

Национальный исследовательский университет «Высшая школа экономики»

КУРСОВАЯ РАБОТА

Байесовский подход в задачах определения индивидуальных
предпочтений

Выполнил: А. А. Кондрашов

Факультет: МИЭФ

Курс: 3

Научный руководитель: Б. Б. Демешев

Москва
2016

Содержание

1 Введение	2
2 Подготовительный этап	3
2.1 R	3
2.2 Установка RStan	3
2.3 Необходимые пакеты	4
3 Введение в Байесовский подход	5
4 Stan	7
4.1 Немного о вероятностном программировании	7
4.2 Как работает Stan?	8
4.3 Примеры моделей в Stan	9
4.3.1 Линейная модель с одной объясняющей переменной	9
4.3.2 Линейная модель с несколькими объясняющими переменными .	13
4.3.3 Модели бинарного выбора	18
4.3.4 Временные ряды	23
4.4 Многоуровневые модели	28
4.5 Способы проверки цепей на сходимость	36
4.6 Прогнозирование в STAN	44
4.6.1 Блок generated quantities	44
4.6.2 Prophet	48
4.7 Rstanarm	50
4.7.1 Линейные модели	50
4.7.2 Модели бинарного выбора	53
4.7.3 Прогнозирование в rstanarm	57
5 Практическая часть	59
5.1 Индивидуальные предпочтения в телекоме	59
6 Заключение	64
7 FAQ	65
7.1 Критерии loo и waic	65
7.2 На что влияет количество цепей и итераций?	65
7.3 Как выбрать априорное распределение?	65
7.4 Что такое warmup и sampling?	66
7.5 Марковские цепи	66
7.6 MCMC — алгоритм Монте-Карло по схеме Марковской цепи	67
Список литературы	69

1 Введение

Байесовский подход — относительно новое слово в сфере статистики, эконометрики, а также искусственного интеллекта. Несмотря на то, что Томас Байес написал свое эссе¹ ещё в 1763 году, развиваться этот метод начал только в конце прошлого века. Это связано с тем, что используемые в подходе алгоритмы требуют серьёзных объёмов вычислений и возможны лишь с помощью ЭВМ. С развитием технологий машинного обучения этот метод набирает популярность.

К сожалению, на данный момент Байесовский подход мало распространён в России, не в последнюю очередь из-за сложности математической части и небольшого количества информации на русском языке. Особенно это касается эконометрических и статистических исследований с помощью Байесовских моделей. С помощью этой работы я хочу внести свой вклад в популяризацию Байесовского подхода, а также языка Stan, который, на мой взгляд, является очень удачным языком моделирования, как в плане эффективности, так и в плане естественности. Как и о Байесовском подходе, о языке Stan имеется крайне малое количество литературы на русском языке, за исключением работ отдельно взятых учёных и исследователей.

Данное руководство рассчитано как на опытных пользователей, так и для новичков в Байесовских методах. Работа состоит из 5 основных частей: в начале будет предложена инструкция к установке Stan и необходимых пакетов в R. Затем следует краткое введение в теорию Байесовского подхода. Далее речь пойдёт о моделях в Stan, с примерами и некоторыми нюансами. После этого идёт небольшая практическая часть: на базе набора данных о выборе индивидов параметров тарифов сотовой связи мы построим модель. После заключения также есть небольшая секция FAQ, в которой я поместил разъяснения для некоторых нюансов.

¹Thomas Bayes, «An Essay towards solving a Problem in the Doctrine of Chances», Philosophical Transactions of the Royal Society of London 53 (1763), 370–418

2 Подготовительный этап

В данной секции я укажу об необходимых (*и, возможно, дополнительных*) элементах для работы со Stan в R.

2.1 R

В первую очередь, нам понадобится сама среда R (или её обновление). Конечно, я считаю, что читатели данной работы компетентны выполнить этот шаг самостоятельно, но во имя формальности я все-таки кратко опишу процедуру установки R:

Последнюю версию можно получить, пройдя по ссылке: <https://cran.r-project.org>. Крайне полезным дополнением будет Rstudio, <https://www.rstudio.com>, в котором можно редактировать код R и Stan. Кроме того, эта программа очень удобна и интуитивно понятна в вопросах интерфейса.

2.2 Установка RStan

Работа Stan подразумевает одновременное взаимодействие нескольких программ (R, компилятора C++ и, собственно, Stan), поэтому установка хоть и не сложная, но выполнять её лучше по подробной инструкции: <https://github.com/stan-dev/rstan/wiki/RStan-Getting-Started>. Суть её сводится к тому, что Вам необходимо установить R → Toolchain (компилятор C++) → настроить конфигурацию в R → и, наконец, установить сам RStan. **Я пропущу установку пререквизитов** и перейду сразу к установке RStan:

```
#Install RStan
install.packages("rstan", repos = "https://cloud.r-project.org/",
                  dependencies = TRUE)
```

Очень важно, чтобы вместе с `rstan` были установлены и все зависимые пакеты. После установки следует **перезагрузить R**. Осталось проверить работоспособность toolchain:

```
fx <- inline::cxxfunction(signature(x = "integer", y = "numeric") ,
                           'return ScalarReal(INTEGER(x)[0] * REAL(y)[0]) ;')
fx( 2L, 5 )
## [1] 10
```

Должно получиться 10. В противном случае, повторите установку.

2.3 Необходимые пакеты

```
# Собственно, запуск самого Stan
library(rstan)
# Пакет для визуального анализа полученной модели на Stan
library(shinystan)
#Данные команды предназначены для ускорения Stan
#путём задействия большего числа ядер процессора
rstan_options(auto_write = TRUE)
options(mc.cores = parallel::detectCores())
# Пакет для сравнения моделей
library(loo)
# Готовые модели Stan
library(rstanarm)
# Вывод графиков с кириллицей в pdf
library(Cairo)
# Всякие полезные функции для данных
library(psych)
library(tidyverse)
library(memisc)
# Для графиков
library(bayesplot)
# Данные по Титанику
library(titanic)
# Для предельных эффектов
library(erer)
# Для загрузки временных рядов
library(quantmod)
# Пакет для временных рядов
library(zoo)
library(forecast)
library(haven)
library(reshape2)
library(prophet)
```

3 Введение в Байесовский подход

Иногда встречаются статистические задачи, в которых информация о состоянии мира может быть изменена после получения набора данных. То есть перед решением задачи мы можем сформировать свое мнение о приемлемых моделях и дать им вероятности в зависимости от нашего доверия той или иной модели. В качестве одного из способов, корректирующего приемлемость этих моделей, можно использовать байесовский подход, в основе которого лежит небезызвестная теорема Байеса.

В отличие от классических методов, применяемых в эконометрике (основывающиеся на работах Фишера, Ньюмена, Пирсона и многих других), байесовские методы стали развиваться преимущественно в последнее время. Это связано в первую очередь с тем, что расчеты, основанные на байесовском подходе, требуют значительных вычислительных затрат, и стали возможны лишь с развитием вычислительной техники. Кроме того, байесовский подход имеет ряд привлекательных преимуществ.

Использование байесовского подхода начинается с расстановки степеней доверия (в виде вероятностей) для разных моделей. Делается это достаточно вольно, но с умом. После получения данных, мы меняем вероятности для первоначальных моделей.

В реальных задачах статистические данные могут отсутствовать, и использование методов, основанных на частотном подходе, может быть ошибочным. Также информация может содержать субъективные оценки (например, экспертные суждения), а может и вовсе быть абсолютно новой. В таких задачах изучаемый нами подход может принести большую пользу.

Как уже говорилось выше, в основе байесовского подхода лежит теорема Байеса. Формула Байеса выглядит следующим образом:

$$Pr(\theta|y) = \frac{Pr(y|\theta)Pr(\theta)}{Pr(y)}$$

А в более общем виде (с несколькими $(\theta_1 \dots \theta_k)$):

$$Pr(\theta_i|y) = \frac{Pr(y|\theta_i)Pr(\theta_i)}{Pr(y)} = \frac{Pr(y|\theta_i)Pr(\theta_i)}{\sum_{j=1}^k Pr(\theta_j)Pr(y|\theta_j)}$$

$Pr(\theta_i)$ интерпретируется как априорное распределение параметра θ_i , а $Pr(\theta_i|y)$ - как апостериорное распределение параметра θ_i , ведь теперь нам известны значения y . Сами θ называются гипотезами, а y - свидетельствами.

Пример 1

Разберем небольшой пример: допустим, что благодаря развитию байесовского подхода, машинного обучения и прочих компьютерных штук, был изобретен искусственный интеллект. Правда, интеллект может быть заточен некую определенную профессию, и, чтобы удовлетворить недостаток специалистов в стране была выпущена партия андроидов-специалистов. Согласно статистике известно, что среди них 25% врачей (med, θ_1), 15% инженеров (eng, θ_2), 50% строителей ($bild, \theta_3$) и 10% - техподдержка андроидов ($tech, \theta_4$) (допустим, что андроидам-инженерам и врачам не стоит

чинить друг друга). Нам нужно узнать, какая профессия была у андроида, случайно отформатировавшего свою память (вместе с серийным номером).

Из задачи мы имеем следующее:

i	1 (med)	2 (eng)	3 (bild)	4 (tech)
$Pr(\theta_i)$	0.25	0.15	0.5	0.1

Теперь перейдем к свидетельствам, или признаками того, что у андроида та или иная работа. Первым признаком андроида специалиста являются зеленая подсветка глаз - это означает, что ИИ исправен и работает корректно. Однако, в связи с условиями работы подсветка может испортиться, а ее замена - дело весьма трудоемкое, и техподдержка не всегда с этим справляется. Допустим следующее:

i	1 (med)	2 (eng)	3 (bild)	4 (tech)
$Pr(\theta_i)$	0.25	0.15	0.5	0.1
$Pr(y_1 \theta_i)$	0.2	0.4	0.7	0.05

Теперь мы можем найти $Pr(\theta_i|y_1)$ или апостериорное распределение для гипотез, используя вышеупомянутую формулу.

$$Pr(\theta_1|y_1) = \frac{Pr(y_1|\theta_1)Pr(\theta_1)}{\sum_{j=1}^k Pr(\theta_j)Pr(y_1|\theta_j)} = \frac{0.2 \times 0.25}{0.25 \times 0.2 + 0.15 \times 0.4 + 0.5 \times 0.7 + 0.1 \times 0.05} = 0.107$$

Аналогичным способом:

$$Pr(\theta_2|y_1) = 0.129$$

$$Pr(\theta_3|y_1) = 0.752$$

$$Pr(\theta_4|y_1) = 0.01$$

Стало быть:

i	1 (med)	2 (eng)	3 (bild)	4 (tech)
$Pr(\theta_i)$	0.25	0.15	0.5	0.1
$Pr(y_1 \theta_i)$	0.2	0.4	0.7	0.05
$Pr(\theta_i y_1)$	0.107	0.129	0.752	0.01

После получения данных y_1 доверие к гипотезам изменилось.

4 Stan

4.1 Немного о вероятностном программировании

В первую очередь хочу упомянуть, что при обсуждении вероятностного программирования стоит абстрагироваться от ассоциаций, которые навязывает слово «программирование». Вероятностное программирование — это, в первую очередь, инструмент для статистического (вероятностного) моделирования, а не для разработки полноценного программного обеспечения. Вероятностная программа работает в двух направлениях: во-первых, она может рассчитать последствия, исходя из предварительных представлений о состоянии мира (иными словами, симулировать выходные данные из имеющихся), а во-вторых, наоборот, может привести возможные объяснения, почему у нас на руках имеются такие данные². Также можно выделить и третье направление:³ мы можем использовать исторические данные, чтобы предсказать будущие значения. Всё это говорит о том, что вероятностный подход достаточно гибкий.

Вероятностное программирование реализуется с помощью соответствующих языков (PPL — probabilistic programming language). Это высокоуровневые языки, предназначенные для облегчения определения моделей и автоматических расчётов по ним. Методы, используемые в процессе вычисления (такой как МСМС, о нём ниже), довольно сложные и требуют глубоких знаний в сфере Байесовского вывода. Поэтому важно, чтобы язык был не только гибким в плане возможностей для описания моделей, но и был достаточно простым и интуитивно понятным, чтобы его могло применять большее число пользователей. Языков, специализирующихся в этой сфере, достаточно много, некоторые из них ещё находятся в разработке. Одним из зарекомендовавших себя языков является Stan.

Stan — это вероятностный язык моделирования, то есть язык, предназначенный для описания вероятностных моделей. Он появился благодаря трудам сотрудников Колумбийского Университета в 2012 году (initial release) и назван в честь Станислава Улама — математика, предложившего вычислительный метод Монте-Карло. Целью авторов было создать язык для Байесовского вывода для многоуровневых генерализованных линейных моделей, поскольку существующие на тот момент решения (такие как BUGS⁴ и JAGS⁵) не могли в полной мере справиться с этими моделями. В результате было принято решение отказаться от семплинга Гиббса в пользу более эффективного, а именно гибридного Монте-Карло (HMC). Дополнив алгоритм своими функциями и модификациями, авторы получили свой алгоритм семплинга (NUTS — No-U-Turn Sampler). Разработка языка продолжается.

С точки зрения обычного исследователя, из всех остальных имплементаций вероятностного программирования Stan положительно выделяется своим по-человечески понятным синтаксисом. В этой работе я предлагаю убедиться в этом.

²Cameron Davidson-Pilon, «Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference», Addison-Wesley Professional, 2016, p. 69

³Avi Pfeffer, Stuart Russel, «Practical Probabilistic Programming», Manning Publications, 2016, p. 9.

⁴Bayesian inference Using Gibbs Sampling

⁵Just Another Gibbs Sampler

4.2 Как работает Stan?

Stan имеет несколько интерфейсов, основными же являются три: cmdStan (командная строка), PyStan (Python) и RStan (R). Код на языке Stan является не алгоритмом, а описанием модели в виде блоков. После получения данных (наблюдений) и кода модели Stan переводит их в программу C++, которая, в свою очередь, компилируется под платформу, на которой запускается. Сначала проверяется код модели, а также корректность введённых данных (y и x). Затем генерируется последовательность зависимых, идентично распределённых выборок $\theta^{(1)}, \theta^{(2)}, \dots$, предельное распределение каждого: $p(\theta|y, x)$. Процесс генерации также называется семплингом (или семплированием). Для его осуществления Stan использует метод Монте-Карло по схеме Марковской цепи (MCMC)⁶, а конкретнее гибридный Монте-Карло (или Hamiltonian Monte-Carlo, HMC). Вкратце, MCMC — это метод оценки параметра путём симуляции ожидаемых значений. Последовательные случайные выборы формируют Марковскую цепь, стационарное распределение которой является искомой⁷. Концепт достаточно сложный, поэтому зачастую пользователи относятся к нему как к чёрному ящику. Тем не менее, это важный элемент работы Stan (и всего Байесовского подхода в целом), с которым желательно ознакомиться.

⁶Подробнее об этом методе можно узнать в секции FAQ

⁷Encyclopedia of Biostatistics

4.3 Примеры моделей в Stan

Перед началом, убедитесь, что пакет `rstan` подключен, а для вычислений задействовано максимально возможное число ядер (все команды указаны в секции 2.3). Процесс вычисления трудоёмкий (для железа) и занимает некоторое время.

4.3.1 Линейная модель с одной объясняющей переменной

Возьмём самый простой пример: одинарную линейную регрессию. В привычном нам виде модель будет выглядеть следующим образом:

$$Y_i = \alpha + \beta X_i + \varepsilon_i, \text{ где } \varepsilon_i \sim \mathcal{N}(0, \sigma^2)$$

С помощью нехитрых преобразований получим:

$$Y_i - (\alpha + \beta X_i) \sim \mathcal{N}(0, \sigma^2) \Rightarrow Y_i \sim \mathcal{N}(\alpha + \beta X_i, \sigma^2)$$

А наша модель будет иметь вид:

```
model_lm <- "
data {
  int<lower=0> n;
  vector[n] x;
  vector[n] y;
}
parameters {
  real alpha;
  real beta;
  real<lower=0> sigma;
}
model {
  y ~ normal(alpha + beta * x, sigma);
}"
```

Собственно, вот и всё описание модели.

- В блоке `data` содержится описание данных для ввода
 - `n` - количество наблюдений,
 - `x` - вектор объясняющей переменной,
 - `y` - вектор объясняемой переменной
- Блок `parameters` содержит, согласно своему названию, операторы
- Блок `model` содержит описание модели

Все просто и понятно, не правда ли? Предлагаю протестировать нашу модель на каких нибудь данных. В качестве примера подойдет набор данных `cars`, встроенный в R: он содержит всего две переменные.

```

# Сформируем данные для Stan
cars_data <- list(n = nrow(cars),
                  x = cars$speed,
                  y = cars$dist)

# Запустим вычисление параметров
lmtest_fit <- stan(model_code = model_lm,
                     data = cars_data,
                     iter = 1000,
                     chains = 4)

```

Обратите внимание, что названия вводимых данных в списке (`cars_data` в данном случае) должны соответствовать указанным в модели (`model_lm`). Как и в R, регистр учитывается. Мы возьмём четыре цепи и 1000 итераций в каждой из них, чтобы у нас была возможность убедиться в том, что цепи сойдутся к апостериорному распределению.

Результатом исполнения команды `stan` станет табличка следующего вида:

```

lmtest_fit

## Inference for Stan model: 1333cca4859e0a7f7b3b49d2c221713f.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##           mean   se_mean     sd    2.5%    25%    50%    75%   97.5% n_eff Rhat
## alpha    -17.58    0.24  6.98  -31.24  -22.21  -17.47  -12.73  -3.96    841    1
## beta      3.93    0.02  0.43     3.09     3.65     3.93     4.21     4.79    813    1
## sigma    15.81    0.05  1.69    12.89    14.58    15.71    16.89    19.43   1209    1
## lp__   -159.47    0.04  1.20  -162.43  -160.07  -159.19  -158.57  -158.05   1028    1
##
## Samples were drawn using NUTS(diag_e) at Fri Mar  3 13:45:53 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).

```

В колонке `mean`, очевидно, указаны средние оценки параметров⁸. `sd` - стандартное отклонение по выборке. `se_mean` - стандартная ошибка, однако, рассчитывается она как `sd/sqrt(n_eff)`, а `n_eff` мы используем так как выборка зависимая. Далее идут квантили оценок, уже упомянутый размер эффективной выборки `n_eff` и фактор уменьшения масштаба `R_hat`, который является мерой сходимости цепи. Если значение `R_hat` равно единице (с небольшими отклонениями), то можно считать, что цепь сошлась. Если же значения значительно разнятся, то цепь не сошлась.

⁸Загадочный параметр `lp__` является логарифмом апостериорной плотности (log posterior density) и его сходимость тоже важна. Подробнее можно прочесть в инструкции Stan (стр. 351 и стр. 578) или в Интернете, например здесь: <https://www.jax.org/news-and-insights/jax-blog/2015/october/lp-in-stan-output>

Также можно вывести более подробный результат, воспользовавшись функцией `summary()`. В отчёте также будет информация по цепям⁹.

```
summary(lmtest_fit)

## $summary
##      mean se_mean    sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
## alpha -17.58 0.2406 6.98 -31.24 -22.21 -17.47 -12.73 -3.96 841 1.003
## beta  3.93 0.0151 0.43   3.09   3.65   3.93   4.21   4.79 813 1.003
## sigma 15.81 0.0487 1.69  12.89  14.58  15.71  16.89 19.43 1209 1.000
## lp__ -159.47 0.0375 1.20 -162.43 -160.07 -159.19 -158.57 -158.05 1028 0.999
##
## $c_summary
## , , chains = chain:1
##
##      stats
## parameter   mean    sd   2.5%   25%   50%   75% 97.5%
## alpha     -17.93 6.75 -30.82 -22.8  -17.74 -13.32 -5.60
## beta      3.96 0.41   3.14   3.7   3.95   4.23   4.76
## sigma     15.78 1.71  12.88  14.5   15.64  16.87 19.48
## lp__     -159.43 1.22 -162.26 -160.0 -159.10 -158.51 -158.04
##
## , , chains = chain:2
##
##      stats
## parameter   mean    sd   2.5%   25%   50%   75% 97.5%
## alpha     -17.38 7.205 -30.86 -22.23 -17.47 -12.7 -3.46
## beta      3.92 0.443   3.01   3.63   3.94   4.2   4.79
## sigma     15.81 1.644  12.92  14.63  15.75  17.0 19.13
## lp__     -159.49 1.130 -162.36 -160.03 -159.23 -158.6 -158.10
##
## , , chains = chain:3
##
##      stats
## parameter   mean    sd   2.5%   25%   50%   75% 97.5%
## alpha     -17.96 7.007 -32.19 -22.44 -17.49 -13.19 -4.74
## beta      3.96 0.442   3.09   3.66   3.94   4.24   4.82
## sigma     15.80 1.618  12.81  14.64  15.80  16.87 19.18
## lp__     -159.48 1.193 -162.37 -160.08 -159.18 -158.62 -158.04
##
## , , chains = chain:4
##
##      stats
## parameter   mean    sd   2.5%   25%   50%   75% 97.5%
## alpha     -17.1 6.911 -30.8 -21.38 -17.03 -12.03 -3.80
## beta      3.9 0.424   3.1   3.59   3.91   4.18   4.73
## sigma     15.8 1.805  12.9  14.49  15.65  16.86 20.18
## lp__     -159.5 1.267 -162.6 -160.15 -159.23 -158.51 -158.04
```

⁹Небольшой совет: командой `options(digits = ...)`, можно уменьшить количество символов в выводе. Так табличка компактнее и не разбивается на части при выводе.

Объект `lmtest_fit` является объектом `stanfit`, как и многие другие сложные объекты, к примеру, объекты с геоданными, `SpatialPolygonDataFrame`, о которых я писал в прошлой работе, состоит из слотов, к которым можно обратиться, введя команду типа `[objectname]@[slotname]`. Нас же больше всего интересует таблица с результатами. В чистом виде (как слот) в объекте `lmtest_fit` она отсутствует, однако, мы можем получить апостериорную выборку параметров введя:

```
lm_array <- as.array(lmtest_fit)
```

Полученный объект является трёхмерным массивом [номер итерации, номер цепи, параметр]. То есть, допустим, если нас интересует параметр β , то мы можем получить его, указав соответствующий номер параметра. Посмотрим на первые 5 значений каждой цепи:

```
head(lm_array[, , 2], 5) #выбран параметр beta

##          chains
## iterations chain:1 chain:2 chain:3 chain:4
##      [1,]    4.486   3.389   3.470   3.496
##      [2,]    4.589   3.493   3.313   3.777
##      [3,]    3.125   3.612   3.883   3.723
##      [4,]    4.041   3.008   4.017   3.577
##      [5,]    4.633   3.005   4.179   3.752
```

В качестве интереса, можно провести небольшое сравнение с обычным МНК:

```
OLS_fit <- lm(dist ~ speed, data = cars)
summary(OLS_fit)

##
## Call:
## lm(formula = dist ~ speed, data = cars)
##
## Residuals:
##     Min      1Q  Median      3Q     Max 
## -29.07  -9.53  -2.27   9.21  43.20 
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)    
## (Intercept) -17.579     6.758   -2.60    0.012 *  
## speed        3.932     0.416    9.46  1.5e-12 *** 
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 15.4 on 48 degrees of freedom
## Multiple R-squared:  0.651, Adjusted R-squared:  0.644 
## F-statistic: 89.6 on 1 and 48 DF,  p-value: 1.49e-12
```

и обнаружить, что полученные значения очень похожи:

```
OLS_fit$coefficients

## (Intercept)      speed
## -17.579       3.932

c(mean(lm_array[, , 1]), mean(lm_array[, , 2]))

## [1] -17.585   3.934
```

4.3.2 Линейная модель с несколькими объясняющими переменными

Теперь построим регрессию от нескольких объясняющих переменных. В качестве примера возьмём набор данных `mtcars`, содержащий 11 переменных. Построим следующую модель:

$$mpg_i = \alpha + \beta_1 wt_i + \beta_2 am_i + \beta_3 hp_i + \varepsilon_i,$$

где $\varepsilon_i \sim \mathcal{N}(0, \sigma^2)$

В Stan же её можно отобразить как:

```
model_lm2 <- "
data {
  int<lower=0> n;
  int<lower=0> k;
  matrix[n, k] X;
  vector[n] y;
}
parameters {
  real alpha;
  vector[k] beta;
  real<lower=0> sigma;
}
model {
  alpha ~ normal(0, 1000);
  y ~ normal(alpha + X * beta, sigma);
}"
```

Существенных изменений нет, однако теперь, поскольку у нас несколько регрессоров, мы указали их как матрицу X (при желании, в неё можно спрятать и единичный вектор). Также я явно ввел априорное распределение для α , допустив, что оно распределено нормально: $\alpha \sim \mathcal{N}(0, 1000)$. При этом Stan неявно задаст равномерное распределение, а именно: $\alpha \sim Uniform[-\infty, +\infty]$. Значения β теперь тоже должны храниться в соответствующем векторе.

Соберём данные для нашей модели:

```
#Сформируем данные для Stan
cars_data <- list(n = nrow(mtcars),
                  k = 3,
                  X = cbind(mtcars$wt, mtcars$am, mtcars$hp),
                  y = mtcars$mpg)
```

И запустим её:

```
lm3_fit1 <- stan(model_code = model_lm2,
                   data = mtcars_data1,
                   iter = 1000,
                   chains = 4)
```

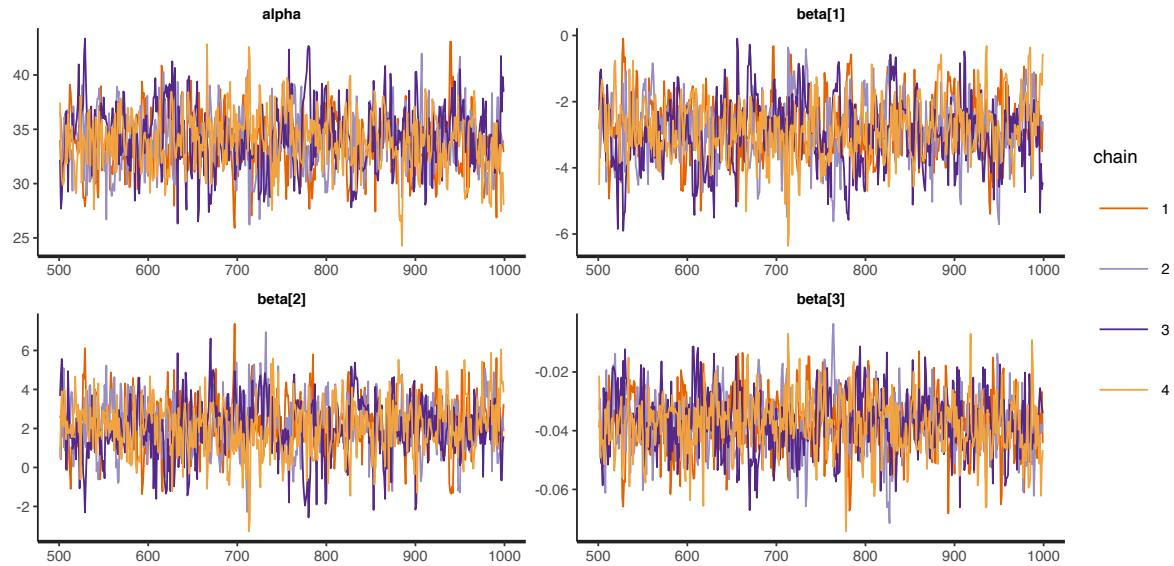
Получим следующий результат:

```
lm3_fit1

## Inference for Stan model: fcc6bab283f8de0c4f27e9e911d54621.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##           mean se_mean    sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
## alpha     33.91    0.12  2.72  28.71  32.13  33.94  35.66 39.37    540 1.01
## beta[1]   -2.85    0.04  0.93  -4.71  -3.46  -2.87  -2.24 -0.96    497 1.01
## beta[2]    2.13    0.06  1.39  -0.82   1.27   2.11   3.04  4.84    613 1.01
## beta[3]   -0.04    0.00  0.01  -0.06  -0.04  -0.04  -0.03 -0.02    804 1.00
## sigma     2.65    0.01  0.36   2.06   2.39   2.64   2.88  3.45   1070 1.00
## lp__    -45.44    0.06  1.62 -49.33 -46.34 -45.12 -44.24 -43.27    716 1.00
##
## Samples were drawn using NUTS(diag_e) at Tue Apr 11 16:11:36 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

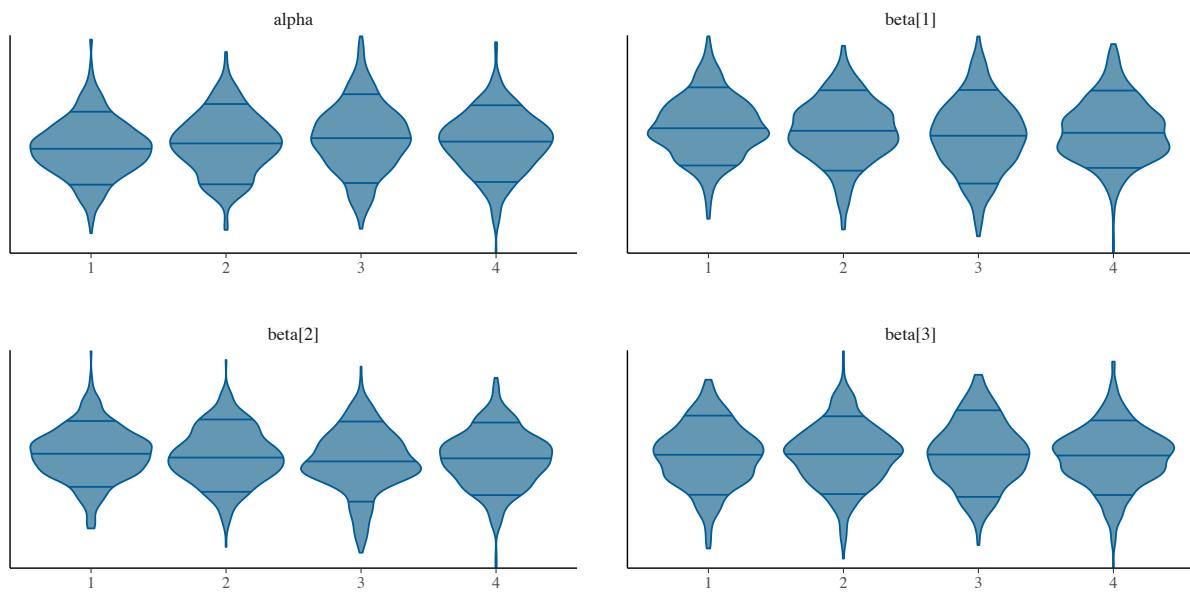
Командой `traceplot()` (или абсолютно аналогичной ей `stan_trace()`) можно попробовать визуально оценить сходимость рядов, предварительно указав искомый параметр. Допустим, меня интересуют основные коэффициенты регрессии.

```
stan_trace(lm3_fit1, pars = c("alpha", "beta[1]", "beta[2]", "beta[3]"))
```



Также добавим скрипичный график:

```
mcmc_violin(as.array(lm3_fit1),
             pars = c("alpha", "beta[1]", "beta[2]", "beta[3]"))
```



Вроде никаких отклонений не заметно.

Предлагаю сделать несколько более строгое предположение касательно α . Допустим, что α распределено нормально, но с маленькой дисперсией. Тогда наша модель примет вид:

```

restrictive_lm2 <- "
data {
  int<lower=0> n;
  int<lower=0> k;
  matrix[n, k] X;
  vector[n] y;
}
parameters {
  real alpha;
  vector[k] beta;
  real<lower=0> sigma;
}
model {
  alpha ~ normal(0, 2);
  y ~ normal(alpha + X * beta, sigma);
}"

```

Запустим вычисления:

```

lm3_fit1_2 <- stan(model_code = restrictive_lm2,
                     data = mtcars_data1,
                     iter = 1000,
                     chains = 4)

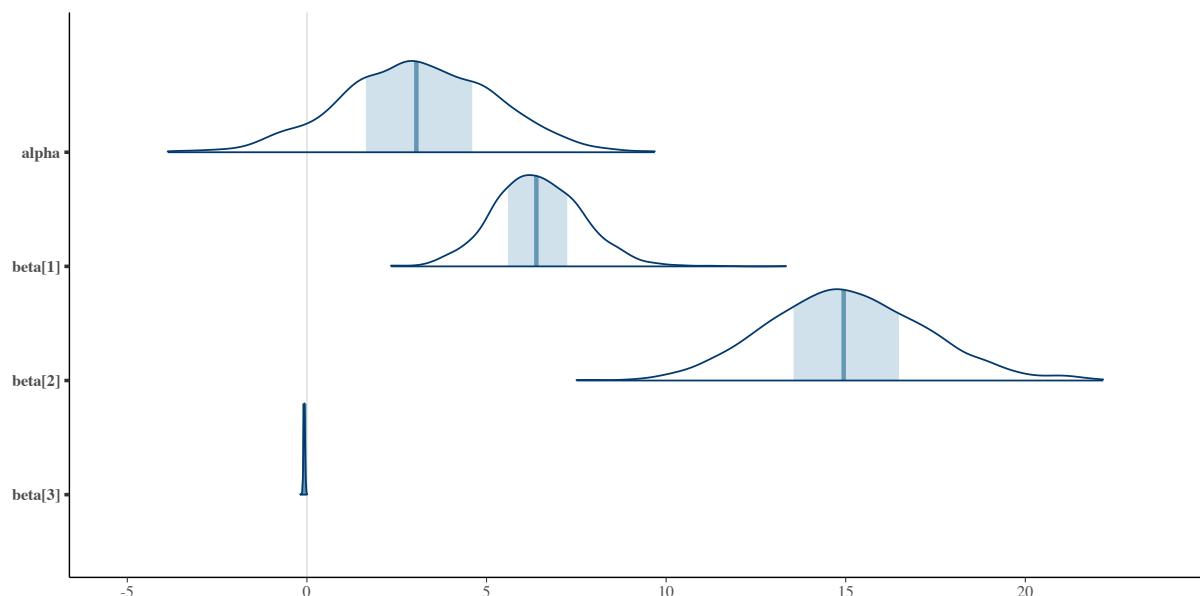
```

И посмотрим на результат:

```

mcmc_areas(as.array(lm3_fit1_2),
            pars = c("alpha", "beta[1]", "beta[2]", "beta[3]"))

```



```
lm3_fit1_2
```

```
## Inference for Stan model: ea31772c637f35fc826b88ef39a761bf.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##          mean se_mean    sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
## alpha     3.10    0.07 2.11 -1.02   1.64   3.05   4.60   7.11 1046 1.00
## beta[1]   6.44    0.05 1.24   4.06   5.60   6.39   7.25   8.95 742 1.01
## beta[2]  15.03    0.06 2.21  10.97  13.55  14.94  16.48  19.51 1203 1.00
## beta[3]  -0.07    0.00 0.02  -0.12  -0.09  -0.07  -0.06  -0.03 789 1.00
## sigma     6.35    0.03 0.97   4.77   5.66   6.24   6.92   8.41 978 1.00
## lp__   -74.10    0.06 1.67 -78.10 -74.92 -73.76 -72.87 -71.88 763 1.00
##
## Samples were drawn using NUTS(diag_e) at Tue Apr 11 16:16:11 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Заметно, что значение α стало значительно ниже. Изменились значения и других параметров. Однако, стоит отметить, что сделанное нами предположение чересчур смелое, и у нас не хватило данных, чтобы оно перестало оказывать влияние на апостериорную выборку. Иными словами, у нас не хватило информации, чтобы изменить первоначальное суждение. Допустим, если в априорном распределении указать $\text{alpha} \sim \text{normal}(0, 10)$, то мы получим следующий результат:

```
lm3_fit1_3
```

```
## Inference for Stan model: 7abdc96e93f248e7817fcc5e11083b8.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##          mean se_mean    sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
## alpha    31.37    0.12 2.85 25.40 29.51 31.45 33.26 36.71 596 1.01
## beta[1]  -2.08    0.04 0.98 -3.99 -2.74 -2.08 -1.48 -0.03 520 1.01
## beta[2]   3.20    0.06 1.50  0.33  2.22  3.17  4.16  6.28 694 1.01
## beta[3]  -0.04    0.00 0.01 -0.06 -0.05 -0.04 -0.03 -0.02 757 1.00
## sigma     2.71    0.01 0.41   2.06   2.42   2.65   2.94   3.64 804 1.00
## lp__   -51.00    0.07 1.74 -55.13 -51.90 -50.68 -49.69 -48.70 666 1.01
##
## Samples were drawn using NUTS(diag_e) at Tue Apr 11 16:27:58 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Который гораздо ближе к тому, что был изначально. Таким образом, наше априорное суждение о распределении параметров обладает влиянием на апостериорное. Графически это можно увидеть, построив графики априорного и апостериорного распределений. Для вывода априорного распределения в Stan, можно попросту написать:

```

for_prior <- "
parameters{
real alpha;
}
model{
alpha ~ normal(0, 1000);
}""

prior <- stan(model_code = for_prior,
iter = 1000,
chains = 4)

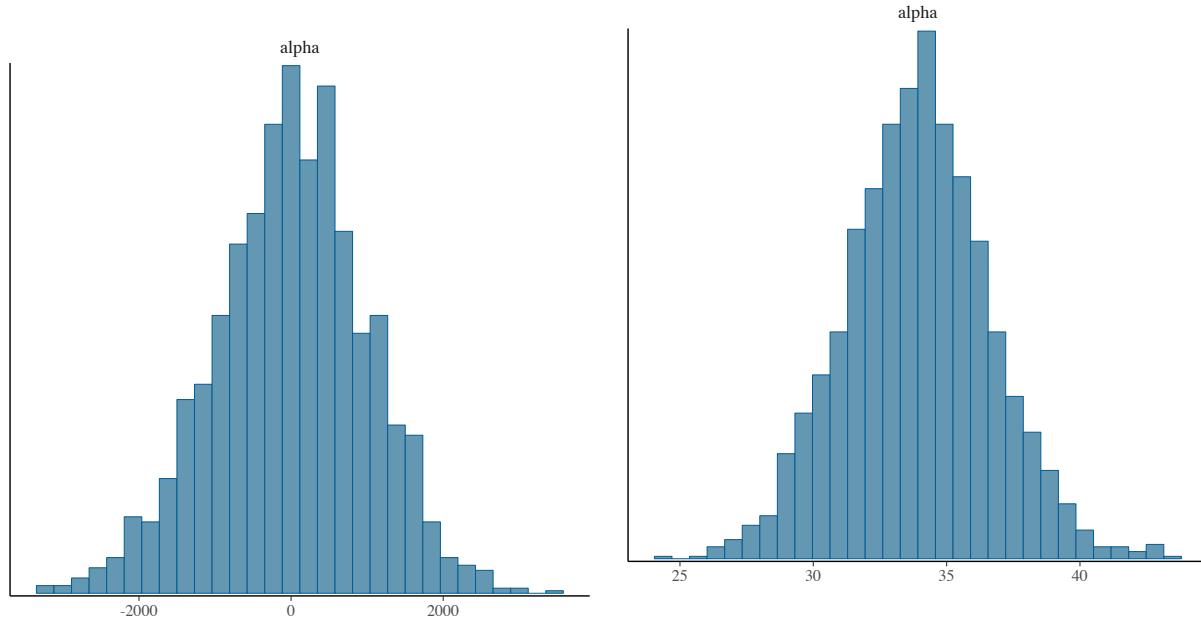
```

А затем построить графики распределений:

```

mcmc_hist(as.array(prior), pars = "alpha")
mcmc_hist(as.array(lm3_fit1), pars = "alpha")

```



Форма распределения осталась прежней, однако постериорное центрировано примерно у 34 и обладает меньшей дисперсией.

4.3.3 Модели бинарного выбора

Перейдём к моделям для предсказывания вероятностей. В этих моделях мы объясняем переменную бинарного типа (y_i), которая может принимать значения 1 или 0. И чтобы её объяснить, мы вводим некую ненаблюдаемую скрытую переменную (y_i^*), так чтобы:

$$y_i = \begin{cases} 0, & \text{если } y_i^* < 0; \\ 1, & \text{если } y_i^* \geq 0 \end{cases}$$

где $y_i^* = \alpha + \beta_1 x_i + \beta_2 z_i + \dots + \varepsilon_i$

В случае логит-модели $\varepsilon_i \sim logistic$, а функция плотности представляет из себя

$$f(t) = \frac{e^{-t}}{(1 + e^{-t})^2}$$

и в целом похожа на $\mathcal{N}(0, 1.6^2)$. В терминах вероятности получим, что:

$$\begin{aligned} P(y_i = 1) &= P(y_i^* \geq 0) = \\ P(\alpha + \beta_1 x_i + \beta_2 z_i + \dots + \varepsilon_i \geq 0) &= P(\varepsilon_i \leq \alpha + \beta_1 x_i + \beta_2 z_i + \dots) = \\ F(\alpha + \beta_1 x_i + \beta_2 z_i + \dots) \end{aligned}$$

Где $F(t)$ - уже функция распределения вероятности. Очевидно, что:

$$P(y_i = 0) = 1 - F(t).$$

Теперь отобразим эту модель в Stan. Практиковаться я предлагаю на наборе данных `titanic`. Отмету, что в R есть и свой, встроенный, набор данных `Titanic`, но он представлен в виде многомерного массива и скучен на переменные. Набор `titanic` можно найти в Интернете, а можно получить, установив пакет `titanic`. Бросим взгляд на наши данные:

```
glimpse(titanic_train, width = 60)

## Observations: 891
## Variables: 12
## $ PassengerId <int> 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, ...
## $ Survived <int> 0, 1, 1, 1, 0, 0, 0, 0, 1, 1, 1, 1, ...
## $ Pclass <int> 3, 1, 3, 1, 3, 3, 1, 3, 3, 2, 3, 1, ...
## $ Name <chr> "Braund, Mr. Owen Harris", "Cumings...
## $ Sex <chr> "male", "female", "female", "female...
## $ Age <dbl> 22, 38, 26, 35, 35, NA, 54, 2, 27, ...
## $ SibSp <int> 1, 1, 0, 1, 0, 0, 0, 3, 0, 1, 1, 0, ...
## $ Parch <int> 0, 0, 0, 0, 0, 0, 1, 2, 0, 1, 0, ...
## $ Ticket <chr> "A/5 21171", "PC 17599", "STON/O2. ...
## $ Fare <dbl> 7.250, 71.283, 7.925, 53.100, 8.050...
## $ Cabin <chr> "", "C85", "", "C123", "", "", "E46...
## $ Embarked <chr> "S", "C", "S", "S", "S", "Q", "S", ...
```

Прогнозировать будем переменную `Survived` по переменным `Sex`, `Age`, `Fare`, `Pclass`. Но для начала надо нужно преобразовать переменную `Sex` в числовой формат, и на всякий случай очистить от пропущенных значений:

```

df_titan <- mutate(titanic_train,
                    Sex = as.numeric(
                        as.factor(titanic_train$Sex)) - 1)
df_titan <- na.omit(df_titan)
glimpse(df_titan$Sex)

##  num [1:714] 1 0 0 0 1 1 1 0 0 0 ...

```

Отлично, теперь `male = 1`, а `female = 0`. Опишем нашу модель в Stan:

```

model_logit <- "
data {
    int<lower=0> N;
    int<lower=0> k;
    vector[N] Sex;
    vector[N] Pclass;
    vector[N] Age;
    vector[N] Fare;
    int<lower=0,upper=1> Surv[N];
}
parameters {
    real alpha;
    real beta[k];
}
model {
    Surv ~ bernoulli_logit(alpha + beta[1] * Sex + beta[2] * Pclass +
                            beta[3] * Age + beta[4] * Fare);
}"

```

Объект с кодом модели пришлось немного преобразить. Во-первых, в этот раз мы ввели каждую переменную отдельно, чтобы их было проще распознавать (и чтобы показать альтернативный способ записи). Во-вторых, описали распределение как `bernoulli_logit`.

Соберем данные и запустим Stan:

```

logit_data <- list(N = nrow(df_titan),
                     k = 4,
                     Sex = df_titan$Sex,
                     Pclass = df_titan$Pclass,
                     Age = df_titan$Age,
                     Fare = df_titan$Fare,
                     Surv = df_titan$Survived)

logittest_fit <- stan(model_code = model_logit,
                       data = logit_data,
                       iter = 1000,
                       chains = 4)

```

Результат:

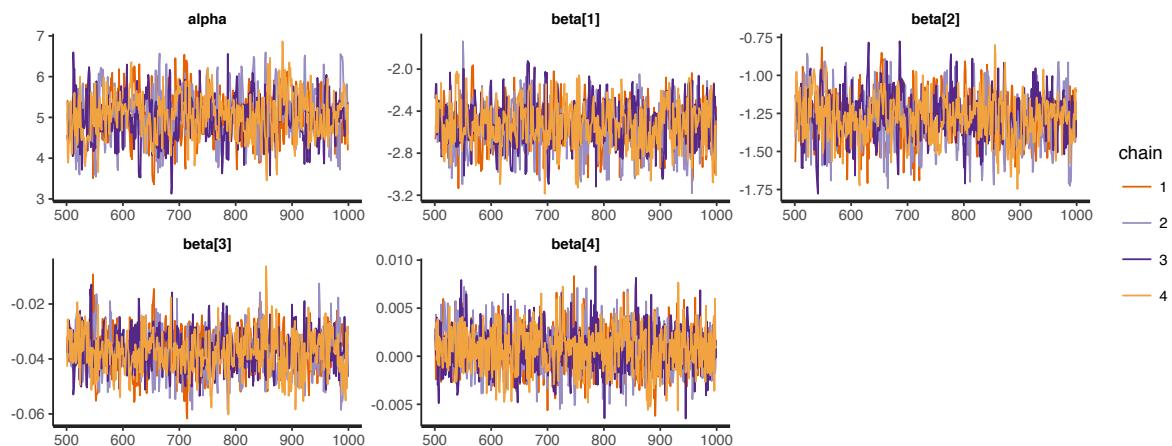
```
logittest_fit

## Inference for Stan model: 831913411f1f6ab48a65b4430b4770fb.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##           mean se_mean    sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
## alpha      5.03   0.02 0.57  3.89  4.64  5.04  5.40  6.17  641  1.00
## beta[1]   -2.54   0.01 0.22 -2.99 -2.68 -2.54 -2.39 -2.12  863  1.00
## beta[2]   -1.28   0.01 0.16 -1.59 -1.38 -1.28 -1.18 -0.97  711  1.00
## beta[3]   -0.04   0.00 0.01 -0.05 -0.04 -0.04 -0.03 -0.02  962  1.00
## beta[4]     0.00   0.00 0.00  0.00  0.00  0.00  0.00  0.01 1377  1.00
## lp__   -326.18   0.06 1.62 -330.03 -327.08 -325.84 -324.98 -324.06  736  1.01
##
## Samples were drawn using NUTS(diag_e) at Mon Apr 24 04:04:37 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

В целом, наши оценки выглядят адекватно.

Проверим ряды на сходимость:

```
stan_trace(logittest_fit)
```



По графикам видно, что цепи сходятся, поэтому будем считать нашу модель приемлемой. Теперь посмотрим на графики распределения:

```
logit_array <- as.array(logittest_fit)
mcmc_areas(logit_array,
            pars = c("alpha",
                     "beta[1]", "beta[2]", "beta[3]", "beta[4]"))
```

На этом графике хорошо видно, что значения коэффициентов при возрасте (`beta[3]`) и тарифе (`beta[4]`) близки к 0.

Схожим образом можно построить и пробит-модель. Для кода Stan нужно ввести некоторые преобразования. Напомню, что в пробит-модели $\varepsilon_i \sim \mathcal{N}(0, 1)$. А функция распределения вероятности имеет вид:

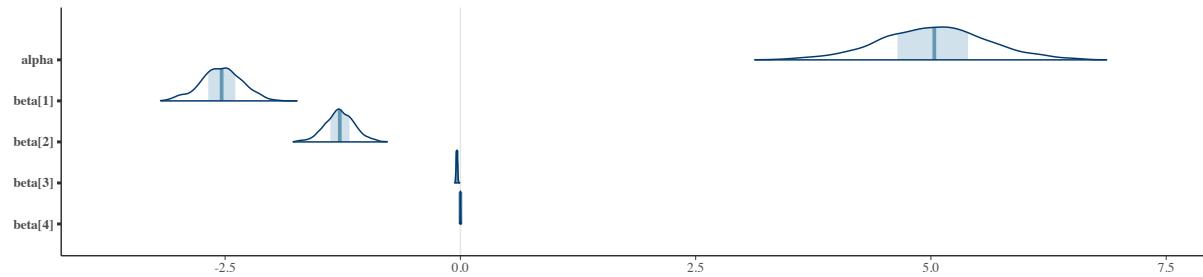
$$\Phi(x) = \int_{-\infty}^x \mathcal{N}(y|0, 1) dy = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-\frac{z^2}{2}} dz$$

В Stan эта функция существует как `Phi`. Также есть более быстрая упрощённая функция `Phi_approx`. В самой модели мы откажемся от векторной формулировки и опишем данные как ряды. В целом ничего не изменится, однако в блоке `model` нужно будет воспользоваться циклом `for`. Запись `for` в Stan такая же, как и в R.

```
model_probit <- "
data {
  int<lower=0> N;
  int<lower=0> k;
  int<lower=0,upper=1> Sex[N];
  int<lower=1,upper=3> Pclass[N];
  real Age[N];
  int<lower=0,upper=1> Surv[N];
}
parameters {
  real alpha;
  real beta[k];
}
model {
  for (n in 1:N)
    Surv[n] ~ bernoulli(Phi(alpha + beta[1] * Sex[n] +
      beta[2] * Pclass[n] + beta[3] * Age[n]));
}"
```

Как всегда, соберем данные и запустим Stan:

```
probit_data <- list(N = nrow(df_titan),
                      k = 3,
```



```

Sex = df_titan$Sex,
Pclass = df_titan$Pclass,
Age = df_titan$Age,
Surv = df_titan$Survived)

probittest_fit <- stan(model_code = model_probit,
                        data = probit_data,
                        iter = 1000,
                        chains = 4)

```

Я отбросил переменную `Fare` чтобы вычисления происходили несколько быстрее. Получится следующий результат:

```

probittest_fit

## Inference for Stan model: ee5b074eb77d2baf547b0c301b675f1a.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##          mean se_mean    sd    2.5%     25%     50%     75%   97.5% n_eff Rhat
## alpha      2.89    0.01  0.27    2.35    2.71    2.89    3.08    3.43   537  1.00
## beta[1]   -1.49    0.00  0.12   -1.73   -1.57   -1.49   -1.41   -1.26   880  1.01
## beta[2]   -0.72    0.00  0.08   -0.88   -0.78   -0.73   -0.67   -0.57   608  1.00
## beta[3]   -0.02    0.00  0.00   -0.03   -0.02   -0.02   -0.02   -0.01   849  1.00
## lp__   -327.05    0.06  1.49 -330.96 -327.74 -326.64 -325.99 -325.27   708  1.00
##
## Samples were drawn using NUTS(diag_e) at Mon Apr 24 21:28:16 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).

```

4.3.4 Временные ряды

В Stan можно описать и модели временных рядов. Данная работа не будет фокусироваться на временных рядах, но я счёл необходимым добавить их в это руководство для полноты картины. Делается это так же, как и в примере с пробит-моделью: в блоке `model` нужно добавить цикл `for` и описать модель с лагом. В качестве примера мы рассмотрим AR модели и прогнозирование временных рядов.

Итак, приступим.

Модель AR(1)

Для начала вспомним, что представляет из себя авторегрессионный процесс первого порядка. Математически он описывается следующим образом:

$$y_t = \beta y_{t-1} + \varepsilon_t, \quad \varepsilon_t \sim iid(0, \sigma^2), \quad t = 1, \dots, n.$$

В Stan эту модель можно описать так:

```

model_AR1 <- "
data {
  int<lower=0> N;
  vector[N] y;
}
parameters {
  real beta;
  real<lower=0> sigma;
}
model {
  for (n in 2:N)
    y[n] ~ normal(beta * y[n-1], sigma);
}"

```

Попрактиковаться я предлагаю на финансовых данных, например, на стоимости акций Google. Нам понадобится пакет `quantmod` и функция `getSymbols`.

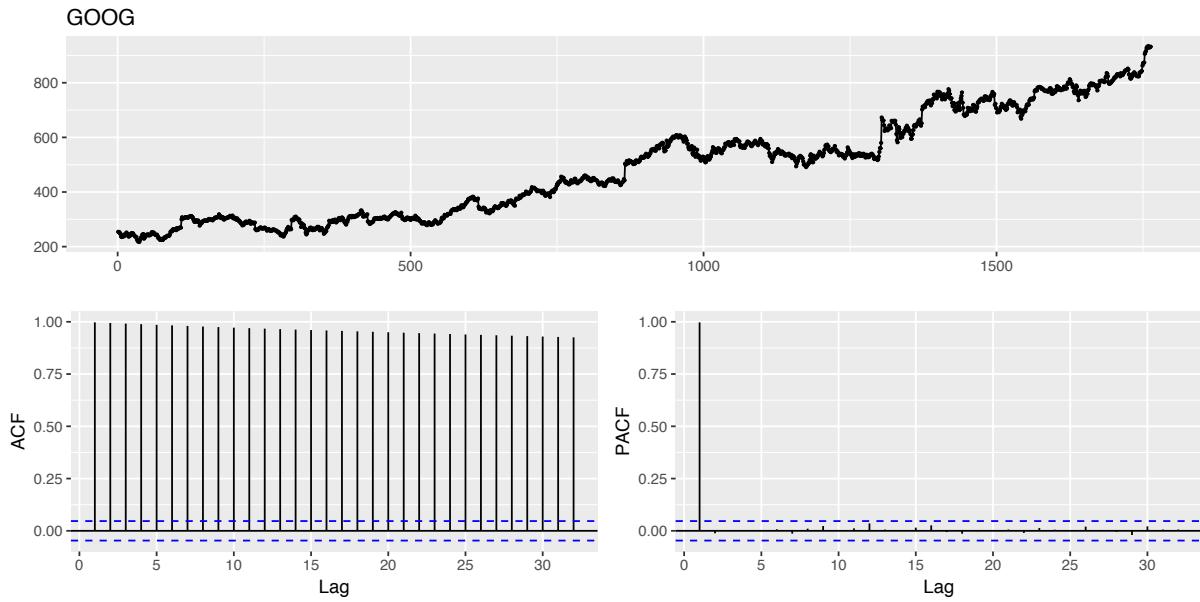
```

Sys.setlocale("LC_TIME", "C") # Перевод времени в англ. формат
getSymbols("GOOG",
  from = "2010-05-13",
  to = "2017-05-13",
  src = "yahoo")
# Нас будут интересовать скорректированные цены закрытия
GOOG <- GOOG[, "GOOG.Adjusted", drop = F]

```

Для начала предлагаю посмотреть, что из себя представляет ряд GOOG:

```
ggtsgdisplay(GOOG)
```



Судя по графикам автокорреляции и частичной автокорреляции, процесс похож на случайное блуждание. Теперь создадим список входных данных и запустим Stan:

```
AR1_data <- list(N = length(GOOG),
                  y = as.vector(GOOG))
AR1_fit <- stan(model_code = model_AR1,
                 data = AR1_data,
                 iter = 1000,
                 chains = 4)
```

Результат:

```
AR1_fit

## Inference for Stan model: 485d5faa6f982c0cdedcd87802b5541f.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##           mean se_mean    sd    2.5%     25%     50%     75%   97.5% n_eff Rhat
## beta      1.0    0.00 0.00    1.00    1.00    1.0    1.00    1.00 1753 1.00
## sigma    18.2    0.02 0.33   17.51   17.99   18.2   18.42   18.85  450 1.01
## lp__ -5130.3    0.04 1.02 -5133.02 -5130.70 -5130.0 -5129.57 -5129.32  550 1.01
##
## Samples were drawn using NUTS(diag_e) at Tue Apr 25 01:14:14 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Модель AR(p)

В случае с авторегрессионным процессом порядка p , изменения в описании небольшие. Количество лагов задаётся тем же путём, что и количество объясняющих переменных в линейных моделях, что мы рассматривали ранее.

```
ARp_model <- "
data {
  int<lower=0> K;
  int<lower=0> N;
  real y[N];
}
parameters {
  real alpha;
  real beta[K];
  real sigma;
}
model {
  for (n in (K+1):N) {
    real mu;
    mu = alpha;
    for (k in 1:K)
      mu = mu + beta[k] * y[n-k];
    y[n] ~ normal(mu, sigma);
  }
}"
```

```
ARp_data <- list(N = length(GOOG),
                   K = 3,
                   y = as.vector(GOOG))
```

```
ARp_fit <- stan(model_code = ARp_model,
                 data = ARp_data,
                 iter = 1000,
                 chains = 4)
```

В данном случае в нашей модели 3 лага и есть константа.

$$y_t = \alpha + \beta_1 y_{t-1} + \beta_2 y_{t-2} + \beta_3 y_{t-3}$$

Получаем следующий результат:

```
ARp_fit
```

```
## Inference for Stan model: 205402774e8262ae1bf04c5bd694ca14.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##           mean se_mean    sd    2.5%    25%    50%    75%   97.5% n_eff Rhat
## alpha     2.66    1.36 2.66   -2.05    0.43    2.89    4.88    7.19     4 1.56
```

```
## beta[1]    1.03   0.00 0.03    0.98    1.01    1.03    1.05    1.08 1485 1.00
## beta[2]   -0.03   0.00 0.04   -0.10   -0.05   -0.03    0.00    0.05 1181 1.00
## beta[3]   -0.01   0.00 0.03   -0.06   -0.02   -0.01    0.01    0.05 1185 1.00
## sigma     18.18   0.01 0.33   17.54   17.96   18.18   18.41   18.84 1469 1.00
## lp__   -5126.62   0.88 2.06 -5131.33 -5127.90 -5126.23 -5125.02 -5123.79      5 1.24
##
## Samples were drawn using NUTS(diag_e) at Tue Apr 25 01:58:29 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

4.4 Многоуровневые модели

Многоуровневые модели — это отдельный класс моделей, в которых параметры могут разниться на разных уровнях. В качестве примера можно привести успеваемость учеников/студентов в разных школах/группах, смертность от некоторого заболевания у пациентов в разных больницах и так далее. Многоуровневые модели также называются иерархическими по двум причинам: во-первых, из-за структуры данных (студенты объединяются в кластеры: классы/школы), а во-вторых, у каждой модели есть своя иерархия — параметры регрессии внутри школы на низшем уровне контролируются гиперпараметрами на высшем уровне¹⁰. Разумно подразумевать, что кластеры не идентичны, но могут быть похожими. Поэтому мы можем использовать знания об одном кластере (или нескольких), чтобы оценить другой. Такой подход увеличивает точность оценивания. Рассмотрим другой пример¹¹: Вы хотите оценить время подачи кофе в нескольких кафе. Вначале Вы ничего не знаете, и делаете (априорное) предположение, что время подачи в среднем 5 минут, а дисперсия мала (допустим, 1). В первом кафе Вам принесли кофе за 4 минуты. Теперь, когда Вы будете заходить во второе кафе, вы будете пользоваться этой информацией, ведь Ваше априорное мнение о времени подачи изменилось: постериорное распределение для первого кафе станет априорным для второго. Повторив этот «эксперимент» много раз, Вы получите некоторое распределение. Если у него большая дисперсия, то его информативность мала, и Вам сложнее оценить время подачи в следующем кафе. В случае с маленькой дисперсией, мы можем более точно угадать время подачи.

Таким образом, многоуровневые модели запоминают свойства разных кластеров, собирая полученную информацию воедино, что улучшает оценки каждого кластера. Среди сильных моментов этих моделей можно выделить:

1. **Улучшенные оценки при повторной выборке.** При взятии нового наблюдения с того-же индивида (места, времени) обычная одноуровневая модель может вызвать недо- или переподгонку (слишком слабая или сильная чувствительность к колебаниям в данных). Часто происходит при большом количестве переменных.
2. **Улучшенные оценки при дисбалансе в выборке.** В случае, когда от одного и того же индивида было получено несколько наблюдений (больше, чем от других), многоуровневые модели самостоятельно справляются с наличием большего количества информации в некоторых кластерах. Более того, они не трансформируют данные (к примеру, усреднение может привести к потере дисперсии).
3. **Дисперсия.** В многоуровневых моделях дисперсия моделируется явно, что может быть полезно, если нас интересует дисперсия среди индивидов или групп.

Вышеперечисленные достоинства достаточно хороши, чтобы считать многоуровневую модель моделью по умолчанию. В большинстве случаев она будет справляться

¹⁰A. Gelman, J. Hill, «Data Analysis Using Regression and Multilevel/Hierarchical Models», Cambridge University press, 2007, p.2

¹¹Richard McElreath, «Statistical Rethinking», 2015, pp.370-371

Школа	y_j	σ_j
A	28	15
B	8	10
C	-3	16
D	7	11
E	-1	9
F	1	11
G	18	10
H	12	18

лучше, чем одноуровневая. И лишь в случае её ненужности, от неё можно отказаться. Освоив иерархические модели, можно даже в какой-то степени закрыть глаза на ошибку в измерении и даже моделировать отсутствующие данные.

Теперь перейдём к примерам¹² в Stan. Поскольку Stan создавался, чтобы работать с иерархическими моделями, его разработчики заранее добавили набор данных Eight Schools¹³. Данные выглядят следующим образом:

А в R:

```
data_8schools <- list(J = 8,
                      y = c(28, 8, -3, 7, -1, 1, 18, 12),
                      sigma = c(15, 10, 16, 11, 9, 11, 10, 18))
```

Модель без объединения информации

Каждую школу будем оценивать отдельно. Код модели:

```
m8schools_nopool <- "
data {
  int<lower=0> J;
  real y[J];
  real<lower=0> sigma[J];
}
parameters {
  real theta[J];
}
model {
  y ~ normal(theta, sigma);
}"
```

Запустим нашу модель:

```
nopool_fit <- stan(model_code = m8schools_nopool, data = data_8schools,
                     iter = 1000, chains = 4)
```

¹²Примеры взяты отсюда: <http://astrostatistics.psu.edu/su14/lectures/Daniel-Lee-Stan-2.pdf>

¹³A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, Aki Vehtari, D. B. Rubin, «Bayesian Data Analysis», CRC Press, 2014, pp.119-124.

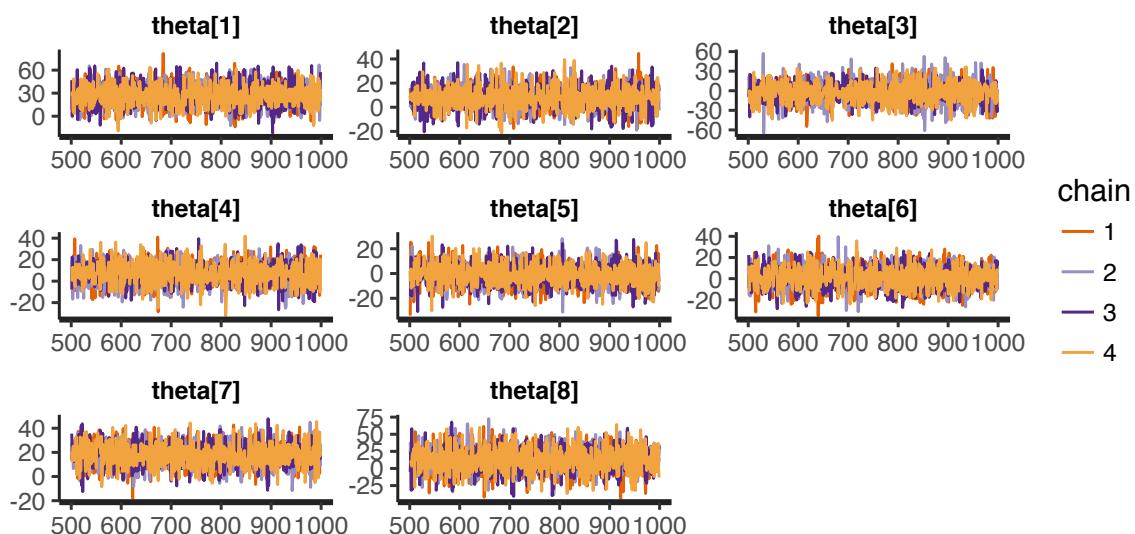
И получим следующий результат:

```
nopool_fit

## Inference for Stan model: 9ea4016a2608ca9f46899b6badf8b2ff.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##           mean se_mean    sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
## theta[1] 27.28     0.34 15.07 -1.19 16.88 26.74 37.57 57.51  2000    1
## theta[2]  8.24     0.22  9.62 -11.23  1.80  8.26 14.89 27.26  2000    1
## theta[3] -3.34     0.36 15.92 -34.75 -13.79 -3.50  6.87 28.14  2000    1
## theta[4]  6.72     0.24 10.65 -13.61 -0.77  6.72 14.17 27.47  2000    1
## theta[5] -0.96     0.21  9.29 -18.39 -7.61 -0.92  5.47 16.52  2000    1
## theta[6]  1.07     0.24 10.82 -19.99 -6.43  1.06  8.46 22.14  2000    1
## theta[7] 17.96     0.22  9.84 -1.61 11.25 18.09 24.73 36.55  2000    1
## theta[8] 12.45     0.41 18.35 -23.61 -0.23 12.39 25.13 48.72  2000    1
## lp__    -3.95     0.06  1.91 -8.49 -4.92 -3.62 -2.58 -1.12 1204    1
##
## Samples were drawn using NUTS(diag_e) at Fri Jun 30 12:40:13 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Полученные оценки θ в целом не отличаются от первоначальных y_j . Также проверим сходимость цепей:

```
stan_trace(nopool_fit)
```



Модель с полным объединением информации

Теперь вся информация о школах будет объединена.

```
m8schools_fullpool <- "
data {
  int<lower=0> J;
  real y[J];
  real<lower=0> sigma[J];
}
parameters {
  real theta; //теперь theta общая
}
model {
  y ~ normal(theta, sigma);
}"

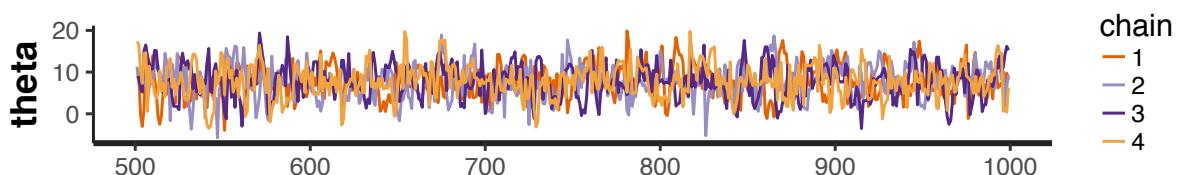
fullpool_fit <- stan(model_code = m8schools_fullpool, data = data_8schools,
                      iter = 1000, chains = 4)
```

```
fullpool_fit

## Inference for Stan model: fec9c74586f9a4e42f683f323df22c7c.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##           mean se_mean    sd  2.5%   25%   50%   75% 97.5% n_eff Rhat
## theta    7.76     0.15 4.01  0.03  5.11  7.71 10.52 15.47    743    1
## lp__   -2.84     0.02 0.67 -4.70 -3.00 -2.58 -2.40 -2.35   1042    1
##
## Samples were drawn using NUTS(diag_e) at Fri Jun 30 21:13:07 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Мы получили средний эффект по всем школам. Сходимость цепей:

```
stan_trace(fullpool_fit)
```



Модель с неполным объединением информации

Добавим некоторое τ , как меру ковариации между школами. Также оценим гиперпараметр μ .

```
m8schools_partpool <- "
data {
  int<lower=0> J;
  real y[J];
  real<lower=0> sigma[J];
  real<lower=0> tau;
}
parameters {
  real theta[J];
  real mu;
}
model {
  theta ~ normal(mu, tau);
  y ~ normal(theta, sigma);
}"
```

```
data_8schools_t25 <- list(J = 8,
                           y = c(28, 8, -3, 7, -1, 1, 18, 12),
                           sigma = c(15, 10, 16, 11, 9, 11, 10, 18),
                           tau = 25)
```

```
partpool_fit <- stan(model_code = m8schools_partpool, data = data_8schools_t25,
                      iter = 1000, chains = 4)
```

```
partpool_fit

## Inference for Stan model: 6f0c5cb6d8d98e0eef3c499566fabb54.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##           mean se_mean    sd   2.5%   25%   50%   75% 97.5% n_eff Rhat
## theta[1] 22.99    0.28 12.64 -1.37 14.61 22.81 31.69 48.04  2000 1.00
## theta[2]  7.86    0.21  9.45 -11.21  1.61  7.89 13.95 26.52  2000 1.00
## theta[3]  0.44    0.31 13.86 -27.40 -8.44  0.31  9.99 26.91  2000 1.00
## theta[4]  7.27    0.23 10.17 -12.94  0.57  7.12 14.04 27.45  2000 1.00
## theta[5]  0.05    0.19  8.28 -15.55 -5.61  0.29  5.84 15.84  2000 1.00
## theta[6]  2.19    0.23 10.25 -17.68 -4.56  2.03  9.33 21.49  2000 1.00
## theta[7] 16.59    0.21  9.48 -1.31  9.58 16.66 23.21 34.91  2000 1.00
## theta[8] 10.57    0.33 14.86 -18.40  0.39 10.22 20.86 39.62  2000 1.00
## mu       8.19    0.22  9.95 -10.68  1.49  8.23 14.91 27.29  2000 1.00
## lp__    -4.95    0.06  1.99 -9.85 -6.12 -4.70 -3.52 -1.82     948 1.01
```

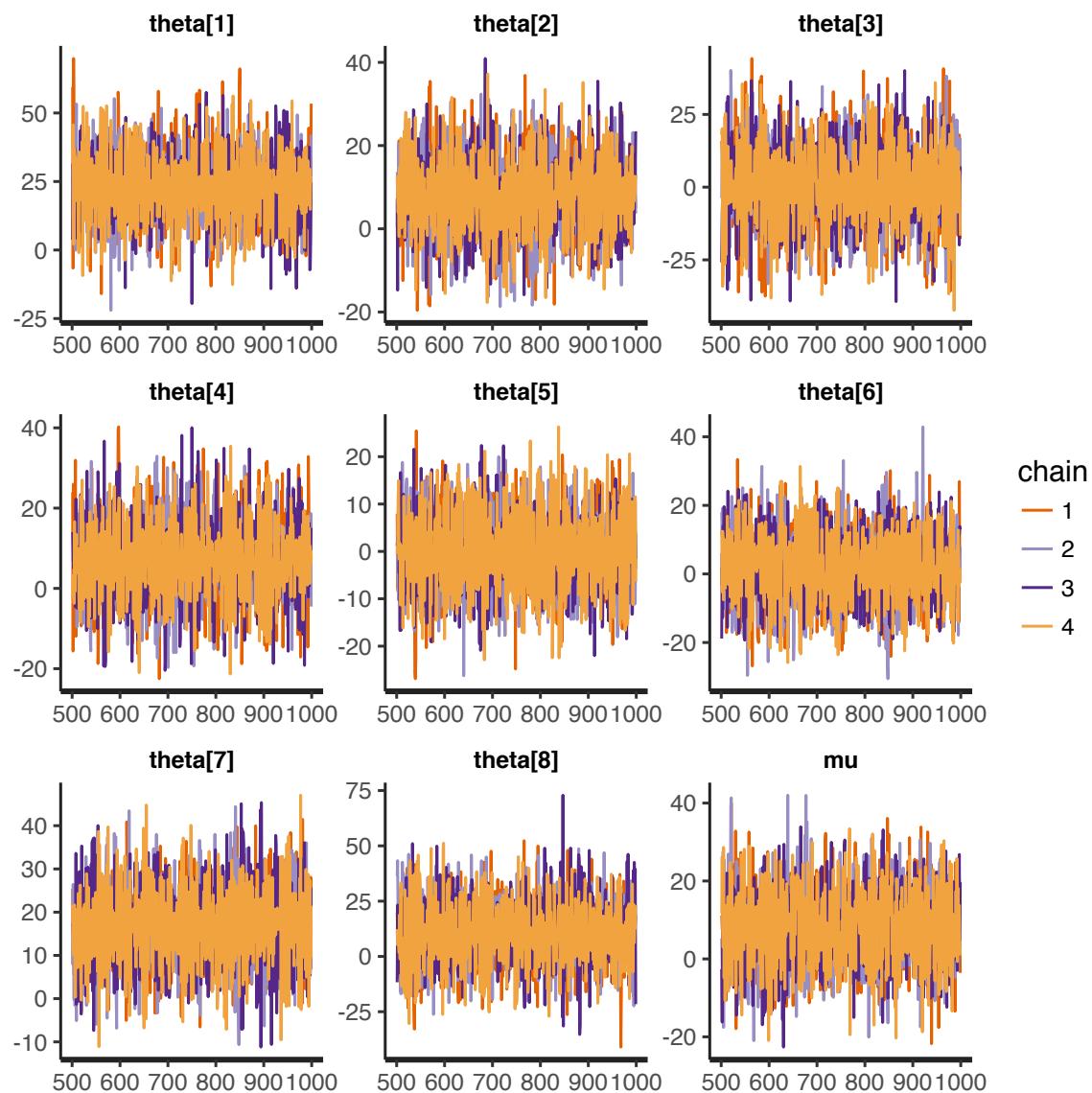
```

## 
## Samples were drawn using NUTS(diag_e) at Fri Jun 30 13:12:25 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).

```

Мы получили оценку гиперпараметра μ (высокий уровень), который влияет на значения θ , индивидуальные для каждого кластера (низший уровень). Сходимость цепей:

```
stan_trace(partpool_fit)
```



Иерархическая модель

Теперь мы оцениваем и гиперпараметр, и ковариацию.

```
m8schools_hier <- "
data {
  int<lower=0> J;
  real y[J];
  real<lower=0> sigma[J];
}
parameters {
  real theta[J];
  real mu;
  real<lower=0> tau;
}
model {
  theta ~ normal(mu, tau);
  y ~ normal(theta, sigma);
}"
```

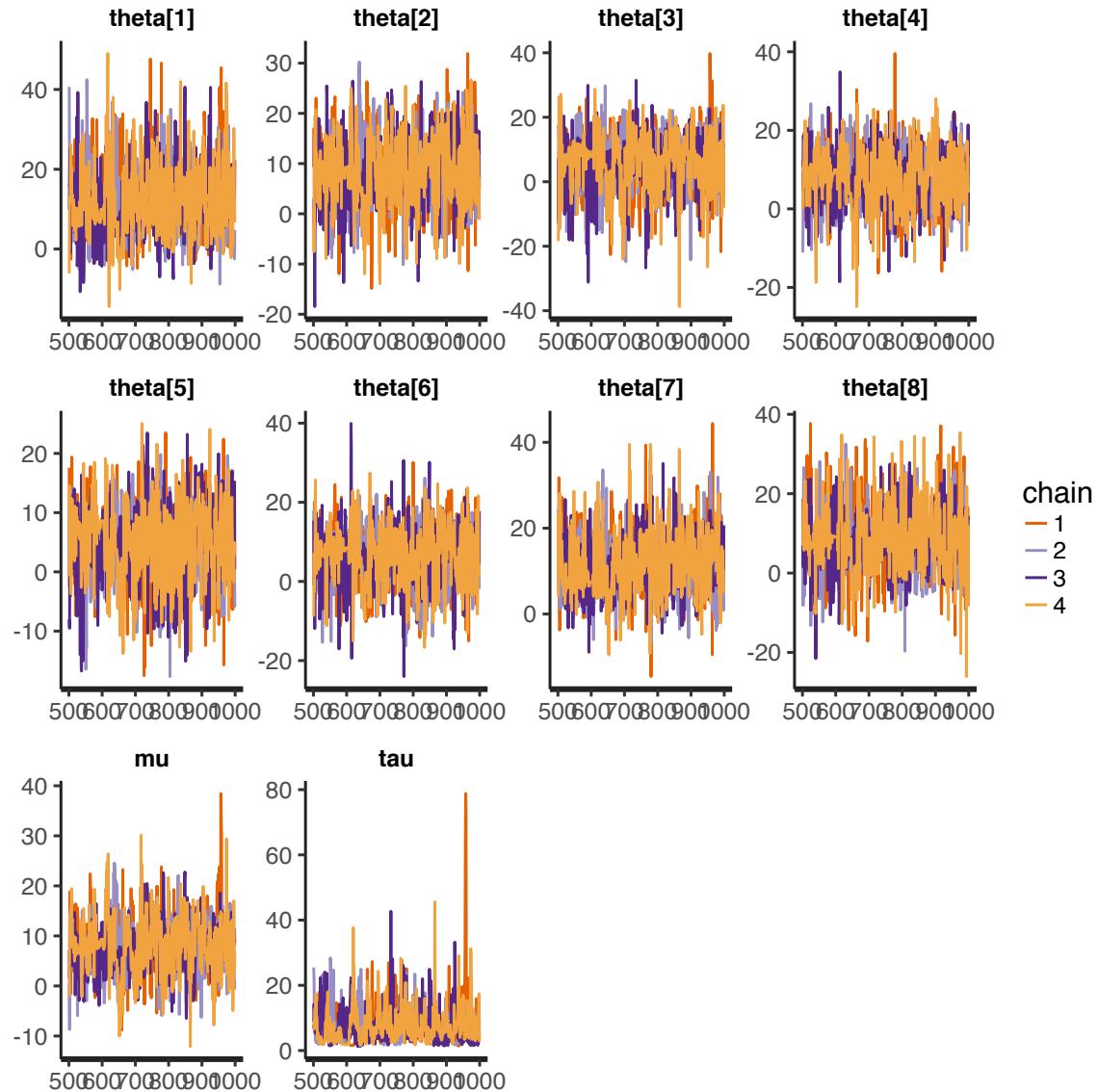
```
hier_fit <- stan(model_code = m8schools_hier, data = data_8schools,
                  iter = 1000, chains = 4)
```

```
hier_fit

## Inference for Stan model: e0480cfab4762f9a1d3354f4dca4778b.
## 4 chains, each with iter=1000; warmup=500; thin=1;
## post-warmup draws per chain=500, total post-warmup draws=2000.
##
##           mean   se_mean     sd    2.5%    25%    50%    75% 97.5% n_eff Rhat
## theta[1]  11.56    0.35  8.26  -2.26   6.28  10.58  15.91 30.99  545  1.00
## theta[2]   8.19    0.27  6.61  -4.84   3.92   8.18  12.31 21.50  594  1.00
## theta[3]   5.92    0.31  8.23 -13.01   1.58   6.55  11.12 20.62  710  1.00
## theta[4]   7.71    0.28  6.95  -6.61   3.37   7.89  12.28 21.01  610  1.00
## theta[5]   5.03    0.27  6.36  -8.46   0.85   5.39   9.32 16.53  546  1.00
## theta[6]   6.15    0.28  7.28  -9.22   1.91   6.55  10.83 19.77  679  1.00
## theta[7]  11.06    0.30  6.93  -1.71   6.52  10.69  15.14 25.84  535  1.00
## theta[8]   8.34    0.29  7.80  -7.06   3.54   8.00  12.93 25.23  734  1.00
## mu        7.93    0.28  5.42  -2.56   4.14   7.87  11.48 19.14  387  1.00
## tau       7.25    0.32  5.68   1.66   3.51   5.87   9.47 20.87  315  1.00
## lp__    -18.42    0.34  4.83 -27.85 -21.71 -18.53 -15.13 -8.85   199  1.01
##
## Samples were drawn using NUTS(diag_e) at Fri Jun 30 13:48:39 2017.
## For each parameter, n_eff is a crude measure of effective sample size,
## and Rhat is the potential scale reduction factor on split chains (at
## convergence, Rhat=1).
```

Мы получили оценки гиперпараметра и ковариации. Сходимость цепей:

```
stan_trace(hier_fit)
```



4.5 Способы проверки цепей на сходимость

Теперь перейдём к проверке сходимости цепей к апостериорному распределению. Замечу, что это очень важный пункт в использовании Байесовского подхода, так как определяет качество модели и сигнализирует о возможном наличии ошибок. Есть несколько способов проверить цепь на сходимость. Разберём некоторые из них на примере нашей простой модели.

Среднее значение цепи

Очевидно, нам нужно оценить среднее для каждой цепи.

```
# Среднее для каждой цепи по отдельности
apply(lm_array[, , 2], 2, mean)

## chain:1 chain:2 chain:3 chain:4
##   3.957   3.917   3.964   3.898

# Среднее по всем цепям
mean(lm_array[, , 2]) #среднее по alpha и beta

## [1] 3.934
```

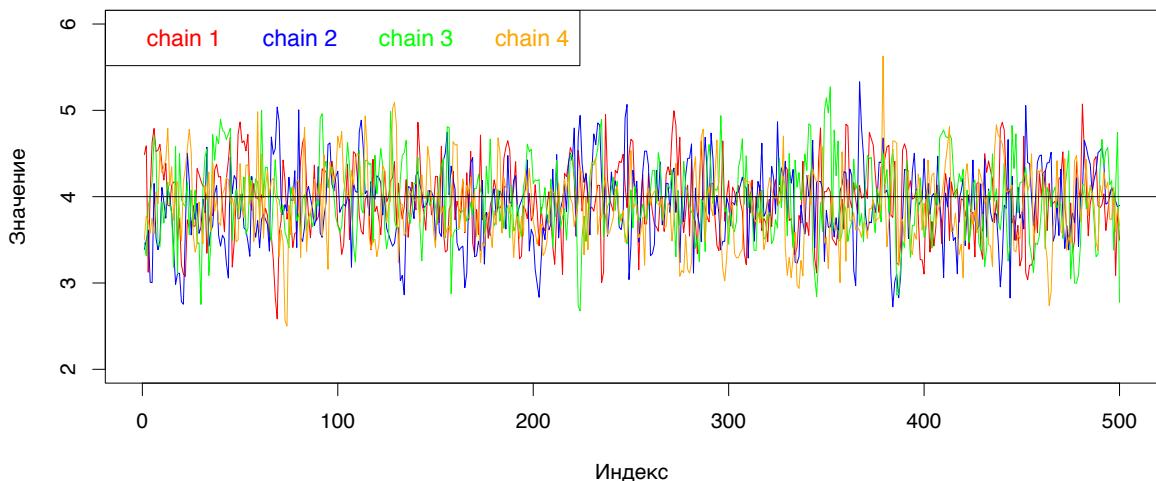
Как мы видим, значения не сильно отличаются друг от друга, так что можно считать, что скорее всего цепи сошлись. Чтобы судить об этом точнее, а главное, без вычислений, мы можем попробовать отобразить цепи графически.

Графические способы

Для начала сделаем это самым простым и банальным способом: с помощью функции `plot()`, предварительно поместив значения β в объект `betas` типа `dataframe`:

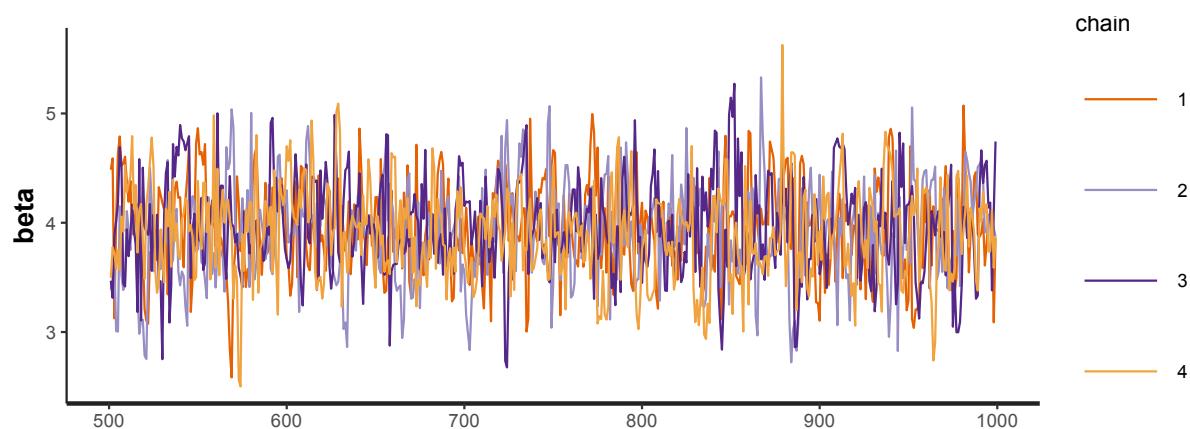
```
betas <- as.data.frame(lm_array[, , 2])
plot(range(1:nrow(betas)), range(2:6),
     type = "n",
     xlab = "Индекс",
     ylab = "Значение",
     main = "Анализ сходимости рядов для beta")
lines(betas$`chain:1`, col = "red")
lines(betas$`chain:2`, col = "blue")
lines(betas$`chain:3`, col = "green")
lines(betas$`chain:4`, col = "orange")
abline(a = 4, b = 0)
labs <- c("chain 1", "chain 2", "chain 3", "chain 4")
legend("topleft", legend = labs, horiz = TRUE,
       bg = "transparent", cex = 1.1,
       text.col = c("red", "blue", "green", "orange"))
```

Анализ сходимости рядов для beta



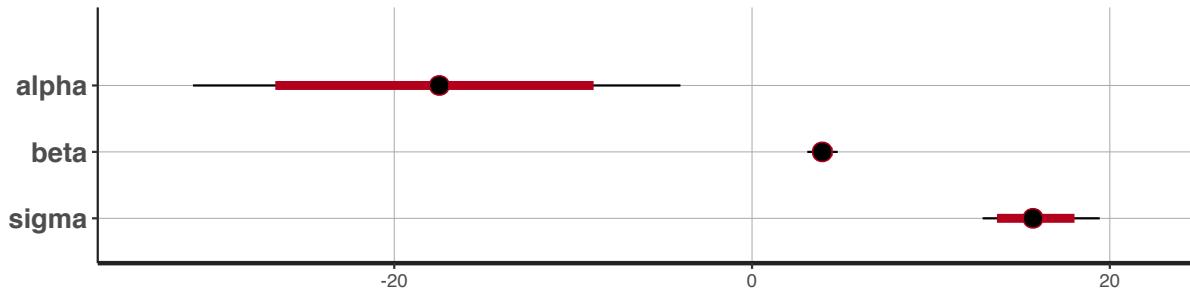
Мы получили нечто в жанре абстракционизма-минимализма. В целом, график похож на сходящиеся ряды, но можно сделать намного проще. В пакет `rstan` уже включена функция `traceplot()` для вызова которой не требуется конвертация в `dataframe`, а также можно вместо индекса переменной указать необходимый параметр, что, несомненно, интуитивно понятнее:

```
traceplot(lmtest_fit, pars = "beta")
```



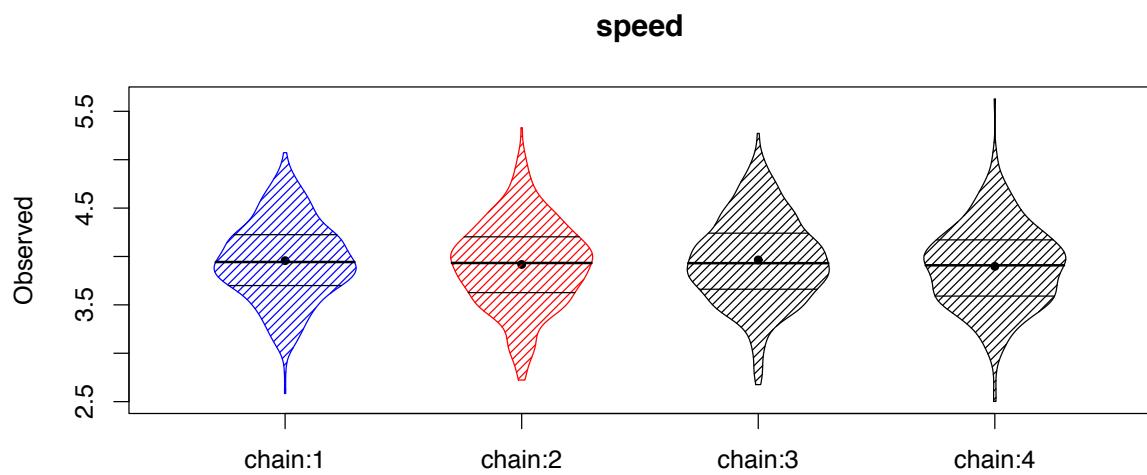
Также мы можем визуализировать интервалы для параметров, введя:

```
plot(lmtest_fit)  
## ci_level: 0.8 (80% intervals)  
## outer_level: 0.95 (95% intervals)
```



Ещё, более наглядным способом, на мой взгляд, будет построить диаграммы в виде скрипки. Сделать это можно стандартной командой `violinBy()` с объектом `array`.

```
#График для beta
violinBy(lm_array[, , 2], main = "speed")
```



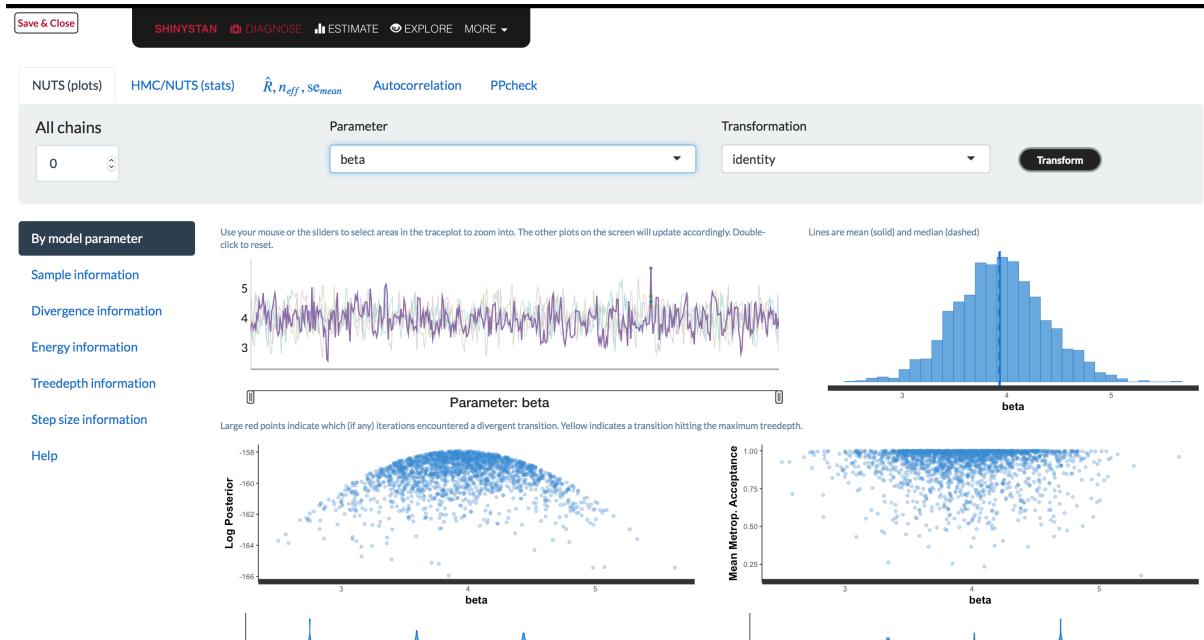
Сходимость цепей можно определить по форме диаграмм. Если они похожи, то закон распределения в каждой цепи одинаковый, и, стало быть, цепи соплись.
Также, для визуального анализа я рекомендую воспользоваться пакетом `shinystan`. С помощью команды `launch_shinystan()`:

```
launch_shinystan(lmtest_fit)
```

откроется окно в браузере (подключение к Интернету не требуется, вычисления происходят локально) с главной страницей `shinytstan`. Перейдя по ссылкам, можно в интерактивном режиме исследовать модель.

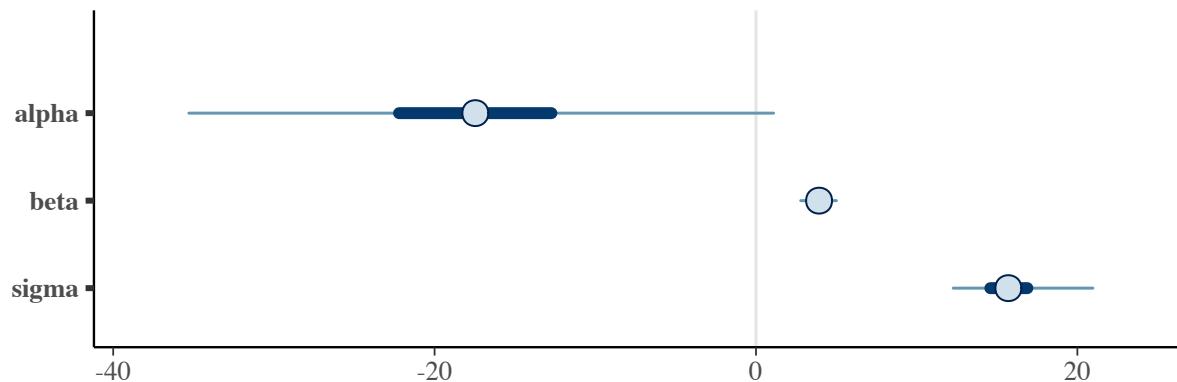


DIAGNOSE ESTIMATE EXPLORE MORE

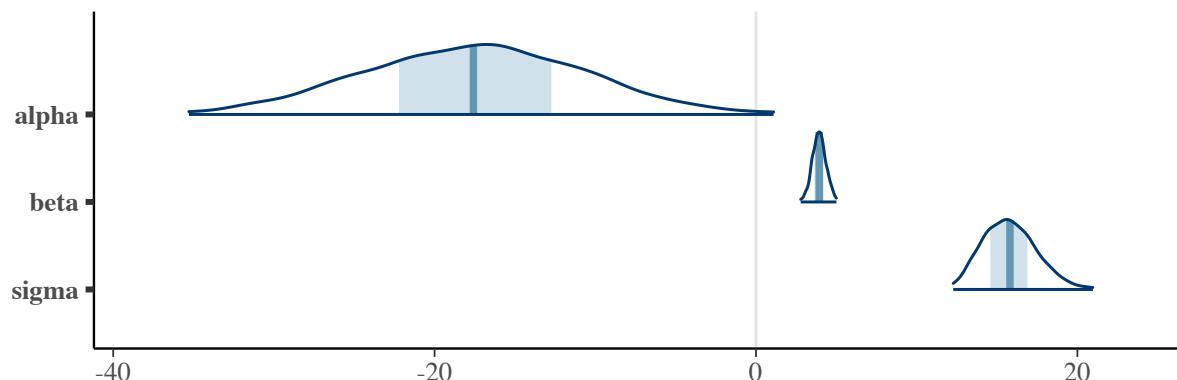


Также для визуализации сходимости и прочих других данных, полученных в процессе анализа, рекомендую использовать пакет `bayesplot`¹⁴. Пакет содержит функции для уже продемонстрированных графиков, а также некоторые собственные. Покажу несколько примеров:

```
#Уже знакомый нам график интервалов,
#можно самому указать границы вероятности:
mcmc_intervals(lm_array, pars = c("alpha", "beta", "sigma"),
                prob = 0.5, prob_outer = 0.99)
```

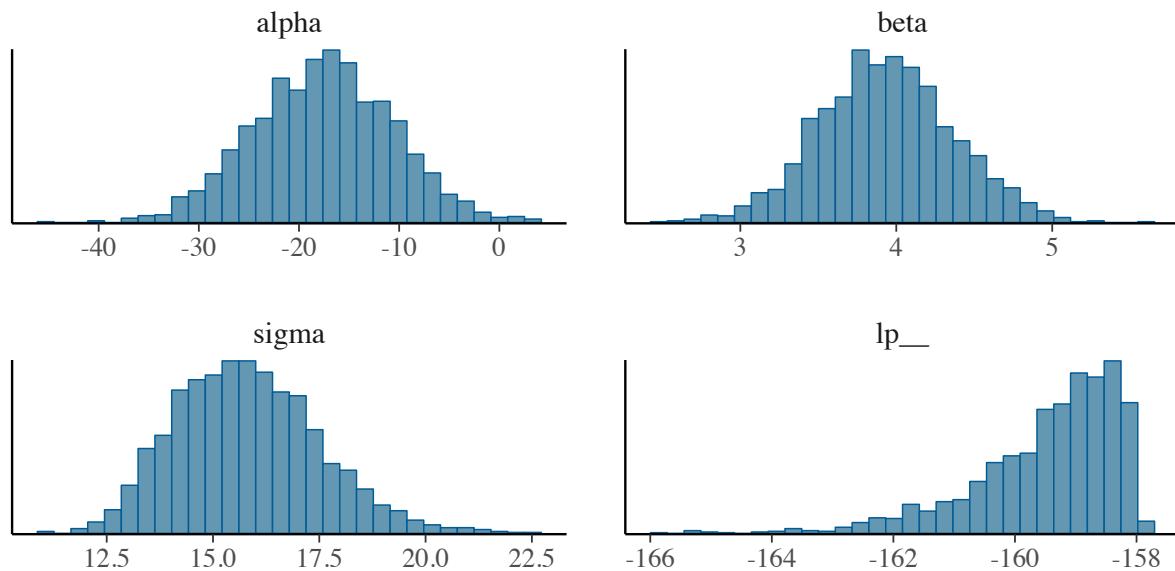


```
#График плотностей, можно самому указать границы вероятности,
#оценить можно среднее или медиану:
mcmc_areas(lm_array,
            pars = c("alpha", "beta", "sigma"),
            prob = 0.5,
            prob_outer = 0.99,
            point_est = "mean")
```

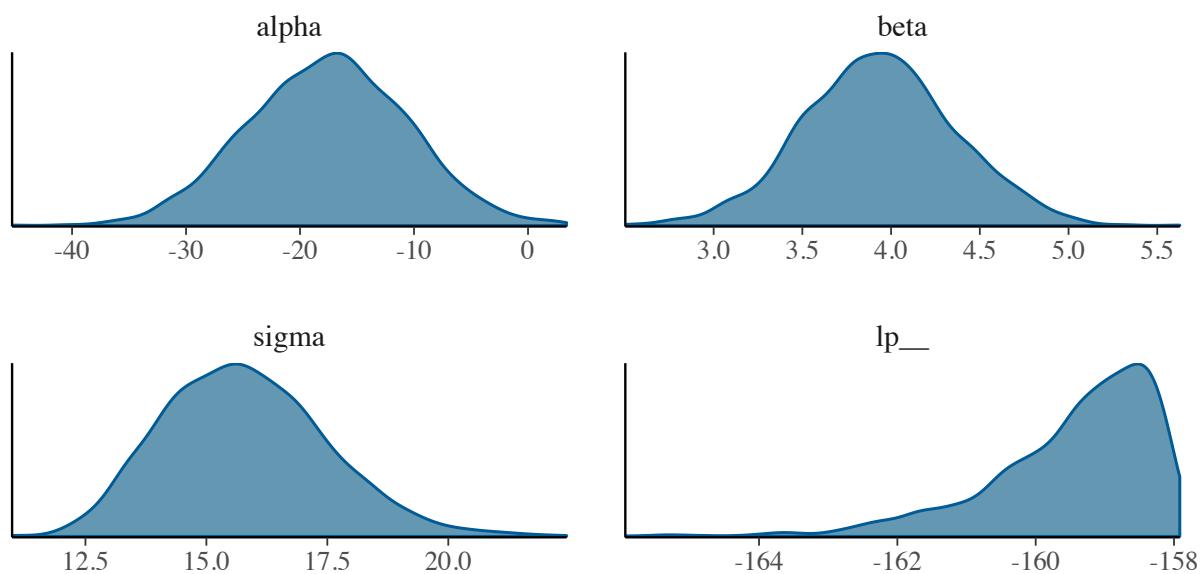


¹⁴Примеры можно найти здесь: <https://cran.r-project.org/web/packages/bayesplot/vignettes/MCMC.html>

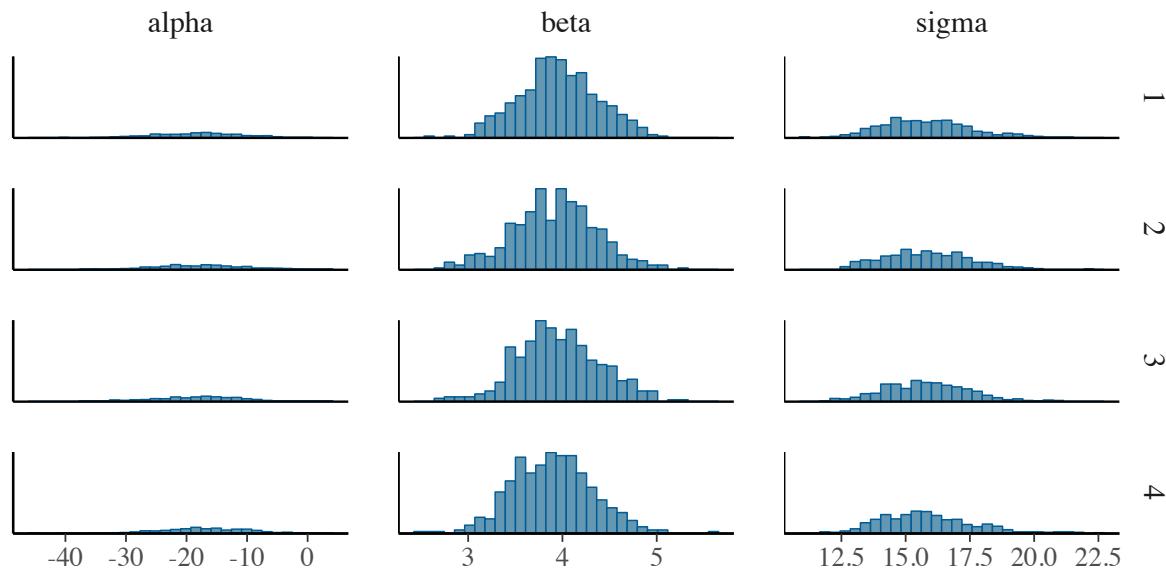
```
#Гистограммы:  
mcmc_hist(lm_array)
```



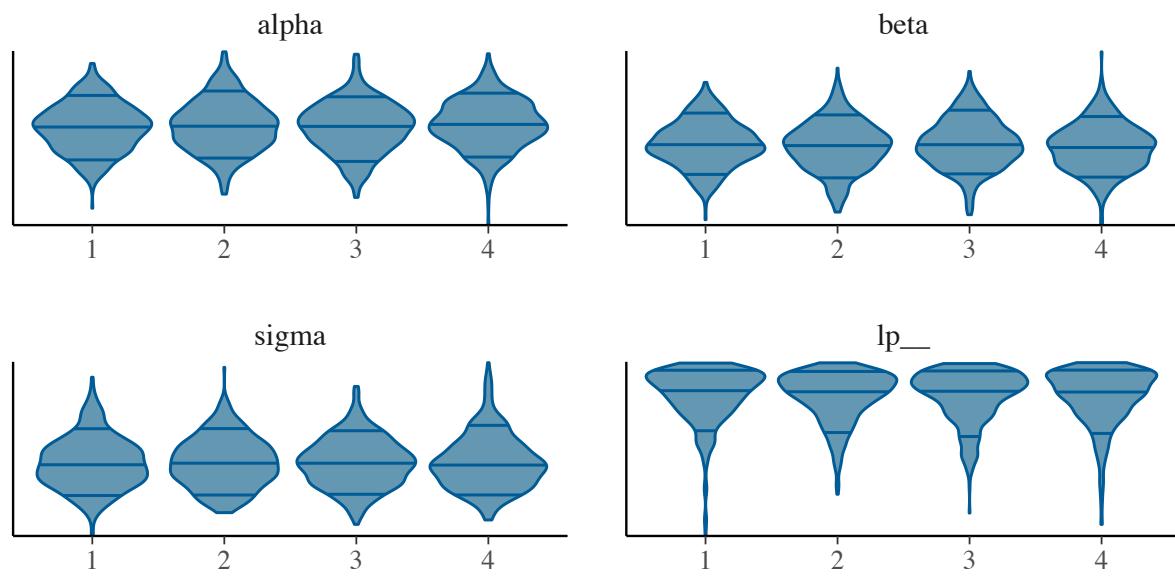
```
#Плотность:  
mcmc_dens(lm_array)
```



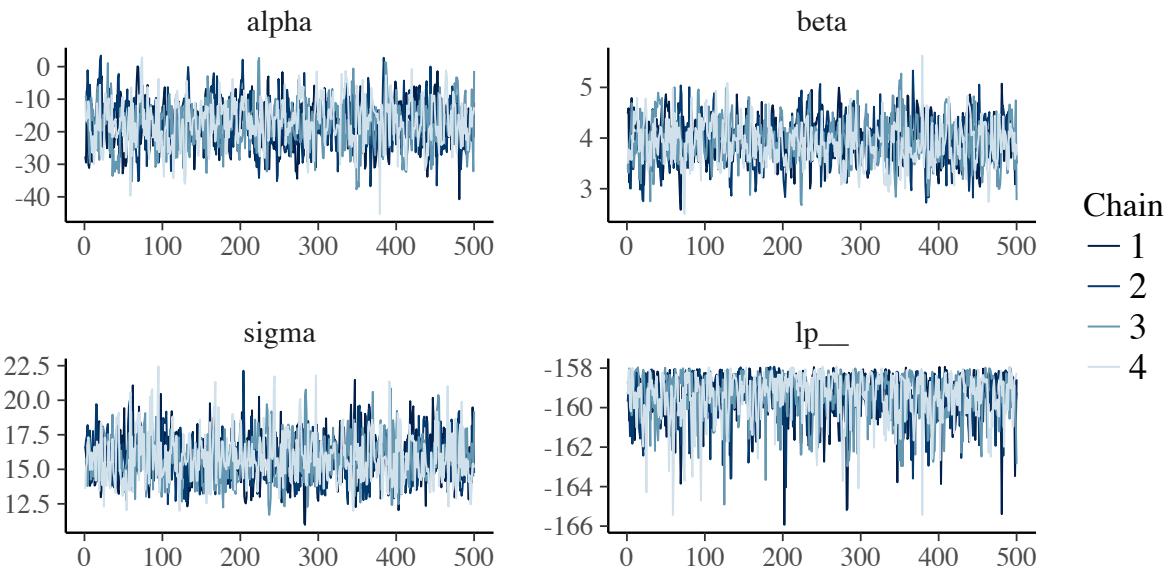
```
# Гистограммы по цепям:  
mcmc_hist_by_chain(lm_array, pars = c("alpha", "beta", "sigma"))
```



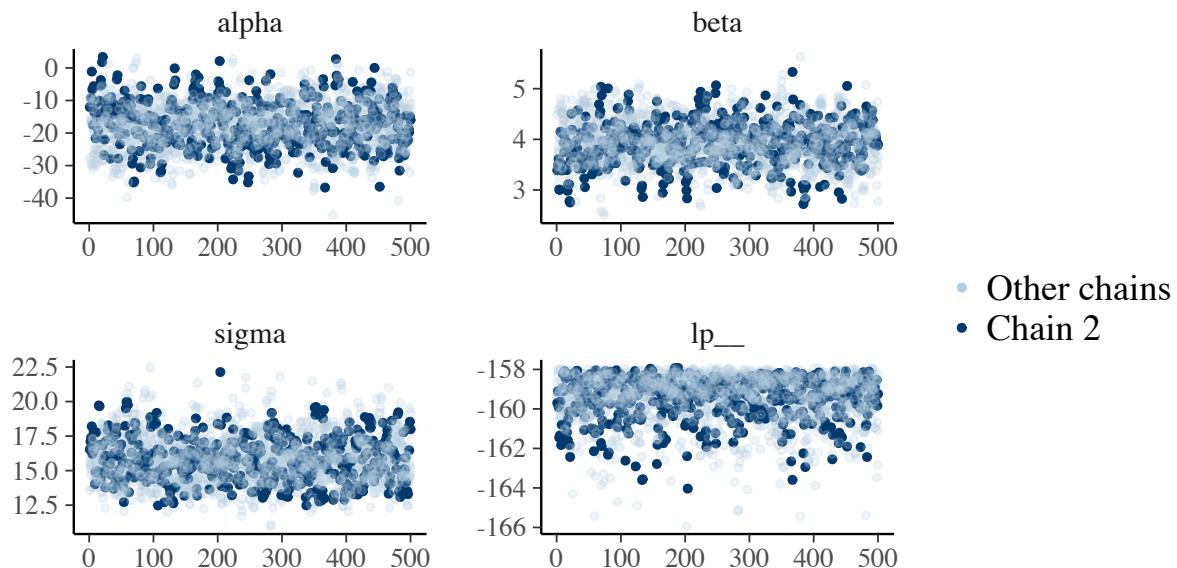
```
#"Скрипкограммы" :)  
mcmc_violin(lm_array)
```



```
#Графики следов цепей:  
mcmc_trace(lm_array)
```



```
#Значения цепей по итерации с выделением отдельной цепи:  
mcmc_trace_highlight(lm_array, highlight = 2)
```



4.6 Прогнозирование в STAN

4.6.1 Блок generated quantities

Хорошая модель должна быть хороша не только в объяснении зависимости одной переменной от других, но и в прогнозировании. В STAN существует отдельный блок, **generated quantities**, в котором с помощью алгоритма можно сгенерировать прогнозируемые значения, согласно оценённой модели.

Рассмотрим пример с моделью AR(3):

```
AR3_w.pred <- "
data {
  int<lower=0> K;
  int<lower=0> N;
  real y[N];
  int<lower=0> T_new;
}
parameters {
  real alpha;
  real beta[K];
  real sigma;
}
model {
  for (n in (K+1):N) {
    real mu;
    mu = alpha;
    for (k in 1:K)
      mu = mu + beta[k] * y[n-k];
    y[n] ~ normal(mu, sigma);
  }
}
generated quantities {
  vector[T_new] y_tilde;
  real mu;
  mu = alpha;
  for (k in 1:K)
    mu = mu + beta[k] * y[1764-k];
  y_tilde[1] = normal_rng(mu, sigma);
  mu = alpha + beta[2] * y[1765-2] +
    beta[3] * y[1765-3] +
    beta[1] * y_tilde[1];
  y_tilde[2] = normal_rng(mu, sigma);
  mu = alpha + beta[3] * y[1766-3] +
    beta[1] * y_tilde[1] +
    beta[2] * y_tilde[2];
  y_tilde[3] = normal_rng(mu, sigma);
  for (t in 4:T_new) {
    mu = alpha;
    for (k in 1:K)
      mu = mu + beta[k] * y_tilde[t-k];
  }
}
```

```

    y_tilde[t] = normal_rng(mu, sigma);
}"}
```

В блоке `generated quantities` необходимо написать алгоритм, с помощью которого будут генерироваться прогнозируемые значения. Идея в данном случае несложная: необходимо создать вектор с новыми значениями \tilde{y} . Важно учесть, что \tilde{y}_1 , \tilde{y}_2 и \tilde{y}_3 зависят от наблюдаемых значений, поэтому их лучше определить заранее. Запустим модель:

```

AR3_data <- list(N = length(GOOG),
                  K = 3,
                  y = as.vector(GOOG),
                  T_new = 365)

AR3_fit_pred <- stan(model_code = AR3_w.pred,
                      data = AR3_data,
                      iter = 3000,
                      chains = 4)
```

В таблице, которую мы получим в результате, помимо коэффициентов будут указаны и прогнозируемые значения (все 365 дней). Для компактности я решил не включать её вывод. Извлечь прогнозы из таблицы их можно с помощью команды `extract()`, содержащейся в пакете `rstan`. Далее, можно создать вектор из медиан симуляций по каждому дню:

```

prediction <- rstan::extract(AR3_fit_pred, "y_tilde")

x <- vector()
for (i in 1:365) {
  x[i] <- median(prediction$y_tilde[, i])
}
```

и создать `dataframe` из прогнозируемых и исторических значений.

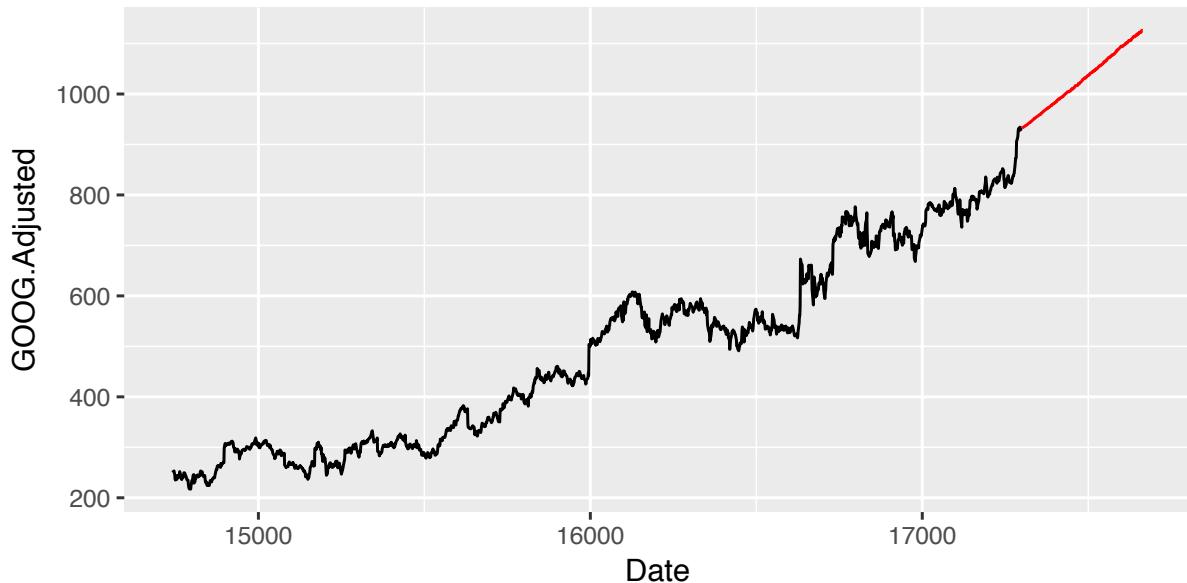
```

dat <- as.data.frame(cbind(
  as.Date(rownames(as.data.frame(GOOG))),
  as.numeric(GOOG)))
colnames(dat) <- c("Date", "GOOG.Adjusted")

pred <- as.data.frame(cbind(
  as.Date(seq(from = as.Date("2017-05-14"),
             to = as.Date("2018-05-13"),
             by = "day")),
  x))
colnames(pred) <- c("Date", "GOOG.Adjusted")
```

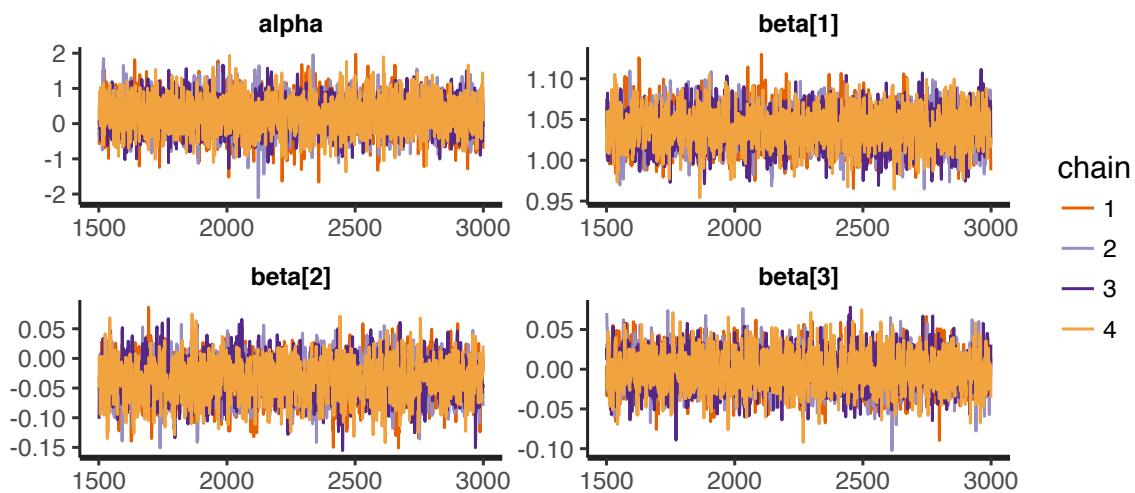
Теперь мы можем отобразить наш прогноз на графике:

```
ggplot() +
  geom_line(data = dat, aes(x = Date,
                             y = GOOG.Adjusted)) +
  geom_line(data = pred, aes(x = Date,
                             y = GOOG.Adjusted),
            col = "red")
```

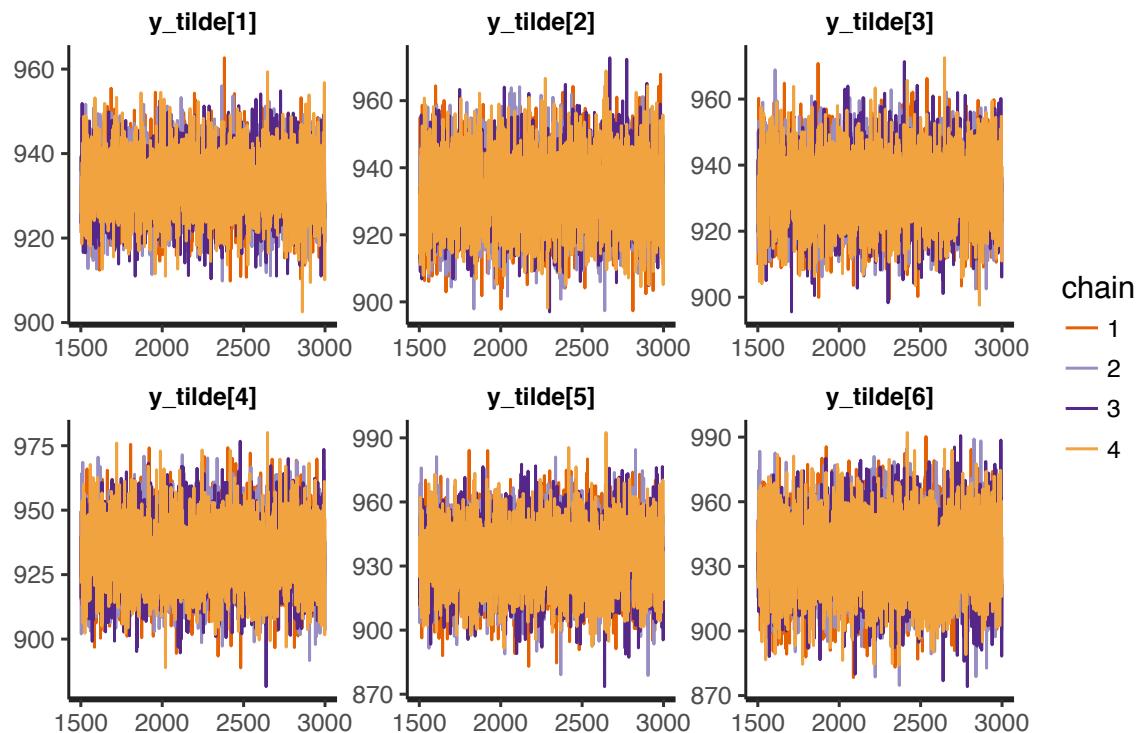


Также стоит проверить ряды на сходимость.

```
stan_trace(AR3_fit_pred, pars = c("alpha", "beta"))
```

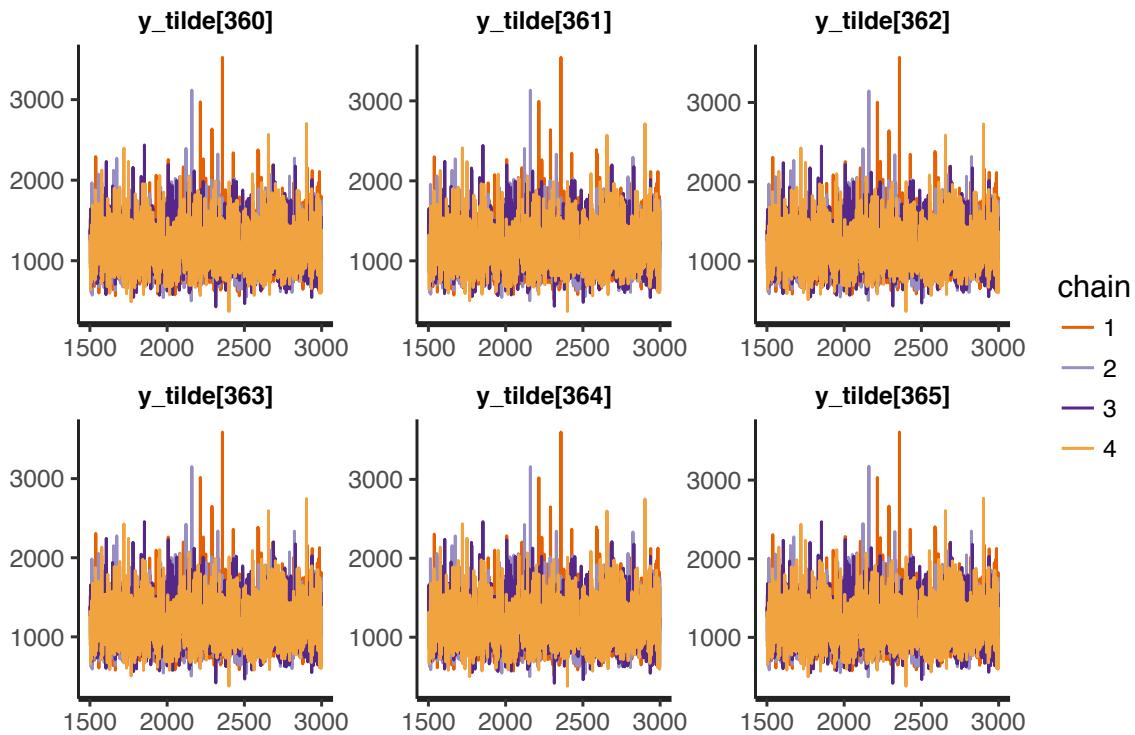


```
stan_trace(AR3_fit_pred, pars = c("y_tilde[1]",
                                 "y_tilde[2]",
                                 "y_tilde[3]",
                                 "y_tilde[4]",
                                 "y_tilde[5]",
                                 "y_tilde[6]"))
```



Для коэффициентов модели и первых 6 прогнозируемых значений всё в порядке. Однако было бы неплохо посмотреть на поведение цепей для последних прогнозов.

```
stan_trace(AR3_fit_pred, pars = c("y_tilde[360]",
                                 "y_tilde[361]",
                                 "y_tilde[362]",
                                 "y_tilde[363]",
                                 "y_tilde[364]",
                                 "y_tilde[365]"))
```



4.6.2 Prophet

Prophet¹⁵ — проект команды Facebook's Core Data Science. Предназначен специально для прогнозов по временным рядам. Модели описываются в Stan (<https://goo.gl/QsKvd2>), пользователю же остаётся воспользоваться несколькими простыми командами. Предлагаю разобрать пример с акциями Facebook.

Примечание: таблица данных должна содержать два столбца: `ds` (дата) и `y` (наблюдения). Важно, чтобы столбец наблюдений имели имя `y`.

```
getSymbols("FB",
           from = "2010-05-13",
           to = "2017-05-13",
           src = "yahoo")
FB <- FB[, "FB.Adjusted", drop = F]
```

Чтобы преобразовать `xts`-объект в `dataframe` с отдельными столбцами для даты и наблюдений, воспользуемся следующими командами:

```
df <- data.frame(ds = index(FB), coredata(FB))
names(df)[2] <- "y"
```

Теперь с помощью команд `prophet` и `predict` мы можем построить прогноз:

¹⁵Ссылка: <https://facebookincubator.github.io/prophet/>

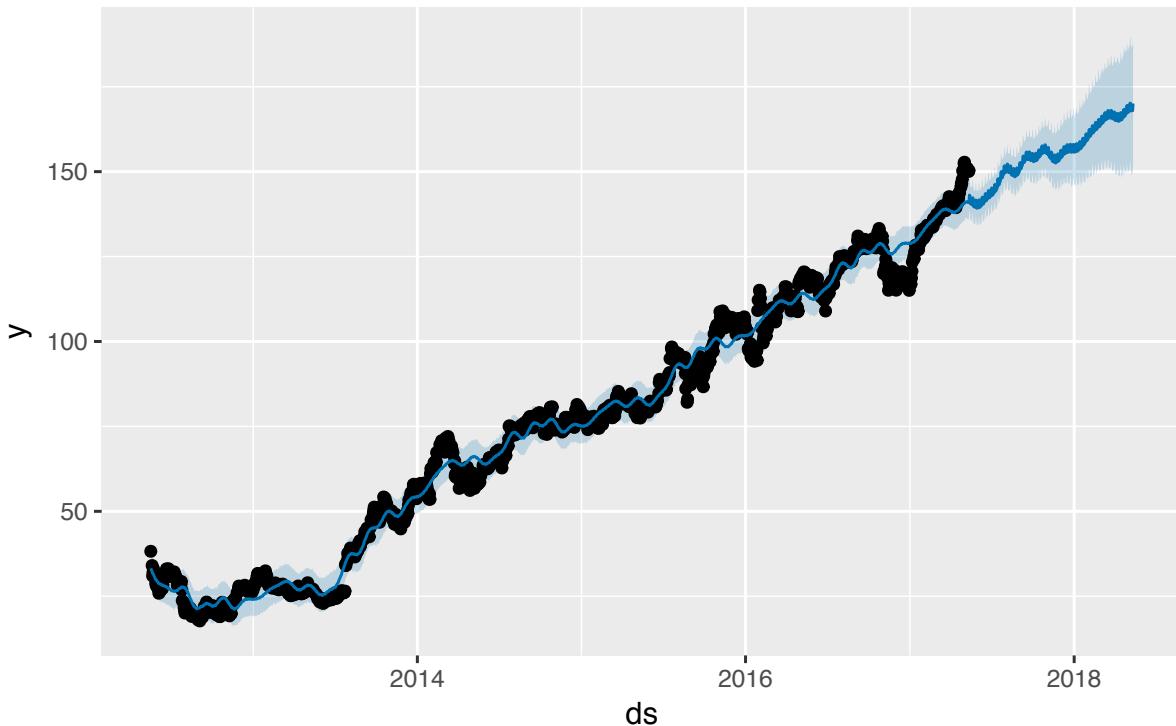
```
m <- prophet(df)

## Initial log joint probability = -13.4282
## Optimization terminated normally:
##   Convergence detected: relative gradient magnitude is below tolerance

future <- make_future_dataframe(m, periods = 365)
forecast <- predict(m, future)
```

и отобразить его на графике:

```
plot(m, forecast)
```



4.7 Rstanarm

Некоторые модели уже существуют в готовом виде в пакете `rstanarm`.

4.7.1 Линейные модели

В инструкции к пакету можно найти функцию `stan_lm()` которая используется для линейного моделирования. Возьмём пример из этой инструкции. В качестве нового параметра появляется `R2` - пропорция дисперсии предсказывающих данных в выходящих данных. Иными словами, это своего рода мера априорного распределения, чем она меньше, тем меньше априорная корреляция между выводом и предсказывающими данными, а априорная плотность сильнее сконцентрирована у нуля.

```
lm3_fit2 <- stan_lm(mpg ~ wt + qsec + am, data = mtcars, prior = R2(0.75),
                      chains = 4, iter = 1000)
```

Этот метод намного быстрее, не правда ли? Дело в том, что в `rstanarm` все модели уже скомпилированы, и поэтому семпллинг происходит быстрее, чем в моделях, которые мы описывали вручную¹⁶. Полученный объект (`lm3_fit2`) не относится к классу `stanfit` (но содержит его, об этом немного дальше), а больше похож на обычные регрессии, которые мы строим с помощью `lm()`. Чтобы убедиться в этом, выведем отчёт по оценке коэффициентов при переменных, введя команду `summary()`:

```
summary(lm3_fit2)

##
## Model Info:
##
## function:
##   family: gaussian [identity]
##   formula: mpg ~ wt + qsec + am
##   algorithm: sampling
##   priors: see help('prior_summary')
##   sample: 2000 (posterior sample size)
##   num obs: 32
##
## Estimates:
##             mean     sd    2.5%   25%   50%   75% 97.5%
## (Intercept) 9.7     7.3   -4.8   5.0   9.8  14.7 23.4
## wt          -3.7    0.7   -5.1  -4.2  -3.8  -3.3 -2.3
## qsec         1.2     0.3    0.6   1.0   1.2   1.4  1.8
## am           2.9     1.4    0.3   1.9   2.9   3.9  5.8
## sigma        2.6     0.4    2.0   2.3   2.6   2.8  3.5
## log-fit_ratio 0.0     0.1   -0.1  -0.1   0.0   0.0  0.1
## R2           0.8     0.1    0.7   0.8   0.8   0.8  0.9
## mean_PPD    20.1    0.7   18.8  19.6  20.1  20.5 21.4
## log-posterior -78.3   2.2  -83.6 -79.5 -77.9 -76.7 -75.1
##
```

¹⁶Код модели можно узнать введя команду `lm3_fit2$stanfit$stanmodel`.

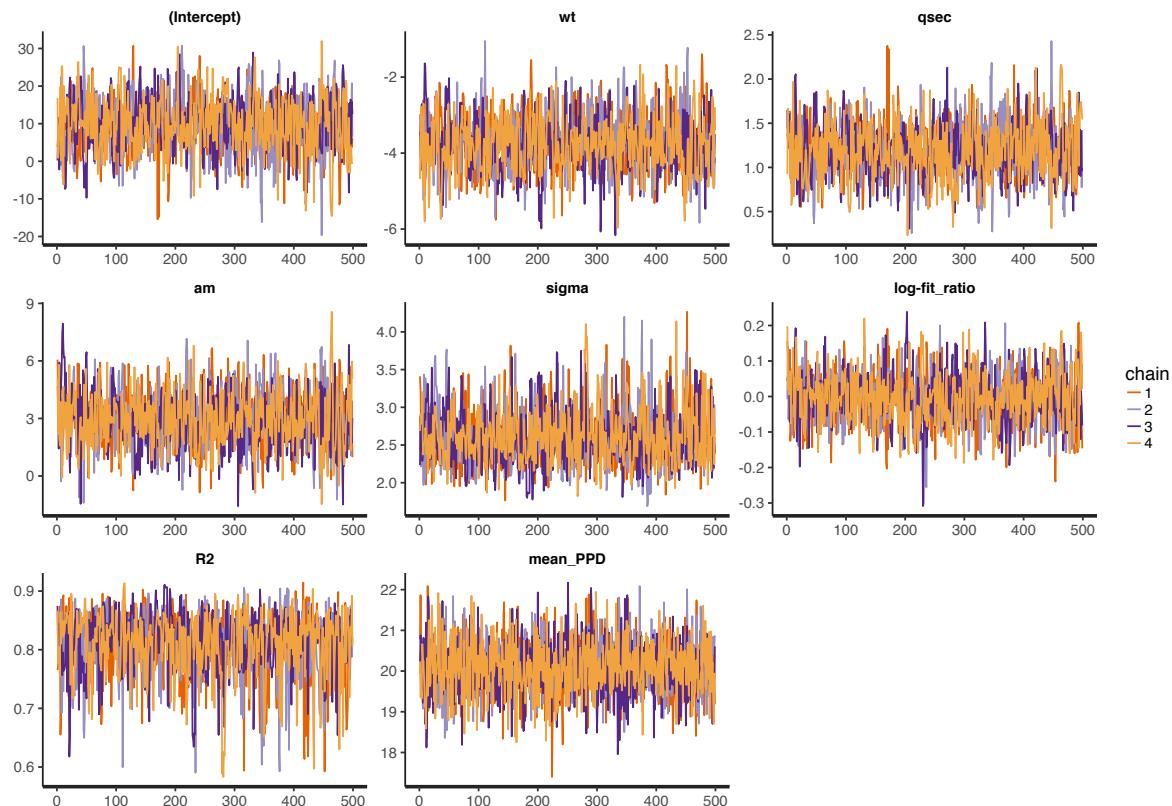
```

## Diagnostics:
##          mcse Rhat n_eff
## (Intercept) 0.3  1.0  818
## wt          0.0  1.0  807
## qsec         0.0  1.0  937
## am          0.0  1.0  994
## sigma        0.0  1.0 1040
## log-fit_ratio 0.0  1.0 1279
## R2          0.0  1.0  854
## mean_PPD    0.0  1.0 2000
## log-posterior 0.1  1.0  482
##
## For each parameter, mcse is Monte Carlo standard error, n_eff is a crude measure of effective

```

Чтобы посмотреть на сходимость цепей, нам нужно заглянуть в `stanfit` часть объекта `lm3_fit2`, указав `$stanfit`:

```
stan_trace(lm3_fit2$stanfit)
```



Если мы построим таким образом нашу первоначальную модель, то мы сможем сравнить их:

```
lm3_fit3 <- stan_lm(mpg ~ wt + hp + am, data = mtcars, prior = R2(0.75),
                      chains = 4, iter = 1000)
```

С помощью пакета `loo` и одноимённой функции можно преобразовать полученные модели в класс `loo` и сравнить их с помощью команды `compare()` или просто вывести явно вероятность получения новых данных согласно модели.

```
loo1 <- loo(lm3_fit2)
loo2 <- loo(lm3_fit3)
loo1

## Computed from 2000 by 32 log-likelihood matrix
##
##           Estimate   SE
## elpd_loo     -77.7 3.4
## p_loo        4.7 1.1
## looic       155.4 6.8
##
## Pareto k diagnostic values:
##                               Count   Pct
## (-Inf, 0.5]    (good)    31   96.9%
## (0.5, 0.7]    (ok)      1    3.1%
## (0.7, 1]      (bad)     0    0.0%
## (1, Inf)      (very bad) 0    0.0%
##
## All Pareto k estimates are ok (k < 0.7)
## See help('pareto-k-diagnostic') for details.

loo2

## Computed from 2000 by 32 log-likelihood matrix
##
##           Estimate   SE
## elpd_loo     -78.7 4.0
## p_loo        4.5 1.2
## looic       157.5 8.0
##
## Pareto k diagnostic values:
##                               Count   Pct
## (-Inf, 0.5]    (good)    29   90.6%
## (0.5, 0.7]    (ok)      3    9.4%
## (0.7, 1]      (bad)     0    0.0%
## (1, Inf)      (very bad) 0    0.0%
##
## All Pareto k estimates are ok (k < 0.7)
## See help('pareto-k-diagnostic') for details.
```

Как можно заметить, все оценки Парето к меньше 0.7. Это означает, что сильно влияющих наблюдений нет (напомню, в наборе `mtcars` 32 наблюдения), и модели можно сравнить командой `compare`:

```
compare(loo1, loo2)

## elpd_diff      se
## -1.0      2.0
```

Мы получили два числа, `elpd_diff` (Expected log pointwise predictive density) и стандартную ошибку. Если значение `elpd_diff / se` находится в диапазоне [-1.96 ; 1.96] ($z_{0.95}$), то отличие моделей несущественно.

4.7.2 Модели бинарного выбора

Как и в прошлой секции, процесс идет намного быстрее и занимает не больше минуты. Для начала построим логит-модель.

```
arm_logit <- stan_glm(data = df_titan, Survived ~ Sex + Pclass + Fare + Age,
                      family = binomial(link = "logit"), x = TRUE,
                      prior = student_t(df = 7, location = 0, scale = 2.5),
                      prior_intercept = normal(0, 10),
                      chains = 4, iter = 1000)
```

И посмотрим на результат:

```
summary(arm_logit)

##
## Model Info:
##
## function:
##   family: binomial [logit]
##   formula: Survived ~ Sex + Pclass + Fare + Age
##   algorithm: sampling
##   priors: see help('prior_summary')
##   sample: 2000 (posterior sample size)
##   num obs: 714
##
## Estimates:
##             mean    sd   2.5%   25%   50%   75%  97.5%
## (Intercept) 5.0    0.6   3.8    4.6   4.9   5.3   6.1
## Sex         -2.5   0.2  -2.9   -2.7  -2.5  -2.4  -2.1
## Pclass       -1.3   0.2  -1.6   -1.4  -1.3  -1.2  -0.9
## Fare          0.0   0.0   0.0    0.0   0.0   0.0   0.0
## Age          0.0   0.0  -0.1    0.0   0.0   0.0   0.0
## mean_PPD    0.4    0.0   0.4    0.4   0.4   0.4   0.4
## log-posterior -333.9  1.6 -337.9 -334.7 -333.6 -332.7 -331.8
##
```

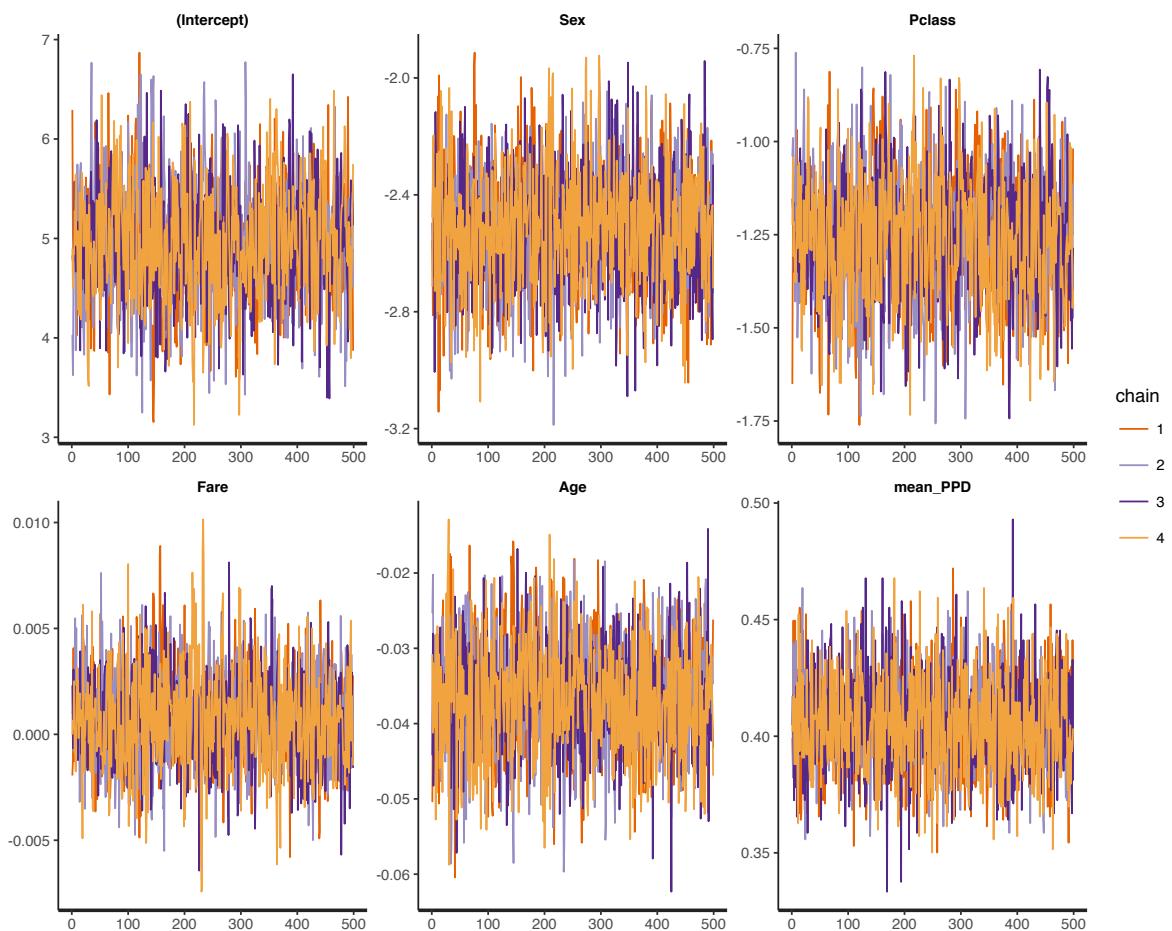
```

## Diagnostics:
##          mcse Rhat n_eff
## (Intercept) 0.0  1.0 1070
## Sex         0.0  1.0 2000
## Pclass      0.0  1.0 1118
## Fare        0.0  1.0 1596
## Age         0.0  1.0 1600
## mean_PPD    0.0  1.0 2000
## log-posterior 0.1  1.0  916
##
## For each parameter, mcse is Monte Carlo standard error, n_eff is a crude measure of effective

```

Полученный результат похож на тот, что мы получили в модели, которую описали сами. Теперь посмотрим на сходимость рядов:

```
traceplot(arm_logit$stanfit)
```



Судя по графикам, всё в порядке.

Пробит-модель строится аналогично, достаточно указать `family = binomial(link = "probit")`.

```
arm_probit <- stan_glm(data = df_titan, Survived ~ Sex + Pclass + Fare + Age,
                        family = binomial(link = "probit"), x = TRUE,
                        prior = student_t(df = 7, location = 0, scale = 2.5),
                        prior_intercept = normal(0, 10),
                        chains = 4, iter = 1000)
```

Результат:

```
summary(arm_probit)

##
## Model Info:
##
##   function: stan_glm
##   family: binomial [probit]
##   formula: Survived ~ Sex + Pclass + Fare + Age
##   algorithm: sampling
##   priors: see help('prior_summary')
##   sample: 2000 (posterior sample size)
##   num obs: 714
##
## Estimates:
##             mean    sd   2.5%   25%   50%   75% 97.5%
## (Intercept) 2.8 0.3 2.2 2.6 2.8 3.0 3.5
## Sex        -1.5 0.1 -1.7 -1.6 -1.5 -1.4 -1.3
## Pclass      -0.7 0.1 -0.9 -0.8 -0.7 -0.6 -0.5
## Fare         0.0 0.0 0.0 0.0 0.0 0.0 0.0
## Age          0.0 0.0 0.0 0.0 0.0 0.0 0.0
## mean_PPD    0.4 0.0 0.4 0.4 0.4 0.4 0.5
## log-posterior -334.9 1.6 -338.9 -335.6 -334.5 -333.7 -332.8
##
## Diagnostics:
##             mcse Rhat n_eff
## (Intercept) 0.0 1.0 1205
## Sex         0.0 1.0 2000
## Pclass       0.0 1.0 1277
## Fare         0.0 1.0 1662
## Age          0.0 1.0 1655
## mean_PPD    0.0 1.0 2000
## log-posterior 0.1 1.0 902
##
## For each parameter, mcse is Monte Carlo standard error, n_eff is a crude measure of effective
```

Предлагаю сравнить модели:

```
loo1 <- loo(arm_logit)
loo2 <- loo(arm_probit)
loo1
```

```

## Computed from 2000 by 714 log-likelihood matrix
##
##           Estimate    SE
## elpd_loo   -328.8 15.6
## p_loo       5.1  0.5
## looic      657.6 31.3
##
## All Pareto k estimates are good (k < 0.5)
## See help('pareto-k-diagnostic') for details.

loo2

## Computed from 2000 by 714 log-likelihood matrix
##
##           Estimate    SE
## elpd_loo   -330.0 15.6
## p_loo       5.0  0.5
## looic      660.1 31.2
##
## All Pareto k estimates are good (k < 0.5)
## See help('pareto-k-diagnostic') for details.

```

Влиятельных наблюдений нет, и в целом значения looic близки, поэтому отличиями в моделях можно пренебречь.

Сравним полученный байесовский результат логит модели с традиционным частотным подходом:

```

freq_logit <- glm(data = df_titan,
                    Survived ~ Sex + Pclass + Fare + Age,
                    family = binomial(link = "logit"), x = TRUE)
summary(freq_logit)

##
## Call:
## glm(formula = Survived ~ Sex + Pclass + Fare + Age, family = binomial(link = "logit"))
##   data = df_titan, x = TRUE)
##
## Deviance Residuals:
##       Min      1Q      Median      3Q      Max
## -2.739  -0.679  -0.395   0.649   2.464
##
## Coefficients:
##             Estimate Std. Error z value Pr(>|z|)
## (Intercept) 4.988040  0.572189   8.72 < 2e-16 ***
## Sex         -2.518197  0.207856  -12.12 < 2e-16 ***
## Pclass       -1.269741  0.158625   -8.00 1.2e-15 ***
## Fare         0.000537  0.002182    0.25     0.81

```

```

## Age           -0.036707   0.007680   -4.78  1.8e-06 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## (Dispersion parameter for binomial family taken to be 1)
##
## Null deviance: 964.52 on 713 degrees of freedom
## Residual deviance: 647.23 on 709 degrees of freedom
## AIC: 657.2
##
## Number of Fisher Scoring iterations: 5

```

Можно заметить сходство полученных оценок. Также можно посмотреть на предельные эффекты для Байесовский и частотной моделей. Для этого нам понадобится функция `maBina()` из пакета `erer`. Чтобы эта функция выдала какой-либо результат, в `glm()` нужно указать `x = TRUE`.

```

maBina(arm_logit)

##          effect error t.value p.value
## (Intercept) 1.152 0.140   8.214  0.000
## Sex        -0.555 0.036 -15.542  0.000
## Pclass      -0.293 0.038  -7.679  0.000
## Fare         0.000 0.001   0.338  0.735
## Age         -0.008 0.002   -4.788  0.000

maBina(freq_logit)

##          effect error t.value p.value
## (Intercept) 1.166 0.137   8.508  0.000
## Sex        -0.555 0.037 -15.173  0.000
## Pclass      -0.297 0.037  -8.109  0.000
## Fare         0.000 0.001   0.246  0.806
## Age         -0.009 0.002   -4.797  0.000

```

Сходство результатов очевидно.

4.7.3 Прогнозирование в `rstanarm`

Rstanarm позволяет и прогнозировать. Сделать это можно стандартным образом, с помощью функции `predict`. Рассмотрим прогноз шансов на выживание в зависимости от возраста на данных Титаника:

```

#новые данные
newdata = data.frame(Age = seq(from = 5, to = 100, length = 100),
                      Sex = 1 , Pclass = 2 , Fare = 100)

```

```

pr_armlogit <- predict(arm_logit, newdata = newdata, se.fit = TRUE)
newdata_pr <- cbind(newdata, pr_armlogit)

newdata_pr <- mutate(newdata_pr, prob = plogis(fit),
                     left_ci = plogis(fit - 1.96*se.fit),
                     right_ci = plogis(fit + 1.96*se.fit))

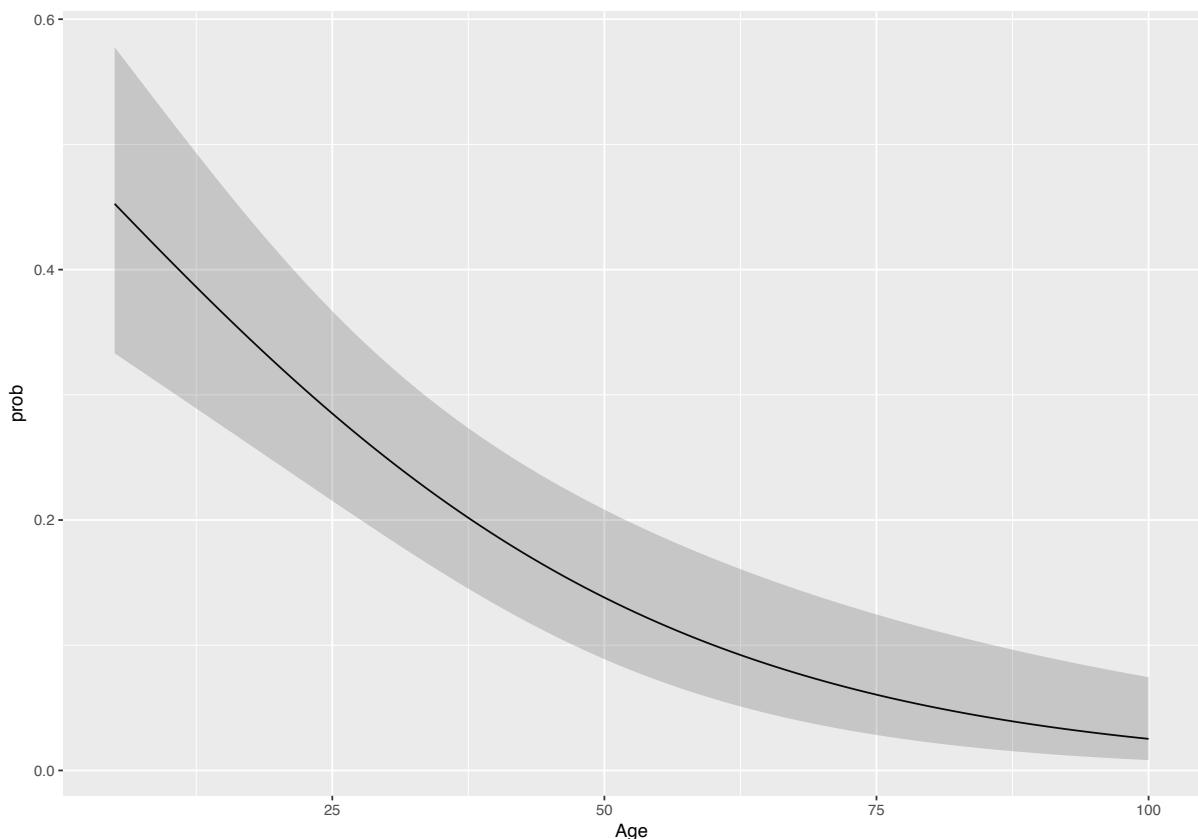
```

Результат отобразим на графике:

```

qplot(data = newdata_pr, x = Age, y = prob, geom = "line") +
  geom_ribbon(aes(ymin = left_ci, ymax = right_ci), alpha = 0.2)

```



5 Практическая часть

5.1 Индивидуальные предпочтения в телекоме

Благодаря широким возможностям, Байесовский подход можно (и нужно) применять в исследованиях в самых разнообразных сферах. В данной секции речь пойдёт о потребительских предпочтениях индивидов при выборе опций сотовой связи¹⁷. Индивидами являются студенты Высшей Школы Экономики. Для начала рассмотрим структуру данных.

```
tele_dat <- read_spss("dat/conjoint_host_sim_dummy.sav")
tele_dat <- tele_dat[, order(names(tele_dat))]
glimpse(tele_dat[, 1:55])

## Observations: 296
## Variables: 55
## $ id                  <dbl> 93, 105, 124, 127, 141, 158, 161, 1...
## $ T1_C1_Gigabytes     <dbl> 1, 7, 4, 1, 10, 3, 4, 5, 9, 6, 9, 8...
## $ T1_C1_Hostprovider1 <dbl> 0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 1, ...
## $ T1_C1_Hostprovider2 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, ...
## $ T1_C1_Hostprovider3 <dbl> 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, ...
## $ T1_C1_Minutes       <dbl> 4, 1, 4, 3, 4, 6, 4, 6, 3, 3, 4, 7, ...
## $ T1_C1_Payment        <dbl> 2, 2, 1, 2, 2, 1, 1, 1, 2, 2, 2, 1, ...
## $ T1_C1_Personalization <dbl> 1, 1, 1, 2, 2, 1, 1, 1, 1, 2, 1, 2, ...
## $ T1_C1_Price          <dbl> 4, 5, 3, 4, 5, 5, 3, 2, 5, 4, 5, 2, ...
## $ T1_C1_Quantitysim2    <dbl> 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, ...
## $ T1_C1_Quantitysim3    <dbl> 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, ...
## $ T1_C2_Gigabytes       <dbl> 5, 2, 3, 8, 3, 9, 3, 6, 2, 9, 8, 7, ...
## $ T1_C2_Hostprovider1   <dbl> 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0, ...
## $ T1_C2_Hostprovider2   <dbl> 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 1, ...
## $ T1_C2_Hostprovider3   <dbl> 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, ...
## $ T1_C2_Minutes         <dbl> 3, 4, 5, 6, 1, 1, 5, 2, 7, 2, 7, 3, ...
## $ T1_C2_Payment          <dbl> 2, 2, 2, 1, 2, 2, 2, 2, 1, 2, 1, ...
## $ T1_C2_Personalization <dbl> 1, 1, 2, 1, 1, 2, 2, 2, 1, 1, 1, 2, ...
## $ T1_C2_Price            <dbl> 5, 2, 5, 2, 3, 4, 5, 5, 3, 5, 4, 3, ...
## $ T1_C2_Quantitysim2      <dbl> 0, 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, ...
## $ T1_C2_Quantitysim3      <dbl> 0, 1, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, ...
## $ T1_C3_Gigabytes        <dbl> 2, 9, 6, 9, 6, 8, 6, 9, 7, 10, 4, 9, ...
## $ T1_C3_Hostprovider1     <dbl> 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ T1_C3_Hostprovider2     <dbl> 0, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, ...
## $ T1_C3_Hostprovider3     <dbl> 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 1, ...
## $ T1_C3_Minutes          <dbl> 7, 2, 6, 4, 6, 7, 6, 1, 1, 7, 1, 5, ...
## $ T1_C3_Payment           <dbl> 1, 2, 1, 1, 1, 2, 1, 2, 1, 2, 1, 2, ...
## $ T1_C3_Personalization <dbl> 2, 1, 1, 1, 2, 2, 1, 2, 1, 2, 2, 1, ...
## $ T1_C3_Price              <dbl> 2, 1, 4, 5, 2, 3, 4, 1, 4, 2, 2, 1, ...
## $ T1_C3_Quantitysim2      <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, ...
## $ T1_C3_Quantitysim3      <dbl> 1, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, ...
## $ T1_C4_Gigabytes         <dbl> 8, 5, 7, 6, 8, 7, 7, 10, 4, 8, 10, ...
```

¹⁷Описание здесь: http://statref.ru/ref_qasrnqasotr.html

```

## $ T1_C4_Hostprovider1 <dbl> 1, 0, 0, 1, 0, 1, 0, 0, 0, 0, 1, 0, ...
## $ T1_C4_Hostprovider2 <dbl> 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, ...
## $ T1_C4_Hostprovider3 <dbl> 0, 0, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, ...
## $ T1_C4_Minutes <dbl> 2, 7, 3, 7, 3, 3, 3, 4, 5, 5, 5, 2, ...
## $ T1_C4_Payment <dbl> 1, 1, 1, 2, 2, 1, 1, 1, 1, 1, 1, 2, ...
## $ T1_C4_Personalization <dbl> 2, 2, 2, 2, 1, 1, 2, 1, 2, 1, 2, 1, ...
## $ T1_C4_Price <dbl> 1, 4, 1, 3, 1, 2, 1, 3, 2, 1, 3, 4, ...
## $ T1_C4_Quantitysim2 <dbl> 0, 1, 1, 0, 1, 1, 1, 0, 1, 1, 0, 0, ...
## $ T1_C4_Quantitysim3 <dbl> 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, ...
## $ T1_C5_Gigabytes <dbl> 3, 8, 1, 3, 2, 1, 1, 1, 6, 5, 2, 5, ...
## $ T1_C5_Hostprovider1 <dbl> 0, 0, 1, 0, 1, 0, 1, 0, 0, 0, 0, 0, ...
## $ T1_C5_Hostprovider2 <dbl> 1, 0, 0, 0, 0, 1, 0, 1, 0, 1, 0, 0, ...
## $ T1_C5_Hostprovider3 <dbl> 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, ...
## $ T1_C5_Minutes <dbl> 1, 3, 1, 2, 5, 5, 1, 5, 2, 1, 3, 4, ...
## $ T1_C5_Payment <dbl> 1, 1, 2, 1, 1, 2, 2, 2, 1, 2, 2, 1, ...
## $ T1_C5_Personalization <dbl> 2, 2, 1, 1, 2, 2, 1, 2, 2, 2, 1, 2, ...
## $ T1_C5_Price <dbl> 3, 3, 2, 1, 4, 1, 2, 4, 1, 3, 1, 5, ...
## $ T1_C5_Quantitysim2 <dbl> 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, ...
## $ T1_C5_Quantitysim3 <dbl> 0, 1, 1, 0, 1, 1, 1, 0, 1, 0, 1, 0, ...
## $ T1_select <dbl> 3, 3, 3, 2, 3, 4, 1, 4, 5, 4, 5, 3, ...
## $ T2_C1_Gigabytes <dbl> 9, 9, 8, 6, 5, 3, 8, 5, 10, 3, 1, 6...
## $ T2_C1_Hostprovider1 <dbl> 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, ...
## $ T2_C1_Hostprovider2 <dbl> 0, 1, 0, 1, 0, 0, 0, 0, 0, 1, 1, ...

```

В наборе данных содержится 296 наблюдений и 359 параметров (можно проверить с `ncol(tele_dat)`). Структура следующая: на 7 карточках (T_1, \dots, T_7) находятся по 5 альтернатив (C_1, \dots, C_5). Указанные числа — это уровни, выбранные пользователями.

```

unique(tele_dat$T1_C1_Minutes)

## [1] 4 1 3 6 7 5 2

unique(tele_dat$T1_C1_Gigabytes)

## [1] 1 7 4 10 3 5 9 6 8 2

unique(tele_dat$T1_C1_Personalization)

## [1] 1 2

unique(tele_dat$T1_C1_Minutes)

## [1] 4 1 3 6 7 5 2

unique(tele_dat$T1_select)

## [1] 3 2 4 1 5

```

Классическая регрессия на частотном подходе здесь неуместна, в случае если мы хотим включить все переменные (недостаточно степеней свободы). Поэтому, конкретно для этой задачи, имеет смысл использовать Байесовский подход.

Теперь нам нужно разделить имеющийся набор данных. Нам нужны: число наблюдений (отдельно их id), число карточек и альтернатив (кластеров), а также колонки `TN_select` и сами наблюдения.

```
n_individuals <- nrow(tele_dat)
n_cards <- 7
n_alternatives <- 5
data_ids <- tele_dat$id
#Получение матриц наблюдений и выборов
df_X <- dplyr::select(tele_dat, -ends_with("select"))
df_y <- dplyr::select(tele_dat, ends_with("select"))
```

Далее нужно привести матрицу с наблюдениями в более структурированный вид (отделить карточки и альтернативы).

```
df_X_melted <- melt(df_X, id.vars = c("id", "version"))
head(df_X_melted)

##      id version      variable value
## 1  93        40 T1_C1_Gigabytes     1
## 2 105        72 T1_C1_Gigabytes     7
## 3 124        97 T1_C1_Gigabytes     4
## 4 127        32 T1_C1_Gigabytes     1
## 5 141        99 T1_C1_Gigabytes    10
## 6 158        62 T1_C1_Gigabytes     3

df_X_sep <- tidyverse::separate(df_X_melted,
                                 variable,
                                 into = c("card",
                                         "alternative",
                                         "variable"),
                                 sep = "_")
head(df_X_sep)

##      id version card alternative variable value
## 1  93        40    T1          C1 Gigabytes     1
## 2 105        72    T1          C1 Gigabytes     7
## 3 124        97    T1          C1 Gigabytes     4
## 4 127        32    T1          C1 Gigabytes     1
## 5 141        99    T1          C1 Gigabytes    10
## 6 158        62    T1          C1 Gigabytes     3
```

Отлично, теперь данные разбиты. Следующим шагом будет создание вектора выбора (`select`) и матрицы наблюдений для каждого индивида. Это можно сделать с помощью цикла `for`:

```

choice_data <- list()
# Для каждого индивида
for (individual_no in 1:n_individuals) {
  # Получаем его id
  person_id <- data_ids[individual_no]
  # Извлекаем из таблицы T?_select значения для индивида
  individual_y <- unlist(df_y[individual_no, ])
  names(individual_y) <- NULL
  # Сортируем по id индивида и убираем столбцы id и version
  person_X_melted <- df_X_sep %>% dplyr::filter(id == person_id) %>%
    dplyr::select(-id, -version)
  # создаем матрицу с наблюдениями по индивиду
  # для каждой карточки и альтернативы
  person_X_df <- dcast(person_X_melted, card + alternative ~ variable,
                        value.var = "value")
  # Удаляем столбцы с метками карточки и альтернативы
  person_X <- dplyr::select(person_X_df, -card, -alternative) %>% as.matrix()
  # Создаем список из матриц наблюдений и векторов выборов
  choice_data[[individual_no]] <- list(y = individual_y, X = person_X)
}

```

В результате мы получим список с матрицами наблюдений (предпочтений) и векторов выбора. Для наглядности, посмотрим на элементы этого списка:

```

head(individual_y)
## [1] 4 2 4 4 2 2

head(person_X_df)

##   card alternative Gigabytes Hostprovider1 Hostprovider2
## 1    T1          C1        4            1            0
## 2    T1          C2        8            0            1
## 3    T1          C3        5            0            0
## 4    T1          C4       10            0            0
## 5    T1          C5        9            0            0
## 6    T2          C1        1            1            0
##   Hostprovider3 Minutes Payment Personalization Price Quantitysim2
## 1              0     7      1            1            1            0
## 2              0     3      2            2            4            0
## 3              1     2      2            2            2            1
## 4              0     5      1            1            3            0
## 5              1     6      1            2            5            0
## 6              0     1      1            2            4            0
##   Quantitysim3
## 1             1
## 2             0
## 3             0
## 4             0
## 5             1
## 6             0

```

В переменной `ttindividual_y` хранятся значения `T1_select`, `T2_select`, ..., `T7_select` для 296 индивида (он был последним, и его данные сохранились в ячейке). А переменная `person_X_df`, хранит его выбор параметра для каждой альтернативы каждой карточки. Переменная `person_X` аналогична последней, но является матрицей, и хранит только числовые значения. То есть, допустим, если я хочу знать данные для 123 индивида, я могу просто ввести `choice_data[[123]]`.

```
head(choice_data[[123]]$X)

##      Gigabytes Hostprovider1 Hostprovider2 Hostprovider3 Minutes
## [1,]         3          0          0          1          7
## [2,]         1          0          1          0          3
## [3,]         2          0          0          0          2
## [4,]         9          1          0          0          5
## [5,]         7          1          0          0          1
## [6,]         8          0          0          0          5
##      Payment Personalization Price Quantitysim2 Quantitysim3
## [1,]     1           1       2          0          0
## [2,]     2           2       3          0          0
## [3,]     1           1       5          0          1
## [4,]     2           2       1          1          0
## [5,]     2           2       4          0          1
## [6,]     1           2       3          1          0

choice_data[[123]]$y

## [1] 1 1 1 4 3 3 2
```

Где получу, что выбор 123-го индивида в карточках Т1–Т3 был 1, для Т4 — 4, 3 для Т5 и Т6, и 2 в Т7. Вторым элементом идёт матрица (`T1_C1` Гигабайты — 3, `T2_C1` Минуты — 5, и так далее). Образно говоря, наш набор данных — это кирпич, по длине которого идут индивиды, ширине — кластеры (карточки и альтернативы), а в глубину — переменные (Гигабайты, минуты и другие).

В целом наш набор готов и представлен в удобном виде.

6 Заключение

В данной работе я продемонстрировал преимущества Байесовского подхода и языка Stan и разобрал один практический пример. Эта работа еще будет дополняться: в ней недостаточно подробно разобраны модели временных рядов и многое другое.

Байесовский подход открывает широкие возможности в сфере исследований и науке о данных и постепенно оттеснит частотный подход, который на сегодняшний день, ввиду исторических событий, занимает доминирующее положение. Это вовсе не значит, что частотный подход неверный, наоборот, в нём есть многие полезные для своих задач инструменты: OLS, LASSO, метод максимального правдоподобия и другие. Байесовский подход сможет решить задачи, в которых частотные методы могут испытывать затруднения. Его развитие плотно сопряжено с развитием вычислительных технологий, а также с количеством преподавателей, способных доходчиво донести достаточно сложные инструменты Байесовского подхода студентам. Выражаю надежду, что читатели данной работы найдут её полезной для своих нужд и заинтересуются изучением этой темы.

7 FAQ

В данной секции я собрал ответы на вопросы, которые могут возникнуть по ходу чтения работы.

7.1 Критерии loo и waic

LOO (Leave-one-out) — информационный критерий, используемый для сравнения Байесовских моделей. В качестве альтернативы можно использовать критерий WAIC (Widely Accepted Information Criterion), однако он менее предпочтителен. Эти критерии предназначены для оценки точности предсказывания моделей, и, в отличие от критерия Акаике (AIC), учитывают априорное распределение. Важно отметить, что в сравниваемых моделях не должно быть влиятельных наблюдений. Их наличие можно определить просто выведя объект, с помощью графика (`plot([loo object])`) или команды (`pareto_k_ids`). Подробнее читать здесь: <http://mc-stan.org/rstanarm/reference/loo.stanreg.html> и здесь: <https://cran.r-project.org/web/packages/loo/loo.pdf>.

7.2 На что влияет количество цепей и итераций?

От числа цепей зависит то, насколько вероятно обнаружение ошибки в модели. При построении моделей и их анализе, Вы наверняка столкнетесь с ситуацией, когда сходятся не все цепи (если они вообще сходятся). В этом случае нельзя доверять и сопшедшемся цепям. Оптимального числа цепей нет, однако имеет смысл брать не менее 3–4, а при наличии большего числа ядер в процессоре даже больше.

Число итераций влияет на то, сойдётся ли цепь. Если выбрать слишком маленькое количество, то не исключено, что цепь «не успеет» сойтись, и при наличии достаточно хороший модели Вам придётся придумать новую. По умолчанию берутся 2000 итераций, хотя как правило, даже 1000 достаточно для большинства задач. Больше число требует большего времени для вычислений (что очевидно), но и уменьшает стандартную ошибку, увеличивает `n_eff` (т.е. увеличивает шансы сходимости) и в целом обеспечивает более точные оценки. Поэтому увеличении итераций рекомендуется в случаях, когда Вас не устраивает значение `n_eff` или необходимо понизить величину ошибки. В данной работе количество итераций было увеличено в примере с прогнозированием, как раз, чтобы поднять `n_eff`.

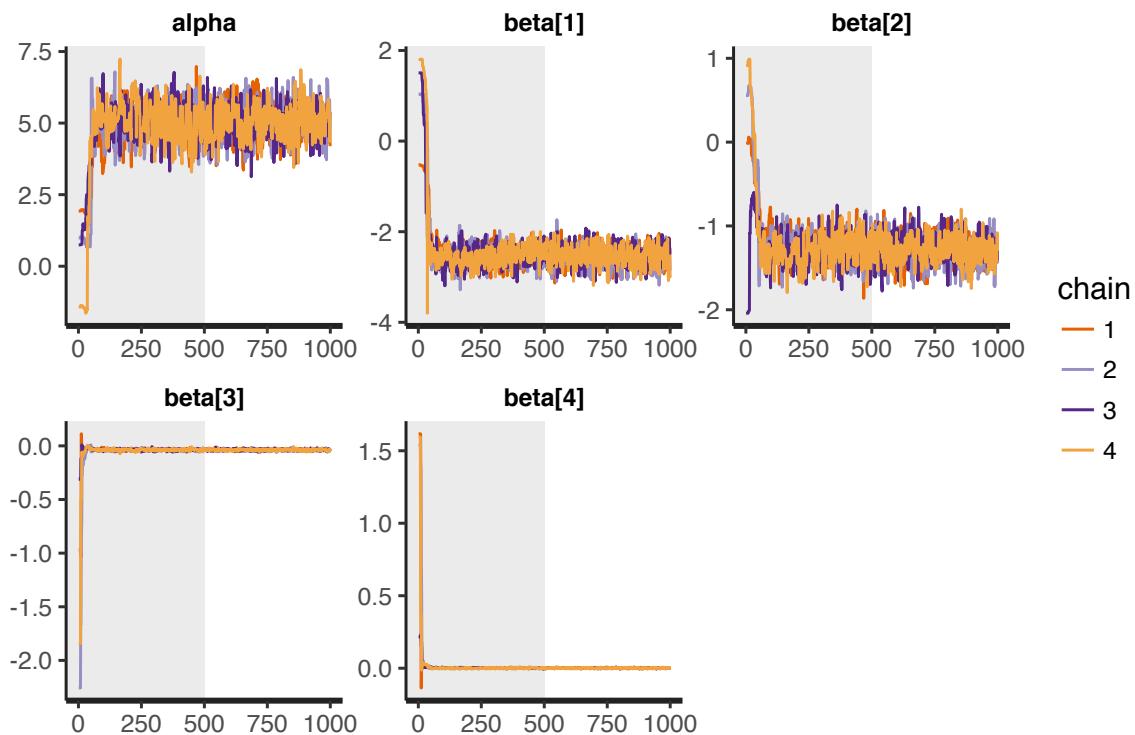
7.3 Как выбрать априорное распределение?

В целом нельзя вывести каких-то строгих правил о выборе априорное распределения. При выборе можно субъективно указать, что параметр θ лежит в каком-то множестве Θ . Также можно руководствоваться существующей, к примеру, исторической информацией и указать априорное распределение, основанное на распределении выборки из этих данных. Можно и вовсе ничего не предполагать касательно параметра. Однако даже в этом случае мы неявно предполагаем, что у θ равные шансы принять любое значение, другими словами, θ распределено равномерно от $-\infty$ до $+\infty$.

7.4 Что такое warmup и sampling?

Процесс генерации цепи делится на две фазы — warmup (или burnin) и sampling. Деление условное, по умолчанию в Stan обе фазы занимают по 50% от всех итераций, и при желании пропорцию можно изменить. Оценки рассчитываются из sampling части цепи (в этом можно убедиться, введя `nrow(as.array([stanfit]))`). На графиках фаза warmup не отображается, но добавив аргумент `inc_warmup = TRUE` можно отобразить и её:

```
stan_trace(logittest_fit, inc_warmup = TRUE)
```

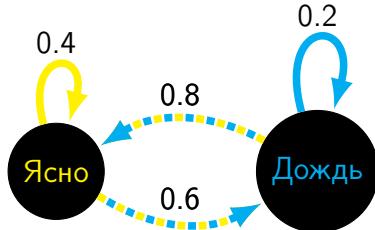


Думаю, из графиков вполне очевидно, почему не стоит убирать фазу разогрева (серая заливка). Также очень часто есть резон уменьшить пропорцию (до 20%–30%) чтобы воспользоваться большим числом итераций в семплинге.

7.5 Марковские цепи

Марковские цепи — последовательности случайных событий, в которых вероятность каждого события зависит исключительно от текущего состояния (но не от прошлых). Чтобы лучше понять суть этого понятия, предлагаю обратиться к классической модели о погоде. У нас есть два состояния: ясный и дождливый (`sunny`, `rainy`) дни, а также вероятности, с которыми эти состояния переходят друг в друга (или в самих

себя). Таблицу с вероятностями также называют матрицей перехода. Схематически нашу модель можно представить следующим образом¹⁸:



	$R = \text{Дождь}$	$S = \text{Ясно}$
$R = \text{Дождь}$	$P(R R) = 0.2$	$(S R) = 0.8$
$S = \text{Ясно}$	$P(R S) = 0.6$	$(S S) = 0.4$

В ходе работы модели, состояние R будет сменяться на состояние S (или снова R) с некоторой вероятностью. Аналогично для состояния S . Замечу, что переход в другое состояние осуществляется в любом случае, поэтому $P(\bullet|S_i) = 1$. Однако, это не значит, что из состояния i мы можем перейти во все остальные $n - 1$, то есть могут быть случаи, когда $P(S_j|S_i) = 0$. Если мы запишем получаемые состояния по хронологии, то и получим Марковскую цепь. С увеличением числа состояний увеличивается и размерность таблицы вероятностей перехода (т.е. при n состояниях получим матрицу вероятностей $n \times n$).

7.6 МСМС — алгоритм Монте-Карло по схеме Марковской цепи

Как было сказано в секции 4.2, МСМС (Markov Chain Monte-Carlo) — это способ оценивания параметров модели путём симуляции ожидаемых значений. Если говорить конкретнее, то МСМС решает проблему семплирования сложного распределения. Допустим, что у нас есть некоторое очень большое (бесконечное) количество листочеков с числами, и нам нужно найти значение числа на листке, который сейчас перевернут. Разумеется, что у чисел на листках есть некоторое распределение D , и мы можем найти точную вероятность получения числа x из этого распределения $p(x)$. Возникает проблема: как нам получить максимально точную вероятность угадывания, при условии, что нам известно вероятностное распределение? Ведь нам неизвестен алгоритм, по которому генерировались эти числа. То есть нам нужно придумать такой случайный алгоритм (функцию) $f(t)$, чтобы вероятность получения числа x была равна $p(x)$. Говоря об оценке параметров, то нам необходимо найти математическое ожидание этой функции.

$$E(f(X)) = \int f(X)p(X)dX$$

Для этого потребуется выборка из априорного распределения

$$X_1, X_2, \dots, X_N \sim p(X)$$

¹⁸Интерактивную визуализацию можно найти здесь: <http://setosa.io/ev/markov-chains/>

Которая, в свою очередь, является марковской цепью. Однако, случайные величины в такой выборке не случайны, и чтобы получить независимый набор, берут каждый t -ый элемент. Стоит отметить, что разные вариации МСМС работают по-разному (есть схема Метрополиса-Гастингса, схема Гиббса), что влияет на их эффективность, число необходимых итераций, насколько хорошо «исследуется» распределение и так далее.

Список литературы

- [1] Richard McElreath, «Statistical Rethinking», 2015
- [2] Stan Development Team, «Stan Modeling Language User's Guide and Reference Manual», Stan Version 2.15.0, 2017
- [3] Andrew Gelman, Daniel Lee, Jiqiang Guo, «Stan: A probabilistic programming language», Journal of Statistical Software, 2015
- [4] A. Gelman, J. Hill, «Data Analysis Using Regression and Multilevel/Hierarchical Models», Cambridge University press, 2007
- [5] A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, Aki Vehtari, D. B. Rubin, «Bayesian Data Analysis», CRC Press, 2014
- [6] Thomas Bayes, «An Essay towards solving a Problem in the Doctrine of Chances», Philosophical Transactions of the Royal Society of London 53 (1763), 370–418
- [7] Cameron Davidson-Pilon, «Bayesian Methods for Hackers: Probabilistic Programming and Bayesian Inference», Addison-Wesley Professional, 2016
- [8] Ветров Д.П., Кропотов Д.А. Байесовские методы машинного обучения, учебное пособие по спецкурсу, 2007
- [9] Alex Rogozhnikov, «Hamiltonian Monte Carlo explained», Dec 19, 2016, http://arogozhnikov.github.io/2016/12/19/markov_chain_monte_carlo.html (на момент 25.06.2017)
- [10] Aki Vehtari, Andrew Gelman, Jonah Gabry, «Practical Bayesian model evaluation using leave-one-out cross-validation and WAIC», arXiv:1507.04544 [stat.CO], 2016
- [11] «Програмное обеспечение для байесовского анализа. Продедура получения новых значений параметров в Sawtooth». http://statref.ru/ref_qasrnaqasotr.html (на момент 25.06.2017)
- [12] Байесовский подход и его применение в задачах страхования, гарантийного обслуживания и принятия решений с проведением экспериментов. <http://levvu.narod.ru/Papers/Bayes.pdf> (на момент 25.06.2017).
- [13] С. А. Айвазян, «Байесовский подход в эконометрическом анализе», Прикладная Эконометрика, №1(9) 2008
- [14] «R Users Will Now Inevitably Become Bayesians» <https://thinkinator.com/2016/01/12/r-users-will-now-inevitably-become-bayesians/> (на момент 25.06.2017)
- [15] Kim Larsen, «Sorry ARIMA, but I'm Going Bayesian» <http://multithreaded.stitchfix.com/blog/2016/04/21/forget-arima/>, 2016 (на момент 25.06.2017)