A Beginner's Guide to Stat-JR's TREE Interface version 1.0.7

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, UK.

**Electronics and Computer Science,

University of Southampton, UK.

September 2019

A Beginner’s Guide to Stat-JR’s TREE interface version 1.0.7

ISBN: To be confirmed

Printed in the United Kingdom

Acknowledgements

The Stat-JR software is very much a team effort and is the result of work funded initially 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. The work then continued with the ESRC grant ES/K007246/1 and more recently via further grants from the British Academy and the ESRC NCRM.

We are therefore grateful to our funders 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.

Thank you also to Rhiannon Moore who has assisted in updating the screen shots for version 1.06.

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 2018.

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 are 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" or 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 its 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, 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, MCMC Estimation in MLwiN, v2.36, 2016) (which can be downloaded from http://www.bristol.ac.uk/cmm/software/mlwin/download/manuals.html) 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 Beginner’s guide

We have written several guides to go with the software: this Beginner’s Guide will cover how to start up and run the software, with a particular focus on the *TREE (Template Reading and Execution Environment)* interface. It will provide some simple examples and is designed for the researcher who wishes to be able to use the software package without worrying too much about how the mathematics behind the modelling works. As such, it does not go into detail on how users can contribute to extending the software themselves: that is covered in the second, Advanced User’s, guide, designed for those who want to understand in greater detail how the system works (a Quick-start guide is also available providing a very brief overview). There is also an E-book User’s guide which deals with the software’s *DEEP (Documents with Embedded Execution and Provenance)* E-book interface. Finally, two guides were released alongside Stat-JR 1.0.5 to support the workflow system, *LEAF (Logging and Execution of Analysis Flows)* and also the Statistical Analysis Assistant features of Stat-JR and a short guide to the SPSS training material generation has been released with this version.

As well as these Guides, we also publish support, such as answers to frequently asked questions, on our website (http://www.bristol.ac.uk/cmm/software/statjr), where you can also find our forum in which users can discuss the software.

In this Beginner’s Guide we look at an example application taken from education research, fitting a Normal response model for a continuous outcome. Here our aim is more to illustrate how to use the software than primarily how to do the best analysis of the dataset in question, and we will demonstrate the interoperability features with some of the other software packages that link to Stat-JR as well. We will then look at a second example from demography that illustrates binomial response models for a discrete outcome.

2 Installing and Starting Stat-JR

2.1 Installing Stat-JR

Stat-JR has a dedicated website (http://www.bristol.ac.uk/cmm/software/statjr) from which you can request a copy of the software, and which contains instructions for installation.

2.2 The use of third party software and licenses

Stat-JR is written primarily in the Python (Rossum) (see https://www.python.org/) package but also makes use of many other third party software packages. We are grateful to the developers of these programs for allowing us to use their products within our package. When you have installed Stat-JR you will find a directory entitled licences in which you can find subdirectories for each package detailing the licensing agreement for each. The list of software packages that we are using can be found in the Appendix to this document.

2.3 Starting up *TREE*

Stat-JR's interface is viewed and operated via a web browser, but it is started by running an executable file.

To start Stat-JR select the *Stat-JR TREE* link from the *Centre for Multilevel Modelling* suite on the start up menu. This action opens 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 we have run *TREE*, you can see commands like the following:

```
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
```

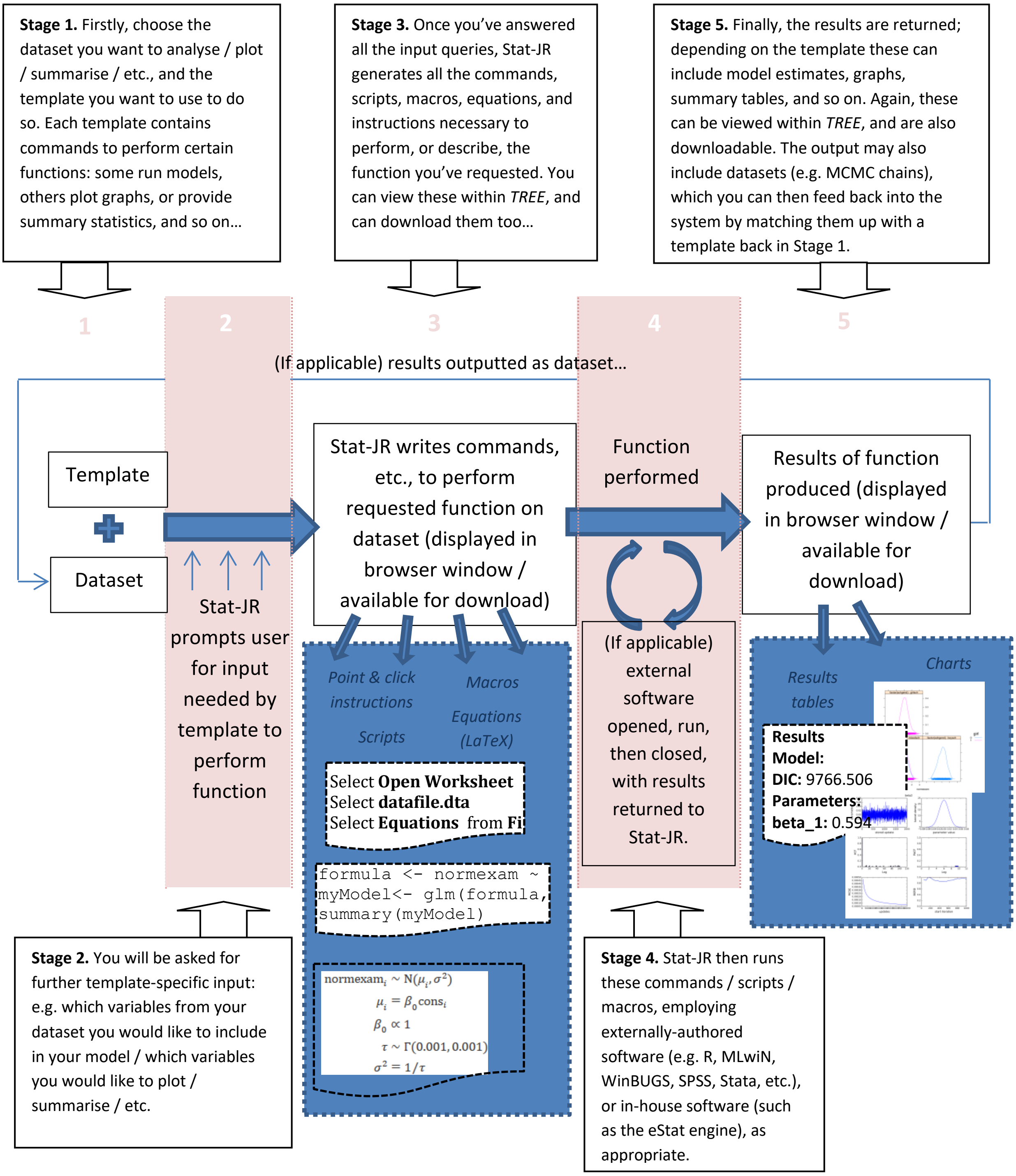
The last line quoted here (although more lines will appear beneath it on start-up) indicates that Stat-JR is locating the default web browser on your machine; once it has done so it will open that web browser and display TREE's welcome page. The lines such as "WARNING:root:Failed to load package GenStat model (GenStat not found)" are not necessarily problematic but are warning you that the Genstat (VSN International, 2015) statistical package – one of the third-party statistical packages with which Stat-JR can interoperate – has not be found (where Stat-JR expects to find it if it is installed) on your particular machine.

Stat-JR works best with either Chrome or Firefox, so if the default browser on your machine is Internet Explorer it is best to open a different browser and copy the html path to it; this will be something like localhost:52228 (although the number will likely differ each time you run Stat-JR). You can change your default browser via Settings in the Chrome menu, or via Options > General in the Firefox menu (both menus are found in top-right of their respective browser windows).

2.4 The structure and layout of the *TREE* interface

Stat-JR can be thought of as a system that manages the use of a set of templates written either by the developers, and supplied with the software, or by users themselves. Each template will perform a specific function: for example, fitting a specific family of models, summarising a dataset, plotting a graph, and so on. The Stat-JR system therefore allows the user to select and use specific templates with their datasets, and to capture and display the outputs that result.

When operating Stat-JR through *TREE*, you generally proceed through the following five stages:



Stage 1. Firstly, choose the dataset you want to analyse / plot / summarise / etc., and the template you want to use to do so. Each template contains commands to perform certain functions: some run models, others plot graphs, or provide summary statistics, and so on…

Stage 3. Once you've answered all the input queries, Stat-JR generates all the commands, scripts, macros, equations, and instructions necessary to perform, or describe, the function you've requested. You can view these within *TREE*, and can download them too…

Stage 5. Finally, the results are returned; depending on the template these can include model estimates, graphs, summary tables, and so on. Again, these can be viewed within *TREE*, and are also downloadable. The output may also include datasets (e.g. MCMC chains), which you can then feed back into the system by matching them up with a template back in Stage 1.

Stat-JR prompts user for input needed by template to perform function

Stat-JR writes commands, etc., to perform requested function on dataset (displayed in browser window / available for download)

Select **Open Worksheet**
Select **datafile.dta**
Select **Equations** from **Fi**

```
formula <- normexam ~
myModel<- glm(formula,
summary(myModel)
```

$$\text{normexam}_i \sim \text{N}(\mu_i, \sigma^2)$$
$$\mu_i = \beta_0 \text{cons}_i$$
$$\beta_0 \propto 1$$
$$\tau \sim \Gamma(0.001, 0.001)$$
$$\sigma^2 = 1/\tau$$

Function performed

(If applicable) external software opened, run, then closed, with results returned to Stat-JR.

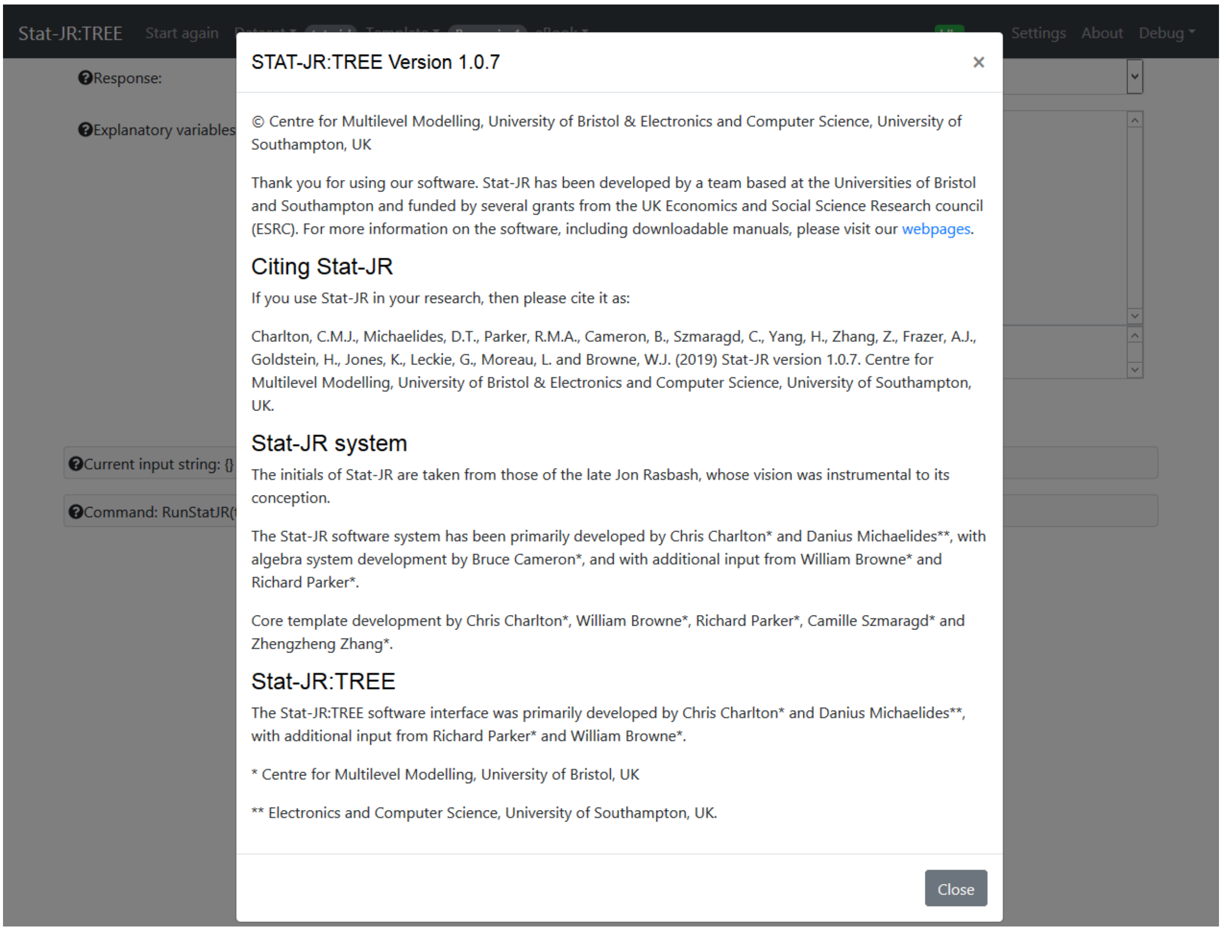
Results of function produced (displayed in browser window / available for download)

Results
Model:
DIC: 9766.506
Parameters:
beta_1: 0.594

Stage 2. You will be asked for further template-specific input: e.g. which variables from your dataset you would like to include in your model / which variables you would like to plot / summarise / etc.

Stage 4. Stat-JR then runs these commands / scripts / macros, employing externally-authored software (e.g. R, MLwiN, WinBUGS, SPSS, Stata, etc.), or in-house software (such as the eStat engine), as appropriate.

Returning to our start-up of the software, when the line http://0.0.0.0:50215/ appears, and after refreshing the web browser, the browser window should appear. We can click on the **About** button to the top right and the following will appear:



STAT-JR:TREE Version 1.0.7 ×

© Centre for Multilevel Modelling, University of Bristol & Electronics and Computer Science, University of Southampton, UK

Thank you for using our software. Stat-JR has been developed by a team 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.

Citing Stat-JR

If you use Stat-JR in 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. (2019) Stat-JR version 1.0.7. Centre for Multilevel Modelling, University of Bristol & Electronics and Computer Science, University of Southampton, UK.

Stat-JR system

The initials of Stat-JR are taken from those of the late Jon Rasbash, whose vision was instrumental to its conception.

The Stat-JR software system has been primarily developed by Chris Charlton* and Danius Michaelides**, with algebra system development by Bruce Cameron*, and with additional input from William Browne* and Richard Parker*.

Core template development by Chris Charlton*, William Browne*, Richard Parker*, Camille Szmaragd* and Zhengzheng Zhang*.

Stat-JR:TREE

The Stat-JR:TREE software interface was primarily developed by Chris Charlton* and Danius Michaelides**, with additional input from Richard Parker* and William Browne*.

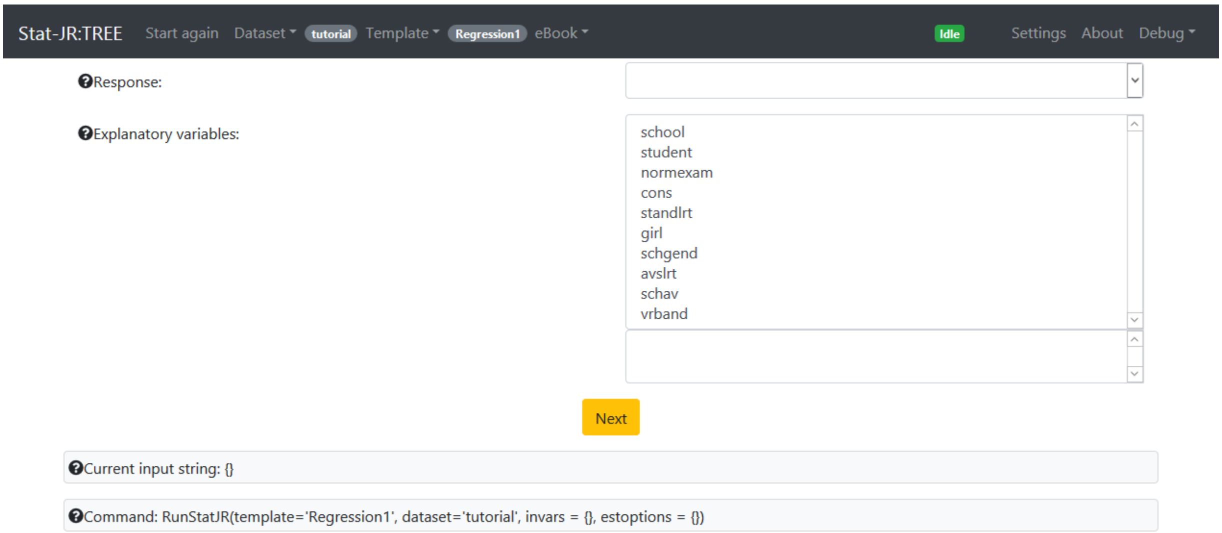
* Centre for Multilevel Modelling, University of Bristol, UK

** Electronics and Computer Science, University of Southampton, UK.

Close

This window contains information on funders, authors, and a link to the Stat-JR website which contains further guidance, such as answers to frequently asked questions, and a user forum.

Pressing **Close** returns us to the main Stat-JR TREE screen:



Response:

Explanatory variables:

school
student
normexam
cons
standlrt
girl
schgend
avslrt
schav
vrband

Current input string: {}

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

At the top you'll see a black **title bar**. From left to right, this contains:

- The words **Stat-JR:TREE** that identifies the interface;

- an option (**Start again**) to clear all inputs the user has chosen for the current template;

- a **Dataset** menu allowing the user to **Choose**, **Drop** (from temporary memory cache), **View** the dataset (as well as summary statistics, the option to add / delete variables, edit data values, edit descriptive labels, duplicate the current dataset, etc.), return a **List** of datasets, and **Upload** / **Download** (see Section 6) datasets. For example, selecting **Dataset > Choose** returns a scrollable list of all the datasets that the system is aware of: i.e. those which are in the user's data folder (by default under *Users\user_name\.statjr\datasets*) and those globally available (in the *datasets* subdirectory of this installation of Stat-JR). This pane can be used to change the selected dataset via the **Use** button; for inputting your own data set you can use the **Upload** button;

- the name of the currently-selected dataset (in the grey box) – if you hover your cursor over this name, it returns a textual description of the dataset if one exists;

- a **Template** menu allowing the user to **Choose**, **List** (described below), **Upload** individual templates not already uploaded in the current session or **Set Inputs** for the current template (as an input string, rather than pointing-and-clicking through the inputs; this option also allows you to retrieve and re-use input values from previous template executions). If you select **Template** > **Choose**, a box appears which contains a scrollable list of all the templates that the system is aware of: i.e. those which are in the user's template folder (by default under *Users\user_name\.statjr\templates*) and those globally available (in the *templates* subdirectory of this installation of Stat-JR). This can be used to change the selected template via the **Use** button. Each template has defined '*tags*' which are terms to describe what it does: these appear as blue phrases in the 'cloud' above the list of templates, whereas the estimation engines supported by each template appear in the cloud in red. When you select a template, its name and description appear to the right of the list. Clicking on the symbol that looks like a baggage label returns the tags for that template, whereas clicking on the 'cog' symbol returns a list of engines that particular template supports;

- the name of the currently-selected template (in the grey box) – again, if you hover your cursor over this name, it also returns a description of the template;

- an eBook menu that allows you to **Load** up an eBook, **Edit** it, **Manage static files** involved with it and **Clear** it. We will not look at this menu in this guide.

- a progress gauge indicating whether Stat-JR is "Idle" (before it has run anything), "Ready" (once it has run something, and is ready for further user input), "Initialising", "Working" or whether it has encountered an "Error";

- a link to a box that pops up containing options to change a variety of **Settings**. This allows the user to specify the path to their datasets, templates, workflows, and the eBook information store (where eBooks loaded into Stat-JR's DEEP interface are saved, although it is unlikely users would wish to edit this folder directly: amendments to it can be made via the DEEP interface itself). These paths refer to *user-specific* folders (by default under *Users\user_name\.statjr*) whilst those datasets, templates and workflows which are *globally* available (to all users) are saved in their respective subdirectories in this installation of Stat-JR. The **Settings** window also displays a number of settings that the program uses with each possible software package: some of these are relatively straightforward, such as where the executables for each package are found, and some are relatively advanced, such as for the *eStat* engine, optimisation, starting values and standalone code options;

- a **Debug** menu; this produces a drop-down list from which one can choose to **Reload templates**, **Reload datasets** or **Reload packages**, allowing users to upload changes to files they make outside the *TREE* interface, without having to start-up *Stat-JR* again. For example, a user could paste a new dataset into the (global or user-specific) datasets directory, or modify a template in the (global or user-specific) templates directory, and reload them so that they appear in their lists in the browser window. In addition, if the user changes the path to a third-party software package (via **Settings**), then **Reload packages** will implement this change in the current session.

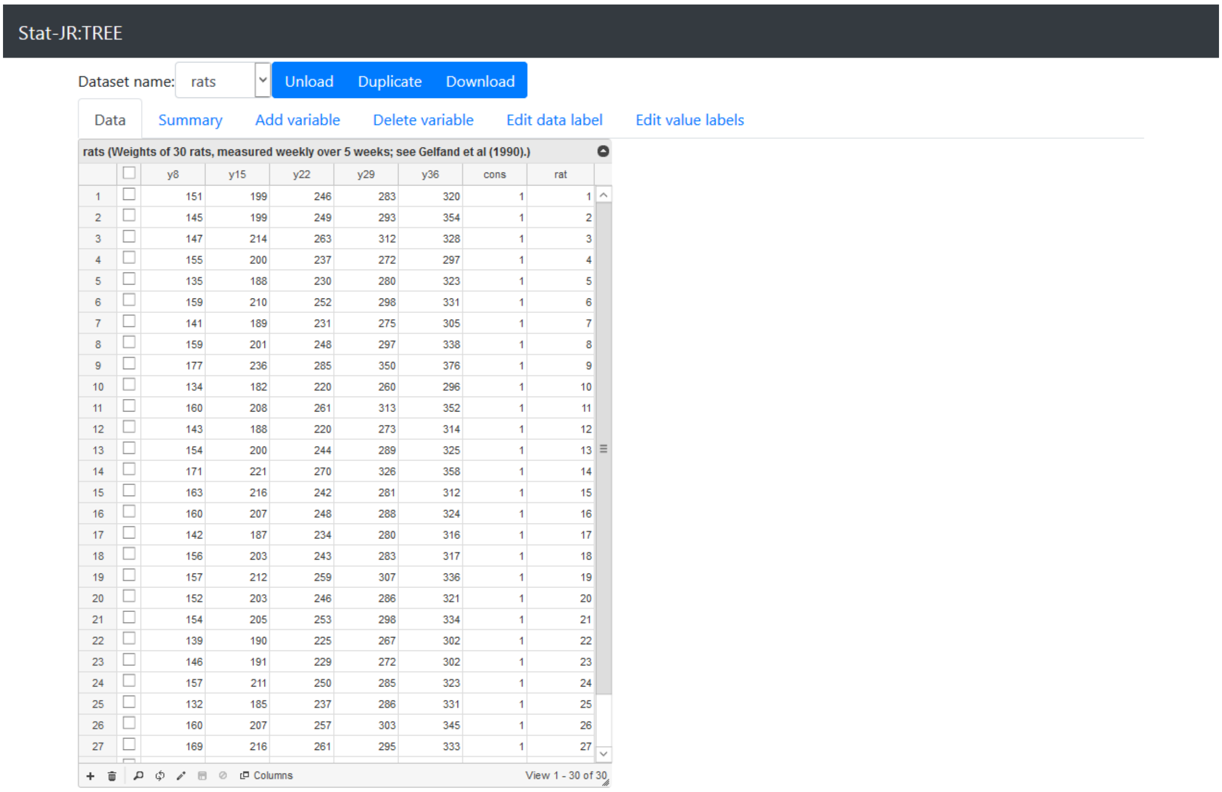
We will now look at The **View dataset** window:

Select **Dataset > Choose** from the menu in the black **title bar**.

Scroll down the dataset list, towards the bottom, and click on **rats**; its name and description will appear to the right of the list.

Click on the **Use** button, and the name of the current dataset (in the grey box in the black title bar at the top) should have changed accordingly.

Select **Dataset > View**; this will open a new tab in your browser: if you click on this you will be able to see the dataset we have just selected, as follows:



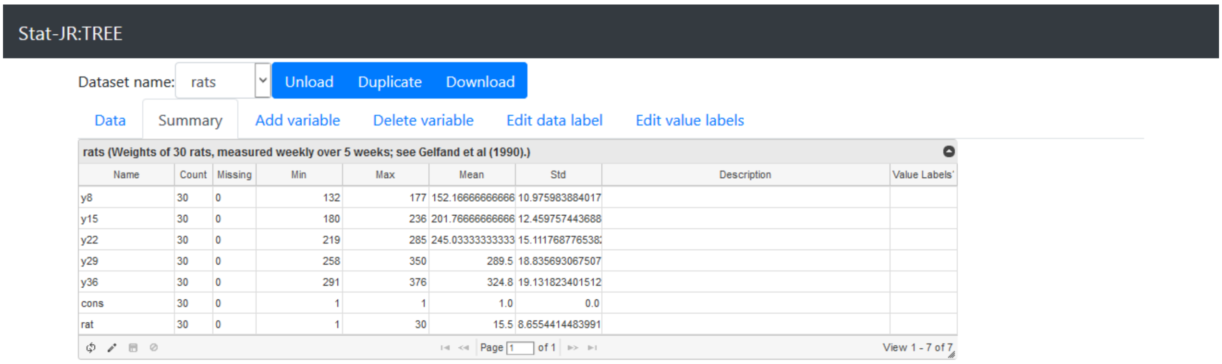
rats (Weights of 30 rats, measured weekly over 5 weeks; see Gelfand et al (1990).)

	□	y8	y15	y22	y29	y36	cons	rat
1	□	151	199	246	283	320	1	1
2	□	145	199	249	293	354	1	2
3	□	147	214	263	312	328	1	3
4	□	155	200	237	272	297	1	4
5	□	135	188	230	280	323	1	5
6	□	159	210	252	298	331	1	6
7	□	141	189	231	275	305	1	7
8	□	159	201	248	297	338	1	8
9	□	177	236	285	350	376	1	9
10	□	134	182	220	260	296	1	10
11	□	160	208	261	313	352	1	11
12	□	143	188	220	273	314	1	12
13	□	154	200	244	289	325	1	13
14	□	171	221	270	326	358	1	14
15	□	163	216	242	281	312	1	15
16	□	160	207	248	288	324	1	16
17	□	142	187	234	280	316	1	17
18	□	156	203	243	283	317	1	18
19	□	157	212	259	307	336	1	19
20	□	152	203	246	286	321	1	20
21	□	154	205	253	298	334	1	21
22	□	139	190	225	267	302	1	22
23	□	146	191	229	272	302	1	23
24	□	157	211	250	285	323	1	24
25	□	132	185	237	286	331	1	25
26	□	160	207	257	303	345	1	26
27	□	169	216	261	295	333	1	27

View 1 - 30 of 30

The **rats** dataset is a small, longitudinal animal growth dataset which contains the weights of 30 laboratory rats on 5 weekly occasions from 8 days of age (see (Gelfand, Hills, Racine-Poon, & Smith, 1990) for more details). The five measurements are labelled *y8, y15, y22, y29* and *y36*, respectively, and the dataset also contains a constant column – a vector of ones, named *cons*, and a rat identifier column, *rat.* Initially, we are going to perform a regression analysis of the initial weight (*y8*) on the final weight (*y36*), including an intercept (*cons*). The tabs above the dataset allow the user to quickly add a new variable or delete an existing variable from the dataset. We can also view a summary of the dataset:

To view a summary of the dataset, click on the **Summary** tab above the data and the screen will look as follows:



rats (Weights of 30 rats, measured weekly over 5 weeks; see Gelfand et al (1990).)

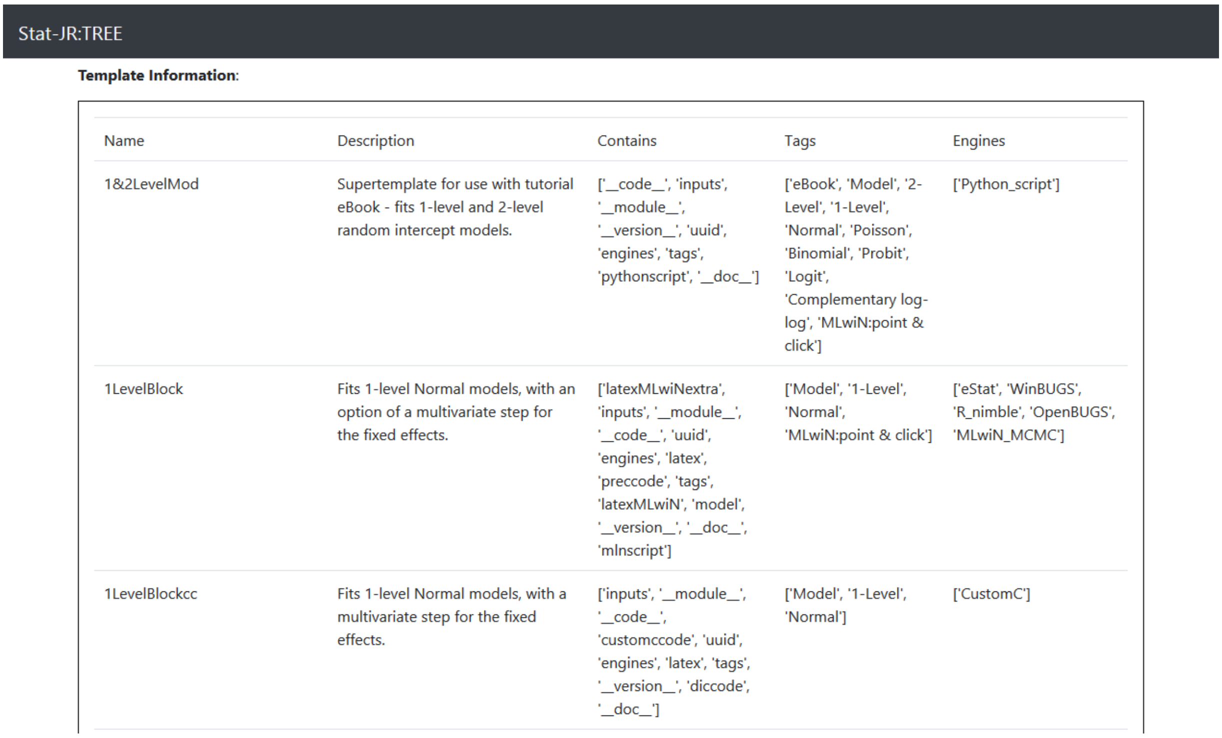
Name	Count	Missing	Min	Max	Mean	Std	Description	Value Labels'
y8	30	0	132	177	152.16666666666	10.975983884017		
y15	30	0	180	236	201.76666666666	12.459757443688		
y22	30	0	219	285	245.03333333333	15.111768776538		
y29	30	0	258	350	289.5	18.835693067507		
y36	30	0	291	376	324.8	19.131823401512		
cons	30	0	1	1	1.0	0.0		
rat	30	0	1	30	15.5	8.6554414483991		

View 1 - 7 of 7

Here we get a very short summary of the dataset, giving, for each variable, the minimum value, maximum value, mean and standard deviation. If the dataset has had descriptions added or has categorical variables then they will appear in the last two columns. More extensive summaries are available by using specific templates to summarise datasets, as we will see later.

Let's now look at the **Template** menu:

Back on the main page, if you click on **Template > List** the following screen will appear in a new tab:



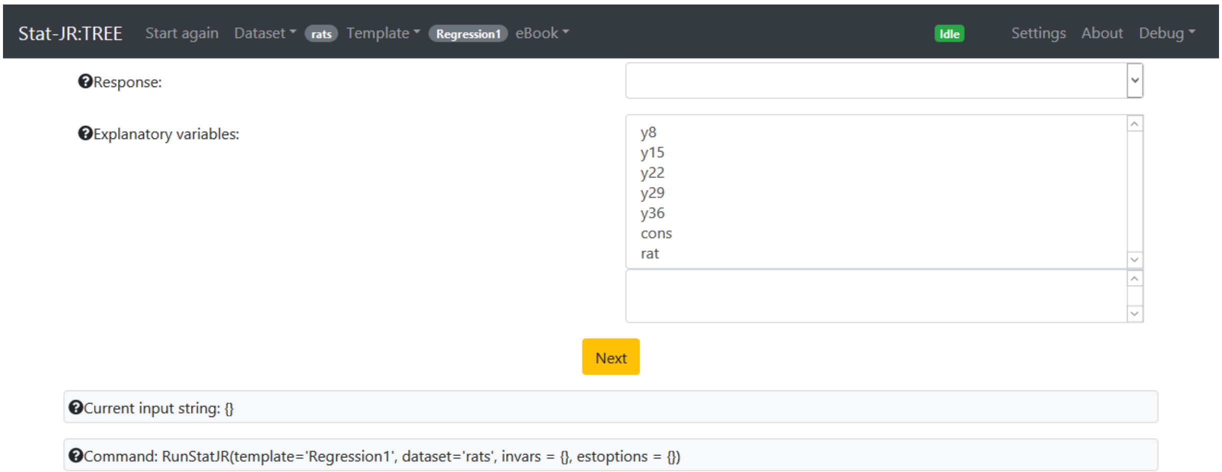
Template Information:

Name	Description	Contains	Tags	Engines
1&2LevelMod	Supertemplate for use with tutorial eBook - fits 1-level and 2-level random intercept models.	['__code__', 'inputs', '__module__', '__version__', 'uuid', 'engines', 'tags', 'pythonscript', '__doc__']	['eBook', 'Model', '2-Level', '1-Level', 'Normal', 'Poisson', 'Binomial', 'Probit', 'Logit', 'Complementary log-log', 'MLwiN:point & click']	['Python_script']
1LevelBlock	Fits 1-level Normal models, with an option of a multivariate step for the fixed effects.	['latexMLwiNextra', 'inputs', '__module__', '__code__', 'uuid', 'engines', 'latex', 'preccode', 'tags', 'latexMLwiN', 'model', '__version__', '__doc__', 'mlnscript']	['Model', '1-Level', 'Normal', 'MLwiN:point & click']	['eStat', 'WinBUGS', 'R_nimble', 'OpenBUGS', 'MLwiN_MCMC']
1LevelBlockcc	Fits 1-level Normal models, with a multivariate step for the fixed effects.	['inputs', '__module__', '__code__', 'customccode', 'uuid', 'engines', 'latex', 'tags', '__version__', 'diccode', '__doc__']	['Model', '1-Level', 'Normal']	['CustomC']

This rather busy screen (we don't reproduce it all here, due to its length) contains, in the two columns on the left, a tabular list of all the templates that are available with a short description of what each template does. The next column is of more interest to advanced users, and contains a list of functions in the template code, whilst the final two columns contain the tags that identify the template type, and the engines that are supported by the template.

We will next demonstrate running a template, using the default **Regression1** template that fits a 1-level Normal response regression model: this is appropriate as the response, the weights of the rats, is a continuous measure.

Return to the main menu screen, which should look as follows:



❓Response:

❓Explanatory variables:

y8
y15
y22
y29
y36
cons
rat

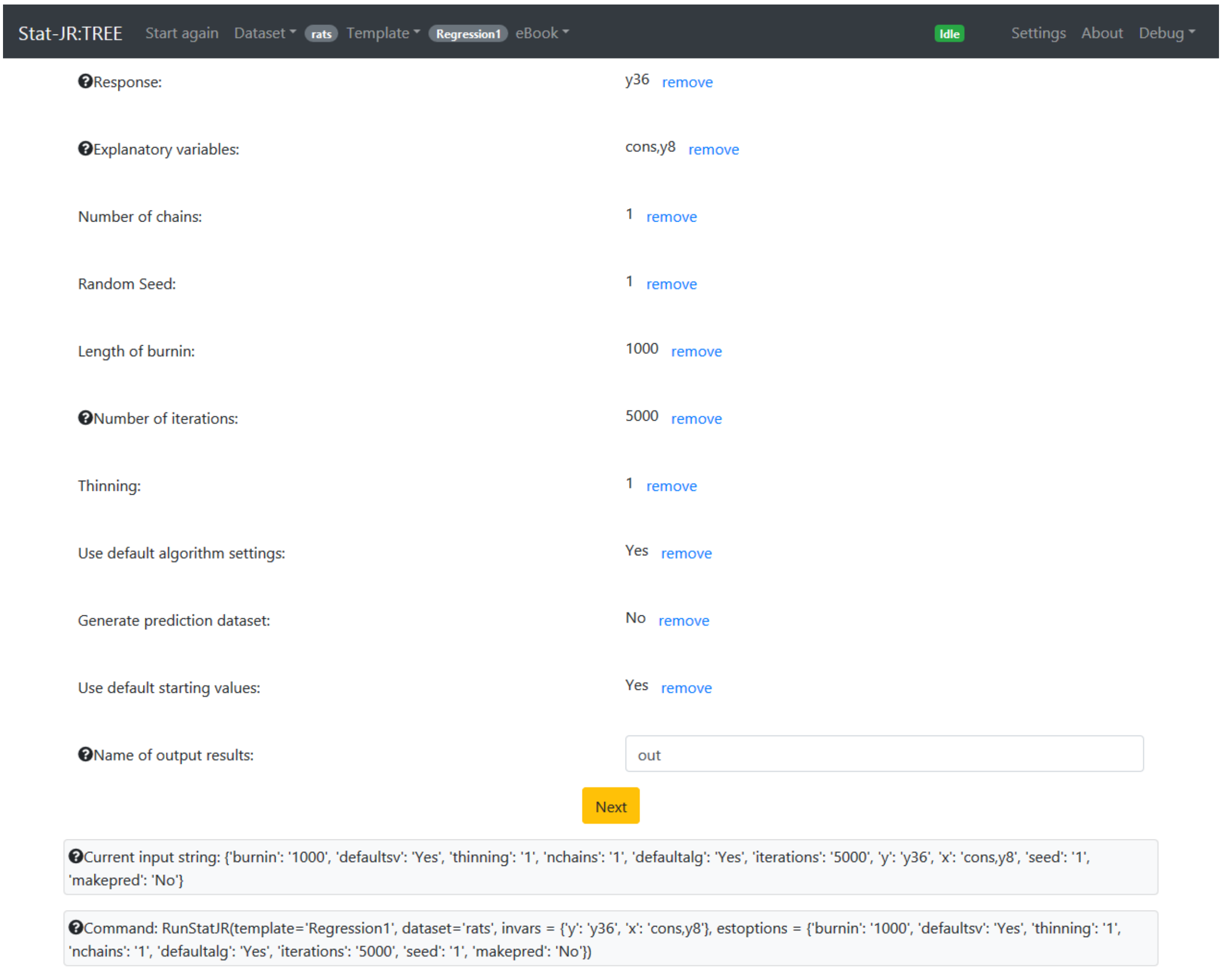
❓Current input string: {}

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

In the middle of the screen you can see the inputs required for this template (these are template-specific, and so will likely change when you use a different template). Since some inputs are conditional (i.e. are only required when earlier inputs take specific values), the opportunity to specify inputs proceeds through sequential steps. Here we see the two initially-required inputs for the **Regression1** template are the **Response** variable and **Explanatory variables.** Since this template only allows for one response variable to be specified, a pull-down list is displayed, but since it allows for several explanatory variables to be specified, a multiple selection list is displayed for that input value. In the case of the latter, variables are selected by clicking on their name in the left-hand list; to de-select them, click on their name in the right-hand list.

The **Start again** link (in the top black bar) will clear any inputs the user has already selected and return you to the first template input screen (i.e. the current screen, in this case), whilst the **Next** button will allow the user to move on and specify further inputs once those on the current screen have all been chosen.

Use the input controls and the **Next** button(s) to fill in the screen as follows:



❓Response:	y36
❓Explanatory variables:	cons,y8
Number of chains:	1
Random Seed:	1
Length of burnin:	1000
❓Number of iterations:	5000
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': '1000', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '1', 'defaultalg': 'Yes', 'iterations': '5000', 'y': 'y36', 'x': 'cons,y8', 'seed': '1', 'makepred': 'No'}

❓Command: RunStatJR(template='Regression1', dataset='rats', invars = {'y': 'y36', 'x': 'cons,y8'}, estoptions = {'burnin': '1000', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '1', 'defaultalg': 'Yes', 'iterations': '5000', 'seed': '1', 'makepred': 'No'})

So here we have entered **Response**: *y36*; **Explanatory variables**: *cons,y8*; **Number of chains**: *1*; **Random Seed**: *1*; **Length of burnin**: *1000*; **Number of iterations**: *5000*; **Thinning**: *1*; **Use default algorithm settings**: *Yes*; **Generate prediction dataset**: *No*; **Use default starting values**: *Yes*; **Name of output results**: *out*.

Note that an option to **remove** appears next to each input previously submitted; this will remove the current input, but keep the other inputs you have specified (as far as it can; if they are conditional on the input you have removed, then they will be, out of necessity, removed too).

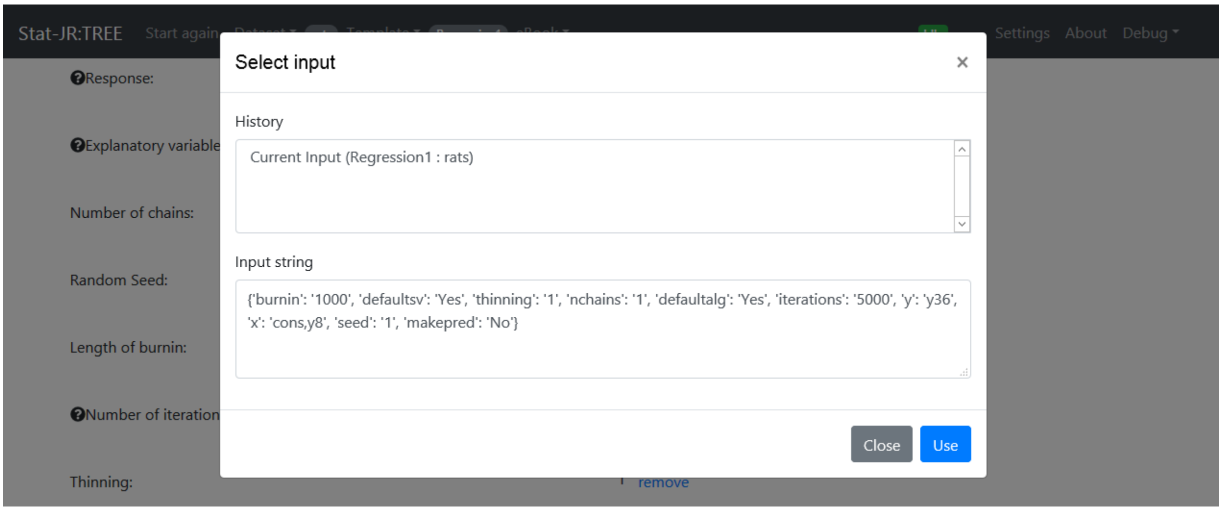
So, here we are performing a regression of the initial weight (*y8*) on the final weight (*y36*), including an intercept (*cons*). The other inputs refer to the Monte Carlo Markov chain (MCMC) estimation procedures in Stat-JR. MCMC estimation methods are simulation-based, and so require certain parameters to be set. The methods involve taking a series of random (dependent) draws from the posterior distribution of the model parameters in order to summarise each parameter. The inputs required here are as follows:

- **Number of chains**: this is the number of starting points from which we will take random draws;
- **Random Seed**: the value from which random numbers are initially drawn. This allows repeatability, as a run using the same starting values and random seed will give the same answers. When multiple chains are used this seed is generally multiplied by the chain number to give a unique seed for each chain;
- **Length of burnin**: the initial length of the chain (i.e. the number of iterations at the start) which are excluded from the parameter summaries (the rationale for this is explained a little further in the example, below, with the *tutorial* dataset);
- **Number of iterations***:* the length of chain following the burnin, from which the parameter summaries are drawn;
- **Thinning**: this determines how often the values are stored: i.e. store every *n*th iteration.

By answering *Yes* to the question **Use default algorithm settings**, we have used defaults for other settings for which we will therefore not be prompted to complete. By answering *No* to **generate prediction dataset** we have chosen not to generate a dataset of predictions from our model. By answering *Yes* to **Use default starting values** we have chosen not to start the chain at values of our choosing, instead accepting Stat-JR's defaults. We will discuss MCMC estimation in slightly more detail in the applications in the next section. The final input we're asked for is the **Name of output results**: this is the name (here we've chosen *out*) given to any dataset of parameter chains that results from running the template.

You will notice, towards the bottom of the window, a box with a rather long text string labelled **Current input string** above it and another labelled as **Command** below it. The input string allows the user to specify all the inputs directly, via the **Set Inputs** option in the **Template** pull-down list, without having to point-and-click through the list as we have done. If you click on **Template > Set Inputs** you will see this input string reproduced in the **Input string** box; clicking on the **Use** button populates the inputs with these values, which obviously will have no effect here, but it would if you first changed a value, or indeed used the inputs from a previously-run template execution, as selected from the **History** box above. The input string needs to be in a certain format, as illustrated below, and note that the inputs will often not have exactly the same name as appears in the prompts you answered earlier (e.g. **Use default starting values**: *Yes* corresponds to 'defaultsv': 'Yes'). In this guide we will reproduce the input string for each example (and also specify the dataset and template, in case it is unclear from the screenshot, etc.) so you can simply copy and paste it in if you need to (although note that the appropriate template will need to be pre-selected – it won't change the template for you). So the input string for the inputs we have just specified is as follows:

Dataset: *rats;* **Template:** *Regression1;* **Input string:** *{'burnin': '1000', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '1', 'defaultalg': 'Yes', 'iterations': '5000', 'y': 'y36', 'x': 'cons,y8', 'seed': '1', 'makepred': 'No'}*



Select input

History

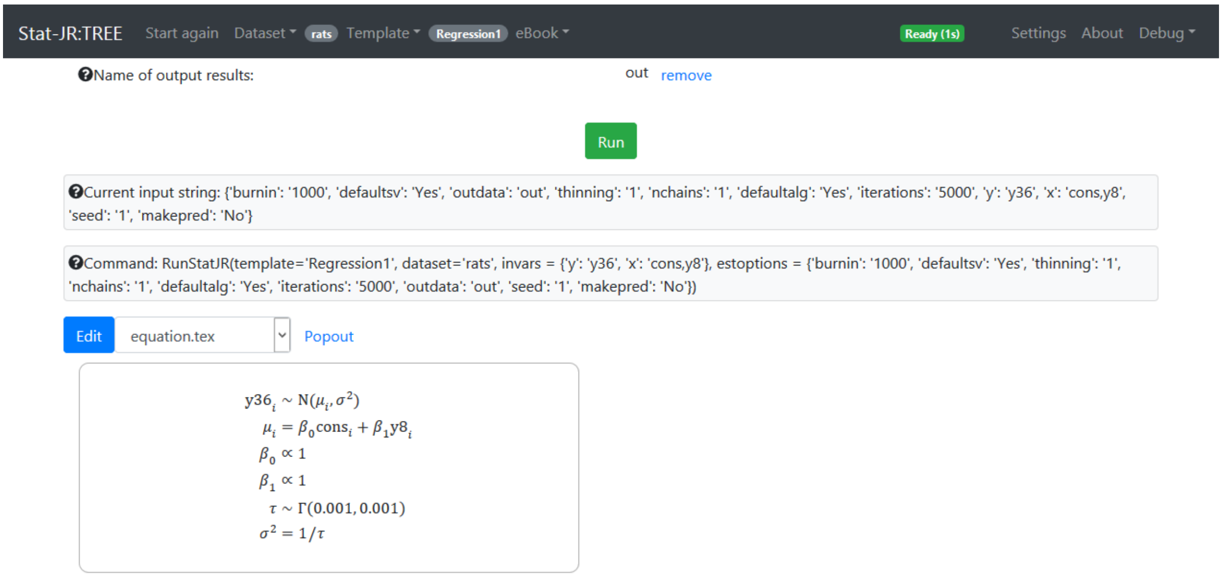
Current Input (Regression1 : rats)

Input string

{'burnin': '1000', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '1', 'defaultalg': 'Yes', 'iterations': '5000', 'y': 'y36', 'x': 'cons,y8', 'seed': '1', 'makepred': 'No'}

Returning to the main window, the second text string (labelled **Command**) can be used by the command-driven version of Stat-JR to perform the same operations.

Clicking on the **Next** button will now pre-process the template inputs; this will result in the following new pane at the bottom of the window:



Name of output results: out

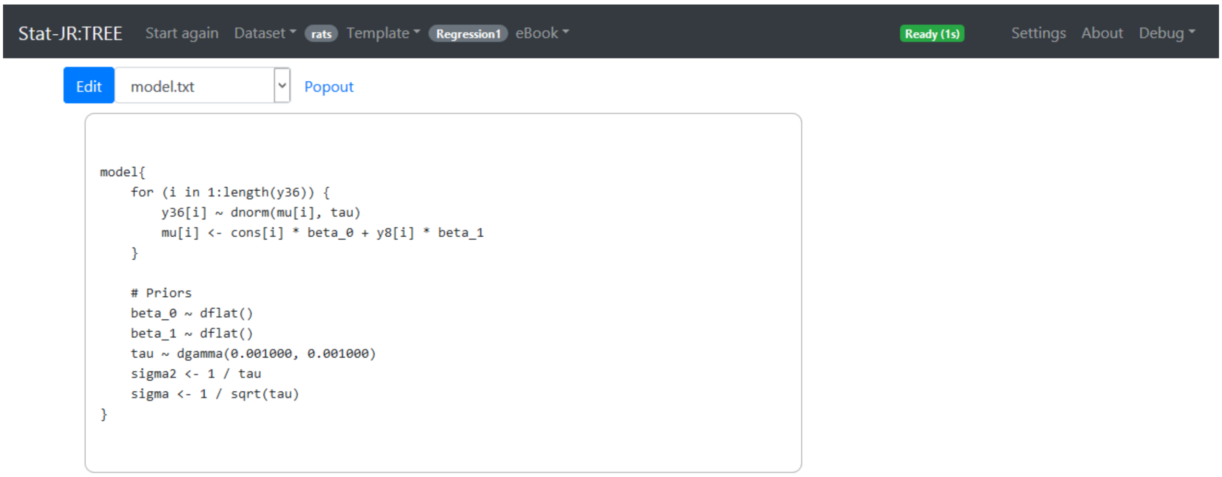
Current input string: {'burnin': '1000', 'defaultsv': 'Yes', 'outdata': 'out', 'thinning': '1', 'nchains': '1', 'defaultalg': 'Yes', 'iterations': '5000', 'y': 'y36', 'x': 'cons,y8', 'seed': '1', 'makepred': 'No'}

Command: RunStatJR(template='Regression1', dataset='rats', invars = {'y': 'y36', 'x': 'cons,y8'}, estoptions = {'burnin': '1000', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '1', 'defaultalg': 'Yes', 'iterations': '5000', 'outdata': 'out', 'seed': '1', 'makepred': 'No'})

equation.tex

$$\begin{aligned}
\mathrm{y36}_i &\sim \mathrm{N}(\mu_i, \sigma^2)\\
\mu_i &= \beta_0 \mathrm{cons}_i + \beta_1 \mathrm{y8}_i\\
\beta_0 &\propto 1\\
\beta_1 &\propto 1\\
\tau &\sim \Gamma(0.001, 0.001)\\
\sigma^2 &= 1/\tau
\end{aligned}$$

The object currently specified in the pull-down list (*equation.tex* is selected by default here) appears in the pane below it. These objects are any outputs constructed by Stat-JR before and during the execution of the template, so here we see a nice mathematical description of the model. If we now select the object *model.txt* from the list we see a description of the regression model that we wish to fit in the language that is used by the eStat engine:



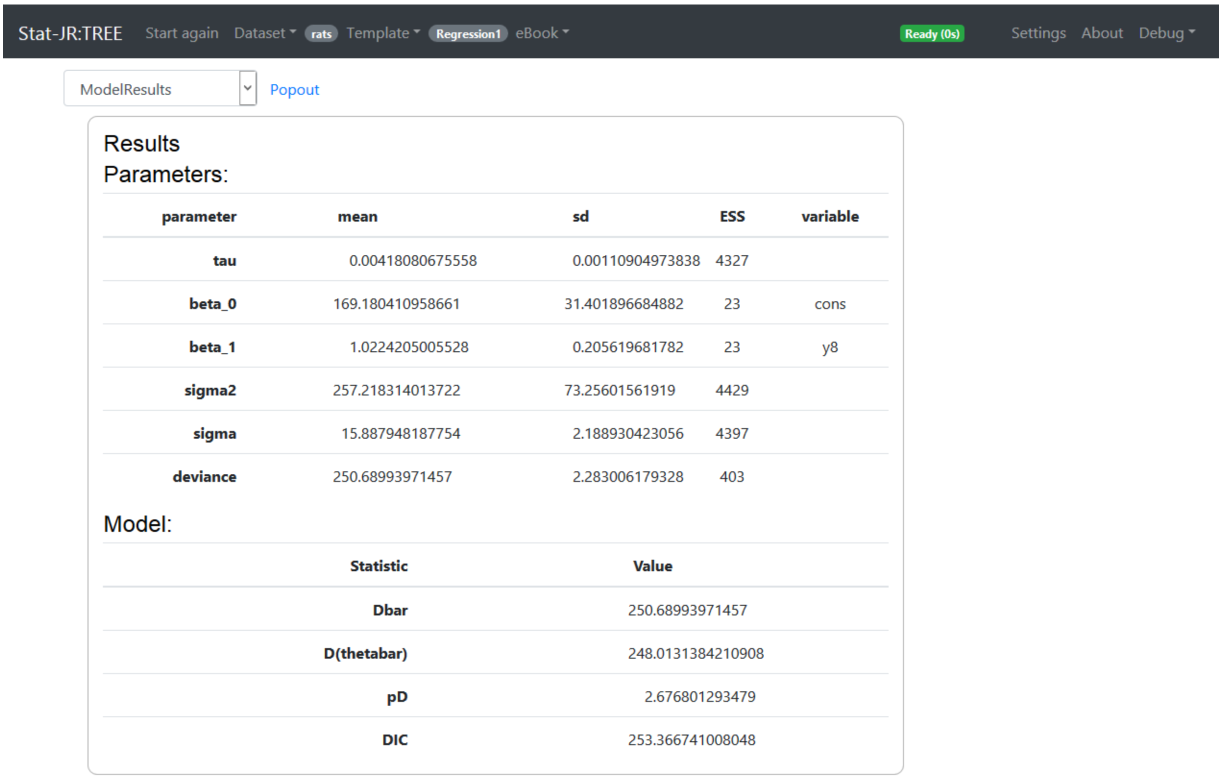
```
model{
    for (i in 1:length(y36)) {
        y36[i] ~ dnorm(mu[i], tau)
        mu[i] <- cons[i] * beta_0 + y8[i] * beta_1
    }

    # Priors
    beta_0 ~ dflat()
    beta_1 ~ dflat()
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    sigma <- 1 / sqrt(tau)
}
```

At this point we haven't actually run the template, and so the objects that can be selected from the pull-down list are those present pre-model run, and include computer code to actually fit the model.

Click the **Run** button to run the template.

Once the progress gauge, towards the right of the black title bar, has changed from "Working" (blue) to "Ready" (green), select *ModelResults* from the pull-down list.

The screen will then look as follows:



Results
Parameters:

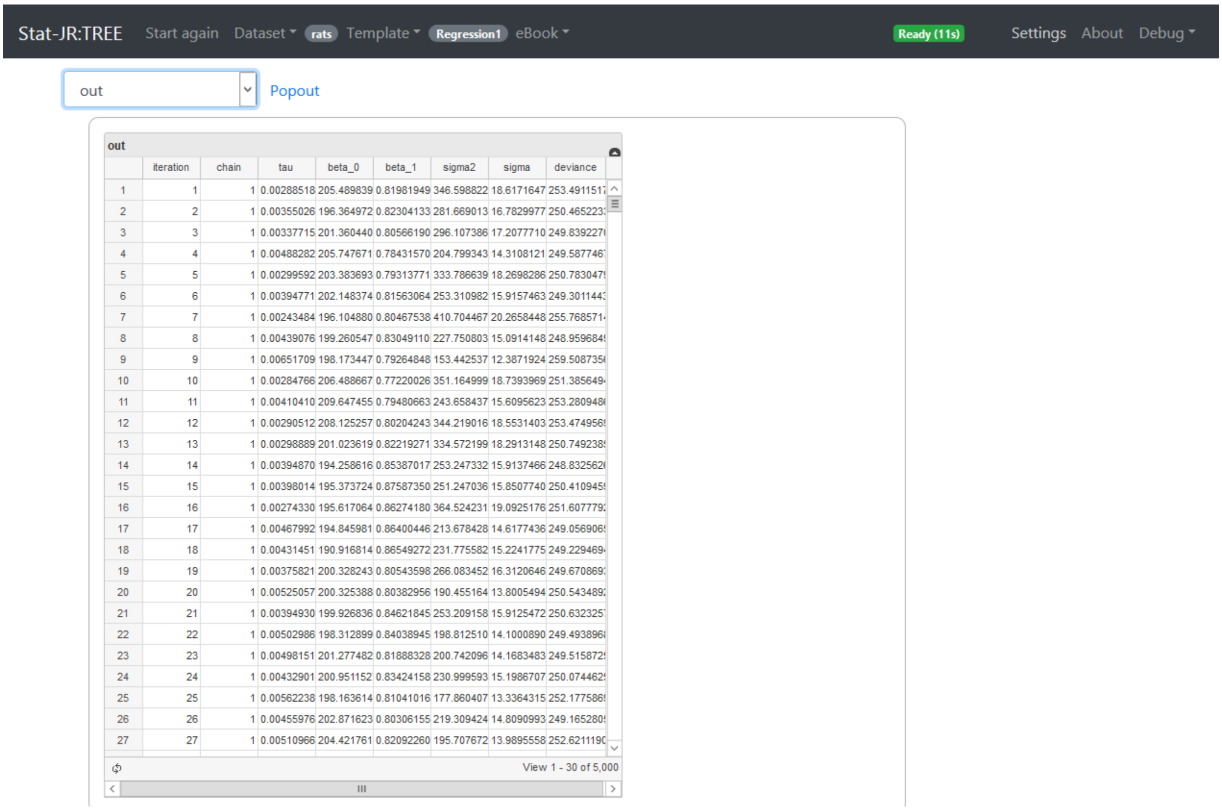
parameter	mean	sd	ESS	variable
tau	0.00418080675558	0.00110904973838	4327	
beta_0	169.180410958661	31.401896684882	23	cons
beta_1	1.0224205005528	0.205619681782	23	y8
sigma2	257.218314013722	73.25601561919	4429	
sigma	15.887948187754	2.188930423056	4397	
deviance	250.68993971457	2.283006179328	403	

Model:

Statistic	Value
Dbar	250.68993971457
D(thetabar)	248.0131384210908
pD	2.676801293479
DIC	253.366741008048

Here we see parameter estimates, along with standard deviations (SDs) as a measure of precision for each parameter. We will explain these further in the next section. At the top of the screen shot above (which is in fact the middle of the full window, vertically-speaking) we now have a few additional buttons. The **Extra Iterations** box, along with the **More** button, will allow us to run for longer (i.e. for a number of iterations additional to those we have already run for). The **Download** button will produce a zipped file that contains a folder with files for many of the objects contained in the pull-down list. Finally, the **Make workflow** button relates to functionality supporting the workflow interface, *LEAF (Logging and Execution of Analysis Flows)*, released with Stat-JR v.1.0.5.

You'll recall that we earlier named the output results *out*, so if we choose this from the pull-down list just above the output pane, we'll be able to view it, as follows:



out

	iteration	chain	tau	beta_0	beta_1	sigma2	sigma	deviance
1	1	1	0.00288518	205.489839	0.81981949	346.598822	18.6171647	253.491151
2	2	1	0.00355026	196.364972	0.82304133	281.669013	16.7829977	250.465223
3	3	1	0.00337715	201.360440	0.80566190	296.107386	17.2077710	249.839227
4	4	1	0.00488282	205.747671	0.78431570	204.799343	14.3108121	249.587746
5	5	1	0.00299592	203.383693	0.79313771	333.786639	18.2698286	250.783047
6	6	1	0.00394771	202.148374	0.81563064	253.310982	15.9157463	249.301144
7	7	1	0.00243484	196.104880	0.80467538	410.704467	20.2658448	255.768571
8	8	1	0.00439076	199.260547	0.83049110	227.750803	15.0914148	248.959684
9	9	1	0.00651709	198.173447	0.79264848	153.442537	12.3871924	259.508735
10	10	1	0.00284766	206.488667	0.77220026	351.164999	18.7393969	251.385649
11	11	1	0.00410410	209.647455	0.79480663	243.658437	15.6095623	253.280948
12	12	1	0.00290512	208.125257	0.80204243	344.219016	18.5531403	253.474956
13	13	1	0.00298889	201.023619	0.82219271	334.572199	18.2913148	250.749238
14	14	1	0.00394870	194.258616	0.85387017	253.247332	15.9137466	248.832562
15	15	1	0.00398014	195.373724	0.87587350	251.247036	15.8507740	250.410945
16	16	1	0.00274330	195.617064	0.86274180	364.524231	19.0925176	251.607779
17	17	1	0.00467992	194.845981	0.86400446	213.678428	14.6177436	249.056906
18	18	1	0.00431451	190.916814	0.86549272	231.775582	15.2241775	249.229469
19	19	1	0.00375821	200.328243	0.80543598	266.083452	16.3120646	249.670869
20	20	1	0.00525057	200.325388	0.80382956	190.455164	13.8005494	250.543489
21	21	1	0.00394930	199.926836	0.84621845	253.209158	15.9125472	250.632325
22	22	1	0.00502986	198.312899	0.84038945	198.812510	14.1000890	249.493896
23	23	1	0.00498151	201.277482	0.81888328	200.742096	14.1683483	249.515872
24	24	1	0.00432901	200.951152	0.83424158	230.999593	15.1986707	250.074462
25	25	1	0.00562238	198.163614	0.81041016	177.860407	13.3364315	252.177586
26	26	1	0.00455976	202.871623	0.80306155	219.309424	14.8090993	249.165280
27	27	1	0.00510966	204.421761	0.82092260	195.707672	13.9895558	252.621119

View 1 - 30 of 5,000

Here we see columns containing the chains of values for each parameter in the model. As well as being able to view this file here, it is also a dataset (stored in temporary memory) and so will appear in the dataset list (at least for the duration of our current session using the software) accessible via the **Dataset** menu in the top title bar (emboldened to indicate that it has been created in this run of the software). This means that we can string template executions together, as we can select *out* as a dataset and perform operations on it using another template.

This ends our whistle-stop tour of many of the windows in Stat-JR. We will next look at a practical application.

3 Application 1: Analysis of the tutorial dataset using the eStat engine

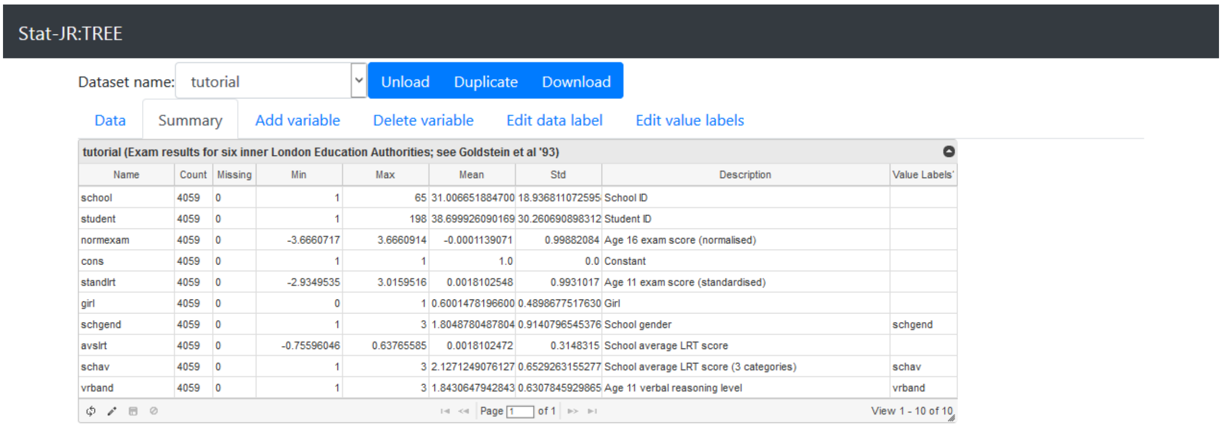
3.1 Summarising the dataset and graphs

In this section we will look at performing some analysis of an example dataset from education. The dataset in question is known as the **tutorial** dataset, and is used as an example in the MLwiN software manuals (see, for example (Browne, MCMC Estimation in MLwiN, v2.36, 2016)). In fact, much of the material here owes a lot to (Browne, MCMC Estimation in MLwiN, v2.36, 2016), which employs similar analysis but using MLwiN (Charlton, Rasbash, Browne, Healy, & Cameron, 2017).

Let us start by looking at the tutorial dataset.

Select **tutorial** via **Dataset > Choose** (see the title bar), then click **Use.**

If you then select **Dataset > View**, and click on the **Summary** tab the following should appear in a new tab in the browser window containing summary information, as follows:



tutorial (Exam results for six inner London Education Authorities; see Goldstein et al '93)

Name	Count	Missing	Min	Max	Mean	Std	Description	Value Labels'
school	4059	0	1	65	31.006651884700	18.936811072595	School ID	
student	4059	0	1	198	38.699926090169	30.260690898312	Student ID	
normexam	4059	0	-3.6660717	3.6660914	-0.0001139071	0.99882084	Age 16 exam score (normalised)	
cons	4059	0	1	1	1.0	0.0	Constant	
standlrt	4059	0	-2.9349535	3.0159516	0.0018102548	0.9931017	Age 11 exam score (standardised)	
girl	4059	0	0	1	0.6001478196600	0.4898677517630	Girl	
schgend	4059	0	1	3	1.8048780487804	0.9140796545376	School gender	schgend
avslrt	4059	0	-0.75596046	0.63765585	0.0018102472	0.3148315	School average LRT score	
schav	4059	0	1	3	2.1271249076127	0.6529263155277	School average LRT score (3 categories)	schav
vrband	4059	0	1	3	1.8430647942843	0.6307845929865	Age 11 verbal reasoning level	vrband

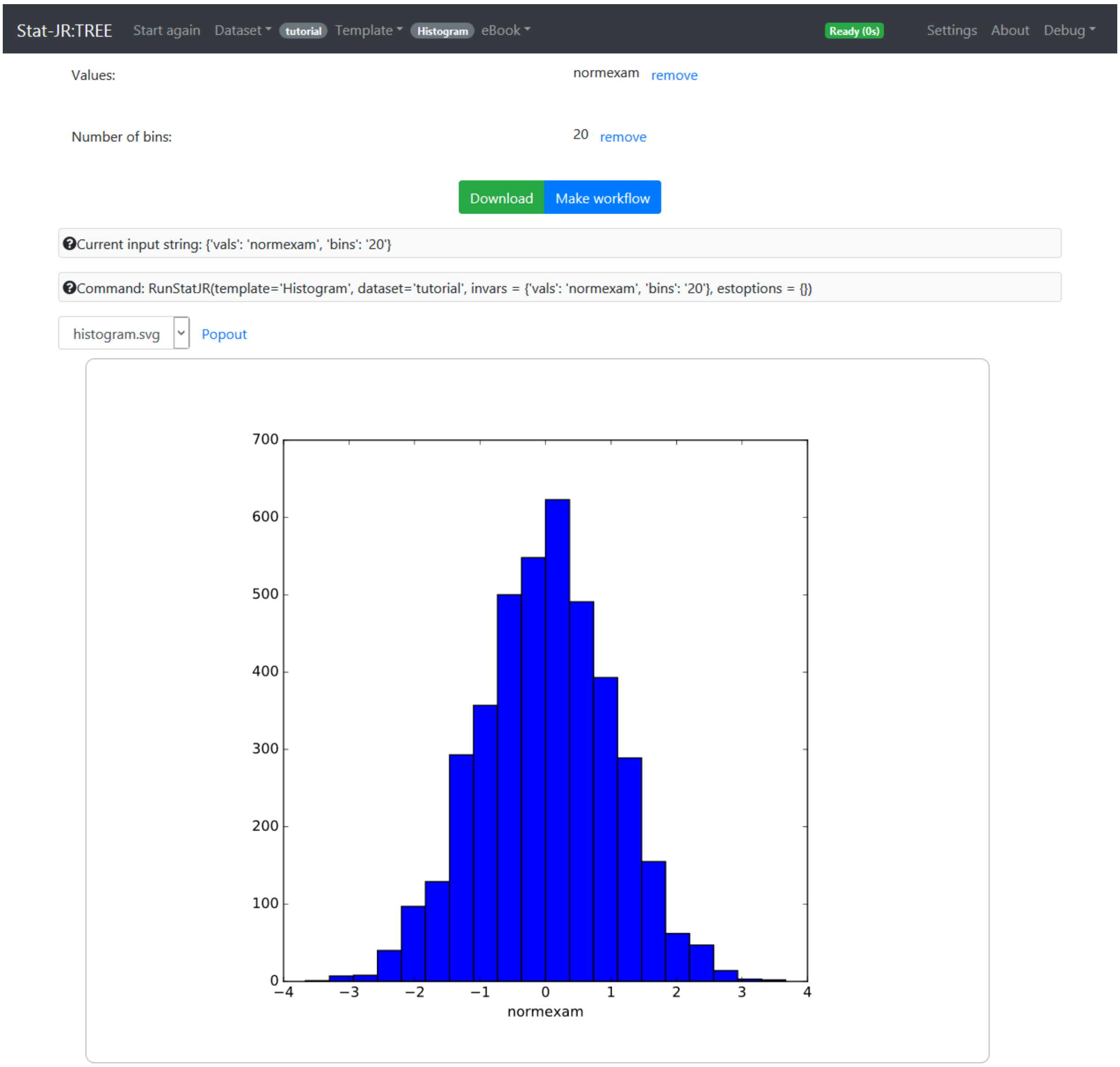
The **tutorial** (Goldstein, et al., 1993) dataset contains data on exam scores of 4059 secondary school children from 65 schools at age 16. These exam scores have been normalised to have a mean of zero and a standard deviation of one and are named *normexam*. There are several predictor variables, including a (standardised) reading test (*standlrt*) taken at age 11, the pupils gender (*girl*), and the school's gender (*schgend*) which takes values 1 for mixed, 2 for boys and 3 for girls. Each variable is described in the **Description** column and if you hover over any of value label names that appear in the **Value Labels** column, the category labels will be displayed.

We can explore the dataset in more detail, prior to fitting any models, by using the many data manipulation templates available in Stat-JR. We will first look at some plots of the data by going back to the main tab and doing the following:

Select **Template > Choose** and then select **Histogram** from the template list that appears and click **Use.**

Fill in the inputs as shown below and click **Next** and then **Run** and select *histogram.svg* from the output list**.**

Dataset: *tutorial;* **Template:** *Histogram;* **Input string:** *{'vals': 'normexam', 'bins': '20'}*



Values: normexam

Number of bins: 20

Current input string: {'vals': 'normexam', 'bins': '20'}

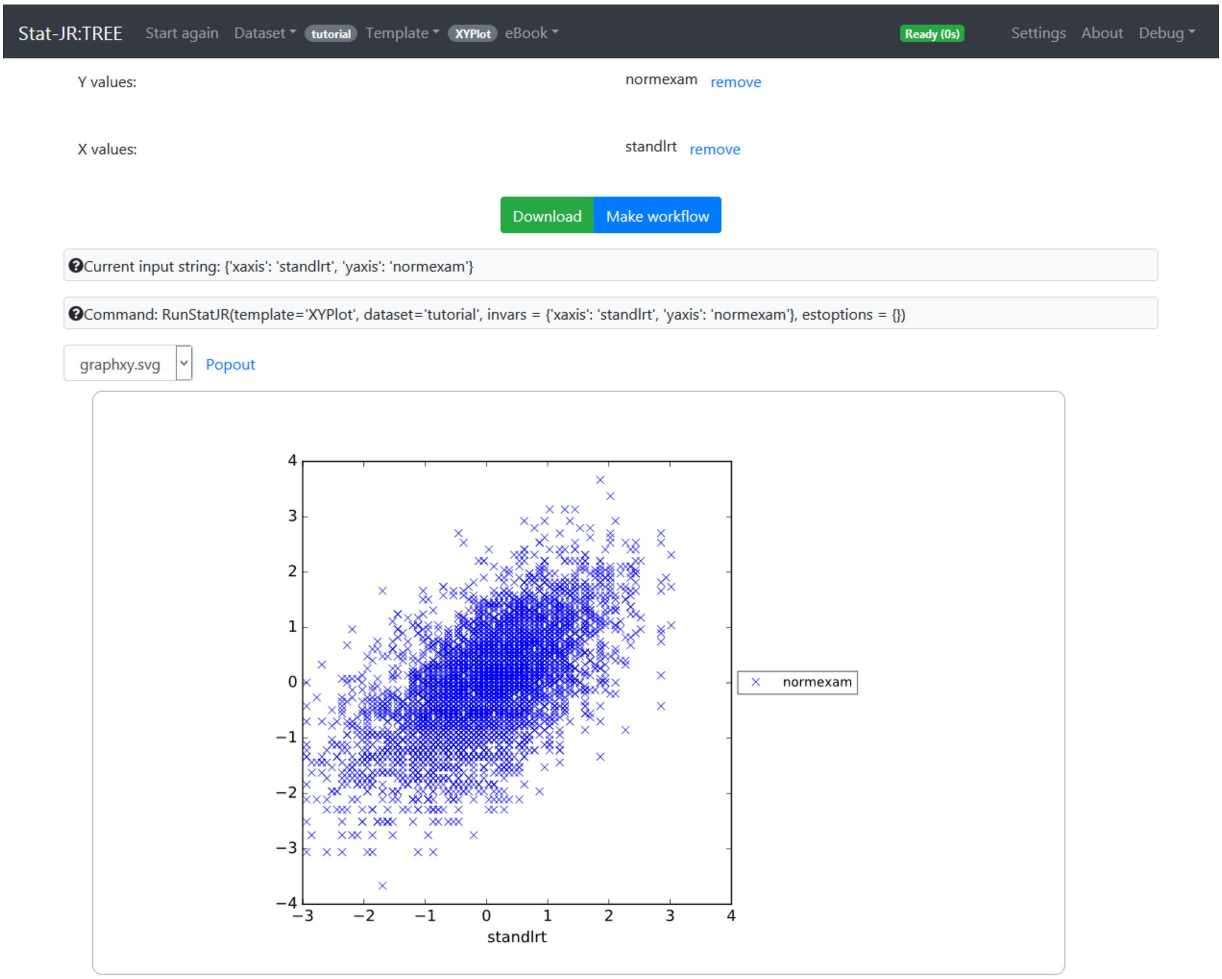
Command: RunStatJR(template='Histogram', dataset='tutorial', invars = {'vals': 'normexam', 'bins': '20'}, estoptions = {})

Here you will see, in the output pane, a histogram plot that shows that the response variable we will model, *normexam*, appears Normally-distributed.

Select **Template > Choose** and this time select **XYPlot** from the template list, then click **Use.**

Fill in the inputs as shown below and click **Next** and then **Run** and select *graphxy.svg* from the list**.**

Dataset: *tutorial;* **Template:** *XYPlot;* **Input string:** *{'xaxis': 'standlrt', 'yaxis': 'normexam'}*



Y values: normexam remove

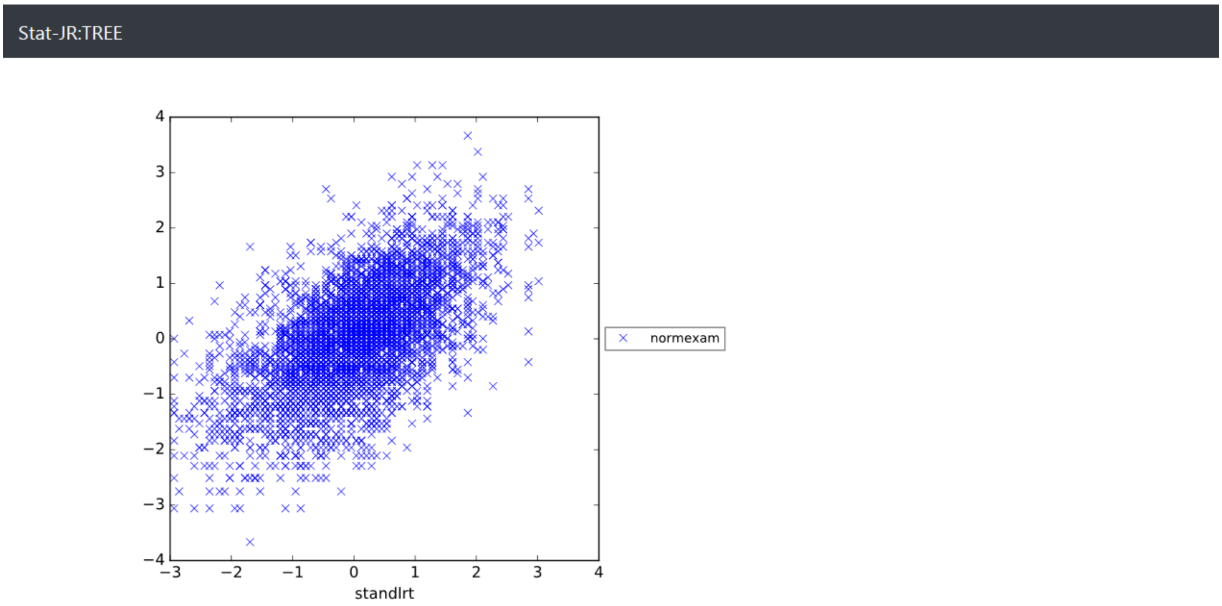
X values: standlrt remove

Current input string: {'xaxis': 'standlrt', 'yaxis': 'normexam'}

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

Here we see that there appears to be a positive relationship between *normexam* and *standlrt*, with pupils that have higher intake scores performing better, on average, at age 16.

We can display the graph in a separate tab in the browser window by clicking on the **Popout** button next to the pull-down list:

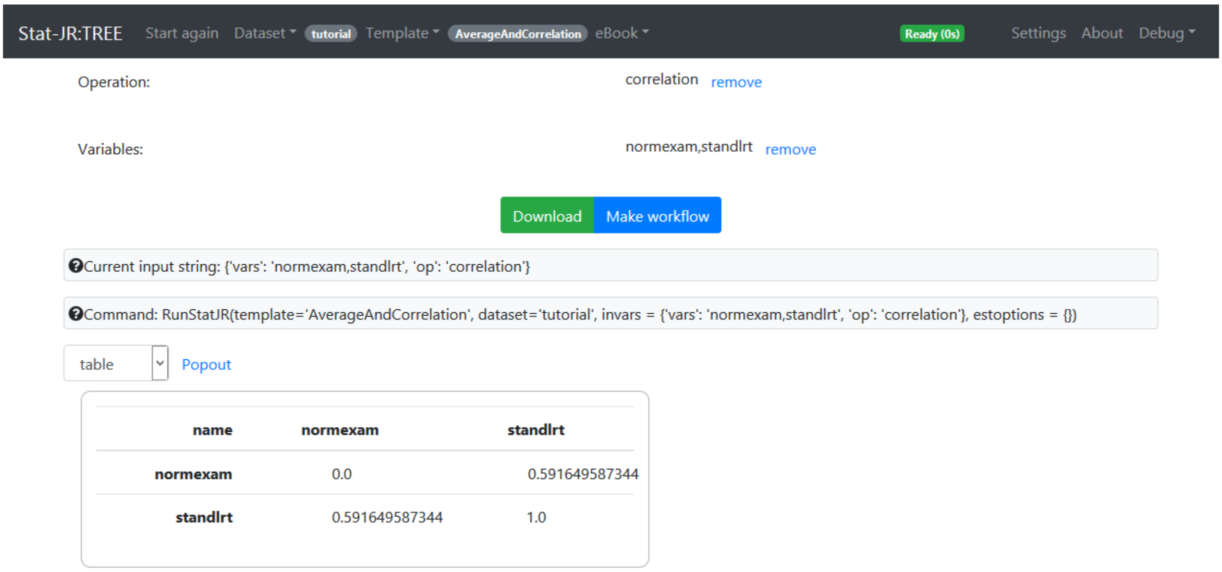


For the sake of brevity, for the remainder of this documentation we will assume you now know how to change template/dataset, and also how to display outputs in separate tabs, so we'll refrain from repeating this information in detail again.

Next, we might like to examine how correlated the two variables, *normexam* and *standlrt*, actually are:

Select **AverageAndCorrelation** as the template, and complete the inputs as follows before clicking on **Next** and **Run** and selecting **table** from the outputs:

Dataset: *tutorial;* **Template:** *AverageAndCorrelation;* **Input string:** *{'vars': 'normexam,standlrt', 'op': 'correlation'}*



Operation: correlation remove

Variables: normexam,standlrt remove

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

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

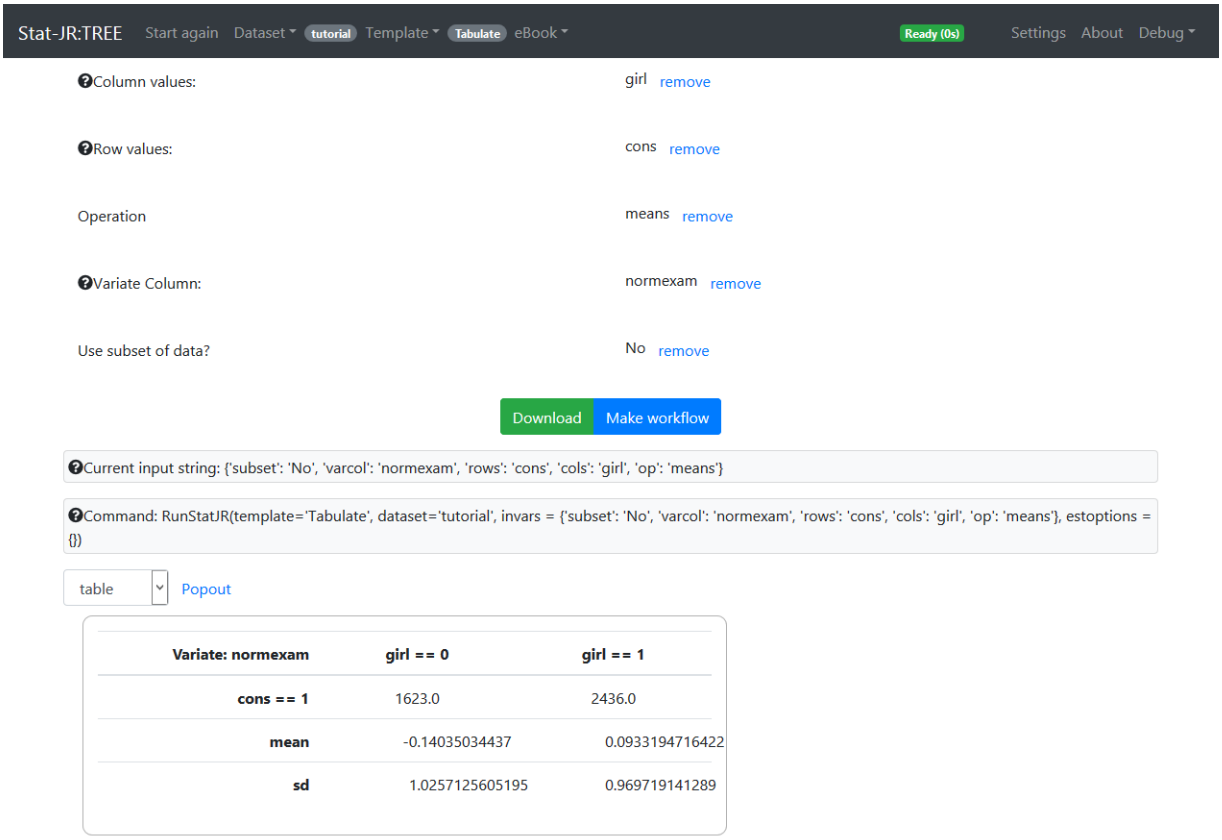
table

name	normexam	standlrt
normexam	0.0	0.591649587344
standlrt	0.591649587344	1.0

Here we see that the correlation is 0.592, so fairly strong and positive. We might also like to look at how exam score varies by gender:

Select **Tabulate** as the template, and complete the inputs as follows, before clicking on **Next** and **Run** and selecting **table** from the output list:

Dataset: *tutorial;* **Template:** *Tabulate;* **Input string:** *{'subset': 'No', 'varcol': 'normexam', 'rows': 'cons', 'cols': 'girl', 'op': 'means'}*



Column values: girl

Row values: cons

Operation: means

Variate Column: normexam

Use subset of data? No

Current input string: {'subset': 'No', 'varcol': 'normexam', 'rows': 'cons', 'cols': 'girl', 'op': 'means'}

Command: RunStatJR(template='Tabulate', dataset='tutorial', invars = {'subset': 'No', 'varcol': 'normexam', 'rows': 'cons', 'cols': 'girl', 'op': 'means'}, estoptions = {})

table

Variate: normexam	girl == 0	girl == 1
cons == 1	1623.0	2436.0
mean	-0.14035034437	0.0933194716422
sd	1.0257125605195	0.969719141289

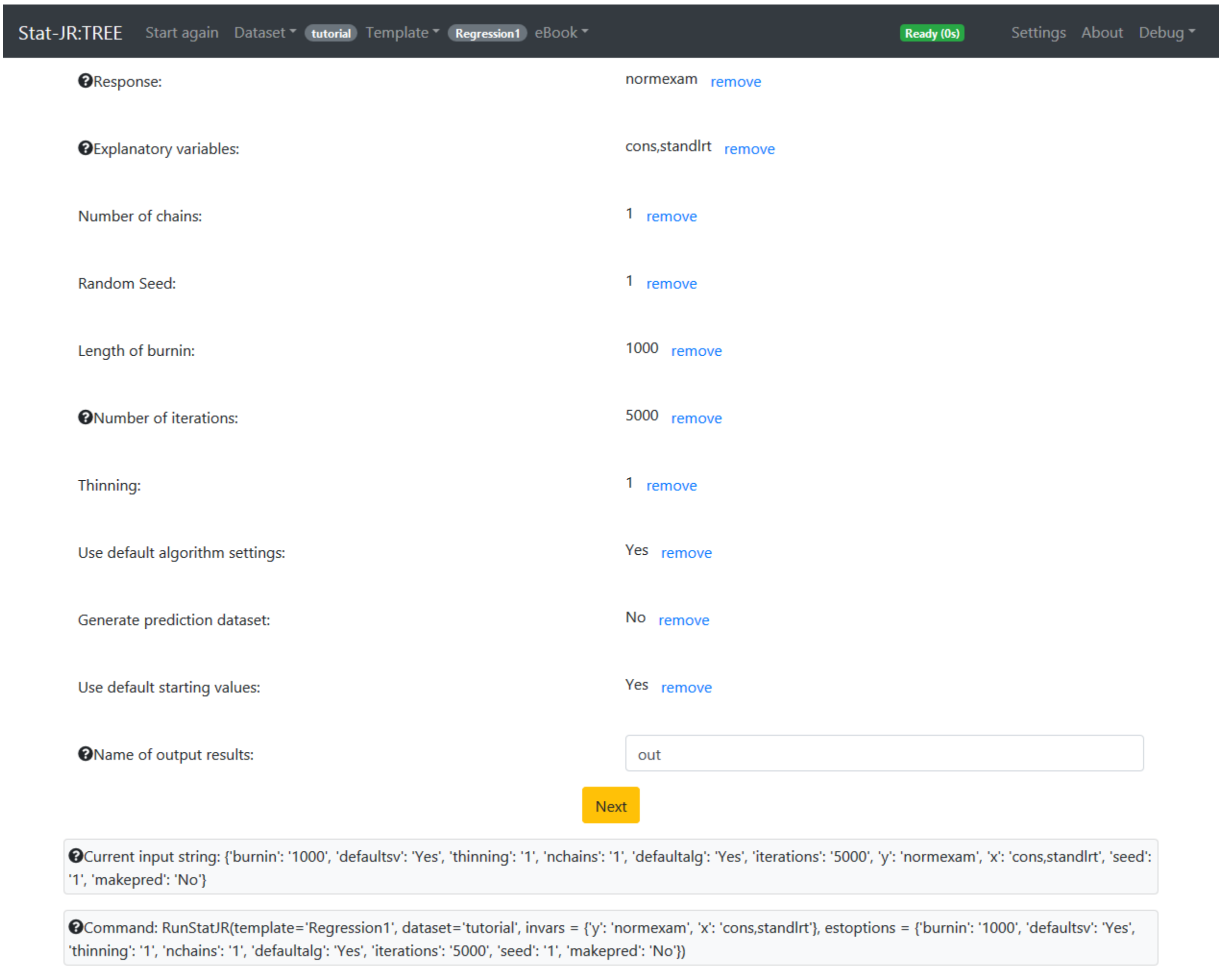
We have to enter variables for column values and row values, and so here we have specified column values as *girl* (taking value 1 for girls and 0 for boys) and row values as *cons* (which is a constant), and then we get 2 columns in the output labelled 0 and 1 for boys and girls, respectively. Looking at the means, it appears that girls do slightly better than boys, and looking at the standard deviations (sds) they are slightly less variable than boys in their scores. Let us now consider performing some statistical modelling on the dataset.

3.2 Single-level Regression

As in the last chapter, with the **rats** dataset, we will start by fitting a simple linear regression model to the **tutorial** dataset. Here we will regress *normexam* on *standlrt* by using a modelling template.

Select **Regression1** as the template and fill it in as follows:

Dataset: *tutorial;* **Template:** *Regression1;* **Input string:** *{'burnin': '1000', 'defaultsv': 'Yes', 'outdata': 'out', 'thinning': '1', 'nchains': '1', 'defaultalg': 'Yes', 'iterations': '5000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1', 'makepred': 'No'}*



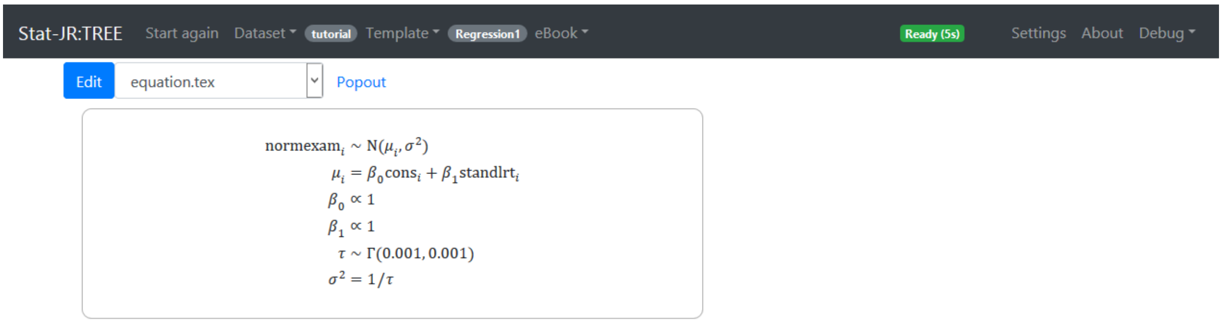
❓Response:	normexam remove
❓Explanatory variables:	cons,standlrt remove
Number of chains:	1 remove
Random Seed:	1 remove
Length of burnin:	1000 remove
❓Number of iterations:	5000 remove
Thinning:	1 remove
Use default algorithm settings:	Yes remove
Generate prediction dataset:	No remove
Use default starting values:	Yes remove
❓Name of output results:	out

❓Current input string: {'burnin': '1000', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '1', 'defaultalg': 'Yes', 'iterations': '5000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1', 'makepred': 'No'}

❓Command: RunStatJR(template='Regression1', dataset='tutorial', invars = {'y': 'normexam', 'x': 'cons,standlrt'}, estoptions = {'burnin': '1000', 'defaultsv': 'Yes', 'thinning': '1', 'nchains': '1', 'defaultalg': 'Yes', 'iterations': '5000', 'seed': '1', 'makepred': 'No'})

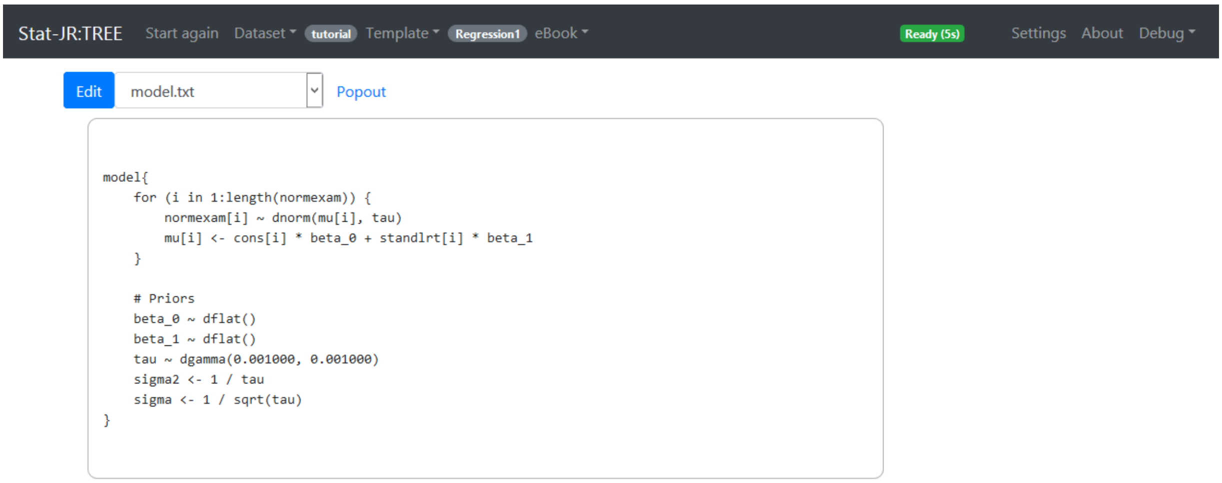
Here we are fitting a linear regression, and so have *standlrt* as an explanatory variable, but also *cons* (which is a column of ones) as we would like to include an intercept as well. For now we have set-up the MCMC estimation options as we did for the **rats** dataset, and we will overwrite the output file *out.*

Clicking on the **Next** button will populate a pull-down list of objects created by Stat-JR at the bottom of the screen and by default we see the object *equation.tex*:



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

In the pane we find a mathematical representation of the chosen model. Note that the file is a LaTeX file that is being rendered in the browser by a piece of software called MathJaX (Cervone, Sorge, Lawson-Perfect, & Krautzberger, 2017), so if you are a LaTeX-user you can copy this file straight into a document. If we instead choose *model.txt* from the list we see the following:



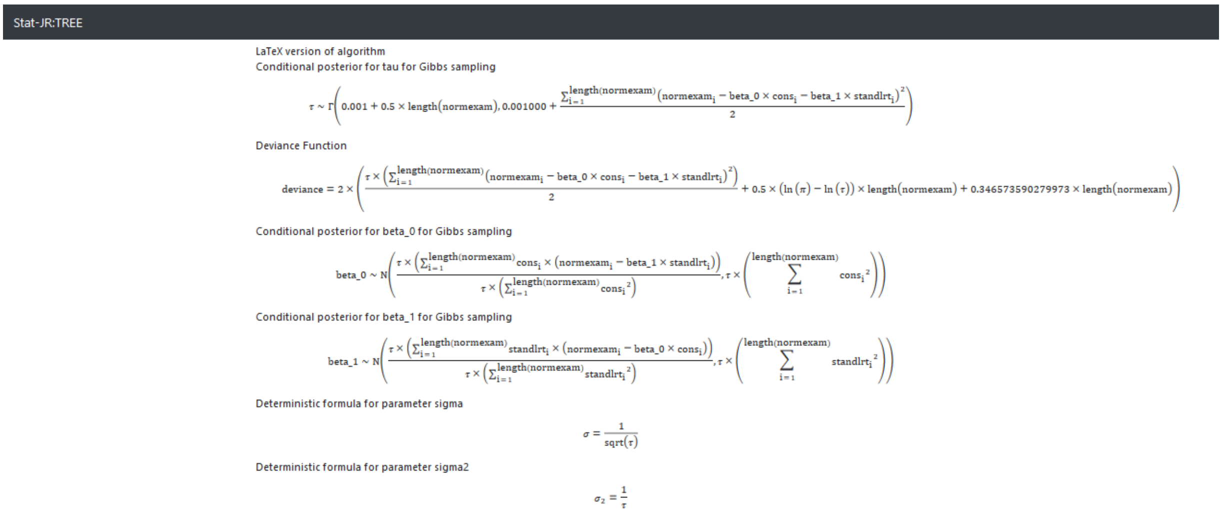
```
model{
    for (i in 1:length(normexam)) {
        normexam[i] ~ dnorm(mu[i], tau)
        mu[i] <- cons[i] * beta_0 + standlrt[i] * beta_1
    }

    # Priors
    beta_0 ~ dflat()
    beta_1 ~ dflat()
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1 / tau
    sigma <- 1 / sqrt(tau)
}
```

Here we see the text file that represents the model we wish to fit in the language that the algebra system used by the built-in eStat engine requires. The **Regression1** template only uses the eStat MCMC-based estimation engine, so as you can see in the mathematical formulae in *equation.tex* we are fitting a Bayesian version of a linear regression, and the last four lines of the output are prior distributions for the unknown parameters, β_0, β_1 and the precision τ (where $\tau=1/\sigma^2$).

Whilst we will keep our description of Bayesian statistics and MCMC estimation to a minimum, and recommend Chapter 1 of (Browne, MCMC Estimation in MLwiN, v2.36, 2016) for more details, in brief we are interested in the joint posterior distribution of all unknown parameters given the data (and the prior distributions specified). In practice, in complex models, this distribution has many dimensions (in our simple regression we have 3 dimensions) and is hard to evaluate analytically. Instead, MCMC algorithms work by simulating random draws from a series of conditional posterior distributions (which can be evaluated). It can then be shown (by some mathematics) that after a period of time (required for the simulations to move from their possibly arbitrary starting point) that the draws will be a dependent sample from the joint posterior distribution of interest. It is common, therefore, to throw away the first *n* draws which are deemed a **burn-in** period.

For the simple linear regression, it is a mathematical exercise to show that the conditional posterior distributions have standard forms and are Normal (for the fixed effect) and Gamma (for the precision = 1 /variance). The eStat engine has a built in algebra system which takes the text file (*model.txt*) and returns the conditional posterior distributions; you can view these as follows:

Select *algorithm.tex* from the list and click on the **Popout** button and the algebra steps will appear in a new tab as follows:



Stat-JR:TREE

LaTeX version of algorithm

Conditional posterior for tau for Gibbs sampling

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

Deviance Function

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

Conditional posterior for beta_0 for Gibbs sampling

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

Conditional posterior for beta_1 for Gibbs sampling

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

Deterministic formula for parameter sigma

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

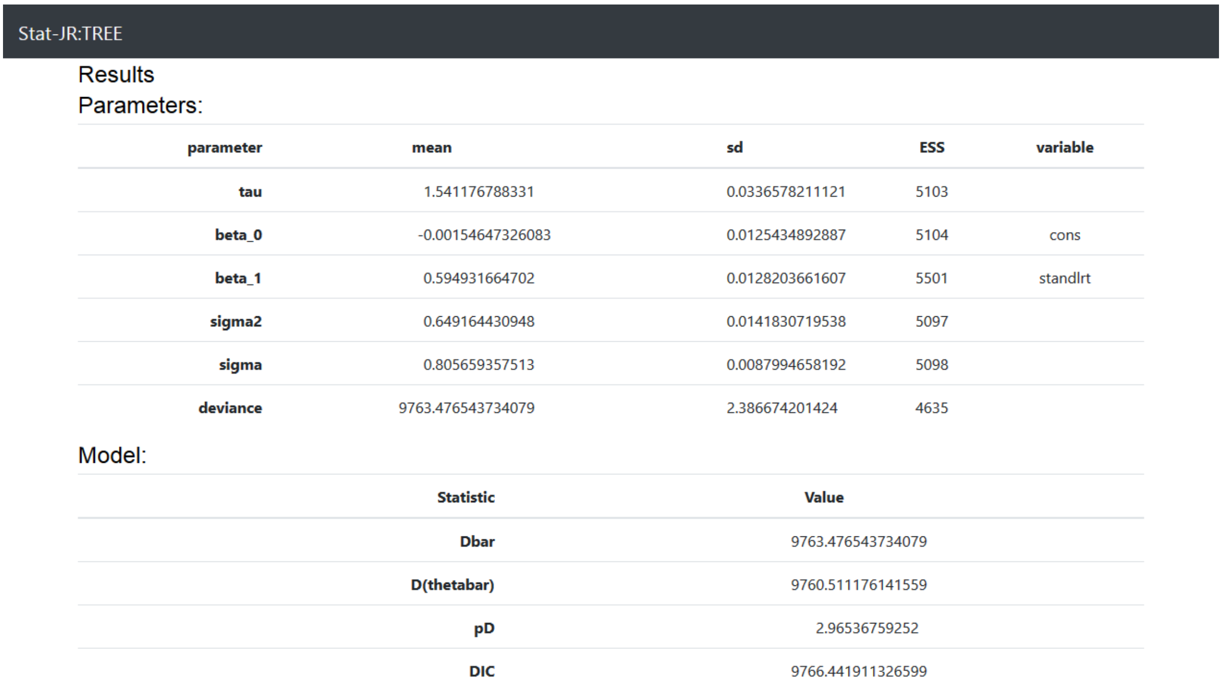
Deterministic formula for parameter sigma2

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

The eStat engine then takes these posterior distributions and wraps them up into computer code (C++ (Stroustrup, 2013)) which it will compile and run for the model. By default this will be several pieces of code that are linked together by Stat-JR, although the **Settings** screen (accessible via a link towards the top of the main menu screen, as we saw earlier) has an option to output completely standalone code that can be taken away and run separately from the Stat-JR system; this is, however, a topic for more advanced users.

Returning to the tab, in the browser window, containing the model template, click on the **Run** button and wait for the model to run.

Then select *ModelResults* from the pull-down list and pop it out into a separate tab.



Stat-JR:TREE

Results

Parameters:

parameter	mean	sd	ESS	variable
tau	1.541176788331	0.0336578211121	5103	
beta_0	-0.00154647326083	0.0125434892887	5104	cons
beta_1	0.594931664702	0.0128203661607	5501	standlrt
sigma2	0.649164430948	0.0141830719538	5097	
sigma	0.805659357513	0.0087994658192	5098	
deviance	9763.476543734079	2.386674201424	4635	

Model:

Statistic	Value
Dbar	9763.476543734079
D(thetabar)	9760.511176141559
pD	2.96536759252
DIC	9766.441911326599

Here the model results can be split into two parts:

The first part of the results (under the heading '*Parameters*') contains the actual parameter estimates. Here, for each parameter, we get 3 numbers: a posterior mean estimate (*mean*), a posterior standard deviation (*sd*), and an effective sample size (*ESS*).

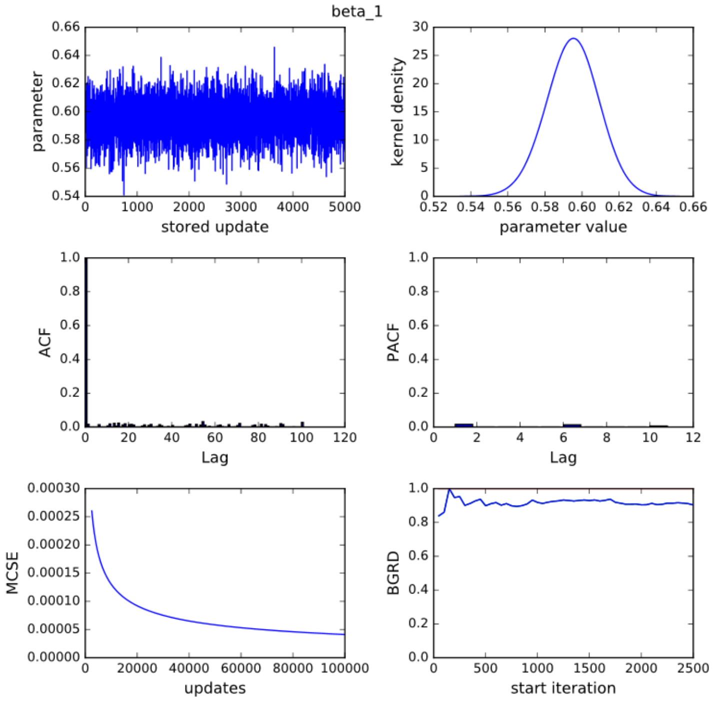
Here we see that beta_0 has a mean estimate of approximately 0, which we would expect as both the response and predictor have been normalised, or standardised. The slope beta_1 has mean 0.595 with standard deviation 0.013, and is highly significant, as its mean estimate is many times its standard deviation (a Bayesian equivalent of a standard error). The value 0.595 represents the average increase in the *normexam* score for a 1-point (1 sd, due to standardising) increase in *standlrt*. The residual variance, sigma2, has value 0.649 showing that, as the initial response variance was 1.0, *standlrt* has explained 35.1% of the variability.

The ESS is a diagnostic which reflects the simulation-based (stochastic) nature of the MCMC estimation procedure: we have based our results on the 5,000 iterations post burn-in, but we know that the method produces dependent samples, and so the ESS gives an equivalent number of independent samples for the parameters involved; in effect a measure of the information content of the chain In this case, all parameters have ESS of > 4000, and so the chains are almost independent.

The second part (under the heading *'Model'*) refers to the model fit for this particular model and the DIC diagnostic (Spiegelhalter, Best, Carlin, & van der Linde, 2002). The DIC diagnostic is an information criterion which is a measure of how good a specific model is, consisting of a combination of how well the model fits the data (here defined by the model deviance) and how complex the model is (here defined by pD: the effective number of parameters). Basically the better fitting the model is, the better the model is, but it has to be penalised by how complex it is. The DIC statistic is defined as the deviance of the mean + 2pD. In this example the deviance at the mean (D(thetabar)) is 9760.5 and pD is ~3 (reflecting the three parameters of the model that are being estimated) and so we have a DIC value of 9766.4. This number is not particularly interesting in isolation but it is when we compare values for several models.

We can also get more information from the diagnostic plots that are available in the list of objects

Return to the model run tab in the browser window, and select *beta_1.svg* from the pull-down list above the output pane and pop it out into a separate tab.



This “*sixway*” plot gives several graphs that are constructed from the chain of 5,000 values produced for *beta_1*. The top-left graph shows the values plotted against iteration number, and is useful to confirm that the chain is ‘mixing well’, meaning that it visits most of the posterior distribution in few iterations. The top-right graph contains a kernel density plot which is like a smoothed histogram and represents the posterior distribution for this parameter. Here the shape is symmetric and looks like a Normal distribution which we expect given theory for fixed effects in a normal model.

The two graphs in the middle row are time series plots known as the autocorrelation (ACF) and partial autocorrelation (PACF) functions. The ACF indicates the level of correlation within the chain; this is calculated by moving the chain by a number of iterations (called the lag) and looking at the correlation between this shifted chain and the original. In this case, the autocorrelation is very small for all lags. The PACF picks up the degree of auto-regression in the chain. By definition a Markov chain should act like an autoregressive process of order 1, as the Markov definition is that the future state of the chain is independent of all the past states of the chain given the current value. If, for example, in reality the chain had additional dependence on the past 2 values, then we would see a significant PACF at lag 2. In this case all PACF values are negligible. All of this suggests that we have good mixing and it would be appropriate to proceed to the interpretation of the parameters.

The bottom-left plot is the estimated Monte Carlo standard error (MCSE) plot for the posterior estimate of the mean. As MCMC is a simulation-based approach this induces (Monte Carlo) uncertainty due to the random numbers it uses. This uncertainty reduces with more iterations, and is measured by the MCSE, and so this graph details how long the chain needs to be run to achieve a specific MCSE. The sixth (bottom-right) plot is a multiple chains diagnostic and doesn’t make much sense when we have run only one chain, and we will therefore consider running multiple chains in the next section.

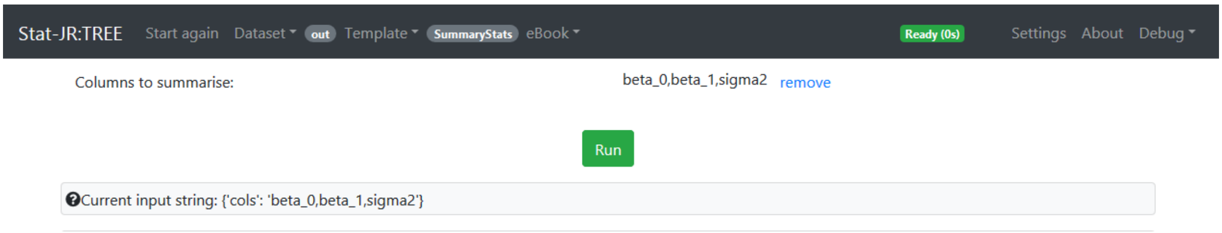
We can also get some other diagnostics and summary statistics for the model as follows:

Click on the **Template** pull-down list at the top of the screen and select **Choose** and **SummaryStats** as the template.

Next click on the **Dataset** pull-down list and select **Choose** and **out** as the dataset.

Run the **SummaryStats** template and select the inputs as follows before clicking on **Run**:

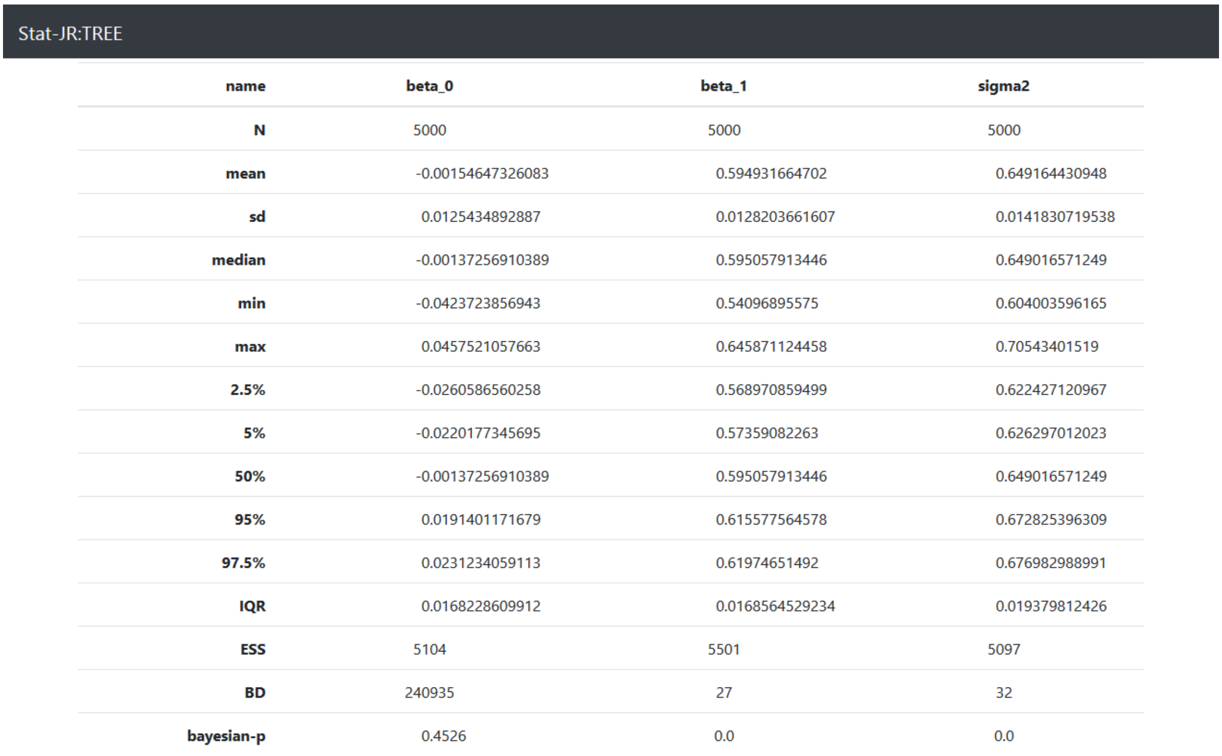
Dataset: *out;* **Template:** *SummaryStats;* **Input string:** *{'cols': 'beta_0,beta_1,sigma2'}*



Columns to summarise: beta_0,beta_1,sigma2

Current input string: {'cols': 'beta_0,beta_1,sigma2'}

Press **Run**, and then select **table** from the drop-down list of outputs, and display it in a separate tab:



name	beta_0	beta_1	sigma2
N	5000	5000	5000
mean	-0.00154647326083	0.594931664702	0.649164430948
sd	0.0125434892887	0.0128203661607	0.0141830719538
median	-0.00137256910389	0.595057913446	0.649016571249
min	-0.0423723856943	0.54096895575	0.604003596165
max	0.0457521057663	0.645871124458	0.70543401519
2.5%	-0.0260586560258	0.568970859499	0.622427120967
5%	-0.0220177345695	0.57359082263	0.626297012023
50%	-0.00137256910389	0.595057913446	0.649016571249
95%	0.0191401171679	0.615577564578	0.672825396309
97.5%	0.0231234059113	0.61974651492	0.676982988991
IQR	0.0168228609912	0.0168564529234	0.019379812426
ESS	5104	5501	5097
BD	240935	27	32
bayesian-p	0.4526	0.0	0.0

Here we see a more extensive summary of the three parameters of interest. This summary table includes various quantiles of the distribution which are calculated by sorting the chain and picking the values that lie *x*% into the sorted chain (where x is 2.5, 5, 50 etc.). These allow for accurate interval estimates that do not rely on a Normal distribution assumption. The inter-quartile range (IQR) is similarly calculated by picking the values that lie 25% and 75% through the sorted list and calculating the distance between them.

The final 3 statistics relate to the MCMC method. The ESS as described earlier gives an idea of how many independent iterations the MCMC chain for each parameter is equivalent to. The Brooks-

Draper diagnostic is based on measuring the mean estimate to a particular accuracy (number of significant figures set to 2 by default). For example, it states that to quote *sigma2* as 0.65 with some desired accuracy only requires 32 iterations. The anomaly here is *beta_0*, however, since the true value is 0 we have difficulty quoting such a value to 2 significant figures! Finally the bayesian-p is a P value for each parameter. Here the number gives the percentage of values in the chain that are less than 0. This give an idea of how significantly different from 0 the parameter is.

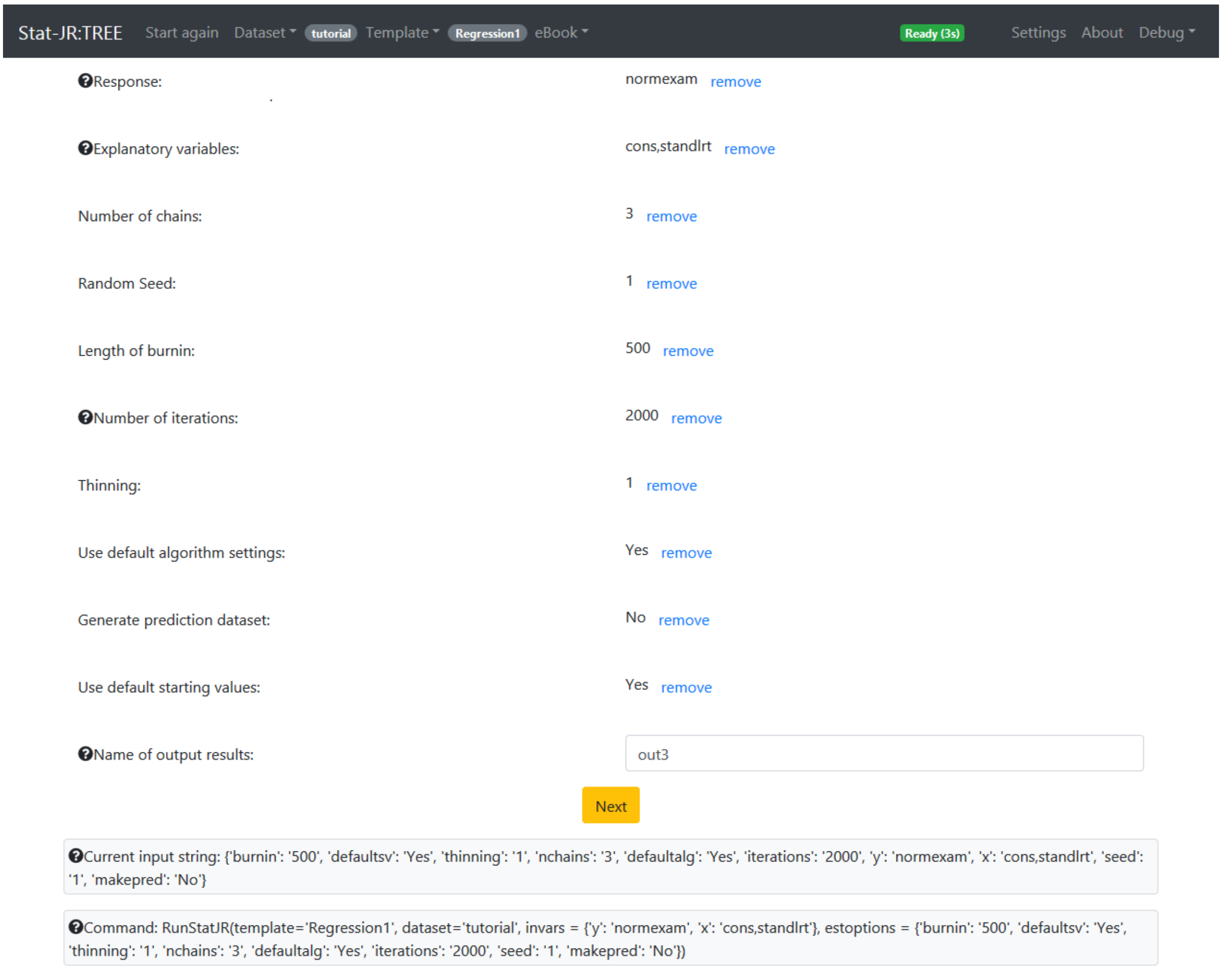
3.3 Multiple chains

MCMC methods are more complicated to deal with than classical methods as we have to specify many estimation parameters, including how long to run the MCMC chains for. The idea of running chains for a longer period is to counteract the fact that the chains are serially-correlated, and therefore are not independent samples from the distribution. Another issue that might cause problems is that the posterior distribution of interest may have several possible maxima (i.e. may be multimodal). This is not usually an issue in the models we cover in this book, but it is still a good idea to start off the estimation procedure from several places, or with several runs with different random number seeds, to confirm we get the same answers.

From the top bar change **Template** and **Dataset** using the respective pull-down lists and **Choose** so you have **Regression1** as the template and **tutorial** as the dataset.

This time fill in the screen as follows:

Dataset: *tutorial;* **Template:** *Regression1;* **Input string:** *{'burnin': '500', 'defaultsv': 'Yes', 'outdata': 'out3', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1', 'makepred': 'No'}*



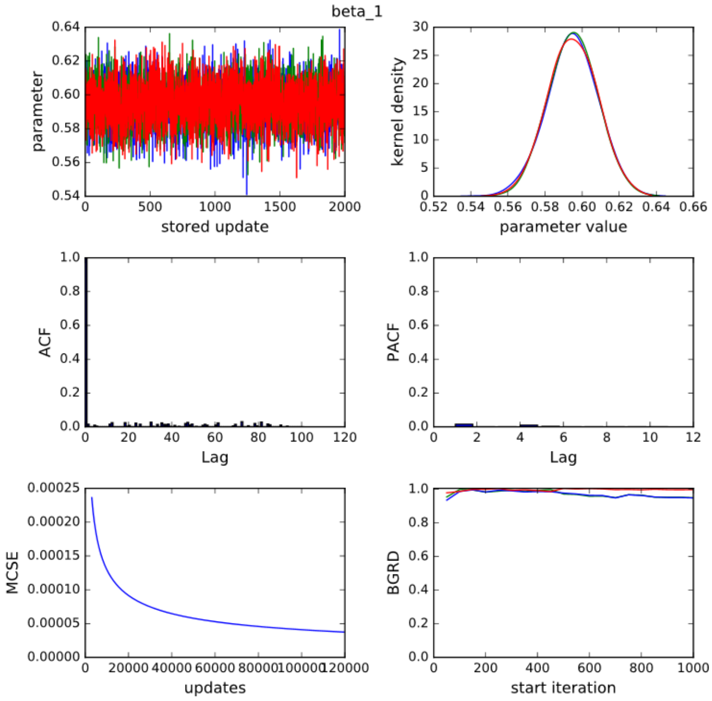
❓Response:	normexam remove
❓Explanatory variables:	cons,standlrt remove
Number of chains:	3 remove
Random Seed:	1 remove
Length of burnin:	500 remove
❓Number of iterations:	2000 remove
Thinning:	1 remove
Use default algorithm settings:	Yes remove
Generate prediction dataset:	No remove
Use default starting values:	Yes remove
❓Name of output results:	out3

❓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'})

Click on the **Next** and **Run** buttons.

When the model has run select *beta_1.svg* from the outputs list and pop it out to view it in a new tab.



Here we see the three chains superimposed on each other in the top-left pane – note the chain looks primarily red simply because this chain (chain 3) has been plotted on top of the other two, and due to good mixing obscures them. Each chain has its own kernel plot in the top-right pane and this also suggests that, by the similarity of shape and position, the chains are mixing well.

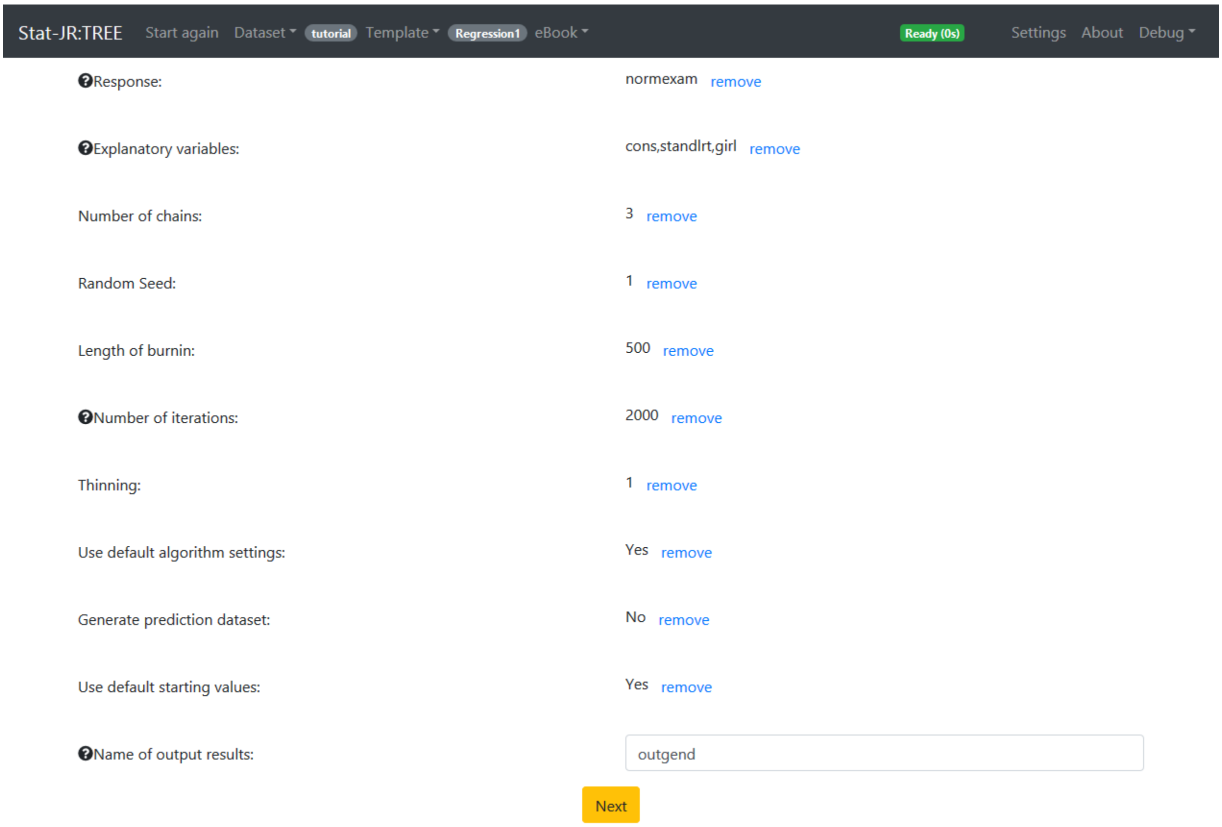
We have previously described what all the graphs here mean in Section 3.2, apart from the Brooks-Gelman-Rubin diagnostic plot (see (Brooks & Gelman, 1998)) in the bottom-right corner. This plot looks at mixing across the chains: the green and blue lines measure variability between and within the chains, and the red is their ratio. For good convergence this red line should be close to 1.0, and here the values get close to 1.0 fairly quickly. We can have a lot of faith in the estimates of our model.

3.4 Adding gender to the model

We have so far been more focused on understanding the MCMC methods but now we will return to modelling. We next wish to look at whether gender has an additional effect on *normexam* on top of that we have observed for intake score (*standlrt*).

To do this, click on the remove link next to explanatory variables in the browser window, and fill-in the template as follows:

Dataset: *tutorial;* **Template:** *Regression1;* **Input string**: *{'burnin': '500', 'defaultsv': 'Yes', 'outdata': 'outgend', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt,girl', 'seed': '1', 'makepred': 'No'}*



Response: normexam

Explanatory variables: cons,standlrt,girl

Number of chains: 3

Random Seed: 1

Length of burnin: 500

Number of iterations: 2000

Thinning: 1

Use default algorithm settings: Yes

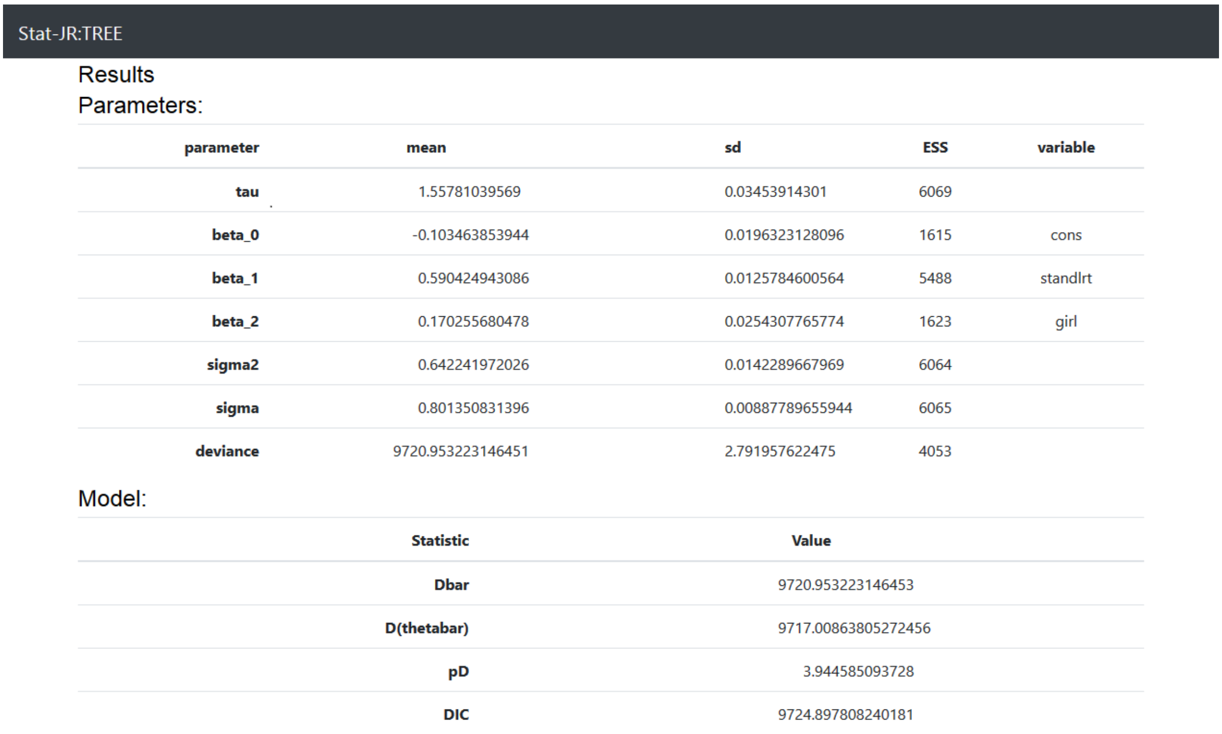
Generate prediction dataset: No

Use default starting values: Yes

Name of output results: outgend

Click on **Next** and then **Run** to run the model.

After the model finishes running select *ModelResults* from the drop-down list of outputs, and display in a new tab.



Results

Parameters:

parameter	mean	sd	ESS	variable
tau	1.55781039569	0.03453914301	6069	
beta_0	-0.103463853944	0.0196323128096	1615	cons
beta_1	0.590424943086	0.0125784600564	5488	standlrt
beta_2	0.170255680478	0.0254307765774	1623	girl
sigma2	0.642241972026	0.0142289667969	6064	
sigma	0.801350831396	0.00887789655944	6065	
deviance	9720.953223146451	2.791957622475	4053	

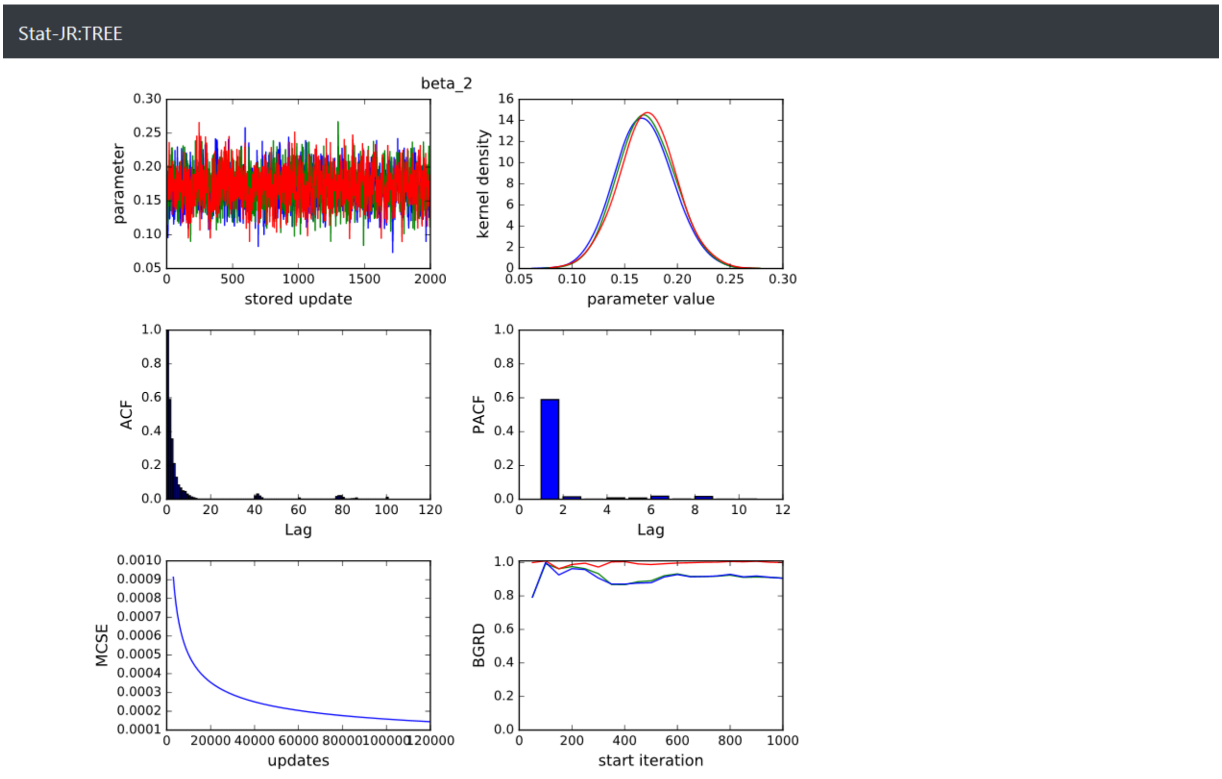
Model:

Statistic	Value
Dbar	9720.953223146453
D(thetabar)	9717.00863805272456
pD	3.944585093728
DIC	9724.897808240181

This new model has one additional fixed effect parameter (beta_2) associated with gender, and we see it has a positive effect (0.170) which appears highly-significant (at least twice its standard deviation, which is 0.025). Note that in our earlier tabulation we saw that the difference in gender means was 0.093- (-0.140) = 0.233 and so the effect here is somewhat smaller, probably due to correlation between gender and intake score.

Looking at the DIC diagnostic to assess whether this model is better we see this has dropped from 9766.4 to 9724.9, which is a big drop, and so the model with gender is indeed much better.

Finally we see that the ESS for two of the parameters is lower (beta_0 and beta_2), at around 1600, so the model doesn't mix quite as well; however, these ESS are still large enough not to require further iterations. Here is the graph for **beta_2.svg**, displayed in a new tab:



We see reasonable mixing, and can clearly see the significance of the effect as well (as the kernel density plot in the top-right corner indicates that 0 is nowhere near the posterior distribution). From a modelling perspective we have thus far ignored the fact that our data is a two-stage sample and that we should account for the clustering of the pupils within secondary schools. To do this we need to fit a 2-level model, and use a different template.

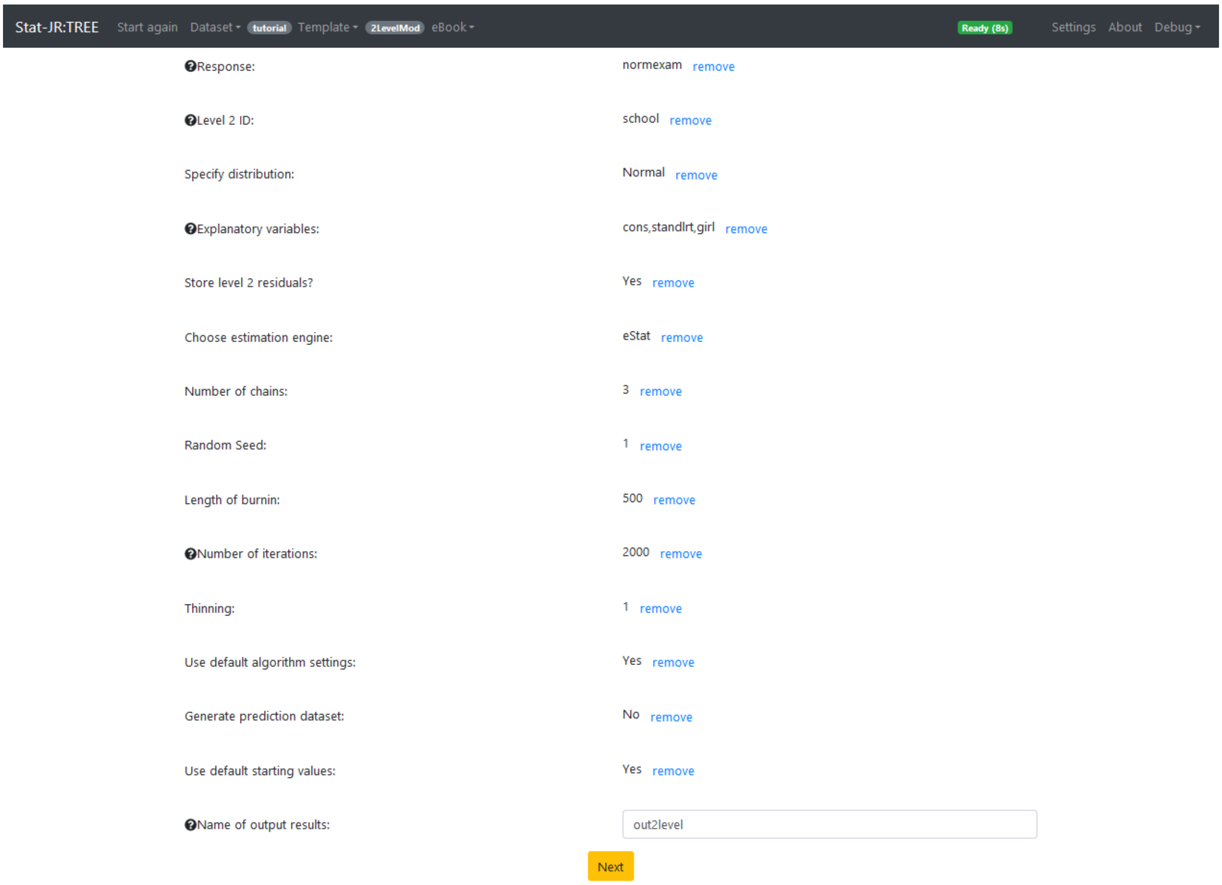
3.5 Including school effects

Stat-JR contains many different model-fitting templates some of which can fit whole families of models and some of which can fit just one or two specific models. We have thus far looked at the rather restrictive **Regression1** template that only fits single level normal response models. To include school effects we will now look at the **2LevelMod** template, which not only includes a set of random effects but also supports different response types and estimation engines, features that we will look at later.

On the **Template** pull-down list at the top of the screen select **Choose** and select **2LevelMod** as the template and stick with **tutorial** for the dataset.

Set-up the inputs as shown below:

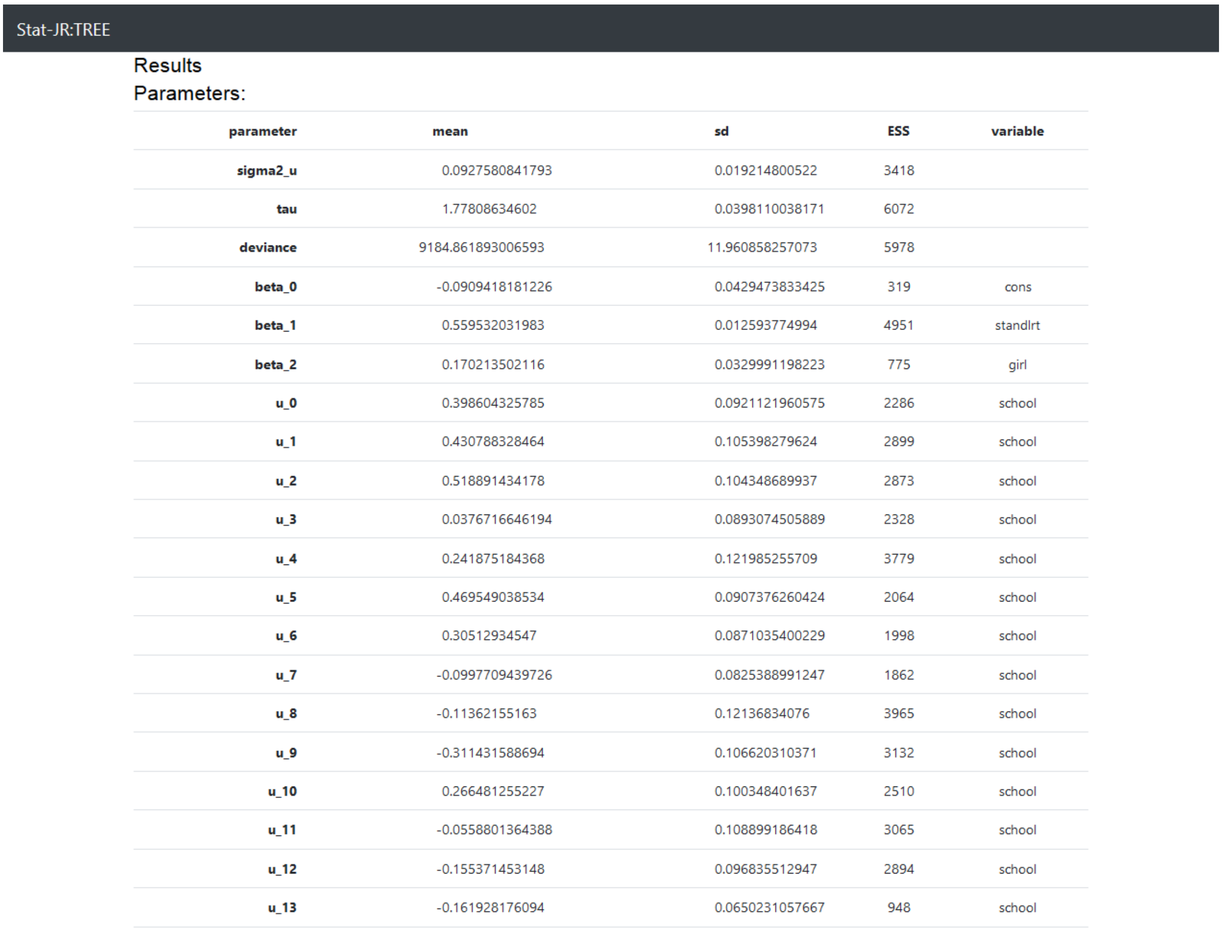
Dataset: *tutorial;* **Template:** *2LevelMod;* **Input string:** *{'Engine': 'eStat', 'L2ID': 'school', 'burnin': '500', 'D': 'Normal', 'outdata': 'out2level', 'storeresid': 'Yes', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt,girl', 'makepred': 'No', 'seed': '1', 'defaultsv': 'Yes'}*



Response:	normexam
Level 2 ID:	school
Specify distribution:	Normal
Explanatory variables:	cons,standlrt,girl
Store level 2 residuals?	Yes
Choose estimation engine:	eStat
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:	out2level

Press **Next** and then **Run** to fit the model. Note that running may take a while as we are storing all 65 school effects and so for each one the software needs to construct diagnostic plots.

When the model finishes select **ModelResults**, from the output list, and show the results in a separate tab.



Results

Parameters:

parameter	mean	sd	ESS	variable
sigma2_u	0.0927580841793	0.019214800522	3418	
tau	1.77808634602	0.0398110038171	6072	
deviance	9184.861893006593	11.960858257073	5978	
beta_0	-0.0909418181226	0.0429473833425	319	cons
beta_1	0.559532031983	0.012593774994	4951	standlrt
beta_2	0.170213502116	0.0329991198223	775	girl
u_0	0.398604325785	0.0921121960575	2286	school
u_1	0.430788328464	0.105398279624	2899	school
u_2	0.518891434178	0.104348689937	2873	school
u_3	0.0376716646194	0.0893074505889	2328	school
u_4	0.241875184368	0.121985255709	3779	school
u_5	0.469549038534	0.0907376260424	2064	school
u_6	0.30512934547	0.0871035400229	1998	school
u_7	-0.0997709439726	0.0825388991247	1862	school
u_8	-0.11362155163	0.12136834076	3965	school
u_9	-0.311431588694	0.106620310371	3132	school
u_10	0.266481255227	0.100348401637	2510	school
u_11	-0.0558801364388	0.108899186418	3065	school
u_12	-0.155371453148	0.096835512947	2894	school
u_13	-0.161928176094	0.0650231057667	948	school

Here if you scroll down we see that the DIC value for the two-level model is 9245, compared with 9725 for the simpler model, showing that it is important to account for the two levels in the data. If you scroll down to the beta fixed effect parameters, as shown in the table below, you will find that their mean estimates have changed little.

Parameter	Single level Mean(sd)	Single level ESS	2level Mean(sd)	2level ESS
beta_0	-0.103 (0.0196)	1615	-0.091 (0.0429)	319
beta_1	0.590 (0.0126)	5488	0.560 (0.0126)	4951
beta_2	0.170 (0.0254)	1623	0.170 (0.0330)	775

The standard deviations for *beta_0* and *beta_2* have increased due to taking account of the clustering, and the ESS values have reduced due to correlation in estimating the fixed effects and level 2 residuals.

3.6 Caterpillar plot

The random effects in the 2-level model are also interesting to look at, and one graph that is often used is a caterpillar plot. This can be produced in Stat-JR using a template specifically designed for producing this plot. This template requires the user to select all the ‘u’s to be displayed in the plot, which can be time-consuming if there are many of them:

From the top bar we need to select **Choose** for **Template** and **Dataset**.

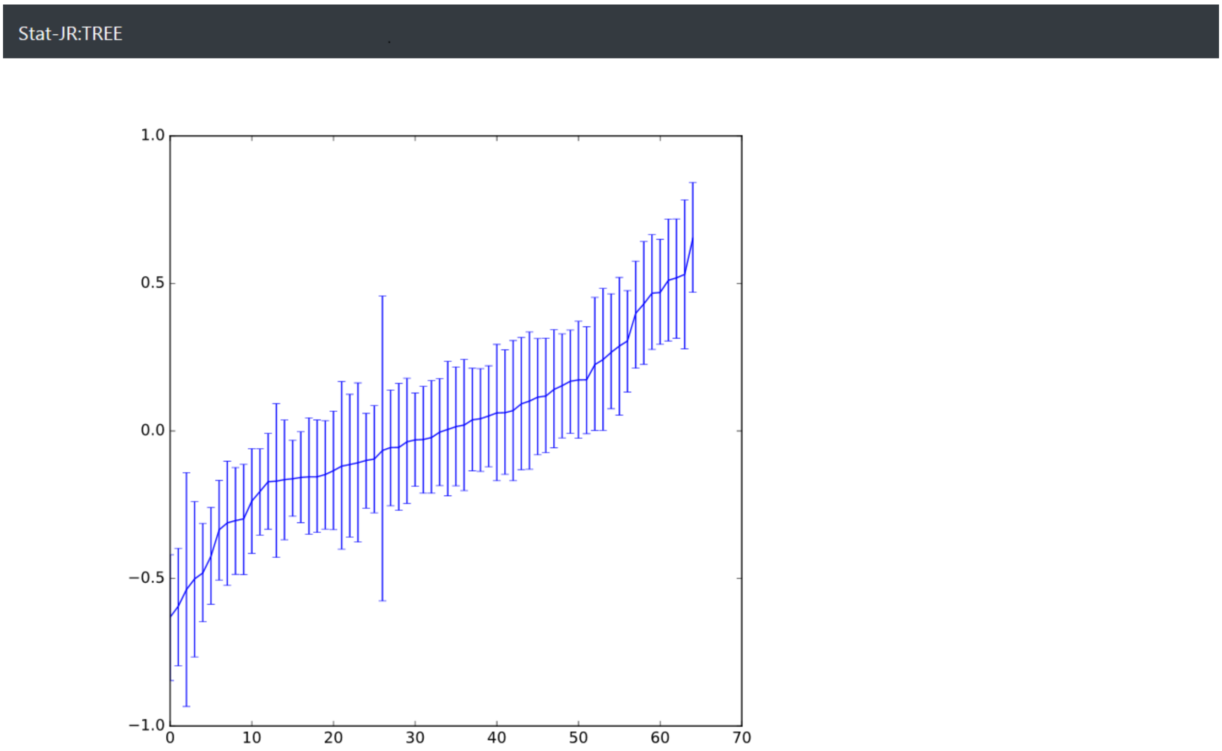
Choose **CaterpillarPlot95** as the template and **out2level** as the dataset.

You need now to select all the ‘u’s from *u0* to *u64* which is best done by clicking on *u0* and holding down the mouse and scrolling down to multiselect all the ‘u’s together

Once all are selected press the **Next** and **Run** buttons.

Select **caterpillar.svg** in the pull-down list and view in a new tab as follows:

Dataset: *out2level;* **Template:** *CaterpillarPlot95;* **Input string:** *{'residuals': 'u_0,u_1,u_2,u_3,u_4,u_5,u_6,u_7,u_8,u_9,u_10,u_11,u_12,u_13,u_14,u_15,u_16,u_17,u_18,u_19, u_20,u_21,u_22,u_23,u_24,u_25,u_26,u_27,u_28,u_29,u_30,u_31,u_32,u_33,u_34,u_35,u_36,u_37, u_38,u_39,u_40,u_41,u_42,u_43,u_44,u_45,u_46,u_47,u_48,u_49,u_50,u_51,u_52,u_53,u_54,u_55, u_56,u_57,u_58,u_59,u_60,u_61,u_62,u_63,u_64'}*



This graph shows the schools in order of ascending mean whilst the bars give a 95% confidence interval around each mean. The school in the middle with the wide confidence interval (i.e. very large bars) has only 2 pupils and so there is much greater uncertainty in the estimate.

In this chapter we have explored fitting three models to the tutorial dataset. This has illustrated how the Stat-JR system works, how to interpret the output from MCMC and eStat, and how to compare models via the DIC diagnostic. There are better models that can be fitted to the dataset: for example, we could look at treating the effect of intake score (*standlrt*) as random, and fit a random slopes model using the template **2LevelRS**; in the future we may add material on this subject to this manual, but for now we leave this as an exercise for the reader. Next we turn to the interoperability features of Stat-JR.

4 Interoperability – a brief introduction

In this section we look at interoperability with other software packages.

In order for Stat-JR to interoperate with a third-party package, the user needs to check that the Stat-JR template he/she wishes to use with it supports interoperability with that third-party software package (this can be checked via **Template > Choose** in the black bar at the top of the TREE interface, and then either using the red cloud terms or by clicking on the 'cog' symbol next to the name of the currently-selected template), that that third-party package is installed, and that Stat-JR knows where to find it (see the paths specified in **Settings** via the black bar at the top of the TREE interface; note that if you change a path, then make sure you press the **Set** button at the bottom of the **Settings** screen and then select **Debug > Reload packages** (via the black bar at the top) to implement this change in the current session).

So, whilst all the templates used in this section support interoperability with the packages we explore, in order to run this section successfully, the user will need to have these packages installed and to have told Stat-JR where to find them.

Stat-JR can interoperate with a variety of third-party statistical packages (see http://www.bristol.ac.uk/cmm/software/statjr/downloads/additionalsoft.html for more details), including the following:

- aML (Lillard & Panis, 2003)
- GenStat (VSN International, 2015)
- Gretl (Cottrell & Lucchetti, 2007)
- JAGS (Plummer, 2003)
- MATLAB (The MathWorks, Inc., 2017)
- Minitab (Minitab, Inc. , 2017)
- MIXREGLS (Hedeker & Nordgren, 2013)
- MLwiN (Charlton, Rasbash, Browne, Healy, & Cameron, 2017)
- Octave (Eaton, Bateman, Hauberg, & Wehbring, 2016)
- OpenBUGS (Lunn, Spiegelhalter, Thomas, & Best, 2009)
- PSPP (Free Software Foundation, 2017)
- R (R Core Team, 2016)
- SABRE (Barry, Francis, & Davies , 1989)
- SAS (SAS Institute Inc., 2011)
- SPSS (IBM Corp., 2017)
- Stata (StataCorp, 2017)
- SuperMix
- WinBUGS (Lunn, Thomas, Best, & Spiegelhalter, 2000)

…as well as, as we've seen, being able to use its own in-house model estimation engine (eStat), and a variety of Python (which is the main language in which Stat-JR is written) functions.

In this section we demonstrate interoperation by selecting a few of these third-party packages.

4.1 So why are we offering interoperability?

There are many motivations that could be given for the benefits of having an interoperability interface. First and foremost it opens up functionality in other software packages through a common interface.

One important feature that the template, **Regression1AML** (which we cover at the end of this chapter), shows is that not all model templates need to use the built-in eStat engine. It would be perfectly reasonable for a user to construct a template that fitted a specific family of models in the WinBUGS software and then novice users would have access to a user-friendly interface to such models without having to understand the subtleties of writing WinBUGS code; it can thus play an important role in introducing packages, such as WinBUGS, to new users. This follows earlier work: for example on the MLwiN-WinBUGS interface that we developed 10 years ago.

It also offers an easy way of comparing different software packages for a multitude of examples, and we will return to this in Section 5.4. Finally it can be thought of as a teaching tool, so that a user familiar with one package can use Stat-JR and directly compare the script files, etc., required for the package with which they are familiar to those required for an alternative package.

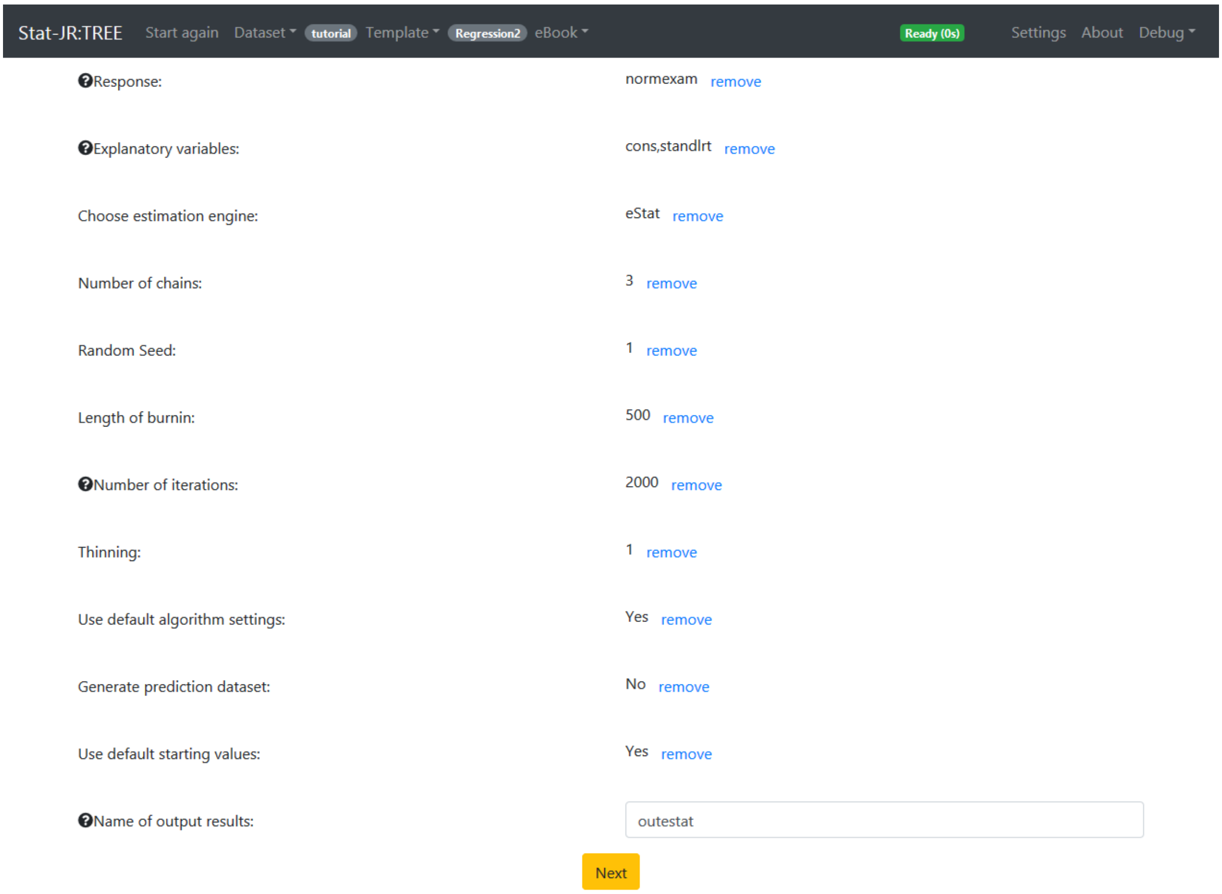
4.2 Regression in eStat revisited

In Section 3 we looked at fitting a few models to the **tutorial** dataset using the built-in eStat engine: 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. It is an MCMC-based estimation method, but is also rather quick. In this chapter we will stick with one simple example, the initial linear regression model that we fitted to the 'tutorial' dataset that we considered in Section 3. We will need to use a new template, **Regression2**, as the **Regression1** template only supports the eStat engine.

We will begin by setting-up the model and running it in eStat:

From the top bar select **Regression2** as the template, and **tutorial** as the dataset using the **Choose** options on the pull-down lists for templates and datasets and set-up the inputs as follows:

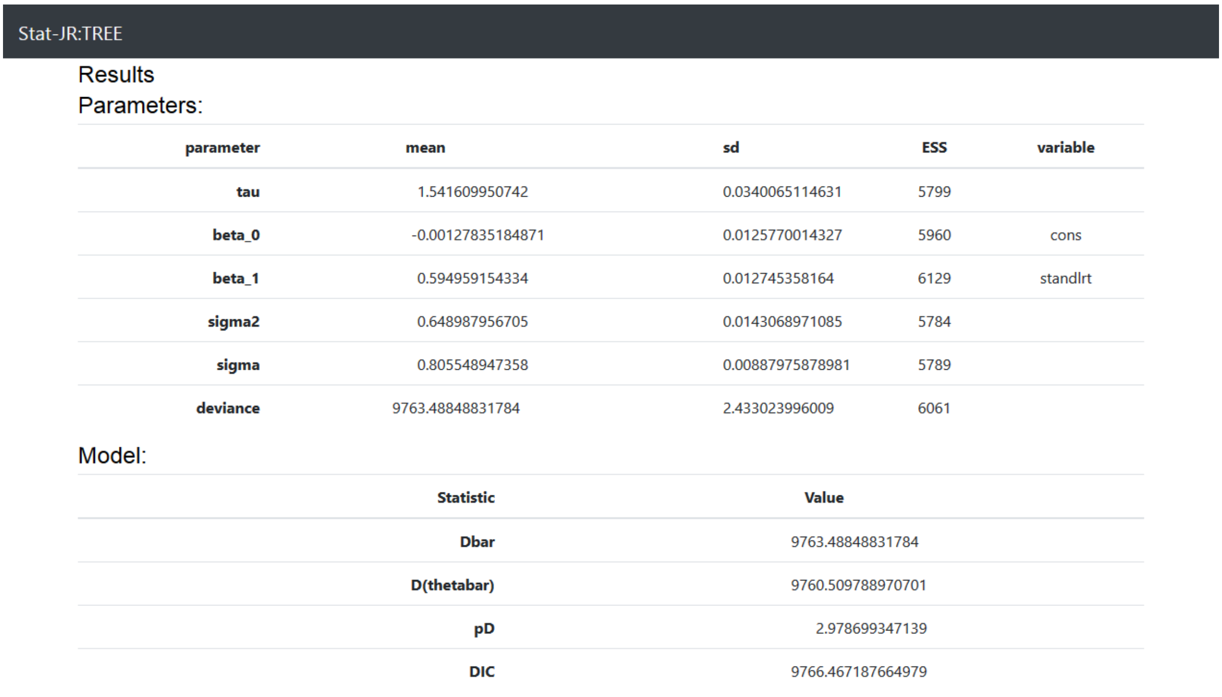
Dataset: *tutorial;* **Template:** *Regression2;* **Input string:** *{'Engine': 'eStat', 'burnin': '500', 'defaultsv': 'Yes', 'outdata': 'outestat', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1', 'makepred': 'No'}*



Response:	normexam remove
Explanatory variables:	cons,standlrt 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:	outestat

Click on **Next** and **Run** to fit the model.

Select **ModelResults** from the pull-down list, and show this output in a new tab which should look as follows:



Results

Parameters:

parameter	mean	sd	ESS	variable
tau	1.541609950742	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	9763.48848831784	2.433023996009	6061	

Model:

Statistic	Value
Dbar	9763.48848831784
D(thetabar)	9760.509788970701
pD	2.978699347139
DIC	9766.467187664979

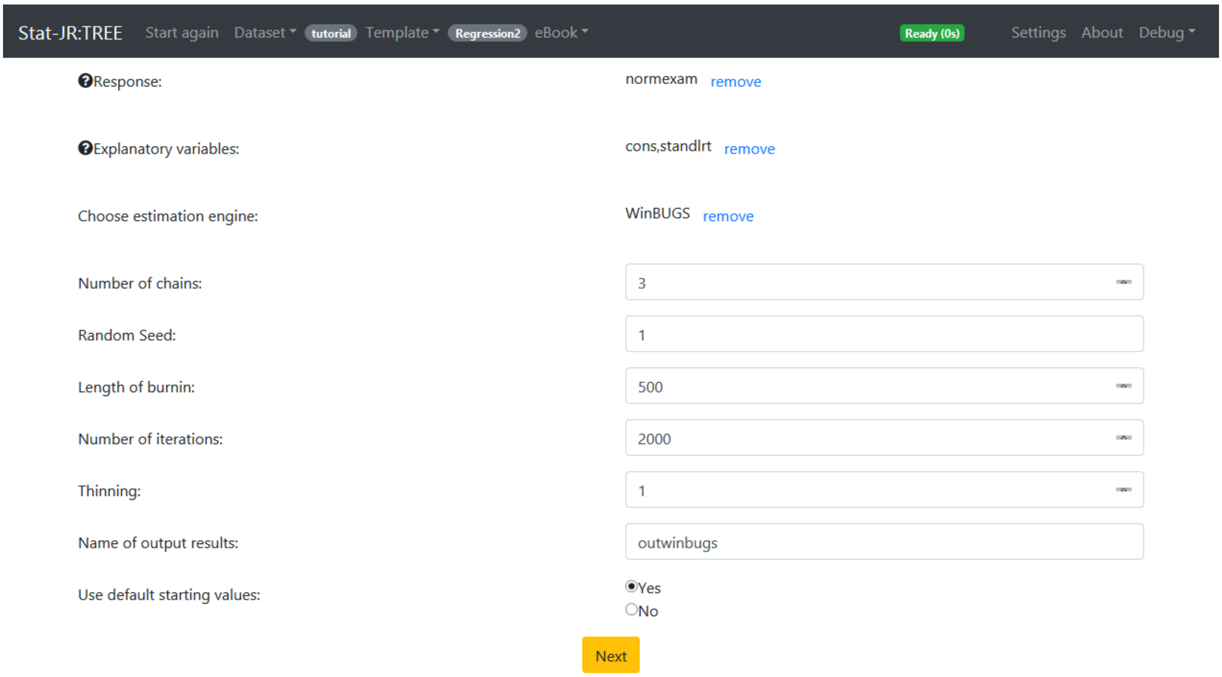
These results are identical to those we obtained using **Regression1** earlier, although we only looked at the plot for *beta_1* in Section 3.3. We will use this as a benchmark, contrasting these results with those we obtain from the other packages, although it is worth noting that all packages will have good mixing and converge quickly for this simple linear regression model. You might like to explore differences between engines / packages for other models yourself after reading this chapter.

4.3 Interoperability with WinBUGS

WinBUGS (Lunn, Thomas, Best, & Spiegelhalter, 2000) is an MCMC-based package developed (as BUGS – Bayesian inference Using Gibbs Sampling) originally in the early 1990s by a team of researchers at the MRC Biostatistics Unit in Cambridge. It is a very flexible package and can fit, in a Bayesian framework, most statistical models, provided you can describe them in its model specification language. In Stat-JR we have borrowed much of this language for our own algebra system, and so many templates feature interoperability with WinBUGS.

To fit the current model using WinBUGS we can click on remove next to the **Choose estimation engine** input and set up the template inputs as follows:

Dataset: *tutorial;* **Template:** *Regression2;* **Input string:** *{'Engine': 'WinBUGS', 'burnin': '500', 'defaultsv': 'Yes', 'outdata': 'outwinbugs', 'thinning': '1', 'nchains': '3', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1'}*



Response: normexam remove

Explanatory variables: cons,standlrt remove

Choose estimation engine: WinBUGS remove

Number of chains: 3

Random Seed: 1

Length of burnin: 500

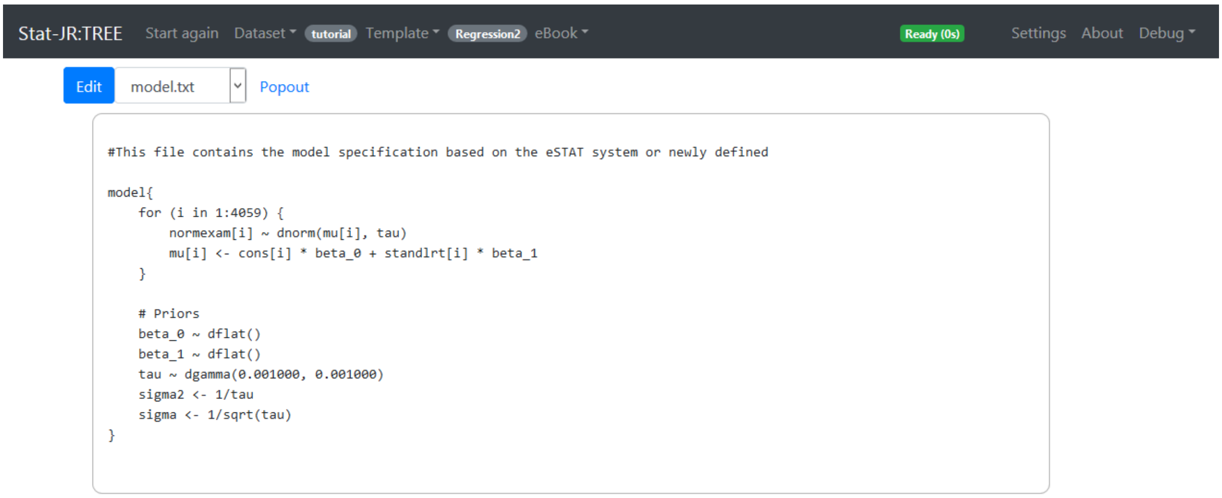
Number of iterations: 2000

Thinning: 1

Name of output results: outwinbugs

Use default starting values: ● Yes ○ No

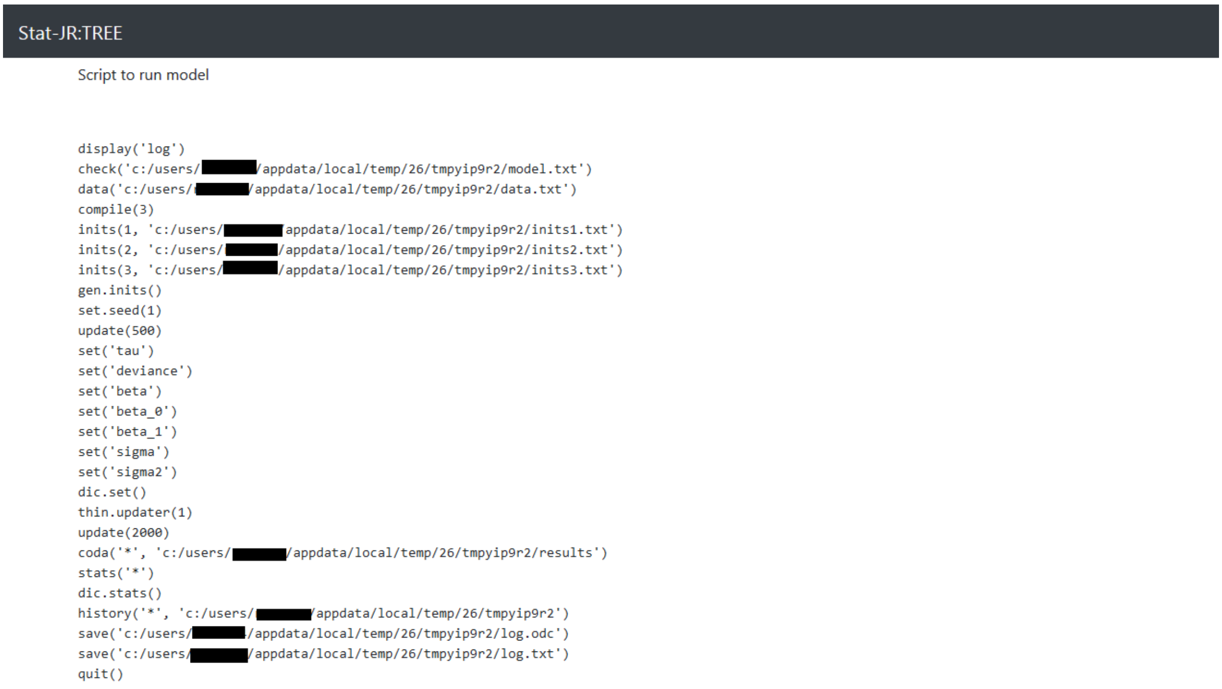
When we press **Next** the Stat-JR software will construct all the files required to run WinBUGS so for example we can choose *model.txt* from the list:



```
#This file contains the model specification based on the eSTAT system or newly defined

model{
    for (i in 1:4059) {
        normexam[i] ~ dnorm(mu[i], tau)
        mu[i] <- cons[i] * beta_0 + standlrt[i] * beta_1
    }

    # Priors
    beta_0 ~ dflat()
    beta_1 ~ dflat()
    tau ~ dgamma(0.001000, 0.001000)
    sigma2 <- 1/tau
    sigma <- 1/sqrt(tau)
}
```

Here we see the model defined in the WinBUGS model specification language in the output pane. This file is almost identical to that used by eStat aside from the expression *length(normexam)* being replaced here by its value 4059.

Selecting *script.txt* from the list and popping out to a new tab gives the following:



Stat-JR:TREE

Script to run model

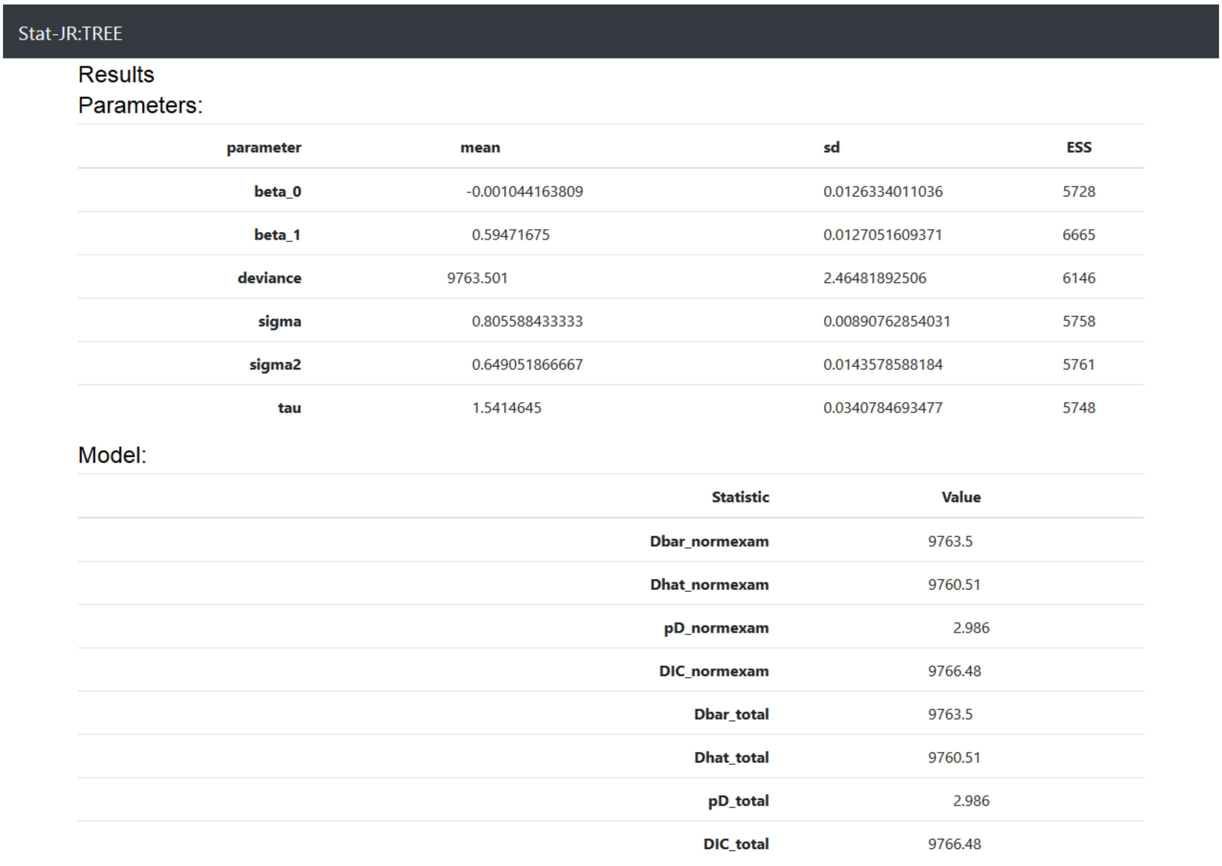
```
display('log')
check('c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2/model.txt')
data('c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2/data.txt')
compile(3)
inits(1, 'c:/users/[redacted]appdata/local/temp/26/tmpyip9r2/inits1.txt')
inits(2, 'c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2/inits2.txt')
inits(3, 'c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2/inits3.txt')
gen.inits()
set.seed(1)
update(500)
set('tau')
set('deviance')
set('beta')
set('beta_0')
set('beta_1')
set('sigma')
set('sigma2')
dic.set()
thin.updater(1)
update(2000)
coda('*', 'c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2/results')
stats('*')
dic.stats()
history('*', 'c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2')
save('c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2/log.odc')
save('c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2/log.txt')
quit()
```

Here we see a list of the commands to be run in the WinBUGS language to fit the model. Note that this is done using a temporary directory and so this pathname appears in many commands.

Return to the tab containing the main page and click on the **Run** button.

The WinBUGS package then pops up in its own window, runs the above script, and returns control to Stat-JR when it has finished estimating the model.

If we look at the *ModelResults* output from the list and pop it out to its own tab we will see the following:



Stat-JR:TREE

Results

Parameters:

parameter	mean	sd	ESS
beta_0	-0.001044163809	0.0126334011036	5728
beta_1	0.59471675	0.0127051609371	6665
deviance	9763.501	2.46481892506	6146
sigma	0.805588433333	0.00890762854031	5758
sigma2	0.649051866667	0.0143578588184	5761
tau	1.5414645	0.0340784693477	5748

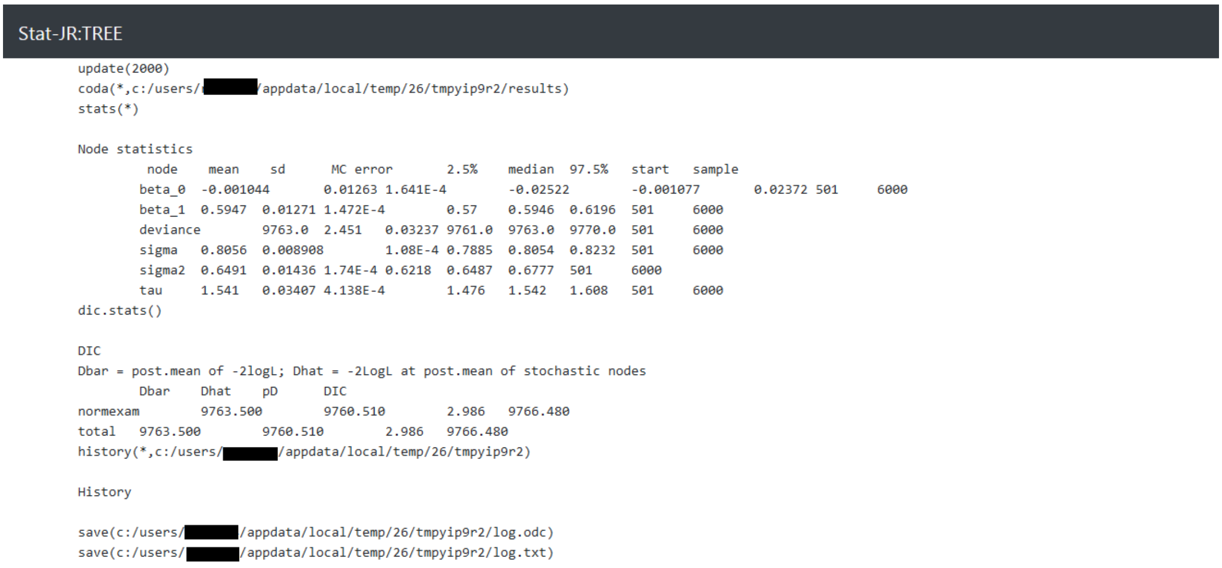
Model:

Statistic	Value
Dbar_normexam	9763.5
Dhat_normexam	9760.51
pD_normexam	2.986
DIC_normexam	9766.48
Dbar_total	9763.5
Dhat_total	9760.51
pD_total	2.986
DIC_total	9766.48

These estimates, as one might expect, are very close to those from eStat, and again all ESS values are around 5,000-6,000. We can also look at the log file from WinBUGS:

Return to the template tab and choose *log.txt* in the outputs list.

Scroll the *log.txt* file down to the bottom, and the screen should look as follows:



Stat-JR:TREE

```
update(2000)
coda(*,c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2/results)
stats(*)

Node statistics
         node    mean    sd      MC error       2.5%    median  97.5%   start   sample
        beta_0  -0.001044       0.01263 1.641E-4        -0.02522        -0.001077       0.02372 501     6000
        beta_1  0.5947  0.01271 1.472E-4        0.57    0.5946  0.6196  501     6000
        deviance        9763.0  2.451   0.03237 9761.0  9763.0  9770.0  501     6000
        sigma   0.8056  0.008908        1.08E-4 0.7885  0.8054  0.8232  501     6000
        sigma2  0.6491  0.01436 1.74E-4 0.6218  0.6487  0.6777  501     6000
        tau     1.541   0.03407 4.138E-4        1.476   1.542   1.608   501     6000
dic.stats()

DIC
Dbar = post.mean of -2logL; Dhat = -2LogL at post.mean of stochastic nodes
        Dbar    Dhat    pD      DIC
normexam        9763.500        9760.510        2.986   9766.480
total   9763.500        9760.510        2.986   9766.480
history(*,c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2)

History

save(c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2/log.odc)
save(c:/users/[redacted]/appdata/local/temp/26/tmpyip9r2/log.txt)
```

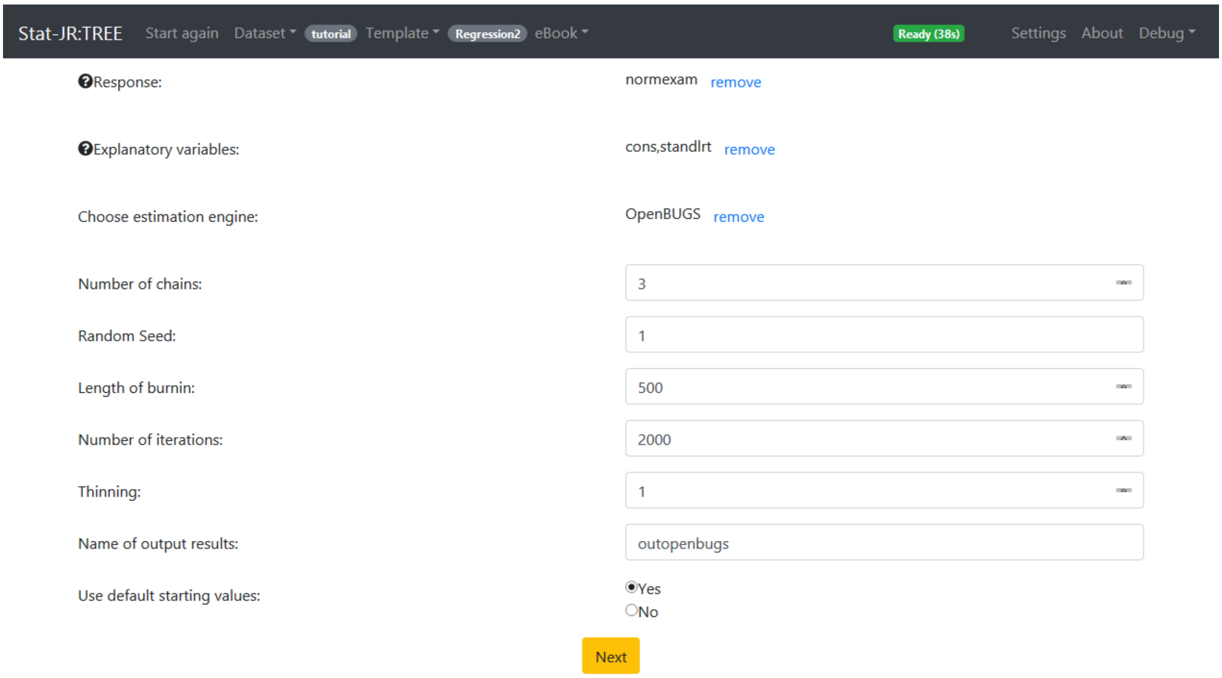
Here we see that the estimates and the DIC diagnostic are embedded in the log file, and take a similar value to eStat. WinBUGS required initial value files for each run (and these are stored in three text files beginning with *inits* and the chain number), together with a data file as well as the model and script files already shown. All of these are available to view and to use again, thus Stat-JR is useful for learning how these other packages, such as WinBUGS, work.

4.4 Interoperability with OpenBUGS

Our next package to consider is OpenBUGS (Lunn, Spiegelhalter, Thomas, & Best, 2009). OpenBUGS was developed by members of the same team who developed WinBUGS, but differs in that it is open source so other coders may get access to the source code, and in theory develop new features in the software.

To run OpenBUGS via Stat-JR click on the word **remove** next to the **Choose Estimation engine** input, set up the template as follows, and then click on **Next**:

Dataset: *tutorial;* **Template:** *Regression2;* **Input string:** *{'Engine': 'OpenBUGS', 'burnin': '500', 'defaultsv': 'Yes', 'outdata': 'outopenbugs', 'thinning': '1', 'nchains': '3', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1'}*



Response: normexam remove

Explanatory variables: cons,standlrt remove

Choose estimation engine: OpenBUGS remove

Number of chains: 3

Random Seed: 1

Length of burnin: 500

Number of iterations: 2000

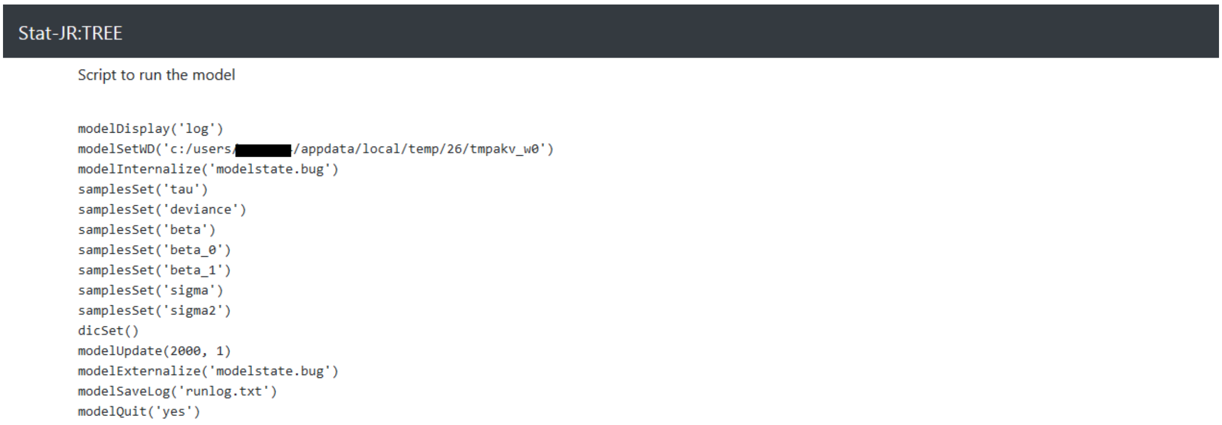
Thinning: 1

Name of output results: outopenbugs

Use default starting values: ● Yes ○ No

Next

This will have set-up the files required for OpenBUGS; these are similar, but not identical, to WinBUGS: the script file, in particular, is somewhat different and is split into three parts called *initscript.txt*, *runscript.txt* (shown below) and *resultsscript.txt* (you can access these from the objects list):

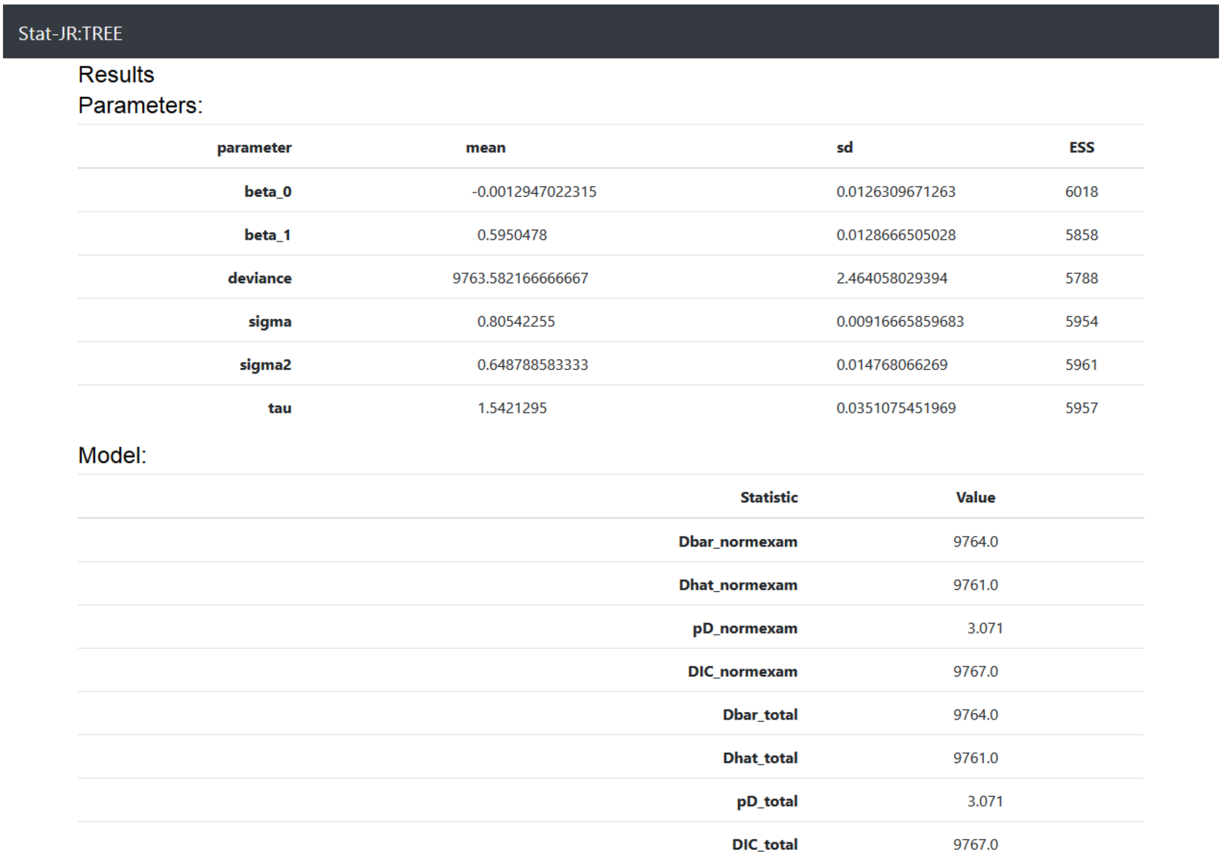


Script to run the model

```
modelDisplay('log')
modelSetWD('c:/users/[redacted]/appdata/local/temp/26/tmpakv_w0')
modelInternalize('modelstate.bug')
samplesSet('tau')
samplesSet('deviance')
samplesSet('beta')
samplesSet('beta_0')
samplesSet('beta_1')
samplesSet('sigma')
samplesSet('sigma2')
dicSet()
modelUpdate(2000, 1)
modelExternalize('modelstate.bug')
modelSaveLog('runlog.txt')
modelQuit('yes')
```

OpenBUGS allows us to change the working directory, and so there is no need for other commands to include the temporary directory path. Unlike WinBUGS, OpenBUGS will run in the background, and so will not appear when we click run.

Clicking on **Run** and selecting *ModelResults* in its own tab gives the following:



Results

Parameters:

parameter	mean	sd	ESS
beta_0	-0.0012947022315	0.0126309671263	6018
beta_1	0.5950478	0.0128666505028	5858
deviance	9763.582166666667	2.464058029394	5788
sigma	0.80542255	0.00916665859683	5954
sigma2	0.648788583333	0.014768066269	5961
tau	1.5421295	0.0351075451969	5957

Model:

Statistic	Value
Dbar_normexam	9764.0
Dhat_normexam	9761.0
pD_normexam	3.071
DIC_normexam	9767.0
Dbar_total	9764.0
Dhat_total	9761.0
pD_total	3.071
DIC_total	9767.0

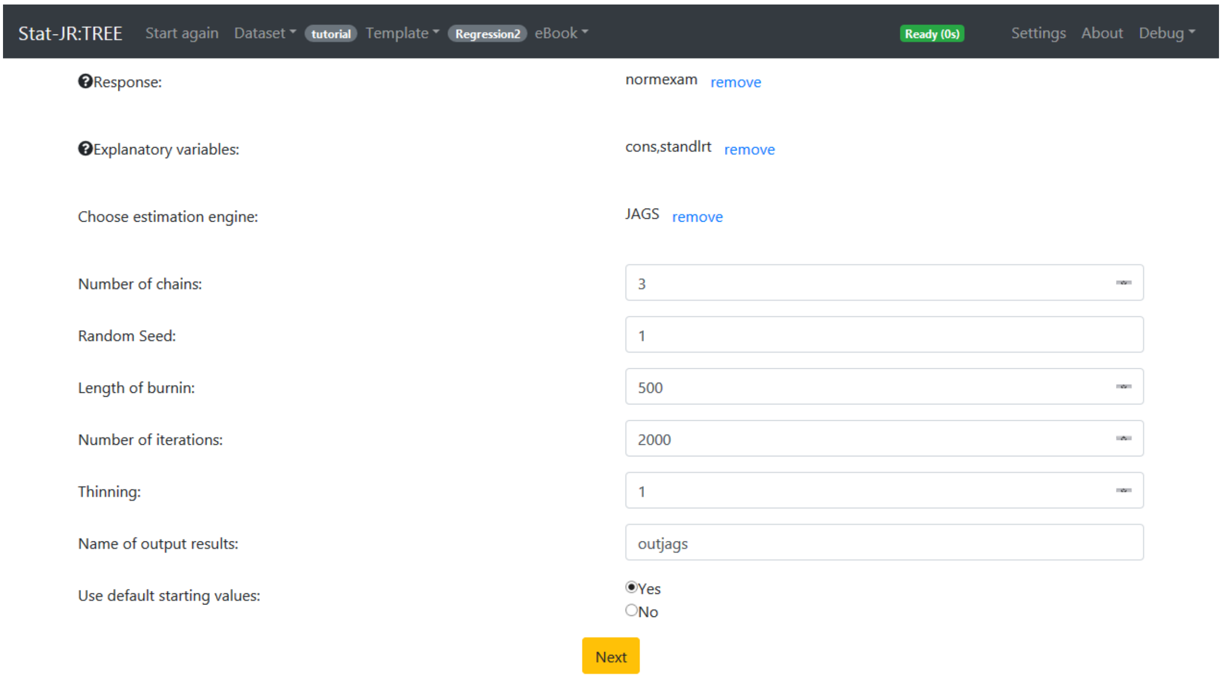
Again, these results are very similar in terms of parameter estimates and ESS values to the other software packages.

4.5 Interoperability with JAGS

The third standalone MCMC estimation engine available, via Stat-JR, is JAGS (“Just Another Gibbs Sampler”), developed by Martyn Plummer ((Plummer, 2003)). JAGS also uses the WinBUGS model language with a few modifications, but has a few differences in terms of script files and data files.

To run JAGS via Stat-JR click on the **remove** text next to **Choose estimation engine** and set-up the template as follows, before clicking on **Next**:

Dataset: *tutorial;* **Template:** *Regression2;* **Input string:** *{'Engine': 'JAGS', 'burnin': '500', 'defaultsv': 'Yes', 'outdata': 'outjags', 'thinning': '1', 'nchains': '3', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1'}*



Response: normexam remove

Explanatory variables: cons,standlrt remove

Choose estimation engine: JAGS remove

Number of chains: 3

Random Seed: 1

Length of burnin: 500

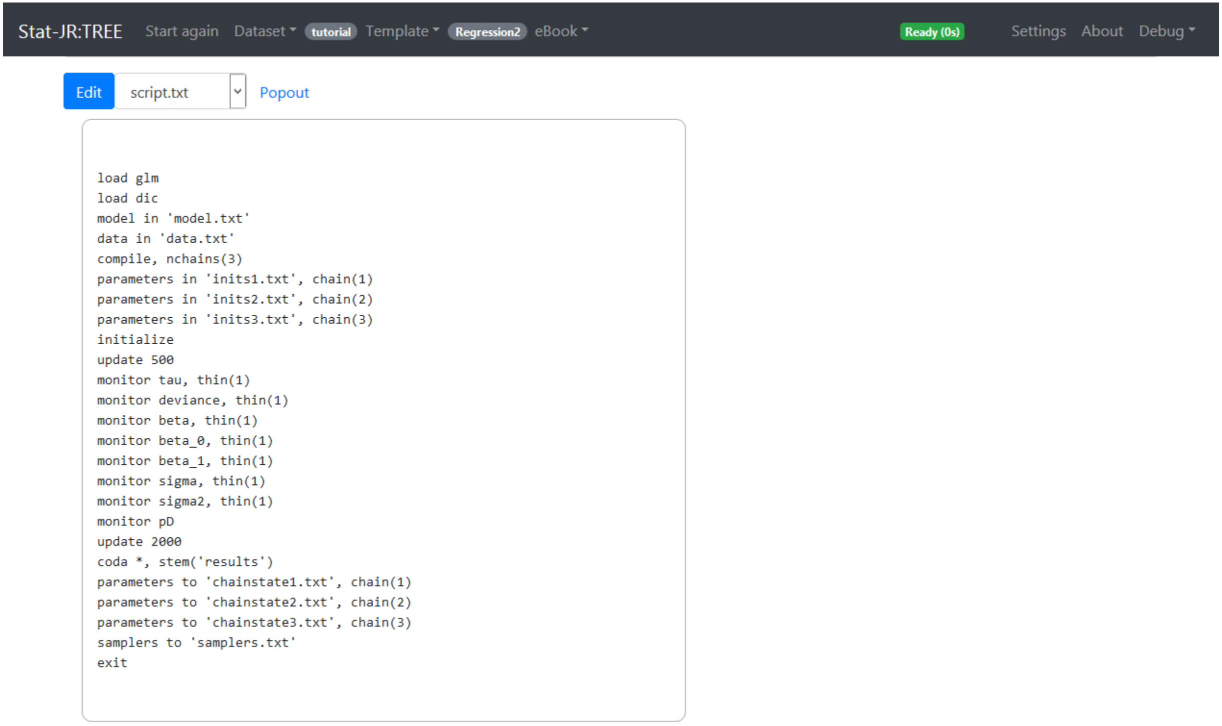
Number of iterations: 2000

Thinning: 1

Name of output results: outjags

Use default starting values: ● Yes ○ No

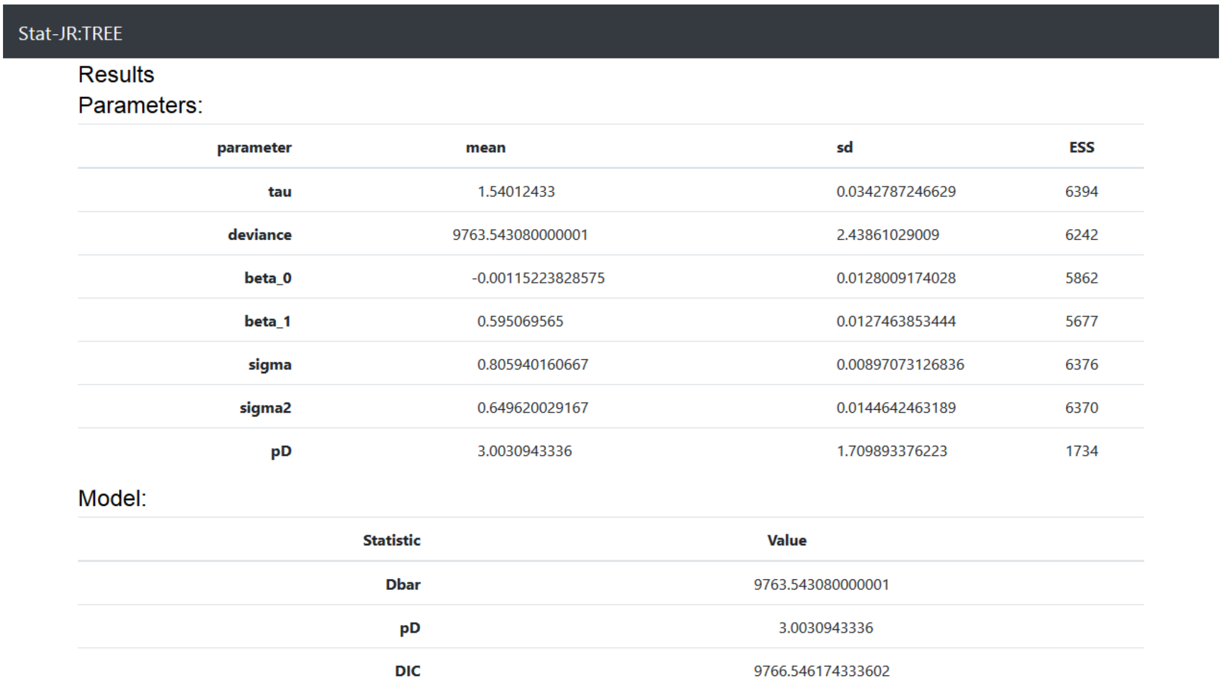
This will set-up the files required for JAGS; for example, here you can see the script file (*script.txt*) which show some differences to those for WinBUGS (as to the initial value file formats):



```
load glm
load dic
model in 'model.txt'
data in 'data.txt'
compile, nchains(3)
parameters in 'inits1.txt', chain(1)
parameters in 'inits2.txt', chain(2)
parameters in 'inits3.txt', chain(3)
initialize
update 500
monitor tau, thin(1)
monitor deviance, thin(1)
monitor beta, thin(1)
monitor beta_0, thin(1)
monitor beta_1, thin(1)
monitor sigma, thin(1)
monitor sigma2, thin(1)
monitor pD
update 2000
coda *, stem('results')
parameters to 'chainstate1.txt', chain(1)
parameters to 'chainstate2.txt', chain(2)
parameters to 'chainstate3.txt', chain(3)
samplers to 'samplers.txt'
exit
```

Like OpenBUGS, JAGS will run in the background (i.e. it will not open as a window on your screen although output will appear in the command window).

Clicking on **Run**, and placing *ModelResults* in a new tab, gives the following:



Results

Parameters:

parameter	mean	sd	ESS
tau	1.54012433	0.0342787246629	6394
deviance	9763.543080000001	2.43861029009	6242
beta_0	-0.00115223828575	0.0128009174028	5862
beta_1	0.595069565	0.0127463853444	5677
sigma	0.805940160667	0.00897073126836	6376
sigma2	0.649620029167	0.0144642463189	6370
pD	3.0030943336	1.709893376223	1734

Model:

Statistic	Value
Dbar	9763.543080000001
pD	3.0030943336
DIC	9766.546174333602

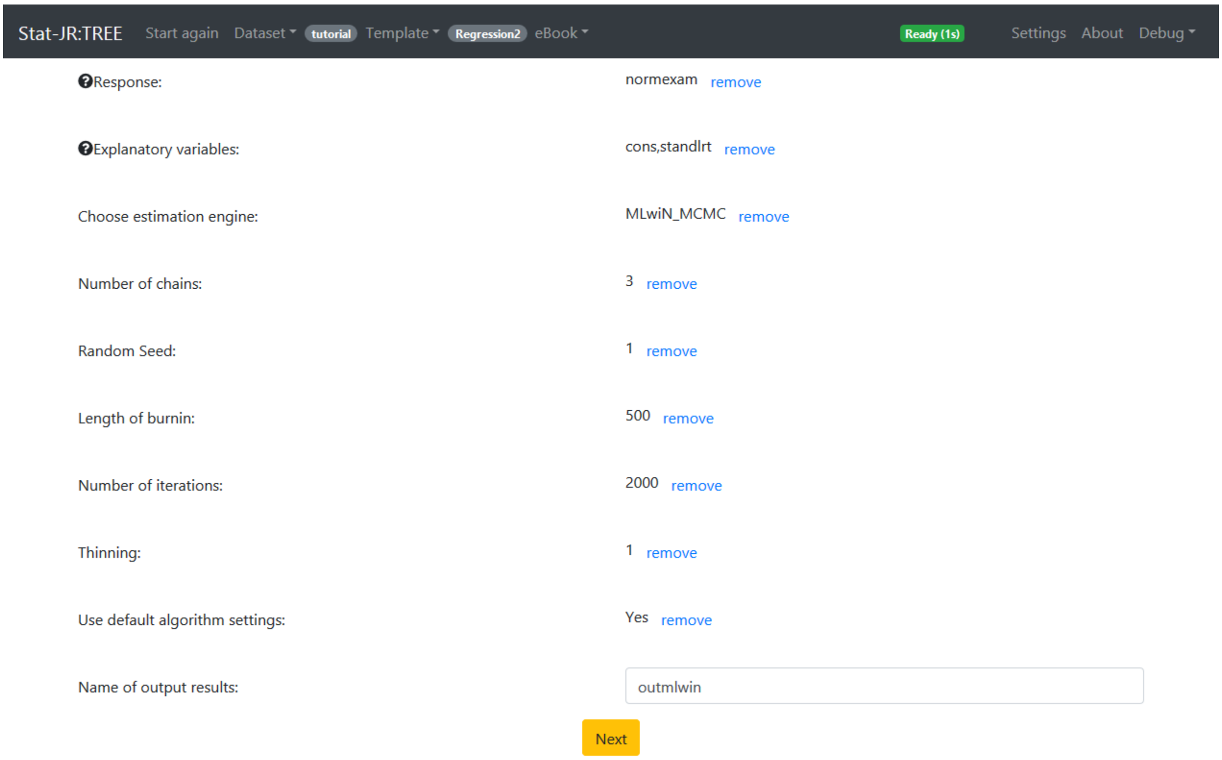
As you can see, we have similar estimates and effective sample sizes to the other estimation methods we've used. Whilst JAGS can be faster than WinBUGS and OpenBUGS, it fits a slightly smaller number of models.

4.6 Interoperability with MLwiN

MLwiN (Charlton, Rasbash, Browne, Healy, & Cameron, 2017) is a software package specifically written to fit multilevel statistical models. It features two estimation engines (for MCMC and likelihood-based (IGLS) methods, respectively) with a menu-driven, point-and-click user interface. It also has an underlying macro language, however, and this is what we use to interoperate with Stat-JR. We will first consider the MCMC engine. As it is limited in the scope of models it fits, this means it is generally quicker than the other MCMC packages. MLwiN is a single chain program, but can be made into a multiple chain engine with Stat-JR, since the latter can start-up three separate instances of MLwiN. At present these are given different random number seeds, but the same starting values, however different starting values can be specified manually by the user.

To run MCMC in MLwiN, via Stat-JR, click on the **remove** text by **Choose estimation engine** input and set-up the template as follows before clicking on **Next**:

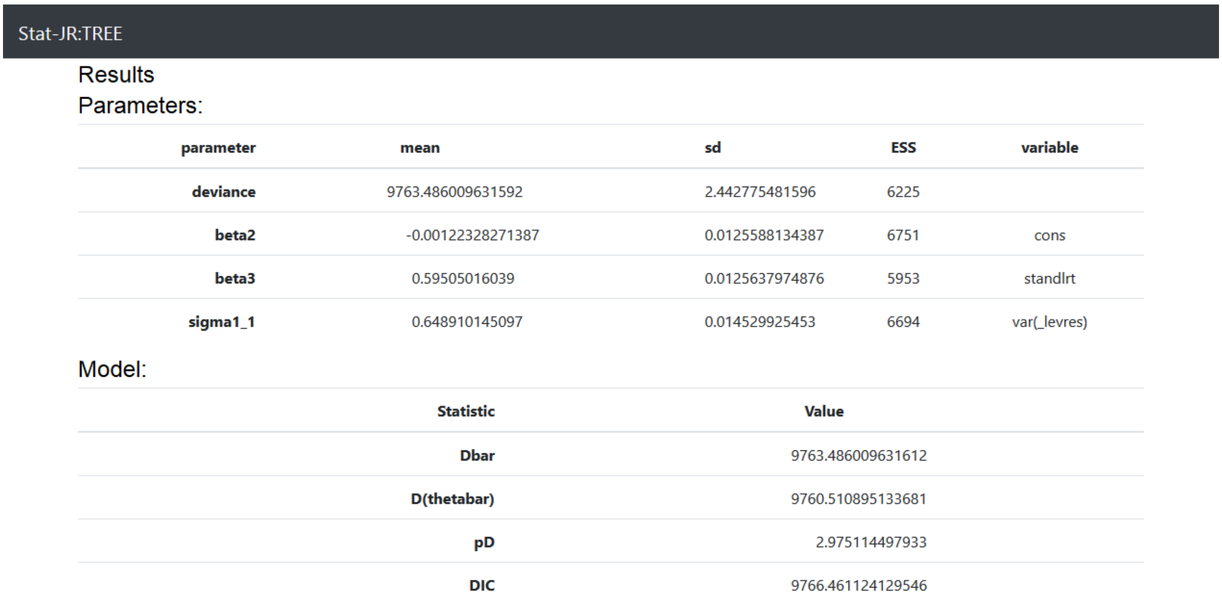
Dataset: *tutorial;* **Template:** *Regression2;* **Input string:** *{'Engine': 'MLwiN_MCMC', 'burnin': '500', 'outdata': 'outmlwin', 'thinning': '1', 'nchains': '3', 'defaultalg': 'Yes', 'iterations': '2000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1'}*



❓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

You can see, in the pull-down list, the dataset (in .dta format) that is used by MLwiN. There are also several MLwiN script files for the multiple chains and the several stages of model fitting.

Clicking on the **Run** button will set off three instances of MLwiN (in the background) and Stat-JR will then collate the results together. Choosing *ModelResults*, and displaying them in a new tab, gives the following:



Results

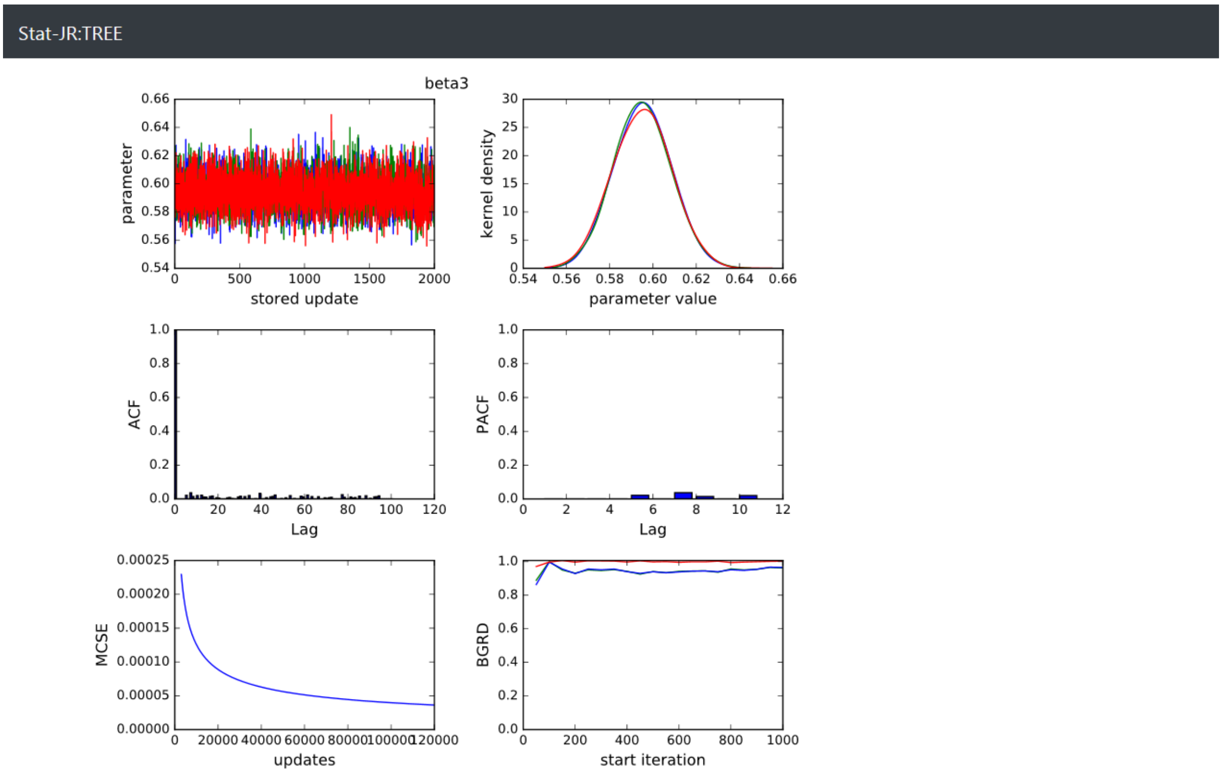
Parameters:

parameter	mean	sd	ESS	variable
deviance	9763.486009631592	2.442775481596	6225	
beta2	-0.00122328271387	0.0125588134387	6751	cons
beta3	0.59505016039	0.0125637974876	5953	standlrt
sigma1_1	0.648910145097	0.014529925453	6694	var(_levres)

Model:

Statistic	Value
Dbar	9763.486009631612
D(thetabar)	9760.510895133681
pD	2.975114497933
DIC	9766.461124129546

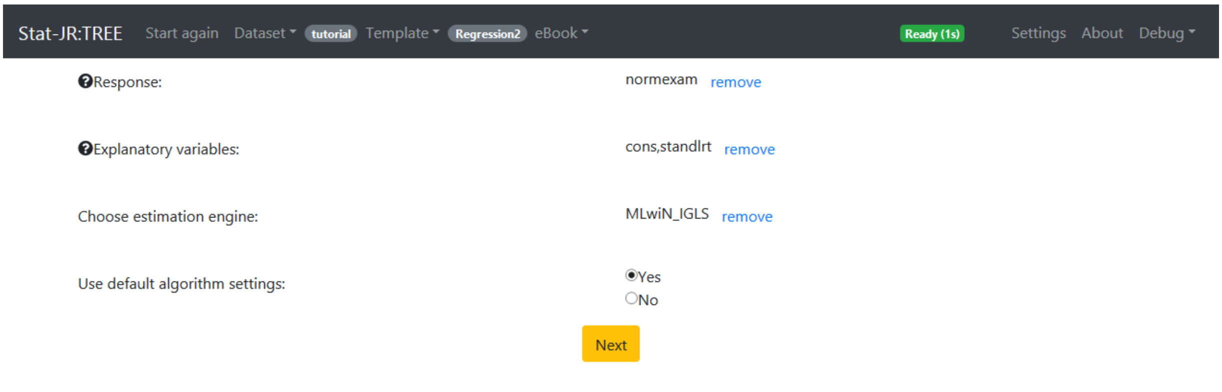
Once again here we have similar estimates, although the naming convention is slightly different for MLwiN. To show that we have multiple chains we can examine the chains for the slope (*beta3*), as shown below:



Stat-JR also offers the option of using the likelihood-based IGLS estimation engine in MLwiN.

To do this in MLwiN, via Stat-JR, click once again on the **remove** text next to the **Choose estimation engine** input and set-up the template as follows, before clicking on **Next**:

Dataset: *tutorial;* **Template:** *Regression2;* **Input string:** *{'y': 'normexam', 'x': 'cons,standlrt', 'Engine': 'MLwiN_IGLS', 'defaultalg': 'Yes'}*



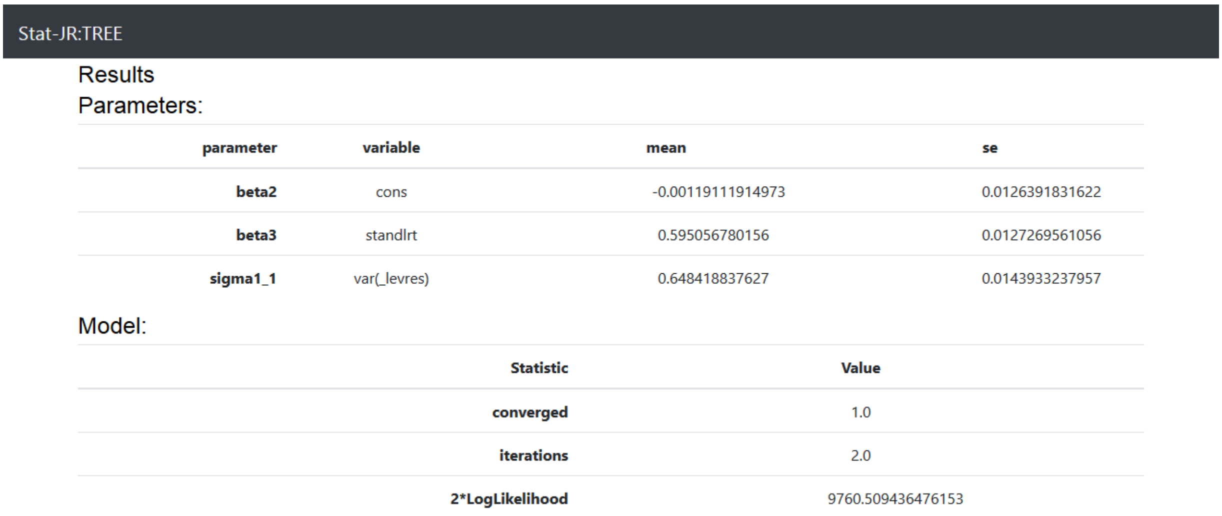
Response: normexam remove

Explanatory variables: cons,standlrt remove

Choose estimation engine: MLwiN_IGLS remove

Use default algorithm settings: ● Yes ○ No

Again the dataset will appears in the output pane, and this time pressing **Run** will give the following in the *ModelResults* output:



Stat-JR:TREE

Results

Parameters:

parameter	variable	mean	se
beta2	cons	-0.00119111914973	0.0126391831622
beta3	standlrt	0.595056780156	0.0127269561056
sigma1_1	var(_levres)	0.648418837627	0.0143933237957

Model:

Statistic	Value
converged	1.0
iterations	2.0
2*LogLikelihood	9760.509436476153

Here we get the *Deviance* (-2*Loglikelihood) value, together with parameter estimates with standard errors. The likelihood-based methods are far faster to run than the MCMC-based methods.

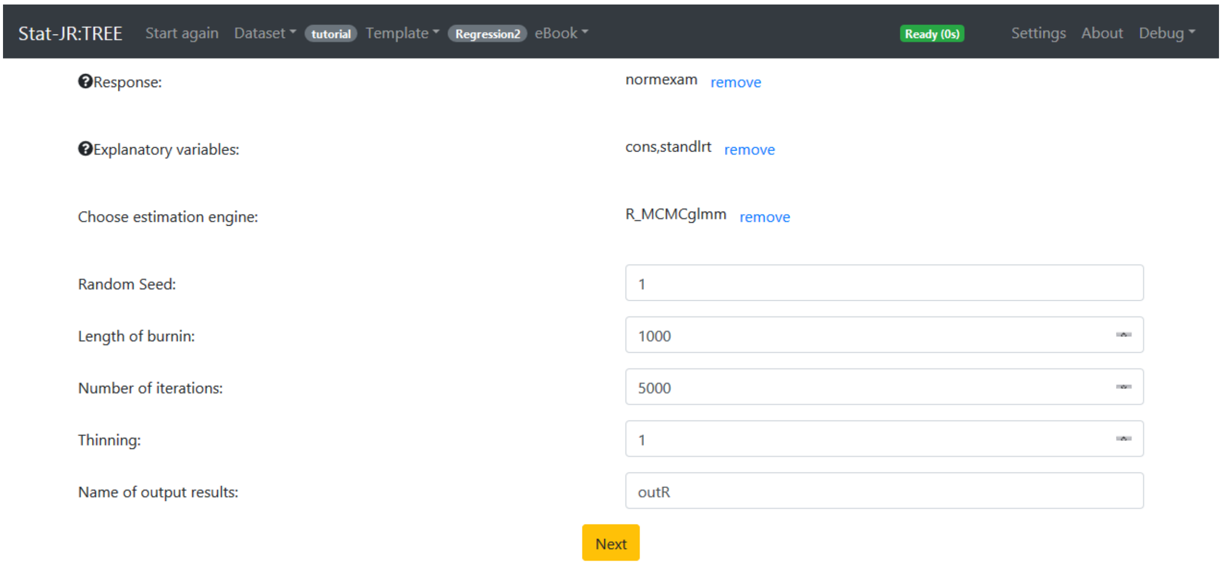
4.7 Interoperability with R

R (R Core Team, 2016) is another more general-purpose package that can be used to fit many statistical models. R has many parallels with Stat-JR in that users can supply functions (like Stat-JR templates) which are then added to the library of R packages. Here we will demonstrate fitting a model using the R package MCMCglmm (Hadfield, 2010), which is MCMC-based, and also using the glm function (from R's stats "base package"), which is a standard regression modelling function[1]. We will firstly demonstrate fitting a model using R's MCMCglmm package.

To run MCMC in R, via Stat-JR, click on the remove text by the **Choose estimation engine** input and set-up the template as follows, and click on **Next**:

[1] Interoperability is also offered via R's nimble package ((de Valpine, Paciorek, Turek, Anderson-Bergman, & Temple Lang, 2016)), although note that this in turn has a dependency on Rtools (https://cran.r-project.org/bin/windows/Rtools/) since it compiles code dynamically.

Dataset: *tutorial;* **Template:** *Regression2;* **Input string:** *{'Engine': 'R_MCMCglmm', 'burnin': '1000', 'outdata': 'outR', 'thinning': '1', 'iterations': '5000', 'y': 'normexam', 'x': 'cons,standlrt', 'seed': '1'}*



Response: normexam remove

Explanatory variables: cons,standlrt remove

Choose estimation engine: R_MCMCglmm remove

Random Seed: 1

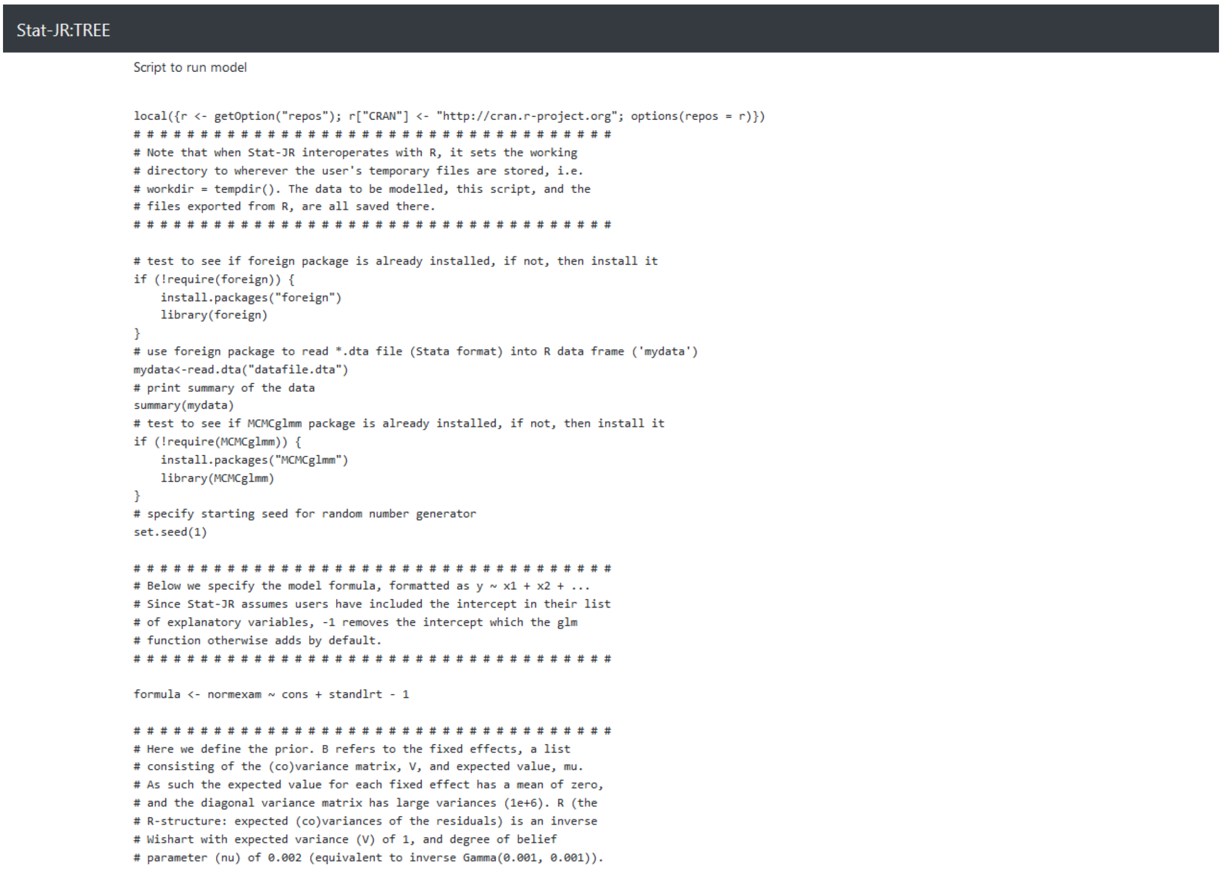
Length of burnin: 1000

Number of iterations: 5000

Thinning: 1

Name of output results: outR

After pressing **Next**, if we look at the script file, *script.R*, which we can select from the outputs list, we see the following:



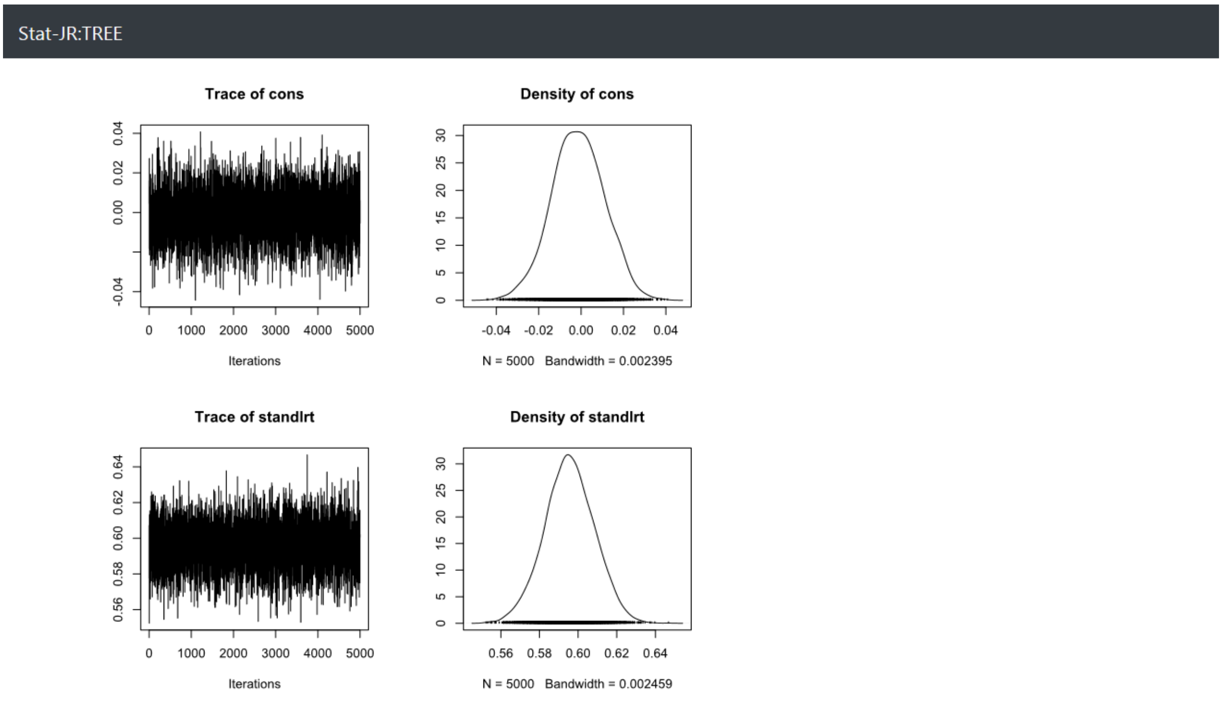
Script to run model

```
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)
}
# use foreign package to read *.dta file (Stata format) into R data frame ('mydata')
mydata<-read.dta("datafile.dta")
# print summary of the data
summary(mydata)
# test to see if MCMCglmm package is already installed, if not, then install it
if (!require(MCMCglmm)) {
    install.packages("MCMCglmm")
    library(MCMCglmm)
}
# specify starting seed for random number generator
set.seed(1)

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# 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

# # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # # #
# Here we define the prior. B refers to the fixed effects, a list
# consisting of the (co)variance matrix, V, and expected value, mu.
# As such the expected value for each fixed effect has a mean of zero,
# and the diagonal variance matrix has large variances (1e+6). R (the
# R-structure: expected (co)variances of the residuals) is an inverse
# Wishart with expected variance (V) of 1, and degree of belief
# parameter (nu) of 0.002 (equivalent to inverse Gamma(0.001, 0.001)).
```

MCMCglmm can fit all forms of generalised linear mixed models, of which a linear regression is a rather trivial case. You will see that the script file contains some setup code which will actually download and install the MCMCglmm library the first time you execute the script (so ensure your machine is connected to the internet) before calling the MCMCglmm command and then producing summaries.

Clicking on **Run** in the main window will create several outputs.

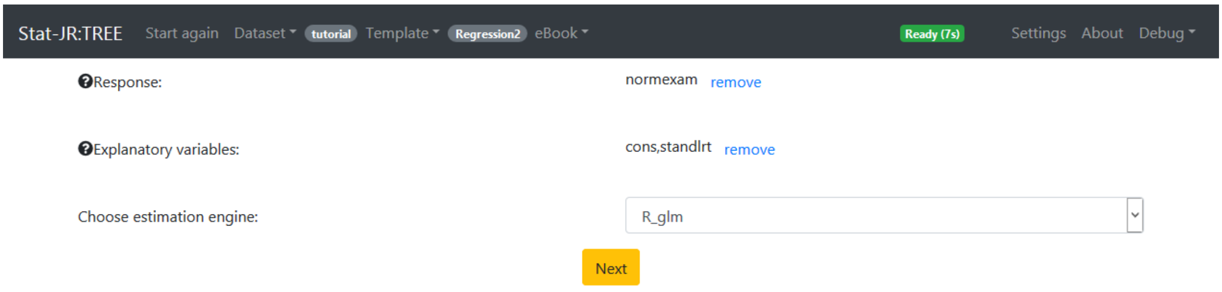
The *ModelResults* are similar to other software but we can also look at diagnostics plots that are specific to R by selecting *DiagPlots1.png*:



Here R gives trace plots and kernel density plots for both the intercept and the slope parameter.

Turning next to the glm function we can click on the **remove** text by **Choose estimation engine** and set-up the template as follows, before clicking on **Next** :

Dataset: *tutorial;* **Template:** *Regression2;* **Input string**: *{'y': 'normexam', 'x': 'cons,standlrt', 'Engine': 'R_glm'}*

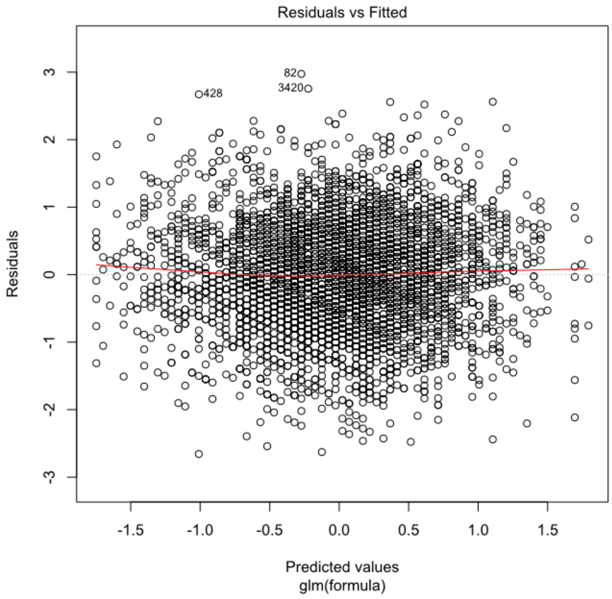


Response: normexam remove

Explanatory variables: cons,standlrt remove

Choose estimation engine: R_glm

Clicking on **Run** will return results in *ModelResults* as usual. There are additional graphical plots that come back from R; for example, below is a plot of residuals of the model fit against fitted values (*ResivsFitted.svg*).



Before finishing with R, we will also demonstrate a non-model Stat-JR template which interoperates with R called **PlotsViaR**; this gives the Stat-JR user access to R's lattice ((Sarkar, 2008)) graphics package through the Stat-JR interface.

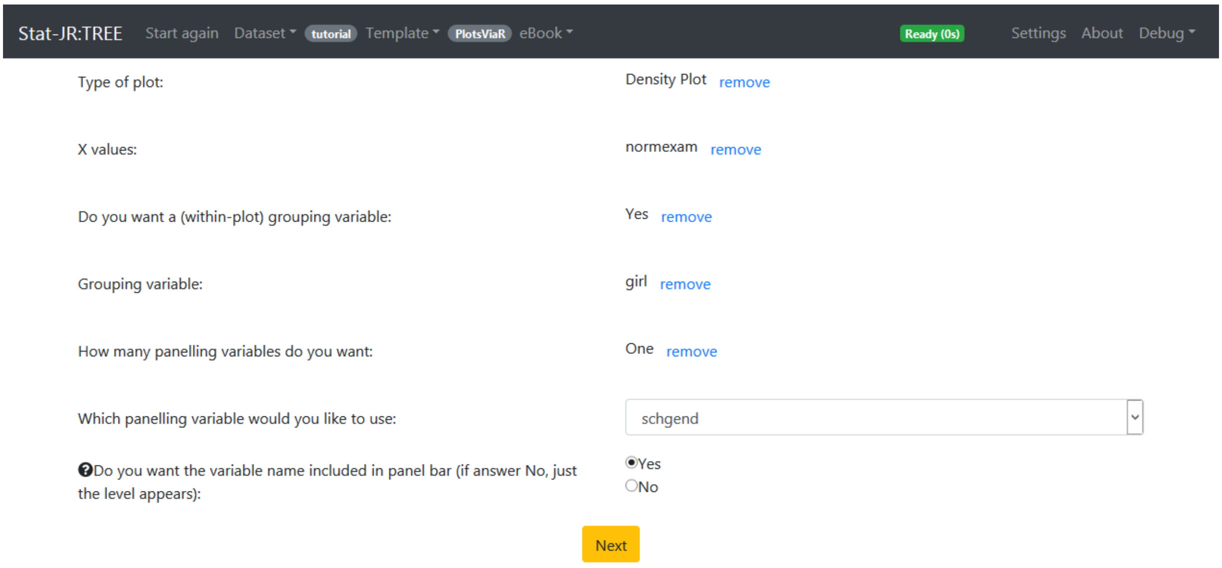
Click on **Choose** from the **Template** pull-down list at the top of the screen to get a list of all the templates. Note that the search cloud is useful with interoperability as it can be used to show which templates offer interoperability with a particular package (the engines are in red).

Click on **Plots** and also **R_script** in the blue tag cloud. You'll see that the list of templates, underneath, is accordingly reduced to just those that draw plots using R.

Select **PlotsViaR** from the list, and click **Use**.

Set up the template inputs as shown below:

Dataset: *tutorial;* **Template:** *PlotsViaR;* **Input string**: *{'var1': 'normexam', 'Gp': 'Yes', 'group': 'girl', 'trellis1': 'schgend', 'howmany': 'One', 'plottypeGUI': 'Density Plot', 'striptitle': 'Yes'}*



Type of plot: Density Plot remove

X values: normexam remove

Do you want a (within-plot) grouping variable: Yes remove

Grouping variable: girl remove

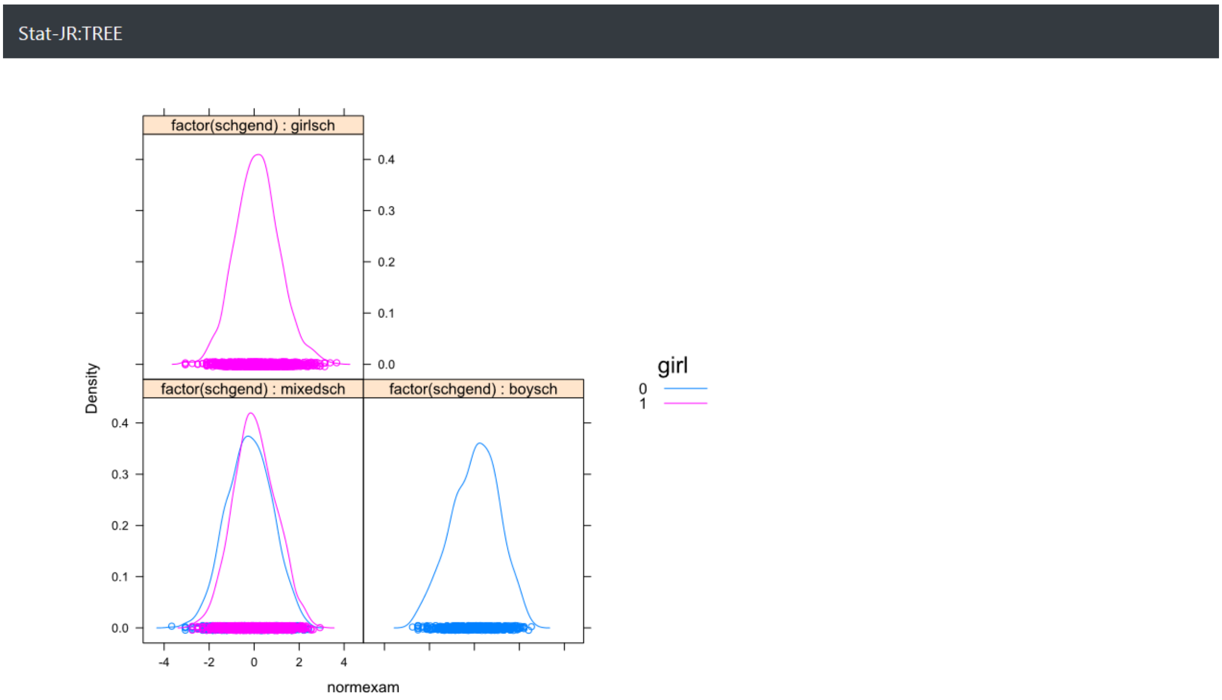
How many panelling variables do you want: One remove

Which panelling variable would you like to use: schgend

Do you want the variable name included in panel bar (if answer No, just the level appears): ● Yes ○ No

Next

These options will display kernel plots for the exam scores of pupils grouped by gender, with separate (panelled or trellised) plots for each school gender type. We can now press **Run** and show the plot (*Plot1.svg*) in a separate tab:



4.8 Interoperability with aML

We will next look at another software package that can fit many statistical models via likelihood-based estimation. aML (Lillard & Panis, 2003) is very useful for fitting multi-process models, but as with other software packages can fit a simple regression as a special case. In our development work on Stat-JR we have written special templates for interoperability with aML as opposed to incorporating interoperability in the standard templates. We therefore need to do the following:

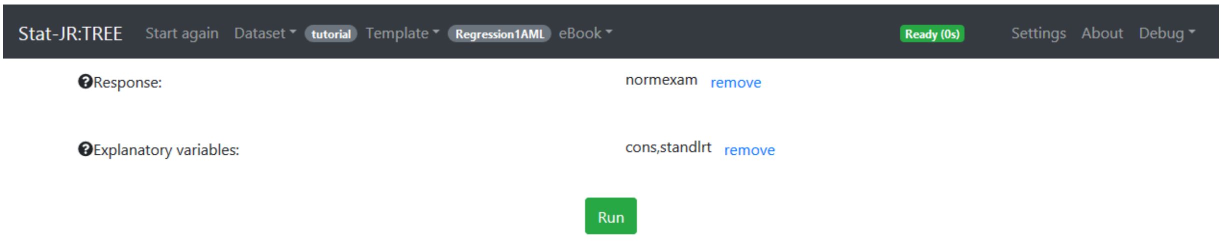
Click on the **Choose** option from the Template pull-down list.

Select **Regression1AML** from the template list and click on **Use**, and stick with the **tutorial** dataset.

Note that if you have earlier clicked on **Plots** and **R_script** in the cloud of terms you will need to either unselect them or click on **[reset]** to see the required template.

Fill in the inputs as follows, and press **Next**:

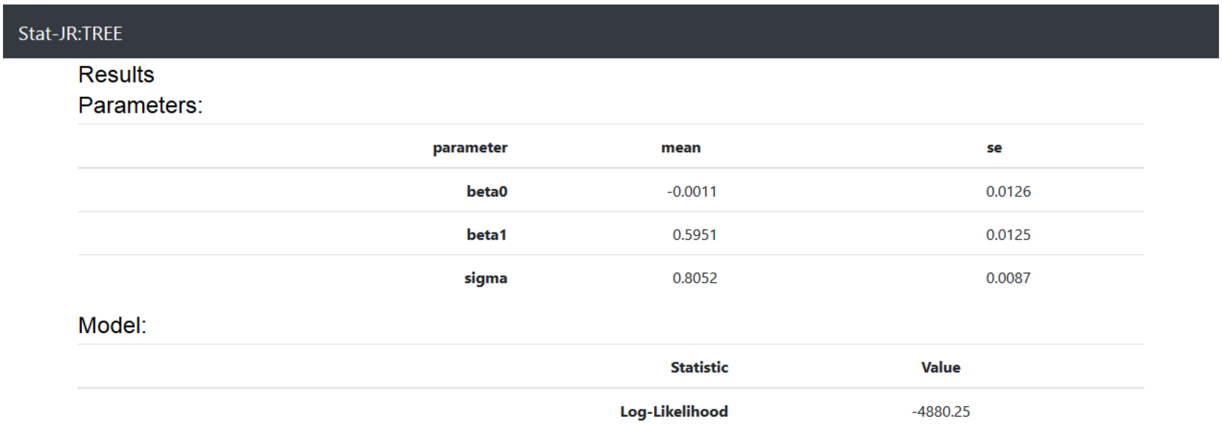
Dataset: *tutorial;* **Template:** *Regression1AML;* **Input string**: *{'y': 'normexam', 'x': 'cons,standlrt'}*



Response: normexam remove

Explanatory variables: cons,standlrt remove

Now click on **Run** to run the model in AML and select *ModelResults* from the list:



Results

Parameters:

parameter	mean	se
beta0	-0.0011	0.0126
beta1	0.5951	0.0125
sigma	0.8052	0.0087

Model:

Statistic	Value
Log-Likelihood	-4880.25

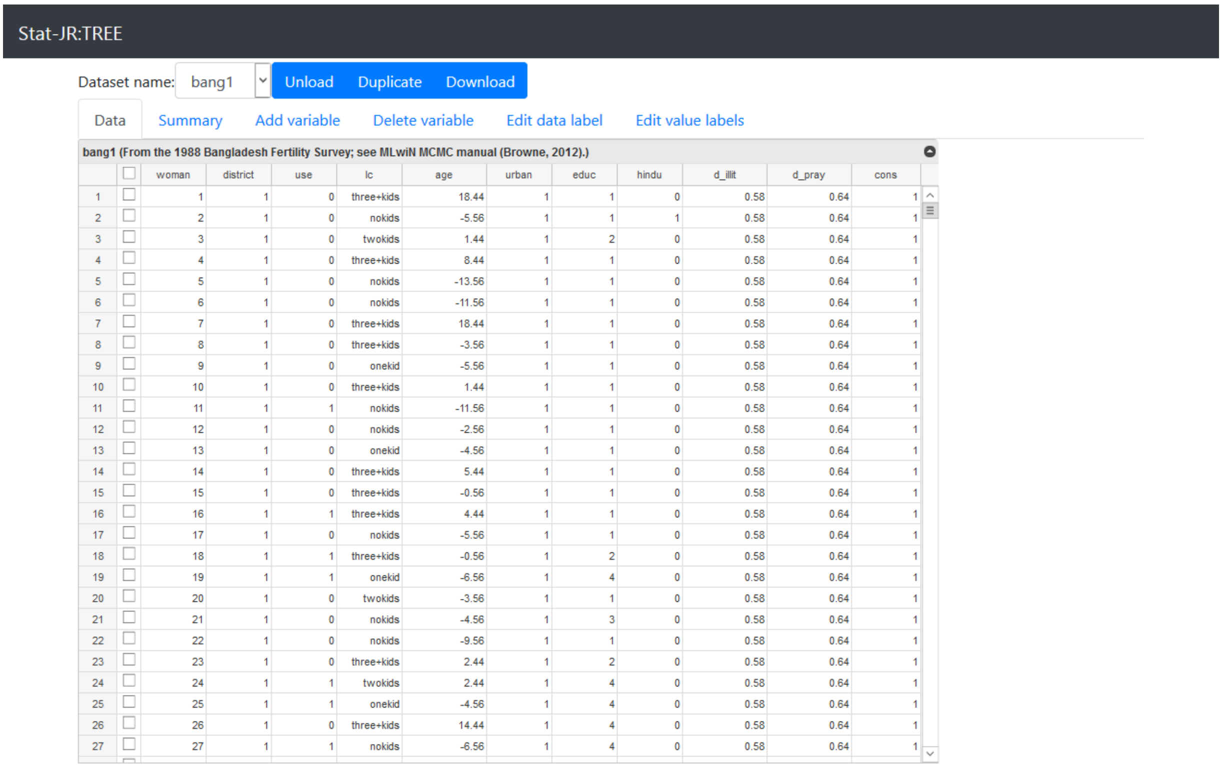
Here we see the model results are similar to other packages. AML has three input datasets: *amlfit.raw*, *amlfit.aml* and *amlfit.r2a*. There are also three additional output files from AML: *amlfit.out, amlfit.tab* and *amlfit.sum*. For more information on how AML works we recommend looking at the reference manual for the software.

5 Application 2: Analysis of the Bangladeshi Fertility Survey dataset

5.1 The Bangladeshi Fertility Survey dataset

The Bangladeshi dataset (**bang1**) is an example dataset from the 1988 Bangladeshi Fertility Survey that is also considered in (Browne, MCMC Estimation in MLwiN, v2.36, 2016). It contains records from 1934 women based in 60 districts in Bangladesh, and we are planning to investigate variables that predict whether the women were using contraception or not at the time of the survey. Let us first look at the data and the variables we will consider.

Select **Choose** and pick **bang1** from the **Dataset** list and click on **Use.**
Click on **View** from the **Dataset** list to view the data as follows:



bang1 (From the 1988 Bangladesh Fertility Survey; see MLwiN MCMC manual (Browne, 2012).)

	□	woman	district	use	lc	age	urban	educ	hindu	d_illit	d_pray	cons
1	□	1	1	0	three+kids	18.44	1	1	0	0.58	0.64	1
2	□	2	1	0	nokids	-5.56	1	1	1	0.58	0.64	1
3	□	3	1	0	twokids	1.44	1	2	0	0.58	0.64	1
4	□	4	1	0	three+kids	8.44	1	1	0	0.58	0.64	1
5	□	5	1	0	nokids	-13.56	1	1	0	0.58	0.64	1
6	□	6	1	0	nokids	-11.56	1	1	0	0.58	0.64	1
7	□	7	1	0	three+kids	18.44	1	1	0	0.58	0.64	1
8	□	8	1	0	three+kids	-3.56	1	1	0	0.58	0.64	1
9	□	9	1	0	onekid	-5.56	1	1	0	0.58	0.64	1
10	□	10	1	0	three+kids	1.44	1	1	0	0.58	0.64	1
11	□	11	1	1	nokids	-11.56	1	1	0	0.58	0.64	1
12	□	12	1	0	nokids	-2.56	1	1	0	0.58	0.64	1
13	□	13	1	0	onekid	-4.56	1	1	0	0.58	0.64	1
14	□	14	1	0	three+kids	5.44	1	1	0	0.58	0.64	1
15	□	15	1	0	three+kids	-0.56	1	1	0	0.58	0.64	1
16	□	16	1	1	three+kids	4.44	1	1	0	0.58	0.64	1
17	□	17	1	0	nokids	-5.56	1	1	0	0.58	0.64	1
18	□	18	1	1	three+kids	-0.56	1	2	0	0.58	0.64	1
19	□	19	1	1	onekid	-6.56	1	4	0	0.58	0.64	1
20	□	20	1	0	twokids	-3.56	1	1	0	0.58	0.64	1
21	□	21	1	0	nokids	-4.56	1	3	0	0.58	0.64	1
22	□	22	1	0	nokids	-9.56	1	1	0	0.58	0.64	1
23	□	23	1	0	three+kids	2.44	1	2	0	0.58	0.64	1
24	□	24	1	1	twokids	2.44	1	4	0	0.58	0.64	1
25	□	25	1	1	onekid	-4.56	1	4	0	0.58	0.64	1
26	□	26	1	0	three+kids	14.44	1	4	0	0.58	0.64	1
27	□	27	1	1	nokids	-6.56	1	4	0	0.58	0.64	1

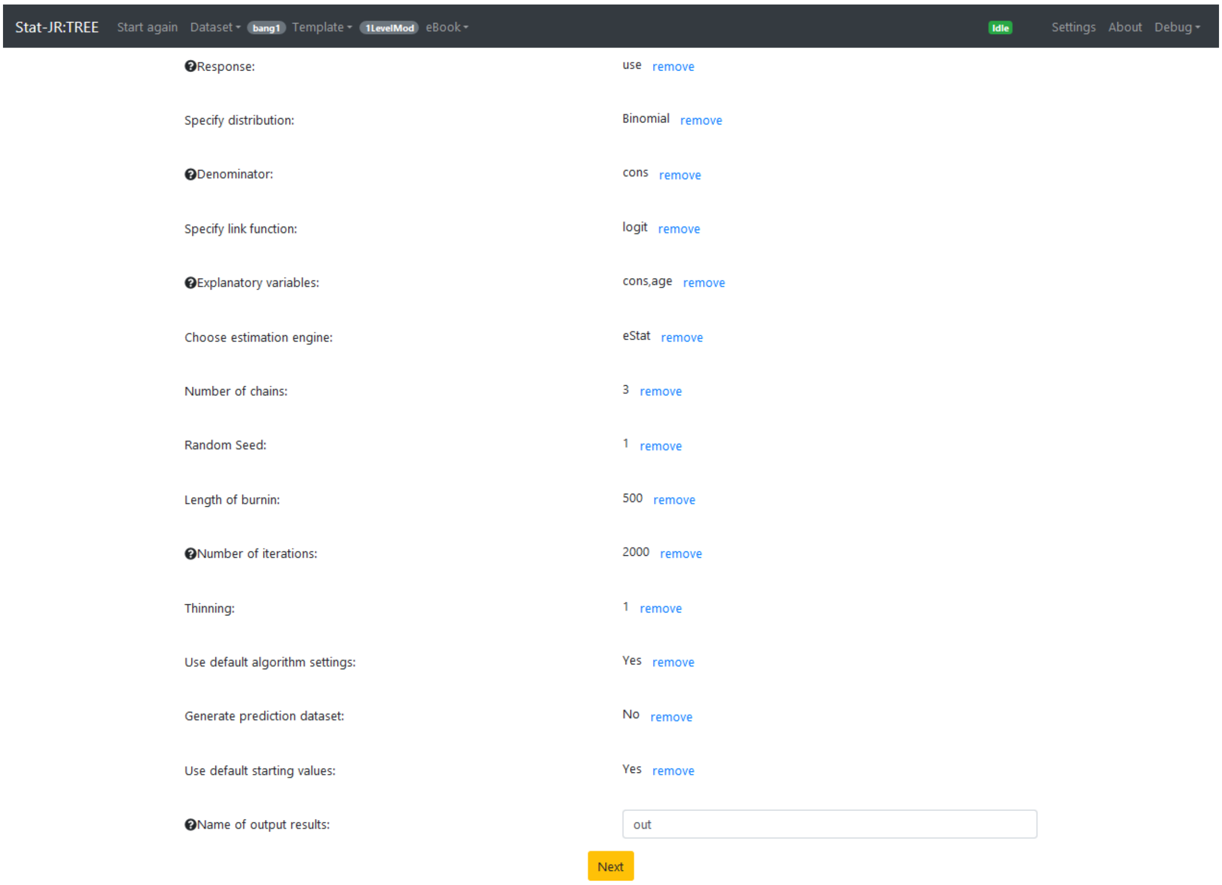
Here we see records for the first 27 women in district 1 displayed. The response variable *use* takes value 1 if the woman was using contraceptives during the time of the survey, and 0 if she was not. There are then several predictor variables, both woman-level and district-level. Here we will focus on just two: the number of living children (*lc*), which is a categorical variable with four categories (no kids, one kid, two kids, three+kids), and the respondents' *age*, which is measured to the nearest year and has been centred around its grand mean. We will now consider modelling the dataset.

5.2 Modelling the data using logistic regression

We will firstly consider a simple linear regression model relating contraception use to the age of the woman.

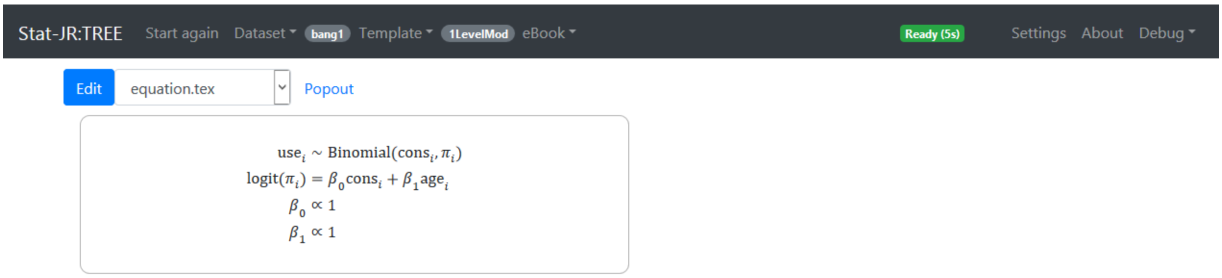
Choose the template **1LevelMod** from the **Template** list and click on **Use**.
Then setup the model with inputs as below.

Dataset: *bang1;* **Template:** *1LevelMod;* **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'}*



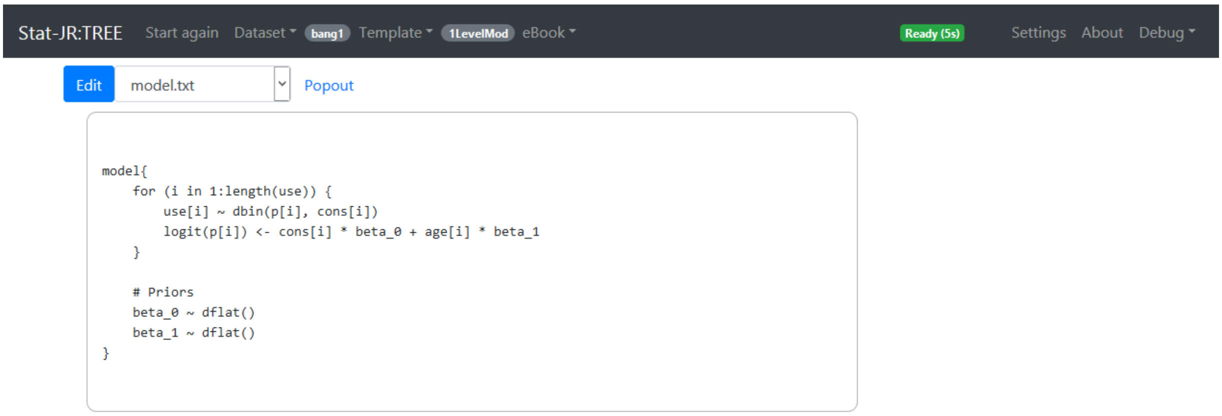
Response:	use remove
Specify distribution:	Binomial remove
Denominator:	cons remove
Specify link function:	logit remove
Explanatory variables:	cons,age remove
Choose estimation engine:	eStat remove
Number of chains:	3 remove
Random Seed:	1 remove
Length of burnin:	500 remove
Number of iterations:	2000 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

Clicking on **Next** and choosing *equation.tex* in the pull-down list and we see the following:



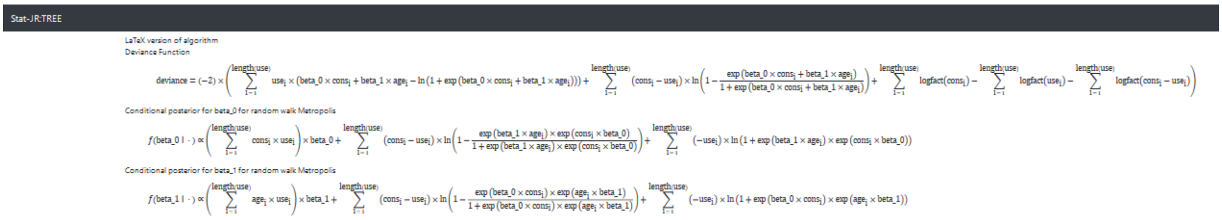
$$\begin{aligned}\mathrm{use}_i &\sim \mathrm{Binomial}(\mathrm{cons}_i, \pi_i)\\ \mathrm{logit}(\pi_i) &= \beta_0 \mathrm{cons}_i + \beta_1 \mathrm{age}_i\\ \beta_0 &\propto 1\\ \beta_1 &\propto 1\end{aligned}$$

Here we the logistic regression model, in LaTeX, in the output pane. If we select *model.txt* we can then see the model code that the algebra system will interpret:



```
model{
    for (i in 1:length(use)) {
        use[i] ~ dbin(p[i], cons[i])
        logit(p[i]) <- cons[i] * beta_0 + age[i] * beta_1
    }

    # Priors
    beta_0 ~ dflat()
    beta_1 ~ dflat()
}
```

Now choosing **algorithm.tex** from the output pane, and placing it in its own tab in the browser window, gives the following:



LaTeX version of algorithm

Deviance Function

$$\text{deviance} = (-2) \times \left(\sum_{i=1}^{\text{length}(\text{use})} \text{use}_i \times (\text{beta\_0} \times \text{cons}_i + \text{beta\_1} \times \text{age}_i - \ln(1 + \exp(\text{beta\_0} \times \text{cons}_i + \text{beta\_1} \times \text{age}_i))) + \sum_{i=1}^{\text{length}(\text{use})} (\text{cons}_i - \text{use}_i) \times \ln\left(1 - \frac{\exp(\text{beta\_0} \times \text{cons}_i + \text{beta\_1} \times \text{age}_i)}{1 + \exp(\text{beta\_0} \times \text{cons}_i + \text{beta\_1} \times \text{age}_i)}\right) + \sum_{i=1}^{\text{length}(\text{use})} \text{logfact}(\text{cons}_i) - \sum_{i=1}^{\text{length}(\text{use})} \text{logfact}(\text{use}_i) - \sum_{i=1}^{\text{length}(\text{use})} \text{logfact}(\text{cons}_i - \text{use}_i) \right)$$

Conditional posterior for beta_0 for random walk Metropolis

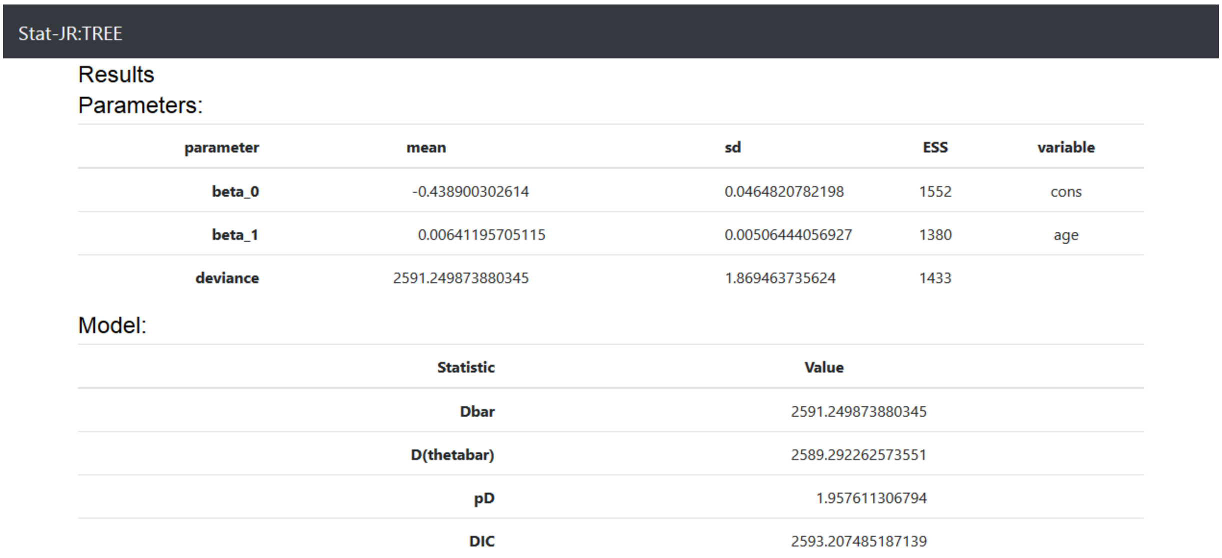
$$f(\text{beta\_0} \mid \cdot) \propto \left(\sum_{i=1}^{\text{length}(\text{use})} \text{cons}_i \times \text{use}_i \right) \times \text{beta\_0} + \sum_{i=1}^{\text{length}(\text{use})} (\text{cons}_i - \text{use}_i) \times \ln\left(1 - \frac{\exp(\text{beta\_1} \times \text{age}_i) \times \exp(\text{cons}_i \times \text{beta\_0})}{1 + \exp(\text{beta\_1} \times \text{age}_i) \times \exp(\text{cons}_i \times \text{beta\_0})}\right) + \sum_{i=1}^{\text{length}(\text{use})} (-\text{use}_i) \times \ln(1 + \exp(\text{beta\_1} \times \text{age}_i) \times \exp(\text{cons}_i \times \text{beta\_0}))$$

Conditional posterior for beta_1 for random walk Metropolis

$$f(\text{beta\_1} \mid \cdot) \propto \left(\sum_{i=1}^{\text{length}(\text{use})} \text{age}_i \times \text{use}_i \right) \times \text{beta\_1} + \sum_{i=1}^{\text{length}(\text{use})} (\text{cons}_i - \text{use}_i) \times \ln\left(1 - \frac{\exp(\text{beta\_0} \times \text{cons}_i) \times \exp(\text{age}_i \times \text{beta\_1})}{1 + \exp(\text{beta\_0} \times \text{cons}_i) \times \exp(\text{age}_i \times \text{beta\_1})}\right) + \sum_{i=1}^{\text{length}(\text{use})} (-\text{use}_i) \times \ln(1 + \exp(\text{beta\_0} \times \text{cons}_i) \times \exp(\text{age}_i \times \text{beta\_1}))$$

Whilst it is a little difficult to see in this screenshot, you will see better on your own screen that the eStat engine uses a different MCMC method, random walk Metropolis, for the steps for the fixed effects (*beta0* and *beta1*) when fitting logistic regression models. We will come back to this modelling decision in Section 5.4 when we compare different software packages.

Returning to the main pane and clicking on **Run** will now run the model.
Once it has finished, if we select *ModelResults* from the list, and look at it in a new tab, we get the following:



Results

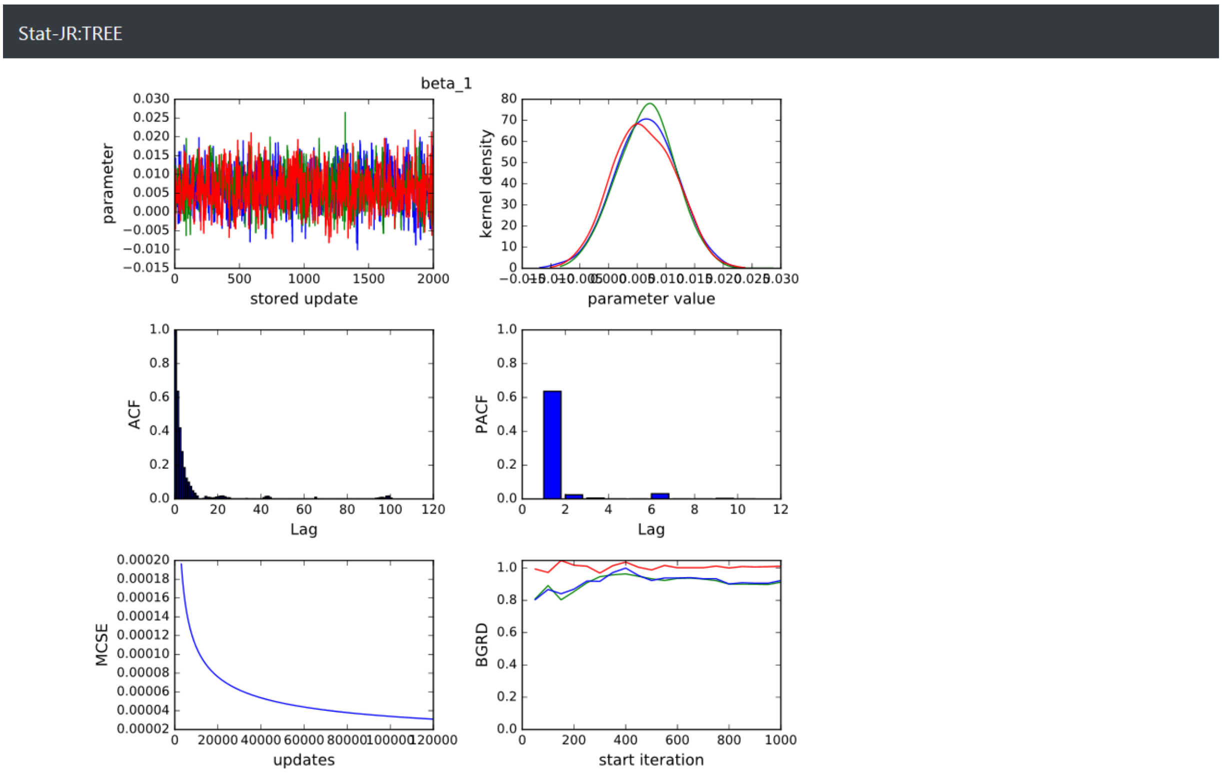
Parameters:

parameter	mean	sd	ESS	variable
beta_0	-0.438900302614	0.0464820782198	1552	cons
beta_1	0.00641195705115	0.00506444056927	1380	age
deviance	2591.249873880345	1.869463735624	1433	

Model:

Statistic	Value
Dbar	2591.249873880345
D(thetabar)	2589.292262573551
pD	1.957611306794
DIC	2593.207485187139

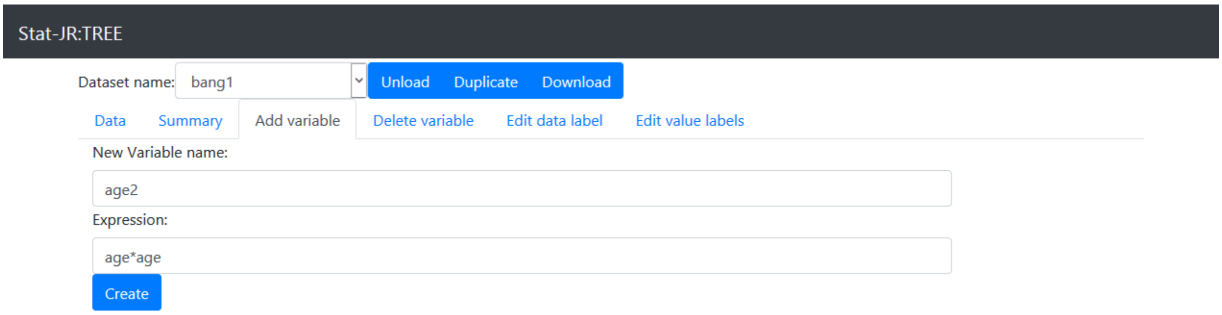
Age doesn’t appear to have a significant effect (its estimate (0.0064) is similar in magnitude to its standard error (0.0051)). To see this more clearly we can look at the graph *beta_1.svg* in its own browser tab:



Here, whilst the values on the x-axis overlap and therefore aren’t particularly clear, we can see that all three chains show strong support for the value 0.00 in the kernel density plot (i.e. it’s comfortably within the distribution). It might be the case, however, that contraceptive use has a non-linear relationship with age (possibly quadratic) and this could also be confounded by how far through their own family-formation process the woman is, which we will model via the variable *lc.* We might also be interested in accounting for any clustering effects of having women nested within districts.

In order to fit a quadratic function to age we will need to construct the variable age^2 which we can easily do via **Dataset > View** and using the variable creation tool.

Return to the main screen and select **View** from the **Dataset** pull-down list at the top of the page Click on the **Add Variable** tab and type the following (**New Variable name:** *age2*; **Expression:** *age*age*):

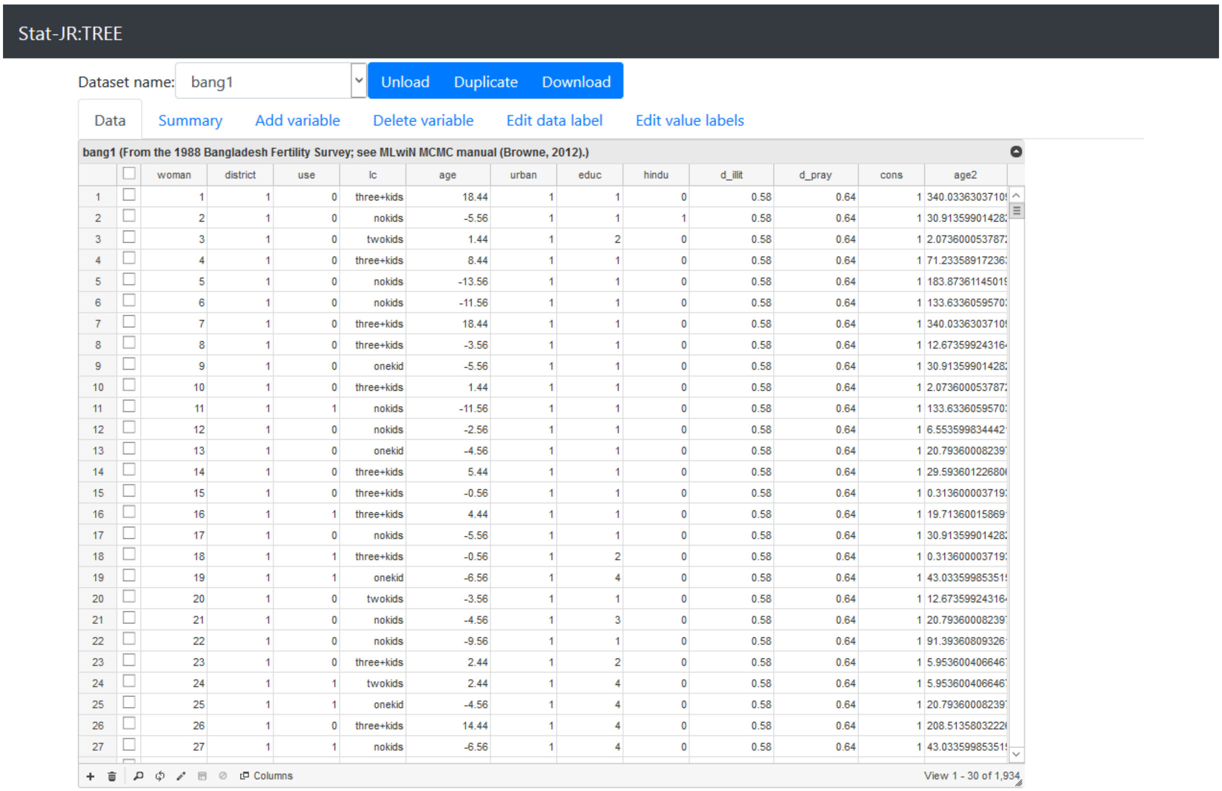


Dataset name: bang1

New Variable name: age2

Expression: age*age

Here we are going to overwrite the existing dataset (at least in temporary memory) with a version to which we have appended an additional column. Clicking on **Create** and looking at the data by (clicking on the **Data** tab) below gives the following:



bang1 (From the 1988 Bangladesh Fertility Survey; see MLwiN MCMC manual (Browne, 2012).)

	□	woman	district	use	lc	age	urban	educ	hindu	d_illit	d_pray	cons	age2
1	□	1	1	0	three+kids	18.44	1	1	0	0.58	0.64	1	340.0336303710
2	□	2	1	0	nokids	-5.56	1	1	1	0.58	0.64	1	30.91359901428
3	□	3	1	0	twokids	1.44	1	2	0	0.58	0.64	1	2.073600053787
4	□	4	1	0	three+kids	8.44	1	1	0	0.58	0.64	1	71.23358917236
5	□	5	1	0	nokids	-13.56	1	1	0	0.58	0.64	1	183.8736114501
6	□	6	1	0	nokids	-11.56	1	1	0	0.58	0.64	1	133.6336059570
7	□	7	1	0	three+kids	18.44	1	1	0	0.58	0.64	1	340.0336303710
8	□	8	1	0	three+kids	-3.56	1	1	0	0.58	0.64	1	12.67359924316
9	□	9	1	0	onekid	-5.56	1	1	0	0.58	0.64	1	30.91359901428
10	□	10	1	0	three+kids	1.44	1	1	0	0.58	0.64	1	2.073600053787
11	□	11	1	1	nokids	-11.56	1	1	0	0.58	0.64	1	133.6336059570
12	□	12	1	0	nokids	-2.56	1	1	0	0.58	0.64	1	6.553599834442
13	□	13	1	0	onekid	-4.56	1	1	0	0.58	0.64	1	20.79360008239
14	□	14	1	0	three+kids	5.44	1	1	0	0.58	0.64	1	29.59360122680
15	□	15	1	0	three+kids	-0.56	1	1	0	0.58	0.64	1	0.313600003719
16	□	16	1	1	three+kids	4.44	1	1	0	0.58	0.64	1	19.71360015869
17	□	17	1	0	nokids	-5.56	1	1	0	0.58	0.64	1	30.91359901428
18	□	18	1	1	three+kids	-0.56	1	2	0	0.58	0.64	1	0.313600003719
19	□	19	1	1	onekid	-6.56	1	4	0	0.58	0.64	1	43.03359985351
20	□	20	1	0	twokids	-3.56	1	1	0	0.58	0.64	1	12.67359924316
21	□	21	1	0	nokids	-4.56	1	3	0	0.58	0.64	1	20.79360008239
22	□	22	1	0	nokids	-9.56	1	1	0	0.58	0.64	1	91.39360809326
23	□	23	1	0	three+kids	2.44	1	2	0	0.58	0.64	1	5.953600406646
24	□	24	1	1	twokids	2.44	1	4	0	0.58	0.64	1	5.953600406646
25	□	25	1	1	onekid	-4.56	1	4	0	0.58	0.64	1	20.79360008239
26	□	26	1	0	three+kids	14.44	1	4	0	0.58	0.64	1	208.5135803222
27	□	27	1	1	nokids	-6.56	1	4	0	0.58	0.64	1	43.03359985351

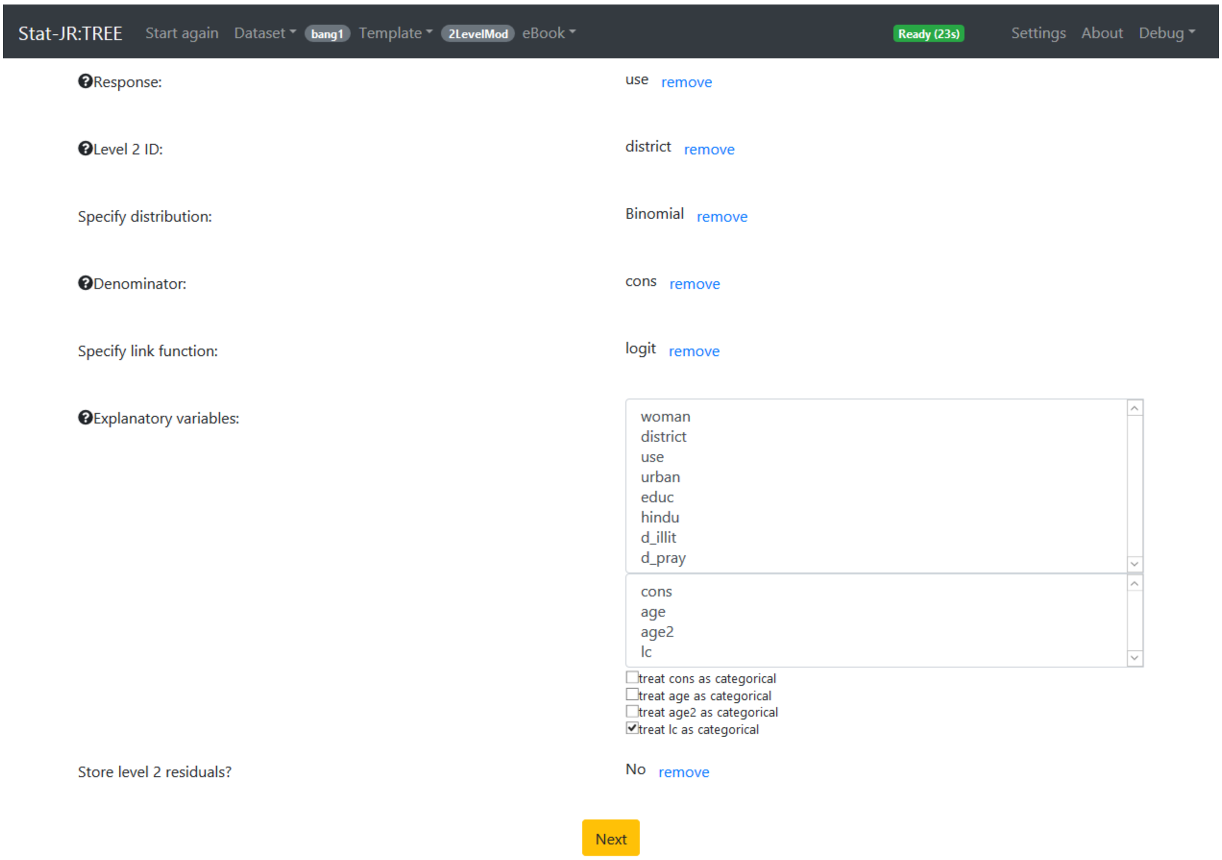
View 1 - 30 of 1,934

Here you see age2 (age^2) appearing in the column on the far right. Whilst we could explore adding further explanatory variables to this 1-level model, we are going to move straight into fitting a 2-level model to account for districts in which we will also investigate the effect of a quadratic function of age.

5.3 Multilevel modelling of the data

We will now require a template that will fit a 2-level logistic regression model to our dataset. In the earlier sections we looked at the template **2LevelMod** and we will once again use it here and also illustrate how to fit categorical predictor variables.

On the main tab, click on **Choose** in the **Template** pull-down list and select **2LevelMod** and click on **Use** button to run this template.
Fill in the template inputs as follows:



Response: use remove

Level 2 ID: district remove

Specify distribution: Binomial remove

Denominator: cons remove

Specify link function: logit remove

Explanatory variables:

woman
district
use
urban
educ
hindu
d_illit
d_pray

cons
age
age2
lc

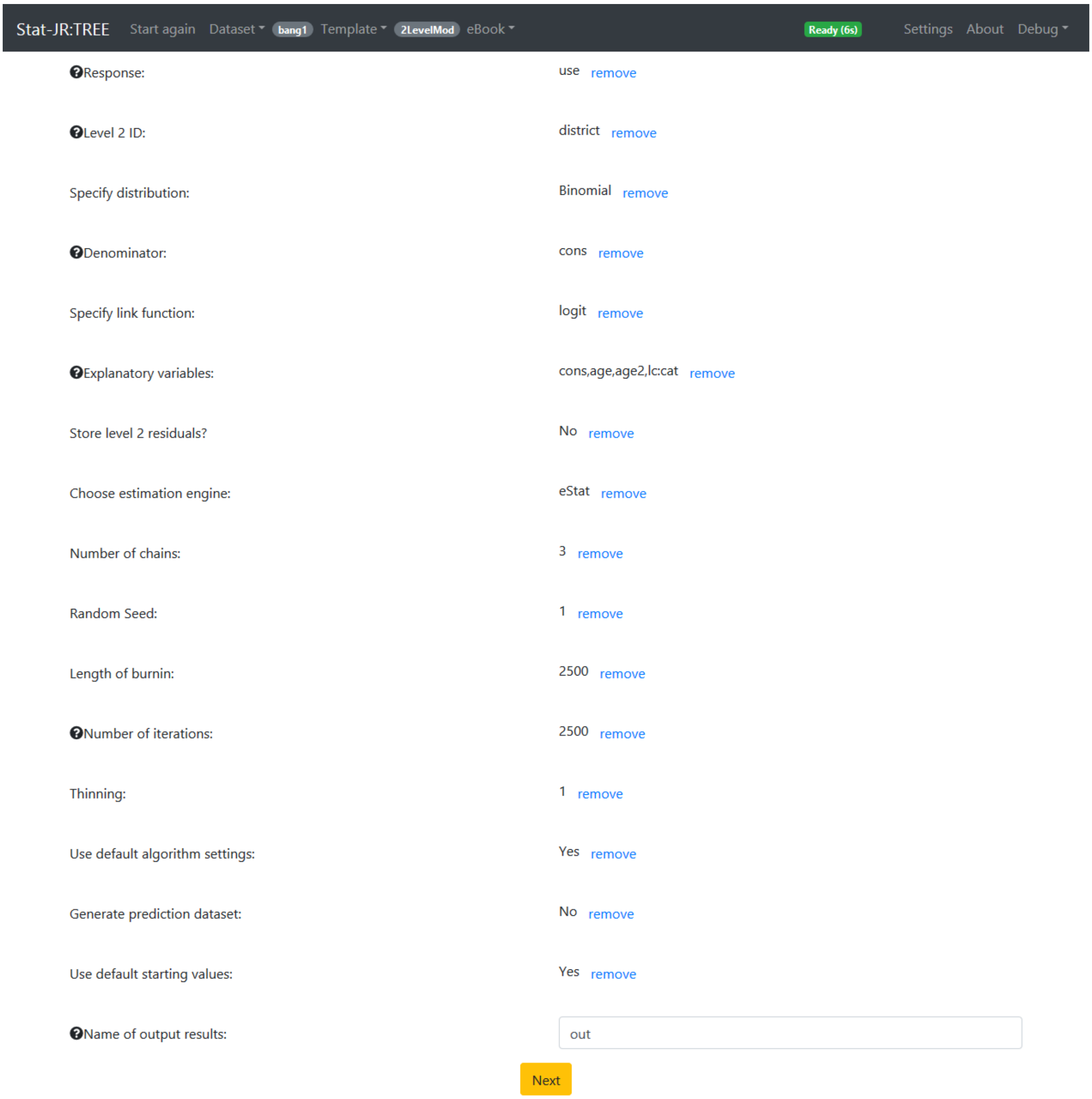
□ treat cons as categorical
□ treat age as categorical
□ treat age2 as categorical
☑ treat lc as categorical

Store level 2 residuals? No remove

Next

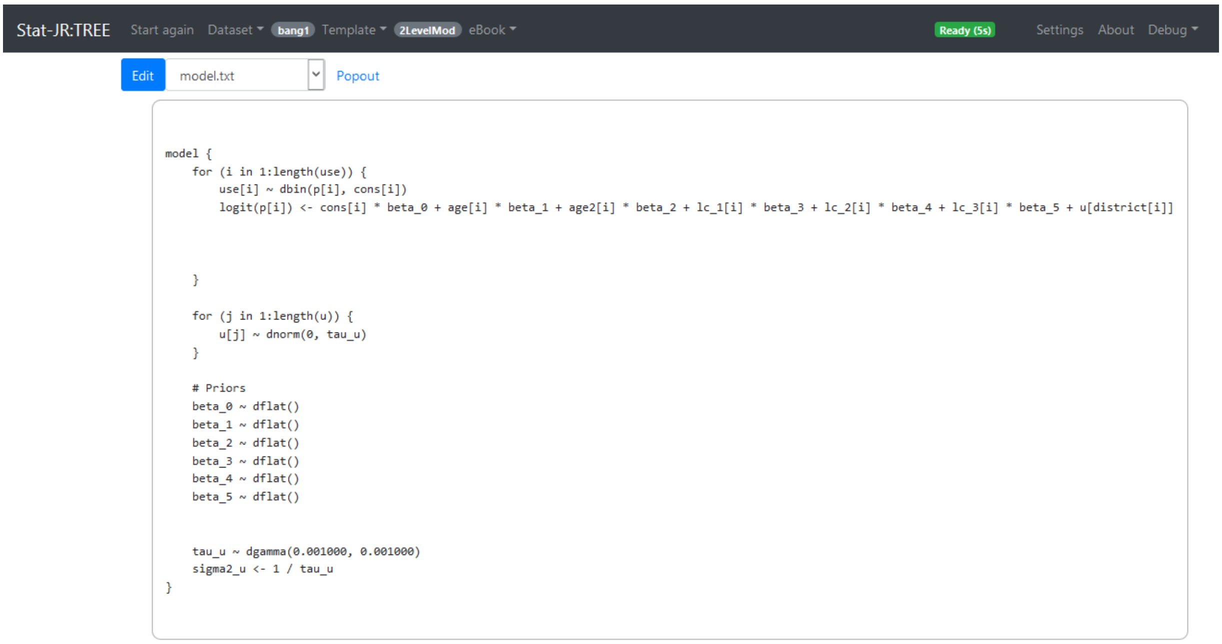
Here we need to specify several extra inputs, including an input for the level 2 identifiers and also to let the software know which predictor variables are categorical (by ticking the box indicating that the variable *lc* is categorical). Continue with the inputs as follows:

Dataset: *bang1;* **Template:** *2LevelMod;* **Input string:** *{'D': 'Binomial', 'storeresid': 'No', 'nchains': '3', 'link': 'logit', 'defaultalg': 'Yes', 'iterations': '2500', 'outdata': 'out', 'seed': '1', 'defaultsv': 'Yes', 'Engine': 'eStat', 'L2ID': 'district', 'burnin': '2500', 'n': 'cons', 'thinning': '1', 'y': 'use', 'x': 'cons,age,age2,lc:cat', 'makepred': 'No'}*



❓Response:	use
❓Level 2 ID:	district
Specify distribution:	Binomial
❓Denominator:	cons
Specify link function:	logit
❓Explanatory variables:	cons,age,age2,lc:cat
Store level 2 residuals?	No
Choose estimation engine:	eStat
Number of chains:	3
Random Seed:	1
Length of burnin:	2500
❓Number of iterations:	2500
Thinning:	1
Use default algorithm settings:	Yes
Generate prediction dataset:	No
Use default starting values:	Yes
❓Name of output results:	out

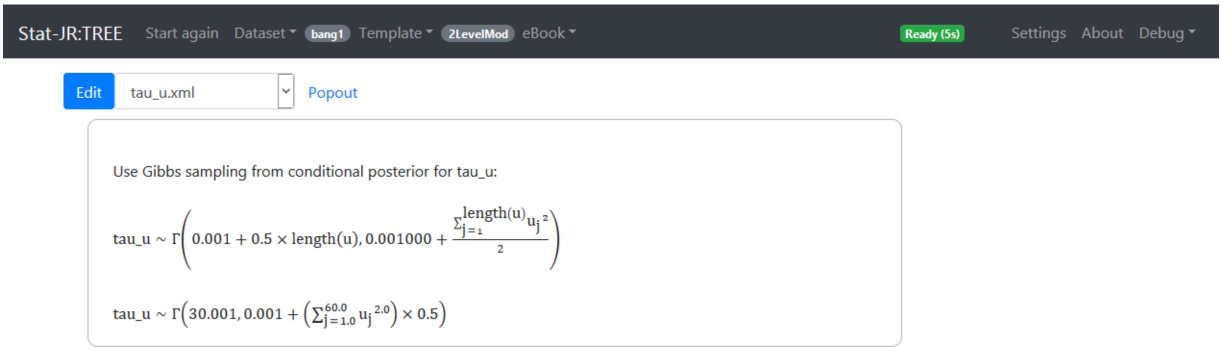
Clicking on **Next** will run the algebra system and set up code to fit the model. If we select *model.txt* in the output list we will see the following:



```
model {
    for (i in 1:length(use)) {
        use[i] ~ dbin(p[i], cons[i])
        logit(p[i]) <- cons[i] * beta_0 + age[i] * beta_1 + age2[i] * beta_2 + lc_1[i] * beta_3 + lc_2[i] * beta_4 + lc_3[i] * beta_5 + u[district[i]]



    }

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

    # Priors
    beta_0 ~ dflat()
    beta_1 ~ dflat()
    beta_2 ~ dflat()
    beta_3 ~ dflat()
    beta_4 ~ dflat()
    beta_5 ~ dflat()


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

Here we see the more complicated model code for this 2-level model in the output pane. Note that the *lc* predictor is treated as categorical and thus appears as three dummy variables (*lc_1*, *lc_2, lc_3*)

If we select *tau_u.xml* in the output list we will see the following:



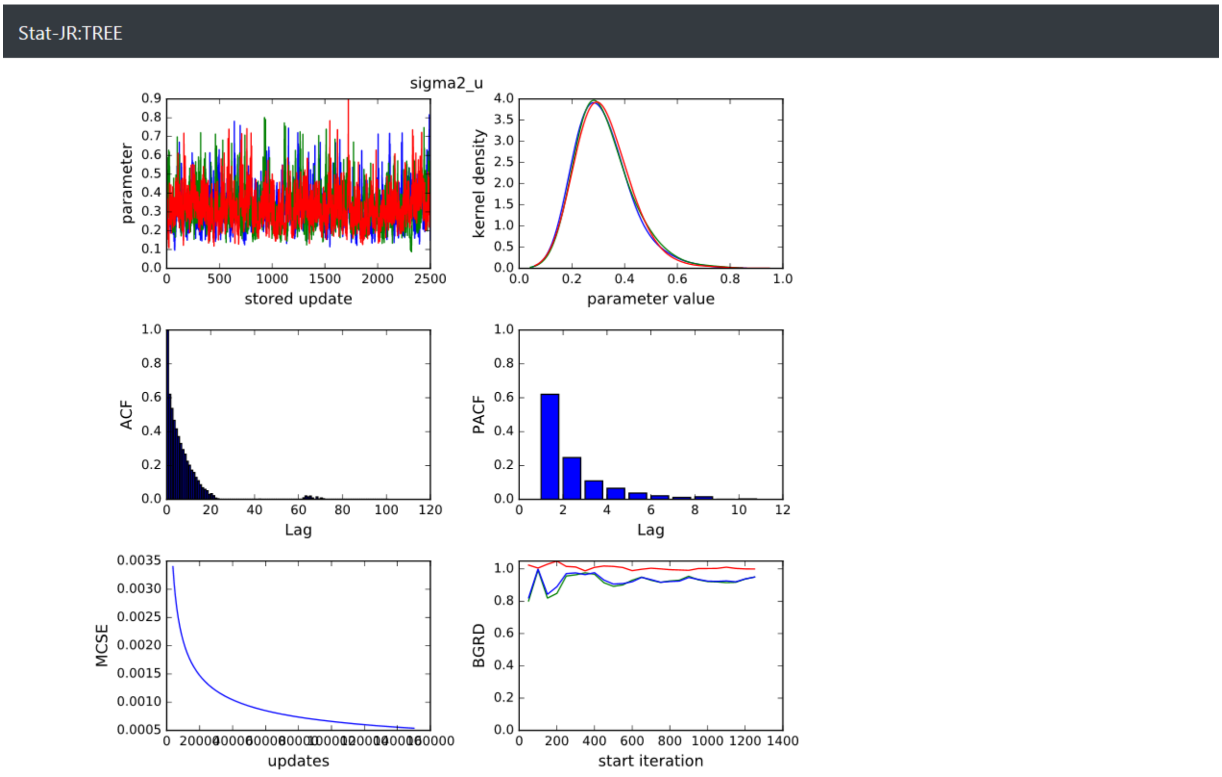
Use Gibbs sampling from conditional posterior for tau_u:

$$\mathrm{tau\_u} \sim \Gamma\left(0.001 + 0.5 \times \mathrm{length(u)}, 0.001000 + \frac{\sum_{j=1}^{\mathrm{length(u)}} u_j{}^2}{2}\right)$$

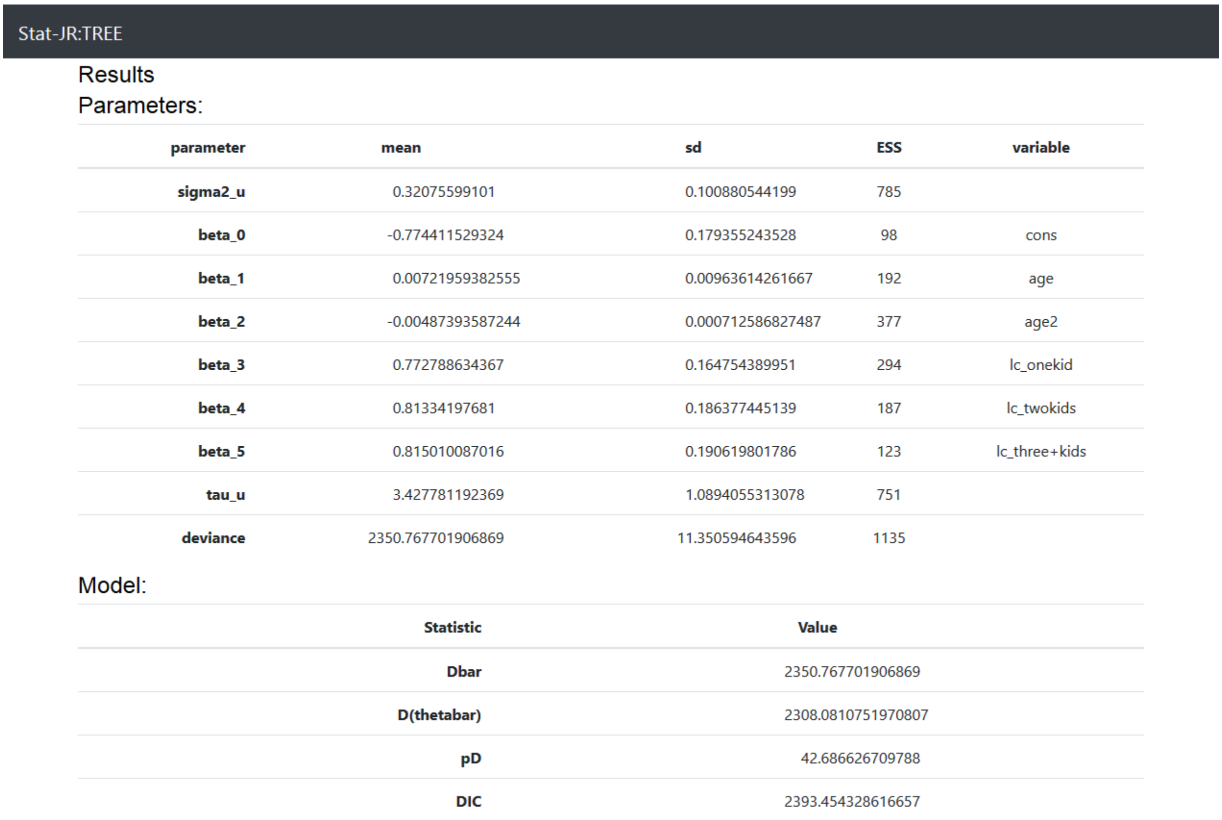
$$\mathrm{tau\_u} \sim \Gamma\left(30.001, 0.001 + \left(\sum_{j=1.0}^{60.0} u_j{}^{2.0}\right) \times 0.5\right)$$

Here we see the algorithm step for the parameter *tau_u*. Although most parameters in this model are updated by Random Walk Metropolis sampling, this parameter is updated by Gibbs Sampling as its conditional posterior distribution has a standard form.

If we now click on **Run** then after 43s (on a machine with Intel Xeon ES-2699 v4; this includes time for compiling and adapting) the model will have run and if we select *sigma2_u.svg* we will see the following:



Here we can see that convergence and mixing, for this parameter at least, are reasonable. In fact, if we look at the diagnostic plots for the other parameters, we see similar convergence there as well. Next we can look at *ModelResults* in its own tab to see the parameter estimates:



Results

Parameters:

parameter	mean	sd	ESS	variable
sigma2_u	0.32075599101	0.100880544199	785	
beta_0	-0.774411529324	0.179355243528	98	cons
beta_1	0.00721959382555	0.00963614261667	192	age
beta_2	-0.00487393587244	0.000712586827487	377	age2
beta_3	0.772788634367	0.164754389951	294	lc_onekid
beta_4	0.81334197681	0.186377445139	187	lc_twokids
beta_5	0.815010087016	0.190619801786	123	lc_three+kids
tau_u	3.427781192369	1.0894055313078	751	
deviance	2350.767701906869	11.350594643596	1135	

Model:

Statistic	Value
Dbar	2350.767701906869
D(thetabar)	2308.0810751970807
pD	42.686626709788
DIC	2393.454328616657

Here we see that *beta_2* (the coefficient estimate for *age2*) is significant and negative (and larger than it's sd) suggesting a quadratic fit to the age predictor. As the data is centred around its mean, this implies that contraceptive use is reduced the further from the mean age the woman is. We will look at this in more detail at the end of the chapter.

The parameters *beta_3* to *beta_5* are all significant, and positive (and of similar magnitude), which suggests that women with children are more likely to use contraceptives than those without (since the reference category here is *nokids*). The parameter *sigma2_u* is fairly large, suggesting there are differences between districts in terms of contraceptive use.

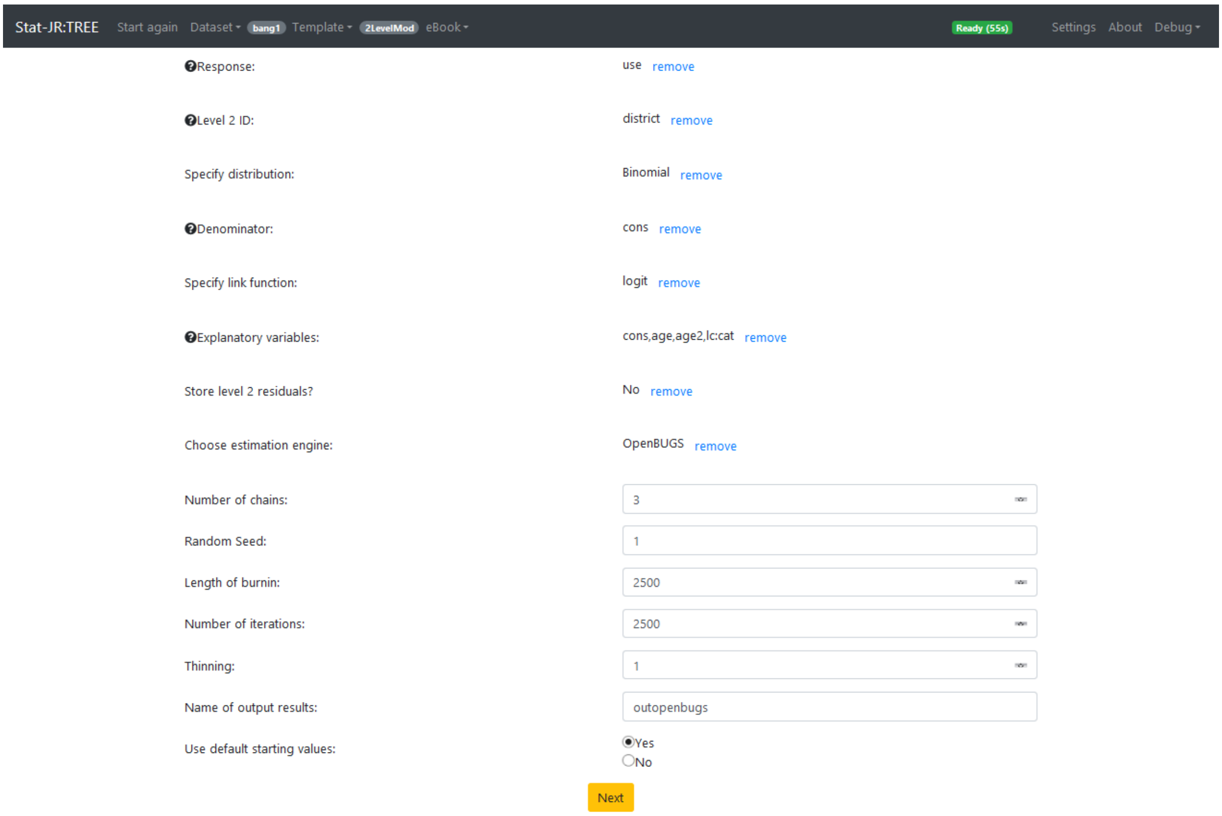
What is slightly disappointing here are the ESS values for all the fixed parameters. We have run each chain, after burnin, for 2,500 iterations resulting in a total of 7,500 actual iterations (i.e. from 3 chains) but the effective sample sizes are of the order of 100-300. As this indicates, the default algorithm in eStat – random walk Metropolis – is not very efficient for this example. We will look at two possible solutions in the next two sections.

5.4 Comparison between software packages

Not all software packages fit the same MCMC algorithm for this model. So here we will show how to fit the same model in another package, OpenBUGS ((Lunn, Spiegelhalter, Thomas, & Best, 2009)), which uses a different method: namely multivariate updating for the fixed effects in a GLMM, as developed by (Gamerman, 1997). This method results in slower estimation, but, as we will see, far better ESS. We will then look at a table comparing all the possible MCMC algorithms in the different packages for this model, which you can verify for yourselves.

To fit the model in OpenBUGS click on the **remove** text next to **Choose estimation engine** and set-up the model as follows:

Dataset: *bang1;* **Template:** *2LevelMod;* **Input string:** *{'Engine': 'OpenBUGS', 'L2ID': 'district', 'burnin': '2500', 'D': 'Binomial', 'outdata': 'outopenbugs', 'storeresid': 'No', 'n': 'cons', 'nchains': '3', 'thinning': '1', 'link': 'logit', 'iterations': '2500', 'y': 'use', 'x': 'cons,age,age2,lc:cat', 'seed': '1', 'defaultsv': 'Yes'}*



Response: use remove

Level 2 ID: district remove

Specify distribution: Binomial remove

Denominator: cons remove

Specify link function: logit remove

Explanatory variables: cons,age,age2,lc:cat remove

Store level 2 residuals? No remove

Choose estimation engine: OpenBUGS remove

Number of chains: 3

Random Seed: 1

Length of burnin: 2500

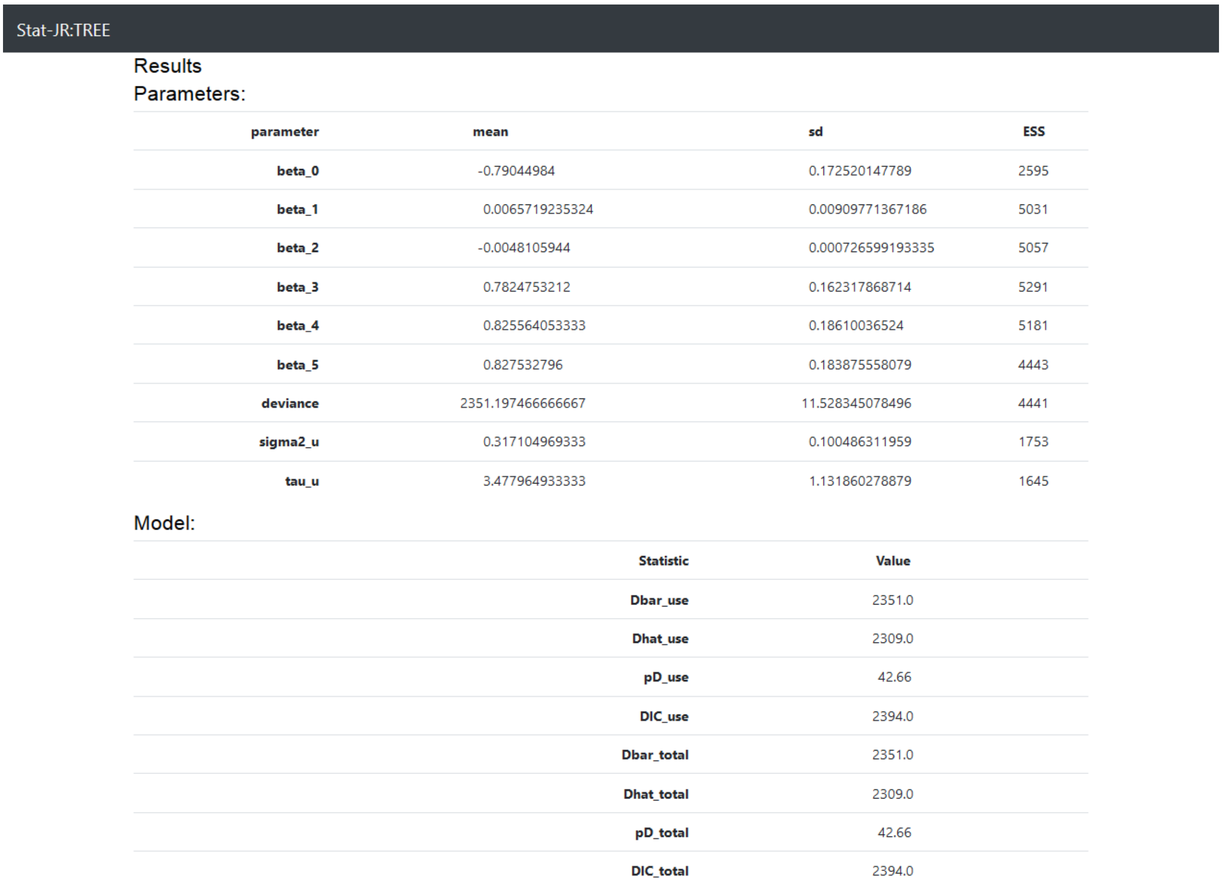
Number of iterations: 2500

Thinning: 1

Name of output results: outopenbugs

Use default starting values: ● Yes ○ No

Clicking on **Next** and **Run** will (after 2 min 49s on my machine) give the following, having selected *ModelResults* from the drop-down box above the output pane, and opening it in a new tab:



Results

Parameters:

parameter	mean	sd	ESS
beta_0	-0.79044984	0.172520147789	2595
beta_1	0.0065719235324	0.00909771367186	5031
beta_2	-0.0048105944	0.000726599193335	5057
beta_3	0.7824753212	0.162317868714	5291
beta_4	0.825564053333	0.18610036524	5181
beta_5	0.827532796	0.183875558079	4443
deviance	2351.197466666667	11.528345078496	4441
sigma2_u	0.317104969333	0.100486311959	1753
tau_u	3.477964933333	1.131860278879	1645

Model:

Statistic	Value
Dbar_use	2351.0
Dhat_use	2309.0
pD_use	42.66
DIC_use	2394.0
Dbar_total	2351.0
Dhat_total	2309.0
pD_total	42.66
DIC_total	2394.0

Here we see far better effective sample size values, with runs of 7,500 iterations translating into ESS values of between 2,500 and 5,500 for the beta parameters.

We can repeat this analysis using WinBUGS, JAGS and MLwiN with the same run lengths. Note for JAGS you will need to edit the initial value files or it will not run. To do this view each in the output window and click on the **Edit** button. If you change the value for beta_2 (the fixed effect associated with age2) from 0.1 to 0.0 in all three initial values files and click **Save** each time then JAGS should run. It should also be noted here that results may vary a little if you have different versions of the third party software packages or have changed options in them.

We could also fit the model using the MCMCglmm package in R, although here we would need to run a single chain and logistic regression models for binary data are the one GLMM where the answers can be a little different as it assumes over-dispersion which is inappropriate in this case.

The table overleaf[2] details the results of fitting many of these options (unless otherwise stated, each with **Dataset:** *bang1;* **Template:** *2LevelMod*):

Input string – eStat: *{'D': 'Binomial', 'storeresid': 'No', 'nchains': '3', 'link': 'logit', 'defaultalg': 'Yes', 'iterations': '2500', 'outdata': 'out', 'seed': '1', 'defaultsv': 'Yes', 'Engine': 'eStat', 'L2ID': 'district', 'burnin': '2500', 'n': 'cons', 'thinning': '1', 'y': 'use', 'x': 'cons,age,age2,lc:cat', 'makepred': 'No'}*

[2] This particular comparison used WinBUGS 1.4.3, OpenBUGS 3.2.3, JAGS 4.3.0 (64-bit), MLwiN 3.01, all run on Windows 64-bit machine with Intel Xeon ES-2699 v4; eStat times are of the form: *including compiling time (excluding compiling time)*.

Input string - WinBUGS: *{'Engine': 'WinBUGS', 'L2ID': 'district', 'burnin': '2500', 'D': 'Binomial', 'outdata': 'outwinbugs', 'storeresid': 'No', 'n': 'cons', 'nchains': '3', 'thinning': '1', 'link': 'logit', 'iterations': '2500', 'y': 'use', 'x': 'cons,age,age2,lc:cat', 'seed': '1', 'defaultsv': 'Yes'}*

Input string - OpenBUGS: {'Engine': 'OpenBUGS', 'L2ID': 'district', 'burnin': '2500', 'D': 'Binomial', 'outdata': 'outopenbugs', 'storeresid': 'No', 'n': 'cons', 'nchains': '3', 'thinning': '1', 'link': 'logit', 'iterations': '2500', 'y': 'use', 'x': 'cons,age,age2,lc:cat', 'seed': '1', 'defaultsv': 'Yes'}

Input string - JAGS (remember to change the initial values files before running– see above): *{'Engine': 'JAGS', 'L2ID': 'district', 'burnin': '2500', 'D': 'Binomial', 'outdata': 'outjags', 'storeresid': 'No', 'n': 'cons', 'nchains': '3', 'thinning': '1', 'link': 'logit', 'iterations': '2500', 'y': 'use', 'x': 'cons,age,age2,lc:cat', 'seed': '1', 'defaultsv': 'No', 'beta_sv_0': '0.1,0.1,0,0.1,0.1,0.1', 'beta_sv_1': '0.1,0.1,0,0.1,0.1,0.1', 'beta_sv_2': '0.1,0.1,0,0.1,0.1,0.1'}*

Input string – MLwiN: *{'Engine': 'MLwiN_MCMC', 'L2ID': 'district', 'burnin': '2500', 'D': 'Binomial', 'outdata': 'outmlwin', 'storeresid': 'No', 'n': 'cons', 'nchains': '3', 'thinning': '1', 'link': 'logit', 'defaultalg': 'Yes', 'iterations': '2500', 'y': 'use', 'x': 'cons,age,age2,lc:cat', 'seed': '1'}*

Input string – eStat – orthogonal parameterisation (see Section 5.5); Template: *NLevelOrthogParamRS*: *{'Engine': 'eStat', 'x1': 'cons', 'burnin': '2500', 'D': 'Binomial', 'outdata': 'outorthog', 'storeresid': 'No', 'thinning': '1', 'n': 'cons', 'nchains': '3', 'orthtype': 'Orthogonal', 'link': 'logit', 'defaultalg': 'Yes', 'iterations': '2500', 'y': 'use', 'x': 'cons,age,age2,lc:cat', 'C1': 'district', 'NumLevs': '1', 'seed': '1', 'useorthog': 'Yes', 'makepred': 'Yes', 'defaultsv': 'Yes'}*

Parameter		eStat	WinBUGS	OpenBUGS	JAGS	MLwiN	eStat orthogonal
Beta0	coeff(sd)	-0.774(0.179)	-0.790(0.176)	-0.790(0.173)	-0.791(0.172)	-0.801(0.183)	-0.782(0.176)
	ESS	98	418	2595	4826	127	878
Beta1	coeff(sd)	0.00722(0.00964)	0.00663(0.00930)	0.00657(0.00910)	0.00656(0.00936)	0.00623(0.00924)	0.00622(0.00937)
	ESS	192	763	5031	4616	275	1855
Beta2	coeff(sd)	-0.00487(0.000713)	-0.00482(0.000737)	-0.00481(0.000727)	-0.00480(0.000724)	-0.00478(0.000730)	-0.00481(0.000710)
	ESS	377	1154	5057	4237	334	1863
Beta3	coeff(sd)	0.773(0.164)	0.782(0.162)	0.782(0.162)	0.782(0.161)	0.789(0.167)	0.781(0.165)
	ESS	294	1021	5291	4965	255	1678
Beta4	coeff(sd)	0.815(0.191)	0.825(0.184)	0.826(0.186)	0.826(0.185)	0.837(0.188)	0.831(0.192)
	ESS	187	745	5181	4705	239	1701
Beta5	coeff(sd)	0.815(0.191)	0.828(0.185)	0.828(0.184)	0.829(0.185)	0.840(0.188)	0.833(0.186)
	ESS	123	512	4443	4758	167	1770
Sigma2u	coeff(sd)	0.321(0.101)	0.318(0.102)	0.317(0.100)	0.321(0.103)	0.315(0.103)	0.323(0.106)
	ESS	751	1809	1753	2108	738	702
Pd		42.69	42.55	42.66	42.01	42.22	43.02
DIC		2393.45	2393.49	2394.0	2392.62	2393.47	2393.91
Time (s)		90 (63)	264	169	74	10	77 (56)

In summary we see that MLwiN is by far the fastest of the packages, with eStat quicker than the other three as well although similar to JAGS. Both MLwiN and eStat use the simple random walk Metropolis algorithm, which is not the best method for this model and gives fairly poor ESS. Interestingly, both WinBUGS and OpenBUGS use the Gamerman method, but in this case OpenBUGS performs better in terms of time taken and ESS. This is somewhat puzzling as when each is run with a single chain, their performance is almost identical. Finally, JAGS is faster than the two BUGS packages with ESS generally as good as OpenBUGS too; however, there have been many comparisons between JAGS and BUGS for different models, and which method is better varies from model to model and versions of the packages, so we need to take care when making comparisons based on just one example. The final column shows another eStat method which we will discuss next.

5.5 Orthogonal parameterisation

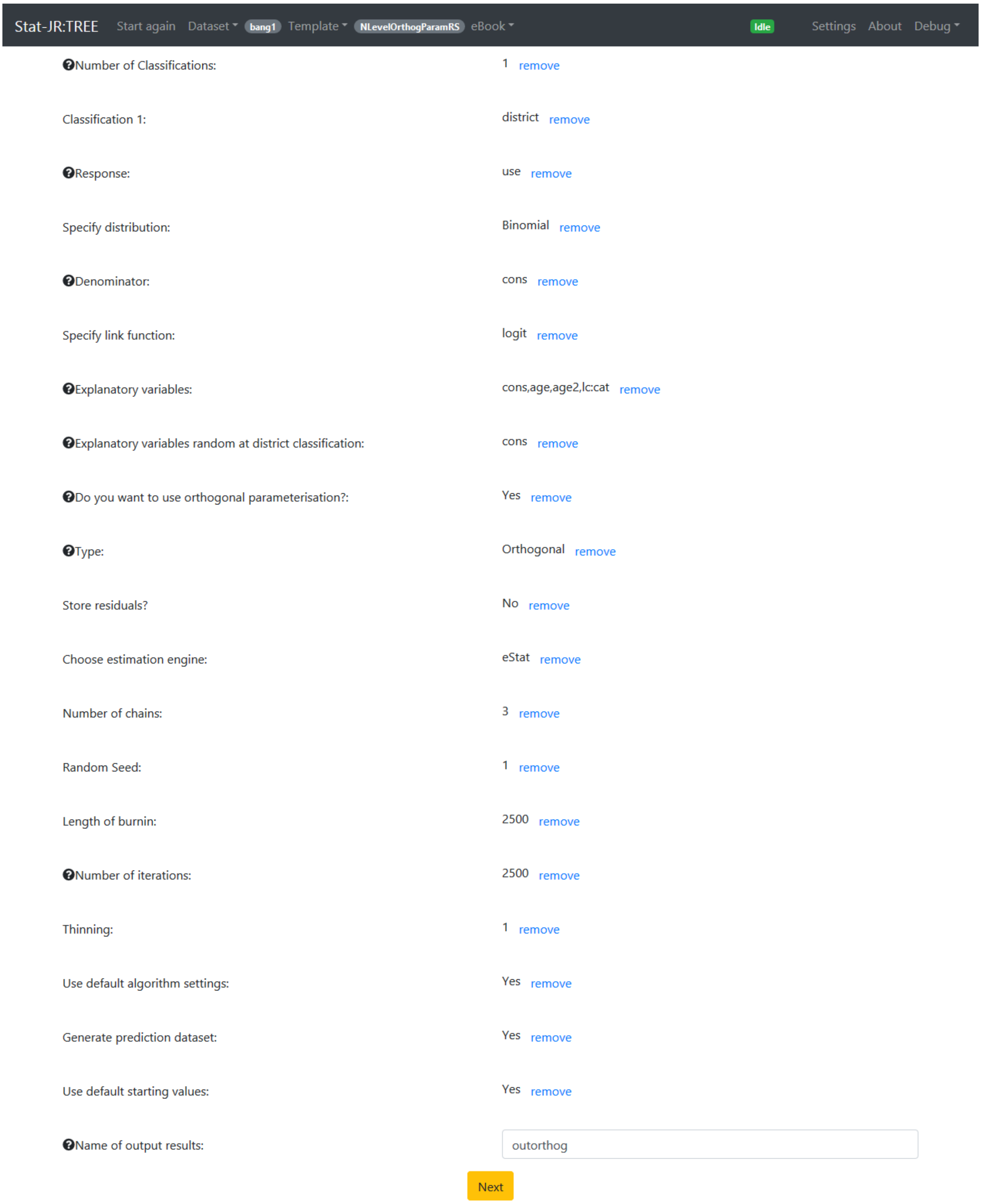
The reason eStat (and MLwiN) perform badly in terms of ESS in this instance is that they are performing single-site updating, and the parameters are correlated. So here we will consider a reparameterisation method that aims to fit parameters that are less correlated, and then translates them back to the original parameters. For this we construct a set of orthogonal vectors from the original predictor variables (see (Browne, Steele, Golalizadeh, & Green, 2009) for details).

We will therefore now look at the **NLevelOrthogParamRS** template in order to use orthogonalisation on our model. This template actually fits a larger family of models: those with any number of higher levels/classifications (hence "NLevel"), allowing for the possibility of random slopes at each of these levels (hence "RS"), and so our 2-level random intercept model is perhaps the simplest case that the template fits.

Click on the **Template** pull-down list and click **Choose** then select **NLevelOrthogParamRS** from the template list.

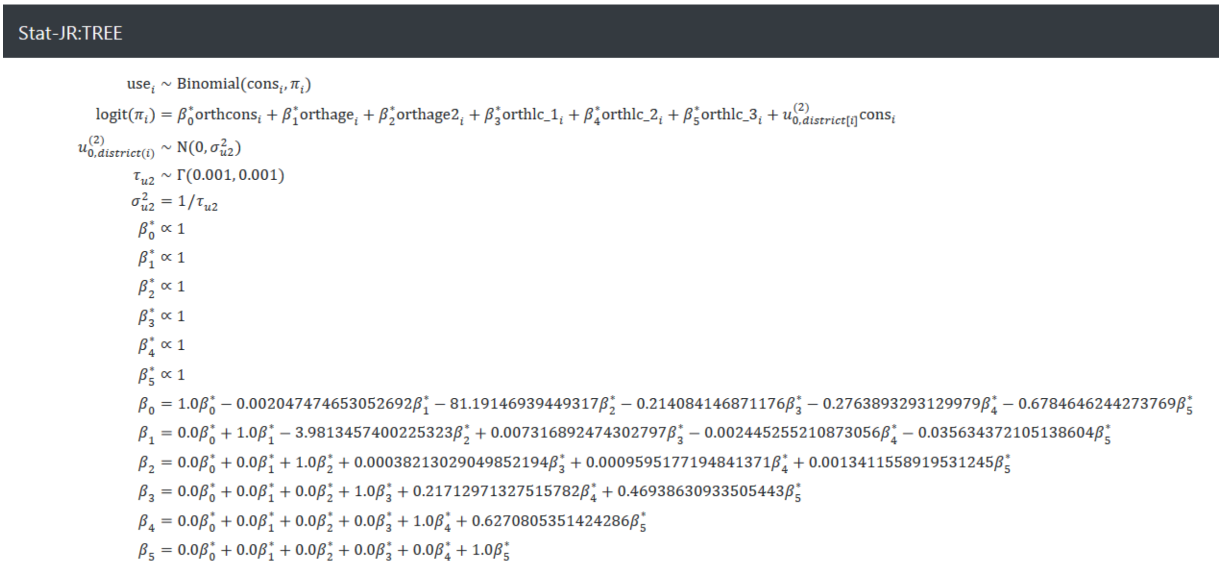
Click on **Use** and fill in the template inputs as follows:

Dataset: *bang1;* **Template:** *NLevelOrthogParamRS;* **Input string:** *{'Engine': 'eStat', 'x1': 'cons', 'burnin': '2500', 'D': 'Binomial', 'outdata': 'outorthog', 'storeresid': 'No', 'thinning': '1', 'n': 'cons', 'nchains': '3', 'orthtype': 'Orthogonal', 'link': 'logit', 'defaultalg': 'Yes', 'iterations': '2500', 'y': 'use', 'x': 'cons,age,age2,lc:cat', 'C1': 'district', 'NumLevs': '1', 'seed': '1', 'useorthog': 'Yes', 'makepred': 'Yes', 'defaultsv': 'Yes'}*



Number of Classifications:	1
Classification 1:	district
Response:	use
Specify distribution:	Binomial
Denominator:	cons
Specify link function:	logit
Explanatory variables:	cons,age,age2,lc:cat
Explanatory variables random at district classification:	cons
Do you want to use orthogonal parameterisation?:	Yes
Type:	Orthogonal
Store residuals?	No
Choose estimation engine:	eStat
Number of chains:	3
Random Seed:	1
Length of burnin:	2500
Number of iterations:	2500
Thinning:	1
Use default algorithm settings:	Yes
Generate prediction dataset:	Yes
Use default starting values:	Yes
Name of output results:	outorthog

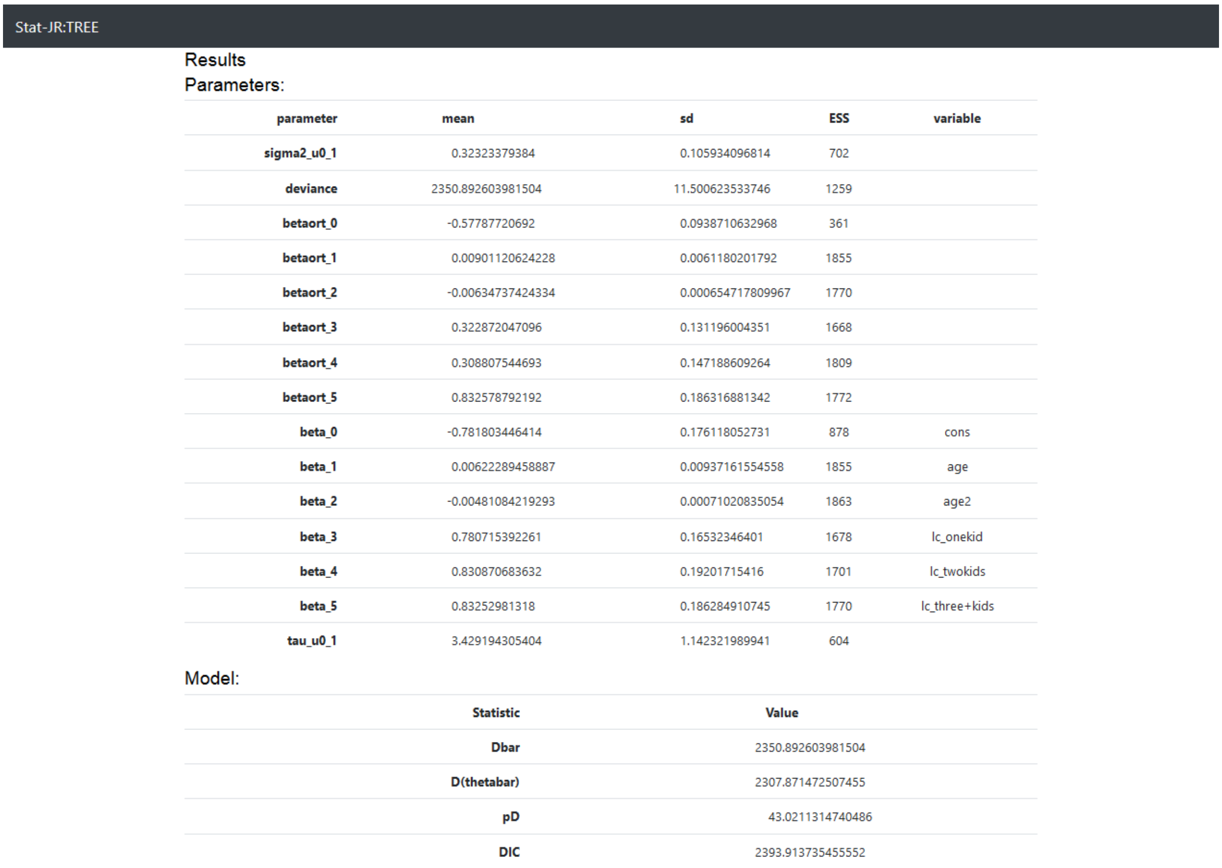
Clicking on **Next** and selecting *equation.tex* in the pull-down list (we've opened it in a new tab) will show the following:



$$
\begin{aligned}
\mathrm{use}_i &\sim \mathrm{Binomial}(\mathrm{cons}_i, \pi_i)\\
\mathrm{logit}(\pi_i) &= \beta_0^*\mathrm{orthcons}_i + \beta_1^*\mathrm{orthage}_i + \beta_2^*\mathrm{orthage2}_i + \beta_3^*\mathrm{orthlc\_1}_i + \beta_4^*\mathrm{orthlc\_2}_i + \beta_5^*\mathrm{orthlc\_3}_i + u_{0,district[i]}^{(2)}\mathrm{cons}_i\\
u_{0,district(i)}^{(2)} &\sim \mathrm{N}(0, \sigma_{u2}^2)\\
\tau_{u2} &\sim \Gamma(0.001, 0.001)\\
\sigma_{u2}^2 &= 1/\tau_{u2}\\
\beta_0^* &\propto 1\\
\beta_1^* &\propto 1\\
\beta_2^* &\propto 1\\
\beta_3^* &\propto 1\\
\beta_4^* &\propto 1\\
\beta_5^* &\propto 1\\
\beta_0 &= 1.0\beta_0^* - 0.002047474653052692\beta_1^* - 81.19146939449317\beta_2^* - 0.214084146871176\beta_3^* - 0.2763893293129979\beta_4^* - 0.6784646244273769\beta_5^*\\
\beta_1 &= 0.0\beta_0^* + 1.0\beta_1^* - 3.9813457400225323\beta_2^* + 0.007316892474302797\beta_3^* - 0.002445255210873056\beta_4^* - 0.035634372105138604\beta_5^*\\
\beta_2 &= 0.0\beta_0^* + 0.0\beta_1^* + 1.0\beta_2^* + 0.00038213029049852194\beta_3^* + 0.0009595177194841371\beta_4^* + 0.001341155891953124 5\beta_5^*\\
\beta_3 &= 0.0\beta_0^* + 0.0\beta_1^* + 0.0\beta_2^* + 1.0\beta_3^* + 0.2171297132751578 2\beta_4^* + 0.46938630933505443\beta_5^*\\
\beta_4 &= 0.0\beta_0^* + 0.0\beta_1^* + 0.0\beta_2^* + 0.0\beta_3^* + 1.0\beta_4^* + 0.6270805351424286\beta_5^*\\
\beta_5 &= 0.0\beta_0^* + 0.0\beta_1^* + 0.0\beta_2^* + 0.0\beta_3^* + 0.0\beta_4^* + 1.0\beta_5^*
\end{aligned}
$$

Here we see that the model code is actually fitting a different set of predictors, each with the prefix ‘orth’ and a corresponding set of coefficients. There is then a set of deterministic statements that translate these coefficient values to the coefficient values for the original predictors (again, see (Browne, Steele, Golalizadeh, & Green, 2009) for details)

Clicking on the **Run** button will run the model (which took 36s on this particular machine, including compiling), after which selecting *ModelResults* from the pull-down list, and popping out into a new tab, gives the following:



Results

Parameters:

parameter	mean	sd	ESS	variable
sigma2_u0_1	0.32323379384	0.105934096814	702	
deviance	2350.892603981504	11.500623533746	1259	
betaort_0	-0.57787720692	0.0938710632968	361	
betaort_1	0.00901120624228	0.0061180201792	1855	
betaort_2	-0.00634737424334	0.000654717809967	1770	
betaort_3	0.322872047096	0.131196004351	1668	
betaort_4	0.308807544693	0.147188609264	1809	
betaort_5	0.832578792192	0.186316881342	1772	
beta_0	-0.781803446414	0.176118052731	878	cons
beta_1	0.00622289458887	0.00937161554558	1855	age
beta_2	-0.00481084219293	0.00071020835054	1863	age2
beta_3	0.780715392261	0.16532346401	1678	lc_onekid
beta_4	0.830870683632	0.19201715416	1701	lc_twokids
beta_5	0.83252981318	0.186284910745	1770	lc_three+kids
tau_u0_1	3.429194305404	1.142321989941	604	

Model:

Statistic	Value
Dbar	2350.892603981504
D(thetabar)	2307.871472507455
pD	43.0211314740486
DIC	2393.913735455552

The estimates, their ESS, and the time taken to run the model are all added to the end of the software comparison table we looked at above. It indicates that, compared to the other method we employed to fit the model in eStat, there is no obvious overhead incurred when performing the orthogonalising algorithm, and it is much faster than OpenBUGS, and the ESS are now much better (if still not as good as OpenBUGS). We therefore have two ways of fitting the model that are reasonably comparable in terms of ESS/s, with little to choose between them. Previously when we have compared JAGS to other approaches it has done poorly but the current version appears to do as well as OpenBUGS but without the time penalty. The orthogonalising approach is also available in MLwiN: this will be faster again, and should have similar ESS to the method in eStat, and therefore may be the best overall in terms of ESS/s and even better than JAGS, but we leave this for the reader to investigate.

5.6 Predictions from the model

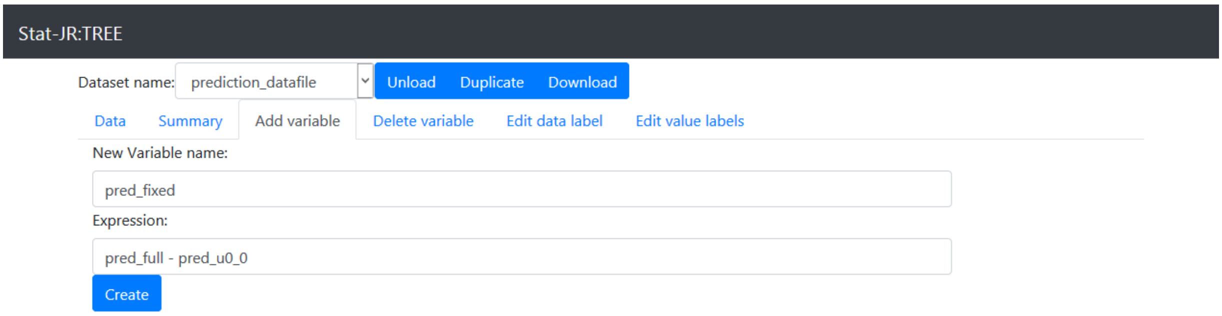
When we ran this model we discussed some interpretation of the fit, but it would be nice to plot some predictions from the model as well. In this latest version of Stat-JR we have added the option to store predictions when fitting the model. So hopefully in the last model fit you will have ticked "Yes" to the generate prediction dataset question. This will generate a new dataset named *prediction_datafile* which contains the original data and several prediction columns formed from the model fit.

To use this dataset we need to select **Choose** on the dataset list and select *prediction_datafile* from the list and click **Use**.

In fact the dataset has a full prediction column called *pred_full* but this also contains the district random effects. We would here like to simply predict from the fixed part of the model so we can construct the variable *pred_fixed* as follows:

Click on **View** from the **Dataset** menu, then choose **Add variable**, and input the new variable *pred_fixed* as indicated below.

Click on **Create** to create the variable

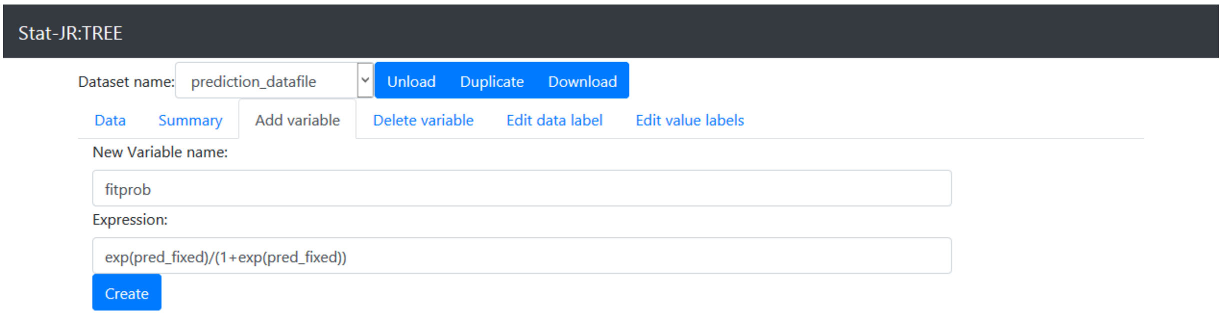


Dataset name: prediction_datafile

New Variable name: pred_fixed

Expression: pred_full - pred_u0_0

This has created a variable on the fixed predictor scale but as we are fitting a logistic regression we need to take an anti-logistic transform to convert these predictions to probabilities. This can be done by creating another column in the dataset as shown below:



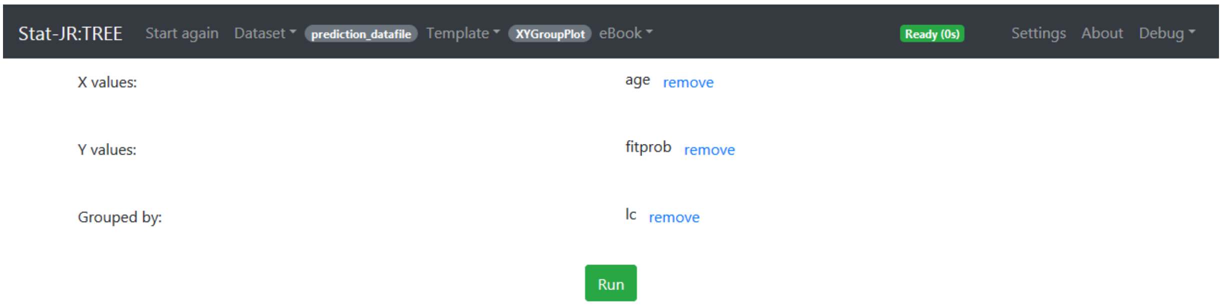
Dataset name: prediction_datafile

New Variable name: fitprob

Expression: exp(pred_fixed)/(1+exp(pred_fixed))

In order to plot separate fitted curves for the various numbers of living children we can use the template **XYGroupPlot** as shown below:

Dataset: *prediction_datafile;* **Template:** *XYGroupPlot;* **Input string:** *{'group': 'lc', 'xaxis': 'age', 'yaxis': 'fitprob'}*

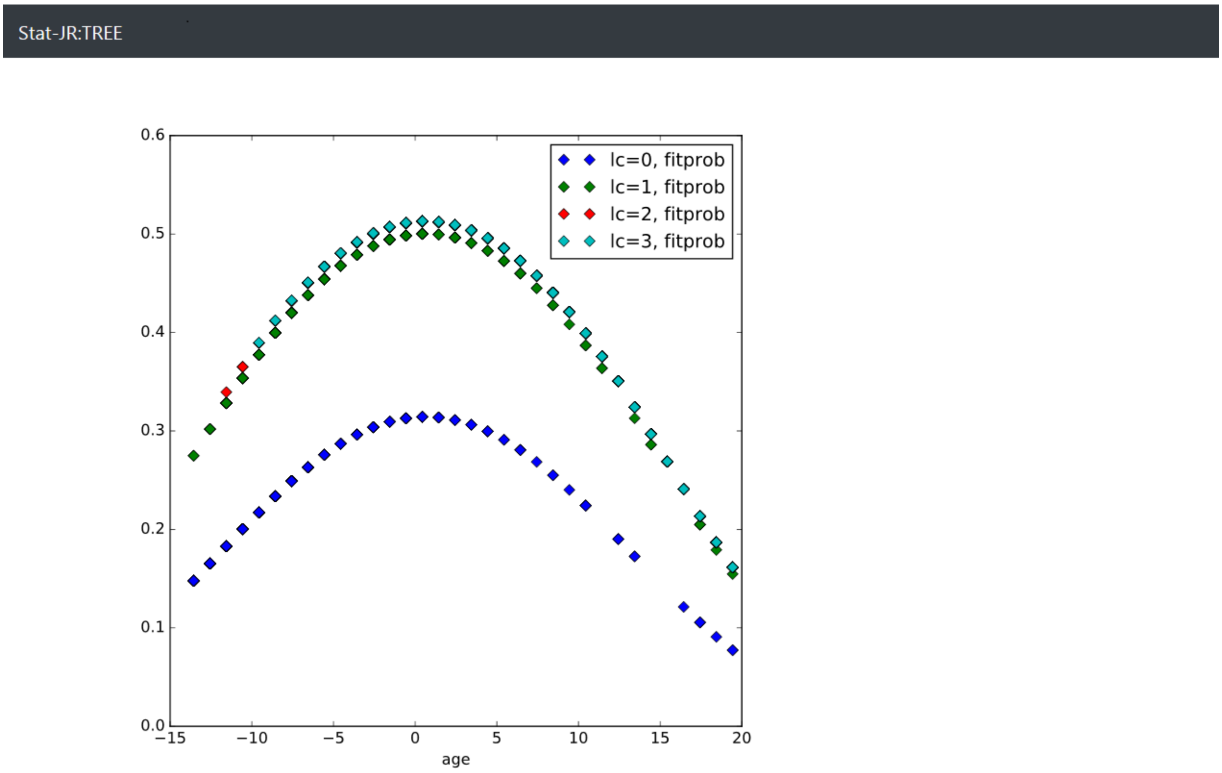


X values: age

Y values: fitprob

Grouped by: lc

Clicking on **Run** and popping out *graphxygroup.svg* gives the following:



Here we see the four curves (although three of them are very close together) which clearly showing that the women with children have higher probabilities of using contraceptives, and that the peak for each group is around the average age of the sample, as discussed earlier.

Hopefully this section has shown firstly that Stat-JR can fit models other than Normal response models; in fact there are a vast number of model templates which fit lots of other model classes. Secondly, we hope we've shown its utility in terms of comparing model-fitting across different software packages for different models, accessing each from a common hub.

6 Miscellaneous other topics e.g. Data Input/Export

Stat-JR works with datasets saved in Stata format, i.e. with a *.dta* extension. It looks for these in the*...\datasets* folder of the Stat-JR install, and also in a folder saved, by default, under your user name, e.g. *C:\Users\YourName\.statjr\datasets* (you can change the path via **Settings** in the black bar at the top of the browser window in the TREE interface; if you do this then make sure you press the **Set** button, and then **Debug > Reload** datasets in the black bar at the top).

6.1 If your dataset is already in .dta format

If your dataset is already in *.dta* format (see below), then you can upload it, in TREE, via (i) **Dataset > Upload** (menu options in the black bar at the top of the browser window), which will upload it into the temporary memory cache, or by (ii) saving your dataset in one of the *datasets* folders (as discussed above), and then selecting **Debug > Reload datasets** (again, accessible via the black bar at the top of the browser window).

In the case of option (i), the dataset will be available for use in the current session, but you then need to download it (as a *.dta* file) via **Dataset > Download** (e.g. saving it into the *StatJR\datasets* folder) for use in the future sessions too. In the case of option (ii), the dataset will be available in future sessions since it has been saved in one of the folders in which Stat-JR searches for datasets on start-up.

6.2 If your dataset is in .txt format

If, instead, you have your dataset saved as a *.txt* file, you can use Stat-JR's *LoadTextFile* template to save it into the temporary memory cache (the template *LoadTextFileMoreOptions* allows the user to specify more particulars, and can also handle string variables).

This dataset will be available for use in the current session, but you then need to download it (as a *.dta* file) via **Dataset > Download** (e.g. saving it into the *StatJR\datasets* folder) for use in the future sessions too.

6.3 Converting your dataset to .dta format

Via the procedure described in Section 6.2 (and downloading), Stat-JR will save your *.txt* dataset as a *.dta* file, but you can also create *.dta* files via Stata, MLwiN and R (e.g. the foreign package in R).

7 References

SAS Institute Inc. (2011). Base SAS® 9.3 Procedures Guide. Cary, NC.

Barry, J., Francis, B., & Davies , R. (1989). SABRE: Software for the Analysis of Binary Recurrent Events. *Decarli A., Francis B.J., Gilchrist R., Seeber G.U.H. (eds) Statistical Modelling. Lecture Notes in Statistics, 57*.

Brooks, S., & Gelman, A. (1998). General Methods for Monitoring Convergence of Iterative Simulations. *Journal of Computational and Graphical Statistics*(7), 434-455.

Browne, W. (2016). *MCMC Estimation in MLwiN, v2.36.* Centre for Multilevel Modelling, University of Bristol.

Browne, W., Steele, F., Golalizadeh, M., & Green, M. (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.

Cervone, D., Sorge, V., Lawson-Perfect, C., & Krautzberger, P. (2017). MathJax version 2.7.2. MathJax.

Charlton, C., Rasbash, J., Browne, W., Healy, M., & Cameron, B. (2017). MLwiN Version 3.00. Centre for Multilevel Modelling, University of Bristol.

Cottrell, A., & Lucchetti, R. (2007). Gretl User's Guide.

de Valpine, P., Paciorek, C., Turek, D., Anderson-Bergman, C., & Temple Lang, D. (2016). nimble: Flexible BUGS-Compatible System for Hierarchical Statistical Modeling and Algorithm Development. R package version 0.5. http://r-nimble.org.

Eaton, J. W., Bateman, D., Hauberg, S., & Wehbring, R. (2016). *GNU Octave version 4.2.0 manual: a high-level interactive language for numerical computions.*

Free Software Foundation. (2017). GNU PSPP (Version 1.0.1). Boston, MA: GNU Project.

Gamerman, D. (1997). Sampling from the posterior distribution ingeneralized linear mixed models. *Statistics and Computing*(7), 57-68.

Gelfand, A., Hills, S., Racine-Poon, A., & Smith, A. (1990). Illustration of Bayesian inference in Normal data models using Gibbs Sampling. *Journal of the American Statistical Association*(85), 972-985.

Goldstein, H., Rasbash, J., Yang, M., Pan, H., Nuttall, D., & Thomas, S. (1993). A multilevel analysis of school examination results. *Oxford Review of Education*(19), 425–433.

Hadfield, J. (2010). MCMC Methods for Multi-Response Generalized Linear Mixed Models: The MCMCglmm R Package. *Journal of Statistical Software*(33(2)), 1-22.

Hedeker, D., & Nordgren, R. (2013). MIXREGLS: A Program for Mixed-Effects Location Scale Analysis. *Journal of Statistical Software, 52*(12), 1--38.

IBM Corp. (2017). IBM SPSS Statistics for Windows, Version 25.0. Armonk, NY.

Lillard, L., & Panis, C. (2003). aML Multilevel Multiprocess Statistical Software, Version 2.0. Los Angeles, California: EconWare.

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

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

Minitab, Inc. . (2017). Minitab 18 Statistical Software . State College, PA.

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).* Vienna, Austria.

R Core Team. (2016). A Language and Environment for Statistical Computing. Vienna, Austria: R Foundation for Statistical Computing.

Rossum, G. v. (n.d.). The Python Language Reference. Python Software Foundation.

Sarkar, D. (2008). *Lattice: Multivariate Data Visualization with R. New York: Springer.*

Spiegelhalter, D., Best, N., Carlin, B., & van der Linde, A. (2002). Bayesian measures of model complexity and fit (with discussion). *Journal of the Royal Statistical Society, Series B*(64), 191-232.

StataCorp. (2017). Stata Statistical Software: Release 15. College Station, TX: StataCorp LLC.

Stroustrup, B. (2013). *The C++ Programming Language.* Addison-Wesley Professional.

The MathWorks, Inc. (2017). MATLAB and Statistics Toolbox Release 2017b. Natick, Massachusetts.

VSN International. (2015). Genstat for Windows 18th Edition. Hemel Hempstead: VSN International.

8 Appendix: List of Third Party Software that are used by Stat-JR

Stat-JR makes use of several third party software products that are included within the distributed code or (in the case of MinGW) need to be downloaded separately. These software products each have a license file that can be viewed from the links in the table below and/or in the licences subdirectory of the installed code.

Package	Link	Licence terms
beautifulsoup	http://bazaar.launchpad.net/~leonardr/beautifulsoup/bs4/view/head:/LICENSE	MIT
BLAS	http://www.netlib.org/blas/faq.html#2	Own licence (Netlib)
Blockly	https://github.com/google/blockly/blob/master/LICENSE	Apache (v2)
Bootstrap	https://github.com/twitter/bootstrap/blob/master/LICENSE	MIT
bottle	https://github.com/bottlepy/bottle/blob/master/LICENSE	MIT
bottle-websocket	https://github.com/zeekay/bottle-websocket/blob/master/LICENSE	MIT
cffi	https://bitbucket.org/cffi/cffi/src/default/LICENSE	MIT
cssselect	https://github.com/scrapy/cssselect/blob/master/LICENSE	BSD
cx_freeze	http://cx-freeze.readthedocs.org/en/latest/license.html	PSF
cycler	https://github.com/matplotlib/cycler/blob/master/LICENSE	BSD
dateutil	https://github.com/dateutil/dateutil/blob/master/LICENSE	Simplified BSD
decorator	https://github.com/micheles/decorator/blob/master/LICENSE.txt	BSD
Font awesome	https://github.com/FortAwesome/Font-Awesome/blob/master/LICENSE.txt	Full Font Awesome Free license
gevent	https://github.com/gevent/gevent/blob/master/LICENSE	MIT
gevent-websocket	https://bitbucket.org/noppo/gevent-websocket/src/0df192940acd288e8a8f6d2dd30329a3381c90f1/LICENSE?fileviewer=file-view-default	Apache(v2)
greenlet	https://github.com/python-greenlet/greenlet/blob/master/LICENSE	PSF
html5lib	https://github.com/html5lib/html5lib-python/blob/master/LICENSE	MIT
isodate	http://www.opensource.org/licenses/bsd-license.php	BSD
kiwisolver	https://github.com/google/kiwi-solver/blob/master/LICENSE	Apache(v2)
jqgrid	https://github.com/tonytomov/jqGrid/blob/v4.6.0/jqGrid.jquery.json	Dual MIT/GPL(v2)

jqtree	https://github.com/mbraak/jqTree/blob/master/LICENSE	Apache(v2)
jquery	http://jquery.org/license	MIT
jQuery File Upload	https://github.com/blueimp/jQuery-File-Upload/blob/master/LICENSE.txt	MIT
jquery-ui	http://jquery.org/license	MIT
jQuery-xpath	http://opensource.org/licenses/MIT	MIT
LAPACK	http://www.netlib.org/lapack/LICENSE.txt	Modified BSD
lxml	http://lxml.de/index.html#license	BSD
mako	http://www.opensource.org/licenses/mit-license.php	MIT
markupsafe	https://github.com/pallets/markupsafe/blob/master/LICENSE	BSD
MathJax	http://cdn.mathjax.org/mathjax/2.0-latest/LICENSE	Apache
matplotlib	http://matplotlib.org/users/license.html	Modified BSD
MinGW-w64	http://mingw-w64.org/doku.php/support	Not distributed with software directly
networkx	https://raw.githubusercontent.com/networkx/networkx/master/LICENSE.txt	BSD
numexpr	https://github.com/pydata/numexpr/blob/master/LICENSE.txt	MIT
numpy	http://numpy.scipy.org/license.html#license	BSD
pandas	http://pandas.pydata.org/pandas-docs/stable/overview.html#license	Modified BSD
patsy	https://github.com/pydata/patsy/blob/master/LICENSE.txt	BSD
peg.js	https://github.com/pegjs/pegjs/blob/master/LICENSE	MIT
ply	http://www.dabeaz.com/ply/README.txt	BSD
popper	https://github.com/FezVrasta/popper.js/blob/master/LICENSE.md	MIT
prov	https://github.com/trungdong/prov/blob/master/LICENSE	MIT
provpy	http://opensource.org/licenses/BSD-2-Clause	BSD
pycparser	https://github.com/eliben/pycparser/blob/master/LICENSE	BSD
pyparsing	http://svn.code.sf.net/p/pyparsing/code/trunk/src/LICENSE	MIT
pyquery	https://github.com/dsc/pyquery/blob/master/LICENSE.txt	BSD
Python	http://docs.python.org/license.html	PSF
pytz	https://github.com/stub42/pytz/blob/master/src/LICENSE.txt	MIT
rdflib	https://github.com/RDFLib/rdflib/blob/master/LICENSE	BSD
scipy	https://github.com/scipy/scipy/blob/master/LICENSE.txt	BSD
setuptools	http://docs.python.org/license.html	PSF

six	https://bitbucket.org/gutworth/six/src/e3da7fd96039a6ed89493f89d121c4f3797e6713/LICENSE?at=default	MIT
sparqlwrapper	https://github.com/RDFLib/sparqlwrapper/blob/master/LICENSE.txt	W3C
statsmodels	https://github.com/statsmodels/statsmodels/blob/master/LICENSE.txt	Modified BSD
tinymce	https://github.com/tinymce/tinymce/blob/master/LICENSE.TXT	LGPL(v2.1)
weave	https://github.com/scipy/weave/blob/master/LICENSE.txt	BSD
webencodings	https://github.com/gsnedders/python-webencodings/blob/master/LICENSE	BSD

9 Index