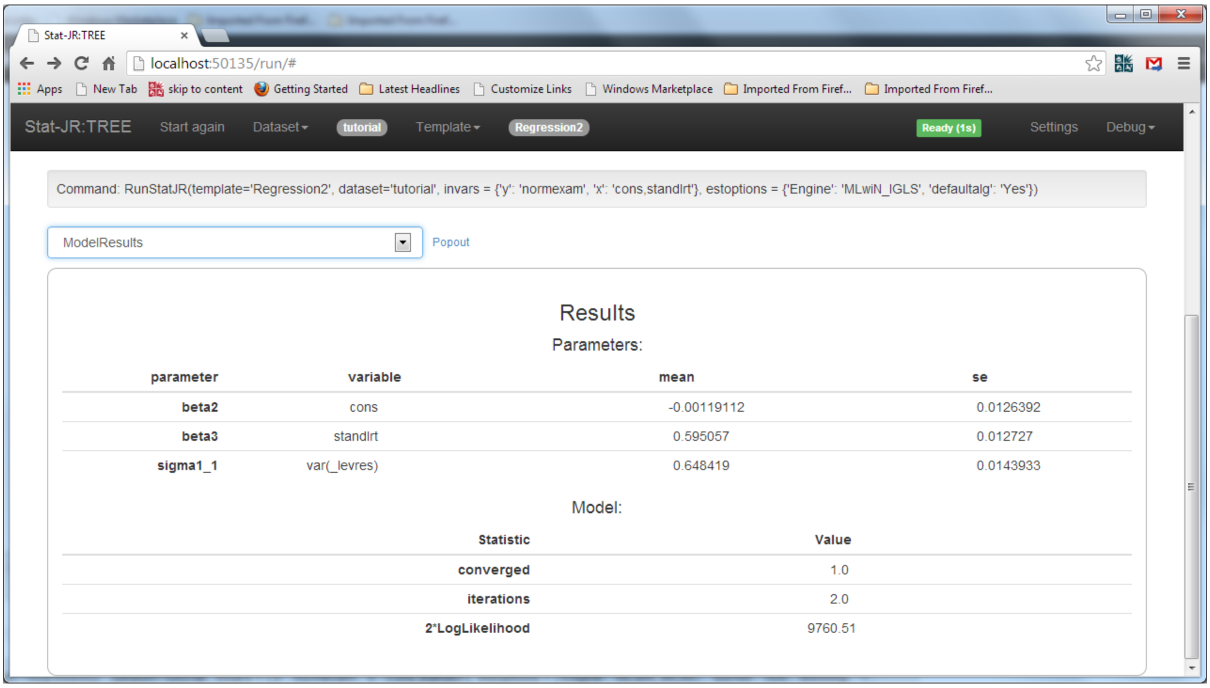
Choose estimation engine: MLwiN_IGLS

Use default algorithm settings: Yes

Current input string: {'y': 'normexam', 'x': 'cons,standlrt', 'Engine': 'MLwiN_IGLS', 'defaultalg': 'Yes'}

```
Command: RunStatJR(template='Regression2', dataset='tutorial', invars = {'y': 'normexam', 'x': 'cons,standlrt'}, estoptions = {'Engine': 'MLwiN_IGLS', 'defaultalg': 'Yes'})
```

Note that as the non-MCMC engines do not create a datafile of the output that question is not asked. Clicking on **Run** you get almost instantaneous answers if you choose **ModelResults**:



Command: RunStatJR(template='Regression2', dataset='tutorial', invars = {'y': 'normexam', 'x': 'cons,standlrt'}, estoptions = {'Engine': 'MLwiN_IGLS', 'defaultalg': 'Yes'})

Results

Parameters:

parameter	variable	mean	se
beta2	cons	-0.00119112	0.0126392
beta3	standlrt	0.595057	0.012727
sigma1_1	var(_levres)	0.648419	0.0143933

Model:

Statistic	Value
converged	1.0
iterations	2.0
2*LogLikelihood	9760.51

You will notice here that the results produced are simply point estimates and standard errors as the method doesn't construct chains. We also do not see the plots that we get with MCMC methods. As we mentioned earlier there are two engines and hence two files in the packages directory, *MLwiN_MCMC.py* and *MLwiN_IGLS.py*. We will discuss briefly *MLwiN_MCMC.py* which is currently, apart from *eStat.py* and an associated eStat engine (*CustomC*) the biggest file in the directory.

As usual the file defines a class, this time for an *MLwiNMCMC* object. The class has the usual *MethodInput*, *init* and *run* (and *runmore*) methods. The *init* method will construct the dataset and script files for running in MLwiN. In fact there are attributes within the code for each of the script files and you will see the code for the *init_script:*

```
init_script = '''
INIT 5 10000 1500 150 30
NOTE input data file
RSTA 'datafile.dta'
NOTE Set up the model

${userscript}

METH 1
BATCH 1
START
STOR "modelstate${chainnum}.wsz"
'''
```

Here the script written within the template gets inserted where we see `${userscript}`, and you will also see that chain number gets inserted in the last line of the macro. The *run* method simply runs MLwiN using the constructed datasets and macro files and then calls the *saveresults* method which creates the *ModelResults* object. Again we omit details of precisely how these code sections work as they are generic code and not template specific. The *MLwiN_IGLS.py* file has a similar form to

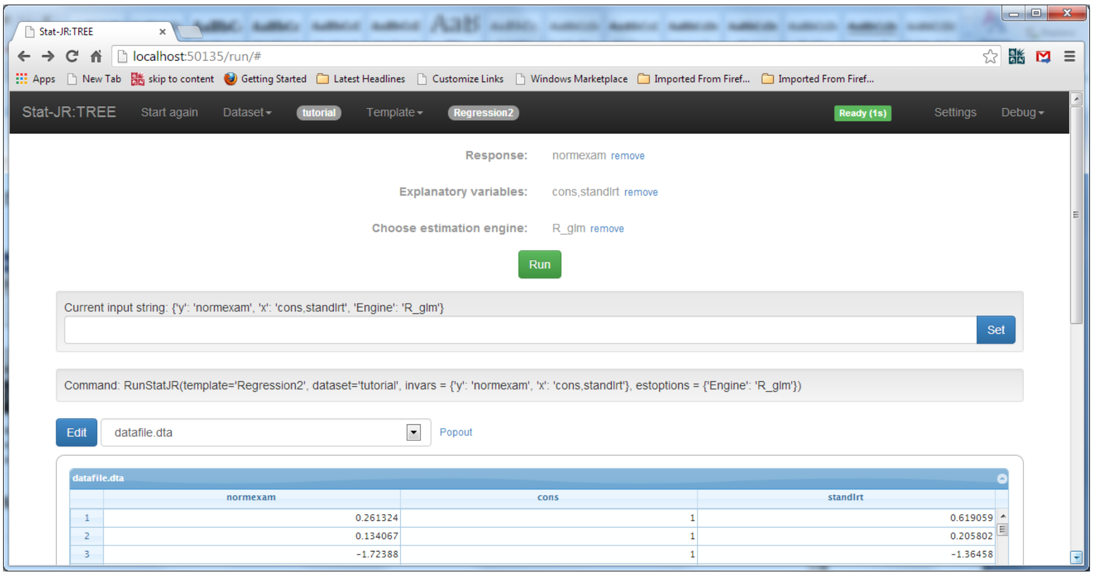
MLwiN_MCMC.py except the macros are slightly shorter and the objects produced in the *saveresults* methods are different. We will leave MLwiN here and move onto another package with some functionality for the use of both MCMC and classical estimation methods, R.

5.5 R

R (R Development Core Team, 2011) is a general purpose statistics programme that consists of a framework of interlinking statistical commands that are known as packages. The R installation consists of a base package containing many of the standard statistical operations and to this can be added user written packages. For our *Regression2* template we will utilise the *glm* function when performing classical inference. For MCMC methods we use the *MCMCglmm* package (Hadfield 2009), a user-defined function that can fit many models using MCMC. As with the earlier programs we need to include details of the location of R in *settings.cfg* on our machine prior to running *webtest*. On my machine this is as follows:

```
[R]
executable=../../../R-3.0.3/bin/x64/R.exe
```

We can again first run the *Regression2* template to see what it gives for R so choose *Regression2* as the *template* and *tutorial* as the *dataset* and then input the following:



Response: normexam remove

Explanatory variables: cons,standlrt remove

Choose estimation engine: R_glm remove

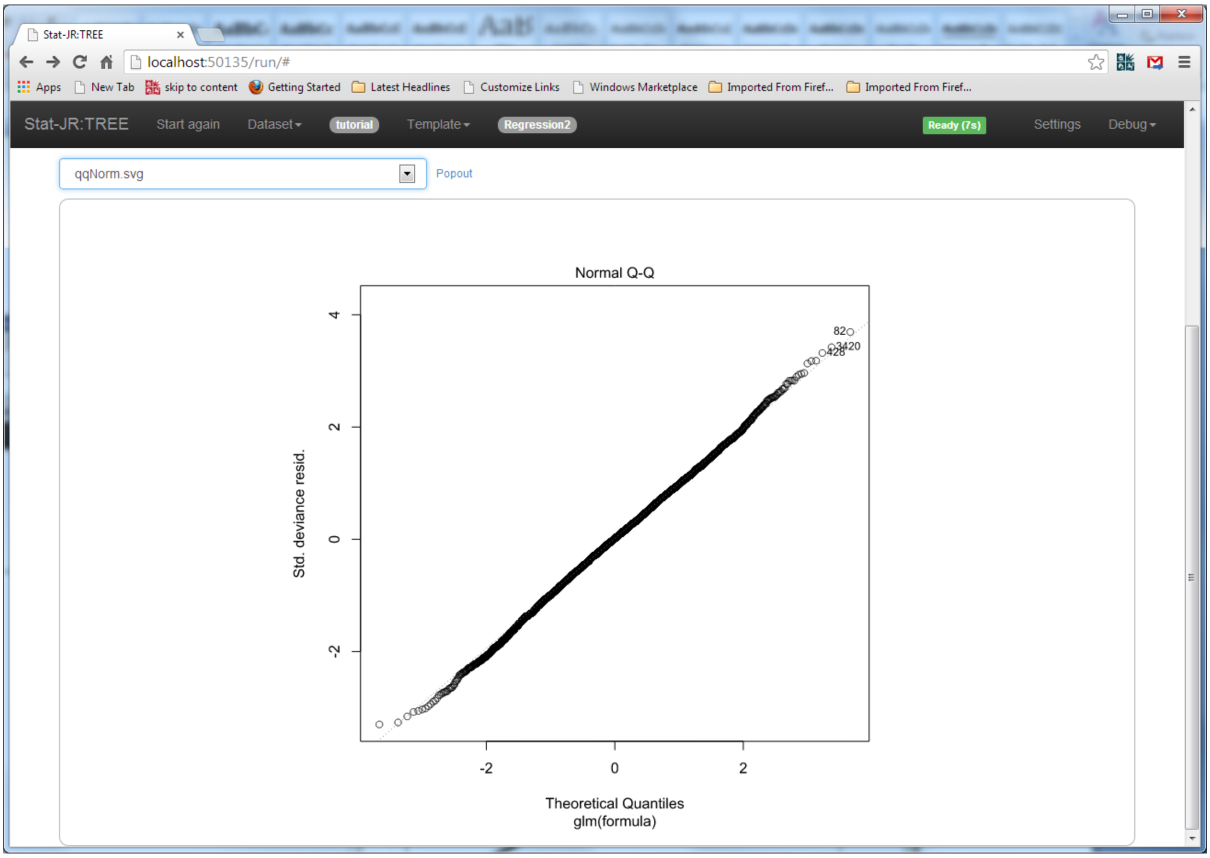
Current input string: {'y': 'normexam', 'x': 'cons,standlrt', 'Engine': 'R_glm'}

Command: RunStatJR(template='Regression2', dataset='tutorial', invars = {'y': 'normexam', 'x': 'cons,standlrt'}, estoptions = {'Engine': 'R_glm'})

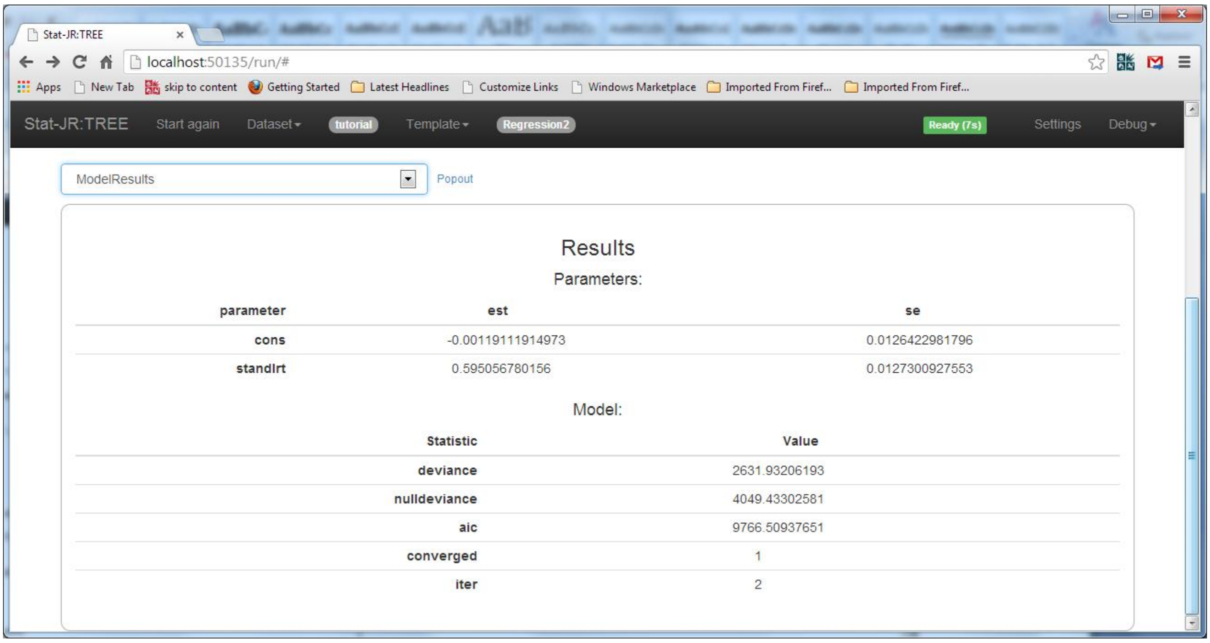
datafile.dta

	normexam	cons	standlrt
1	0.261324	1	0.619059
2	0.134067	1	0.205802
3	-1.72388	1	-1.36458

Clicking on **Run** will fire off R in the background window and then give the following output if we choose *qqNorm.svg* from the output list:



Here we see a quantile-quantile plot, which is one of the outputs produced from the script sent to R. We can select *ModelResults* in the output list to get the following:



Results

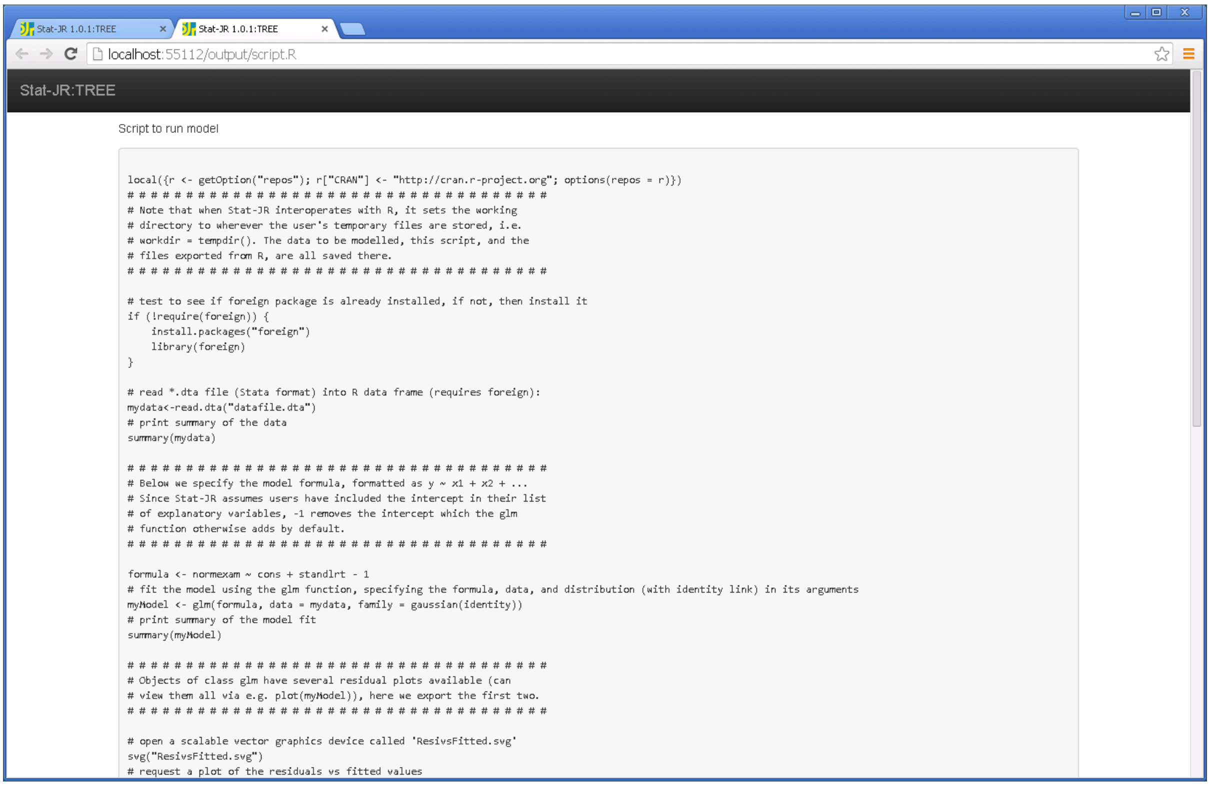
Parameters:

parameter	est	se
cons	-0.00119111914973	0.0126422981796
standlrt	0.595056780156	0.0127300927553

Model:

Statistic	Value
deviance	2631.93206193
nulldeviance	4049.43302581
aic	9766.50937651
converged	1
iter	2

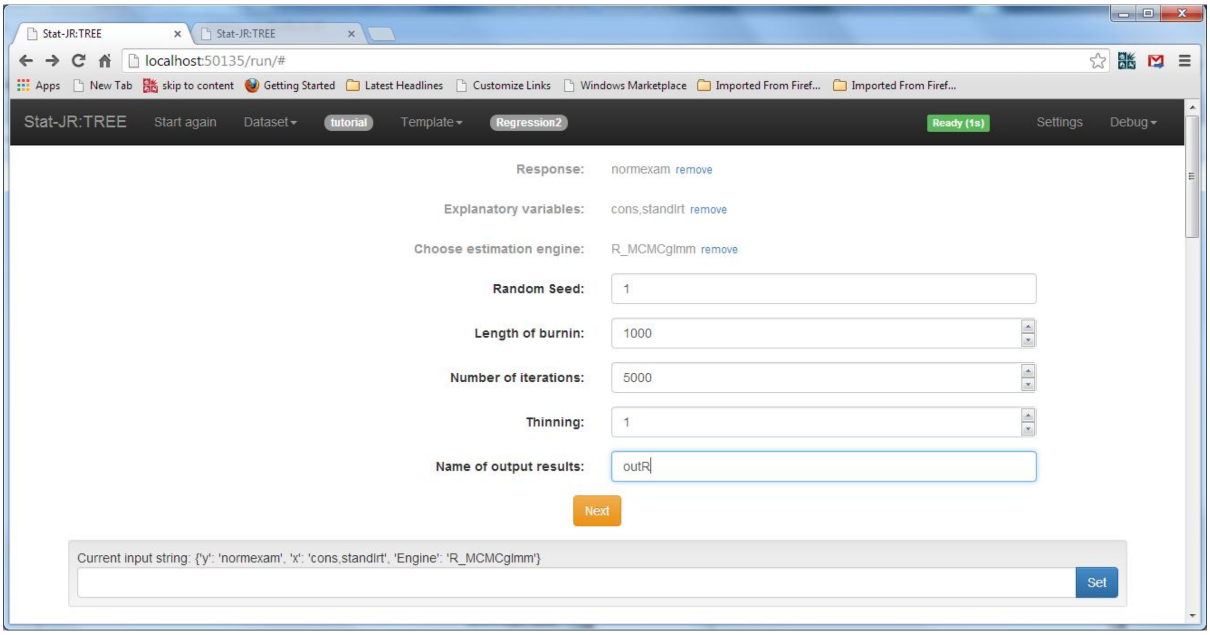
Here you will see that as the method is maximum likelihood we get only estimates and standard errors and the AIC statistic. It is also possible to view the full log file from R and a plot of residuals against fitted values in the output pane as this is also created by the R script. If we wish to look at the script itself we can select *script.R* from the output list and pop it out as follows:



Script to run model

```
local({r <- getOption("repos"); r["CRAN"] <- "http://cran.r-project.org"; options(repos = r)})
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Note that when Stat-JR interoperates with R, it sets the working
# directory to wherever the user's temporary files are stored, i.e.
# workdir = tempdir(). The data to be modelled, this script, and the
# files exported from R, are all saved there.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# test to see if foreign package is already installed, if not, then install it
if (!require(foreign)) {
    install.packages("foreign")
    library(foreign)
}

# read *.dta file (Stata format) into R data frame (requires foreign):
mydata<-read.dta("datafile.dta")
# print summary of the data
summary(mydata)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Below we specify the model formula, formatted as y ~ x1 + x2 + ...
# Since Stat-JR assumes users have included the intercept in their list
# of explanatory variables, -1 removes the intercept which the glm
# function otherwise adds by default.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

formula <- normexam ~ cons + standlrt - 1
# fit the model using the glm function, specifying the formula, data, and distribution (with identity link) in its arguments
myModel <- glm(formula, data = mydata, family = gaussian(identity))
# print summary of the model fit
summary(myModel)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Objects of class glm have several residual plots available (can
# view them all via e.g. plot(myModel)), here we export the first two.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# open a scalable vector graphics device called 'ResivsFitted.svg'
svg("ResivsFitted.svg")
# request a plot of the residuals vs fitted values
```

Here we find some heavily commented R code. We can see the call to *glm* near the bottom of the window and this is followed by some code to generate the 2 plots that appear in the output list. If we wish to instead use MCMC estimation we would give the following inputs:



Response: normexam remove

Explanatory variables: cons,standlrt remove

Choose estimation engine: R_MCMCglmm remove

Random Seed: 1

Length of burnin: 1000

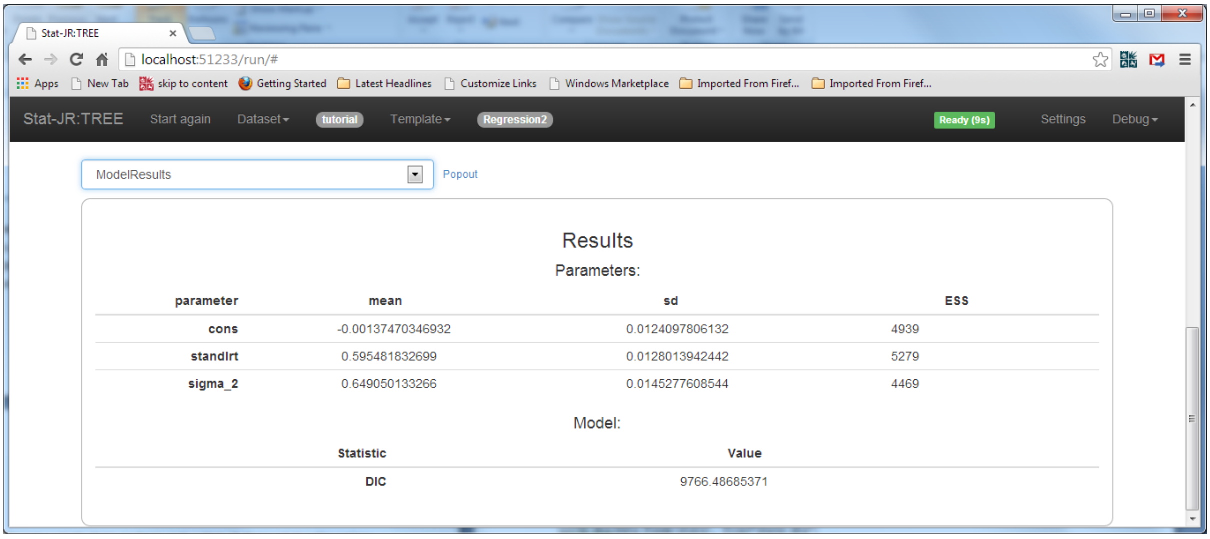
Number of iterations: 5000

Thinning: 1

Name of output results: outR

Current input string: {'y': 'normexam', 'x': 'cons,standlrt', 'Engine': 'R_MCMCglmm'}

Clicking on **Next** and then **Run** will give the following upon running and selecting *ModelResults* from the output list:



Results

Parameters:

parameter	mean	sd	ESS
cons	-0.00137470346932	0.0124097806132	4939
standlrt	0.595481832699	0.0128013942442	5279
sigma_2	0.649050133266	0.0145277608544	4469

Model:

Statistic	Value
DIC	9766.48685371

It should be noted that MCMCglmm is a single chain package and so Stat-JR does not give the option for multiple chains here although in theory R could be called several times to give parallel chains as we did for MLwiN. It also only gives the actual DIC estimate and not pD. Let us now look at how interoperability is performed in the code. At present R interoperability is possibly the opposite extreme to MLwiN interoperability. In the packages directory there are files for each R package, here we are interested in *R_glm.py* and *R_MCMCglmm.py* which are quite short files defining classes for each object and then in the *Regression2* template file there is a longer attribute *rscript* which is used to construct the R script for model fitting and in fact is one script for all methods with Python conditional statements to split up the code for glm and for MCMCglmm.

Looking firstly at *R_glm.py*, it has a *MethodInput* method but this doesn't have any additional inputs as they are not required for this package. The *inits* method code begins as follows:

```
def init(self):
        self.WriteData()
```

Here we see a call to the *WriteData* method that constructs the data file that needs to be sent to R. Then we begin to construct the script file for R:

```
script = '''
<% from EStat.Utils import Rmmult %>\\
local({r <- getOption("repos"); r["CRAN"] <- "http://cran.r-project.org"; options(repos = r)})
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Note that when Stat-JR interoperates with R, it sets the working
# directory to wherever the user's temporary files are stored, i.e.
# workdir = tempdir(). The data to be modelled, this script, and the
# files exported from R, are all saved there.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
....
```

which here initially involves loading up the data and installing the R packages required if they are not already present. Next is the call to the template specific code via the *rscript* attribute as shown below:

```
      script += self.template.rscript
```

Here the substitutions into the template specific code are made and it is added to the script file. Finally some more generic code to interrogate the output produced by R and create datafiles to be used as output objects in Stat-JR is written.

```
        script += '''
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Here an empty list called 'stats' is created, to which various
# components of the model just fitted are added. If you are fitting this
# model yourself in R, you can of course access these components directly
# yourself, rather than necessarily copying them to a list for export.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# create an empty list called 'stats'
stats <- list()
# add (residual) deviance to 'stats'
stats$deviance <- myModel$deviance
# add null deviance to 'stats'
stats$nulldeviance <- myModel$null.deviance
# add Akaike information criterion to 'stats'
stats$aic <- myModel$aic
# add convergence status (logical TRUE / FALSE) to 'stats'
stats$converged <- myModel$converged
# add number of iterations taken to fit the model to 'stats'
stats$iter <- myModel$iter

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# A variety of parameters / model fit statistics are saved to the
# working directory as *.dta files; Stat-JR imports these and
# translates them into its own format for presentation to the user.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# save 'stats' as 'stats.dta'
write.dta(data.frame(stats), file="stats.dta")
# create 'estimates.dta', consisting of the coefficient estimates on one row,
# and their standard errors on another
write.dta(data.frame(rbind(myModel$coefficients, sqrt(diag(vcov(myModel))))),
file="estimates.dta")
# save the residuals as 'residuals.dta'
write.dta(data.frame(myModel$residuals), file="residuals.dta")
'''
```

Looking at the template specific code in the template *Regression2* we see the following code at the beginning of the method:

```
rscript = '''
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Here we specify the model formula, formatted as y ~ x1 + x2 + ...
# Since Stat-JR assumes users have included the intercept in their list
# of explanatory variables, -1 removes the intercept which the glm
# function otherwise adds by default.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

formula <- ${y} ~ ${Rmmult(x)} - 1
```

This first line here simply defines the formula for the model which is common to both estimation methods and then the code goes on to specific code for the glm package.

```
% if Rpackage == 'glm':
# fit the model using the glm function, specifying the formula, data, and distribution (with
identity link) in its arguments
myModel <- glm(formula, data = mydata, family = gaussian(identity))
# print summary of the model fit
summary(myModel)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Objects of class glm have several residual plots available (can
# view them all via e.g. plot(myModel)), here we export the first two.
# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #

# open a scalable vector graphics device called 'ResivsFitted.svg'
svg("ResivsFitted.svg")
```

```
# request a plot of the residuals vs fitted values
plot(myModel, 1)
# close the device
dev.off()
# open a scalable vector graphics device called 'qqNorm.svg'
svg("qqNorm.svg")
# request a Q-Q plot
plot(myModel, 2)
# close the device
dev.off()
% endif
```

This second chunk is the code specific to the glm engine and consists of the call to that function followed by calls to a summary function and two plotting functions. There is then a large chunk of code specific for the *MCMCglmm* function which essentially runs that code and then the script is returned. So we should now see how the full script file for R is formed and what needs to be placed in the template.

The *run* method (within *R_glm.py*) that is executed when the **Run** button is pressed is in this case very short and basically consists of a command to run the script followed by a call to the *saveresults* method.

```
def run(self):
        self.eng.run('script.R')

        try:
            self.saveresults()
        except:
            logging.error('There was a problem running the model')
```

The *saveresults* method itself is also quite short and creates the *ModelResults* object. Here it involves interrogation of the two output files from R, *estimates.dta* and *stats.dta* to extract the appropriate numbers for the *ModelResults* object.

```
def saveresults(self):
        results = ModelOutput()

        dta = self.eng.outputs['estimates.dta']
        for var in dta.variables.keys():
            results.add(var, 'est', dta.variables[var]['data'][0])
            results.add(var, 'se', dta.variables[var]['data'][1])

        statsdta = self.eng.outputs['stats.dta']
        for stat in statsdta.variables.keys():
            results.add('model', stat, statsdta.variables[stat]['data'][0])

        self.eng.outputs['ModelResults'] = results
```

The *RMCMCglmm.py* file performs the equivalent operations when this engine is called from Stat-JR and the same methods are present. The code is slightly more involved and in this case the *MethodInputs* method has inputs to tell R how long to run MCMC for. There are also other differences in the functions to account for the method being MCMC and hence returning chains to be summarised. These methods are however similar to those for MLwiN and WinBUGS and we will not detail them here.

Finally here we should point out that there are R specific templates that perform other functions. For example *PlotsViaR.py* which allows the user to use the R *lattice* graphical functions from within Stat-JR. These templates generally have a very simple structure: the *engine* attribute is set to *R_script*, the *inputs* attribute gives all the inputs required by the template and the *rscript* attribute is used to construct a script file to be used in R to perform the required operations. There is therefore a file in the *packages* directory named *R_Script.py* which handles such templates. It has a very simple structure and contains the usual methods we have become familiar with when looking at the files in this directory. It basically contains code to construct the data file for R, call the template code to construct the script and then run the script and store the output objects. We will give no further details here but finish by mentioning the other packages supported by Stat-JR.

5.6 Other packages

For our *Regression2* template you will see that we also offer interoperability with other packages: Stata, SPSS, SAS, Minitab, SABRE, Matlab, GenStat, Gretl and PyMC. Each of these packages will have a python file in the packages directory which deals with getting the data in the correct format for the package, calling the template specific code for the package and interrogating the output files received back by Stat-JR from the package. Some packages will have two python files in the packages directory, for example for Stata we have files StataModel.py and StataScript.py, and here the distinction is between calls from templates that fit models and thus need to create a ModelResults object and templates that use other functionality e.g. graphs from within the package.

For our *Regression2* template you will see that for most packages the code is quite short, for example for Stata, we have:

```
statascript = '''
local family gaussian
local link identity

glm ${y} \\
% for p in x:
${p} \\
% endfor
, family(`family') link(`link') \\
##always remove the intercept
noconstant


## produce diagnostic plots
predict yhat, mu
predict ehat, response
predict rhat, response standardized
scatter ehat yhat
graph export "ResivsFitted.png", replace
egen rank=rank(rhat)
gen nscore=invnormal(rank/(_N+1))
scatter rhat nscore
graph export "QQ_Plot.png", replace
'''
```

and here the code not only fits a model but also produces two plots. This ends our whirlwind description of the interoperability features in the Stat-JR program. The interoperability features are still a work in progress and although they are present in many of the templates that we will describe in later sections we will not be going into details on this aspect of these templates. The interested reader can look at these templates and see how they perform interoperability and try writing their own interoperability code for their own templates.

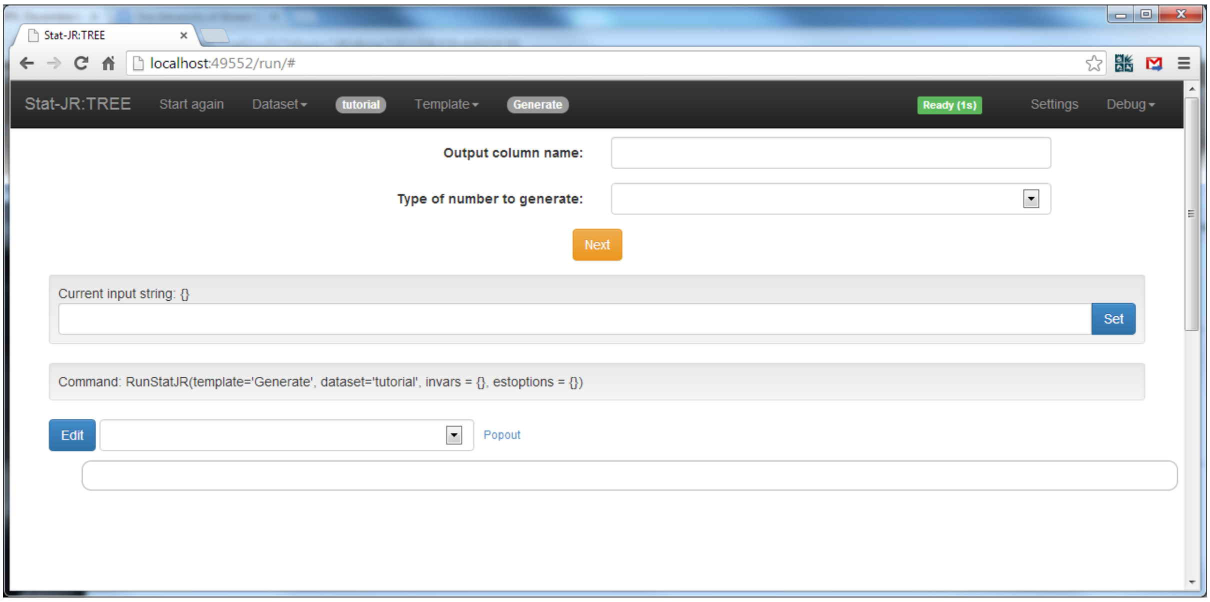
6 Input, Data manipulation and output templates

The Stat-JR system does not simply consist of templates for fitting models to datasets. There are in addition templates that allow the user to input their own datasets, manipulate datasets and plot features of datasets. In many ways these templates are much simpler to write and understand. We will here look at a few examples of the templates along with their code and explain how they fit into the *TREE* interface.

6.1 Generate template (generate.py)

Our first template to look at is used for generating columns to add to a dataset. These columns can be constants, sequences, repeated sequences or random numbers. As this template doesn't have any exciting outputs we will not see much happen after execution. Let's look at an example of adding a vector of uniform random numbers to the *tutorial* dataset.

We firstly choose the *Generate* template from the template list on the main window and press the **Use** button. The template will look as follows:



Output column name:

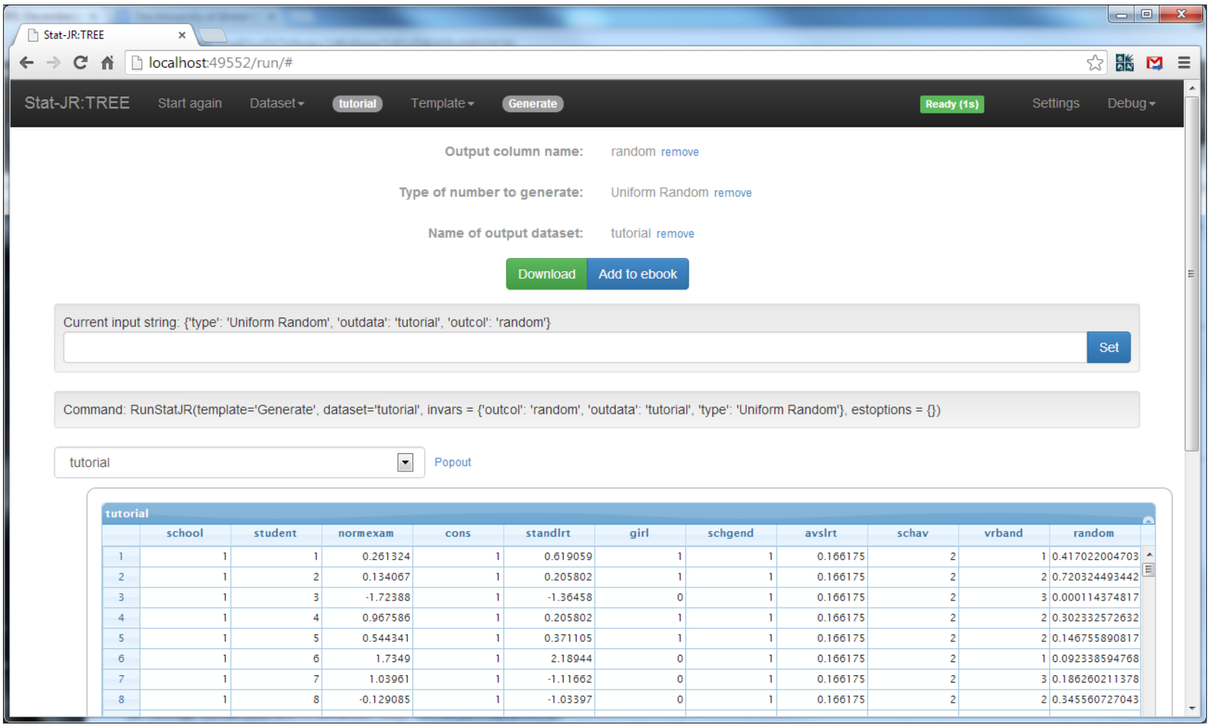
Type of number to generate:

Current input string: {}

Command: RunStatJR(template='Generate', dataset='tutorial', invars = {}, estoptions = {})

Now we select *random* for the output column name and choose *Uniform Random* for the type. After clicking **Next** we are asked for a name of output results, and here if we enter *tutorial* the new column will be appended onto the dataset and the tutorial dataset (in memory) will have an additional column. If we choose a new name then a new dataset containing all the columns from *tutorial* along with this new column will be formed (in memory) and *tutorial* will persist without the new column.

Pressing **Next** will finish the inputs and Pressing **Run** will run the template and the **Run** button will then disappear. If we then select *tutorial* from the pull down list we will see:



Output column name: random remove

Type of number to generate: Uniform Random remove

Name of output dataset: tutorial remove

Current input string: {'type': 'Uniform Random', 'outdata': 'tutorial', 'outcol': 'random'}

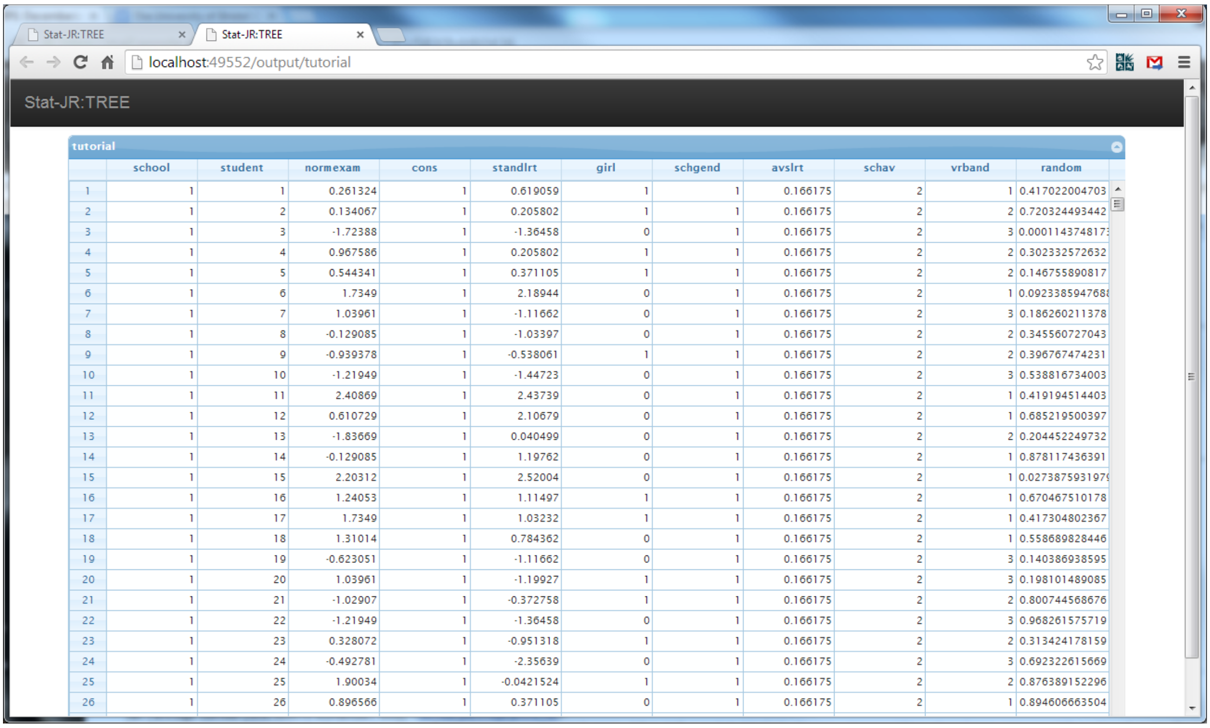
Command: RunStatJR(template='Generate', dataset='tutorial', invars = {'outcol': 'random', 'outdata': 'tutorial', 'type': 'Uniform Random'}, estoptions = {})

tutorial

tutorial

	school	student	normexam	cons	standlrt	girl	schgend	avslrt	schav	vrband	random
1	1	1	0.261324	1	0.619059	1	1	0.166175	2	1	0.417022004703
2	1	2	0.134067	1	0.205802	1	1	0.166175	2	2	0.720324493442
3	1	3	-1.72388	1	-1.36458	0	1	0.166175	2	3	0.000114374817
4	1	4	0.967586	1	0.205802	1	1	0.166175	2	2	0.302332572632
5	1	5	0.544341	1	0.371105	1	1	0.166175	2	2	0.146755890817
6	1	6	1.7349	1	2.18944	0	1	0.166175	2	1	0.092338594768
7	1	7	1.03961	1	-1.11662	0	1	0.166175	2	3	0.186260211378
8	1	8	-0.129085	1	-1.03397	0	1	0.166175	2	2	0.345560727043

To see what the template has done if you popout the tutorial object in a new tab you will see a new column labelled *random* to the right of the dataset:



tutorial

	school	student	normexam	cons	standlrt	girl	schgend	avslrt	schav	vrband	random
1	1	1	0.261324	1	0.619059	1	1	0.166175	2	1	0.417022004703
2	1	2	0.134067	1	0.205802	1	1	0.166175	2	2	0.720324493442
3	1	3	-1.72388	1	-1.36458	0	1	0.166175	2	3	0.000114374817
4	1	4	0.967586	1	0.205802	1	1	0.166175	2	2	0.302332572632
5	1	5	0.544341	1	0.371105	1	1	0.166175	2	2	0.146755890817
6	1	6	1.7349	1	2.18944	0	1	0.166175	2	1	0.092338594768
7	1	7	1.03961	1	-1.11662	0	1	0.166175	2	3	0.186260211378
8	1	8	-0.129085	1	-1.03397	0	1	0.166175	2	2	0.345560727043
9	1	9	-0.939378	1	-0.538061	1	1	0.166175	2	2	0.396767474231
10	1	10	-1.21949	1	-1.44723	0	1	0.166175	2	3	0.538816734003
11	1	11	2.40869	1	2.43739	0	1	0.166175	2	1	0.419194514403
12	1	12	0.610729	1	2.10679	0	1	0.166175	2	1	0.685219500397
13	1	13	-1.83669	1	0.040499	0	1	0.166175	2	2	0.204452249732
14	1	14	-0.129085	1	1.19762	0	1	0.166175	2	1	0.878117436391
15	1	15	2.20312	1	2.52004	0	1	0.166175	2	1	0.027387593197
16	1	16	1.24053	1	1.11497	1	1	0.166175	2	1	0.670467510178
17	1	17	1.7349	1	1.03232	1	1	0.166175	2	1	0.417304802367
18	1	18	1.31014	1	0.784362	0	1	0.166175	2	1	0.558689828446
19	1	19	-0.623051	1	-1.11662	0	1	0.166175	2	3	0.140386938595
20	1	20	1.03961	1	-1.19927	1	1	0.166175	2	3	0.198101489085
21	1	21	-1.02907	1	-0.372758	1	1	0.166175	2	2	0.800744568676
22	1	22	-1.21949	1	-1.36458	0	1	0.166175	2	3	0.968261575719
23	1	23	0.328072	1	-0.951318	1	1	0.166175	2	2	0.313424178159
24	1	24	-0.492781	1	-2.35639	0	1	0.166175	2	3	0.692322615669
25	1	25	1.90034	1	-0.0421524	1	1	0.166175	2	2	0.876389152296
26	1	26	0.896566	1	0.371105	0	1	0.166175	2	1	0.894606663504

Examining the code it is first worth noting that the template has

```
engines = ['Python_script']
```

This tells Stat-JR that this template is not a model template and therefore needs to be treated differently. The template has an *inputs* attribute as shown below:

```
inputs = '''
outcol = Text('Output column name: ')
type = Text('Type of number to generate: ', ['Uniform Random', 'Binomial
Random', 'Chi Squared Random', 'Exponential Random', 'Gamma Random',
'Normal Random', 'Poisson Random', 'Constant', 'Sequence', 'Repeated
sequence'])
#carry = DataMatrix('Carried Data: ')

if type == 'Binomial Random':
    prob = Text('Probability')
    numtrials = Integer('Number of Trials')

if type == 'Chi Squared Random':
    degreefree = Integer('Degrees of Freedom')

if type == 'Gamma Random':
    shape = Text('Shape')

if type == 'Poisson Random':
    exp = Text('Expectation')

if type == 'Constant':
    value = Text('Value')

if type == 'Sequence':
    start = Integer('Starting Value')
    step = Integer('Step')

if type == 'Repeated sequence':
    max = Integer('Maximum number')
    repeats = Integer('Repeats per block')

outdata=Text('Name of output dataset: ')
'''
```

Here we see that there are two main inputs, a name for the column to add (*outcol*) and a *type* of column to generate. Depending on the type there may be additional inputs and these are catered for through a set of if statements in Python. So for example if we want a constant column we will have an additional attribute, *value* which gives the value of the constant. Note that the length of the vector is controlled by the lengths of the columns already in the dataset, as a dataset is currently restricted to be a set of columns of equal length.

As this template is not a model template there is no *model* or *latex* attributes instead the computations are performed within a method called *pythonscript* which basically performs the required calculation in Python and adds the column to the output. The method code is as follows:

```
pythonscript = '''
import numpy

import EStat
from EStat.Templating import *
from EStat.DTAFile import DTAFile

retval = DTAFile()
```

```
retval.nobs = datafile.nobs
for k in datafile.variables.keys():
    retval.addvariable(k, data = datafile.variables[k]['data'])

datalen = datafile.nobs

if type == 'Uniform Random':
    outvar = numpy.random.uniform(size = datalen)
if type == 'Binomial Random':
    outvar = numpy.random.binomial(float(numtrials), float(prob), size =
datalen)
if type == 'Chi Squared Random':
    outvar = numpy.random.chisquare(float(degreefree), size = datalen)
if type == 'Exponential Random':
    outvar = numpy.random.exponential(size = datalen)
if type == 'Gamma Random':
    outvar = numpy.random.gamma(float(shape), size = datalen)
if type == 'Normal Random':
    outvar = numpy.random.normal(size = datalen)
if type == 'Poisson Random':
    outvar = numpy.random.poisson(float(exp), size = datalen)
if type == 'Constant':
    outvar = numpy.ones(datalen) * float(value)
if type == 'Sequence':
    outvar = numpy.arange(int(start), int(start) + (datalen * int(step)) -
1, int(step))
if type == 'Repeated sequence':
    outvar = numpy.array(list(numpy.repeat(numpy.arange(1, int(max) + 1),
int(repeats))) * (datalen / int(max) * int(repeats)))

retval.addvariable(str(outcol), data = outvar)
outputs[str(outdata)] = retval
'''
```

Here although the function is long this is in the main due to the many if statements to cope with each type of vector to be generated. So for example if we wanted a vector of Uniform random numbers to be stored in random then the only lines to be executed are:

```
retval = DTAFile()
retval.nobs = datafile.nobs
for k in datafile.variables.keys():
    retval.addvariable(k, data = datafile.variables[k]['data'])
datalen = datafile.nobs
outvar = numpy.random.uniform(size = datalen)
retval.addvariable(str(outcol), data = outvar)
outputs[str(outdata)] = retval
```

Here the code calculates the length of vector in the fifth line, uses the numpy random generator in the next line to create the column of numbers in outvar. In the remaining lines we link the column into the dataset and finally return the dataset to the output name we gave as an input.

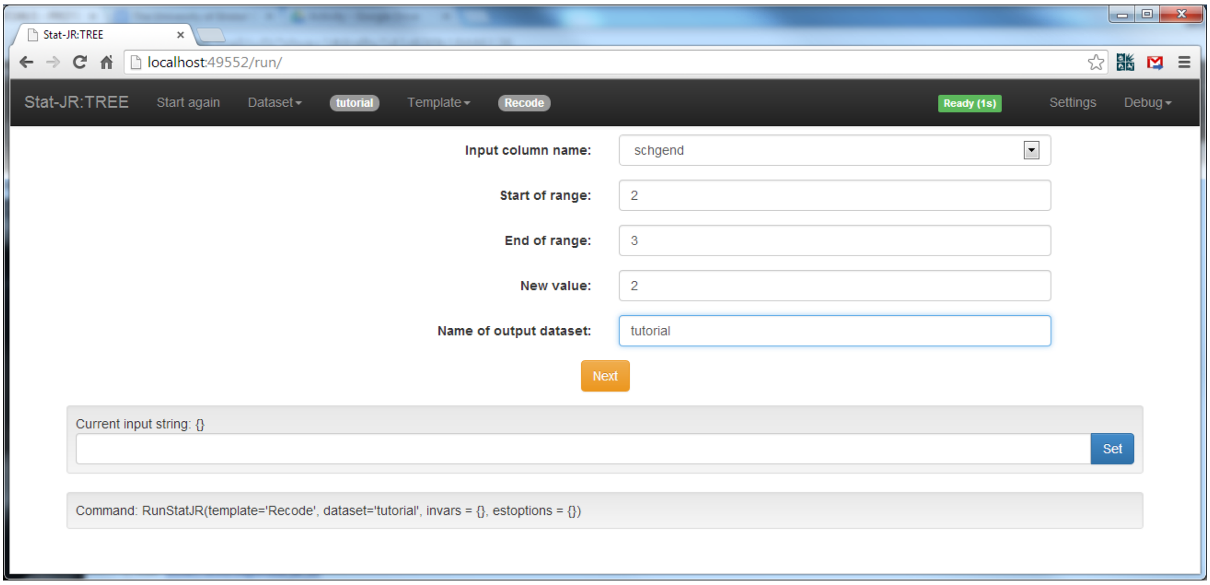
Exercise 2

Try modifying this template so that it only offers the random number generators. Try expanding the inputs so for example the Normal random generator will allow a mean and a variance, the Gamma has a scale parameter and the exponential has a rate parameter.

6.2 Recode template (recode.py)

The *Generate* template allows the user to add new columns to their existing dataset. There are many templates that expand or manipulate a dataset and we will here look at a second template, the *Recode* template. The *Recode* template as the name suggests recodes values – in this case recoding values within a contiguous range to a specific new value. This can be useful for creating categorical values – although this might involve several repeated uses of the *Recode* template!

We will demonstrate this with the **tutorial** dataset and look at recoding the school gender (*schgend*) column. In the original dataset *schgend* takes values 1 for a mixed school, 2 for a boys school and 3 for a girls school. We might want to recode this to take values 1 for mixed and 2 for single sex i.e. convert the 3s for girls schools to 2s. To do this first we select *Recode* from the template list and hit the **Use** button. Next we select *schgend* from the list of columns and select the other inputs as below:



Input column name: schgend

Start of range: 2

End of range: 3

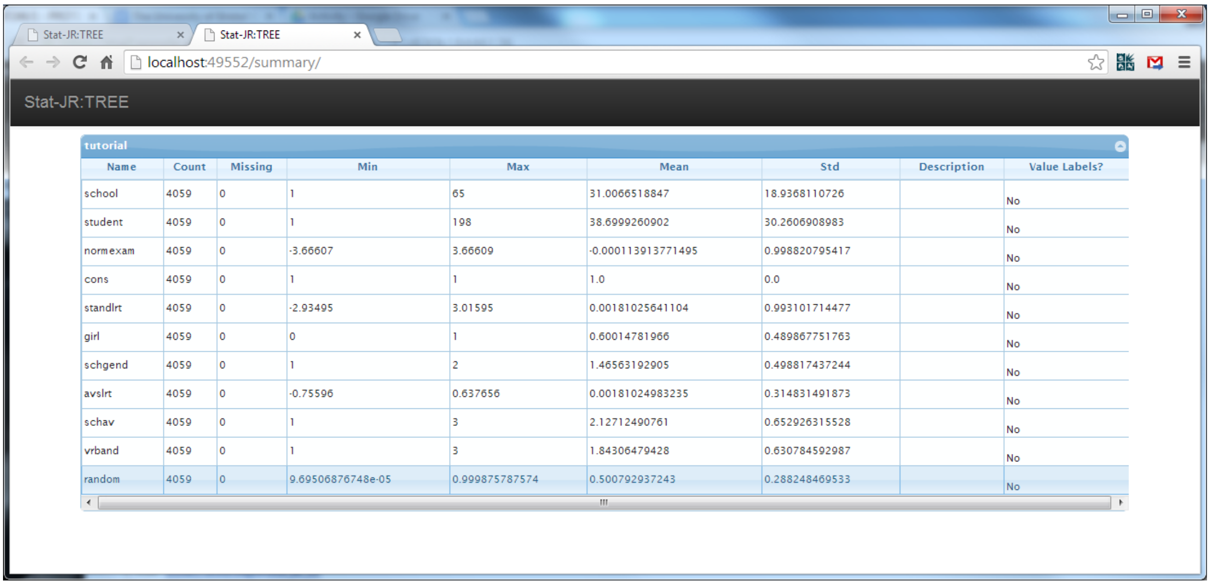
New value: 2

Name of output dataset: tutorial

Current input string: {}

Command: RunStatJR(template='Recode', dataset='tutorial', invars = {}, estoptions = {})

Clicking on **Next** and **Run** will run the template. Selecting **Summary** from the *Dataset* pull down list at the top of the screen shows a dataset summary:



Name	Count	Missing	Min	Max	Mean	Std	Description	Value Labels?
school	4059	0	1	65	31.0066518847	18.9368110726		No
student	4059	0	1	198	38.6999260902	30.2606908983		No
normexam	4059	0	-3.66607	3.66609	-0.000113913771495	0.998820795417		No
cons	4059	0	1	1	1.0	0.0		No
standlrt	4059	0	-2.93495	3.01595	0.00181025641104	0.993101714477		No
girl	4059	0	0	1	0.60014781966	0.489867751763		No
schgend	4059	0	1	2	1.46563192905	0.498817437244		No
avslrt	4059	0	-0.75596	0.637656	0.00181024983235	0.314831491873		No
schav	4059	0	1	3	2.12712490761	0.652926315528		No
vrband	4059	0	1	3	1.84306479428	0.630784592987		No
random	4059	0	9.69506876748e-05	0.999875787574	0.500792937243	0.288248469533		No

Here we see that schgend now goes from 1 to 2 as expected. Let us now look at the code for this template. As with *generate* this template has an *inputs* and a *pythonscript* attribute. These are both quite short:

```
inputs = '''
incol = DataVector('Input column name: ')
rangestart = Text('Start of range: ')
rangeend = Text('End of range: ')
newval = Text('New value: ')
#carry = DataMatrix('Carried Data: ')
outdata=Text('Name of output dataset: ')
'''
```

Here the *inputs* attribute contains the five inputs that we saw when running the template. Next the *pythonscript* attribute:

```
    pythonscript = '''
import numpy
import numexpr

import EStat
from EStat.Templating import *
from EStat.DTAFile import DTAFile
retval = DTAFile()
retval.nobs = datafile.nobs
for k in datafile.variables.keys():
    retval.addvariable(k, data = datafile.variables[k]['data'])

# Copy data into numpy array for processing
var = numpy.array(datafile.variables[incol]['data'])

var[(var >= float(rangestart)) & (var <= float(rangeend))] = float(newval)

retval.addvariable(incol, data = var)

outputs[str(outdata)] = retval
'''
```

After some importing lines, the *pythonscript* attribute firstly copies the original column to the object *var* and then performs the recoding by finding the values in the original column within the correct range and then replacing them with the *newval*. Note the >= and <= operators mean that the range is inclusive of its end points. Finally when *var* is modified it is then linked back to the input column and the dataset is returned.

Exercise 3

This template applies the recoding by copying the recoded column over itself. As an exercise, try modifying the template so that it will place the recoded column into a new location i.e. have another name that is where to output the column to. Note the code for **Generate** should help here.

6.3 AverageAndCorrelation template

Another template that one might consider using prior to fitting a model is the *AverageAndCorrelation* template. This template will give either some summary statistics (including the averages) for a series of columns or the correlation matrix for a set of columns.
The template has a very short *inputs* attribute:

```
inputs = '''
op = Text('Operation: ', ['averages', 'correlation'])
vars = DataMatrix('Variables: ')
'''
```

Here *op* allows the user to choose between averages and correlations whilst *vars* stores which columns to perform the operation on. This template again uses the *pythonscript* attribute but this time creates an output called *table* which will give the averages or correlations in tabular form.
The code for *pythonscript* is as follows:

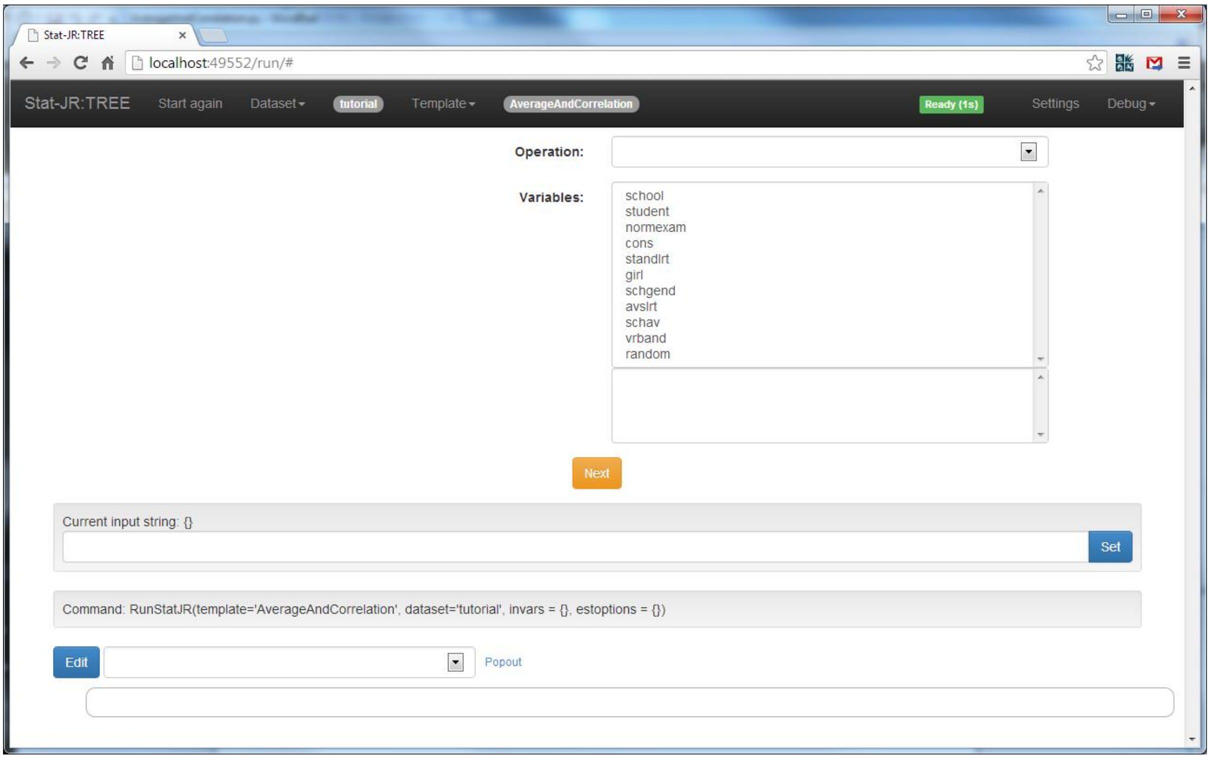
```
    pythonscript = '''
import numpy
import EStat
from EStat.Templating import *

tabout = TabularOutput()
if op == 'averages':
    tabout.column_headings = ['name', 'count', 'mean', 'sd']
    for i in range(0, len(vars)):
        var = numpy.array(datafile.variables[vars[i]]['data'])
        tabout.add_row(vars[i], [len(var), var.mean(), var.std()])

if op == 'correlation':
    invars = numpy.empty((len(vars), datafile.nobs))
    for i in range(0, len(vars)):
        invars[i, :] = datafile.variables[vars[i]]['data'][:]

    corrs = numpy.corrcoef(invars)
    tabout.column_headings = ['name']
    for j in range(0, len(vars)):
        tabout.column_headings.append(vars[j])

    for i in range(0, len(vars)):
        row = []
        for j in range(0, len(vars)):
            row.append(corrs[i, j])
        tabout.add_row(vars[i], row)
```

```
outputs['table'] = tabout
'''
```

You will see here separate chunks of code for averages and correlations. The average code basically initializes a table output with column heading and then loops through the columns in *vars* setting each in turn as a numpy array stored in *var*. An array of text strings are then constructed and added to the tabular output, *tabout*, and here we are utilising the len function to get the number of data items and the built in numpy functions *mean* and *std* to get the mean and standard deviation respectively.

The correlation code is slightly longer, we here firstly need to construct the data as a matrix *invars* from which we can construct the correlations (*corrs*) by a call to the numpy.corrcoef function. Then we again format the output nicely into *tabout*.

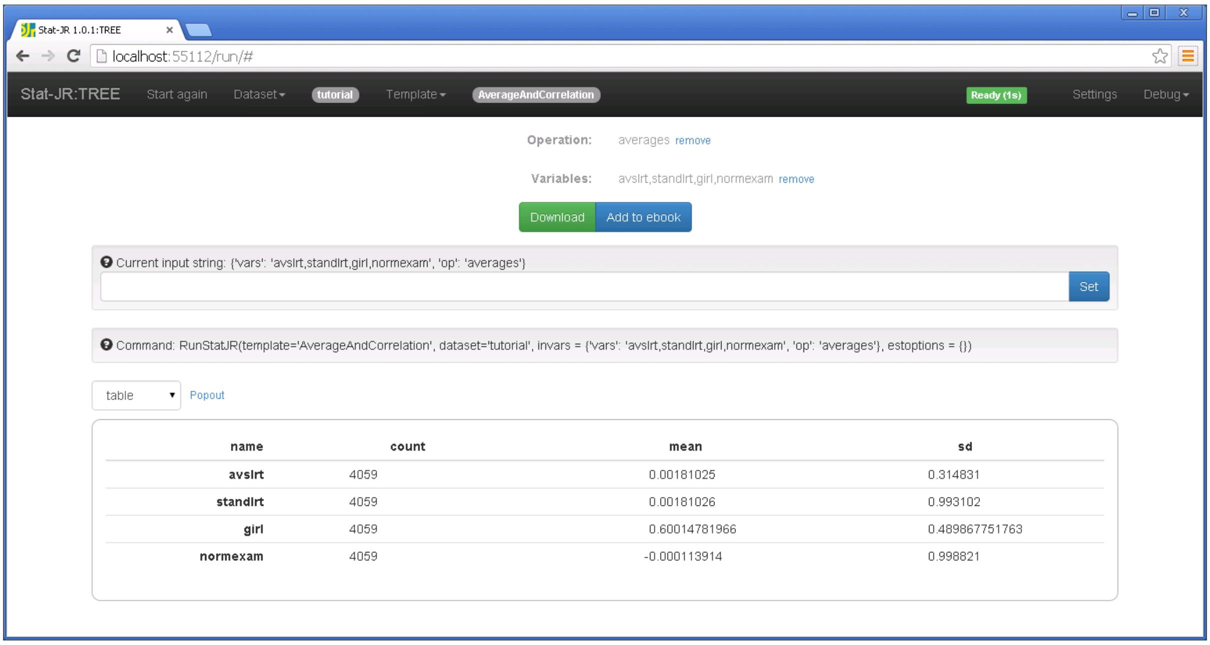
The line `outputs['table'] = tabout` creates the table object which is then included in the output object list. If we consider using this template with the *tutorial* dataset we first need to select it from the template list and select **Use** to get the default screen:



Current input string: {}

Command: RunStatJR(template='AverageAndCorrelation', dataset='tutorial', invars = {}, estoptions = {})

We can now try this with some of the variables and in turn averages and correlation. Here is an example of averages – note we select *table* from the objects list:



Operation: averages remove

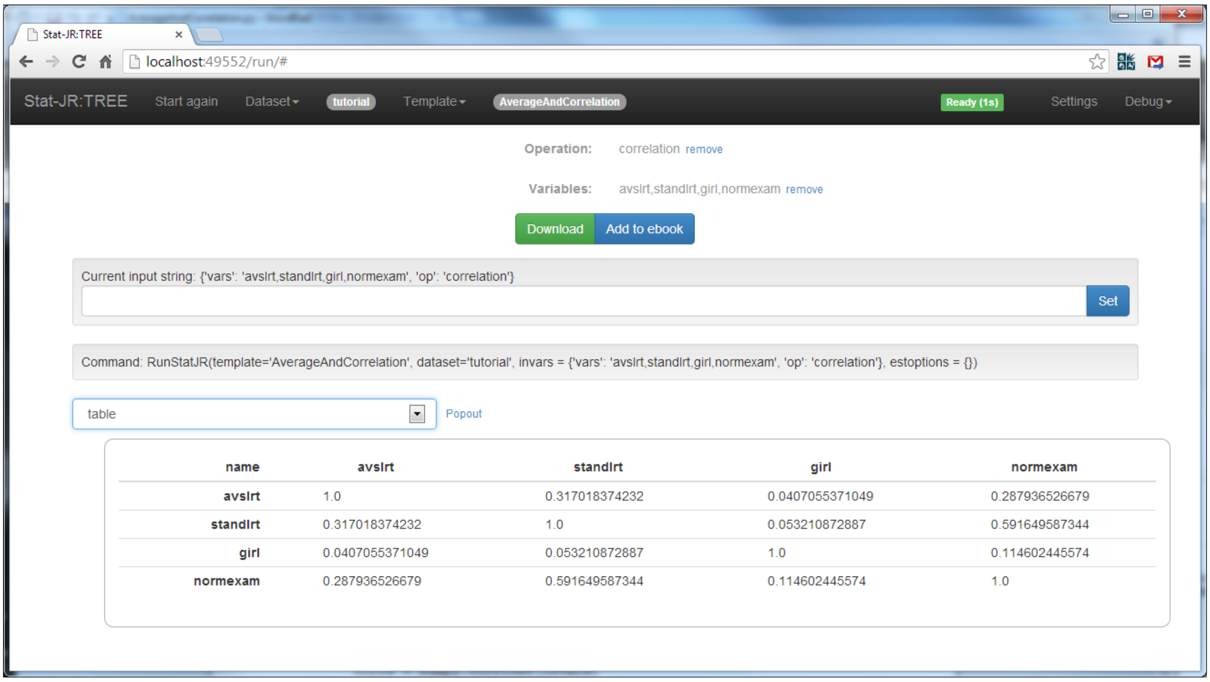
Variables: avslrt,standlrt,girl,normexam remove

Current input string: {'vars': 'avslrt,standlrt,girl,normexam', 'op': 'averages'}

Command: RunStatJR(template='AverageAndCorrelation', dataset='tutorial', invars = {'vars': 'avslrt,standlrt,girl,normexam', 'op': 'averages'}, estoptions = {})

name	count	mean	sd
avslrt	4059	0.00181025	0.314831
standlrt	4059	0.00181026	0.993102
girl	4059	0.60014781966	0.489867751763
normexam	4059	-0.000113914	0.998821

and the correlations for the same four variables:



Operation: correlation remove

Variables: avslrt,standlrt,girl,normexam remove

Current input string: {'vars': 'avslrt,standlrt,girl,normexam', 'op': 'correlation'}

Command: RunStatJR(template='AverageAndCorrelation', dataset='tutorial', invars = {'vars': 'avslrt,standlrt,girl,normexam', 'op': 'correlation'}, estoptions = {})

name	avslrt	standlrt	girl	normexam
avslrt	1.0	0.317018374232	0.0407055371049	0.287936526679
standlrt	0.317018374232	1.0	0.053210872887	0.591649587344
girl	0.0407055371049	0.053210872887	1.0	0.114602445574
normexam	0.287936526679	0.591649587344	0.114602445574	1.0

Exercise 4

Why not try and add the option to this template to give the standard error of the mean and also to allow the template to output both averages and correlations together for the same variables. Remember to rename the template first!

6.4 XYPlot template

Our final template in our whistle-stop tour of non-model templates is a graphing template. Python has excellent graphing facilities and so we have created a few very basic graphing templates that demonstrate some of these facilities. The *xyplot* template basically allows the user to plot one or more Y variables against an X variable on the same plot.

The template has an *inputs* attribute as shown below:

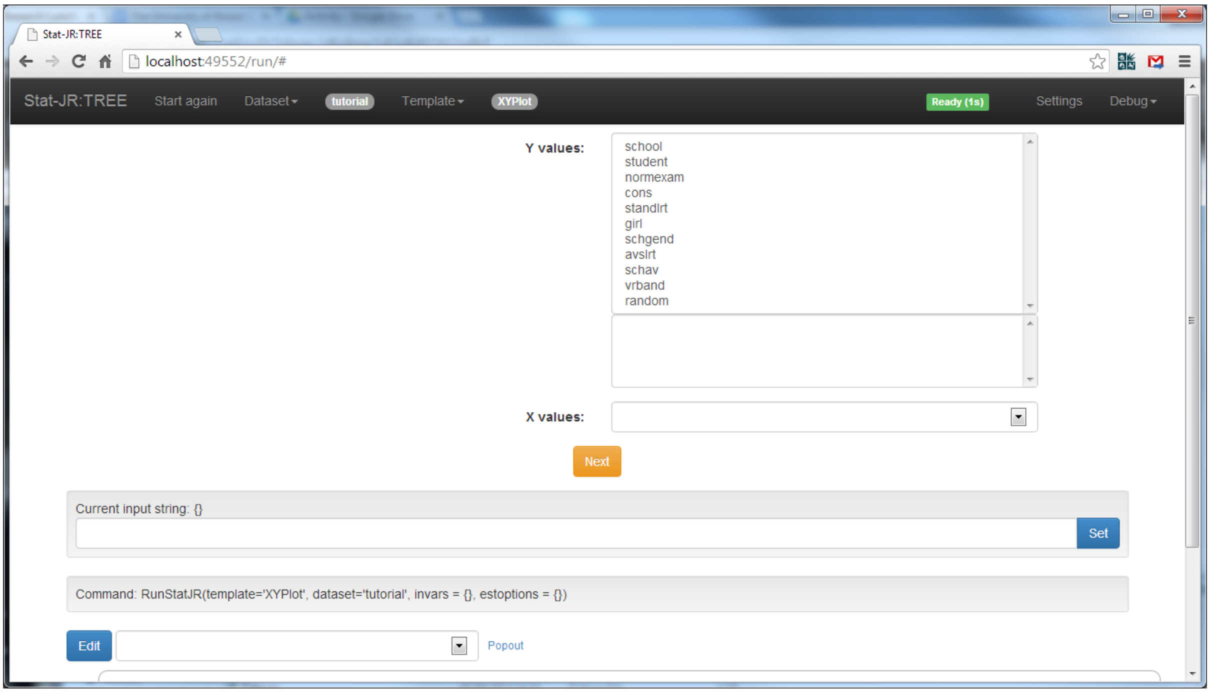
```
inputs = '''
yaxis = DataMatrix('Y values: ')
xaxis = DataVector('X values: ')
'''
```

Here we have two inputs, the various Y variables and the corresponding X variable to plot against. For a graph template we once again use the *pythonscript* attribute but this time the method constructs an object called *graphxy* which is in fact a .svg image file and is constructed by a function called ImageOutput.

The *pythonscript* code is as follows:

```
pythonscript = '''
from io import BytesIO

from matplotlib.figure import Figure
import matplotlib.lines as lines
from matplotlib.backends.backend_agg import FigureCanvasAgg

import EStat
from EStat.Templating import *

fig = Figure(figsize=(8,8))
ax = fig.add_subplot(100 + 10 + 1, xlabel = str(xaxis))
for n in yaxis:
    ax.plot(datafile.variables[xaxis]['data'],
datafile.variables[n]['data'], 'x', label = n)
    ax.legend()

canvas = FigureCanvasAgg(fig)
buf = BytesIO()
canvas.print_figure(buf, dpi=80, format='svg')

buf.seek(0)
outputs['graphxy.svg'] = ImageOutput(buf.getvalue())
buf.close()
'''
```

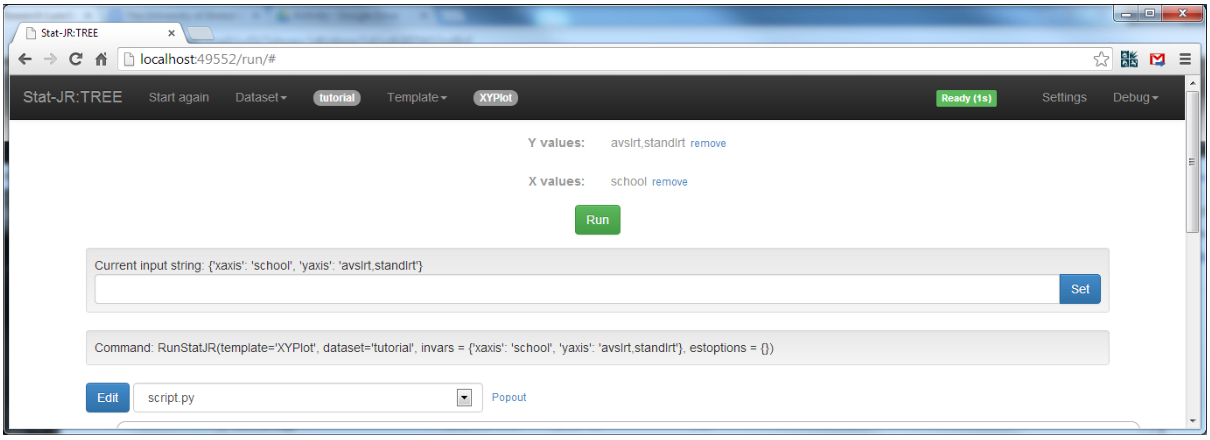
Here we have to firstly import lots of Python libraries in order to call the graphics functions. The function we are using is the *Figure* function from the *matplotlib* package. We then make a blank plot sticking on the axes labels before looping over the y variables and plotting their points. The ‘x’ is the symbol to be plotted for each plot. The last six lines are used to store the plotted figure as a .svg file. To see this template in action we will pick it from the *TREE* main screen (along with the *tutorial* dataset) and we will be greeted by the following in the browser:



Current input string: {}

Command: RunStatJR(template='XYPlot', dataset='tutorial', invars = {}, estoptions = {})

Perhaps the simplest plot here would be to plot *normexam* against *standlrt* which you can try yourself. Here we illustrate instead the use of more than one *y* variable by making the following selections:



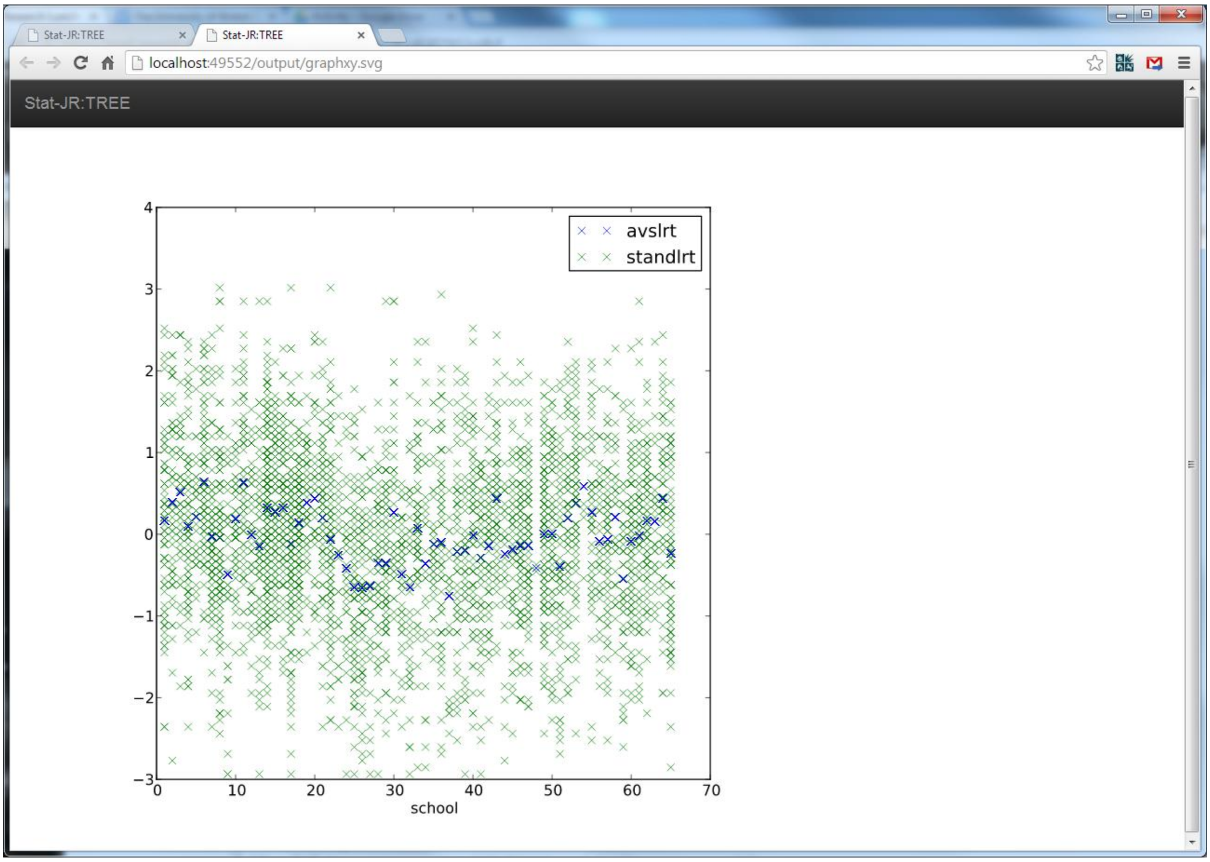
Y values: avslrt,standlrt remove

X values: school remove

Current input string: {'xaxis': 'school', 'yaxis': 'avslrt,standlrt'}

Command: RunStatJR(template='XYPlot', dataset='tutorial', invars = {'xaxis': 'school', 'yaxis': 'avslrt,standlrt'}, estoptions = {})

Clicking on the **Run** button and choosing to popout *graphxy.svg* gives the following graph:



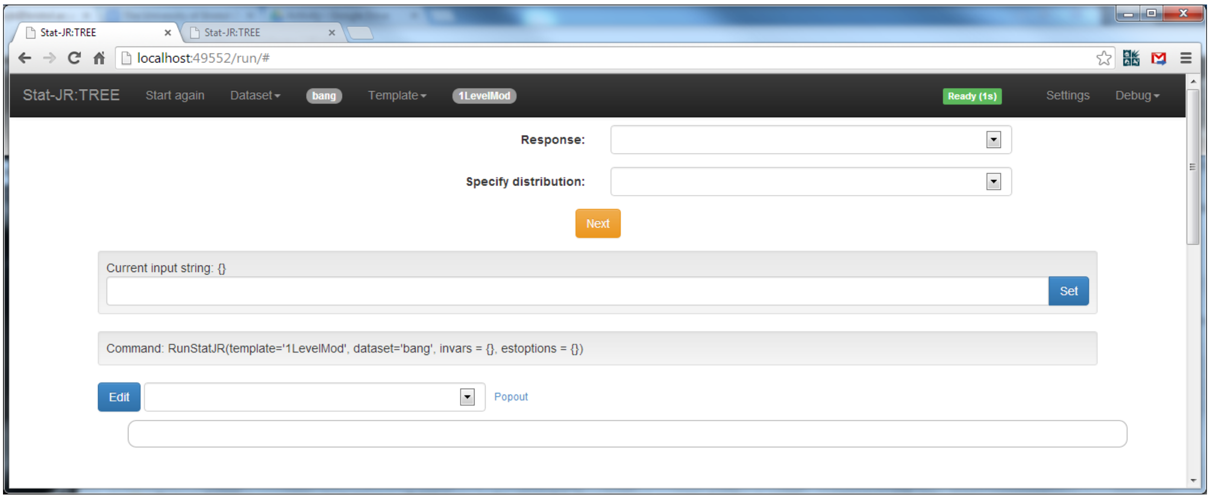
Here we see plotted the actual intake scores for each pupil against school number in green and the school average in blue.

<u>Exercise 5</u>

Simplify this template to only allow a single y variable. Try adding a main title to the graph and varying the symbol and colours – maybe make this an option for the user to choose. Remember to rename the template before you start!

7 Single level models of all flavours – A logistic regression example

We have so far met two model templates, *Regression1* which could be used to fit normal response multiple regression models in the Stat-JR built in MCMC engine and *Regression2* which allowed the same models to be fitted in other statistics packages. We will now look at a generalisation of these templates, *1LevelMod* that allows other response types including Binomial and Poisson responses. This template will illustrate the use of conditional statements within the *inputs* and *model* functions. We will begin by looking at the template in action in *Stat-JR.* The template should be able to fit all the models that *Regression1* fits and so you could test the earlier regressions but here we will look at a logistic regression. So from the main menu headings we need to set the *template* to be *1LevelMod* and the *dataset* to be *bang*, our example binary response dataset taken from the 1988 Bangladeshi Fertility Survey. Clicking on **Use** gives the following output in the browser:



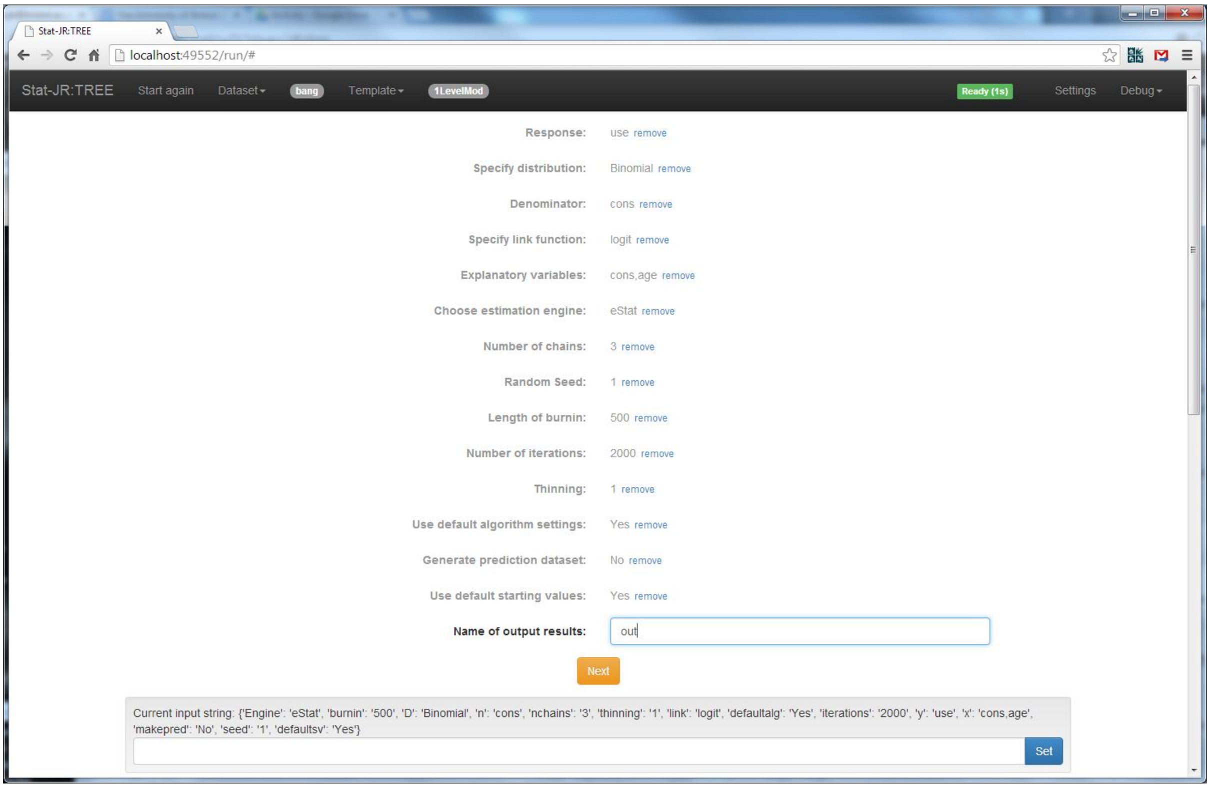
Response:

Specify distribution:

Current input string: {}

Command: RunStatJR(template='1LevelMod', dataset='bang', invars = {}, estoptions = {})

We will now set up the various inputs and the screen will look as follows:



Response: use remove

Specify distribution: Binomial remove

Denominator: cons remove

Specify link function: logit remove

Explanatory variables: cons,age remove

Choose estimation engine: eStat remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

Use default algorithm settings: Yes remove

Generate prediction dataset: No remove

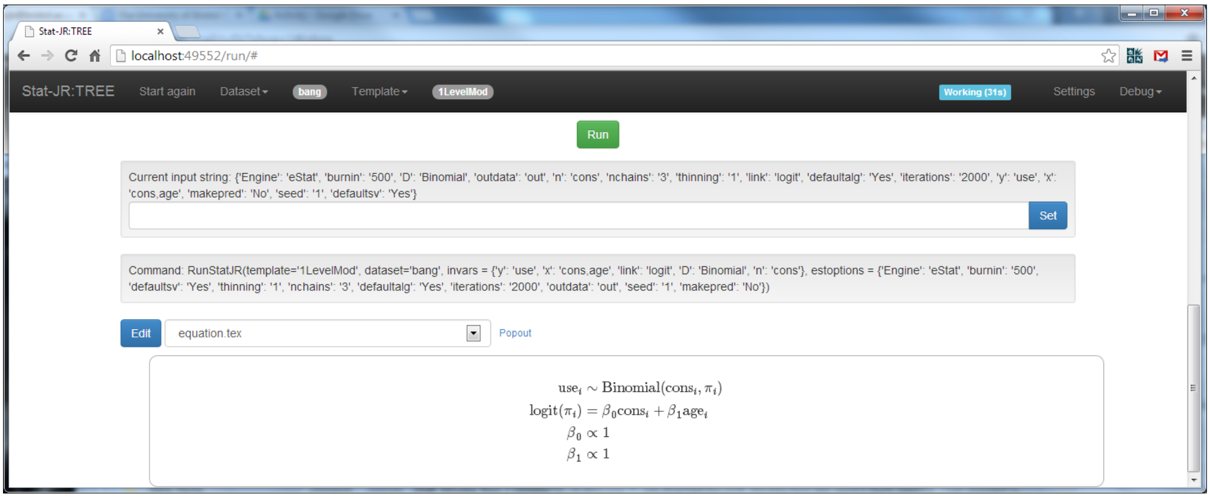
Use default starting values: Yes remove

Name of output results: out

Current input string: {'Engine': 'eStat', 'burnin': '500', 'D': 'Binomial', 'n': 'cons', 'nchains': '3', 'thinning': '1', 'link': 'logit', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'use', 'x': 'cons,age', 'makepred': 'No', 'seed': '1', 'defaultsv': 'Yes'}

We are here fitting a logistic regression to the response variable which is whether the women in Bangladesh in the dataset use contraception or not. We are regressing this against age and using the Stat-JR built-in MCMC engine with some default settings for estimation.

If we click on **Next, equation.tex** will display the model:

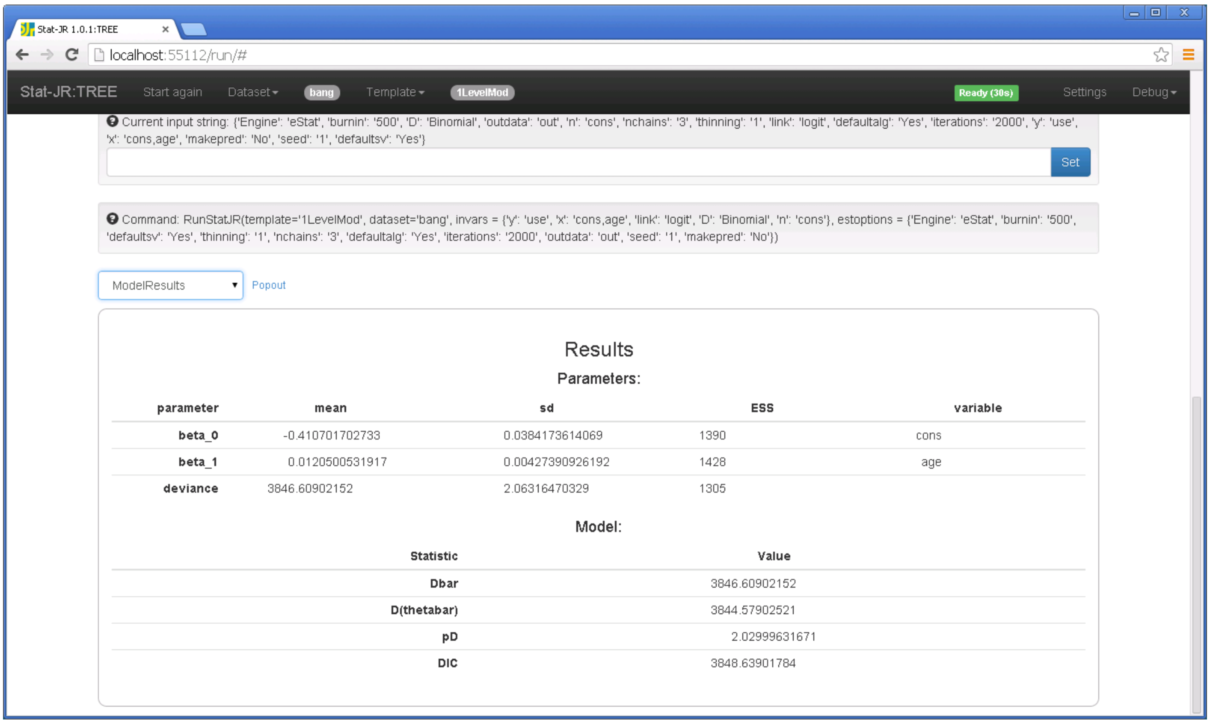


Current input string: {'Engine': 'eStat', 'burnin': '500', 'D': 'Binomial', 'outdata': 'out', 'n': 'cons', 'nchains': '3', 'thinning': '1', 'link': 'logit', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'use', 'x': 'cons,age', 'makepred': 'No', 'seed': '1', 'defaultsv': 'Yes'}

Command: RunStatJR(template='1LevelMod', dataset='bang', invars = {'y': 'use', 'x': 'cons,age', 'link': 'logit', 'D': 'Binomial', 'n': 'cons'}, estoptions = {'Engine': 'eStat', 'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'outdata': 'out', 'seed': '1', 'makepred': 'No'})

$$\begin{aligned}\mathrm{use}_i &\sim \mathrm{Binomial}(\mathrm{cons}_i, \pi_i)\\ \mathrm{logit}(\pi_i) &= \beta_0\mathrm{cons}_i + \beta_1\mathrm{age}_i\\ \beta_0 &\propto 1\\ \beta_1 &\propto 1\end{aligned}$$

We will next click on **Run** to run the model and then get the following results by selecting **ModelResults** from the objects list:



Current input string: {'Engine': 'eStat', 'burnin': '500', 'D': 'Binomial', 'outdata': 'out', 'n': 'cons', 'nchains': '3', 'thinning': '1', 'link': 'logit', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'use', 'x': 'cons,age', 'makepred': 'No', 'seed': '1', 'defaultsv': 'Yes'}

Command: RunStatJR(template='1LevelMod', dataset='bang', invars = {'y': 'use', 'x': 'cons,age', 'link': 'logit', 'D': 'Binomial', 'n': 'cons'}, estoptions = {'Engine': 'eStat', 'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'outdata': 'out', 'seed': '1', 'makepred': 'No'})

Results

Parameters:

parameter	mean	sd	ESS	variable
beta_0	-0.410701702733	0.0384173614069	1390	cons
beta_1	0.0120500531917	0.00427390926192	1428	age
deviance	3846.60902152	2.06316470329	1305	

Model:

Statistic	Value
Dbar	3846.60902152
D(thetabar)	3844.57902521
pD	2.02999631671
DIC	3848.63901784

Here we see that the age coefficient is positive and significant meaning that older women are more likely to use contraceptives. So we now want to look at the template to see what the code looks like. We will only concern ourselves with the Stat-JR built in engine here and so will not look at how the template works with interoperability as this will be an extension of the code for *Regression2* in section 5.

7.1 Inputs

The code for *inputs* is as follows:

```
inputs = '''
y = DataVector('Response: ', help= 'a.k.a. <em>Y</em>, Outcome variable,
Dependent variable, etc.')
```

```
D = Text('Specify distribution: ', ['Normal', 'Binomial', 'Poisson', '-ve
Binomial'])
if D == 'Binomial':
    n = DataVector('Denominator: ')
    link = Text('Specify link function: ', ['logit', 'probit', 'cloglog'])
if D == 'Poisson':
    link = Text(value = 'ln')
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('Offset: ')
if D == '-ve Binomial':
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('Offset: ')

x = DataMatrix('Explanatory variables: ', allow_cat = True, help= "<p
style='text-align:left'>A.k.a. X, Predictor variables, Independent
variables, etc.</p><p style='text-align:left'><strong>Note:</strong> if you
wish to include an <strong>intercept</strong> then you need to add it (e.g.
a constant of ones) as one of the explanatory variables.</p><p style='text-
align:left'>Once you've selected a variable, you have the opportunity to
indicate whether it's categorical or not; if categorical, dummy variables
will be added to the model on your behalf.</p>")

if D == 'Normal':
    tau = ParamScalar()
    sigma = ParamScalar(modelled = False)
    sigma2 = ParamScalar(modelled = False)
if D == '-ve Binomial':
    alpha = ParamScalar()
    rho = ParamVector(parents = [y])
beta = ParamVector(parents = [x], as_scalar = True)
deviance = ParamScalar(modelled = False)
'''
```

Compared to *Regression1* you will see that we have introduced an input *D* for distribution and that we introduce conditional statements (if statements). The distribution *D* is defined as a *Text* input and you will see that there are a limited number of choices given as a second argument to the statement. The *TREE* program will treat this as a pull-down list input with the limited number of choices populating the list.

As we saw in our example when fitting a Binomial model we introduce additional inputs *n* – the denominator column and *link* a *Text* based input to indicate the link function. We also see that for non-normal models there is no level 1 variance and so the quantities *tau, sigma* and *sigma2* are not included.

7.2 Engines

This template allows many estimation engines as shown below:

```
engines = ['eStat', 'WinBUGS', 'OpenBUGS', 'JAGS', 'MLwiN_MCMC',
'MLwiN_IGLS', 'R_MASS', 'R_MCMCglmm', 'R_MCMCpack',
'Stata_model','SPSS_model', 'SAS_model', 'R_INLA', 'R_RStan']
```

when we originally wrote Stat-JR each template had it's own inputs for these engines defined in a *Methodinput* function but now these are generic inputs and so simply by including an engine here, Stat-JR knows which inputs to use.

7.3 Model

The *model* attribute now also contains conditional statements as shown below:

```
model = '''
model{
    for (i in 1:length(${y})) {
        ${y}[i] ~ \\
        % if D == 'Normal':
dnorm(mu[i], tau)
        mu[i] <- \\
        % endif
        % if D == 'Binomial':
dbin(p[i], ${n}[i])
        ${link}(p[i]) <- \\
        % endif
        % if D == 'Poisson':
dpois(p[i])
        ${link}(p[i]) <- \\
        % if offset:
${n}[i] + \\
        % endif
        % endif
        % if D == '-ve Binomial':
dpois(p[i])
        p[i] <- rho[i] * exp(\\
        % if offset:
${n}[i] + \\
        % endif
        % endif
        % if D == '-ve Binomial':
${mmult(x, 'beta', 'i')})
        rho[i] ~ dgamma(alpha, alpha)
        % else:
${mmult(x, 'beta', 'i')}
        % endif
    }

    # Priors
    % for i in range(0, x.ncols()):
    beta_${i} ~ dflat()
    % endfor
    % if D == 'Normal':
    tau ~ dgamma(0.001000, 0.001000)
    sigma <- 1 / sqrt(tau)
    sigma2 <- 1 / tau
    % endif
    % if D == '-ve Binomial':
    alpha ~ dlnorm(0, 0.001)
    % endif
}
        '''
```

Basically in the *model* code, conditional statements are started by a %if and the code to be conditionally executed is ended by a %endif. The conditional statements can be hierarchical for example the line

```
% if offset:
```

is within another %if statement and now the %endif will correspond to the latest %if. In our example we have D == 'Binomial' and so the code simplifies to:

```
model = '''
model{
    for (i in 1:length(${y})) {
        ${y}[i] ~ \\
dbin(p[i], ${n}[i])
        ${link}(p[i]) <- \\
${mmult(x, 'beta', 'i')}
    }
    # Priors
    % for i in range(0, x.ncols()):
    beta_${i} ~ dflat()
    % endfor
}
        '''
```

and as we demonstrated for *Regression1* we can fill in the $ calls and unwind the %for loop and the $mmult function to get the code we can view in the *model.txt* output object.

7.4 Latex

Finally the *latex* method now also contains conditional statements.

```
latex = r'''
\begin{aligned}
%if D == 'Normal':
 \mbox{${y}}_i & \sim \mbox{N}(\mu_i, \sigma^2) \\
\mu_i & =
%endif
%if D == 'Binomial':
 \mbox{${y}}_i & \sim \mbox{Binomial}(\mbox{${n}}_i, \pi_i) \\
\mbox{${link}}(\pi_i) & =
%endif
%if D == 'Poisson':
 \mbox{${y}}_i & \sim \mbox{Poisson}(\pi_i) \\
\mbox{${link}}(\pi_i) & =
%if offset:
\mbox{${n}}_i +
%endif
%endif
  ${mmulttex(x, r'\beta', 'i')} \\
%if str(Engine) in ['eStat', 'WinBUGS', 'OpenBUGS', 'JAGS' , 'MLwiN_MCMC',
'R_MCMCglmm' ]:
%for i in range(0, len(x)):
\beta_{${i}} & \propto 1 \\
%endfor
%if D == 'Normal':
\tau & \sim \Gamma (0.001,0.001) \\
\sigma^2 & = 1 / \tau
%endif
%endif
\end{aligned}
'''
```

and as with the *model* function we achieve conditional operations via the %if and %endif pairs. Again for our example we can strip out the conditionals to get

```
latex = r'''
\begin{aligned}
\mbox{${y}}_i & \sim \mbox{Binomial}(\mbox{${n}}_i, \pi_i) \\
\mbox{${link}}(\pi_i) & =
  ${mmulttex(x, r'\beta', 'i')} \\
%for i in range(0, len(x)):
\beta_{${i}} & \propto 1 \\
%endfor
\end{aligned}
'''
```

If you look at the code you will see other functions for the various other software packages but we will not discuss these here.

Exercise 6

Convert the more general *1LevelMod* template into a specific logistic regression template. To do this copy *1LevelMod.py* to *1LevelLogit.py* and simply remove the conditional statements and additional options so that the template only allows the user to fit logistic regression models. You can check the template works by attempting the example given in the section with your new template.

8 Including categorical predictors

Originally in Stat-JR all predictor variables were assumed to be continuous and so if a predictor was categorical, for example school gender in the tutoral dataset, we would need some method to transform the original form to a series of dummy variables. To this end several templates were created that performed this transformation as part of the template in an attribute called *preparedata*. We will look at an example of this in a minute with the *1LevelCatRef* template which allows the user to specify variables as categorical and to specify which category is the reference. In the latest version of Stat-JR functionality has been built in to allow the user to specify that an input might be categorical and so for example in the *inputs* in *1LevelMod* you will see the line:

```
x = DataMatrix('Explanatory variables: ', allow_cat = True, help= "<p
style='text-align:left'>A.k.a. X, Predictor variables, Independent
variables, etc.</p><p style='text-align:left'><strong>Note:</strong> if you
wish to include an <strong>intercept</strong> then you need to add it (e.g.
a constant of ones) as one of the explanatory variables.</p><p style='text-
align:left'>Once you've selected a variable, you have the opportunity to
indicate whether it's categorical or not; if categorical, dummy variables
will be added to the model on your behalf.</p>")
```

which tells Stat-JR to ask for each element of *x* whether it is categorical or not and then within each package file there is code to construct the dummy variables. This method is restricted in that it always chooses the first category to be the reference. We will here look at an alternative template that has built in functionality for constructing these categorical variables within the template. This template is called *1LevelCatRef* and we will first look at its *inputs* attribute to see how it gets the user to input the model structure before demonstrating its use on the *tutorial* dataset.

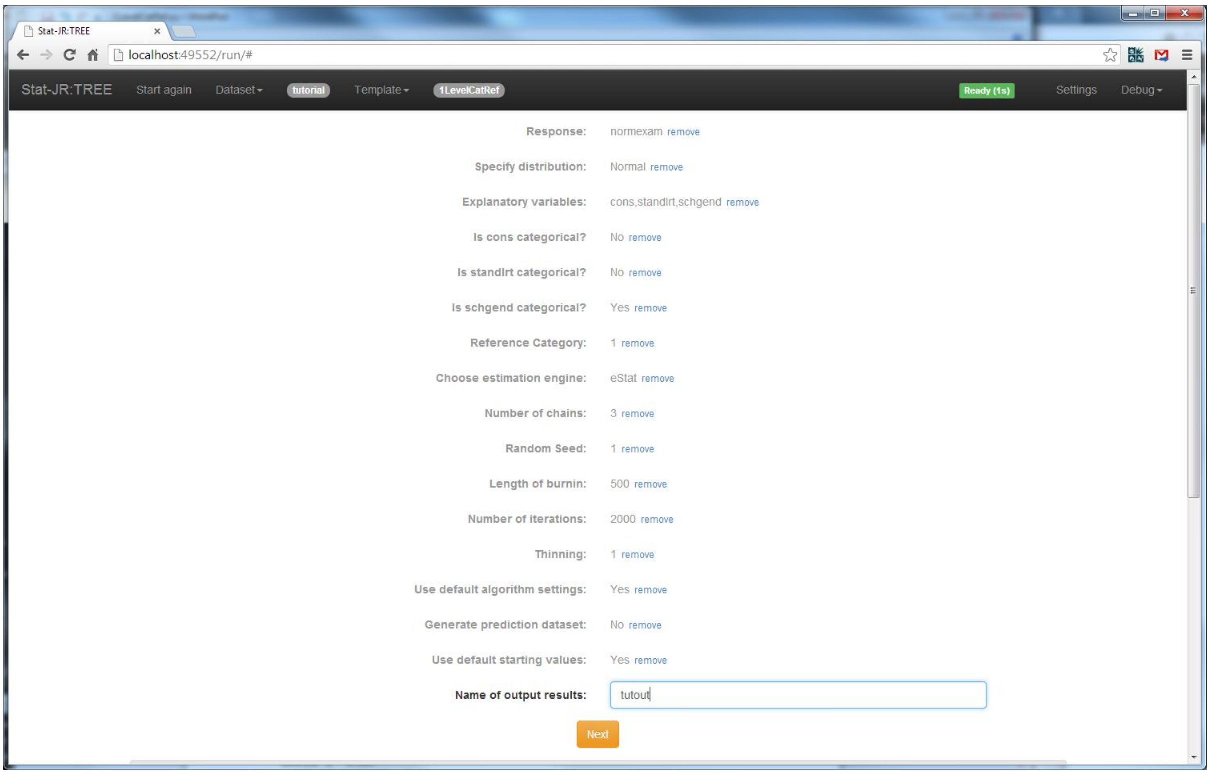
The *inputs* code is as follows:

```
inputs = '''
y = DataVector('Response: ')
D = Text('Specify distribution: ', ['Normal', 'Binomial', 'Poisson'])
if D == 'Binomial':
    n = DataVector('Denominator: ')
    link = Text('Specify link function: ', ['logit', 'probit', 'cloglog'])
if D == 'Poisson':
    link = Text(value = 'ln')
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('Offset: ')
x = DataMatrix('Explanatory variables: ')
for var in x:
    context[var + '_cat'] = Boolean('Is ' + var + ' categorical? ')
    if context[var + '_cat']:
        context[var + '_ref'] = Text('Reference Category: ')

origx = Text(value = [])
if D == 'Normal':
    tau = ParamScalar()
    sigma = ParamScalar(modelled = False)
    sigma2 = ParamScalar(modelled = False)
beta = ParamVector(parents = [x])
deviance = ParamScalar(modelled = False)
'''
```

This code section is much the same as that in *1LevelMod* (aside from not doing negative binomial) upto the point that *x* is input. We next see a *for* loop that includes the use of the *context* statement which is used to construct attribute names that are a combination of text and variable names. If for example *x* contains the three variable list [‘cons’,’standlrt’,’schgend’] then the context statements will create 3 variables ‘cons_cat’, ‘standlrt_cat’ and ‘schgend_cat’ which will store the text strings ‘yes’ or ‘no’ depending on whether the variables are categorical or not. If yes then a further context statement is used to construct a variable to house the reference category for that variable. The line

```
origx = Text(value = [])
```

will be used to store the original x variables prior to manipulating the categorical variables. By setting its value in the assignment we will not get an input widget appearing in the browser.
Let us demonstrate fitting this model so choose *1LevelCatRef* from the template list and *tutorial* as the dataset. Note if you have previously used the **recode** template on this dataset, on the main menu click on **Debug/Reload datasets** to get back the original *tutorial* dataset. Firstly we will choose the inputs as follows:



Response: normexam remove

Specify distribution: Normal remove

Explanatory variables: cons,standlrt,schgend remove

Is cons categorical? No remove

Is standlrt categorical? No remove

Is schgend categorical? Yes remove

Reference Category: 1 remove

Choose estimation engine: eStat remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

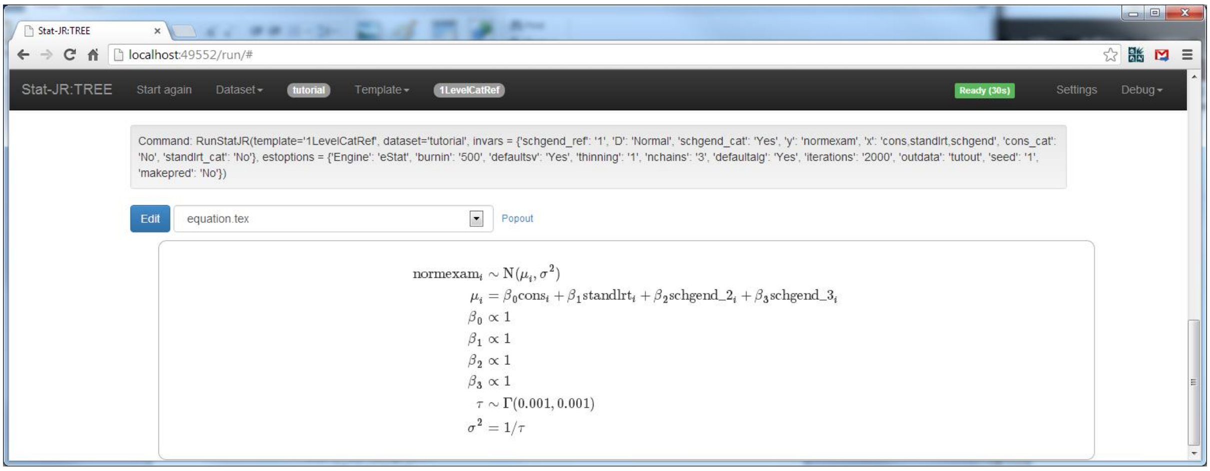
Use default algorithm settings: Yes remove

Generate prediction dataset: No remove

Use default starting values: Yes remove

Name of output results: tutout

Next we click on the **Next** button and we will be able to look at the equation for the model:



```
Command: RunStatJR(template='1LevelCatRef', dataset='tutorial', invars = {'schgend_ref': '1', 'D': 'Normal', 'schgend_cat': 'Yes', 'y': 'normexam', 'x': 'cons,standlrt,schgend', 'cons_cat': 'No', 'standlrt_cat': 'No'}, estoptions = {'Engine': 'eStat', 'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'outdata': 'tutout', 'seed': '1', 'makepred': 'No'})
```

$$\begin{aligned}
\mathrm{normexam}_i &\sim \mathrm{N}(\mu_i, \sigma^2)\\
\mu_i &= \beta_0 \mathrm{cons}_i + \beta_1 \mathrm{standlrt}_i + \beta_2 \mathrm{schgend\_2}_i + \beta_3 \mathrm{schgend\_3}_i\\
\beta_0 &\propto 1\\
\beta_1 &\propto 1\\
\beta_2 &\propto 1\\
\beta_3 &\propto 1\\
\tau &\sim \Gamma(0.001, 0.001)\\
\sigma^2 &= 1/\tau
\end{aligned}$$

Here we see that in the maths that the expression for the linear predictor has two terms to represent two of the possible categories for school gender (*schgend_2* and *schgend_3*).
The important attribute here is *preparedata*. The *preparedata* attribute allows for template specific data manipulations to be executed prior to the model run. In this case the code is as overleaf:

```
preparedata = '''
mydata = data['datafile']
for var in x:
    origx.name.append(var) # Save user's original selection

del x[:]
for var in origx.name:
    if context[var + '_cat']:
        uniqvals = list(set(mydata.variables[var]['data'].compressed()))
        uniqvals.sort()
        uniqvals.remove(int(context[var + '_ref']))
        for i in uniqvals:
            if int(i)<0:
                lab = 'neg'+str(abs(int(i)))
            else :
                lab = str(int(i))
            mydata.addvariable(var + '_' + lab, data =
(mydata.variables[var]['data'][:] == i).astype(float))
            x.name.append(var + '_' + lab)
    else:
        # TODO: fix this
        x.name.append(var)

beta.ncols = len(x)
'''
```

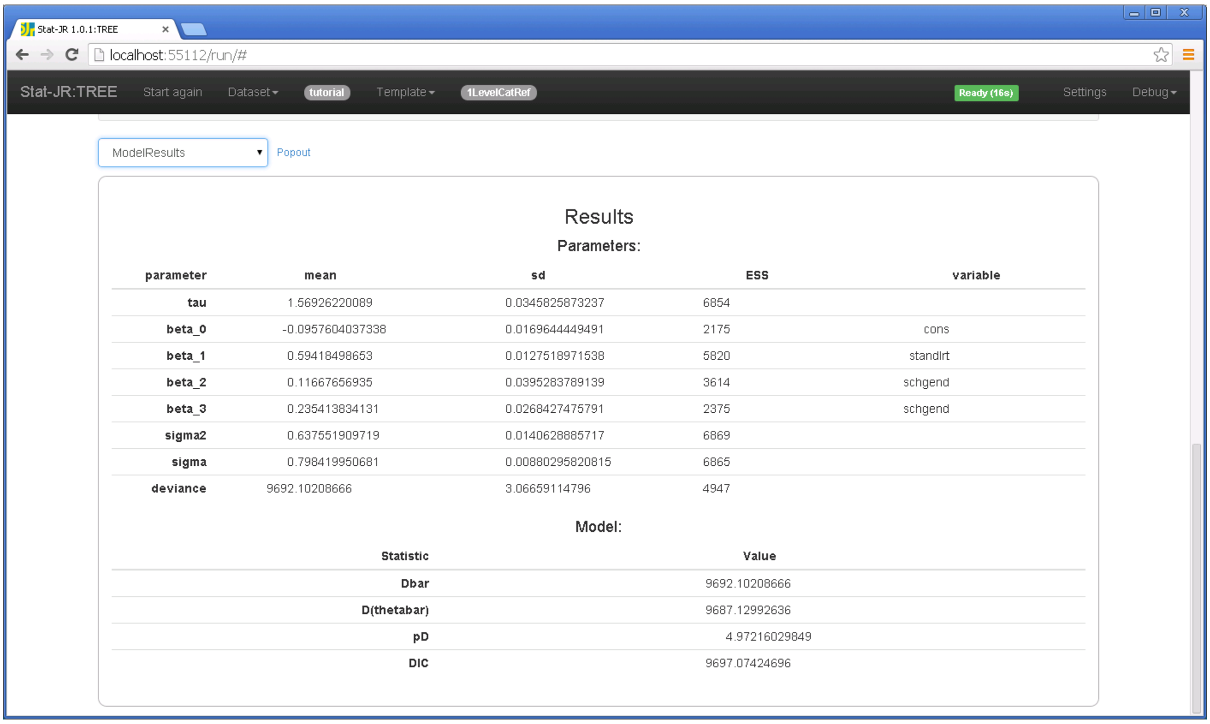
This code firstly retains the named predictor variables in *origx* by copying the contents of *x* to *origx* and then deleting them from *x*. Then the code loops over the variables via the second *for* statement and conditionally (the *if* statement) on a particular variable being categorical does some processing. The lines

```
                uniqvals =
list(set(mydata.variables[var]['data'].compressed()))
                uniqvals.sort()
                uniqvals.remove(int(context[var + '_ref']))
```

firstly find all unique values in the categorical predictor which are then stored in *uniqvals*. We then sort these into ascending order before removing the user defined reference category as it will play the role as the base category in the model. We then have a second loop over this list of *uniqvals* where we create the dummy variables. The lines

```
mydata.addvariable(var + '_' + lab, data =
(mydata.variables[var]['data'][:] == i).astype(float))
            x.name.append(var + '_' + lab)
```

firstly construct an array which takes value 1 if the original variable has value *i* or 0 otherwise. This newly constructed predictor variable is then appended to the new variable list. If the variable is not categorical it is simply added to this new variable list itself. We finally adjust the length of beta to account for the expansion of the categorical variables and return the new dataset.
This *preparedata* method is run before the *model* and *latex* attributes and so these are similar to those we saw in *1LevelMod*. To continue running the example we can press the **Run** button and then select the **ModelResults** object from the list to get the following results:



Results

Parameters:

parameter	mean	sd	ESS	variable
tau	1.56926220089	0.0345825873237	6854	
beta_0	-0.0957604037338	0.0169644449491	2175	cons
beta_1	0.59418498653	0.0127518971538	5820	standlrt
beta_2	0.11667656935	0.0395283789139	3614	schgend
beta_3	0.235413834131	0.0268427475791	2375	schgend
sigma2	0.637551909719	0.0140628885717	6869	
sigma	0.798419950681	0.00880295820815	6865	
deviance	9692.10208666	3.06659114796	4947	

Model:

Statistic	Value
Dbar	9692.10208666
D(thetabar)	9687.12992636
pD	4.97216029849
DIC	9697.07424696

This completes this section and is the last single level model we will meet for a while. Another extension would be to allow the inclusion of interactions into the model. This has been done in the template *1LevelInteractions* (which is available from the template repository but not part of the core release). Here the modifications are done in the *inputs* and *model/latex* methods as no new predictor variables are created. Instead the model code includes multiplications between the variables. We will leave you to try out this template as an exercise.

9 Multilevel models

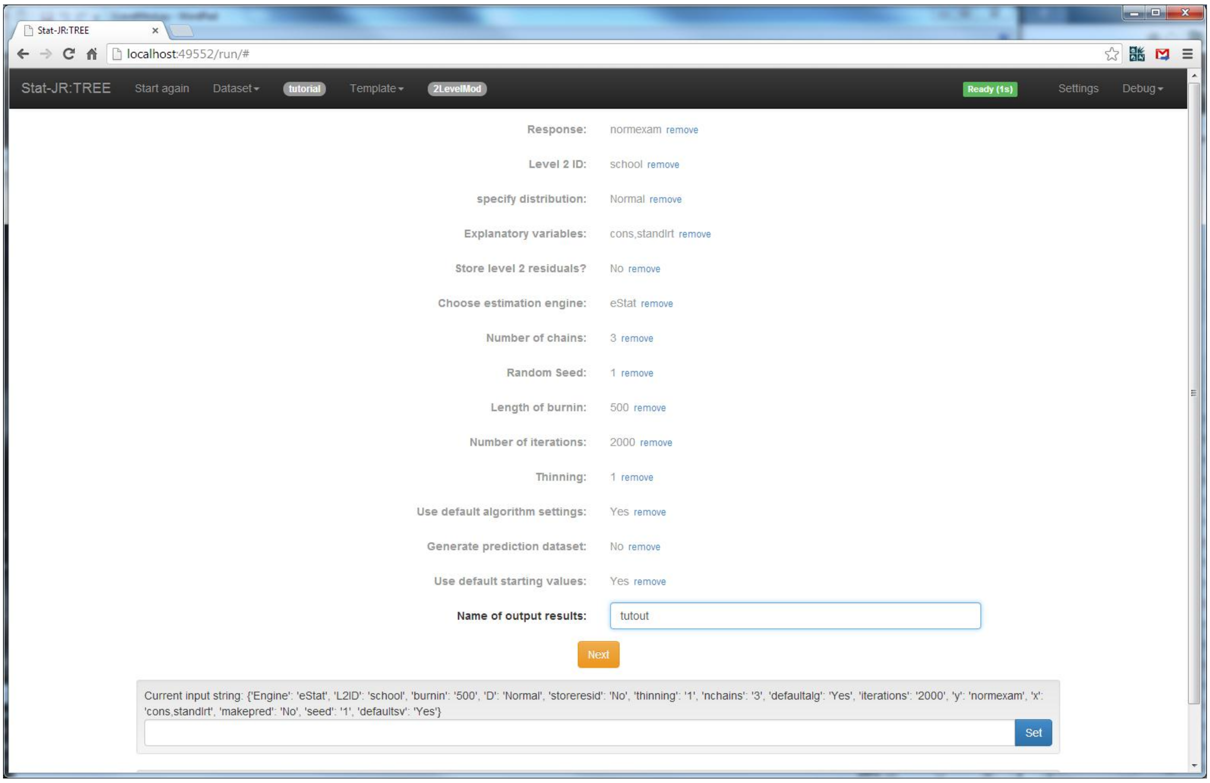
Our next step is to move onto templates for models for more complex data structures. In this section we look at multilevel modelling templates – templates that allow random effects to account for clustering in the data. We will look at two templates of increasing complexity, firstly a template for fitting models that have 2 levels i.e. 1 higher level of clustering and then secondly a more general template that will fit models with any number of levels clustering whether nested or crossed. Note here that these templates allow only random intercepts in the models we are fitting.

9.1 2LevelMod template

We will begin our investigation of *2LevelMod* by looking at its *inputs* attribute. Note that here and later we have stripped out the help text from some of the inputs for readability:

```
inputs = '''
y = DataVector('Response: ')
L2ID = IDVector('Level 2 ID: ')
D = Text('specify distribution: ', ['Normal', 'Poisson', 'Binomial'])
if D == 'Binomial':
    n = DataVector('Denominator: ')
    link = Text('Specify link function: ', ['logit', 'probit', 'cloglog'])
if D == 'Poisson':
    link = Text(value = 'ln')
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('Offset: ')

x = DataMatrix('Explanatory variables: ', allow_cat = True)
storeresid = Boolean('Store level 2 residuals?')

if D == 'Normal':
    tau = ParamScalar()
    sigma2 = ParamScalar(modelled = False)

beta = ParamVector(parents=[x], as_scalar=True)

if storeresid:
    u = ParamVector(parents=[L2ID], as_scalar=False)
else:
    u = ParamVector(parents=[L2ID], as_scalar=False, monitor=False)
tau_u = ParamScalar()
sigma2_u = ParamScalar(modelled = False)
deviance = ParamScalar(modelled = False)
'''
```

If you compare this with the *inputs* function for *1LevelMod* you will see we have added 2 additional inputs: *L2ID* to allow the user to input the column containing the level 2 identifiers and *storeresid*, a Boolean indicator of whether to store the level 2 residuals or not. We also have three additional parameters *u, tau_u* and *sigma2_u* (to represent the level 2 residuals, their precision and variance respectiviely) that have been included. We can try out an example of these inputs by selecting the template *2LevelMod* and the dataset *tutorial* and applying the following inputs:



Response: normexam remove

Level 2 ID: school remove

specify distribution: Normal remove

Explanatory variables: cons,standlrt remove

Store level 2 residuals? No remove

Choose estimation engine: eStat remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

Use default algorithm settings: Yes remove

Generate prediction dataset: No remove

Use default starting values: Yes remove

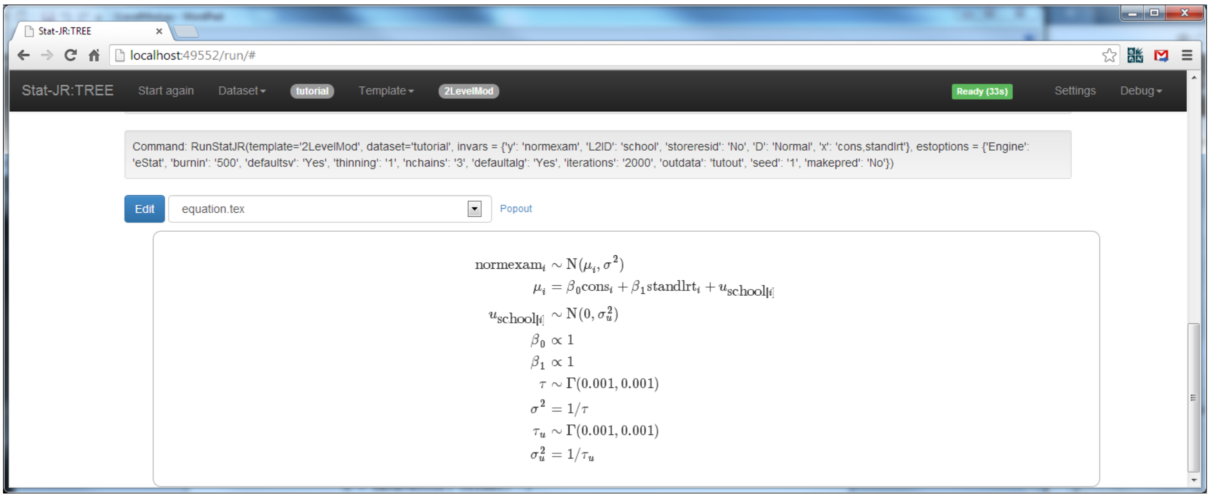
Name of output results: tutout

Next

Current input string: {'Engine': 'eStat', 'L2ID': 'school', 'burnin': '500', 'D': 'Normal', 'storeresid': 'No', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'makepred': 'No', 'seed': '1', 'defaultsv': 'Yes'}

Set

Clicking on **Next** we will see **equation.tex** in the bottom pane:



Command: RunStatJR(template='2LevelMod', dataset='tutorial', invars = {'y': 'normexam', 'L2ID': 'school', 'storeresid': 'No', 'D': 'Normal', 'x': 'cons,standlrt'}, estoptions = {'Engine': 'eStat', 'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'outdata': 'tutout', 'seed': '1', 'makepred': 'No'})

$$\begin{aligned}
\mathrm{normexam}_i &\sim \mathrm{N}(\mu_i, \sigma^2)\\
\mu_i &= \beta_0 \mathrm{cons}_i + \beta_1 \mathrm{standlrt}_i + u_{\mathrm{school[}i]}\\
u_{\mathrm{school[}i]} &\sim \mathrm{N}(0, \sigma_u^2)\\
\beta_0 &\propto 1\\
\beta_1 &\propto 1\\
\tau &\sim \Gamma(0.001, 0.001)\\
\sigma^2 &= 1/\tau\\
\tau_u &\sim \Gamma(0.001, 0.001)\\
\sigma_u^2 &= 1/\tau_u
\end{aligned}$$

Here we see a mathematical representation of the model created in *latex*. Let's look next at *model*:

```
model = '''
model {
    for (i in 1:length(${y})) {
        ${y}[i] ~ \\
        % if D == 'Normal':
dnorm(mu[i], tau)
        mu[i] <- \\
        % endif
        % if D == 'Binomial':
dbin(p[i], ${n}[i])
        ${link}(p[i]) <- \\
        % endif
        % if D == 'Poisson':
dpois(p[i])
        ${link}(p[i]) <- \\
        % if offset:
${n}[i] + \\
        % endif
        % endif
${mmult(x, 'beta', 'i')} + u[${L2ID}[i]]
    }

    for (j in 1:length(u)) {
        u[j] ~ dnorm(0, tau_u)
    }

    # Priors
    % for i in range(0, x.ncols()):
    beta_${i} ~ dflat()
    % endfor

    % if D == 'Normal':
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    % endif

    tau_u ~ dgamma(0.001000, 0.001000)
    sigma2_u <- 1 / tau_u
}
'''
```

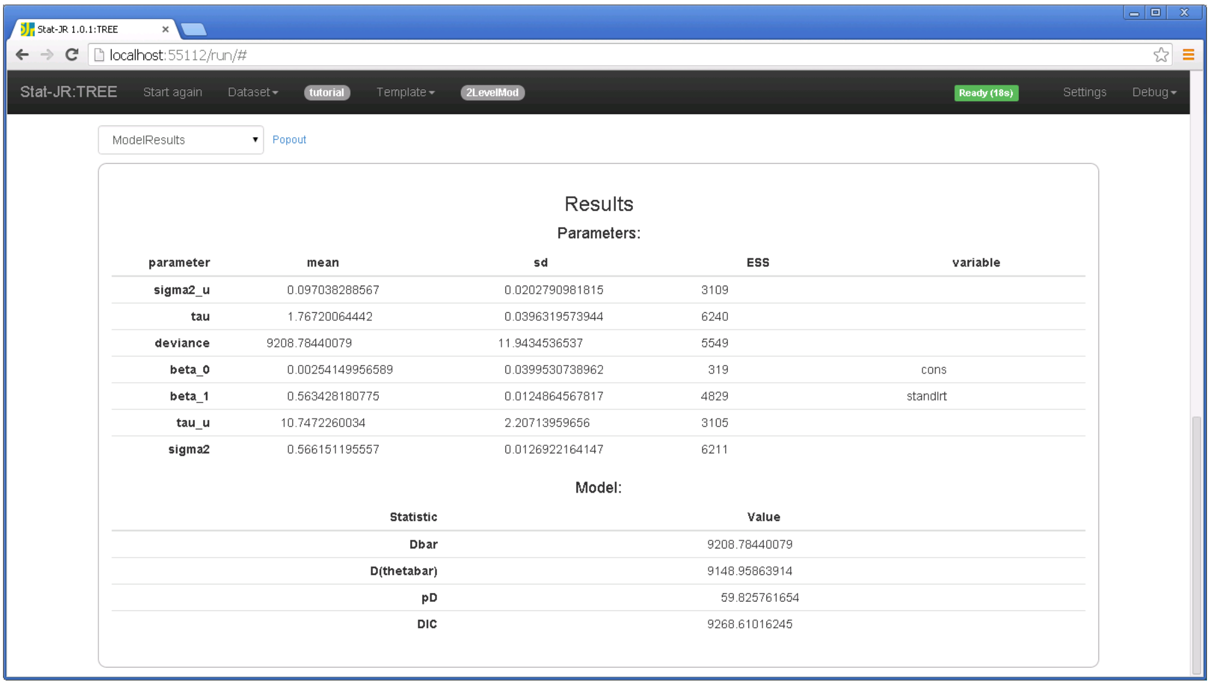
The code has become quite long mainly due to the conditional statements for the different distribution types. We see that the term `u[${L2ID}[i]]` has been appended to the linear predictor where L2ID is inserted for a particular model. The chunk of code

```
    for (j in 1:length(u)) {
        u[j] ~ dnorm(0, tau_u)
    }
```

then gives the random effect distribution and finally the chunk

```
    tau_u ~ dgamma(0.001000, 0.001000)
    sigma2_u <- 1 / tau_u
```

gives a prior distribution for the variance of the random effects. The *latex* function is adapted in very similar ways and so for brevity we omit this code here. We will finish off this template by running it and looking at **ModelResults** (in a new tab). If we do this we only get results for the variables that have had their chains stored:



Results

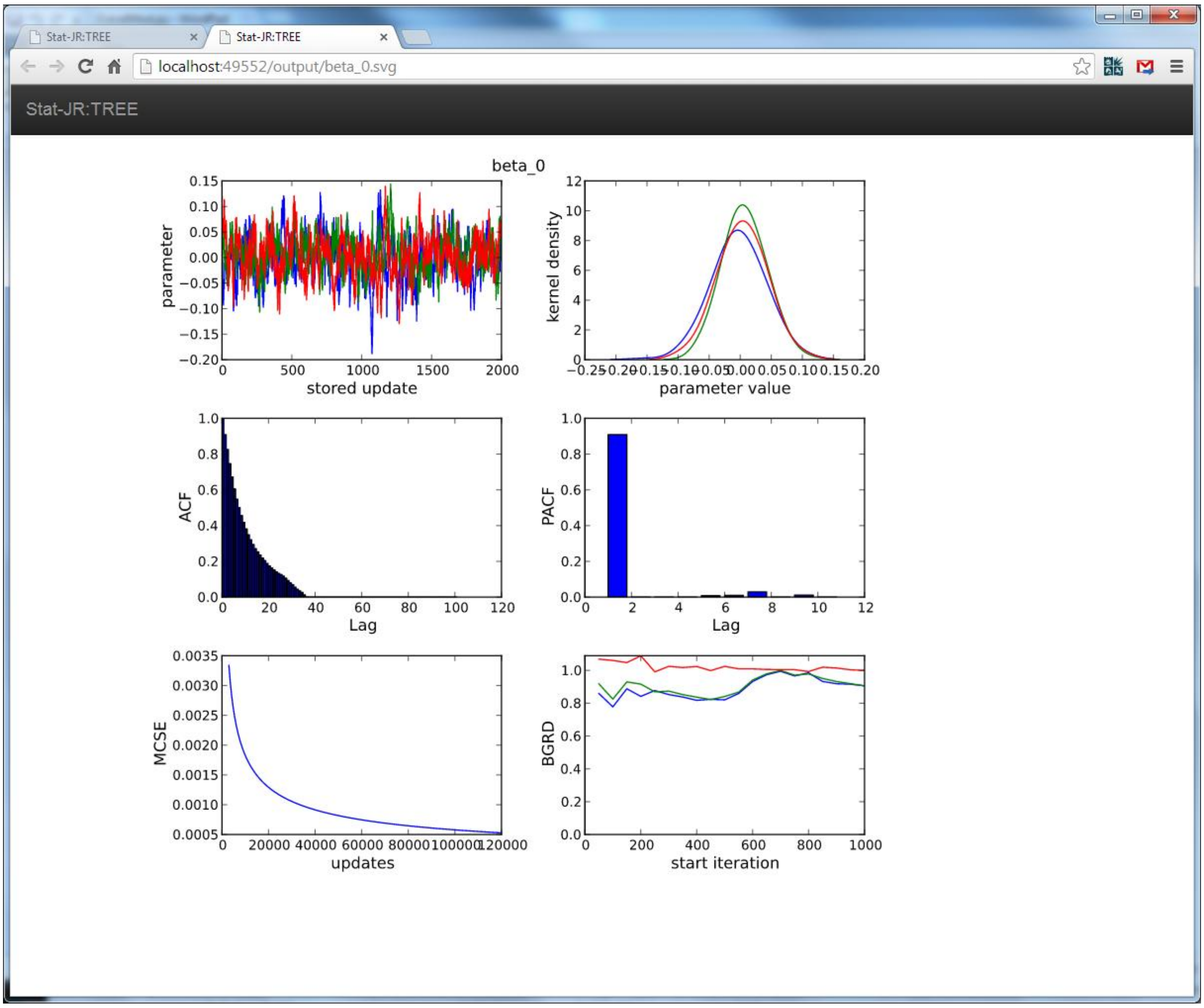
Parameters:

parameter	mean	sd	ESS	variable
sigma2_u	0.097038288567	0.0202790981815	3109	
tau	1.76720064442	0.0396319573944	6240	
deviance	9208.78440079	11.9434536537	5549	
beta_0	0.00254149956589	0.0399530738962	319	cons
beta_1	0.563428180775	0.0124864567817	4829	standlrt
tau_u	10.7472260034	2.20713959656	3105	
sigma2	0.566151195557	0.0126922164147	6211	

Model:

Statistic	Value
Dbar	9208.78440079
D(thetabar)	9148.95863914
pD	59.825761654
DIC	9268.61016245

Basically although we didn't store chains for each of the 65 random effects *u* we can store summary statistics for them but by default we do not display them unless we change the output options on the *settings* screen. As usual we also can get the MCMC plots e.g. for *beta_0.svg:*



Exercise 7

Try adapting this template so that it allows the user to incorporate interactions.

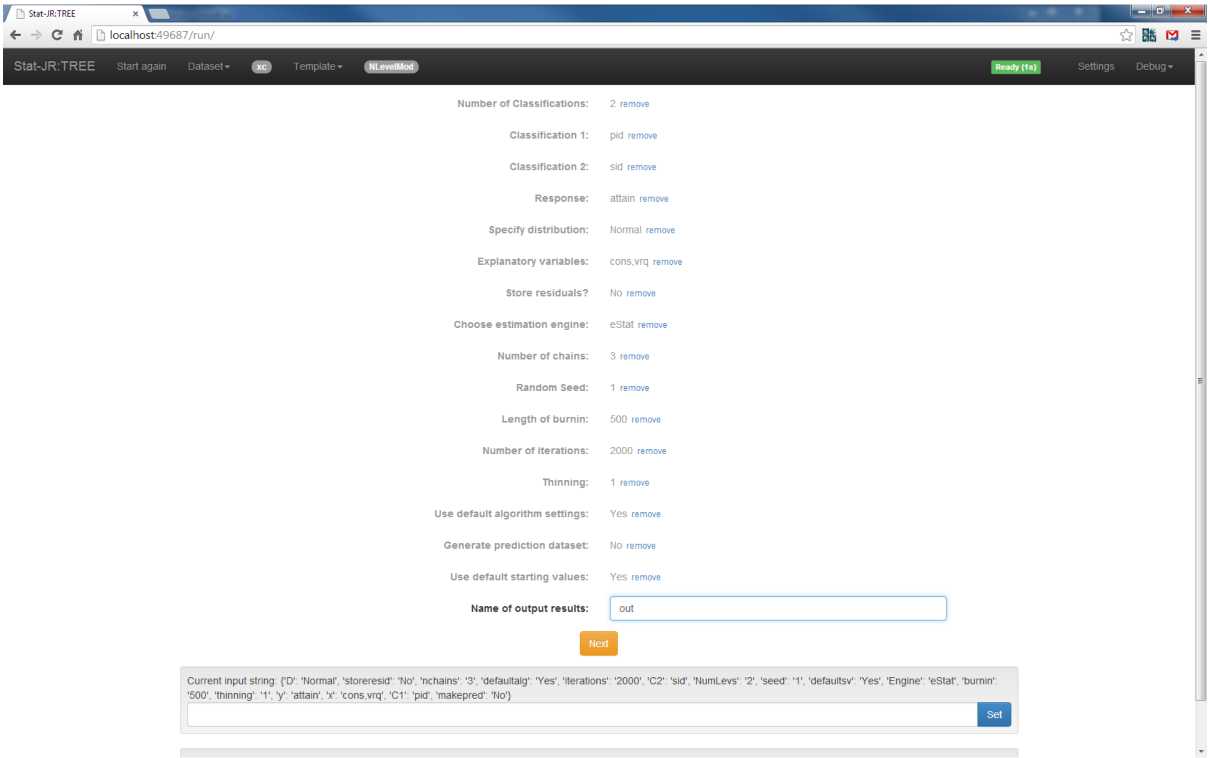
9.2 NLevelMod template

The *NlevelMod* template as the name suggests extends the *2LevelMod* template to an unlimited number (input by the user) of levels of clustering. Note that these clusters can be either nested or cross-classified. We will once again start by looking at the *inputs* attribute to see how it differs from *2LevelMod*:

```
inputs = '''
NumLevs = Integer('Number of Classifications: ')
for i in range(0, int(NumLevs)):
    selstr = 'Classification ' + str(i + 1) + ': '
    context['C' + str(i + 1)] = IDVector(selstr)

y = DataVector('Response: ')
D = Text('Specify distribution: ', ['Normal', 'Binomial', 'Poisson'])
if D == 'Binomial':
    n = DataVector('Denominator: ')
    link = Text('Specify link function: ', ['logit', 'probit', 'cloglog'])
if D == 'Poisson':
    link = Text(value = 'ln')
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('Offset: ')
x = DataMatrix('Explanatory variables: ', allow_cat = True)
storeresid = Boolean('Store residuals?')

beta = ParamVector(parents=[x], as_scalar=True)
if D == 'Normal':
    tau = ParamScalar()
    sigma2 = ParamScalar(modelled = False)

for i in range(0, int(NumLevs)):
    if storeresid:
        context['u' + str(i + 1)] = ParamVector(parents=[context['C' +
str(i + 1)]], as_scalar=False)
    else:
        context['u' + str(i + 1)] = ParamVector(parents=[context['C' +
str(i + 1)]], as_scalar=False, monitor=False)
    context['tau_u' + str(i + 1)] = ParamScalar()
    context['sigma2_u' + str(i + 1)] = ParamScalar(modelled = False)

deviance = ParamScalar(modelled = False)
'''
```

Here we have needed to replace the code for inputting the level 2 identifier with code to input the number of classifications (levels of clustering) and then we have looped over the number of classifications constructing both the names of the columns that contain the classification vectors (which will labelled *C1*, *C2* …) and, towards the bottom of the code, the new parameters associated with each classification (*u1, tau_u1* and *sigma2_u1* etc). To achieve these inputs we have used the *context* command to construct attribute names by concatenating strings and also a simple string concatenation to create *selstr* which contains the question associated with inputting each classification name. The rest of the code is similar to before. It should be noted that a 2 level model in this template has 1 classification as we are not considering level 1 here.

We can consider using this template on a cross-classified example with two higher classifications. This example is a dataset from Fife in Scotland where we are looking at the impact of both primary school and secondary school on the attainment of children at age 16. To do this select the template *NLevelMod* from the *template* list and the dataset *xc* from the *dataset* list. Select the inputs as shown:



Number of Classifications: 2 remove

Classification 1: pid remove

Classification 2: sid remove

Response: attain remove

Specify distribution: Normal remove

Explanatory variables: cons,vrq remove

Store residuals? No remove

Choose estimation engine: eStat remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

Use default algorithm settings: Yes remove

Generate prediction dataset: No remove

Use default starting values: Yes remove

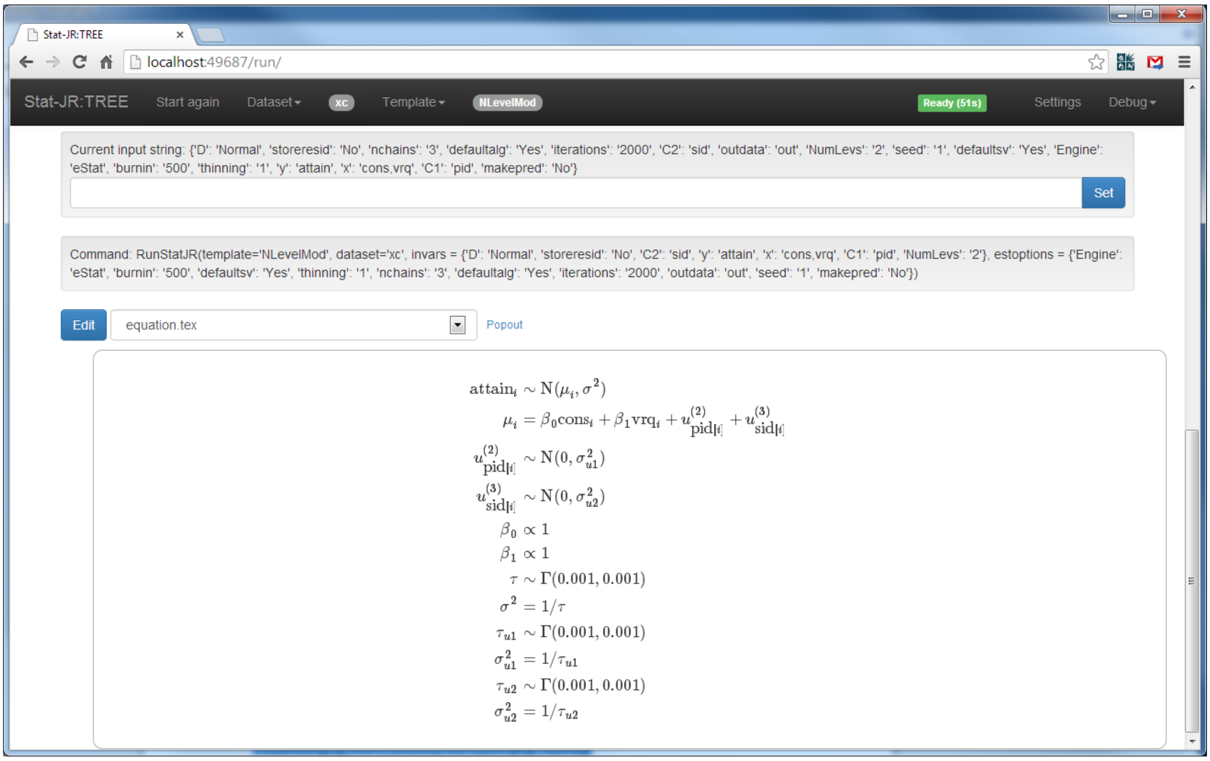
Name of output results: out

Next

Current input string: {'D': 'Normal', 'storeresid': 'No', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'C2': 'sid', 'NumLevs': '2', 'seed': '1', 'defaultsv': 'Yes', 'Engine': 'eStat', 'burnin': '500', 'thinning': '1', 'y': 'attain', 'x': 'cons,vrq', 'C1': 'pid', 'makepred': 'No'}

Set

Clicking on the **Next** button will display the mathematical formulation of the model (**equation.tex)** in the pull down list:



Current input string: {'D': 'Normal', 'storeresid': 'No', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'C2': 'sid', 'outdata': 'out', 'NumLevs': '2', 'seed': '1', 'defaultsv': 'Yes', 'Engine': 'eStat', 'burnin': '500', 'thinning': '1', 'y': 'attain', 'x': 'cons,vrq', 'C1': 'pid', 'makepred': 'No'}

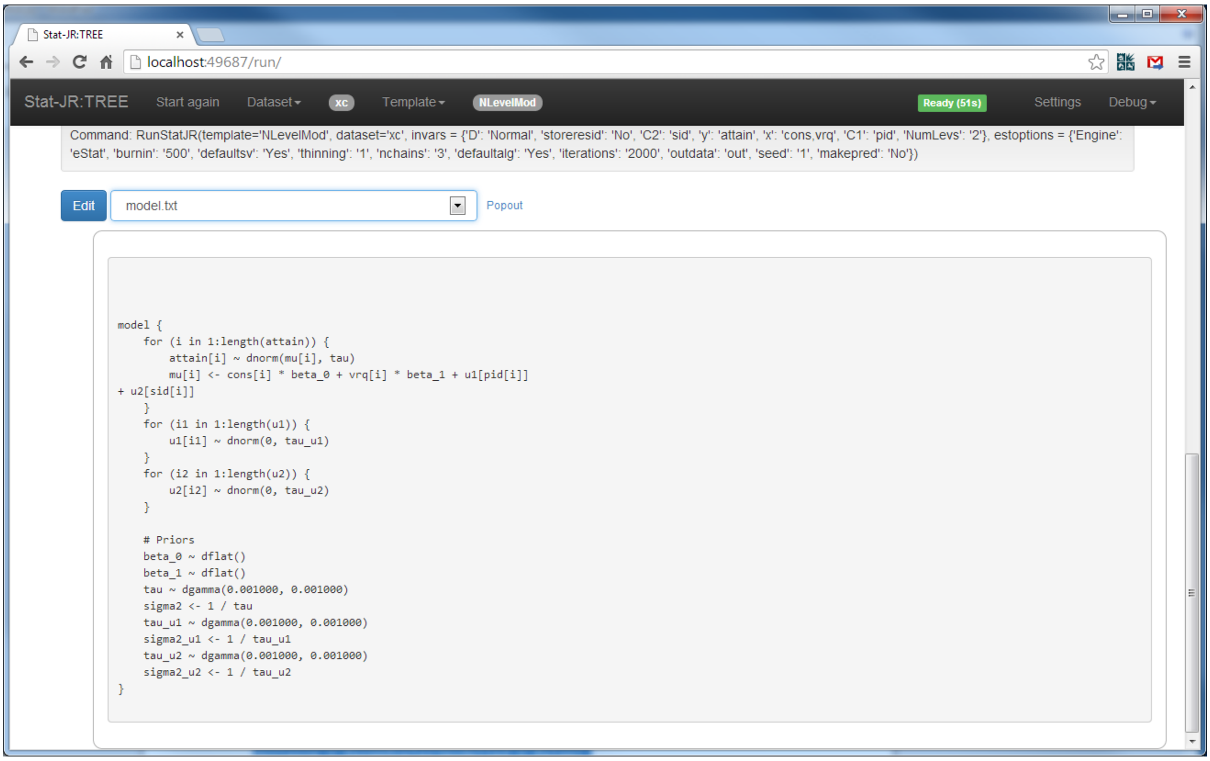
Set

Command: RunStatJR(template='NLevelMod', dataset='xc', invars = {'D': 'Normal', 'storeresid': 'No', 'C2': 'sid', 'y': 'attain', 'x': 'cons,vrq', 'C1': 'pid', 'NumLevs': '2'}, estoptions = {'Engine': 'eStat', 'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'outdata': 'out', 'seed': '1', 'makepred': 'No'})

Edit equation.tex Popout

$$
\begin{aligned}
\mathrm{attain}_i &\sim \mathrm{N}(\mu_i, \sigma^2)\\
\mu_i &= \beta_0\mathrm{cons}_i + \beta_1\mathrm{vrq}_i + u^{(2)}_{\mathrm{pid}[i]} + u^{(3)}_{\mathrm{sid}[i]}\\
u^{(2)}_{\mathrm{pid}[i]} &\sim \mathrm{N}(0, \sigma^2_{u1})\\
u^{(3)}_{\mathrm{sid}[i]} &\sim \mathrm{N}(0, \sigma^2_{u2})\\
\beta_0 &\propto 1\\
\beta_1 &\propto 1\\
\tau &\sim \Gamma(0.001, 0.001)\\
\sigma^2 &= 1/\tau\\
\tau_{u1} &\sim \Gamma(0.001, 0.001)\\
\sigma^2_{u1} &= 1/\tau_{u1}\\
\tau_{u2} &\sim \Gamma(0.001, 0.001)\\
\sigma^2_{u2} &= 1/\tau_{u2}
\end{aligned}
$$

If we select **model.txt** from the list we can see the model code. It is similar to that we had for the 2 level model and *2LevelMod*.



Command: RunStatJR(template='NLevelMod', dataset='xc', invars = {'D': 'Normal', 'storeresid': 'No', 'C2': 'sid', 'y': 'attain', 'x': 'cons,vrq', 'C1': 'pid', 'NumLevs': '2'}, estoptions = {'Engine': 'eStat', 'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'outdata': 'out', 'seed': '1', 'makepred': 'No'})

```
model {
    for (i in 1:length(attain)) {
        attain[i] ~ dnorm(mu[i], tau)
        mu[i] <- cons[i] * beta_0 + vrq[i] * beta_1 + u1[pid[i]]
+ u2[sid[i]]
    }
    for (i1 in 1:length(u1)) {
        u1[i1] ~ dnorm(0, tau_u1)
    }
    for (i2 in 1:length(u2)) {
        u2[i2] ~ dnorm(0, tau_u2)
    }

    # Priors
    beta_0 ~ dflat()
    beta_1 ~ dflat()
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    tau_u1 ~ dgamma(0.001000, 0.001000)
    sigma2_u1 <- 1 / tau_u1
    tau_u2 ~ dgamma(0.001000, 0.001000)
    sigma2_u2 <- 1 / tau_u2
}
```

The model code is created by the *model* attribute and here we see the code:

```
    model = '''
<% numlevs = int(NumLevs) %>

model {
    for (i in 1:length(${y})) {
        ${y}[i] ~ \\
        % if D == 'Normal':
dnorm(mu[i], tau)
        mu[i] <- \\
        % endif
        % if D == 'Binomial':
dbin(p[i], ${n}[i])
        ${link}(p[i]) <- \\
        % endif
        % if D == 'Poisson':
dpois(p[i])
        ${link}(p[i]) <-
        % if offset:
${n}[i] + \\
        % endif
        % endif
${mmult(x, 'beta', 'i')} \\
        % for i in range(0, numlevs):
+ u${i + 1}[${context['C' + str(i + 1)]}[i]]
        % endfor
    }
    % for i in range(0, numlevs):
    for (i${i + 1} in 1:length(u${i + 1})) {
        u${i + 1}[i${i + 1}] ~ dnorm(0, tau_u${i + 1})
    }
    % endfor
    # Priors
    % for i in range(0, x.ncols()):
    beta_${i} ~ dflat()
    % endfor
    % if D == 'Normal':
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    % endif
    % for i in range(0, numlevs):
    tau_u${i + 1} ~ dgamma(0.001000, 0.001000)
    sigma2_u${i + 1} <- 1 / tau_u${i + 1}
    % endfor
}
'''
```

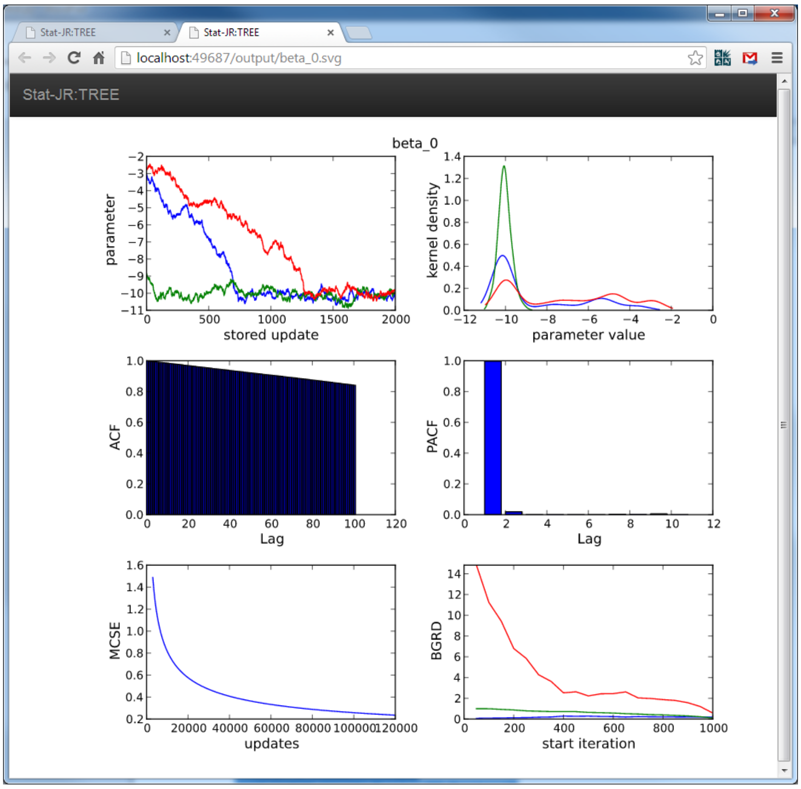
Here we introduce the use of local variable *numlevs*. Basically the *model* attribute is a text string with substitutions. Then if we wish to include a Python statement, whilst inside the text string, we place it within a <% and a %>. In this case we set a value to *numlevs* and then use it as a looping upper bound later in the code. You will see that the rest of the code contains many of the features we have discussed in earlier examples. You do however have to be careful as the code is such a mixture of WinBUGS like model code and Python code. For example if we consider the chunk:

```
% for i in range(0, numlevs):
    for (i${i + 1} in 1:length(u${i + 1})) {
        u${i + 1}[i${i + 1}] ~ dnorm(0, tau_u${i + 1})
    }
    % endfor
```

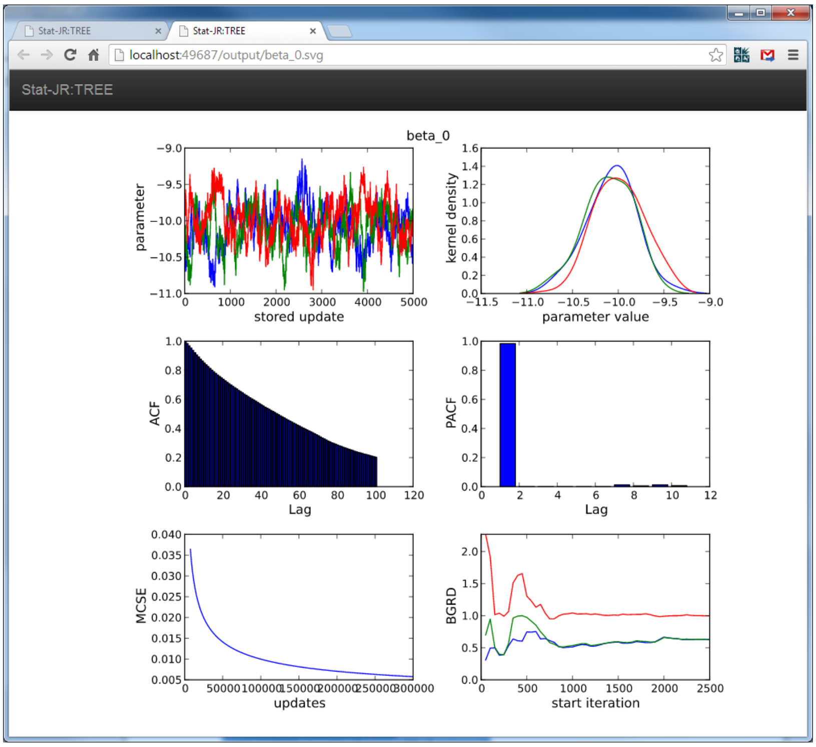
with our cross-classified model. Here we are using *i* in both the model code we are constructing and as a python variable. So *numlevels* in our example is 2 and so the outside %for (Python) can be expanded out and the *i* substitutions made and we get:

```
for (i1 in 1:length(u1)) {
        u1[i1] ~ dnorm(0, tau_u1)
    }
for (i2 in 1:length(u2)) {
        u2[i2] ~ dnorm(0, tau_u2)
    }
```

as we see in the browser. Once again the *latex* function which creates the LaTeX output will have similar substitutions via Python but we will not describe this in detail here.
Clicking on **Run** will run the model and the output contains information for all the parameters but not the residuals as we didn't ask to store them and the usual MCMC plots are available.
Interestingly the ESS values for beta_0 and beta_1 are poor and if we look at the object **beta_0.svg** We see the following:



Clearly in this case the 500 burnin was not long enough and convergence has not been achieved. If we modify the burnin to 2000 and main run to 5000 we get the following:



There are several other N level modelling templates included with the software that you can also look at. We will describe one further such template (*NLevelRS*) which allows random slopes in section 11. This template will need to utilise the *preccode* feature and so we will first explain this with a simpler 1 level example.

Exercise 8
Try adapting this template to allow interactions between predictors calling your new templates *nlevelint*.

10 Using the Preccode method

One of the aims of the Stat-JR system is to allow other estimation engines aside from our built-in MCMC engine to be used with templates. We saw in section 5 details of how the system can interact with third-party software. In this section (and in fact the following three sections) we will see how through the inclusion of additional C++ code the user can increase the set of models and methods that can be fitted using the built-in eStat engine. At present the methods we describe are partly to advance the modelling but also partly to cover current limitations in the algebra system which may eventually be rectified. As the names suggest the *preccode* function will involve writing C++ code and so some knowledge of the C/C++ languages would be useful. The examples given here will however allow the user with some modification to use similar chunks of code for their examples. We begin in this chapter with a simple example of a 1 level probit regression model.

10.1 The 1LevelProbitRegression template

We have seen already that the *1LevelMod* template can be used to fit binary response models and we have demonstrated a logistic regression model for the *bang* dataset. A probit regression is similar to a logistic regression but uses a different link function. One interesting feature of a probit regression is that the link function is the inverse normal distribution cdf. This means that we can interpret the model using latent variables in an interesting way.

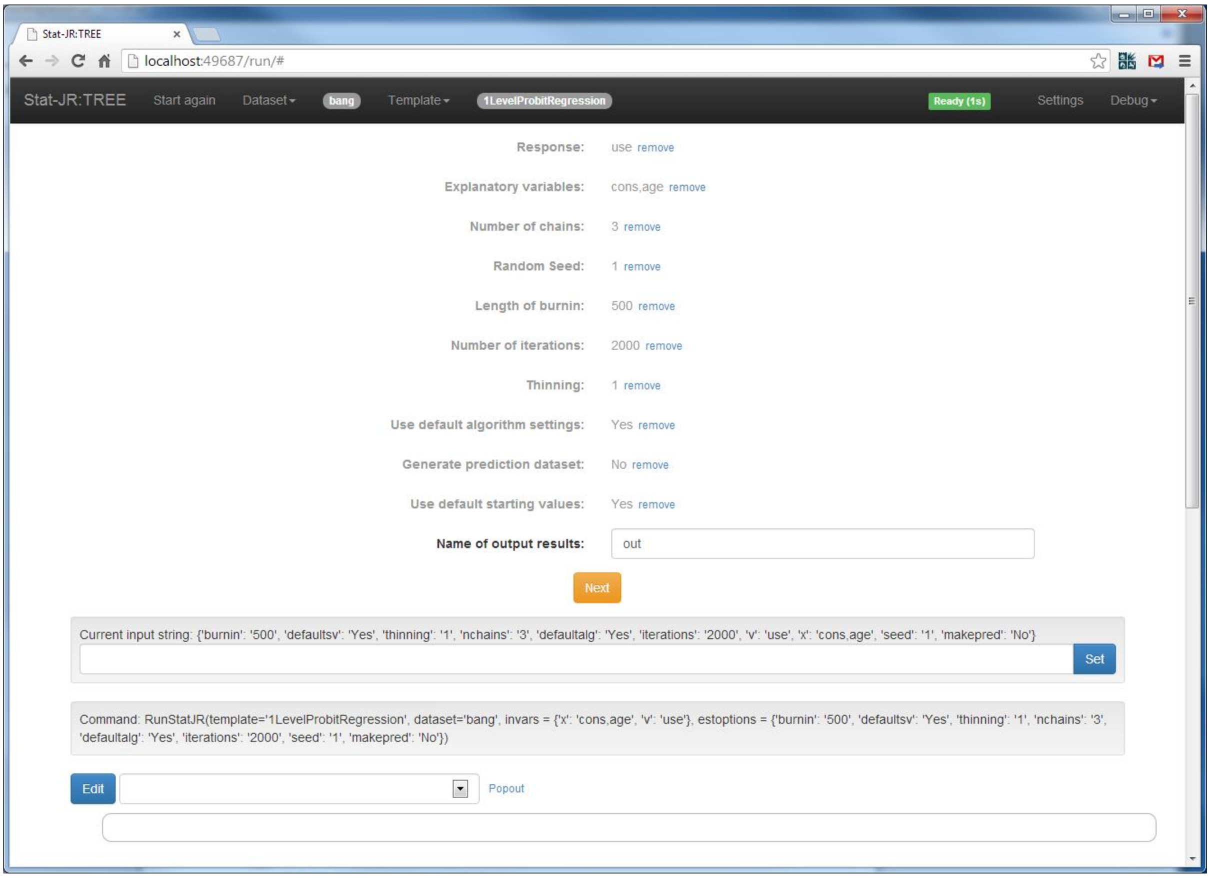
Imagine that you had a variable which was a continuous measurement but that we can only observe a binary indicator as to whether the variable was above or below a threshold, for example in education we might have a mark in an exam but the student is only told whether they pass or fail. If we model the pass/fail indicators using a probit regression then this is equivalent to assuming the unobserved (latent) continuous measure follows a normal distribution (with threshold 0 and variance 1).

We can use this fact in our modelling when we use MCMC by generating the latent continuous variables as part of the algorithm. Then having generated the latent variables we have a normal response model for these variables which is easy to fit. The 1LevelProbitRegression template therefore fits a probit regression using this technique and we will add the step to update the continuous response variables via the *preccode* methods.

We will start as usual by looking at the *inputs* attribute which is quite short:

```
   inputs = '''
v = DataVector('Response: ')
y = ParamVector(parents=[v], as_scalar=False, customstep=True, 
monitor=False)
x = DataMatrix('Explanatory variables: ', allow_cat = True)
beta = ParamVector(parents=[x], as_scalar=True)
deviance = ParamScalar(customstep=True)
'''
```

Here you will see that the column containing the 0/1 response is actually stored as v in this template as we will use y to be the underlying continuous response. As *y* is latent it is defined as a *Paramvector* rather than data, and the parents term links the lengths of the two vectors together which basically ensures that the continuous response vector *y* is the same length as the observed binary response vector *v*. The argument *customstep=True* tells Stat-JR that this parameter will have it's own C code step and the *monitor=False* argument tells Stat-JR not to store a chain for each element of y. As always it helps to demonstrate the template with an example so we will fit a probit regression model (equivalent to the logistic regression in section 7) to the Bangladeshi dataset. Select *1LevelProbitRegression* from the *template* list and *bang* from the *dataset* list and then fill in the template as shown below:



Response: use

Explanatory variables: cons,age

Number of chains: 3

Random Seed: 1

Length of burnin: 500

Number of iterations: 2000

Thinning: 1

Use default algorithm settings: Yes

Generate prediction dataset: No

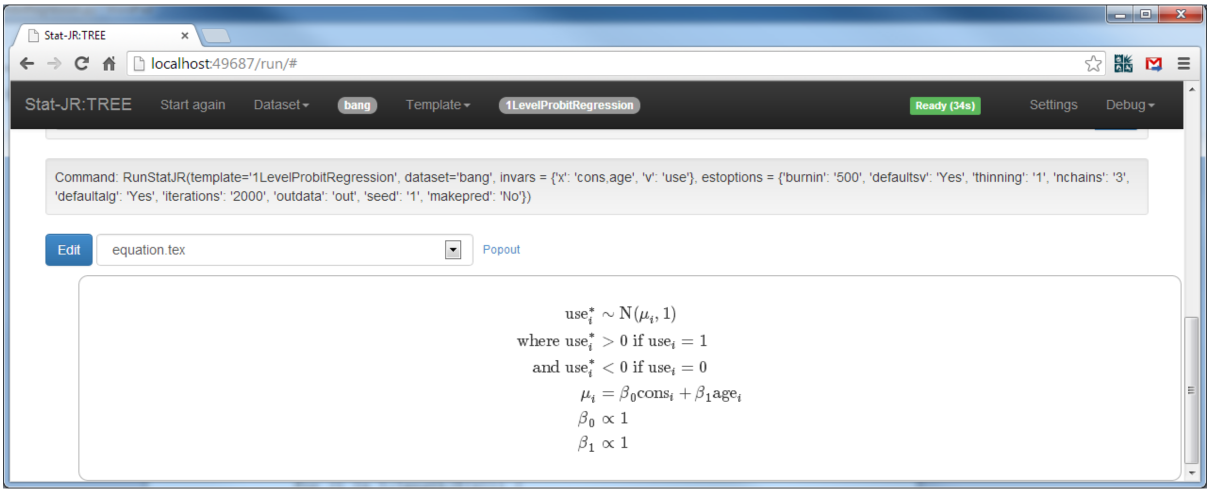
Use default starting values: Yes

Name of output results: out

```
Current input string: {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'v': 'use', 'x': 'cons,age', 'seed': '1', 'makepred': 'No'}
```

```
Command: RunStatJR(template='1LevelProbitRegression', dataset='bang', invars = {'x': 'cons,age', 'v': 'use'}, estoptions = {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'seed': '1', 'makepred': 'No'})
```

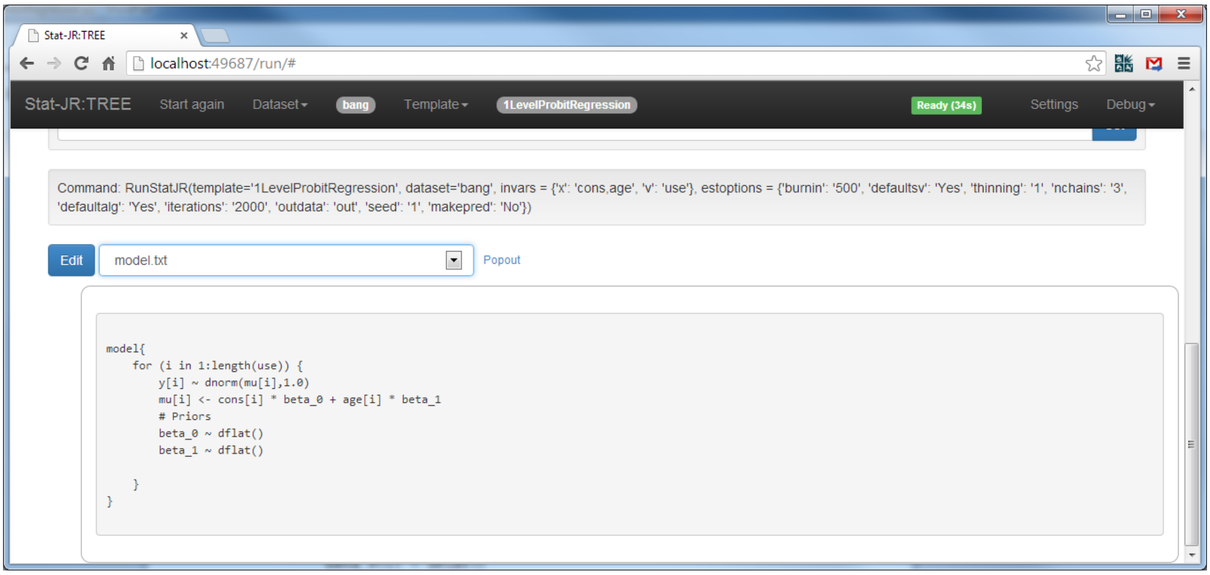
Clicking on the **Next** button, *equation.tex* will appear in the output pane and we see the model described mathematically:



Command: RunStatJR(template='1LevelProbitRegression', dataset='bang', invars = {'x': 'cons,age', 'v': 'use'}, estoptions = {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'outdata': 'out', 'seed': '1', 'makepred': 'No'})

$$\begin{aligned} \mathrm{use}^*_i &\sim \mathrm{N}(\mu_i, 1) \\ \text{where } \mathrm{use}^*_i &> 0 \text{ if } \mathrm{use}_i = 1 \\ \text{and } \mathrm{use}^*_i &< 0 \text{ if } \mathrm{use}_i = 0 \\ \mu_i &= \beta_0 \mathrm{cons}_i + \beta_1 \mathrm{age}_i \\ \beta_0 &\propto 1 \\ \beta_1 &\propto 1 \end{aligned}$$

Here you see how use* is the latent continuous variable written as *y* in the model code. If we look at model.txt (below) we see that the model code is really just fitting a normal model as if we already know the values of *y.*



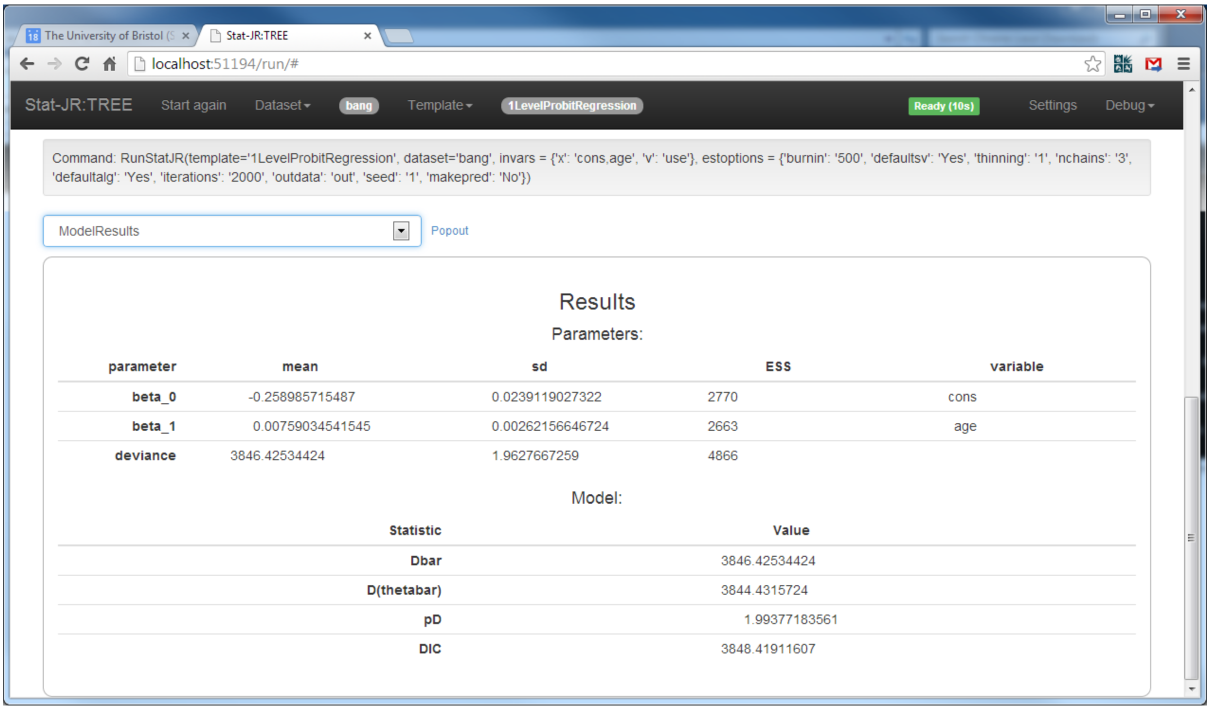
Command: RunStatJR(template='1LevelProbitRegression', dataset='bang', invars = {'x': 'cons,age', 'v': 'use'}, estoptions = {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'outdata': 'out', 'seed': '1', 'makepred': 'No'})

```
model{
    for (i in 1:length(use)) {
        y[i] ~ dnorm(mu[i],1.0)
        mu[i] <- cons[i] * beta_0 + age[i] * beta_1
        # Priors
        beta_0 ~ dflat()
        beta_1 ~ dflat()

    }
}
```

If we look at the *model* attribute we can see that clearly.

```
model = '''
model{
    for (i in 1:length(${v})) {
        y[i] ~ dnorm(mu[i],1.0)
        mu[i] <- ${mmult(x, 'beta', 'i')}
        # Priors
        % for i in range(0, x.ncols()):
        beta_${i} ~ dflat()
        % endfor

    }
}
'''
```

Here the code is fairly straightforward so the interesting thing is how we actually include a step for *y* to make this the correct model. You will recall that y will need a custom step and will not be monitored. To see this in practice we will continue our example and press the **Run** button. When finished if you select **ModelResults** then the results will look as follows:



Command: RunStatJR(template='1LevelProbitRegression', dataset='bang', invars = {'x': 'cons,age', 'v': 'use'}, estoptions = {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'outdata': 'out', 'seed': '1', 'makepred': 'No'})

Results

Parameters:

parameter	mean	sd	ESS	variable
beta_0	-0.258985715487	0.0239119027322	2770	cons
beta_1	0.00759034541545	0.00262156646724	2663	age
deviance	3846.42534424	1.9627667259	4866	

Model:

Statistic	Value
Dbar	3846.42534424
D(thetabar)	3844.4315724
pD	1.99377183561
DIC	3848.41911607

We see that the model has run and got reasonable ESS values and has returned a DIC value. To see how this happened we need to look at the **preccode** and **deviancecode** attributes

10.3 preccode and deviancecode attributes

So we have seen that the code works but we need now to look and see how the step for updating the latent y variable is incorporated into the code. This is done via the *preccode* attribute which for this template looks as follows:

```
    preccode = '''
<%!
    def mmult(names, var, index):
        out = ""
        count = 0
        for name in names:
            if count > 0:
                out += ' + '
            out += 'double(' + name + '[' + index + ']) * ' + var + '_' +
str(count)
            count += 1
        return out
%>
        double mean;
        for(int i=0;i<length(y);i++)
        {
           mean = ${mmult(x, 'beta', 'i')};
```

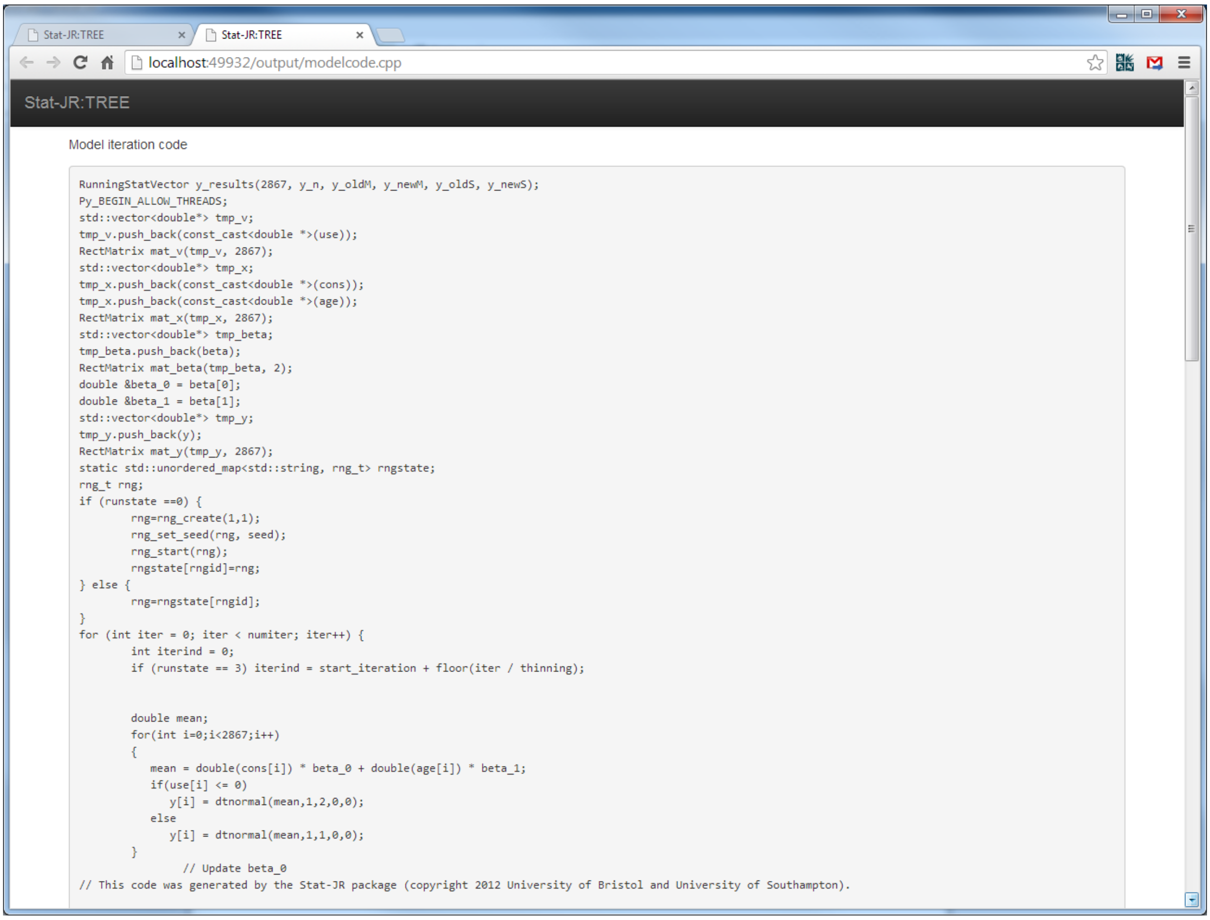
```
      if(${v}[i] <= 0)
         y[i] = dtnormal(mean,1,2,0,0);
      else
         y[i] = dtnormal(mean,1,1,0,0);
   }
   '''
```

This function could have been even shorter except we need to include in here a definition for the mmult function that is used to construct the linear predictor, this time as C++ code.
The actual C++ code is the chunk

```
double mean;
for(int i=0;i<length(y);i++)
{
   mean = ${mmult(x, 'beta', 'i')};
   if(${v}[i] <= 0)
      y[i] = dtnormal(mean,1,2,0,0);
   else
      y[i] = dtnormal(mean,1,1,0,0);
}
```

Here we see that the code involves looping over all data points via a *for* loop and for each point evaluating the mean value which is the linear predictor calculated via the substitution. Then depending on the value of the binary response a call is made to the truncated normal random number generator via the dtnormal function. Here dtnormal takes 5 arguments, the mean, the sd, the type of truncation with 1 left truncation, 2 right truncation and 3 both, and finally the left and right truncation values.

To see this in action choose **modelcode.cpp** from the list and choose to pop it out:



Model iteration code

```
RunningStatVector y_results(2867, y_n, y_oldM, y_newM, y_oldS, y_newS);
Py_BEGIN_ALLOW_THREADS;
std::vector<double*> tmp_v;
tmp_v.push_back(const_cast<double *>(use));
RectMatrix mat_v(tmp_v, 2867);
std::vector<double*> tmp_x;
tmp_x.push_back(const_cast<double *>(cons));
tmp_x.push_back(const_cast<double *>(age));
RectMatrix mat_x(tmp_x, 2867);
std::vector<double*> tmp_beta;
tmp_beta.push_back(beta);
RectMatrix mat_beta(tmp_beta, 2);
double &beta_0 = beta[0];
double &beta_1 = beta[1];
std::vector<double*> tmp_y;
tmp_y.push_back(y);
RectMatrix mat_y(tmp_y, 2867);
static std::unordered_map<std::string, rng_t> rngstate;
rng_t rng;
if (runstate ==0) {
        rng=rng_create(1,1);
        rng_set_seed(rng, seed);
        rng_start(rng);
        rngstate[rngid]=rng;
} else {
        rng=rngstate[rngid];
}
for (int iter = 0; iter < numiter; iter++) {
        int iterind = 0;
        if (runstate == 3) iterind = start_iteration + floor(iter / thinning);


        double mean;
        for(int i=0;i<2867;i++)
        {
           mean = double(cons[i]) * beta_0 + double(age[i]) * beta_1;
           if(use[i] <= 0)
              y[i] = dtnormal(mean,1,2,0,0);
           else
              y[i] = dtnormal(mean,1,1,0,0);
        }
                // Update beta_0
// This code was generated by the Stat-JR package (copyright 2012 University of Bristol and University of Southampton).
```

Here we see that after some initial setup lines, the **preccode** chunk, as the name suggests appears before the steps for other parameters (in this case beta_0). This is important as the y variable needs initialising before the other parameters are updated and updating it first ensures y is positive when *use* is 1 and negative when *use* is 0.

So we have added in a step via preccode to enable the MCMC algorithm to work correctly. Of course the model that the algebra system has been sent is the simpler normal model and so the deviance for this model would be returned by it and thus as this is used to construct the DIC diagnostic then we would get a model fit diagnostic for the wrong model. To rectify this the attribute *deviancecode* can be used to overwrite the definition of the deviance. This attribute contains a piece of C++ code for the deviance step. This code is then included in both the iteration loop and the DIC code and thus the DIC diagnostic is calculated correctly. The code can be seen within the template but will not be repeated here.

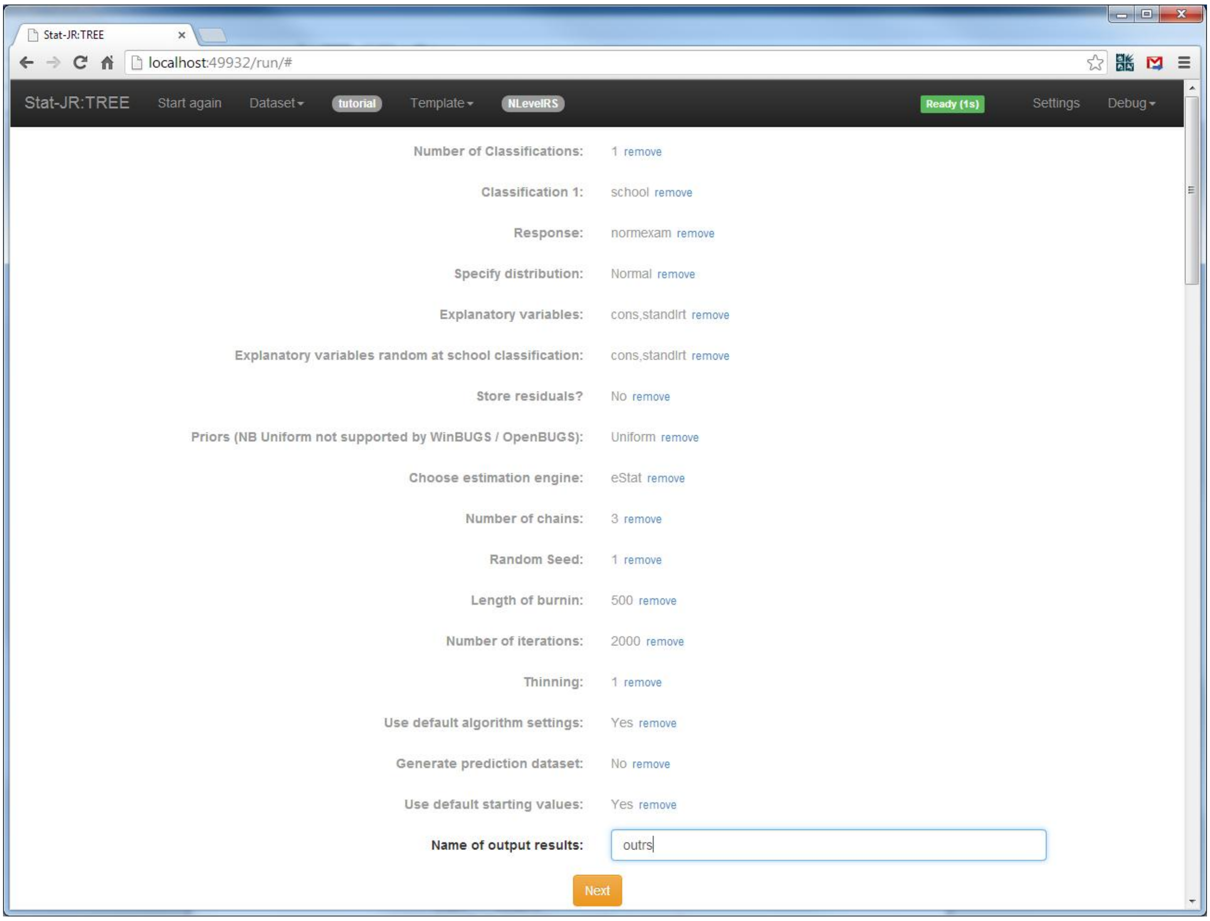
11 Multilevel models with Random slopes and the inclusion of Wishart priors

One limitation of the algebra system in its current form is that it treats all parameters as scalars. This means for example that for the *Regression1* template, the set of beta parameters are all updated individually through univariate normal steps. We will investigate the implications of this in section 12. In section 9 we introduced our first multilevel models all of which only had random intercepts. To extend such models to include random slopes requires (assuming slopes and intercepts are correlated) the use of a multivariate normal distribution for the random effects.

Multivariate normal distributions by their nature have vector and not scalar parameters and so our model code diverges from standard WinBUGS model code here (and hence this is an example template where template specific methods are required for WinBUGS). Our improvised model code depends on the number of response variables i.e. we have bivariate, trivariate etc normal distributions. We will see how these work in practice via the template *NLevelRS*. It should be noted that we also have templates that completely circumvent the algebra system and simply write custom C code. These templates have the postfix 'cc' at their end for example *1LevelModcc.*

11.1 An example with random slopes

Firstly select *NLevelRS* from the *template* list and *tutorial* from the *data* list. Then choose the inputs as follows:



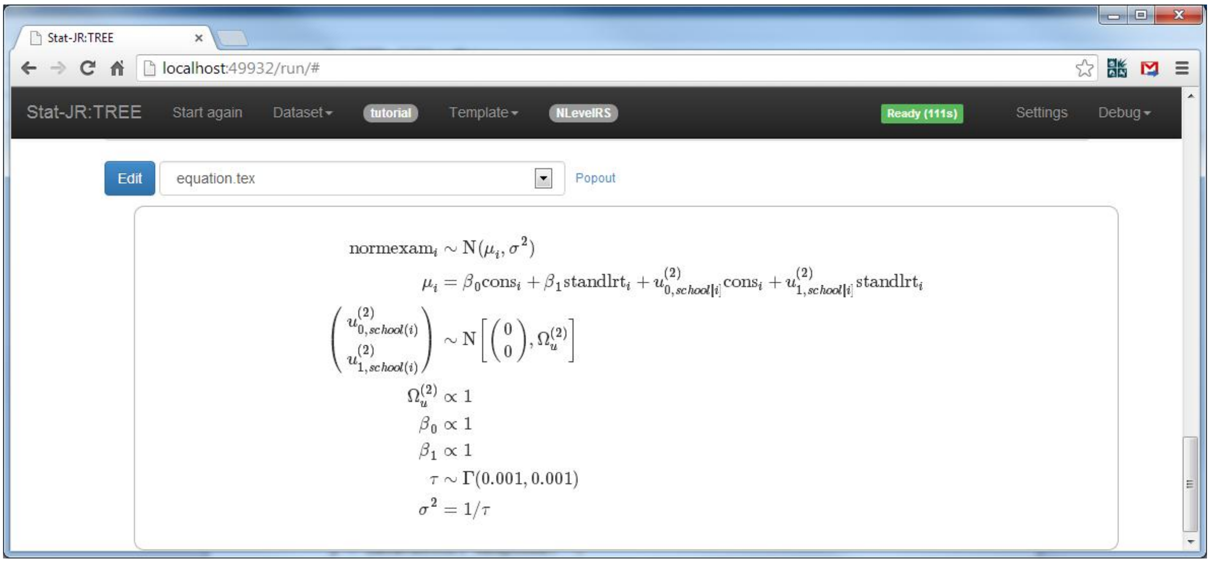
Number of Classifications: 1 remove
Classification 1: school remove
Response: normexam remove
Specify distribution: Normal remove
Explanatory variables: cons,standlrt remove
Explanatory variables random at school classification: cons,standlrt remove
Store residuals? No remove
Priors (NB Uniform not supported by WinBUGS / OpenBUGS): Uniform remove
Choose estimation engine: eStat remove
Number of chains: 3 remove
Random Seed: 1 remove
Length of burnin: 500 remove
Number of iterations: 2000 remove
Thinning: 1 remove
Use default algorithm settings: Yes remove
Generate prediction dataset: No remove
Use default starting values: Yes remove
Name of output results: outrs

You will see that there are lots of inputs here and correspondingly the *inputs* function for this template is therefore quite long as we see below:

```
inputs = '''
NumLevs = Integer('Number of Classifications: ')
for i in range(0, int(NumLevs)):
    selstr = 'Classification ' + str(i + 1) + ': '
    context['C' + str(i + 1)] = IDVector(selstr)
y = DataVector('Response: ')
D = Text('Specify distribution: ', ['Normal', 'Binomial', 'Poisson'])
if D == 'Binomial':
    n = DataVector('Denominator: ')
    link = Text('Specify link function: ', ['logit', 'probit', 'cloglog'])
if D == 'Poisson':
    link = Text(value = 'ln')
    offset = Boolean('Is there an offset: ')
    if offset:
        n = DataVector('Offset: ')
if D == 'Normal':
    tau = ParamScalar()
    sigma = ParamScalar(modelled = False)
x = DataMatrix('Explanatory variables: ', allow_cat = True)

for i in range(0, int(NumLevs)):
    context['x'+str(i+1)] = DataMatrix('Explanatory variables random at ' +
context['C' + str(i + 1)] + ' classification: ', allow_cat = True)


storeresid = Boolean('Store residuals?')

beta = ParamVector(parents=[x], as_scalar=True)

for i in range(0, int(NumLevs)):
    for var in range(0, len(context['x'+str(i+1)])):
        if storeresid:
            context['u' + str(var) + '_' + str(i)] =
ParamVector(parents=[context['C' + str(i + 1)]], as_scalar=False)
        else:
            context['u' + str(var) + '_' + str(i)] =
ParamVector(parents=[context['C' + str(i + 1)]], as_scalar=False,
monitor=False)
    num = len(context['x'+str(i+1)])
    if num == 1:
        context['tau_u0_'+str(i+1)] = ParamScalar()
        context['sigma_u0_'+str(i+1)] = ParamScalar(modelled = False)
    else:
        context['omega_u'+str(i+1)] = ParamMatrix(modelled = False,
customstep=True)
        context['omega_u'+str(i+1)].size = num
        context['d_u'+str(i+1)] = ParamMatrix(customstep=True)
        context['d_u'+str(i+1)].size = num
        context['priors' + str(i)] = Text('Priors (NB Uniform not supported
by WinBUGS / OpenBUGS): ', ['Uniform', 'Wishart'])
        if context['priors' + str(i)] == 'Wishart':
            context['R' + str(i)] = List('R matrix: ')
            context['v' + str(i)] = Integer('Degrees of Freedom:')

deviance = ParamScalar(modelled = False)
'''
```

The template is initially like the *NLevelMod* template but then has an additional section that is used to input the variables that have random effects associated with them (at each level), and then any priors at those levels are input. You will see that we use the *context* functionality to construct variable names a lot and that there are different parameters for classifications where there is a single random parameter and where there are more than one. In brief parameters beginning *tau_u0* and *sigma_u0* are the precision and variance of the random effects if there is a single set of random effects; those beginning *omega_u* and *d_u* are the variance matrix and precision matrix if we have multiple sets of random effects at a classification. Finally in this case there are two possible priors and for the (informative) Wishart priors an estimate (beginning with R) and degrees of freedom (beginning with v) parameter are required.

Having completed our inputs we now need to click on **Next** to see what the model looks like (as shown by *equation.tex*):



$$\mathrm{normexam}_i \sim \mathrm{N}(\mu_i, \sigma^2)$$

$$\mu_i = \beta_0 \mathrm{cons}_i + \beta_1 \mathrm{standlrt}_i + u^{(2)}_{0,school[i]} \mathrm{cons}_i + u^{(2)}_{1,school[i]} \mathrm{standlrt}_i$$

$$\begin{pmatrix} u^{(2)}_{0,school(i)} \\ u^{(2)}_{1,school(i)} \end{pmatrix} \sim \mathrm{N}\left[\begin{pmatrix} 0 \\ 0 \end{pmatrix}, \Omega^{(2)}_u\right]$$

$$\Omega^{(2)}_u \propto 1$$

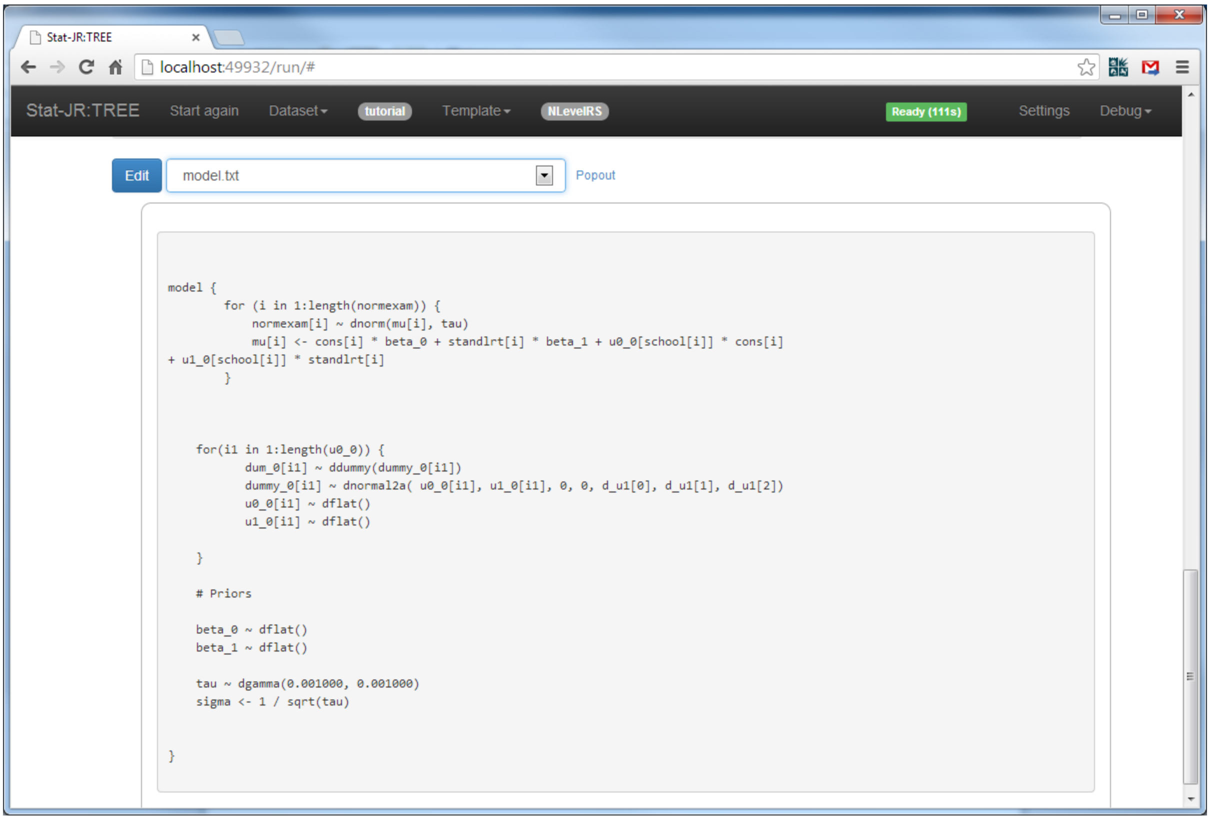
$$\beta_0 \propto 1$$

$$\beta_1 \propto 1$$

$$\tau \sim \Gamma(0.001, 0.001)$$

$$\sigma^2 = 1/\tau$$

Here we see the LateX code including the multivariate normal distribution for the random intercepts and slopes. To see the model specification we choose *model.txt* from the list:



```
model {
        for (i in 1:length(normexam)) {
            normexam[i] ~ dnorm(mu[i], tau)
            mu[i] <- cons[i] * beta_0 + standlrt[i] * beta_1 + u0_0[school[i]] * cons[i]
+ u1_0[school[i]] * standlrt[i]
        }



    for(i1 in 1:length(u0_0)) {
           dum_0[i1] ~ ddummy(dummy_0[i1])
           dummy_0[i1] ~ dnormal2a( u0_0[i1], u1_0[i1], 0, 0, d_u1[0], d_u1[1], d_u1[2])
           u0_0[i1] ~ dflat()
           u1_0[i1] ~ dflat()

    }

    # Priors

    beta_0 ~ dflat()
    beta_1 ~ dflat()

    tau ~ dgamma(0.001000, 0.001000)
    sigma <- 1 / sqrt(tau)


}
```

The multivariate normal distribution is written in the model code as follows:

```
for(i1 in 1:length(u0_0)) {
        dum_0[i1] ~ ddummy(dummy_0[i1])
        dummy_0[i1] ~ dnormal2a( u0_0[i1], u1_0[i1], 0, 0, d_u1[0],
d_u1[1], d_u1[2])
           u0_0[i1] ~ dflat()
           u1_0[i1] ~ dflat()
    }
```

Basically the *dnormal2a* distribution has as its first two arguments the two responses. Next we get the 2 means and then the 3 parameters that make up the precision matrix. As the algebra system expects all parameters to appear on the left-hand side we complete our workaround for a multivariate Normal distribution by including the two dflat statements which do not change the posterior but mean that the *u0_0[i1]* and *u1_0[i1]* are regarded in the algebra system as parameters. Note that the *dummy_0* parameters are simply placeholders as each distribution needs a scalar left-hand side. The definition of dnormal2a does not depend on the left hand side term. The *dummy_0* parameters also appears on the right hand side in the *ddummy* statement and this is to trick the algebra system into thinking that *dummy_0* is truly a parameter so that the dnormal2a statement is not considered part of the likelihood for calculating DIC etc.

The code for creating the model code is in *model* but doesn't contain anything very new that needs reporting here. The *latex* code might interest those trying to learn LateX as it contains a chunk to produce the multivariate Normal line as follows:

```
\left(
\begin{array}{l}
% for i in range(0, len(context['x'+str(lev+1)])):
u^{(${lev + 2})}_{${i},${context['C' + str(lev + 1)]}(i)}
% if i != len(context['x'+str(lev+1)]) -1:
\\
% endif
% endfor
\end{array}
\right) & \sim \mbox{N}
\left[ \left(
\begin{array}{l}
% for i in range(0, len(context['x'+str(lev+1)])):
0
% if i != len(context['x'+str(lev+1)]) -1:
\\
% endif
% endfor
\end{array}
\right), \Omega^{(${lev + 2})}_{u} \right] \\
```

Here we use Python %ifs and %fors to allow conditional code and the array environment and \left and \right (for big brackets) in LaTeX to deal with vectors and matrices. The actual code that is produced can be looked at by right clicking on the LateX and selecting show source and selecting the appropriate lines. It looks as follows:

```
\left(
\begin{array}{l}
u^{(2)}_{0,school(i)}
\\
u^{(2)}_{1,school(i)}
\end{array}
\right) & \sim \mbox{N}
\left[ \left(
\begin{array}{l}
0
\\
0
\end{array}
\right), \Omega^{(2)}_{u} \right] \\
```

Looking at the model code we have not included a prior for *d_u1* and so here we again resort to writing our own *preccode* chunk.

11.2 Preccode for NLevelRS

We will here look at the *preccode* in chunks. The *preccode* is being used to add a step for updating the precision matrix *d_u1* and the corresponding variance matrix *omega_u1*. Looking at the start of the code overleaf:

```
    preccode = '''
    {
<% numlevs = int(NumLevs) %>\\
        bool fail = false;
% for i in range(0, numlevs):
<% n = len(context['x'+str(i + 1)]) %>\\
% if n > 1 :

        std::vector<double*> tmp_u${i};
% for j in range(0, n):
        tmp_u${i}.push_back(u${j}_${i});
% endfor
        RectMatrix mat_u${i}(tmp_u${i}, length(u0_${i}));

        SymMatrix sb${i + 1} = mat_u${i}.T() * mat_u${i};
```

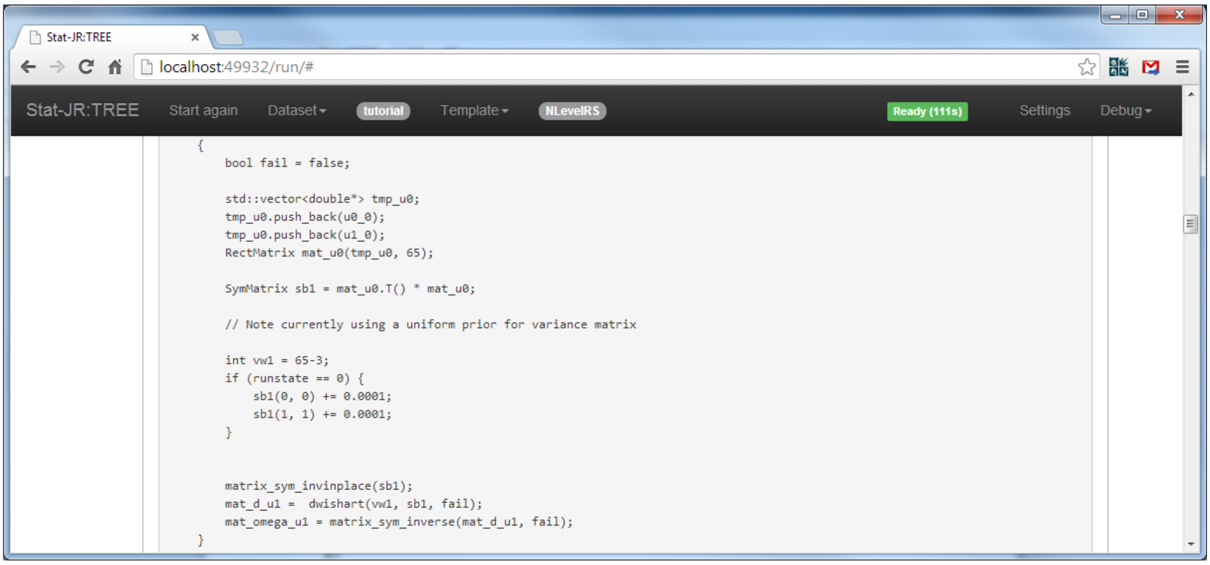
This first section stores the number of levels (*numlevs*) for looping purposes and also within the loop the number of random effects are constructed (as *n*) because for classifications with only 1 set of random effects nothing needs doing as the algebra system has evaluated the posterior required. We take the u's and store them in a matrix so that we can do matrix arithmetic. We next construct a matrix variable *sb1* which initially stores the crossproduct matrix of the residuals before moving to the next chunk of code:

```
% if context['priors' + str(i)] == 'Uniform':
        int vw${i+1} = length(u0_${i})-${n + 1};
        if (runstate == 0) {
% for j in range(0, n):
            sb${i + 1}(${j}, ${j}) += 0.0001;
% endfor
        }
% endif
% if context['priors' + str(i)] == 'Wishart':
        int vw${i+1} = length(u0_${i}) + ${context['v' + str(i)]};
% endif

% if context['priors' + str(i)] == 'Wishart':
<%
import numpy
Rmat = numpy.empty([n, n])
count = 0
for j in range(0, n):
    for k in range(0, j + 1):
        Rmat[j, k] = float(context['R' + str(i)].name[count])
        Rmat[k, j] = Rmat[j, k]
        count += 1
%>

<% count = 0 %>
% for j in range(0, n):
% for k in range(j, n):
        sb${i+1}(${j}, ${k}) += ${str(Rmat[j, k] * float(context['v' + 
str(i)]))};
% endfor
% endfor
% endif
```

In this chunk of code we have different blocks of code depending on prior distribution types. For the uniform prior we simply construct the degrees of freedom parameter (*vw1*), which equals the number of higher level units minus the number of sets of random effects + 1. We also have some code for the first iteration (*runstate = 0*) to avoid numerical problems as the residual starting values may all be the same. For the Wishart prior we have to add the prior parameters to the *sb1* and *vw1* parameters. Next we have:

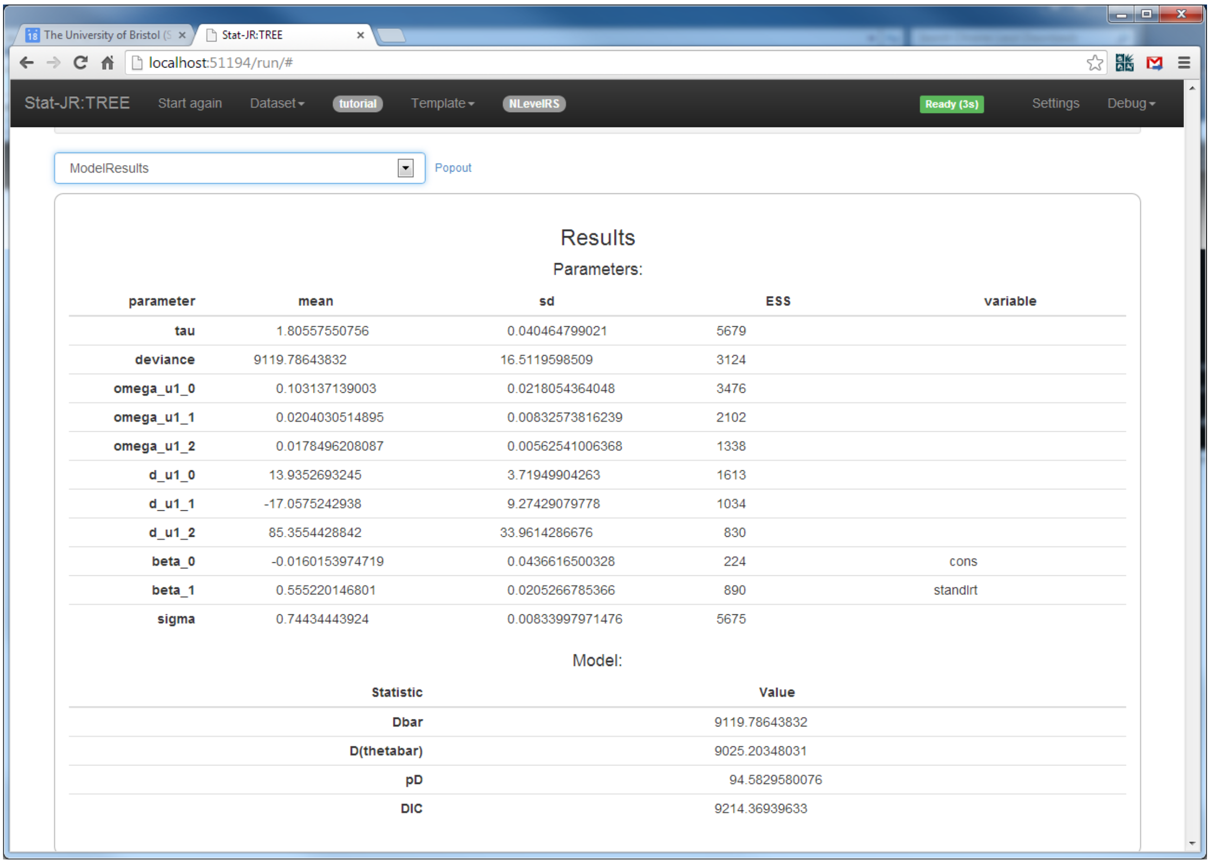
```
        matrix_sym_invinplace(sb${i+1});
        mat_d_u${i + 1} =  dwishart(vw${i+1}, sb${i+1}, fail);
        mat_omega_u${i + 1} = matrix_sym_inverse(mat_d_u${i + 1}, fail);
%endif
%endfor
    }
'''
```

In this last chunk of code we invert the *sb1* parameter before drawing the new precision matrix which we store in *mat_d_u1* and the inverse matrix to the vector *mat_omega_u1*. To see the code that the *preccode* method generates for our example we can select **modelcode.cpp** and scroll down a few lines as shown below:



```
    {
        bool fail = false;

        std::vector<double*> tmp_u0;
        tmp_u0.push_back(u0_0);
        tmp_u0.push_back(u1_0);
        RectMatrix mat_u0(tmp_u0, 65);

        SymMatrix sb1 = mat_u0.T() * mat_u0;

        // Note currently using a uniform prior for variance matrix

        int vw1 = 65-3;
        if (runstate == 0) {
            sb1(0, 0) += 0.0001;
            sb1(1, 1) += 0.0001;
        }


        matrix_sym_invinplace(sb1);
        mat_d_u1 =  dwishart(vw1, sb1, fail);
        mat_omega_u1 = matrix_sym_inverse(mat_d_u1, fail);
    }
```

As the name preccode suggests the code appears before the other steps in the algorithm.
We can finally run the template by clicking on the **Run** button and selecting **ModelResults** from the list. The results appear as follows:



Results

Parameters:

parameter	mean	sd	ESS	variable
tau	1.80557550756	0.040464799021	5679	
deviance	9119.78643832	16.5119598509	3124	
omega_u1_0	0.103137139003	0.0218054364048	3476	
omega_u1_1	0.0204030514895	0.00832573816239	2102	
omega_u1_2	0.0178496208087	0.00562541006368	1338	
d_u1_0	13.9352693245	3.71949904263	1613	
d_u1_1	-17.0575242938	9.27429079778	1034	
d_u1_2	85.3554428842	33.9614286676	830	
beta_0	-0.0160153974719	0.0436616500328	224	cons
beta_1	0.555220146801	0.0205266785366	890	standlrt
sigma	0.74434443924	0.00833997971476	5675	

Model:

Statistic	Value
Dbar	9119.78643832
D(thetabar)	9025.20348031
pD	94.5829580076
DIC	9214.36939633

and as usual we also get MCMC output graphs for the fixed effect parameters, variances and precisions via the pulldown list.

<u>Exercise 9</u>

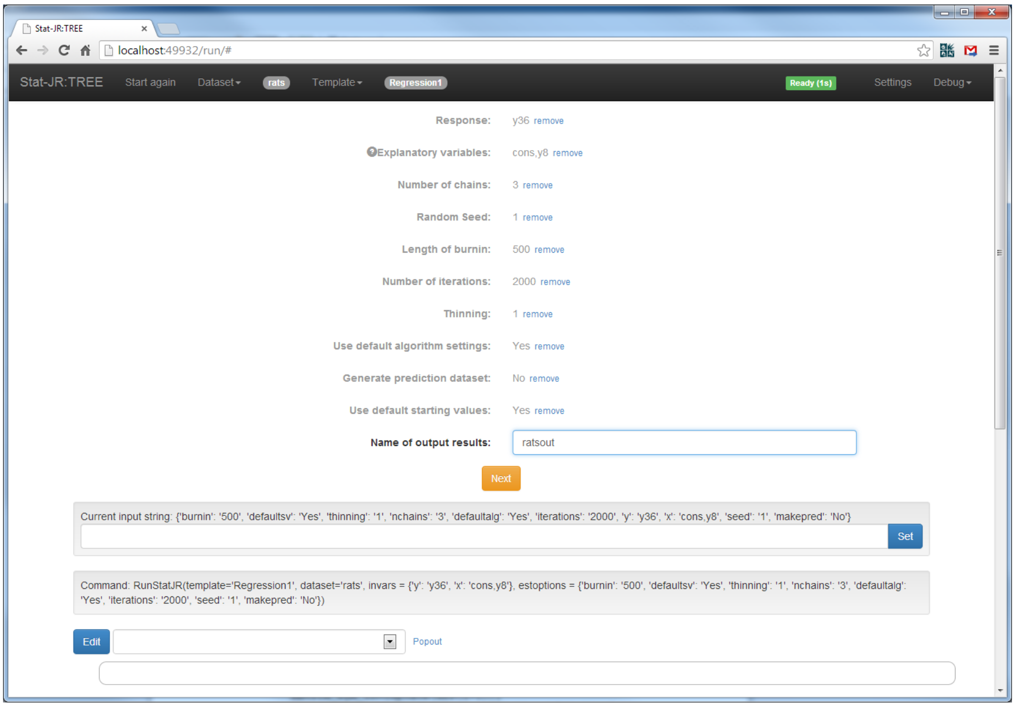
Try adapting the *NLevelRS* template so that it only allows one higher classification and compare your results with the *2LevelRS* template. This exercise will be in essence a merging of features of two templates, *2LevelMod* and *NLevelRS* and will test your understanding of the various chunks of code.

12 Improving mixing (1LevelBlock and 1LevelOrthogParam)

In this section we will return once again to our first template *Regression1* but use it on a different dataset, *rats*. This dataset consists of the weights of 30 laboratory rats at weekly intervals from 8 days old and here we will consider a regression looking at the impact on their final weight at 36 days of their initial weight at 8 days old.

12.1 Rats example

We will set up a simple regression for this rather small dataset as follows:



Response: y36 remove

Explanatory variables: cons,y8 remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

Use default algorithm settings: Yes remove

Generate prediction dataset: No remove

Use default starting values: Yes remove

Name of output results: ratsout

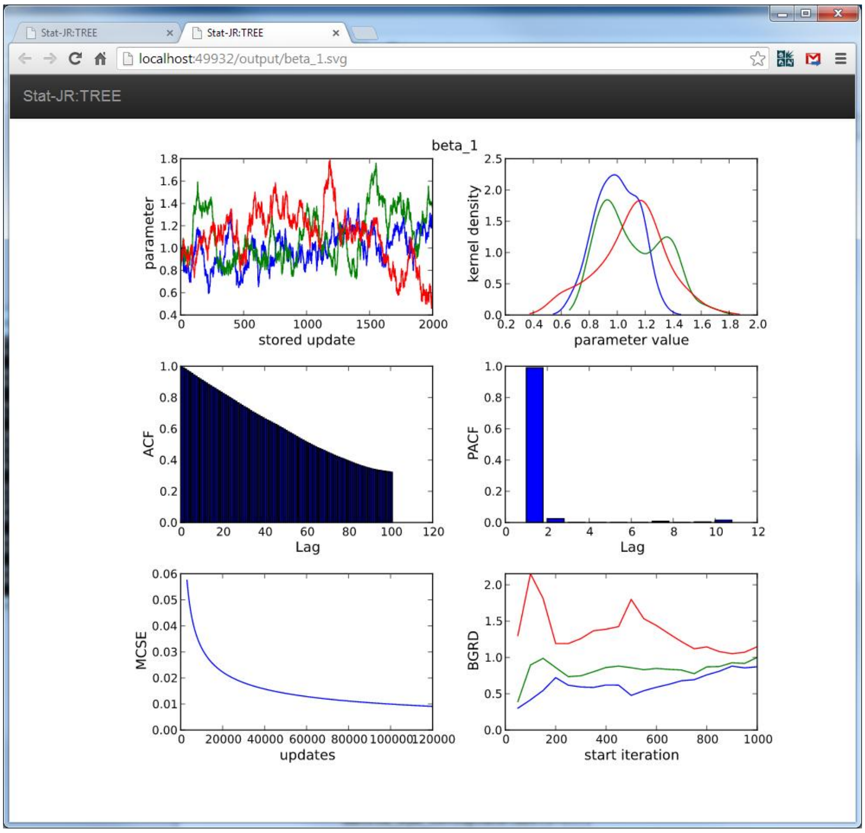
Next

```
Current input string: {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'y36', 'x': 'cons,y8', 'seed': '1', 'makepred': 'No'}
```

Set

```
Command: RunStatJR(template='Regression1', dataset='rats', invars = {'y': 'y36', 'x': 'cons,y8'}, estoptions = {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'seed': '1', 'makepred': 'No'})
```

We will then run the model by clicking the **Next** and **Run** buttons. If we look at the output and change the pull down list to *beta_1.svg* and pop it out so that we have the MCMC plot for *beta1* visible we will see the following:



In *ModelResults* we see that both the regression coefficients have very small effective sample sizes (32 and 32 respectively) and the chains we observe in the graphs above are not mixing well. Aside from being a small dataset a difference between the *rats* and the *tutorial* dataset is that the data have not been centred. This means that the joint posterior distribution of *beta0* and *beta1* has a very large correlation between the pair of parameters and so if we update them separately we will have problems. We will look at two templates that will rectify this problem.

12.2 The 1LevelBlock template

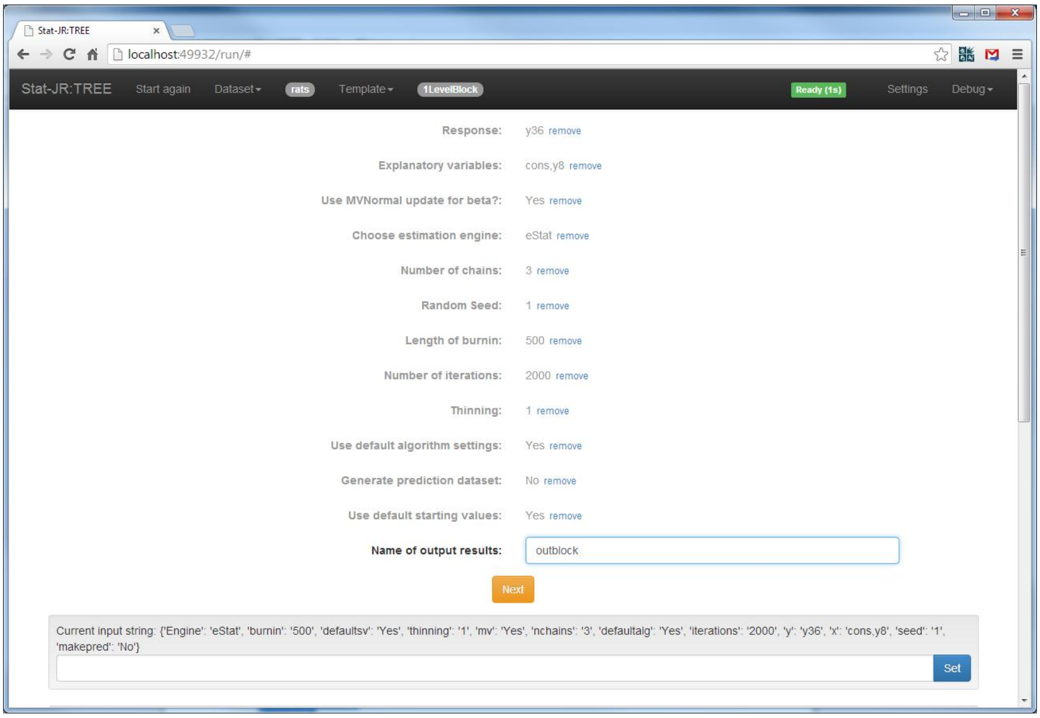
Most MCMC algorithms implemented in software packages will update the parameters *beta0* and *beta1* together in one multivariate normal block. As we have seen, the current algebra system in Stat-JR does not produce multivariate posterior distributions. We can however work out the correct posterior distribution by hand and plug this into the code via the *preccode* options we have seen earlier. This is performed by the *1LevelBlock* template. If you look at the template code you will see it has an initial input in the *inputs* attributes as to whether or not to block the fixed effects and conditional on this we let Stat-JR know whether *beta* is to be updated via a custom step.

```
mv = Boolean('Use MVNormal update for beta?: ')
if mv:
    beta = ParamVector(parents=[x], as_scalar=True, customstep=True)
else:
    beta = ParamVector(parents=[x], as_scalar=True)
```

This code is informing the code generator that customsteps are to be used for the beta parameters when the block updating option (*mv*) is selected and that it should ignore whatever has been returned from the algebra system for these steps. The *preccode* method then contains the code to update the beta vector which has mean (in matrix form) $(X^TX)^{-1}X^Ty$ and variance $(X^TX)^{-1}$ times the residual variance. The code which uses matrix classes is as follows:

```
    preccode = '''
% if mv:
            {
                bool fail = false;
                static RectMatrix xtxchol(${len(x)}, ${len(x)});
                static RectMatrix mean(${len(x)}, 1);

                // Setting up constant terms for beta step
                if (runstate == 0) {
                    xtxchol = matrix_cholesky(mat_x.T() * mat_x, false,
fail);
                    RectMatrix xty = mat_x.T() * mat_y;
                    mean = matrix_cholsolve(xtxchol, xty);
                }

                // Multivariate step for beta
                DiagMatrix taudiag = identity(${len(x)}) / tau;
                SymMatrix variance = matrix_cholsolve(xtxchol, taudiag);
                mat_beta = dmultnormal(mean, variance, fail);
            }
% endif
'''
```

If we want to test this template we can choose it (along with *rats*) from the template list and set up the inputs as follows:



Response: y36 remove

Explanatory variables: cons,y8 remove

Use MVNormal update for beta?: Yes remove

Choose estimation engine: eStat remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

Use default algorithm settings: Yes remove

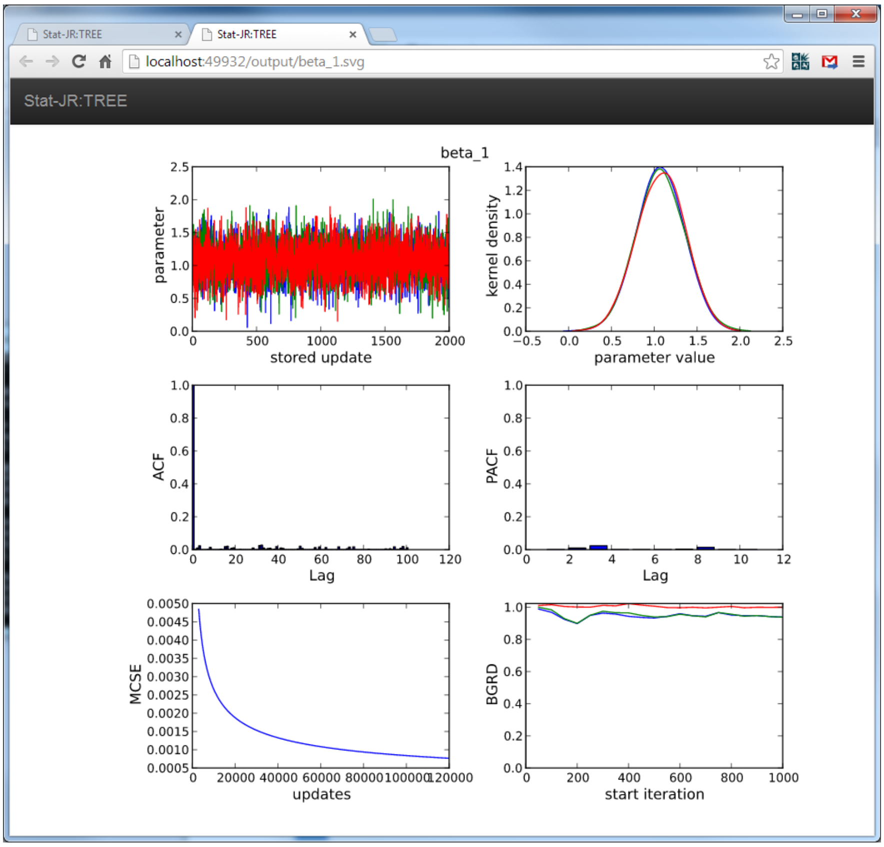
Generate prediction dataset: No remove

Use default starting values: Yes remove

Name of output results: outblock

Current input string: {'Engine': 'eStat', 'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'mv': 'Yes', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'y36', 'x': 'cons,y8', 'seed': '1', 'makepred': 'No'}

Running the template by pressing the **Next** and **Run** buttons results in the following output. Note here we have selected *beta_1.svg* for comparison with the *Regression1* output.



We can see that the method has given much better mixing for *beta1*. Looking at the *ModelResults* the effective sample size values have increased from 32/32 to 5745/5780 for *beta_0* and *beta_1* respectively! We have in other templates (*2LevelBlock* and *NLevelBlock*) implemented similar block updating of fixed effects for multilevel models. We will next look at an alternative method that has the advantages of not needing to use *preccode* and also of being useful for non-normal response models.

12.3 The 1LevelOrthogParam template

The alternative approach to blocking variables that are correlated is to reparameterise the parameters to a configuration that are less correlated. We will achieve this by using an orthogonal parameterisation for the fixed effects rather than the standard parameterisation.

The template we will use is called *1LevelOrthogParam* and the inputs are very similar to the *1LevelMod* template (as this approach also works for non-normal responses). The template does have 2 additional inputs in *inputs* which are used to find out whether or not to use a transformed parameterisation and if so whether to use an orthogonal or orthonormal parameterisation.

This can be seen in the following lines:

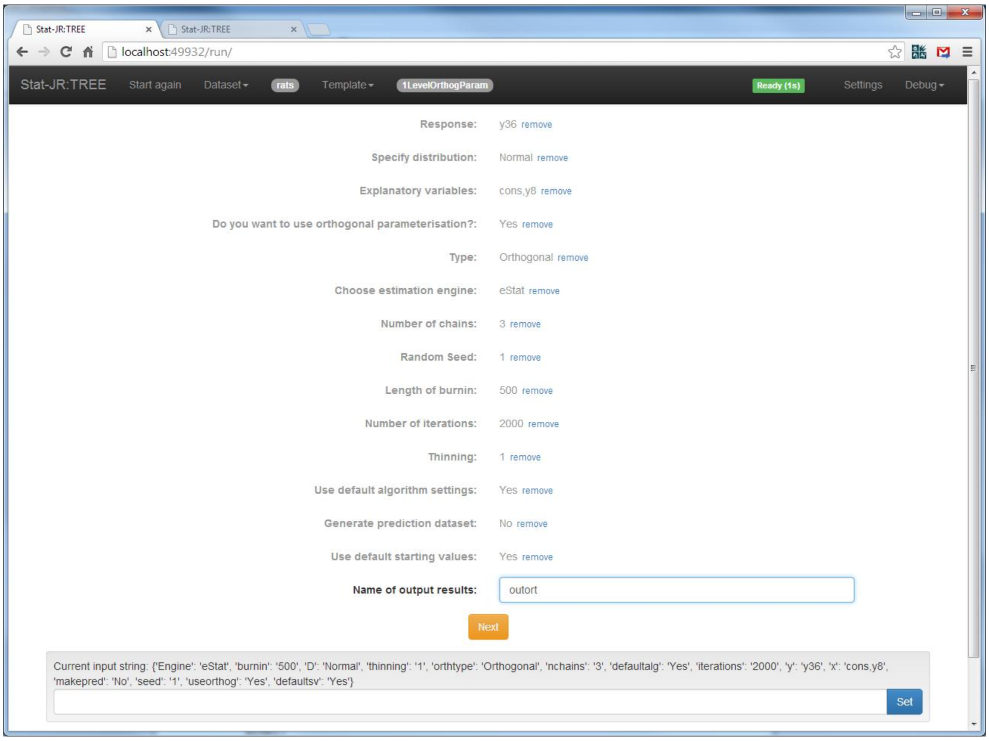
```
useorthog = Boolean('Do you want to use orthogonal parameterisation?: ')
if useorthog:
    orthtype = Text('Type:', ['Orthogonal', 'Orthonormal'])
```

and the following lines are conditional on using the orthogonal parameterisation:

```
if useorthog:
    betaort = ParamVector(parents=[x], as_scalar=True)
    orthogmat = List(value = [])
    #orthogmat = ParamVector()
    #orthogmat.ncols = -(len(x) * len(x))
    beta = ParamVector(parents=[x], as_scalar=True, modelled = False)
else:
    beta = ParamVector(parents=[x], as_scalar=True)
```

Here we add an additional vector of responses, *betaort* if the orthogonal parameterisation is to be used and the standard *beta* vector is now not modelled but becomes deterministically calculated.

Let us try out the template on the *rats* example so choose *1LevelOrthogParam* from the template list and input the following:



Response: y36 remove

Specify distribution: Normal remove

Explanatory variables: cons,y8 remove

Do you want to use orthogonal parameterisation?: Yes remove

Type: Orthogonal remove

Choose estimation engine: eStat remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

Use default algorithm settings: Yes remove

Generate prediction dataset: No remove

Use default starting values: Yes remove

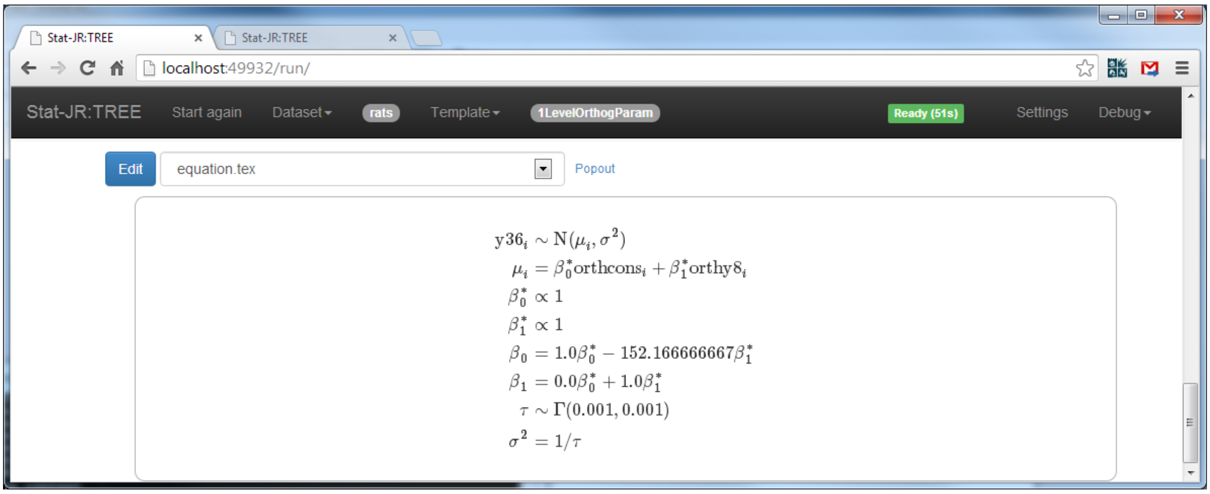
Name of output results: outort

Next

Current input string: {'Engine': 'eStat', 'burnin': '500', 'D': 'Normal', 'thinning': '1', 'orthtype': 'Orthogonal', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'y36', 'x': 'cons,y8', 'makepred': 'No', 'seed': '1', 'useorthog': 'Yes', 'defaultsv': 'Yes'}

Set

Clicking on the **Next** button will give the following output for the model:



$$
\begin{aligned}
\mathrm{y36}_i &\sim \mathrm{N}(\mu_i, \sigma^2)\\
\mu_i &= \beta_0^*\mathrm{orthcons}_i + \beta_1^*\mathrm{orthy8}_i\\
\beta_0^* &\propto 1\\
\beta_1^* &\propto 1\\
\beta_0 &= 1.0\beta_0^* - 152.166666667\beta_1^*\\
\beta_1 &= 0.0\beta_0^* + 1.0\beta_1^*\\
\tau &\sim \Gamma(0.001, 0.001)\\
\sigma^2 &= 1/\tau
\end{aligned}
$$

The method of using an orthogonal parameterisation is mentioned in Browne et al. (2009) for non-normal examples and has also been implemented in MLwiN. For details on how we construct orthogonal vectors we refer the reader to Browne et al. (2009) but note that a function to do the procedure named *orthog* that is stored elsewhere is used in this template. Here you will see that we fit a model with the parameters *betaort* placed in the linear predictor along with data vectors *orthcons* and *orthy8*. These data vectors are constructed in the *preparedata* attribute that we detail here:

```
    preparedata = '''
from EStat.stats.utils.orthog import orthog
import numpy
mydata = data['datafile']
```

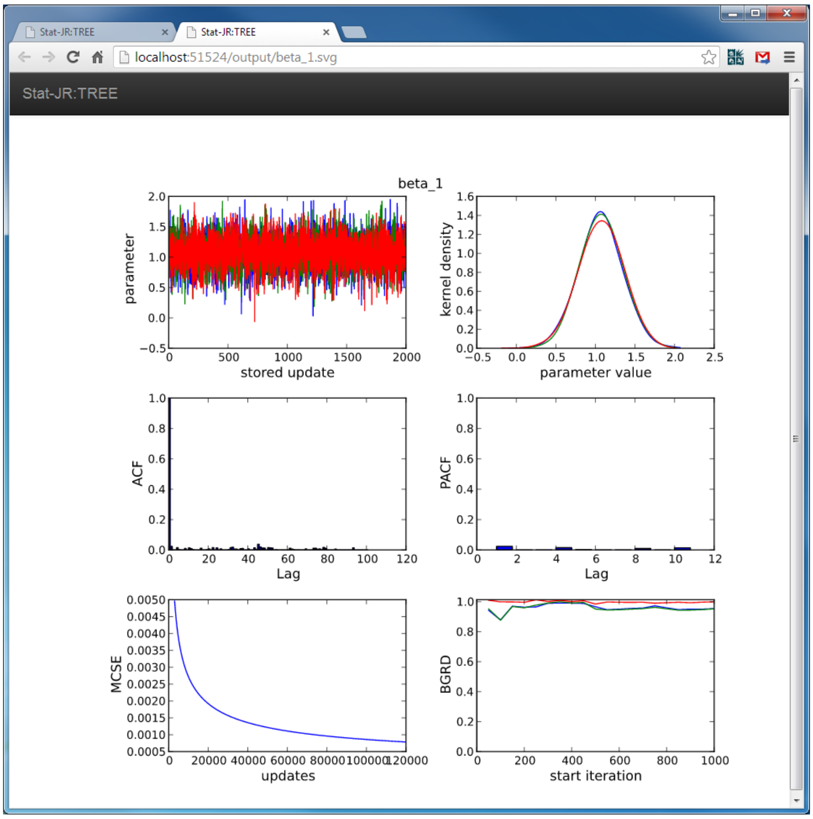
```
if useorthog:
    orth = numpy.zeros([len(mydata.variables[x[0]]['data']), len(x)])

    for i in range(0, len(x)):
        orth[:, i] = mydata.variables[x[i]]['data']

    if orthtype == 'Orthogonal':
        (tmp, om) = orthog(orth)

        orthogmat[:] = [str(i) for i in om.flat]

        for n in range(0, len(x)):
            mydata.addvariable('orth' + x[n], data = numpy.array(tmp[:,
n]).flatten())

    if orthtype == 'Orthonormal':
        (tmp, om) = numpy.linalg.qr(numpy.mat(orth))

        orthogmat[:] = [str(i) for i in om.I.flat]

        for n in range(0, len(x)):
            mydata.addvariable('orth' + x[n], data = numpy.array(tmp[:,
n]).flatten())

    x[:] = ['orth' + n for n in x]
'''
```

We begin by constructing a blank list 'orthogmat' and an empty matrix *orth*. We then implement the orthogonalising algorithm by filling *orth* with the original x variable vectors and then calling the *orthog* function. This results in *tmp* which is the matrix of orthogonal versions of the predictors and *om* which is the matrix that performs the orthogonalisation. We store this as a vector in the object 'orthogmat' . A slightly different routine is given if the user chooses Orthonormal instead here. The columns of this *tmp* matrix are then placed in objects that have the string 'orth' appended to the front of the original x variables names. Finally the original x variable names are replaced with these new orthogonal variable names before the data is returned. The *model* attribute then constructs the model code:

```
model = '''
model{
    for (i in 1:length(${y})) {
        ${y}[i] ~ \\
        % if D == 'Normal':
dnorm(mu[i], tau)
        mu[i] <- \\
        % endif
        % if D == 'Binomial':
dbin(p[i], ${n}[i])
        ${link}(p[i]) <- \\
        % endif
        % if D == 'Poisson':
dpois(p[i])
        ${link}(p[i]) <- \\
        % if offset:
${n}[i] + \\
        % endif
        % endif
%if useorthog:
${mmult(x, 'betaort', 'i')}
% else:
${mmult(x, 'beta', 'i')}
% endif
    }

    # Priors
    % for i in range(0, beta.ncols):
%if useorthog:
    betaort_${i} ~ dflat()
% else:
    beta_${i} ~ dflat()
% endif
    % endfor

% if useorthog:
<% count = 0%>
    % for i in range(0, beta.ncols):
    beta_${i} <- \\
    % for j in range(0, beta.ncols):
${orthogmat[count]} * betaort_${j}\\
% if j == (beta.ncols - 1):

% else:
% if float(orthogmat[count+1])  >= float(0.0) :
 + \\
%endif
% endif
<% count += 1 %>\\
    % endfor
    % endfor
% endif


    % if D == 'Normal':
    tau ~ dgamma(0.001000, 0.001000)
    sigma <- 1 / sqrt(tau)
    sigma2 <- 1 / tau
    % endif
}
        '''
```

Here we see that a different *mmult* function is performed for the orthogonal parameterisation and priors are given for *betaort* rather than *beta* in this case. Finally code is given to allow us to recover *beta* from *betaort* deterministically. We construct the product of the *orthogmat* terms and the *betaort*s placing + signs between the terms unless the *orthogmat* term is negative.

We can run the model by clicking on the **Run** button and we will see the following results for *beta_1* if we select *beta_1.svg* in the list:



We again see good mixing of the chains and very similar estimates to the blocking approach (Effective sample sizes for beta0 and beta1 are 5686 and 5725 respectively). The other advantage of this orthogonal approach is in it's generalisability to non-normal response models. In these cases Metropolis Hastings algorithms are used and so a blocking approach is not so straight forward.

Exercise 10

Convert this template so that it is analogous to the *Regression1* template but uses the orthogonal parametrisation. Call this new template *orthogregression*.

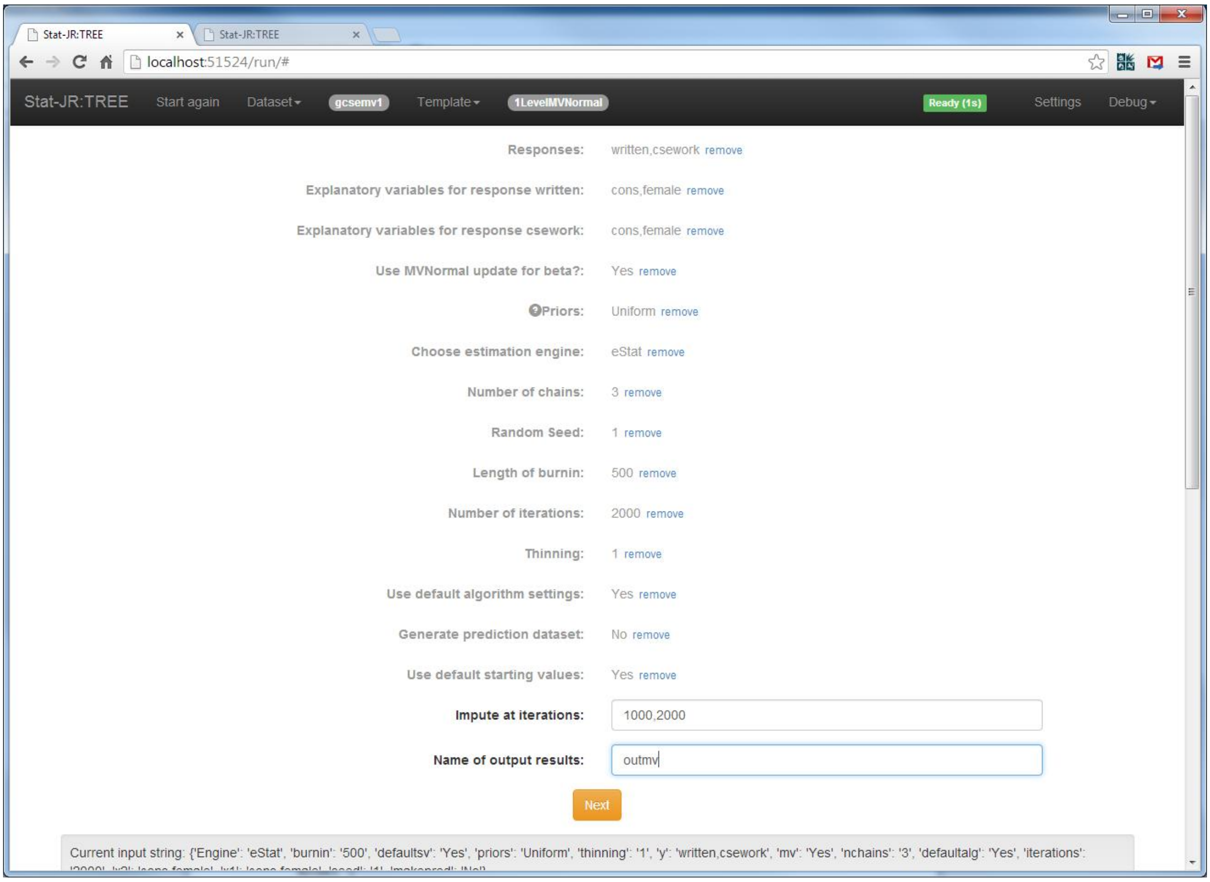
12.4 Multivariate Normal response models

Having established a method of including multivariate distributions for use with random slopes in the *preccode* we can reuse the same method to allow us to fit multivariate Normal response models We will here consider the template for fitting 1 level multivariate response models, *1LevelMVNormal.py*. This template can be used to fit models with missing data for some responses which is achieved by a method similar to that used for the probit regression and so the *preccode* will generate (at least) two steps, one for the variance matrix of the responses and an intial step to set up the missing responses. Looking at the *inputs* attribute we see the following:

```
    inputs = '''
y = DataMatrix('Responses: ')
for i in range(0, len(y)):
    context['x'+str(i+1)] = DataMatrix('Explanatory variables for response
' + y[i] + ': ', allow_cat = True)

mv = Boolean('Use MVNormal update for beta?: ')

lenbeta = 0
```

```
for i in range(0, len(y)):
    lenbeta += len(context['x'+str(i+1)])
    context['miss'+y[i]] = ParamVector(monitor=False)
n = len(y)
if n == 1:
    tau = ParamScalar()
    sigma = ParamScalar(modelled = False)
else:
    omega_e = ParamMatrix(modelled = False, customstep=True)
    omega_e.size = n
    d_e = ParamMatrix(customstep=True)
    d_e.size = n
    priors = Text('Priors: ', ['Uniform', 'Wishart'], help="<p>Note:
Uniform <em>not</em> supported by WinBUGS / OpenBUGS)</p>")
    if priors == 'Wishart':
        R = List('R matrix: ')
        v = Integer('Degrees of Freedom:')


if mv:
    beta = ParamVector(customstep=True)
else:
    beta = ParamVector()
beta.ncols = lenbeta
deviance = ParamScalar(modelled=False)
'''
```

Here you will notice that we construct parameter vectors that are a combination of the string ‘miss’ and the *y* variable names input using a *context* statement and these will be used in the model. Note that in line with the *1LevelBlock* template we have also given the option to update *beta* as a block but for now we will ignore this here. Let us run the template with the *gcsemv1* dataset that contains two responses for secondary school pupils taking General Certificate of Secondary Education (GCSE) exams in 1989, a written and a coursework test score. We will set up the inputs as follows:



Responses: written,csework remove

Explanatory variables for response written: cons,female remove

Explanatory variables for response csework: cons,female remove

Use MVNormal update for beta?: Yes remove

Priors: Uniform remove

Choose estimation engine: eStat remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

Use default algorithm settings: Yes remove

Generate prediction dataset: No remove

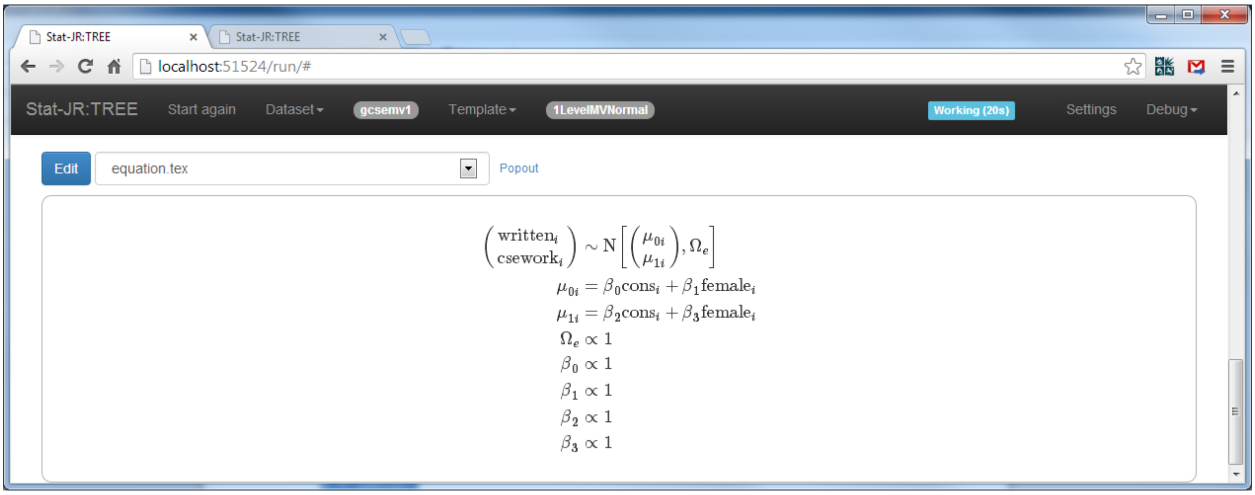
Use default starting values: Yes remove

Impute at iterations: 1000,2000

Name of output results: outmv

Current input string: {'Engine': 'eStat', 'burnin': '500', 'defaultsv': 'Yes', 'priors': 'Uniform', 'thinning': '1', 'y': 'written,csework', 'mv': 'Yes', 'nchains': '3', 'defaultalg': 'Yes', 'iterations':

Here we allow the two responses to both depend on one predictor female. Note that both responses contain missing values as there are some pupils with only a written score and some with only a coursework score. The missing values are given the value `-9.999e29` and this value will be looked for in the *preccode* function. You will also note the extra input for imputing datasets. Here we will return datasets with the current values of missing data at the prescribed iteration numbers for each chain. Clicking on the **Next** button and looking at the model output, *equation.tex* in the output pane we see:



$$\begin{pmatrix}\text{written}_i\\ \text{csework}_i\end{pmatrix} \sim \text{N}\left[\begin{pmatrix}\mu_{0i}\\ \mu_{1i}\end{pmatrix}, \Omega_e\right]$$

$$\mu_{0i} = \beta_0\text{cons}_i + \beta_1\text{female}_i$$

$$\mu_{1i} = \beta_2\text{cons}_i + \beta_3\text{female}_i$$

$$\Omega_e \propto 1$$

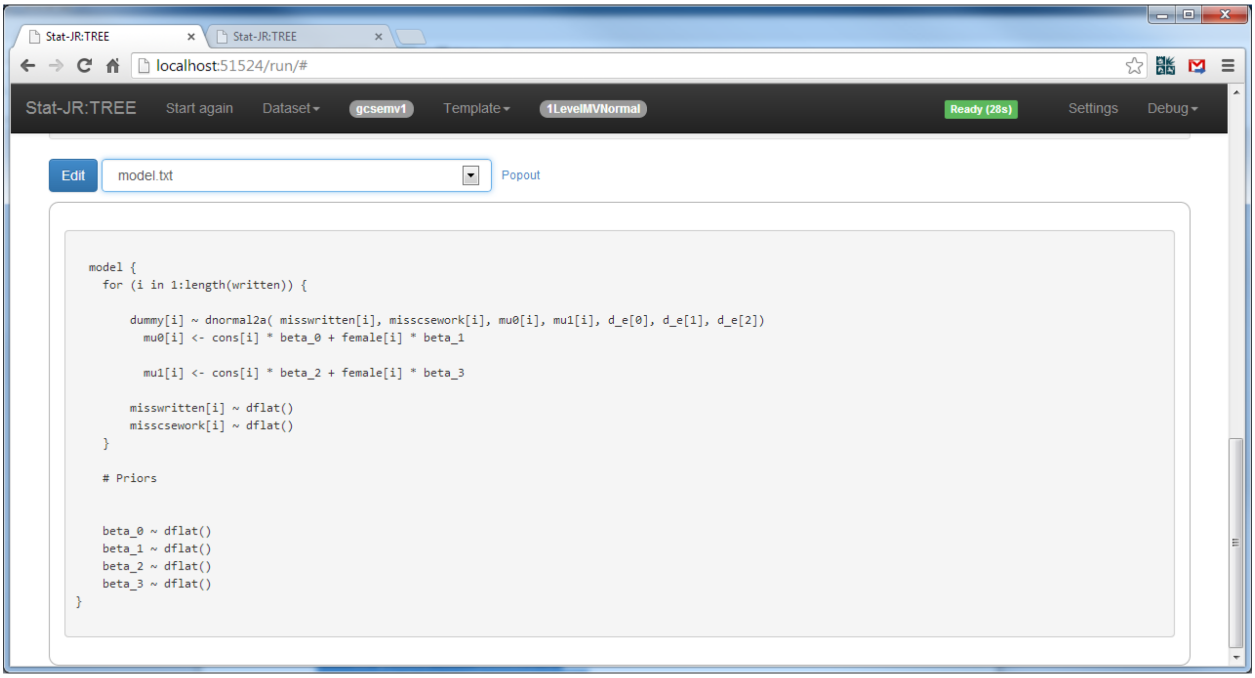
$$\beta_0 \propto 1$$

$$\beta_1 \propto 1$$

$$\beta_2 \propto 1$$

$$\beta_3 \propto 1$$

If we select *model.txt* we can see the model code thus:



```
model {
  for (i in 1:length(written)) {

      dummy[i] ~ dnormal2a( misswritten[i], misscsework[i], mu0[i], mu1[i], d_e[0], d_e[1], d_e[2])
        mu0[i] <- cons[i] * beta_0 + female[i] * beta_1

        mu1[i] <- cons[i] * beta_2 + female[i] * beta_3

      misswritten[i] ~ dflat()
      misscsework[i] ~ dflat()
  }

  # Priors


  beta_0 ~ dflat()
  beta_1 ~ dflat()
  beta_2 ~ dflat()
  beta_3 ~ dflat()
}
```

Here we see again the use of the *dnormal2a* function and also that we have included *dflat* statements for both the *misswritten* and *misscsework* responses to let the algebra system know that these are parameters. We will not look in detail at the *model* method as we can see the output it produces on the screen.

There is a *preparedata* attribute that is used to set the length of the missing data vector to equal the original response vector:

```
    preparedata = '''
mydata = data['datafile']
for i in range(0, len(y)):
    context['miss'+y[i]].ncols = -1*len(mydata.variables[y[i]]['data'])
'''
```

We next turn our attention to the *preccode* function.

12.5 The preccode function for this template

We will deal with the code here in chunks. We begin with a definition of the *mmult2* function that we will use to work out the linear predictors for each response. The *mmult2* function is specifically useful for multivariate response models as it contains a count parameter which informs us which element of beta to start with in the linear predictor:

```
    preccode = '''
<%!
    def mmult2(names, var, index,count):
        out = ""
        first = True
        for name in names:
            if first == False:
                out += ' + '
            else:
                first = False
            out += 'double(' + name + '[' + index + ']) * ' + var + '_' + 
str(count)
```

```
            count += 1
        return out
%>
            {
<% n = len(y) %>\\
```

Next we have the code chunk for generating the step for the level 1 variance matrix. This is almost identical to the random slopes code except we need the crossproduct of the level 1 residuals e (instead of the higher level random effects u) and this needs constructing which is done in the initial code using the mmult2 function:

```
                SymMatrix sb(${n});
                for(int i = 0; i < length(miss${y[0]}); i++) {
<% lenbeta = 0 %>\\
% for i in range(0, n):
                    double e${i} = double(miss${y[i]}[i]) -
(${mmult2(context['x' + str(i+1)], 'beta', 'i', lenbeta)});
<% lenbeta += len(context['x' + str(i + 1)]) %>\\
% endfor
% for i in range(0, n):
% for j in range(i, n):
                    sb(${i}, ${j}) += e${i} * e${j};
% endfor
% endfor
                }
```

Once constructed the remainder of the code follows the same pattern as random slopes and so for brevity we omit this code here, it can be viewed in *1LevelMVNormal.py*.

The one thing we have not mentioned is how the missing data is updated and here this is currently done in a slightly undesirable way, and relies on the parameter name beginning with the character string *miss.* To see how this is done we once again have to delve deeper into the code. In the subdirectory of Stat-JR with path src/lib/EStat/templates you will find some of the files that are used in the code generation. The file *gibbsstep.cpp* contains the template that is used by Stat-JR to convert the step from the algebra system into C code and in here we can modify what precisely is written in the C code. You will notice a few statements that involve the "miss" prefix at the start and end of the code:

```
% if "miss" in theta:
<% temp = theta.replace('miss', '',1) %>
if (${temp} <= -9.999e29) {
% endif
```

and

```
% if "miss" in theta:
}
% endif
```

This code recognises the prefix "miss" in a variable name and places the condition statements around the update step for that parameter. There are also some more complicated reliance on various prefixes involving "mis" but these are primarily for the mixed response modelling which we do not discuss here . Basically for the case "miss" we have:

```
% if fn == "dnorm" and "mis" in theta:
% if
% elif "miss" in theta:
        ${theta} = ${expr};
% endif
% endif
```

which simply translates to equating the variable name of interest (*theta*) to the expression the algebra system gives for its posterior (*expr*). This reliance on the parameter name is undesirable and we will hopefully come up with a better method for making such algorithmic changes in later releases.

It would be good at this point to look at the code generated for this example. To do this choose *modelcode.cpp* from the object list and scroll down. Here we see the step for the variance matrix *omega_e* near the top of the code:

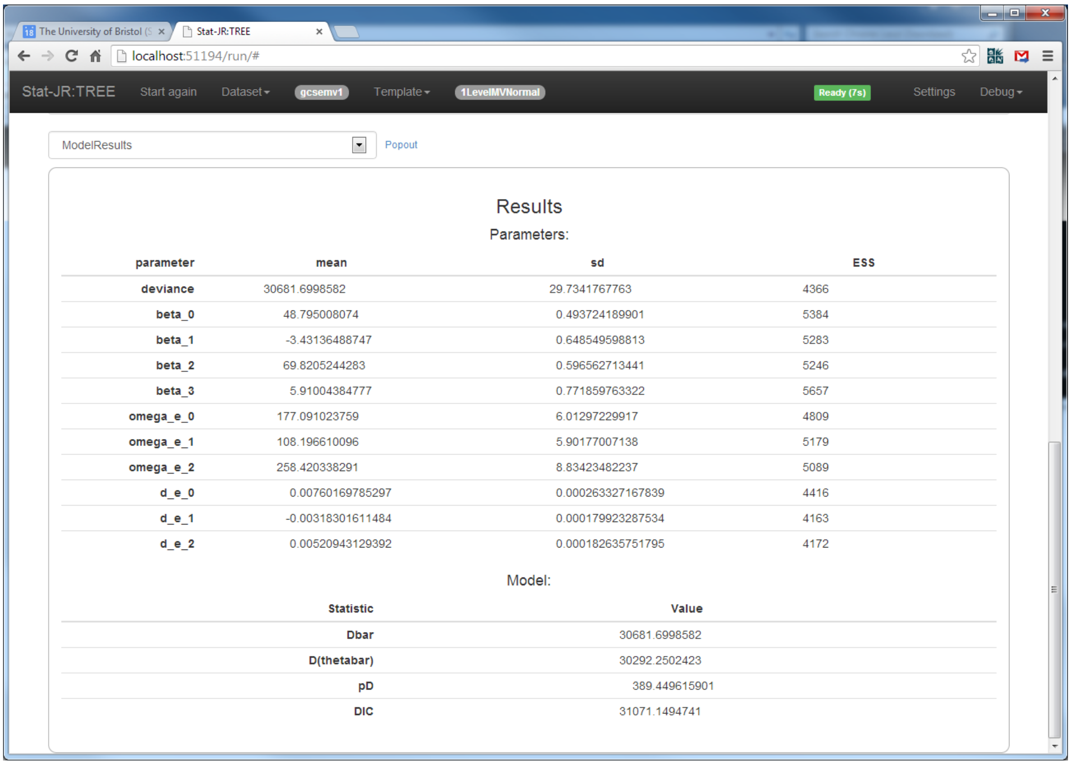
```
{

                // Note currently using a uniform prior for variance matrix
                SymMatrix sb(2);
                for(int i = 0; i < 1905; i++) {
                    double e0 = double(misswritten[i]) - (double(cons[i]) *
beta_0 + double(female[i]) * beta_1);
                    double e1 = double(misscsework[i]) - (double(cons[i]) *
beta_2 + double(female[i]) * beta_3);
                    sb(0, 0) += e0 * e0;
                    sb(0, 1) += e0 * e1;
                    sb(1, 1) += e1 * e1;
                }
                if (runstate == 0) {
                    sb(0, 0) += 0.0001;
                    sb(1, 1) += 0.0001;
                }
                matrix_sym_invinplace(sb);
                int vw = 1905 - 3;
                bool fail = false;
                mat_d_e = dwishart(vw, sb, fail);
                mat_omega_e = matrix_sym_inverse(mat_d_e, fail);
```

and later on the steps for the missing data:

```
// Update misswritten
               for(unsigned int i=0; i<1905; i++){
// This code was generated by the Stat-JR package (copyright 2012
University of Bristol and University of Southampton).
    {

if (written[i] <= -9.999e29) {
        misswritten[i] =
dnorm(((cons[i]*beta_0)+(beta_1*female[i])+((d_e[1]*misscsework[i]*pow(d_e[
0],(-1.0)))*(-1.0))+(d_e[1]*beta_2*pow(d_e[0],(-
1.0))*cons[i])+(d_e[1]*beta_3*female[i]*pow(d_e[0],(-1.0)))),d_e[0]);
}
    }
               }
// Update misscsework
               for(unsigned int i=0; i<1905; i++){
// This code was generated by the Stat-JR package (copyright 2012
University of Bristol and University of Southampton).
```

```
     {
if (csework[i] <= -9.999e29) {
          misscsework[i] = dnorm((((d_e[1]*pow(d_e[2],(-
1.0))*misswritten[i])*(-1.0))+(d_e[1]*pow(d_e[2],(-
1.0))*beta_0*cons[i])+(d_e[1]*pow(d_e[2],(-
1.0))*beta_1*female[i])+(beta_2*cons[i])+(beta_3*female[i])),d_e[2]);
}
     }
                   }
```

We can run the template by clicking on the **Run** button and choosing the **ModelResults** in the list we see:



Results

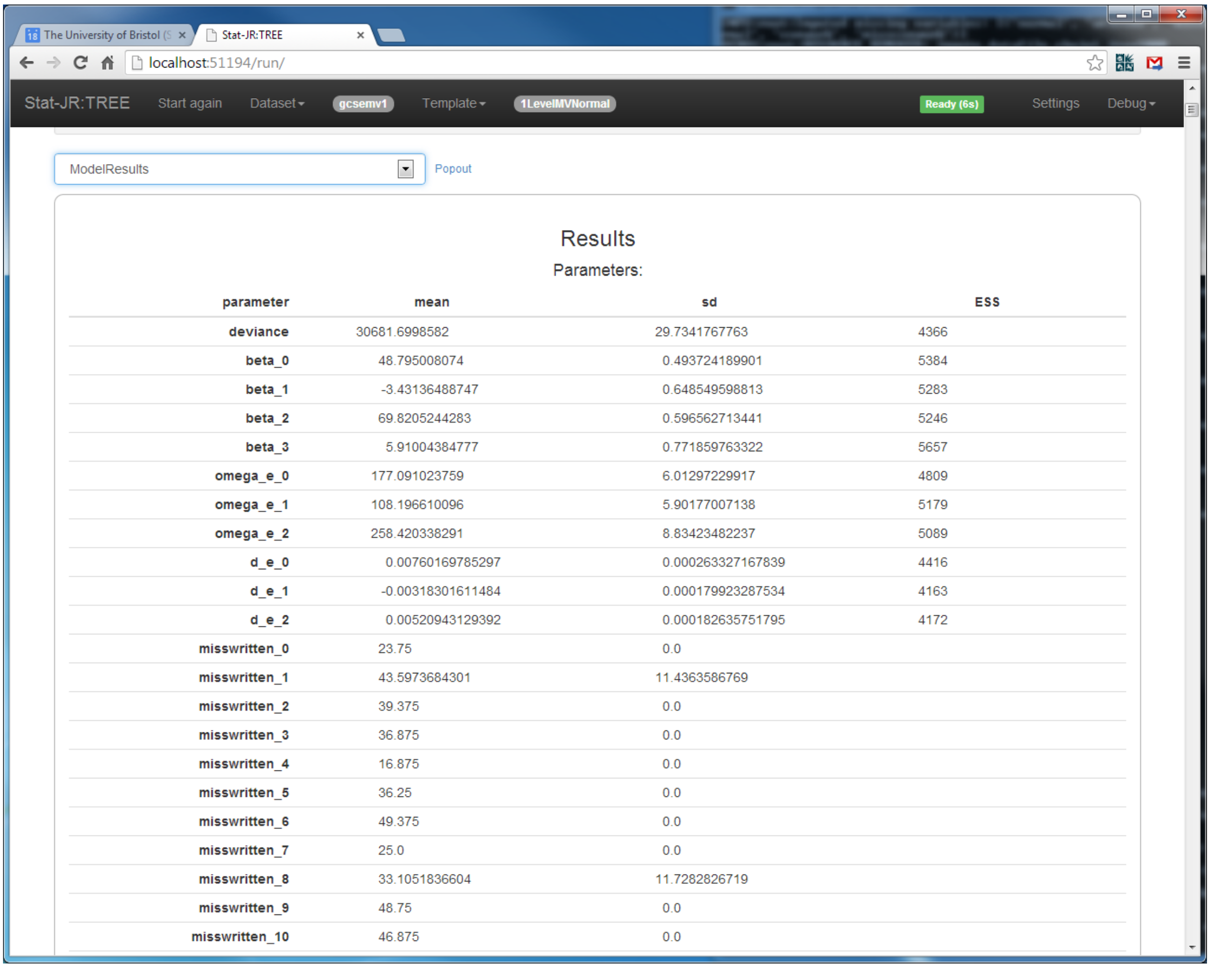
Parameters:

parameter	mean	sd	ESS
deviance	30681.6998582	29.7341767763	4366
beta_0	48.795008074	0.493724189901	5384
beta_1	-3.43136488747	0.648549598813	5283
beta_2	69.8205244283	0.596562713441	5246
beta_3	5.91004384777	0.771859763322	5657
omega_e_0	177.091023759	6.01297229917	4809
omega_e_1	108.196610096	5.90177007138	5179
omega_e_2	258.420338291	8.83423482237	5089
d_e_0	0.00760169785297	0.000263327167839	4416
d_e_1	-0.00318301611484	0.000179923287534	4163
d_e_2	0.00520943129392	0.000182635751795	4172

Model:

Statistic	Value
Dbar	30681.6998582
D(thetabar)	30292.2502423
pD	389.449615901
DIC	31071.1494741

Note here we do not see the missing values as by default non-monitored nodes are not displayed in **ModelResults**. To view the missing values you would need to return to the **Settings** screen (from the main screen) and click on the tick box that allows you to *Include unmonitored values in results* and click on **Set** before setting up the model again. At present due to the number of nodes this takes a very long time in the browser so we do not advise you to try however if you do then eventually the screen will look as follows:

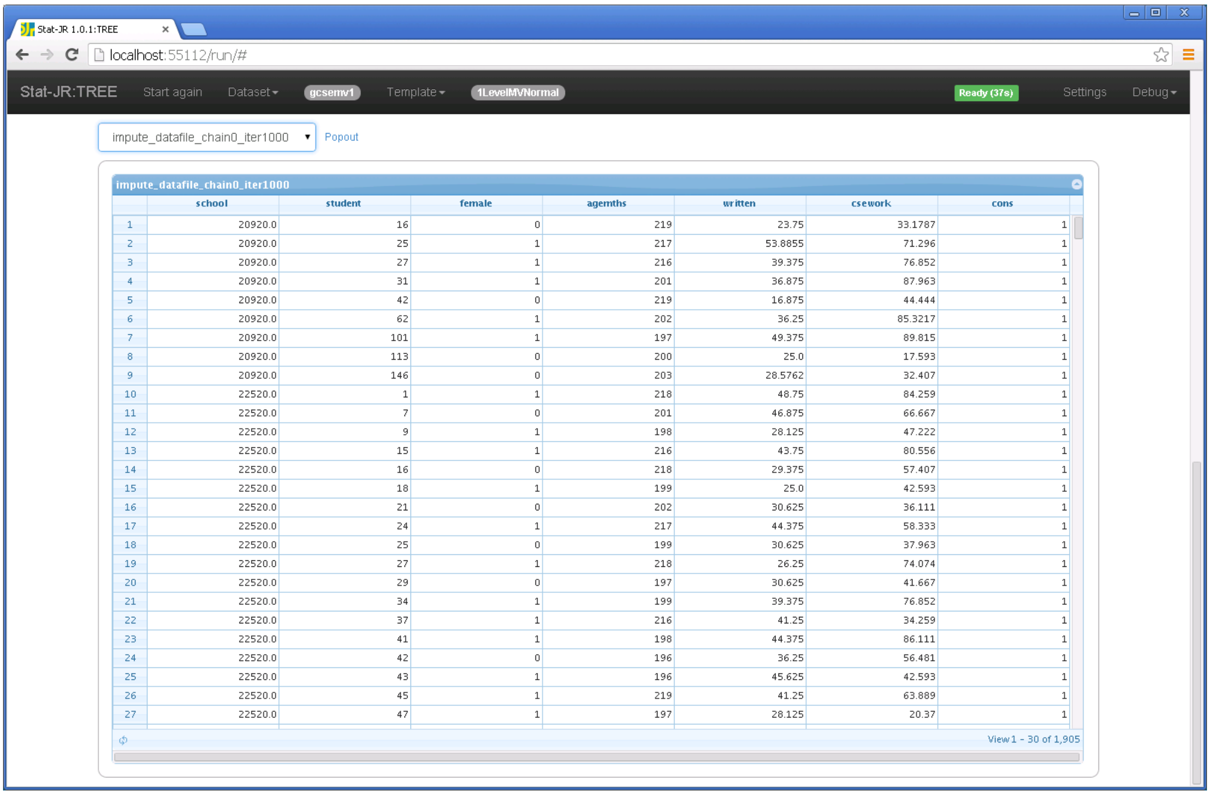


Results

Parameters:

parameter	mean	sd	ESS
deviance	30681.6998582	29.7341767763	4366
beta_0	48.795008074	0.493724189901	5384
beta_1	-3.43136488747	0.648549598813	5283
beta_2	69.8205244283	0.596562713441	5246
beta_3	5.91004384777	0.771859763322	5657
omega_e_0	177.091023759	6.01297229917	4809
omega_e_1	108.196610096	5.90177007138	5179
omega_e_2	258.420338291	8.83423482237	5089
d_e_0	0.00760169785297	0.000263327167839	4416
d_e_1	-0.00318301611484	0.000179923287534	4163
d_e_2	0.00520943129392	0.000182635751795	4172
misswritten_0	23.75	0.0	
misswritten_1	43.5973684301	11.4363586769	
misswritten_2	39.375	0.0	
misswritten_3	36.875	0.0	
misswritten_4	16.875	0.0	
misswritten_5	36.25	0.0	
misswritten_6	49.375	0.0	
misswritten_7	25.0	0.0	
misswritten_8	33.1051836604	11.7282826719	
misswritten_9	48.75	0.0	
misswritten_10	46.875	0.0	

Here you will see that for the missing data variables the ones that correspond to actual data have standard deviation zero in the output as they shouldn’t change from iteration to iteration so for example the first and third written scores were observed. We can also look at the values of these missing data at prescribed iterations/chains and so selecting impute_datafile_chain0_iter1000 gives the following:



impute_datafile_chain0_iter1000

	school	student	female	agemths	written	csework	cons
1	20920.0	16	0	219	23.75	33.1787	1
2	20920.0	25	1	217	53.8855	71.296	1
3	20920.0	27	1	216	39.375	76.852	1
4	20920.0	31	1	201	36.875	87.963	1
5	20920.0	42	0	219	16.875	44.444	1
6	20920.0	62	1	202	36.25	85.3217	1
7	20920.0	101	1	197	49.375	89.815	1
8	20920.0	113	0	200	25.0	17.593	1
9	20920.0	146	0	203	28.5762	32.407	1
10	22520.0	1	1	218	48.75	84.259	1
11	22520.0	7	0	201	46.875	66.667	1
12	22520.0	9	1	198	28.125	47.222	1
13	22520.0	15	1	216	43.75	80.556	1
14	22520.0	16	0	218	29.375	57.407	1
15	22520.0	18	1	199	25.0	42.593	1
16	22520.0	21	0	202	30.625	36.111	1
17	22520.0	24	1	217	44.375	58.333	1
18	22520.0	25	0	199	30.625	37.963	1
19	22520.0	27	1	218	26.25	74.074	1
20	22520.0	29	0	197	30.625	41.667	1
21	22520.0	34	1	199	39.375	76.852	1
22	22520.0	37	1	216	41.25	34.259	1
23	22520.0	41	1	198	44.375	86.111	1
24	22520.0	42	0	196	36.25	56.481	1
25	22520.0	43	1	196	45.625	42.593	1
26	22520.0	45	1	219	41.25	63.889	1
27	22520.0	47	1	197	28.125	20.37	1

View 1 - 30 of 1,905

Here the second written test score has value 53.89, and the ninth written score is 28.58, while their means are 43.60 and 33.11 respectively across all iterations and chains. We have extended these multivariate normal modelling templates to more levels and to include random slopes. They also form the basis for the mixed response templates which allow other response types via the use of latent variables and mimic and extend the functionality that exists in the REALCOM software program. You will see that these templates are pretty big and involve coding in several languages (Python, WinBUGS model code, LaTeX and C++). It is hoped that with advances in the algebra system that the reliance on the *preccode* functions will reduce but if you want to look at the other multivariate templates you will see many similarities in the code in these functions. This is one of the plus points of the ability to view the code in the templates within the Stat-JR system.

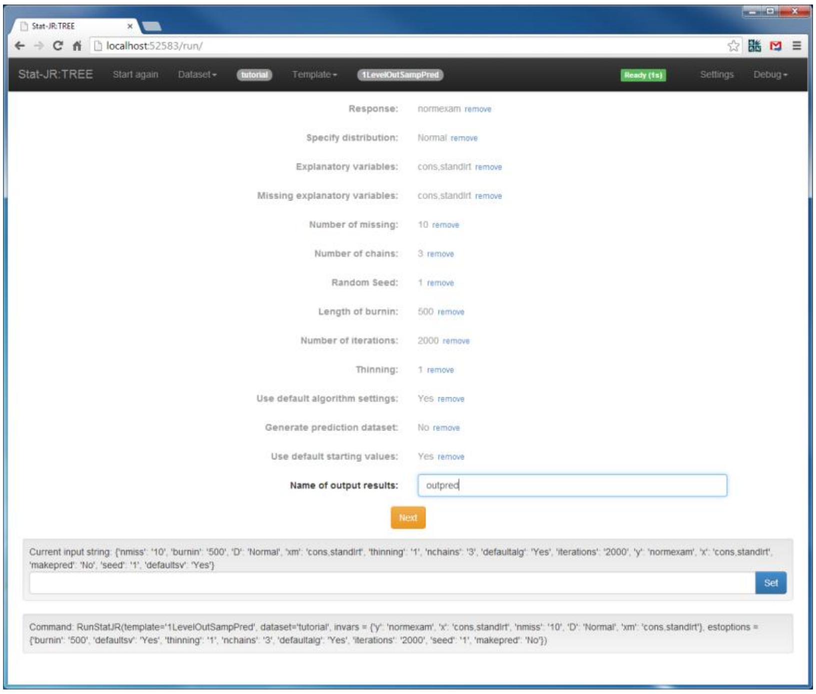
We will finish this documentation by considering one more example of getting more from the MCMC estimation engine.

13 Out of sample predictions

Most of the statistical modelling templates we have thus far created are primarily being used for statistical inference. We might however be interested in using the model to predict future responses. The advantage of a simulation-based approach is that we can easily get confidence intervals about these predictors at the same time as we estimate the model. We do however have to be careful that we do not feedback the results of our predicting into the estimation part of the model. WinBUGS has a method to do this with its cut function and we have developed a similar method which we will demonstrate here.

13.1 The 1LevelOutSampPred template – using the zxfd trick

We will illustrate our approach on a 1 level model which we can fit using the *1LevelOutSampPred* template. We will firstly choose this template along with the *tutorial* dataset and then select the following inputs:



Response: normexam remove

Specify distribution: Normal remove

Explanatory variables: cons,standlrt remove

Missing explanatory variables: cons,standlrt remove

Number of missing: 10 remove

Number of chains: 3 remove

Random Seed: 1 remove

Length of burnin: 500 remove

Number of iterations: 2000 remove

Thinning: 1 remove

Use default algorithm settings: Yes remove

Generate prediction dataset: No remove

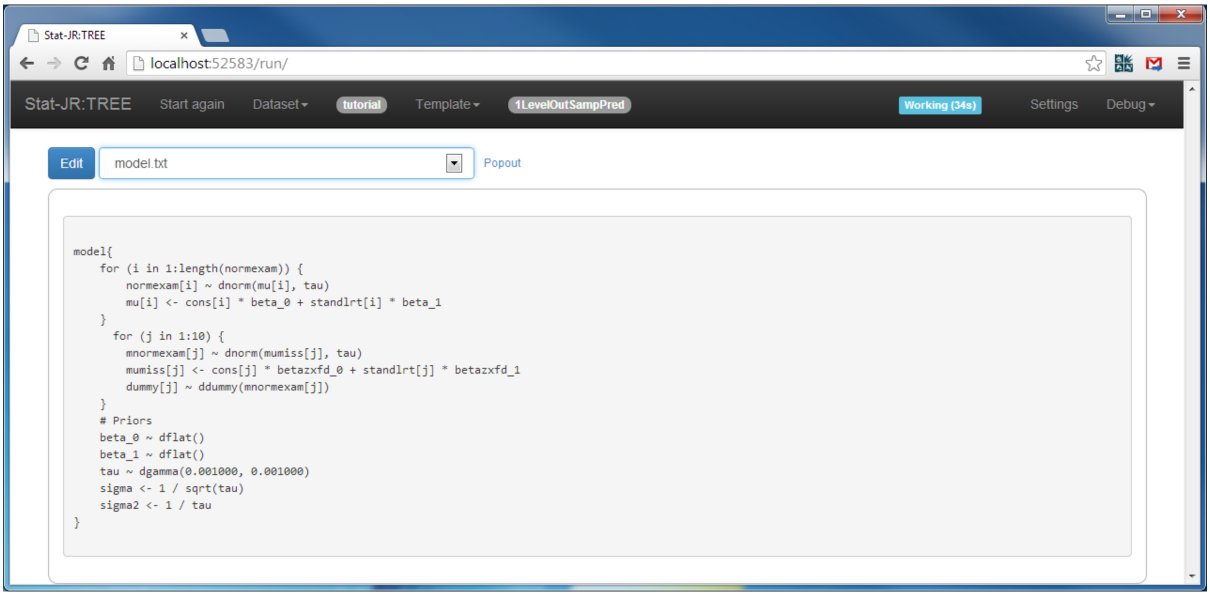
Use default starting values: Yes remove

Name of output results: outpred

```
Current input string: {'nmiss': '10', 'burnin': '500', 'D': 'Normal', 'xm': 'cons,standlrt', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'makepred': 'No', 'seed': '1', 'defaultsv': 'Yes'}
```

```
Command: RunStatJR(template='1LevelOutSampPred', dataset='tutorial', invars = {'y': 'normexam', 'x': 'cons,standlrt', 'nmiss': '10', 'D': 'Normal', 'xm': 'cons,standlrt'}, estoptions = {'burnin': '500', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'seed': '1', 'makepred': 'No'})
```

To explain what is going on we are planning to fit a regression model to *normexam* with predictor *standlrt* as we have done previously using the *1LevelMod* template. We will then use the predictors given in 'missing explanatory variables' to predict the 10 individuals who in this case have the same scores as the first 10 in the model actually fit. Note if you want to predict other individuals you need to form new columns of the same length as the data although the values below the 'Number of missing' row will be ignored. Clicking on the **Next** button and choosing *model.txt* gives the following output:



```
model{
    for (i in 1:length(normexam)) {
        normexam[i] ~ dnorm(mu[i], tau)
        mu[i] <- cons[i] * beta_0 + standlrt[i] * beta_1
    }
      for (j in 1:10) {
        mnormexam[j] ~ dnorm(mumiss[j], tau)
        mumiss[j] <- cons[j] * betazxfd_0 + standlrt[j] * betazxfd_1
        dummy[j] ~ ddummy(mnormexam[j])
    }
    # Priors
    beta_0 ~ dflat()
    beta_1 ~ dflat()
    tau ~ dgamma(0.001000, 0.001000)
    sigma <- 1 / sqrt(tau)
    sigma2 <- 1 / tau
}
```

Here you will notice that we have an additional *j* loop in the model code for the out-of-sample predictions which will be stored in *mnormexam*. There are two interesting parts to this code: Firstly the line

```
mumiss[j] <- cons[j] * betazxfd_0 + standlrt[j] * betazxfd_1
```

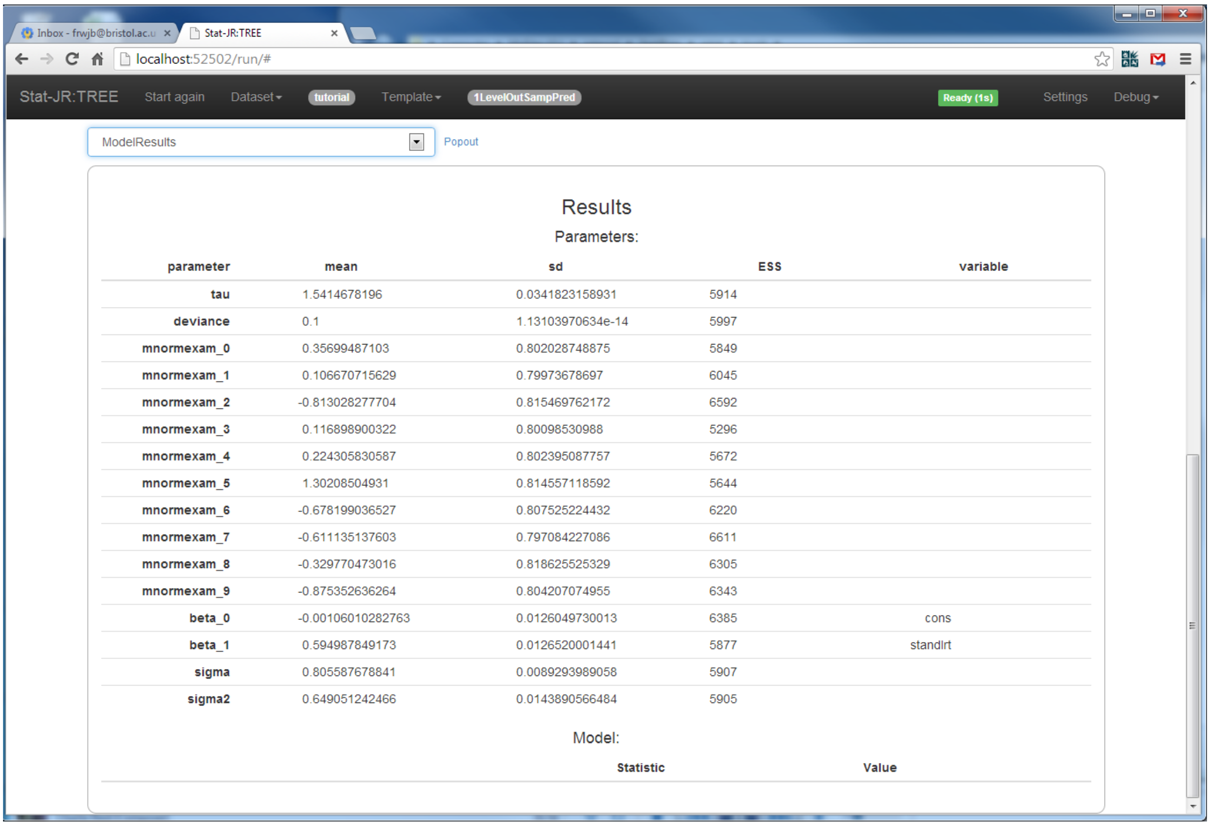
has the strange string *zxfd* placed in the middle of the two parameter names. This is our way of stopping the predictions from feeding back to the model parameter estimation (equivalent to performing the cut function in WinBUGS). Basically as the predictors in this line are not *beta0* and *beta1* then this line will not influence the posteriors for the fixed effects. The posterior for *mnormexam* will be calculated but this is only because we include the line

```
dummy[j] ~ ddummy(mnormexam[j])
```

so that *mnormexam* appears on both the left and right hand side within the model code to differentiate it from data. The algebra system will formulate the posterior which will depend on *betazxfd0* and *betazxfd1*. Of course in practice we want these replaced by the correct *beta0* and *beta1* and this is done in the bowels of the code generator with the lines:

```
elif type == 'variable':
                    result=l['name'].replace('.', '_').replace('zxfd','')
```

which is in code that is not currently available to view by the user. If we run the template we get the following output for *ModelResults*:



Results

Parameters:

parameter	mean	sd	ESS	variable
tau	1.5414678196	0.0341823158931	5914	
deviance	0.1	1.13103970634e-14	5997	
mnormexam_0	0.35699487103	0.802028748875	5849	
mnormexam_1	0.106670715629	0.79973678697	6045	
mnormexam_2	-0.813028277704	0.815469762172	6592	
mnormexam_3	0.116898900322	0.80098530988	5296	
mnormexam_4	0.224305830587	0.802395087757	5672	
mnormexam_5	1.30208504931	0.814557118592	5644	
mnormexam_6	-0.678199036527	0.807525224432	6220	
mnormexam_7	-0.611135137603	0.797084227086	6611	
mnormexam_8	-0.329770473016	0.818625525329	6305	
mnormexam_9	-0.875352636264	0.804207074955	6343	
beta_0	-0.00106010282763	0.0126049730013	6385	cons
beta_1	0.594987849173	0.0126520001441	5877	standlrt
sigma	0.805587678841	0.0089293989058	5907	
sigma2	0.649051242466	0.0143890566484	5905	

Model:

Statistic	Value

Here you can see that the out of sample predictions, *mnormexam* have been estimated with standard errors. We do not get a DIC diagnostic here as this would include the missing data and be a little misleading. We have recently incorporated the ability to use additional datasets in a template and we may in the future update this template to allow the data to be predicted to be in a different dataset. We hope that this and other templates give you a flavour of the possibilities that are available in the Stat-JR package. The package is still evolving and so we very much welcome feedback and suggestions for improvement. We also encourage you to send us your own templates for inclusion with the software.

Exercise 11

Try modifying the *regression1* template to allow for out of sample predictions. Call the new template *regression1pred*.

14 Solutions to the exercises

Rather than fill many pages with Python code we will place potential solutions to each of the exercises in a solutions directory on the Stat-JR website at a later date. Below we list the filenames for each of the exercises:

Exercise 1 LinReg.py
Exercise 2 Random.py
Exercise 3 RecodeNew.py
Exercise 4 AvandCorr.py
Exercise 5 XYPlotNew.py
Exercise 6 1LevelLogit.py
Exercise 7 2levelinteractions.py
Exercise 8 nlevelinteractions.py
Exercise 9 2levelwithRS.py
Exercise 10 orthogregression.py
Exercise 11 regression1pred.py

References

Browne, W.J., Steele F., Golalizadeh, M., and Green M.J. (2009) The use of simple reparameterizations to improve the efficiency of Markov chain Monte Carlo estimation for multilevel models with applications to discrete time survival models *Journal of Royal Statistical Society, Series A.* 172: 579-598

Hadfield, J. D. (2009) MCMC Methods for Multi-Response Generalized Linear Mixed Models: The MCMCglmm R Package. *Journal of Statistical Software*, **33**:1-22

Hetland, M.L. (2005) *Beginning Python: From Novice to Professional.* Springer-Verlag. New York.

Lillard, L.A. and Panis C.W.A. (2003) *aML Multilevel Multiprocess Statistical Software, Version 2.0.* EconWare, Los Angeles, California.

Lunn, D.J., Thomas, A., Best, N., and Spiegelhalter, D. (2000). WinBUGS - a Bayesian modelling framework: concepts, structure, and extensibility. *Statistics and Computing*, **10**: 325--337.

Lunn, D., Spiegelhalter, D., Thomas, A. and Best, N. (2009). The BUGS project: Evolution, critique, and future directions, *Statistics in Medicine*, **28**, 3049-3067.

Lutz, M. and Ascher, D. (2005) *Learning Python, Second Edition*. O'Reilly Media, Sebastopol, CA.

Plummer, M. (2003). JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling, *Proceedings of the 3rd International Workshop on Distributed Statistical Computing* (DSC 2003), March 20–22, Vienna, Austria. ISSN 1609-395X.

R Development Core Team (2011). *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0

Rasbash, J., Charlton, C., Browne, W.J., Healy, M. and Cameron, B. (2009). *MLwiN Version 2.1*. Centre for Multilevel Modelling, University of Bristol.