LUDWIG-
MAXIMILIANS-
UNIVERSITÄT
MÜNCHEN

INSTITUT FÜR STATISTIK
SONDERFORSCHUNGSBEREICH 386

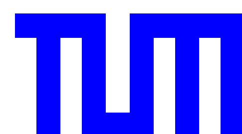
Lang, Brezger:

BayesX - Software for Bayesian Inference based on Markov Chain Monte Carlo simulation techniques

Sonderforschungsbereich 386, Paper 187 (2000)

Online unter: http://epub.ub.uni-muenchen.de/

Projektpartner



BayesX

Software for Bayesian Inference based on
Markov Chain Monte Carlo simulation techniques

Version 0.5

by

Stefan Lang and Andreas Brezger

University of Munich,
Ludwigstr. 33, 80539 Munich
email: lang@stat.uni-muenchen.de
andib@stat.uni-muenchen.de

Licensing agreement:
The authors of this software grant to any individual or non-commercial organization the right to use and to make an unlimited number of copies of this software. Usage by commercial entities requires a license from the authors. You may not decompile, disassamble, reverse engineer, or modify the software. This includes, but is not limited to modifying/changing any icons, menues, or displays associated with the software. This software cannot be sold without written authorization from the authors. This restriction is not intended to apply for connect time charges, or flat rate connection/download fees for electronic bulletin board services. The authors of this program accept no responsibility for damages resulting from the use of this software and make no warranty on representation, either express or implied, including but not limited to, any implied warranty of merchantability or fitness for a particular purpose. This software is provided as is, and you, its user, assume all risks when using it.

BayesX is available under `http://www.stat.uni-muenchen.de/~lang/bayesx/bayesx.html`

Contents

1 What is *BayesX* **4**

2 Getting started **8**

2.1 Installing BayesX 8
2.2 General usage of *BayesX* 8
2.3 Windows 11
2.3.1 The command window 11
2.3.2 The output window 11
2.3.3 The review window 11
2.3.4 The object browser 11
2.4 Description of dataset examples 12
2.4.1 Rents for flats 12
2.4.2 Credit scoring 13

3 Special Commands **14**

3.1 Exiting *BayesX* 14
3.2 Opening and closing log-files 14
3.3 Saving the contents of the output window 15
3.4 Changing the delimiter 16
3.5 Using batch-files 16
3.6 Dropping objects 17

4 dataset objects **18**

4.1 Method 'drop' 18
4.2 Functions and Expressions 20
4.2.1 Operators 20
4.2.2 Functions 21
4.2.3 Explicit subscribing 22
4.2.4 Constants 23
4.2.5 Expression syntax errors 24
4.3 Method 'generate' 24
4.4 Method 'infile' 25

4.5 Method ‘outfile’ 28
4.6 Method ’rename’ 29
4.7 Method ‘replace’ 30
4.8 Method ‘set obs’ 31
4.9 Method ‘sort’ 31
4.10 Variable names 32
4.11 Examples 32
4.11.1 The credit scoring dataset 32
4.11.2 Simulating complex statistical models 33

5 map objects **35**
5.1 Method ’infile’ 35

6 bayesreg objects **39**
6.1 Method ’regress’ 39
6.2 Method ’autocorr’ 55
6.3 Method ’getsample’ 57
6.4 S-Plus functions for visualizing estimation results 59
6.4.1 Installation of the functions 59
6.4.2 Plotting nonparametric functions 59
6.4.3 Drawing geographical maps 62
6.4.4 Plotting autocorrelation functions 66
6.4.5 Plotting sampled parameters 67
6.5 Global options 68
6.6 Examples 69
6.6.1 Credit scoring 69
6.6.2 Rents for flats 74

Index

Chapter 1

What is *BayesX*

BayesX is a Software tool for Bayesian inference based on Markov Chain Monte Carlo (MCMC) inference techniques. This is the first (test) version of the program, so only selected number of statistical procedures are available. The main feature of *BayesX* so far, is a very powerful regression tool for Bayesian semiparametric regression within the Generalized linear models framework. *BayesX* is able to estimate nonlinear effects of metrical covariates, trends and flexible seasonal patterns of time scales, structured and/or unstructured random effects of spatial covariates (geographical data) and unstructured random effects of unordered group indicators. Moreover, *BayesX* is able to estimate varying coefficients models with metrical and even spatial covariates as effectmodifiers. The distribution of the response can be either Gaussian, binomial or Poisson. In addition, *BayesX* has some useful functions for handling and manipulating datasets and geographical maps.

We have taken great pain in making *BayesX* as user-friendly as possible, to enable even non-experts in Bayesian statistics, to use *BayesX* successfully. However, Markov Chain Monte Carlo simulation techniques are subject of current research with many questions remaining and (of course) a lot of unsolved problems. Therefore, at least some basic knowledge about Bayesian inference with MCMC techniques is strongly recommended.

To give a first impression about the capabilities and the usage of *BayesX* for estimating complex semiparametric regression models, we give here an example of a study on unemployment durations that has been carried out with *BayesX*. The purpose of the study is to estimate the probability that an unemployed person gets a new job given some covariates. The data are available on an individual basis with more than 100000(!) observations. Using *BayesX*, we estimated a logistic regression model with predictor

$$\eta = f_1(t) + f_2^{Trend}(D) + f_3^{Season}(D) + f_4(A) + f_5^{str.}(C) + f_6^{unstr.}(C) + others$$

where $f_1(t)$ and $f_4(A)$ are nonlinear effects of duration time t and age A, $f_2^{Trend}(D)$ and $f_3^{Season}(D)$ are a nonlinear trend and a time varying seasonal effect of calendar time D, and $f_5^{str.}(C)$ and $f_6^{unstr.}(C)$ are structured and unstructured spatial random effects of the district C in which the unemployed live. The estimation of this model has been carried out in *BayesX* using the following program code:

```
delimiter = ;
dataset m;
m.infile , maxobs=120000 using c:\data\male.raw;
```

```
map ma;
ma.infile using c:\maps\westgermany.bnd;

bayesreg b;
b.outfile = c:\results\male;
b.regress y = time(rw2,1,5) + date(rw2,2,6) + date(season,12,3,8) + age(rw2,2,6)
+ region(spatial,ma) + region(random) , maxint=300 burnin=2000 iterations=52000
step=50 family=binomial using m;

delimiter = return;
```

The second and the third statement in the program code are used to create a *dataset object* 'm' and to read in the data, which are stored in the external ASCII-file 'c:\data\male.raw'. In the following two statements a *map object* 'ma' is created and the map of West Germany is stored therein. The map of Germany is used later for estimating the structured spatial random effect $f_5^{str.}(C)$. Finally, a so called *bayesreg object* is created and a Bayesian regression model is estimated using the *regress* command of *bayesreg objects*.

Estimation results are shown in Figures 1.1 and 1.2. The Figures have been created using a couple of (easy to use) S-Plus functions that are available together with *BayesX*. More details about the presented study on unemployment durations can be found in Fahrmeir, Lang (1999), see the reference Section in Chapter 6 about *bayesreg objects*. Other examples about the usage of the regression procedure in *BayesX* can be found in Chapter 6.6 of this manual.

We plan to extend the current first version of *BayesX* in the near future. The following additional features will be incorporated into *BayesX* (in order of expected publication):

- Functions for **prediction**, which will be based on the posterior predictive distribution.
- Functions for estimating **multivariate** regression models, such as cumulative models for ordered categorical response or models with unordered categorical response.
- **Bayesian P-Splines** for estimating nonlinear effects of metrical covariates.
- **2-dimensional surface fitting**.
- **Edge preserving smoothing**.
- Additional and improved functions for handling and manipulating datasets.
- Additional and improved functions for handling geographical maps.

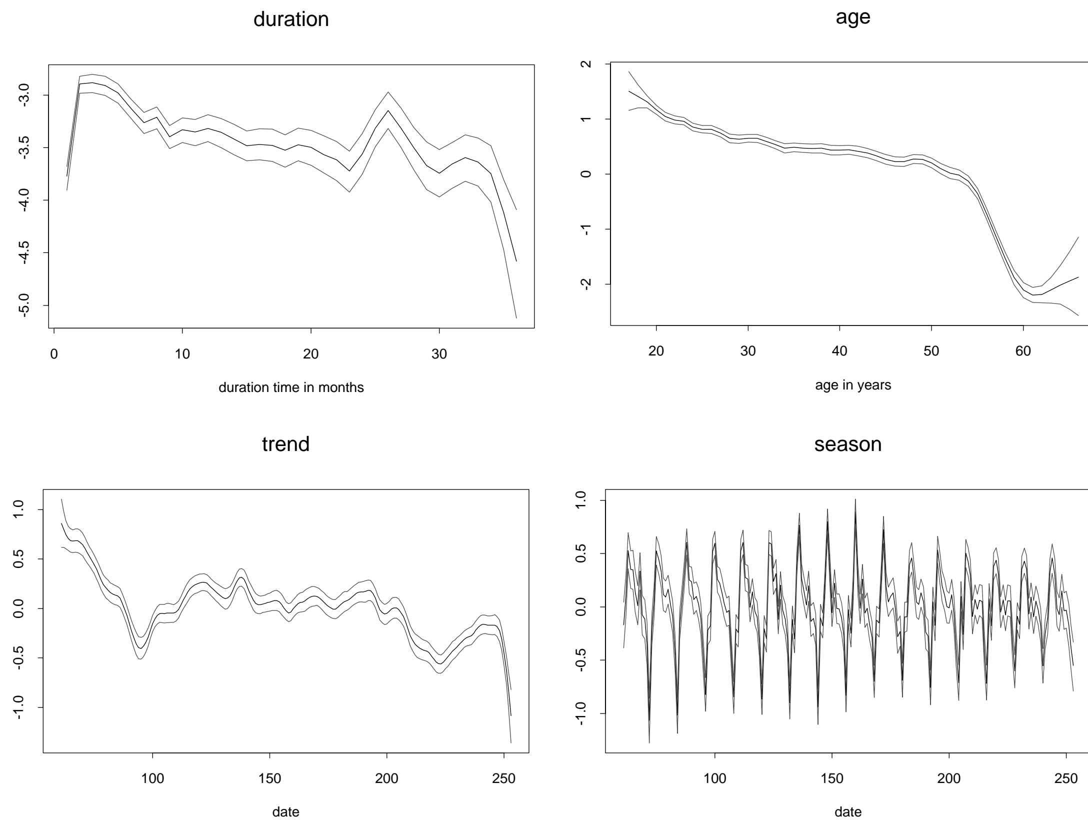


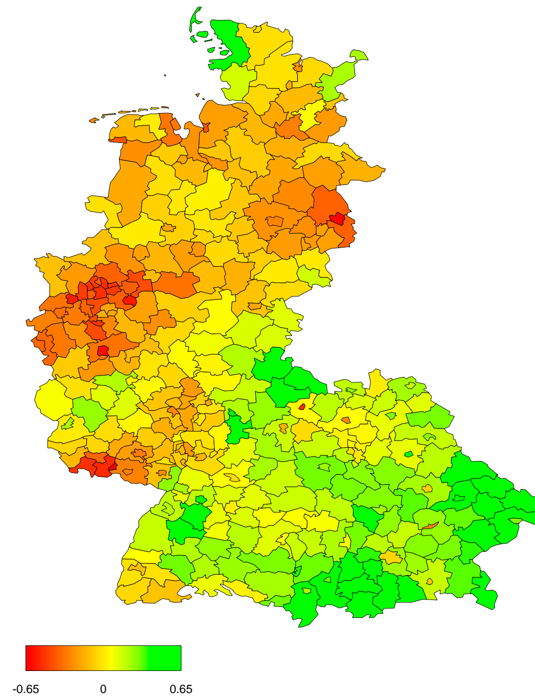
Figure 1.1: Estimated effects of duration time, age and calendar time. Shown is the posterior mean with 80 % credible regions.

Acknowledgement:
The development of *BayesX* has been supported by grants from the German National Science Foundation, Sonderforschungsbereich 386.

Special thanks goes to (in alphabetical order of first names):

Alexander Jerak for testing the program and suggestions for improvement;
Andrea Hennerfeind for testing the program and suggestions for improvement. Thanks also for carefully reading and correcting the manual.
Dieter Gollnow for computing and providing the map of Munich (a really hard job);
Leo Knorr-Held for advertising the program;
Ludwig Fahrmeir for his patience with finishing the program and for carefully reading and correcting the manual;
Ngianga-Bakwin Kandala for using the program;
Petra Kragler for being the first user of *BayesX*, extensive testing of the program (a real power tester) and finding really unexpected errors. Thanks also for carefully reading and correcting the manual.

a) structured random effect



b) unstructured random effect

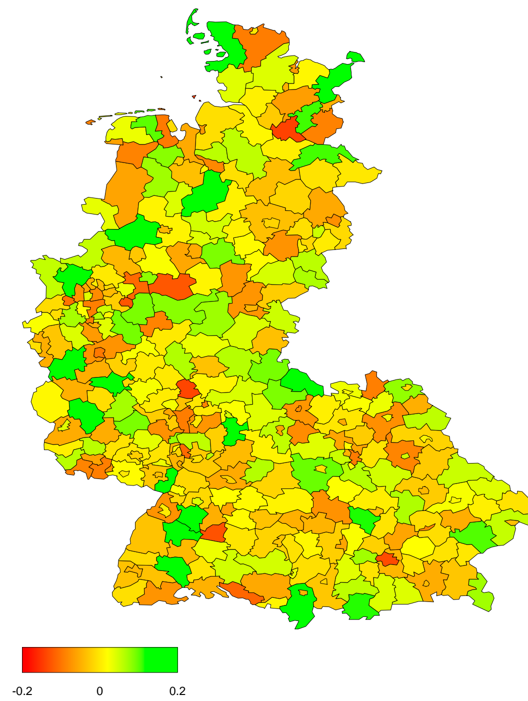


Figure 1.2: Estimated spatial structured and unstructured random effects (colored). Shown is the posterior mean.

Chapter 2

Getting started

This chapter provides information on how to install *BayesX* on your computer and explains the purpose of the different windows appearing after starting *BayesX*. In a third section the general usage of the program and the features of the current version are described. Finally we describe two datasets, which serve as the basis for most of the illustrating examples in this manual.

2.1 Installing BayesX

Installing *BayesX* is very simple. Suppose you have already downloaded the installation file *bayesx.zip*. Unzip this file now using the *winzip* program and store the unzipped files in a temporary directory. Start the program *setup.exe*, for example by double clicking it in the *Windows explorer*. Now follow the instructions of the setup routine to install *BayesX*. After the successfull installation of *BayesX* your installation directory contains five additional subdirectories, namely the directories *doc*, *examples*, *output*, *sfunctions* and *temp*. The *doc* directory contains the program documentation, that is this manual. The *examples* directory contains two datasets, 'credit.raw' and 'rents.raw'. These datasets exemplify many of the statistical functions and routines described in the following chapters. A detailed description of the two datasets is given in Subsection 2.4. The examples directory contains also some tutorial programs that illustrate the usage of most of the functions of *BayesX*, see the chapters below. The *output* directory is the default directory for the program output. The output directory can be redefined by the user. The *sfunctions* directory contains some S-Plus functions for visualizing estimation results obtained with *bayesreg objects*, see Chapter 6 for *bayesreg objects* and Section 6.4 for a detailed description of the S-Plus functions. Finally in the *temp* directory some temporary files will be stored. Normally you will never use this directory.

The created directories and their contents are briefly summarized in Table 2.1.

After a successful installation, *BayesX* can be started via the *Windows Start* button.

2.2 General usage of *BayesX*

BayesX is object oriented, that is the first thing to do during a session is to create some objects. Currently there are three different object types available, *dataset objects*, *map*

Directory	Contents
doc	contains the program description
examples	contains dataset examples and tutorial programs
output	default directory for estimation output
sfunctions	contains some S-Plus functions for visualizing output
temp	stores temporary files

Table 2.1: Subdirectories of the installation directory and their content

objects and *bayesreg objects*. Dataset objects are used to handle and manipulate datasets, see Chapter 4 for details. Map objects are used to handle geographical maps, see Chapter 5. The main purpose of map objects is to serve as auxiliary objects for estimating spatial covariate effects with *bayesreg objects*. The most important object type (with the current version) is the *bayesreg object*. *bayesreg objects* are used to estimate Bayesian semiparametric regression models using modern Markov Chain Monte Carlo simulation techniques. See Section 6 for a detailed description of *bayesreg objects*. The object oriented concept does not go too far, that is inheritance or other concepts of object oriented programs or languages as S-Plus or C++ are not supported.

Creating a new object during a session is very easy. The syntax for creating a new object is:

> *objecttype objectname*

To create for example a dataset object with name 'mydata', simply type:

> dataset mydata

Note that there are restrictions for the name of objects, that is some object names are not allowed. One rule for example is, that object names must begin with a (uppercase or lowercase) letter rather than a number. See Section 4.10 for valid object names. The section is about valid variable names for datasets, but the same is true for object names.

After the successful creation of an object you can apply methods for that particular object. With dataset objects for example you can read in data stored in an ASCII-file using method 'infile', create new variables using method 'generate', modify existing variables using method 'replace' and so on. The syntax for applying methods of the objects is similar for all methods and independent of the particular object type. The general syntax is:

> *objectname.methodname* [*model*] [weight *varname*] [if *boolean expression*]
[, *option(s)*] [using *usingtext*]

Table 2.2 explains the syntax parts in more detail.

Note that [...] indicate that this part of the syntax is optional and may be omitted. Moreover for most methods only some (or even none) of the syntax parts above are meaningful and therefore specifying the rest is not allowed and will cause an error message.

We illustrate the concept with some simple methods of dataset objects. Suppose we have

Syntax part	Description
objectname	the name of the object to apply the method
methodname	the name of the method
model	a model specification (for example a regression model)
weight varname	specifies *varname* as a weight variable
if *boolean expression*	indicates that the method should be applied only if a certain condition holds
, *options*	define (or modify) options for the method
using *usingtext*	indicates that another object or file should be used to apply the particular method

Table 2.2: Syntax parts of methods for objects

already created a dataset object with name 'mydata' and want to create some variables for that dataset. We first have to tell BayesX how many observations we want to create. This can be done with the 'set' command, see also Section 4.8. For example

```
> mydata.set obs = 1000
```

indicates that the dataset 'mydata' should have 1000 observations. Here the method name is *set* and the "model" is *obs = 1000*. Since no other syntax parts (for example if statements) are meaningful for this method they are not allowed. For example specifying an additional weight variable 'x' by typing

```
> mydata.set obs = 1000 weight x
```

will cause the error message:

```
ERROR: weight statement not allowed
```

In a second step we can now create a new variable 'X', say, that contains Gaussian (pseudo)random numbers with mean 2 and standard deviation 0.5:

```
> mydata.generate X = 2+0.5*normal()
```

Here 'generate' is the method name and *X = 2+0.5*normal()* is the model. In this case the model consists of the specification of the new variable name, followed by the equal sign '=' and a mathematical expression for the new variable. As is the case with the 'set' command other syntax parts are not meaningful and therefore not allowed. Suppose now we want to replace the negative values of X with the constant 0. This may be done with the 'replace' command by typing:

```
> mydata.replace X = 0 if X < 0
```

Here an additional if statement is obviously meaningful and is therefore allowed, but not

necessary.

2.3 Windows

After starting *BayesX* you can see a main window with a menue bar and four additional windows within the main window. The four windows are the *command window*, the *output window*, the *review window* and the *object browser*. The purpose of these windows is described in the following four subsections.

2.3.1 The command window

The command window is used to enter and execute commands. By default, a command will be executed if you press the return key. You can change this default delimiter using the 'delimiter' command, see Section 3.4.

2.3.2 The output window

In the *output window* all commands entered in the *command window* or executed through a batch file (see Section 3.5) are printed together with the program output. The contents of the output window may be edited using the *edit menue*. You can write additional text/comments and highlight some text passages for example by underlining them, or by printing in bold fonts. Moreover, the contents of the output window can be saved and processed with your favorite text editor. For saving the output, enter the *file menue* and click on *Save output* or *Save output as*. The contents of the output window can be saved under two different file formats. The default is the rich-text format. The second choice is to store the output window in plain ASCII format. However, the ASCII format has the disadvantage that all text highlights (for example bold letters) will disappear in the saved file.

2.3.3 The review window

In many cases subsequent commands change only slightly. The *review window* gives you a convenient way to bring back and edit past commands. In the *review window* all past commands entered during a session are shown. Click once on one of these past commands and it is copied to the command window, where the command or a slightly modified version can be executed once again.

2.3.4 The object browser

The object browser is used to view the contents of the objects currently in memory. The window is split into two parts. The left part shows the different object types currently supported by *BayesX*. These are for the moment only *dataset objects*, *bayesreg objects* and *map objects*. By clicking on one of the object types the names of all objects of this type will appear in the right part of the object browser. Double clicking on one of the names gives a visualization of the object. The visualization method is depending on the respective object type. Double clicking on dataset objects, for example, will open a spreadsheet where you

Variable	Description
R	monthly rent per square meter in German marks
F	floor space in square meters
A	year of construction
L	Location of the building in subquarters

Table 2.3: Variables of the rent dataset

can inspect the variables and the observations of the dataset. Clicking on map objects opens another window that contains a graphical representation of the map.

2.4 Description of dataset examples

This section describes the two datasets that are used to illustrate many of the features of *BayesX*. The two datasets are stored columnwise in plain ASCII-format. The first row of each dataset contains the variable names separated by blanks. Subsequent rows contain the observations, one observation per row.

2.4.1 Rents for flats

According to the German rental law, owners of appartments or flats can base an increase in the amount that they charge for rent on "average rents" for flats comparable in type, size, equipment, quality and location in a community. To provide information about these "average rents", most larger cities publish "rental guides", which can be based on regression analysis with rent as the dependent variable. The 'rent94.raw' file stored in the *examples* directory is a subsample of data collected in 1994 for the rental guide in Munich. The variable of primary interest is the monthly rent per square meter in German Marks. Covariates characterizing the flat were constructed from almost 200 variables out of a questionnaire answered by tenants of flats. The present dataset contains a small subset of these variables that are sufficient for demonstration. Table 2.3 describes the variables of the dataset. The dataset will be used in the following chapters to demonstrate the usage of *BayesX*, see primarily Section 6.6.2 for a Bayesian regression analysis of the dataset.

Additional to the dataset, the *examples* directory contains the file 'munich.bnd' that contains a map of Munich. This map proves to be useful for visualizing regression results for the explanatory variable location L in the dataset. See Chapter 5 for a description on how to incorporate geographical maps into *BayesX*.

References

Fahrmeir, L., Lang, S. (1999): Bayesian Inference for Generalized Additive Mixed Models Based on Markov Random Field Priors. Discussion Paper 169, Sonderforschungsbereich 386, Ludwigs-Maximilians-Universität München. Available under `http://www.stat.uni-muenchen.de/sfb386/publikation.html`.

Variable	Description
y	creditability, dichotomous with $y = 0$ for creditworthy, $y = 1$ for not creditworthy
account	running account, trichotomous with categories “no running account” (= 1), “good running account” (= 2), “medium running account” (“less than 200 DM”) (= 3)
duration	duration of credit in months, metrical
amount	amount of credit in 1000 DM, metrical
payment	payment of previous credits, dichotomous with categories “good” (= 1), “bad” (= 2)
intuse	intended use, dichotomous with categories “private” (= 1) or “professional” (= 2)
marstat	marital status, with categories “married” (= 1) and “living alone” (= 2).

Table 2.4: Variables of the credit scoring dataset

2.4.2 Credit scoring

The aim of credit scoring is to model or predict the probability that a client with certain covariates (“risk factors”) is to be considered as a potential risk, and therefore will probably not pay back his credit as agreed upon by contract. The data set consists of 1000 consumers credits from a South German bank. The response variable is “creditability”, which is given in dichotomous form ($y = 0$ for creditworthy, $y = 1$ for not creditworthy). In addition, 20 covariates assumed to influence creditability were collected. The present dataset (stored in the *examples* directory) contains a subset of these covariates that proved to be the main influential variables on the response variable, see Fahrmeir and Tutz (1997, ch. 2.1). Table 2.4 contains a description of the variables of the dataset. Usually a binary logit model is applied to estimate the effect of the covariates on the probability of being not creditworthy. As in the case of the rents for flats example, this dataset is used to demonstrate the usage of certain features of *BayesX* , see primarily Section 6.6.1 for a Bayesian regression analysis of the dataset.

References

Fahrmeir, L., Lang, S. (1999): Bayesian Inference for Generalized Additive Mixed Models Based on Markov Random Field Priors. Discussion Paper 169, Sonderforschungsbereich 386, Ludwigs-Maximilians-Universität München. Available under `http://www.stat.uni-muenchen.de/sfb386/publikation.html`.

Fahrmeir, L., Tutz, G. (1997): Multivariate Statistical Modelling based on Generalized Linear Models. New York: Springer–Verlag.

Chapter 3

Special Commands

This chapter describes some commands that are not connected with a particular object type. Among others, there are commands for exiting *BayesX*, opening and closing log-files, saving program output, dropping objects etc..

3.1 Exiting *BayesX*

You can exit *BayesX* by simply typing either

> exit

or

> quit

3.2 Opening and closing log-files

In a log-file, program output and commands entered by the user, are stored in plain ASCII format. This makes it easy to further use the program output, for example results of statistical procedures, in your favourite word processor. Another important application of log-files is the documentation of your work. You open a log-file by typing:

> logopen [, *option*] using *filename*

This opens a log-file that will be saved in 'filename'. After opening a log-file, all commands entered and all program output appearing on the screen will be saved in that file. If the log-file specified in 'filename' is already existing, new output is appended at the end of the file. To overwrite an existing log-file the 'replace' option must be specified in addition. Note that it is not allowed to open more than one log-file simultaneously. An open log-file can be closed by simply typing:

> logclose

Note that exiting *BayesX* automatically closes a log-file that is currently open.

Possible errors

- **ERROR: log-file is already open**
  You tried to open a log-file though there is another log-file already open. It is not allowed to open more than one log-file simultaneously.

- **ERROR: currently no log-file open**
  You tried to close a log-file using the 'logclose' command though there is no log-file currently open.

3.3 Saving the contents of the output window

You can save the contents of the output window not only with the *file->save output* or *file->save output as* menue, but also using the 'saveoutput' command. Saving the output window with the saveoutput command may be in particular useful in batch files, see Section 3.5. The syntax for saving the output window is

> saveoutput [*, options*] using *filename*

where 'filename' is the file in which the contents of the output is saved.

Options

- **replace**
  By default, an error will be raised if one tries to store the contents of the output window in a file that is already existing. This preserves you to overwrite a file unintendedly. An already existing file can be overwritten by explicitly specifying the 'replace' option.

- **type = rtf | txt**
  The output window can be saved under two different file types. By default, the contents of the window will be saved in rich-text format. The second possibility is to store the output window in plain ASCII–format. This can be done by specifying 'type = txt'. To explicitly store the file in rich text format *type = rtf* must be specified.
  DEFAULT: type = rtf

Possible Errors

- **ERROR: file *filename* is already existing**
  You tried to save the output window in a file that is already existing. Specify the replace option to overwrite an already existing file.

3.4 Changing the delimiter

By default, commands entered using the command window will be executed by pressing the return key. This can be inconvenient, in particular if your statements are large. In that case it may be more favourable to split a statement into several lines, and execute the command using a different delimiter than the return key. You can change the delimiter using the 'delimiter' command. The syntax is

delimiter = *newdel*

where 'newdel' is the new delimiter. There are only two different delimiters allowed, namely the 'return' key and the ';' (semicolon) key. To specify the ';' key as the delimiter, type

delimiter = ;

and press return. To return to the 'return' key as the delimiter, type

delimiter = return;

Note that the above statement must end with a semicolon, since this was previously set to the current delimiter.

3.5 Using batch-files

You can execute commands stored in a file just as if they were entered from the keyboard. This may be useful if you want to rerun a certain analysis more than once (possibly with some minor changes) or if you want to run time consuming statistical methods such as Bayesian regression based on MCMC simulation techniques (see Chapter 6). You can run such batch files by simply typing

> usefile *filename*

This executes the commands stored in 'filename' successively. *BayesX* will not stop the execution if an error occurs in one ore more commands. Note that it is allowed to invoke another batch file within a batch file currently running.

Comments

Comments in batch files are allowed, that is every line starting with a '%' sign is ignored by the program.

Changing the delimiter

In particular in such batch-files, the readability of your program code may be improved

if some (large) commands are split up over several lines. Normally this will cause errors, because *BayesX* interprets each line in your program as one statement. To overcome this problem one simply has to change the delimiter using the 'delimiter' command, see Section 3.4.

3.6 Dropping objects

You can eliminate objects by typing

drop *objectlist*

This drops the objects specified in 'objectlist'. The names of the objects in 'objectlist' must be separated by blanks.

Possible Errors

- **ERROR: objectlist required**
  You did not specify the objects to be eliminated in your drop statement.
- **ERROR: object *objectname* is not existing**
  The object 'objectname' specified in the 'objectlist' is not existing and can therefore not be eliminated.

Chapter 4

dataset objects

dataset objects are used to manage and manipulate data. A new dataset object is created by typing

> dataset *objectname*

where objectname is the name of the dataset. After the creation of a dataset object you can apply the methods for manipulating and managing datasets discussed below.
Note that in the current version of *BayesX* **only numerical variables are allowed**. Hence, string valued variables, for example, are not yet supported by *BayesX*.

4.1 Method 'drop'

Description

'drop' eliminates variables or observations from the dataset.

Syntax

objectname.drop *varlist*

objectname.drop if *expression*

The first command may be used to eliminate the variables specified in 'varlist' from the dataset. The second statement may be used to eliminate certain observations. An observation will be removed from the dataset if 'expression' is true, i.e. the value of the expression is one.

Options

not allowed

Examples

The statement

> credit.drop account duration

drops the variables 'account' and 'duration' from the credit scoring dataset. With the statement

> credit.drop if marstat = 2

all observations with 'marstat =2', i.e. all persons living alone, will be dropped from the credit scoring dataset. The following statement

> credit.drop account duration if marstat = 2

will raise the error

ERROR: dropping variables and observations in one step not allowed

It is not allowed to drop variables and certain observations in one single command.

Possible Errors:

- **ERROR: variable *varname* can not be found**
  The variable *varname* specified in *varlist* is not existing and can therefore not be eliminated from the dataset.

- **ERROR: varlist or boolean expression expected**
  There was neither a 'varlist' to drop certain variables nor a 'boolean expression' to drop certain observations in the 'drop' statement.

- **ERROR: dropping variables and observations in one step not allowed**
  In the drop statement both a *varlist* and a *boolean* expression is specified. You can not drop variables and observations with one single command. If you want to do both, you first have to drop some variables with the drop command, and then drop observations with a second drop statement or vice versa.

See also Section 4.2 **Functions and expressions** for possible expression syntax errors.

4.2 Functions and Expressions

The primary use of expressions is to generate new variables or change existing variables, see Sections 4.3 and 4.7, respectively. Expressions may also be used in 'if' statements to force *BayesX* to apply a method only to observations where the boolean expression in the 'if' statement is true. The following are all examples of expressions:

2+2
log(amount)
1*(age <= 30)+2*(age > 30 & age <= 40)+3*(age > 40)
age=30
age+3.4*age^2+2*age^3
amount/1000

4.2.1 Operators

BayesX has three different types of operators: arithmetic, relational and logical. Each of the types is discussed below.

Arithmetic operators

The arithmetic operators are + (addition), - (subtraction), * (multiplication), / (division), ^ (raise to a power) and the prefix - (negation). Any arithmetic operation on a missing value or an impossible arithmetic operation (such as division by zero) yields a missing value.

Example

The expression

(x+y^(3-x))/x*y

denotes the formula

$$\frac{x + y^{3-x}}{x \cdot y}$$

and evaluates to missing if x or y is missing or zero.

Relational operators

The relational operators are > (greater than), < (less than), >= (greater than or equal), <= (less than or equal), = (equal), and != (not equal). Relational expressions are either 1 (i.e. the expression is true) or 0 (i.e. the expression is false).

Examples

Relational operators may be used to create indicator variables. The following statement generates a new variable 'amountcat' (out of the already existing variable 'amount'), whose value is 1 if 'amount $\leq$ 10' and 2 if 'amount > 10'.

> credit.generate amountcat = 1*(amount<=10)+2*(amount>10)

Another useful application of relational operators is in 'if' statements. For example, changing an existing variable only when a certain condition holds can be done by the following command:

> credit.replace amount = NA if amount <= 0

This sets all observations missing where 'amount $\leq$ 0'.

Logical operators

The logical operators are & (and) and | (or).

Example

Suppose you want to generate a variable 'amountind' whose value is 1 for married people with amount greater than 10 and 0 otherwise. This can be done by typing

> credit.generate amountind = 1*(marstat=1 & amount > 10)

Order of evaluation of the operators

The order of evaluation (from first to last) of operators is
^
/,*
-, +
!=, >, <, <=, >=, =
&, |.

Brackets may be used to change the order of evaluation.

4.2.2 Functions

Functions may appear in expressions. Functions are indicated by the function name, an open and a close paranthesis. Inside the paranthesis one or more arguments may be specified. The argument(s) of a function may be any expression, including other functions. Multiple

arguments are separated by commas. All functions return 'missing' when given missing values as arguments or when the result is undefined.

Functions reference

Table 4.1 references all mathematical functions;Table 4.2 references all statistical functions.

Function	Description
abs(x)	absolute value
cos(x)	cosine of radians
exp(x)	exponential
floor(x)	returns the integer obtained by truncating x.
	Thus floor(5.2) evaluates to 5 as floor(5.8).
lag(x)	lag operator
log(x)	natural logarithm
log10(x)	log base 10 of x
sin(x)	sine of radians
sqrt(x)	square root

Table 4.1: List of mathematical functions.

4.2.3 Explicit subscribing

Individual observations on variables can be referenced by subscribing the variables. Explicit subscripts are specified by the variable name with square brackets that contain an expression. The result of the subscript expression is truncated to an integer, and the value of the variable for the indicated observation is returned. If the value of the subscript expression is less than 1 or greater than the number of observations in the dataset, a missing value is returned.

Examples

Explicit subscribing combined with the constant $_n$ can be used to create lagged values on a variable. For example the lagged value of a variable 'x' in a dataset 'data' can be created by

> data.generate xlag = x[_n-1]

Note that *xlag* can also be generated using the *lag* function

> data.generate xlag = lag(x)

Function	**Description**
bernoulli(p)	returns Bernoulli distributed random numbers with probability of success p. If p is not within the interval $[0;1]$, a missing value will be returned.
binomial(n,p)	returns $B(n;p)$ distributed random numbers. Both, the number of trials n and the probability of success p may be expressions. If $n < 1$, a missing value will be returned. If n is not integer valued, the number of trials will be $[n]$. If p is not within the interval $[0;1]$, a missing value will be returned.
cumul(x)	cumulative distribution function
cumulnorm(x)	cumulative distribution function Φ of the standard normal distribution.
exponential(λ)	returns exponential distributed random numbers with parameter λ. If $\lambda <\leq 0$, a missing value will be returned.
gamma(μ,ν)	returns gamma distributed random numbers with mean μ and variance μ^2/ν If μ and/or ν are smaller than zero, a missing value will be returned.
normal()	returns standard normal distributed random numbers; $N(\mu, \sigma^2)$ distributed random numbers may be generated with $\mu + \sigma$*normal().
uniform()	uniform pseudo random number function; returns uniformly distributed pseudo-random numbers on the interval $(0, 1)$

Table 4.2: List of statistical functions

4.2.4 Constants

Table 4.3 lists all constants that may be used in expressions.

Constant	Description
_n	contains the number of the current observation.
_N	contains the total number of observations in the dataset.
_pi	contains the value of π.
NA	indicates a missing value
.	indicates a missing value

Table 4.3: List of constants

Examples

The following statement generates a variable 'obsnr' whose value is 1 for the first observation, 2 for the second and so on.

```
> credit.generate obsnr = _n
```

The command

> credit.generate nrobs = _N

generates a new variable *nrobs* whose values are equal to the total number of observations, say 1000, for all observations.

4.2.5 Expression syntax errors

The following is a list of possible expression syntax errors:

- **ERROR: expression syntax error in *expression***
  This error message indicates a general syntax error in an expression. One possible reason for this error message is a forgotten closing bracket.

- **ERROR: *functionname* unknown function**
  A function with name 'functionname' is specified that is unknown for *BayesX*. See Section 4.1 and 4.2 for functions supported by *BayesX*.

- **ERROR: invalid number of arguments for function 'binomial'**
  The function 'binomial' expects two arguments separated by a comma. The first argument is assumed to be the number of trials, and the second the probability of success.

- **ERROR: argument not allowed in function 'functionname'**
  You specified an argument in function 'functionname' though no arguments are allowed for the specified function. See Tables 4.1 and 4.2 for correct function specifications.

- **ERROR: variable *varname* not found**
  A variable with name 'varname' specified in the expression statement is not existing.

4.3 Method 'generate'

Description

'generate' is used to create a new variable.

Syntax

objectname.generate *newvar* = *expression*

Method 'generate' creates a new variable with name 'newvar'. See Section 4.10 **Variable names** for valid variable names. The values of the new variable are specified by 'expression'. The details of valid expressions are covered in Section 4.2 **Functions and expressions**.

Options

not allowed

Examples

The following command generates a new variable called 'amount2' whose values are the square of amount in the credit scoring dataset.

> credit.generate amount2 = amount^2

If you try to change the variable currently generated, for example by typing

> credit.generate amount2 = amount^0.5

the error message

> ERROR: variable amount2 is already existing

will occur. This prevents you to change an existing variable unintendedly. An existing variable may be changed with method 'replace', see Section 4.7.
If you want to generate an indicator variable 'largeamount' whose value is 1 if amount exceeds a certain value, say 3.5, and 0 otherwise, the following will produce the desired result:

> credit.generate largeamount = 1*(amount>3.5)

Possible Errors:

- **ERROR: variable *varname* is already existing**
  This error message will appear if one tries to generate a variable with name 'varname' that is already existing in the dataset. To change an existing variable make use of method 'replace', see Section 4.7.

- **ERROR: invalid variable name specification**
  You tried to generate a new variable with an invalid variable name. See Section 4.10 **Variable names** for correct variable names.

See also Section 4.2 **Functions and expressions** for possible expression syntax errors.

4.4 Method 'infile'

Description

Reads in data saved in an ASCII-file.

Syntax

objectname.infile [*varlist*] [, *options*] using *filename*

Reads in data stored in 'filename'. The variables are given names specified in 'varlist'. If 'varlist' is empty, i.e. there is no 'varlist' specified, it is assumed that the first row of the datafile contains the variable names separated by blanks or tabs. It is not required that the observations in the datafile are stored in a special format, except that successive observations should be separated by one or more blanks (or tabs). The first value read from the file will be the first observation of the first variable, the second value will be the first observation of the second variable, and so on. However, an error will occur if for some variables no values can be read for the last observation.

It is assumed that a period '.' or 'NA' indicates a missing value.

Note that in the current version of *BayesX* **only numerical variables are allowed**. Thus, the attempt to read in string valued variables, for example, will cause an error.

Options

- ***missing = missingsigns***
  By default a period '.' or 'NA' indicates a missing value. If you have a dataset where missing values are indicated by different signs than the period '.' or 'NA', you can force *BayesX* to recognize these signs as missing values by specifying the 'missing' option. For example 'missing = MIS' defines MIS as an indicator for a missing value. Note that periods '.' and 'NA' remain valid indicators for missing values, even if the missing option is specified.

- ***maxobs = intvalue***
  If you work with large datasets, you may observe the problem that reading in a dataset using the 'infile' command is very time consuming. The reason for this problem is that *BayesX* does not know the number of observations to store in memory in advance. The effect is that new memory must be allocated whenever a certain amount of memory is used. To avoid this problem the 'maxobs' option may be used, leading to a considerable reduction of computing time. This option forces *BayesX* to allocate in advance enough memory to store at least 'intvalue' observations before new memory must be reallocated. Suppose for example that your dataset consists approximately of 100000 observations. Then specifying 'maxobs = 105000' allocates enough memory to read in the dataset quickly. Note that 'maxobs = 105000' does not mean that your dataset cannot hold more than 105000 observations. This means only that new memory will/must be allocated when the number of observations of your dataset exceeds the 105000 observations limit.

Examples

Suppose we want to read a dataset stored in 'c:\data\testdata.raw' containing two variables 'var1' and 'var2'. The first few rows of the datafile could look like this:

var1 var2
2 2.3
3 4.5
4 6
...

To read in this dataset, we first have to create a new dataset object, say 'testdata', and then read the data using the 'infile' command. The following two commands will produce the desired result.

```
> dataset testdata
> testdata.infile using c:\data\testdata.raw
```

If the first row in the dataset file contains no variable names, the second command must be modified to:

```
> testdata.infile var1 var2 using c:\data\testdata.raw
```

Suppose furthermore that the dataset you want to read in is a pretty large dataset with 100000 observations. In that case the maxobs option is very useful to reduce reading time. Typing for example

```
> testdata.infile var1 var2 , maxobs=101000 using c:\data\testdata.raw
```

will produce the desired result.

Posible Errors

- **ERROR: *observation* cannot be read as a number**
  The dataset contains a value 'observation' that can not be interpreted as a real valued number. For example, 'observation' could contain a letter. In the current version of *BayesX* **only numerical variables are allowed**.
- **ERROR: missing observations for one or more variable**
  There are not enough values for one or more variable for the last observation.
- **ERROR: *varname* invalid variable name**
  The variable name 'varname' appearing in 'varlist' or in the first row of the data file is invalid. See Section 4.10 **Variable names** for valid variable names.

4.5 Method 'outfile'

Description

'outfile' writes data to a disk file in ASCII format. The saved data can be read back using the 'infile' command, see Section 4.4.

Syntax

objectname.outfile [*varlist*] [if *expression*] [, *options*] using *filename*

'outfile' writes the variables specified in 'varlist' to the disk file with name 'filename'. If 'varlist' is omitted in the outfile statement, *all* variables in the dataset are written to disk. Each row in the data file corresponds to one observation. Different observations are separated by blanks. Optionally, an 'if' statement may be used to write only those observations to disk where a certain boolean expression, specified in 'expression', holds.

Options

- ***header***
  Specifying the 'header' option forces *BayesX* to write the variable names in the first row of the created data file.
- ***replace***
  The 'replace' option allows *BayesX* to overwrite an already existing data file. If 'replace' is omitted in the option list and the file specified in 'filename' is already existing, an error will be raised. This prevents you to overwrite an existing file unintendedly.

Examples

The statement

> credit.outfile using c:\data\cr.dat

writes the complete credit scoring dataset to 'c:\data\cr.dat'. To generate two different ASCII data sets for married people and people living alone, you could type

> credit.outfile if marstat = 1 using c:\data\crmarried.dat
> credit.outfile if marstat = 2 using c:\data\cralone.dat

Suppose you only want to write the two variables 'y' and 'amount' to disk. You could type

> credit.outfile boni amount using c:\data\cr.dat

This will raise the error message

ERROR: file c:\data\cr.dat is already existing

because 'c:\data\cr.dat' has already been created. You can overwrite the file using the 'replace' option

> credit.outfile boni amount , replace using c:\data\cr.dat

Possible Errors

- **ERROR: file *filename* is already existing**
  You tried to write to a file that is already existing. To prevent the user to overwrite an existing file unintentionally, *BayesX* must be forced explicitly to overwrite an existing file using the 'replace' option.
- **ERROR: variable *varname* can not be found**
  You tried to write the observations of a variable with name 'varname' to 'filename', which is not existing.

4.6 Method 'rename'

Description

'rename' is used to change variable names.

Syntax

objectname.rename *varname newname*

'rename' changes the name of 'varname' to 'newname'. 'newname' must be a valid variable name, see Section 4.10 on how to create valid variable names.

Options

not allowed

Possible Errors

- **ERROR: *newname* invalid varname**
  The specified new variable name *newname* is not a correct variable name. See Section 4.10 for valid variable names.

- **ERROR: variable *oldname* can not be found**
  You tried to rename a variable that is not existing in the dataset.
- **ERROR: variable *newname* is already existing**
  You tried to give a variable a new variable name that is already reserved for another variable in the dataset.

4.7 Method 'replace'

Description

'replace' changes the values of an existing variable.

Syntax

objectname.replace *varname* = *expression* [if *boolexp*]

'replace' changes the values of the existing variable 'varname'. If 'varname' is not existing, an error will be raised. The new values of the variable are specified in *expression*. Expressions are covered in Section 4.2. An optional 'if' statement may be used to change the values of the variable only if the boolean expression 'boolexp' is true.

Options

not allowed

Example

The statement

> credit.replace amount = NA if amount < 0

changes the values of the variable 'amount' in the credit scoring dataset to missing if $amount < 0$.

Possible Errors

- **ERROR: variable *varname* not found**
  You tried to change a variable that is not yet existing.

See also Section 4.2 **Functions and expressions** for possible expression syntax errors.

4.8 Method ‘set obs’

Description

’set obs’ changes the current number of observations in a dataset.

Syntax

objectname.set obs = *intvalue*

’set obs’ raises the number of observations in the dataset to ’intvalue’, which must be greater or equal to the current number of observations. This prevents you to destroy part of the data currently in memory. Observations may be eliminated using the ’drop’ statement, see Section 4.1. The values of the additionally created observations will be set missing.

Possible Errors

- **ERROR: new number of observations must be greater than the current number of observations** It is not allowed to specify a smaller number than the current number of observations. This prevents you to destroy part of the data currently in memory.

4.9 Method ‘sort’

Description

Sorts the dataset.

Syntax

objectname.sort *varlist* [*, options*]

Sorts the dataset with respect to the variables specified in varlist. Missing values are interpreted to be larger than any other number and are thus placed last.

Option

- ***descending***
  If this option is specified, the dataset will be sorted in descending order. The default is ascending order.

Possible errors

- **ERROR: variable *varname* can not be found**
  One of the variables specified in ’varlist’ is not existing.

4.10 Variable names

A valid variable name is a sequence of letters (A-Z and a-z), digits (0-9), and underscores (_). The first character of a variable name must be either a letter or an underscore. *BayesX* respects upper and lower case letters, that is 'myvar', 'Myvar' and 'MYVAR' are three distinct variable names.

4.11 Examples

This section contains two examples on how to work with *dataset objects*. The first example illustrates some of the methods described above, using one of the example datasets stored in the 'examples' directory, the credit scoring dataset. A description of this dataset can be found in Section 2.4.2. The second example shows how to simulate complex statistical models.

4.11.1 The credit scoring dataset

In this section we illustrate how to code categorical variables according to one of the coding schemes, dummy or effect coding. This will be useful in regression models, where all categorical covariates must be coded in dummy or effect coding before they are incorporated into the model.
We first create a *dataset object* 'credit' and read in the data using the *infile* command.

```
> dataset credit
> credit.infile using c:\bayes\examples\credit.raw
```

We can now generate new variables to obtain dummy coded versions of the categorical covariates 'account', 'payment', 'intuse' and 'marstat':

```
> credit.generate account1 = 1*(account=1)
> credit.generate account2 = 1*(account=2)
> credit.generate payment1 = 1*(payment=1)
> credit.generate intuse1 = 1*(intuse=1)
> credit.generate marstat1 = 1*(marstat=1)
```

The reference categories are chosen to be 3 for 'account' and 2 for the other variables. Alternatively, we could code the variables according to effect coding. This is achieved with the following program code:

```
> credit.generate account_eff1 = 1*(account=1)-1*(account=3)
> credit.generate account_eff2 = 1*(account=2)-1*(account=3)
> credit.generate payment_eff1 = 1*(payment=1)-1*(payment=2)
> credit.generate intuse_eff1 = 1*(intuse=1)-1*(intuse=2)
> credit.generate marstat_eff1 = 1*(marstat=1)-1*(marstat=2)
```

4.11.2 Simulating complex statistical models

In this section we illustrate how to simulate complex regression models. Suppose first we want to simulate data according to the following Gaussian regression model:

$$y_i = 2 + 0.5x_{i1} + sin(x_{i2}) + \epsilon_i, \quad i = 1, \dots, 1000 \quad (4.1)$$
$$x_{i1} \sim U(-3, 3) \quad i.i.d. \quad (4.2)$$
$$x_{i2} \sim U(-3, 3) \quad i.i.d. \quad (4.3)$$
$$\epsilon_i \sim N(0, 0.5^2) \quad i.i.d. \quad (4.4)$$

We first have to create a new dataset 'gsim', say, and specify the desired number of observations:

```
> dataset gsim
> gsim.set obs = 1000
```

In a second step the covariates x1 and x2 have to be created. In this first example we assume that the covariates are uniformly distributed between -3 and 3. To generate them, we must type:

```
> gsim.generate x1 = -3+6*uniform()
> gsim.generate x2 = -3+6*uniform()
```

In a last step we can now create the response variable by typing

```
> gsim.generate y = 2 + 0.5*x1+sin(x2)+0.5*normal()
```

You could now (if you want) estimate a Gaussian regression model with the generated dataset using a *bayesreg object*, see Chapter 6. Of course, more refined models could be simulated. We may for example drop the assumption of a constant variance 0.5^2 in the error term. Suppose the variance is heteroscedastic and growing with order log(i) where i is the observation index. We can simulate such a heteroscedastic model by typing:

```
> gsim.replace y = 2 + 0.5*x1+sin(x2)+0.1*log(_n+1)*normal()
```

In this model the standard deviation is

$$\sigma_i = 0.1 * log(i + 1), \quad i = 1, \dots, 1000.$$

Suppose now that we want to simulate data from a logistic regression model. In a logistic regression model it is assumed that (given covariates) the response variable y_i, $i = 1, \dots, n$, is binomial distributed with parameters n_i and π_i where n_i is the number of replications and π_i is the probability of success. For π_i one assumes that it is related to a linear predictor η_i via the logistic distribution function, that is

$$\pi_i = \frac{\exp(\eta_i)}{1 + \exp(\eta_i)}.$$

To simulate such a model we have to specify the linear predictor η_i and the number of replications n_i. We specify a similar linear predictor as in the example above for Gaussian response, namely

$$\eta_i = -1 + 0.5 x_{i1} - sin(x_{i2}).$$

For simplicity, we set $n_i = 1$ for the number of replications. The following commands generate a dataset 'bin' according to the specified model:

```
> dataset bin
> bin.set obs = 1000
> bin.generate x1 = -3+6*uniform()
> bin.generate x2 = -3+6*uniform()
> bin.generate eta = -1+0.5*x1-sin(x2)
> bin.generate pi = exp(eta)/(1+exp(eta))
> bin.generate y = binomial(1,pi)
```

Note that the last three statements can be combined into a single command:

```
> bin.generate y = binomial(1,exp(-1+0.5*x1-sin(x2))/(1+exp(-1+0.5*x1-sin(x2)))
```

However, the first version is much easier to read and should therefore be prefered.

Chapter 5

map objects

map objects are used to handle and store geographical maps. For the moment map objects serve more or less as auxiliary objects for *bayesreg objects*, where the effect of spatial covariates on a dependent variable can be modelled via Markov Random Field priors. The main purpose of *map objects* in this context is to provide the neighborhood structure of the map, and to compute weights associated with this neighborhood structure. The typical approach is as follows: A map object is created and the boundary information of a geographical map is read from an external file and stored in the map object. This can be achieved using the 'infile' command, see Section 5.1 below. Based on the boundary information, the map object automatically computes the neighborhood structure of the map and the weights associated with the neighborhood structure. Since there are several proposals in the spatial statistics literature for defining the weights, the user is given the choice between a couple of alternative weight definitions. After the correct initialization, the map object can be passed to the 'regress' function of a *bayesreg object* in order to estimate regression models with spatial covariates, see Chapter 6, in particular Section 6.1, and the subsections about spatial covariates therein.

5.1 Method 'infile'

Description

Method 'infile' is used to read the boundary information of a geographical map stored in an external file. This file is called a *boundary file*, since it must contain the information about the boundaries of the different regions of the map. It is assumed that the boundary of each region is stored in form of a closed polygone, that is the boundary is represented by a set of connected straight lines. A detailed description of the structure of boundary files is given below.

Syntax

objectname.infile [*, options*] using *filename*

Method 'infile' reads the map information stored in the boundary-file 'filename'. The required structure of boundary files is as follows: A boundary file provides the boundary information of a geographical map. For each region of the map the boundary file must contain the identifying name of the region, the polygones that form the boundary of the region, and the number of lines the polygone consists of. The first line always contains the region name surrounded by quotation marks and the number of lines the polygone of the region consists of. The name and the number of lines must be separated by a comma. The subsequent lines contain the coordinates of the straight lines that form the boundary of the region. The straight lines are represented by the coordinates of their end points. Coordinates must be separated by a comma.

To give an example we print a (small) part of the boundary file of (former) West Germany:

```
      ⋮
"6634",31
2319.26831,4344.48828
2375.45435,4399.50391
2390.67139,4446.32520
2470.26807,4405.35645
2576.78735,4379.60449
2607.22144,4337.46533
2627.12061,4356.19385
2662.23682,4355.02344
2691.50024,4311.71338
2726.61646,4310.54248
2716.08154,4256.69775
2710.22900,4227.43408
2680.96533,4234.45752
2583.81055,4165.39551
2568.59351,4096.33398
2520.60132,4042.48901
2535.81836,3941.82251
2490.16724,3920.75269
2451.53955,3903.19458
2437.49292,3924.26440
2369.60156,3933.62866
2359.06665,3951.18677
2285.32275,3969.91553
2258.40015,4061.21753
2197.53223,4049.51221
2162.41602,4086.96948
2204.55542,4091.65161
2192.85010,4125.59717
2284.15210,4220.41113
2339.16748,4292.98438
2319.26831,4344.48828

      ⋮
```

The corresponding graph to the section of the boundary file above can be found in Figure 5.1. Note that the first and the last point must be identical (see the example above) to obtain a closed ploygone.

In some cases it might happen that a region is separated into subregions that are not connected. As an illustrative example compare Figure 5.2 showing a region of Germany that is

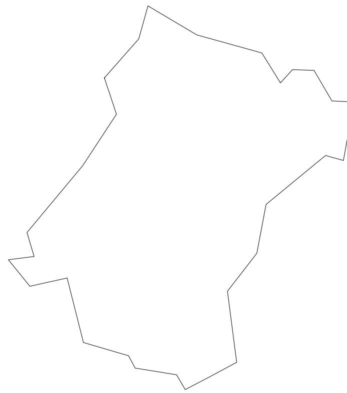


Figure 5.1: Corresponding graph of the section of the boundary file

divided into 8 subregions. In this case the boundary file must contain the polygones of all subregions. The first row for these subregions must contain the regions name and the number of lines the polygone of the respective subregion consists of. Note that it is not necessary that the polygones of the subregions are stored in subsequent order in the boundary file.

Another special case that might happen is illustrated in figure 5.3. Here a region is totally surrounded by another region. In this case an additional line must be added to the boundary description of the *surrounded* region. The additional line must be placed just after the first line and must contain the name of the *surrounding* region. The syntax is:

is.in,"*region name*"

The following lines show a section of the boundary file of West Germany where region "9361" is totally surrounded by region "9371":

```
  ⋮
"9361",7
is.in,"9371"
4155.84668,2409.58496
4161.69922,2449.38330
4201.49756,2461.08862
4224.90820,2478.64673
4250.66016,2418.94922
4193.30371,2387.34448
4155.84668,2409.58496

  ⋮
```

Finally, we want to call attention to an important limitation in the current version of *BayesX*. In most cases *map objects* serve as auxiliary objects to estimate spatial random effects with *bayesreg objects*. In this case the names of the regions of the map and the values of the spatial covariate, whose effect is estimated, must match. Since there are only numerical variables allowed in *dataset objects* (and no string valued variables), the names of the regions in the corresponding *map object* must necessarily be numbers, although there is in principle no limitation for the names of regions in *map objects*.

Option

- **weightdef=adjacency | combnd | centroid**
  Option 'weightdef' allows to specify how the weights associated with each pair of neighbors are computed. Currently there are three weight specifications available, 'weightdef=adjacency', 'weightdef=centroid' and 'weightdef=combnd'. If 'weightdef=adjacency' is specified, for each pair of neighbors the weights are equal to one. This so called adjacency weights are most common in spatial statistics. Specifying 'weightdef=centroid' results in weights proportional to the distance of the centroids of neighboring regions. Setting 'weightdef=combnd' results in weights proportional to the length of the common boundary.

Figure 5.2: Example for a region that is divided into subregions

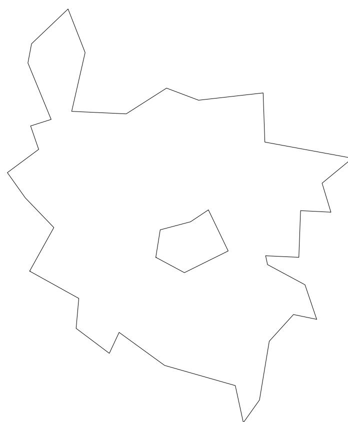


Figure 5.3: Example for a region that is totally surrounded by another region

Chapter 6

bayesreg objects

bayesreg objects are used to fit generalized linear models with a parametric, semiparametric or nonparametric predictor. Besides the possibility of incorporating nonlinear effects of metrical covariates, the procedure also allows to estimate (unstructured) random effects, time varying seasonal effects as well as nonlinear effects of spatial covariates. In addition, *bayesreg objects* provide several possibilities for estimating interaction effects between two covariates, which is based mainly on the varying coefficients framework introduced by Hastie, Tisbshirani (1993). Inference is fully Bayesian via Markov Chain Monte Carlo (MCMC) techniques. It would be beyond the scope of this manual to describe the models and estimation techniques used in full detail, though Section 6.1 covers the methodology in some detail. A detailed description of the estimation techniques can be found in Fahrmeir, Lang (1999), but see also Knorr-Held(1996), Knorr-Held(1999) and Fahrmeir, Knorr-Held (1997). A German description can be found in Lang (1996). Good introductions to Generalized Linear Models are the monographs of Fahrmeir, Tutz (1997) and Mc Cullagh, Nelder (1989). For an introduction to Bayesian Generalized Linear Mixed Models see Clayton (1996). Introductions to semi- and nonparametric models are given in Green, Silverman (1994), Hastie, Tibshirani (1990) and Hastie, Tibshirani (1993). Smoothing methods for spatial data, in particular geographical data, are covered in the papers by Besag, York and Mollie (1991) and Besag, Kooperberg (1995). Finally, the paper of Chib and Greenberg (1995) and the monograph *Markov Chain Monte Carlo in Practice* edited by Gilks, Richardson and Spiegelhalter (1996) give good surveys on MCMC simulation techniques. A German introduction into MCMC techniques can be found in Lang (1996) or Biller (1999).

6.1 Method 'regress'

Description

Method 'regress' fits generalized linear models and nonparametric extensions in a Bayesian framework. Generalized linear models assume that, given covariates w, the distribution of the dependent or response variable y belongs to an exponential family with mean $\mu = E(y|w)$ linked to a predictor η by $\mu = h(\eta)$. Here h is a known link or response function. Traditionally the effect of the covariates on the response is assumed to be linear, i.e.

$$\eta = \beta_1 w_1 + \cdots + \beta_q w_q = \beta' w. \tag{6.1}$$

In a Bayesian framework suitable priors for the unknown parameters $\beta_1, \ldots, \beta_q$ have to be chosen. In the absence of any prior knowledge diffuse priors are the appropriate choice, i.e.

$$\beta_i \propto const \quad i = 1, \ldots, q.$$

Another common choice also supported by *BayesX* are informative multivariate Gaussian priors with mean μ_0 and covariance matrix Σ_0.

Moreover *BayesX* allows semiparametric or even fully nonparametric predictors. Suppose that we have additional covariates $x = (x_1, \ldots, x_p)$ whose effect is assumed to be nonlinear. Then we may replace the simple linear predictor (6.1) by a more flexible semiparametric additive predictor

$$\eta = f_1(x_1) + \cdots + f_p(x_p) + \beta' w$$

where the unknown functions $f_1, \ldots, f_p$ are (more or less) smooth functions of the covariates. A prior for a function f depends on the type of covariate x and on prior beliefs in smoothness of f.

Time scales and metrical covariates

Suppose first that x is a time scale or metrical covariate with equally spaced ordered observations

$$x_{(1)} < x_{(2)} < \cdots < x_{(m)}.$$

Here m denotes the number of *different* observed values for x in the dataset. Define $f(t) := f(x_{(t)})$ for $t = 1, \ldots, m$. Common smoothness priors for the vector $f = (f(1), \ldots, f(m))'$ of function evaluations are first or second order random walk models

$$f(t) = f(t-1) + u(t) \quad \text{or} \quad f(t) = 2f(t-1) - f(t-2) + u(t)$$

with Gaussian errors $u(t) \sim N(0; \tau^2)$ and diffuse priors $f(1) \propto$ const, and $f(1)$ and $f(2) \propto$ const, for initial values, respectively. Both specifications act as smoothness priors penalizing too rough functions f. A first order random walk penalizes too abrupt jumps $f(t) - f(t-1)$ between successive states and a second order random walk penalizes deviations from the linear trend $2f(t-1) - f(t-2)$. In the case of non-equally spaced observations slight modifications of the priors defined above are necessary, see Fahrmeir, Lang (1999) for details. If x is a time scale we may also introduce an additional seasonal effect of x. A common smoothness prior for a seasonal component is

$$f(t) + f(t-1) + \ldots + f(t - per - 1) = u(t) \sim N(0, \tau^2). \tag{6.2}$$

where *per* is the period, for example $per = 12$ for monthly data. Compared to a dummy variable approach this specification has the advantage that it allows for a time varying seasonal effect rather than a time constant seasonal pattern.

The amount of smoothness of a function f is controlled by the variance parameter τ^2. For a fully Bayesian analysis, a hyperprior for τ^2 is introduced in a further stage of the

hierarchy. This allows for simultaneous estimation of the unknown function and the amount of smoothness. A common choice is a highly dispersed inverse gamma prior

$$p(\tau^2) \propto IG(a,b).$$

Usual values for the hyperparameters a and b are $a = b$, for example $a = b = 0.00001$, leading to diffuse priors for the variance parameters. An alternative proposed, for example, in Besag et al. (1995) is $a = 1$ and a small value for b, such as $b = 0.005$, which is the default in *BayesX*.

Spatial covariates

Let us now turn our attention to a spatial covariate x, where the values of x represent the location or site in connected geographical regions. For example in the rents for flats example (see Section 2.4.1) x indicates the location of the flats in Munich. A common way to deal with spatial covariates is to assume that neighboring sites are more alike than two arbitrary sites. Thus for a valid prior definition a set of neighbors for each site x_t must be defined. For geographical data as considered here one usually assumes that two sites x_t and x_j are neighbors if they share a common boundary.
The simplest spatial smoothness prior for the function evaluations $f(t)$, $t = 1, \dots, m$, of the m different sites x_t is

$$f(t)|f(j)\, j \neq t, \tau^2 \sim N\left(\sum_{j \in \partial_j} f(j)/N_t, \tau^2/N_t\right), \tag{6.3}$$

where N_t is the number of adjacent sites and $j \in \partial_t$ denotes that site x_j is a neighbor of site x_t. Thus the (conditional) mean of $f(t)$ is an unweighted average of function evaluations of neighboring sites.
A more general prior including (6.3) as a special case is given by

$$f(t)|f(j)\, j \neq t, \tau^2 \sim N\left(\sum_{j \in \partial_t} w_{tj}/w_{t+} f(j), 1/w_{t+}\tau^2\right), \tag{6.4}$$

where w_{tj} are known (not necessarily) equal weights and $+$ denotes summation over the missing subscript. Such a prior is called a Gaussian intrinsic autoregression, see Besag, York and Mollie (1991) and Besag and Kooperberg (1995). Other weights than $w_{tj} = 1$ as in (6.3) are based on the common boundary length of neighboring sites, i.e. $w_{tj} =$ *length of the common boundary*, or on the distance of the centroids of two sites. All these spatial priors are supported by *BayesX*.
As is the case for metrical covariates, the amount of smoothness is controlled by the variance parameter τ^2 for which once again a highly dispersed gamma prior is chosen.

It turns out that all smoothness priors discussed so far can be written in terms of a penalty matrix K, i.e.

$$f|\tau^2 \propto \exp\left(-\frac{1}{2\tau^2} f' K f\right). \tag{6.5}$$

For example, for a random walk of first order with equidistant x-values, the penalty matrix is given by:

$$K = \begin{pmatrix} 1 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 1 \end{pmatrix}$$

Random effects

The observation models discussed above may be appropriate if heterogeneity among units is sufficiently described by covariates. A common way to deal with the problem of unobserved heterogeneity is the inclusion of additive random effects leading to the predictor

$$\eta = f_1(x_1) + \cdots + f_p(x_p) + \beta' w + b_g, \tag{6.6}$$

where $g \in \{1, \ldots, G\}$ is the grouping variable. In most cases the grouping variable is a index that identifies different units. It may also be the district in which a certain person lives to incorporate spatial heterogeneity. Of course, estimating more than one random effect according to some other grouping variable is possible and supported by *BayesX*. For random effects, a usual assumption for the prior is that the b_g's are i.i.d. Gaussian,

$$b_g | v^2 \sim N(0, v^2), \quad g = 1, \ldots, G \tag{6.7}$$

and define again a highly dispersed hyperprior for v^2.

Apparently, there is only a slight difference to the additive effects $f_1, \ldots, f_p$. In fact, instead of specifying first or second order random walk priors for a function f the random effects prior (6.7) may also be specified. The main difference between the two specifications is the amount of smoothness allowed for a function f. With a random effects specification succeeding parameters are allowed to vary more or less *unrestricted*, whereas random walk priors guarantee that succeeding parameters vary smoothly over the range of x.

Interactions

Suppose that we want to model an interaction effect between a dichotomous variable x_1 and a metrical covariate x_2. A common way to deal with such interactions are varying coefficients models introduced by Hastie and Tibshirani (1993). In varying coefficients models it is assumed that the effect of a particular covariate (here x_1) is not fixed but varies smoothly over the domain of a second covariate (here x_2) leading to the predictor

$$\eta = \cdots + f(x_2)x_1 + \cdots. \tag{6.8}$$

In the context of varying coefficients models covariate x_2 is called the effect modifier. For the unknown function f an appropriate smoothness prior must be defined. As for simple additive model terms first or second order random walks are possible specifications.

Several generalizations of the varying coefficients model specified in (6.8) are possible. Obviously, we are not restricted to a categorical covariate as the interacting variable, that is

x_1 may also be metrical. This defines an interaction effect between two metrical covariates. However, this is a very special interaction effect and one has to check carefully whether such a specification is justified or not. Alternatives are 2-dimensional surfaces $f(x_1, x_2)$ where f is now a smooth 2-dimensional function, but so far this is not supported by *BayesX*. Another possibility is to allow also spatial covariates as the effect modifier and define the spatial smoothness priors (6.3) or (6.4) for f. If for example the interacting variable x_1 is time t this would define a (special) kind of space-time interaction. Although such models are supported by *BayesX* we should stress that a large amount of data is necessary to estimate such models accurately.

Bayesian Inference

Bayesian inference is based on the posterior distribution of the model. In many practical situations (as is the case here) the posterior distribution is numerically intractable. A common technique to overcome these problems are Markov Chain Monte Carlo (MCMC) simulation methods that became very popular recently. MCMC methods allow the drawing of random numbers from the numerically intractable posterior distribution and in this way the estimation of characteristics of the posterior like means, standard deviations or quantiles via their empirical analogue. The main idea is very simple. Instead of drawing directly from the posterior (which is impossible in most cases anyway) a markov chain is created, whose iterations of the transition kernel converge to the posterior. In this way a sample of dependent random numbers of the posterior is obtained. As a rule, the first part of the sample is discarded to take into account the time the algorithm needs for convergence to the posterior. This is known as burnin period. In *BayesX* the user has some control over the MCMC simulations to fit a certain model by specifying certain options. Among others there are options to specify the number of burnin iterations and the total number of iterations, see the options list below.

The exact MCMC simulation techniques used are described in detail in Fahrmeir, Lang (1999) and Fahrmeir, Lang (2000). Here we only give a brief overview.

Suppose first that the distribution of the response variable is Gaussian. In that case full conditionals for fixed effects, nonparametric functions f_j and also for random effects are multivariate Gaussian. Thus a simple Gibbs sampler can be used where posterior samples are drawn directly from the multivariate Gaussian distributions. For example the full conditional $\beta|\cdot$ for fixed effects with diffuse priors is Gaussian with mean

$$E(\beta|\cdot) = (W'W)^{-1}W'(y - \tilde{\eta})$$

and covariance matrix

$$Cov(\beta|\cdot) = \sigma^2(W'W)^{-1}$$

where W is the designmatrix of fixed effects and $\tilde{\eta}$ is the part of the linear predictor associated with the other effects in the model (for example nonparametric terms). Similarly the full conditional for a nonparametric function $f = (f(1), \ldots, f(m))$ is Gaussian with mean

$$E(f|\cdot) = \left(\frac{1}{\sigma^2}X'X + \frac{1}{\tau^2}K\right)^{-1} \frac{1}{\sigma^2}X'(y - \tilde{\eta})$$

and covariance matrix

$$Cov(f|\cdot) = \left(\frac{1}{\sigma^2}X'X + \frac{1}{\tau^2}K\right)^{-1}.$$

Here the matrix X is a $n \times m$ matrix whose element in the i-th row of the j-th column is one if observation i belongs to parameter $f(j)$ and zero else. It is easy to see, that $X'X$ is a $m \times m$ diagonal matrix where the j-th diagonal element is the number of observations belonging to the respective parameter $f(j)$. The matrix K is the penalty matrix. Although the full conditional is Gaussian, drawing random samples in an efficient way is not trivial, since linear equation systems with a high dimensional precision matrix must be solved in every iteration of the MCMC scheme. However, since all matrices involved are band matrices, random samples from the full conditional can be drawn in a very efficient way using cholesky decompositions for band matrices. Moreover the resulting Markov chain has superior mixing properties leading to almost independent samples. More details can be found in Harvard-Rue (2000).

Let us now turn our attention to models with non-Gaussian response. In this case full conditionals are no longer Gaussian, so that more refined algorithms are needed. For fixed effects and also random effects we use a slightly modified version of the weighted least squares proposal suggested by Gamerman (1997), see Fahrmeir, Lang (1999) for details. For updating a smooth function f we adopted and extended a MH-algorithm with conditional prior proposals developed recently by Knorr-Held (1999) in the related context of dynamic models. Here the proposal is drawn from the conditional prior distribution

$$p(f(t)|f(l), l \neq t, \tau^2)$$

of $f(t)$ given the remaining parameters and the variance parameter. Convergence and mixing is considerably improved by block moves, where blocks $f[r,s] = (f(r), \dots, f(s))$ of parameters are updated instead of single parameters $f(t)$, i.e. drawing proposals from

$$p(f[r,s]|f(l), l \notin [r,s], \tau^2),$$

which can be shown to be multivariate Gaussian. The blocksize depends on the analysed dataset and has to be specified by the user. To be more specific, a minimal and maximal blocksize must be specified, then in every iteration of the MCMC simulation the procedure chooses the blocksize randomly between the specified minimal and maximal size. If the mixing of the chain is poor, blocksizes should be increased, if the acceptance rates are too small, the blocksizes should be decreased. As a rule of thumb, acceptance rates between 30 and 80 percent guarantee a good mixing of the chain in most cases.

Less complicated is the updating of a variance parameter τ^2 (for Gaussian as well as for non Gaussian response). Since the full conditional of τ^2 is still an inverse gamma distribution, updating can be done by simple Gibbs steps, drawing directly from the full conditional distribution.

We summarize the resulting hybrid MCMC algorithm for non Gaussian response in the following overview:

(i) Partition vector $f_j, j = 1, 2, \dots$ of function evaluations into subvectors $f_j = (f_j^{(1)}, ..., f_j^{(r)}, \dots)$ and draw from

$$p(f_j^{(r)}|\cdot), \quad r = 1, 2, \dots, \quad j = 1, 2, \dots$$

with MH steps using conditional prior proposals.

(ii) Draw samples for random parameters $b_g, g = 1, \dots, G$, and fixed parameters β from

$$p(b_g|\cdot) \quad \text{and} \quad p(\beta|\cdot)$$

by weighted least squares proposals.

(iii) Draw samples for variances $\tau_j^2, j = 1, \dots$ from inverse Gamma posteriors

$$p(\tau_j^2|\cdot) \sim IG(a_j', b_j')$$

with updated parameters a_j', b_j'.

(iv) Repeat steps (i), (ii), (iii) until enough samples are obtained, omitting samples from the burnin phase.

For Gaussian response the scheme is similar and therefore omitted.

Syntax

objectname.regress *model* [weight *weightvar*] [if *expression*] [, *options*] using *dataset*

'regress' estimates the regression model specified in 'model' using the data specified in 'dataset'. 'dataset' must be the name of a dataset object created before. The distribution of the response variable can be either Gaussian, binomial or Poisson. It is specified using option 'family', see the options list below for details. The default is 'family=binomial'. The details of correct models are covered in the next subsection. An 'if' statement may be specified to analyse only a part of the dataset, i.e. the observations where 'expression' is true. An optional weight variable *weightvar* may be specified to estimate weighted regression models. If the response distribution is binomial, it is assumed that the values of the weight variable correspond to the number of replications and that the values of the response variable correspond to the number of successes. If weight is omitted, *BayesX* assumes that the number of replications is one, i.e. the values of the response must be either zero or one.

Syntax for models

The syntax for models is:

$depvar = term_1 + term_2 + \cdots + term_r$

'depvar' specifies the dependent variable in the model and $term_1$,...,$term_r$ define in which way the covariates influence the dependent or response variable. The different terms must be separated by '+' signs. There are several different possibilities to specify the influence of a certain covariate on the response.

Fixed effects

Traditionally the effect of covariates W1,W2,...,Wq on the response, say Y, is assumed to be linear, i.e. the predictor η of the model is

$$\eta = \beta_0 + \beta_1 W1 + \cdots + \beta_q Wq.$$

The following model statement causes *BayesX* to estimate this model:

Y = const + W1 + W2 + W3 + $\cdots$ + Wq

The additional *const* expression indicates a constant intercept in the model. To estimate a model without the intercept one simply has to omit *const* in the above model expression.

In many cases covariates with fixed effects are categorical. In this case one typically has to recode the categorical covariate according to a coding scheme. Common coding schemes for categorical covariates are dummy coding and effect coding, see for example Fahrmeir, Tutz (1997) for details. One possible way of dealing with categorical covariates in *BayesX* is to generate a set of dummy variables and incorporate them into the model statement as fixed effects. In most cases this is very time consuming and therefore *BayesX* provides a better possibility of incorporation categorical covariates into the model. As a side effect a lot of computing time is saved, because the special structure of the designmatrix of categorical covariates can be used to speed up computations. Suppose for example that covariate W1 is categorical with categories 1,2,3,4 and 5. Then the model statement

X1(cat,2)

forces *BayesX* to treat X1 as a categorical covariate meaning that X1 is automatically coded in effect coding with reference category 2. Thus the second argument in the above statement specifies the value of the reference category. This argument must not be omitted.

A disadvantage of the proposed approach is that in some cases the mixing of sampled parameters may worsen because parameter blocks become smaller.

Nonlinear effects

In many practical situations, however, the effect of a covariate is clearly nonlinear. Suppose we have an additional metrical covariate $X1$, whose effect is assumed to be nonlinear. Then we may extend the simple predictor above by

$$\eta = \beta_0 + f_1(X1) + \beta_1 W1 + \beta_2 W2 + \cdots + \beta_q Wq,$$

where f_1 is assumed to vary smoothly over the course of X1. In *BayesX*, smoothness of f_1 is guaranteed by specifying a smoothness prior for f_1. Appropriate priors for metrical covariates currently supported by *BayesX* are first and second order random walk models. The following model statement defines a second order random walk prior for f_1.

Y = const + X1(rw2,4,8) + W1 + W2 + ⋯ + Wq

Here the expression X1(rw2,4,8) indicates, that the effect of X1 should be incorporated nonparametrically into the model using a second order random walk prior. The second and third argument in the expression above are more technical and specific to the used MCMC inference technique. These arguments define the minimal and maximal blocksize for the block move updates of the parameters of f_1. In the example above, *BayesX* is forced to choose the blocksize between 4 and 8. Note that the specification of the minimum and maximum blocksize is optional and can be omitted. In that case the minimum and maximum blocksize are chosen to be one, leading to a single move algorithm. For Gaussian response the specification of a minimum and maximum blocksize is not meaningful, since in this case parameters are updated in one step by drawing directly from the full conditional, which is multivariate Gaussian.

Similarly a first order random walk is specified in the model statement by modifying the first argument in X1(rw2,4,8) from 'rw2' to 'rw1' yielding the term X1(rw1,4,8). Suppose now we have p covariates X1,...,Xp with possibly nonlinear effects. In that case the following model statement is one possibility to estimate such a model:

Y = const + X1(rw2,4,8) + X2(rw1,3,7) + ... + Xp(rw2,2,9) + W1 + ⋯ + Wq

This corresponds to the predictor

$$\eta = \beta_0 + f_1(X1) + \cdots + f_p(Xp) + \beta_1 W1 + \beta_2 W2 + \cdots + \beta_q Wq.$$

However, this is meaningful only if all X-covariates are metrical.

As an extension to existing methods for estimating semiparametric models, *BayesX* also allows the incorporation of an additional seasonal effect for a time scale and even the appropriate incorporation of spatial covariates using one of the Markov random field priors (6.3) or (6.4) for spatial data. A seasonal component for a time scale X1 is specified for example by

X1(seasonal,12,4,8).

Here the second argument specifies the period of the seasonal effect. In the example above the period is 12, corresponding to monthly data. Note that the second argument is not optional and an error will be raised if omitted. In analogy to the specification of first or second order random walks the third and fourth argument define the minimal and maximal blocksize. The specification of a Markov random field prior for spatial data has only two arguments. Here the second argument must be the name of a map object (see Chapter 5) that holds all necessary spatial information about the geographical map, including the polygones of the different geographical regions, the neighbors of each region and the weights that should be associated to the neighbors. For example the statement

X1(spatial,germany)

defines a Markov random field prior for X1 where the geographical information is stored in the map object 'germany'. An error will be raised if 'germany' is not existing. The blocksize

for spatial effects is alway chosen to be one. Although a block move is in principle possible experience shows that in many cases the conditional precision matrices are not invertable because of highly unstructured penalty matrices.

Finally we note that *BayesX* also supports additive random effects to cope with unobserved heterogeneity among units. Suppose the analysed dataset contains a index variable 'indivnr' that gives information about the individuum a certain observation belongs to. Then an individuum specific random effect is incorporated through the term

indivnr(random).

Of course more than one random effects term in the model is allowed.

Interaction effects

BayesX provides several possibilities of incorporating interaction effects based on varying coefficients models. For example a varying coefficients term with a second order random walk smoothness prior is defined as follows:

X1*X2(rw2,5,8)

This corresponds to the predictor

$$\eta = \ldots + f(X2)X1 + \ldots$$

where the effect of X1 varies smoothly over the course of X2. Similar to simple additive effects, the second and third argument specify the minimum and maximum blocksize for blockmoves. Of course a first order random walk as smoothness prior is also possible. Moreover not only metrical covariates are allowed as the effect modifier but also spatial covariates. For example the statement

X1*X2(spatial,germany)

defines a varying coefficients term with the spatial covariate X2 as the effect modifier and the spatial smoothness prior (6.3) or the more general prior (6.4) depending on the weight definition in the map object 'germany'.

Tables 6.2 and 6.3 summarize the different possibilities of incorporating covariates into the model, see pages 78 and 79.

Options

Options for specifying prior distributions

Options for specifying prior distributions are listed in alphabetical order.

- **a = *realvalue***
  Defines the value of the hyperparameter a for the inverse gamma prior of variance

parameters in nonparametric terms. *realvalue* must be a positive real valued number.
DEFAULT: a = 1

- **aresp = *realvalue***
  Defines the value of the hyperparameter a for the inverse gamma prior of the overall variance parameter σ^2, if the response distribution is gaussian. *realvalue* must be a positive real valued number.
  DEFAULT: aresp = 1

- **b = *realvalue***
  Defines the value of the hyperparameter b for the inverse gamma prior of variance parameters in nonparametric terms. *realvalue* must be a positive real valued number.
  DEFAULT: b = 0.005

- **bresp = *realvalue***
  Defines the value of the hyperparameter b for the inverse gamma prior of the overall variance parameter σ^2, if the response distribution is gaussian. *realvalue* must be a positive real valued number.
  DEFAULT: bresp = 0.005

Options for controlling MCMC simulations

Options for controlling MCMC simulations are listed in alphabetical order.

- **burnin = *intvalue***
  Changes the number of burnin iterations to *intvalue*, where *intvalue* must be a positive integer number or zero (i.e. no burnin period). The number of burnin iterations must be smaller than the number of iterations (see option iterations).
  DEFAULT: burnin = 2000

- **condprior**
  This option is meaningful only for Gaussian response. In this case full conditionals for nonparametric terms (as well as for fixed effects) are all multivariate Gaussian, that is MH-steps for updating parameters can be replaced by simple Gibbs steps by drawing random numbers directly from the full conditional. However, since updating via conditional prior proposals is still possible, the user is given the choice between the two alternatives. By default, parameters of a nonparametric term are updated in one single block by drawing random numbers directly from the full conditional. To force *BayesX* to update parameters via conditional prior proposals, simply specify 'condprior' as an additional option in the option list. Note that the specification of 'condprior' will have no effect if the response distribution is non-Gaussian.

- **iterations = *intvalue***
  Changes the number of MCMC iterations to *intvalue*, where *intvalue* must be a

positive integer number. The number of iterations must be larger than the number of burnin iterations.
DEFAULT: iterations = 52000

- **maxint** = ***invalue***
  If first or second order random walk priors are specified, in some cases the data will be grouped slightly. That is the range between the minimal and maximal observed covariate values will be divided into (small) intervals, and for each interval one parameter will be estimated. The grouping has almost no effect on estimation results as long as the number of intervals is large enough. With the 'maxint' option the amount of grouping can be determined by the user. *intvalue* is the maximum number of intervals allowed. For equidistant data, maxint=150 for example, means that no grouping will be done as long as the number of *different* observations is equal to or below 150. For non equidistant data some grouping may be done even if the number of different observations is below 150.
  DEFAULT: maxint=150

- **step** = ***intvalue***
  Defines the thinning parameter for MCMC simulation. For example, step=50 means, that only every 50th sampled parameter will be stored und used to compute characteristics of the posterior distribution as means, standard deviations or quantiles. The aim of thinning is to reach a considerable reduction of disk storing.
  DEFAULT: step = 50

Further options

- **family** = ***stringvalue***
  Defines the distribution of the response variable in the model. Families supported are gaussian, binomial and poisson. As link function, *BayesX* always uses the natural link, i.e. for gaussian distributed response the identity link, for binomial distributed response the logit link, and for poisson distributed response the log link.
  DEFAULT: family = binomial

Estimation output

The way the estimation output is presented depends on the estimated model. Estimation results of fixed effects are displayed in a tabular in the 'output window' and in a log-file (if created before). Shown will be the posterior mean, standard deviation, 10, 50 and 90 percent quantiles. Estimation effects of nonlinear effects of metrical and spatial covariates as well as (unstructured) random effects are presented in a different way. Since there are currently no capabilities for visualizing estimated effects, results are stored in an external ASCII-file

whose contents can be read into a general purpose statistics program (e.g. STATA, S-Plus) to further analyse and/or visualize the results. To ease the visualization of estimation results somehow, a couple of S-Plus functions are shipped together with *BayesX*, that allow plotting of nonlinear effects very easily. They are described in detail in Section 6.4 below. However, since the structure of the files containing estimation results is very simple, any other (statistics) software package with plotting facilities may be used instead. The structure of the files is as follows:

There will be one file for every nonparametric term in the model. To ease locating the files, the name of the files together with the storing directory are displayed in the 'output-window'. Each of the files contains six columns. The first column contains a parameter index (starting with one), the second column contains the values of the covariate whose effect is estimated. In the next four columns the estimated effects are printed, namely the posterior mean together with the 10, 50 (median) and 90 percent quantiles. As an example compare the following few lines, that are the beginning of a file containing the results of a covariate X1:

intnr	X1	X1mean	X1qu10	X1med	X1qu90
1	17	0.952843	0.78761	0.956815	1.11972
2	18	0.909783	0.816364	0.908112	1.00225
3	19	0.858095	0.805797	0.857704	0.90865
4	20	0.792924	0.757262	0.794104	0.828346
5	21	0.700773	0.668315	0.7008	0.735272
6	22	0.599102	0.565967	0.599408	0.632037
7	23	0.496422	0.461891	0.496439	0.529504
8	24	0.386172	0.353547	0.384964	0.421142

Note that the first row of the files always contains the names of the six columns.

Examples

We give here only a few examples about the usage of method *regress*. More detailed examples can be found in Section 6.6.

Suppose that we have a dataset 'test' with variables 'y', 'x1' 'x2' and 't', where 't' is assumed to be a time scale measured in months. Suppose further that we have already created a *bayesreg object* 'b'.

Fixed effects

We first specify a model with 'y' as the response variable and fixed effects for the covariates 'x1', 'x2' and 'x3'. Hence the predictor is

$$\eta = \beta_0 + \beta_1 x1 + \beta_2 x2 + \beta_3 x3$$

This model is estimated by typing:

```
> b.regress y = const + x1 + x2 + x3 , iterations=12000 burnin=2000 family=binomial step=10 using test
```

Additive models

Suppose now that we want to allow for possibly nonlinear effects of 'x2' and 'x3'. Defining a second order random walk as smoothness priors we obtain

```
> b.regress y = const + x1 + x2(rw2,4,8) + x3(rw2,4,10) , iterations=12000 burnin=2000
family=binomial step=10 using test
```

which corresponds to the predictor

$$\eta = \beta_0 + \beta_1 x1 + f_1(x2) + f_2(x3).$$

Here we choose 4 and 8 as the minimum and maximum blocksize for blockmove updates of 'x2', and 4 and 10 for covariate 'x3'. Note that the specification of the minimum and maximum blocksize is useless (although allowed) if the response is Gaussian, because in this case all full conditionals are Gaussian allowing to update the parameters in one large block. In this case we could simply type

```
> b.regress y = const + x1 + x2(rw2) + x3(rw2) , iterations=12000 burnin=2000
family=gaussian step=10 using test
```

time scales

In our next step we extend the model by incorporating an additional trend and a flexible seasonal component for the time scale t:

```
> b.regress y = const + x1 + x2(rw2,4,8) + x3(rw2,4,10) + t(rw2,5,12) + t(season,12,6,14)
, iterations=12000 burnin=2000 family=binomial step=10 using test
```

Note that we passed four arguments to specify the seasonal component. The second argument is the period of the seasonal effect, the third and the fourth argument are the minimum and maximum blocksize for blockmove updates.

spatial covariates

Suppose now that we have an additional spatial covariate 'region', which indicates the geographical region an observation belongs to. To incorporate a structured spatial random effect, we first have to create a *map object* and read in the boundary information of the different regions (polygones that form the regions, neighbors etc.).

```
> map m
> m.infile using c:\maps\map.bnd
```

More details about *map objects* can be found in Chapter 5. We can now extend our predictor

with a spatial random effect:

> b.regress y = const + x1 + x2(rw2,4,8) + x3(rw2,4,10) + t(rw2,5,12) + t(season,12,6,14) + region(spatial,m) , iterations=12000 burnin=2000 family=binomial step=10 using test

In some situations it may be reasonable to incorporate also an unstructured spatial random effect into the model in order to split the total spatial effect into a structured and an unstructured component. This can be done by typing

> b.regress y = const + x1 + x2(rw2,4,8) + x3(rw2,4,10) + t(rw2,5,12) + t(season,12,6,14) + region(spatial,m) + region(random) , iterations=12000 burnin=2000 family=binomial step=10 using test

Note again that the syntax simplifies for Gaussian response. In this case we could type:

> b.regress y = const + x1 + x2(rw2) + x3(rw2) + t(rw2) + t(season,12) + region(spatial,m) + region(random) , iterations=12000 burnin=2000 family=binomial step=10 using test

Possible Errors

- **ERROR: no valid (nonmissing) observations**
  The dataset used for estimation is empty.
- **ERROR: number of iterations must exceed number of burnin iterations by 100**
  To obtain reliable estimates, the sample size must be large enough. An error will be raised if the total number of samples is below 100, i.e. the number of burnin iterations must exceed the burnin (at least) by 100.
- **ERROR: thinning parameter too large**
  This error will be raised if the thinning parameter (option step) is larger than the number of iterations minus the burnin.
- **ERROR: " + *datasetname* + is not existing**
  The specified dataset for estimation is not existing.
- **ERROR: " + *datasetname* + " is not a dataset object**
  The specified dataset for estimation is an existing object but is not a *dataset object*.
- **ERROR: " + *mapobject* + is not existing**
  The specified *map object* for estimating spatial random effects is not existing.
- **ERROR: " + *mapobject* + " is not a map object**
  The specified map for estimating spatial random effects is an existing object but is not a *map object*.
- **ERROR: variable *varname* is not existing.**
  Variable *varname* specified in the model statement is not existing.

References

Besag, J., Kooperberg, C. (1995): *On conditional and intrinsic autoregressions.* Biometrika, **82**, 733-746.

Besag, J., York, J., Mollie, A. (1991): *Bayesian image restoration with two applications in spatial statistics (with discussion).* Ann. Inst. Statist. Math., **43**, 1-59.

Biller, C. (1999): *Bayesianische Ansätze zur nonparametrischen Regression.* PhD-thesis, University of Munich.

Clayton, D. (1996): *Generalized linear mixed models.* In: Gilks, W., Richardson S. and Spiegelhalter D. (eds), Markov Chain Monte Carlo in Practice. London: Chapman and Hall, 275-301.

Chib, S., Greenberg, E. (1995): *Understanding the Metropolis–Hastings Algorithm.* The American Statistician, **49**, 327-335.

Fahrmeir, L., Knorr-Held, L. (1997): *Dynamic discrete time duration models.* Sociological Methodology, **27**, 417-452.

Fahrmeir, L., Lang, S. (1999): *Bayesian Inference for Generalized Additive Mixed Models Based on Markov Random Field Priors.* Discussion Paper 169, Sonderforschungsbereich 386, Ludwigs-Maximilians-Universität München. Available under `http://www.stat.uni-muenchen.de/sfb386/publikation.html`.

Fahrmeir, L., Lang, S. (2000): *Bayesian Semiparametric Regression Analysis of Multicategorical Time-Space Data. Proceedings of the International Symposium on Frontiers of Time Series Modeling in Tokio.* Available under `http://www.stat.uni-muenchen.de/~lang/` .

Fahrmeir, L., Tutz, G. (1997): *Multivariate Statistical Modelling based on Generalized Linear Models.* New York: Springer–Verlag.

Gamerman, (1997): *Efficient Sampling from the posterior distribution in generalized linear models.* Statistics and Computing, **7**, 57-68.

Gamerman, D. (1998): *Markov Chain Monte Carlo for dynamic generalized linear models.* Biometrika **85**, 215-227.

Green, P.J., Silverman, B. (1994): *Nonparametric Regression and Generalized Linear Models.* Chapman and Hall, London.

Hastie, T., Tibshirani, R. (1990): *Generalized additive models.* Chapman and Hall, London.

Hastie, T., Tibshirani, R. (1993): *Varying-coefficient Models.* Journal of the Royal Statistical Society, **B 55**, 757-796.

Knorr-Held, L. (1996): *Hierarchical Modelling of Discrete Longitudinal Data.* PhD-thesis, University of Munich.

Knorr-Held, L. (1999): *Conditional Prior Proposals in Dynamic Models.* Scandinavian Journal of Statistics, , **26**, 129-144.

Lang, S. (1996): *Bayesianische Inferenz in Modellen mit variierenden Koeffizienten.* Diplomarbeit, Universität München

Rue, H. (2000): *Fast Sampling of Gaussian Markov Random Fields with Applications.* Technical report. Available under `http://www.math.ntnu.no/preprint/statistics/2000/s1-2000.ps`.

6.2 Method 'autocorr'

Description

This method is a post estimation method, that is it is only meaningful if method 'regress' has been applied before. By using method 'autocorr', the autocorrelation functions of all sampled (and stored) parameters will be computed. Since there are no graphics facilities in the current version of *BayesX*, the computed functions will be written to an external file and must be visualized with other programs. However, since the structure of the file is very simple, the visualization of the functions should be easy. Beyond it, a S-Plus function for automated plotting of the autocorrelation functions is available, see Section 6.4 below.

Syntax

objectname.autocorr [*, options*]

This command computes autocorrelation functions for all sampled and stored parameters. An error will be raised if regression results are not yet available. The computed functions will be stored in an external file. The storing directory will be the current output directory of the *bayesreg object*. By default, this directory is <INSTALLDIRECTORY>\output, but the current output directory may be changed by redefining the global option 'outfile', see Section 6.5. The filename will be the current output name extended by the ending '_autocorr.raw'. By default, the output name is the name of the particular *bayesreg object*, thus if for example your *bayesreg objects* name is 'bayes', the complete filename will be 'bayes_autocorr.raw'. Once again, the default output name may be changed using the globaloption 'outfile' (Section 6.5). Note that the autocorrelation file will be overwritten whenever method 'autocorr' is applied. A remedy for that problem is to change the current output directory and/or output name *before* every new estimation, using the global option 'outfile', see Section 6.5. The structure of the file with the stored autocorrelation functions is the following:

The computed functions are stored in a matrix like fashion. For every parameter the autocorrelation function will be stored columnwise, with autocorrelation for lag 1 in row 1, for lag 2 in row 2 and so on. The first column of the file contains the lag number. In addition, for each term in the estimated model minimum, mean and maximum autocorrelations will be computed and stored. Note finally that the very first row of the file contains the column names.

The stored autocorrelation functions are visualized most simply using the S-plus function 'plotautocor', see Section 6.4 for details.

Option

- **maxlag =** ***intvalue***
  With the 'maxlag' option, the maximum lag number for computing autocorrelations may be specified. 'intvalue' must be a positive integer valued number.
  DEFAULT: maxlag = 250

Examples

Suppose we defined a bayesreg object 'b' and estimated a (simple) regression model using the following 'regress' statement

```
> b.regress Y = const + X using d
```

where 'd' is the analysed dataset. The model contains only a fixed effect for covariate X and a constant. We may now want to check the mixing of the sampled parameters (one for the constant and one for X) by computing autocorrelation functions. The following statement computes autocorrelations up to lag 100 and stores the result in the default output directory with filename 'b_autocorr.raw':

```
> b.autocorr , maxlag=100
```

The default output directory is '<INSTALLDIRECTORY>\output'. So if for example *BayesX* is installed in 'c:\bayes', the autocorrelation functions will the stored in 'c:\bayes\output\b_autocorr.raw'. If you wish to store the file in another directory, say 'c:\data', and under another name, for example 'estimate1', you must use the global option 'outfile' before estimation (see also Section 6.5). The following commands produce the desired result (program output between the different statements omitted):

```
> b.outfile = c:\data\estimate1
> b.regress Y = const + X using d
> b.autocorr , maxlag=100
```

Now the autocorrelation functions will be stored under 'c:\data\estimate1_autocorr.raw'. The beginning of the file looks like this:

```
lag FixedEffects_1 FixedEffects_2 FixedEffects_min FixedEffects_mean FixedEffects_max
1   -0.0250767   0.017502    -0.0250767   -0.00378733  0.017502
2   -0.0582127   0.0141592   -0.0582127   -0.0220268   0.0141592
3   -0.00734053  0.0176357   -0.00734053  0.00514759   0.0176357
4   0.0209137    0.0186829   0.0186829    0.0197983    0.0209137
```

The computed autocorrelation functions may now be visualized easily using the S-Plus function 'plotautocor'. The function plots the autocorrelation functions for all estimated parameters (in our example only two) against the lag number. If the option 'mean.autocor=T' is specified, only minimum, mean and maximum autocorrelations for each term in the model are plotted against the lag number. Obviously, this is much faster than the first alternative, where the autocorrelation functions of all parameters are drawn.
The following statement in S-Plus plots and stores the autocorrelations in the postscript-file 'c:\data\estimate1_autocorr.ps':

plotautocor1("c:\\data\\estimate1_autocorr.raw", "c:\\data\\estimate1_autocorr.ps")

Note that double backslashes are required in S-Plus to specify the directory of a file correctly. For more details about the function 'plotautocor' compare Section 6.4.

Possible Errors

- **ERROR: no regression results**
  You tried to compute autocorrelation functions although no estimation results are present. Use method 'regress' to get some estimation results.
- **ERROR: integer value expected**
  You used the 'maxlag' option to specify the maximum lag number, but specified a number that is not integer valued.
- **ERROR: value between 1 and 500 expected**
  You used the 'maxlag' option to specify the maximum lag number. Only values between 1 and 500 are allowed as maximum lag numbers.

6.3 Method 'getsample'

Description

This method is a post estimation method, that is it is only meaningful if method 'regress' has been applied before. With method 'getsample' all sampled parameters will be stored in (one or more) ASCII file(s). Afterwards, sampling paths can be plotted and stored in a postscript file using the S-Plus function 'plotsample' or with other programs with graphics capacities. See also Section 6.4 for the S-Plus routine 'plotsample'.

Syntax

objectname.getsample

This command stores all sampled parameters in ASCII file(s). An error will be raised, if regression results are not yet available. The storing directory will be the current output directory of the *bayesreg object*. By default, this directory is <INSTALLDIRECTORY>\output,

but you can change the current output directory by redefining the global option 'outfile', see Section 6.5. The filenames will be the current output name extended by an ending depending on the type of the estimated effect. For example for fixed effects, the complete filename will be 'b_FixedEffects_sample.raw', if 'b' is the name of the bayesreg object. The total number of created files and their filenames are printed in the 'output window'. By default, the output name is the name of the *bayesreg object*. Once again, the default name may be changed using the global option 'outfile' (Section 6.5). Note that it can happen that some or all files will be overwritten, if method 'getsample' is applied more than once with the same *bayesreg object*. To avoid such problems change the current output directory and/or output name *before* every new estimation using the global option 'outfile', see Section 6.5.

The structure of the created files is as follows: The very first row contains the parameter names. In the following lines, the parameters are stored in a matrix like fashion. In the first row (to be precise the second row, since the first contains the names) the first sampled value of each parameter seperated by blanks is stored . In the second row the second value is stored and so on. In the first column of each row the sampling number is printed.

The stored sampled parameters are visualized most simply using the S-Plus function 'plotsample', see Section 6.4. Of course, other programs for visualization may also be used.

Examples

Suppose we already have defined a bayesreg object 'b' and estimated a (simple) regression model using the following 'regress' statement:

```
> b.regress Y = const + X using d
```

The model contains only a fixed effect for covariate X and a constant. We may now want to check the mixing of the sampled parameters (one for the constant and one for X) by storing sampled parameters in an ASCII-file and visualizing sampling paths. The (simple) statement

```
> b.getsample
```

forces *BayesX* to store the sampled parameters in a file named 'b_FixedEffects_sample.raw'. The storing directory is the current output directory, which is by default '<INSTALLDIRECTORY>\output'. The current output directory can be changed using the global option 'outfile', see Section 6.5. For example the two commands

```
> b.outfile = c:\data\estimate1
> b.getsample
```

have the effect that the sampled parameters will now be stored in file 'c:\data\estimate1_FixedEffects_sample.raw'.

The first few lines of the file look like this:

```
intnr   b_1   b_2
```

```
1   -2.00499   2.02167
2   -1.97745   2.01813
3   -1.98498   1.98328
4   -1.97108   1.98054
5   -2.00004   1.98099

        ⋮
```

6.4 S-Plus functions for visualizing estimation results

Since *BayesX* provides no capabilities for visualizing estimation results, some S-Plus functions for plotting estimated functions are shipped together with *BayesX*. These functions can be found in the subdirectory 'sfunctions' of the installation directory. Table 6.1 gives a first overview over the different functions and their abilities. The usage of the functions is very simple so that also users not familiar with the S-Plus environment should be able to apply the functions without any problems. The following subsections describe how to install the functions in S-Plus and give a detailed descripiton of the usage of the respective functions.

Functionname	**Description**
plotnonp	visualizes estimated nonparametric functions
plotautocor	visualizes autocorrelation functions
plotsample	visualizes sampling pathes of sampled parameters
readbndfile	reads in boundaries of geographical maps
drawmap	visualizes estimation results for spatial covariates

Table 6.1: Overview over S-Plus functions

6.4.1 Installation of the functions

Installation of the different functions is very easy. The S-Plus code for the functions is stored in the directory <INSTALLDIRECTORY>\sfunctions in the ASCII text file 'plot.txt'. To install the functions you first have to start S-Plus. Afterwards the functions will be installed by entering

source("<INSTALLDIRECTORY>\\sfunctions\\plot.txt")

in the 'Commands window' of S-Plus. Note that a double backslash is required in S-Plus to specify a directory correctly.

6.4.2 Plotting nonparametric functions

This subsection describes the usage of the function 'plotnonp' for visualizing nonparametric function estimates.

Suppose that a Bayesian regression model has already been estimated with predictor

$$\eta = \ldots + f(X) + \ldots,$$

where the effect of X is modelled nonparametrically using for example a first or second order random walk prior. Unless the directory for estimation output has been changed using the global option 'outfile' (see Section 6.5), estimation results for the nonparametric effect of X are stored in the directory

<INSTALLDIRECTORY>\output

that is in the subdirectory 'output' of the installation directory. The filename is

*objectname*_nonpX.res

that is it is composed of the name of the bayesreg object and the covariate name. For the following we assume that 'c:\bayes' is the installation directory and 'b' is the name of the bayesreg object. In this case results for the effect of X are stored in:

c:\bayes\output\b_nonpX.res

The structure of the file has already been described in Section 6.1. Although it is possible (and very easy) to visualize the estimated nonparametric function with any software package that has plotting capabilities, a fast and easy way of plotting estimation results without knowing the particular structure of the results-file is desirable. This is the task of the S-plus function 'plotnonp'.

The function has only one required and many optional arguments. The required argument is the directory and the filename where nonparametric estimation results are stored. For example by entering the command

plotnonp("c:\\bayes\\output\\b_nonpX.res")

a S-plus graphic-window will be opened with the plotted function estimate. The function always plots the posterior mean together with the posterior 10 and 90 percent quantiles. One advantage of the function is, that after its application no permanent objects will remain in the S-plus environment.

Besides the required argument a lot of optional arguments may be passed to the function. Among others there are options for plotting the graphs in a postscript file rather than the screen, labeling the axes, specifying the minimum/maximum value on the x/y axis and so on. The following lists all optional arguments, that can be passed to 'plotnonp':

- psname = "file (including path)"
  Name of the postscript output file. If 'psname' is specified the graph will be stored in a postscript file and will not appear on the screen.
- ylimtop=numerical value
  Specifies the maximum value on the y-axis (vertical axis)

- ylimbottom = numerical value
  Specifies the minimum value on the y-axis
- xlab = "character string"
  'xlab' is used to label the x-axis (horizontal axis)
- ylab ="character string"
  'ylab' is used to label the y-axis.
- maintitle = "character string"
  Adds a title to the graph
- subtitle = "character string"
  Adds a subtitle to the graph.

As an illustration compare the following S-Plus statement:

plotnonp("c:\\bayes\\b_nonpX.res", psname="c:\\bayes\\b_nonpX.ps",
maintitle="Maintitle",ylab="effect of X",xlab="X")

This statement draws the estimated effect of X and stores the graph in the postscript file "c:\\bayes\\b_nonpX.ps". A title, a x-axis and y-axis label is added to the graph. For illustration purposes, the resulting graph is shown in figure 6.1.

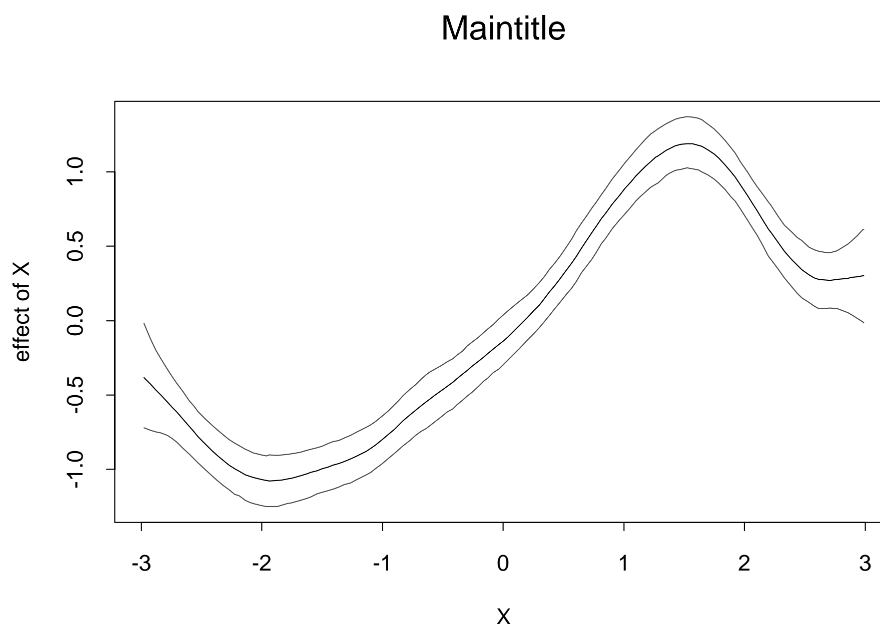


Figure 6.1: Illustration for the usage of 'plotnonp'

In some situations the effect of a covariate representing dates must be plotted. Suppose for example that a covariate has values ranging from 1 to 192 representing the time period from January 1980 to December 1995. In this case, we naturally prefer that the x-axis is

labeled in terms of dates rather than in the origional coding (form 1 to 192). To achieve this, function 'plotnonp' provides the three additional options 'year', 'month' and 'step'. Options 'year' and 'month' are used to specify the year and the month (1 for January,2 for February,...) corresponding to the minimum covariate value. In the example mentioned above year=1980 and month=1 will produce the correct result. In addition, option 'step' may be specified to define the periodicity in which your data are collected. For example step=12 (the default) corresponds to monthly data, while step = 4, step = 2 and step = 1 correspond to quarterly, half yearly and yearly data. We illustrate the usage of 'year', 'month' and 'step' with our example. Suppose we estimated the effect of calendar time D, say, on a certain dependent variable, where the range of the data is as described above. Then the following S-Plus function call will produce the postscript file shown in figure 6.2:

plotnonp("c:\\bayes\\b_nonpD.res", psname="c:\\bayes\\b_nonpD.ps",
year=1980,month=1,step=12,xlab="date", ylab= " ")

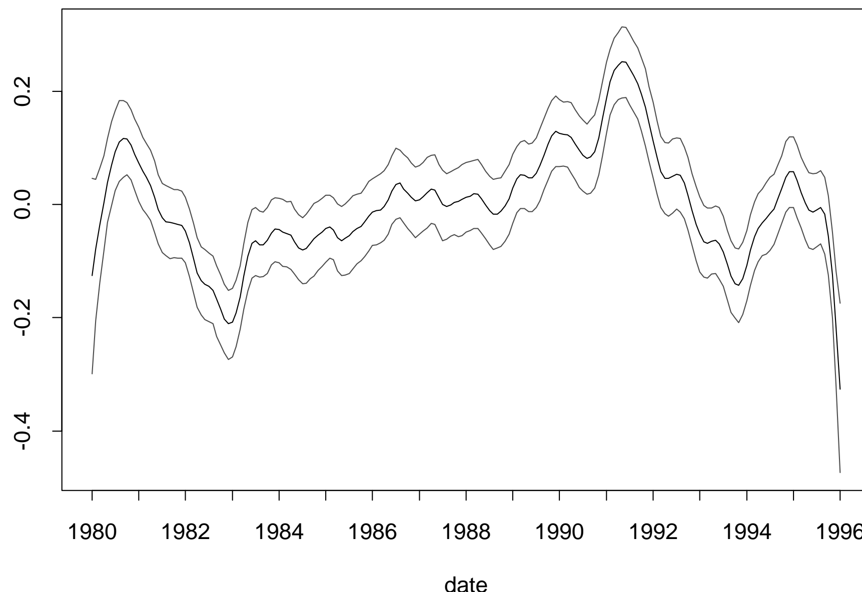


Figure 6.2: Illustration for the usage of 'plotnonp'

Note, that ylab=" " forces S-Plus to omit the y axis label. If ylab (as well as xlab) is omitted, default labels will be given to the two axis.

Finally, we note that all options that can be passed to the 'plot' function of S-Plus may also be passed to function 'plotnonp'. Thus, function 'plotnonp' is more or less a specialized version of the S-Plus 'plot' function.

6.4.3 Drawing geographical maps

This subsection describes how to visualize estimation results of spatial covariates, where the observations represent the location or site in connected geographical regions. A typical example for a spatial covariate is given in the 'rents for flats' example, see Section 2.4.1, where the covariate 'L' indicates the location (in subquarters) of the flat in Munich. Figure

Figure 6.3: Map of Munich

6.3 shows a map of Munich separated into subquarters.

Typically, the effect of such a spatial covariate is incorporated into a regression model via an unstructured or structured random effect. In the latter case a spatial smoothness prior for the spatial covariate is specified that penalizes too abrupt changes of the estimated effect in neighboring sites. In some situations the incorporation of both, an unstructured and a structured effect, may also be appropriate. Details on how to incorporate spatial covariates into a semiparametric regression model are given in Section 6.1. For the rest of this section we assume that an effect of a spatial covariate has already been estimated and that we now want to visualize the estimation results. This can be easily done with the two S-Plus functions 'readbndfile' and 'drawmap'. Function 'readbndfile' is used to read the boundary information of a map that is stored in a so called boundary-file and to store this information as a permanent S-Plus map object. The boundary file contains mainly the polygones which form the different geographical regions of the map. The required structure of such a file is described below. After the successfull reading of the boundary information of a map, the second function 'drawmap' may be used to draw and print the map either on the screen or into a postscript file. There are several possible ways to draw the map. In the simplest case the map can be drawn without any estimation effects, i.e. only the boundaries of the different regions or sites are drawn, see Figure 6.3 for an example. In practice, however, one usually wants to color the regions of the map according to some numerical characteristics. As an example compare Figure 6.4 in which the subquarters of Munich are colored according to the frequency of flats in the rent dataset located in the respective subquarter. Subquarters colored in red contain less flats compared to subquarters colored in green. In striped areas no observations are available.

In the following subsections we give a detailed description of the usage of the functions 'readbndfile' and 'drawmap'. The structure of boundary files required to apply the functions 'readbndfile' and 'drawmap' is described in detail in Chapter 5 about map objects.

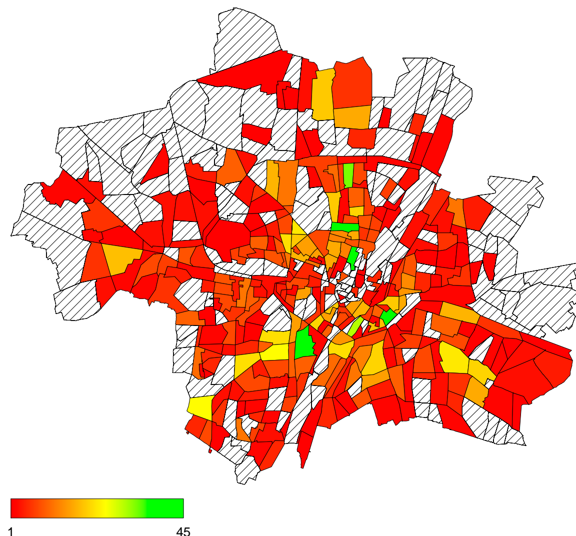


Figure 6.4: relative frequencies of observed flats in the "rent dataset"

Function 'readbndfile'

Function 'readbndfile' is used to read in boundary information into S-Plus that is stored in a boundary file. The function has two required arguments. The first argument is the filename of the boundary file to read in. The second argument specifies the name of the map object in S-Plus (recall that the map information is stored as a permanent S-Plus object). To give an example, suppose that *BayesX* is installed in the directory 'c:\bayes' and that we want to read in the map of Munich. In this case the boundary file of the map is stored in the subdirectory 'examples' of the installation directory, that is in 'c:\bayes\examples'. The name of the boundary file is simply 'munich.bnd'. The following function call reads in the boundary information of Munich and stores the map permanently in S-Plus:

```
readbndfile("c:\\bayes\\examples\\munich.bnd","munich")
```

Once again note that double backslashes are required in S-Plus to specify a directory. The second argument in the statement above is "munich", i.e. the name of the map object is simply munich. To refer to the map of munich in subsequent statements and function calls, the quotation marks must be omitted.

Function 'drawmap'

Function 'drawmap' is used to draw geographical maps and color the regions according to some numerical characteristics. There is only one required argument that must be passed to 'drawmap', that is the name of the map to be drawn. Provided that the map has already been read into S-Plus (via function 'readbndfile'), the following statement draws the map of Munich in a S-Plus graphic-window on the screen:

```
drawmap(map=munich)
```

Storing the map in a postscript file rather than drawing it on the screen can be achieved by specifying the name of the postscript file using the 'outfile' option. For example the command

drawmap(map=munich,outfile="c:\\bayes\\munich.ps")

produces a postscript file named 'munich.ps' with the map of Munich.

However, in most cases one not only wants to draw the boundaries of a geographical map, but also color the regions according to some numerical characteristics. Suppose for example that we have already estimated a location specific effect on the monthly rents in the 'rents for flats' dataset. Suppose further that the estimated effects are stored in 'c:\bayes\output\b_spatialL.res'. The structure of the file is described in detail in Section 6.1. It contains 6 columns, one column with a row counter, one column containing the values of the covariate (here the location) and four columns containing estimated effects for the values of the covariate. These are the posterior mean and the posterior 10, 50 and 90 percent quantiles. The first row contains the column names. For the covariate 'L' (location) for example, the first row of the file is given by:

intnr L Lmean Lqu10 Lmed Lqu90

Suppose now that we want to visualize estimation results for the spatial covariate location by coloring the subquarters of Munich according to the estimated posterior mean. Compared to the S-Plus statement above (at least) three more arguments must be passed to function 'drawmap'; the argument 'dfile' that specifies the filename of estimated results, the argument 'plotvar' that specifies the variable to be plotted and the argument 'regionvar' that specifies which column of the file, containing estimation results, stores the region names. The following statement produces the desired result:

drawmap(map=munich,outfile="c:\\bayes\\munich.ps",
dfile="c:\\bayes\\output\\b_spatialL.res", plotvar=Lmean,regionvar=L)

Besides the arguments discussed so far there are some more optional arguments that can be passed to 'drawmap'. They are listed and described below together with a summary of the arguments already described:

- **map=Name of the map**
  Name of the S-Plus map object. Use function 'readbndfile' to read in geographical maps into S-Plus.

- **dfile="filename (including path)"**
  Name (including path) of the file containing numerical characteristics of the regions of the map. The file must contain at least two columns, one column that lists the names of the regions and one column containing the numerical characteristics of the respective regions. It is important that the names of the regions listed match with the region names stored in the S-Plus map object. The first row of the file must contain the names of the columns.

- **outfile="filename (including path)"**
  Name (including path) of the postscript file where the map should be stored.

- **regionvar=character string**
  Name of the column in the data file containing the region names (see also argument 'dfile').

- **plotvar = character string**
  Name of the column in the data file containing the numerical characteristics of the regions (see also argument 'dfile').

- **lowerlimit=number**
  Lower limit of the range to be drawn. If 'lowerlimit' is omitted, the minimum numerical value in the 'plotvar' column will be used instead as the lower limit.

- **upperlimit=number**
  Upper limit of the range to be drawn. If 'upperlimit' is omitted, the maximum numerical value in the 'plotvar' column will be used instead as the upper limit.

- **shades=number**
  To color the regions according to their numerical characteristics, the data are devided into a (typically large) number of ordered categories. Afterwards a color is associated with each category. The 'shades' option can be used to specify the number of categories (and with it the number of different colors). The maximum number of colors is 1000, which is also the default value.

- **pstitle=character string**
  Adds a title to the graph.

- **color =T/F**
  The 'color' option allows to choose between a grey scale for the colors and a colored scale. The default is 'color=F', which means a grey scale.

- **legend=T/F**
  By default a legend is drawn into the graph. To omit the legend in the graph, 'legend=F' must be passed as an additional argument.

- **drawnames=T/F**
  In some situations it may be favorable to print the names of the regions into the graph (although the result may be confusing in most cases). This can be done by specifying the additional option 'drawnames=T'. By default the names of the regions are omitted in the graph.

- **swapcolors=T/F**
  In some situations is may be favorable to swap the order of the colors, i.e. red shades corresponding to large values and green shades corresponding to small values. This is achieved by specifying 'swapcolors=T'. By default small values are colored in red shades and large values in green shades.

6.4.4 Plotting autocorrelation functions

This section describes how to visualize autocorrelation functions of sampled parameters using the S-Plus function 'plotautocor'.

To compute autocorrelation functions, the post-estimation command 'autocorr' must be applied, see Section 6.2 for details. For the rest of this section we assume that autocorrelations are already computed and stored in file:

c:\bayes\output\b_autocorr.raw

The minimum number of arguments required for the function is one, namely the file where the computed autocorrelation functions are stored. In this case a S-Plus graphic window will be opened and the autocorrelation functions are plotted on the screen. To store autocorrelations in a postscript file, an ouput filename must be specified as a second argument. Thus, the S-Plus command

```
plotautocor("c:\\bayes\\output\\b_autocorr.raw")
```

prints autocorelations on the screen, while the statement

```
plotautocor("c:\\bayes\\output\\b_autocorr.raw", "c:\\bayes\\output\\b_autocorr.ps")
```

forces S-Plus to store the autocorrelation graphs in the postscript file 'c:\bayes\output\b_autocorr.ps'.

In particular for regression models with a large number of parameters the execution of function 'plotautocor' can be very time consuming. Moreover, the size of the resulting postscript file can be very large. To avoid such problems 'plotautocor' provides the additional argument 'mean.autocor'. If 'mean.autocor = T' is specified for each lag number and model term only minimum, mean and maximum autocorrelations are plotted, leading in most cases to a considerable reduction in computing time and storing size.

6.4.5 Plotting sampled parameters

This section describes how to plot sampled parameters using the S-Plus function 'plotsample'. Before applying function 'plotsample', sampled parameters must be stored in ASCII-format using the post-estimation command 'getsample'. See Section 6.3 for details, but note that sampled parameters will be stored in several different files, typically one file for each term in the model.

Suppose now that we want to visualize sampling paths for the parameters of the nonlinear effect of a covariate X. Assume further that sampled parameters are stored in the ASCII file

c:\bayes\output\b_X_samples.raw.

As most other functions, 'plotsample' provides two possibilities of drawing sampled parameters. The first possibility is to print the graphs on the screen, and the second is to store them into a postscript file. To print the sampling paths on the screen, only the file name (including path) of the ASCII file where sampled parameters are stored must be passed to the function. For the example mentioned above the corresponding command is:

plotsample(”c:\\bayes\\output\\b_X_samples.raw”)

If sampling paths should be drawn into a postscript file rather than on the screen, the filename of the resulting postscript file must be specified as a second argument. Thus, for our example we get:

plotsample(”c:\\bayes\\output\\b_X_samples.raw”, ”c:\\bayes\\output\\b_X_samples.ps”)

In addition, all options that are available for the S-Plus function ’plot’ may be passed to function ’plotsample’, see the S-Plus documentation for details.

6.5 Global options

The purpose of global options is to affect the global behavior of a *bayesreg object*. The main characteristic of global options is, that they are not associated with a certain method.
The syntax for specifying global options is

objectname.optionname = *newvalue*

where *newvalue* is the new value of the option. The type of the value depends on the respective option.

The following global options are currently available for *bayesreg objects*:

- **outfile = *filename***
  By default, the estimation output produced by the *bayesreg* ’regress’ procedure will be written to the default output directory, which is

  <INSTALLDIRECTORY>\output.

  The default filename is composed of the name of the *bayesreg object* and the type of the file. For example, if you estimated a nonparametric effect for a covariate X, say, then the estimation output will be written to

  <INSTALLDIRECTORY>\output\b_nonpX.res

  where ’b’ is the name of the ’bayesreg’ object. In most cases, however, it may be necessary to save estimation results into a different directory and/or under a different filename than the default. This can be done using the ’outfile’ option. With the outfile option you have to specify the directory where the output should be stored to and in addition a base filename. The base filename should not be a complete filename. For example specifying

  outfile = c:\data\res1

  would force *BayesX* to store the estimation result for the nonparametric effect of X in file

  c:\data\res1_nonpX.res

- **iterationsprint** = ***intnumber***
  By default, the current iteration number is printed in the output window (or in an additional log-file) after every 100 th iteration. This can lead to rather big and complex output files. The 'iterationsprint' option allows to redefine after how many iterations the current iteration number is printed. For example 'iterationsprint = 1000' forces *BayesX* to print the current iterations number only after every 1000 th iteration rather than after every 100 th iteration.

6.6 Examples

In this Section we present two complex examples about the usage of *bayesreg objects* to estimate linear and nonlinear effects within a Bayesian framework. The first example contains a reanalysis of the 'credit scoring dataset' that is described in Chapter 2.4.2, which contains also a (incomplete) list of some publications where the credit scoring dataset has already been analysed. The analysis here is based mainly on Fahrmeir, Lang (1999). The second example is a Bayesian analysis of the 'rents dataset' described in Chapter 2.4.1. Both datasets are shipped together with *BayesX* and are stored in the directory 'examples', which is a subdirectory of the installation directory. Since the main focus here is on illustrating the usage of *bayesreg objects*, we omit any interpretation of estimated effects.

6.6.1 Credit scoring

All *BayesX* statements of this section can be found in the 'examples' directory in the file 'credit.prg'. In principle, the commands in 'credit.prg' can be executed using the 'usefile' command for running batch files, see Section 3.5. But note that the specified directories therein may not exist at your computer. Thus, to avoid errors, the file must be modified first to execute correctly.

In order to analyse the 'credit scoring dataset', we first have to load the dataset into *BayesX*. For the rest of this section we assume that *BayesX* is installed in the directory 'c:\bayes'. In this case, the credit scoring dataset can be found in 'c:\bayes\examples' under the name 'credit.raw'. With the following two commands (entered in the command window) we first create a *dataset object* 'credit' and afterwards we load the dataset into *BayesX* using the 'infile' command:

```
> dataset credit
> credit.infile using c:\bayes\examples\credit.res
```

Since the first row of the file already contains the variable names, it is not necessary to specify variable names in the 'infile' statement. We compute now effect coded versions of the categorical covariates 'account', 'payment', 'intuse' and 'marstat':

```
> credit.generate account1 = 1*(account=1)-1*(account=3)
> credit.generate account2 = 1*(account=2)-1*(account=3)
> credit.generate payment1 = 1*(payment=1)-1*(payment=2)
> credit.generate intuse1 = 1*(intuse=1)-1*(intuse=2)
```

> credit.generate marstat1 = 1*(marstat=1)-1*(marstat=2)

The reference categories for the covariates are chosen to be 3 for 'account' and 2 for the others. Before we are able to estimate Bayesian regression models, we first have to create a *bayesreg object*:

> bayesreg b
> b.outfile = c:\results\credit

The second command changes the default output directory and name (which is 'c:\bayes\output\b') to c:\results\credit1. We can now start estimating models. We first estimate a logistic regression where all covariates enter the model as fixed effects.

> b.regress y = const + account1 + account2 + duration + amount + payment1 + intuse1 + marstat1 , iterations=5000 burnin=2000 step=1 family=binomial using credit

Here we specified 5000 iterations, 2000 burnin iterations and one as the thinning parameter, i.e. every sampled parameter will be stored and used for estimation. Note that there is an alternative way of specifying our model. Alternativeley we could type:

> b.regress y = const + account(cat,3) + duration + amount + payment(cat,2)
+ intuse(cat,2) + marstat(cat,2) , iterations=5000 burnin=2000 step=1
family=binomial using credit

The difference here compared to the first specification is that the covariates 'account', 'payment', 'intuse' and 'marstat' are treated as categorical covariates, thus effect coding is automatically incorporated. However, since the effect of every categorical covariate is updated without considering the other effects, mixing of sampled parameters is in general slower compared to the first alternative, where all parameters are updated in one block.
Executing the command yields the following output (simulation output omitted):

SIMULATION TERMINATED

SIMULATION RUN TIME: 1 minute 38 seconds

SIMULATION RESULTS:

FixedEffects1

Acceptance rate: 26.06 %

Variable	mean	Std. Dev.	10% quant.	median	90% quant.
const	-1.24311	0.183257	-1.48361	-1.2446	-1.01388
account1	-1.08499	0.120584	-1.24854	-1.07775	-0.934988
account2	0.859005	0.104591	0.729654	0.850584	0.993064
duration	0.0345912	0.00766953	0.0242133	0.0342926	0.0441488
amount	0.0300396	0.0310772	-0.0109083	0.0311249	0.0690022
payment1	-0.456014	0.124162	-0.608557	-0.462533	-0.292689

FixedEffects2

Acceptance rate: 54.98 %

Variable	mean	Std. Dev.	10% quant.	median	90% quant.
intuse1	-0.233724	0.084493	-0.341506	-0.235033	-0.124342
marstat1	-0.269354	0.0785152	-0.373707	-0.267377	-0.170913

Somewhat surprisingly, we observe that the amount of credit seems to have no ("significant") influence on the response. To check this phenomenon more carefully, we run a second estimation, now allowing for possibly nonlinear effects of the metrical covariates 'amount' and 'duration'. We choose second order random walks as smoothness priors and modify the 'regress' statement above according to the new model:

```
> b.regress y = const + account1 + account2 + duration(rw2,4,9) + amount(rw2,15,20)
+ payment1 + intuse1 + marstat1 , iterations=5000 burnin=2000 step=1
family=binomial using credit
```

Blocksizes are chosen to be between 4 and 9 for covariate 'duration' and between 15 and 20 for covariate 'amount'. We get the following output (output for fixed effects omitted):

duration_rw2

Acceptance rate: 61.08 %

Results are stored in file c:\results\credit_nonpduration.res
Results may be visualized using the S-Plus functions
'plotnonp' and 'drawmap' for spatial data

Estimated variance parameter (sample mean): 0.00315944

amount_rw2

Acceptance rate: 58.4 %

Results are stored in file c:\results\credit_nonpamount.res
Results may be visualized using the S-Plus functions

’plotnonp’ and ’drawmap’ for spatial data

Estimated variance parameter (sample mean): 0.0040042

We visualize estimated effects for ’amount’ and ’duration’ using the S-Plus function ’plotnonp’. We first have to install the function by entering in S-Plus the command:

source(”c:\\bayes\\sfunctions\\plot.txt”)

This installs not only function ’plotnonp’, but also all the other S-Plus functions shipped with *BayesX*, see Section 6.4. We can now apply function ’plotnonp’ by typing (in S-Plus)

plotnonp(”c:\\results\\credit_nonpduration.res”, ”c:\\results\\credit_nonpduration.ps”)

and

plotnonp(”c:\\results\\credit_nonpamount.res”, ”c:\\results\\credit_nonpamount.ps”)

This produces the graphs (stored in postscript files) shown in Figure 6.5.
To obtain more sophisticated graphs, for example by labeling the axis, some more options must be passed to ’plotnonp’, see Section 6.4.2 for details.
We now want to check the mixing of the generated Markov chains. For that reason we compute and plot sampled parameters as well as autocorrelation functions. We first store sampled parameters and compute autocorrelations by typing in *BayesX*:

```
> b.getsample
> b.autocorr
```

This yields the following program output (in the Output window):

Storing sampled parameters...
Sampled parameters will be stored in file(s):

c:\results\credit_FixedEffects1_sample.raw
c:\results\credit_duration_rw2_sample.raw
c:\results\credit_amount_rw2_sample.raw

Storing completed

Sampled parameters may be visualized using the S-plus
function ’plotsample’.

Computing autocorrelation functions...
Autocorrelation functions computed and stored in file

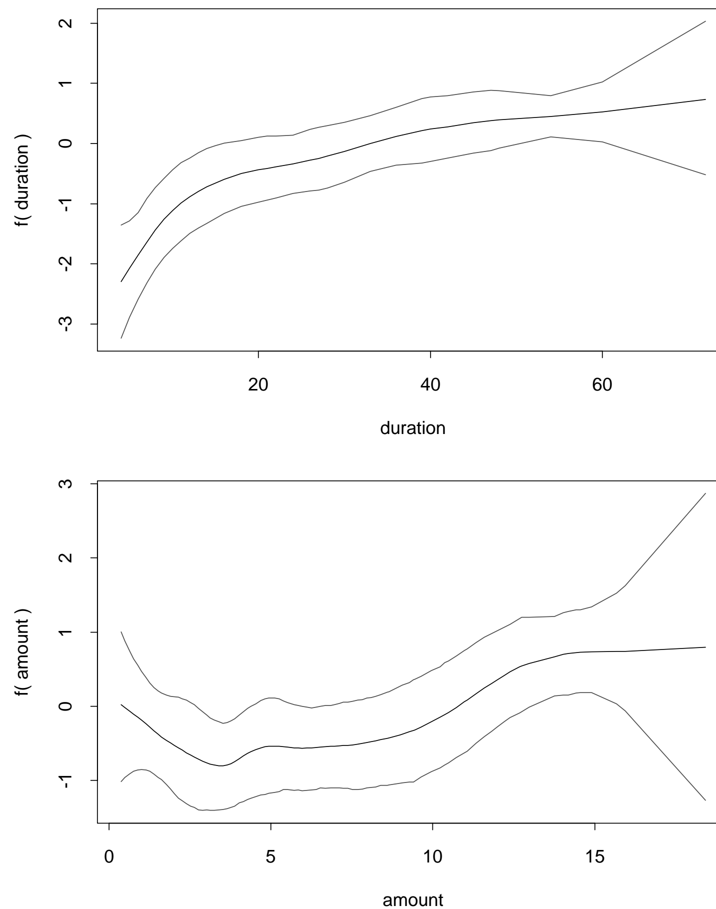


Figure 6.5: Estimated effects of duration and amount of credit. Shown is the posterior mean within 80 % credible regions.

c:\results\credit_autocorr.raw.

They may be visualized using the S-Plus function 'plotautocor'.

Plotting sampling paths and autocorrelations is done in S-Plus by typing

plotsample("c:\\results\\credit_FixedEffects1_sample.raw",
"c:\\results\\credit_FixedEffects1_sample.ps")
plotsample("c:\\results\\credit_duration_rw2_sample.raw",
"c:\\results\\credit_duration_rw2_sample.ps")
plotsample("c:\\results\\credit_amount_rw2_sample.raw",
"c:\\results\\credit_amount_rw2_sample.ps")

and

plotautocor("c:\\results\\credit_autocorr.raw",
"c:\\results\\credit_autocorr.ps",mean.autocor=T)

Note that we only printed minimum, mean and maximum autocorrelations by specifying 'mean.autocor=T' as an additional option. Plotting *all* autocorrelation functions is achieved by simply omitting 'mean.autocor=T' in the function call above. Because of the large number of graphs produced by the four function calls, we omit printing the resulting graphs here. But all postscript files containing the graphs can be found in the 'examples' directory. A look at the plots shows that all autocorrelations disappear until lag 50. Thus we can run a final simulation with 52000 iterations and a burnin of 2000 iterations:

```
> b.outfile = c:\results\creditfinal
> b.regress y = const + account1 + account2 + duration(rw2,4,9) + amount(rw2,15,20)
+ payment1 + intuse1 + marstat1 , iterations=52000 burnin=2000 step=50
family=binomial using credit
```

The thinning parameter is set to 50, i.e. every 50th-parameter is stored and used for estimation. Since the obtained results differ only slightly from the findings above, we omit printing our final estimation results.

6.6.2 Rents for flats

All *BayesX* statements of this section can be found in the 'examples' directory in the file 'rent.prg'. In principle, the commands in 'rent.prg' can be executed using the 'usefile' command for running batch files, see Section 3.5. But note that the specified directories therein may not exist at your computer. Thus, to avoid errors, the file must be modified first to execute correctly.

We first load the dataset into *BayesX*. For the rest of this section we assume that *BayesX* is installed in the directory 'c:\bayes'. In this case the rent dataset can be found in 'c:\bayes\examples' under the name 'rent94.raw'. The following commands create a dataset 'rent' and load the data into *BayesX*:

```
> dataset rent
> rent.infile using c:\bayes\examples\rent94.raw
```

Since the first row of the file already contains the variable names, it is not necessary to specify variable names in the 'infile' statement.

To be able to estimate Bayesian nonparametric regression models we create a *bayesreg object* 'b' and specify a output directory and filename:

```
> bayesreg b
> b.outfile = c:\results\rent
```

We first estimate a model with possibly nonlinear effects of floor space 'F' and year of construction 'A' on the monthly rent per square meters. To allow for spatial heterogenity we incorporate an additional unstructured random effect for location 'L'. In principle, a structured spatial random effect is also possible. However, a look at the distribution of

the flats over the subquarters in Munich (see Figure 6.4) shows that we have too many subquarters where no flats are observed. The described model can be estimated by typing

> b.regress R = const + F(rw2) + A(rw2) + L(random) , iterations=22000 burnin=2000 step=20 family=gaussian using rent

Executing the command yields the following output (simulation output omitted):

SIMULATION RESULTS:

Estimation results for the scale parameter:

Mean:	19.051
Std. Dev.:	0.963657
10% Quantile:	18.036
50% Quantile:	18.9007
90% Quantile:	20.2507

FixedEffects1

Acceptance rate: 100 %

Variable	mean	Std. Dev.	10% quant.	median	90% quant.
const	11.2007	0.584888	10.411	11.2376	11.941

F

Acceptance rate: 100 %

Results are stored in file c:\results\rent_nonpF.res
Results may be visualized using the S-Plus functions
'plotnonp1' and 'plotnonp2 or 'drawmap' for spatial data

Estimated variance parameter (sample mean): 0.00322753

A

Acceptance rate: 100 %

Results are stored in file d:\results\rent_nonpA.res
Results may be visualized using the S-Plus functions
'plotnonp1' and 'plotnonp2 or 'drawmap' for spatial data

Estimated variance parameter (sample mean): 0.00390422

L_random

Acceptance rate: 100 %
Results are stored in file d:\results\rent94_randomL.res
Estimated variance parameter (sample mean): 3.02058

Visualizing estimation results can be done easily in S-Plus using function 'plotnonp' for plotting nonlinear effects and 'drawmap' for visualizing the distribution of the spatial random effects. We type (in S-Plus):

```
plotnonp("c:\\results\\rent_nonpA.res", "c:\\results\\rent_nonpA.ps")
plotnonp("c:\\results\\rent_nonpF.res", "c:\\results\\rent_nonpF.ps")
readbndfile("c:\\bayes\\examples\\munich.bnd" ,"munich")
drawmap(map=munich,dfile="c:\\results\\rent_randomL.res"
,outfile="c:\\results\\rent_randomL.ps", regionvar=L,plotvar=Lmean,color=T)
```

The first two statements produce the plots printed in Figure 6.6. Both graphes are stored in a postscript file rather than printed on the sreen.
The third statement reads the map of Munich and stores it as a permanent S-Plus map object with name 'munich'. The last statement draws the map of Munich and colors the subquarters according to the estimated spatial random effect. The resulting graph is stored in the postscript file 'c:\results\rent_randomL.ps'. Figure 6.7 shows the graph.
A look at the effect of year of construction 'A' shows pretty large credible intervals until the year 1890. The reason is that we have only a couple of observations for the time period from 1800 to 1890. For that reason it may be reasonable to restrict our analysis only to flats built after 1890. This is achieved using the following statement

```
b.regress R = const + F(rw2) + A(rw2) + L(random) if A > 1890
, iterations=22000 burnin=2000 step=20 family=gaussian using rent
```

where we added an additional 'if' statement to the 'regress' command. To save space, printing estimation results is omitted.

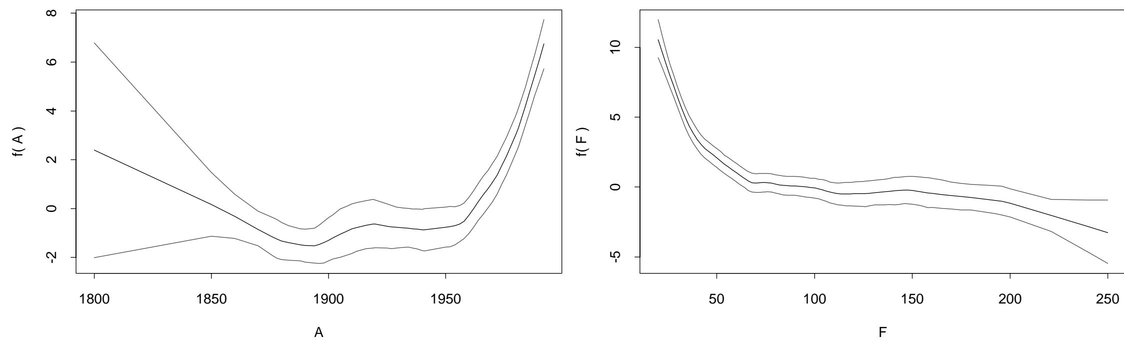


Figure 6.6: Estimated effects of floor space and year of construction. Shown is the posterior mean within 80 % credible regions.

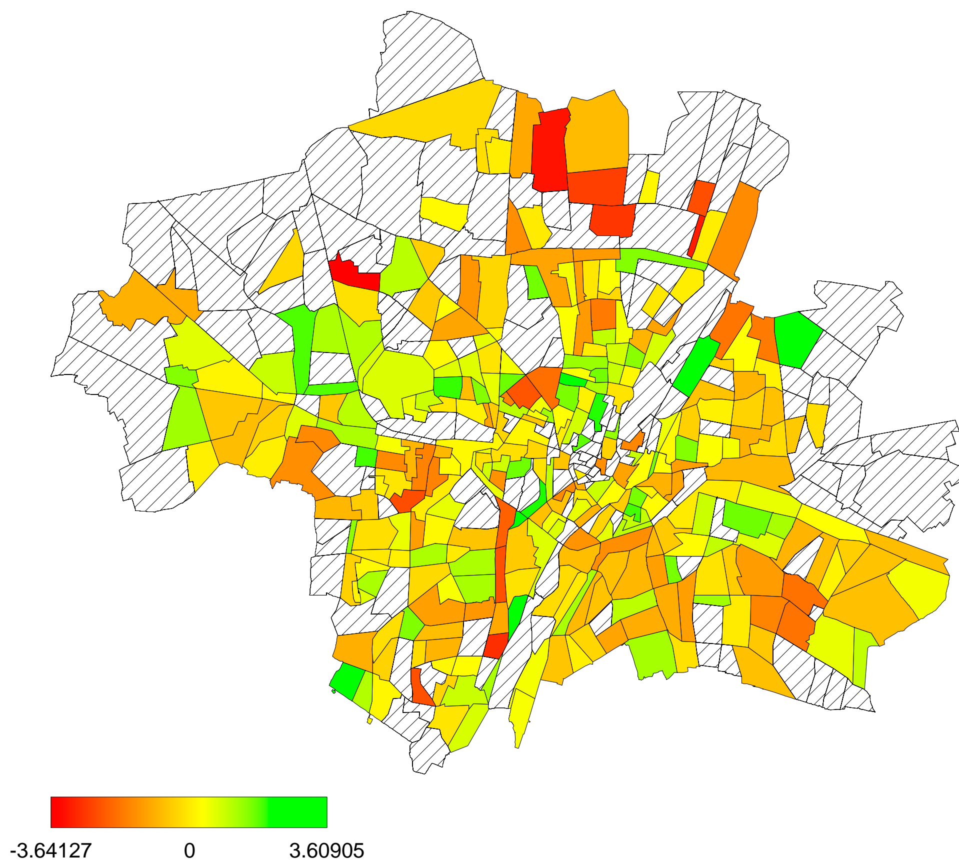


Figure 6.7: Estimated effect of location. Shown is the posterior mean.

Syntax	**Predictor**	**Description**
const	$\eta = +\beta_0+$	Constant Intercept.
X1	$\eta = +\beta_1 X1+$	Incorporates covariate X1 as a fixed effect into the model.
X1(rw1[,minsize,maxsize])	$\eta = +f_1(X1)+$	Defines a first order random walk model for the effect of X1. The blocksize for block move updates is between 'minsize and 'maxsize'. Note that the last two arguments are optional. In this case the minimum and maximum blocksize is chosen to be one (single move). If only 'maxsize' is omitted, the maximum blocksize is chosen to be equal to the minimum size.
X1(rw2[,minsize,maxsize])	$\eta = +f_1(X1)+$	Defines a second order random walk model for the effect of X1. The blocksize for block move updates is between 'minsize' and 'maxsize'. Note that the last two arguments are optional. In this case the minimum and maximum blocksize is chosen to be one (single move). If only 'maxsize' is omitted, the maximum blocksize is chosen to be equal to the minimum size.
X1(season,per[,minsize,maxsize])	$\eta = +f_1^S(X1)+$	Defines a seasonal effect of X1 with period 'per'. The blocksize for block move updates is between 'minsize' and 'maxsize'. This is meaningful only provided that X1 is a time scale. The blocksize for block move updates is between 'minsize' and 'maxsize'. Note that the last two arguments are optional. In this case the minimum and maximum blocksize is chosen to be one (single move). If only 'maxsize' is omitted, the maximum blocksize is chosen to be equal to the minimum size.
X1(spatial,mapobj)		Defines a spatial Markov random field prior for the spatial covariate X1 with geographical information stored in the map object 'mapobj'. Parameters are updated using a single move algorithm.
grvar(random)	$\eta = +b_{grvar}+$	Defines a random effect with respect to grouping variable 'grvar'.

Table 6.2: List of possible model terms

Syntax	**Predictor**	**Description**
X1*X2(rw1[,minsize,maxsize])	$\eta = +f(X2)X1+$	Defines a varying coefficient term, where the effect of X1 varies smoothly over the course of X2. Covariate X2 is the effectmodifier. The smoothness prior for f is a first order random walk. The blocksize is between 'minsize' and 'maxsize'.
X1*X2(rw2[,minsize,maxsize])	$\eta = +f(X2)X1+$	Defines a varying coefficient term, where the effect of X1 varies smoothly over the course of X2. Covariate X2 is the effectmodifier. The smoothness prior for f is a second order random walk. The blocksize is between 'minsize' and 'maxsize'.

Table 6.3: List of possible modelterms (continued)

Index

autocorrelation functions, 50
  computing of, 50
  plotting, 61

batch files, 13
Bayesian Inference, 41
Bayesian semiparametric regression, 36
bayesreg object, 36
  autocorr command, 50
  getsample command, 53
  global options, 63
  regress command, 36
boundary files, 31

changing existing variables, 26
Command window, 8
Conditional prior proposals, 42
credit scoring, 10

dataset, 15
  drop command, 15
  generate command, 21
  infile command, 22
  outfile command, 24
  rename command, 26
  replace command, 26
  simulation of, 28
  sort command, 28
dataset examples, 9
  credit scoring, 10
  rents for flats, 9
dataset objects, 15
delimiter, 13
dropping objects, 14
dropping observations, 15
dropping variables, 15

Exiting BayesX, 11
Expressions, 16
  constants, 19
  explicit subscribing, 19
  operators, 17

Functions, 18
  abs, 18
  bernoulli distributed random numbers, 18
  binomial distributed random numbers, 18
  cos, 18
  cumulative distribution function, 18
  exp, 18
  exponential distributed random numbers, 18
  floor, 18
  gamma distributed random numbers, 18
  lag, 18
  logarithm, 18
  normal distributed random numbers, 18
  sin, 18
  square root, 18
  uniformly distributed random numbers, 18

General syntax, 6
generalized additive models, 36
generalized linear models, 36
generating new variables, 21

Installation, 5
Interactions, 39
Interctions
  between a categorical and a metrical covariates, 39
  between categorical covariates, 39
  between metrical covariates, 39
  space time, 39

Log files, 11

map object, 31

boundary files, 31
infile command, 31
Markov chain monte carlo, 36
MCMC, 36, 41
Gaussian Response, 41
non-Gaussian Response, 42
metrical covariates, 37

Object browser, 8
Objects, 5
create, 5
dropping, 14
Operators, 17
arithmetic, 17
logical, 18
order of evaluation, 18
relational, 17
Output window, 8
saving the contents, 12

Plotting autocorrelation functions, 61
Plotting sampled parameters, 62

random effects, 39
Reading boundary files, 59
reading data from ASCII files, 22
renaming variables, 26
rents for flats, 9
Review window, 8

S-Plus
drawing geographical maps, 57
drawmap, 59
plotting autocorrelation functions, 61
plotting sampled parameters, 62
reading boundary files, 59
S-Plus functions, 54
Installation, 55
Plotting nonparametric functions, 55
sampled parameters, 53
saveoutput, 12
saving data in an ASCII file, 24
Simualtion of artificial datasets, 28
sorting variables, 28
spatial covariates, 38
Syntax, 6

time scales, 37

variables names, 28
varying coefficients models, 36

Weighted least squares proposals, 42
Windows, 8
command, 8
output, 8
review, 8
writing data to a file, 24