BSR 1.0: Bayesian Sparse Regression

Víctor de Buen

E-mail address: vdebuen@gmail.com
URL: www.tol-project.org

BSR 1.0: Bayesian Sparse Regression

Un sistema de simulación MCMC de modelos de regresión lineal sparse

Guía del programador y manuales de usuario

Octubre de 2008

SUMMARY. En este documento se presenta la versión inicial 1.0, aún en desarrollo, de un sistema abierto escrito en TOL[1] para la simulación e inferencia bayesianas de tipo MonteCarlo-Markov Chain (MCMC) mediante el algoritmo de Gibbs, sobre modelos de regresión lineal sparse con estructura arbitraria (jerárquicos, redes bayesianas, ...) con restricciones de desigualdad lineal, con tratamiento de omitidos y filtros no lineales, ambas tanto en el input como en el output, así como con estructuras ARIMA.

Gracias a Jorge Suit Pérez y Luis M. Artiles Martínez por su paciente revisión de errores, a José Almagro por la inspiración y a todo el equipo de Bayes en general por su apoyo incondicional.

Documento en construcción!

[1] ver `www.tol-project.org`

Índice general

Introducción

BSR es un sistema de estimación de modelos por simulación bayesiana, cuyo objetivo es satisfacer de la forma más eficiente posible las necesidades más comunes que nos encontramos los analistas de datos a la hora de resolver los problemas de previsión que nos proponen nuestros clientes. Muy a menudo dichos problemas suelen ser planteables como modelos jerárquicos lineales, en adelante HLM (Hierarchical Lineal Model), en los que aparecen ciertas no linealidades en el nivel observacional, como la presencia de estructura ARIMA, datos omitidos en el input o el output, heterocedasticidad, etc.; así como algunos efectos aditivos no lineales más complejos como funciones de transferencia (deltas) y otros más particulares que pueden aparecer en algunos proyectos.

Cada uno de estos problemas es más o menos complicado de resolver, y el método utilizado para hacerlos cooperar es la simulación MCMC de Gibbs. El mayor problema técnico con diferencia proviene paradójicamente del teóricamente más sencillo, que es la parte lineal. Esto es debido a que, incluso en problemas no muy grandes, conlleva una cantidad de cálculo insoportable (complejidad cúbica), si no se trata de una forma específica para aprovechar la gran cantidad de ceros que contiene la matriz del diseño (sparse matrix). Se trata de un sistema bastante complejo que ha servido de motor de evolución de TOL, al arrastrar como elementos previos indispensables la incorporación de NameBlock, un nuevo tipo de datos orientado al objeto (OOP) y VMatrix, matrices con un sistema de operaciones diseñado para hacer transparente al usuario las cuestiones estructurales y algorítmicas del álgebra matricial, tanto denso (BLAS-LAPACK) como sparse (CHOLMOD). En versiones ulteriores, se incorporará también la posibilidad de trabajo en paralelo en memoria compartida (OpenMP) y distribuida (Open MPI), para poder atacar problemas de grandes dimensiones (cientos de miles o millones de variables).

Un sistema complejo y potente supone en principio una gran fuente de errores así como una dificultad extrema en su uso y aprendizaje, por lo que desde el primer momento se ha tenido en cuenta la triple vertiente: eficiencia en el diseño algorítmico, robustez de los resultados mediante tests diagnósticos y flexibilidad de manejo. Esta última característica se logra merced a un sistema de capas de interfaz (API's) de distintos niveles, las cuales se apoyan cada una en la anterior. Las capas más interiores son más complicadas de utilizar, pero son más flexibles y permiten diseñar modelos más complejos, como redes bayesianas, y cualquier tipo de regresión lineal generalizada con las extensiones comentadas anteriormente, e incluso cualquier otra para la que se pueda dar un método de simulación condicionado al resto de componentes. Las capas exteriores son más rígidas pues se restringen a modelos cada vez más ceñidos a una clase muy concreta de modelos jerárquicos, pero a cambio ofrecen un manejo muy cómodo en el que el usuario define el modelo en unas pocas líneas de código TOL que queda registrado en una base de datos relacional (BsrHlm) que incluirá finalmente un resumen diagnóstico de los resultados obtenidos en la simulación.

Esta documentación se encuentra dividida en tres partes en función del tipo de lector objetivo, lo cual se indica en el prólogo de cada una, así que debe ser éste el que decida, en función de dichas indicaciones, qué partes debe leer con mayor atención y cuáles puede ojear o incluso eludir. Aunque cada una puede ser leída de forma independiente, las partes se presentan en el orden natural de dependencias:

- Parte I: Documentación matemática de BSR
- Parte II: Guía del programador de BSR
- Parte III: Manual de usuario de BSR-HLM

Parte 1

Documentación matemática de BSR

Esta parte interesa a los desarrolladores matemáticos de BSR y a todos aquellos que deseen comprender cómo funcionan las cosas por dentro desde el punto de vista del cálculo numérico y la estadística. Aquí se describen los conceptos y algoritmos matemáticos manejados por BSR para la simulación, diagnosis e inferencia bayesianas de la clase de modelos admisible.

Estos son los requisitos matemáticos previos necesarios para comprender sin dificultad los conceptos y algoritmos presentados en esta parte:

- Simulación e inferencia bayesianas
  - Generalidades de los métodos de MonteCarlo
  - Cadenas de Markov
  - Método de Gibbs
  - Método univariante ARMS
  - Diagnosis (CODA)
- Modelos de regresión gaussiana
  - Regresión lineal
    - Distribución condicionada del bloque lineal
    - Distribución condicionada de la varianza
  - Regresión ARIMA
    - Procesos estacionarios
    - Ecuaciones en diferencias regulares y estacionales (factorización)
    - La función de autocovarianzas
  - El modelo jerárquico
    - Nodos observacionales
    - Nodos latentes
    - Nodos a priori
  - Redes bayesianas

Estos son los resúmenes de cada capítulo:

1. Definición de la clase de modelos admisibles en BSR: describe la clase de modelos admisibles en BSR así como el método de Gibbs por bloques aplicado para la simulación.
2. El bloque lineal con restricciones lineales: Casi todos los modelos de regresión contienen uno o varios bloques lineales que suelen acumular la mayor parte de las variables por lo que es imprescindible atacar su simulación de la manera más eficiente posible, especialmente si se trata de matrices sparse como es bastante habitual, sobre todo en modelos estructurados como los modelos jerárquicos o las redes bayesianas. También es muy importante poder forzar restricciones en forma de inecuaciones sobre los parámetros de este bloque lineal. Para ello es necesario manejar algoritmos sumamente eficientes de generación de multinormales truncadas, descomposiciones de Cholesky y resolución de sistemas lineales sparse y densos, que permitan al mismo tiempo conjugarse con otros bloques de variables (Sigma, ARMA, Missing, ...) dentro del marco del método de simulación de Gibbs.
3. El bloque de varianzas: En un modelo de regresión con función de contraste arbitraria puede haber uno o varios bloques de de residuos que se postulan independientes de los que se quiere estimar su varianza.
4. El bloque ARIMA: En la simulación de Gibbs de un modelo ARMA se puede utilizar el método de simulación univariante ARMS para obtener realizaciones de los parámetros ARMA, si se fuerza a que los polinomios AR y MA estén factorizados en monomios y binomios. La matriz de covarianzas de un proceso ARMA genérico de longitud m es una matriz de Toeplitz simétrica que es más densa cuanto más cerca de la unidad estén las raíces de la parte AR. Para el caso MA puro es una matriz muy sparse de densidad aproximada q/m. En este artículo se describe una descomposición factorial de la matriz de covarianzas en el caso ARMA general mediante matrices sparse que se obtienen con coste computacional $O\left(n^2 m\right)$, en lugar del coste $O\left(m^3\right)$ del método de Cholesky denso, donde $n = \text{máx}\left(p, q\right) + 1 + p$, siendo p y q los grados de los polinomios AR y MA respectivamente.

5. El bloque de omitidos: En la simulación de Gibbs de una regresión lineal generalizada con datos omitidos en el input y en el output, si se condiciona al resto de parámetros, se obtiene otra regresión lineal cuyos parámetros son los valores omitidos.
6. El bloque no lineal: Muy a menudo se observan comportamientos de inputs que no actúan de forma lineal en el filtrado de fenómenos complejos. En el caso concreto de que la forma de actuar sea aditiva con respecto al ruido del modelo, es posible formalizar un método genérico ARMS de simulación de Gibbs de cada parámetro condicionado al resto. Se expone también el caso concreto de la función de transferencia no lineal (deltas).
7. Diagnosis: Cuando ya se dispone de una cadena de Markov para un modelo comienza la etapa de diagnosis que certifique la calidad de los resultados obtenidos e informe de la potencia alcanzable por las ulteriores inferencias que se puedan extraer de él.
8. Mantenimiento y producción: Una vez que acaba la etapa de análisis y diseño de un modelo se pasa a la etapa de producción en la que el modelo se usará para hacer inferencia de una u otra forma: predicción, decisión, etc. Durante esta etapa el modelo debe mantenerse en estado de revista, es decir, debe ir adaptándose a los nuevos datos que vayan conociéndose e incorporando eventuales nuevas variables.

CAPíTULO 1

Definición de la clase de modelos admisibles en BSR

Se formula el tipo de modelos que es capaz de tratar BSR en la forma más general posible, así como el método de Gibbs por bloques aplicado para la simulación.

1.1. Evolución de la modelación en TOL

Uno de los tipos de modelos más utilizados en la previsión de series temporales es el modelo ARIMA con función de transferencia lineal (omega)[1]. El estimador máximo verosímil de TOL para esta clase de modelos es la función Estimate y durante años ha servido para modelar fenómenos realmente complicados, gracias a su eficiencia de cálculo y a su facilidad de manejo, el tratamiento de omitidos en el output, etc. Sin embargo, presenta graves problemas inherentes al propio método de estimación máximo verosímil.

El más grave de todos ocurre cuando existen correlaciones múltiples entre grupos de variables de forma que los parámetros capturan bien el efecto conjunto pero no el particular de cada uno. Variables que deberían dar valores positivos se estiman como negativos y al revés simplemente porque ocurren casi de la misma forma que otros efectos de signo contrario. A veces es suficiente reformular los inputs de forma que sean lo más ortogonales entre sí que sea posible, pero otras veces no es viable hacerlo sin perder expresabilidad en el modelo o simplemente no hay datos suficientes en la muestra para poder distinguir inputs muy distintos en el fondo pero iguales en la forma para la muestra disponible. En estos casos es necesario que el analista pueda expresar toda la información a priori de que disponga para ayudar al estimador. En general si no se dispone de una adecuada información a priori, al hacer previsiones, el modelo funciona de forma muy distinta si éstas son extramuestrales o intramuestrales. En general, cuanto más información a priori fidedigna haya menos discrepancia se encontrará. Mucho cuidado con inventarse información a priori no justificada porque eso es seguro mucho peor que no dar ninguna información. Hubo algún intento de introducir mecanismos en este sentido para el estimador MLE pero no se llegó a ningún resultado aceptable.

Por razones de diferente tipo, a veces se conoce el signo o un intervalo de dominio de determinados parámetros, o bien que el efecto de unos debe ser mayor que el de otros (relaciones de orden), o, en general, se conoce cualquier tipo de restricción lineal. Este tipo de restricciones complican la convergencia de los métodos de optimización usados en MLE por lo que nunca se consiguió incorporarlos adecuadamente.

A la hora de manejar datos de clientes reales es muy frecuente que falten datos, no sólo de las series output a analizar, sino también de los inputs empleados en su explicación. Para un estimador MLE esto supone una fuente más de no linealidad que complica aún más la implementación y la convergencia del método.

Por los mismos motivos nunca llegaron a funcionar debidamente en el MLE otros sistemas de componentes no lineales, como funciones de transferencia amortiguadas (deltas) y otras propias del negocio en cuestión que no es posible ajustar a una forma no lineal paramétrica concreta sino que pueden tener formas arbitrarias cualesquiera.

Lo anterior se refiere a la descripción del modelo de regresión univariante de un nodo de observación concreto, como puede ser cada uno de los puntos de venta de un producto concreto, o cada una de las variantes de un producto o cualquier tipo de entidad que se quiera analizar en detalle. Pero también existen relaciones entre los parámetros de diferentes nodos de observación.

[1] Ver [**Box-Jenkins**]

Estas relaciones pueden ser expresadas a veces mediante una estructura jerárquica lineal mediante hiperparámetros no observables que se agrupan en nodos que llamaremos latentes y que se relacionan linealmente con los parámetros de los nodos obervados, o bien, con otros hiperparámetros de nodos latentes de niveles inferiores. Estas estructuras dan lugar a un aumento brutal del número de variables que un sistema MLE no puede soportar.

Este tipo de modelos jerárquicos se pueden ver como un caso particular de regresión lineal sparse, una vez filtrados el resto de efectos (ARIMA, omitidos, ...) de los nodos observacionales. Pero existen otras clases de modelos, como las redes bayesianas en los que las relaciones entre los parámetros e hiperparámetros no tienen porqué darse sólo en un sentido jerárquico, sino de forma arbitraria, manteniéndose sin embargo la característica sparse de la matriz de diseño conjunta. No es posible ni siquiera soñar con un estimador MLE capaz de trabajar con estas clases de modelos. En esta generalización, lo que realmente es común a todas ellas es que existen nodos de información o segmentos de regresión, observados o no, que introducen ecuaciones de regresión que involucran a parámetros del llamado bloque lineal, que pueden estar sujetos a restricciones de desigualdad lineal. Cada uno de estos segmentos tienen asociado un vector de residuos que puede tener o no una estructura de serie temporal, estructura ARIMA, datos omitidos, filtros no lineales, etc.

Todas estas razones han llevado a la conclusión irrevocable de que es preciso un sistema de estimación bayesiana basado en simulaciones de Montecarlo de cadenas de Markov (MCMC) generadas con métodos como Gibbs o Metropolis-Hastings. Varios han sido los intentos de implementación de estos sistemas, destacando entre todos BLR y HLM, y aunque los resultados han sido en general de una alta calidad, no se puede decir lo mismo respecto a la generalidad de los modelos admitidos ni de la eficiencia en los cálculos.

1.2. Formalización de la clase de modelos de BSR

A continuación se va a describir el modelo de regresión con restricciones lineales subdivisible en K segmentos de regresión cuyo único nexo en común es el vector $\beta \in \mathbb{R}^n$ de los parámetros a estimar en el bloque lineal conjunto de la regresión. Para el k-ésimo segmento de regresión, sean $Y^{(k)} \in \mathbb{R}^{M_k}$ el vector de output original a analizar, y $X^{(k)} \in \mathbb{R}^{M_k \times n}$ la matriz de inputs originales explicativos del segmento. La relación de regresión no se da, en general, sobre estos términos originales sino que se precisa hacer una serie de filtrados que se describen a continuación.

Tanto el input como el output originales pueden tener omitidos, los cuales deben ser sustituidos por ceros, para permitir realizar las operaciones aritméticas y algebraicas necesarias para su simulación y filtrado. Llamaremos $v^{(k)}$y $u^{(k)}$ a los vectores de parámetros correspondientes a los omitidos del output y el input respectivamente, y $\delta^{v^{(k)}}$y $\delta^{u^{(k)}}$ a los operadores lineales de ubicación[2] . De esta manera resultan el output y los inputs filtrados de omitidos, o simplemente filtrados sin más

$$\check{Y}^{(k)} = Y^{(k)} + \delta^{v^{(k)}} \cdot v^{(k)} \tag{1.2.1}$$

$$\check{X}^{(k)} = X^{(k)} + \delta^{u^{(k)}} \cdot u^{(k)} \tag{1.2.2}$$

La relación entre ambos es lo que define propiamente el segmento de regresión lineal en los parámetros lineales

$$\check{Y}^{(k)} = \check{X}^{(k)}\beta + w^{(k)} \tag{1.2.3}$$

El vector $w^{(k)} \in \mathbb{R}^{M_k}$ es el que se llamará ruido ARIMA o ruido sin diferenciar del segmento. Si se define la matriz $\triangle^{(k)} \in \mathbb{R}^{m_k \times M_k}$ como el operador lineal de rango $m_k \leq M_k$ de filtrado determinista, normalmente definido como la matriz de Toeplitz correspondiente a un polinomio de raíces unitarias, se obtendrá el ruido ARMA o ruido diferenciado $z^{(k)} \in \mathbb{R}^{m_k}$

$$z^{(k)} = \triangle^{(k)} w^{(k)} \tag{1.2.4}$$

[2]Se aconseja no pararse ahora demasiado en los detalles que se explicarán bien a fondo en el capítulo 5.1

Si no hay estructuras de este tipo se establece $m_k = M_k$ y $\triangle^{(k)} = I_{m_k}$. La distribución de este ruido ARMA viene dada por la ecuación en diferencias del modelo ARMA[3]

$$\begin{array}{c} \phi^{(k)}(B)\, z_t^{(k)} = \theta^{(k)}(B)\, e_t^{(k)} \\ e^{(k)} \sim N\left(0, \sigma_k^2 I_{m_k}\right) \end{array} \tag{1.2.5}$$

siendo $\sigma_k^2 \in \mathbb{R}^+$ el parámetro varianza[4] de $e^{(k)} \in \mathbb{R}^{m_k}$, los residuos incorrelados del segmento. En algunos segmentos puede venir prefijado el valor de la varianza como un dato a priori del modelo en lugar de como un parámetro a simular.

Sea la matriz

$$L^{(k)\,-1} = \frac{1}{\sigma_k} L^{-1}\left(\phi^{(k)}, \theta^{(k)}, m_k\right) \tag{1.2.6}$$

de descomposición simétrica de la matriz de covarianzas del modelo ARMA[5]. Si se premultiplican por ella el output e input filtrados y diferenciados

$$\begin{array}{l} Y'^{(k)} = L^{(k)\,-1} \triangle^{(k)} \check{Y}^{(k)} \\ X'^{(k)} = L^{(k)\,-1} \triangle^{(k)} \check{X}^{(k)} \end{array} \tag{1.2.7}$$

se obtiene la expresión del segmento de regresión lineal estandarizada

$$\begin{array}{c} Y'^{(k)} = X'^{(k)} \check{\beta} + \epsilon^{(k)} \\ \epsilon^{(k)} \sim N\left(0, I_{m_k}\right) \end{array} \tag{1.2.8}$$

donde $\epsilon^{(k)} = \frac{1}{\sigma_k} e^{(k)} \in \mathbb{R}^{m_k}$ son los residuos estandarizados del segmento.

Si se definen de forma conjunta para todos los segmentos de regresión, los vectores de output y residuos conjuntos y la matriz conjunta de inputs,

$$\begin{array}{c} Y'^{\,T} = \left(Y'^{(1)\,T}, \ldots, Y'^{(K)\,T}\right) \\ \epsilon^T = \left(\epsilon^{(1)T}, \ldots, \epsilon^{(K)T}\right) \\ X' = \left(X'^{(1)}, \ldots, X'^{(K)}\right) \end{array} \tag{1.2.9}$$

se obtiene la regresión lineal principal conjunta del modelo

$$\begin{array}{c} Y' = X' \cdot \check{\beta}^T + \epsilon \\ \epsilon \sim N\left(0, I_m\right) \\ m = \sum_{k=1}^{K} m_k \end{array} \tag{1.2.10}$$

donde el vector conjunto de parámetros estará sujeto opcionalmente a un sistema de inecuaciones lineales arbirarias de la forma

$$C \cdot \check{\beta} \leq c \tag{1.2.11}$$

siendo $C \in \mathbb{R}^{\rho \times n}$ la matriz de coeficientes de restricción y $c \in \mathbb{R}^{\rho}$ el vector de frontera, ambos conocidos y fijos[6]

En resumen, es posible clasificar los símbolos empleados asegún su papel en la simulación

- Datos: vienen dados en la propia definición del modelo y son $C, c, Y^{(k)}, X^{(k)}, \triangle^{(k)}, \delta^{v^{(k)}}, \delta^{u^{(k)}}$ y algunos σ_k^2 prefijados

[3] Más detalles en el capítulo 4.1
[4] Más detalles en 3.1
[5] Más detalles en 4.4
[6] Más detalles en 2.1

- Parámetros: son los valores que se quieren simular para poder hacer luego inferencia bayesiana sobre ellos o sobre funciones de ellos, lo cual incluye al resto de los σ_k^2 no prefijados y a $\beta, \alpha^{(k)}, \phi^{(k)}, \theta^{(k)}, v^{(k)}, u^{(k)}$
- Ruidos: son variables aleatorias cuya distribución se propone como hipótesis del modelo y son $w^{(k)}, z^{(k)}, e^{(k)}, \epsilon^{(k)}$
- Filtros: son variables auxiliares que se hacen necesarias durante la simulación de las diferentes componentes, como $\check{Y}^{(k)}, \check{X}^{(k)}, Y^{'(k)}, X^{'(k)}$y otras que irán surgiendo en la explicación detallada de cada bloque.

1.3. Extensiones de la clase de modelos BSR

Esta clase de modelos recién presentada puede ser ampliada mediante comoposición con otros modelos de tipo BSR o de otro tipo, como de bloques no lineales, heterocedasticidad, etc. De hecho, tanto la parte ARIMA como los omitidos podrían haber sido considerados como extensiones de la clase de modelos lineales, pero la estrecha relación entre todos estos bloques y las múltiples implicaciones entre sus respectivos filtrados hacen que sean más eficientes los algoritmos si se consideran dentro de un mismo marco. Cuando quepa duda de sobre qué se está hablando se especificará bien la clase básica de modelos BSR o bien la clase extendida de modelos BSR.

1.4. El modelo jerárquico lineal HLM

A continuación se verá cómo el modelo jerárquico se puede expresar como un caso particular de la clase de modelos recién descrita.

Los nodos observacionales admitirían opcionalmente parte ARIMA, datos omitidos y filtros no lineales. Todos los parámetros de este nivel son propiedad exclusiva de cada nodo observacional, incluidos los del bloque lineal, por lo que no hay ningún parámetro en común entre nodos observacionales.

La ecuación típica de un nodo latente es de la forma

$$\vartheta \sim N\left(X \cdot \eta, \sigma^2 I\right) \tag{1.4.1}$$

El símbolo η de los hiperparámetros del nodo latente formará parte del bloque lineal β general del modelo y el símbolo ϑ es un vector de parámetros del bloque lineal de nodos previamente definidos, es decir, bien parámetros β o α de nodos observacionales, o bien hiperparámetros η de nodos latentes previos. La varianza σ^2 puede ser un valor conocido dado o bien un parámetro a estimar, aunque esto último puede dar problemas de convergencia si no hay suficiente superficie de respuesta, es decir si hay pocas observaciones con respecto al número de variables.

Si se reescribe la ecuación del nodo latente de esta forma

$$\begin{aligned} 0 &= \vartheta - X \cdot \eta + e \\ e &\sim N\left(0, \sigma^2 I\right) \end{aligned} \tag{1.4.2}$$

es decir, si se define el vector de output como 0, entonces se puede escribir como un segmento de regresión de BSR muy simple, sin parte ARIMA, ni datos omitidos ni filtro no lineal.

La ecuación típica de un nodo a priori es de la forma

$$\vartheta \sim N\left(\mu, \sigma^2 I\right) \tag{1.4.3}$$

El símbolo ϑ es un vector de parámetros del bloque lineal de nodos previamente definidos, es decir, bien parámetros β o α de nodo observacionales , o bien hiperparámetros η de nodos latentes previos. La varianza σ^2 suele ser un valor conocido dado en este tipo de nodos, aunque perfectamente podría ser un parámetro a estimar, con los mismos problemas referidos para los nodos latentes.

Si se reescribe la ecuación del nodo latente de esta forma

$$\begin{aligned} \mu &= \vartheta + e \\ e &\sim N\left(0, \sigma^2 I\right) \end{aligned} \tag{1.4.4}$$

es decir, si se define el vector de output como μ, entonces se puede escribir como un segmento de regresión de BSR muy simple, sin parte ARIMA, ni datos omitidos ni filtro no lineal.

El bloque lineal de un modelo HLM tendría pues la siguiente forma de regresión lineal con residuos independientes

$$\begin{pmatrix} Y^O \\ Y^L \\ Y^P \end{pmatrix} = \begin{pmatrix} X^O & 0 \\ \nabla_O^L & -X^L \\ \nabla_O^P & \nabla_L^P \end{pmatrix} \cdot \begin{pmatrix} \beta \\ \vartheta \end{pmatrix} + \begin{pmatrix} e^O \\ e^L \\ Ye^P \end{pmatrix} \tag{1.4.5}$$

$$Y = \begin{pmatrix} Y^O \\ Y^L \\ Y^P \end{pmatrix} = \begin{pmatrix} Y^{O_1} \\ \vdots \\ Y^{O_{K_O}} \\ 0 \\ \vdots \\ 0 \\ \mu^{P_1} \\ \vdots \\ \mu^{P_{K_P}} \end{pmatrix} \tag{1.4.6}$$

$$X = \begin{pmatrix} X^O & 0 \\ \nabla_O^L & X^L \\ \nabla_O^P & \nabla_L^P \end{pmatrix} = \begin{pmatrix} X^{O_1} & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \ddots & \vdots & \vdots & & \vdots \\ 0 & \cdots & X^{O_{K_O}} & 0 & \cdots & 0 \\ \nabla_{O_1}^{L_1} & \cdots & \nabla_{O_{K_O}}^{L_1} & -X^{L_1} & \cdots & 0 \\ \vdots & & \vdots & \vdots & \ddots & \vdots \\ \nabla_{O_1}^{L_{K_L}} & \cdots & \nabla_{O_{K_O}}^{L_{S_L}} & 0 & \cdots & -X^{L_{K_L}} \\ \nabla_{O_1}^{P_1} & \cdots & \nabla_{O_{K_O}}^{P_1} & \nabla_{L_1}^{P_1} & \cdots & \nabla_{L_{K_L}}^{P_1} \\ \vdots & & \vdots & \vdots & & \vdots \\ \nabla_{O_1}^{P_{K_P}} & \cdots & \nabla_{O_{K_O}}^{P_{K_P}} & \nabla_{L_1}^{P_{K_P}} & \cdots & \nabla_{L_{K_L}}^{P_{K_P}} \end{pmatrix} \tag{1.4.7}$$

$$e = \begin{pmatrix} e^O \\ e^L \\ e^P \end{pmatrix} = \begin{pmatrix} e^{O_1} \\ \vdots \\ e^{O_{K_O}} \\ e^{L_1} \\ \vdots \\ e^{L_{K_L}} \\ e^{P_1} \\ \vdots \\ e^{P_{K_P}} \end{pmatrix} \wedge e^k \sim N\left(0, \sigma_k^2 I\right) \tag{1.4.8}$$

Los índices $O_1 = 1, \ldots, O_{K_O}$ se refieren a los segmentos de regresión correspondientes a los nodos observacionales, $L_1 = O_{K_O} + 1, \ldots, L_{K_L} = O_{K_O} + O_{K_L}$ a los latentes y $P_1 = O_{K_O} + O_{K_L+1}, \ldots, P_{K_P} = K$, de forma que todos los segmentos existentes son $K = K_O + K_L + K_P$.

Las dimensiones de los bloques son las siguientes

$$
\begin{gathered}
X^{O_j} \in \mathbb{R}^{m_{O_j} \times n_{O_j}} \\
X^{L_j} \in \mathbb{R}^{m_{L_j} \times n_{L_j}} \\
\nabla_i^{L_j} \in \mathbb{R}^{m_{L_j} \times n_{O_i}} \\
\nabla_i^{P_j} \in \mathbb{R}^{m_{P_j} \times n_{O_i}} \\
n_O = \sum_{k=1}^{K_O} n_{O_k} \wedge n_L = \sum_{k=1}^{K_L} n_{L_k} \wedge n = n_O + n_L \\
m_O = \sum_{k=1}^{K_O} m_{O_k} \wedge m_L = \sum_{k=1}^{K_L} m_{L_k} \wedge m_P = \sum_{k=1}^{K_P} m_{P_k} \wedge m = m_O + m_L + m_P
\end{gathered} \tag{1.4.9}
$$

Las matrices de coeficientes que ligan los nodos observacionales con los latentes $\nabla_{O_j}^{L_i}$, al igual que las de $\nabla_{O_j}^{P_i}$ y los $\nabla_{L_j}^{P_i}$, son todas ellas diagonales con valores 1 ó 0 y estan conjuntamente restringidas a que sólo una de entre todas ellas puede tener un 1 en una fila dada, y lo mismo para una columna específica. Es decir, una variable sólo puede estar relacionada con una sola ecuación de un nodo, bien latente bien a priori. Es decir, el bloque de X

$$
\nabla_O^* = \begin{pmatrix} \nabla_O^L \\ \nabla_O^P \end{pmatrix} \tag{1.4.10}
$$

tiene todas las celdas nulas excepto un máximo de n_O celdas iguales a 1. Análogamente el bloque

$$
\nabla_*^P = \begin{pmatrix} \nabla_O^P & \nabla_L^P \end{pmatrix} \tag{1.4.11}
$$

tiene todas las celdas nulas excepto un máximo de $n = n_O + n_L$ celdas iguales a 1; y al mismo tiempo ∇_L^P tiene todas las celdas nulas excepto un máximo de n_L. La finalidad de estas restricciones es que se asegure que no existen ecuaciones incongruentes entre sí para un mismo parámetro hijo.

Se pone ahora de manifiesto la estructura sparse el modelo HLM de la que se venía hablando anteriormente.

1.5. El modelo jerárquico lineal de output combinado HLM-OC

Una generalización del modelo jerárquico que aparece de forma bastante natural es aquella en la que el output de un nodo latente o a priori no es directamente un único parámetro de un nodo hijo sino que puede ser cualquier combinación lineal de parámetros hijo. La estructura matricial de bloques es exactamente igual que la expresada en la ecuación 1.4.7, sólo que ahora las matrices de bloque ∇_*^* no tienen porqué ser de valores 1 ó 0, sino que pueden tener valores arbitrarios, aunque eso sí, se mantienen las mismas restricciones entre ellas, es decir, si un parámetro aparece en uno de esos bloques no debería aparecer en ningún otro, y además deben ser todas ellas de rango completo igual al número de filas de cada una.

La ecuación del nodo latente con output combinado sería de la forma

$$
\nabla \cdot \vartheta \sim N\left(X \cdot \eta, \sigma^2 I\right) \tag{1.5.1}
$$

reescribible como

$$
\begin{gathered}
0 = \nabla \cdot \vartheta - X \cdot \eta + e \\
e \sim N\left(0, \sigma^2 I\right)
\end{gathered} \tag{1.5.2}
$$

La ecuación del nodo a priori con output combinado sería de la forma

$$
\nabla \cdot \vartheta \sim N\left(\mu, \sigma^2 I\right) \tag{1.5.3}
$$

reescribible como

$$
\begin{gathered}
\mu = \nabla \cdot \vartheta - X \cdot \eta + e \\
e \sim N\left(0, \sigma^2 I\right)
\end{gathered} \tag{1.5.4}
$$

Resulta evidente que no deja de ser un caso particular de la clase de modelos de BSR, e incluso lo seguiría siendo sin cumplir esas restricciones, las cuales tienen por único objetivo el asegurar que los modelos son congruentes. A veces, por comodidad, se relajará la notación y se llamará simplemente modelo jerárquico a esta generalización.

1.6. Simulación de Gibbs

El método de simulación de Gibbs[7] consiste en tomar cada variable o bloque de variables del cual se conoce un método de simulación para su distribución condicionada al resto de variables del modelo. Luego se actualizan los valores de ese bloque y toda aquella información auxiliar derivada de las mismas para pasar a simular otro bloque. Existen diversas estrategias para el orden de los bloques: ciclos, secuencias simétricas, selección aleatoria simple, selección de permutaciones aleatorias, etc. Empíricamente se ha observado que el mejor orden para esta clase de modelos es el siguiente:

1. Bloques de output omitidos
2. Bloques de input omitidos
3. Bloques ARMA
4. Bloque de varianzas
5. Bloque lineal

La simulación de cada bloque supone una serie de actuaciones sobre ciertas componentes del modelo que se ven afectadas por los cambios en dicho bloque. Por ejemplo, al cambiar los omitidos del output $v^{(k)}$ se modifica el output filtrado $\check{Y}^{(k)}$ y por tanto el diferenciado $Y^{'(k)}$, y lo mismo le ocurre a éste último si se tocan los parámetros ARMA del output. Es de vital importancia por tanto que estén perfectamente identificados todas las componentes auxiliares que comunican a unos bloques con otros y que el orden de simulación esté perfectamente establecido para que los condicionamientos se propaguen adecuadamente.

[7] Ver [**Bayesian Data Analysis**]

CAPíTULO 2

El bloque lineal con restricciones lineales

Casi todos los modelos de regresión contienen uno o varios bloques lineales que suelen acumular la mayor parte de las variables por lo que es imprescindible atacar su simulación de la manera más eficiente posible, especialmente si se trata de matrices sparse como es bastante habitual, sobre todo en modelos estructurados como los modelos jerárquicos o las redes bayesianas. También es muy importante poder forzar restricciones en forma de inecuaciones sobre los parámetros de este bloque lineal. Para ello es necesario manejar algoritmos sumamente eficientes de generación de multinormales truncadas, descomposiciones de Cholesky y resolución de sistemas lineales sparse y densos, que permitan al mismo tiempo conjugarse con otros bloques de variables (Sigma, ARMA, Missing, ...) dentro del marco del método de simulación de Gibbs.

2.1. Descripción

Sea el modelo de regresión lineal estandarizada de rango completo con restricciones lineales

$$\begin{gathered} Y = X \cdot \beta + \varepsilon \\ C \cdot \beta \leq c \\ \varepsilon \sim N(0, I) \\ Y, \varepsilon \in \mathbb{R}^m \\ \beta \in \mathbb{R}^n \wedge X \in \mathbb{R}^{m \times n} \\ c \in \mathbb{R}^\rho \wedge C \in \mathbb{R}^{\rho \times n} \\ m > n \\ rank(X) = n \end{gathered} \tag{2.1.1}$$

donde

- β son los parámetros a estimar de la regresión,
- n es el número de parámetros a estimar de la regresión,
- m es el número de datos de contraste del modelo y ha de ser mayor que el de variables,
- ρ es el número de restricciones de inecuaciones lineales,
- Y es la matriz de output estandarizado del modelo y es completamente conocida, aunque puede cambiar de una simulación para otra si estamos en el marco de una simulación de Gibbs con más bloques involucrados. Para estandarizarla normalmente se habrá dividido previamente cada segmento por su correspondiente parámetro de desviación típica, y también se habrá filtrado eventualmente de las partes no lineales: ARMA, omitidos, etc.
- X es la matriz de iputs del modelo y es completamente conocida y de rango completo, aunque puede cambiar de una simulación para otra lo mismo que el output. Para estandarizarla normalmente se hará igual que con el output.
- ε son los residuos estandarizados del modelo cuya distribución se propone como hipótesis principal del mismo,
- C es la matriz de coeficientes de restricción, que es conocida y también podría cambiar en cada simulación
- c es el vector de frontera de restricción que es igualmente conocida aunque no necesariamente fija.

2.2. Simulación del bloque lineal de rango completo

Condicionando a la varianza σ^2, la distribución de β es de sobras conocida y no ofrece mayores dificultades a nivel teórico, pues se trata simplemente de una multinormal truncada

$$\beta \sim TN\left(\mu, \sigma^2\Sigma, C \cdot \beta \leq c\right) \tag{2.2.1}$$

o bien una multinormal sin truncar si no hay restricciones, es decir, si $\rho = 0$

$$\beta \sim N\left(\mu, \sigma^2\Sigma\right) \tag{2.2.2}$$

En ambos casos la media y la matriz de covarianzas son las mismas

$$\begin{aligned} \Sigma &= \left(X^T \cdot X\right)^{-1} \\ \mu &= \Sigma \cdot X^T \cdot Y \end{aligned} \tag{2.2.3}$$

Si se dispone de una descomposición simétrica de la matriz de información, como podría ser la de Cholesky o cualquier otra que cumpla

$$\begin{aligned} X^T \cdot X &= L \cdot L^T \\ \Sigma &= L^{-T} \cdot L^{-1} \end{aligned} \tag{2.2.4}$$

entonces es posible definir el cambio de variable de estandarización

$$\begin{aligned} \beta &= L^{-T} \cdot z + \mu \\ z &= L^T \cdot (\beta - \mu) \\ E[z] &= 0 \\ Cov[z] &= L^T \cdot \Sigma \cdot L = I \end{aligned} \tag{2.2.5}$$

Las restricciones sobre esta nueva variable quedarían así

$$\begin{aligned} C \cdot \left(L^{-T} \cdot z + \mu\right) &\leq c \\ C \cdot L^{-T} \cdot z &\leq c - C \cdot \mu \end{aligned} \tag{2.2.6}$$

y definiendo

$$\begin{aligned} D &= C \cdot L^{-T} \\ d &= c - C \cdot \mu \end{aligned} \tag{2.2.7}$$

se obtiene que la distribución de es una multinormal independiente truncada

$$z \sim TN\left(0, I, D \cdot z \leq d\right) \tag{2.2.8}$$

2.2.1. Algoritmo de factorización. Aunque a nivel teórico el método presentado no ofrece gran dificultad, no es trivial obtener la descomposición de Cholesky de una forma eficiente cuando el número de datos m y sobre todo el de variables n empiezan a crecer. Para empezar, algo tan aparentemente inofensivo como el producto $X^T \cdot X$ precisa nada menos que de $n^2 \cdot m$ productos y $n^2 \cdot (m-1)$ sumas de números reales. Con unos pocos cientos de variables y apenas unos miles de datos se va a los centenares de millones de operaciones. Cuando X es fija a lo largo de las simulaciones no tiene ninguna importancia, pero si no es así hay que repetir el proceso miles o decenas de miles de veces con lo cual se alcanza sin dificultad los billones de operaciones aritméticas.

Si X cambia dependiendo de forma univariante o como mucho bivariante es posible utilizar un método de interpolación para ahorrar tiempo de computación. Este es el caso de la matriz de diseño extendido de un nodo observacional dentro de un modelo jerárquico.

Si X es sparse entonces existen métodos de descomposición de $X^T \cdot X$ sin necesidad de construir explícitamente el producto. Concretamente el paquete CHOLMOD [**User Guide for CHOLMOD**] dispone de un método muy eficiente tanto en la descomposición, como en la resolución de los sistemas triangulares asociados.

Para el cálculo de μ conviene disponer los cálculos así

$$\mu = L^{-T} \cdot \left(L^{-1} \cdot \left(Y^T \cdot X\right)\right) \tag{2.2.9}$$

2.2.2. Algoritmo de factorización precondicionada. Cuando la cadena de Markov ha convergido es de esperar que no haya grandes diferencias entre las matrices L de simulaciones consecutivas. Si es así puede ser de gran ayuda usar la descomposición L_0 de la iteración anterior como precondicionador de la actual, puesto que se tendrá lo siguiente

$$\begin{aligned} \dot{X} &= X \cdot L_0 \\ \dot{X}^T \cdot \dot{X} &= \dot{L} \cdot \dot{L}^T \simeq I \end{aligned} \tag{2.2.10}$$

lo cual hace suponer que las $\dot{X}$ y $\dot{L}$ transformadas son más sparse que las correspondientes matrices originales, y por tanto más rápidas de factorizar. Es posible que para que realmente sean sparse sea necesario truncar los valores próximos a 0 según un parámetro de tolerancia predefinido. Esto será útil también independientemente de que se precondicione o no, si existen circunstancias que provoquen valores próximos a 0 que puedan ser truncados, como por ejemplo si existe parte ARMA, puede ocurrir que variables originalmente sparse se conviertan en falsas densas con valores cercanos a 0. Una vez realizada esta factorización se obtendrá la original teniendo en cuenta que

$$X^T \cdot X = L_0^{-T} \cdot \dot{X}^T \cdot \dot{X} \cdot L_0^{-1} = L_0^{-T} \cdot \dot{L} \cdot \dot{L}^T \cdot L_0^{-1} \tag{2.2.11}$$

es decir

$$\begin{aligned} L &= L_0^{-T} \cdot \dot{L} \\ L^{-1} &= \dot{L}^{-1} \cdot L_0^T \end{aligned} \tag{2.2.12}$$

En principio no sería necesario tampoco actualizar la L_0 en cada iteración sino tras un determinado número de iteraciones que incluso podría ser creciente conforme la cadena se va estabilizando. Un criterio razonable es el tiempo que lleva el cálculo realizado con precondicionamiento con respecto al tiempo que llevó la última vez que se factorizó la matriz $X^T \cdot X$ original. Mientras el tiempo sea sustancialmente menor merecerá la pena continuar con la misma matriz de precondicionamiento. Si ya la primera iteración cuesta más o apenas poco menos el método de factorización original, entonces se vuelve a este durante un número dado de iteraciones. Si este número se define como infinito o simplementente mayor que la longitud de la cadena, entonces nunca se intenta el método precondicionado y todo queda como estaba.

2.2.3. Algoritmo de simulación de la multinormal estandarizada truncada. El algoritmo utilizado para generar los z está basado en el Sampler TN2 descrito en las páginas 9-11 del artículo [**Sampling Truncated Multinormal**], y de hecho es conceptualmente idéntico salvo que en la implementación de la fórmula 24 se evitan los productos $D_{-j} \cdot z_{-j}$ actualizando el vector $D \cdot z$ que se calcula sólo una vez al principio.

El algoritmo es el siguiente

1. Se parte de un vector z que cumple las restricciones $D \cdot z \leq d$
2. Se calcula y almacena el producto $\zeta = D \cdot z \in \mathbb{R}^\rho$
3. Para cada columna $j = 1 \ldots n$ de D se siguen los siguientes pasos
   a) Se extrae el vector $\zeta^j = (D_{j,1} \cdot z_j, \ldots, D_{j,r} \cdot z_j)^T \in \mathbb{R}^r$
   b) Se calcula $\zeta^{-j} = \zeta - \zeta^j$
   c) Se calcula el límite inferior $\lambda_j = \max\limits_{i=1\ldots\rho} \left\{ \frac{d_i - \zeta_{-j}}{D_{i,j}} \middle| D_{i,j} < 0 \right\}$
   d) Se calcula el límite superior $\kappa_j = \min\limits_{i=1\ldots\rho} \left\{ \frac{d_i - \zeta_i^{-j}}{D_{i,j}} \middle| D_{i,j} > 0 \right\}$
   e) Se simula la normal truncada estándar univariante $\xi_j \sim TN(0, 1, \lambda_j, \kappa_j)$
   f) Se modifica $\zeta \leftarrow \zeta^{-j} + (D_{j,1} \cdot \xi_j, \ldots, D_{j,r} \cdot \xi_j)^T$
   g) Se modifica la componente igualándola al valor simulado $z_j \leftarrow \xi_j$

2.2.4. Inicialización de la cadena de Markov con una solución factible. Pendiente!!

2.3. Descomposición de bloques lineales

Cuando el bloque lineal de un modelo tiene demasiadas variables puede ser necesario dividirlo en varias partes para acometerlo bien secuencialmente o bien en paralelo si sus dimensiones sobrepasan la capacidad de la máquina y el diseño del modelo lo permite.

2.3.1. Descomposición del modelo jerárquico. El modelo jerárquico HLM presentado en 1.4, es un caso típico de modelo divisible de forma natural. Cada nodo observacional es independiente del resto de nodos observacionales y está relacionado con una reducida selección de ecuaciones de los nodos latentes y a priori de forma exclusiva, es decir, cada ecuación latente o a priori sólo afecta a un nodo observacional como máximo, ya que alternativamente podría afectar a un nodo latente. Aunque esto mismo se podría decir para los nodos latentes de cada nivel con respecto al nivel superior, lo normal es que, en los modelos masivos donde puede ser interesante la descomposición, el tamaño relativo de la parte latente y a priori es insignificante con respecto a la observacional por lo que no es necesario descomponerlos.

Se dispone por tanto de una partición inconexa de las ecuaciones del modelo en una serie de bloques con las ecuaciones ligadas a cada nodo observacional, más otro bloque con el resto de ecuaciones ligadas a los nodos latentes. En cada bloque observacional se debe introducir las ecuaciones respetando los segmentos de donde provienen, es decir, debe haber un segmento para las ecuaciones del nodo observacional y otro segmento para las ecuaciones de cada nodo padre, latente o a priori. Nótese que la varianza de los nodos latentes que no estén fijas serán un parámetro más de condicionamiento externo para los bloques observacionales, mientras que las varianzas de estos no serán parámetros condicionantes del bloque de latentes, que sólo dependen de las variables del bloque lineal de sus hijos.

Esta descomposición en bloques permite además ejecutar las simulaciones de los nodos observacionales en paralelo como bloques de Gibbs condicionados al bloque único de los nodos latentes, y después habría que simular éste bloque conjunto de latentes condicionado a todos los observacionales.

Lo dicho anteriormente reza para el modelo HLM sin restricciones de desigualdad ya que si existiera alguna de ellas que afectera a varios nodos al mismo tiempo entonces la distribución de los mismos ya no sería independiente y habría que mantenerlos dentro de un mismo bloque de Gibbs. En este caso sería necesario dividir los nodos en clases de equivalencia con la relación 'existe una restricción de desigualdad entre variables de ambos nodos'. En el peor de los casos se podría tener todos los nodos en una misma clase y no sería posible la simulación en paralelo.

Los nodos que estuvieran relacionados aún se podrían dividir en un bloque BSR para cada uno, sólo que la simulación habría que hacerla secuencial dentro de la clase de nodos relacionados, puesto que todos dependen de todos. Esto podría ser útil si no fuera posible almacenar en RAM el bloque conjunto de toda la clase de nodos relacionados.

2.3.2. Descomposición del modelo jerárquico con output combinado. El modelo jerárquico con output combinado HLM-OC presentado en 1.5, es también un modelo divisible aunque en este caso sólo son separables los nodos que no comparten variables con coeficiente no nulo en los bloques ∇_*^* de ninguna ecuación latente ni a priori. En este caso sería necesario dividir los nodos en clases de equivalencia con la relación 'existe o bien una restricción de desigualdad, o bien una combinación de output latente o a priori, entre variables de ambos nodos '.

2.3.3. Simulación del bloque lineal de rango incompleto. Otro caso en el que puede ser útil descomponer un modelo BSR en varios ocurre cuando hay menos datos que variables en el bloque lineal, o en general, cuando se tiene un modelo de rango incompleto. Si la matriz X no fuera de rango completo, $rank\,(X) = r < n\ '$, siempre que no tenga ninguna columna totalmente nula, aún sería posible simular los β rompiéndolos en J trozos

$$X \cdot \beta \ = \begin{pmatrix} X^{(1)} & \cdots & X^{(J)} \end{pmatrix} \cdot \begin{pmatrix} \beta^{(1)} \\ \vdots \\ \beta^{(J)} \end{pmatrix} \tag{2.3.1}$$

$$\beta^{(j)} \in \mathbb{R}^{n_j} \wedge X^{(j)} \in \mathbb{R}^{m \times n_j} \forall j = 1 \dots J \tag{2.3.2}$$

elegiendo los bloques de forma que

$$n_j = rank\left(X_j\right) < m \tag{2.3.3}$$

. Es decir, si hay más variables que datos hay que romper en tantos bloques como haga falta para que cada bloque tenga al menos tantos datos como variables, y si hay colinealidades hay que romperlas poniendo una de las columnas implicadas en un bloque distinto.

En estos modelos la varianza debería ser dada como un parámetro fijo y no ser simulada, o bien serlo con algún tipo de restricciones o distribución a priori

Otra cuestión aparte es que la simulación los modelos de rango incompleto realmente llegue a converjer, es decir, una cosa es que se pueda simular algo condicionando a otro algo y otra cosa es que eso converja a alguna parte. La experiencia demuestra que incluso con modelos de rango completo pero con mucha correlación entre variables, la cadena no converge sino que se da paseos entre lo que soluciones prácticamente equivalentes. Si eso ocurre sin llegar a ser colineal, está claro que con rango incompleto es de esperar problemas de ese estilo.

Incluso en el caso en que convergiera la superficie de respuesta de este tipo de modelos, es menor que 1, $m/n < 1$, por lo que habría que ser extraordinariamente cautos a la hora de hacer inferencia con ellos, pues incluso con superficies de respuesta de $m/n = 10$ pueden aparecer facilmente problemas de sobreajuste, de forma que se obtienen buenos resultados intra-muestrales pero las inferencias extra-muestrales son significativamente peores.

Nótese que en este caso no cabe la posibilidad de ejecución en paralelo ya que cada bloque depende de todos los demás, por lo que el condicionamiento ha de ser secuencial.

En cualquier caso, siempre que sea posible, será preferible añadir información a priori razonable de forma que la matriz del diseño resulte de rango completo. Por ejemplo, en un modelo con k_{obs} nodos observacionales con m_{obs} datos y n_{obs} variables relativas a los mismos conceptos en cada nodo, si tiene sentido aplicar un hiperparámetro a cada una de las n_{obs} entonces se añaden $k_{obs} \cdot n_{obs}$ ecuaciones latentes y si además se dispone de información a priori sobre dichos hiperparámetros, se agregarán otras n_{obs} ecuaciones a priori.

CAPíTULO 3

El bloque de varianzas

En un modelo de regresión con función de contraste arbitraria puede haber uno o varios bloques de de residuos que se postulan independientes de los que se quiere estimar su varianza.

3.1. Descripción

Un modelo genérico de regresión normal puede verse como un conjunto de funciones arbitrarias de contraste de segmentos de residuos normales independientes con varianzas desconocidas

$$\begin{gathered} e_k = \mho^k \left(Y^k, \Upsilon\right) \forall k = 1 \dots K \\ e_k \sim N\left(0, \sigma_k^2 I\right) \\ \Upsilon \in \mathbb{R}^n \\ Y^k, e_k \in \mathbb{R}^{m_k} \\ \sum_{k=1}^{K} m_k > n \end{gathered} \tag{3.1.1}$$

donde

- σ_k^2 es el parámetro del bloque de varianzas o σ-block referido al k-ésimo segmento de regresión,
- Υ son todos los parámetros de la regresión excepto los del bloque de varianzas,
- n es el número de parámetros a estimar de la regresión,
- Y^k es el vector de output conocido del k-ésimo segmento de regresión.
- $\mho^k$ es la función arbitraria de contraste del k-ésimo segmento de regresión y formaliza las hipótesis que se desean contrastar sobre el output.
- m_k es el número de datos de contraste del k-ésimo segmento,
- e_k son los residuos del k-ésimo segmento de regresión, y su distribución se propone como hipótesis principal del mismo,

3.2. Simulación del bloque de varianzas

Condicionando al resto de parámetros Υse obtienen los vectores de residuos de un segmento concreto, prescindiendo del índice k

$$e = f\left(Y, \Upsilon\right) \tag{3.2.1}$$

La suma de sus cuadrados dividida por la varianza se distribuyen como una χ^2_{m-1}

$$\tfrac{1}{\sigma^2} \sum_{t=1}^{m} e_t^2 \sim \chi^2_{m-1} \tag{3.2.2}$$

luego σ_k^2 se puede simular así

$$\begin{gathered} \sigma^2 \sim \frac{s^2}{\chi^2_{m-1}} \\ s^2 = \sum_{t=1}^{m} e_t^2 \end{gathered} \tag{3.2.3}$$

3.3. Información a priori

En [**Bayesian Data Analysis**], sección 14.8, subsección "Prior information about variance parameters" página 384, se propone un método de incluir información a priori sobre la varianza de un modelo de regresión con residuos normales, equivalente a añadir m_0 residuos auxiliares con un valor central de varianza a priori σ_0.

$$\sigma^2 \sim \frac{m_0\sigma_0^2+s^2}{\chi^2_{m_0+m-1}} \tag{3.3.1}$$

Para que el prior no sea dependiente de la longitud actual del segmento, es preferible definirlo en virtud del peso del valor central a priori en la distribución a posteriori

$$\lambda_0 = \frac{m_0}{m_0+m} \in [0,1] \tag{3.3.2}$$

obteniéndose el valor de m_0 en función de m

$$m_0 = m\frac{\lambda_0}{1-\lambda_0} \in [0,1] \tag{3.3.3}$$

Obsérvese que dar un peso $\lambda_0 = 0$ es equivalente a no tener ningún a priori para σ^2, o más exactamente a suponer a priori uniforme en $(0,\infty)$, y un peso $\lambda_0 = 1$ es equivalente a un prior determinista, es decir a fijar $\sigma^2 = \sigma_0^2$. Este parámetro peso se puede interpretar como la credibilidad del a priori.

3.4. Modelos heterocedásticos

A veces ocurre que la varianza dentro de un segmento no permanece constante sino que cambia de magnitud en virtud de ciertos fenómenos endógenos y exógenos, es decir, que se puede considerar que la serie de varianzas, bajo cierta transformación tiene su propio modelo ARIMA con función de transferencia, lo cual es una generalización natural de los modelos ARCH y GARCH [1].

La forma de tratar este problema en BSR es la composición de bloques de modelos en el que se crea un modelo BSR anexo al principal en el que para cada segmento heterocedástico del principal, habrá un segmento de regresión cuyo output será la serie estandarizada de cuadrados de residuos heterocedásticos

$$u^2_{t,k} = \left(\frac{1}{m}\sum_{t=1}^{m} e^2_{k,t}\right)^{-1} e^2_{k,t} \tag{3.4.1}$$

Una vez simulado el modelo producirá unas previsiones normalizadas $\hat{u}^2_{t,k}$, es decir, tales que

$$\frac{1}{m}\sum_{t=1}^{m} \hat{u}^2_{k,t} = 1 \tag{3.4.2}$$

que se utilizarán para reconstruir el modelo lineal en términos homocedásticos

$$\begin{aligned} \dot{Y}_{k,t} &= \frac{1}{\hat{u}_{k,t}} Y_{k,t} \\ \dot{X}_{k,t,j} &= \frac{1}{\hat{u}_{k,t}} X_{k,t,j} \end{aligned} \tag{3.4.3}$$

[1] Ver [**ARCH in UK**] y [**GARCH**]

CAPíTULO 4

El bloque ARIMA

En la simulación de Gibbs de un modelo ARMA se puede utilizar el método de simulación univariante ARMS para obtener realizaciones de los parámetros ARMA, si se fuerza a que los polinomios AR y MA estén factorizados en monomios y binomios. La matriz de covarianzas de un proceso ARMA genérico de longitud m es una matriz de Toeplitz simétrica que es más densa cuanto más cerca de la unidad estén las raíces de la parte AR. Para el caso MA puro es una matriz muy sparse de densidad aproximada q/m. En este artículo se describe una descomposición factorial de la matriz de covarianzas en el caso ARMA general mediante matrices sparse que se obtienen con coste computacional $O\left(n^2 m\right)$, en lugar del coste $O\left(m^3\right)$ del método de Cholesky denso, donde $n = \text{máx}\left(p, q\right) + 1 + p$, siendo p y q los grados de los polinomios AR y MA respectivamente.

4.1. Descripción

Sea el proceso ARIMA

$$\begin{gathered} \phi\left(B\right) \triangle\left(B\right) w_t = \theta\left(B\right) e_t \\ e_t \sim N\left(0, \sigma^2\right) \\ z_t = \triangle\left(B\right) w_t \\ \deg\left(\phi\left(B\right)\right) = p \\ \deg\left(\theta\left(B\right)\right) = q \\ t = 1 \ldots m \end{gathered} \tag{4.1.1}$$

tal que los polinomios AR y MA se encuentran factorizados en monomios o binomios de estacionalidades s_k arbitrarias de la siguiente manera

$$\begin{gathered} \phi\left(B\right) = 1 - \sum_{j=1}^{p} \phi_j B^j = \prod_{k=1}^{K} \phi^k\left(B^{s_k}\right) \\ \theta\left(B\right) = 1 - \sum_{j=1}^{q} \theta_j B^j = \prod_{k=1}^{K} \theta^k\left(B^{s_k}\right) \\ \deg\left(\phi^k\left(B\right)\right) = p_k \\ \deg\left(\theta^k\left(B\right)\right) = q_k \\ \phi^k\left(B^{s_k}\right) = \begin{cases} 1 - \phi_1^k B^{s_k} & si\ p_k = 1 \\ 1 - \phi_1^k B^{s_k} - \phi_2^k B^{2s_k} & si\ p_k = 2 \end{cases} \\ \theta^k\left(B^{s_k}\right) = \begin{cases} 1 - \theta_1^k B^{s_k} & si\ q_k = 1 \\ 1 - \theta_1^k B^{s_k} - \theta_2^k B^{2s_k} & si\ q_k = 2 \end{cases} \\ p = \sum_{k=1}^{K} p_k s_k \\ q = \sum_{k=1}^{K} q_k s_k \end{gathered} \tag{4.1.2}$$

El método ARMS[1] es capaz de simular cualquier distribución de probabilidad univariante dada una función proporcional a la función de densidad y el intervalo de dominio en el que se

[1] Ver[**ARMS**]

encuentra definida. Normalmente se utiliza el logaritmo de la densidad salvo una constante que ofrece la misma información con menos problemas numéricos.

4.2. Distribución condicionada de los valores iniciales

Para poder evaluar la ecuación en diferencias es preciso conocer p valores iniciales del ruido

$$z^0 = \{z_t\}_{t=1-p\ldots 0} \tag{4.2.1}$$

y q valores iniciales de los residuos

$$e^0 = \{e_t\}_{t=1-q\ldots 0} \quad \in \mathbb{R}^q \tag{4.2.2}$$

De esta forma se puede escribir la ecuación en diferencias en notación matricial

$$\begin{pmatrix} -\phi_p & \cdots & -\phi_1 & 1 & 0 & & & \cdots & & 0 \\ \vdots & \ddots & \vdots & -\phi_1 & \ddots & \ddots & & & & \vdots \\ 0 & \cdots & -\phi_p & \vdots & -\phi_1 & 1 & 0 & & & 0 \\ 0 & \cdots & 0 & -\phi_p & \cdots & -\phi_1 & 1 & 0 & & 0 \\ 0 & \cdots & 0 & 0 & -\phi_p & \cdots & -\phi_1 & 1 & \ddots & \vdots \\ \vdots & & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 & -\phi_p & \cdots & -\phi_1 & 1 \end{pmatrix} \cdot \begin{pmatrix} z_{1-p} \\ \vdots \\ z_0 \\ z_1 \\ \vdots \\ z_m \end{pmatrix}$$

$$= \tag{4.2.3}$$

$$\begin{pmatrix} -\theta_p & \cdots & -\theta_1 & 1 & 0 & & & \cdots & & 0 \\ \vdots & \ddots & \vdots & -\theta_1 & \ddots & \ddots & & & & \vdots \\ 0 & \cdots & -\theta_p & \vdots & -\theta_1 & 1 & 0 & & & 0 \\ 0 & \cdots & 0 & -\theta_p & \cdots & -\theta_1 & 1 & 0 & & 0 \\ 0 & \cdots & 0 & 0 & -\theta_p & \cdots & -\theta_1 & 1 & \ddots & \vdots \\ \vdots & & \vdots & \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & 0 & \cdots & 0 & -\theta_p & \cdots & -\theta_1 & 1 \end{pmatrix} \cdot \begin{pmatrix} e_{1-q} \\ \vdots \\ e_0 \\ z_1 \\ \vdots \\ z_m \end{pmatrix}$$

Dividiendo el sistema en bloques se puede escribir como

(4.2.4)

$$\begin{pmatrix} \Phi^0 & \Phi \end{pmatrix} \cdot \begin{pmatrix} z^0 \\ z \end{pmatrix} = \begin{pmatrix} \Theta^0 & \Theta \end{pmatrix} \cdot \begin{pmatrix} e^0 \\ e \end{pmatrix}$$

$$\Phi^0 = \begin{pmatrix} -\phi_p & \cdots & -\phi_1 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & -\phi_p \\ 0 & \cdots & 0 \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{pmatrix} \in \mathbb{R}^{m\times p} \wedge \Phi = \begin{pmatrix} 1 & 0 & & & \cdots & & 0 \\ -\phi_1 & \ddots & \ddots & & & & \vdots \\ \vdots & -\phi_1 & 1 & 0 & & & 0 \\ -\phi_p & \cdots & -\phi_1 & 1 & 0 & & 0 \\ 0 & -\phi_p & \cdots & -\phi_1 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & -\phi_p & \cdots & -\phi_1 & 1 \end{pmatrix} \in \mathbb{R}^{m\times m}$$

$$\Theta^0 = \begin{pmatrix} -\theta_p & \cdots & -\theta_1 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & -\theta_p \\ 0 & \cdots & 0 \\ 0 & \cdots & 0 \\ \vdots & & \vdots \\ 0 & \cdots & 0 \end{pmatrix} \in \mathbb{R}^{m\times q} \wedge \Theta = \begin{pmatrix} 1 & 0 & & & \cdots & & 0 \\ -\theta_1 & \ddots & \ddots & & & & \vdots \\ \vdots & -\theta_1 & 1 & 0 & & & 0 \\ -\theta_p & \cdots & -\theta_1 & 1 & 0 & & 0 \\ 0 & -\theta_p & \cdots & -\theta_1 & 1 & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & 0 \\ 0 & \cdots & 0 & -\theta_p & \cdots & -\theta_1 & 1 \end{pmatrix} \in \mathbb{R}^{m\times m}$$

$$z^0 \in \mathbb{R}^p \wedge z \in \mathbb{R}^p$$
$$e^0 \in \mathbb{R}^q \wedge e \in \mathbb{R}^q$$

o lo que es lo mismo

$$\Phi^0 \cdot z^0 + \Phi \cdot z = \Theta^0 \cdot e^0 + \Theta \cdot e \tag{4.2.5}$$

De la última fórmula se deduce la siguiente expresión matricial con distribución conocida

$$e = \Theta^{-1} \cdot \left(\Phi \cdot z + \Phi^0 \cdot z^0 - \Theta^0 \cdot e^0\right) \sim N\left(0, \sigma^2 I_m\right) \tag{4.2.6}$$

Si se define el vector conjunto de valores iniciales

$$u = \begin{pmatrix} z^0 \\ e^0 \end{pmatrix} \tag{4.2.7}$$

entonces, dados σ^2, ϕ, θ se tiene la siguiente regresión lineal

$$\begin{array}{c} \Pi \cdot z = G \cdot u + e \\ \Pi = \Theta^{-1}\Phi \qquad u = \left(z^0, e^0\right) \\ G = \Theta^{-1} \begin{pmatrix} -\Phi^0, & \Theta^0 \end{pmatrix} \end{array} \tag{4.2.8}$$

con lo que su distribución condicionada es

$$u \ \left|\phi, \theta, \sigma^2 \ \sim N\left(\tfrac{1}{\sigma^2}\left(G^T \cdot G\right)^{-1} \cdot G^T \cdot \Pi \cdot z, \ \tfrac{1}{\sigma^2}\left(G^T \cdot G\right)^{-1}\right)\right. \tag{4.2.9}$$

Nótese que las matrices $\Phi^0, \Theta^0, \Phi, \Theta$ son matrices de Toeplitz triangulares por lo que las operaciones de productos y resolución de sistemas lineales son desarrollables como ecuaciones en diferencias.

4.3. Distribución condicionada de los parámetros ARMA

Dados la varianza σ^2, los valores iniciales u y todos los parámetros ARMA salvo uno cualquiera de ellos, digamos ϑ, por el teorema de Bayes, el logaritmo de su densidad condicionada al resto de parámetros es proporcional a la de los residuos resultantes de aplicar la ecuación en diferencias para cada posible valor de dicho parámetro, que tomando logaritmos es

$$\lg f(\vartheta) = Cte_1 - \frac{m}{2}\lg 2\pi\sigma^2 - \frac{1}{2\sigma^2}e(\vartheta)^T e(\vartheta) = Cte - \frac{1}{2\sigma^2}e(\vartheta)^T e(\vartheta) \tag{4.3.1}$$

Dado un valor cualquiera de ϑ quedan completamente determinados los polinomios ϕ, θ; así que es posible calcular los residuos, para lo cual no es preciso recurrir a la fórmula del párrafo anterior, pues resulta evidente que es completamente equivalente a despejar los residuos en la ecuación en diferencias del modelo

$$e_t = z_t - \sum_{j=1}^{p}\phi_j z_{t-j} + \sum_{j=1}^{q}\theta_j e_{t-j} \tag{4.3.2}$$

que da un método de cálculo mucho más eficiente de los mismos.

Si existe información a priori o de tipo latente sobre todos o algunos de los parámetros ϑ, sólo es necesario poder escribirla en términos de una distribución normal univariante

$$\vartheta \sim N\left(\mu_\vartheta, \sigma_\vartheta^2\right) \tag{4.3.3}$$

En el caso de ser información a priori μ_ϑ será constante y si es latente será el valor actual del correspondiente hiperparámetro o combinación lineal de hiperparámetros. Esto se traduce a una simple ampliación de los residuos estandarizados

$$\lg f(\vartheta) = Cte - \frac{1}{2\sigma^2}e(\vartheta)^T e(\vartheta) - \frac{1}{2}\left(\frac{\vartheta - \mu_\vartheta}{\sigma_\vartheta}\right)^2 \tag{4.3.4}$$

4.3.1. Dominio de estacionariedad de monomios y binomios. Para poder aplicar el método ARMS es necesario además explicitar en qué dominio es evaluable la función de densidad, lo cual en este caso consiste en saber en qué intervalo es estacionario el factor polinomial al que pertenece el parámetro ϑ de turno. Basta recordar que un polinomio es estacionario si todas sus raíces están fuera del círculo unidad. En los casos de grado 1 y 2 se puede establecer esa condición de una forma analítica.

En un monomio de la forma $1 - \alpha x$ esto se cumple sí y sólo sí

$$\begin{array}{c} \alpha x - 1 = 0 \Longleftrightarrow x = \frac{1}{\alpha} \\ \|x\|_2 > 1 \Longleftrightarrow |\alpha| < 1 \end{array} \tag{4.3.5}$$

En un binomio de la forma $1 - \beta x - \alpha x^2$ esto se cumple sí y sólo sí

$$\begin{array}{c} \alpha x^2 + \beta x - 1 = 0 \Longleftrightarrow x = \frac{-\beta \pm \sqrt{\beta^2+4\alpha}}{2\alpha} \\ \|x\|_2 > 1 \Longleftrightarrow \left\| \frac{-\beta \pm \sqrt{\beta^2+4\alpha}}{2\alpha} \right\|_2 > 1 \end{array} \tag{4.3.6}$$

Si $4\alpha < \beta^2$ las raíces son complejas conjugadas y su módulo es el mismo para ambas

$$\begin{array}{c} \left\| \frac{-\beta \pm \mathbf{i}\sqrt{-(\beta^2+4\alpha)}}{2\alpha} \right\|_2^2 = \frac{|\beta|^2 + \left|\beta^2+4\alpha\right|}{4\alpha^2} = \frac{\beta^2 - \beta^2 - 4\alpha}{4\alpha^2} = \frac{-1}{\alpha} > 1 \Longleftrightarrow \alpha > -1 \\ -1 < \alpha < \frac{\beta^2}{4} \end{array} \tag{4.3.7}$$

Si $4\alpha > \beta^2$ las raíces son reales y su módulo puede ser distinto para cada una. Las fronteras de la región se completan resolviendo las ecuaciones

$$\begin{array}{c} \frac{-\beta \pm \sqrt{\beta^2+4\alpha}}{2\alpha} = 1 \Longleftrightarrow \alpha = 1 - \beta \\ \frac{-\beta \pm \sqrt{\beta^2+4\alpha}}{2\alpha} = -1 \Longleftrightarrow \alpha = 1 + \beta \end{array} \tag{4.3.8}$$

Las condiciones de estacionariedad son por tanto las siguientes (ver figura 4.3.1):

- Si $\vartheta = \alpha$ es un parámetro de un monomio $1 - \alpha x$ entonces la anterior fórmula es evaluable en $\alpha \in (-1, 1)$

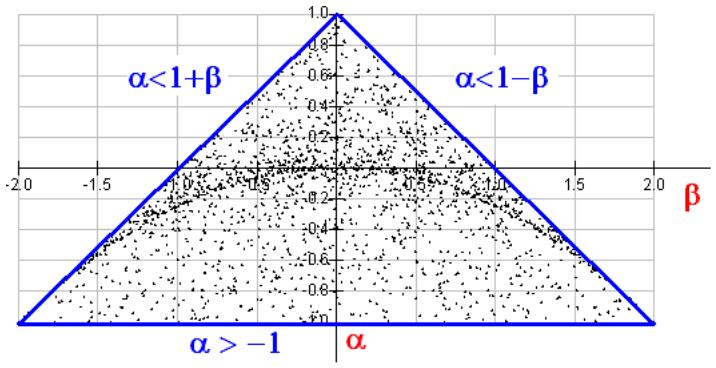

FIGURA 4.3.1.
Gráfico de simulaciones de polinomios estacionarios de grado 2. El sector parabólico inferior corresponde a las raíces complejas y el resto hacia arriba a las reales.

- Si $\vartheta = \alpha$ es el parámetro de orden 2 de un binomio $1 - \beta x - \alpha x^2$ entonces la región factible es en $\alpha \in (-1, 1 - |\beta|)$
- Si $\vartheta = \beta$ es el parámetro de orden 1 de un binomio $1 - \beta x - \alpha x^2$ entonces es estacionario en $\beta \in (\alpha - 1, 1 - \alpha)$

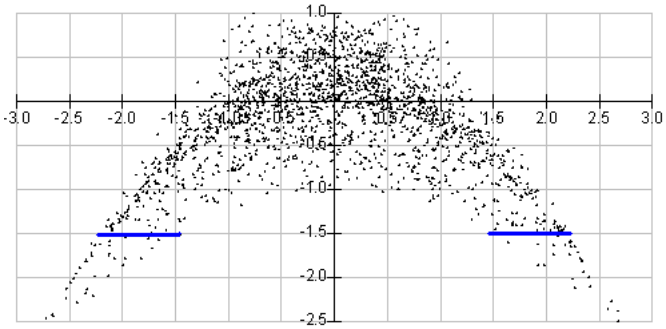

FIGURA 4.3.2.
Gráfico de simulaciones de polinomios estacionarios de grado 3. El dominio de estacionariedad no es una región conexa. Fijado el coeficiente de grado 2 el de grado 3 no está delimitado en un intervalo sino en la unión de dos intervalos inconexos y no se puede aplicar el método ARMS

La razón por la que se obliga a disponer de factores de grado máximo 2 es que la región de estacionariedad de un polinomio de grado 1 es simplemente el intervalo $[-1, 1]$ y para grado 2 es un triángulo, lo cual permite, una vez fijado uno de los parámetros obtener el intervalo en el que el otro permanece en la región de estacionariedad. Para grados superiores las regiones son complicadas de definir analíticamente por lo que resulta bastante complicado fijar el intervalo de estacionariedad de cada parámetro condicionado al resto (ver figura 4.3.2).

Por el teorema fundamental del álgebra todo polinomio de coeficientes reales es factorizable en monomios y binomios, por lo cual esta exigencia no supone ninguna restricción en la clase de modelos. Incuso es posible forzar la presencia de raíces reales imponiendo la factorización en binomios si es que existe alguna razón de peso para pensar que debe ser así.

4.4. Condicionamiento a los parametros ARMA

El modelo ARIMA con función de transferencia lineal se puede escribir así

$$\begin{aligned} \triangle \cdot Y &= \triangle \cdot X \cdot \beta + z \\ z &\sim N\left(0, \sigma^2 \Sigma\left(\phi, \theta, m\right)\right) \end{aligned} \tag{4.4.1}$$

donde

$$\Sigma(\phi,\theta,m)=\left\{\sigma_{i,j}(\phi,\theta)=\gamma_{i-j}(\phi,\theta)\right\}_{i,j=1\ldots m} \tag{4.4.2}$$

es la matriz de covarianzas de orden m de la serie estandarizada Z/σ, que depende exclusivamente de (ϕ,θ,m) y es la matriz de Toeplitz generada por las autocovarianzas del proceso ARMA

$$\left\{\gamma_k(\phi,\theta)=E\left[z_t,z_{t-k}\right]\right\}_{k=0,1,2,\ldots} \tag{4.4.3}$$

que son los coeficientes del polinomio de retardo infinito simétrico

$$\gamma(B+F)=\frac{\theta(B)\,\theta(F)}{\phi(B)\,\phi(F)} \tag{4.4.4}$$

Vista la matriz de covarianzas de una forma más gráfica quedaría así:

$$\Sigma(\phi,\theta,m)=\begin{pmatrix}\gamma_0 & \gamma_1 & \gamma_2 & \cdots & \gamma_{m-1}\\ \gamma_1 & \gamma_0 & \gamma_1 & \ddots & \vdots\\ \gamma_2 & \gamma_1 & \gamma_0 & \ddots & \gamma_2\\ \vdots & \ddots & \ddots & \ddots & \gamma_1\\ \gamma_{m-1} & \cdots & \gamma_2 & \gamma_1 & \gamma_0\end{pmatrix}\in\mathbb{R}^{m\times m} \tag{4.4.5}$$

Si se dispone de una descomposición simétrica como la de Cholesky

$$\Sigma(\phi,\theta,m)=L(\phi,\theta,m)\cdot L^T(\phi,\theta,m) \tag{4.4.6}$$

ésta se podrá emplear para filtrar las matrices de input y output de la regresión lineal, resolviendo sistemas lineales en $L(\phi,\theta,m)$

$$\begin{aligned}Y^{'}&=L^{-1}(\phi,\theta,m)\cdot\triangle\cdot Y\\ X^{'}&=L^{-1}(\phi,\theta,m)\cdot\triangle\cdot X\\ e&=L^{-1}(\phi,\theta,m)\,Z\end{aligned} \tag{4.4.7}$$

resultando la regresión lineal

$$\begin{aligned}Y^{'}&=X^{'}\cdot\beta+e\\ e&\sim N\left(0,\sigma^2 I\right)\end{aligned} \tag{4.4.8}$$

que permite realizar la simulación de los parámetros lineales y la varianza condicionados a los parámetros ARMA.

Efectuar la descomposición de Cholesky genérica tendría complejidad cúbica en el orden de la matriz $O\left(m^3\right)$ y aunque existen diferentes métodos de calcular la descomposición de Cholesky con complejidades superlineales $O\left(m\cdot\lg^2 m\right)$, luego la matriz $L(\phi,\theta,m)$ no tiene ninguna estructura especial y la resolución de los sistemas referidos tiene complejidad cuadrática $O\left(m^2\right)$, lo cual supone tiempos de cálculo insoportables con apenas unos cientos de datos.

A continuación se presenta un método implícito de descomposición sintética mucho más rápido de obtener los mismos resultados.

4.4.1. Descomposición sintética.

Lema 1. Dada la descomposición de Cholesky de la matriz de covarianzas de orden m del proceso AR puro

$$\Sigma(\phi,1,m)=L(\phi,1,m)\cdot L^T(\phi,1,m) \tag{4.4.9}$$

la inversa de dicha descomposición $L^{-1}(\phi,1,m)$ para cualquier $m>n=\max(p,q)+1+p$ es de Toeplitz a partir de la fila n por lo que es conocida sin coste adicional alguno sin más que repetir

en cada fila de $L^{-1}(\phi,1,n)$ los valores de la fila n desplazados una columna hacia delante:

$$L^{-1}(\phi,1,m)=\begin{pmatrix}\vartheta_{1,1} & \cdots & 0 & \cdots & 0\\ \vdots & \ddots & \vdots & \ddots & \vdots\\ \vartheta_{n,1} & \cdots & \vartheta_{n,n} & & 0\\ \vdots & \ddots & & \ddots & \vdots\\ 0 & \cdots & \vartheta_{n,1} & \cdots & \vartheta_{n,n}\end{pmatrix}\in\mathbb{R}^{m\times m} \tag{4.4.10}$$

Eso la convierte además en una matriz de Toeplitz de bandas con ceros fuera de las n primeras bandas que es muy sparse cuando crece m pues el número de datos no ceros es menor que $n\cdot m$. Nótese que la matriz de descomposición $L(\phi,1,m)$ propiamente dicha no es sparse en general, aunque para raíces muy lejanas del círculo unidad los términos más alejados de la unidad se acercan rápidamente a cero.

Lema 2. La matriz de covarianzas de orden $m>n$ del proceso ARMA filtrada simétricamente de la descomposición de las covarianzas del proceso AR puro, es decir,

$$\Upsilon(\phi,\theta,m)=L^{-1}(\phi,1,m)\cdot\Sigma(\phi,\theta,m)\cdot L^{-T}(\phi,1,m) \tag{4.4.11}$$

también es simétrica y de Toeplitz partir de la fila y columna n por lo que es conocida sin coste adicional alguno, una vez calculada $\Upsilon(\phi,\theta,m)$, sin más que repetir en cada fila (y columna) los valores de la fila (y columna) n desplazados una columna (fila) hacia delante (abajo). Se trata por ende de una matriz de Toeplitz de bandas con ceros fuera de las n primeras bandas que es muy sparse cuando crece m pues el número de datos no ceros es menor que $(2\cdot n-1)\cdot m$:

$$\Upsilon(\phi,\theta,m)=\begin{pmatrix}\upsilon_{1,1} & \cdots & \upsilon_{n,1} & \cdots & 0\\ \vdots & \ddots & \vdots & \ddots & \vdots\\ \upsilon_{n,1} & \cdots & \upsilon_{n,n} & & \upsilon_{n,1}\\ \vdots & \ddots & & \ddots & \vdots\\ 0 & \cdots & \upsilon_{n,1} & \cdots & \upsilon_{n,n}\end{pmatrix}\in\mathbb{R}^{m\times m} \tag{4.4.12}$$

Por construcción del método de Cholesky $\Upsilon(\phi,\theta,m)=\Lambda(\phi,\theta,m)\cdot\Lambda^{T}(\phi,\theta,m)$ se tiene que el factor $\Lambda(\phi,\theta,m)$ es igualmente de bandas con ceros fuera de las n primeras bandas, siendo el número de datos no ceros menor que $n\cdot m$:

$$\Lambda(\phi,\theta,m)=\begin{pmatrix}\lambda_{1,1} & \cdots & 0 & \cdots & 0\\ \vdots & \ddots & \vdots & \ddots & \vdots\\ \lambda_{n,1} & \cdots & \lambda_{n,n} & & 0\\ \vdots & \ddots & & \ddots & \vdots\\ 0 & \cdots & \lambda_{n,1} & \cdots & \lambda_{n,n}\end{pmatrix}\in\mathbb{R}^{m\times m} \tag{4.4.13}$$

Corolario 1. Mezclando las dos fórmulas anteriores

$$\Upsilon(\phi,\theta,m)=\Lambda(\phi,\theta,m)\cdot\Lambda^{T}(\phi,\theta,m)=L^{-1}(\phi,1,m)\cdot\Sigma(\phi,\theta,m)\cdot L^{-T}(\phi,1,m) \tag{4.4.14}$$

$$L(\phi,1,m)\cdot\Lambda(\phi,\theta,m)\cdot\Lambda^{T}(\phi,\theta,m)\cdot L^{T}(\phi,1,m)=\Sigma(\phi,\theta,m)=L(\phi,\theta,M)\cdot L^{T}(\phi,\theta,M) \tag{4.4.15}$$

se obtiene de forma inmediata una expresión de la descomposición de Cholesky de la matriz de covarianzas para cualquier entero m

$$L(\phi,\theta,m)=L(\phi,1,m)\cdot\Lambda(\phi,\theta,m) \tag{4.4.16}$$

que, aunque no es en general sparse.

Corolario 2. Sin embargo sí que permite un mecanismo de filtrado sparse en dos fases. En el caso del filtrado ARMA comentado anteriormente se tiene que calcular V dado U de forma que

$$U = L(\phi, \theta, m) \cdot V \tag{4.4.17}$$

Para ello se calcula primero

$$W = L^{-1}(\phi, 1, m) \cdot U = \Lambda(\phi, \theta, m) \cdot V \tag{4.4.18}$$

teniendo en cuenta que $L^{-1}(\phi, 1, m)$ es sparse, para luego despejar V en el sistema sparse

$$W = \Lambda(\phi, \theta, m) \cdot V \tag{4.4.19}$$

Corolario 3. En un proceso AR puro es $\Lambda(\phi, \theta, m) = I_m$ y $L^{-1}(\phi, \theta, m) = L^{-1}(\phi, 1, m)$
Corolario 4. En un proceso MA puro es $L^{-1}(\phi, 1, m) = I_m$ y $L(\phi, \theta, m) = \Lambda(\phi, \theta, m)$

4.4.2. Algoritmo de la descomposición ARMA sintética.

Entrada. Los polinomiosϕ, θ y el tamaño de la serie m
Salida. Dos matrices sparse $L^{-1}(\phi, 1, m)$ y $\Lambda(\phi, \theta, m)$
Método.

1. Se calcula $n = \text{máx}(p, q) + 1 + p$
2. Se calcula $\Sigma(\phi, 1, n)$
3. Se calcula la descomposición de Cholesky de orden n tal que $\Sigma(\phi, 1, n) = L(\phi, 1, n) \cdot L^T(\phi, 1, n)$ que será en general más densa cuanto más cerca de la unidad estén las raíces ϕ, siendo la complejidad máxima de este paso $O\left(n^3\right)$
4. Se calcula la inversa de dicha descomposición $L^{-1}(\phi, 1, n)$ con coste $O\left(n^3\right)$ a lo peor.
5. Si $m > n$ entonces se expande la última fila de $L^{-1}(\phi, 1, n)$ otras $m - n$ veces para formar la matriz sparse $L^{-1}(\phi, 1, m)$. De lo contrario se extrae como el menor principal de orden m. El número de celdas no nulas a almacenar es como máximo $n \cdot m$
6. Se calcula $\Sigma(\phi, \theta, n)$
7. Se calcula $\Upsilon(\phi, \theta, n) = L^{-1}(\phi, 1, n) \cdot \Sigma(\phi, \theta, n) \cdot L^{-T}(\phi, 1, n)$ lo cual conlleva un coste menor o igual que $O\left(n^3\right)$
8. Si $m > n$ entonces se expanden las últimas fila y columna de $\Upsilon(\phi, \theta, n)$ otras $m - n$ veces para formar la matriz sparse simétrica $\Upsilon(\phi, \theta, m)$. De lo contrario se extrae como el menor principal de orden m
9. Se calcula la descomposición de Cholesky de orden m de la matriz sparse $\Upsilon(\phi, \theta, m) = \Lambda(\phi, \theta, m) \cdot \Lambda^T(\phi, \theta, m)$ con complejidad $O\left(n^2 m\right)$y tamaño a almacenar de $n \cdot m$ celdas como máximo

4.4.3. Algoritmo de filtrado ARMA sintético.

Entrada. Las matrices sparse $L^{-1}(\phi, 1, m)$ y $\Lambda(\phi, \theta, m)$ generadas por el algoritmo anterior
La matriz o vector U a filtrar, que puede ser densa o sparse.
Salida. La matriz filtrada $V = \Lambda(\phi, \theta, m) \cdot L^{-1}(\phi, 1, m) \cdot Z$
Método.

1. Se calcula el producto de matriz sparse por matriz o vector $W = L^{-1}(\phi, 1, m) \cdot Z$
2. Se resuelve el sistema triangular sparse $W = \Lambda(\phi, \theta, m) \cdot V$

CAPíTULO 5

El bloque de omitidos

En la simulación de Gibbs de una regresión lineal generalizada con datos omitidos en el input y en el output, si se condiciona al resto de parámetros, se obtiene otra regresión lineal cuyos parámetros son los valores omitidos.

5.1. Descripción

Sea el modelo de regresión lineal con omitidos

$$\begin{gathered} \triangle \cdot (Y + \delta^v \cdot v - (X + \delta^u \cdot u) \cdot \beta) = a \\ a \sim N(0, \Sigma) \\ a \in \mathbb{R}^m \\ Y, V = \delta^v \cdot v \in \mathbb{R}^M \\ \nabla \in \mathbb{R}^{m \times M} \wedge M \geqslant m \\ \beta \in \mathbb{R}^n \wedge X, U = \delta^u \cdot u \in \mathbb{R}^{M \times n} \\ v \in \mathbb{R}^{K_0}, u \in \mathbb{R}^K \end{gathered} \tag{5.1.1}$$

- β son los parámetros del bloque lineal de la regresión,
- n es el número de parámetros a estimar de la regresión,
- Y es la matriz de output del modelo y es completamente conocida, aunque puede cambiar de una simulación para otra si estamos en el marco de una simulación de Gibbs con más bloques involucrados. Donde es desconocida se coloca un cero.
- La matriz booleana $\delta^v \in \{0,1\}^{M \times K_0}$ es el operador lineal de ubicación que transforma el vector de parámetros v en el vector de outputs omitidos V ubicados en los índices adecuados.
- M es el número de datos del output
- X es la matriz de iputs del modelo y es completamente conocida y de rango completo, aunque puede cambiar de una simulación para otra lo mismo que el output. Donde es desconocida se coloca un cero.
- El tensor tridimensional booleano $\delta^u \in \{0,1\}^{M \times n \times K}$ es el operador lineal de ubicación que transforma el vector de parámetros u en la matriz de inputs omitidos U ubicados en los índices adecuados.
- m es el número de datos de contraste del modelo, o número de residuos, y ha de ser mayor que el de variables,
- La matriz $\triangle$ es el operador lineal de filtrado determinista, normalmente definido como la matriz de Toeplitz correspondiente a un polinomio de raíces unitarias. Este operador puede ser degenerado, cuando es $M > m$
- a es el ruido del modelo cuya distribución se propone como hipótesis principal del mismo,
- Σ es la matriz de covarianzas, usualmente será o bien una matriz diagonal o bien la matriz de autocovarianzas de un proceso ARMA

Al conjunto de índices en los que se dan interrupciones en el output se le llamará

$$\mathcal{I}^0 = \left\{ t_1^0, \ldots, t_{K_0}^0 \right\} \tag{5.1.2}$$

La matriz de datos conocidos de output Y valen 0 en las interrupciones y las matriz de datos desconocidos de output V vale 0 fuera de las interrupciones y un valor a estimar dentro

$$\begin{array}{c} Y_t = 0 \forall t \in \mathcal{I}^0 \\ V_t = 0 \forall t \notin \mathcal{I}^0 \\ V_{t_k^0} = v_k \forall k = 1 \dots K_0 \\ v = (v_1 \dots v_{K_0})^T \in \mathbb{R}^{K_0} \end{array} \tag{5.1.3}$$

De forma análoga para cada input $j = 1 \dots n$ se define el conjunto de índices en los que se dan interrupciones

$$\mathcal{I}^j = \left\{ t_1^j, \dots, t_{K_j}^j \right\} \tag{5.1.4}$$

La matriz de datos conocidos de input X valen 0 en las interrupciones y las matriz de datos desconocidos de output U vale 0 fuera de las interrupciones y un valor a estimar dentro

$$\begin{array}{c} X_{j,t} = 0 \forall t \in \mathcal{I}^j \\ U_{t,j} = 0 \forall t \notin \mathcal{I}^j \\ U_{t_k^j,j} = w_k^j \forall k = 1 \dots K_j \end{array} \tag{5.1.5}$$

Si se concatenan los vectores w^j se obtiene un único vector de omitidos de inputs

$$\begin{array}{c} u^T = \left(w_1^{1T} \dots w_{K_1}^{1T}, w_1^{2T} \dots w_{K_2}^{2T}, ..., w_1^{nT} \dots w_{K_n}^{nT} \right) \in \mathbb{R}^K \\ K = \sum_{j=1}^n K_j \end{array} \tag{5.1.6}$$

Esto es equivalente a establecer la función de indexación $S(j,k)$ tal que

$$u_{S(j,k)} = w_k^j \tag{5.1.7}$$

Es obvio que condicionando a los vectores de los bloques de omitidos u, v queda una regresión lineal. Veremos a continuación que esta relación es recíproca.

5.2. Distribución condicionada de los omitidos del output

Si se conocen $\beta, \sigma^2, \Sigma, U$ y una descomposición simétrica $\Sigma = L \cdot L^T$ entonces, premultiplicando por L^{-1} y llevando al lado izquierdo todo lo conocido, a lo cual llamaremos D, se tiene que

$$\begin{array}{c} D = \dot{L}^{-1} \cdot (Y - (X + U) \cdot \beta) = -\dot{L}^{-1} \cdot V + \varepsilon \\ \dot{L}^{-1} = L^{-1} \cdot \triangle \\ \varepsilon = L^{-1} \cdot a \sim N(0, I_m) \end{array} \tag{5.2.1}$$

Extrayendo de $-L^{-1}$ las columnas correspondientes a las interrupciones en una matriz H resulta una regresión lineal en los parámetros de output desconocidos v

$$\begin{array}{c} D = H \cdot v + \varepsilon \\ \varepsilon \sim N(0, I_m) \\ H_{t,k} = -\dot{L}^{-1}_{t,t_k^0} \forall k = 1 \dots K_0 \end{array} \tag{5.2.2}$$

Si existe algún conocimiento a priori, sobre los valores que pueden adoptar los omitidos, y éste es expresable como una multinormal truncada,

$$\begin{array}{c} v \sim N(\mu_v, \Sigma_v) \\ A_v \cdot v \leq a_v \end{array} \tag{5.2.3}$$

y se dispone de una descomposición simétrica $\Sigma_v^{-1} = L_v \cdot L_v^T$, entonces no hay más que añadir a la regresión las ecuaciones e inecuaciones pertinentes

$$\begin{array}{c} D = H \cdot v + \varepsilon \\ L_v \cdot \mu_v = L_v \cdot v + \varepsilon_v \\ A_v \cdot v \leq a_v \\ \varepsilon \sim N(0, I_m) \\ \varepsilon_v \sim N(0, I_{K_0}) \end{array} \tag{5.2.4}$$

En cualquier caso, definiendo

$$\breve{D} = \begin{pmatrix} D \\ L_v \cdot \mu_v \end{pmatrix} \tag{5.2.5}$$

$$\breve{H} = \begin{pmatrix} H, & L_v \end{pmatrix} \tag{5.2.6}$$

$$\breve{\varepsilon} = \begin{pmatrix} \varepsilon \\ \varepsilon_v \end{pmatrix} \tag{5.2.7}$$

se tiene una regresión lineal estandarizada con restricciones

$$\begin{array}{c} \breve{D} = \breve{H} \cdot v + \breve{\varepsilon} \\ A_v \cdot v \leq a_v \\ \breve{\varepsilon} \sim N\left(0, I_{m+K_0}\right) \end{array} \tag{5.2.8}$$

que se puede tratar como se explicaba en el capítulo 2.1

5.3. Distribución condicionada de los omitidos del input

Si se conocen $\beta, \sigma^2, \Sigma, V$ y una descomposición simétrica $\Sigma = L \cdot L^T$ entonces, premultiplicando por L^{-1} y llevando al lado izquierdo todo lo conocido, a lo cual llamaremos C, se tiene que

$$\begin{array}{c} C = \dot{L}^{-1} \cdot (Y + V - X \cdot \beta) = \dot{L}^{-1} \cdot U \cdot \beta + \varepsilon \\ \dot{L}^{-1} = L^{-1} \cdot \nabla\triangle \\ \varepsilon = L^{-1} \cdot a \sim N\left(0, I_m\right) \end{array} \tag{5.3.1}$$

El producto de matrices $L^{-1} \cdot U \cdot \beta$ se puede expresar alternativamente como otro producto $G \cdot u$ en el que sólo interviene el vector de inputs omitidos

$$\begin{array}{c} \sum_{t=1}^{m} \sum_{i=1}^{m} \dot{L}_{t,i}^{-1} U_{i,j} \beta_j = \sum_{t=1}^{m} \sum_{j=1}^{n} \sum_{k=1}^{K_j} \dot{L}_{t,t_k^j}^{-1} \beta_j w_k^j = \sum_{t=1}^{m} \sum_{s=1}^{K} G_{t,s} u_s \\ G_{t,s} = L_{t,t_k^j}^{-1} \beta_j \wedge s = S\left(j,k\right) \\ G \in \mathbb{R}^{M \times K} \wedge w \in \mathbb{R}^K \end{array} \tag{5.3.2}$$

Resulta pues una regresión lineal en los parámetros de input desconocidos u

$$\begin{array}{c} C = G \cdot u + \varepsilon \\ \varepsilon \sim N\left(0, I_m\right) \end{array} \tag{5.3.3}$$

Si existe algún conocimiento a priori, sobre los valores que pueden adoptar los omitidos, y éste es expresable como una multinormal truncada,

$$\begin{array}{c} v \sim N\left(\mu_u, \Sigma_u\right) \\ A_u \cdot u \leq a_u \end{array} \tag{5.3.4}$$

y se dispone de una descomposición simétrica $\Sigma_u^{-1} = L_u \cdot L_u^T$, entonces no hay más que añadir a la regresión las ecuaciones e inecuaciones pertinentes

$$\begin{array}{c} C = G \cdot u + \varepsilon \\ L_u \cdot \mu_u = L_u \cdot u + \varepsilon_u \\ A_u \cdot u \leq a_u \\ \varepsilon \sim N\left(0, I_m\right) \\ \varepsilon_u \sim N\left(0, I_K\right) \end{array} \tag{5.3.5}$$

En cualquier caso, definiendo

$$\breve{C} = \begin{pmatrix} C \\ L_u \cdot \mu_u \end{pmatrix} \tag{5.3.6}$$

$$\breve{G} = \begin{pmatrix} G, & L_u \end{pmatrix} \tag{5.3.7}$$

$$\breve{\varepsilon} = \begin{pmatrix} \varepsilon \\ \varepsilon_u \end{pmatrix} \tag{5.3.8}$$

se tiene una regresión lineal estandarizada con restricciones

$$\begin{gathered} \breve{C} = \breve{G} \cdot u + \breve{\varepsilon} \\ A_u \cdot u \leq a_u \\ \breve{\varepsilon} \sim N\left(0, I_{m+K}\right) \end{gathered} \tag{5.3.9}$$

que se puede tratar como se explicaba en el capítulo 2.1

5.4. Distribución condicionada conjjunta de los omitidos del input y el output

Lo anterior es válido bajo el supuesto de que los conjuntos de omitidos del input y el output son disjuntos, pues de lo contrario sería imposible formular el condicionamiento mutuo de ambos bloques y no habría otro remedio que simularlos conjuntamente en un vector $w \in \mathbb{R}^{K_0+K-K'}$ que englobara los parámetros u y v, sin repetición de los K' comunes, como es lógico.

En tal caso, si se conocen β, σ^2, Σ y una descomposición simétrica $\Sigma = L \cdot L^T$ entonces, premultiplicando por L^{-1} y llevando al lado izquierdo todo lo conocido, a lo cual llamaremos C, se tiene que

$$\begin{gathered} E = \dot{L}^{-1} \cdot (Y - X \cdot \beta) = \dot{L}^{-1} \cdot (V - U \cdot \beta) + \varepsilon \\ \dot{L}^{-1} = L^{-1} \cdot \nabla\triangle \\ \varepsilon = L^{-1} \cdot a \sim N\left(0, I_m\right) \end{gathered} \tag{5.4.1}$$

Siguiendo los razonamientos de los puntos anteriores, existirá una matriz J tal que

$$V - U \cdot \beta = J \cdot w \tag{5.4.2}$$

de forma que podríamos escribir todo lo anterior como una sola regresión lineal

$$\begin{gathered} E = J \cdot w + \varepsilon \\ L_w \cdot \mu_w = L_w \cdot w + \varepsilon_w \\ A_w \cdot w \leq a_w \\ \varepsilon \sim N\left(0, I_m\right) \\ \varepsilon_w \sim N\left(0, I_{K_0+K-K'}\right) \end{gathered} \tag{5.4.3}$$

CAPíTULO 6

El bloque de filtro no lineal

Muy a menudo se observan comportamientos de inputs que no actúan de forma lineal en el filtrado de fenómenos complejos y de outputs que hay que transformar de forma no lineal para obtener relaciones lineales con los algunos inputs concretos. En ambos casos se trata de modificar bien un output o bien uno o varios inputs en el modelo de regresión lineal, y en ambos es posible formalizar un método genérico ARMS de simulación de Gibbs de cada parámetro condicionado al resto. Se expone también el caso concreto de la función de transferencia no lineal (deltas).

6.1. Descripción

Sea el modelo de regresión con partes no lineales

$$\begin{gathered} \triangle \cdot \left(Y - F_0 - X \cdot \beta - \sum_{r=1}^{R} F_r \cdot \alpha_r \right) + a \\ a \sim N(0, \Sigma) \\ Y, a \in \mathbb{R}^m \\ \beta \in \mathbb{R}^n \wedge X \in \mathbb{R}^{m \times n} \\ F_r = f_r(\tau_r) \in \mathbb{R}^{m \times h_r} \\ \alpha_r \in \mathbb{R}^{h_r} \\ \tau_r \in \mathbb{R}^{R_r} \\ F_0 = f_0(\tau_0) \in \mathbb{R}^m \\ \tau_0 \in \mathbb{R}^{R_0} \end{gathered} \tag{6.1.1}$$

donde

- β son los parámetros del bloque lineal original de la regresión,
- n es el número de parámetros a estimar de la regresión,
- M es el número de datos del output,
- m es el número de datos de contraste del modelo y ha de ser mayor que el de variables,
- X es la matriz de iputs estrictamente lineales del modelo y es completamente conocida y de rango completo.
- La matriz $\triangle$ es el operador lineal de filtrado determinista, normalmente definido como la matriz de Toeplitz correspondiente a un polinomio de raíces unitarias. Este operador puede ser degenerado, cuando es $M > m$ (ver 4.1)
- a son los residuos del modelo cuya distribución se propone como hipótesis principal del mismo,
- Σ es la matriz de covarianzas, usualmente será o bien una matriz diagonal o bien la matriz de autocovarianzas de un proceso ARMA (ver 4.1)
- τ_0 y τ_r son los vectores de parámetros de los filtros no lineales del output y el input respectivamente y f_0 y f_r sus correspondientes funciones de filtrado no lineal
- F_0 es el vector de filtro de output generado por el filtro no lineal.
- F_r es la r-ésima matriz de o inputs filtrados.
- α_r son ampliaciones de los parámetros del bloque lineal para los nuevos input,

En el caso de covarianzas diagonales o ARMA, este modelo es una extensión del modelo correspondiente a un segmento de regresión de la clase BSR, ya que condicionando a los parámetros no lineales τ, es obvio que resulta una regresión lineal expresable por bloques

$$\triangle \cdot Y = Z = \triangle \cdot F_0 + \triangle \cdot \begin{pmatrix} X & F_1 & \cdots & F_R \end{pmatrix} \begin{pmatrix} \beta \\ \alpha_1 \\ \vdots \\ \alpha_R \end{pmatrix} + a \tag{6.1.2}$$

Cada filtro no lineal aditivo del input viene dado por una función real arbitraria f_r que devuelve una matriz de efectos para un vector de parámetros dados dentro de un dominio convexo y cerrado Ω_r^f

$$F_r = f_r(\tau_r) : \Omega_r^f \subseteq \mathbb{R}^{R_r} \to \mathbb{R}^{M \times h_r} \tag{6.1.3}$$

Estos filtros tienen asociado un vector de parámetros lineales α_r que serán estimados como el de un input lineal una vez fijados los parámetros no lineales τ_r.

El filtro no lineal del output viene dado por una función real arbitraria f_0 que devuelve un vector para un vector de parámetros dados dentro de un dominio convexo y cerrado Ω_0^f que se restará al output original Y

$$F_0 = f_0(\tau_0) : \Omega_0^f \times \mathbb{R}^M \subseteq \mathbb{R}^{R_0} \times \mathbb{R}^M \to \mathbb{R}^M \tag{6.1.4}$$

Nótese que la función no lineal de filtro del input f_r podría hacerse cargo de los parámetros α_r y añadirse este efecto al filtro no lineal del output, por lo que en apariencia no aportan nada dichos α_r y podría definirse el modelo usando sólo el filtro del output sin pérdida de generalidad. La diferencia está en el método de cálculo de las simulaciones usado internamente. El filtro del output tiene la ventaja de que es más rápido en ejecución, especialmente si X y Σ son fijos o existe algún método rápido de descomposición para ambas, pues en tal caso no es necesario recalcular las desde cero en cada simulación. Sin embargo pueden dar problemas si alguno de los parámetros del filtro no lineal tiene alta correlación con otros lineales o no, pues en ese caso la cadena de Markov puede resultar demasiado autocorrelada y precisar de longitudes mayores para hacer thinin. Por esta razón se ofrecen ambas posibilidades de definición para aplicar en cada caso la más adecuada, o cualquier mezcla de ambas.

6.2. Distribución condicionada de los parámetros no lineales

Si se conocen todos los vectores de parámetros no lineales salvo un τ_k del input y los α, β, Σ y una descomposición simétrica $\Sigma = L \cdot L^T$ entonces, premultiplicando por L^{-1} y llevando todo al lado derecho, salvo los residuos cuya distribución es conocida, se tiene la siguiente igualdad

$$\begin{aligned} \varepsilon &= L^{-1} \cdot \triangle \cdot (Z_k - f_k(\tau_k) \cdot \alpha_k) \\ Z_k &= -F_0 - X \cdot \beta - \sum_{\substack{r=1 \\ r \neq k}}^{R} F_r \cdot \alpha_r \\ \varepsilon &= -L^{-1} \cdot a \sim N(0, I_m) \end{aligned} \tag{6.2.1}$$

Si fijamos ahora todos los componentes de $\tau_k = (\tau_{k,1}, \ldots, \tau_{k,R_k})$ salvo cierto $\tau_{k,i} \wedge i \in \{1, \ldots, R_k\}$ entonces es posible aplicar el método ARMS si se dispone de un método que permita construir el dominio de $\tau_{k,i}$ condicionado por los $\{\tau_{k,j}\}_{j \neq i}$, para lo que es preciso disponer de dos funciones tales que dicho dominio se exprese como el intervalo cerrado

$$\omega_{k,i}^f(\tau_k) \leq \tau_{k,i} \leq \Omega_{k,i}^f(\tau_k) \tag{6.2.2}$$

Nótese que al exigirse convexidad y cierre al dominio genérico Ω_r^f del vector τ_k, el dominio de cada componente es necesariamente un intervalo cerrado, por ser la intersección de aquel con un hiperplano.

Por el teorema de Bayes, el logaritmo de la densidad de $\tau_{k,i}$ condicionada al resto de parámetros es proporcional a la de los residuos resultantes de aplicar la ecuación en diferencias para cada posible valor de dicho parámetro, que tomando logaritmos es

$$\lg h\left(\tau_{k,i}\right) = Cte_1 - \frac{m}{2}\lg 2\pi - \frac{1}{2}\varepsilon^T\varepsilon = Cte - \frac{1}{2}\varepsilon^T\varepsilon \tag{6.2.3}$$

Si existe información a priori o de tipo latente sobre todos o algunos de los parámetros no lineales, sólo es necesario poder escribirla en términos de una distribución normal univariante

$$\tau_{k,i} \sim N\left(\mu_{\tau_{k,i}}, \sigma^2_{\tau_{k,i}}\right) \tag{6.2.4}$$

En el caso de ser información a priori $\mu_{\tau_{k,i}}$ será constante y si es latente será el valor actual del correspondiente hiperparámetro o combinación lineal de hiperparámetros. Esto se traduce a una simple ampliación de los residuos estandarizados

$$\lg h\left(\tau_{k,i}\right) = Cte - \frac{1}{2}\varepsilon^T\varepsilon - \frac{1}{2}\left(\frac{\tau_{k,i} - \mu_{\tau_{k,i}}}{\sigma_{\tau_{k,i}}}\right)^2 \tag{6.2.5}$$

Para el filtro no lineal del output el proceso de simulación es prácticamente el mismo: si se conocen todos los vectores de parámetros no lineales salvo el del output τ_0 del input y los α, β, Σ y una descomposición simétrica $\Sigma = L \cdot L^T$ entonces, premultiplicando por L^{-1} y llevando todo al lado derecho, salvo los residuos cuya distribución es conocida, se tiene la siguiente igualdad

$$\begin{gathered} \varepsilon = L^{-1} \cdot \triangle \cdot \left(Z_0 - f_0\left(\tau_0\right)\right) \\ Z_0 = X \cdot \beta + \sum_{r=1}^{R} F_r \cdot \alpha_r \\ \varepsilon = L^{-1} \cdot a \sim N\left(0, I_m\right) \end{gathered} \tag{6.2.6}$$

A partir de ahí es exactamente igual que con el input por lo que no se explicitan las fórmulas.

Hay que hacer notar que en este esquema los resultados de los filtros no lineales no pueden tener omitidos. Si los datos originales a partir de los cuales se construye, y que sólo el propio filtro conoce, tuviera omitidos, entonces el propio filtro debería encargarse de su simulación. Una posible solución de carácter general sería que los valores desconocidos fueran simplemente una variable más que se simula con ARMS igual que los demás. Eso sería muy sencillo de tratar pero probablemente no sería muy eficiente si hay muchos omitidos.

6.3. Casos particulares

6.3.1. Función de transferencia no lineal (deltas). Sea el modelo de series temporales con función de transferencia no lineal

$$\begin{gathered} y_t = \frac{\omega(B)}{\delta(B)} x_t + a_t \\ a \sim N\left(0, \Sigma\right) \\ t = 1 \ldots m \end{gathered} \tag{6.3.1}$$

donde los polinomios implicados son de la forma

$$\begin{aligned} \omega\left(B\right) &= \sum_{i=0}^{h} \omega_i B^i \\ \delta\left(B\right) &= 1 - \sum_{j=1}^{d} \delta_j B^j \end{aligned} \tag{6.3.2}$$

siendo $\delta\left(B\right)$ estacionario, es decir, tal que sus raíces están todas fuera del círculo unidad.

Esto es expresable dentro de la clase de modelos descritos anteriormente

$$\begin{array}{c} Y = f\left(\delta, z^0, x^0\right) \cdot \omega + a \\ a \sim N\left(0, \Sigma\right) \\ Y, a \in \mathbb{R}^m \\ F = f\left(\delta, z^0, x^0\right) \in \mathbb{R}^{m \times h} \end{array} \tag{6.3.3}$$

donde la función de filtro no lineal se forma a partir del resultado de la ecuación en diferencias

$$\begin{array}{c} z_t = \sum_{j=1}^{d} \delta_j z_{t-j} + x_t \\ F_{t,i} = z_{t-i} \forall i = 1 \ldots h \end{array} \tag{6.3.4}$$

Los valores inciales $x^0 = (x_{1-h}, \ldots, x_0)$ del input original x_t pueden ser conocidos o no, en cuyo caso se incluirán como variables a simular, lo mismo que si existen otros valores desconocidos en cualquier otro punto de x_t se deberían adjuntar con estos valores iniciales. Los valores iniciales $z^0 = (z_{1-k}, \ldots, z_0)$, con $k = \text{máx}\left(h, d\right)$, son siempre desconocidos y deben ser simulados. Lo más sencillo sería considerarlos todos ellos variables auxiliares no lineales sin más y que sea el propio método ARMS el que los simule. Para todos estos parámetros el dominio es siempre la recta real, aunque es recomendable dar un intervalo lo más acotado que sea posible con la información de que se disponga, así como se debe tener en cuenta la información a priori sobre los mismos.

Para los parámetros δ rigen las mismas condiciones de estacionariedad descritas en 4.3.1 en la página 28 para los parámetros AR y MA por lo que es necesario imponer la misma restricción sobre el grado que no puede ser mayor que 2. Si fuera preciso usar un grado mayor habría que factorizar $\delta\left(B\right)$ de forma similar a la allí descrita, sin ninguna pérdida de generalidad.

6.3.2. Filtrado en términos originales. Ocurre a veces que la mayor parte de los efectos que inciden sobre una serie output ocurren bajo cierta transformación que produce además residuos más homocedásticos que en términos originales. Muy a menudo se trata de la transformación logarítmica, es decir, se tiene una mayoría de efectos multiplicativos en términos originales, sin que eso sea obstáculo para que pueda existir algún que otro efecto aditivo. Este modelo de regresión no lineal se escribiría así

$$\begin{array}{c} Log\left(Y - X^{\alpha} \cdot \alpha\right) = X^{\beta} \cdot \beta + a \\ a \sim N\left(0, \Sigma\right) \end{array} \tag{6.3.5}$$

Despejando la serie transformada

$$\begin{array}{c} Log\left(Y\right) - Log\left(Y\right) + Log\left(Y - X^{\alpha} \cdot \alpha\right) = X^{\beta} \cdot \beta + a \\ Log\left(Y\right) = Log\left(Y\right) - Log\left(Y - X^{\alpha} \cdot \alpha\right) + X^{\beta} \cdot \beta + a \end{array} \tag{6.3.6}$$

lo cual se puede escribir dentro de la clase expuesta mediante un filtro no lineal de output

$$f_0\left(\alpha\right) = Log\left(Y\right) - Log\left(Y + X^{\alpha} \cdot \alpha\right) \tag{6.3.7}$$

6.3.3. Modelo probit. Con una pequeña extensión en la definición de los filtros, que permita sustituir las funciones f_0 por generadores de variables aleatorias, con o sin parámetros, es posible simular modelos cualitativos como el probit. En estos modelos el output original es una variable cualitativa booleana, es decir, $Y_t \in \{0, 1\}$. Para ello se definirá como filtro del output el generador aleatorio condicionado al bloque lineal β

$$f_{0,t} \leftarrow \begin{cases} Y_t - TN\left(X \cdot \beta, 1, f_{0,t} \leq X \cdot \beta\right) & \forall Y_t = 0 \\ Y_t - TN\left(X \cdot \beta, 1, X \cdot \beta \leq f_{0,t}\right) & \forall Y_t = 1 \end{cases} \tag{6.3.8}$$

Una vez generado este filtro de output se tiene de nuevo un modelo de regresión lineal perfectamente asumible por BSR.

$$\begin{aligned} Y &= f_0 + X \cdot \beta + \epsilon \\ \epsilon &\sim N(0, I) \end{aligned} \tag{6.3.9}$$

En este caso no hay ningún parámetro que simular, si no que se simula directamente el output, por lo que evidentemente no sería necesario incluir los métodos de obtención de intervalos. Matemáticamente no tiene sentido decir que f_0 sea una función, pues no depende de ningún parámetro, pero no habría ningún problema en pensar que existe un parámetro τ_0 irrelevante que luego nos abstenemos de simular. Informáticamente se tratará sin ningún problema como un caso particular de dimensión del vector de parámetros igual 0.

CAPíTULO 7

Diagnosis

Cuando ya se dispone de una cadena de Markov para un modelo comienza la etapa de diagnosis que certifique la calidad de los resultados obtenidos e informe de la potencia alcanzable por las ulteriores inferencias que se puedan extraer de él.

7.1. Convergencia de la cadena de Markov

Lo primero que hay que comprobar antes de poder hacer ningún otro tipo de inferencia es que la cadena de Markov recién construida ha convergido a un punto La diagnosis de convergencia debe cerciorarse de que la etapa de simulaciones desechadas (burning) ha sido en efecto suficiente para asegurar la convergencia del paseo aleatorio hacia la distribución a posteriori. Se debe comprobar también que la secuencia simulada para cada parámetros no esté pobremente mezclada (poorly mixing) y que no exista una alta correlación en dicha secuencia. El paquete CODA de R dispone de métodos de diagnosis de convergencia como los de Geweke, Gelman & Rubin, Raftery & Lewis y otros que se describen en [**CODA**]. Los tests más confiables son los que, como el de Gelman & Rubin, toman dos o más cadenas creadas a partir de distintos valores iniciales, pero son también los más complicados de utilizar y pueden suponer un coste de simulación extra muy alto al multiplicar el burning así como una compleja logística informática de procesamiento. De los tests de una sola cadena el más completo es el de Raftery & Lewis pues da estimaciones bastante aceptables del burning y el thinning.

7.2. Significación de los parámetros

Un parámetro es necesario o significativo en un modelo si realmente el modelo con dicho parámetro es mejor que sin él. Hay por tanto tantas formas de definir la significación como de comparar la calidad de modelos. Por otra parte no parece muy eficiente estimar todos los modelos con y sin cada uno de los parámetros. La medida que se utiliza para medir esta significación es la probabilidad de que el parámetro sea cero. Cuando la distribución del parámetro es aproximadamente normal esta probabilidad se puede medir como el p-value del clásico test de significación de la t-Student para una muestra. En el caso de BSR se tendrán parámetros cuya distribución marginal a posteriori sea normal o normal truncada para los parámetros lineales y de omitidos, chi cuadradas para la varianza y otras distribuciones de las que ni siquiera se tiene una representación analítica, como los parámetros ARMA y los de los filtros no lineales. Por esta razón se precisa un test de significación más general como es el One Sample Sign Test `http://www.alglib.net/statistics/hypothesistesting/signtest.php` que es el de mayor potencia dentro de los tests no paramétricos válidos para cualquier distribución [**One Sample Wilcoxon Test**].

7.3. Alta correlación múltiple entre parámetros

Si en la cadena de Markov se resta la media de las realizaciones de cada columna y se multiplica su traspuesta por sí misma se obtiene la matriz de covarianzas muestrales de los parámetros. Si la longitud de la cadena es mucho mayor que el número de variables se tendrá una estimación insesgada de la matriz de covarianzas poblacional (ver `http://en.wikipedia.org/wiki/Estimation_of_covariance_matrices`). Dividiendo filas y columnas por su correspondiente elemento diagonal se obtiene la matriz de correlaciones muestrales y calculando su descomposición de Jordan se puede comprobar si hay autovalores cercanos a la unidad. Se toma el autovector v correspondiente

al mínimo autovalor λ y se pretende contrastar la hipótesis de que es nulo, $\lambda = 0$, es decir que existe colinealidad entre las variables Υ

$$v^T \cdot \Upsilon = \lambda \cdot \Upsilon = 0 \tag{7.3.1}$$

Si se toma una componente no nula cualquiera $v_k \neq 0$, la ecuación anterior es equivalente a

$$\sum_{j \neq k} v_j \cdot \Upsilon_j = -v_k \cdot \Upsilon_k \tag{7.3.2}$$

Para contrastar la hipótesis así expresada se puede usar el test no paramétrico como el Mann-Whitney U-test `http://www.alglib.net/statistics/hypothesistesting/mannwhitneyu.php`

Si el p-value es mayor que cierto umbral se procede a repetir el test para el siguiente autovalor para detectar otras posibles colinealidades.

7.4. Normalidad e independencia de los residuos

La hipótesis fundamental del modelo es que los residuos estandarizados de cada segmento de regresión tienen distribución multinormal estándar independiente. Como los residuos no son conocidos sino que se deben calcular para cualquier posible combinación de parámetros lo que se hará es hacer las comprobaciones para los residuos correspondientes a los valores medios de los parámetros en la cadena de Markov. Para contrastar la normalidad se puede usar un test como el Jarque-Bera test `http://www.alglib.net/statistics/hypothesistesting/jarqueberatest.php`. Para la independencia el test más completo y usado es el Ljung-Box test `http://en.wikipedia.org/wiki/Ljung-Box_test`

Detector de outliers.

Detector de heterocedasticidad.

7.5. Polinomios ARMA no estacionarios

Dada una realización de los coeficientes de un polinomio AR ó MA de grado máximo 2, es trivial calcular el módulo de sus raíces reales o complejas. Si se repite el proceso para cada fila de la cadena de Markov simulada previamente entonces se obtienen cadenas de Markov de los módulos de las raíces y se puede aplicar el anteriormente nombrado One Sample Sign Test `http://www.alglib.net/statistics/hypothesistesting/signtest.php` para contrastar si es igual a 1.

CAPíTULO 8

Mantenimiento y producción

Una vez que acaba la etapa de análisis y diseño de un modelo se pasa a la etapa de producción en la que el modelo se usará para hacer inferencia de una u otra forma: predicción, decisión, etc. Durante esta etapa el modelo debe mantenerse en estado de revista, es decir, debe ir adaptándose a los nuevos datos que vayan conociéndose e incorporando eventuales nuevas variables.

8.1. Reestimación

Durante la etapa de mantenimiento de un modelo se dispone siempre de una cadena de Markov correspondiente a la última sesión de simulación de las variables del modelo. Periódicamente, se añadirá un intervalo de nuevos datos observados de input y output, sin que desaparezcan variables ni se modifique la formulación del modelo, salvo quizás por la incorporación de alguna variable nueva que trate de explicar algún acontecimiento del que no había precedentes en la muestra. Entonces es lícito suponer que las variables que no tienen datos en ese nuevo intervalo ni correlacionan con otras que sí los tengan, no deberían cambiar nada con respecto a las ya simuladas. Es decir, esta modificación supone dividir las variables del modelo en dos categorías:

- las variables consideradas como vigentes que no dependen de los nuevos datos incorporados, y
- las variables susceptibles de haber quedado obsoletas.

Si es sostenible la hipótesis de que la distribución de las variables vigentes condicionadas por las obsoletas no se puede ver afectada por la incorporación de los nuevos datos y variables, entonces no es preciso simular de nuevo dichas variables, sino que basta con leerlas directamente de la cadena de Markov previamente almacenada. En tal caso sólo será preciso volver a simular las variables obsoletas condicionadas a las vigentes recién leídas. No se podría por tanto ampliar la longitud de la nueva cadena de Markov.

8.2. Simulación parcial

Tanto si se ha fijado un valor constante para un parámetro como si se leen sus valores de una cadena previamente almacenada, los pasos a seguir a partir de ahí son básicamente los mismos dependiendo del bloque concreto al que pertenece el parámetro. A esta técnica le llamaremos simulación parcial y acontinuación se detallan algunas notas sobre su implementación en cada caso.

8.2.1. El bloque lineal. Si llamamos β^0 a los parámetros que no es necesario simular pues ya se dispone de un valor, fijo o leído de una MCMC previa, y β^1 a los que hay que simular, entonces tendremos

$$\begin{gathered} Y = X^0 \cdot \beta^0 + X^1 \cdot \beta^1 + \varepsilon \\ C^0 \cdot \beta^0 + C^1 \cdot \beta^1 \leq c \\ \varepsilon \sim N(0, I) \\ Y, \varepsilon \in \mathbb{R}^m \\ \beta^0 \in \mathbb{R}^{n_0} \wedge X^0 \in \mathbb{R}^{m \times n_0} \\ \beta^1 \in \mathbb{R}^{n_1} \wedge X^1 \in \mathbb{R}^{m \times n_1} \\ c \in \mathbb{R}^{\rho} \wedge C^0 \in \mathbb{R}^{\rho \times n_0} \wedge C^1 \in \mathbb{R}^{\rho \times n_1} \\ m > n = n_0 + n_1 \\ rank(X) = n \end{gathered} \tag{8.2.1}$$

Puesto que β^0 es conocido definiendo

$$\begin{aligned} Y^1 &= Y - X^0 \cdot \beta^0 \\ c^1 &= c - C^0 \cdot \beta^0 \end{aligned} \tag{8.2.2}$$

se tendrá la regresión con restricciones

$$\begin{aligned} Y^1 &= X^1 \cdot \beta^1 + \varepsilon \\ C^1 \cdot \beta^1 &\leq c^1 \\ \varepsilon &\sim N(0, I) \end{aligned} \tag{8.2.3}$$

Nótese que en la nuevas matrices de restricciones C^1y c^1 habría que eliminar todas las filas que no tengan coeficientes no nulos para los β^1.

En el caso particular en que todos los parámetros del bloque lineal estuvieran fijados, es decir, $n = n_0$, o lo que es lo mismo, $n_1 = 0$, no habría nada que simular en este bloque lineal, por lo que bastaría con filtrar el output que sería por sí mismo el vector de ruido estandarizado del modelo: $Y^1 = Y - X^0 \cdot \beta^0 = \varepsilon$

8.2.2. El bloque de omitidos. Los omitidos se simulan en realidad como un bloque lineal, por lo que la solución es idéntica al caso anterior

8.2.3. El bloque de varianzas. Como se simulan de una en una de forma independiente no ofrece ninguna dificultad, simplemente se simulan las que hay que simular y las otras no.

8.2.4. El bloque ARMA. Internamente se simulan usando un ARMS, aunque en un futuro es posible que se usen otros métodos más eficientes, especialmente para la parte AR, por lo que sería recomendable no permitir la fijación de cada parámetro AR ó MA en particular, sino de toda la parte ARMA de un segmento concreto. Eso sí, unos segmentos podrían tener la parte ARMA fija y otros no.

En cualquier caso la parte de filtrado ARIMA del resto de bloques habría que seguir manteniéndola en cualquier caso.

8.2.5. Los bloques de filtros no lineales. Cabe decir lo mismo que en el apartado del bloque ARMA.

8.3. Remodelación

Cada cierto tiempo se acumulan demasiados cambios eventuales que deben reformularse de una forma más genérica, o bien ocurre que los nuevos datos contradicen las hipótesis que antes funcionaban correctamente, bien por error en el diseño del modelo, o bien porque el propio fenómeno está cambiando y evolucionando, lo cual es muy habitual por ejemplo en los nuevos mercados, pero que es inherente a casi cualquier proceso complejo.

En estos casos ya no es posible reestimar el modelo viejo con las nuevas incorporaciones sino que hay que reestimar todas conjuntamente, aunque sí es posible a veces introducir como información a priori una aproximación relajada de la distribución a posteriori de la vieja simulación. En un modelo jerárquico esto es equivalente a sustituir la jerarquía de nodos latentes por un solo nivel a priori que permite la simulación en paralelo de cada nodo observacional como un modelo independiente de los demás, lo cual proporciona un mecanismo de escalabilidad sumamente eficiente.

8.4. Previsión bayesiana

Lo primero que hay que dejar claro es que en el sentido bayesiano, dar una previsión de una variable aleatoria no consiste en dar un número específico; ni la media, ni la moda ni ningún otro estadístico concreto, si no una formulación de la distribución de probabilidad de dicha variable. En algunas clases de modelos esa distribución se puede concretar en base a ciertos parámetros, por ejemplo si es una normal basta con la media y la varianza, pero el método no paramétrico más empleado es dar una muestra suficientemente larga de simulaciones, lo cual en nuestro caso se concreta en una cadena de Markov-Montecarlo (MCMC).

Si en un modelo en el que algunos segmentos de regresión corresponden a series temporales, incluimos en el intervalo muestral las fechas para las que se quiere estimar la previsión introduciendo omitidos para los outputs, pues lógicamente son desconocidos, por eso se quieren prever, entonces la cadena de Markov correspondiente a esos omitidos tras la simulación del modelo nos da un conjunto de realizaciones de la distribución conjunta a posteriori de las previsiones correspondientes. Esto es tanto como decir que la previsión es sólo un caso particular de simulación de datos omitidos referidos al futuro.

Si se conjuga este concepto con el anterior de reestimación (8.1) entonces se puede ver que una forma rápida de construir las previsiones de un modelo ya estimado cuando aparecen nuevos datos input y/o output es considerar como vigentes todas las variables salvo los omitidos del output correspondientes al periodo de previsión. El proceso de reestimación así definido tendría como resultado una cadena de Markov que contendría la distribución conjunta de las previsiones en el periodo especificado. En esta redefinición del modelo original sólo sería necesario introducir los últimos datos de cada segmento observacional necesarios para la evaluación ARIMA.

Por supuesto que se puede considerar obsoleta alguna variable más, lo cual queda al albur del analista que debe decidirlo en función de la magnitud de los cambios que se aprecien en los datos recién llegados, y de este modo se obtiene la reestimación y la previsión en un sólo paso de simulación parcial.

8.5. Inferencia y decisión bayesianas

Una vez obtenida la MCMC $\{\Upsilon_s\}_{s=1\ldots S}$ de la distribución a posteriori de la previsión Υ, hacer inferencia bayesiana es algo básicamente trivial, lo cual no sucede por casualidad, sino porque ese es precisamente el objetivo de generar muestras de cualquier tipo.

- Para obtener un estadístico de tipo esperanza basta con calcular su promedio a lo largo de las muestras:

$$E\left[f\left(\Upsilon\right)\right]=\int_{\Omega}f\left(\Upsilon\right)\mathrm{d}\Upsilon=\frac{1}{S}\sum_{s=1}^{S}f\left(\Upsilon_s\right)\pm O\left(\frac{1}{\sqrt{S}}\right) \tag{8.5.1}$$

- Para la media, por ejemplo, sería como es obvio

$$\overline{\Upsilon}=\frac{1}{S}\sum_{s=1}^{S}\Upsilon_s \tag{8.5.2}$$

- Una decisión bayesiana basada en una función de coste es también un caso particular de estadístico esperanza. Una función de coste muy habitual en los sistemas de atención dinámica de la demanda, y que nos servirá perfectamente de ilustración del método, es la función de coste asimétrico de servicio

$$C\left(\widetilde{\Upsilon},\Upsilon\right)=\begin{cases}a\left(\Upsilon-\widetilde{\Upsilon}\right) & \forall\Upsilon>\widetilde{\Upsilon}\\ b+c\left(\widetilde{\Upsilon}-\Upsilon\right) & \forall\Upsilon\leq\widetilde{\Upsilon}\end{cases} \tag{8.5.3}$$

que devuelve el coste de tomar la decisión $\widetilde{\Upsilon}$ y que el valor real sea finalmente Υ teniendo en cuenta que si sobran unidades, cada una de ellas con lleva un coste $a>0$, el llamado coste unitario de devolución, mientras que si faltan unidades puede haber un coste de oportunidad $b\geq 0$ independiente de cuantas se hayan dejado de servir, además del coste $c>0$ de no haber vendido cada unidad que faltó. La decisión óptima desde el punto de vista bayesiano sería la que minimiza la esperanza del coste total

$$E\left[C\left(\widetilde{\Upsilon},\Upsilon\right)\right]\simeq\overline{C\left(\widetilde{\Upsilon},\Upsilon\right)}=\frac{1}{S}\sum_{s=1}^{S}C\left(\widetilde{\Upsilon},\Upsilon_s\right) \tag{8.5.4}$$

Para encontrarla, siempre que se trate de una distribución unimodal, se puede utilizar el método de la bisectriz o el de Fibonacci evaluando $\overline{C\left(\widetilde{\Upsilon},\Upsilon\right)}$ dentro del

intervalo $\widetilde{\Upsilon} \in \left[\min_{s=1\ldots S} \Upsilon_s, \max_{s=1\ldots S} \Upsilon_s\right]$. Si no es unimodal existen métodos más complicados aunque es una situación realmente extraña por lo que no merece la pena extenderse en ello.

- Para obtener un estadístico de tipo cuantil simplemente se calcula dicho cuantil sobre la muestra.
  - Para la mediana, por ejemplo, se ordena la cadena de menor a menor

$$\Upsilon_{s_1} \leq \Upsilon_{s_2} \leq \ldots \leq \Upsilon_{s_S} \tag{8.5.5}$$

  y se toma el valor

$$Q_{1/2} = \begin{cases} \Upsilon_{s_{S/2}} & \forall s \bmod 2 = 1 \\ \frac{\Upsilon_{s_{(S-1)/2}} + \Upsilon_{s_{(S+1)/2}}}{2} & \forall s \bmod 2 = 0 \end{cases} \tag{8.5.6}$$

  - Para dar un intervalo simétrico de confianza $1-\alpha$ se calculan los cuantiles muestrales $Q_{\alpha/2}$ y $Q_{1-\alpha/2}$.
- Para estimar la moda o valor más probable, siempre que se trate de una distribución unimodal, claro está, se debe utilizar algún método de estimación de la función de densidad, como el kernel-density `http://en.wikipedia.org/wiki/Kernel_density_estimation`, y luego aplicar un método de optimización adecuado para encontrar su máximo.

Parte 2

Guía del programador de BSR

Esta parte interesa a todos los desarrolladores de BSR y a todos aquellos que deseen comprender cómo funciona BSR por dentro desde el punto de vista informático.así como a los usuarios de BSR que se enfrenten a problemas complejos que no se pueden representar dentro del clásico modelo jerárquico lineal, bien porque se tengan estructuras reticulares, porque haya componentes lineales o por cualquier otra causa.

Aquí se describe la metodología de la implementación, tanto en C++ como en TOL y SQL de todo lo relacionado con el sistema BSR desde los algoritmos matemáticos a las API de definición de modelos; y también cómo sacarle el máximo partido a BSR utilizando la API virtual de definición de modelos así como la incorporación de las extensiones no lineales y cualquier otra modificación del modelo que sea representable como un bloque de Gibbs, independientemente del método de simulación interno al bloque, que puede ser Metropolis-Hastings o cualquier otro.

Para llegar a una comprensión adecuada se debe haber entendido previamente la Parte I de este libro. Estos son los requisitos previos sobre TOL necesarios para comprender sin dificultad la implementación presentada en esta parte:

- Nivel avanzado en las características de TOL hasta la versión 1.1.6
- Ciertos conocimientos mínimos sobre las nuevas características de la nueva versión de TOL 1.1.7, la primera sobre la que funciona BSR
  - NameBlock: `http://www.tol-project.org/es/node/40`
  - VMatrix: `http://www.tol-project.org/es/node/90`

Estos son los resúmenes de cada capítulo:

1. BysMcmc: Se describen los miembros y métodos generales del NameBlock BysMcmc cuyo propósito es aglutinar todas las funcionalidades relacionadas con la inferencia bayesiana basada en métodos de simulación MCMC (MonteCarlo Markov Chain)
2. BysMcmc::Bsr::Gibbs: Se describen los miembros y métodos generales del **NameBlock BysMcmc::Bsr::Gibbs** que implementa los métodos de simulación propiamente dichos de la clase de modelos de BSR y sus extenciones.

CAPíTULO 9

BysMcmc

Se describen los miembros y métodos generales del NameBlock BysMcmc cuyo propósito es aglutinar todas las funcionalidades relacionadas con la inferencia bayesiana basada en métodos de simulación MCMC (MonteCarlo Markov Chain)

9.1. Descripción

El fichero principal de este apartado es `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/_bysMcmc.tol`, en el cual se define el **NameBlock BysMcmc** que incluye métodos genéricos para el manejo de cadenas de Markov independientemente del método concreto utilizado para generarlas e incluso de la clase de modelos que se simulan. Se encarga de la definición de la parte común a todos los bloques de variables con sus correspondientes métodos de acceso y actualización, así como de la logística del proceso de llamadas al simulador concreto que se esté usando, el almacenamiento de la cadena en caché RAM y su escritura periódica en disco y sus correspondientes mensajes de estado de proceso. También incluye funciones de diagnosis sobre la convergencia de la cadena de Markov y de inferencia sobre la misma, pero sólo de aquellos métodos completamente genéricos, es decir, que son independientes de la clase de modelos concreta y de todo tipo de semántica asociada a las variables.

Por último, incluye el simulador la clase de modelos BSR y los de sus extensiones, que por el momento son las únicas con soporte en BysMcmc, aunque en el futuro podrá haber otras clases de modelos estándar que cuelguen de aquí. Por supuesto que el analista puede crear su propio simulador para una clase concreta de modelos y llamar a los métodos de BysMcmc sin tener que incluirlo dentro de este NameBlock.

9.2. Opciones y configuración

El miembro **NameBlock Options** centraliza todas las opciones de los valores que pueden adoptar los parámetros cualitativos de configuración de los distintos componentes de BysMcmc

```
////////////////////////////////////////////////////////////////////////////
//Configuration default values over all BysMcmc related components
////////////////////////////////////////////////////////////////////////////
  //Options used over all BysMcmc related components
  NameBlock Options =
  [[
    //Scalar samplers draws a univariant distribution given the logarithm of
    //density function and a domain interval
    //The API of a scalar sampler  is given by a function like this
    //  Real (Code logDens, Real x0, Real x1)
    NameBlock Scalar.Sampler =
    [[
      Real Arms(Code logDens, Real x0, Real x1)
      {
        ARMS(logDens, x0, x1)
      };
      Real Slice(Code logDens, Real x0, Real x1)
      {
```

```
        MatDat(SliceSampler1D(logDens, x0, x1),1,1)
      }
    ]]
  ]];
////////////////////////////////////////////////////////////////////////////
```

El miembro **NameBlock Config** centraliza todos los parámetros de configuración de los distintos componentes de BysMcmc que contiene los valores por defecto de cada uno de los parámetros de configuración.

```
////////////////////////////////////////////////////////////////////////////
//Configuration default values over all BysMcmc related components
////////////////////////////////////////////////////////////////////////////
  Set _config =
  [[
  //MCMC dimensions
    //The final number realizations that will remain after burnin
    Real mcmc.sampleLength = 2000;
    //The frequency of simulation notifying and chain cache updating
    Real mcmc.cacheLength = 100;
    //The number of first realizations that will be skiped at inference and
    //diagnostic time
    Real mcmc.burnin = 100;
    //Only one of each this number of simulations will be selected for
    //inference and diagnostic
    Real mcmc.thinning = 1; //thinningg
    //Maximum of time in seconds dedicated to simulation
    //Process will be stoped after mcmc.sampleLength + mcmc.burnin
    //simulations or mcmc.maxTime seconds will be elapsed
    Real mcmc.maxTime = 1/0;
  //MLS step configuration
    Text mls.method  = Options::Mls.Method::None;
    Real mls.maxIter = 100;
    Real mls.tolerance = 0.01;
    Real mls.FunnelGibbs.sigmaFactor = 0.01;
  //Basic master configuration
    //If solution error is great than this value a warning will be shown
    Real bsr.cholesky.epsilon = 1.E-13;
    //The notifying frequency of Cholesky warning
    Real bsr.cholesky.warningFreq = 100;
    //The number of burnin realizations in Gibbs drawing of truncated
    //multinormal
    Real bsr.truncMNormal.gibbsNumIter = 0;
    //Scalar sampler used in ARMA block of basic BSR can be a member of
    //BysMcmc::Options::Scalar.Sampler or an API equivalent function
    Code bsr.scalarSampler.armaBlock = Options::Scalar.Sampler::Slice;
    //Scalar sampler used in ARMA block of basic BSR can be a member of
    //BysMcmc::Options::Scalar.Sampler or an API equivalent function
    Code bsr.scalarSampler.nonLinBlock = Options::Scalar.Sampler::Slice;
    //Arima filter method
    Text bsr.arimaFilter = Options::Arima.Filter::FastChol;
  //CODA and other diagnostic report configuration
    //The quantile to be estimated in Raftery test
    Real report.raftery.diag.q = 0.025;
    //The desired margin of error of the estimate in Raftery test
```

```
    Real report.raftery.diag.r = 0.007;
    //The probability of obtaining an estimate in the interval (q-r,q+r) in
    //Raftery test
    Real report.raftery.diag.s = 0.950;
    //Precision required for estimate of time to convergence in Raftery test
    Real report.raftery.diag.eps = 0.001;
    //Lag of AutoCorrelation Function
    Real report.acf.lag= 20;
    //Number of divisions in histograms
    Real report.histogram.parts = 100;
    //Number of divisions in kernel density aproximations
    Real report.kerDens.points = 0;
  //Generic flags
    //If positive then is the number of existent realizations to be recovered
    //from disk
    //If negative then is the number of last existent realization to be
    //skyped
    //If zero then no resuming will be made and a new Markov chain will be
    //generated
    Real do.resume = 0;
    //Enables or disables CODA and other diagnostic reports
    Real do.report = True;
    //Enables or disables in-sample evaluation of forecasting, filter, noise,
    //residuals, ...
    Real do.eval = True;
    //Enables or disables in-sample evaluation of individual linear effects
    Real do.linear.effects = True
  ]];
////////////////////////////////////////////////////////////////////////////
```

El método **GetCfgVal** busca un parámetro en el NameBlock de configuración del usuario y, si no lo encuentra, devuelve esl valor por defecto definido en BysMcmc::Config

```
////////////////////////////////////////////////////////////////////////////
//Searches a parameter in the user defined configuration NameBlock and, if
//not exists returns default value of BsyMcmc::Config
////////////////////////////////////////////////////////////////////////////
Anything GetCfgVal(NameBlock config, Text member)
{
  StdLib::GetConfigValue(config,member,Eval("BysMcmc::Config::"+member))
}
////////////////////////////////////////////////////////////////////////////
```

9.3. Método NameBlock BysMcmc::BuildCycler()

En `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/_build.tol` se define este método fundamental de la clase usada para obtener el constructor y manejador de la cadena de Markov, el cual se ocupa de la logística de almacenamiento y trazado del proceso, de forma independiente del algoritmo de simulación, siendo el propio BSR un caso particular, el cual será el argumento principal (**NameBlock modelSampler**) de este método, debiendo contener la siguiente API obligatoria:

```
////////////////////////////////////////////////////////////////////////////
//Identificador del modelo
Text get.name (Real unused)
//Identificador de la sesión de trabajo
```

```
Text get.session (Real unused)
//Camino donde se almacena la cadena de Markov en formato binario BBM
Text get.path (Real unused)
//Nombres de los parámetros de la cadena de Markov
Set get.colNames (Real unused)
//Simula una realización de la cadena de Markov
Matrix draw (Real numSim)
//Evalúa los componentes (filtro, ruido, residuos, previsión, ...) de cada
//segmento de la regresión para un vector concreto de parámetros.
//Este método no es necesario para la simulación pero sí es recomendable
//para su posterior diagnosis e inferencia.
Set eval(Matrix values)
////////////////////////////////////////////////////////////////////////////
```

El argumento del método es **NameBlock config** centraliza todos los parámetros de configuración de los distintos componentes, es decir, los de BysMcmc::Config vistos en la sección anterior. Sólo es preciso incluir aquellos que se quieran cambiar, aunque no hay ningún problema en explicitar los valores por defecto. Usualmente se deseará modificar al menos los parámetros de configuración de la longitud de la cadena, por ejemplo así

```
////////////////////////////////////////////////////////////////////////////
NameBlock my_config =
[[
  Real mcmc.burnin        =  500;
  Real mcmc.sampleLength  = 5000
]];
////////////////////////////////////////////////////////////////////////////
```

El último argumento del método es **NameBlock notifier**, el cual se utiliza como canal especializado de notificaciones de eventos relacionados con la, como puede ser una base de datos (Ver **NameBlock dbNotifier** en la función **BysMcmc::Bsr::DynHlm::DBApi::Estim** de `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_db_api.estim.tol`) , una barra de progreso en una interfaz gráfica, o cualquier otro sistema especializado que cumpla la API obligatoria:

```
////////////////////////////////////////////////////////////////////////////
//Notifica la inicialización de la cadena de Markov,
//informando del número de variables y el path de almacenamiento
Real initializing(Real numVar, Text path)
//Notifica que se está simulando la realización
//numSim de la cadena de Markov
Real simulating(Real numSim)
////////////////////////////////////////////////////////////////////////////
```

El NameBlock devuelto por `BysMcmc::BuildCycler` cuenta con una serie de métodos y miembros que permiten crear y manejar con comodidad las cadenas de Markov.

El método **Real generate(Real sampleLength)** genera la cadena de Markov y la almacena temporalmente en una matriz en memoria que se irá añadiendo, con llamadas a la función built-in **MatAppendFile,** al fichero binario permanente cada cierto número de simulaciones dado por el valor del campo **Real cacheLength_**, cuyo valor por defecto es 100 pero que el usuario puede cambiar a su antojo. Este campo también define la frecuencia con que ocurren las notificaciones de la simulación en curso. Normalmente el número de simulaciones que se le pasará será **mcmc.burnin+mcmc.sampleLength*mcmc.thinning**. Si se supera el tiempo máximo **mcmc.maxTime**, entonces se finalizará la simulación aunque no se hayan realizado todas las muestras requeridas.

Mediante el miembro **Real resume_=config::do.resume**, que por defecto está deshabilitado, es posible especificar que se reutilicen total o parcialmente las realizaciones de una cadena

de Markov previamente simulada y almacenada en disco, en un archivo con el mismo camino actual, para no perder el trabajo realizado tras cualquier circunstancia que aborte la simulación. Si es positivo se utilizarán las especificadas o bien las exietntes si hay menos. Si es negativo se eliminarán las últimas almacenadas utilizando sólo las restantes. Si es cero no se reutiliza nada y se comienza una nueva cadena de Markov.

El método **Matrix loadMcmc(Real burnin, Real length, Real thinin)** recarga un total de **length** simulaciones de una cadena de Markov previamente almacenada descartando las **burnin** primeras simulaciones y extrayendo del resto una de cada **thinin**. Esto permite hacer inferencia con diferentes submuestras para contrastar los resultados.

El método **Set report(Real burnin, Real length, Real thinin)** devuelve un informe de diagnosis e inferencia básica (ver 9.6) sobre la subcadena construida según se indica en el párrafo anterior. El miembro **bysInf_** permite modificar la configuración de dicho informe y tiene la estructura **BysInf.Report.Config** que se puede ver en `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/_structs.tol` y que se comentarán más adelante en9.6.

Antes de comenzar a simular se almacena en un fichero ***.recover.oza** la información necesaria para poder hacer informes parciales de la cadena MCMC sin necesidad de esperar a que acabe la simulación. Este fichero se recarga con el método **BysMcmc::Get.Recover** o bien se puee llamar directamente a **BysMcmc::RecoveAndReport** para que lo recupere y lance el informe con **report**.

9.4. Método NameBlock BysMcmc::DefineBlock()

En `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/_defineBlock.tol` se define este método que crea el manejador de la información común a cada bloque de variables de una MCMC, como el conjunto indexado de los nombres de los parámetros, que deben ser únicos, para poder acceder por nombre a los mismos; una matriz para almacenar los valores en curso de los parámetros; las posiciones de ubicación de los mismos dentro de la cadena MCMC completa; un flag para habilitar o deshabilitar el bloque, etc.

Se trata de un método de uso interno del sistema por lo que el usuario final no necesita acceder nunca a este miembro.

El programador de un nuevo simulador debería fijarse en cómo se usa en el caso de los bloques de BSR para crear sus propios bloques de simulación.

9.5. Miembro BysMcmc::_.cmsg

En el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/_cmsgs.tol` se define este miembro que incluye los mensajes de error empleados por los métodos propios y los de los NameBlock hijos que lo precisen. Es de uso interno del sistema por lo que el usuario no necesita acceder nunca a este miembro. Se hace uso del paquete **CMsg::Coded** implementado en `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/general/system/cmsg/_coded.tol`. Los mensajes codificados como **"Stop"** se entienden como errores irrecuperables del proceso de compilación TOL que se estuviera llevando a cabo, y provocarán la parada inmediata del mismo mediante la orden **Real Stop**, para no perder más tiempo, pues ya es seguro que no se va a llegar a ninguna parte, e incluso es posible que de esta forma se proteja al ejecutable de una caída incontrolada.

9.6. Método BysMcmc::Inference

En `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/_inference.tol` se define este miembro que incluye los métodos para la diagnosis e inferencia bayesianas a partir de cadenas de Markov previamente simuladas definidos en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/_inference_functions.tol`. Todos los métodos se llaman internamente salvo el método **report,** que les llama al resto y es a su vez llamado por el estimador al final de la simulación, y el **compare.reports** que puede ser usado manualmente por el analista para

comparar los informes de dos cadenas de simulaciones sobre el mismo modelo con los mismos parámetros. El contenido de un informe devuelto por **report** es el siguiente

1. **Text name**: Identificador del modelo
2. **Set colNames**: Nombres de las variables de la cadena de Markov
3. **Set repCfg**: Parámetros de configuración usados
4. **Matrix mcmc**: la cadena de Markov en la que cada fila es una simulación concreta y cada columna se refiere al parámetro de la misma posición en colNames
5. **Set sample**: conjunto de matrices columna extraídas de la anterior para facilitar la visualización gráfica de la cadena de Markov de cada variable.
6. **Set coda.raftery.diag**: Una tabla con una fila para el resultado del test de Raftery de CODA [**CODA**] de cada variable con la estructura **BysInf.Coda.Diag.Raftery** que se puede ver en `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/_structs.tol`. Los campos del miembro **bysInf_** que comienzan por **raftery.diag.** afectan al test de Raftery según se explica en la documentación de CODA.
7. **Set coda.summary**: Una tabla con una fila para el resultado del sumario CODA [**CODA**] de cada variable con la estructura **BysInf.Coda.Summary.Stats** que se puede ver en `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/_structs.tol`
8. **Set acf**: Los **bysInf_->acf.lag_** primeros términos de la función de autocorrelación observada de la cadena de Markov de cada parámetro
9. **Set histogram**: Histograma de frecuencias de **bysInf_->histogram.parts_** partes de la cadena de Markov de cada parámetro
10. **Set acf**: Función de autocorrelación observada de la cadena de Markov de cada parámetro
11. **Set kerDens**: Aproximación kernel-density de la densidad de probabilidad de la cadena de Markov de cada parámetro. Por defecto está deshabilitada porque lleva bastante tiempo de cálculo y a menudo no se utiliza para nada. Si se desea analizar hay que cambiar **bysInf_->kerDens.points** de 0 al número de puntos en los que se quiera evaluar la densidad aproximada, antes de llamar a **report**.

El metodo **loadAndReport** permite recuperar de disco una cadena MCMC y efectuar el informe llamando a **report** con ella.

9.7. Miembro BysMcmc::Bsr

En `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/_bsr.tol` se define este miembro que incluye el simulador específico de BSR que se ocupa de todo el manejo específico de la clase de modelos BSR y que consta a su vez de otros tres ficheros

- En `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_gibbs.tol` se define el miembro **NameBlock BysMcmc::Bsr::Gibbs** que incluye el simulador de Gibbs de BSR propiamente dicho que se explica en detalle en 10
- Otro componente fundamental de BSR es la API de definición de modelos en ficheros de datos ASCII con formato .bsr, el cual está implementado en la función built-in C++ de TOL Bsr.Parse que se describe en 11
- En `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/_import.tol` se define el miembro **NameBlock BysMcmc::Bsr::Import** que contiene la API genérica de definición de modelos de BSR cuya descripción se encuentra en 12
- En `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_dynhlm.tol` se define el miembro **NameBlock BysMcmc::Bsr::DynHlm** que contiene la API de definición de modelos de la clase restringida al caso jerárquico lineal HLM dinámico, en el sentido de que las observaciones ocurren en el tiempo. Para un estudio en profundidad de este tema ver 13

- En `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/_oneNode.tol` se define el miembro **NameBlock BysMcmc::Bsr::OneNode** que contiene una API simplificada que construye directamente un BSR.ModelDef a partir de un NameBlock con una estructura bastante sencilla para el caso de un modelo BSR con un solo nodo observacional. En estos casos no se necesita crear un archivo BSR ASCII ni mucho menos almacenar información alguna en la base de datos. Contiene el método Estim para modelos BSR básicos con filtros no lineales opcionales y el método EstimProbit especializado para el caso del modelo binario Probit.

CAPíTULO 10

BysMcmc::Bsr::Gibbs

Se describen los miembros y métodos generales del **NameBlock BysMcmc::Bsr::Gibbs** que implementa los métodos de simulación propiamente dichos de la clase de modelos de BSR y sus extenciones.

10.1. Descripción

Se encuentra en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_gibbs.tol` y se trata de la pieza fundamental de BSR y sus extensiones pues contiene los métodos de simulación de cada uno de los bloques posibles por separado, así como el ensamblaje de los bloques del BSR básico en un sólo simulador maestro para poder ser utilizados recursivamente como un bloque dentro de otro simulador maestro.

Cada bloque será un NameBlock que deberá contener al menos lo siguiente:

- Un miembro **NameBlock _.blockDef** creado con el método **BysMcmc::DefineBlock** explicado en 9.4 que incluye las características comunes a todos los bloques de simulación.
- Un método **Real setStore(Matrix values)** que cambia el valor de los parámetros llamando al método **_.blockDef::setStore(values)** y se ocupa luego, si procede, de actualizar todos los miembros internos necesarios para el condicionamiento a dichos valores en el resto de bloques que dependen de él. Para ello debe conocer perfectamente la semántica de dichos bloques.
- Un método **VMatrix draw(Real numSim, ...)** que genera la **numSim**-ésima simulación de los valores de los parámetros en función de una serie de argumentos condicionantes opcionales. El método maestro que ensambla los diferentes bloques debe conocer la sintaxis de los métodos **draw** de cada uno de sus bloques hijo. Este método debe llamar finalmente a **setStore** antes de devolver el vector de valores simulados para asegurar la cadena de condicionamientos.

Cada maestro será un NameBlock que deberá contener al menos los métodos exigidos al argumento **NameBlock modelSampler** del método **BysMcmc::BuildCycler()** según se describe en 9.3.

También incluye los métodos de llamada a la estimación bayesiana (simulación+diagnosis+inferencia) para ser aplicados sobre un master de simulación concreto de la forma más transparente posible.

10.2. Método `BysMcmc::Bsr::Gibbs::BasicMaster()`

El método **NameBlock BysMcmc::Bsr::Gibbs::BasicMaster(BSR.ModelDef modelDef, NameBlock config)** se implementa en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_basicMaster.tol` y sirve para crear el simulador maestro de un modelo BSR básico, es decir sin extensiones, tal y como se define en 1.2. El argumento **modelDef** se almacena en el miembro **Set _.modelDef** pues contiene toda la información necesaria para definir el modelo y poder simular sus parámetros, organizada según la estructura **BSR.ModelDef**:

- Set DocInfo: Información documental del modelo con estructura BSR.DocInfo
  - **Text Model.Name**: Nombre del modelo que identifica el objetivo del mismo sin distinguir entre diferentes sesiones o implementaciones del mismo.
  - **Text Model.Description**: Descripción del objetivo del modelo.
  - **Text Session.Name**: Nombre de la sesión o implementación concreta del modelo.

  - **Text Session.Description**: Descripción de los detalles de implementación, forma concreta del modelo, etc.
  - **Text Session.Authors**: Nombre o mejor e-mail del autor o autores separados por comas
  - **Text Session.Creation**: Fecha de creación de la sesión
  - **Text Path**: Camino absoluto del directorio donde almacenar resultados auxiliares y la propia cadena de Markov.
- **Set LinearBlock**: Tabla de parámetros del bloque lineal principal y sus valores iniciales dada por filas de estructura BSR.LinearBlock
  - **Text Name**: Nombre del parámetro
  - **Real InitialValue**: Valor inicial. Los valores iniciales deberían cumplir las restricciones pues aún no existe un método seguro y eficiente de encontrarlos de forma genérica.
- **Set InputMissingBlock**: Tabla de omitidos del input con elementos de estructura **BSR.MissingBlock**
  - **Text Name**: Nombre de la variable de omisión.
  - **Real Index**: El índice $k = 1 \dots K_0$ identificador del omitido.
  - **Real Row**: El número de ecuación $t = 1 \dots M$ en la que aparece el omitido.
  - **Real Col**: El número de columna aquí no tiene sentido pues el output sólo tiene una columna.
  - **Text Prior**: Identificador del tipo de a priori. Puede ser la cadena vacía, **"Normal"** o **"TruncatedNormal"**
  - **Real Nu**: La media de la distribución a priori o el valor inicial si no hay a priori
  - **Real Sigma2**: La varianza de la distribución a priori o $+\infty$ si no hay a priori
  - **Real MinBound**: El límite inferior o $-\infty$ si se desconoce o no existe.
  - **Real MaxBound**: El límite superior o $+\infty$ si se desconoce o no existe.
- **Set OutputMissingBlock**: Tabla de omitidos del output con elementos de esa misma estructura **BSR.MissingBlock,** salvo que el número de columna aquí no tiene mucho sentido pues el output sólo tiene una columna y se rellena siempre a 1.
- **Set NoiseDistrib**: Tabla de definición de la distribución del ruido de cada segmento y del propio segmento, definido cada uno por un elemento de estructura BSR.NoiseDistrib
  - **Text Name**: Identificador del segmento. Por ejemplo en un modelo jerárquico se suele usar el nombre del nodo.
  - **Real Nu**: Media del ruido. De momento debe ser cero y es posible que desaparezca.
  - **Text SigmaName**: Si no es la cadena vacía indica que la varianza de este segmento es un parámetro desconocido a simular en el modelo.
  - **Real SigmaIndex**: En el caso de el campo anterior no sea la cadena vacía indica la posición del parámetro varianza dentro del bloque de varianzas.
  - **Set Arima**: Estructura ARIMA del segmento. Puede ser un conjunto vacío o de elementos con estructura **ARIMAStruct** como se describe en 10.5
  - **Set ArimaAuxInfo**: Información auxiliar que se creará y manejará internamente en el bloque ARIMA como se describe en 10.5. Se debe pasar el conjunto vacío.
  - **VMatrix Cov**: Matriz de covarianzas del segmento. Por el momento debe ser diagonal.
  - **VMatrix L**: Descomposición de Choleski de Cov.
  - **VMatrix Li**: Inversa de L.
  - **Set EquIdx**: Índices de las ecuaciones contenidas en el segmento. Deben ser consecutivas para mayor eficiencia.
  - **Set TimeInfo**: Información temporal del segmento si es que la dimensión del mismo es el tiempo. Se utiliza en la creación de informes de evaluación de resultados pero no en la simulación en sí misma. Si no la tiene es el conjunto vacío y en oto caso se especifica mediante un conjunto con estructura BSR.NoiseTimeInfo:
    - **TimeSet Dating**: Fechado de las series involucradas.
    - **Date FirstDate**: Fecha de inicio.

    - **Date LastDate**: Fecha final.
- **VMatrix Y**: Matriz de output en términos originales con todos los segmentos ordenados secuencialmente.
- **VMatrix X**: Matriz sparse de input en términos originales con todos los segmentos ordenados secuencialmente.
- **VMatrix a**: Matriz de frontera de las inecuaciones lineales que restringen los parámetros del bloque lineal
- **VMatrix A**: Matriz de coeficientes de las inecuaciones lineales que restringen los parámetros del bloque lineal

Los bloques manejados por el maestro básico son los siguientes

1. **NameBlock _.lin.blk**: El bloque lineal principal se crea llamando a **NameBlock BysMcmc::Bsr::Gibbs::StdLinearBlock** (ver 10.3)
2. **NameBlock _.sig.blk**: El bloque de varianzas se crea llamando a **NameBlock BysMcmc::Bsr::Gibbs::SigmaBlock** (ver 10.4)
3. **NameBlock _.arm.blk**: El bloque ARMAs se crea llamando a **NameBlock BysMcmc::Bsr::Gibbs::ArimaBlock** (ver 10.5)
4. **NameBlock _.inpMis.blk**: El bloque de omitidos del input se crea llamando a **NameBlock BysMcmc::Bsr::Gibbs::InputMissingBlock** (ver 10.6)
5. **NameBlock _.outMis.blk**: El bloque de varianzas se crea llamando a **NameBlock BysMcmc::Bsr::Gibbs::OutputMissingBlock** (ver 10.7)

Todos ellos se almacenan dentro de un miembro **Set _.blocks** que garantiza el orden ejecución de algunos métodos genéricos de bloques.

El método **VMatrix findMatchingBeta(VMatrix givenInitVal)** comprueba que los valores iniciales son congruentes con las restricciones y en caso contrario intenta encontrar otros que sí lo sean aunque esto último no está garantizado por el momento

El método **Real initialize(Real unused)** llama a **findMatchingBeta** e inicializa todos los bloques.

El método principal **VMatrix draw(Real numSim)** muestrea todos los bloques en el orden adecuado, almacena resultados intermedios para mayor eficiencia y finalmente concatena las muestras de los bloques según el orden establecido.

Como es obvio, este NameBlock implementa los métodos obligatorios de un simulador maestro según se describe en 9.3, y además el método **Real setStore(Matrix values)** que se utilizará para usar el simulador como un bloque de otro simulador maestro, tal y como se explica en 10.8.

10.3. Método BysMcmc::Bsr::Gibbs::StdLinearBlock()

El método **NameBlock BysMcmc::Bsr::Gibbs::StdLinearBlock(Text name, Set colNames, Real numBlock, Real firstCol, NameBlock config)** se implementa en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_stdLinearBlock.tol` y sirve para crear el simulador de un bloque lineal estandarizado, no necesariamente el bloque lineal principal de una regresión, sino que también puede ser el utilizado en los tratamientos de omitidos y cualquier otra situación que requiera simular los parámetros de una regresión lineal con restricciones de desigualdad lineal opcionales, tal y como se describe en el capítulo 2.

En tiempo de creación del bloque no se sabe apenas nada aún de las dimensiones y aún menos del contenido de sus miembros que serán introducidos posteriormente mediante una llamada al método **Real initialize(VMatrix beta0, VMatrix A, VMatrix a)**, que se ocupa de asignar valores iniciales de los parámetros β_0, almacenar las restricciones $A \cdot \beta \leq a$ que han de ser, si las hay, inmutables a lo largo de las simulaciones, y de comprobar que se cumplen para los valores iniciales, es decir que $A \cdot \beta_0 \leq a$. Lógicamente, la matriz **A**, si no está vacía, debe tener tantas columnas como filas tenga **beta0**, y tantas filas como **a**. También se ocupa de asignar el método de muestreo al miembro oculto **_drawer**, ya que se dispone de métodos diferenciados de muestreo para el caso de haber restricciones **VMatrix draw.constrained(Real numSim, VMatrix Y,**

VMatrix X), o no haberlas **VMatrix draw.unconstrained(Real numSim, VMatrix Y, VMatrix X).**

En ambos casos la matriz **X** debe tener tantas columnas como filas tuviera la matriz **beta0** en la llamada a **initialize**, pero el número de filas puede variar en cada iteración siempre que coincida con el de la matriz **Y.** Los dos métodos llaman inicialmente al método **NameBlock solve(Real numSim, VMatrix Y, VMatrix X)** para calcular el vector de medias y la descomposición de la matriz de covarianzas de la multinormal asociada a la regresión lineal tal y como se describe en la ecuación 2.2.3. En el caso sin restricciones el método implementa las ecuaciones 2.2.5 de forma casi trivial. Si hay restricciones se implementa el algoritmo 2.2.3.

El método principal **VMatrix draw(Real numSim, VMatrix Y, VMatrix X)** llamará finalmente al **_drawer** si el bloque está habilitado, pues en otro caso devuelve el último valor almacenado de forma constante, lo cual, si no hay ninguna injerencia externa, supondrá devolver siempre **beta0**.

Hay que hacer notar que, para cumplir con la condición de estandarización, es preciso dividir cada fila de Y y de X por la sigma correspondiente al segmento en el que se encuentre, y, si existe parte ARIMA, es necesario también filtrar previamente tanto el input como el output antes de llamar a este método draw.

10.4. Método BysMcmc::Bsr::Gibbs::SigmaBlock()

El método **NameBlock BysMcmc::Bsr::Gibbs::SigmaBlock(Text name, Set NoiseDistrib, Real numBlock, Real firstCol, NameBlock config)** se implementa en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_sigmaBlock.tol` y sirve para crear el simulador de un bloque de varianzas tal y como se describe en el capítulo 3. El bloque contiene los parámetros σ_k^2 correspondientes a las varianzas de cada uno de los segmentos de regresión con varianza desconocida, los cuales se referencian en el miembro **Set _.sigmaBlockVar** creado en tiempo de creación del manejador a partir del argumento de llamada **Set NoiseDistrib** con elementos de estructura **BSR.NoiseDistrib**, uno para cada segmento, y que se corresponde con el campo de igual nombre de la estructura **BSR.ModelDef** almacenado en el miembro **Set _.modelDef** del simulador maestro creado con **BysMcmc::Bsr::Gibbs::BasicMaster()**.

El método principal **VMatrix draw(Real numSim, VMatrix noise)** llamará a **VMatrix draw.invRandChisq(Real numSim, VMatrix noise)** si el bloque está habilitado, pues en otro caso devuelve el último valor almacenado de forma constante. En este caso será el generador maestro quien se ocupe de asignar un valor inicial pues el valor por defecto es cero.

Quedan por implementar los siguientes aspectos:

- Introducción de información a priori sobre cada varianza
- Introducción de información latente para un grupo de varianzas
- Modelos heterocedásticos

10.5. Método BysMcmc::Bsr::Gibbs::ArimaBlock()

El método **NameBlock BysMcmc::Bsr::Gibbs::ArimaBlock(Text name, Set NoiseDistrib, Real numBlock, Real firstCol, NameBlock config)** se implementa en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_arimaBlock.tol` y sirve para crear el simulador del bloque de parámetros ARMA tal y como se describe en el capítulo 4. El bloque contiene los parámetros ϕ, θ correspondientes a los polinomios AR y MA respectivamente de cada uno de los segmentos de regresión para los que se haya definido dicha estructura, los cuales se referencian en el miembro **Set _.armaParamBlock** creado en tiempo de creación del manejador a partir del argumento de llamada **Set NoiseDistrib** con elementos de estructura **BSR.NoiseDistrib**, uno para cada segmento,que se almacena en el miembro **Set _.NoiseDistrib** y que se corresponde con el campo de igual nombre de la estructura **BSR.ModelDef** almacenado en el miembro **Set _.modelDef** del simulador maestro creado con **BysMcmc::Bsr::Gibbs::BasicMaster()**. En el miembro **Set _.arimaParamBlock** se

almacenan los índices de todos aquellos que tengan estructura ARIMA, es decir los de **_.armaParamBlock** más los que tengan sólo estructura determinista definida habitualmente mediante un polinomio unitario, también llamado de diferencias pues suele ser de la forma $(1 - B^k)^n$. Estas estructuras deterministas obviamente no tienen parámetros que simular y se usan sólo para filtrar las matrices de input y output de la regresión lineal y construir así el ruido diferenciado o ruido ARMA.

Los polinomios AR y MA están en todo momento recogidos en el campo **Set Arima** de cada elemento del miembro **_.NoiseDistrib**. Cada elemento corresponde a un factor de estacionalidad arbitraria y tiene estructura **ARIMAStruct**, siendo los grados de los factores AR y MA 0, 1 ó 2. Al margen de ello se almacena toda la información auxiliar, alguna aparentemente ?, necesaria para acelerar los cálculos en el campo **Set ArimaAuxInfo** que no tiene una estructura fija pues su contenido depende de si hay parte ARMA, sólo diferencias o nada de parte ARIMA.

Toda esta información auxiliar se asigna mediante el método **Real initialize(Real unused)** que se llama antes de la propia creación del almacén genérico _.blockDef , ya que antes de eso ni siquiera se conocen aún el número ni los nombres de las variables AR y MA implicadas.

Una vez simulados los parámetros ARMA, se llama al método **Real calcAllArimaBlockLi(Real numSim)** llama para cada segmento **_.armaParamBlock** al método **Real calcArimaBlockLi(BSR.NoiseDistrib segment)** que llama a su vez a **ARMAProcess::FastCholeskiCovFactor** implementado en `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/arima/_arma_process.tol`, para calcular la descomposición sintética explicada en 4.4.2. También se le llama al principio del todo para que se pueda utilizar el filtro ARIMA con los valores iniciales.

El método **VMatrix filter(VMatrix X)** se ocupa de filtrar de la parte ARIMA conjunta a una matriz (input,output,noise,etc.) referida a todos los segmentos que no sean de ruido blanco puro, filtrando primero la parte determinista mediante una llamada a **VMatrix filter.deterministic(VMatrix X)**; y luego se encarga de la parte estocástica el método **VMatrix filter.arma(VMatrix DX)** que llama en ciclo para cada segmento al método **VMatrix filterJustBlock(BSR.NoiseDistrib segment, VMatrix X)** que se ocupa de filtrar la parte ARMA de un segmento concreto aplicando el algoritmo 4.4.3.

Si el bloque está habilitado, pues en otro caso devuelve el último valor almacenado de forma constante, el método principal **VMatrix draw(Real numSim, VMatrix noise, VMatrix sigBlk)** llamará a **VMatrix draw.ARMS(Real numSim, VMatrix noise, VMatrix sigBlk)** que aplica el método ARMS de simulación de los parámetros AR y MA uno por uno condicionados al resto, tal y como se describe en 4.3. Los límites de muestreo de cada factor AR ó MA se calculan mediante ARMAProcess::StationarityBounds.2 implementada también en `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/arima/_arma_process.tol`. Para ello, primero simula los valores iniciales condicionados a los valores actuales de los productos ARMA, mediante el método **ARMAProcess::Eval.Almagro(...)::Draw.U_** implementado en `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/arima/_arma_process.tol`, pero no los almacena en la cadena de Markov ya que son fácilmente reproducibles en el remoto caso en que hicieran falta, y pueden ocupar mucho espacio si la estacionalidad de los factores es grande. Luego se simulan todos los parámetros ARMA condicionados a dichos valores iniciales con el método ARMS aplicado ala función de densidad implementada en ese mismo fichero con el nombre **ARMAProcess::Eval.Almagro(...)::LogLH.Z_cond_U(0)**.

Quedan por implementar los siguientes aspectos:

- Introducción de información a priori sobre cada parámetro ARMA
- Introducción de información latente para un grupo de parámetros referidos a un mismo parámetro (mismos indicadores de factor, estacionalidad, AR ó MA, grado) en diferentes segmentos con estructura ARIMA equivalente.
- Pruebas con el método Slice Sampler alternativo a ARMS que además no precisa de un intervalo exacto de límites de simulación.

10.6. Método BysMcmc::Bsr::Gibbs::InputMissingBlock()

El método **NameBlock BysMcmc::Bsr::Gibbs::InputMissingBlock(Text name, Set inputMissingBlock, Real numBlock, Real firstCol, NameBlock config)** se implementa en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_inputMissingBlock.tol` y sirve para crear el simulador de un bloque de valores omitidos en el input, tal y como se describe en el capítulo 5. El bloque contiene los parámetros u correspondientes a los valores desconocidos de todos los inputs de cada uno de los bloques de regresión.

El acceso a los datos en el modelo BSR lo proporciona el argumento **inputMissingBlock** con elementos de estructura **BSR.MissingBlock**, uno para dato omitido, y que viene dado siempre como el campo **_.modelDef->InputMissingBlock** de la estructura **BSR.ModelDef** correspondiente al miembro **Set _.modelDef** del simulador maestro creado con **BysMcmc::Bsr::Gibbs::BasicMaster()**. Este argumento incluye en cada elemento **inputMissingBlock**[k] con la estructura **BSR.MissingBlock** definida en 10.2

En resumen, la distribución a priori puede ser una cualquiera de estas

- no informativa: si es **Sigma2** igual a $+\infty$ y **MinBound** y **MaxBound** son $-\infty$ y $+\infty$ respectivamente.
- uniforme en un intervalo: si es **Sigma2** igual a $+\infty$ y al menos uno de los dos límites **MinBound** o **MaxBound** son finitos.
- normal: si es **Sigma2** un valor finito y **MinBound** y **MaxBound** son $-\infty$ y $+\infty$ respectivamente.
- normal truncada: si es **Sigma2** un valor finito y al menos uno de los dos límites **MinBound** o **MaxBound** son finitos.

La información a priori se recoge en los siguientes miembros, todos ellos son inmutables a lo largo de las simulaciones y referidos a la fórmula 5.3.5:

- **VMatrix _.prior.**si: Puesto que la distribución a priori es independiente para cada omitido su covarianza es diagonal y se puede definir la inversa de su descomposición de Choleski, es decir L_u, como un vector.
- **VMatrix _.prior.X**: La matriz de Cholesky L_u extendida con ceros donde no hay información a priori.
- **VMatrix _.prior.Y**: El producto $L_u \cdot \mu_u$ extendido con ceros donde no hay información a priori.
- **VMatrix _.prior.A**: La matriz de coeficientes de las inecuaciones A_u
- **VMatrix _.prior.a**: La matriz de frontera de las inecuaciones a_u

Los campos **Index**, **Row**, y **Col** definen la matriz booleana $\delta^v \in \{0,1\}^{M \times K_0}$de ubicación de los omitidos. Para optimizar los procesos de actualización y extracción de la información necesaria se crean una serie de miembros de indexación creados a partir de los campos de ubicación, y por tanto inmutables durante la simulación:

- **Matrix _.tj**: La matriz de pares (ecuación, variable) ó (Row,Col) ó (t, j)
- **Matrix _.tk**: La matriz de pares (ecuación, omitido) ó (Row,Index) ó (t, k)
- **Set _.j**: El conjunto de variables afectadas por omitidos

Todos estos miembros inmutables se asignan mediante el método **Real initialize(Real unused)** que se llama en la propia creación.

Con toda esta información se genera internamente un bloque lineal llamando a **NameBlock BysMcmc::Bsr::Gibbs::StdLinearBlock** (ver 10.3) que se almacenará como el miembro **NameBlock _.lin.blk**

El método **VMatrix draw(Real numSim, NameBlock arima, VMatrix si, VMatrix beta, VMatrix Z)** , construye las matrices de input **X01** y output **Y01** de la regresión estandarizada y simula los omitidos llamando a **_.lin.blk::draw(numSim, Y01, X01)**.

10.7. Método BysMcmc::Bsr::Gibbs::OututMissingBlock()

El método **NameBlock BysMcmc::Bsr::Gibbs::OutputMissingBlock(Text name, Set outputMissingBlock, Real numBlock, Real firstCol, NameBlock config)** se implementa en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_outputMissingBlock.tol` y sirve para crear el simulador de un bloque de valores omitidos en el output, tal y como se describe en el capítulo 5. El bloque contiene los parámetros v correspondientes a los valores desconocidos de todos los inputs de cada uno de los bloques de regresión.

El acceso a los datos en el modelo BSR lo proporciona el argumento **outputMissingBlock** con elementos de estructura **BSR.MissingBlock**, uno para dato omitido, y que viene dado siempre como el campo **_.modelDef->OutputMissingBlock** de la estructura **BSR.ModelDef** correspondiente al miembro **Set _.modelDef** del simulador maestro creado con **BysMcmc::Bsr::Gibbs::BasicMaster()**. Este argumento incluye en cada elemento **outputMissingBlock**[k] con estructura **BSR.MissingBlock** definida en 10.2, salvo que aquí el campo **Real Col** del número de columna no tiene ningún sentido pues el output sólo tiene una columna. Por eso, en este caso sólo los campos **Index**, **Row** definen la matriz booleana $\delta^v \in \{0,1\}^{M \times K_0}$ de ubicación de los omitidos. Para optimizar los procesos de actualización y extracción de la información necesaria se crea el miembro de indexación **Set _.t** que almacena el conjunto de ecuaciones afectadas por omitidos en el output y se crea a partir de los campos de ubicación, y es por tanto inmutable durante la simulación.

Los campos que almacenan la información a priori son idénticos a los del caso de omitidos del input recién vistos.

Todos los miembros inmutables se asignan mediante el método **Real initialize(Real unused)** que se llama en la propia creación.

Con toda esta información se genera internamente un bloque lineal llamando a **NameBlock BysMcmc::Bsr::Gibbs::StdLinearBlock** (ver 10.3) que se almacenará como el miembro **NameBlock _.lin.blk**

El método **VMatrix draw(Real numSim, NameBlock arima, VMatrix si, VMatrix beta, VMatrix Z)** , construye las matrices de input **X01** y output **Y01** de la regresión estandarizada y simula los omitidos llamando a **_.lin.blk::draw(numSim, Y01, X01)**.

10.8. Método BysMcmc::Bsr::Gibbs::BsrAsBlock()

El método **NameBlock BysMcmc::Bsr::Gibbs::BsrAsBlock(NameBlock bsrInstance, Real numBlock, Real firstCol, NameBlock config)** se implementa en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_bsrAsBlock.tol` y permite usar el simulador maestro del modelo básico BSR como un bloque de otro simulador maestro. El argumento **bsrInstance** define el modelo BSR básico que fue generado previamente con **NameBlock BysMcmc::Bsr::Gibbs::BasicMaster** (ver 10.2) y el bloque se define simplemente como la concatenación de los bloques internos del BSR.

10.9. Método BysMcmc::Bsr::Gibbs::NonLinBlock()

El método **NameBlock BysMcmc::Bsr::Gibbs::NonLinBlock(NameBlock filterEval, NameBlock bsr, Real numBlock, Real firstCol, NameBlock config)** se implementa en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_nonLinBlock.tol` y permite simular un bloque correspondiente a un filtro no lineal arbitrario **filterEval**, como los definidos en 6 que afecta al output o a parte del input de un modelo BSR básico **bsr** creado con **BasicMaster** (ver 10.2).

Un fitro no lineal es un NameBlock que tiene los siguientes métodos obligatorios:

- **Text get.name(Real unused)**: Identificador del filtro
- **Text get.segmentName(Real unused)**: Identificador del segmento
- **Text get.colNames(Real unused)**: Nombres de los parámetros del bloque no lineal
- **Set get.bounds(i,param)**: Devuelve los límites inferior y superior del i-ésimo parámetro dados los valores actuales del resto.

- **Matrix eval(Matrix param)**: Evalúa el filtro no lineal para un vector de valores concretos de los parámetros. El número de filas debe ser la longitud del segmento y el de columnas será 1 si es un filtro del output, o el cardinal de **get.linBlkNames(0)** si es del input.
- **Set get.linBlkNames(Real unused)**: Nombres de los parámetros del bloque lineal afectados por un bloque lineal del input. Lógicamente este método sólo debe existir si es un filtro del input, y de hecho es la única forma que tiene el bloque de saber si es del output o del input, si no existe es del output y si existe es del input.

Si llamamos W al valor actual del ruido ARIMA o ruido no diferenciado del model BSR básico

$$W = F_0 - X \cdot \beta - \sum_{r=1}^{R} F_r \cdot \alpha_r \tag{10.9.1}$$

entonces el output de la regresión no lineal del bloque lineal del input es (ver 6.2.1):

$$Z_k = F_0 - X \cdot \beta - \sum_{\substack{r=1 \\ r \neq k}}^{R} F_r \cdot \alpha_r = F_k \cdot \alpha_k + W \tag{10.9.2}$$

y el del output es (ver 6.2.6):

$$Z_0 = X \cdot \beta + \sum_{r=1}^{R} F_r \cdot \alpha_r = F_0 - W \tag{10.9.3}$$

Esto ofrece una forma rápida de evaluar dichos output que además es prácticamente idéntica para el input y el output, salvo el signo de W.

El método principal **VMatrix draw(Real numSim)** llamará a **VMatrix draw.ARMS(Real numSim)** si el bloque está habilitado, pues en otro caso devuelve el último valor almacenado de forma constante.

Una vez simulados los valores se debe evaluar el filtro resultante de los mismos y actualizar, o bien el output o bien la parte de input correspondiente, de lo cual se encarga el método **Real setStore(Matrix values)** llamando a los métodos **_.bsr::setSegmentOutput** ó **_.bsr::setSegmentInputCol** respectivamente. Es importante destacar que las matrices que se le pasen a dichos métodos tengan la misma estructura sparse que los correspondientes bloques en las matrices originales para asegurar una actualización eficiente sin transformaciones internas de la matriz. Si no se está seguro de que esta estructura sea inmutable independientemente de los parámetros no lineales entonces lo más aconsejable es considerar dichos bloques matriciales como densos en la definición original del modelo. Para ello basta con explicitar los coeficientes cero en las expresiones de la regresión que toquen en el fichero ASCII con formato BSR (ver 11)

10.10. Método BysMcmc::Bsr::Gibbs::DeltaTransfer()

El método **NameBlock DeltaTransfer (Text segmentName, Text inputName, Set linBlkNames, Polyn omega, Polyn delta, Matrix x0, Matrix x)** se implementa en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_deltaTransfer.tol` y define un filtro no lineal correspondiente a una función de transferencia con deltas como el explicado en 6.3.1, y que se puede usar como argumento de **NonLinBlock** (Ver 10.9), es decir cumple con la API mínima exigida.

El método **get.bounds** cacula los límites de muestreo de cada factor delta mediante ARMAProcess::StationarityBounds.2 implementada también en `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/arima/_arma_process.tol`. Los límites de los valores iniciales de transferencia no están definidos pero como ARMS los requiere se le pasan valores suficientemente grandes.

El método de evaluación **eval** no es más que resolver la ecuación en diferencias y devolver una matriz con una columna para cada monomio omega, con su correspondiente retardo.

Quedan por implementar los siguientes aspectos:

- Posibilidad de incluir los omitidos que hagan falta tanto en los valores iniciales x_0 como en los datos del input original propiamente dicho x.
- Pruebas con el método Slice Sampler alternativo a ARMS que además no precisa de un intervalo exacto de límites de simulación.

10.11. Método BysMcmc::Bsr::Gibbs::NonLinMaster()

El método **NameBlock BysMcmc::Bsr::Gibbs::NonLinMaster(BSR.ModelDef modelDef, Set nonLinFilter, NameBlock config)** se implementa en el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/gibbs/_nonLinMaster.tol` y permite simular un modelo BSR extendido con filtros no lineales arbitrarios como los definidos en 6.

En primer lugar se crea el master del BSR básico a partir del **modelDef**, llamando a **BasicMaster** (ver 10.2) y luego se crea un bloque a partir de ese master llamando a **BsrAsBlock** (ver 10.8)

Luego para cada filtro no lineal se genera un bloque que lo gestione con **NonLinBlock** (ver 10.9).

Todos los bloques, el de BSR y los no lineales, se almacenan dentro de un miembro **Set _.blocks** que garantiza el orden ejecución de algunos métodos genéricos de bloques.

El método principal **VMatrix draw(Real numSim)** simplemente muestrea todos los bloques en el orden adecuado y concatena las muestras de los bloques según el orden establecido.

Como es obvio, este NameBlock implementa los métodos obligatorios de un simulador maestro según se describe en 9.3, y además el método **Real setStore(Matrix values)** que se podrá utilizar para usar el simulador como un bloque de otro simulador maestro definido por el usuario.

CAPíTULO 11

El formato ASCII-BSR

Se describe el formato ASCII-BSR de definición de modelos con la función built-in BSR.Parse

11.1. Descripción

La estructura de datos que deine un modelo básico BSR explicada en 10.2 está diseñada para la máxima eficiencia en la simulación de todos los bloques, pero es realmente complicada de manejar por el analista, e incluso por sistemas automatizados escritos en TOL, es muy difícil de manejar y visualizar y es prácticamente imposible de mantener en un sistema en producción.

Para paliar este problema existe un formato de definición de modelos en ficheros ASCII con extensión .bsr que permite escribir el modelo con una sintaxis muy próxima a las expresiones matemáticas involucradas, que en casos de modelos pequeños se podría incluso escribir a mano, pero que además es bastante fácil de generar mecánicamente desde programas TOL o incluso desde cualquier otro lenguaje de programación que permita escritura formateada.

11.2. Ejemplo

En los directorios test_01, test_02 y test_03 disponibles en `http://cvs.tol-project.org/viewcvs.cgi/tol/stdlib/math/stat/models/bayesian/bysMcmc/` se puede ver ejemplos de modelos generados automáticamente de forma aleatoria en los que los archivos def.tol construyen ficheros .bsr a partir de ciertos parámetros dimensionales para el chequeo de calidad del simulador BSR básico.

A continuación se puede ver un ejemplo generado automáticamente con el ejemplo test_03 de un modelo HLM muy sencillo pero completo, con sólo dos nodos observacionales con parte ARIMA y omitidos en el input y en el output, y con un nodo latente para cada variable observacional y otro a priori para los hiperparámetros latentes.

```
1  ////////////////////////////////////////////////////////////////////////////
2  // Bayesian Sparsed Regression
3  ////////////////////////////////////////////////////////////////////////////
4
5  $BEGIN
6
7  Model.Name = "bysMcmc.test_03";
8  Model.Description =
9  "Randomly generation of a hierarchycal lineal model with these simple\n"
10 "structure:\n"
11 "  1: a set of K1 observational nodes with the same number n of variables\n"
12 "     and data length m\n"
13 "  2: a set of n latent nodes with one central parameter for each distinct\n"
14 "     observational variable with specified standard deviation\n"
15 "  3: a central prior equation for each latent variable with specified\n"
16 "     standard deviation\n"
17 "  4: a set of sparse linear inequalities over the observational variables\n"
18 "";
19 Session.Name = "RS_1483339472";
20 Session.Description =
21 "Random seed=1483339472";
22 Session.Authors = "vdebuen@tol-project.org";
23
24 ////////////////////////////////////////////////////////////////////////////
25 // Regression Variables Declarations with initial values
26 ////////////////////////////////////////////////////////////////////////////
27
28 NodeObs.1_Var.1 <- +0.53803037013858557;
29 NodeObs.1_Var.2 <- +0.46934376657009125;
30 NodeObs.1_Var.3 <- -0.22560517396777868;
31 NodeObs.2_Var.1 <- +0.86625403119251132;
```

```
32  NodeObs.2_Var.2 <- -0.15567512763664126;
33  NodeObs.2_Var.3 <- -0.075121806003153324;
34  NodeLat.1_Var.nu <- +0.15836594998836517;
35  NodeLat.2_Var.nu <- -0.85943601280450821;
36  NodeLat.3_Var.nu <- +0.18600673880428076;
37
38  ////////////////////////////////////////////////////////////////////////////////
39  // Regression Missing Input Declarations
40  ////////////////////////////////////////////////////////////////////////////////
41
42  MissInp.NodeObs.3.13 ? ~ Normal(0.896203693933785,0.2007952652551403);
43  MissInp.NodeObs.6.30 ? ~ Normal(0.6046165255829692,0.09139028575200531);
44  MissInp.NodeObs.3.2 ? ~ TruncatedNormal(0.9820013050921261,0.2410816408006598,0,9.820013050921261);
45  MissInp.NodeObs.5.40 ? ~ Normal(-0.7318465355783701,0.1338998379095156);
46
47  ////////////////////////////////////////////////////////////////////////////////
48  // Regression Missing Output Declarations
49  ////////////////////////////////////////////////////////////////////////////////
50
51  MissOut.NodeObs.31 ? <- -0.4157893265383703;
52  MissOut.NodeObs.40 ? <- -0.3029720514576624;
53
54  ////////////////////////////////////////////////////////////////////////////////
55  // Regression Residuals Declarations Normal(0,sigma^2)
56  ////////////////////////////////////////////////////////////////////////////////
57
58  NodeObs.1_Residuals[20 dating Daily from y2008m08d09 until y2008m08d28] ~ Normal(0,NodeObs.1_Sigma2*ARIMA.COV([[
        ARIMAStruct( 1, 1-1.709702122239 66*B+0.9668938382528 72*B^2, 1, 1), ARIMAStruct( 7, 1, 1, 1)]]));
59  NodeObs.2_Residuals[20 dating Daily from y2008m08d09 until y2008m08d28] ~ Normal(0,NodeObs.2_Sigma2*ARIMA.COV([[
        ARIMAStruct( 1, 1-0.0191113898593219*B+0.129901138134301*B^2, 1, 1), ARIMAStruct( 7, 1, 1, 1)]]));
60  NodeLat.1_Residuals[2] ~ Normal(0,0.03276045051869933);
61  NodeLat.2_Residuals[2] ~ Normal(0,0.030080604978733022);
62  NodeLat.3_Residuals[2] ~ Normal(0,0.010264774427016805);
63  NodePri.1_Residuals[3] ~ Normal(0,0.011844979275860658);
64
65  ////////////////////////////////////////////////////////////////////////////////
66  // Regression equations
67  ////////////////////////////////////////////////////////////////////////////////
68
69  +5.2435581658297865 == NodeObs.1_Residuals[1]; //Reg.Equ.1
70  +2.1913126132131682 == NodeObs.1_Residuals[2]+MissInp.NodeObs.3.2*NodeObs.1_Var.3 ; //Reg.Equ.2
71  +0.12304427448649291 == NodeObs.1_Residuals[3]+0.63090779585763812*NodeObs.1_Var.1 +0.54491685889661312*NodeObs
        .1_Var.2 +0.94600624032318592*NodeObs.1_Var.3 ; //Reg.Equ.3
72  -2.6088045980742529 == NodeObs.1_Residuals[4]-0.66879721963778138*NodeObs.1_Var.2 ; //Reg.Equ.4
73  -6.0452993628906073 == NodeObs.1_Residuals[5]; //Reg.Equ.5
74  -6.3822864805891921 == NodeObs.1_Residuals[6]+0.95094704767689109*NodeObs.1_Var.1 -0.54092238331213593*NodeObs.1
        _Var.2 ; //Reg.Equ.6
75  -6.6295145939371594 == NodeObs.1_Residuals[7]+0.93608348118141294*NodeObs.1_Var.1 +0.9823420150205493*NodeObs.1
        _Var.2 ; //Reg.Equ.7
76  -5.5561393799341996 == NodeObs.1_Residuals[8]+0.7666153940372169*NodeObs.1_Var.2 -0.66276752250269055*NodeObs.1
        _Var.3 ; //Reg.Equ.8
77  -3.1295982695386888 == NodeObs.1_Residuals[9]-0.56221867958083749*NodeObs.1_Var.1 -0.90145999845117331*NodeObs.1
        _Var.3 ; //Reg.Equ.9
78  +0.76951244117245921 == NodeObs.1_Residuals[10]; //Reg.Equ.10
79  +2.9132946114405867 == NodeObs.1_Residuals[11]+0.95396953029558063*NodeObs.1_Var.3 ; //Reg.Equ.11
80  +5.7416648977148181 == NodeObs.1_Residuals[12]-0.95996885281056166*NodeObs.1_Var.3 ; //Reg.Equ.12
81  +6.3831794474164543 == NodeObs.1_Residuals[13]-0.64427804760634899*NodeObs.1_Var.2 +MissInp.NodeObs.3.13*NodeObs
        .1_Var.3 ; //Reg.Equ.13
82  +4.0282180180201248 == NodeObs.1_Residuals[14]-0.90212121233344078*NodeObs.1_Var.1 ; //Reg.Equ.14
83  +4.3505297967804113 == NodeObs.1_Residuals[15]+0.85422466974705458*NodeObs.1_Var.1 -0.78263330226764083*NodeObs
        .1_Var.3 ; //Reg.Equ.15
84  -1.8762849597824245 == NodeObs.1_Residuals[16]-0.71285796910524368*NodeObs.1_Var.1 -0.58573955669999123*NodeObs
        .1_Var.3 ; //Reg.Equ.16
85  -5.6888365292228968 == NodeObs.1_Residuals[17]+0.98112034006044269*NodeObs.1_Var.2 -0.79486409109085798*NodeObs
        .1_Var.3 ; //Reg.Equ.17
86  -8.3673617279440613 == NodeObs.1_Residuals[18]; //Reg.Equ.18
87  -8.0537960538725955 == NodeObs.1_Residuals[19]+0.52584361424669623*NodeObs.1_Var.2 +0.67250693123787642*NodeObs
        .1_Var.3 ; //Reg.Equ.19
88  -5.9263487197705214 == NodeObs.1_Residuals[20]-0.96396565064787865*NodeObs.1_Var.2 +0.8175902683287859*NodeObs.1
        _Var.3 ; //Reg.Equ.20
89  -0.6068887599917272 == NodeObs.2_Residuals[1]+0.56295138271525502*NodeObs.2_Var.2 ; //Reg.Equ.21
90  +2.0939562031264218 == NodeObs.2_Residuals[2]+0.65345998154953122*NodeObs.2_Var.1 -0.71942238695919514*NodeObs.2
        _Var.3 ; //Reg.Equ.22
91  +0.60038804953399949 == NodeObs.2_Residuals[3]-0.84704046417027712*NodeObs.2_Var.2 ; //Reg.Equ.23
92  -1.0713915154528588 == NodeObs.2_Residuals[4]-0.93844964541494846*NodeObs.2_Var.1 ; //Reg.Equ.24
93  -0.32125826101136556 == NodeObs.2_Residuals[5]+0.51997147314250469*NodeObs.2_Var.1 -0.63292960310354829*NodeObs
        .2_Var.2 ; //Reg.Equ.25
94  -0.54762880703286565 == NodeObs.2_Residuals[6]-0.50578109361231327*NodeObs.2_Var.1 ; //Reg.Equ.26
95  +2.4653275135908528 == NodeObs.2_Residuals[7]+0.54335346445441246*NodeObs.2_Var.1 -0.93712931545451283*NodeObs.2
        _Var.2 -0.72416583448648453*NodeObs.2_Var.3 ; //Reg.Equ.27
```

```
96  +0.527148126841143 65 == NodeObs.2_Residuals[8]+0.95070930058136582*NodeObs.2_Var.1 -0.97863368503749371*NodeObs
        .2_Var.3 ; //Reg.Equ.28
97  -1.02703702737663 12 == NodeObs.2_Residuals[9]-0.88577614538371563*NodeObs.2_Var.1 +0.54846574366092682*NodeObs.2
        _Var.3 ; //Reg.Equ.29
98  +0.25652362241255156 == NodeObs.2_Residuals[10]-0.61568234255537391*NodeObs.2_Var.2 +MissInp.NodeObs.6.30*
        NodeObs.2_Var.3 ; //Reg.Equ.30
99  +MissOut.NodeObs.31 == NodeObs.2_Residuals[11]-0.73601436568424106*NodeObs.2_Var.3 ; //Reg.Equ.31
100 +0.38486138593475 == NodeObs.2_Residuals[12]; //Reg.Equ.32
101 -0.11539390961624 94 == NodeObs.2_Residuals[13]-0.52979899849742651*NodeObs.2_Var.1 ; //Reg.Equ.33
102 +0.20153337099189605 == NodeObs.2_Residuals[14]; //Reg.Equ.34
103 -0.60618407368998284 == NodeObs.2_Residuals[15]-0.61933115171268582*NodeObs.2_Var.1 ; //Reg.Equ.35
104 +1.0049067997452399 == NodeObs.2_Residuals[16]+0.86020477814599872*NodeObs.2_Var.2 -0.98850312503054738*NodeObs
        .2_Var.3 ; //Reg.Equ.36
105 -0.72396483131240752 == NodeObs.2_Residuals[17]+0.87632823176681995*NodeObs.2_Var.3 ; //Reg.Equ.37
106 +0.27870237026751166 == NodeObs.2_Residuals[18]; //Reg.Equ.38
107 -0.74122059405715657 == NodeObs.2_Residuals[19]-0.75435957591980696*NodeObs.2_Var.1 -0.91726320842280984*NodeObs
        .2_Var.2 ; //Reg.Equ.39
108 +MissOut.NodeObs.40 == NodeObs.2_Residuals[20]-0.72136184806004167*NodeObs.2_Var.1 +MissInp.NodeObs.5.40*NodeObs
        .2_Var.2 ; //Reg.Equ.40
109 0 == NodeLat.1_Residuals[1]-NodeObs.1_Var.1 +NodeLat.1_Var.nu ; //Reg.Equ.41
110 0 == NodeLat.1_Residuals[2]-NodeObs.2_Var.1 +NodeLat.1_Var.nu ; //Reg.Equ.42
111 0 == NodeLat.2_Residuals[1]-NodeObs.1_Var.2 +NodeLat.2_Var.nu ; //Reg.Equ.43
112 0 == NodeLat.2_Residuals[2]-NodeObs.2_Var.2 +NodeLat.2_Var.nu ; //Reg.Equ.44
113 0 == NodeLat.3_Residuals[1]-NodeObs.1_Var.3 +NodeLat.3_Var.nu ; //Reg.Equ.45
114 0 == NodeLat.3_Residuals[2]-NodeObs.2_Var.3 +NodeLat.3_Var.nu ; //Reg.Equ.46
115 -0.6096816505305469 == NodePri.1_Residuals[1]-NodeLat.1_Var.nu ; //Reg.Equ.47
116 +0.66229855222627521 == NodePri.1_Residuals[2]-NodeLat.2_Var.nu ; //Reg.Equ.48
117 +0.46403526607900858 == NodePri.1_Residuals[3]-NodeLat.3_Var.nu ; //Reg.Equ.49
118
119 ////////////////////////////////////////////////////////////////////////////////
120 // Restriction inequations
121 ////////////////////////////////////////////////////////////////////////////////
122
123 +1.1888731951872438 >= +NodeObs.1_Var.1 ;
124 +0.50465998947620394 >= +NodeObs.1_Var.2 ;
125 -0.21083058095071464 >= +NodeObs.1_Var.3 ;
126 +0.9361133812088519 >= +NodeObs.2_Var.1 ;
127 -0.13837098166579381 >= +NodeObs.2_Var.2 ;
128 -0.0033603257639333584 >= +NodeObs.2_Var.3 ;
129 +1.1404310412146701 >= +NodeObs.1_Var.1 ;
130 +0.52799691481748601 >= +NodeObs.1_Var.2 ;
131 -0.73132390667684377 >= -NodeObs.1_Var.1 +NodeObs.1_Var.3 ;
132 +1.3114740782628398 >= -NodeObs.1_Var.2 +NodeObs.2_Var.1 ;
133 +0.13644857598003002 >= -NodeObs.1_Var.3 +NodeObs.2_Var.2 ;
134 -0.86476675785612311 >= -NodeObs.2_Var.1 +NodeObs.2_Var.3 ;
135
136 $END
```

11.3. API modular

Desde la versión 2.0.1 de TOL es posible definir un modelo BSR de forma modular para resolver tres problemas fundamentales:

1. Por una parte, escribir explícitamente las ecuaciones de nodos observacionales densos y grandes es demasiado lento. (Ver ticket #634)
2. Por otra parte, los modelos masivos con multitud de nodos requieren demasiados recursos si se intentan cargar monolíticamente. (Ver ticket #628)
3. Además, la sintaxis actual no incorpora los filtros no lineales por loque los medelos están inclompletos y no pueden ser ejecutados en sesiones TOL ulteriores.

Para resolver estos problemas se plantea un nuevo esquema en el que la idea inicial será tener tres tipos de módulos o archivos ASCII-BSR:

1. primary: los módulos primarios son capaces de leer un sólo segmento de regresión (nodo en el argot jerárquico) basándose en código TOL para la construcción de la matriz. Normalmente lo usaremos para nodos densos observacionales pero eso a BSR no le importa en absoluto, pues no sabe qué cosa es observacional ni latente ni nada. En esos módulos sólo cabe pues un nodo, con su declaración de variables lineales y missing, estructura de la varianza (sigma-ARIMA), sus filtros no lineales y sus restricciones de desigualdad internas, es decir, todo aquello que no afecta a ningún otro. Un módulo primario puede funcionar de forma autónoma, por ejemplo cuando queremos simular un nodo observacional por sí mismo, sin tener en cuenta estructuras jerárquicas, a priori, ni nada por el

estilo, aunque eso no creo que se dé muy a menudo, pues lo normal es que la menos haya información a priori.

2. joint: los módulos de mezcla serían los actualmente permitidos, llamados así por su capacidad de mezclar variables de varios segmentos de regresión. Pueden funcionar autónomamente como hasta ahora, definiendo todas las variables y ecuaciones de forma explícita, o bien integrarse dentro de un módulo máster que se encargue de cargar previamente las variables de las que depende.
3. master: los módulos maestros simplemente incluyen a otros módulos de cualquier tipo, incluidos otros maestros, si se desea tener los modelos estructurados en varios niveles, de forma que en cada nivel cada uno de los sub-master podría ser ejecutado autónomamente.

11.3.1. Estructura del módulo primario. Un módulos primario se definirá del siguiente modo:

1. En primer lugar se incluye una cláusula que indique el tipo de módulo que se va a definir

   Module.Type = primary;

2. Las variables del bloque lineal se definen con la misma sintaxis que rige actualmente para el módulo joint.
3. El segmento de regresión es único pero se define con la misma sintaxis que en el módulo joint. En ambos casos hay una novedad: la posibilidad de introducir los filtros no lineales si los hay en la propia definición del ruido. La estructura del campo `NoiseDistrib` será distinta a la actual `BSR.NoiseDistrib` para que pueda incluir un nuevo campo en el que almacenar los filtros no lineales.
4. Luego se introducirán la matriz de output y la de input. El orden de definición de las variables del bloque lineal debe coincidir con el orden de las columnas en la matriz de inputs.
5. Después se introducirán si las hay, las variables de omitidos con una sintaxis simular a la de los módulos joint pero añadiéndoles la información de su ubicación en las matrices de input u output, las cuales deberían tener un valor omitido en la celda correspondiente, cosa que debe por tanto comprobarse.
6. Por último se añadirán si las hay, las restricciones de desigualdad sobre las variables del bloque lineal.

Obviamente el Import deberá saber que está escribiendo un módulo primario y pedir las matrices en lugar de los coeficientes así como la ubicación de los omitidos.

A continuación se presenta una plantilla de cómo podría quedar el fichero ASCII de un módulo primario.

```
1  ////////////////////////////////////////////////////////////////////////
2  // Bayesian Sparse Regression
3  ////////////////////////////////////////////////////////////////////////
4
5  $BEGIN
6
7  Module.Type = primary;
8
9  Model.Name = "...";
10 Model.Description = "...";
11 Session.Name = "...";
12 Session.Description = "...";
13 Session.Authors = "...";
14
15 ////////////////////////////////////////////////////////////////////////
16 // Defining header of Regression Variables
17 ////////////////////////////////////////////////////////////////////////
18
19 LinearBlk::<node_name>::<variable-name>::Coef <-
20   {+ or -}<initial value or ?>;
21 ...
22 LinearBlk::<node_name>::<variable-name>::Coef <-
23   {+ or -}<initial value or ?>;
24
25 ////////////////////////////////////////////////////////////////////////
26 // Defining header of Noise Distribution
27 ////////////////////////////////////////////////////////////////////////
```

```
28
29  <node_name>::Noise [length and optional time info]
30  ~ <distribution>
31  // Optional defininition of Non Linear Filters
32  with non linear flters {'Set of NameBlock's expresion'};
33
34  ////////////////////////////////////////////////////////////////////////////
35  // Defining Regression Equations
36  ////////////////////////////////////////////////////////////////////////////
37
38  Output = {'Matrix or VMatrix expresion'};
39  Input = {'Matrix or VMatrix expresion'};
40
41  ////////////////////////////////////////////////////////////////////////////
42  // Optional defininition of Ouput Missing Variables
43  ////////////////////////////////////////////////////////////////////////////
44
45  MissingBlk::<MIXTURE or node name>::<missing_name> ?
46    at row <row_index> for output
47    ~ <distribution>;
48  ...
49  MissingBlk::<MIXTURE or node name>::<missing_name> ?
50    at row <row_index> for output
51    ~ <distribution>;
52
53  ////////////////////////////////////////////////////////////////////////////
54  // Defining defininition of Input Missing Variables
55  ////////////////////////////////////////////////////////////////////////////
56
57  MissingBlk::<MIXTURE or node name>::<missing_name> ?
58    at row <row_index> for input <column_index>
59    ~ <distribution>;
60  ...
61  MissingBlk::<MIXTURE or node name>::<missing_name> ?
62    at row <row_index> for input <column_index>
63    ~ <distribution>;
64
65  ////////////////////////////////////////////////////////////////////////////
66  // Defining Parameter Constraints
67  ////////////////////////////////////////////////////////////////////////////
68
69  {+|-}<number> {<=|>=}
70    {+|-}<number>*LinearBlk::<node_name>::<variable-name>::Coef
71    ...
72    {+|-}<number>*LinearBlk::<node_name>::<variable-name>::Coef;
73  ...
74
75  $END
```

11.3.2. Cambios en la estructura del módulo joint. El módulo joint se queda prácticamente como está con estas salvedades:

1. Es posible, aunque no obligatorio por compatibilidad hacia atrás, especificar el tipo de módulo en la cabecera del archivo:

   Module.Type = joint;

2. En un módulo joint puede aparecer una referencia a una variable que ha sido definida en otro módulo previo. El reconocedor sintáctico permitirá pues que aparezcan en las ecuaciones e inecuaciones variables no definidas. En la estructura BSR.LinearBlock se añade un campo Real Extern cuyo valor será puesto a TRUE en estos casos para que puedan resolverse más tarde las referencias.

11.3.3. Mezcla de módulos en el master. El master debe ocuparse de mezclar los modelos o submodelos resultantes de de los módulos en un solo modelo BSR. El modo de hacerlo depende del mecanismo de generación que se vaya a seguir, que puede ser de dos formas

11.3.3.1. Mecanismo de generación monofase. Es el único mecanismo de generación implementado actualmente, en el que todos los segmentos y variables lineales se aúnan en un solo sistema lineal de ecuaciones sparse.

En este caso los BSR.ModelDef que resultan de cada módulo se deben agregar en un sólo BSR.ModelDef que defina el modelo completo siguiendo los isguientes pasos con sumo cuidado. Se indicará al principio del módulo máster mediante la sintaxis

```
Modular.Schema = monophasic;
```

1. La información documental (campo DocInfo) que prevalece es la del módulo master mientras que la de los sub-módulos simplemente se olvida.
2. Se obtiene la lista conjunta de variables (campo {{{LinearBlock}}}) del bloque lineal de cada uno de los módulos sin repeticiones.
3. Se reubican las posiciones de las variables en cada uno de los módulos 1. Se modifica el número de columnas y las posiciones de las columnas de la matriz X de inputs para hacerlas coincidir con la posición de las variables en la lista global.
4. Se modifica el número de columnas y las posiciones de las columnas de la matriz A de las inecuaciones para hacerlas coincidir con la posición de las variables en la lista global.
5. Se modifica el campo Col de los registros BSR.MissingBlock de definición de omitidos del input en cada módulo 1. Se modifica el número de filas y columnas de las matrices Cov, L y Li de cada registro BSR.NoiseDistrib correspondiente a cada segmento, así como el conjunto de índices EquIdx de las filas a las que correponde.
6. Se concatenan las filas de las matrices Y[k], X[k], a[k], A[k] de todos los módulos en las correspondientes Y, X, a, A conjuntas.
7. Se concatenan los conjuntos de omitidos del input (campo InputMissingBlock) y el output (campo OutputMissingBlock)
8. Se concatenan los conjuntos de segmentos de regresión (campo NoiseDistrib)

Intentar hacer todas estas operaciones mientras se está leyendo un BSR actual es algo extremadamente complicado de implementar, pues algunas cosas aún no existen y otras están a medias, no se conocen algunas dimensiones, etc. Precisamente por esta razón propongo todo este tema de la modularización de la sintaxis de los ficheros ASCII-BSR.

11.3.3.2. Mecanismo de generación multifase. En este caso cada módulo se generará condicionalmente al resto de módulos, bien secuencialmente, bien en paralelo cuando tal cosa sea posible. Esto no es necesario que se implemente en una primera versión pero se puede ir adelantando algo del diseño. Se indicará al principio del módulo máster mediante la sintaxis

```
Modular.Schema = sequential;
```

ó bien

```
Modular.Schema = parallel;
```

según sea el caso.

En ambos casos la mezcla de módulos es distinta al caso de generación monofase, pues en realildad no se hace agregación física de los módulos sino que simplemente se enlazarán las dependencias de variables entre ellos para que luego BSR pueda actualizar sus valores en las sucesivas simulaciones condicionadas, de forma similar a como lo ahcía el antiguo HLM, del cual se podrán tomar bastantes ideas.

11.3.3.3. Plantilla de un módulo master. Aunque cada módulo internamente especifica con la cláusula Module.Type de que tipo de módulo se trata es conveniente repetir esa información en la sentencia de inclusión del módulo hijo en el módulo master para llamar directamente al parser adecuado, ya que en realidad lo que habrá es 3 parsers distintos, el de módulos primarios, el de los joint, que es básicamente el actual y el de los masters. La cláusula Module.Type puede usarse como mecanismo de control para evitar posibles errores de diseño del modelo. Así podría quedar la sintaxis de los módulos master

```
1  ////////////////////////////////////////////////////////////////////////////
2  // Bayesian Sparse Regression
3  ////////////////////////////////////////////////////////////////////////////
4  $BEGIN
5  Module.Type = master;
6  Model.Name = "...";
7  Model.Description = "...";
8  Session.Name = "...";
9  Session.Description = "...";
10 Session.Authors = "...";
11 Modular.Schema = { monophasic, sequential or parallel } ;
```

```
12  Include { primary, joint or master } module "file_relative_path.bsr" ; ... Include { primary, joint or master }
        module "file_relative_path.bsr" ;
13  $END
```

CAPíTULO 12

API BysMcmc::Bsr::Import

Se describen los miembros y métodos generales del NameBlock BysMcmc::Bsr::Import

12.1. Descripción

El `API BysMcmc::Bsr::Import`, en lo adelante `API Import`, es un nivel de abstracción que permite definir modelos **BSR** sin necesidad de conocer el formato en archivo de dichos modelos. Semánticamente esta orientado a los conceptos ya definidos en la sección 1.2 y se centran en los segmentos de regresión y la estructura del error.

Por recordar, un modelo **BSR** define una regresion lineal restringida compuesta de segmentos de regresion con errores independientes entre sí y un sistema de restricciones lineales entre los parámetros.

El `API Import` es un mecanismo para definir las caracteristicas de cada uno de los segmentos de regresión tales como:

- estructura temporal de las observaciones
- distribución del error
- parámetros lineales
- omitidos en el output e inputs
- restricciones entre los parámetros

Para cada una de las categorías anteriores deben definirse un conjunto de métodos que dan acceso a los elementos de las mismas, por ejemplo en la definición de los parámetros lineales de un segmento debemos tener un método para retornar el número de parámetros del segmento y otro para retornar la información del i-ésimo parámetro. Este estilo de interfaz se ha escogido para dar flexibilidad a la hora de definir el modelo (amplia un poco más esto).

De forma global el modelo **BSR** debe satisfacer también unos requerimientos de interfaz como son retornar el número de segmentos que lo componen, el i-ésimo segmento y las restricciones entre parámetros de diferentes bloques. El segmento retornado debe ser un `NameBlock` que cumpla con el API del segmento requerida.

En las secciones siguientes describiremos en detalle los metodos que deben implementarse a nivel global del modelo BSR como a nivel de segmento. Finalmente describiremos mediante un ejemplo el uso del `API Import`.

12.2. NameBlock del sistema BSR

Un sistema BSR se implementa a través de un NameBlock que debe responder a un interfaz que permita conocer atributos como: nombre del modelo, directorio de salida del modelo .bsr, nombre de la session, número de segmentos de regresion así como el acceso a cada uno de los segmentos, entre otros.

A partir de un NameBlock o instancia que cumpla los requeriemiento del API Import podemos generar el archivo .bsr asociado mediante la función `Text BysMcmc::Bsr::Import::Write(NameBlock BSR)`. Esta función retorna el camino del archivo generado.

El nombre del archivo y la localización es determinado haciendo uso de metodos que deben estar implementados en el NameBlock

A continuación se describe en detalles los métodos requeridos en el API de un `NameBlock` para un sistema `BSR`. Para ligar la descripción a un ejemplo concreto usaremos el siguiente modelo simple $ARMA(2,1)$ para un nodo de observación:

$$
\begin{aligned}
y_t &= \beta_0 + \textstyle\sum_1^k \beta_i x_{t,i} + z_t \\
\phi\left(B\right) z_t &= \theta\left(B\right) \varepsilon_t, \varepsilon_t \sim N(0, \sigma^2) \\
\phi(B) &= 1 - \phi_1 B - \phi_2 B^2 \\
\theta(B) &= 1 - \theta_1 B \\
\beta_1 &\geq \beta_2
\end{aligned}
\tag{12.2.1}
$$

Get.Doc.Model.Name: el método `Text Get.Doc.Model.Name(Real void)` debe retornar un identificador del modelo. Ejemplo:

```
Text Get.Doc.Model.Name(Real void)
{
  Text "BSRExample"
};
```

Get.Doc.Model.Description: el método `Text Get.Doc.Model.Description(Real void)` implementa una función que retorna una descripción del modelo.

```
Text Get.Doc.Model.Description(Real void)
{
  Text "Modelo simple ARMA(2,1)"
};
```

Get.Doc.Session.Name: el método `Text Get.Doc.Session.Name(Real void)` implementa una función que retorna un identificador de la sesión. Para un mismo modelo podemos tener diferentes sesiones que consiste en una parametrización distinto. Poder mantener diferentes sesiones de un modelo nos permite almacenar y comparar los resultado para diferentes versiones del modelo. El modelo BSR para una sesión dada se escribe en directorio indicado por el valor de retorno de la función `Text Get.Doc.Path(Real void)`. Ejemplo:

```
Text Get.Doc.Session.Name(Real void)
{
  Text _.session.id
};
```

Get.Doc.Session.Description: el método `Text Get.Doc.Session.Description(Real void)` implementa una función que retorna una descripción de la sesión. Ejemplo:

```
Text Get.Doc.Session.Description(Real void)
{
  Text "Session "+_.session.id+" para el modelo: " +
       Get.Doc.Model.Name(0) + " " + Get.Doc.Model.Description(0)
};
```

Get.Doc.Session.Authors: el método `Text Get.Doc.Session.Authors(Real void)` retorna una cadena que contenga los autores de la sesión. Ejemplo:

```
Text Get.Doc.Session.Authors(Real void)
{
  Text "user@bayesforecast.com"
};
```

Get.Doc.Path: el método `Text Get.Doc.Path(Real void)` retorna el directorio de salida donde se escribirá el archivo .bsr asociado al modelo. Ejemplo:

```
Text Get.Doc.Path(Real void)
{
  Text _.path.out
};
```

Get.LinReg.Size: el método `Real Get.LinReg.Size(Real void)` retorna el número de segmentos independientes del sistema BSR. Ejemplo:

```
Real Get.LinReg.Size(Real void)
{
  // para el ejemplo actual solo tenemos un segmento de regresión
  Real 1
};
```

Get.LinReg.Segment: el método `NameBlock Get.LinReg.Segment(Real iS)` retorna el `NameBlock` que implementa el API import para el segmento con ndice `iS`. El API que debe satisfacer este `NameBlock` lo describimos en la sección 12.3. Ejemplo:

```
NameBlock Get.LinReg.Segment(Real iS)
{
  // en este caso retornamos el NameBlock para el único segmento
  // que se modela para mas de un segmento de regresión se suele
  // usar un Set que es indexado por iS: _bsr.blocks[iS]
  NameBlock _.bsr.ARMA.2_1
};
```

Get.Constraints.Handler: el método `NameBlock Get.Constraints.Handler(Real void)` retorna el `NameBlock` que implementa el interfaz necesaria para definir las restricciones entre parametros de diferentes segmentos. En la sección 12.4 se describe el API que debe satisfacerse. Para nuestro ejemplo como solo tenemos un segmento de regresión las restricciones están definidas en dicho segmento, no obstante también las podíamos haber definido en el las restricciones globales. Ejemplo:

```
NameBlock Get.Constraints.Handler(Real void)
{
  // retornamos un NameBlock predefinido para el caso
  // no restringido
  NameBlock BysMcmc::Bsr::Import::Unconstrained(0)
};
```

Finalmente el una función que retorna el NameBlock que implementa el API del sistema BSR del para el modelo anterior sería similar a:

```
NameBlock Build.Example.BSR(Text path.out, Text session.id,
                            Serie Y, Set X)
{
  NameBlock [[
    Text _.path.out = path.out;
    Text _.session.id = session.id;
    NameBlock _.bsr.ARMA.2_1 = [[
      Text _.dating.name = DatingName(Y);
      Date _.first = First(Y);
      Date _.last = Last(Y),
      Matrix _.Y = SerMat(Y);
      Matrix _.X = SerSetMat(X);
      /* ... */
      /* aquí implementamos el NameBlock para el nodo de observación */
    ]];

    Text Get.Doc.Model.Name(Real void)
    {
      Text "BSRExample"
    };
```

```
    Text Get.Doc.Model.Description(Real void)
    {
      Text "Modelo simple ARMA(3,1)"
    };

    Text Get.Doc.Session.Name(Real void)
    {
      Text _.session.id
    };

    Text Get.Doc.Session.Description(Real void)
    {
      Text "Session "+_.session.id+" para el modelo: " +
           Get.Doc.Model.Name(0) + " " + Get.Doc.Model.Description(0)
    };

    Text Get.Doc.Session.Authors(Real void)
    {
      Text "user@bayesforecast.com"
    };

    Text Get.Doc.Path(Real void)
    {
      Text _.path.output
    };

    Real Get.LinReg.Size(Real void)
    {
      // para el ejemplo actual solo tenemos un segmento de regresión
      Real 1
    };

    NameBlock Get.LinReg.Segment(Real iS)
    {
      // en este caso retornamos el NameBlock para el único segmento
      // que se modela para mas de un segmento de regresión se suele
      // usar un Set que es indexado por iS: _bsr.blocks[iS]
      NameBlock _.bsr.ARMA.2_1
    };

    NameBlock Get.Constraints.Handler(Real void)
    {
      // retornamos un NameBlock predefinido para el caso
      // no restringido
      NameBlock BysMcmc::Bsr::Import::Unconstrained(0)
    }
  ]]
};
```

12.3. API del segmento BSR

Después de haber visto el API global del un sistema BSR, ahora pasamos al API de un segmento BSR concreto

Get.Name: la función `Get.Name(Real void)` retorna el nombre del segmento BSR. Debe ser un nombre único entre todos los segmentos del sistema BSR. En el ejemplo que llevamos pudiera quedar así:

```
Text Get.Name(Real void)
{
  Text "ObsY"
};
```

Get.Sigma2: `la función Anything Get.Sigma2(Real void)` retorna un identificador o un valor para el parámetro σ del segmento de regresión. Si se retorna un `Text` entonces se asume un identificador de parámetroσ desconocido y se simulará según su distribución condicional, en tal caso el identificador debe ser único entre todos los nombres de parémetros σ de todos los segmentos. Si el valor retornado es un `Real` entonces se asume una σ conocida y no se simula. En nuestro ejemplo σ es desconocido por tanto le damos un nombre:

```
Text Get.Sigma2(Real void)
{
  Text "sigma2"
};
```

Get.TimeInfo: si el segmento tiene estructura temporal, el método `Set Get.TimeInfo(Real void)` retorna información asociada al fechado del segmento. Ejemplo:

```
Set Get.TimeInfo(Real void)
{
  Set BSR.NoiseTimeInfo(TimeSet Eval(_.dating.name), _.first, _.last)
};
```

el objeto resultado debe ser de tipo `BSR.NoiseTimeInfo` que es una estructura definida internamente en TOL y que tiene la siguiente definición:

```
Struct BSR.NoiseTimeInfo
{
  TimeSet Dating,
  Date FirstDate,
  Date LastDate
};
```

Get.ARIMA.Size: la función `Get.ARIMA.Size(Real void)` retorna el número de factores `ARIMA` para el segmento `BSR`. Si el segmento es no dinámico o es un ruido blanco entonces se debe retornar 0. Recuérdese que en la versión actual un factor `ARIMA` no debe superar el grado 2. Un polinomio de grado superior a 2 debe darse descompuesto en factores de grado 2 ó 1. Para el modelo 12.2 con un factor bastaría.

```
Real Get.ARIMA.Size(Real void)
{
  Real 1
};
```

Get.ARIMA.Factor: la función `Get.ARIMA.Factor(Real f)` retorna el f-ésimo factor del ruido `ARIMA`. El valor del argumento `f` va desde 1 hasta `n=Get.ARIMA.Size(0)`. El valor de retorno de la función debe ser un Set con estructura `ARIMAStruct`:

```
Struct ARIMAStruct
{
  Real Periodicity,
  Polyn AR,
  Polyn MA,
  Polyn DIF
```

```
};
```

Para el modelo 12.2 la función sería:

```
Set Get.ARIMA.Factor(Real f)
{
  Set ARIMAStruct(1, 1-0.1*B-0.1*B^2, 1-0.1*B, 1)
};
```

Get.Param.Size: la función `Real Get.Param.Size(Real void)` retorna el número de parámetros lineales de la regresión. En el caso del ejemplo que llevamos se corresponde `Rows(X)=3` ya que los variables predictivas están almacenadas por filas:

```
Real Get.Param.Size(Real void)
{
  Real Rows(_.X)
};
```

Get.Param: la función `Set Get.Param(Real numParam)` retorna la información para el i-ésimo parámetro. El valor del argumento `iP` va desde `1` hasta `n=Get.Param.Size(0)`. El valor de retorno de la función es `Set` con estructura `Bsr.Param.Info`:

```
Struct Bsr.Param.Info
{
  Text Name,              // identificador único del parámetro
  Real InitValue,         // valor inicial conocido o ?
  Real Prior.LowerBound,  // cota inf. del intervalo de definición o -1/0
  Real Prior.UpperBound   // cota sup. del intervalo de definición o +1/0
};
```

En nuestro caso retornamos los nombres `beta_1`, `beta_2`,..., `beta_k` y todos no acotados:

```
Set Get.Param(Real iP)
{
  Set Bsr.Param.Info("beta_"+IntText(iP), Real Rand(0,1), -1/0, 1/0)
};
```

Get.Missing.Size: la función `Real Get.Missing.Size(Real void)` retorna el número de parámetros asociados a los valores omitidos de las observaciones y variables. En nuestro caso no tenemos omitidos por lo que retornamos 0.

```
Real Get.Missing.Size(Real void)
{
  Real 0
};
```

Get.Missing: la función `Set Get.Missing(Real iP)` retorna la información acerca del i-ésimo valor/parámetro omitido. El valor del argumento `iP` va desde `1` hasta `n=Get.Missing.Size(0)`. El valor de retorno es la información a priori para este parámetro y que será usada en el muestreo del parámetro. Esta información es dada como un Set con estructura Bsr.Missing.Info:

```
Struct Bsr.Missing.Info
{
  Text Name,              // identificador único para el valor missing
  Real Prior.Average,     // prior media or initial value
  Real Prior.Sigma,       // sigma del prior or +1/0 si no inf.
  Real Prior.LowerBound,  // límite inf. del intervalo o -1/0
  Real Prior.UpperBound   // límite sup. del intervalo o +1/0
};
```

como no tenemos omitidos retornamos el conjunto vacío:

```
Set Get.Missing(Real iP)
{
  Set Empty
};
```

Get.Equation.Size: la función `Real Get.Equation.Size(Real void)` retorna el número de ecuaciones del segmento. Para nuestro ejemplo tendríamos la implementación:Para nuestro ejemplo como solo tenemos un segmento de regresión las restricciones están definidas en dicho segmento, no obstante también las podíamos haber definido en el las restricciones globales.

```
Real Get.Equation.Size(Real void)
{
  // _.X e _.Y están dispuestas por filas
  Real Columns(_.X);
};
```

Get.Equation.Output: la función `{Text|Real} Get.Equation.Output(Real iE)` retorna el valor de la observación i-ésima para el segmento. El valor del parámetro `iE` va desde `1` hasta `n=Get.Equation.Size(0)`. El valor de retorno puede ser un `Text` o un `Real`. Cuando el valor de la observación es omitido debemos retornar el identifcador de parámetro definido con `Get.Missing`. Si el valor es conocido se debe retornar el valor observado. En nuestro ejemplo las observaciones están en la variable `_.Y`:

```
Real Get.Equation.Output(Real iE)
{
  Real MatDat(_.Y,1,iE)
};
```

Get.Equation.Input.Size: la función `Real Get.Equation.Input.Size(Real iE)` retorna el número de términos distintos de 0 que tiene la ecuación i-esima. El valor del parámetro `iE` va desde 1 hasta `n=Get.Equation.Size(0)`. En nuestro ejemplo la matriz `X` es densa y todas las ecuaciones tienen el mismo número de términos distintos de `0`:

```
Real Get.Equation.Input.Size(Real iE)
{
  Real Rows(_.X)
};
```

Get.Equation.Input.Coef: la función `{Text|Real} Get.Equation.Input.Coef(Real iE, Real iT)` retorna el coeficiente t-ésimo término de la e-ésima ecuación. El valor del parámetro `iE` va desde `1` hasta `n=Get.Equation.Size(0)`, mientras que el valor del parámetro `iT` va desde `1` hasta `n=Get.Equation.Input.Size(0)`. El valor de retorno puede ser un `Text` o un `Real`. Cuando el valor de la variable es omitido debemos retornar el identificador de parámetro definido con `Get.Missing`. Si el valor es conocido se debe retornar el valor observado de la variable. En nuestro ejemplo todos los valores son conocidos y los coeficiente de las ecuaciones están por columnas:

```
Real Get.Equation.Input.Coef(Real iE, Real iT)
{
  Real MatDat(_.X, iT,iE)
};
```

Get.Equation.Input.Param: la función `Text Get.Equation.Input.Param(Real iE, Real iT)` retorna el identificador del parámetro del t-ésimo término en la e-ésima ecuación. Los identificadores de parámetros pueden estar definidos en cualquiera de los segmentos del sistema BSR. En nuestro ejemplo los identificadores de parámetros son del tipo `beta_k`:

```
Text Get.Equation.Input.Param(Real iE, Real iT)
{
  Text "beta_"+IntText(iT)
```

```
};
```

Get.Constraints.Handler: la función `NameBlock Get.Constraints.Handler(Real void)` el método `NameBlock Get.Constraints.Handler(Real void)` retorna el `NameBlock` que implementa el interfaz necesaria para definir las restricciones entre parametros de este segmentos. En la sección 12.4 se describe el API que debe satisfacerse. En el modelo planetado en 12.2 solo tenemos una restricción lineal de tipo orden $\beta 1 \geq \beta 2$ y pudiera quedar:

```
NameBlock Get.Constraints.Handler(Real void)
{
  NameBlock Explicit.Constraints([["0>=-beta_1+beta_2"]])
};
```

12.4. Restricciones en BSR

Las restricción en un sistema `BSR` la podemos definir dentro de un segmento de regresión en el caso en que sean restricciones que involucren a parámetros propios de ese segmento o a nivel global si es que se involucran parámetros de diferentes segmentos.

La definición de las restricciones es dada mediante un `NameBlock` que debe satisfacer unos equerimientos de interfaz. Primeramente definimos unas estructuras que son necesarias en el API de restricciones.

Bsr.OrderRelation.Info: es una estructura para codificar las relaciones de orden entre dos parámetros

```
////////////////////////////////////////////////////////////////////
// Lower <= Upper
////////////////////////////////////////////////////////////////////
Struct Bsr.OrderRelation.Info
{
  Text Lower, //Left parameter
  Text Upper  //Right parameter
};
```

por ejemplo en nuestro caso tenemos la restricción $\beta 1 \geq \beta 2$ la cual quedaría codificada de esta manera: `Set Bsr.OrdeRelation.Info("beta_2","beta_1")`

Bsr.LinearCombTerm: es una estructura para codificar un término de una combinación lineal de parámetros

```
////////////////////////////////////////////////////////////////////
// Coefficient * Parameter
////////////////////////////////////////////////////////////////////
Struct Bsr.LinearCombTerm
{
  Real Coefficient,
  Text Parameter
};
```

Bsr.GenericConstraint.Info: es una estructura para codificar una combinación lineal de parámetros acotada a un intervalo.

```
////////////////////////////////////////////////////////////////////
// LowerBound <= c_1*alfa_1 + ... + c_n*alfa_n <= UpperBound
////////////////////////////////////////////////////////////////////
Struct Bsr.GenericConstraint.Info
{
  Real LowerBound, // cota inferior de la combinación lineal o -1/0
  Set  LinearComb, // un conjutno de Bsr.LinearCombTerm
  Real UpperBound  // cota superior de la combinación lineal o +1/0
```

```
};
```

A continuación describimos los requerimientos del API de restricciones expresadas en un conjunto de funciones que deben estar implementadas en el NameBlock:

Get.OrderRelation.Size: la función `Real Get.OrderRelation.Size (Real void)` retorna el número de restricciones de orden. Para nuestro ejemplo tenemos solo una:

```
Real Get.OrderRelation.Size (Real void)
{
  Real 1
};
```

Get.OrderRelation: la función `Set Get.OrderRelation (Real r)` retorna la r-ésima relación de orden entre dos parámetros. El parámetro r va desde 1 hasta `n=Get.OrderRelation.Size(0)`. El valor de retorno es una estructura del tipo `Bsr.OrdeRelation.Info`. Siguiendo nuestro ejemplo:

```
Set  Get.OrderRelation (Real r)
{
  Set Bsr.OrdeRelation.Info("beta_2","beta_1")
};
```

Get.GenericConstraint.Size: la función `Real Get.GenericConstraint.Size (Real void)` retorna el número de restricciones genéricas. Las restricciones genéricas son aquella que no pueden definirse como restricciones de dominio de un parámetro o como relaciones de orden entre dos parámetros. Este es el tipo de restricción líneal más general que podemos expresar y por tanto los otros tipos de restricciones, como las de orden, pueden expresarse también de este tipo. En nuestro caso como ya hemos definido la restricción como de orden no la definimos como genérica:

```
Real Get.GenericConstraint.Size (Real void)
{
  Real 0
};
```

Get.GenericConstraint: la función `Set Get.GenericConstraint (Real r)` retorna la r-ésima restricción genérica. El argumento `r` va desde 1 hasta `n=Get.GenericConstraint.Size(0)`. El valor de retorno es un `Set` con estructura `Bsr.GenericConstraint.Info`. Por ejemplo la restricción de orden ya definida anteriormente podríamos definirla de esta manera:

```
Set  Get.GenericConstraint (Real r)
{
  Set Bsr.GenericConstraint.Info
  ( 0.0,
    [[Set Bsr.LinearCombTerm(1.0,  "beta_1"),
      Set Bsr.LinearCombTerm(-1.0, "beta_2")]],
    +1/0 )
};
```

Get.ExplicitConstraint.Size: la función `Real Get.ExplicitConstraint.Size(Real void)` retorna el número de restricciónes explícitas. Las restricciones explícitas son restricciones en formato texto tal y como se escriben en el archivo `.bsr`. Para nuestro ejemplo no definimos ninguna restricción del tipo explícita ya que han sido convenientemente definidas anteriormente:

```
Real Get.ExplicitConstraint.Size(Real void)
{
  Real 0
};
```

Get.ExplicitConstraint: la función `Text Get.ExplicitConstraint(Real r)` retorna la r-ésima restricción explícita.

```
Text Get.ExplicitConstraint(Real r)
{
  Text ""
};
```

Se dan también funciones de conveniencia para construir un `NameBlock` que satisfaga el `API` de definición de restricciones. Estas funciones son:

Order.Relations: la función `NameBlock Order.Relations(Set order.relations)` crea un `NameBlock` de restricciones en la que solo están definidas restricciones de orden. Ejemplo:

```
NameBlock Order.Relations([[Bsr.OrdeRelation.Info("beta_2","beta_1")]])
```

Generic.Constraints: la función `NameBlock Generic.Constraints(Set generic.constraints)` crea un `NameBlock` de restricciones en la que solo están definidas restricciones genéricas. Ejemplo:

```
NameBlock Generic.Constraints
(
[[
  Bsr.GenericConstraint.Info
  ( 0.0,
    [[Set Bsr.LinearCombTerm(1.0,  "beta_1"),
      Set Bsr.LinearCombTerm(-1.0, "beta_2")]],
    +1/0 )
]]
)
```

Explicit.Constraints: la función `NameBlock Explicit.Constraints(Set explicit.constraints)` crea un `NameBlock` de restricciones en la que solo están definidas restricciones explícitas. Ejemplo:

```
NameBlock Explicit.Constraints([["0<=+beta_1-beta_2"]])
```

Constraints: la función `NameBlock Constraints(Set order.relations,Set generic.constraints,Set explicit.constraints)` crea un `NameBlock` de restricciones a partir de los 3 tipos de restricciones posibles. Ejemplo:

```
NameBlock Constraints(Empty,Empty,[["0<=+beta_1-beta_2"]])
```

Unconstrained: la función `NameBlock Unconstrained(Real void)` crea un `NameBlock` de restricciones vacío. Ejemplo:

```
NameBlock Unconstrained(0)
```

12.5. Ejemplo de API Import

El código completo para el ejemplo dado en las ecuaciones 12.2 quedaría de la siguiente manera:

```
NameBlock Build.Example.BSR(Text path.out, Text session.id,
                            Serie Y, Set X)
{
  Real beta1 = Rand(-1,1);
  Real beta2 = Rand(-1,beta1);
  NameBlock [[
    Text _.path.out = path.out;
    Text _.session.id = session.id;
    NameBlock _.bsr.ARMA.2_1 = [[
      Text _.dating.name = DatingName(Y);
```

```
_.first = First(Y);
_.last = Last(Y),
Matrix _.Y = SerMat(Y);
Matrix _.X = SerSetMat(X);
Set _.beta.ini = [[beta1, beta2]];

Text Get.Name(Real void)
{
  Text "ObsY"
};

Text Get.Sigma2(Real void)
{
  Text "sigma2"
};

Set Get.TimeInfo(Real void)
{
  Set BSR.NoiseTimeInfo(TimeSet Eval(_.dating.name), _.first, _.last)
};

Real Get.ARIMA.Size(Real void)
{
  Real 1
};

Set Get.ARIMA.Factor(Real f)
{
  Set ARIMAStruct(1, 1-0.1*B-0.1*B^2, 1-0.1*B, 1)
};

Real Get.Param.Size(Real void)
{
  Real Rows(_.X)
};

Set Get.Param(Real iP)
{
  Real value = If(iP>2,Real Rand(-1,1),_.beta.ini[iP]);
  Set Bsr.Param.Info("beta_"+IntText(iP), value, -1/0, 1/0)
};

Real Get.Missing.Size(Real void)
{
  Real 0
};

Set Get.Missing(Real iP)
{
  Set Empty
};

Real Get.Equation.Size(Real void)
{
```

```
    // _.X e _.Y están dispuestas por filas
    Real Columns(_.X)
  };

  Real Get.Equation.Output(Real iE)
  {
    Real MatDat(_.Y,1,iE)
  };

  Real Get.Equation.Input.Size(Real iE)
  {
    Real Rows(_.X)
  };

  Real Get.Equation.Input.Coef(Real iE, Real iT)
  {
    Real MatDat(_.X, iT,iE)
  };

  Text Get.Equation.Input.Param(Real iE, Real iT)
  {
    Text "beta_"+IntText(iT)
  };

  NameBlock Get.Constraints.Handler(Real void)
  {
    NameBlock BysMcmc::Bsr::Import::Explicit.Constraints([["0>=-beta1+beta2"]])
  }
]];

Text Get.Doc.Model.Name(Real void)
{
  Text "BSRExample"
};

Text Get.Doc.Model.Description(Real void)
{
  Text "Modelo simple ARMA(2,1)"
};

Text Get.Doc.Session.Name(Real void)
{
  Text _.session.id
};

Text Get.Doc.Session.Description(Real void)
{
  Text "Session "+_.session.id+" para el modelo: " +
       Get.Doc.Model.Name(0) + " " + Get.Doc.Model.Description(0)
};

Text Get.Doc.Session.Authors(Real void)
{
  Text "user@bayesforecast.com"
```

```
    };

    Text Get.Doc.Path(Real void)
    {
      Text _.path.out
    };

    Real Get.LinReg.Size(Real void)
    {
      // para el ejemplo actual solo tenemos un segmento de regresión
      Real 1
    };

    NameBlock Get.LinReg.Segment(Real iS)
    {
      // en este caso retornamos el NameBlock para el único segmento
      // que se modela para mas de un segmento de regresión se suele
      // usar un Set que es indexado por iS: _bsr.blocks[iS]
      NameBlock _.bsr.ARMA.2_1
    };

    NameBlock Get.Constraints.Handler(Real void)
    {
      // retornamos un NameBlock predefinido para el caso
      // no restringido
      NameBlock BysMcmc::Bsr::Import::Unconstrained(0)
    }
  ]]
};
```

12.6. API modular

Como se ha visto anteriormente 11.3, desde la versión 2.0.1 de TOL es posible definir un modelo BSR de forma modular. Para que el Import sea capaz de crear este tipo de expresión de modeos se le han añadido una serie métodos que entrarán en acción si el NameBlock de definición del usuario implementa a su vez los siguientes métodos opcionales

Get.ModuleType: retorna el tipo de módulo que puede ser "primary","joint" o "master". Si no existe método se sobreentiende "joint" para guardar compatibilidad hacia atrás

```
Text Get.ModuleType(Real unused)
```

Si se trata de un módulo de tipo "master", entonces deberá implementar estos métodos en lugar de los definidos en la sección anterior y que corresponden al actual "joint"

Get.Modular.Schema: retorna el esquema modular o modo en el que se resolverán las simulaciones que puede ser "monophasic", "sequential" ó "parallel"

```
Text Get.Modular.Schema(Real unused)
```

Get.Modules: retorna el conjunto de módulos, cada uno de los cuales es un NameBlock utilizable a su vez por el Import de forma recursiva

```
Set Get.Modules(Real unused)
```

Si se trata de un módulo de tipo "primary", entonces sólo puede tener un segmento el cual debe tener estos métodos en lugar de los descritos en la sección Get.Equation.Output, Get.Equation.Input.Size, Get.Equation.Input.Coef y Get.Equation.Input.Param

Get.OutputVMatrix: devuelve la matriz de output del segmento de regresión

```
VMatrix Get.OutputVMatrix(Real unused)
```

Get.InputVMatrix: devuelve la matriz de input del segmento de regresión

```
VMatrix Get.InputVMatrix(Real unused)
```

El método Get.Equation.Size se mantiene porque el reconocedor sintáctico no va a caragr la matriz y desconoce por tanto sus dimensiones.

Tanto en los módulos "primary" como en los "joint" los segmentos pueden ahora añadir la información referida a los fitrosno lineales que antes se introducían a BSR de forma externa. Para ello se deberá definir el método

Get.NonLinearFilters: Devuelve un conjunto de NameBlock's cada uno de los cuales define un filtro no lineal sobre el mismo segmento

```
Set Get.NonLinearFilters(Real unused)
```

CAPíTULO 13

BysMcmc::Bsr::DynHlm

Se describen los miembros y métodos generales del NameBlock BysMcmc::Bsr::DynHlm

13.1. Descripción

Como ya se ha dicho anteriormente el caso del modelo jerárquico aparece de forma natural en multitud de ocasiones y su especial estructura sparse por bloques permite simplificaciones importantes a la hora de la definición del modelo. En particular es especialemente apta para ser almacenada en una base de datos que facilite la inserción, modificación y mantenimiento en general de diversos modelos y versiones ó sesiones del mismo, de forma que se automatice al máximo las tareas de definición, diagnosis, inferencia y todas las tareas que conlleva un sistema de modelación masiva tanto en la etapa de desarrollo como en la de producción.

13.2. El script de la base de datos del sistema Bsr-Hlm

De todo ellos se ocupan los miembros y métodos de este **NameBlock BysMcmc::Bsr::DynHlm** implementado en `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_dynhlm.tol`

El script SQL `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/db_bsrhlm_mysql.sql` contiene todo lo necesario para convertir una base de datos existente en el sistema de gestión de modelos de **BysMcmc::Bsr::DynHlm** que llamaremos en lo sucesivo sistema Bsr-Hlm.

En el capítulo 14 se explica con detalle todo lo relativo a las tablas del sistema y sus relaciones. Baste decir aquí que existen varios grupos de tablas

- Dimensionales: Aquellas que definen los valores que pueden adoptar determinados campos de otras tablas
  - Del sistema: aquellas cuyos valores vienen dados por el propio sistema y no deben ser modificadas por el usuario.
  - De usuario: aquellas que vienen vacías de fábrica y en las que el usuario, mediante el programa de interfaz que elija, debe introducir datos propios.
- ModSes: Aquellas que definen una sesión concreta de un modelo específico. Deben ser rellenadas por el programa de interfaz de usuario. El índice de todas estas tablas comienza siempre por los campos (id_model, id_session) que definen el objetivo y la versión concreta del modelo en cuestión.
  - Genéricas: aquellas que no se refieren a ningún tipo de nodo en particular.
  - Observacionales: aquellas que se refieren a los nodos observacionales.
  - Latentes: aquellas que se refieren a los nodos latentes.
  - A priori: aquellas que se refieren a los nodos a priori.
  - Mixtas: aquellas que se refieren a los nodos mixtos, es decir que pueden relacionar diversos tipos de nodos.
- Resultados: Aquellas que recojen resultados de la simulación y que son rellenadas por el propio sistema Bsr-Hlm durante y después del proceso.

13.3. NameBlock BysMcmc::Bsr::DynHlm::DbApi

En el fichero `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_db_api.tol` se implementa la cabecera del **NameBlock BysMcmc::Bsr::DynHlm::DbApi** que incluye los métodos específicos de manejo de la base de datos del sistema Bsr-Hlm.

Todos los métodos precisan que previamente haya sido abierta y activa la conexión a la base de datos del sistema Bsr-Hlm.

Se han clasificado dichos métodos en distintos ficheros según el tipo de datos a los que afectan:

- `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_db_api.tools.tol`: Contiene las utilidades SQL genéricas y las de manejo de las tablas dimensionales:
  - **Real SetEngineToMySQL(Real unused)**: Establece en el miembro **_.engine** el motor de base de datos a emplear. De momento el único admitido es MySql.
  - **Set SqlTableTrace(Text query, Text structure)**: Ejecuta una consulta de tipo **SELECT** y devuelve el contenido en un conjunto con la estructura dada, si no es la cadena vacía en cuyo caso se devuelve una tabla no estructurada. Si el miembro de configuración doQueryTrace está habilitado se escribirá primero la consulta por la salida estándar.
  - **Real SqlExecTrace(Text query)**: Ejecuta una consulta de modificación (**INSERT**, **DELETE**, **UPDATE**, **ALTER**, ...) y devuelve 1 en caso de éxito. Si el miembro de configuración doQueryTrace está habilitado se escribirá primero la consulta por la salida estándar.
  - **Text SqlFormatDate(Date dte)**: Transforma una fecha en un texto admisible como fecha por el motor de la base de datos dado en el miembro **_.engine.** Si la fecha no es válida devuelve el texto **'NULL'** para que sea tratado como tal por la base de datos.
  - **Text SqlFormatReal(Real x)**: Transforma un número real en un texto admisible como número de precisión doble por el motor de la base de datos dado en el miembro **_.engine.** Si el número no es finito devuelve el texto **'NULL'** para que sea tratado como tal por la base de datos.
  - **Real CreateSkeleton(Real unused)**: Ejecuta el script de la base de datos del sistema (ver 13.2) en la base de datos activa.
  - **Real ForeignKey.Disable(Real unused)**: Desactiva las claves ajenas que relacionan las tablas para permitir hacer modificaciones controladas. No se debe usar si no se está seguro de que las modificaciones serán congruentes con dichas claves.
  - **Real ForeignKey.Enable(Real unused)**: Reestablece las claves ajenas que relacionan las tablas después de haber hecho modificaciones controladas.
  - **Real Mod.Create(Text dbName, Text model, Text description)**: Crea un nuevo registro de modelo o modifica uno existente.
  - **Real Mod.Delete(Text dbName, Text model)**: Elimina, si existe, un registro de modelo y todos los registros asociados a todas las sesiones del mismo en todas las tablas del sistema. Se debe usar por tanto con mucho cuidado pues supone la eliminación de todo rastro del modelo.
  - **Real Mod.SetBlocked(Text dbName, Text model, Real blocked)**: Bloquea o desbloquea todas las model-session asociadas a un modelo.
  - **Real Ses.Create(Text dbName, Text model, Text description)**: Crea un nuevo registro de sesión o modifica uno existente.
  - **Real Ses.Delete(Text dbName, Text model)**: Elimina, si existe, un registro de sesión y todos los registros asociados a todas los modelos de la misma en todas las tablas del sistema. Se debe usar por tanto con mucho cuidado pues supone la eliminación de todo rastro de la sesión.
  - **Real Ses.SetBlocked(Text dbName, Text model, Real blocked)**: Bloquea o desbloquea todas las model-session asociadas a una sesión.

- `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_db_api.tools.modses.tol`: Contiene las utilidades de manejo de las tablas de tipo ModSes genéricas:
  - **Real ModSes.CheckBlocked(Text dbName, Text model, Text session, Real forzeStop)**: Comprueba si una model-session está bloqueada y en tal caso, si **forzeStop** es cierto se para el proceso de compilación TOL en curso con **Real Stop**.
  - **Real ModSes.SetBlocked(Text dbName, Text model, Text session, Real blocked)**: Establece el estado de bloqueo de una model-session al valor **blocked**.
  - **Real ModSes.Create(Text dbName, Text model, Text session)**: Crea un registro de model-session. Un modelo puede tener diferentes sesiones y una sesión se puede referir a varios modelos.
  - **Real ModSes.SetLevels(Text dbName, Text model, Text session, Real maxLatLevel, Real hasPrior, Real hasMixture)**: Establece los niveles de jerarquía admnisibles del modelo.
  - **ModSes.Node.Create(Text dbName, Text model, Text session, Text node, Text nodeType, Real level, Real number, Text description)**: Crea un nuevo registro de nodo. Si ya existiera uno con ese nombre en esa model-session dará error de SQL.
  - **Real ModSes.Delete(Text dbName, Text model, Text session)**: Borra todos los registros asociados a una model-session en todas las tablas del sistema. Se debe usar por tanto con mucho cuidado pues supone la eliminación de todo rastro de la model-session.
  - **Real ModSes.Replicate(Text from_dbName, Text from_model, Text from_session, Text to_dbName, Text to_model, Text to_session)**: Hace una copia de una model-session. Como es lógico, al menos uno de los tres descriptores de destino **to_dbName, to_model, to_session** debe ser distintos de los correspondientes descriptores de origen **from_dbName, from_model, from_session**. Si es **to_dbName** distinta de **from_dbName** esto permite transmitir modelos entre distintas bases de datos dentro del mismo gestor.
  - **Real ModSes.Move(Text from_dbName, Text from_model, Text from_session, Text to_dbName, Text to_model, Text to_session)**: Copia una model-session de origen en otra de destino como se explica en la función anterior y luego borra la de origen.
  - **Real ModSes.Node.Rename(Text dbName, Text model, Text session, Text from_node, Text to_node)**: Renombra un nodo dentro de una model-session extendiendo el cambio a todas las tablas afectadas para mantener la integridad referencial.
  - **Real ModSes.Node.Delete(Text dbName, Text model, Text session, Text from_node)**: Borra un nodo un nodo dentro de una model-session extendiendo el cambio a todas las tablas afectadas para mantener la integridad referencial.
- `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_db_api.tools.modses.obs.tol`: Contiene las utilidades de manejo de las tablas de tipo ModSes Observacionales:
  - **Real ModSes.Obs.Output.Create(...)**: Crea el registro de output
  - **Real ModSes.Obs.Input.Create(...)**: Crea un registro de input
  - **Real ModSes.Obs.TransFun.Create(...)**: Crea un registro de función de transferencia omega(B)/delta(B)
  - **Real ModSes.Obs.SetArima(Text dbName, Text model, Text session, Text node, Set arima)**: Crea un registro de factor ARIMA para cada elemento con estructura ARIMAStruct del argumento **arima**.
  - **Real ModSes.Obs.Create(Text dbName, Text model, Text session, NameBlock node)**: Crea todos los registros necesarios para la definición de un nodo observacional a partir de un manejador TOL del nodo con esta API mínima

```
NameBlock node =
[[
  //##Nombre del nodo
  Text _.id_node;
  //##Descripción del nodo
  Text _.description;
  //##Devuelve una serie del nodo dado un identificador único
  Serie GetSerie(Text id_serie);
  //##Información adicional del modelo sobre el output
  NameBlock _.output =
  [[
    //##Nombre con el que reconoce la serie en la base de datos
    Text  _.name;
    //##Serie output
    Serie _.serie;
    //##Número de sigmas para la distribución a priori de los
    //omitidos se multiplicará por la desviación típica muestral
    //de la serie. La media a priori será la de la propia serie.
    Real  _.mis_pri_sig_fac;
    //##Valor mínimo a priori para los omitidos
    Real  _.mis_pri_min;
    //##Valor máximo a priori para los omitidos
    Real  _.mis_pri_max;
    //##Fecha inicial de estimación del nodo. Si la serie empieza
    //más tarde se rellenará con omitidos
    Date  _.firstDate;
    //##Fecha final de estimación del nodo. Si la serie acaba antes
    //se rellenará con omitidos, lo cual es un método de previsión
    //muy simple
    Date  _.lastDate;
    //##Parte ARIMA asociada
    Set   _.arima;
  ]];
  //##Información adicional del modelo sobre los inputs
  //Cada elemento es un NameBlock cuyo nombre será el identificador
  //en la base de datos de BsrHlm y es obligatorio que se llame
  //igual que la correspondiente serie input
  Set _.input = SetOfNameBlock(
    //## input.1.name es el nombre con el que reconoce la serie en
    //la base de datos
    NameBlock input.1.name =
    [[
      //##Serie input
      Serie _.serie;
      //##Si es falso no se usará este input
      Real  _.enabled;
      //#Función de transferencia. Es opcional con valor por defecto 1/1
      Ratio _.transferFunction;
      //##Número de sigmas para la distribución a priori de los omitidos
      Real  _.mis_pri_sig_fac,
      //##Valor mínimo a priori para los omitidos
      Real  _.mis_pri_min;
      //##Valor máximo a priori para los omitidos
      Real  _.mis_pri_max;
```

```
      //##Valor inicial del parámetro
      Real _.initValue;
      //##Valor mínimo a priori del parámetro
      Real _.minimum;
      //##Valor máximo a priori del parámetro
      Real _.maximum
    ]],
    ...
  )
]];
```

- **Real ModSes.Obs.CreateAll(Text dbName, Text model, Text session, Set nodes)**: Crea un conjunto de nodos observacionales a partir de un conjunto de manejadores TOL del nodo con la API mínima recién definida.
- **Real ModSes.Obs.Output.Serie.Rename(Text dbName, Text model, Text session, Text from_serie, Text to_serie)**: Renombra una serie output en todas las tablas donde es necesario.
- **Real ModSes.Obs.Input.Serie.Rename(Text dbName, Text model, Text session, Text from_serie, Text to_serie)**: Renombra una serie input en todas las tablas donde es necesario.

▪ `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_db_api.tools.modses.lat.tol`: Contiene las utilidades de manejo de las tablas de tipo ModSes Latentes

- **Real ModSes.Lat.SigBlk.Create(...)**: Crea el registro del Sigma-Block. de un nodo latente.
- **Real ModSes.Lat.Equ.Create(...)**: Crea el registro de una ecuación de un nodo latente.
- **Real ModSes.Lat.Output.Create(...)**: Crea el registro de un término output para una ecuación de un nodo latente.
- **Real ModSes.Lat.Input.Create(...)**: Crea el registro de un término input para una ecuación de un nodo latente.
- **Real ModSes.Lat.Homog_1.Create(Text dbName, Text model, Text session, Text id_node, Text description, Real level, Real number, Text id_parameter, Real initValue, Real sigma, Real minimum, Real maximum, Set node_childs)**: Crea todos los registros necesarios para definir un nodo latente de homogenidad simple sobre un conjunto de variables de los nodos hijo con esta API

```
Set node_childs = SetOfNameBlock(
  NameBlock child.1 =
  [[
    Text _.id_node;
    Text _.id_parameter;
    Real _.sigma_factor
  ]],
  ...
)
]];
```

- **Real ModSes.Lat.Create(Text dbName, Text id_model, Text id_session, Text id_node, Text description, Real level, Real number, Set father_parameters, Set node_childs, Real sigma, Matrix X)**: Crea todos los registros necesarios para definir un nodo latente genérico de la forma: $\beta_i = \sum_j X^{ij}\lambda_j + e_i$.

El **Set father_parameters** es un conjunto de NameBlocks con información relativa a cada uno de los hiperparámetros λ_j, este conjunto tendrá tantos Nameblocks como hiperparámetros $\lambda_j$haya:

```
Set father_parameters = [[
    NameBlock Lam1 =
     ModSes.Lat.Father.Param.Def(Text id_parameter,
        Real init_value, Real min, Real max),
    ........
]];
```

En el ejemplo citado vemos la construcción explícita de un NameBlock que corresponde al hiperparámetro λ_1 y que se realiza a través de la función **ModSes.Lat.Father.Param.Def** (Text id_parameter, Real init_value, Real min, Real max). Los argumentos de esta función son los siguientes:
Text id_parameter: texto identificativo del hiperparámetro.
Real init_value: valor inicial para la simulación.
Real min: valor mínimo del dominio de muestreo para λ_1.
Real max: valor máximo del dominio de muestreo para λ_1.

El **Set node_childs** es un conjunto de NameBlocks con información relativa a los parámetros β_i, este conjunto tedrá tantos NameBlocks como parámetros $\beta_i$haya:

```
Set node_childs = [[
  NameBlock Param1 =
      ModSes.Lat.Child.Param.Def  (Text id_node,
       Text id_paramater, Real sigma_factor),
      .....
  ]];
```

En el ejemplo vemos la construcción explícita de un NameBlock que corresponde al parámetro β_1, y se realiza a través de la función **ModSes.Lat.Child.Param.Def** (Text id_node, Text id_parameter, Real sigma_factor). Los argumentos de esta función son los siguientes:
Text id_node: Texto identificativo del segmento o nodo donde se encuentra el parámetro β_i.
Text id_parameter: Texto identificativo del parámetro β_i.
Real sigma_factor: Valor real para el parámetro sigma factor, común a todo el nodo de regresión.

La matrix X es la matrix X^{ij}.

- **Real ModSes.Lat.OutComb.Create(Text dbName, Text id_model, Text id_session, Text id_node, Text description, Real level, Real number, Set father_parameters, Set node_childs, Real sigma, Matrix X, Matrix Y, Set SigmaBlock)**: Crea todos los registros necesarios para definir un nodo latente genérico con output combinado, es decir de la forma: $\sum_j Y^{ij}\beta_j = \sum_l X^{il}\lambda_l + e_i (i = 1, ..., n)$.
  El Set father_parameters y el Set node_childs tienen el mismo significado y se construyen de la misma forma que en la función anterior Real **ModSes.Lat.Create.**
  El Set SigmaBlock es un conjunto de valores reales que fijan el valor de $\sigma_i (i = 1, ..., n)$para cada una de las ecuaciones del nodo latente.

- http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_db_api.tools.modses.pri.tol: Contiene las utilidades de manejo de las tablas de tipo ModSes A priori
  - **Real ModSes.Pri.Equ.Create(...)**: Crea el registro de una ecuación de un nodo a priori.
  - **Real ModSes.Pri.Output.Create(...)**: Crea el registro de un término output para una ecuación de un nodo a priori.
  - **Real ModSes.Pri.Homog_1.Create(Text dbName, Text model, Text session, Text id_node, Text description, Set node_childs)**: Crea todos los registros necesarios para definir un nodo latente de homogenidad simple sobre un conjunto de variables de los nodos hijo con esta API

```
  Set node_childs = SetOfNameBlock(
    NameBlock child.1 =
    [[
      Text _.id_node;
      Text _.id_parameter;
      Real _.average;
      Real _.sigma
    ]],
    ...
  )
]];
```

  - **Real ModSes.Pri.OutComb.Create(Text dbName, Text model, Text session, Text id_node, Text description, Real number, Set PriorInfo, Set node_childs, Matrix Y)**: Crea todos los registros necesarios para definir un nodo a priori genérico con output combinado: $\sum_j Y^{ij}\beta_j \sim N(\mu_i, \sigma_i)$.
    El Set PriorInfo es un conjunto de conjuntos que especifica los valores μ_i, σ_i para cada una de las ecuaciones del nodo a priori.

```
Set PriorInfo = [[
    [[ Real mu_1, Real sigma_1 ]],
    [[ Real mu_2, Real sigma_2 ]],
    .....
]]
```

    En el ejemplo anterior se construye explícitamente el conjunto PriorInfo, donde aparecen dos conjuntos con los valores de μ_1, σ_1 y μ_2, σ_2 para dos ecuaciones de nodo apriori.
    http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_db_api.tools.modses.mix.tol: Contiene las utilidades de manejo de las tablas de tipo ModSes Mixtas
  - **Real ModSes.Mix.Param.Create(...)**: Crea el registro de un parámetro a simular en el modelo.
  - **Real ModSes.Mix.Param.Delete(...)**: Borra el registro de un parámetro a simular en el modelo en todas las tablas donde aparece.
  - **Real ModSes.Mix.OrderRelation.Create(...)**: Crea el registro de una relación de orden entre dos parámetros del bloque lineal.
  - **Real modSes.Mix.Constraint.Create(...):** Crea el registro de una relación de orden compuesta por una combinación lineal de parámetros
    de bloques lineales.

- http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_db_api.load.tol: Contiene los métodos de carga de modelos previamente definidos en la base de datos
  - **NameBlock CreateSeriesHandler(Set obs.node.handlers)**: Crea el manejador de series del modelo a partir de un conjunto de NameBlock's como el definido en la función **ModSes.Obs.Create** (ver 13.3)
  - **NameBlock LoadNodeConstraints(Text dbName, Text model, Text session, Text id_node)**: Carga el importador a BSR-ASCII de las restricciones que sólo afectan a parámetros de un nodo determinado, o bien, si **id_node** es igual a **"MIXTURE"**, de las restricciones que afectan a más de un nodo.
  - **NameBlock LoadObsNode(...)**: Carga el importador a formato ASCII-BSR de un nodo observacional.
  - **NameBlock LoadLatNode(...)**: Carga el importador a formato ASCII-BSR de un nodo latente.
  - **NameBlock LoadPriNode(...)**: Carga el importador a formato ASCII-BSR de un nodo a priori.
  - **NameBlock LoadModelDef(...)**: Carga el importador a formato ASCII-BSR de un modelo Bsr-Hlm completo.
- http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_db_api.estim.tol: Contiene los métodos de estimación de modelos una vez cargados los datos de definición del modelo.
  - **Real SaveParamStats(Text dbName, Text model, Text session, Set estim)**: Guarda en la base de datos los resultados de las estadísticas de diagnosis sobre la estimación de los parámetros resultante de la simulación de un modelo.
  - **Set GetLinearBlkEffects(...)**: Crea las series de efectos lineales correspondientes a cada input o función de transferencia de cada nodo.
  - **Set Estim(Text dbName, Text model, Text session, Text resultRootPath, NameBlock seriesHandler, NameBlock config)**: Simula y devuelve los resultados de diagnosis y de evaluación según se especifique en la configuración dada, y escribe en la base de datos la información de estado durante la simulación y, al final de la misma, las estadísticas de diagnosis sobre la estimación de los parámetros.

13.4. NameBlock BysMcmc::Bsr::DynHlm::BuildNode

En el fichero http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_build.node.tol se implementa la cabecera del **NameBlock BysMcmc::Bsr::DynHlm::BuildNode** que incluye los métodos específicos para la importación de nodos jerárquicos al formato BSR-ASCII, es decir los NameBlock de gestión de segmentos explicados en 12.3

Se han clasificado dichos métodos en distintos ficheros según el tipo de nodos a los que afectan:

- http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_build.node.obs.tol: Contiene las funciones relativas a los nodos observacionales
  - **NameBlock Obs.Serie.Info(...)**: Crea el manejador de una serie de observaciones, independientemente de si es un unput o el output del nodo, incluyendo la información relativa a los omitidos y a los valores iniciales, si es que se trata de un input con función de transferencia.
  - **NameBlock Obs.Output(...)**: Crea el manejador del output.
  - **NameBlock Obs.Input(...)**: Crea el manejador de un input sin función de transferencia.
  - **NameBlock Obs.TransFun(...)**: Crea el manejador de un input con función de transferencia.
  - **NameBlock Obs.TransFun(...)**: Crea el manejador de un input con función de transferencia.

  - **NameBlock Obs.TransFun(...)**: Crea el manejador de un input con función de transferencia.
  - **NameBlock Obs.TransFun(...)**: Crea el manejador de un input con función de transferencia.
  - **NameBlock Obs(...)**: Crea el importador de un nodo observacional.
- `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_build.node.lat.tol`: Contiene las funciones relativas a los nodos latentes.
  - **NameBlock Lat.Homog(...)**: Crea el importador de un nodo latente homogéneo simple.
- `http://cvs.tol-project.org/viewcvs.cgi/*checkout*/tol/stdlib/math/stat/models/bayesian/bysMcmc/bsr/dynhlm/_build.node.pri.tol`: Contiene las funciones relativas a los nodos a priori.
  - **NameBlock Lat.Homog(...)**: Crea el importador de un nodo a priori homogéneo simple.

Parte 3

Manual de usuario de BSR-HLM

En esta parte interesa a todos los usuarios de BSR que se enfrenten a problemas que se pueden representar dentro del clásico modelo jerárquico lineal, con las extensiones de estructura ARIMA y datos omitidos en el nivel observacional. Se describe la base de datos del sistema BSR-HLM, los métodos para interactuar con ella a diferentes niveles, usando la interfaz gráfica o una plantilla de modelo en TOL según las necesidades de cada caso. También se detallan los resultados obtenidos tras una simulación, los métodos de diagnosis y en general todo lo relacionado con el manejo básico de BSR tanto en la fase de análisis como en la de mantenimiento de un proyecto de modelación.

Estos son los resúmenes de cada capítulo:

... pendiente!

CAPíTULO 14

La base de datos del sistema Bsr-Hlm

Se describen las diferentes utilidades de la base de datos del sistema de definición de modelos BSr-Hlm

14.1. Descripción

La base de datos del sistema Bsr-Hlm no es exactamente una base de datos sino un mero conjunto de tablas relacionadas entre sí cuyos nombres comienzan por el prefijo **bsrhlm_** y que se pueden imbricar en una base de datos existente conviviendo con otras tablas de datos más o menos relacionados con el proyecto en cuestión, o bien estar aisladas en una base de datos dedicada para uno o varios proyectos no demasiado grandes. Por el momento sólo está disponible para el gestor de MySql pero más adelante se irá independizando la implementación del gestor concreto, incluyendo al menos los gestores de Postgree Sql, Oracle y Microsoft Sql Server.

14.2. Tablas

A continuación se describen cada una de las tablas y sus campos. Los que se presentan subrayados forman la clave primaria. Las claves ajenas se representan mediante la palabra clave FK y un enlace al campo de la tabla maestra.

14.2.1. bsrhlm_d_gibbs_block. Se introducén los distintos tipos de bloques GIBBS

1. Texto **id_gibbs_blk**: Identificador del tipo de bloque GIBBS
2. Texto **ds_gibbs_blk**: Descripción del tipo de bloque GIBBS

Los valores admitidos actualmente son:

- 'LinearBlk';'Main linear regression'
- 'SigmaBlk';'Variance of each segment of regression'
- 'ArmaBlk';'AutoRegressive Moving Average'
- 'MissingBlk';'Missing values of input and output'

14.2.2. bsrhlm_d_node_type. Se introducen los tipos de nodos que van a tener los modelos y su descripción.

1. Texto **id_node_type**: Identificador del tipo de nodo
2. Texto **ds_node_type**: Descrición del tipo de nodo

Hay 4 tipos de nodos, observacionales, latentes, a priori y mixtos. La descrición de cada uno de los nodos esta a continuación.

- 'OBS':Los nodos observacionales están en el primer nivel de la jerarquía
- 'LAT':'Latent nodes are in the second and upper levels of hierarchical tree'
- 'PRI':'Prior nodes stablish normality hypotesys about parameters of observational and latent nodes'
- 'MIX':'The mixture node is used to represent entities that are not linked to a specific node but to an unespecified set of nodes. If its used, it must be just one node called MIXTURE of type MIX at table node'

14.2.3. bsrhlm_d_level_node_type. Se introducen los tipos de nodos y el nivel que tienen asociado

1. Texto **id_node_type**: Identificador del tipo de nodo. FK[1] bsrhlm_d_node_type.id_node_type
2. Entero **nu_level**: Nivel del tipo de nodo asociado

Los registros dados por el sistema como válidos en principio son

- 'OBS';0
- 'LAT';1
- 'LAT';2
- 'LAT';3
- 'LAT';4
- 'LAT';5
- 'LAT';6
- 'LAT';7
- 'LAT';8
- 'LAT';9
- 'PRI'; 888888888
- 'MIX'; 999999999

14.2.4. bsrhlm_d_model. Se introducen los modelos que se van a estimar

1. Texto **id_model**: Identificador del modelo
2. Texto **ds_model**: Descripción del modelo

14.2.5. bsrhlm_d_session. Tabla con las sesiones de estimación.

1. Texto **id_session**: Identificador de la sesión
2. Texto **ds_session**: Descripción de la sesión
3. Texto **te_authors**: Autores de la sesión
4. Fecha **dh_creation**: Fecha y hora de creación de la sesión

14.2.6. bsrhlm_d_model_session. Tabla que asocial modelo con sesión

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión.

14.2.7. bsrhlm_d_level. Tabla que describe para cada par modelo y sesión que tipos de nodos tiene.

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node_type**: Identificador del tipo de nodo
4. Entero **nu_level**: Nivel del nodo

14.2.8. bsrhlm_d_node. Tabla que describe los nodos que tiene cada modelo

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node**: Identificador del nodo
4. Texto **id_node_type**: Identificador del tipo de nodo
5. Entero **nu_level**: Nivel del nodo
6. Entero **nu_node**:
7. Texto **ds_node**: Descripción del nodo

14.2.9. bsrhlm_v_mix_parameter. Tabla con los parámetros que tiene cada nodo observacional

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node**: Identificador del nodo
4. Texto **id_parameter**: Identificador del parámetro
5. Texto **id_gibbs_blk**: Tipo de bloque del parámetro

6. Real **vl_prm_ini**: Valor inicial del parámetro
7. Real **vl_prm_min**: Valor mínimo del parámetro
8. Real **vl_prm_max**: Valor máximo del parámetro

14.2.10. bsrhlm_v_mix_non_lin_filter. Tabla con los parámetros que tiene cada nodo observacional

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node**: Identificador del nodo
4. Texto **id_series**: Identificador del input/output modificado
5. Entero **nu_lin_blk_param**: Número de parámetros afectados en el bloque lineal
6. Entero **nu_non_lin_blk_param**: Número de parámetros simulados en el bloque no lineal

14.2.11. bsrhlm_v_mix_order_relation. Tabla que define las restricciones de orden entre parámetros

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node_lower**: Identificador del nodo que va a tener el menor de los parámetros
4. Texto **id_parameter_lower**: Identificador del menor parámetro
5. Texto **id_node_upper**: Identificador del nodo que va a tener el mayor de los parámetros
6. Texto **id_parameter_upper**: Identificador del parámetro mayor

14.2.12. bsrhlm_v_mix_cnstrnt_border. Tabla que define las ecuaciones de las restricciones de orden a<=f(x)>=b :

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: identificador de la sesión
3. Entero **nu_inequation**: Identificador de la ecuación
4. Texto **id_node**: Identificador del nodo mixto
5. Real **vl_left_border**: Limite izquierdo de la ecuación
6. Real **vl_right_border**: Limite derecho de la ecuación

14.2.13. bsrhlm_v_mix_cnstrnt_lin_cmb. Tabla que define las ecuaciones de las restriciones de orden a<=f(x)>=b :

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Entero **nu_inequation**: Identificador de la ecuación
4. Texto **id_node**: Identificador del nodo que va a tener uno de los parámetros de la ecuación
5. Texto **id_parameter**: Identificador del parámetro
6. Real **vl_coef**: Valor que va a multiplicar al parámetro en la ecuación

14.2.14. bsrhlm_v_obs_output. Tabla con los output de los nodos observacionales

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node**: Identificador del nodo observacion
4. Texto **id_series**: Nombre de la serie output
5. Real **vl_mis_pri_sig_fac**: Número de sigmas entre los que pueden variar los omitidos respecto a la media
6. Real **vl_mis_pri_min**: Valor minimo de los omitidos
7. Real **vl_mis_pri_max**: Valor máximo de los omitidos
8. Fecha **dh_start**: Fecha de inicio de la estimación
9. Fecha **dh_end**: Fecha de fin de la estimación

14.2.15. bsrhlm_v_obs_arima_block. Tabla para definir el modelo ARIMA de los observacionales

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node**: Identificador del nodo observacional
4. Entero **nu_factor**: Identificador de cada periodo
5. Entero **nu_period**: Periodo
6. Texto **te_ar**: Parte AR
7. Texto **te_ma**: Parte MA
8. Texto **te_dif**: Parte diferencial

14.2.16. bsrhlm_v_obs_input. Definición de los inputs de los observacionales

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: identificador de la sesión
3. Texto **id_node**: Identificador del nodo
4. Texto **id_parameter**: Identificador del parámetro. Usualmente es id_series+"::Coef".
5. Texto **id_series**: Nombre de la serie input
6. Real **vl_mis_pri_sig_fac**: Número de sigmas entre los que pueden variar los omitidos respecto a la media
7. Real **vl_mis_pri_min**: Valor mínimo de los omitidos
8. Real **vl_mis_pri_max**: Valor máximo de los omitidos

14.2.17. bsrhlm_v_obs_transferFunction. Definición de las funciones de transferencia de los inputs observacionales

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: identificador de la sesión
3. Texto **id_node**: Identificador del nodo
4. Texto **id_transferFunction**: Identificador de la funciópn de transferencia. Puede haber varias para una misma serie input.
5. Texto **te_omega**: Polinomio numerador de la función de transferencia
6. Texto **te_delta**: Polinomio denominador de la función de transferencia
7. Texto **id_series**: Nombre de la serie input
8. Real **vl_mis_pri_sig_fac**: Número de sigmas entre los que pueden variar los omitidos respecto a la media
9. Real **vl_mis_pri_min**: Valor mínimo de los omitidos
10. Real **vl_mis_pri_max**: Valor máximo de los omitidos

14.2.18. bsrhlm_v_lat_sigma_block. Tabla para definir el sigma block de los modelos jerárquicos

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node**: Identificador del nodo latente
4. Real **vl_sigma**: Sigma block del nodo latente

14.2.19. bsrhlm_v_lat_equ. Tabla para definir las ecuaciones de un nodo latente

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node_father**: Identificador del nodo latente padre
4. Entero **nu_equation**: Número de ecuación
5. Real **vl_sigma_factor**: Valor por el que se altera el sigma-block del nodo latente para esta ecuación

14.2.20. bsrhlm_v_lat_output. Tabla para definir las combinaciones de los parámetros hijos en el output de cada ecuación

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node_father**: Identificador del nodo latente padre
4. Entero **nu_equation**: Número de ecuación
5. Texto **id_node_child**: Identificador del nodo hijo
6. Texto **id_parameter_child**: Identificador del parámetro del hijo
7. Real **vl_coef**: Coeficiente de la variable hija en el output

14.2.21. bsrhlm_v_lat_input. Tabla para definir las combinaciones de los parámetros padre en el input de cada ecuación

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node_father**: Identificador del nodo latente padre
4. Entero **nu_equation**: Número de ecuación
5. Texto **id_parameter_father**: Identificador del parámetro padre
6. Real **vl_coef**: Valor del parámetro de relación del nodo padre con el nodo hijo

14.2.22. bsrhlm_v_pri_equ. Tabla para definir las ecuaciones de un nodo a priori

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node_father**: Identificador del nodo a priori padre
4. Entero **nu_equation**: Número de ecuación
5. Real **vl_average**: Media a priori
6. Real **vl_sigma**: Varianza a priori

14.2.23. bsrhlm_v_pri_output. Tabla para definir las combinaciones de los parámetros hijos en el output de cada ecuación

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node_father**: Identificador del nodo a priori
4. Entero **nu_equation**: Número de ecuación
5. Texto **id_node_child**: Identificador del nodo que tiene el parámetro
6. Texto **id_parameter_child**: Identificador del parámetro hijo
7. Real **vl_coef**: Coeficiente de la variable hija en el output

14.2.24. bsrhlm_v_est_status. Tabla que almacena los resultados del proceso de estimación estimación, tiempo,...

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Entero **in_blocked**: Bloquea el modelo-sesión para proteger los datos
4. Texto **te_path_data_out**: Path del log de salida
5. Entero **nu_mcmc_var**: Número de variables
6. Entero **nu_mcmc_burnin**: Número de burning
7. Entero **nu_mcmc_sampleLength**: Número de iteraciones del modelo
8. Entero **nu_mcmc_cacheLength**:
9. Entero **nu_mcmc_currentLength**:
10. Real **vl_mcmc_time_avg**: Tiempo medio de estimación
11. Entero **nu_error**: Número de errores
12. Entero **nu_warning**: Número de warnings
13. Fecha **dh_loading**: Inicio de la carga
14. Fecha **dh_parsing**: Inicio del parseo
15. Fecha **dh_initializing**: Hora de inicializacion
16. Fecha **dh_simulating**: Hora de simulación

17. Fecha **dh_reporting**: Hora del reporte
18. Fecha **dh_evaluating**: Hora de la evaluación
19. Fecha **dh_saving**: Hora del salvado
20. Fecha **dh_ending**: Hora de la finalización

14.2.25. bsrhlm_v_est_param_stats. Tabla que almacena los parámetros del modelo

1. Texto **id_model**: Identificador del modelo
2. Texto **id_session**: Identificador de la sesión
3. Texto **id_node**: Identificador del nodo
4. Texto **id_parameter**: Identificador del parámetro
5. Texto **id_gibbs_blk**: Identificador del bloque gibbs
6. Entero **nu_mcmc_index**:
7. Real **vl_mean**: Media
8. Real **vl_sd**: Desviación estandar
9. Real **vl_naive_se**:
10. Real **vl_tmser_se**:
11. Real **vl_q_001**: quantil 1 %
12. Real **vl_q_025**: quantil 2.5 %
13. Real **vl_q_250**: quantil 25 %
14. Real **vl_q_500**: quantil 50 %
15. Real **vl_q_750**: quantil 75 %
16. Real **vl_q_975**: quantil 97.5 %
17. Real **vl_q_999**: quantil 99 %
18. Real v**l_raftery_length**:
19. Real **vl_raftery_burnin**:
20. Real **vl_raftery_size**:
21. Real **vl_raftery_size_min**:
22. Real **vl_raftery_dep_factor**:
23. Real **vl_raftery_remain**:

14.3. Diagrama de relaciones entre las tablas

En la figura siguiente se muestran las relaciones de clave ajena existente entre las diferentes tablas del sistema Bsr-Hlm

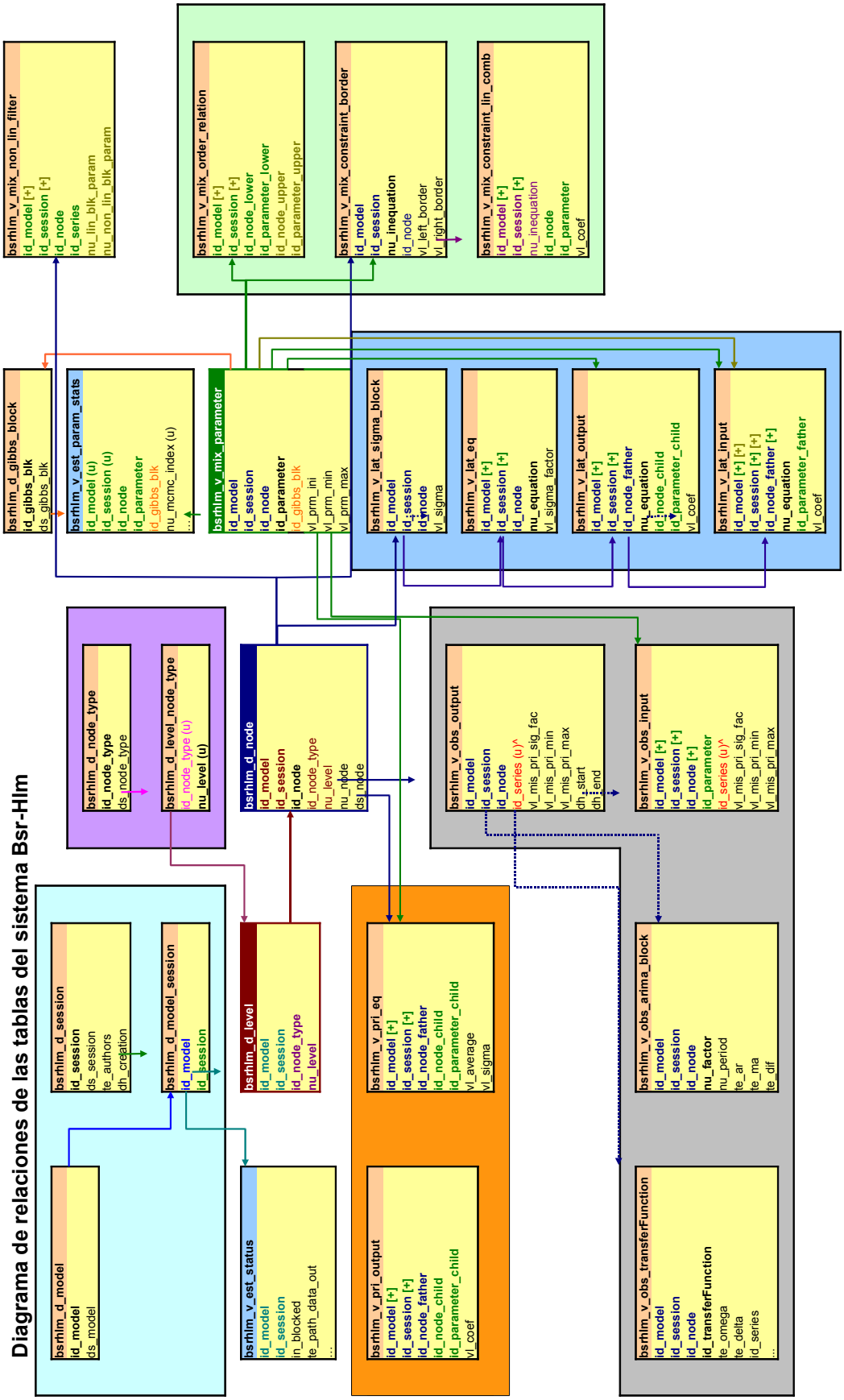
Diagrama de relaciones de las tablas del sistema Bsr-Hlm
bsrhlm_d_model
id_model
ds_model
bsrhlm_d_session
id_session
ds_session
te_authors
dh_creation
bsrhlm_d_model_session
id_model
id_session
bsrhlm_v_est_status
id_model
id_session
in_blocked
te_path_data_out
...
bsrhlm_d_level
id_model
id_session
id_node_type
nu_level
bsrhlm_d_node_type
id_node_type
ds_node_type
bsrhlm_d_level_node_type
id_node_type (u)
nu_level (u)
bsrhlm_d_node
id_model
id_session
id_node
id_node_type
nu_level
nu_node
ds_node
bsrhlm_v_pri_eq
id_model [+]
id_session [+]
id_node_father
id_node_child
id_parameter_child
vl_average
vl_sigma
bsrhlm_v_pri_output
id_model [+]
id_session [+]
id_node_father
id_node_child
id_parameter_child
vl_coef
bsrhlm_v_obs_output
id_model
id_session
id_node
id_series (u)^
vl_mis_pri_sig_fac
vl_mis_pri_min
vl_mis_pri_max
dh_start
dh_end
bsrhlm_v_obs_input
id_model [+]
id_session [+]
id_node [+]
id_parameter
id_series (u)^
vl_mis_pri_sig_fac
vl_mis_pri_min
vl_mis_pri_max
bsrhlm_v_obs_arima_block
id_model
id_session
id_node
nu_factor
nu_period
te_ar
te_ma
te_dif
bsrhlm_v_obs_transferFunction
id_model
id_session
id_node
id_transferFunction
te_omega
te_delta
id_series
...
bsrhlm_d_gibbs_block
id_gibbs_blk
ds_gibbs_blk
bsrhlm_v_est_param_stats
id_model (u)
id_session (u)
id_node
id_parameter
id_gibbs_blk
nu_mcmc_index (u)
...
bsrhlm_v_mix_parameter
id_model
id_session
id_node
id_parameter
id_gibbs_blk
vl_prm_ini
vl_prm_min
vl_prm_max
bsrhlm_v_lat_sigma_block
id_model
id_session
id_node
vl_sigma
bsrhlm_v_lat_eq
id_model [+]
id_session [+]
id_node
nu_equation
vl_sigma_factor
bsrhlm_v_lat_output
id_model [+]
id_session [+]
id_node_father
nu_equation
id_node_child
id_parameter_child
vl_coef
bsrhlm_v_lat_input
id_model [+] [+]
id_session [+] [+]
id_node_father [+]
nu_equation
id_parameter_father
vl_coef
bsrhlm_v_mix_non_lin_filter
id_model [+]
id_session [+]
id_node
id_series
nu_lin_blk_param
nu_non_lin_blk_param
bsrhlm_v_mix_order_relation
id_model [+]
id_session [+]
id_node_lower
id_parameter_lower
id_node_upper
id_parameter_upper
bsrhlm_v_mix_constraint_border
id_model
id_session
nu_inequation
id_node
vl_left_border
vl_right_border
bsrhlm_v_mix_constraint_lin_comb
id_model [+]
id_session [+]
nu_inequation
id_node
id_parameter
vl_coef

FIGURA 14.3.1.

CAPíTULO 15

Plantilla del modelo Bsr-Hlm

Se describe la plantilla del modelo Bsr-Hlm para usar el sistema sin interfaz gráfica, empleando tan sólo código TOL.

15.1. Descripción

A continuación exponemos las partes necesarias de código en las que consiste una plantilla ejemplo para el uso de BSr-Hlm:

Este NameBlock especifica los datos necesarios para la conexión con la base da datos BsrHlm

```
NameBlock DBConnection = [[
  NameBlock BsrHlm = DBConnect::Create("BsrHlm","bysfor","bys!98.",
    "mysql","bysNPIDSO","d-mysql01","Almacenamiento de
     modelos BSR-DynHLM");
  Real check = Real DBConnect::CheckAll(0)
                          ]];
```

Activamos la base de datos BsrHlm:

```
Real DBConnection::BsrHlm::Activate(0);
```

Identificamos el nombre del modelo:

```
Text id_model =  "BSR_Lat.Reg.Test_III";
```

Descripción del objetivo del modelo:

```
Text ds_model = "Pruebas para nodos latentes regresivos";
```

Identificamos la sesión:

```
Text id_session = "SES_LAT.REG_TEST.20081006_22";
```

Descripción de la sesión:

```
Text ds_session = "Pruebas nodos regresivos latentes";
```

Listado de autores:

```
Text ds_authors = "mafernandez@bayesinf.com;";
```

Inserción del modelo en la base de datos:

```
Real BysMcmc::Bsr::DynHlm::DBApi::Mod.Create(
    DBConnection::BsrHlm::_.defaultDB, id_model, ds_model);
```

Inserción de la sesión en la base de datos:

```
Real BysMcmc::Bsr::DynHlm::DBApi::Ses.Create(
     DBConnection::BsrHlm::_.defaultDB, id_session, ds_session,
     ds_authors);
```

Inserción del modelo-sesión en la base de datos:

```
Real BysMcmc::Bsr::DynHlm::DBApi::ModSes.Create(
      DBConnection::BsrHlm::_.defaultDB, id_model, id_session);
```

Se crean en la base da datos los niveles de jerarquía .

```
Real BysMcmc::Bsr::DynHlm::DBApi::ModSes.SetLevels(
     DBConnection::BsrHlm::_.defaultDB,id_model,id_session,
     Real maxLatLevel = 1,
     Real hasPrior    = True,
     Real hasMixture  = False)};
```

Es el momento de crear el NameBlock node.series.handler, responsable de manejar las series asociada a un nodo observacional. Describimos a continuación los elementos necesarios que aparecen en este NameBlock, los comentarios realizados con ## indican variables que son obligatorias:

```
////////////////////////////////////////////////////////
NameBlock node.series.handler (Real node)
////////////////////////////////////////////////////////
{ [[
    // Se forma la etiqueta del nodo.
    Text _node = FormatReal(node,"%1.0f");
    //## Nombre del nodo
    Text _.description = "Nodo observacional referido al output "+_node;
    //# Contiene todas las series output e input con nombres únicos dentro
    //de cada nodo
     Set _.series = { SetOfSerie (
        Output = .....
        Input1 = .....
        Input2 = .....
        ....
               )
                    };
    //# Se indexa para acelerar búsquedas
      Real _check_unique = SetIndexByName(_.series);
    //## Devuelve serie del nodo dado un identificaador único
      GetSerie (Text id_serie)
      {
       _.series[id_serie]
      }
    //##Información adicional del modelo sobre el output
     NameBlock _.output =
      {[[
      //##Nombre con el que reconoce la serie en la base de datos
```

```
Text _.name = "Output";
//##Serie output
Serie _.serie = _.series::Output;
//##Número de sigmas para la distribución a priori de los omitidos
Real _.mis_pri_sig_fac = 5;
//##Valor mínimo a priori para los omitidos
Real _.mis_pri_min     = 0;
//##Valor máximo a priori para los omitidos
Real _.mis_pri_max     = 2;
//##Fecha inicial de estimación del nodo
Date _.firstDate =  y2001m01;
//##Fecha final de estimación del nodo
Date _.lastDate  =  y2013m01;
// Desviación típica de los residuos para el nodo observacional.
// Si es ? se estimará
Real _.sigma = ?;
//##Parte ARIMA asociada
Set _.arima = SetOfSet (
   ARIMAStruct (1,1,1,1)
)
]]};
//##Información adicional del modelo sobre los inputs
//Cada elemento es un NameBlock cuyo nombre será el identificador en
//la base de datos de BsrHlm y es obligatorio que se llame igual que
//la corespondiente serie input
Set _.input = {
SetOfNameBlock(
NameBlock Input1= [[
//##Serie input
Serie _.serie = _.series::Input1;
//##Si es falso no se usará este input
Real _.enabled = 1;
//##Número de sigmas para la distribución a priori de los omitidos
Real _.mis_pri_sig_fac =  5;
//##Valor mínimo a priori para los omitidos
Real _.mis_pri_min     =  0;
//##Valor máximo a priori para los omitidos
Real _.mis_pri_max     =  2;
//##Valor inicial del parámetro
Real _.initValue       = 0.1;
//##Valor mínimo a priori del parámetro
Real _.minimum         = -1;
//##Valor máximo a priori del parámetro
Real _.maximum         =  1
]],
NameBlock Input2 = [[
 .....
 .....
]],
```

```
      .....
    // Se indexa para acelerar busquedas
    Real _checkUniqueInput = SetIndexByName(_.input)
 ]]
};
```

Ahora se llama al manejador de series del modelo, notar que el bucle For debe correr a todos los nodos observacionales que se quieran calcular:

```
////////////////////////////////////////////////////
NameBlock series.handler =
  BysMcmc::Bsr::DynHlm::DBApi::CreateSeriesHandler(
  For (1,num_nodos, NameBlock (Real node)
  {
   WriteLn("[series.handler] creando de series del nodo "+ IntText(node));
   NameBlock aux = node.series.handler(node);
   Eval(aux::_.id_node+"=aux")
}));
```

Ahora se crea en la base de datos los nodos observacionales :

```
Real BysMcmc::Bsr::DynHlm::DBApi::ModSes.Obs.CreateAll(
      DBConnection::BsrHlm::_.defaultDB, id_model, id_session,
      series.handler::_.obs.node);
```

Ahora viene la parte del código dónde se usarán las funciones necesarias para implementar en el modelo los nodos latentes, nodos a priori etc... Incluimos algunos ejemplos del uso de estas funciones

Nodos latentes homogéneos.

```
Real {
 BysMcmc::Bsr::DynHlm::DBApi::ModSes.Lat.Homog_1.Create
 (DBConnection::BsrHlm::_.defaultDB,
  id_model,
  id_session,
  // identificación del nodo latente
  id_node = "Lat."+child_input,
  // descripción del node latente
  description = "Fuerza la homogeneidad de la variable ...",
  // Nivel de latencia en la jerarquía
  Real level = 1,
  //Orden de aparición dentro del nivel
  Copy(level.1_number := level.1_number+1),
  //Nombre del hiperparámetro
  Text id_parameter = "hiper",
  //Valor inicial del hiperparámetro
  Real initValue = -0.1,
  //Desviación típica de los residuos. Si es ? se estimará
  Real sigma = 0.05,
  //Valor mínimo del hiperparámetro
  Real minimum = -1,
```

```
  //Valor máximo del hiperparámetro
  Real maximum = 10,
  //Selección de los parámetros y nodos hijos con el input habilitado
  //Es un Set formado por tantos NameBlocks como parémetroas hijos
  //(betas) formen el nodo latente homogéneo
  [[
    NameBlock beta1 = [[
       Text _.id_parameter = "Nombre del nodo donde está beta1"
       Text _.id_node      = "Nodo donde está situado el beta1"
       Real _.sigma_factor = 1
                      ]],
    NameBlock beta2 = [[ ...
                         ...]],
    ...
    ...
  ]]
 )};
```

Nodos latentes generalizados.

```
Real {
  // Set de hiperparámetros. Es un set con tantos NameBlocks como
  // hiperparámetros tiene el nodo lantente.
  Set father_parameters = [[
      NameBlock Lam1 =  ModSes.Lat.Father.Param.Def (
                        Text id_parameter = "hiper1",
                        Real init_value   = 0.1,
                        Real min          = 0,
                        Real max          = 1
                        ),
      NameBlock Lam2 = ModSes.Lat.Father.Param.Def (
                        ......
                        ......
                        ),
      ........
      ........
                          ]];
   // Set de parámetros hijo. Es un set con tantos NameBlocks como
   // paramétros hijo tenga el nodo latente regresivo.
   Set node_childs = [[
      NameBlock beta1 = ModSes.Lat.Child.Param.Def (
                        Text id_node = "node1",
                        Text id_parameter = "beta1",
                        Real sigma_factor = 1
                        ),
      NameBlock beta2 = ModSes.Lat.Child.Param.Def (
                        .....
                        .....
                        ),
      ......
      ......
```

```
                        ]];
Real BysMcmc::Bsr::DynHlm::DBApi::ModSes.Lat.Create (
      DBConnection::BsrHlm::_.defaultDB,
      id_model,
      id_session,
      // Nombre del nodo latente
      id_node = "Lat.Reg",
      // Descripcion del nodo latente
      description = "Establece un nodo regresivo latente...",
      //Nivel de latencia en la jerarquía
      Real level = 1,
      // Orden de aparición dentro del nivel
      Copy(level.1_number := level.1_number+1),
      father_parameters,
      node_childs,
      // Valor de la sigma del bloque latente conjunto
      Real sigma = 0.1,
      // Matrix X del nodo latente
      Matrix X
      )
};
```

Nodos latentes con combinacion de output.

```
Real {
   Set father_parameters = [[
      NameBlock Lam1 =  ModSes.Lat.Father.Param.Def (
                        Text id_parameter = "hiper1",
                        Real init_value   = 0.1,
                        Real min          = 0,
                        Real max          = 1
                        ),
      NameBlock Lam2 = ModSes.Lat.Father.Param.Def (
                        ......
                        ......
                        ),
      ........
      ........
                          ]];
   // Set de parámetros hijo. Es un set con tantos NameBlocks como
   // paramétros hijo tenga el nodo latente regresivo.
   Set node_childs = [[
      NameBlock beta1 = ModSes.Lat.Child.Param.Def (
                        Text id_node = "node1",
                        Text id_parameter = "beta1",
                        Real sigma_factor = 1
                        ),
      NameBlock beta2 = ModSes.Lat.Child.Param.Def (
                        .....
                        .....
                        ),
```

```
        ......
        ......
                        ]];

        Real ModSes.Lat.OutComb.Create (
        DBConnection::BsrHlm::_.defaultDB,
        id_model,
        id_session,
        // Nombre del nodo latente
        id_node = "Lat.Reg",
        // Descripción del nodo latente
        description = "Establece un nodo regresivo latente...",
        // Nivel de latencia en la jerarquía
        Real level = 1,
        // Orden de aparición dentro del nivel
        Copy(level.1_number := level.1_number+1),
        father_parameters,
        node_childs,
        // Valor de Sigma para todo el nodo latente
        Real sigma = 0.1,
        // Matriz X del nodo latente
        Matrix X,
        // Matriz Y del nodo latente
        Matrix Y,
        // Un Set de Reales. Cada valor real especifica el i-ésimo valor
        // de sigma de la i-ésima ecuación del nodo latente
        Set SigmaLat = [[
        Real sigma1 = 0.1
        Real sigma2 = 0.2
        ....
        ....
        ]])
};
```

Nodo a priori.

```
Real{
  BysMcmc::Bsr::DynHlm::DBApi::ModSes.Pri.Homog_1.Create(
    DBConnection::BsrHlm::_.defaultDB,
    id_model,
    id_session,
    // identificación del nodo
    id_node = "Pri",
    // descripción del nodo
    description = "Prior para..."
    // Conjunto de NameBlock. Cada NameBlock especifica que
    // parámetro o hiperparámetro es el que tiene el a priori.
    [[
       NameBlock hiper1 = [[
                          // Nombre del nodo donde está hiper1
```

```
                                Text _.id_node = "node_hiper1",
                                // Nombre del parámetro que tiene el prior
                                Text _.id_parameter = "Lam1",
                                // Valor medio de la Normal prior
                                Real _.average = 0.1,
                                // Valor de sigma de la Normla prior
                                Real _.sigma = 0.01
                              ]]
        NameBlock hiper2 = [[
                                ....
                                ....
                              ]]
        ....

    ]]
    )
};
```

Nodo a priori combinación del output.

```
Real {
  BysMcmc::Bsr::DynHlm::DPApi::ModSes.Pri.Out.Comb.Create(
    DBConnection::BsrHlm::_.defaultDB,
    id_model,
    id_session,
    // identificación del nodo
    id_node = "Pri.OutComb",
    // descripción del nodo
    description = "Prior para...",
    Copy(level.2_number := level.2_number+1),
    // Set de sets, cada set contiene dos reales que especifican
    // el valor de la media y sigma para cada ecuacion del nodo a priori
    [[
      [[ mu_1, sigma_1 ]],
      [[ mu_2, sigma_2 ]],
         ............
    ]],
    // Set de NameBlocks, tantos NameBlocks como parámetros entran
    // en la combinación lineal del output
    [[
      NameBlock param1 = [[
                          // Nombre de ese parámetro
                          Text _.id_node = "param1",
                          // Localización de ese parámetro
                          Text _.id_parameter = "node.param1"
                         ]],
      NameBlock param2 = [[
                          .....
                         ]]
    ]],
```

```
    // Matriz Y responsable de la combinación lineal del output
    Matrix Y
    )
}
```

Nodo restricción de orden.

```
Real {
  BysMcmc::Bsr::DynHlm::DBApi::ModSes.Mix.OrderRelation.Create(
    DBConnection::BsrHlm::_.defaultDB,
    id_model,
    id_session,
    id_node_param1, "param1",
    /* <= */ // param1 <= param2
    id_node_param2, "param2")
     };
```

Nodo restricción de orden con combinación lineales.

```
Real{
  BysMcmc::Bsr::DynHlm::DBApi::ModSes.Mix.Constraint.Create(
    DBConnection::BsrHlm::_.defaultDB,
    id_model,
    id_session,
    id_session,
    // El límite por la izquierda puede ser -1/0 o ? en el caso de
    // que no haya límite por la izquierda
    Real leftBound = -1/0,
    // Set de NameBlocks que especifica la información de cada
    // uno de los parámetros que entran en la combinación lineal
    [[
      NameBlock param1 =  [[
                            // Localización del parámetro
                               Text _.id_node = node.param1,
                            // Nombre del parámetro
                               Text _.id_parameter = param1,
                            // Coeficiente asociado a la combinación lineal
                               Real _.coef = 0.3
                          ]]
      NameBlock param2 =  [[
                               .....
                               .....
                          ]]


    ]],
    // El límite por la derecha puede ser -1/0 o ? en el caso de
    // que no haya límite por la derecha
    Real rightBound = 0
  )
}
```

Finalmente la llamada al estimador BSR se realiza por el siguiente código:

```
NameBlock bsr.config =
[[
  //MCMC dimensions
  Real mcmc.burnin    =  100;
  Real mcmc.sampleLength = 2000;
  Real mcmc.cacheLength  =  100;
  //Basic master configuration
  Real basic.cholesky.epsilon = 1.E-10;
  Real basic.cholesky.warningFreq = 100;
  Real basic.truncMNormal.gibbsNumIter = 5;
  //Report configuration
  Real report.raftery.diag.q   = 0.025;
  Real report.raftery.diag.r   = 0.007;
  Real report.raftery.diag.s   = 0.950;
  Real report.raftery.diag.eps = 0.001;
  Real report.acf.lag          = 20,
  Real report.histogram.parts  = 100;
  Real report.kerDens.points   = 0;
  Real report.kerDens.numIter  = 2;
  Real report.kerDens.epsilon  = 0.001;
  //Generic flags
  Real do.resume          =  False;
  Real do.report          =  True;
  Real do.eval            =  True;
  Real do.linear.effects  =  True
]];
////////////////////////////////////////////////////////////

////////////////////////////////////////////////////////////
// Estimación del modelo con el sistema Bsr::DynHlm
////////////////////////////////////////////////////////////
Set bsr.estim = {BysMcmc::Bsr::DynHlm::DBApi::Estim(
  DBConnection::BsrHlm::_.defaultDB,
  id_model,
  id_session,
  Path::_.data.out,
  series.handler,
  bsr.config
)};

// Desconectamos de la base de datos
  Real DBConnection::BsrHlm::Close(0);
```

CAPíTULO 16

Casuística

Se describen diferentes casos y modalidades de uso del sistema Bsr-Hlm, especialmente indicados para sistemas de modelación masiva de series de demanda, pero que pueden muchas veces extrapolarse a otros fenómenos.

16.1. Introducción

En la modelación masiva de los Sistemas de Atención Dinámica de la Demanda es bastante habitual que el número de variables crezca desmesuradamente dada la complejidad de los sistemas de promoción, efectos calendario, meteorológicos, etc.; que se entrelazan entre sí en formas que no siempre son fáciles de parametrizar escuetamente. Al aumentar los grados de libertad, los modelos estimados por máxima verosimilitud de forma independiente pueden sobreajustar los datos, con consecuencias fatales en la inferencia extramuestral que de ellos se pueda desprender.

Uno de los objetivos fundamentales de la estimación bayesiana de un Modelo Jerárquico Lineal (HLM, Hierarquical Lineal Model) es restringir de forma probabilística el conjunto de valores que pueden adoptar los parámetros que intervienen en los distintos modelos, merced a todo el conocimiento a priori de que se disponga, para reducir los grados de libertad reales y alcanzar una mayor superficie de contraste.

Por razones de eficiencia y simplicidad algorítmica, las restricciones probabilísticas han de tener concretamente distribuciones multinormales truncadas. Por lo tanto, las relaciones entre un conjunto de variables aleatorias se deberán expresar paramétricamente diciendo que dichas variables, o cierta combinación lineal de ellas, tienen una distribución conjunta multinormal truncada por un conjunto de restricciones desigualdad lineal, con un vector de medias y una matriz de covarianzas que debe explicitar el analista para ajustar su conocimiento a priori.

16.2. Restricciones de homogeneidad

Este tipo de restricciones aparecen cuando se sabe que determinados parámetros no deberían ser demasiado distintos de ciertos otros por responder a fenómenos del mismo tipo en circunstancias parecidas. Se trata simplemente de aplicar el principio de continuidad que es uno de los pilares de la ciencia moderna, restringiendo de forma probabilística el conjunto de valores que pueden adoptar los parámetros que intervienen para reducir la libertad de movimiento de los parámetros y alcanzar una mayor superficie de contraste sin restar la dimensionalidad necesaria para adaptarse a la complejidad intrínseca a la naturaleza del problema.

16.2.1. Homogeneidad simple o unifactorial. Tenemos m nodos observados $\left\{N_j^{\text{obs}}\right\}_{j=1\ldots m}$ en los que aparece en cada uno una variable $\beta^{(1)}_{k[i,j]}$ referida a un mismo input i para todos ellos, y que presumimos que no debe variar mucho de uno a otro. Con $k[i,j]$ representamos la función de indexación que devuelve la variable asociada al i -ésimo input para el j-ésimo nodo observado. En el caso de distribución de prensa valdría de ejemplo una promoción concreta que afecta a una región geográfica específica.

La restricción de homogeneidad que se desea imponer es que dichas variables han de estar en torno a un mismo valor fijo con errores independientes entre sí. Para ello se definirá un nodo latente con una sola variable por cada input y matriz de covarianzas diagonal dependiente tanto del input como del output.

El usuario tiene que definir los valores σ_k con los que controlará el grado de homogeneidad que quiere forzar. En ausencia de información más concreta, un criterio bastante razonable para

reducir las hipótesis *a priori* a un sólo parámetro, es hacer que las varianzas sean inversamente proporcionales a las longitudes muestrales de cada nodo observado.

$$\sigma^2_{k[i,j]} = \sigma^2_{k[i,1]} \frac{T_j}{T_1} \forall j = 1 \dots m \tag{16.2.1}$$

Por supuesto que el usuario puede establecer otro tipo de configuración de varianzas e incluso establecer covarianzas no nulas entre los distintos nodos observados, pero ello debería de estar fundamentado en un conocimiento *a priori* al menos tan sólido como la proporcionalidad de la varianza muestral respecto al inverso de la longitud de la muestra.

Si además se quiere forzar que las variables de los nodos observados estén en el entorno de un valor fijo $\overset{\circ}{\mu}_i$ para cada input conocido como resultado, por ejemplo, de una estimación global previa, entonces bastaría con añadir a lo anterior un nodo *a priori* con matriz de covarianzas diagonal

$$\beta_i^{(2)} \sim N(\overset{\circ}{\mu}_i, \overset{\circ}{\Sigma}_i) \wedge \overset{\circ}{\Sigma}_{i,i'} = 0 \forall i \neq i' \wedge \overset{\circ}{\Sigma}_{i,i} = \overset{\circ}{\sigma}_i^2 \tag{16.2.2}$$

Parte 4

Referencias

Bibliografía

[Bayesian Data Analysis] Bayesian Data Analysis, by Andrew Gelman, John B. Carlin, Hal S. Stern, and Donald B. Rubin

[Box-Jenkins] George Box, Gwilym M. Jenkins, Time Series Analysis: Forecasting & Control, 1976

[ARMS] Gilks, W. R., Best, N. G. and Tan, K. K. C. (1995) Adaptive rejection Metropolis sampling. Applied Statistics, 44, 455-472.

[Sampling Truncated Multinormal] Efficient Gibbs Sampling of Truncated Multivariate Normal with Application to Constrained Linear Regression. Gabriel Rodriguez-Yam, Richard A. Davis, and Louis L. Scharf. http://www.stat.colostate.edu/~rdavis/papers/CLR.pdf

[One Sample Wilcoxon Test] Small Sample Power of the One Sample Wilcoxon Test for Non-Normal Shift Alternatives. Harvey J. Arnold. Source: Ann. Math. Statist. Volume 36, Number 6 (1965), 1767-1778. http://projecteuclid.org/DPubS?service=UI&version=1.0&verb=Display&handle=euclid.aoms/1177699805

[ARCH in UK] Robert F. Engle, Autoregressive Conditional Heteroscedasticity with Estimates of Variance of United Kingdom Inflation, 1982

[GARCH] Tim Bollerslev, Generalized Autoregressive Conditional Heteroskedasticity, 1986

[User Guide for CHOLMOD] User Guide for CHOLMOD: a sparse Cholesky factorization and modification package. Timothy A. Davis Dept. of Computer and Information Science and Engineering Univ. of Florida, Gainesville, FL. http://www.cise.ufl.edu/research/sparse/cholmod/CHOLMOD/Doc/UserGuide.pdf

[The quadprog R Package] The quadprog Package: Functions to solve Quadratic Programming Problems.S original by Berwin A. Turlach <berwin.turlach@anu.edu.au> R port by Andreas Weingessel <Andreas.Weingessel@ci.tuwien.ac.at>. http://cran.r-project.org/web/packages/quadprog/quadprog.pdf

[CODA] CODA: Convergence diagnosis and output analysis software for Gibbs sampling output, Version 0.30 (1995) by Nicky Best, Mary Kathryn Cowles, Karen Vines http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.48.6445&rep=rep1&type=pdf