Анализ данных с R (III).

© С. В. Петров*, Е. М. Балдин†, В. Е. Лявшук‡

*p2004r@gmail.com
†E.M.Baldin@inp.nsk.su
‡lve@tut.by

Эмблема **R** взята с официального сайта проекта `http://developer.r-project.org/Logo/`

Оглавление

Глава 8

Размножаем реальность (bootsrapping на примере)

Довольно легко оценить эффективность хоккеиста по числу забитых голов и голевых передач, но что делать с оценкой эффективности более сложных действий? Давайте оценим эффективность рассылки спама...

Ну, не совсем того спама, о котором вы вероятно подумали. Рассылка осуществлялась университетом с целью информировать абитуриентов о своём существовании. Общаясь с абитуриентами можно заметить одну очень любопытную особенность: выбор ВУЗа и уж тем более факультета часто делается достаточно случайным образом. Безусловно есть группа молодых людей, которые почти с раннего детства знают куда поступать, но большинство серьёзно об этом моменте почему-то особо не задумываются. Очевидно что абсолютно всех абитуриентов конкретного высшего учебного заведения объединяет одно: они все об этом ВУЗе *знают*.

Задача Факультет математики и информатики Гродненского государственного университет имени Янки Купалы (ГрГУ им. Я. Купалы) рассылал потенциальным абитуриентам (не кому попало, а именно подходящим для обучения на этом факультете молодым людям) письма-приглашения о поступлении. Необходимо количественно оценить экономическую эффективность такой деятельности.

Эта проблема не нова, так реклама является необходимым инструментом стимулирования продаж. С течением времени на получение одинакового покупательского отклика необходимо тратить всё больше средств на рекламу и, при этом, диверсифицировать каналы распространения рекламы. Отсюда естественно желание, в нашем случае, руководства вуза оптимизировать затраты на рекламу на базе разумных расчётов, учитывающих реальность конкретной ситуа-

ции рекламирования, а не идеальную модель из учебника. За эту задачу взялись сотрудники университета Ю. А. Войтукевич, В. Е. Лявшук и С. В. Петров.

Реальные условия задачи В 2009 г. в эксперимент по повышению качества абитуриентов «на входе» в образовательный процесс в ГрГУ им. Я. Купалы включился физико-технический факультет. Для определения целевой аудитории рекламной рассылки при помощи методов многомерной статистики были одновременно проанализированы результаты тестов по физике и математике 4116 абитуриентов. Из них для списка рассылки по указанной выше методике отобрано 598 потенциальных кандидатов к поступлению на факультет, которым были высланы персональные (то есть с указанием фамилии, имени, отчества) приглашения к поступлению на физтех ГрГУ. Число 598 определялось ограничениями бюджета на рекламную кампанию. По итогам приемной кампании 2009 г. на физико-технический факультет было принято 146 абитуриентов. Из принятых абитуриентов 61 в свое время получил персональное письмо-приглашение. Теперь хочется оценить количественный эффект от персонифицированной рассылки, если, естественно, этот эффект есть.

Оценка эффективности рекламы является весьма нетривиальной задачей. Рекламодатель всегда может точно указать величину запланированных или уже сделанных затрат на рекламу. А вот с оценкой величины не только будущего, но даже уже имеющегося эффекта от рекламирования у заказчика обычно возникают трудности. Более-менее ясную картину можно получить в ситуации вывода на рынок абсолютно нового товара/услуги («истинной новинки»), когда информирование покупателей начинается «с нуля». В остальных же случаях сложно определить, сколько покупателей пришло за покупкой под влиянием конкретной рекламы по конкретному каналу распространения информации, а сколько под влиянием иных факторов. В тоже время, только точное определение величины эффекта от воздействия конкретного канала распространения рекламы позволяет адекватно планировать, контролировать и вовремя корректировать рекламную кампанию.

Решение Если мы знаем количество откликов в результате проведенной кампании и количество контактов потенциальных покупателей с определенным каналом распространения рекламы, то можем, построив статистическую модель случайного поведения покупателей, многократно (пусть «многократно» — это 10 тысяч и более раз) сравнить реальную картину поведения покупателей с моделью. Тогда удастся с какой-то точностью оценить, сколько покупателей приняло решение о покупке под влиянием рекламы по конкретному каналу.

Фактически выше сформулирована задача для метода размножения выборок или бутстреп-анализа (bootstrap resampling technique или bootsrapping). Этот метод был предложен американским статистиком Бредли Эфроном (Bradley Efron) в 1977 г. Он отличается от традиционных методов статистического анализа тем, что предполагает многократную обработку различных частей одних и тех же

данных, как бы поворот их «разными гранями», и сопоставление полученных таким образом результатов. Бутстреп-процедура не требует информации о виде закона распределения изучаемой случайной величины и в этом смысле может рассматриваться как метод непараметрической статистики.

Есть мнение, что использовать этот метод истинный исследователь будет только от полной безнадёги и если имеются хоть какие-нибудь рабочие гипотезы, позволяющие предсказать поведение модели, то лучше заняться их развитием. Предположим, что таковые гипотезы у нас отсутствуют или нам их просто лень их развивать.

Рассчитаем насколько полученный результат приёмной кампании близок к случайному. Очевидно, что для этого достаточно рассчитать распределение вероятности количества зачисленных абитуриентов получивших письмо-приглашение в численном эксперименте по случайной рассылке 598 писем.

Для этого случайным образом выбираем 598 человек из 4165 возможных кандидатов и проверяем, сколько среди них находится тех, кто действительно поступил на физический факультет. Данный эксперимент повторяем 10 тысяч раз и строим распределение числа выбранных таким образом поступивших на факультет. Данные 10 тысяч попыток позволяют построить распределение числа поступивших абитуриентов, получивших письмо приглашение случайным образом.

При некоторой фантазии вспомнив о подвигах барона Мюнхгаузена «bootstrap» можно перевести как «вытягивание себя из болота за шнурки от ботинок». Что, собственно говоря, мы и делаем. Число 10 тысяч взято не спроста. Бутстреп-процедура позволяет только асимптотически приблизиться к верному ответу, поэтому число тестов должно быть очень большим.

К вопросу о данных Информация об абитуриентах и поступивших представлена в виде двух таблиц данных: `abitura` — список потенциальных сдававших тест абитуриентов (тех, о ком была информация) и `priem` — список принятых. В каждой из этих таблиц данных присутствуют поля Фамилия, Имя и Отчество, которые по идее однозначно определяют конкретного абитуриента. Совершенно логично в качестве уникального ключа представить комбинацию этих полей.

Техническое исполнение На языке среды статистических расчетов **R** данная задача выглядит примерно следующим образом:

```
# Обнуляем вектор результатов
> bstr <- 1
#  Записываем в вектор результатов 10000 вариантов
# попадания писем абитуриентам (крутим цикл)
> for (n in 1:10000) {
#  На каждом шаге выбираем 598 случайных абитуриентов
# из 4116 и подсчитываем сколько из них оказались в числе принятых
```

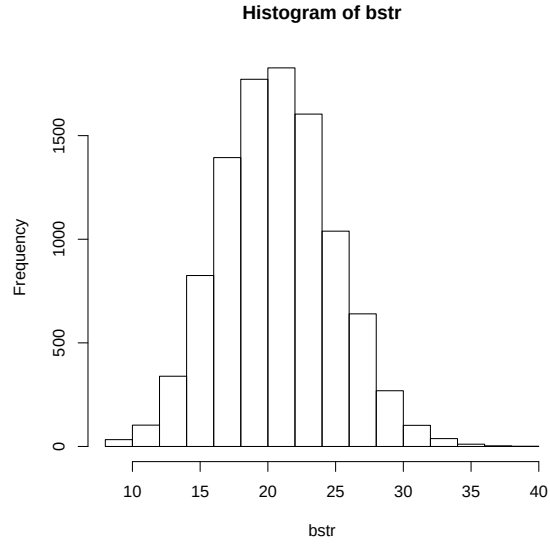

Рис. 8.1. Результат размножения выборок.

```
+  bstr[n] <- nrow(merge(
+    abitura[sample(1:4116, 598, replace= FALSE),], priem,
+    by.x=c("Фамилия","Имя","Отчество"),
+    by.y=c("Фамилия","Имя","Отчество")))
+ }
#  Строим гистограмму распределение числа поступивших
> hist(bstr)
```

Разберём что же было сделано чуть-чуть поподробнее. Команда sample отбирает случайным образом из таблицы данных `abitura` 598 случайных абитуриентов. Команда merge сливает полученную выборку и таблицу данных `priem` по уникальному ключу составленному из Фамилии, Имени и Отчества. При этом требуется наличие уникального ключа в обоих выборках. Команда nrow просто подсчитывает число совпавших в обоих списках (случайная выборка из всех потенциальных абитуриентов сдававших тест и список поступивших) абитуриентов.

Что-то похожее на результат На рисунке 8.1 представлена гистограмма, составленная по результатам 10000 попыток в ходе бутстрепа, в случае случайной рассылки 598 писем среди 4165 человек. Из рисунка очевидно, что вероятность получить среди зачисленных абитуриентов более 37 человек с таким письмом-приглашением составляет менее 0,1%. Напоминаем, что таких на самом деле было

61 человек. на этом простом модельном эксперименте мы показали, что рассылка работает. При выполнении вышеописанной операции делается масса допущений, которые наверняка искажают результаты анализа, но мы договорилось, что это делается «от полной безнадёги».

Уточнение задачи Однако, рассылать письма-приглашения всем без исключения абитуриентам, которые сдавали тест не очень осмысленно. Поэтому в качестве следующего шага изучим распределения плотности вероятности количества поступивших абитуриентов в случае рассылки приглашений с отбрасыванием всех абитуриентов, имеющих оценки ниже установленного минимума.

Дополнительные параметры Каждый из потенциальных абитуриентов сдавал тесты по физике и математике. В таблице данных `abitura` эти баллы нормируются на максимальный таким образом, что «волшебное число» абитуриентов 598 имеет балл и по физике и по математике больше 1, а минимальная оптимальная граница требований лежит в районе 0 (таких примерно четверть, а остальные три четверти, проходивших тестирование, имеют отрицательные результаты). То, как это сделано, выходит за рамки этой статьи.

Решение продолжается Оформляем функцию бутсреп-эксперимента с двумя параметрами характеризующими минимальные границы по физике и математике.

```
# Определяем функцию boot.fiz.mat.
#  fiz.min - граница отсева по физике,
#  mat.min - граница отсева по математике.
> boot.fiz.mat <- function (fiz.min, mat.min) {
#  Обнуляем вектор
+  bstr <- 1
#  Подсчитываем число потенциальных абитуриентов
# удовлетворяющих условиям отбора
+  nn <- nrow(abitura[abitura[,"физика"]> fiz.min &
+                     abitura[,"математика"]> mat.min,])
#  Крутим цикл
+  for (n in 1:10000) {
#  На каждом шаге выбираем 598 случайных абитуриентов
# из числа прошедших отбор и подсчитываем сколько
# из них оказались в числе принятых
+   bstr[n] <- nrow(merge(
+     abitura[abitura[,"физика"]> fiz.min &
+             abitura[,"математика"]> mat.min,]
+           [sample(1:nn, 598, replace= FALSE),], priem,
+     by.x=c("Фамилия","Имя","Отчество"),
```

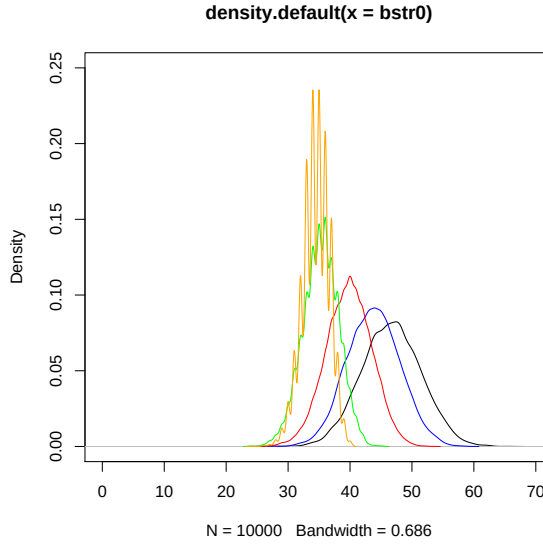

Рис. 8.2. Результат размножения выборок в зависимости от границы отсева.

```
+      by.y=c("Фамилия","Имя","Отчество")))
+  }
# Возвращаем результат
+ bstr}
```

Теперь проводим 5 бутстреп-экспериментов последовательно повышая границу отсева абитуриентов:

```
> bstr0 <- boot.fiz.mat(0, 0.)
> bstr02 <- boot.fiz.mat(0.2, 0.2)
> bstr04 <- boot.fiz.mat(0.4, 0.4)
> bstr06 <- boot.fiz.mat(0.6, 0.6)
> bstr08 <- boot.fiz.mat(0.8, 0.8)
```

и отображаем полученные результаты:

```
plot(density(bstr0),xlim=c(0,70),ylim=c(0,0.25),col="black")
lines(density(bstr02),col="blue")
lines(density(bstr04),col="red")
lines(density(bstr06),col="green")
lines(density(bstr08),col="orange")
```

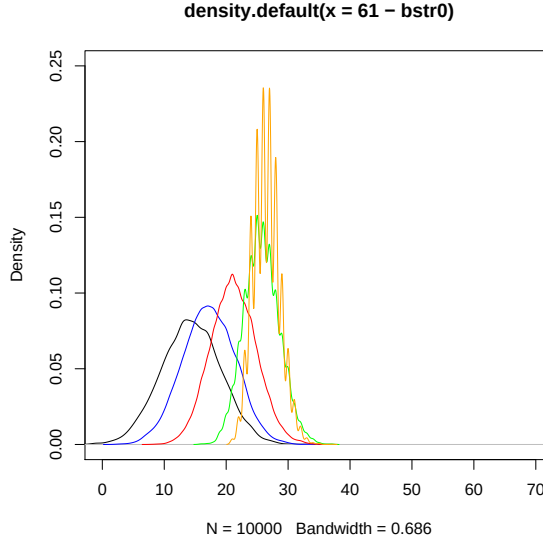

Рис. 8.3. Разница между фактическим принятыми получателями рассылки и результатами модельных экспериментов.

Следует отметить, что процедура не из быстрых. Даже на такой не сильно большой выборке на более-менее современном персональном компьютере счёт идёт на десятки минут.

Из рисунка 8.2 видно, что при повышении уровня требований от оптимальной границы среднее смещается влево. Это показывает, что процедура отсечения только баллам является не достаточно эффективной по сравнению с используемой при рассылке. Даже в самом удачном случае (чёрный график) вероятность получить в числе зачисленных 61 и более абитуриентов с письмом приглашением составляет менее 0,0022 (22 случая на 10000 попыток). Этот практически крайний случай задан оптимальной нижней границей требований. Все другие случаи отсечения абитуриентов по нижней границе результатов тестирования дают ещё меньшую вероятность получить на выходе 61 отклик. Но даже распределение с оптимальной нижней границей демонстрирует максимальную плотность вероятности в районе 46 человек. Это говорит о крайне высокой достоверности наличия эффекта рассылаемых писем-приглашений.

Попробуем теперь оценить эффект от рассылки количественно. Для этого немного изменим график, построив распределение разницы между числом фактически принятых во время вступительной кампании 2009 г. абитуриентов получивших приглашение и эффектом от случайной рассылки приглашений:

```
plot(density(61-bstr0),xlim=c(0,70),ylim=c(0,0.25),col="black")
lines(density(61-bstr02),col="blue")
lines(density(61-bstr04),col="red")
lines(density(61-bstr06),col="green")
lines(density(61-bstr08),col="orange")
```

Результат На рисунке 8.3 отражена картина, когда от реального результата приемной кампании (61 абитуриент с письмом-приглашением в числе 146 зачисленных) отнимаются гипотетические результаты, которые можно получить в результате случайной рассылки 598 приглашений всем, чьи результаты выше оптимальной границы требований. Видно, что разница между реальным результатом и гипотетически наиболее вероятного равна примерно 15. Это именно те пятнадцать человек, о которых можно сказать, что они пришли на факультет исключительно благодаря целенаправленной рассылке писем-приглашений.

Для пятнадцати человек письмо-приглашение сыграло роль последнего импульса к принятию решения. Естественно мы не имеем возможности определить эти 15 по фамилиям, но в эффекте целенаправленной рассылки мы в какой-то степени уверены. Зная сколько денег пошло на рассылку, теперь легко оценить сколько ресурсов было потрачено на одного качественного дополнительного абитуриента. В пересчёте с белорусских рублей на российские затраты в данном случае составляли около 250 рублей на одного хорошего человека. Много это или мало решать заказчику.

Да, то, что рассылка эффективна на самом деле было установлено опытным путём. В 2009 году факультет математики и информатики отказался от практики рассылки, а физико-технический факультет воспользовался этими наработками и впервые за многие годы получил достоверно более качественный набор (обычно на физику идут не так активно, как на математику по причине, что, как таковой, физики в школе гораздо меньше чем математики, не говоря уж об информатике), чем слишком поздно пожалевшие о своём неправильном шаге математики. Так что множество наукообразных слов, составляющих эту статью, на самом деле в какой-то степени верны. Экспериментальный факт.

Глава 9

Интерфейс для пользователя с мышкой (GUI на примере)

Про наличие графического пользовательского интерфейса в **R** уже упоминалось в более ранних статьях Linux Format. Теперь покажем как этот интерфейс можно использовать с пользой для дела.

Отдавать команды компьютеру можно исключительно из командной строки **R**, аналогично **R Commander** почти всё позволяет сделать мышкой. Очевидно, что часто требуется нечто среднее между этими двумя «мирами». Назовём этот средний путь «point-and-click» интерфейс. Особенно этот путь эффективен когда мы визуализируем обрабатываемые данные.

Анализ данных часто сводится к написанию своей специализированной программы. Если данных действительно много, то программирование для исследователя становится неизбежной необходимостью. Невозможно предусмотреть всё многообразие данных, пока их не исследуешь. Поэтому процесс анализа идёт параллельно с написанием всё более и более интеллектуальной системы этого самого анализа.

Построение динамического графического point-and-click интерфейса программы анализа должен быть прост, незатейлив и не отвлекать от самого главного, а именно от собственно анализа данных. Он должен быть тесно интегрирован с возможностями командной строки. В полной мере отвечает таким качествам пакет **rpanel**.

9.1. rpanel

Пакет **rpanel** — это крайне простая и одновременно высокоуровневая надстройка над Tcl/Tk расширением среды **R**. Tcl является наравне с Perl и Python «клас-

сическим» скриптовым языком высокого уровня, который обычно используют совместно с кроссплатформенной библиотекой графических элементов Tk. Многие языки программирования как и **R** используют Tk для построения простых графических интерфейсов.

Если пакет **rpanel** отсутствует в системе, то скачиваем и устанавливаем его с помощью команды

```
> install.packages("rpanel")
```

После этого загружаем саму библиотеку:

```
> library(rpanel)
```

Логика использования пакета довольно проста:

1) С помощью функции rp.control нужно создать объект `panel` и объявить в нём нужные нам переменных;

2) Далее необходимо «населить» объект `rpanel` различными элементами графического интерфейса (ползунками, кнопками, картинками и т. д.) и связанными с ними переменными;

3) В самом конце требуется определить функцию, которая будет выполняться в ответ на взаимодействие пользователя с графическим интерфейсом.

Разберём простейший пример, в котором функция выводит на экран одно из двух: или гистограмму полученных данных, или боксплот («ящик-с-усами») в зависимости от выбранного значения в элементе rp.radiogroup.

Объявим объект `panel`, и данные которые будем отображать:

```
> panel <- rp.control(x=rnorm(50))
```

Здесь мы «исследуем» вектор данных `x`, имеющих гауссовское распределение.

Теперь поместим в `panel` переключатель rp.radiogroup и свяжем его с переменной `plot.type` и с функцией hist.or.boxp:

```
> rp.radiogroup(panel,          # сам объект
+   plot.type,                  # переменная
+   c("histogram", "boxplot"),  # значения переключателя
+   title="Plot␣type",          # название переключателя
+   action = hist.or.boxp)      # пользовательская функция
```

В заключении объявим функцию которая будет выполнятся в ответ на взаимодействие с переключателем:

```
> hist.or.boxp <- function(panel) {
+   if (panel$plot.type == "histogram")
+     hist(panel$x)
```

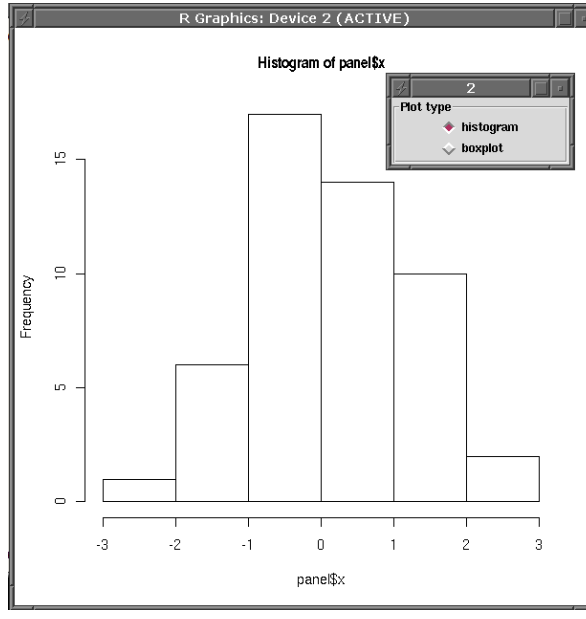

Рис. 9.1. Пример простого переключателя.

```
+   else
+     boxplot(panel$x)
+   panel
+ }
```

Доступ к переменным внутри объекта `panel` производится путём обращения вида `panel$имя_переменной`.

Теперь всё готово. У нас есть готовая панель с переключателем, который в зависимости от выбора показывает либо гистограмму, либо боксплот.

Если хочется, то можно «вмонтировать» график прямо в панель как один из элементов графического интерфейса. Для это понадобится воспользоваться командами rp.tkrplot и rp.tkrreplot из пакета **tkrplot**.

Для установки нового пакета необходимо убедиться, что установлены системные пакеты **tk-dev** и **tcl-dev**:

```
=> sudo aptitude install tk-dev tcl-dev
```

Затем, как обычно, устанавливаем и загружаем сам пакет:

```
> install.packages("tkrplot")
> library(tkrplot)
```

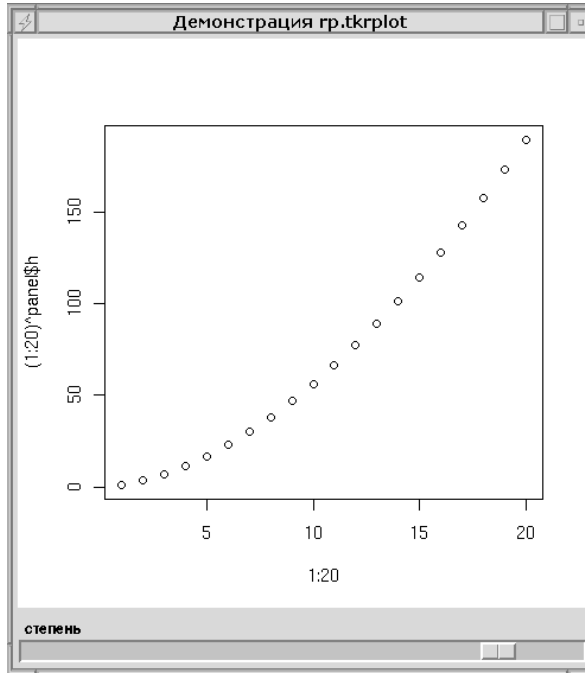

Рис. 9.2. Пример ползунка и графика как графического элемента.

Открываем панель `rpplot` и объявляем переменную `h`:

```
> rpplot <- rp.control(title = "Демонстрация rp.tkrplot",
+                      h = 1)
```

Помещаем на панели `rpplot` график `tkrp`:

```
> rp.tkrplot(rpplot,
+    tkrp,
+    function(panel) plot(1:20,(1:20)^panel$h))
```

В зависимости от параметра `h` на графике отображается степенная функция.

Определяем пользовательскую функцию для перерисовки графика:

```
> redraw <- function(panel) {
+  rp.tkrreplot(panel, tkrp)
+ }
```

Для динамически меняющегося параметра `h` в панель `rpplot` добавим ползунок и свяжем его с пользовательской функцией `redraw`:

```
# Помещаем на панель ползунок
> rp.slider(rpplot,           # панель
+    h,                       # переменная ползунка
+    action = redraw,         # пользовательская функция
+    from = 0.05, to = 2.00,  # параметры ползунка
+    resolution = 0.05,
+    title="степень")       # название ползунка
```

9.2. Tcl/Tk

Если возможностей **rpanel** не хватает, то можно спуститься уровнем ниже и обратиться напрямую к Tcl/Tk. Нужный пакет так и называется **tcltk**.

Тулкит Tk предоставляет богатый набор элементов интерфейса. К ним относятся окна редактирования текста, полосы прокрутки, поля ввода текста, кнопки, метки, перечни и рисунки. Всё это можно комбинировать и получать сложный графический интерфейс.

Простейший пример программы «Здравствуй, мир!»:

```
> library(tcltk)       # Загружаем пакет Tcl/Tk
> tt <- tktoplevel()   # Объявляем контейнер
# Объявляем метку
> lbl <- tklabel(tt,text="Здравствуй, мир!")
> tkpack(lbl)          # Размещаем метку в контейнере
```

В результате выполнения этого кода на экран будет выведено окно с популярной надписью «Здравствуй, мир!». Картинку этого шедевра программистского искусства специально не предоставляем, так как она достаточно банальна. Выполните пример самостоятельно — он не сложен.

Структура построения графического интерфейса всегда похожа: объявляется контейнер, и в нём с помощью менеджера геометрии (tkpack) размещаются дочерние элементы графического интерфейса.

В контейнер можно добавить сколько угодно графических элементов. Процесс построения сложного приложения как раз и заключается в комбинировании элементарных блоков. Например добавим в наше окно кнопку «Выполнить!»:

```
> but <- tkbutton(tt, text="Выполнить!")
> tkpack(but)
```

На кнопку можно нажимать, хотя никаких действий с ней не связано.

Окно контейнера в котором выведен наш пример называется «1» (значение по умолчанию). Изменим это:

```
> tktitle(tt) <- "Моё окно"
```

Рис. 9.3. 4 кнопки.

В Tcl/Tk доступно три менеджера геометрии. Простейший из них placer почти никогда не используется. Более популярны оставшиеся два: packer и grid. У packer есть два параметра: «порядок» и «направление».

В предыдущем примере, если попробовать изменять размеры окна мышью, то видно, что элементы графического интерфейса располагаются по центру окна у верхней его границы. Если начать уменьшать размер окна примера по вертикали, то увидим, что сначала сожмётся и исчезнет кнопка «Выполнить!», а потом начнёт изменяться надпись «Здравствуй, мир!». Попытки ручной коррекции размеров окна приводят к ещё одному эффекту: теперь окно не будет автоматически корректировать свои размеры при размещении в нем новых элементов. Восстановить автоматическую подгонку размеров окна можно выполнив команду:

```
> tkwm.geometry(tt,"")
```

Элементы графического интерфейса можно привязать к любой из сторон основного окна. Например, вот так (рис. 9.3):

```
> tkdestroy(tt)          # Убираем предыдущий пример
> tt <- tktoplevel()
# Английские названия используются и как имена кнопок,
#и как параметры
> edge <- c("top","right","bottom","left")
> for ( i in 1:4 )         # Четыре кнопки
+   tkpack(buttons[[i]], side=edge[i],
+          fill="both")
```

Элемент интерфейса при этом занимает достаточную для своего расположения полосу, параллельно стороне привязке. Указанный параметр `fill` заставляет элемент занять полосу целиком. Аналогично, если указать параметр `expand=TRUE`, то элементы интерфейса займут всё свободное место.

Когда размещаемый графический элемент интерфейса не занимает полосу целиком, то его можно «заякорить», подав «морские» команды вида «n» (Nord или Север или Верх основного окна) или «sw» (Зюйд-Вест или Внизу-слева).

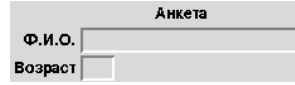

Рис. 9.4. Пример использования tkgrid.

Если нам к примеру понадобится разместить в основном окне область ввода текста с полосой прокрутки, то мы:

1) Добавляем полосе прокрутки свойство `fill="y"`;

2) Добавляем области ввода текста `fill="both"`и `expand=TRUE`, чтобы она заняла все свободное пространство;

3) Первой в основное окно размещаем полосу прокрутки, а потом область ввода текста, чтобы полоса прокрутки не сокращалась при изменении размеров окна пользователем.

Если же вы задумались как объединить ряд кнопок и область ввода текста, то нет ничего проще: надо всего лишь объединить кнопки во фрейм. Каждый фрейм — это контейнер со своим собственным менеджером геометрии. Гибкость в размещении графических элементов интерфейса достигается этими средствами весьма изрядная. Однако, иногда такой способ требует от пользователя изощрённых подходов. Например, достаточно трудно выровнять ряды кнопок одновременно и по вертикали и по горизонтали. В случае если вас замучила именно эта проблема, то вас спасёт менеджер геометрии grid. Он размещает графические элементы интерфейса стройными рядами и колоннами.

Следующий пример рекомендуется выполнить построчно, наблюдая за изменением вывода в основное окно (рис. 9.4):

```
> t2 <- tktoplevel()
> heading <- tklabel(t2, text="Анкета")
> l.name <- tklabel(t2, text="Ф.И.О.")
> l.age <- tklabel(t2, text="Возраст")
> e.name <- tkentry(t2, width=30)
> e.age <- tkentry(t2, width=3)
> tkgrid(heading, columnspan=2)
> tkgrid(l.name, e.name)
> tkgrid(l.age, e.age)
> tkgrid.configure(e.name, e.age, sticky="w")
> tkgrid.configure(l.name, l.age, sticky="e")
```

С таким достаточно приличным и доставшимся «малой кровью» графическим интерфейсом уже хочется общаться. А именно: получать данные от построенного интерфейса и отправлять в него результаты обработки. Для этого есть два основных способа «управляющие переменные» и «обратные вызовы» (callbacks).

Для создания Tcl управляющей переменной служит функция tclVar(), которая создаёт объект класса `tclVar`. На стороне Tcl созданные переменные будут поименованы автоматически, и в обычной ситуации беспокоится о задании какого-то специального имени переменной нет необходимости. Получить значение созданной Tcl переменной можно с помощью функции tclvalue().

Чтобы поместить содержимое в созданную переменную, не столкнувшись при этом с проблемами кавычек и других символов имеющих служебное значение в строках Tcl, следует задействовав функцию tclObj():

```
# Создаем переменную, в R имя foo
> foo <- tclVar()
# Заполняем foo символами
> tclObj(foo) <- c(pi,exp(1))
# Выводим значение foo
> tclvalue(foo)
[1] "3.141592653589793␣2.718281828459045"
# Разбираем строку на символьные значения
> as.character(tclObj(foo))
[1] "3.141592653589793" "2.718281828459045"
# Разбираем строку на вещественные значения
> as.double(tclObj(foo))
[1] 3.141593 2.718282
# Чего в строке нет, того нет
> as.integer(tclObj(foo))
[1] NA NA
# Округляем до целых самостоятельно.
> round(as.double(tclObj(foo)))
[1] 3 3
```

Доступ через tclObj() экономит значительные усилия. Например, так выглядит подготовка списка названий месяцев:

```
# Имена месяцев в представлении R
> month.name
 [1] "January"   "February"  "March"     "April"
 [5] "May"       "June"      "July"      "August"
 [9] "September" "October"   "November"  "December"
# Теперь строка пригодная для экспорта в Tcl
> paste(month.name, collapse="␣")
[1] "January␣February␣March␣April␣May␣June␣July
+␣August␣September␣October␣November␣December"
```

Заменяется достаточно простым выражением:

```
> mylist <- tclVar()
```

Рис. 9.5. Список месяцев.

Рис. 9.6. Список месяцев (множественный выбор).

```
> tclObj(mylist) <- month.name
> tclObj(mylist)
<Tcl> January February March April May June July
+ August September October November December
```

Теперь очень просто имея переменную Tcl общаться с созданным интерфейсом. Построим форму со списоком названий месяцев:

```
> tkpack(lb <- tklistbox(tt <- tktoplevel(),
+                listvariable = mylist))
```

В любой момент мы можем узнать какой месяц выбран в данный момент (рис. 9.5). При этом помним, что в Tcl индексы начинаются с 0:

```
> as.character(tclObj(mylist))[as.integer(tkcurselection(lb))+1]
[1] "July"
```

Возвращаемый из Tcl вызовов значения так же являются объектами `tclObj`. Например, мы можем включить в окне с нашим построенном списком месяцев режим множественного выбора:

```
> tkconfigure(lb, selectmode="multiple")
```

Выбрав нужные позиции списка (рис. 9.6) мы легко получаем их в виде номеров строк:

```
> as.integer(tkcurselection(lb))
[1] 1 3 5
```

Теперь попробуем привязать к кнопке какое либо действие, например пусть кнопка сообщает о своем нажатии в **R**, и меняя надпись:

```
# Создаём основное окно
> t <- tktoplevel()
# Создаем кнопку
> b <- tkbutton(t, text = "Не жми на меня!")
# Размещаем её в основном окне
> tkpack(b)
# Объявляем функцию кнопки
> change.text <- function() {
+   cat("ОГО!\n")
+   tkconfigure(b, text = "Говорю тебе: не жми на меня!")
+ }
# Прикрепляем функцию к кнопке
> tkconfigure(b, command = change.text)
```

Сейчас кнопка реагирует на нажатия, а именно, при каждом нажатии на кнопку в консоли появляются восклицания «ОГО!» и меняется текст на самой кнопке. Это простой пример — попробуйте его в динамике.

Кнопку для краткости можно сразу объявить с нужной нам функцией:

```
> b <- tkbutton(t, text="Не жми на меня!",
+   command=expression(cat("ОГО!\n"),
+   tkconfigure(b, text = "Говорю тебе: не жми на меня!")))
```

Каждый из элементов графического интерфейса Tk имеет массу полезных применений. Например, окно для ввода текста представляет из себя практически готовый текстовый редактор. В нём можно вводить и редактировать введенный текст, перемещаться по тексту клавишами курсора, выделять куски текста для операций копирования и вставки.

Что бы получить текст из окна надо выполнить команду

```
> tkget(txt, "0.0", "end")
```

Поскольку Tcl имеет дело со строками, то и передаём мы только строки. «Первая строка, первый символ» — это `0.0`, а `end` — это, соответственно, конец текста. Если мы захотим получить текст в **R** также разбитым на строки, как в текстовом окне, то нам надо конвертировать Tcl строку в вектор состоящий из строк:

```
> strsplit(tkget(txt, "0.0", "end"),"\n")
```

Похожим способом мы можем получить и выделение текста в окне:

```
> X <- tkget(txt, "sel.first", "sel.last")
```

Здесь нам уже не обойтись без проверки ошибок, а то вдруг выделение текста не было сделано? Проверить это просто, путём сравнения с пустой строкой:

```
> tktag.ranges(txt, "sel") != ""
```

Можно получить информацию и о нажатии кнопки мыши, например, получить координаты её курсора и не переместить при этом текстовый курсор:

```
> tkbind(txt, "<Control-Button-1>",
+   expression(function(x,y) cat(x,y,"\n"),
+                break))
```

Добавить текст, к примеру, в конец текста расположенного в окне:

```
> tkinsert(txt, "end",    # Добавляем в конец текста
          string)          # Строка которую добавляем
```

Естественно, что если мы хотим передать в Tcl несколько строк сразу, то нам понадобится объединить их, например вот таким способом:

```
> tkinsert(txt, "end",
          paste(deparse(ls), collapse="\n"))
```

Если нам надо переместить текстовый курсор в определенное место окна, то мы можем воспользоваться сочетанием:

```
# Переместить курсор в начало первой строки
> tkmark.set(txt, "insert", "0.0")
# Сделать видимой область с курсором
> tksee(txt, "insert")
```

Мы также можем контролировать куда будет смещаться вставляемый текст, если ему не хватает места. По умолчанию он вставляется перед выделением, но это легко изменить:

```
> tkmark.gravity(txt, "insert", "left")
```

Если наше приложение достаточно сложное, то ему безусловно понадобится меню. Меню у нас тоже есть — это отдельный элемент. Можно использовать всплывающее меню, но чаще меню размещают в основном окне в качестве полоски, или кнопки «Пуск», или как меню вложенное внутри главного меню приложения.

Наиболее практично меню создавать по шагам. Сначала нужно объявить меню командой tkmenu, затем наполняем его элементами меню с помощью tkadd. Это позволит легко воспользоваться многочисленными опциями команд создания элементов меню.

Существует множество разновидностей элементов меню. Это эквивалентные уже знакомым нам кнопкам `command entry`. Вложенное меню получают вставляя `Cascade entries`. `Checkbutton` и `radiobutton` позволяют делать различные

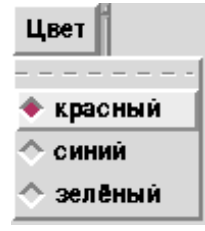

Рис. 9.7. Пример меню.

варианты выбора параметров. Есть и доступ к элементам организующим содержимое меню: это `separators`, отделяющие элементы меню визуально друг от друга, и `tear-offs`, позволяющие открeплять меню от родительского и делать его плавающим.

Чтобы не быть голословными создадим кнопку меню с тремя `radiobutton` (рис. 9.7):

```
# Объявляем переменную изменяемую из меню
> color <- tclVar
#и её значение по умолчанию
> color<-"красный"
> tt <- tktoplevel()
# Размещаем кнопку меню
> tkpack(mb <- tkmenubutton(tt, text="Цвет"))
# Объявляем вложенное меню
> m <- tkmenu(mb)
# Присоединяем вложенное меню к кнопке
> tkconfigure(mb,menu=m)
# Создаём элементы меню
> for ( i in c("красный", "синий", "зелёный"))
   tkadd(m, "radio", label=i, variable=color, value=i)
```

Опросить что же выбрано пользователем из вариантов меню, можно обратившись к функции

```
> tclvalue(color)
[1] "красный"
```

Вот теперь мы во всеоружии, то есть можем не просто наслаждаться сами высокопродуктивной обработкой данных в **R**, но и сделать свои наработки доступными для не столь искушённых в **R** окружающих.

К сожалению нет в мире совершенства. Вдруг нам понадобится сделать доступным наше предложение через корпоративный вебсервер? Или нам не нравится

вид Tk? Если Вы еще не привыкли к тому что в **R** как в Греции есть всё, то вот оно решение: **gWidgets**.

Вместо того, что бы опускаться вглубь конкретной библиотеки графического интерфейса, можно просто сделать свое приложение библиотеко-независимым. Но это тема для отдельного рассказа.

Глава 10

Высокопроизводительные вычисления

Компьютеры становятся быстрее, места на диске всё больше, но темп с которым растёт объём данных и постоянно повышающаяся сложность их обработки всё равно *заставляет* думать как ускорить вычисления.

Традиционно для сложных вычислений используются кластеры. Многие слышали про список TOP500, но сейчас пора осознать что кластерные технологии потихоньку проникают и на обычные, можно сказать домашние, компьютеры. Почти у каждого процессора сейчас по несколько ядер. Так почему бы этим не воспользоваться? Но для начала постараемся понять как оценить выигрыш от подобных приёмов.

10.1. Анализ эффективности программы

Измерить скорость вычислений в **R** и, соответственно, оценить эффективность написанного кода можно несколькими способами:

- Использовать system.time() для простых измерений;
- Использовать Rprof() для профилирования написанного кода;
- Использовать Rprofmem() для профилирования использования памяти.

Для визуализации накопленных с помощью Rprof() данных можно использовать пакеты **profr** и **proftools**.

Применим средства профилирования к простой задаче. Пусть в ходе моделирования нам многократно требуется находить параметры линейной регрессии,

для чего воспользуемся функцией lm(). Единственный её недостаток заключается в том, что она делает много лишнего. Она просто слишком универсальна. Для простой линейной регрессии вполне достаточно воспользоваться уточнённым вызовом lm.fit().

Насколько же эффективней окажется прямой вызов? Попробуем с помощью обоих функций обработать набор макроэкономических показателей `longley`. Будем строить зависимость между числом работающих и остальными показателями. Для оценки проделав по 1000 вычислений за раз.

```
# Загружаем данные
> data(longley)
# Записываем профиль в файл lm.out
> Rprof("lm.out")
# Выполняем lm() 1000 раз
> invisible(replicate(1000,lm(Employed ~ .-1, data=longley)))
# Отключаем профилирование
> Rprof(NULL)
# Готовим данные для lm.fit()
> longleydm <- data.matrix(data.frame(longley))
# Записываем профиль в файл lm.fit.out
> Rprof("lm.fit.out")
# Выполняем lm.fit() 1000 раз
> invisible(replicate(1000,lm.fit(longleydm[,-7],longleydm[,7])))
# Отключаем профилирование
> Rprof(NULL)
```

Записанные в файлах данные профилирования можно проанализировать с помощью встроенной команды summaryRprof(). Например, для того чтобы выяснить время работы программы достаточно обратиться к переменной `sampling.time`:

```
> summaryRprof("lm.out")$sampling.time
[1] 6.42
> summaryRprof("lm.fit.out")$sampling.time
[1] 0.44
```

Данные профилирования можно отобразить и графически, с помощью пакета **profr**:

```
# Устанавливаем пакет (если нужно)
> install.packages("profr")}
> library("profr")
> plot(parse_rprof("lm.out"),main="Profile␣of␣lm()")
> plot(parse_rprof("lm.fit.out"),main="Profile␣of␣lm.fit()")
```

Из рис. 10.1 и 10.2 видно сколько времени проводит программа в каждой из функций. Неудивительно что функция lm.fit() работает в 14 раз быстрее.

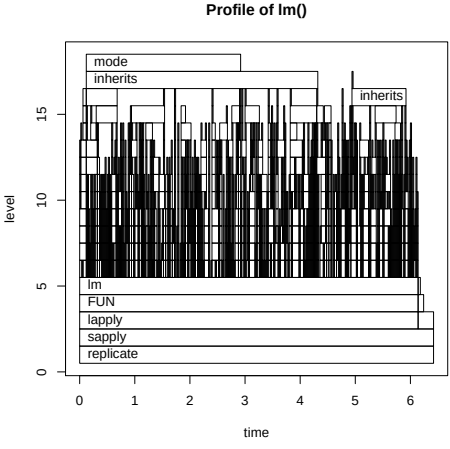

Рис. 10.1. Профилирование lm().

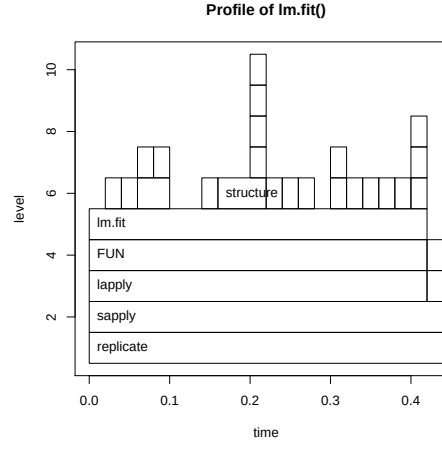

Рис. 10.2. Профилирование lm.fit().

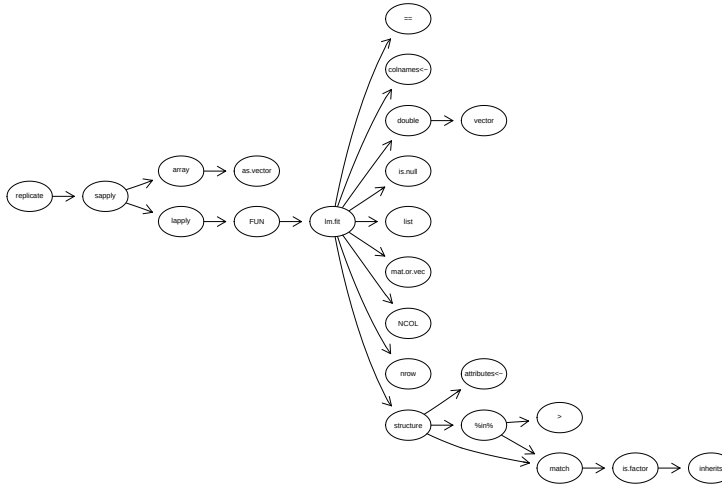

Рис. 10.3. Граф вызовов, полученный с помощью пакета **proftools**.

Для представления зависимостей вызовов в виде графа можно воспользоваться пакетом **proftools**. Для этого необходимо установить системный пакет **graphviz-dev**

```
=> aptitude install graphviz-dev
```

пакет **Rgraphviz** и сам **proftools**

```
> source("http://bioconductor.org/biocLite.R")
> biocLite("Rgraphviz")
> install.packages("proftools")}
```

Для отображения графа (рис. 10.3) необходимо выполнить следующие команды:

```
library("Rgraphviz")
library("proftools")
lmfitprod <- readProfileData("lm.fit.out")
plotProfileCallGraph(lmfitprod)
```

Для отладки потребления памяти нужно использовать специально модифицированную версию **R**, которая собрана с опцией `enable-memory-profiling`. При этом для анализа используется команда Rprofmem аналогично использованию Rprof.

Для контроля за избыточным потреблением памяти можно воспользоваться функцией tracemem. Она срабатывает каждый раз когда происходит копирование какого либо объекта.

10.2. Встроенные функции — ключ к ускорению

Достижение максимальной скорости во многом вопрос знания встроенных функций **R**. Их появление, как правило, обусловлено тем, что выполняемые ими функции встречаются в обработке данных часто. Подобный подход достаточно часто существенно сокращает усилия по вводу кода программы, а также повышает скорость выполнения кода, за счёт используемых во встроенных функциях оптимальных алгоритмов.

Очень часто встроенные функции реализованы на более низкоуровневом языке, который может предоставлять более широкий доступ к в возможностям оборудования. Поясним это небольшим примером. Для этого напишем программу складывающую числа от 1 до указанного с помощью обычного цикла, и сравним это с вариантом программы использующей встроенные функции **R**.

```
> mysum <- function(N) { a <- 0;
+    for (i in 1:N) a <- a + i;
+    return(a) }
> system.time(mysum(10000000))
```

```
   user  system elapsed
  7.456   0.024   7.482

> system.time(sum(as.numeric(seq(1,10000000))))
   user  system elapsed
  0.052   0.060   0.112
```

Разница очевидна: второй вариант подсчёта суммы работает более чем в 70 раз быстрее! То что сейчас было проделано носит название «векторизация». На такой способ расчётов трудно переключиться после императивного стиля программирования, но именно в векторизации заключается необычайно высокая продуктивность работы в **R**. Уменьшилось не только время выполнения программы, но и уменьшился её размер!

Возьмем простую тестовую задачу: «Найти распределение детерминанта матрицы 2x2 в которую занесены независимо и случайно изменяющиеся значения. Допустимые значения 0, 1, . . . , 9.» Она эквивалентна задаче найти все сочетания $ab - cd$, где a, b, c, d — это цифры.

Наивное императивное решение «с циклами» выглядит привычно и даже благодаря синтаксису **R** довольно «компактно»:

```
> dd.for.0 <- function()
+  {
+    val <- NULL
+    for (a in 0:9) for (b in 0:9) for (d in 0:9) for (e in 0:9)
+      val <- c(val, a*b - d*e)
+    table(val)
+  }
> system.time(dd.for.0())
   user  system elapsed
  0.196   0.000   0.195
```

Время от запуска к запуску слегка меняется и так как в подобных случаях распределение ненормальное, то лучше всего находить медиану нескольких попыток:

```
median(replicate(20, system.time(dd.for.0())["elapsed"]))
[1] 0.177
```

Попробуем добится от этого наивного варианта большего, например выделим память под расчёты сразу в начале программы:

```
> dd.for.1 <- function()
+  {
+    val <- double(10000)    # преалоцируем val
+    nval <- 0
```

```
+    for (a in 0:9) for (b in 0:9) for (d in 0:9) for (e in 0:9)
+      val[nval <- nval + 1] <- a*b - d*e
+    table(val)
+  }

> median(replicate(20, system.time(dd.for.1())["elapsed"]))
[1] 0.059
```

Улучшение очевидно: код ускорился более чем в три раза. Поскольку наши данные целые числа, то посмотрим что даст использование встроенной функции tabulate():

```
> dd.for.3 <- function()
+  {
+    val <- double(10000)
+    nval <- 0
+    for (a in 0:9) for (b in 0:9) for (d in 0:9) for (e in 0:9)
+      val[nval <- nval + 1] <- a*b - d*e
+    tabulate(val)
+  }
> median(replicate(20, system.time(dd.for.3())["elapsed"]))
[1] 0.057
```

Чуть-чуть улучшилось, но это все полумеры, делаем решительный шаг и вспоминаем что прародителем **R** был и язык APL. Запишем решение задачи как операцию над массивами:

```
> dd.fast <- function()
+  {
+    val <- outer(0:9, 0:9, "*")
+    val <- outer(val, val, "-")
+    tabulate(val)
+  }

> median(replicate(20, system.time(dd.fast())["elapsed"]))
[1] 0.001
```

Лучшее решение использующее циклы обойдено более чем в 50 раз, а «традиционное» почти в 200!

Что же делать если от цикла нельзя избавится? Есть вариант использовать среду **R** в которую встроена возможность компиляции кода.

Вариант сборки среды **R** с возможностью jit (just-in-time compilation) позволяет рассчитывать на ускорение кода содержащего циклы примерно в полтора раза.

Операции с матрицами являются в **R** такими производительными, потому что они опираются на процедуры библиотеки blass («basic linear algebra subprogram»). **R** может быть скомпилирована с различными вариантами реализации Blas: это и свободная библиотека Atlas (пакет **atlas3-base**), и платная Goto, и библиотеки от двух основных производителей процессоров Intel и AMD. Библиотеки не только имеют более производительный код, но и автоматически задействуют в вычислениях все имеющиеся ядра процессора персонального компьютера. Дополнительный прирост производительности можно получить настроив Atlas под конкретно используемый в расчётах персональный компьютер.

Независимо от наличия библиотеки BLAS можно поэкспериментировать с экспериментальным пакетом **pnmath0** от Люка Тьерни (Luke Tierney). Пакет можно найти по адресу `http://www.stat.uiowa.edu/~luke/R/experimental/`. Пакет заменят реализацию встроенных векторных функций **R** на параллельные варианты, используя Pthreads. Пока эта возможность не встроена в **R** и её придётся установить самостоятельно. Следует отметить, что параллельные вычисления будут активироваться только при достаточно длине векторов аргументов.

Если на компьютере установлена видеокарта которая поддерживает вычисления на своем GPU (Пока только CUDA и CUBLAS), то при установке пакета **gputools**, появляется возможность выполнять с очень высокой скоростью иерархический кластерный анализ, классификацию с обучением (По алгоритму SVM) и расчёт коэффициентов корреляции.

Несмотря на использование высокопроизводительных векторных операций и компиляции в режиме just-in-time, бывают моменты когда на счету каждый такт процессора. В этом случае есть два механизма оперативно встроить в расчет выполненный в среде **R** низкоуровневый код императивного языка программирования:

- Для простой вставки небольшого фрагмента кода **inline**;
- **Rcpp** — для облегчённого процесса интеграции сложного кода на C++.

Пакет **inline** предоставляет функцию cfunction(), умеющую автоматически встраивать код написанный на Fortran, C, C++. Для выполнения следующего простого примера на Fortran, естественно необходимо установить и загрузить сам **inline**:

```
# Не забудьте про отступы! Fortran такой Fortran.
> code <- "
+␣␣␣␣␣␣␣do␣i␣=␣1,␣n(1)
+␣␣␣␣␣␣␣␣x(i)␣=␣x(i)**3
+␣␣␣␣␣␣␣end␣do"
> cubefn <- cfunction(signature(n="integer", x="numeric"),
+                     code, convention=".Fortran")
> x <- as.numeric (1:10)
> n <- as.integer(10)
> cubefn(n,x)$x
```

```
[1]    1    8   27   64  125  216  343  512  729 1000
```

10.3. Параллелельные вычисления

Среда **R** работающая в 64х битном окружении практически не имеет ограничений по объёму обрабатываемых данных. Современным ответом в области вычисления с гигантскими объёмами данных является пакет **iterators** от REvolution Computing. В комплекте с возможностью поэлементно обработать структуру не помещающуюся в памяти крайне удачно сочетается второй пакет **foreach**. Данный пакет вводит возможность циклически обработать созданный итератор и вернуть суммарный результат. Отсутствие побочных эффектов позволяет выполнить оптимизируемую операцию параллельно.

Отсутствие побочных эффектов это именно то, что позволяет не заботится где выполняется та, или иная часть кода. Компьютеры не только стали мощнее и у них больше ядер. Компьютеров прежде всего стало *много* больше. Под рукой практически у каждого имеются 5–10 машин, многочисленные ЦПУ которых если посмотреть пристально никогда не загружены даже на 10% от своей возможности. Вся эта мощь доступна пользователю когда он использует **R**.

Кластерные вычисления настолько естественные для векторных операций **R**, что существует несколько способов их реализации в этой среде:

- **Rmpi** — это Message Passing Interface, является стандартом в области паралельных вычислений;
- **NWS** — это написанная на Python альтернативная реализация MPI;
- **snow** — высокоуровневая надстройка над MPI, PVM, NWS, sockets;
- **papply** — параллелизация функции apply через MPI;
- **multicore** — параллельные вычисления на многоядерных машинах.

Поскольку наша цель — это быстро и «глобально» задействовать мощь окружающих нас фактически простаивающих компьютеров для нашей же пользы, то сразу же воспользуемся высокоуровневым средством, а именно пакетом **snow**.

А что бы рассказ не был пустым теоретизированием попробуем решить реальную практическую задачу:

Задача Число телефонных пар, которые проходят рядом, и передают сигнал без помех ограничено. Безусловно влияет и длинна кабеля, и его ёмкость, и диаметр жил. Однако в номограммах оценки характеристик телефонной пары не предусмотрено свыше 25 ADSL пар в одном кабеле.

Известны нормативные документы российских провайдеров ADSL определяющме емкость одиночного кабеля в 18%. Скорее всего для кабелей небольшой

ёмкости всё же верен предел в 18 жил в одном кабеле занятых одновременно работающими ADSL модемами. А для кабелей более 100 пар ёмкости верно процентное ограничение.

В городе Нске например ёмкость ADSL сети Нсктелеком в этом году превысила 10 тыс. абонентов. Попробуем оценить какие проблемы встретит сеть при своем развитии.

Модель Город 300 тыс населения, в средней семье 3 человека определяет следующие исходные условия:

```
# кол-во абонентов услуги
> n.abonentov <- 10000
# ко-во домов
> n.domov <- 1000
# кол-во квартир в доме
> n.kvartir <- 100
# критическое число абонентов для ADSL в одном доме
> n.kritic <- 18
```

Получаем модель города в виде вектора, каждый элемент которого квартира помеченная номером дома

```
> gorod <- rep(1:n.domov, each = n.kvartir)
```

Поскольку вычисления ресурсоёмкие загрузим сразу библиотеку для паралельных вычислений основных функций пакета (предполагается, что эта экспериментальная библиотека уже установлена):

```
> library("pnmath0")
```

Делаем выборку случайных `n.abonentov` в векторе `gorod`.

```
> vyborka <- as.factor(gorod)[sample(1:length(gorod),
+             n.abonentov, replace= FALSE)]
```

Подсчитываем сколько в каждом доме попало абонентов, и строим гистограмму

```
> hist(as.numeric(tapply(rep(1,n.abonentov), vyborka, sum)))
```

Сколько домов испытывают трудности

```
> length(as.numeric(tapply(rep(1,n.abonentov),
+          vyborka, sum))[as.numeric(tapply(rep(1,n.abonentov),
+                           vyborka, sum))>n.kritic])
[1] 4
```

Сколько абонентов испытывают трудности

```
> sum(as.numeric(tapply(rep(1,n.abonentov),
+        vyborka, sum))[as.numeric(tapply(rep(1,n.abonentov),
+                    vyborka, sum))>n.kritic])
[1] 81
```

Это только одна реализация. Попробуем построить бутстреп процедуру и оценить какова доля домов в которых будет свыше `n.kritic` абонентов. Понадобится сделать не менее 10000 тысяч вычислительных экспериментов:

```
> nn <- 10000
> bstr.dom <- numeric(nn)
> bstr.abonent <- numeric(nn)
>  for (n in 1:nn) {
+    vyborka <- as.factor(gorod)[sample(1:length(gorod),
                 n.abonentov, replace= FALSE)]
+    rr <- as.numeric(tapply(rep(1,n.abonentov),vyborka , sum))
+    bstr.abonent[n] <- sum(rr[rr>n.kritic])
+    bstr.dom[n] <- length(rr[rr>n.kritic])
+  }
# Проблемные абоненты
> hist(bstr.abonent)
# Проблемные дома
> hist(bstr.dom)
```

Проведём эксперимент с подсчётом: сколько процентов абонентов и какое колво домов окажется с плохим качеством услуги при росте абонентской базы от 10000 до 20000 абонентов

Оформим функцию:

```
> my.boot.adsl <- function (n.abonentov) {
+  nn <- 10000
+  bstr.dom <- numeric(nn)
+  bstr.abonent <- numeric(nn)
+  for (n in 1:nn) {
+    vyborka <- as.factor(gorod)[sample(1:length(gorod),
+                      n.abonentov, replace= FALSE)]
+    rr <- as.numeric(tapply(rep(1,n.abonentov),vyborka , sum))
+    bstr.abonent[n] <- sum(rr[rr>n.kritic])
+    bstr.dom[n] <- length(rr[rr>n.kritic])
+  }
+  return(abonent=bstr.abonent, dom=bstr.dom)
+}
```

Рассчитаем оценку числа проблемных абонентов в диапазона от 10000 до 20000 размере абонентской базы:

```
> xx <- c(NA)
> for (n in 1:10) {
+  xx[n] <- my.boot.adsl(10000+(n*1000))
+}
```

Нормируем на размер абонентской базы

```
> xx.norm <- xx
> xx.norm[[1]] <- xx[[1]]/11000
> for (n in 2:10) {
>  xx.norm[[n]] <- xx[[n]]/(10000+(n*1000))
> }
```

Отобразим в виде боксплота

```
> boxplot(xx.norm,names=seq(11000,20000,by = 1000))
```

Данный модельный город предполагал наличие в городе только 100 квартирных домов. Повторим вычисления на данных о реальном количестве квартир в городе Нске.

Реальность Ввиду ресурсоёмкости вычислений напишем параллельную версию программы. Для этого загружаем библиотеки кластера

```
> library(snow)
```

Создаем кластер из двух узлов

```
 > cl <- makeCluster(c("localhost","localhost"), type = "SOCK")
```

Критическое число абонентов для ADSL в одном доме

```
> n.kritic <- 18
> clusterExport(cl, "n.kritic")
```

Получаем модель города в виде вектора, каждый элемент которого квартира помеченная номером дома. Обработаем реальные дома города Гродно. Загрузим список максимальных номеров квартир в доме

```
> Nsk <- na.omit(read.table("data_Nsk.txt"))
> Nsk.sort <- sort(t(Nsk))
```

Файл `data_Nsk.txt` — просто колонка числе, которую можно загрузить, например, тут: `http://www.inp.nsk.su/~baldin/data_Nsk.txt`.

Создадим модель города

```
> gorod <- unlist(mapply(rep,
+       1:length(Nsk.sort),Nsk.sort))
> clusterExport(cl, "gorod")
```

Отведём место под результаты оценки числа проблемных домов и числа проблемных абонентов. Для уменьшения накладных расходов при выделение памяти.

```
> nn <- 10000
> bstr.dom <- numeric(nn)
> clusterExport(cl,"bstr.dom")
> bstr.abonent <- numeric(nn)
> clusterExport(cl,"bstr.abonent")
```

Функция расчёта числа проблемных абонентов при случайном распределении

```
> my.boot.func <- function (n, n.abonentov) {
+  vyborka <- as.factor(gorod)[sample(1:length(gorod),
+                    n.abonentov, replace= FALSE)]
+  rr <- as.numeric(tapply(rep(1,n.abonentov),vyborka , sum))
+  bstr.abonent[n] <- sum(na.omit(rr[rr>n.kritic]))
+}
```

Рассчитаем оценку числа проблемных абонентов в диапазона от 3000 до 32000 размера абонентской базы

```
> xx <- cbind(sapply(1000+((1:10)*3000),
+                 function (n.abonentov) parSapply(cl,
+                      c(1:nn), my.boot.func, n.abonentov )))
```

Нормируем на размер абонентской базы

```
> xx.norm <- xx
> xx.norm[,1] <- xx[,1]/4000
> for (n in 2:10) {
+  xx.norm[,n] <- xx[,n]/(1000+(n*3000))
+ }
```

Отобразим в виде боксплота

```
> boxplot(xx.norm, names=seq(4000,31000,by = 3000))
```

Остановим кластер

```
> stopCluster(cl)
```

А в чём выигрыш? Приведённые выше вычисления тестировались на процессоре семейства Intel Core Duo модель T2050 с частотой 1.60 ГГц. При использовании распараллеливания время вычисления составляло 6470 секунд, а без оного 12754 секунд. Иными словами два ядра примерно в два раза лучше чем одно. **ЧТД**.

Настоящая реальность В соответствии с нормативными данными в номограммах для расчета помехозащищенности вообще не предусмотрен случай, когда число задействованных пар в одном кабеле превышает 25%. Это хорошо согласуется с тем, что работает даже в час пик только одна пятая часть всех абонентов. Иными словами полностью заполненный абонентами 100 парный кабель будет все таки работоспособен.

Однако есть два «но»:

1) реальный постсоветский кабель не держит более 18 нагруженных ADSL пар (в интернете есть нормативные документы российских провайдеров);

2) развитие IP – TV приводит абонентов с совсем другим подходом к использованию услуги. Ни о какой доле активных абонентов в одну пятую речи не идет. В утренние и вечерние часы телевизор смотрят 90%. Причем поток достигает максимальных скоростей и соответственно создаёт максимальные помехи для соседних в кабеле пар. И кол-во абонентов в самом лучшем случае не превысит 25%, даже при переходе к ADSL-2 (оптика на группу домов).

Получается что на настоящий день достигнув по агентурным данным размера абонентской базы в 15 тыс квартир, реальная компания Белтелеком в городе Гродно имеет долю клиентов имеющих периодически принципиально не устранимые проблемы в 30-38%. Причём каждый подключенный абонент ip-tv «съедает» ресурс сети как 5 абонентов передачи данных.

Глава 11

Поиск зависимостей

R можно и нужно применять в реальных исследованиях. Грамотный исследователь с помощью своего интеллекта, знаний статистики и возможностей, предоставляемых **R**, с лёгкостью может ответить на массу загадок мироздания. Если есть данные, естественно.

11.1. Кто оценит преподавателя?

Студенты знают, что за ними постоянно следят и их ценят, а каждый экзамен становится праздником на которых их оценивают. С этим всё в порядке. Но кто оценит преподавателей? Говорят анкетирование поможет. Проверим это.

Предыстория Давным давно в одной «далёкой галактике» провели анкетирование на тему «Преподаватель глазами студентов» и благополучно об этом забыли. После того как с анкет была сдута пыль веков, выяснилось, что данные опроса представлены таблицами. Каждая строка опросной таблицы состоит из средней по группе студентов оценки по соответствующей графе анкеты. Строка также характеризуется курсом, предметом, преподавателем и кафедрой. Формат записей CSV, то есть что-то вроде:

```
Кафедра истории и философии;Галактическая история;Галактион Иванович
Планетный;5-й курс;4,6;4,7;4,3;4,8;4,6;3.6;2,7;...
```

Поиск зависимостей Импортируем и объединяем таблицы:

```
> data <- rbind(read.csv2("анкета1.csv"),
+               read.csv2("анкета2.csv"),
```

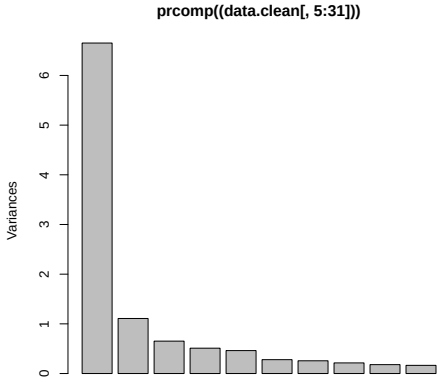

Рис. 11.1. График нагрузок основных компонент

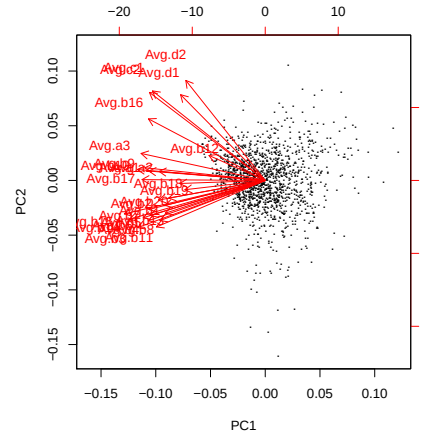

Рис. 11.2. Двойная диаграмма (biplot)

```
+                 ...
+                 read.csv2("анкетаN.csv"))
```

Увы, в реальной жизни никуда не деться от пропущенных значений, поэтому очищаем данные опросов от них:

```
> data.clean <- na.exclude(data)
```

Воспользуемся методом основных компонент, чтобы уменьшить размерность анализируемых данных. Строим график нагрузок основных компонент с помощью команды prcomp. В расчёте участвуют только ответы на вопросы анкеты:

```
> plot(prcomp((data.clean[,5:31])))
```

Присутствует простая структура из двух факторов. Довольно странно, что ответ на анкету из 26 вопросов у студента происходит, исходя всего из *двух* взаимонезависимых факторов. Иными словами студенты отвечали на все многочисленные вопросы анкеты фактически реально исходя всего лишь из двух причин.

Посмотрим на получившуюся простую структуру с целью идентификации двух выделенных в анализе факторов:

```
> biplot(prcomp(data.clean[,5:31]),
+        xlabs= rep( ".", length(data.clean[,3])))
```

Выделенные две взаимно независимые компоненты, путем сопоставления преподавателей участвующих в оценке, с местом которое они заняли в пространстве факторов можно идентифицировать. Первая компонента отражает ось «Было

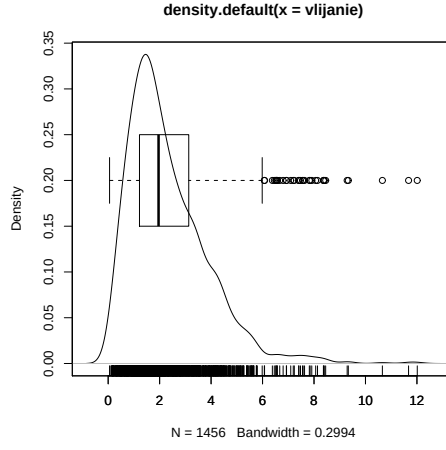

Рис. 11.3. Анализ анкеты

сложно на предмете — Было просто». Вторая компонента отражает ось «Добрый преподаватель — Строгий преподаватель». По соображениям этикета подробности этого процесса мы вынужденно пропускаем.

У составителя анкеты, да и у оцениваемого преподавателя, естественно возникает философский вопрос: «Что лучше? Быть добрым или злым? И что делать если тебе досталась сложная для студентов учебная программа?». Дело в том, что любое простое суммирование баллов набранных преподавателями, с целью получить интегральную оценку, будет эквивалентно проведению под определенным углом через построенное нами факторное пространство оси измерения отражающего данную интегральную оценку.

На самом деле мера определяющая качество взаимодействия в системе «Студент — Преподаватель» имеется. Задумаемся над вопросом: «Кто из преподавателей получил оценки вблизи начала координат факторной плоскости?». Кто всегда получает среднюю оценку? Очевидный ответ: тот, о котором ничего особенного *не помнят*!

Если преподаватель взаимодействовал со студентом достаточно сильно в процессе обучения, то он оставил в его сознании чёткий след. Этот след может отражать и доброту, и строгость, и сложность, и простоту ..., а также их любое сочетание! Иными словами сила взаимодействия в системе «Преподаватель — Студент» — это длина вектора проведенного из начала координат к точке на факторной плоскости соответствующей анкете оцениваемого преподавателя.

Нам остается воспользоваться теоремой великого Пифагора

```
> result.pca <-prcomp((data.clean[,6:32]))$x[,1:2]
> vlijanie <- (result.pca[,1]^2 + result.pca[,2]^2)^0.5
```

Посмотрим каково распределение полученной величины.

```
> plot(density(vlijanie))  # рисуем плотность распределения данных
> rug(jitter(vlijanie))    # добавляем по оси x штрихи значений данных
> boxplot(vlijanie,
+         add= TRUE,         # добавляем на тот же график
+         horizontal=TRUE,  # располагаем boxplot по горизонтали
+         at= 0.2,           # позиция срединной линии boxplot
+         boxwex = 0.2)      # масштаб ширины boxplot
```

Оцененная сила воздействия преподавателя на студентов неоднозначная величина. Безусловно плохо когда преподавателя студенты «не помнят», но что с противоположным случаем? Может ли быть воздействие слишком сильным? Очевидно по аналогии с другими рецепторами организма любое слишком сильное воздействие может вести к повреждению и уж точно ощущается воспринимающим как очевидный дискомфорт.

«Золотая середина» получилась действительно в середине. И это породило определённые трудности. Выделить «лучших», как обычно стремятся организаторы таких опросов принципиально не возможно, так как они плавно «перетекают» в «не лучших».

Дело в том, что вопросы задаваемые в анкете не позволяют измерить именно то, о чём они напрямую спрашивают. Это крайне наивный, как показывает наш анализ, подход. Анкета позволяет только измерить некоторые объективные характеристики исходя из которых отвечает заполняющий анкету.

Выводы Анализируемая анкета (да и любая другая на эту тему) позволит всего лишь узнать кто из преподавателей вообще остался не замечен студентами и его надо подогнать. Ну и присмотреться к тем преподавателям, у кого сила взаимодействия с студентом необычайно сильна.

11.2. Кадровая политика ордена иезуитов

Есть те, кто считают, что история — это не наука. Она, как правило, не говорит на языке математики и в какой-то мере даёт простор для спекуляций, то есть надуманных субъективных оценок. Объективность безусловно всячески приветствуется, но где её взять? Основную опасность для исторического исследования представляет искушение описать ушедшую реальность при помощи классификаций и понятий современности. Но такие понятия как «государство», «экономика», «собственность» содержат внутри себя клише, сложившиеся в контексте современной культуры. Описание прошлого на основе примерки к нему подобных клише несет в себе риск модернизации и ставит под вопрос валидность исторического исследования. Парадокс в том, что одни и те же данные, хранящиеся

в архивных документах, позволяют выдвигать и доказывать диаметрально противоположные гипотезы. В этом случае лучше послушать, что говорят данные сами, без нашей подсказки.

Предыстория Не будем как в случае анкет сдувать архивную пыль. Это вредно для здоровья. Возьмём в руки скучный документ. Его оригинал находится в Риме в Главном архиве ордена иезуитов Archivum Romanum Societatis Iesu (ARSI). Пятый фонд в этом архиве называется Germania и касается жизни Германской Ассистенции Товарищества, куда входили также Польская и Литовская провинция ордена.

Том номер 130 из фонда Germania фактически представляет собой канцелярскую книгу, в которую для сведения генерала заносились предложения провинциалов по кандидатурам на должность настоятеля определенного дома ордена и следующего провинциала. Таких записей с кандидатурами в период с 1684 — 1705 г. в книге зарегистрировано 412, т. е. ведение книги начато при правлении генерала Шарля де Нойэля[1] (1682-86) и окончено со смертью генерала Тирса Гонсалеса [2] (1687–1705). Генералу, как правило, представлялись три кандидатуры (по латыни terno), из которых он обычно выбирал первую по счету (отступлений от правил в книге зарегистрировано только 9). Если же ни одна из трёх предложенных кандидатур не устраивала генерала, то тогда его канцелярия запрашивала у провинциала новое представление. Так произошло в 53 случаях из 412 (доля отвергнутых представлений равна 13%), большинство из которых пришлось на конец правления де Ноэля и первую половину правления Гонсалеса, что свидетельствует об активном воздействии генералов на кадровую политику провинций.

Данные в Germ.130 представлены в виде последовательных записей вида:

[1]NOYELLE Charles de, род. 18 VII 1615 в Брюсселе, вступил в орден 29 IX 1630 в Бельгийской провинции, ум. 12 XII 1686 в Риме. С 1661 г. ассистент Германской ассистенции Товарищества Иисуса. Избран генералом ордена 5 VII 1682 на XII Генеральной Конгрегации. Управлял орденом почти 4 года. В это время в лоне французской церкви усилились галликанские течения, стремящиеся к независимости местной церкви от Рима. В 1683 г. под Веной христианские польско-австрийско-германские войска под командованием Яна III Собеского разбили армию Османской империи. Иезуиты принимали участие в военной кампании в качестве походных капелланов. В 1684 г. генерал инициировал открытие миссии чешских иезуитов в Москве, в которой также приняли участие иезуиты Литовской провинции. Поддерживал деятельность о. Маврикия Вотта ОИ (VOTA (Votta Carlo Maurizio) при дворе Яна III Собеского.

[2]Thyrsus Gonzalez de Santalla, род. 18 I 1624 в Арганза (Испания), вступил в орден 3 III 1643, ум. 27 X 1705 в Риме. Профессор философии и теологии в Саламанке в 1655-65 и 1676-87 гг. Миссионер. Избран генералом ордена 6 VII 1687 на XIII Генеральной Конгрегации и исполнял обязанности до смерти в 1705 г. В 1688 г. направил в Польшу визитатора Игнатия Дертинса (Diertins). Оказал влияние на обращение в католическую веру саксонского электора Фридриха Августа (будущего короля Речи Посполитой Августа II) и поддерживал вовлеченность в политику о. Маврикия Вотта ОИ. Активно боролся против пробабилизма в нравственном богословии (от лат. probabilis — приемлемый, возможный, вероятный — взгляд, согласно которому знание является только вероятным, т. к. истина недостижима), которому приписывал падение нравов. Выступал в печати, в том числе против янсенизма.

	Pro Domo Vilnensi	Pro Domo Nesvisiensis	Pro Domo Varsaviensi
Zawistowski Franciscus	0	2	0
Dzieniszewski Albertus	0	0	0
Rymgayło Josephus	0	1	0
Kołakowski Martinus	0	1	2
Narmunth Nicolaus	4	0	2

Таблица 11.1. Кросстаблица (небольшой кусочек)

```
Pro rectoratu Vilnensi, 1684, Kitnowski Petrus, Krasnodebski Adamus,
Wyrwicz Andreas
```

То есть запись содержит название должности, год, и три кандидатуры, предложенные провинциалом. Иногда (крайне редко) провинциал предлагал менее, чем три кандидатуры за раз, и, видимо, отдельно обосновывал своё предложение. За двадцать лет так случилось только 8 раз (2 раза запись содержит две кандидатуры и 6 раз только одну кандидатуру). Иногда, наоборот, провинциал предлагал больше, чем три кандидатуры на выбор. В период с 1684 г. по 1695 г. так произошло 20 раз (16 раз по 4 кандидатуры, 3 раза по 5 кандидатур и 1 раз 7 кандидатур). В последующие 10 лет ведения книги подобных случаев не зафиксировано. Примечательно, что в несколько чаще нарушение правила выбора из трех кандидатур происходило в случае домов Польской провинции. Так на должность провинциала Польской провинции предлагалось в 1687 г. 4 кандидата, в 1691 г. — 7 кандидатов, в 1695 — 5 кандидатов.

Поиск зависимостей Используя утилиты обработки текстовой информации **awk** и **sed** удалось получили список пар «претендент — должность». Причём должность фактически означает географическую привязку. Данный список был преобразован (свёрнут) в таблицу (в статье приведён лишь небольшой её кусочек), строки которой означали географически локализованные ректорские должности, а столбцы — претендентов:

```
> library("xtable")
> iesu_table <- table(read.table("data_iesu.txt", sep=",")
+                       )[c(137, 27, 112, 55, 92),2:4]
> colnames(iesu_table)<-c("Pro~Domo␣Vilnensi",
+                         "Pro~Domo␣Nesvisiensis",
+                         "Pro~Domo␣Varsaviensi")
> xtable(iesu_table,
> align="lp{1.6cm}p{1.6cm}p{1.6cm}")
```

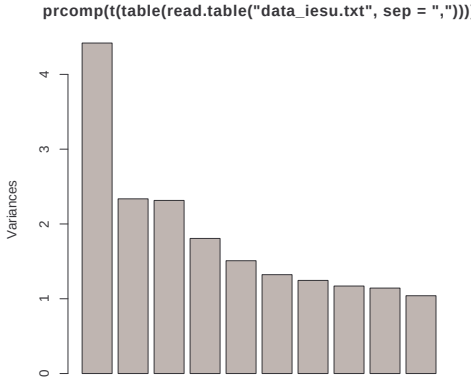

Рис. 11.4. Руководители Ордена иезуитов при принятии кадровых решений руководствовались лишь тремя факторами

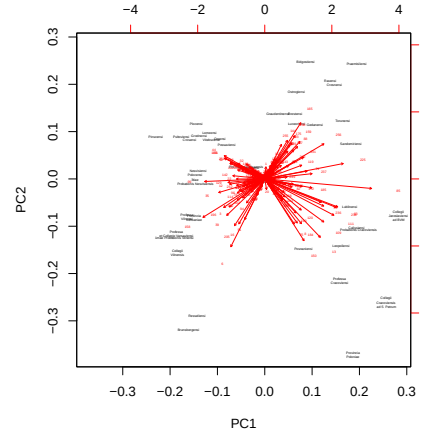

Рис. 11.5. Двойная диаграмма первых двух факторов

На пересечение строки и столбца помещалось количество выдвижений данного кандидата на данную должность за весь период наблюдений. Таблица была обработана методом основных компонент.

```
> plot(prcomp(t(table(read.table("data_iesu.txt",
+                                sep=","")))))
```

Вскрытая анализом факторная структура состоит из трёх факторов. Следующий шаг: посмотрим на реальные данные в пространстве первых двух факторов.

```
> labelxs <- gsub("_",
+                 "␣",
+                 rownames(t(table(read.table("data_iesu.txt",
+                                             sep=","))))))
> labelxs <- sub("(Pro␣rectoratu)␣", "", labelxs)
> labelxs <- sub("(Pro␣Domo)␣", "", labelxs)
> labelxs <- sub("(tiae)␣", "tiae\n␣", labelxs)
> labelxs <- sub("(Pro␣regenda)␣", "", labelxs)
> labelxs <- sub("(Provincia␣)", "Provincia\n", labelxs)
> labelxs <- sub("(Professa␣)", "Professa\n", labelxs)
> labelxs <- sub("(Collegii␣)", "Collegii\n", labelxs)
> labelxs <- sub("(␣ad␣)", "\n␣ad␣", labelxs)
```

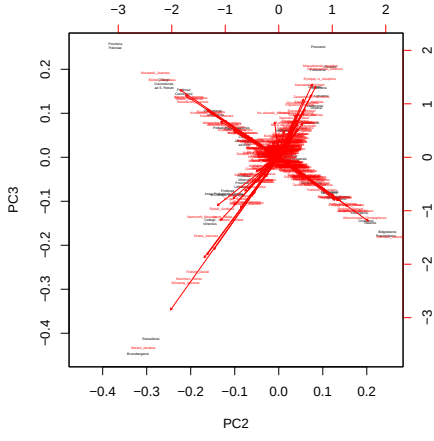

Рис. 11.6. Двойная диаграмма второго и третьего фактора

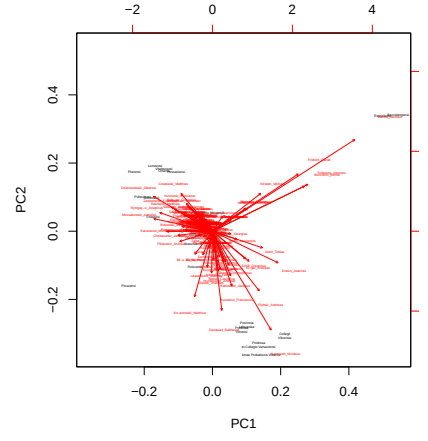

Рис. 11.7. Третий фактор в Литве

```
> biplot(prcomp(t(table(read.table("data_iesu.txt",
+                                  sep=",")))),
+        cex=c(0.25,0.25),
+        arrow.len = 0.025,
+        ylabs = 1:length(t(table(read.table("data_iesu.txt",
+                                  sep=",")))[1,]),
+        xlabs = labelxs,
+        )
```

Красным нанесены переменные, в которых оценивались должности. На основе вклада переменных и были выведены оси–факторы. Горизонтальная ось (1-й фактор) отражает наличие естественной группировки, следовательно это проявление ложного коэффициента корреляции. Этот корреляционный вклад вносят две несвязанные группы. Какие? По названием коллегиумов, заглянув в энциклопедию (`http://www.jezuici.krakow.pl/bibl/enc.htm`) обнаружим, что они расположены в разных провинциях: польской и литовской. Обе провинции находятся на территории одного федеративного государства Речи Посполитой, в котором было несколько административных центров: Краков, Варшава, Вильно. Вертикальная ось отражает иерархию существующую внутри провинций. Об этом свидетельствует концентрация в нижней части графика руководящих должностей в столичных городах: Варшаве, Вильно и Кракове. Таким образом выделенную ось мы можем с полным правом назвать «Столичность — Глубинка».

```
> biplot(prcomp(t(table(read.table("data_iesu.txt",
```

```
+                                  sep=","))))  ,
+       choices= c(2,3),
+       xlabs = labelxs,
+       arrow.len = 0.025,
+       cex=0.25)
```

Третий фактор показывает стоящую отдельно группу должностей Пруссии, входившей формально в Литовскую провинцию. Это хороший повод отдельно рассмотреть факторную структуру Литовской провинции (подмножество из файла с данными `data_iesu.txt`).

```
> labelxs <- gsub("_",
+               "␣",
+               rownames(t(table(read.table("data_litwa.csv",
+                                           sep=","))))
> labelxs <- sub("(Pro␣rectoratu)␣", "", labelxs)
> labelxs <- sub("(Pro␣Domo)␣", "", labelxs)
> labelxs <- sub("(tiae)␣", "tiae\n␣", labelxs)
> labelxs <- sub("(Pro␣regenda)␣", "", labelxs)
> labelxs <- sub("(Provincia␣)", "Provincia\n", labelxs)
> labelxs <- sub("(Professa␣)", "Professa\n", labelxs)
> labelxs <- sub("(Collegii␣)", "Collegii\n", labelxs)
> labelxs <- sub("(␣ad␣)", "\n␣ad␣", labelxs)


> biplot(prcomp(t(table(read.table("data_litwa.csv",
+                                  sep=",")))),
+       arrow.len = 0.025,
+       xlabs = labelxs,
+       cex=0.25)
```

Мы видим слева Пинск, а справа прусские Брунсберга и Решель. Что их отличает эти должности друг от друга? Национальная принадлежность кандидатов. Посмотрев на имена вы сразу же заметите явную концентрацию славянских фамилий (оканчание на -ски) слева, а иностранных (немецко звучащих) — справа.

Первый фактор явно отражает национальные различия как в пространстве признаков, так и в пространстве значений. Полюса оси — это «Прусскость — Белорусскость».

Второй фактор знакомая нам «Столичность — Провинциальность», отражающая иерархию в организационной структуре провинции. Данный переход подтверждает спектр фамилий руководящих кадров ордена. Слева видны польские фамилии, а справа не польские (чаще немецкие).

Эта же картина переносится на должности. В провинциальных городах немецкая часть спектра фамилий отсутствует вообще. столицах же присутствуют и немцы, и поляки, причём они расположены согласно оси первого фактора.

И что с того? Что мы имеем в сухом остатке. Что вскрыла такая обработка исторических сведений? Что собой представляет пространство факторов? Оно представляет собой пространство принятия кадровых решений (фрагмент соответствующей карты мира) в сознании провинциалов ордена и двух генералов ордена иезуитов в период с 1684 по 1705 год.

Что получили в результате историки? Фактическую информацию для проверки своих гипотез. Мы увидели иерархию о которой никто никогда явно не говорил, но все учитывали когда принимали решения. Естественно эту зависимость можно обнаружить и «методом пристального вглядывания». Статистика всего лишь инструмент, но в ряде случаев инструмент удобный.