**С.С. Задорожный**

# Статистическая обработка данных на языке R

Учебно-методическое пособие

Москва
Физический факультет МГУ им. М.В. Ломоносова
2023 г.

Задорожный Сергей Сергеевич

**Статистическая обработка данных на языке R**
М. Физический факультет МГУ им. М.В. Ломоносова, 2023. – 104с.

R – это мощный язык для статистических вычислений и графики, который может справиться поистине с любой задачей в области обработки данных. Он работает во всех важных операционных системах и поддерживает тысячи специализированных модулей и утилит. Все это делает R замечательным средством для извлечения полезной информации из гор сырых данных.

Обработка данных, получаемых из различных источников является очень важной составляющей физических исследований (как теоретических, так и экспериментальных). Использование языка R для этой цели позволит существенно сократить время обработки и получить результаты, которые достаточно трудоёмко получаются другими методами.

Пособие может использоваться как для изучения языка R в рамках спецкурса у студентов 3 курса, так и для самостоятельного освоения сотрудниками и студентами старших курсов. Для лучшего понимания материала желательно иметь навыки программирования на других языках и начальные знания по математической статистике.

Автор – сотрудник кафедры математического моделирования и информатики физического факультета МГУ.

Рецензенты: доцент физического факультета МГУ Митин И.В.,
профессор физического факультета МГУ Голубцов П.В.

---

Подписано в печать февраля 2023 г.
Формат 60x90/16. Объём 6,62 п.л.
Тираж 30 экз. Заказ №

---

Отпечатано в Отделе оперативной печати физического факультета МГУ
Физический факультет МГУ им. М.В. Ломоносова
119991 Москва, ГСП-1, Ленинские горы, д.1, стр. 2

© Физический факультет МГУ им. М.В. Ломоносова, 2023 г.
© Задорожный С.С. , 2023 г.

# Оглавление


1. **Предисловие ... 4**
2. **Основы программирования в R ... 5**
3. **Что такое данные и зачем их обрабатывать. ... 25**
4. **Типы данных ... 31**
5. **Пример работы в R ... 43**
6. **Анализ одномерных данных ... 49**
7. **Анализ связей: двумерные данные ... 60**
8. **Анализ структуры: data mining ... 79**
9. **Анализ временныхрядов ... 91**
10. **Статистическая разведка ... 100**
11. **Самое необходимое ... 102**
12. **Литература ... 103**

# 1. Предисловие

Обработка данных, получаемых из различных источников является очень важной составляющей физических исследований.

Высокопроизводительные компьютеры и доступное программное обеспечение позволяют реализовать полный цикл обработки данных, который состоит, в общем случае, из следующих шагов:

- *Доступ к обрабатываемым данным (их загрузка из разных источников и создание совокупности взаимосвязанных исходных таблиц);*
- *Редактирование загруженных данных (замена или удаление пропущенных значений, преобразование в более удобный вид);*
- *Аннотирование данных (чтобы помнить, что представляет собой каждый их фрагмент);*
- *Получение общих сведений о структуре данных (вычисление описательных статистик для того, чтобы охарактеризовать анализируемые показатели);*
- *Графическое представление данных и результатов вычислений в понятной информативной;*
- *Моделирование данных (нахождение зависимостей и тестирование статистических гипотез);*
- *Оформление результатов виде наглядных таблиц и диаграмм.*

В условиях, когда к услугам пользователя имеются десятки пакетов прикладных программ, актуальна проблема выбора: какое программное обеспечение анализа данных следует предпочесть для своей практической работы?

Комбинировать различные типы анализа, иметь доступ к промежуточным результатам, управлять стилем отображения данных, добавлять собственные расширения программных модулей и оформлять итоговые отчеты в необходимом виде позволяют коммерческие вычислительные системы, такие как Matlab, SPSS и др.

Однако прекрасной альтернативой им является бесплатная программная среда R, являющаяся современной и постоянно развивающейся статистической платформой общего назначения.

Сегодня R является безусловным лидером среди свободно распространяемых систем статистического анализа. Ведущие университеты мира, аналитики крупнейших компаний и исследовательских центров постоянно используют R при проведении научно-технических расчетов и создании крупных информационных проектов. Широкое преподавание статистики на базе пакетов этой среды и всемерная поддержка научным сообществом обусловили то, что приведение скриптов R постепенно становится общепризнанным "стандартом" как в журнальных публикациях, так и при неформальном общении ученых всего мира.

Язык R представляет собой набор программных средств для работы с данными, вычислений и графического отображения. Он является средством разработки новых методов интерактивного анализа данных, поэтому быстро развивается и расширяется большой коллекцией пакетов.

R включает следующие возможности:

- *Эффективная обработка и хранение данных.*
- *Набор операторов для обработки матриц.*
- *Цельная, непротиворечивая, комплексная коллекция утилит для анализа данных.*
- *Графические средства для анализа данных и визуализации.*
- *Хорошо развитый, простой и эффективный язык программирования.*

# 2. Основы программирования в R

### *Инструменты для обработки данных*

В принципе, обрабатывать данные можно и без компьютера. Однако многие статистические расчеты настолько тяжеловесны, что уже в XIX веке стали придумывать способы их автоматизации.

Почти в любом компьютере с предустановленной системой есть про грамма-калькулятор. Такая программа обычно умеет выполнять четыре арифметических действия, часто — считать квадратные корни и степени, иногда логарифмы. В принципе, этого достаточно для того, чтобы делать простейшую обработку: считать среднее значение, стандартное отклонение, некоторые тесты.

Вообще говоря, для того чтобы делать тесты, кроме калькулятора потребуются еще и статистические таблицы, из которых можно узнать примерные значения так называемых статистик — величин, характеризующих данные в целом. Таблицы используются потому, что точный подсчет многих статистик слишком сложен, поэтому используются оценочные значения. Статистические таблицы можно найти во многих книгах по классической статистике, они также «встроены» во многие специализированные программы.

Главный недостаток калькулятора — сложность работы с сериями чисел.

Чтобы работать с сериями более эффективно (и не только для этого), были придуманы электронные таблицы. Их сила прежде всего в том, что они помогают визуализировать данные. Однако для обработки данных нужен гораздо более специализированный инструмент. Конечно, программы типа MS Excel, имеют, среди прочего, набор статистических функций. Но поскольку это не основной их компонент, то набор статистических тестов невелик.

Выход — в использовании специализированных статистических программ.

В России широко распространена система STATISTICA. Она отличается мощной графической частью с гибкой настройкой. Другим, и очень серьезным, преимуществом STATISTICA является наличие переведенной на русский язык системы помощи, свободно доступной в Интернете. Однако, популярным этот пакет является в основном только в России. Поэтому легко можно представить проблемы с обменом данными. Гибкость STATISTICA велика, но только в пределах так называемых модулей. Если надо скомбинировать работу нескольких модулей, то придется отойти от графического подхода — например, начать писать макросы.

Одной из наиболее продвинутых систем анализа данных является SAS. Это коммерческая, очень мощная система, обладающая развитой системой помощи и имеющая долгую историю развития. Создавалась она для научной и экономической обработки данных и до сих пор является одним из лидеров в этом направлении. Однако пользоваться ей поначалу не очень легко даже человеку, знакомому с командной строкой и программированием. А стоимость самой системы просто запредельная — многие тысячи долларов!

### *Из истории S и R*

R — это среда для статистических расчетов. R задумывался как свободный аналог среды S-Plus, которая, в свою очередь, является коммерческой реализацией языка расчетов S. Язык S — довольно старая разработка. Он возник еще в 1976 году в компании Bell Labs и был назван, естественно, «по мотивам» языка С. Первая реализация S была написана на FORTRAN и работала под управлением операционной системы GCOS. В 1980 г. реализация была переписана под UNIX, и с этого момента S стал распространяться в основном в научной среде. Начиная с третьей версии (1988 г.), коммерческая реализация S называется S-Plus. Последняя распространялась компанией Insightful, а сейчас

распространяется компанией TIBCO Software. Версии S-Plus доступны под Windows и различные версии UNIX — естественно, за плату, причем весьма и весьма немаленькую. Собственно, высокая цена и сдерживала широкое распространение этого во многих отношениях замечательного продукта. Тут-то и начинается история R.

В августе 1993 г. двое молодых новозеландских ученых анонсировали свою новую разработку, которую они назвали R (буква «R» была выбрана просто потому, что она стоит перед «S», тут есть аналогия с языком программирования C, которому предшествовал язык B). По замыслу создателей (это были Robert Gentleman и Ross Ihaka), это должна была быть новая реализация языка S, отличающаяся от S-Plus некоторыми деталями. Фактически же они создали не аналог S-Plus, а новую статистическую систему. Когда в проекте появилось достаточно много возможностей, в том числе уникальная по легкости система написания дополнений, все большее количество людей стало переходить на R. Количество книг, написанных про R, за последние годы выросло в несколько раз, а количество пакетов уже приближается к трем с половиной тысячам!

Коротко говоря, R применяется везде, где нужна работа с данными. Это не только статистика в узком смысле слова, но и «первичный» анализ (графики, таблицы сопряженности) и продвинутое математическое моделирование. В принципе, R может использоваться и там, где в настоящее время принято использовать специализированные программы математического анализа, такие как MATLAB или Octave. Но, разумеется, чаще всего его применяют для статистического анализа — от вычисления средних величин до вейвлет-преобразований и временных рядов. Географически R распространен тоже очень широко. Трудно найти американский или западноевропейский университет, где бы не работали с R.

У R два главных преимущества: неимоверная гибкость и свободный код. Гибкость позволяет создавать приложения (пакеты) практически на любой случай жизни. Нет, кажется, ни одного метода современного статистического анализа, который бы не был сейчас представлен в R. Свободный код – это не просто бесплатность программы, но и возможность разобраться, как именно происходит анализ, а если в коде встретилась ошибка – самостоятельно исправить ее и сделать исправление доступным для всех.

Конечно, у R есть и недостатки. Самый главный из них — это трудность обучения программе. Команд много, вводить их надо вручную, запомнить все трудно, а привычной системы меню нет. Поэтому порой очень трудно найти, как именно сделать какой-нибудь анализ.

Не стоит забывать, однако, что сила R — там же, где его слабость. Интерфейс командной строки позволяет делать такие вещи, которых рядовой пользователь других статистических программ может достичь только часами ручного труда. Второй недостаток R — относительная медлительность. Но этот недостаток преодолевается, хотя и медленно.

### ***Как скачать и установить R***

Поскольку R — свободная система, то его можно скачать и установить без всякой оплаты.

Самую свежую версию можно скачивать R из так называемого CRAN (Comprehensive R Archive Network) сервера. У этого сайта довольно много зеркал (копий), так что выбирайте подходящее. Скачать можно как исходный код для последующей компиляции, так и собранный бинарный пакет. Кстати говоря, исходный код R компилируется очень просто.

Интересно заметить, что Windows-инсталляция R является, как это принято сейчас

говорить, «portable» и может запускаться, например, с флэшки или лазерного диска. R делает пару записей в реестр, но для работы они совершенно некритичны. Единственное, что надо при этом иметь в виду, — рабочая папка должна быть открыта для записи, иначе некуда будет записывать результаты работы. Еще одна вещь, важная для всех операционных систем: R держит все свои вычисления в оперативной памяти, поэтому если в процессе работы, скажем, выключится питание, результаты сессии, не записанные явным образом в файл, пропадут.

### *Как начать работать в R*

Каждая операционная система имеет свои особенности работы. Но в целом можно сказать, что под все операционные системы существует так называемый «терминальный» способ запуска, а под Mac и Windows имеется свой графический интерфейс(GUI) с некоторыми дополнительными возможностями (разными в разных операционных системах).

В Windows необходимо вызывать программу `Rgui.exe`. GUI построен так, что всё равно запускается терминал-подобное окно – общение с R проходит в режиме диалога. Полноценного GUI с R не поставляется, хотя существуют многочисленные попытки создать такую систему.

### *Первые шаги*

Перед тем как начать работать, надо понять, как выйти. Для этого надо ввести одну команду `q()` и нажать `Enter`.

Уже такой простой пример показывает, что в R любая команда имеет аргумент в круглых скобках. Если аргумент не указан, скобки все равно нужны.

Как узнать, как правильно вызывать функцию? Для этого надо научиться получать справку. Есть два пути. Первый — вызвать команду справки:

```
> help(q)
```

или

```
> ?q
```

И вы увидите отдельное окно помощи в основном окне программы.

Предыдущую команду легко вызвать, нажав клавишу-стрелку «вверх». Если при выходе из системы сохранить сессию в файл `.Rhistory`, то команды будут доступны и в следующей сессии — при условии вызова R из той же самой папки.

### *Базовые объекты языка R*

Любой объект языка R имеет набор атрибутов (attributes). Этот набор может быть разным для объектов разного вида, но каждый объект обязательно имеет два встроенных атрибута:

- длина (length);
- тип (mode).

Смысл атрибута «длина» достаточно очевиден. Тип объекта — болеесложная сущность. Так например, тип объекта `vector` определяется типом элементов, из которых он состоит.

### *Вектор*

Вектор является простейшим объектом, объединяющим элементы одного примитивного типа. Создать пустой вектор можно при помощи функции `vector()`:

```
> v <- vector(mode = "logical", length = 0)
```

Здесь аргумент `length` задает длину вектора (то есть количество содержащихся в нем элементов), а `mode` — примитивный тип элементов.Объекты примитивного типа `numeric` служат для представления обычных чисел с плавающей точкой. Отметим, что, кроме «обычных» значений, объекты типа `numeric` могут принимать ряд специальных. Таковыми являются бесконечность `Inf` (которая, в свою очередь, можетбыть положительной или отрицательной) и `NaN` (Not-A-Number). Последнее служит для отображения результатов

операций в том случае, когда этот результат не определен (например, деление ноля на ноль, вычитание из бесконечности и т. п.).

Еще одно специальное значение, `NA` (Not Available), служит для отображения «пропусков» в данных. Использование отдельного значения для пропущенных наблюдений позволяет решить универсальным образом множество проблем при работе с реальными данными. Сразу же отметим, что значение `NA` могут принимать объекты любого примитивного типа.

Объекты типа `complex` содержат комплексные числа. Мнимая часть комплексного числа записывается с символом `i` на конце. Примером комплексного вектора длины 3 может служить

```
> c(5.0i, -1.3+8.73i, 2.0)
```

Комплексные векторы могут принимать те же специальные значения, что и числовые.

Объекты типа *logical* могут принимать одно из трех значений: `TRUE`, `FALSE` и `NA`. Кроме этого, существуют (в основном в целях обратной совместимости) глобальные переменные `T` и `F`, имеющие значение `TRUE` и `FALSE` соответственно.

Каждый элемент вектора типа *character* является, как следует из названия типа, строкой. Каждая такая строка может иметь произвольную длину.

Как и в языке С, символ обратной косой черты является специальным и предназначен для ввода так называемых escape-последовательностей. В частности, обычный символ обратной косой черты (backslash)вставляется при помощи последовательности «`\\`», символ табуляции – при помощи «`\t`», а перехода на новую строку — при помощи «`\n`».

Вместо вызова функции `vector()` с указанием типа элемента удобно вызывать функции, создающие объект нужного типа сразу:

```
> v1 <- numeric(length = 0)
> v2 <- character(length = 0)
> v3 <- complex(length = 0)
> v4 <- logical(length = 0)
```

### *Список*

Вектор содержит элементы исключительно одного типа. Это делает его простым, но в то же время несколько ограничивает сферу его применения. В этом полной противоположностью вектору служит список. Список является объектом типа `list`, его длина — это количество компонентов, из которых он состоит. Каждый компонент списка является произвольным объектом языка R.

Так, например, первый компонент списка может быть числовым вектором длины 10, второй компонент — списком, а третий — строкой. Пустой список можно получить вызовом функции `vector()`:

```
> l <- vector(mode = "list", length = 3)
```

Список с одновременной инициализацией создается функцией `list()`:

```
> l <- list(c(1, 5, 7), "string", c(TRUE, FALSE))
```

Отдельные компоненты списка могут иметь имя. Для его задания необходимо передать функции `list()` пары «имя=значение»:

```
> l <- list(A=c(1, 5, 7), "string", B=c(TRUE, FALSE))
```

Преобразовать список в вектор можно командой `unlist()`, а в матрицу — при помощи `do.call()`.

### *Матрица и многомерная матрица*

Как было отмечено выше, каждый объект языка R обладает некоторым набором атрибутов. Атрибуты — это список, каждый компонент которого имеет имя (именованный список). Матрицы являются хорошим примером применения атрибутов на практике. Так, матрица фактически представляет собой просто вектор с дополнительным атрибутом `dim` и, опционально,

атрибутом dimnames.

Атрибут dim представляет собой числовой вектор, длина которого равна размерности матрицы (таким образом, типичная двумерная матрица имеет атрибут dim длины 2). Элементы вектора dim показывают, сколько строк, столбцов и т. д. содержится в матрице. Произведение всех элементов вектора dim должно совпадать с длиной объекта. Атрибут dimnames позволяет задать название для каждой размерности матрицы.

Многомерная матрица (многомерный массив) создается функцией *array()*:

```
> a <- array(data = NA, dim = length(data), dimnames = NULL)
```

Здесь data — исходные данные для матрицы, dim, dimnames — значения соответствующих атрибутов. Двумерная матрица (как наиболее часто используемый вид) может быть создана при помощи специальнойфункции matrix():

```
> m <- matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE,
+ dimnames = NULL)
```

Здесь data — исходное содержимое матрицы, nrow, ncol — количество строк и столбцов. Аргумент byrow позволяет задать, каким образом заполнять матрицу из вектора data: по столбцам (по умолчанию) или по строкам, dimnames позволяет задать имена строк и столбцов. Кроме этого, создать матрицу или многомерную матрицу можно простым назначением атрибута dim вектору:

```
> v <- numeric(6)
> dim(v) <c(2, 3); v
     [,1] [,2] [,3]
[1,]  0     0     0
[2,]  0     0     0
```

### *Факторы*

Объект типа factor является способом представления номинальных(категориальных) и шкальных типов данных в R. Качественные данные принимают значения в некотором конечном множестве. Поэтому для задания подобных данных достаточно задать это множество (способ кодирования этих данных) — множество *градаций*, или *уровней* (levels) фактора. Чаще всего для кодирования градаций берут обычные числа. Создать фактор можно при помощи функции factor(data, levels, labels=levels). Здесь data — отдельные значения, levels — вектор всех возможных значений для градаций, labels — опциональный вектор, содержащий текстовые обозначения градаций, используемые при выводе фактора на экран. Задание дополнительно аргумента order позволяет получить упорядоченный фактор.

```
> F <- factor(c(1, 1, 2, 1, 3),levels = 1:5,
+ labels=c("A","B","C", "D", "E"))
> f
[1] A A B A C Levels: A B C D E
> f <- factor(c(1, 1, 2, 1, 3), levels = 1:5,
+ labels = c("A", "B", "C", "D", "E"), ordered = TRUE)f
[1] A A B A C Levels: A < B < C < D < E
```

### *Таблица данных*

Матрицы и многомерные матрицы позволяют хранить элементы только одного типа и поэтому не очень хорошо приспособлены для работы с реальными данными. Реальные данные, как правило, представляют собой набор однотипных *наблюдений*. Каждое наблюдение, в свою очередь, есть совокупность элементов разного типа (например, для медицинских данных характерно наличие поля «имя» строкового типа, категориального поля «пол» и числового поля «возраст»).

Таким образом, помещая отдельные наблюдения в строки, приходим к своеобразной матрице, *таблице данных*, в которой элементы в одном столбце должны обязательно иметь одинаковый тип, но этот тип может меняться от столбца к столбцу. В статистической литературе традиционно отдельное наблюдение называется *индивидом*, а столбец с данными – *признаком*.

С другой стороны, объект «таблица данных» (data frame) может рассматриваться как обычный список векторов одинаковой длины (собственно, он таковым и является). Создать таблицу данных можно при помощи функции `data.frame(..., row.names = NULL)`.

В качестве аргументов здесь передаются столбцы создаваемого набора данных либо в виде пар типа *имя = значение*, либо просто «как есть» (в таком случае столбец получит автоматически назначаемое имя `V1`, `V2`, ...). Аргумент `row.names` позволяет задать вектор с названиями для отдельных строк.

Отметим, что большинство объектов могут быть преобразованы в объект типа `data.frame` «естественным» образом, например:

```
> data.frame(matrix(1:6, nrow = 2, ncol = 3))
  X1 X2 X3
1  1  3  5
2  2  4  6
```

### *Выражение*

Язык R предоставляет доступ к конструкциям самого языка. Так, функция является объектом типа `function`, причем его содержимое – это просто код функции. Поэтому с функциями можно в некотором смысле работать как с обыкновенными переменными (переименовывать, передавать в качестве аргументов, сохранять в списки и т. д.)

Объект типа `expression` представляет собой еще *не выполненное* выражение языка R, то есть попросту строку или набор строк кода.

Такие выражения можно выполнять или, в более общем случае, подставлять в них значения каких-либо переменных, преобразовывать и т. д. Создать выражение можно при помощи функции `expression()`:

```
> e <- expression((x + y) / exp(z))
> x <- 1; y <- 2; z <- 0; eval(e)
[1] 3
```

Ее аргументом является выражение языка R. Вычислить выражение можно при помощи функции `eval()`. По умолчанию переменные для подстановки берутся из текущего окружения. Это поведение можно переопределить, передав требуемое окружение через аргумент `envir`. В качестве примера нетривиального преобразования выражений отметим функцию символьного дифференцирования `D()`, первым аргументом которой является выражение, которое необходимо дифференцировать, а вторым — название переменной дифференцирования:

```
> e <- expression((x + y) / exp(z))
> D(e, "x")
1/exp(z)
```

Результатом функции `D()` снова является объект типа `expression`.

### *Операторы доступа к данным*

Все операции доступа к данным можно разделить на две категории: *извлечения* и *замены*. В первом случае оператор доступа к данным стоит справа от оператора присваивания, во втором – слева:

```
> value <- x[sub] # извлечение
>  x[sub] <- value   # замена
```

### *Оператор [ с положительным аргументом*

Оператор «одинарная левая квадратная скобка» служит для извлечения элементов с

индексами, соответствующими элементам переданного вектора. Например, следующая конструкция извлечет из вектора v первые 3 элемента:

```
> v <- 1:5
> v[1:3]
[1] 1 2 3
```

Результат имеет ту же длину, что и вектор индексов. Никаких ограничений ни на длину, ни на содержимое вектора индексов нет (таким образом, любой индекс может встречаться произвольное число раз). Индексы, равные NA, или имеющие значение больше, чем длина вектора, приводят в результате к NA (или к пустым строкам).

В случае использования операции замены длина результата равна максимальному индексу. Элементы, значение которым не присвоено, получают значение NA:

```
> v <- 1:5
> v[10] <- 10
> v
1  2  3  4  5 NA NA NA NA 10
```

***Оператор [ с отрицательным аргументом***

В случае, когда вектор индексов содержит отрицательные значения, результат содержит все элементы, кроме указанных. Например:

```
> v <- 1:5
> v[-c(1,5)]
[1] 2 3 4
```

Нулевые, повторяющиеся и индексы больше длины вектора игнорируются. Пропущенные значения в индексах приводят к ошибке, равно как и смесь положительных и отрицательных индексов.

***Оператор [ со строковым аргументом***

В качестве индекса можно использовать строку. В таком случае поиск элементов идет не по номеру, а по имени.

```
> v <- 1:5
> names(v) <- c("first", "second", "third", "fourth", "fifth")
> v
first second  third fourth  fifth
1       2       3       4     5
> v[c("first",  "third")]
first third
  1     3
```

***Оператор [ с логическим аргументом***

Типичным примером использования оператора [ с логическим вектором в качестве аргумента является конструкция вида

```
> v[v < 0]
```

извлекающая из вектора v только отрицательные элементы. Предполагается, что длина аргумента совпадает с длиной вектора, в противном случае аргумент повторяется столько раз, сколько необходимо для достижения длины вектора. В результат попадают только те элементы вектора, элемент вектора индексов для которых совпадает с TRUE. Индекс NA выбирает элемент NA (с «никаким» индексом). Таким образом, длина результата совпадает с общим количеством индексов TRUE и NA:

```
> v <- 1:50
> even <- c(FALSE, TRUE)
```

```
> v[even]
[1]  2  4  6  8 10 12 14 16 18 20 22 24 26 28 30 32 34 36
38 40 42 44 46 48 50
> three <- rep(c(FALSE, FALSE, TRUE), length = length(v))
> v[even & three]
[1]  6 12 18 24 30 36 42 48
```

Отметим, что в случае замены индексы NA допускаются, только если длина вектора, стоящего справа от оператора присваивания, равна 1 (в противном случае непонятно, следует ли использовать очередной элемент замещающего вектора или нет; при длине замещающего вектора = 1 эта неоднозначность, очевидно, не влияет на результат):

```
v[c(TRUE, NA)] <- 1:50
Error in v[c(TRUE, NA)] <- 1:50 :
NAs are not allowed in subscripted assignments
> v[c(TRUE, NA)] <- 0
> v
[1]  0  2  0  4  0  6  0  8  0 10  0 12  0 14  0 16  0 18
0 20  0 22  0 24  0
[26] 26  0 28  0 30  0 32  0 34  0 36  0 38  0 40  0 42
0 44  0 46  0 48  0 50
```

### *Оператор* $

Оператор $ служит для извлечения отдельного элемента из списка. Синтаксис очень прост: слева от знака $ стоит список, элемент которогоследует извлечь, а справа – имя извлекаемого элемента:

```
> .Machine$double.eps
[1] 2.220446e-16
```

Оператор $ обладает одной полезной особенностью: имя элемента нет необходимости передавать полным, достаточно лишь префикса, определяющего нужный элемент однозначно.

```
> .Machine$double.ep
> [1] 2.220446e-16
```

В случае, если элемент не определяется однозначно, будет возвращено значение NULL:

```
> names(.Machine)[grep("double.e",  names(.Machine))]
[1] "double.eps"   "double.exponent"
> .Machine$double.e
NULL
```

Следует, однако, предельно осторожно обращаться с неполными именами при выполнении замены: произойдет добавление *нового* значения с коротким именем. Старое значение с полным именем останется неизменным:

### *Оператор* [[

Вообще говоря, оператор $ может быть удален из языка без потери какой-либо функциональности. Оператор [[ позволяет осуществлять все те же операции и даже больше. Выражение типа .Machine$double.eps полностью эквивалентно .Machine[["double.eps"]]. Первое, правда,короче на 5 символов и выглядит как единое целое.

Оператор $ не может быть использован, если элемент списка не имеет имени: доступ по индексу возможен только посредством оператора [[. Кроме этого, оператор [[ следует применять в том случае, когда интересующее нас имя элемента уже содержится в некоторой переменной:

```
> l <- list(A=1:10)
```

```
> cname <- "A"
> l$cname
NULL
> l[[cname]]
[1]  1  2  3  4  5  6  7  8  9 10
```

Отметим также, что по умолчанию оператор `[[` требует точного совпадения переданного названия элемента. Это поведение можно переопределить при помощи аргумента `exact`:

```
> l <- list(abc=1:10)
> l$a
[1]  1  2  3  4  5  6  7  8  9 10
> l[["a"]]
NULL
> l[["a", exact=FALSE]]
[1]  1  2  3  4  5  6  7  8  9 10
```

Необходимо всегда иметь в виду важный факт: разобранный ранее оператор `[` работает не только с векторами, но и со списками. Отличие заключается в том, что оператор `[[` всегда возвращает *элемент списка*, а `[` – *список*:

```
> .Machine[[1]]
[1] 2.220446e-16
> .Machine[1]
$double.eps
[1] 2.220446e-16
```

Кроме этого, аргумент оператора `[[` всегда имеет длину 1, а аргумент оператора `[` может быть вектором с семантикой, описанной выше.

***Доступ к табличным данным***

Табличные данные, организованные в матрицы или таблицы данных, имеют свои правила доступа: каждое измерение может быть индексировано независимо. Вектор индексов для каждого измерения может принимать любую из форм, допустимых для оператора `[`, как было описано выше (хотя векторы строк будут индексировать не имена элементов, а имена измерений):

```
> state.x77[1:2, c("Area", "Population")]
        Area Population
Alabama  50708    3615
Alaska  566432    365
```

В случае если одно (или больше) измерение результата будет иметь длину 1, то это измерение будет «схлопнуто»: так, ниже видно, что

```
> state.x77[1:2, "Area"]
Alabama  Alaska
50708  566432
```

является вектором, а не матрицей. Если такое поведение нежелательно, то его можно переопределить заданием аргумента `drop`:

```
> state.x77[1:2, "Area", drop=FALSE]
            Area
Alabama     50708
Alaska      566432
```

Так как матрицы являются обычными векторами, то их элементы можно индексировать по порядку так же, как если бы измерения отсутствовали. Двумерные матрицы при этом

индексируются по столбцам; многомерные – так, что первое измерение меняется наиболее часто, а последнее – наиболее редко:

```
> m <-matrix(1:6, nrow = 3)
> m
     [,1] [,2]
[1,]  1    4
[2,]  2    5
[3,]  3    6
> m[2:4]
[1] 2 3 4
```

Такая гибкость очень часто приводит к трудноуловимым ошибкамв случае пропуска запятых при индексировании.

Объект типа `data. frame` является обычным списком из векторов, поэтому доступ к нему при помощи оператора `[` аналогичен доступу к списку.

***Пустые индексы***

Отдельные индексирующие векторы могут быть опущены. В таком случае выбирается все измерение и нет необходимости дополнительно узнавать его размер. Наиболее часто это используется при выборке отдельных строк или столбцов из матрицы:

```
> m <-matrix(1:6, nrow = 3)
> m[c(2,3), ]
     [,1] [,2]
[1,]  2    5
[2,]  3    6
> m[, 2]
[1] 4 5 6
```

Пустой индекс можно использовать и с векторами:

```
> v <- 1:10
> v[] < -0
> v
[1] 0 0 0 0 0 0 0 0 0 0
```

Здесь заменяются все элементы вектора `v` на 0, но при этом сохраняет все атрибуты (например, имена отдельных элементов). В некоторых случаях это может быть более предпочтительно, чем, скажем, конструкция вида

```
> v <-rep(0, length(v))
```

***Функции и аргументы***

Функции являются объектами типа `function`. Единственное их отличие, скажем, от вектора заключается в наличии в языке дополнительного оператора вызова функции `()`. Создать функцию можно при помощи вызова функции `function()`:

```
function(arglist) expr
```

Здесь `arglist` — список аргументов в виде имен или пар *имя* = *значение*, а `expr` — объект типа `expression`, составляющий тело функции. Выполнение функции происходит до вызова функции `return()`, аргумент которой становится возвращаемым значением, либо до выполнения последнего выражения в теле функции. В последнем случае именно значение последнего выражения будет возвращено из функции.Возвращаемое значение можно замаскировать при помощи функции `invisible()`.

```
> norm <-function(x) sqrt(x%*%x)
> fib <-function(n)
```

```
+ {
+    if (n<=2) {
+       if (n>=0) 1 else 0
+    }
+    else {
+       return(Recall(n-1) + Recall(n-2))
+    }
+ }
```

Обратите внимание на использование фигурных скобок. Несмотря на то, что во многих случаях они могут быть опущены, рекомендуется их использовать чаще, особенно для функций, циклов и условных конструкций. Наоборот, точка с запятой в конце строки, хотя и допускается, но совершенно не обязательна, и лучше ее опускать. Так как функции суть обычные переменные, то они могут быть переименованы естественным образом:

```
> fibonacci <- fib; rm(fib)
> fibonacci(10)
```

Конструкция `Recall()` позволяет вызвать «текущую функцию» независимо от используемого имени, тем самым сохраняя работоспособность рекурсивных функций даже после переименования.

Аргументы функций можно разделить на обязательные и необязательные. Наличие возможности передавать необязательные аргументы (опции) является одной из отличительных черт языка R.

Так как функции может быть передан не весь набор аргументов, то, естественно, должен существовать механизм *назначения аргументов*. Так, аргументы могут передаваться по имени (и поскольку их число конечно, то стандартные правила сокращения имен до уникального префикса работают и здесь). Есть также способ передачи аргументов по их позиции в операторе вызова функции.

Назначение значений аргументам происходит в три этапа:

- *Аргументы, переданные в виде пары имя = значение, имена которых совпадают явно.*
- *Аргументы, переданные в виде пары имя = значение, имена которых совпали по уникальному префиксу.*
- *Все остальные аргументы по порядку.*

Ключевое слово «`...`» (*ellipsis*) позволяет создавать функции с произвольным числом аргументов. Его наличие несколько усложняет процедуру назначения аргументов, так как происходит разделение аргументов на те, что находятся до троеточия, и располагающихся после. Если к первым применяются стандартные правила назначения аргументов, то последние можно назначить исключительно по полному имени, все неназначенные аргументы «съедаются» троеточием и доступны изнутри функции в виде именованного списка.

Как правило, аргументы функции, идущие после троеточия, в справке представлены в виде пары *имя = значение*. Кроме того, так как аргументы назначаются по имени, то при вызове функции они могут находиться на любой позиции, нет необходимости передавать их последними. Рассмотрим простой пример:

```
> f <- function(aa, bb, cc, ab, ..., arg1 = 5, arg2 = 10) {
+ print(c(aa, bb, cc, ab, arg1, arg2)); print(list(...))
+ }
> f(arg1 = 7, aa = 1, a = 2, ac = 3, 4, 5, 6)
[1]  1  4  5  2  7 10
$ac
[1] 3
```

```
[[2]]
[1] 6
```

Назначение аргументов здесь происходит так:

- *Назначаются аргументы arg1 и aa, так как их имена совпадают целиком.*
- *Значение аргументу aa уже было присвоено на предыдущем шаге, поэтому ab может быть назначен по совпадению префикса a, ставшего уникальным.*
- *Вследствие совпадения имени для переданного аргумента ac происходит добавление аргумента к списку (. . . ). arg2 получает значение по умолчанию. После этого этапа именованных аргументов не остается.*
- *Аргументы bb и cc назначаются по порядку значениями 4 и 5 соответственно. Аргумент 6 добавляется к списку list(...).*

При отсутствии оператора троеточия вызов функции оканчивается ошибкой при попытке назначить аргументы на шагах 3 и 4.

Функция `args(fun)` выводит список всех аргументов функции `fun`. В заключение отметим еще одну важную особенность, отличающую R от многих других языков программирования: аргументы функций вычисляются «лениво», то есть не в момент вызова функции, а в момент использования (этот факт объясняется просто: язык R является интерпретируемым). В связи с этим надо соблюдать большую осторожность, скажем, при передаче в качестве аргументов результатов вызовов функций со сторонними эффектами: порядок их вызова может отличатьсяот ожидаемого.

С другой стороны, такое поведение позволяет в некоторых случаях существенно упростить задание значений по умолчанию для аргументов:

```
f <-function(X, L = N %/% 2)
{
   N <- length(X)
   do.something(X, L)
}
```

Здесь вычисление аргумента `L` произойдет где-то внутри функции `do.something()`. К этому моменту переменной `N` уже будет назначено значение, и выражение `N %/% 2` будет корректно определено.

### *Циклы и условные операторы*

Как и многие языки программирования, R предоставляет конструкции, позволяющие управлять исполнением программы в зависимости от внешних условий: операторы цикла и условия. Хотя, в отличие от «обычных» языков декларативного программирования, они используются существенно реже.

Условное выполнение кода производится при помощи оператора `if`:

```
if (cond) cons.expr else alt.expr
```

Здесь `cond` – логический вектор длины 1, `cons. expr`, `alt. expr` выражения, которые будут выполнены в случае, если `cond` истинно или ложно соответственно. Значение `NA` для условия не допускается. Размер вектора в условии больше 1, то используется только *первый элемент* и выдается предупреждение.

Данный факт является источником многочисленных недоразумений: оператор `==` работает покомпонентно, поэтому в конструкции вида

```
if (v1 == v2) do.something(v1, v2)
```

будет происходить сравнение только первых элементов векторов,а не их содержимого целиком. Ожидаемого поведения можно достичь припомощи функций `identical()` или `all. equal()` в

зависимости от того, требуется точное или приближенное равенство векторов.

Циклы в R реализуются при помощи операторов while и for. Синтаксис первого следующий:

```
while (cond) expr
```

Здесь cond – условие выполнения тела цикла (правила вычисления условия совпадают с таковыми для оператора if), expr — собственно, тело цикла.

Оператор for позволяет перечислить все элементы последовательности:

```
for (idx in seq) expr
```

Здесь idx – переменная цикла, seq — перечисляемая последовательность, а expr тело цикла. Последовательность seq вычисляется до первого выполнения тела цикла, ее переопределение внутри тела цикла не влияет на число итераций, аналогично назначение какого-либо значения переменной idx не влияет на следующие итерации цикла. Цикл можно прервать оператором break, закончить текущую итерацию и перейти к следующей – оператором next.

### *R как СУБД*

Несмотря на то что к R написаны интерфейсы ко многим системам управления базами данных (СУБД) и даже есть специальный пакет sqldf, который позволяет управлять таблицами данных посредством команд SQL, стоит обратить внимание и на базовые особенности R, позволяющие превратить его, практически не расширяя, в организатор связанных (реляционных) текстовых баз данных.

Тем, кто знаком с основами языка SQL, могут показаться интересными соответствия между операторами этого языка и командами R. Некоторые из них приведены в таблице:

| SELECT | [ и subset() |
|---|---|
| JOIN | merge() |
| GROUP BY | aggregate(), tapply() |
| DISTINCT | unique() и duplicated() |
| ORDER BY | order(), sort(), rev() |
| WHERE | which(), %in%, == |
| LIKE | grep() |
| INSERT | rbind() |
| EXCEPT | ! и - |

Соответствия эти не однозначные и не абсолютные, но хорошо видно, что операции SQL могут быть без особых проблем выполнены «изнутри» R. Единственным серьезным недостатком является то, что многие из этих функций выполняются весьма медленно. Грешит этим и очень важная функция merge(), которая позволяет связывать разные таблицы на основании общей колонки («ключа» в терминологии базданных).

Вот пример пользовательской функции, которая работает быстрее:

```
> recode <- function(var, from, to)
> + {
> + x <- as.vector(var)
> + x.tmp <- x
> + for (i in 1:length(from)) {x <- replace(x, x.tmp == from[i],
> + to[i])}
> + if(is.factor(var)) factor(x) else x
> + }
```

Она делает то, чего не умеет делать встроенная функция replace(), –перекодирование

всех значений по определенному правилу:

```
> replace(rep(1:10,2), c(2,3,5), c("a","b","c"))
[1] "1"  "a"  "b"  "4"  "c"  "6"  "7"  "8"  "9"  "10"
"1"  "2"  "3"  "4"  "5"
[16] "6"  "7"  "8"  "9"  "10"
> recode(rep(1:10,2), c(2,3,5), c("a","b","c"))
[1] "1"  "a"  "b"  "4"  "c"  "6"  "7"  "8"  "9"  "10"
"1"  "a"  "b"  "4"  "c"
[16] "6"  "7"  "8"  "9"  "10"
```

Как видите, `replace()` заменил только первые значения, в то время как `recode()` заменил их все.

Теперь мы можем оперировать несколькими таблицами как одной. Это очень важно для иерархически организованных данных. Например, мы можем работать в разных регионах и всюду делать похожие операции (скажем, что-то измерять). Тогда удобнее иметь не одну таблицу, а две: в первой будут данные по регионам, а во второй – данные измерений объектов. Для связывания таблиц нужно, чтобы в каждой из них была одна и та же колонка, например, номер региона. Вот как это можно организовать:

```
> locations <- read.table("eq-l.txt", h=T, sep=";")
> measurements <- read.table("eq-s.txt", h=T, sep=";")
> head(locations)
     N.POP WHERE    SPECIES
1      1 Tverskaja  arvense
2      2 Tverskaja  arvense
3      3 Tverskaja  arvense
4      4 Tverskaja  arvense
5      5 Tverskaja pratense
6      6 Tverskaja palustre
> head(measurements)
 N.POP DL.R DIA.ST  N.REB N.ZUB DL.OSN.Z DL.TR.V DL.BAZ DL.PER
 1 1  424  2.3    13    12    2.0    5      3.0   25
 2 1  339  2.0    11    12    1.0    4      2.5   13
 3 1  321  2.5    15    14    2.0    5      2.3   13
 4 1  509  3.0    14    14    1.5    5      2.2   23
 5 1  462  2.5    12    13    1.1    4      2.1   12
   1  350  1.8     9     9    1.1    4      2.0   15
> loc.N.POP <- recode(measurements$N.POP, locations$N.POP,
> + as.character(locations$SPECIES))
> head(cbind(species=loc.N.POP, measurements))
  species N.POP DL.R DIA.ST N.REB N.ZUB DL.OSN.Z DL.TR.V DL.BAZ
1 arvense 1     424  2.3     13    12    2.0       5      3.0
2 arvense 1     339  2.0     11    12    1.0       4      2.5
3 arvense 1     321  2.5     15    14    2.0       5      2.3
4 arvense 1     509  3.0     14    14    1.5       5      2.2
5 arvense 1     462  2.5     12    13    1.1       4      2.1
6 arvense 1     350  1.8      9     9    1.1       4      2.0
...
```

Здесь показано, как работать с двумя связанными таблицами и командой `recode()`. В

одной таблице записаны местообитания (locations), а в другой — измерения растений (measurements). Названия видов есть только в первой таблице. Если мы хотим узнать, каким видам какие признаки соответствуют), то надо слить первую и вторую таблицы. Можно использовать для этого `merge()`, но `recode()` работает быстрее и эффективнее. Надо только помнить о типе данных, чтобы факторы не превратились в цифры. Ключом в этом случае является колонка `N.POP` (номер местообитания).

### *Правила переписывания. Векторизация*

Встроенные операции языка R *векторизованы*, то есть выполняются поэлементно. В таком случае достаточно быстро встает вопрос, каким образом осуществляются операции в случае, если операнды имеют разную длину (например, при сложении вектора длины 2 и длины 4). За это отвечают так называемые *правила переписывания* (recycling rules):

- *Длина результата совпадает с длиной операнда наибольшей длины.*
- *Если длина операнда меньшей длины делит длину второго операнда, то такой операнд повторяется (переписывается) столько раз, сколько нужно до достижения длины второго операнда. После этого операция производится поэлементно над операндами одинаковой длины.*
- *Если длина операнда меньшей длины не является делителем длины второго операнда (то есть она не укладывается целое число раз в длину большего операнда), то такой операнд повторяется столько раз, сколько нужно для перекрытия длины второго операнда. Лишние элементы отбрасываются, производится операция и выводится предупреждение.*

Как следствие этих правил, операции типа сложения числа (то есть вектора единичной длины) с вектором выполняются естественным образом:

```
> 2 + c(3, 5, 7, 11)
[1]  5  7  9 13
> c(1, 2) + c(3, 5, 7, 11)
[1]  4  7  8 13
> c(1, 2, 3) + c(3, 5, 7, 11)
[1]  4  7 10 12
Warning message:
In c(1, 2, 3) + c(3, 5, 7, 11) :
longer object length is not a multiple of shorter objectlength
```

Большинство встроенных функций языка R так или иначе векторизованы, то есть выдают «естественный» результат при передаче вкачестве аргумента вектора. К этому необходимо стремиться при написании собственных функций, так как это, как правило, является ключевым фактором, влияющим на скорость выполнения программы. Разберем простой пример, написанный человеком хорошо знакомым с языком типа C, но малознакомым с R:

```
> p <- 1:20
> lik <-0
> for (i in length(p))lik <-lik + log(p[i])
```

Это же самое действие можно реализовать существенно проще и короче (кроме того, обеспечив корректную работу в случае, если вектор `p` имел бы нулевую длину):

```
> lik <sum(log(p))
```

Отметим, что «проще» имеется в виду не только с точки зрения количества строк кода, но и вычислительной сложности: первый образецкода выполняется полностью на интерпретаторе, второй же использует эффективные и быстрые встроенные функции.

Второй образец кода работает потому, что функции `log()` и `sum()` векторизованы.

Функция `log()` векторизована в обычном смысле: скалярная функция применяется поочередно к каждому элементу вектора, таким образом результат `log(c(1,2))` идентичен результату `c(log(1), log(2))`.

Функция `sum()` векторизована в несколько ином смысле: она берет

на вход вектор и считает что-то, зависящее от вектора целиком. В данном случае вызов `sum(x)` полностью эквивалентен выражению `x[1] + x[2] + ... + x[length(x)]`.

Очень часто векторизация кода появляется сама собой за счет наличия встроенных функций, правил переписывания для арифметических операций и т. п. Однако (особенно часто это происходит при переписывании кода с других языков программирования) код следует изменить для того, чтобы он стал векторизованным. Например, код

```
> v <- NULL
> v2 <- 1:10
> for (i in length(v2))
+ {
+    if (v2[i] %% 2 == 0)
+    {
+       v <-c(v, v2[i]) # 7 строка
+    }
+ }
```

плох сразу по двум причинам: он содержит цикл там, где его можно избежать, и, что совсем плохо, он содержит вектор, растущий внутри цикла: в седьмой строке происходят выделение памяти под новый вектор `v` и копирование в этот новый вектор элементов из старого. Таким образом, вычислительные затраты этого цикла (в терминах числа копирования элементов) пропорциональны квадрату длины вектора `v2`!

Оптимальное же решение в данном случае является очень простым:

```
> v <- v2[v2 %% 2 == 0]
```

Векторизацию отдельной функции можно произвести при помощи функции `Vectorize()`, однако не стоит думать, что это решение всех проблем – это исключительно изменение внешнего интерфейса функции, внутри она по-прежнему будет вызываться для каждого элемента по отдельности (хотя в отдельных случаях такого решения оказывается достаточно).

Стандартной проблемой при векторизации является оператор `if`. Один из вариантов замены его векторизации был рассмотрен выше, но очень часто подобного рода преобразования невозможны. Например, рассмотрим код вида

```
if (x > 1) y <- -1 else y <- 1
```

В случае когда `x` имеет длину 1, получаемый результат вполне соответствует ожиданиям. Однако, как только длина `x` становится больше 1, все перестает работать. В данном случае код можно векторизоватьпри помощи функции `ifelse()`:

```
ifelse(x > 1, -1, 1)
```

В общем случае синтаксис функции следующий:

```
ifelse(cond, vtrue, vfalse)
```

Результатом функции `ifelse()` является вектор той же длины, что и вектор-условие `cond`. В вектор результата записывается элемент вектора `vtrue` в случае, если соответствующий элемент вектора `cond` равен `TRUE`, либо элемент вектора `vfalse`, если элемент `cond` равен `FALSE`. В противном случае в результат записывается `NA`.

Стандартным желанием при выполнении векторизации является использование функции из `apply()`-семейства: `lapply()`, `sapply()` и т. п. В большинстве случаев результат получится гораздо хуже, чем если бы векторизации не было вообще (в этом примере мы предварительно явно

выделили память под вектор v, чтобы исключить влияние менеджера памяти на результат):

```
> d <- 1:1000000
> v <- numeric(length(d))
> system.time(for (i in length(d)) v[i] <- pi*d[i]^2/4)
user  system elapsed
4.463 0.009 4.504
> system.time(v <- sapply(d, function(d) pi*d^2/4))
user  system elapsed
8.062 0.165 8.383
> system.time(v <- pi*d^2/4)
user  system elapsed
0.024 0.001 0.024
```

Такие результаты для времени выполнения вполне объяснимы. Первый фрагмент кода выполняется целиком на интерпретаторе. Последний за счет использования векторизованных операций проводит большую часть времени в ядре среды R (то есть в неинтерпретируемом коде). Второй же фрагмент совмещает худшие стороны первого и третьего фрагментов:

- Функция sapply() исполняется в неинтерпретируемом коде.
- Второй аргумент функции sapply() является интерпретируемой функцией.

Таким образом, для вычисления одного элемента вектора d необходимо запустить интерпретатор для тривиальной функции, выполнить эту функцию и получить результат. Как следствие накладные расходы, необходимые на запуск интерпретатора, доминируют во времени исполнения кода.

Излишняя векторизация может приводить не только к увеличению времени выполнения, но и к существенному расходу памяти. Предположим, что встала задача заменить все отрицательные элементы таблицы данных df на 0. Естественно, это можно сделать в полностью векторизованном виде:

```
df[df < 0] <- 0
```

Однако, как только таблица данных df станет очень большой, эта конструкция будет требовать памяти в два раза больше, нежели размер таблицы данных, тем самым потенциально приводя к нехватке свободной памяти для среды.

Альтернативой может быть заполнение по строкам:

```
for (i in nrow(df)) df[i, df[i, ] < 0] <- 0
```

или же по столбцам:

```
for (i in ncol(df)) df[df[, i] < 0, i] <- 0
```

Выбор того или иного варианта зависит от соотношения числа строки столбцов в таблице данных и от того, что является более важным – скорость исполнения кода или потребление памяти (хотя, как только заканчивается доступная физическая память и начинается использование файла подкачки, ни о какой производительности не может быть и речи).

Наряду с функциями apply()-семейства, существует еще несколько полезных векторизованных функций. Самая интересная из них, наверное, функция do.call(), которая вызывает выражение из своего первого аргумента для каждого элемента своего второго аргумента-списка. Это очень удобно, когда нужно преобразовать список в матрицу или таблицу данных, поскольку ни cbind(), ни rbind() не могут обрабатывать списки. Вот как это делается:

```
> (spisok <- strsplit(c("Вот как это делается",
> + "Другое немного похожее предложение"), " "))
[[1]]
[1] "Вот" "как" "это" "делается"
 [[2]]
```

```
[1] "Другое" "немного" "похожее" "предложение"
> do.call("cbind", spisok)
       [,1]     [,2]
[1,] "Вот"   "Другое"
[2,] "как"   "немного"
[3,] "это"   "похожее"
[4,] "делается" "предложение"
> do.call("rbind", spisok)
      [,1]          [,2]         [,3]         [,4]
[1,] "Вот"          "как"        "это"        "делается"
[2,] "Другое"       "немного"    "похожее"    "предложение"
```

Мы сделали здесь полезную вещь – разобрали две строки текста на слова и разместили их по ячейкам. Это часто требуется при конвертации данных. Обратите внимание на первую команду – она выводит свой результат одновременно и на экран, и в объект spisok! Таков эффект наружных круглых скобок. Это похоже на то, что делает в UNIX команда tee.

***Отладка***

Возможности отладки кода в среде R достаточно разнообразны. Однако они могут показаться несколько непривычными для тех, кто уже пользовался интегрированными средами разработки для других языков. Эти отличия вполне объяснимы: все механизмы отладки должны работать одинаково хорошо, как из консольного неитерактивного режима, так и из какой-либо среды-надстройки.

При возникновении ошибки естественно желание (как правило, оновозникает первым) узнать, в какой функции ошибка возникла. Такую возможность предоставляет функция traceback(): она показывает стек вызовов функций до момента возникновения последней ошибки. Отметим, что в последних версиях R сообщения об ошибках существенно улучшились, и в большинстве случаев функция traceback() становится ненужной.

```
> foo <- function(x) { print(1); bar(2) }
> bar <- function(x) { x + a.variable.which.does.not.exist }
> foo(2)
[1] 1
Error in bar(2) : object ’a.variable.which.does.not.exist’
not found
> traceback()
2: bar(2) at #1
1: foo(2)
```

Функция browser() позволяет расставить точки остановки: вызов этой функции внутри кода приводит к появлению мини-отладчика (по сути дела – просто к остановке выполнения программы и вызову консоли R с окружением текущей функции в качестве основного окружения).

Команды отладчика:

c (или cont, или просто нажатие клавиши Enter) приводит к выходу из отладчика и продолжению выполнения кода со следующей строки.

n вводит отладчик в режим пошагового просмотра кода. В этом случае смысл команд c и n меняется:

n приводит к выполнению текущей строки и остановке на следующей;

c выполняет код до конца текущего контекста, то есть до закрывающей фигурной скобки, конца цикла, условного оператора, до выхода из функции и т. п.

where выводит стек вызовов функций до текущей.

Q приводит к завершению работы как отладчика, так и выполняемого кода.

Все остальные выражения, набранные в консоли отладчика, интерпретируются как обычные выражения языка R, исполненные в окружении вызванной функции: в частности, если набрать имя объекта и нажать Enter, то этот объект будет напечатан; вызов функции ls() позволяет получить имена всех переменных в окружении вызванной функции (в случае, если потребуется вывести содержимое переменных с именами, совпадающими с именами команд отладчика, это можно сделать посредством явного вызова функции print() с соответствующей переменной в качестве аргумента).

Функция recover() обладает схожей функциональностью, но позволяет просматривать и отлаживать не только функцию, из которой она была вызвана, но и любую активную в данный момент (то есть любую по стеку вызовов). Впрочем, ее основное предназначение — работа в качестве postmortem-отладчика, вызов этой функции можно «повесить» на всякое возникновение ошибки:

```
options(error = recover)
foo <- function(x) { print(1); bar(2) }
bar <- function(x) { x + a.variable.which.does.not.exist }foo(2)
[1] 1
Error in bar(2) : object 'a.variable.which.does.not.exist' not
found
Enter a frame number, or 0 to exit1: foo(1)
2: bar(2) Selection:
...
```

Функция debug() позволяет поставить глобальный *флаг отладки* на функцию: при любом ее вызове произойдет переход в отладочный режим. Команды отладчика те же, что и при вызове функции browser() в пошаговом режиме. Флаг отладки остается на функции до тех пор,пока не будет снят при помощи функции undebug().

Функция debugonce() позволяет поставить флаг отладки на функцию только на одно выполнение.

```
foo <function(x) { print(1); bar(2) }
bar <function(x) { x + a.variable.which.does.not.exist }
debug(bar)
foo(2)
[1] 1
debugging in: bar(2)
debug: {
   x + a.variable.which.does.not.exist
}
Browse[2]>
```

Функция debug() не требует модификации исходного кода для отладки. Аналогичную (только более широкую функциональность) предоставляет функция trace(), позволяющая вставить вызовы произвольных функций (например, browser() или recover()) в произвольные места других функций. Мы не будем детально останавливаться на всех ее (весьма обширных) возможностях. Рассмотрим только несколько примеров.

- Вставка вызова произвольного кода при выходе из данной функции осуществляется при помощи аргумента exit (значением аргумента может быть произвольное невычисленное выражение, не только имя функции)

```
> foo <- function(x) { print(1); }
```

```
> trace(foo, exit = browser)
> foo(1)
[1] 1
Tracing foo(1) on exit
Called from: eval(expr, envir, enclos)
Browse[1]>
```

- В сложных случаях можно отредактировать текст функции и вставить в нужные места код отладки посредством передачи аргумента edit:

```
> foo <- function(x) { print(1); }
> trace(foo, edit = TRUE)
```

Отменить трассировку функции можно вызовом функции untrace().

Интерактивную отладку предоставляет пакет debug. Среди его возможностей стоит отметить поддержку множественных окон отладки,содержащих выполняемый код, возможность назначения точек остановки (в том числе по условию) и т. д. Всю информацию работе с этим пакетом, можно получить из встроенной справки:

```
> install.packages("debug") # Установка пакета debug
> library(debug)    # Подключение пакета
> help(package=debug)      # Вызов справки
```

### *Элементы объектно-ориентированного программирования*

Объектно-ориентированное программирование является простой, но, в то же время, достаточно мощной идеей, существенно упрощающей работу с похожими сущностями. Язык R предоставляет достаточно богатые инструменты для объектно-ориентированного программирования (ООП). Более того, в рамках языка существуют две различные концепции ООП: *версии 3* (S3) и *версии 4* (S4). Мы кратко остановимся исключительно на версии S3 и только в объеме, необходимом для понимания, «что тут вообще происходит».

ООП в стиле S3 выглядит достаточно просто. Функция может быть *универсальной* (generic), то есть производить различные действия в зависимости от типа переданного аргумента. Универсальные функции связаны с одним или несколькими *методами* (methods). Метод – это, собственно, функция, которая вызывается для объекта данного типа. Таким образом, универсальные функции осуществляют диспетчеризацию вызова в зависимости от типа переданного объекта на функцию-метод. Типичными примерами таких универсальных функций, с которыми приходится иметь дело постоянно, являются функции print(), plot(), summary().

За «объектно-ориентированность» отвечает атрибут class, являющийся просто текстовым вектором. Этот вектор просматривается слева направо, при этом ищется функция-метод, отвечающая данному «классу». Если таковая функция найдена, то она вызывается. В противном случае (если просмотрен весь вектор class) вызывается метод по умолчанию. В случае отсутствия такового выдается сообщение об ошибке. Имя функции-метода обычно состоит из имени универсальной функции, точки и имени класса. Так, функция-метод print() для класса data.frame имеет имя print.data.frame(), а функция-метод по умолчанию называется print.default().

Наследование при такой модели ООП происходит тривиальным образом: достаточно в качестве атрибута class использовать вектор длины 2 и больше. Например, в случае если бы нам захотелось создать свой собственный объект, со структурой, похожей на таблицу данных, то мы могли бы использовать в качестве атрибута class нечто типа

```
c("custom.data.frame", "data.frame")
```

Таким образом, мы могли бы переопределить часть функций-методов для нашего класса (например, print() и summary()), по при этом оставить остальные функции от таблицы данных

(например, [).

Получить список всех методов для данной универсальной функцииможно при помощи methods:

```
> methods(summary)
[1] summary.aov                  summary.aovlist*
[3] summary.aspell*             summary.check_packages_in_dir*
[5] summary.connection           summary.data.frame
…
```

Подробную информацию о модели ООП S3 можно получить в разделе справки S3Methods, по модели S4 — в разделе Methods.

## 3. Что такое данные и зачем их обрабатывать.

### *Получение данных*

Способов получения данных много, и особенно важными являются *наблюдения* и *эксперименты*.

*Наблюдением* называется такой способ получения данных, при котором воздействие наблюдателя на наблюдаемый объект сведено к минимуму. *Эксперимент* тоже включает наблюдение, но сначала на наблюдаемый объект оказывается заранее рассчитанное воздействие. Для наблюдения очень важно это «сведение воздействия к минимуму». Если этого не сделать, мы получим данные, отражающие не свойства объекта, а его реакцию на наше воздействие.

От правильного подбора данных серьезным образом будет зависеть качество получаемых данных. Собственно говоря, есть два основных принципа составления выборки: *повторность* и *рандомизация*.

*Повторности* нужны для того, чтобы быть более уверенными в полученных результатах. Существуют, специальные методы, которые позволяют посчитать, сколько надо собрать данных, для того чтобы с определенной вероятностью высказать некоторое утверждение. Это так называемые «тесты мощности».

*Рандомизация* — еще одно условие создания выборки. Каждый объект генеральной совокупности должен иметь равные шансы попасть в выборку. Очень часто исследователи полагают, что выбрали свои объекты случайно (проделали рандомизацию), в то время как на самом деле их материал был подобран иначе.

Анализ данных необходим всегда, когда результат неочевиден,и часто даже тогда, когда он *кажется очевидным*. Разберемся, к каким последствиям он может привести.

- *Анализ данных умеет давать общие характеристики для больших выборок, такие, как среднее значение и дисперсия. Среднее значение определяет вокруг чего «разбросаны» данные, дисперсия – насколько сильно они разбросаны.*
- *Сравнение данных при помощи статистических тестов позволяет выяснить, насколько велика вероятность, что различия между группами вызваны случайными причинами.*
- *Третий тип результата – это сведения о взаимосвязях. Изучение взаимосвязей – это самый серьезный и самый развитый раздел анализа данных. Существует множество методик выяснения и, главное, проверки «качества» связей.*
- *Многомерный анализ данных позволяет выполнить проверку качества классификации объектов.*

Существует и другой подход к результатам анализа данных, когда все методы делятся на *предсказательные и описательные*. К первой группе методов относится все, что можно статистически оценить, тоесть выяснить, *с какой вероятностью* может быть верным или неверным наш вывод. Ко второй — методы, которые «просто» сообщают информацию о

данных без подтверждения какими-либо вероятностными методами. В последние годы все для большего числа методов находятсяспособы их вероятностной оценки, и поэтому первая группа все время увеличивается.

***Как загружать данные***

Сначала о том, как набрать данные прямо в R. Можно, например, использовать команду `c()`:

```
> a <- c(1,2,3,4,5)
> a
[1] 1 2 3 4 5
```

Здесь мы получаем объект `a`, состоящий из чисел от одного до пяти. Можно использовать команды `rep()`, `seq()`, `scan()`, а также оператор двоеточия:

```
> b <- 1:5
> b
[1] 1 2 3 4 5
```

Можно воспользоваться встроенной в R подпрограммой — электронной таблицей наподобие сильно упрощенного Excel, для этого надо набрать команду

```
> data.entry(b)
```

В появившейся таблице можно редактировать данные «на месте», то есть, все, что вы ввели, непосредственно скажется на содержании объекта. Это несколько противоречит общим концепциям, заложенным в R, и поэтому есть похожая функция `de(b)`, которая не меняет объект, а выдает результат «наружу». Если же есть таблица данных, можно использовать команды `fix()`.

Функция `edit()`, вызванная для другого типа объекта, запустит его редактирование в текстовом редакторе.

Однако лучше всего научиться загружать в R файлы, созданные при помощи других программ, скажем, при помощи Excel.

В целом данные, которые надо обрабатывать, бывают двух типов — текстовые и бинарные. Текстовые данные для статистической обработки — это обычно текстовые таблицы, в которых каждая строка соответствует строчке таблицы, а колонки определяются при помощи разделителей (обычно пробелов, знаков табуляции, запятых или точек с запятой). Для того чтобы R «усвоил» такие данные, надо, во первых, убедиться, что текущая рабочая папка в R и та папка, откуда будут загружаться ваши данные, — это одно и то же. Для этого в запущенной сессии R надо ввести команду

```
> getwd()
[1] "d:/Documents/Documents"
```

Поменять рабочую папку можно командой:

```
> setwd("d:\\wrk\\temp")
```

Данные загружает команда `read. table()`:

```
> read.table("mydata.txt", sep=";", head=TRUE)
1 1 2 3
2 4 5 6
3 7 8 9
```

Функция `read. table()` очень хороша, но не настолько умна, чтобы определять формат данных «на лету». Поэтому придётся выяснить нужные сведения заранее, скажем, в том же самом текстовом редакторе. Есть и способ выяснить это через R, для этого используется команда `file. show(″mydata. txt″)`.

Она выводит свои данные таким же образом, как и help(),– отдельным окном или в отдельном режиме основного окна.

В R многие команды, в том числеи read.table(), имеют умалчиваемые значения аргументов. Например, значение sep по умолчанию ″ ″, что в данном случае означает, что разделителем является любое количество пробелов или знаков табуляции. Отметим еще несколько важных вещей:

Русский текст в файлах обычно читается без проблем. Если, все же возникли проблемы, то лучше файл перевести в кодировку UTF-8 и добавить соответствующую опцию:

```
> read.table("mydata.txt",sep=";",head=TRUE,encoding="UTF-8")
```

Данные, которые выдают многие русифицированные программы, в качестве десятичного разделителя обычно используют запятую, а не точку. В этих случаях надо указывать аргумент dec:

```
>read.table("data/mydata3.txt", dec=",", sep=";", h=T)
```

Обратите внимание на сокращенное обозначение аргумента и его значения.

Итак, один из возможных способов работы с данными в R такой:

- *данные набирают в какой-нибудь «внешней» программе;*
- *записывают их в текстовый файл с разделителями;*
- *загружают как объект в R и работают с этим объектом, возможно, внося изменения;*
- *если были изменения, командой write.table() записывают обратно в файл;*
- *импортируют как текстовый файл в исходную программу и работают дальше.*

Такой способ позволяет использовать все преимущества программ по набору электронных таблиц и текстовых редакторов.

С электронными таблицами в текстовом формате больших проблем обычно не возникает. Разные экзотические текстовые форматы, как правило, можно преобразовать к «типичным», если не с помощью R, то с помощью каких-нибудь текстовых утилит (вплоть до «тяжеловесов» типа языка Perl). А вот с «посторонними» бинарными форматами дело гораздо хуже. Здесь возникают прежде всего проблемы, связанные с закрытыми и/или недостаточно документированными форматами, такими, например, как формат программы MS Excel. Вообще говоря, ответ на вопрос, как прочитать бинарный формат в R, обычно сводится к тому, чтобы найти способ, как преобразовать бинарные данные в текстовые таблицы. Второй путь – найти способ прочитать данные в R без преобразования. В R есть пакет foreign, который может читать бинарные данные, выводимые различными пакетами. Чтобы узнать про это подробнее, надо загрузить пакет (командой library(foreign)) и вызвать общую справку по его командам, например, командой help(package=foreign).

Что касается, например, форматов MS Excel, то есть не меньше пяти разных способов, как загружать в R эти файлы, но все они имеют ограничения. Из всех способов наиболее привлекательным представляется обмен с R через буфер. Если открыт Excel, то можно скопировать в буфер любое количество ячеек, а потом загрузить их в R командой

```
> read.table("clipboard").
```

R может загружать изображения. Для этого есть сразу несколько пакетов, наиболее разработанный из них – pixmap. R может также загружать и другие данные (пакеты maps, maptools) и вообще много чего еще. Чтобы загрузить такие пакеты, нужно, в отличие от foreign, их сначала скачать из репозитория. Для этого используется меню (под Windows) или команда install.packages().

У R есть собственный бинарный формат. Он быстро записывается и быстро загружается, но его нельзя использовать с другими программами:

```
> x <- "apple"
> save(x, file="x.rd") # Сохранить объект "x"
> exists("x")
[1] TRUE
> rm(x)
> exists("x")
[1] FALSE
> dir()
[1] "x.rd" ...
> load("x.rd") # Загрузить объект "x"
> x
[1] "apple"
```

Для R написано множество интерфейсов к базам данных, в частности для MySQL, PostgresSQL и sqlite (последний может вызываться прямо из R, см. документацию к пакетам RSQLite и sqldf).

***Как сохранять результаты***

Начинающие работу с R обычно просто копируют результаты работы (скажем, данные тестов) из консоли R в текстовый файл. И, действительно, на первых порах этого может быть достаточно. Однако рано или поздно возникает необходимость сохранять объемные объекты (скажем, таблицы данных), созданные в течение работы. Можно, как уже говорилось, использовать внутренний бинарный формат, но это не всегда удобно. Лучше всего сохранять таблицы данных в виде текстовых таблиц, которые потом можно будет открывать другими (в частности, офисными) приложениями. Для этого служит команда `write.table()`:

```
> write.table(file="trees.csv",trees,row.names=FALSE, sep=";",
+ quote=FALSE)
```

В текущую папку будет записан файл `trees.csv`, созданный из встроенной в R таблицы данных `trees`. «Встроенная таблица» означает, что эти данные доступны в R безо всякой загрузки, напрямую. Кстати говоря, узнать, какие таблицы уже встроены, можно командой `data()`.

А что, если надо записать во внешний файл результаты выполнения команд? В этом случае используется команда `sink()`:

```
> sink("1.txt", split=TRUE)
> 2+2
[1] 4
> sink()
```

Тогда во внешний файл запишется строчка «`[1] 4`», то есть, результат выполнения команды. Параметр `split=TRUE` задаётся для того, чтобы вывести данные еще и на экран.

Для сохранения истории команд служит `savehistory()`, а для сохранения всех созданных объектов —`save.image()`.

***R как калькулятор***

Самый простой способ использования R — это арифметические вычисления. Важной особенностью является то, что вычисления производятся не с одним числом, а сразу с

набором (вектором). Например, выражение

```
> log10(sqrt(sum(c(1,2,6)))^2+1)
```

вычисляется так:

- *Создается вектор: c(1,2,6).*
- *Подсчитывается сумма его членов: 1+2+6=9.*
- *Извлекается квадратный корень: sqrt(9)=3.*
- *Он возводится в квадрат: 3^2=9.*
- *К результатуприбавляется 1: 9+1=10.*
- *Вычисляется десятичный логарифм: log10(10)=1.*

### *Графики*

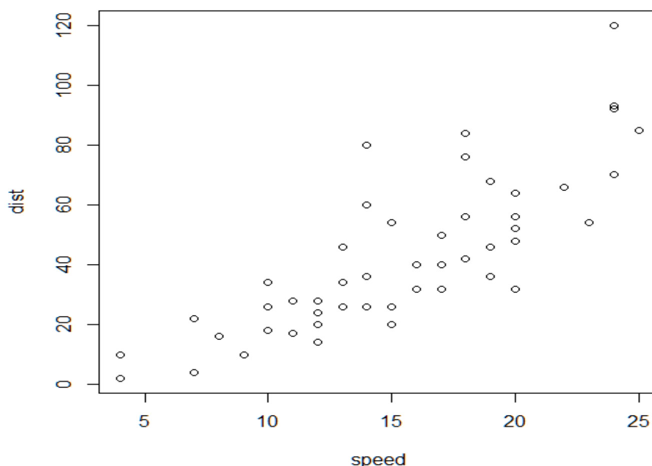
Одним из основных достоинств статистического пакета служит разнообразие типов графиков, которые он может построить. В базовом наборе есть несколько десятков типов графиков, еще больше в рекомендуемом пакете `lattice`, и еще больше — в пакетах с CRAN. При этом они достаточно хорошо настраиваются, то есть пользователь легко может настраивать их на свой вкус. Остановимся на нескольких фундаментальных принципах, понимание которых должно существенно облегчить построение графиков в R.

Рассмотрим такой пример:

```
> plot(1:20, main="Заголовок")
> legend("topleft", pch=1, legend="Мои любимые точки")
```

Первая команда рисует график «с нуля», а вторая добавляет к уже нарисованному графику детали. Это и есть два типа графических команд, используемых в базовом графическом наборе R. `plot()` — основная графическая команда Она распознает тип объекта, который подлежит рисованию, и строит соответствующий график. Например, в приведенном примере `1:20` — это последовательность чисел от 1 до 20, то есть *вектор,* а для «одиночного» вектора предусмотрен график, где по оси абсцисс — индексы, а по оси ординат — сами эти элементы. Если в аргументе команды будет что-то другое, будет, скорее всего, построен иной график. Вот пример:

```
> plot(cars)
> title(main="Автомобили двадцатых годов")
```




Здесь тоже есть команды обоих типов, но немного иначе оформленные. Таблица данных `cars` (встроенная в R) состоит из двух колонок — `speed` и `distance`. Функция

plot() автоматически нарисует коррелограмму (scatterplot), где по оси x откладывается значение одной переменной (колонки), а по оси y – другой, и присвоит осям имена этих колонок.

Очень многие пакеты расширяют графические функции R. Можно, например, применить к тем же данным (1:20) функцию qplot() из пакета ggplot2 (рис. 3):

```
> library(ggplot2)
> qplot(1:20, 1:20, main="Заголовок")
```

Команда library() загружает нужный пакет. Если пакета ggplot2 в системе нет, то его нужно скачать из Интернета и установить. Делается это через меню или командой install.packages("ggplot2").

***Графические устройства***

Когда вводится команда plot(), то R открывает так называемое экранное графическое устройство и начинает вывод на него. Если следующая команда того же типа (то есть не добавляющая), то R «сотрет» старое изображение и начнет выводить новое. Команда dev.off()закрывает графическое окно. Экранных устройствв R предусмотрено несколько, в каждой операционной системе — свое. Но все это не так важно, пока вы не захотите строить графики и сохранять их в файлы автоматически. Тогда надо будет познакомиться с другими графическими устройствами. Их несколько (количество опять-таки зависит от операционной системы),а пакеты предоставляют еще около десятка. Работают они так:

```
> png(file="1-20.png", bg="transparent")
> plot(1:20)
> dev.off()
```

Команда png() открывает одноименное графическое устройство, причем задается параметр, включающий прозрачность базового фона. Как только вводится команда dev.off(), устройство закрывается,и на диске появляется файл 1-20.png. R поддерживает и векторные форматы, например PDF. Здесь, однако, могут возникнуть специфические для русскоязычного пользователя трудности со шрифтами. Вот как надо «правильно» создавать PDF-файл, содержащий русский текст:

```
> pdf("1-20.pdf", family="NimbusSan", encoding="CP1251.enc")
> plot(1:20, main="Заголовок")
> dev.off()
> embedFonts("1-20.pdf")
```

Как видим, требуется указать, какой шрифт мы будем использовать, а также кодировку. Затем нужно закрыть графическое устройство и *встроить в полученный файл* шрифты. В противном случае кириллица может не отобразиться! Важно отметить, что шрифт «NimbusSan» и возможность встраивания шрифтов командой embedFonts() обеспечивается взаимодействием R с «посторонней» программой Ghostscript, в поставку которой входят шрифты, содержащие русские буквы.

Кроме PDF, R «знает» и другие векторные форматы, например, PostScript, xfig и picTeX. Есть отдельный пакет svglite, который поддерживает популярный векторный формат SVG. График в этом формате можно, например, открыть и видоизменить в свободном векторном редакторе Inkscape.

***Графические опции***

Как уже говорилось, графика в R настраивается в очень широкихпределах. Один из способов настройки — это видоизменение встроенных графических опций. Вот, к

примеру, распространенная задача — нарисовать два графика один над другим на одном рисунке. Чтобы это сделать, надо изменить исходные опции — разделить пространство рисунка на две части, примерно так:

```
> old.par <- par(mfrow=c(2,1))
> hist(cars$speed, main="")
> hist(cars$dist, main="")
> par(old.par)
```

Ключевая команда здесь — `par()`. В первой строчке изменяется один из ее параметров, `mfrow`, который регулирует, сколько изображений и как будет размещено на «листе». Значение `mfrow` по умолчанию — `c(1, 1)`, то есть один график по вертикали и один по горизонтали. Что- бы не печатать каждый раз команду `par()` со всеми ее аргументами, запоминаем старое значение в объекте `old.par`, а в конце возвращаем состояние к запомненному. Команда `hist()` строит график-гистограмму.

***Интерактивная графика***

Интерактивная графика позволяет выяснить, где именно на графике расположены нужные точки, поместить объект (скажем, подпись) в нужное место, а также проследить «судьбу» одних и тех же точек на разных графиках. Кроме того, если данные многомерные, то можно вращать облако точек в плоскости разных переменных, с тем чтобы выяснить структуру данных. Вот так, например, можно добавлять подписи в указанную мышкой область графика (после того, как введена вторая команда, надо щелкнуть левой кнопкой мыши на какой-нибудь точке в середине, а затем щелкнуть в любом месте графика правой кнопкой мыши и в получившемся меню выбрать `Stop`):

```
> plot(1:20)
> text(locator(), "Моя любимая точка", pos=4)
```

Интерактивная графика других типов реализована командой `identify()`, а также дополнительными пакетами, в том числе `iplot`, `manipulate`, `playwith`, `rggobi`, `rpanel`, и `TeachingDemos`.

# 4. Типы данных

Чтобы обрабатывать данные, надо не просто их получить. Надо еще перевести их на язык цифр. Сделать это можно самыми разными способами, иногда — удачно, иногда — с натяжками.

***Интервальныеданные***

Очень важно, что, например, температура изменяется плавно и непрерывно. Это значит, что если у нас есть две разные температуры, то всегда можно представить температуру, промежуточную между ними. Любые два показателя температуры или расстояния представляют собой интервал, куда «умещается» бесконечное множество других показателей. Такие данные называются *интервальными*.

Не всегда, однако, интервальные данные изменяются плавно и непрерывно от минус бесконечности к плюс бесконечности. Например, температура соответствует не прямой, а лучу, потому что она ограничена абсолютным нулем. Но на остальном протяжении этого луча показатели температуры можно уподобить действительным числам. Еще интереснее измерять углы. Угол изменяется непрерывно, но вот после $360^{\circ}$ следует $0^{\circ}$ — вместо прямой имеем отрезок без отрицательных значений. Есть даже особый раздел статистики, так называемая *круговая статистика* (directional, or circular statistics), которая работает с углами.

Другой случай связан с данными представленными натуральными числами. У этих чисел есть отношение порядка, но вот промежуточное значение не существует. Они представляют собой *дискретный* тип данных.

С интервальностью и непрерывностью данных неразрывно связан важный водораздел в методах статистики. Эти методы часто делят на две большие группы: *параметрические* и *непараметрические*. Параметрические тесты предназначены для обработки так называемых параметрических данных. Для того чтобы данные считались параметрическими, должны одновременно выполняться три условия:

- *распределение данных близко к нормальному;*
- *выборка – большая (обычно не менее 30 наблюдений);*
- *данные – интервальные и непрерывные.*

Если *хотя бы одно* из этих условий не выполняется, данные считаются непараметрическими и обрабатываются непараметрическими методами. Несомненным достоинством непараметрических методов является их способность работать с «неидеальными» данными. Зато параметрические методы имеют бóльшую мощность (то есть при прочих равных вероятность не заметить существующую закономерность ниже). Этому есть простое объяснение: непараметрические данные (если они, как это очень часто бывает, дискретны) имеют свойство «скрывать» имеющиеся различия, объединяя отдельные значения в группы.

Так как параметрические методы доступнее непараметрических (например, в курсах статистики изучают в основном параметрические методы), то часто хочется как-нибудь «параметризировать» данные. На распределение данных мы, естественно, никак повлиять не можем (хотя иногда преобразования данных могут «улучшить» распределение и даже сделать его нормальным). Что мы можем сделать, так это постараться иметь достаточно большой объем выборки (что, как вы помните, увеличивает и ее репрезентативность), а также работать с непрерывными данными.

В R интервальные данные представляют в виде числовых векторов (*numerical vectors*). Чаще всего один вектор — это одна выборка. Допустим, у нас есть данные о росте семи сотрудников небольшой компании. Вот так можно создать из этих данных простейший числовой вектор:

```
> x <- c(174, 162, 188, 192, 165, 168, 172.5)
```

Собственно, R и работает в основном с объектами и функциями. У объекта имеется своя структура:

```
> str(x)
num [1:7] 174 162 188 192 165 168 172.5
```

То есть, `x` — это числовой (`num`, «numeric») вектор. В R нет скаляров, «одиночные» объекты трактуются как векторы из одного элемента. Вот так можно проверить, вектор ли перед нами:

```
> is.vector(x)
[1] TRUE
```

Вообще говоря, в R есть множество функций «`is.что-то()`» для подобной проверки, например:

```
> is.numeric(x)
[1] TRUE
```

А еще есть функции конверсии «`as.что-то()`». Называвать объекты можно, в принципе, как угодно, но лучше придерживаться некоторых правил:

- *Использовать для названий только латинские буквы, цифры и точку (имена объектов не должны начинаться с точки или цифры).*
- *Помнить, что R чувствителен к регистру, X и x — это разные имена.*
- *Не давать объектам имена, уже занятые распространенными функциями (типа c()), а также ключевыми словами (особенно T, F, NA, NaN, Inf, NULL, а также pi —*

*единственное встроенное в R число).*

Для создания «искусственных» векторов очень полезен оператор «`:`», обозначающий интервал, а также функции создания последовательностей («sequences») `seq()` и повторения («replications») `rep()`.

***Шкальные данные***

Если интервальные данные можно получить непосредственно (например, посчитать) или при помощи приборов (измерить), то шкальные данные не так просто сопоставить числам. Предположим, нам надо составить, а затем проанализировать данные опросов об удобстве мебели. Ясно, что «удобство» — вещь субъективная, но игнорировать ее нельзя, надо что-то с ней сделать. Как правило, «что-то» – это шкала, где каждому баллу соответствует определенное описание, которое и включается в опрос. Кроме того, в такой шкале все баллы часто можно ранжировать, в нашем случае — от наименее удобной мебели к наиболее удобной.

Число, которым обозначено значение шкалы, — вещь более чем условная. По сути, можно взять любое число. Зато есть отношение порядка и, более того, подобие непрерывности. Например, если удобную во всех отношениях мебель мы станем обозначать цифрой «5», а несколько менее удобную — цифрой «4», то в принципе можно представить, какая мебель могла бы быть обозначена цифрой «4.5». Именно поэтому к шкальным данным применимы очень многие из тех методов, которые используются для обработки интервальных непрерывных данных. Однако к числовым результатам обработки надо подходить с осторожностью, всегда помнить об условности значений шкалы.

Больше всего трудностей возникает, когда данные измерены в разных шкалах. Разные шкалы часто очень нелегко перевести друг в друга. По умолчанию R будет распознавать шкальные данные как обычный числовой вектор. Однако для некоторых задач может потребоваться преобразовать его в так называемый упорядоченный фактор. Если же стоит задача создать шкальные данные из интервальных, то можно воспользоваться функцией `cut(..., ordered=TRUE)`.

Для статистического анализа шкальных данных всегда требуются непараметрические методы. Если же хочется применить параметрические методы, то нужно иначе спланировать сбор данных, чтобы в результате получить интервальные данные. Например, при исследованиях размеров листьев не делить их визуально на «маленькие», «средние» и «большие», а измерить их длину и ширину при помощи линейки.

Вот еще один пример перекодирования. Предположим, вы изучаете высоту зданий в различных городах земного шара. Можно в графе «город» написать его название. Это, конечно, проще всего, но тогда вы не сможете использовать эту переменную в статистическом анализе данных. Можно закодировать города цифрами в порядке их расположения, например с севера на юг (если вас интересует географическая изменчивость высоты зданий в городе) — тогда получатся шкальные данные, которые можно обработать непараметрическими методами. И наконец, каждый город можно обозначить его географическими координатами или расстоянием от самого южного города — тогда мы получим интервальные данные, которые можно будет попробовать обработать параметрическими методами.

***Номинальные данные***

*Номинальные* данные (их часто называют «категориальными»), в отличие от шкальных, нельзя упорядочивать. Поэтому они еще дальше от чисел в строгом смысле слова, чем шкальные данные. Вот, например, пол. Даже если мы присвоим мужскому и женскому полам какие-нибудь числовые значения (например, 1 и 2), то из этого не будет следовать, что какой-то пол «больше» другого. Да и промежуточное значение (1.5) здесь непросто представить. В принципе, можно обозначать различные номинальные показатели не цифрами, а буквами, целыми словами или специальными значками — суть от этого не изменится.

Обычные численные методы для номинальных данных неприменимы. Однако существуют способы их численной обработки. Самый простой — это счет, подсчет количеств данных разного типа в общем массиве данных. Эти количества и производные от них числа уже гораздо легче поддаются обработке.

Особый случай как номинальных, так и шкальных данных — *бинарные данные*, то есть такие, которые проще всего передать числами 0 и 1. Например, ответы «да» и «нет» на вопросы анкеты. Или наличие/отсутствие чего-либо. Бинарные данные иногда можно упорядочить (скажем, наличие и отсутствие), иногда – нет (скажем, верный и неверный ответы). Можно бинарные данные представить и в виде логического вектора, то есть набора значений `TRUE` или `FALSE`. Самая главная польза от бинарных данных — в том, что в них можно перекодировать практически все остальные типы данных (хотя иногда при этом будет потеряна часть информации). После этого к ним можно применять специальные методы анализа. Для обозначения номинальных данных в R есть несколько способов,разной степени «правильности». Во-первых, можно создать текстовый вектор:

```
> sex <- c("male","female","male","male","female","male","male")
> is.character(sex)
[1] TRUE
> is.vector(sex)
[1] TRUE
> str(sex)
chr [1:7] "male" "female" "male" "male" "female" "male" ...
> sex
[1] "male" "female" "male" "male" "female" "male" "male"
```

Объект-ориентированные команды R кое-что понимают про объект `sex`, например команда `table()`:

```
> table(sex)
female      male
2           5
```

А вот команда `plot()`, увы, не умеет ничего хорошего сделать с таким вектором. Сначала нужно сообщить R, что этот вектор надо рассматривать как *фактор* (то есть номинальный тип данных). Делается

это так:

```
> sex.f <- factor(sex)
> sex.f
[1] male female male male female male male
Levels: female male
```

И теперь команда `plot()` уже «понимает», что ей надо делать —строить столбчатую диаграмму:

```
> plot(sex.f)
```

Это произошло потому, что перед нами специальный тип объекта, предназначенный для категориальных данных, — фактор с двумя уровнями (градациями) (`levels`):

```
> is.factor(sex.f)
[1] TRUE
> is.character(sex.f)
[1] FALSE
> str(sex.f)
Factor w/ 2 levels "female","male": 2 1 2 2 1 2 2
```

Очень многие функции R (скажем, тот же самый plot()) предпочитают факторы текстовым векторам, при этом некоторые умеют конвертировать текстовые векторы в факторы, а некоторые — нет, поэтому надо быть внимательным. Факторы (в отличие от текстовых векторов) можно легко преобразовать в числовые значения:

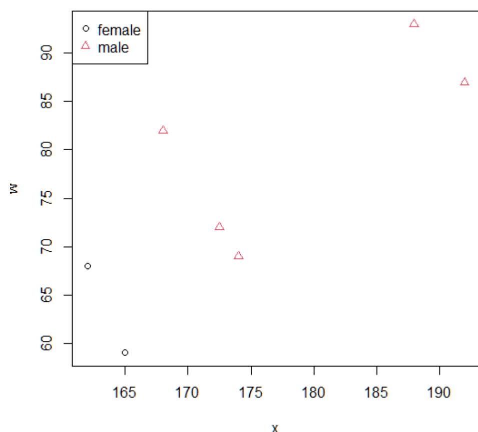
```
> as.numeric(sex.f)
[1] 2 1 2 2 1 2 2
```

Зачем это нужно, становится понятным, если рассмотреть вот такой пример. Положим, кроме роста, у нас есть еще и данные по весу сотрудников:

```
> x <- c(174, 162, 188, 192, 165, 168, 172.5)
> w <- c(69, 68, 93, 87, 59, 82, 72)
```

Мы хотим построить такой график, на котором были бы видны одновременно рост, вес и пол. Вот как это можно сделать:

```
> plot(x, w, pch=as.numeric(sex.f), col=as.numeric(sex.f))
> legend("topleft", pch=1:2, col=1:2, legend=levels(sex.f))
```




Здесь pch и col – параметры, предназначенные для определения соответствия типа значков и их цвета на графике. Таким образом, в зависимости от того, какому полу принадлежит данная точка, она будет изображена кружком или треугольником и черным или красным цветом. При условии, разумеется, что все три вектора соответствуют друг другу.

Факторы можно упорядочивать, превращая их в один из вариантов шкальных данных. Введем четвертую переменную — размер маек для тех же самых гипотетических восьмерых сотрудников:

```
> m <- c("L", "S", "XL", "XXL", "S", "M", "L")
> m.f <- factor(m)
> m.f
[1] L S XL  XXL S M L
Levels: L M S XL XXL
```

Как видно, уровни расположены просто по алфавиту, а нам надо, чтобы S(small) шел первым. Кроме того, надо как-то сообщить R, что перед нами – шкальные данные. Делается это так:

```
> m.o <- ordered(m.f, levels=c("S", "M", "L", "XL", "XXL"))
> m.o
[1] L S XL XXL S  M L
Levels: S < M < L < XL < XXL
```

Теперь R «знает», какой размер больше. Это может сыграть критическую роль — например, при вычислениях коэффициентов корреляции.

Работая с факторами, нужно помнить и об одной опасности. Если возникла необходимость перевести фактор в числа, то вместо значений вектора мы получим числа, соответствующие уровням фактора! Чтобы этого не случилось, надо сначала преобразовать фактор, состоящий из значений-чисел, в текстовый вектор, а уже потом — в числовой:

```
> a <- factor(3:5)
> a
[1] 3 4 5
Levels: 3 4 5
> as.numeric(a) # Неправильно!
[1] 1 2 3
> as.numeric(as.character(a)) # Правильно!
[1] 3 4 5
```

Когда файл данных загружается при помощи команды `read. table()`,то все столбцы, где есть хотя бы одно нечисленное значение, будут преобразованы в факторы. Если хочется этого избежать (для того, например, чтобы не столкнуться с вышеописанной проблемой), то нужно задать дополнительный параметр: `read. table(..., as. is=TRUE)`.

***Доли, счет и ранги: вторичные данные***

Из названия ясно, что такие данные возникают в результате обработки первичных, исходных данных.

Наибольшее применение вторичные данные находят при обработке шкальных и в особенности номинальных данных, которые нельзя обрабатывать «в лоб». Например, счет («counts») — это просто количество членов какой-либо категории. Такие подсчеты и составляют суть статистики в бытовом смысле этого слова. Проценты (доли, «rates») тоже часто встречаются в быту, так что подробно описывать их, наверное, не нужно. Одно из самых полезных их свойств — они позволяют вычленить числовые закономерности там, где исходные данные отличаются по размеру.

Для того чтобы визуализировать счет и проценты, придумано немало графических способов. Самые распространенные – это графики-пироги и столбчатые диаграммы. Однако надо заметить, что столбчатые диаграммы и в особенности «пироги» — неудачный способ представления информации. Многочисленные эксперименты доказали, что читаются такие графики гораздо хуже остальных.

На замену им в R есть так называемые точечные графики (dotplots).

```
> romashka.t <- read.table("romashka.txt", sep="\t")
> romashka <- romashka.t$V2
> dotchart(romashka)
```

Если счет и доли получают обычно из номинальных данных, то отношения и ранги «добывают» из данных количественных. Отношения особенно полезны в тех случаях, когда изучаемые явления или вещи имеют очень разные абсолютные характеристики. Например, вес людей довольно трудно использовать в медицине напрямую, а вот соотношение между ростом и весом очень помогает в диагностике ожирения.

Чтобы получить ранги, надо упорядочить данные по возрастанию и заменить каждое значение на номер его места в полученном ряду.

Например, обычная методика вычисления медианы основывается на рангах. Ранги особенно широко применяются при анализе шкальных и непараметрических интервальных данных. Ранговые методики анализа, как правило, устойчивы, но менее чувствительны, чем параметрические. Это и понятно, ведь в процессе присваивания рангов часть информации теряется:

```
> a1 <- c(1,2,3,4,4,5,7,7,7,9,15,17)
> a2 <- c(1,2,3,4,5,7,7,7,9,15,17)
> names(a1) <- rank(a1)
> a1
1 2 3 4.5 4.5 6 8 8 8 10 11 12
1 2 3 4   4   5 7 7 7  9 15 17
> names(a2) <- rank(a2)
> a2
1  2  3  4  5  7  7  7  9 10 11
1  2  3  4  5  7  7  7  9 15 17
```

Как видно, ранги могут быть не целыми, а половинными. Это бывает тогда, когда одинаковых чисел четное число. Кроме того, у одинаковых чисел ранги тоже одинаковые. Это называется «ties», буквально «ничья». Ничья эта несколько мешает при выполнении некоторых статистических операций, например, при вычислении непараметрических тестов, основанных на рангах:

```
> wilcox.test(a2)
[... результаты теста пропущены ...]
Warning message:
In wilcox.test.default(a2):cannot compute exact p-value with ties
```

В этих случаях R всегда сообщает о проблеме.

***Пропущенные данные***

Не существует ни совершенных наблюдений, ни совершенных экспериментов. Чем больше массив данных, тем больше вероятность встретить в нем различные недочеты, прежде всего *пропущенные данные*, которые возникают по тысяче причин — от несовершенства методик, от случайностей во время фиксации данных, от ошибок компьютерных программ и т. д. Строго говоря, пропущенные данные бывают нескольких типов. Самый понятный — это «unknown», неизвестное значение, когда данные просто не зафиксированы (или потеряны, что, увы, бывает нередко). Есть еще «both», когда в процессе наблюдений возникло состояние, отвечающее сразу нескольким значениям. Например, если мы наблюдаем за погодой и отмечаем единицей солнечный день, а нулем — пасмурный, то день с переменной облачностью будет «both» (обычно это значит, что методика наблюдений разработана плохо). Наконец, «not applicable», неприменимое значение, возникает тогда, когда обнаружено нечто, логически не совместимое с тем признаком, который надо фиксировать.

Опыт показывает, что обойтись без пропущенных данных практически невозможно. Более того, их отсутствие в сколько-нибудь крупном массиве данных может служить основанием для сомнений в их достоверности.

В R пропущенные данные принято обозначать двумя большими буквами латинского алфавита «NA». Предположим, что у нас имеется результат опроса тех же самых семи сотрудников. Их спрашивали, сколько в среднем часов они спят, при этом один из опрашиваемых отвечать отказался, другой ответил «не знаю», а третьего в момент опроса просто не было на рабочем месте. Так возникли пропущенные данные:

```
> h <- c(8, 10, NA, NA, 8, NA, 8)
> h
[1]  8 10 NA NA  8 NA  8
```

Как видим, NA надо вводить без кавычек, а R нимало не смущается, что среди цифр находится вроде бы текст. Отметим, что пропущенные данные очень часто столь же разнородны, как и в нашем примере. Однако кодируются они одинаково, и об этом не

нужно забывать.

Чтобы высчитать среднее от «непропущенной» части вектора, можно поступить одним из двух способов:

```
> mean(h, na.rm=TRUE)
[1] 8.5
> mean(na.omit(h))
> [1] 8.5
```

Первый способ разрешает функции `mean()` принимать пропущенные данные, а второй делает из вектора `h` временный вектор без пропущенных данных (они просто выкидываются из вектора). Какой из способов лучше, зависит от ситуации.

Часто возникает еще одна проблема — как сделать подстановку пропущенных данных, скажем, заменить все `NA` на среднюю по выборке. Вот распространенное (но не очень хорошее) решение:

```
> h[is.na(h)] <- mean(h, na.rm=TRUE)
> h
[1]  8.0 10.0  8.5  8.5  8.0  8.5  8.0
```

В левой части первого выражения осуществляется индексирование, то есть выбор нужных значений `h` — таких, которые являются пропущенными (`is.na()`). После того как выражение выполнено, «старые» значения *исчезают навсегда*, поэтому рекомендуем сначала сохранить старый вектор, скажем, под другим названием:

```
> h.old <- h
```

Есть много других способов замены пропущенных значений, в том числе и очень сложные, основанные на регрессионном, а также дискриминантном анализе. Некоторые из них реализованы в пакетах `mice` и `cat`, существует даже пакет `MissingDataGUI`, предоставляющий графический интерфейс для «борьбы» с пропущенными данными.

### *Выбросы и как их найти*

К сожалению, после набора данных возникают не только «пустые ячейки». Очень часто встречаются просто ошибки. Чаще всего это опечатки, которые могут возникнуть при ручном наборе. Если данных немного, то можно попытаться выявить такие ошибки вручную. Хуже, если объем данных велик – скажем, более тысячи записей. В этом случае могут помочь методы обработки данных, прежде всего те, которые рассчитаны на выявление выбросов(outliers). Самый простой из них – нахождение минимума и максимума, а для номинальных данных — построение таблицы частот. Но такие методы помогают лишь отчасти. Легко найти опечатку в таблице данных роста человека, если кто-то записал 17 см вместо 170 см. Однако ее практически невозможно найти, если вместо 170 см написано 171 см. В этом случае остается надеяться лишь на статистическую природу данных – чем их больше, тем менее заметны будут ошибки, и на так называемые робастные (устойчивые к выбросам) методы обработки.

### *Основные принципы преобразования данных*

Если в исследовании задействовано несколько разных типов данных — параметрические и непараметрические, номинальные и непрерывные, проценты и подсчеты и т. п., то самым правильным будет привести их к какому-то «общему знаменателю».

Иногда такое преобразование сделать легко. Даже номинальные данные можно преобразовать в непрерывные, если иметь достаточно информации. Скажем, пол (номинальные данные) можно преобразовать в уровень мужского гормона тестостерона в крови (непрерывные); правда, для этого нужна дополнительная информация. Распространенный вариант преобразования – обработка дискретных данных так, как будто они непрерывные. В целом это безопасно, но иногда приводит к неприятным последствиям. Совершенно неприемлемый вариант – преобразование номинальных данных в шкальные. Если данные по своей природе не упорядочены, то их искусственное упорядочение может

радикально сказаться на результате.

Часто данные преобразуют для того, чтобы они больше походили на параметрические. Если у распределения данных длинные «хвосты», если график распределения лишь отчасти «колоколообразный», можно прибегнуть к логарифмированию.

Это, наверное, самое частое преобразование. В графических командах R есть даже специальный аргумент

```
..., log="ось",
```

где вместо слова ось надо подставить `x` или `y`, и тогда соответствующая ось графика отобразится в логарифмическом масштабе.

Вот самые распространенные методы преобразований с указаниями,как их делать в R (мы предполагаем, что ваши данные находятся в векторе `data`):

- *Логарифмическое: log(data + 1). Если распределение скошено вправо, то может дать нормальное распределение. Может также делать более линейными зависимости между переменными и уравнивать дисперсии. «Боится» нулей в данных, поэтому рекомендуется прибавлять единицу.*
- *Квадратного корня: sqrt(data). Похоже по действию на логарифмическое. «Боится» отрицательных значений.*
- *Обратное: 1/(data + 1). Эффективно для стабилизации дисперсии. «Боится» нулей.*
- *Квадратное: data^2. Если распределение скошено влево, можетдать нормальное распределение. Линеаризует зависимости и выравнивает дисперсии.*
- *Логит: log(p/(1-p)). Чаще всего применяется к пропорциям. Линеаризует так называемую сигмовидную кривую. Кроме логитпреобразования, для пропорций часто используют и арксинус преобразование, asin(sqrt(p))*

При обработке многомерных данных очень важно, чтобы они были одной размерности. Ни в коем случае нельзя одну колонку в таблице записывать в миллиметрах, а другую — в сантиметрах.

В многомерной статистике широко применяется и нормализация данных— приведение разных колонок к общему виду (например, к одному среднему значению). Вот как можно, например, нормализовать два разномасштабных вектора:

```
> a <- 1:4
> b <- seq(100, 400, 100)
> d <- data.frame(a, b)
> d
      a   b
1     1  100
2     2  200
3     3  300
4     4  400
> scale(d)
         a          b
[1,] -1.1618950 -1.1618950
[2,] -0.3872983 -0.3872983
[3,]  0.3872983  0.3872983
[4,]  1.1618950  1.1618950
```

Как видим, команда `scale()` приводит векторы «к общему знаменателю». Обратите внимание на то, что поскольку вектор `b` был, по сути, просто увеличенным в сто раз вектором `a`, после преобразования они стали совершенно одинаковыми.

### *Матрицы, списки и таблицы данных*

Матрицы — очень распространенная форма представления данных, организованных в форме таблицы. В R матриц, как таковых, по сути, нет. Матрица – это просто специальный тип вектора, обладающий некоторыми добавочными свойствами (атрибутами), позволяющими интерпретировать его как совокупность строк и столбцов. Предположим, мы хотим создать простейшую матрицу 2 2. Для начала создадим ее из числового вектора:

```
> m <- 1:4
> m
[1] 1 2 3 4
> ma <- matrix(m, ncol=2, byrow=TRUE)
> ma
     [,1] [,2]
[1,]  1     2
[2,]  3     4
> str(ma)
int [1:2, 1:2] 1 3 2 4
> str(m)
int [1:4] 1 2 3 4
```

Как видно, структура объектов `m` и `ma`, различается, по сути, лишь их выводом на экран. Еще очевиднее единство между векторами и матрицами прослеживается, если создать матрицу несколько иным способом:

```
> mb <- m
> mb
[1] 1 2 3 4
> attr(mb, "dim") <- c(2,2)
> mb
     [,1] [,2]
[1,]  1     3
[2,]  2     4
```

Выглядит как некий фокус. Однако все просто: мы присваиваем вектору `mb` атрибут `dim` («dimensions», размерность) и устанавливаем значение этого атрибута в `c(2, 2)`, то есть 2 строки и 2 столбца.

Это лишь два способа создания матриц, в действительности их гораздо больше. Очень популярно, например, «делать» матрицы из векторов-колонок или строк при помощи команд `cbind()` или `rbind()`. Если результат нужно «повернуть» на 90 градусов (транспонировать), используется команда `t()`.

Наиболее распространены матрицы, имеющие два измерения, однако никто не препятствует сделать многомерную матрицу (массив):

```
> m3 <- 1:8
> dim(m3) <- c(2,2,2)
```

`m3` — это трехмерная матрица (или, по-другому, трехмерный массив). Естественно, показать в виде таблицы ее нельзя, поэтому R выводит ее на экран в виде серии таблиц. Многомерные матрицы в R принято называть «arrays».

**Списки** — еще один важный тип представления данных. Создавать их, особенно на первых порах, скорее всего, не придется, но знать их особенности необходимо – прежде всего потому, что очень многие функциив R выдают «наружу» именно списки.

```
> l <- list("R", 1:3, TRUE, NA, list("r", 4))
```

Список — это своего рода ассорти. Он может состоять из чего угодно, в том числе и из других списков.

Элементы вектора выбираются, как мы помним, при помощи функции — квадратной скобки:

```
> h[3]
[1] 8.5
```

Элементы матрицы выбираются так же, только используются два аргумента (номер строки и номер столбца):

```
> ma[2, 1]
[1] 3
```

А вот элементы списка выбираются тремя различными методами.

Во-первых, можно использовать квадратные скобки:

```
> l[1]
[[1]]
[1] "R"
> str(l[1])
List of 1
$ : chr "R"
```

Здесь очень важно, что полученный объект *тоже будет списком*.

Во-вторых, можно использовать двойные квадратные скобки:

```
> l[[1]]
[1] "R"
> str(l[[1]])
chr "R"
```

В этом случае полученный объект будет того типа, какого он был бы до объединения в список (поэтому первый объект будет текстовым вектором, а вот пятый — списком).

И в-третьих, для индексирования можно использовать имена элементов списка. Но для этого сначала надо их создать:

```
> names(l) <- c("first", "second", "third", "fourth", "fifth")
> l$first
[1] "R"
> str(l$first)
chr "R"
```

Для выбора по имени употребляется знак доллара, а полученный объект будет таким же, как при использовании двойной квадратной скобки. На самом деле имена в R могут иметь и элементы вектора, истроки и столбцы матрицы:

```
> names(w)<-c("Коля","Женя","Петя","Саша","Катя","Вася","Жора")
> w
Коля Женя Петя   Саша Катя   Вася Жора
69    68    93    87    59    82    72
> rownames(ma) <- c("a1","a2")
> colnames(ma) <- c("b1","b2")
> ma
   b1 b2
a1  1  2
a2  3  4
```

Единственное условие — все имена должны быть разными. Однако знак доллара можно использовать только со списками. Элементы вектора по имени можно отбирать так:

```
> w["Женя"]
Женя
68
```

Теперь о самом важном типе представления данных — **таблицах данных** (data frame). Именно таблицы данных больше всего похожи на электронные таблицы Excel и аналогов, и поэтому с ними работают чаще всего. Таблицы данных — это гибридный тип представления, одномерный список из векторов *одинаковой длины*. Таким образом, каждая таблица данных — это список колонок, причем внутри одной колонки все данные должны быть одноготипа (а вот сами колонки могут быть разного типа). Проиллюстрируем это на примере созданных ранее векторов:

```
> m <- c("L", "S", "XL", "XXL", "S", "M", "L")
> m.f <- factor(m)
> m.o <- ordered(m.f, levels=c("S", "M", "L", "XL", "XXL"))
> x <- c(174, 162, 188, 192, 165, 168, 172.5)
> w <- c(69, 68, 93, 87, 59, 82, 72)
> names(w)<-c("Коля","Женя","Петя","Саша","Катя","Вася","Жора")
> d <- data.frame(weight=w, height=x, size=m.o, sex=sex.f)
> d
    weight height size  sex
Коля  69    174     L    male
Женя  68    162     S    female
Петя  93    188     XL   male
Саша  87    192     XXL  male
Катя  59    165     S    female
Вася  82    168     M    male
Жора  72    172.5   L    male
> str(d)
'data.frame':      7 obs. of  4 variables:
$ weight: num  69 68 93 87 59 82 72
$ height: num  174 162 188 192 165 168 172.5
$ size:Ord.factor w/5 levels "S"<"M"<"L"<"XL"<..: 3 1 4 5 1 2 3
$ sex : Factor w/ 2 levels "female","male": 2 1 2 2 1 2 2
```

Поскольку таблица данных является списком, к ней применимы все методы индексации списков. Таблицы данных можно индексировать и как двумерные матрицы. Вот несколько примеров:

```
> d$weight
[1] 69 68 93 87 59 82 72
> d[[1]]
[1] 69 68 93 87 59 82 72
> d[,1]
[1] 69 68 93 87 59 82 72
> d[,"weight"]
[1] 69 68 93 87 59 82 72
```

Очень часто бывает нужно отобрать несколько колонок. Это можно сделать разными способами:

```
> d[,2:4]
```

```
     height size   sex
Коля 174    L      male
Женя 162    S      female
Петя 188    XL     male
Саша 192    XXL    male
Катя 165    S      female
Вася 168    M      male
Жора 172.5  L      male
```

Тот же результат даст команда

```
> d[,-1]
```

К индексации имеет прямое отношение еще один тип данных R — *логические векторы*. Как, например, отобрать из нашей таблицы только данные, относящиеся к женщинам? Вот один из способов:

```
> d[d$sex=="female",]
     weight  height size sex
Женя 68      162     S   female
Катя 59      165     S   female
```

Чтобы отобрать нужные строки, мы поместили перед запятой логическое выражение d$sex==female. Его значением является логический вектор:

```
> d$sex=="female"
[1] FALSE  TRUE FALSE FALSE  TRUE FALSE FALSE
```

Таким образом, после того как «отработала» селекция, в таблице данных остались только те строки, которые соответствуют TRUE, то есть строки 2 и 5. Знак «==», а также знаки «&», «|» и «!» используются для замены соответственно «равен?», «и», «или» и «не».

Более сложным случаем отбора является сортировка таблиц данных. Для сортировки одного вектора достаточно применить команду sort(), а вот если нужно, скажем, отсортировать наши данные сначала по полу, а потом по росту, приходится применить операцию посложнее:

```
> d[order(d$sex, d$height),]
        weightheight size sex
  Женя   68    162     S    female
  Катя   59    165     S    female
  Вася   82    168     M    male
  Жора   72    172.5  L    male
  Коля   69    174     L    male
  Петя   93    188    XL   male
  Саша   87    192    XXL  male
```

Команда order() создает не логический, а числовой вектор, который соответствует будущему порядку расположения строк.

Заключая разговор о типах данных, следует отметить, что эти типы вовсе не так резко разграничены. Поэтому если трудно с первого взгляда сказать, к какому именно типу относятся ваши данные, нужно вспомнить главный вопрос — *насколько хорошо данные соотносятся с числовой прямой?* Если соотношение хорошее, то данные, скорее всего, интервальные и непрерывные, если плохое — шкальные или даже номинальные. И во-вторых, не следует забывать, что весьма часто удается найти способ преобразования данных в требуемый тип.

## 5. Пример работы в R

Рассмотрим пример по обработке данных. Попробуйте «просто» последовательно

выполнить приведенные ниже команды. Желательно не копировать их откуда-либо, а набрать с клавиатуры: таким способом запомнить и, стало быть, научиться будет гораздо проще. Хорошо, если про каждую выполненную команду вы прочтете соответствующий раздел справки (напоминаем, это делается командой help(команда)).

Все команды будут относиться к файлу данных о воображаемых жуках, состоящему из четырех столбцов (признаков): пол жука (POL: самки = 0 и самцы = 1), цвет жука: (CVET: красный=1, синий=2, зеленый=3), вес жука в граммах (VES) и длина жука в миллиметрах (ROST):

```
POL  CVET  VES    ROST
0    1     10.68  9.43
1    1     10.02  10.66
0    2     10.18  10.41
1    1     8.01   9
0    3     10.23  8.98
1    3     9.7    9.71
1    2     9.73   9.09
0    3     11.22  9.23
1    1     9.19   8.97
1    2     11.45  10.34
```

Создаем на жестком диске рабочую директорию и копируем туда файл данных в текстовом формате с расширением *.txt и разделителем-табуляцией. Пусть файл называется *zhuki.txt*.

Открываем программу R. Указываем директорию, где находится файл данных, при помощи меню: Файл -> Изменить папку -> выбираем директорию, или печатаем команду *setwd(...),* в аргументе которой должен стоять полный путь к вашей директории (для указания пути надо использовать прямые слэши «/».

Читаем файл данных (создаем в памяти программы объект под названием data, который представляет собой копию файла данных):

```
> data <- read.table("zhuki.txt", h=TRUE)
```

Если в файле дробная часть отделена запятой, а не точкой, то нужно указать это в параметре *dec*:

```
> data <- read.table("zhuki.txt", h=TRUE, dec=",")
```

Посмотрим получившуюся таблицу (если файл большой, то можно использовать специальную команду head(data)):

```
> data
```

Внутри R вносить изменения в данные не очень удобно. Разумно вносить их в файл данных (открыв его, например,в Excel), а потом заново читать его в R.

Посмотрим на структуру файла данных. Сколько объектов (obs. =observations), сколько признаков (variables), как названы признаки и в каком порядке они следуют в таблице:

```
> str(data)
'data.frame':   10 obs. of  4 variables:
 $ POL : int  0 1 0 1 0 1 1 0 1 1
 $ CVET: int  1 1 2 1 3 3 2 3 1 2
 $ VES : num  10.68 10.02 10.18 8.01 10.23 ...
 $ ROST: num  9.43 10.66 10.41 9 8.98 ...
```

Отметьте для себя, что POL и CVET загружены как числа, в то время как на самом деле это *номинальные* данные.

Создадим в памяти еще один объект с данными, куда отберем данные только для самок

*(POL = 0):*

```
> data.f <- data[data$POL == 0,]
```

А теперь — отдельный объект с данными для крупных (больше10 мм) самцов:

```
> data.m.big <- data[data$POL == 1 & data$ROST > 10,]
```

Знаки «==» и «&» используются для формирования нужного логического выражения определяющего критерий отбора. Кроме того, обязательно нужны квадратные скобки, а поскольку данные табличные, то внутри квадратных скобок обязательна запятая, разделяющая выражения для строк и столбцов.

Добавим еще один признак к нашему файлу: удельный вес жука (отношение веса жука к его длине) – `VES.R`:

```
> data$VES.R <- data$VES/data$ROST
```

Проверьте, что новый признак появился, при помощи уже использованной нами команды `str(data)` (стрелка вверх!).

Новый признак добавлен только к данным в памяти программы. Чтобы сохранить изменения в файл нужно написать:

```
> write.table(data, "zhukinew.txt", quote=FALSE)
```

Охарактеризуем выборку. Для этого посмотрим на основные характеристики каждого признака:

```
> summary(data)
   POL           CVET            VES              ROST             VES.R
Min.   :0.0   Min.   :1.00   Min.   : 8.010   Min.   : 8.970   Min.  :0.8900
1st Qu.:0.0   1st Qu.:1.00   1st Qu.: 9.707   1st Qu.: 9.023   1st Qu.:0.9832
Median :1.0   Median :2.00   Median :10.100   Median : 9.330   Median :1.0475
Mean   :0.6   Mean   :1.90   Mean   :10.041   Mean   : 9.582   Mean   :1.0496
3rd Qu.:1.0   3rd Qu.:2.75   3rd Qu.:10.568   3rd Qu.:10.182   3rd Qu.:1.1263
Max.   :1.0   Max.   :3.00   Max.   :11.450   Max.   :10.660   Max.   :1.2156
```

Конечно, команду `summary()`, как и многие прочие, можно применять как ко всем данным, так и к любому отдельному признаку:

```
> summary(data$VES)
```

А можно вычислять эти характеристики по отдельности:

```
> min(data$VES)
> max(data$VES)
> median(data$VES)
> mean(data$VES)
> colMeans(data)
```

К сожалению, предыдущие команды не работают, если есть пропущенные значения. Для того чтобы посчитать среднее для каждого из признаков, избавившись от пропущенных значений, надо ввести

```
> mean(data, na.rm=TRUE)
```

Кстати, строки с пропущенными значениями можно удалить из таблицы данных так:

```
> data.o <- na.omit(data)
```

Иногда бывает нужно вычислить сумму всех значений признака:

```
> sum(data$VES)
```

или сумму всех значений одной строки (попробуем на примере второй):

```
> sum(data[2,])
```

... или сумму значений всех признаков для каждой строки:

```
> apply(data, 1, sum)
```

Для номинальных признаков имеет смысл посмотреть, сколько раз встречается в выборке каждое значение признака (заодно узнаем, какие значения признак принимает):

```
> table(data$POL)
```

А теперь выразим частоту встречаемости значений признака в процентах, приняв за 100% общее число объектов (округлив значения процентов до целых чисел):

```
> round(100*table(data$POL)/length(data$POL), 0)
```

Одна из основных характеристик разброса данных вокруг среднего значения (наряду с абсолютным и межквартильным разбросом) – стандартное отклонение (standard deviation). Вычислим его:

```
> sd(data$VES)
```

Вычислим и безразмерный коэффициент вариации (CV):

```
> 100*sd(data$VES)/mean(data$VES)
```

Можно вычислять характеристики любого признака отдельно для самцов и для самок. Попробуем на примере среднего арифметического для веса:

```
> tapply(data$VES, data$POL, mean)
```

Посмотрим, сколько жуков разного цвета среди самцов и самок:

```
> table(data$CVET, data$POL)
```

А теперь то же самое, но не в штуках, а в процентах от общего числа жуков:

```
> 100*table(data$CVET, data$POL)/sum(data$CVET, data$POL)
```

И наконец, вычислим средние значения веса жуков отдельно для всех комбинаций цвета и пола (для красных самцов, красных самок, зеленых самцов, зеленых самок...):

```
> tapply(data$VES, list(data$POL, data$CVET), mean)
```

Теперь порисуем диаграммы...

Проверим сначала, как распределены данные, нет ли выбросов. Для этого построим гистограммы для каждого признака:

```
> hist(data$VES, breaks=20)
```

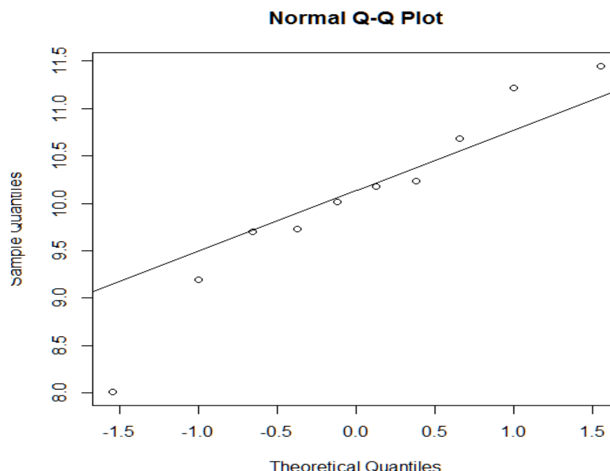
Детализацию гистограммы можно изменять, варьируя число интервалов (`breaks`).

Можно задать точную ширину интервалов значений признака на гистограмме (зададим ширину в 20 единиц, а значения признака пусть изменяются от 0 до 100):

```
> hist(data$VES, breaks=c(seq(0,100,20)))
```

Более точно проверить нормальность распределения признака можно при помощи двух команд:

```
> qqnorm(data$VES); qqline(data$VES)
```



Чем больше распределение точек на этом графике отклоняется от прямой линии, тем дальше распределение данных от нормального. Кстати, для того чтобы открыть новое графическое окно (новый график будет нарисован рядом со старым, а не вместо него), можно использовать команду `dev.new()`.

Построим диаграмму рассеяния, на которой объекты будут обозначены кружочками. Рост будет по оси абсцисс (горизонтальная ось), вес – по оси ординат (вертикальная ось):

```
> plot(data$ROST, data$VES, type="p")
```

Можно изменять размер кружочков (параметр `cex`). Сравните:

```
> plot(data$ROST, data$VES, type="p", cex=0.5)
```

Можно изменить вид значка, который обозначает объект (параметр pch):

```
>  plot(data$ROST, data$VES, type="p", pch=2)
```

Можно вместо значков обозначить объекты на диаграмме кодом пола (0/1):

```
> plot(data$ROST, data$VES, type="n")
> text(data$ROST, data$VES, labels=data$POL)
```

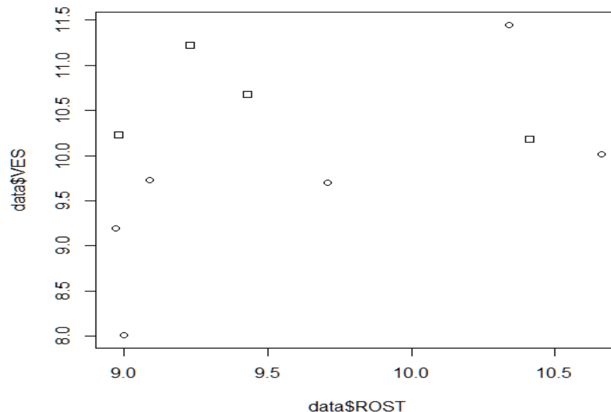
Обе команды здесь действуют на один и тот же график. Первая печатает пустое поле, вторая добавляет туда значки.

Еще можно сделать так, чтобы разные цифры-обозначения имели и разные цвета (мы добавили «+1», иначе бы значки для самок печатались нулевым, прозрачным цветом):

```
> plot(data$ROST, data$VES, type="n")
> text(data$ROST, data$VES, labels=data$POL, col=data$POL+1)
```

А можно самцов и самок обозначить разными значками:

```
> plot(data$ROST, data$VES, type="n")
> points(data$ROST, data$VES, pch=data$POL)
```




Сохраняем график при помощи меню: *Файл -> Сохранить как...*->PNG->graph.png или печатаем *две* команды:

```
> dev.copy(png, filename="graph.png")
> dev.off()
```

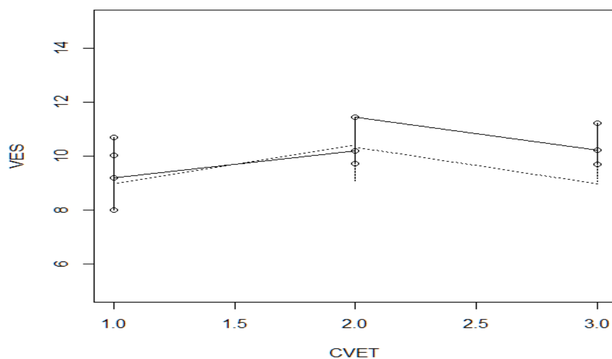
Рисуем коррелограмму:

```
> plot(data[order(data$ROST), c("ROST", "VES")], type="o")
```

Здесь мы отсортировали жуков по росту, потому что иначе вместо графика получилось бы множество пересекающихся линий.

А теперь нарисуем две линии на одном графике:

```
> plot(data[order(data$CVET), c("CVET", "VES")], type="o",ylim=c(5, 15))
> lines(data[order(data$CVET), c("CVET", "ROST")], lty=3)
```



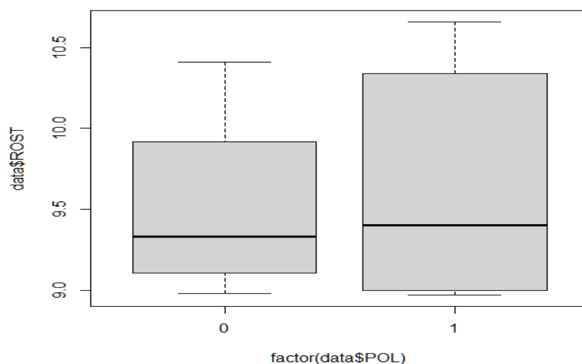

Аргумент ylim задает длину оси ординат (от 0 до 50). Аргумент lty задает тип линии («3» — это пунктир).

Рисуем «ящик-с-усами», или, по-другому, boxplot (он покажет выбросы, минимум–максимум, квартильный разброс и медиану):

```
> boxplot(data$ROST)
```

... а теперь — для самцов и самок по отдельности:

```
> boxplot(data$ROST ~ factor(data$POL))
```




Статистические тесты

Достоверность различий для параметрических данных (тест Стьюдента), для зависимых переменных:

```
> t.test(data$VES, data$ROST, paired=TRUE)
```

... и для независимых переменных:

```
> t.test(data$VES, data$ROST, paired=FALSE)
```

... если нужно сравнить значения одного признака для двух групп:

```
> t.test(data$VES ~ data$POL)
```

Если p-value < 0.05, то различие между выборками достоверно. В R по умолчанию не требуется проверять, одинаков ли разброс данных относительно среднего.

Достоверность различий для непараметрических данных (тест Вилкоксона):

```
> wilcox.test(data$VES, data$ROST, paired=TRUE)
```

Достоверность различий между тремя и более выборками параметрических данных (вариант однофакторного дисперсионного анализа):

```
> oneway.test(data$VES ~ data$CVET)
```

Посмотрим, какие именно пары выборок достоверно различаются:

```
> pairwise.t.test(data$VES, data$CVET, p.adj="bonferroni")
```

А теперь проверим достоверность различий между несколькими выборками

непараметрических данных:

```
> kruskal.test(data$VES ~ data$CVET)
```

Достоверность соответствия для категориальных данных (тест Пирсона, хи-квадрат):

```
> chisq.test(data$CVET, data$POL)
```

Достоверность различия пропорций (тест пропорций):

```
> prop.test(c(sum(data$POL)), c(length(data$POL)), 0.5)
```

Здесь мы проверили – правда ли, что доля самцов достоверно отличается от 50%?

Достоверность линейной связи между параметрическими данными (корреляционный тест Пирсона):

```
> cor.test(data$VES, data$ROST, method="pearson")
```

... и между непараметрическими (корреляционный тест Спирмена):

```
> cor.test(data$VES, data$ROST, method="spearman")
```

Дисперсионный анализ при помощи линейной модели:

```
> anova(lm(data$ROST ~ data$POL))
```

Сохраняем историю команд через меню или при помощи команды

```
> savehistory("zhuki.r")
```

Все введенные вами команды сохранятся в файл с расширением `*.r`,который можно открыть в любом текстовом редакторе и исправлять, дополнять или копировать в строку ввода программы в следующий раз.Этот файл можно выполнить целиком (запустить), для этого используется команда `source(″zhuki.r″)`.

## 6. Анализ одномерных данных

Теперь, наконец, можно обратиться к статистике. Начнем с самых элементарных приемов анализа — вычисления общих характеристик одной-единственной выборки.

### *Как оценивать общую тенденцию*

У любой выборки есть две самые общие характеристики: *центр* (центральная тенденция) и *разброс* (размах). В качестве центра чаще всего используются *среднее* и *медиана*, а в качестве разброса — *стандартное отклонение* и *квартили*. Среднее отличается от медианы прежде всего тем, что оно хорошо работает в основном тогда, когда распределение данных близко к нормальному. Медиана не так зависит от характеристик распределения, как говорят статистики, она более *робастна* (устойчива). Понять разницу легче всего на таком примере. Возьмем опять наших гипотетических сотрудников. Вот их зарплаты (в тыс. руб.):

```
> salary <- c(21, 19, 27, 11, 102, 25, 21)
```

Разница в зарплатах обусловлена, в частности, занимаемой должностью.

```
> mean(salary); median(salary)
[1] 32.28571
[1] 21
```

Получается, что из-за одной высокой зарплаты среднее гораздо хуже отражает «типичную», центральную зарплату, чем медиана. Медиана — это значение, которое отсекает половину упорядоченной выборки. Для того чтобы лучше это показать, вернемся к тем двум векторам, на примере которых в предыдущей главе было показано, как присваиваются ранги:

```
> a1 <- c(1,2,3,4,4,5,7,7,7,9,15,17)
> a2 <- c(1,2,3,4,5,7,7,7,9,15,17)
> median(a1)
[1] 6
> median(a2)
[1] 7
```

В векторе a1 всего двенадцать значений, то есть четное число. В этом случае медиана определяет среднее между двумя центральными числами. У вектора a2 все проще, там одиннадцать значений, поэтому для медианы просто берется середина.

Кроме медианы, для оценки свойств выборки очень полезны *квартили*, то есть те значения, которые отсекают соответственно 0%, 25%, 50%, 75% и 100% от всего распределения данных. Медиана — это третий квартиль (50%). Первый и пятый квартили — это соответственно минимум и максимум, а второй и четвертый квартили используют для робастного вычисления разброса. Можно понятие «квартиль» расширить и ввести специальный термин для значения, отсекающего любой процент упорядоченного распределения (не обязательно по четвертям) — это называется «*квантиль*».

Для характеристики разброса часто используют и параметрическую величину — *стандартное отклонение*. Широко известно «правило трех сигм», которое утверждает, что если средние значения двух выборок различаются больше чем на тройное стандартное отклонение, то эти выборки разные, то есть взяты из разных генеральных совокупностей. Это правило очень удобно, но, к сожалению, подразумевает, что обе выборки должны подчиняться нормальному распределению. Для вычисления стандартного отклонения в R предусмотрена функция sd().

Кроме среднего и медианы, есть еще одна центральная характеристика распределения, так называемая *мода*, самое часто встречающееся в выборке значение. Мода применяется редко и в основном для номинальных данных. Вот как посчитать ее в R (мы использовали для подсчета переменную sex из предыдущей главы):

```
> sex<-c("male","female","male","male","female","male","male")
> t.sex <- table(sex)
> mode <- t.sex[which.max(t.sex)]
> mode
male
5
```

Таким образом, мода нашей выборки — male.

Часто стоит задача посчитать среднее (или медиану) для целой таблицы данных. Есть несколько облегчающих жизнь приемов. Покажем их на примере встроенных данных trees:

```
> attach(trees) # Первый способ
> mean(Girth)
[1] 13.24839
> mean(Height)
[1] 76
> mean(Volume/Height)
[1] 0.3890012
> detach(trees)
> with(trees, mean(Volume/Height)) # Второй способ
[1] 0.3890012
> lapply(trees, mean) # Третий способ
$Girth
[1] 13.24839
$Height
[1] 76
$Volume
```

```
[1] 30.17097
```

Первый способ (при помощи attach()) позволяет присоединить колонки таблицы данных к списку текущих переменных. После этого к переменным можно обращаться по именам, не упоминая имени таблицы. Важно не забыть сделать в конце detach(), потому что велика опасность запутаться в том, что вы присоединили, а что — нет. Если присоединенные переменные были как-то модифицированы, на самой таблице это не скажется.

Второй способ, в сущности, аналогичен первому, только присоединение происходит внутри круглых скобок функции with(). Третий способиспользует тот факт, что таблицы данных — это списки из колонок. Для строк такой прием не сработает, надо будет запустить apply().

Стандартное отклонение, дисперсия (его квадрат) и так называемый межквартильный разброс вызываются аналогично среднему:

```
> sd(salary); var(salary); IQR(salary)
[1] 31.15934
[1] 970.9048
[1] 6
```

Последнее выражение, дистанция между вторым и четвертым квартилями IQR робастен и лучше подходит для примера с зарплатой, чем стандартное отклонение.

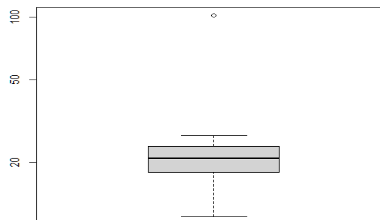
Применим эти функции к встроенным данным trees:

```
> attach(trees)
> mean(Height)
[1] 76
> median(Height)
[1] 76
> sd(Height)
[1] 6.371813
> IQR(Height)
[1] 8
> detach(trees)
```

Видно, что для деревьев эти характеристики значительно ближе друг к другу. Разумно предположить, что распределение высоты деревьев близко к нормальному.

В наших данных по зарплате — всего 7 чисел. А как понять, есть ли какие-то «выдающиеся» значения в данных большого размера? Для этого есть графические функции. Самая простая — так называемый «ящик-с-усами», или boxplot. Для начала добавим к нашим данным еще тысячу гипотетических работников с зарплатой, случайно взятой из межквартильного разброса исходных данных (рис. 9):

```
> new.1000 <- sample((median(salary) - IQR(salary)) :
+ (median(salary) + IQR(salary)), 1000, replace=TRUE)
> salary2 <- c(salary, new.1000)
> boxplot(salary2, log="y")
```



Это интересный пример еще и потому, что в нем впервые представлена техника получения случайных значений. Функция `sample()` способна выбирать случайным образом данные из выборки. В данном случае мы использовали `replace=TRUE`, поскольку нам нужно было выбрать много чисел из гораздо меньшей выборки. Если писать на R имитацию карточных игр, то надо использовать `replace=FALSE`, потому что из колоды нельзя достать опять ту же самую карту.

Как видно, зарплата руководителя представлена высоко расположенной точкой (настолько высоко, что нам даже пришлось вписать параметр `log=″y″`, чтобы нижележащие точки стали видны лучше). Сам бокс, то есть главный прямоугольник, ограниченсверху и снизу квартилями, так что высота прямоугольника — это IQR. Так называемые «усы» по умолчанию обозначают точки, удаленные на полтора IQR. Линия посередине прямоугольника — это, как легко догадаться, медиана. Точки, лежащие вне «усов», рассматриваются как выбросы и поэтому рисуются отдельно.

Есть две функции, которые связаны с боксплотами. Функция `quantile()` по умолчанию выдает все пять квартилей, а функция `fivenum()` — основные характеристики распределения.

Другой способ графического изображения — это гистограмма:

```
> hist(salary2, breaks=20, main="")
```

В нашем случае `hist()` по умолчанию разбивает переменную на 10 интервалов, но их количество можно указать вручную, как в предложенном примере. Численным аналогом гистограммы является функция`cut()`. При помощи этой функции можно выяснить, сколько данных какого типа у нас имеется:

```
> table(cut(salary2, 20))
```

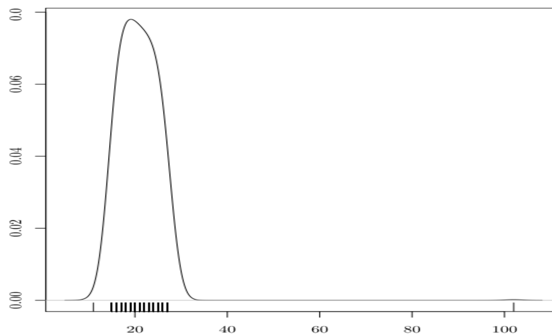
Есть еще две графические функции, «идеологически близкие» к гистограмме. Во-первых, это `stem()` – псевдографическая (текстовая) гистограмма:

```
> stem(salary, scale=2)
```

Это очень просто — значения данных изображаются не точками, а цифрами, соответствующими самим этим значениям.

Другая функция тоже близка к гистограмме, но требует гораздо более изощренных вычислений. Это график плотности распределения:

```
> plot(density(salary2, adjust=2), main="")
> rug(salary2)
```

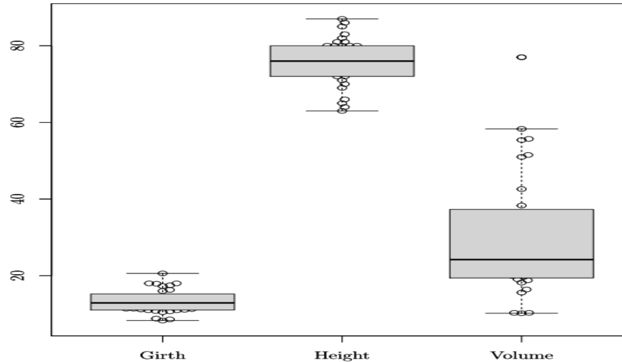



Графическая функция `rug()` используется,чтобы выделить места с наиболее высокой плотностью значений.

По сути, перед нами *сглаживание* гистограммы — попытка превратить ее в непрерывную гладкую функцию. Насколько гладкой она будет, зависит от параметра `adjust` (по умолчанию он равен единице). Результат сглаживания называют еще *графиком распределения*.

Кроме boxplot и различных графиков «семейства» гистограмм, в R много и других одномерных графиков. График «улей», например, отражает не только плотность распределения значений выборки, но и то, как расположены сами эти значения (точки). Для того чтобы его построить, потребуется загрузить (а возможно, еще и установить сначала) пакет beeswarm. После этого можно поглядеть на сам «улей»:

```
> library("beeswarm")
> beeswarm(trees)
> boxplot(trees, add=TRUE)
```




Здесь не просто построен график-улей, но еще и добавлен boxplot, чтобы стали видны квартили и медиана. Для этого нам понадобился аргумент add=TRUE.

Ну и, наконец, самая главная функция, summary():

```
> lapply(list(salary, salary2), summary)
event                mag              station         dist
Min. :   1.00     Min. :  5.000     117 : 5      Min. :      0.50
1st Qu.: 9.00     1st Qu.:5.300     1028 :       4 1st Qu.:11.32
Median :18.00     Median :6.100     113 : 4      Median :  23.40
Mean :  14.74     Mean :  6.084     112 : 3      Mean :     45.60
3rd Qu.:20.00     3rd Qu.:6.600     135 : 3      3rd Qu.:   47.55
Max. :  23.00     Max. :  7.700     (Other):147 Max. :    370.00
NA's : 16
```

Фактически она возвращает те же самые данные, что и fivenum() с добавлением среднего значения (Mean). Заметьте, кстати, что у обеих «зарплат» медианы одинаковы, тогда как средние существенно отличаются. Это еще один пример неустойчивости средних значений — ведь с добавлением случайно взятых «зарплат» вид распределения не должен был существенно поменяться.

Переменная station (номер станции наблюдений) — фактор, и к тому же с пропущенными данными, поэтому отображается иначе.

Перед тем как завершить рассказ об основных характеристиках выборки, надо упомянуть еще об одной характеристике разброса. Для сравнения изменчивости признаков (особенно таких, которые измерены в разных единицах измерения) часто применяют безразмерную величину — *коэффициент вариации*. Это просто отношение стандартного отклонения к среднему, взятое в процентах. Вот так можно сравнить коэффициент вариации для разных признаков деревьев (встроенные данные trees):

```
> 100*sapply(trees, sd)/colMeans(trees)
Girth        Height     Volume
```

```
23.686948  8.383964 54.482331
```

Здесь применена функция `sapply()` — вариант `lapply()` с упрощенным выводом, и `colMeans()`, которая вычисляет среднее для каждой колонки. Сразу отметим, что подобных функций вR несколько. Например, широко используются функции `colSums()` и `rowSums()`, которые выдают итоговые суммы соответственно по колонкам и по строкам (главная функция электронных таблиц!). Есть, разумеется, еще и `rowMeans()`.

### *Ошибочные данные*

Способность функции `summary()` указывать пропущенные данные, максимумы и минимумы служит очень хорошим подспорьем на самом раннем этапе анализа данных — проверке качества. Предположим, унас есть данные, набранные с ошибками:

```
> err<-read.table("errors.txt",h=TRUE,sep="\t",stringsAsFactors=TRUE)
> str(err)
'data.frame': 7 obs. of  3 variables:
$ AGE : Factor w/ 6 levels "12","22","23",..: 3 4 3 5 1 6 2
$ NAME  : Factor w/ 6 levels "","John","Kate",..: 2 3 1 4 5 6 2
$ HEIGHT: num  172 163 161 16.1 132 155 183
> summary(err)
  AGE   NAME      HEIGHT
  12:1       :1   Min.  : 16.1
  22:1  John :2   1st Qu.:143.5
  23:2  Kate :1   Median :161.0
  24:1  Lucy :1   Mean  :140.3
  56:1  Penny:1   3rd Qu.:167.5
  a :1  Sasha:1   Max.  :183.0
```

Кроме команды `summary()`, здесь использована также очень полезная команда `str()`. Как видно, переменная `AGE` (возраст) почему-то стала фактором, и `summary()` показывает, почему: в одну из ячеек закралась буква `a`.Кроме того, одно из имен пустое, скорее всего, потому, что в ячейку забыли поставить `NA`. Наконец, минимальный рост — 16.1 см! Такого не бывает обычно даже у новорожденных, так что можно с уверенностью утверждать, что наборщик просто случайно поставил точку.

### *Одномерные статистические тесты*

Закончив разбираться с описательными статистиками, перейдем к простейшим статистическим тестам. Начнем с так называемых «одномерных», которые позволяют проверять утверждения относительно того, как распределены исходные данные.

Предположим, мы знаем, что средняя зарплата в нашем первом примере — около 32 тыс. руб. Проверим теперь, насколько эта цифра достоверна:

```
> salary <- c(21, 19, 27, 11, 102, 25, 21)
> t.test(salary, mu=mean(salary))
  One Sample t-testdata:  salary
t = 0, df = 6, p-value = 1
alternative hypothesis: true mean is not equal to 32.28571
95 percent confidence interval:
      3.468127 61.103302
sample estimates:
mean of x
32.28571
```

Это вариант теста Стьюдента для одномерных данных. Статистические тесты (в том числе и этот) пытаются высчитать так называемую тестовую статистику, в данном случае

статистику Стьюдента (t- статистику). Затем на основании этой статистики рассчитывается «p-величина» (p-value), отражающая вероятность *ошибки первого рода*. А ошибкой первого рода (ее еще называют «ложной тревогой»), в свою очередь, называется ситуация, когда мы принимаем так называемую альтернативную гипотезу, в то время как *на самом деле* верна нулевая(гипотеза «по умолчанию»). Наконец, вычисленная p-величина используется для сравнения с заранее заданным порогом (уровнем) *значимости*. Если p-величина ниже порога, нулевая гипотеза отвергается, если выше — принимается.

В нашем случае нулевая гипотеза состоит в том, что истинное среднее (то есть среднее генеральной совокупности) равно вычисленному нами среднему (то есть 32.28571).

Перейдем к анализу вывода функции. Статистика Стьюдента при шести степенях свободы (df=6, поскольку у нас всего 7 значений) дает единичное p-значение, то есть 100%. Какой бы распространенный порог мы не приняли (0.1%, 1% или 5%), это значение все равно больше. Следовательно, мы принимаем нулевую гипотезу. Поскольку альтернативная гипотеза в нашем случае — это то, что «настоящее» среднее (среднее исходной выборки) не равно вычисленному среднему, то получается, что «на самом деле» эти цифры статистически не отличаются. Кроме всего этого, функция выдает еще и доверительный интервал (confidence interval), в котором, по ее «мнению», может находиться настоящее среднее. Здесь он очень широк – от 3.5 тысяч до 61 тысячи рублей.

Непараметрический (то есть не связанный предположениями о распределении) аналог этого теста тоже существует. Это так называемый ранговый тест Уилкоксона:

```
> new.1000 <- sample((median(salary) - IQR(salary)) :
+ (median(salary) + IQR(salary)), 1000, replace=TRUE)
> salary2 <- c(salary, new.1000)
> wilcox.test(salary2, mu=median(salary2), conf.int=TRUE)
  Wilcoxon signed rank test with continuity correction
data:  salary2
V = 221949, p-value = 0.8321
alternative hypothesis: true location is not equal to 21
95 percent confidence interval:
20.99999 21.00007
sample estimates:
(pseudo)median
21.00004
```

Эта функция и выводит практически то же самое, что и t.test() выше. Обратите, однако, внимание, что этот тест связан не со средним, а с медианой. Вычисляется (если задать conf.int=TRUE) и доверительный интервал. Здесь он значительно у́же, потому что медиана намного устойчивее среднего значения.

Понять, соответствует ли распределение данных нормальному (или хотя бы близко ли оно к нормальному), очень и очень важно. Например, все параметрические статистические методы основаны на предположении о том, что данные имеют нормальное распределение. Поэтому в R реализовано несколько разных техник, отвечающих на вопрос о нормальности данных. Во-первых, это статистические тесты. Самый простой из них – тест Шапиро-Уилкса:

```
> shapiro.test(salary2)
    Shapiro-Wilk normality test
data:  salary2
W = 0.74472, p-value < 2.2e-16
```

Здесь функция выводит гораздо меньше, чем в предыдущих случаях. Более того, даже встроенная справка не содержит объяснений того, какая здесь, например,

альтернативная гипотеза. Разумеется, можно обратиться к литературе, благо справка дает ссылки на публикации. А можно просто поставить эксперимент:

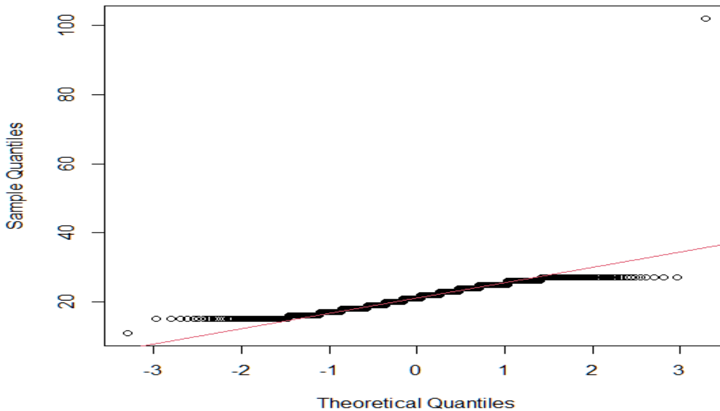
```
> set.seed(1638)
> shapiro.test(rnorm(100))
Shapiro-Wilk normality test
data:  rnorm(100)
W = 0.9934, p-value = 0.9094
```

rnorm() генерирует вектор случайных чисел распределенные по нормальному закону с нулевым средним зачением. Раз мы получили высокое p-значение, то это свидетельствует о том, что альтернативная гипотеза в данном случае: «распределение не соответствует нормальному». Функция set. seed(), регулирующая встроенный в R генератор случайных чисел, вызвана для того, чтобы результаты при вторичном воспроизведении были теми же.

Таким образом, распределение данных в salary2 отличается от нормального.

Другой популярный способ проверить, насколько распределение похоже на нормальное – графический:

```
> qqnorm(salary2, main="")
> qqline(salary2, col=2)
```




Для каждого элемента вычисляется, какое место он занимает в сортированных данных (так называемый «выборочный квантиль») и какоеместо он должен был бы занять, если распределение нормальное (теоретический квантиль). Прямая проводится через квартили. Если точки лежат на прямой, то распределение нормальное. В нашем случае многие точки лежат достаточно далеко от красной прямой, а значит, не похожи на распределенные нормально.

Для проверки нормальности можно использовать и более универсальный тест Колмогорова-Смирнова, который сравнивает любые два распределения, поэтому для сравнения с нормальным распределением ему надо прямо указать «pnorm», то есть так называемую накопленную функцию нормального распределения (она встроена в R):

```
> ks.test(salary2, "pnorm")
One-sample Kolmogorov-Smirnov test
data:  salary2
D = 1, p-value < 2.2e-16 alternative hypothesis: two-sided
```

Он выдает примерно то же, что и текст Шапиро-Уилкса.

***Как создавать свои функции***

Тест Шапиро-Уилкса всем хорош, но не векторизован, как и многие другие тесты в R, поэтому применить его сразу к нескольким колонкам таблицы данных не получится.

Сделано это нарочно, для того чтобы подчеркнуть нежелательность множественных парных сравнений. Но в нашем случае парных сравнений нет, а сэкономить время хочется. Можно, конечно, нажимая «стрелку вверх», аккуратно повторить тест для каждой колонки, но более правильный подход — создать пользовательскую функцию. Вот пример такой функции:

```
> normality <- function(data.f)
+ {
+     result <- data.frame(var=names(data.f), p.value=rep(0,
+     ncol(data.f)), normality=is.numeric(names(data.f)))
+     for (i in 1:ncol(data.f))
+     {
+           data.sh <- shapiro.test(data.f[, i])$p.value
+           result[i, 2] <- round(data.sh, 5)
+           result[i, 3] <- (data.sh > .05)
+     }
+     return(result)
+ }
```

Чтобы функция заработала, надо скопировать эти строчки в окно консоли или записать их в отдельный файл (желательно с расширением *.r), а потом загрузить командой `source()`. После этого ее можно вызвать:

```
> normality(trees)
var   p.value normality
  1 Girth 0.08893  TRUE
  2 Height 0.40342 TRUE
  3 Volume 0.00358 FALSE
```

Функция не только запускает тест Шапиро-Уилкса несколько раз, но еще и разборчиво оформляет результат выполнения. Разберем функцию чуть подробнее. В первой строчке указан ее аргумент — `data.f`. Дальше, в окружении фигурных скобок, находится тело функции. На третьей строчке формируется пустая таблица данных такой размерности, какая потребуется нам в конце. Дальше начинается цикл: для каждой колонки выполняется тест, а потом (это важно!) из теста извлекается p-значение. Процедура эта основана на знании структуры вывода теста — списка, где элемент `p-value` содержит p-значение. Все p-значения извлекаются, округляются, сравниваются с пороговым уровнем значимости (в данном случае 0.05) и записываются в таблицу. Затем таблица выдается «наружу». Предложенная функция совершенно не оптимизирована. Ее легко можно сделать чуть короче и к тому же несколько «смышленее», скажем так:

```
> normality2 <- function(data.f, p=.05)
+ {
+     nn <- ncol(data.f)
+     result <- data.frame(var=names(data.f), +
      p.value=numeric(nn),
+     normality=logical(nn))
+     for (i in nn)
+     {
+           data.sh <- shapiro.test(data.f[, i])$p.value
+           result[i,2:3]<-list(round(data.sh, 5),data.sh > p)
+     }
```

```
+     return(result)
+ }
> normality2(trees)
```

Результаты, разумеется, не отличаются. Зато видно, как можно добавить аргумент, причем сразу со значением по умолчанию. Теперь можно писать:

```
> normality2(trees, 0.1)
var   p.value normality
  1  Girth  0.08893 FALSE
  2  Height 0.40341 TRUE
  3  Volume 0.00358 FALSE
```

То есть если вместо 5% взять десятипроцентный порог, то уже и дляпервой колонки можно отвергнуть нормальное распределение.

Уже не раз говорилось, что циклов в R следует избегать. Можно ли сделать это в нашем случае? Оказывается, да:

```
> lapply(trees, shapiro.test)
$Girth
Shapiro-Wilk normality testdata:  X[[1L]]
W = 0.9412, p-value = 0.08893
$Height
...
```

Как видите, все еще проще! Если мы хотим улучшить зрительный эффект, можно сделать так:

```
> lapply(trees, function(.x) ifelse(shapiro.test(.x)$p.value >
+ .05, "NORMAL", "NOT NORMAL"))
$Girth
[1] "NORMAL"
$Height
[1] "NORMAL"
$Volume
[1] "NOT NORMAL"
```

Здесь применена так называемая *анонимная функция*, функция без названия, обычно употребляемая в качестве последнего аргумента команд типа `apply()`. Кроме того, используется логическая конструкция `ifelse()`.

И наконец, на этой основе можно сделать третью пользовательскую функцию:

```
> normality3 <- function(df, p=.05)
+ {
+    lapply(df,function(.x)ifelse(shapiro.test(.x)$p.value>p,
+      "NORMAL", "NOT NORMAL"))
+ }
> normality3(list(salary, salary2))
> normality3(log(trees+1))
```

Примеры тоже интересны. Во-первых, нашу третью функцию можно применять не только к таблицам данных, но и к «настоящим» спискам, с неравной длиной элементов. Во-вторых, простейшее логарифмическое преобразование сразу же изменило «нормальность» колонок.

### *Всегда ли точны проценты*

Полезной характеристикой при исследовании данных является пропорция (доля). В статистике под пропорцией понимают отношение объектов с исследуемой особенностью к общему числу наблюдений. Поскольку доля — это часть целого, то отношение части к целому находится в пределах от 0 до 1. Для удобства восприятия доли ее умножают на 100% и получают процент — число в пределах от 0% до 100%.

Рассмотрим проблему, которая часто встречается при проведении статистических исследований. Как узнать, отличается ли вычисленный нами процент от «истинного» процента, то есть доли интересующих нас объектов в генеральной совокупности?

Вот пример. В больнице есть группа из 476 пациентов, среди которых 356 курят. Мы знаем, что в среднем по больнице доля курящих составляет 0.7 (70%). А вот в нашей группе она чуть побольше — примерно 75%. Для того чтобы проверить гипотезу о том, что доля курящих в рассматриваемой группе пациентов отличается от средней доли по больнице, мы можем задействовать так называемый биномиальный тест:

```
> binom.test(x=356, n=476, p=0.7, alternative="two.sided")
Exact binomial test
data:  356 and 476
number of successes = 356, number of trials = 476,p-value =
0.02429
alternative hypothesis: true probability of success is notequal to
0.7
95 percent confidence interval:
0.7063733 0.7863138
sample estimates:
probability of success
0.7478992
```

Поскольку p-значение значительно меньше 0.05 и альтернативная гипотеза состоит в том, что доля курильщиков (она здесь довольно издевательски называется «probability of success», «вероятность успеха») не равна 0.7, то мы можем отвергнуть нулевую гипотезу и принять альтернативную, то есть решить, что наши 74% отличаются от средних по больнице 70% не случайно. В качестве опции мы использовали `alternative="two.sided"`, но можно было поступить иначе — проверить гипотезу о том, что доля курящих в рассматриваемой группе пациентов *превышает* среднюю долю курящих по больнице. Тогда альтернативную гипотезу надо было бы записать как `alt="greater"`.

Кроме биномиального, мы можем применить здесь и так называемый *тест пропорций*. Надо заметить, что он применяется шире, потому что более универсален:

```
> prop.test(x=356, n=476, p=0.7, alternative="two.sided")
1-sample proportions test with continuity correction
data:  356 out of 476, null probability 0.7
X-squared = 4.9749, df = 1, p-value = 0.02572
alternative hypothesis: true p is not equal to 0.7
95 percent confidence interval:
0.7059174 0.7858054
sample estimates:
p
0.7478992
```

Как видим, результат практически такой же.

Тест пропорций можно проводить и с двумя выборками, для этого используется все та же функция `prop.test()` (для двухвыборочного теста пропорций), а также `mcnemar.test()` (для теста Мак-Немара, который проводится тогда, когда выборки связаны

друг с другом).

```
> power.prop.test(p1=0.48, p2=0.52, power=0.8)
     Two-sample comparison of proportions power calculation
              n = 2451.596
             p1 = 0.48
             p2 = 0.52
      sig.level = 0.05
          power = 0.8
    alternative = two.sided
NOTE: n is number in *each* group
```

Получается, что опросить надо было примерно 5 тысяч человек!

Мы использовали здесь так называемый power тест, который часто применяется для прогнозирования эксперимента. Мы задали значение `power = 0.8`, потому что именно такое значение является общепринятым порогом значимости (оно соответствует `p-value < 0.1`).

# 7. Анализ связей: двумерные данные

Разберемся, как работать с двумя выборками. Если унас два набора цифр, то первое, что приходит в голову — сравнить их. Для этого нам потребуются статистические тесты.

### *Что такое статистический тест*

Настала пора подробнее познакомиться с ядром статистики — тестами и гипотезами. В предыдущей главе мы уже коротко объясняли,как строятся статистические гипотезы, но если анализ одной выборки можно сделать, не вдаваясь подробно в их смысл, то в анализе связей без понимания этих принципов не обойтись.

### *Статистические гипотезы*

Мы знаем, что статистическая выборка должна быть репрезентативной (то есть адекватно характеризовать генеральную совокупность, или, как ее еще называют, популяцию). Хотя мы и обеспечиваем репрезентативность выборки соблюдением двух основных принципов ее создания (рандомизации и повторности), неопределенность все же остается. Кроме того, если мы принимаем вероятностную точку зрения на происхождение данных (которые получены путем случайного выбора), то все дальнейшие суждения, основанные на этих данных, будут иметь вероятностный характер. Таким образом, мы никогда не сможем на основании нашей выборки со стопроцентной уверенностью судить о свойствах генеральной совокупности! Мы можем лишь *выдвигать гипотезы* и *вычислять их вероятность*.

Великие философы науки (например, Карл Поппер) постулировали,что мы ничего не можем доказать, мы можем лишь что-нибудь опровергнуть. Действительно, если мы соберем 1000 фактов, подтверждающих какую-нибудь теорию, это не будет значить, что мы ее доказали.

Поэтому в любом статистическом тесте выдвигаются две противоположные гипотезы. Одна — это то, что мы *хотим* доказать (но не можем!) — называется *альтернативной гипотезой* (ее обозначают $H_1$). Другая — противоречащая альтернативной — называется *нулевой гипотезой* (обозначается $H_0$). Нулевая гипотеза всегда является предположением об *отсутствии* чего-либо (например, зависимости одной переменной от другой или различия между двумя выборками). Мы не можем *доказать* альтернативную гипотезу, а можем лишь *опровергнуть* нулевую гипотезу и *принять* альтернативную. Если же мы не можем опровергнуть нулевую гипотезу, то мы вынуждены принять ее.

### *Статистические ошибки*

Естественно, что когда мы делаем любые предположения (в нашем случае — выдвигаем статистические гипотезы), мы можем ошибаться(в нашем случае — делать статистические ошибки). Всего возможночетыре исхода:

| Популяция / Выборка | Верна $H_0$ | Верна $H_1$ |
|---|---|---|
| Принимаем $H_0$ | Правильно! | Статистическая ошибка *второго рода* |
| Принимаем $H_1$ | Статистическая ошибка *первого рода* | Правильно! |

Если мы приняли для выборки $H_0$ (нулевую гипотезу) и она верна для генеральной совокупности (популяции), то мы правы, и все в порядке. Аналогично и для $H_1$ (альтернативной гипотезы). Ясно, что мы не можем знать, что в действительности верно для генеральной совокупности, и сейчас просто рассматриваем все логически возможные варианты.

А вот если мы приняли для выборки альтернативную гипотезу, а она оказалась неверна для генеральной совокупности, то мы совершили так называемую статистическую ошибку первого рода (нашли несуществующую закономерность). Вероятность того, что мы совершили эту ошибку (это и есть p-значение, «p-value»), всегда отображается при проведении любых статистических тестов. Очевидно, что если вероятность этой ошибки достаточно высока, то мы должны отвергнуть альтернативную гипотезу. Возникает естественный вопрос: какую вероятность считать достаточно высокой? Так же, как и с размером выборки, однозначного ответа на этот вопрос нет. Общепринято, что пороговым значением надо считать 0.05 (то есть альтернативная гипотеза отвергается, если вероятность ошибки при ее принятии больше или равна 5%).

Таким образом, решение о результатах статистических тестов принимается главным образом на основании вероятности статистической ошибки первого рода. Степень уверенности исследователя в том, что заключение, сделанное на основании статистической выборки, будет справедливо и для генеральной совокупности, отражает статистическая достоверность. Допустим, если вероятность статистической ошибки первого рода равна 3%, то говорят, что найденная закономерность достоверна с вероятностью 97%. А если вероятность статистической ошибки первого рода равна, например, 23%, то говорят, что достоверной закономерности не найдено.

В случае, если мы принимаем нулевую гипотезу для выборки, в то время как для генеральной совокупности справедлива альтернативная гипотеза, то мы совершаем статистическую ошибку второго рода (не замечаем существующей закономерности). Этот параметр характеризует так называемую *мощность* (power) статистического теста. Чем меньше вероятность статистической ошибки второго рода (то есть чем меньше вероятность не заметить несуществующую закономерность), тем более мощным является тест.

### *Тестирование двух выборок*

При ответе на этот вопрос нужно всегда помнить, что упомянутые ниже тесты проверяют различия только по центральным значениям (например, средним) и подразумевают, что разброс данных в выборках примерно одинаков. Например, выборки с одинаковыми параметрами средней тенденции и разными показателями разброса данных, `c(1, 2,3, 4, 5, 6, 7, 8, 9)` и `c(5, 5, 5, 5, 5, 5, 5, 5, 5)` различаться не будут.

Для проведения статистического теста нужно выдвинуть две статистические гипотезы. Нулевая гипотеза здесь: «различий между (двумя) выборками нет» (то есть обе выборки взяты из одной генеральной совокупности). Альтернативная гипотеза: «различия между (двумя) выборками есть».

Напоминаем, что ваши данные должны быть организованы в виде двух векторов — отдельных или объединенных в лист или таблицу данных. Например, если нужно узнать, различается ли достоверно ростмужчин и женщин, то в одном векторе должен быть указан

рост мужчин, а в другом — рост женщин (каждая строчка — это один обследованный человек).

Если данные параметрические, то нужно провести параметрический «t-тест» (или «тест Стьюдента»). Если при этом переменные, которые мы хотим сравнить, были получены *на разных объектах*, мы будем использовать двухвыборочный t-тест для независимых переменных (twosample t-test), который запускается при помощи команды `t.test()`. Например, если две сравниваемые выборки записаны в первом и втором столбцах таблицы данных `data`, то команда

```
> t.test(data[,1], data[,2])
```

по умолчанию выдаст нам результаты двухвыборочного t-теста для независимых переменных.

Если же пары сравниваемых характеристик были получены *на одном объекте*, то есть, переменные *зависимы* (например, частота пульса до и после физической нагрузки измерялась у одного и того же человека), надо использовать парный t-тест (paired t-test). Для этого в команде `t.test()` надо указать параметр `paired=TRUE`. В нашем примере, если данные зависимы, надо использовать команду вида

```
> t.test(data[,1],data[,2], paired=TRUE).
```

Второй тест, тест для зависимых переменных, более мощный. Представьте себе, что мы измеряли пульс до нагрузки у одного человека,а после нагрузки — у другого. Тогда было бы не ясно, как объяснить полученную разницу: может быть, частота пульса увеличилась после нагрузки, а может быть, этим двум людям вообще свойственна разная частота пульса. В случае же «двойного» измерения пульса каждый человек как бы является своим собственным контролем, и разница междусравниваемыми переменными (до и после нагрузки) обусловливается только тем фактором, на основе которого они выделены (наличием нагрузки).

Если мы имеем дело с *непараметрическими* данными, то нужно провести непараметрический *двухвыборочный тест Вилкоксона*, «Wilcoxon test» (он известен еще и как как *тест Манна-Уитни*, «Mann- Whitney test»). Для этого надо использовать команду `wilcox.test()`. В случае с зависимыми выборками, аналогично t-тесту, надо использовать параметр `paired=TRUE`.

Приведем несколько примеров. Для начала воспользуемся классическим набором данных, который использовался в оригинальной работе Стьюдента (псевдоним ирландского математика Уильяма Сили Госсета). В упомянутой работе производилось сравнение влияния двух различных снотворных на увеличение продолжительности сна. В **R** эти данные доступны под названием `sleep`. В столбце `extra` содержится среднее приращение продолжительности сна после начала приема лекарства (по отношению к контрольной группе), а в столбце `group` — код лекарства (первое или второе).

```
> plot(extra ~ group, data = sleep)
```

Здесь использована так называемая «формула модели»: в рассматриваемом случае `extra ~ group` означает, что `group` используется для разбивки `extra`.

Влияние лекарства на каждого человека индивидуально, но среднее увеличение продолжительности сна можно считать вполне логичным показателем «силы» лекарства. Основываясь на этом предположении, попробуем проверить при помощи t-теста, значимо ли различие в средних для этих двух выборок (соответствующих двум разным лекарствам):

```
> with(sleep, t.test(extra[group == 1], extra[group == 2],
+ var.equal = FALSE))
Welch Two Sample t-test

data:  extra[group == 1] and extra[group == 2]
```

```
t = -1.8608, df = 17.776, p-value = 0.0794
alternative hypothesis:true difference in means is not equal to 0
95 percent confidence interval:
-3.3654832  0.2054832
sample estimates:
mean of x mean of y
0.75        2.33
```

Параметр `var.equal` позволяет выбрать желаемый вариант критерия: оригинальный t-критерий Стьюдента в предположении, что разбросы данных одинаковы (`var.equal = TRUE`), или же t-тест в модификации Уэлча (Welch), свободный от этого предположения (`var.equal = FALSE`).

Хотя формально мы не можем отвергнуть нулевую гипотезу (о равенстве средних), p-значение (0.0794) все же достаточно маленькое, чтобы попробовать другие методы для проверки гипотезы, увеличить количество наблюдений, еще раз убедиться в нормальности распределений и т. д. Может прийти в голову провести односторонний тест — ведь он обычно чувствительнее. Так вот, этого делать нельзя! Нельзя потому, что большинство статистических тестов рассчитаны на то, что они будут проводиться без знания какой-либо дополнительной информации.

Для сравнения двух выборок существуют, разумеется, и непараметрические тесты. Один из них, *тест знаков*, настолько прост, что его нетв R. Можно, однако, легко сделать его самому. Тест знаков вычисляет разницу между всеми парами элементов двух одинаковых по размеру выборок (то есть это парный тест). Затем можно отбросить все отрицательные значения и оставить только положительные. Если выборки взяты из одной генеральной совокупности, то положительных разниц будет примерно половина, и тогда уже знакомый нам биномиальный тест не найдет отличий между 50% и пропорцией положительных разниц. Если выборки разные, то пропорция положительных разниц будет значимо больше или меньше половины.

Перейдем тем временем к более сложным непараметрическим тестам. Рассмотрим пример. Стандартный набор данных `airquality` содержит информацию о величине озона в воздухе города Нью-Йорка с мая по сентябрь 1973 года. Концентрация озона представлена округленными средними значениями за день, поэтому анализировать ее «от греха подальше» лучше непараметрическими методами.

Проверим, например, гипотезу о том, что распределение уровня озона в мае и в августе было одинаковым:

```
> wilcox.test(Ozone ~ Month, data = airquality,
+ subset = Month %in% c(5, 8))
Wilcoxon rank sum test with continuity correctiondata:  Ozone by
Month
W = 127.5, p-value = 0.0001208
alternative hypothesis: true location shift is not equal to 0
```

Поскольку `Month` дискретен (это просто номер месяца), то значения `Ozone` будут сгруппированы помесячно. Кроме того, мы использовали параметр `subset` вместе с оператором `%in%`, который выбирает изо всех месяцев май и август (пятый и восьмой месяцы).

Критерий отвергает гипотезу о согласии распределений уровня озона в мае и в августе с достаточно большой надежностью. В это достаточно легко поверить, потому что уровень озона в воздухе сильно зависит от солнечной активности, температуры и ветра.

Можно использовать t-тест и тест Вилкоксона и для одной выборки, если стоит задача сравнить ее с неким «эталоном». Поскольку выборка одна, то и соответствующие тесты называют одновыборочными. Нулевую гипотезу в этом случае надо формулировать как

равенство выборочной средней или медианы (для t-теста и теста Вилкоксона, соответственно) заданному числу $\mu$ .

**Задача**. В файле данных otsenki.txt записаны оценки одних и тех же учеников некоторого школьного класса за первую четверть (значение A1 во второй колонке) и за вторую четверть (A2), а также оценки учеников другого класса за первую четверть (B1). Отличаются ли результаты класса А за первую и вторую четверть? Какой класс учился в первой четверти лучше — А или В?

Попробуем проанализировать данные при помощи статистических тестов:

```
> otsenki <- read.table("data/otsenki.txt")
> klassy <- split(otsenki$V1, otsenki$V2)
```

Можно было сделать и по-другому, но функция split() — самое быстрое решение для разбивки данных на три группы, и к тому же она выдает список, элементы которого можно будет проверить на нормальность при помощи созданной нами функции normality3() :

```
> normality3(klassy)
$A1
[1] "NOT NORMAL"
$A2
[1] "NOT NORMAL"
$B1
[1] "NOT NORMAL"
```

Увы, параметрические методы неприменимы. Надо работать непараметрическими методами:

```
> lapply(klassy, function(.x) median(.x, na.rm=TRUE))
$A1
[1] 4
$A2
[1] 4
$B1
[1] 5
```

Мы применили анонимную функцию для того чтобы избавиться от пропущенных значений (болевшие ученики?). Похоже, что в классе А результаты первой и второй четвертей похожи, а вот класс В учился в первой четверти лучше А. Проверим это:

```
> wilcox.test(klassy$A1, klassy$A2, paired=TRUE)
Wilcoxon signed rank test with continuity correction
data: klassy$A1 and klassy$A2
V = 15.5, p-value = 0.8605
alternative hypothesis: true location shift is not equal to 0
> wilcox.test(klassy$B1, klassy$A1, alt="greater")
Wilcoxon rank sum test with continuity correction
data: klassy$B1 and klassy$A1
W = 306, p-value = 0.02382
alternative hypothesis: true location shift is greater than 0
```

Для четвертей мы применили парный тест: ведь оценки получали одни и те же ученики. А для сравнения разных классов использован односторонний тест, поскольку такие тесты обычно чувствительнее двусторонних и поскольку здесь как раз и можем проверить, учится ли класс В лучше А, а не просто отличаются ли их результаты.

Итак, результаты класса А за первую и вторую четверти достоверно не отличаются, при этом класс В учился в первую четверть статистически значимо лучше класса А.

### *Есть ли соответствие, или Анализ таблиц*

А как сравнить между собой две выборки из номинальных (категориальных) данных? Для этого часто используют таблицы сопряженности (contigency tables). Построить таблицу сопряженности можно с помощью функции `table()`:

```
> with(airquality, table(cut(Temp, quantile(Temp)), Month))
Month
          5  6  7  8  9
(56,72] 24  3  0  1 10
(72,79]  5 15  2  9 10
(79,85]  1  7 19  7  5
(85,97]  0  5 10 14  5
```

Строки этой таблицы — четыре интервала температур (по Фаренгейту), а столбцы — месяц года. На пересечении каждой строки и столбца находится число, которое показывает, сколько значений в данном месяце попадает в данный интервал температур.

Если факторов больше двух, то R будет строить многомерную таблицу и выводить ее в виде серии двумерных таблиц, что не всегда удобно.

Можно, однако, построить «плоскую» таблицу сопряженности, когда все факторы, кроме одного, объединяются в один «многомерный» фактор, чьи градации используются при построении таблицы. Построить такую таблицу можно с помощью функции `ftable()`:

```
> ftable(Titanic, row.vars = 1:3)
                    Survived No  Yes
 Class Sex    Age
 1st   Male   Child          0   5
              Adult          118 57
       Female Child          0   1
              Adult          4   140
 2nd   Male   Child          0   11
              Adult          154 14
       Female Child          0   13
              Adult          13  80
 3rd   Male   Child          35  13
              Adult          387 75
       Female Child          17  14
              Adult          89  76
 Crew  Male   Child          0   0
              Adult          670 192
       Female Child          0   0
              Adult          3   20
```

Параметр `row.vars` позволяет указать номера переменных в наборе данных, которые следует объединить в один-единственный фактор, градации которого и будут индексировать строки таблицы сопряженности. Параметр `col.vars` проделывает то же самое, но для столбцов таблицы.

Функцию `table` можно использовать и для других целей. Самое простое — это подсчет частот. Например, можно считать пропуски:

```
> d<-factor(rep(c("A","B","C"),10),levels=c("A","B","C","D","E"))
> is.na(d) <- 3:4
> table(factor(d, exclude = NULL))
```

```
A   B   C  <NA>
9  10   9    2
```

При помощи функции chisq.test() можно статистически проверить гипотезу о независимости двух факторов. К данным будет применен тест хи-квадрат (Chi-squared test). Например, проверим гипотезу о независимости цвета глаз и волос (встроенные данные HairEyeColor):

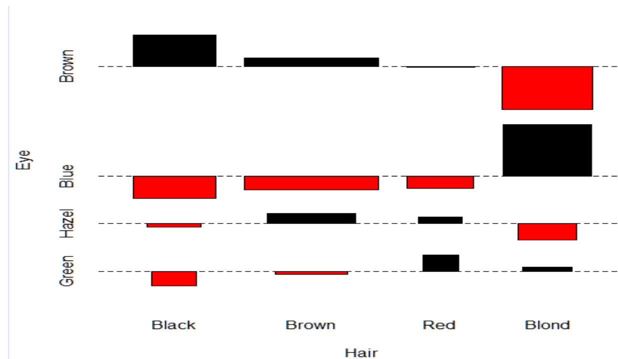
```
> x <- margin.table(HairEyeColor, c(1, 2))
> chisq.test(x)
Pearson's Chi-squared testdata: x
X-squared = 138.2898, df = 9, p-value < 2.2e-16
```

Такого же эффекта можно добиться, если функции summary() передать таблицу сопряженности в качестве аргумента.

Набор данных HairEyeColor — это многомерная таблица сопряженности. Для суммирования частот по всем «измерениям», кроме двух, использовалась функция margin.table. Таким образом, в результате была получена двумерная таблица сопряженности. Тест хи-квадрат в качестве нулевой гипотезы принимает независимость факторов, так что в нашем примере (поскольку мы отвергаем нулевую гипотезу) следует принять, что факторы *обнаруживают соответствие*.

Чтобы графически изобразить эти соответствия, можно воспользоваться функцией assocplot():

```
> x <- margin.table(HairEyeColor, c(1,2))
> assocplot(x)
```




На графике видны отклонения ожидаемых частот от наблюдаемых величин. Высота прямоугольника показывает абсолютную величину этого отклонения, а положение — знак отклонения. Отчетливо видно, что для людей со светлыми волосами характерен голубой цвет глаз и совсемне характерен карий цвет, а для обладателей черных волос ситуация обратная.

Чтобы закрепить изученное о таблицах сопряженности, рассмотрим еще один интересный пример. Однажды большая компания статистиков-эпидемиологов собралась на банкет. На следующее утро после банкета, в воскресенье, многие встали с симптомами пищевого отравления, а в понедельник в институте много сотрудников отсутствовало по болезни. Поскольку это были статистики, и не просто статистики, а эпидемиологи, то они решили вспомнить, кто что ел на банкете, и тем самым выяснить причину отравления. Получившиеся данные имеют следующий формат:

```
> tox <- read.table("otravlenie.txt", h=TRUE)
> head(tox)
  ILL CHEESE CRABDIP CRISPS BREAD CHICKEN RICE CAESAR TOMATO
1 1    1      1     1     2      1     1     1     1
```

```
2 2    1     1    1    2     1    2    2    2
3 1    2     2    1    2     1    2    1    2
4 1    1     2    1    1     1    2    1    2
5 1    1     1    1    2     1    1    1    1
6 1    1     1    1    1     1    2    1    1
  ICECREAM CAKE JUICE WINE COFFEE
1 1      1     1    1   1
2 1      1     1    1   2
3 1      1     2    1   2
4 1      1     2    1   2
5 2      1     1    1   1
6 2      1     1    2   2
```

Первая переменная (ILL) показывает, отравился участник банкета (1) или нет (2), остальные переменные соответствуют разным блюдам.Простой взгляд на данные ничего не даст, ведь на банкете было 45 человек и 13 блюд! Значит, надо попробовать статистические методы.

Так как данные номинальные, то можно попробовать проанализировать таблицы сопряженности:

```
> for (m in 2:ncol(tox))
+ {
+    tmp <- chisq.test(tox$ILL, tox[,m])
+    print(paste(names(tox)[m], tmp$p.value))
+ }
[1] "CHEESE 0.840899679390882"
[1] "CRABDIP 0.94931385140737"
[1] "CRISPS 0.869479670886473"
[1] "BREAD 0.349817724258644"
[1] "CHICKEN 0.311482217451896"
[1] "RICE 0.546434435905853"
[1] "CAESAR 0.000203410168460333"
[1] "TOMATO 0.00591250292451728"
[1] "ICECREAM 0.597712594782716"
[1] "CAKE 0.869479670886473"
[1] "JUICE 0.933074267280188"
[1] "WINE 0.765772843686273"
[1] "COFFEE 0.726555246056369"
```

Цикл for позволил нам не писать тест тринадцать раз подряд. Без функции print() цикл бы ничего не напечатал, а paste() помогла оформить результат. Ну а результат таков, что есть два значимых соответствия – с салатом (CAESAR) и с помидорами (TOMATO).

Виновник найден! Почти. Ведь вряд ли испорченными были сразу оба блюда. Можно теперь попробовать разобраться, что же было главной причиной. Для этого придется познакомимся с логистической регрессией.

Кроме методов работы с таблицами сопряженности (таких, например, как тест хи-квадрат), номинальные и шкальные данные можно обрабатывать различными, более специализированными методами. Например, для сравнения результатов экспертных оценок популярны так называемые тесты согласия (concordance). Среди этих тестов широко распространен тест Коэна (Cohen), который вычисляет так называемую *каппу Коэна* (Cohen’s kappa) — *меру согласия*, изменяющуюся от 0 до 1, и вдобавок вычисляет p-значение для нулевой гипотезы (о том, что каппа равна 0).

Приведем такой пример: в 2003 году две группы независимо (и примерно в одно и то же время года) обследовали один и тот же остров Белого моря. Целью было составить список всех видов растений, встреченных на острове. Таким образом, данные были бинарными (0 — вида на острове нет, 1 — вид на острове есть). Результаты обследований записаны в файл pokorm03. dat:

```
> pok <- read.table("pokorm_03.dat", h=TRUE, sep=";")
> library(psych)
> cohen.kappa(pok)
Cohen Kappa and Weighted Kappa correlation coefficientsand
confidence boundaries
lower estimate upperunweighted kappa  0.55 0.67  0.8
weighted kappa     0.55  0.67  0.8Number of subjects = 193
```

Каппа довольно высока, нижний предел 95% доверительного интервала выше 0.5. Это значит, что результаты исследования можно считать согласованными друг с другом.

**Задача**. В файле данных prorostki. txt находятся результаты эксперимента по проращиванию семян васильков, зараженных различными грибами (колонка CID, CID=0 — это контроль, то есть незараженные семена). Всего исследовали по 20 семян на каждый гриб, тестировали три гриба, итого с контролем — 80 семян. Отличается ли прорастание семян, зараженных грибами, от контроля?

Загрузим данные, посмотрим на их структуру и применим тест хи-квадрат:

```
> pr <- read.table("prorostki.txt", h=TRUE)
> head(pr)
CID GERM. 14
1    63      1
2    63      1
3    63      1
4    63      1
5    63      1
6    63      1
> chisq.test(table(pr[pr$CID %in% c(0,105),]))
Pearson’s Chi-squared test with Yates’ continuity correctiondata:
table(pr[pr$CID %in% c(0, 105), ])
X-squared = 8.0251, df = 1, p-value = 0.004613
> chisq.test(table(pr[pr$CID %in% c(0,80),]))
Pearson’s Chi-squared test with Yates’ continuity correctiondata:
table(pr[pr$CID %in% c(0, 80), ])
X-squared = 22.7273, df = 1, p-value = 1.867e-06
> chisq.test(table(pr[pr$CID %in% c(0,63),]))
Pearson’s Chi-squared test with Yates’ continuity correctiondata:
table(pr[pr$CID %in% c(0, 63), ])
X-squared = 0.2778, df = 1, p-value = 0.5982
Warning message:
In chisq.test(table(pr[pr$CID %in% c(0, 63), ])) :Chi-squared
approximation may be incorrect
```

Как видим, эффекты двух грибов (CID105 и CID80) отличаются отконтроля, а третьего (CID63) — нет.

***Анализ корреляций***

Мерой линейной взаимосвязи между переменными является коэффициент

корреляции Пирсона (обозначается латинской буквой *r*). Значения коэффициента корреляции могут изменяться по модулю от нуля до единицы. Нулевой коэффициент корреляции говорит о том, что значения одной переменной не связаны со значениями другой переменной, а коэффициент корреляции, равный единице (или минус единице), свидетельствует о четкой линейной связи между переменными. Положительный коэффициент корреляции говорит о положительной взаимосвязи (чем больше, тем больше), отрицательный — об отрицательной(чем больше, тем меньше).

*Степень* взаимосвязи между переменными отражает *коэффициентдетерминации*: это коэффициент корреляции, возведенный в квадрат. Эта величина показывает, какая доля изменений значений одной переменной сопряжена с изменением значений другой переменной. Например, если коэффициент корреляции увеличится вдвое (0.8), то степень взаимосвязи между переменными возрастет в четыре раза ($0.8^2 = 0.64$).

Повторим еще раз, что коэффициент корреляции характеризует меру линейной связи между переменными. Две переменные могут быть очень четко взаимосвязаны, но если эта связь не линейная, а, допустим, параболическая, то коэффициент корреляции будет близок к нулю. Поэтому, перед тем как оценить взаимосвязь, нужно посмотреть на ее графическое выражение. Лучше всего здесь использовать диаграмму рассеяния, или коррелограмму (scatterplot) — именно она вызывается в R командой `plot()` с двумя аргументами-векторами.

Очень важно также, что всюду в этом разделе речь идет о *наличии* и *силе* взаимосвязи между переменными, а не о *характере* этой взаимосвязи. Если мы нашли достоверную корреляцию между переменнымиА и Б, то это может значить, что:

```
А зависит от Б;
Б зависит от А;
А и Б зависят друг от друга;
А и Б зависят от какой-то третьей переменной В, а между собой не
имеют ничего общего.
```

Например, хорошо известно, что объем продаж мороженого и число пожаров четко коррелируют. Странно было бы предположить, что поедание мороженого располагает людей к небрежному обращению с огнем или что созерцание пожаров возбуждает тягу к мороженому. Все гораздо проще — оба этих параметра зависят от температуры воздуха!

Для вычисления коэффициента корреляции в R используется функция `cor()`:

```
> cor(5:15, 7:17)
[1] 1
> cor(5:15, c(7:16, 23))
[1] 0.9375093
```

В простейшем случае ей передаются два аргумента (векторы одинаковой длины). Кроме того, можно вызвать ее с одним аргументом, если это — матрица или таблица данных. В этом последнем случае функция `cor()` вычисляет так называемую *корреляционную матрицу*, составленную из коэффициентов корреляций между столбцами матрицы или набора данных, взятых попарно, например:

```
> cor(trees)
   Girth      Height    Volume
Girth   1.0000000 0.5192801   0.9671194
Height  0.5192801 1.0000000   0.5982497
Vol4ume 0.9671194 0.5982497   1.0000000
```

Если все данные присутствуют, то все просто, но что делать, когда есть пропущенные наблюдения? Для этого в команде `cor` есть параметр `use`. По умолчанию он равен `all.obs`,

что при наличии хотя бы одного пропущенного наблюдения приводит к сообщению об ошибке. Если use приравнять к значению complete.obs, то из данных автоматически удаляются все наблюдения с пропусками. Может оказаться так, что пропуски раскиданы по исходному набору данных достаточно хаотично и их много, так что после построчного удаления от матрицы фактически ничего не остается. В таком случае поможет попарное удаление пропусков, то есть удаляются строчки с пропусками не из всей матрицы сразу, а только лишь из двух столбцов непосредственно перед вычислением коэффициента корреляции. Для этого опцию use следует приравнять к значению pairwise.complete.obs (надо иметь в виду, что тогда коэффициенты корреляции вычисляются по *разному* количеству наблюдений и сравнивать их друг с другом может быть опасно).

Если данные непараметрические, нужно использовать ранговый коэффициент корреляции Спирмена (Spearman) $\rho$. Он менее подвержен влиянию случайных «выбросов» в данных, чем коэффициент Пирсона. Для подсчета $\rho$ достаточно приравнять параметр method к значению spearman:

```
> x < rexp(50)
> cor(x, log(x), method="spearman")
[1] 1
```

Если корреляционная матрица большая, то читать ее довольно трудно. Поэтому существует несколько способов визуального представления таких матриц. Можно, например, использовать функцию symnum(), которая выведет матрицу в текстовом виде, но с заменой чисел на буквы в зависимости от того, какому диапазону принадлежало значение:

```
> symnum(cor(longley))
GNP. GNP U A P Y E
GNP.deflator 1
GNP           B   1
Unemployed    ,   ,   1
Armed.Forces  .   .   1
Population    B   B   , . 1
Year          B   B   , . B 1
Employed      B   B   . . B B 1
attr(,"legend")
[1] 0 ‘ ’ 0.3 ‘.’ 0.6 ‘,’ 0.8 ‘+’ 0.9 ‘*’ 0.95 ‘B’ 1
```

Эта функция имеет большое количество разнообразных настроек,но по умолчанию они все выставлены в значения, оптимальные для отображения корреляционных матриц.

Второй способ — это графическое представление корреляционных коэффициентов. Идея проста: нужно разбить область от 1 до +1 на отдельные диапазоны, назначить каждому свой цвет, а затем все это отобразить. Для этого можно воспользоваться функциями image() и axis():

```
> cor.l < cor(longley)
> image(1:ncol(cor.l), 1:nrow(cor.l), cor.l,
+ col=heat.colors(22), axes=FALSE, xlab="", ylab="")
# Подписи к осям:
> axis(1, at=1:ncol(cor.l), labels=abbreviate(colnames(cor.l)))
> axis(2,at=1:nrow(cor.l),labels=abbreviate(rownames(cor.l)),las = 2)
```

(Мы сократили здесь длинные названия строк и столбцов при помощи команды abbreviate().)

Полученный график часто называют «heatmap» («карта темпера туры»).

Еще один интересный способ представления корреляционной матрицы

предоставляется пакетом ellipse. В этом случае значения коэффициентов корреляции рисуются в виде эллипсов. Чем ближе значениекоэффициента корреляции к +1 или 1 — тем более вытянутым становится эллипс. Наклон эллипса отражает знак. Для получения изображения необходимо вызвать функцию plotcorr (рис. 21):

```
> library(ellipse)
> cor.l < cor(longley)
> colnames(cor.l) < abbreviate(colnames(cor.l))
> rownames(cor.l) < abbreviate(rownames(cor.l))
> plotcorr(cor.l, type="lower", mar=c(0,0,0,0))
```

А как проверить статистическую значимость коэффициента корреляции? Это равносильно проверке статистической гипотезы о равенстве нулю коэффициента корреляции. Если гипотеза отвергается, то связь одного признака с другим считается *значимой*. Для проверки такой гипотезы используется функция cor. test():

```
> with(trees, cor.test(Girth, Height))
Pearson's product-moment correlationdata: Girth and Height
t = 3.2722, df = 29, p-value = 0.002758
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:0.2021327 0.7378538
sample estimates:
cor0.5192801
```

Логика рассуждений здесь абсолютно такая же, как и в рассмотренных выше тестах. В данном случае нам нужно принять альтернативную гипотезу о том, что корреляция действительно существует. Обратите внимание на доверительный интервал – тест показывает, что реальное значение корреляции может лежать в интервале 0.2–0.7.

***Какая связь, или Регрессионный анализ***

Корреляционный анализ позволяет определить, зависимы ли переменные, и вычислить силу этой зависимости. Чтобы определить тип зависимости и вычислить ее параметры, используется регрессионный анализ. Наиболее известна двумерная линейная регрессионная модель:

$m = b_0 + b_1 \times x$,

где $m$ — модельное (predicted) значение зависимой переменной $y$, $x$ – независимая переменная, а $b_0$ и $b_1$ – параметры модели (коэффициенты).

Такая модель позволяет оценить среднее значение переменной $y$ при известном значении $x$. Разность между истинным значением и модельным называется *ошибкой* («error») или *остатком* («residual»):

$E = y - m$

В идеальном варианте остатки имеют нормальное распределение с нулевым средним и неизвестным, но постоянным разбросом $\sigma^2$, которыйне зависит от значений $x$ и $y$. В этом случае говорят о гомогенности остатков. Если же разброс зависит еще от каких-либо параметров, то остатки считаются гетерогенными. Кроме того, если обнаружилось, что средние значения остатков зависят от $x$, то это значит, что между $y$ и$x$ имеется нелинейная зависимость.

Для того чтобы узнать значения параметров $b_0$ и $b_1$, применяют метод наименьших квадратов. Затем вычисляют среднеквадратичное отклонение $R^2$:

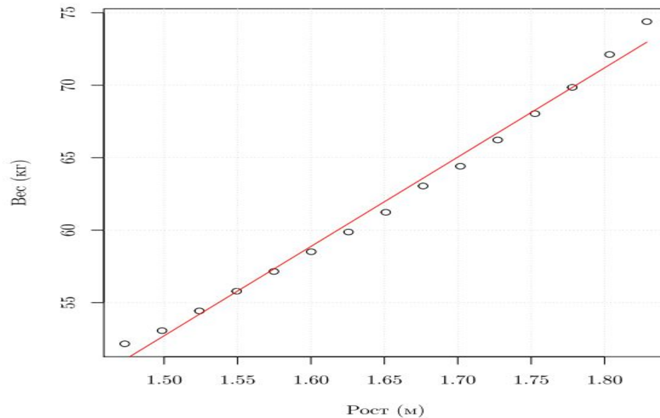
$$R^2 = 1 - \sigma^2_m / \sigma^2_y,$$

где $\sigma^2_y$ — разброс переменной y.

Для проверки гипотезы о том, что модель значимо отличается от нуля, используется так называемая F-статистика (статистика Фишера). Как обычно, если p-значение меньше уровня значимости (обычно 0.05),то модель считается значимой.

Вот пример. Во встроенной таблице данных women содержатся 15 наблюдений о росте (дюймы) и весе (фунты) женщин в возрасте от 30 до 39 лет. Попробуем узнать, как вес зависит от роста (рис. 22):

```
# Преобразование в метрическую систему:
> women.metr < women
> women.metr$height <- 0.0254*women.metr$height
> women.metr$weight <- 0.45359237*women.metr$weight
# Вычисление параметров уравнения регрессии:
> lm.women<-lm(formula = weight ~ height, data = women.metr)
> lm.women$coefficients
(Intercept) height
-39.69689   61.60999
# Вывод модельных значений:
> b0 < lm.women$coefficient[1]
> b1 < lm.women$coefficient[2]
> x1 < min(women.metr$height)
> x2 < max(women.metr$height)
> x < seq(from = x1, to = x2, length.out =100)
> y < b0 + b1*x
# Вывод графика зависимости:
> plot(women.metr$height, women.metr$weight, main="",
+ xlab="Рост (м)", ylab="Вес (кг)")
> grid()
> lines(x, y, col="red")
```




Для просмотра результатов линейной аппроксимации можно исполь зовать функцию

```
> summary(lm.women)
Call:
lm(formula = weight ~ height, data = women.metr)
Residuals:
Min        1Q     Median     3Q      Max
-0.7862 -0.5141 -0.1739  0.3364  1.4137
Coefficients:
                Estimate Std. Error t value Pr(>|t|)
(Intercept)  -39.697     2.693     -14.74 1.71e-09 ***
```

```
height         61.610    1.628      37.85 1.09e-14 ***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

Residual standard error: 0.6917 on 13 degrees of freedomMultiple R-
squared: 0.991,Adjusted R-squared: 0.9903
F-statistic:  1433 on 1 and 13 DF,  p-value: 1.091e-14
on 1 and 13 DF,  p-value: 1.091e-14
```

Отсюда следует, что:

- *Получена линейная регрессионная модель вида*
- *Вес(мод) = -39.697 + 61.61 * Рост*
- *Увеличение роста на 10 см соответствует увеличению весапримерно на 6 кг.*
- *Наибольшее положительное отклонение истинного значения от клика от модельного составляет 1.4137 кг, наибольшее отрицательное −0.7862 кг.*
- *Почти половина остатков находится в пределах от первой квартили (1Q= −0.5141 кг) до третьей (3Q= 0.3364 кг).*
- *Все коэффициенты значимы на уровне p-value < 0.001; это показывают *** в строке Coefficients) и сами значения p-value Pr(>|t|): 1.71e-09для $b_0$ («(Intercept)») и 1.09e-14 («height»)для $b_1$.*
- *Среднеквадратичное отклонение (Adjusted R-squared) для данной модели составляет $R^2$ = 0.9903. Его значение близко к 1, чтоговорит о высокой значимости.*
- *Значимость среднеквадратичного отклонения подтверждает и высокое значение F-статистики, равное 1433, и общий уровень значимости (определяемый по этой статистике): p-value: 1.091e-14, что много меньше 0.001.*
- *Если на основе этого анализа будет составлен отчет, то надо будет указать еще и значения степеней свободы df: 1 и 13 (по колонками по строкам данных соответственно).*

На самом деле это еще далеко не конец, а только начало долгого пути анализа и подгонки модели. В этом случае, например, параболическая модель (где используется не рост, а квадрат роста), будет еще лучше вписываться в данные (как выражаются, будет «лучше объяснять» вес). Можно попробовать и другие модели, ведь о линейных, обобщенных линейных и нелинейных моделях написаны горы книг! Но нам сейчас было важно сделать первый шаг.

Время от времени требуется не просто получить регрессию, а сравнить несколько регрессий, основанных на одних и тех же данных. Вот пример. Известно, что в методике анализа выборов большую роль играет соответствие между процентом избирателей, проголосовавших за данного кандидата, и процентом явки на данном избирательном участке. Эти соответствия для разных кандидатов могут выглядеть по-разному, что говорит о различии их электоратов. Данные из файла `vybory. txt`содержат результаты голосования за трех кандидатов по более чем сотне избирательных участков. Посмотрим, различается ли для разных кандидатов зависимость их результатов от явки избирателей. Сначала загрузим и проверим данные:

```
> vybory < read.table("vybory.txt", h=TRUE)
> str(vybory)
'data.frame': 153 obs. of  7 variables:
$ UCHASTOK: int  1 2 3 4 5 6 7 8 9 10 ...
```

```
$ IZBIR     : int  329786 144483 709903 696114 ...
$ NEDEJSTV  : int  2623 859 5656 4392 3837 4715 ...
$ DEJSTV: int  198354 97863 664195 619629 ...
$ KAND.1  : int  24565 7884 30491 54999 36880 ...
$ KAND.2  : int  11786 6364 11152 11519 10002 ...
$ KAND.3  : int  142627 68573 599105 525814 ...
> head(vybory)
UCHASTOK IZBIR NEDEJSTV DEJSTV KAND.1 KAND.2 KAND.3
1      1 329786    2623   198354  24565  11786 142627
2      2 144483    859      97863  7884   6364  68573
3      3 709903    5656   664195  30491  11152 599105
4      4 696114    4392   619629  54999  11519 525814
5      5 717095    3837   653133  36880  10002 559653
6      6 787593    4715   655486  72166  25204 485669
```

Теперь присоединим таблицу данных, для того чтобы с ее колонками можно было работать как с независимыми переменными:

```
> attach(vybory)
```

Вычислим доли проголосовавших за каждого кандидата и явку:

```
> DOLJA < cbind(KAND.1, KAND.2, KAND.3) / IZBIR
> JAVKA < (DEJSTV + NEDEJSTV) / IZBIR
```

Посмотрим, есть ли корреляция:

```
> cor(JAVKA, DOLJA)
KAND.1       KAND.2       KAND.3[1,] -0.3898262 -0.41416 0.9721124
```

Корреляция есть, хотя и невысокая для первых двух кандидатов. Похоже, что голосование по третьему кандидату серьезно отличалось. Проверим это:

```
> lm.1 < lm(KAND.1/IZBIR ~ JAVKA)
> lm.2 < lm(KAND.2/IZBIR ~ JAVKA)
> lm.3 < lm(KAND.3/IZBIR ~ JAVKA)
> lapply(list(lm.1, lm.2, lm.3), summary)
Call:
lm(formula = KAND.1/IZBIR ~ JAVKA)
Residuals:
   Min       1Q        Median          3Q      Max
-0.046133 -0.013211 -0.001493  0.012742  0.057943
Coefficients:
             Estimate  Std. Error t value Pr(>|t|)
(Intercept)  0.117115     0.008912  13.141  < 2e-16 ***
JAVKA       -0.070726    0.013597  -5.202 6.33e-07 ***
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
...
Call:
lm(formula = KAND.3/IZBIR ~ JAVKA)
Residuals:
  Min         1Q        Median    3Q       Max
-0.116483 -0.023570 0.000518 0.025714 0.102810
Coefficients:
            Estimate Std. Error t value Pr(>|t|)
```

```
(Intercept) -0.43006 0.01702    -25.27  <2e-16 ***
JAVKA        1.32261 0.02597     50.94  <2e-16 ***
---
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Коэффициенты регрессии у третьего кандидата сильно отличаются. Да и $R^2$ гораздо выше. Третий кандидат имеет электорат, отличающийся по своему поведению от электоратов двух первых кандидатов. Разумеется, хочется проверить это статистически, а не только визуально. Для того чтобы сравнивать регрессии, используется анализ ковариаций (ANCOVA). Покажем, как можно применить анализ ковариаций для данных о выборах:

```
> vybory2 < cbind(JAVKA, stack(data.frame(DOLJA)))
> names(vybory2) < c("javka","dolja","kand")
> str(vybory2)
'data.frame': 459 obs. of  3 variables:
$ javka: num  0.609 0.683 0.944 0.896 0.916 ...
$ dolja: num  0.0745 0.0546 0.043 0.079 0.0514 ...
$ kand : Factor w/ 3 levels "KAND.1","KAND.2",..: 1 1 1 ...
> head(vybory2, 3)

   javka       dolja      kand
1 0.6094164  0.07448770  KAND.1
2 0.6832776  0.05456697  KAND.1
3 0.9435810  0.04295094  KAND.1
```

Мы создали и проверили новую таблицу данных. В ней две интересующие нас переменные (доля проголосовавших за каждого кандидата и явка) расположены теперь в один столбец, а третий столбец – это фактор с именами кандидатов. Чтобы получить такую таблицу, мы использовали функцию `stack()`.

```
> ancova.v < lm(dolja ~ javka * kand, data=vybory2)
> summary(ancova.v)
Coefficients:
                  Estimate Std. Error t value Pr(>|t|)
(Intercept)       0.117115  0.011973   9.781   < 2e-16 ***
javka            -0.070726 0.018266  -3.872 0.000124 ***
kandKAND.2       -0.023627 0.016933  -1.395 0.163591
kandKAND.3       -0.547179 0.016933 -32.315 < 2e-16 ***
javka:kandKAND.2 0.004129  0.025832   0.160 0.873074
javka:kandKAND.3 1.393336  0.025832  53.938 < 2e-16 ***
```

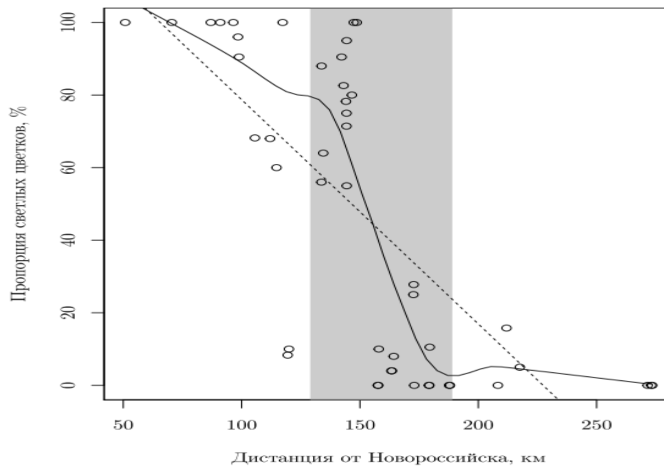
Анализ ковариаций использует формулу модели «*отклик ~ воздействие * фактор*», где звездочка (*) обозначает, что надо проверить одновременно отклик на воздействие, отклик на фактор и взаимодействие между откликом и фактором. (Напомним, что линейная регрессия использует более простую формулу, «*отклик ~ воздействие*»). В нашем случае откликом была доля проголосовавших, воздействием — явка, а фактором — имя кандидата. Больше всего нас интересовало, правда ли, что имеется сильное взаимодействие между явкой и третьим кандидатом. Полученные результаты (см. строку `javka:kandKAND.3`) не оставляют в этом сомнений, взаимодействие значимо.

Довольно сложно анализировать данные, если зависимость нелинейная. Здесь на помощь могут прийти методы визуализации. Более того, часто их бывает достаточно, чтобы сделать выводы, пусть и предварительные. Рассмотрим такой пример. По берегу Черного моря, от Новороссийска до Сочи, растут первоцветы, или примулы. Самая замечательная их черта — это то, что окраска цветков постепенно меняется при продвижении от

Новороссийска на юго-запад, вдоль берега: сначалабольшинство цветков белые или желтые, а ближе к Дагомысу появляется все больше и больше красных, малиновых, фиолетовых и розовых тонов. В файле данных primula.txt записаны результаты десятилетнего исследования береговых популяций первоцветов. Попробуем выяснить, *как именно* меняется окраска цветков по мере продвижения на юго-запад (рис. 24):

```
> prp.coast < read.table("data/primula.txt",
+ as.is=TRUE, h=TRUE)
> plot(yfrac ~ nwse, data=prp.coast, type="n",
+ xlab="Дистанция от Новороссийска, км",
+ ylab="Пропорция светлых цветков, %")
> rect(129, -10, 189, 110, col=gray(.8), border=NA); box()
> mtext("129", at=128, side=3, line=0, cex=.8)
> mtext("189", at=189, side=3, line=0, cex=.8)
> points(yfrac ~ nwse, data=prp.coast)
> abline(lm(yfrac ~ nwse, data=prp.coast), lty=2)
> lines(loess.smooth(prp.coast$nwse, prp.coast$yfrac), lty=1)
```

В этом небольшом анализе есть несколько интересных для нас моментов. Во-первых, окраска цветков закодирована пропорцией, для того, чтобы сделать ее интервальной. Во-вторых, линейная регрессия, очевидно, не отражает реальной ситуации. В-третьих, для изучения нелинейного взаимоотношения мы применили здесь так называемое сглаживание, «LOESS» (Locally weighted Scatterplot Smoothing), которое помогло увидеть нам общий вид возможной кривой.




### *Вероятность успеха, или Логистическаярегрессия*

Номинальные данные очень трудно обрабатывать статистически. Тест пропорций + хи-квадрат — практически все, что имеется в нашем арсенале для одно и двумерных номинальных данных. Однако часто встречаются задачи, в которых требуется выяснить не просто достоверность соответствия, а его характер, то есть требуется применить к номинальным данным нечто вроде регрессионного анализа. Вот, например, данные о тестировании программистов с разным опытом работы. Их просили за небольшое время (скажем, двадцать минут) написать несложную программу, не тестируя ее на компьютере. Потом написанные программы проверяли, и если программа работала, писали «S» («success», успех), а если не работала — писали «F» («failure», неудача):

```
> l < read.table("data/logit.txt", stringsAsFactors=TRUE)
> head(l)
```

```
  V1  V2
1 14  F
2 29  F
3  6  F
4 25  S
5 18  S
6  4  F
```

Как узнать, влияет ли стаж на успех? Таблицы сопряженности тутподходят мало, ведь стаж (V1) — интервальная переменная. Линейная регрессия не подходит, потому что для нее требуется не только интервальное воздействие (в нашем случае стаж), но еще и интервальный отклик. У нас же отклик — бинарный. Оказывается, существует выход. Для этого надо в качестве отклика исследовать не успех/неуспех, а *вероятность успеха*, которая, как всякая вероятность, непрерывно изменяется в интервале от 0 до 1. Мы не будем здесь вдаваться в мате матические подробности, а просто покажем, как это работает для данных о программистах:

```
> l.logit < glm(formula=V2 ~ V1, family=binomial, data=l)
> summary(l.logit)
Call:
glm(formula = V2 ~ V1, family = binomial, data = l)
Deviance Residuals:
  Min       1Q      Median      3Q         Max
-1.9987  -0.4584  -0.2245     0.4837      1.5005
Coefficients:
Estimate  Std. Error z value Pr(>|z|)
(Intercept)-4.9638      2.4597   -2.018   0.0436 *
V1          0.2350      0.1163    2.021   0.0432 *
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
(Dispersion parameter for binomial family taken to be 1)
Null deviance:     18.249 on 13 degrees of freedom
Residual deviance: 10.301 on 12 degrees of freedom
AIC: 14.301
Number of Fisher Scoring iterations: 5
```

Итак, оба параметра логистической регрессии значимы, и p-значения меньше пороговых. Этого достаточно, чтобы сказать: опыт программиста значимо влияет на успешное решение задачи.

### *Если выборок больше двух*

А что если теперь мы захотим узнать, есть ли различия между *тремя* выборками? Первое, что приходит в голову (предположим, что это параметрические данные), – это провести серию тестов Стьюдента: между первой и второй выборками, между первой и третьей и, наконец, между второй и третьей – всего три теста. К сожалению, число необходимых тестов Стьюдента будет расти чрезвычайно быстро с увеличением числа интересующих нас выборок. Например, для сравнения попарно шести выборок нам понадобится провести уже 15 тестов!А представляете, как обидно будет провести все эти 15 тестов толь ко для того, чтобы узнать, что все выборки не различаются между собой! Но главная проблема заключена не в сбережении труда исследователя (все-таки обычно нам нужно сравнить не больше 3–4 выборок). Дело в том, что при повторном проведении статистических тестов, основанных на вероятностных понятиях, на одной и той же выборке *вероятность обнаружить достоверную закономерность по ошибке возрастает*. Допустим, мы считаем различия достоверными при p-value< 0.05, при этом мы будем

ошибаться (находить различия там, где их нет) в 4 случаях из 100 (в 1 случае из 25). Понятно, что если мы проведем 25 статистических тестов на одной и той же выборке, то, скорее всего, однажды мы найдем различия просто по ошибке.

Поэтому для сравнения трех и более выборок используется специальный метод — однофакторный дисперсионный анализ (ANOVA, от английского «ANalysis Of VAriance»). Нулевая гипотеза здесь будет утверждать, что выборки не различаются между собой, а альтернативная гипотеза — что *хотя бы одна пара* выборок различается между собой. Обратите внимание на формулировку альтернативной гипотезы! Результаты этого теста будут одинаковыми и в случае, если различаетсятолько одна пара выборок, и в случае, если различаются все выборки.

Чтобы узнать, *какие именно* выборки отличаются, надо будет проводить специальные тесты.

Данные для ANOVA лучше организовать следующим образом: задать две переменные, и в одной из них указать все значения всех сравниваемых выборок (например, рост брюнетов, блондинов и шатенов), а во второй — коды выборок, к которым принадлежат значения первой переменной (например, будем ставить напротив значения роста брюнета «br», напротив роста блондина — «bl» и напротив роста шатена — «sh»). Тогда мы сможем задействовать уже знакомую нам логическую регрессию (при этом отклик будет количественный, а вот воздействие —качественное). Сам анализ проводится при помощи двойной функции anova(lm()):

```
> set.seed(1683)
> VES.BR < sample(70:90, 30, replace=TRUE)
> VES.BL < sample(69:79, 30, replace=TRUE)
> VES.SH < sample(70:80, 30, replace=TRUE)
> ROST.BR < sample(160:180, 30, replace=TRUE)
> ROST.BL < sample(155:160, 30, replace=TRUE)
> ROST.SH < sample(160:170, 30, replace=TRUE)
> data < data.frame(CVET=rep(c("br", "bl", "sh"), each=30),
+ VES=c(VES.BR, VES.BL, VES.SH),
+ ROST=c(ROST.BR, ROST.BL, ROST.SH))
> boxplot(data$ROST ~ data$CVET)
> anova(lm(data$ROST ~ data$CVET))
Analysis of Variance Table
Response: data$ROST
          Df Sum Sq Mean Sq F value   Pr(>F)
data$CVET  2 2185.2 1092.58  81.487  < 2.2e-16 ***
Residuals 87 1166.5   13.41
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
```

Надо теперь объяснить, что именно мы сделали. В R дисперсионный анализ проводится при помощи той же самой функции линейной модели lm(), что и регрессия, но формула модели здесь другая: *отклик~ фактор*, где фактором является индикатор группы (в нашем случае c("br", "bl", "sh")). Такой анализ сначала строит три модели. Каждую модель можно условно представить в виде вертикальной линии, по которой распределены точки; вертикальной потому, что все точки со ответствуют одному значению индикатора группы (скажем, bl). Потом функция anova() сравнивает эти модели и определяет, есть ли между ними какие-нибудь значимые отличия. Мы создали наши данные искусственно, как и данные о зарплате в главе про одномерные данные. Ну и, естественно, получили ровно

то, что заложили, – надо принять альтернативную гипотезу о том, что хотя бы одна выборка отличается от прочих. Можно предположить, что это — блондины. Проверим предположение при помощи множественного t-теста (pairwise t-test):

```
> pairwise.t.test(data$ROST, data$CVET)
Pairwise comparisons using t tests with pooled SDdata:  data$ROST
and data$CVET
      bl      br
br <2e-16   -
sh 1.1e-11  1.2e-05
P value adjustment method: holm
```

И вправду, из полученной таблички видно, что блондины значимо (оба p-значения много меньше 0.05) отличаются от остальных групп.

Важно помнить, что ANOVA — параметрический метод, то есть наши данные должны иметь нормальное (или близкое к нормальному) распределение. Кроме того, стандартные отклонения выборок должны быть, по крайней мере, похожи. Последнее условие можно обойти при помощи функции `oneway.test()`, которая не предполагает по умолчанию равенства разбросов. Но если есть сомнения в параметричности исходных данных, то лучше всего использовать непараметрический аналог ANOVA, тест Краскала-Уоллиса (Kruskal-Wallis test).

# 8. Анализ структуры: data mining

Фразу «data mining» можно все чаще увидеть в Интернете и на обложках книг по анализу данных. Говорят, даже, что эпоха статистики одной-двух переменных закончилась, и наступило новое время – время анализа больших и сверхбольших массивов данных.

На самом деле под методами «data mining» подразумеваются любые методы, как визуальные, так и аналитические, позволяющие «нащупать» структуру в данных, особенно в данных большого размера. Данные для такого анализа используются, как правило, многомерные, то есть такие, которые можно представить в виде таблицы из нескольких колонок-переменных. Поэтому более традиционное название для этих методов — «многомерный анализ», или «многомерная статистика».Но «data mining» звучит, конечно, серьезнее. Кроме многомерности и большого размера (сотни, а то и тысячи строк и столбцов), используемые данные отличаются еще и тем, что переменные в них могут быть совершенно разных типов (интервальные, шкальные, номинальные).

Грубо говоря, многомерные методы делятся на методы визуализации и методы классификации с обучением. В первом случае результат можно анализировать в основном зрительно, а во втором — возможна статистическая проверка результатов. Разумеется, граница между эти ми группами нерезкая, но для удобства мы станем рассматривать их именно в этом порядке.

### *Рисуем многомерные данные*

Самое простое, что можно сделать с многомерными данными, – это построить график. Разумеется, для того чтобы построить график нескольких переменных, надо свести все разнообразие к двум или, в крайнем случае, к трем измерениям. Это называют «сокращением размерности».

Но иногда можно обойтись и без сокращения размерности. Например, если переменных три.

### *Диаграммы рассеяния*

Если все три переменные — непрерывные, то поможет пакет `RGL`, который позволяет создавать настоящие трехмерные графики.

Для примера всюду, где возможно, мы будем использовать встроенные в R данные `iris`. Эти данные, заимствованные из работы знаменитого математика (и биолога) Р. Фишера, описывают разнообразие нескольких признаков трех видов ирисов.

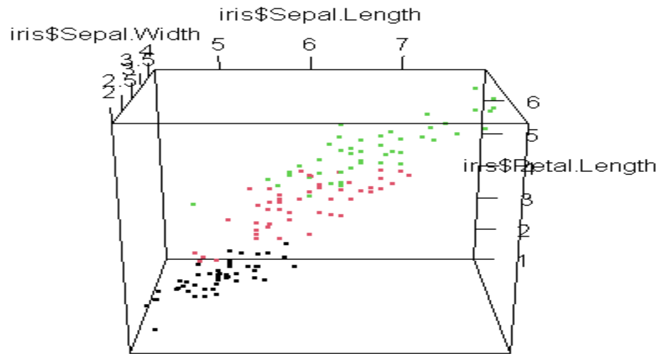
Соответственно, в них 5 переменных (колонок), причем последняя — это название вида.

Вот как можно изобразить 4 из 5 колонок при помощи RGL:

```
> library(rgl)
> plot3d(iris$Sepal.Length,iris$Sepal.Width,iris$Petal.Length,
+ col=as.numeric(iris$Species), size=3)
```

Размер появившегося окна и проекцию можно (и нужно) менять припомощи мышки.

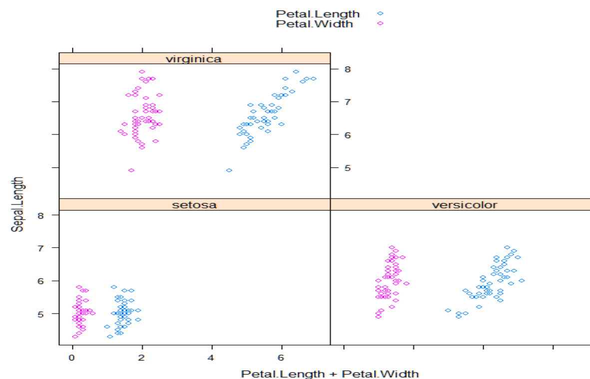
Сразу видно, что один из видов (*Iris setosa*) хорошо отличаетсяот двух других по признаку длины лепестков (Petal.Length). Кстати, в том, что мы изобразили именно 4 признака, не было оговорки, ведь вид ириса – тоже признак, закодированный в данном случае цве том. Можно обойтись и без RGL, тогда вам может потребоваться пакет scatterplot3d, содержащий одноименную функцию.




Но, вообще говоря, трехмерными графиками лучше не злоупотреблять –они, скорее, не раскрывают, а затемняют суть явления. Правда, в случае RGL это компенсируется возможностью свободно менять «точку обзора».

Другой способ визуализации многомерных данных — это построение составных графиков. Здесь у R колоссальные возможности, предоставляемые пакетом lattice, который предназначен для так называемой панельной(Trellis) графики. Вот как можно изобразить четыре признака ирисов (рис. 31):

```
> library(lattice)
> xyplot(Sepal.Length ~ Petal.Length + Petal.Width | Species,
+ data=iris, auto.key=TRUE)
```

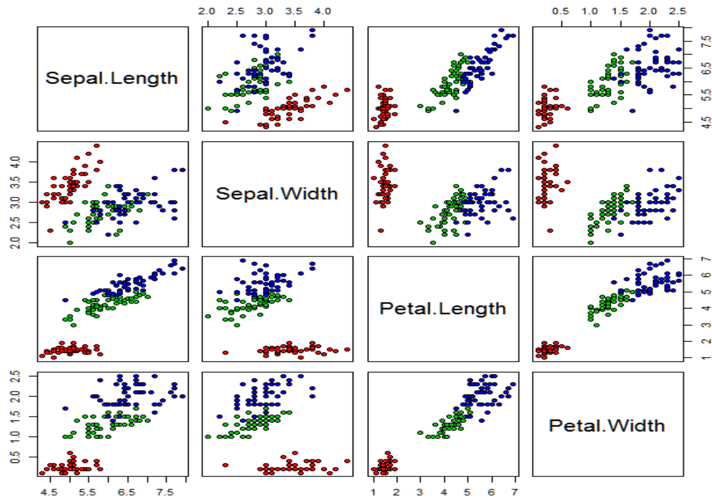



Получилась картинка зависимости длины чашелистиков как от длины, так и от

ширины лепестков для каждого из трех видов.

Матричный график рисуется при помощи функции pairs()

```
> pairs(iris[1:4], pch=21, bg=c("red", "green3", "blue")
+ [unclass(iris$Species)])
```




Мы получили зависимость значения каждого признака от каждого признака, причем заодно и «покрасили» точки в цвета видов (для этого пришлось «деклассировать» фактор iris$Species, превратив его в текстовый вектор). Таким образом, нам удалось отобразить сразу пять переменных.

***Анализ главных компонент***

Перейдем теперь к методам сокращения размерности. Самый распространенный из них – это анализ главных компонент. Суть его в том, что объекты можно представить, как точки в $n$-мерном пространстве, где $n$ – это число анализируемых признаков. Через полученное облако точек проводится прямая, так, чтобы учесть наибольшую долю изменчивости признаков, то есть, пронизывая это облако вдоль наиболее вытянутой его части (это можно представить себе как грушу, надетую на стальной стержень), так получается первая главная компонента. Затем через это облако проводится вторая, перпендикулярная первой прямая, так, чтобы учесть наибольшую оставшуюся долю изменчивости признаков, это будет вторая главная компонента. Эти две компоненты образуют плоскость, на которую проецируются все точки.

В результате все признаки-колонки преобразуются в компоненты, причем наибольшую информацию о разнообразии объектов несет первая компонента, вторая несет меньше информации, третья — еще меньше и т. д. Таким образом, хотя компонент получается столько же, сколь ко изначальных признаков, в первых двух-трех из них сосредоточена почти вся нужная нам информация. Поэтому их можно использовать для визуализации данных на плоскости, обычно первой и второй (реже первой и третьей) компоненты. Компоненты часто называют «фактора ми», и это порождает некоторую путаницу с похожим на анализ глав ных компонент факторным анализом, преследующим, однако, совсем другие цели (мы его рассматривать не будем).

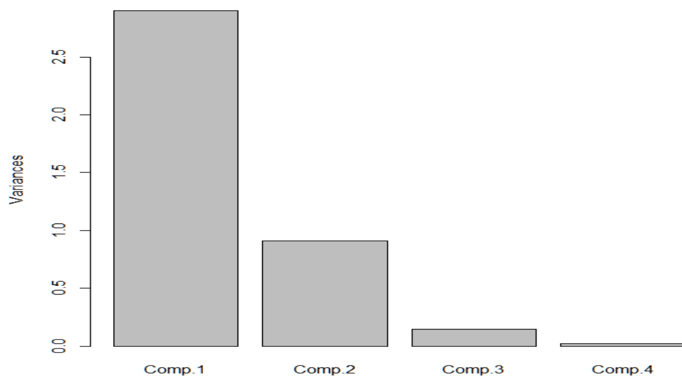
Вот как делается анализ главных компонент на наших данных про ирисы:

```
> iris.pca <- princomp(scale(iris[,1:4]))
```

Мы употребили функцию scale() для того, чтобы привести все четыре переменные к одному масштабу.

Теперь посмотрим на сами компоненты:

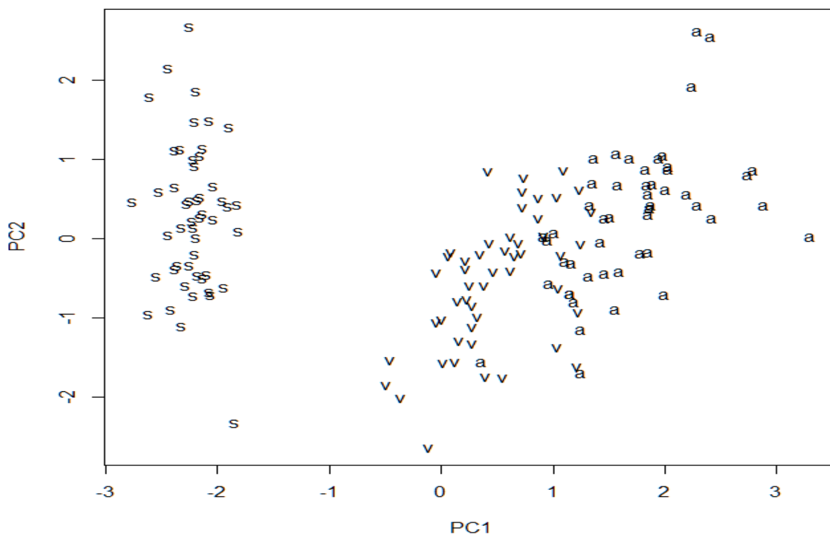
```
> plot(iris.pca, main="")
```




Это служебный график, так называемый «screeplot» (в буквальном переводе «график осыпи»), показывающий относительные вклады каждой компоненты в общий разброс данных. Хорошо видно, что компонент четыре, как и признаков, но, в отличие от первоначальных признаков, наибольший вклад вносят первые две компоненты.

Перейдем к собственно визуализации:

```
> iris.p <- predict(iris.pca)
> plot(iris.p[,1:2], type="n", xlab="PC1", ylab="PC2")
> text(iris.p[,1:2],labels=abbreviate(iris[,5],1,method="both.sides"))
```




Вот так мы визуализировали разнообразие ирисов. Получилось, что *Iris setosa* (обозначен буквой s) сильно отличается от двух остальных видов, *Iris versicolor* (v) и *Iris virginica* (a). Функция `predict()` позволяет расположить исходные случаи (строки) в пространстве вновь найденных компонент.

Иногда полезной бывает и функция `biplot()`:

```
> biplot(iris.pca)
```

Понять, насколько силен вклад каждого из четырех исходных признаков можно используя функцию `loadings()`:

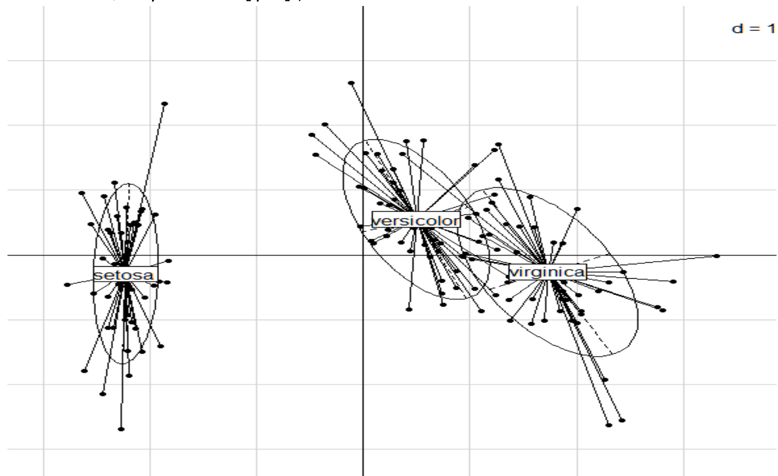
```
> loadings(iris.pca)
Loadings:
              Comp.1 Comp.2 Comp.3 Comp.4
```

```
Sepal.Length  0.521  0.377  0.720  0.261
Sepal.Width  -0.269  0.923 -0.244 -0.124
Petal.Length  0.580        -0.142 -0.801
Petal.Width   0.565        -0.634  0.524
               Comp.1 Comp.2 Comp.3 Comp.4
SS loadings      1.00   1.00   1.00   1.00
Proportion Var   0.25   0.25   0.25   0.25
Cumulative Var   0.25   0.50   0.75   1.00
```

В первой части вывода каждая ячейка таблицы соответствует вкладу признака в определенный компонент. Чем ближе это значение по модулю к единице, тем больше вклад.

Пакеты `ade4` и `vegan` реализуют множество вариаций анализа главных компонент, но самое главное — содержат гораздо больше возможностей для визуализации. Вот как можно проанализировать те же данные в пакете `ade4`:

```
> library(ade4)
> iris.dudi <- dudi.pca(iris[,1:4], scannf=FALSE)
> s.class(iris.dudi$li, iris[,5])
```




Не правда ли, на получившемся графике различия видны яснее? Кроме того, можно проверить качество разрешения между классами (в данном случае видами ирисов):

```
> iris.between < bca(iris.dudi, iris[,5], scannf=FALSE)
> randtest(iris.between)
Monte-Carlo test
Call: randtest.between(xtest = iris.between)
Observation: 0.7224358
Based on 999 replicates Simulated p-value: 0.001 Alternative
hypothesis: greater
```

Классы (виды ирисов) различаются хорошо. Об этом говорит основанное на 999 повторениях (со слегка различающимися параметрами) анализа значение `Observation`. Если бы это значение было меньше 0.5 (то есть 50%), то нам пришлось бы говорить о нечетких различиях. Как видно, использованный метод уже ближе к «настоящей статистике», нежели к типичной визуализации данных.

***Классификация без обучения, или кластерный анализ***

Другим способом снижения размерности является классификация без обучения (упорядочение, или ординация), проводимая на основаниизаранее вычисленных значений сходства между всеми парами объектов(строк). В результате этой процедуры получается

квадратная матрица расстояний, диагональ которой обычно составлена нулями (ведь расстояние между объектом и им же самим равно нулю). За десятилетия развития этой области статистики придуманы сотни коэффициентов сходства, из которых наиболее употребительными являются евклидово и квартальное (манхеттеновское), применимые в основном к непрерывным переменным. Коэффициент корреляции тоже может быть мерой сходства. Балльные и бинарные переменные в общем случае требуют других коэффициентов, но в пакете `cluster` реализована функция `daisy()`, способная распознавать тип переменной и применять соответствующие коэффициенты, а в пакете `vegan` реализовано множество дополнительных коэффициентов сходства.

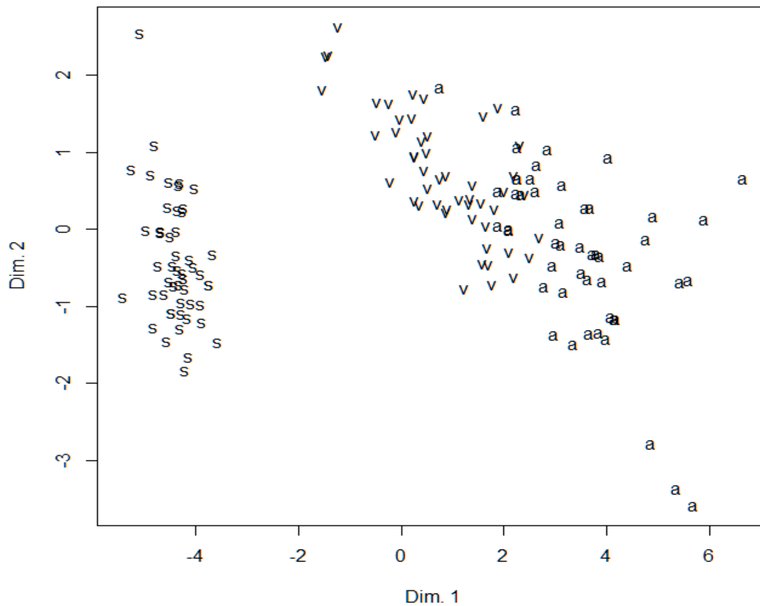
Вот как можно построить матрицу сходства (лучше ее все-таки называть матрицей различий, поскольку в ее ячейках стоят именно расстояния) для наших ирисов:

```
> library(cluster)
> iris.dist <- daisy(iris[,1:4], metric="manhattan")
```

(Указанное нами в качестве аргумента манхеттеновское расстояние будет вычислено только для интервальных данных, для данных других типов функция `daisy()` самостоятельно рассчитает более универсальное расстояние Говера, «Gower's distance»).

С полученной матрицей можно делать самые разные вещи. Одно из самых простых применений — сделать многомерное шкалирование (или, как его еще иногда называют, «анализ главных координат»). Сутьметода можно объяснить так. Допустим, мы измерили по карте расстояние между десятком городов, а карту потеряли. Задача — восстановить карту взаимного расположения городов, зная только расстояния между ними. Такую же задачу решает многомерное шкалирование. Причем этоне метафора – вы можете запустить `example(cmdscale)` и посмотреть(на примере 21 европейского города), как это происходит на самом деле.А вот для наших ирисов многомерное шкалирование можно применить так:

```
> iris.c <- cmdscale(iris.dist)
> plot(iris.c[,1:2], type="n", xlab="Dim. 1", ylab="Dim. 2")
> text(iris.c[,1:2], labels=abbreviate(iris[,5],1,method="both.sides"))
```

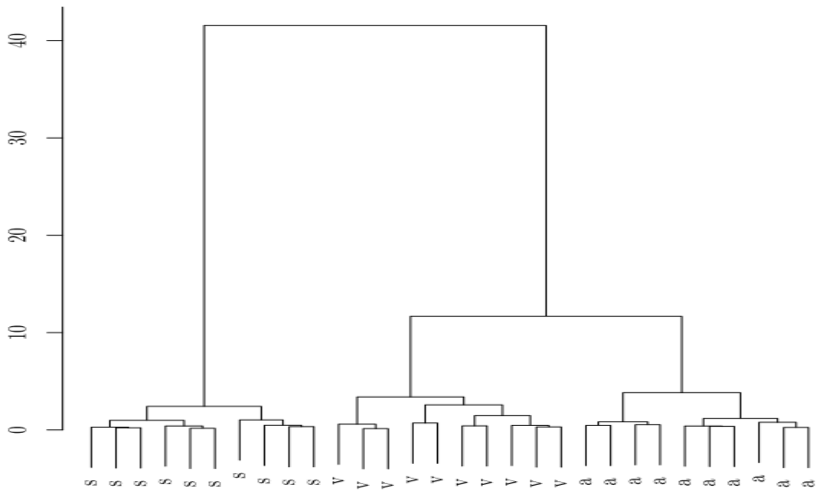



Как видно, результат очень похож на результат анализа главныхкомпонент, что не удивительно – ведь внутренняя структура данных(которую нам и надо найти в процессе «data mining») не изменилась. Кроме `cmdscale()`, советуем обратить внимание на

непараметрическийвариант этого метода, осуществляемый при помощи функции isoMDS().

Другой вариант работы с матрицей различий — это кластерный анализ. Существует множество его разновидностей. Наиболее употребительными являются иерархические методы, которые вместо уже привычных нам двумерных графиков производят «полуторамерные» деревья классификации, или дендрограммы. Вот как это делается (рис. 44):

```
> iriss < iris[seq(1,nrow(iris),5),]
> iriss.dist < daisy(iriss[,1:4])
> iriss.h < hclust(iriss.dist, method="ward.D")
> plot(iriss.h,labels=abbreviate(iriss[,5],1,
method="both.sides"), main="")
```




Поскольку на выходе получается «дерево», мы взяли для его по строения каждую пятую строку данных, иначе ветки сидели бы слишком плотно (это, кстати, недостаток иерархической кластеризации как метода визуализации данных). Метод Уорда (ward.D) дает очень хорошо очерченные кластеры (конечно, если их удается найти), и поэтому неудивительно, что в нашем случае все три вида разделились, причем отлично видно, что виды на «v» (*Iris versicilor* и *I. virginica*) разделяются на более низком уровне (Height 12), то есть сходство между ними сильнее, чем каждого из них с третьим видом.

Кластерный анализ этого типа весьма привлекателен тем, что дает готовую классификацию. Однако не сто́ит забывать, что это всего лишь визуализация, и кластерный анализ может «навязать» данным структуру, которой у них на самом деле нет. Насколько «хороши» получившиеся кластеры, проверить порой непросто, хотя и здесь создано множество методов. Один метод реализует пакет cluster — это так называемый «silhouette plot» (за примером можно обратиться к example(agnes)). Другой, очень «модный» метод, основанный на так называемой bootstrap-репликации, реализует пакет pvclust (рис. 45):

```
> library(pvclust)
> isst <- t(iriss[,1:4])
> colnames(irisst)<-paste(abbreviate(iriss[,5],3),colnames(irisst))
> iriss.pv <- pvclust(irisst, method.dist="manhattan",
+ method.hclust="ward.D", nboot=100)
Bootstrap (r = 0.5)... Done.
```

…

```
> plot(iriss.pv, col.pv=c(au=0, bp="darkgreen", edge=0), main="")
```

Получим аналогичное изображение. Но над каждым узлом печатаются p-значения (au), связанные с устойчивостью кластеров в процессе репликации исходных данных. Значения, близкие к 100, считаются «хорошими».

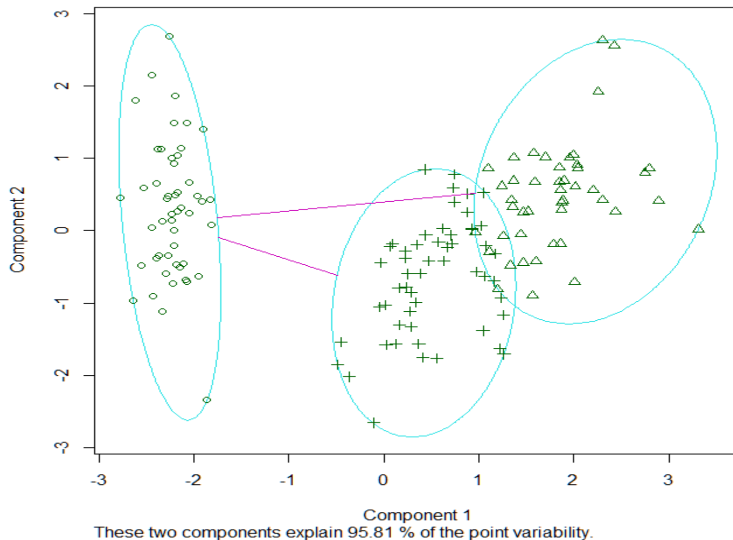
Кроме иерархических методов кластеризации, существуют и другие.Есть, например, «методы средних», которые стараются получить заранее заданное количество кластеров. Приведем пример. В файле eq. txt записаны данные измерений разных частей растений у двух сильно от личающихся видов хвощей. Попробуем проверить, сможет ли такой метод распознать эти виды, зная лишь признаки. Используем функцию kmeans():

```
> eq <- read.table("eq.txt", h=TRUE)
> eq.k <- kmeans(eq[,-1], centers=2)
> table(eq.k$cluster, eq$SPECIES)arvense fluviatile
  arvense fluviatile
1     37     5
2      1     41
```

Получилось! Ошибка классификации довольно низкая.

Очень интересны так называемые *нечеткие методы* кластеризации, основанные на идее того, что каждый объект может принадлежать к нескольким кластерам сразу – но с разной «силой». Вот как реализуется такой метод в пакете cluster:

```
> iris.f <- fanny(iris[,1:4], 3)
> plot(iris.f, which=1, main="")
```




```
> head(data.frame(sp=iris[,5], iris.f$membership))

  sp          X1         X2         X3
1 setosa 0.9142273 0.03603116 0.04974153
2 setosa 0.8594576 0.05854637 0.08199602
3 setosa 0.8700857 0.05463714 0.07527719
4 setosa 0.8426296 0.06555926 0.09181118
5 setosa 0.9044503 0.04025288 0.05529687
6 setosa 0.7680227 0.09717445 0.13480286
```

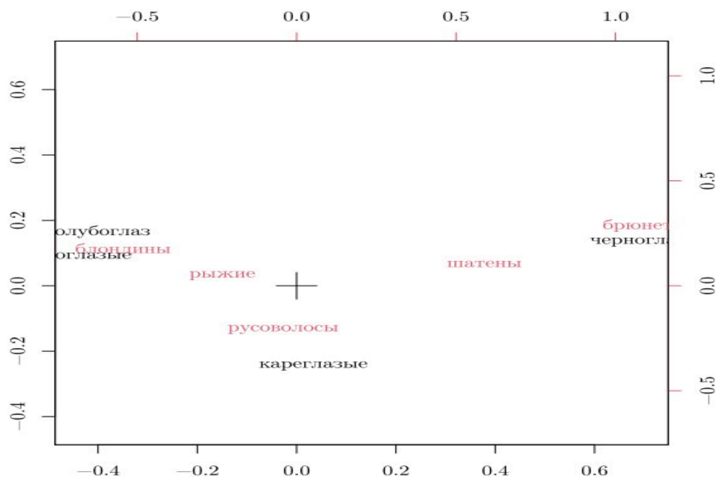
Подобный график мы уже неоднократно видели, здесь нет ничего принципиально

нового. А вот текстовый вывод интереснее. Для каждойстрочки указан «membership» — показатель «силы» связи, с которой данный элемент «притягивается» к каждому из трех кластеров. Как видно, шестая особь, несмотря на то, что почти наверняка принадлежитк первому кластеру, тяготеет и к третьему. Недостатком этого методаявляется необходимость заранее указывать количество получающихся кластеров. Подобный метод реализован и в пакете `e1071` – функция называется `cmeans()`, но в этом случае вместо количества кластеров можно указать предполагаемые центры, вокруг которых будут группироваться элементы.

Методы классификации без обучения могут работать не только с интервальными и шкальными, но и с номинальными данными. Для этого надо лишь закодировать номинальные признаки в виде нулей и единиц. Дальше умные функции типа `daisy()` сами выберут коэффициент сходства, а можно и указать его вручную (например, так: `dist(..., method="binary")`). А есть ли многомерные методы, ко торые могут работать с таблицами сопряженности? Оказывается, такие не просто существуют, но широко используются в разных областях, например в экологии.

Анализ связей (correspondence analysis) – один из них. Как и любой многомерный метод, он позволяет визуализировать структуру данных, причем работает он через составление таблиц сопряженности. Простой вариант анализа связей реализован в пакете `MASS` функцией `corresp()`:

```
> library(MASS)
> caith.ru <- caith
> row.names(caith.ru) <- abbreviate(c("голубоглазые",
+ "сероглазые","кареглазые","черноглазые"), 10, method="both")
> names(caith.ru) <- abbreviate(c("блондины","рыжие",
+ "русоволосые","шатены","брюнеты"), 10, method="both")
> caith.ru
  блонд рыжие русов шатен брюне
голуб 326   38    241   110   3
серог 688   116   584   188   4
карег 343   84    909   412   26
черно 98    48    403   681   85
> biplot(corresp(caith.ru, nf = 2))
```




Данные представляют собой статистическую таблицу, созданную в результате переписи населения, где отмечались, среди прочего, цвета глаз и волос. На получившемся графике удалось визуализовать обе переменные, причем чем чаще два признака (скажем,

голубоглазость и светловолосость) встречаются вместе у одного человека, тем ближе на графике эти надписи.

Эта возможность (визуализация одновременно нескольких наборов признаков) широко используется в экологии при анализе сообществ животных и растений, обитающих на одной территории. Если есть, напри мер, данные по 10 озерам, где указаны их названия и то, какие виды рыбв них обитают, то можно будет построить график, где названия озер будут расположены ближе к тем названиям рыб, которые для этих самых озер более характерны. Более того, если в данные добавить еще и абиотические признаки озер (скажем, максимальную глубину, прозрачностьводы, придонную температуру летом), то можно будет показать сразу три набора признаков! Пакет `vegan` содержит несколько таких функций, например `cca()` и `decorana()`.

### *Классификация с обучением, или дискриминантный анализ*

Перейдем теперь к методам, которые лишь частично могут назы ваться «визуализацией». В зарубежной литературе именно их приня то называть «методами классификации». Для того чтобы работать с методами классификации с обучением, надо сначала освоить технику «обучения». Как правило, выбирается часть данных с известной групповой принадлежностью. На основании анализа этой части (трениро вочной выборки) строится гипотеза о том, как должны распределяться по группам остальные, неклассифицированные данные. В результате анализа, как правило, можно узнать, насколько хорошо работает та, или иная, гипотеза. Кроме того, методы классификации с обучением можно с успехом применять и для других целей, например, для выяснения важности признаков для классификации.

Один из самых простых методов в этой группе — линейный дискриминантный анализ (linear discriminant analysis). Его основная идея – создание функций, которые на основании линейных комбинаций значений признаков (это и есть классификационная гипотеза) «сообщают», куда нужно отнести данную особь. Вот как можно применить такой анализ:

```
> library(MASS)
> iris.train < iris[seq(1,nrow(iris),5),]
> iris.unknown < iris[-seq(1,nrow(iris),5),]
> iris.lda < lda(iris.train[,1:4], iris.train[,5])
> iris.ldap < predict(iris.lda, iris.unknown[,1:4])$class
> table(iris.ldap, iris.unknown[,5])
iris.ldap setosa versicolor virginica
setosa       40     0             0
versicolor  0     40             7
virginica   0      0            33
```

Наша тренировочная выборка привела к построению гипотезы, по которой все виды (за исключением части `virginica`) попали в «свою» группу. Заметьте, что линейный дискриминантный анализ не требует шкалирования (функция `scale()`) переменных.

Результат дискриминантного анализа можно проверить статистически. Для этого используется многомерный дисперсионный анализ (MANOVA), который позволяет выяснить соответствие между данными и моделью (классификацией, полученной при помощи дискриминационного анализа):

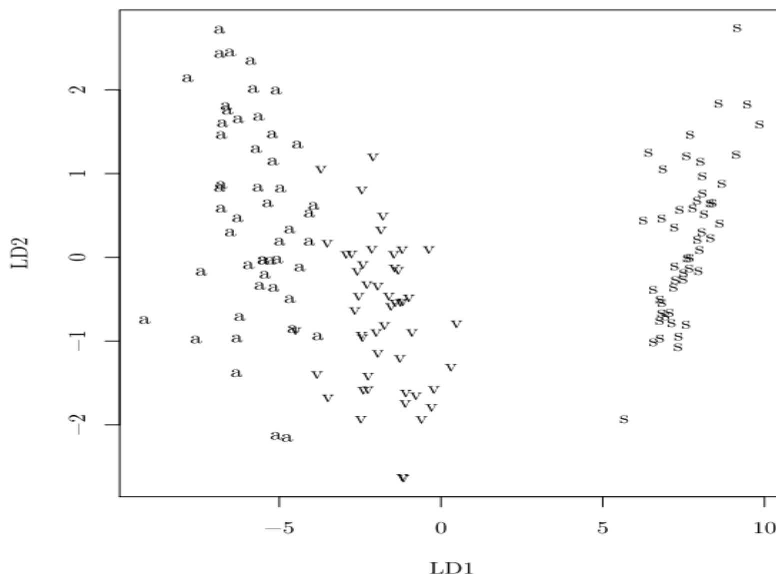
```
> ldam <- manova(as.matrix(iris.unknown[,1:4]) ~ iris.ldap)
> summary(ldam, test="Wilks")
Df    Wilks approx F num Df den Df    Pr(>F)
iris.ldap   2 0.026878  145.34       8     228 < 2.2e-16 ***
Residuals 117
```

```
---
Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Здесь имеет значение и значимость статистики Фишера (F), и само значение «Wilks», поскольку это – «likeli hood ratio», то есть отношение правдоподобия, в данном случае вероятность того, что группы не различаются. Чем ближе статистика Wilks к нулю, тем лучше классификация. В данном случае есть и достоверные различия между группами, и качественная классификация.

Линейный дискриминантный анализ можно использовать и для визуализации данных, например так:

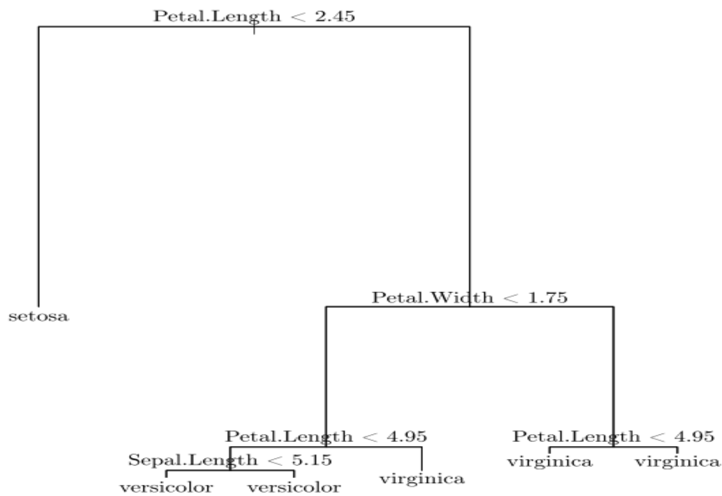
```
> iris.lda2 <- lda(iris[,1:4], iris[,5])
> iris.ldap2 <- predict(iris.lda2, dimen=2)$x
> plot(iris.ldap2, type="n", xlab="LD1", ylab="LD2")
> text(iris.ldap2,labels=abbreviate(iris[,5],1,
+ method="both.sides"))
```




Здесь мы в качестве тренировочной использовали всю выборку целиком. Как видно, виды на «v» разделились даже лучше, чем в предыдущих примерах, поскольку дискриминантный анализ склонен часто *переоценивать* различия между группами. Это свойство, а также параметричность метода привели к тому, что в настоящее время этот тип анализа используется все реже.

На замену дискриминантному анализу придумано множество методов с похожим принципом работы. Один из самых оригинальных это так называемые «деревья классификации», или «деревья решений»(«classification trees», или «decision trees»). Они позволяют выяснить, какие именно показатели могут быть использованы для разделения объектов на заранее заданные группы. В результате строится ключ, в котором на каждой ступени объекты делятся на две группы:

```
> library(tree)
> iris.tree <- tree(iris[,5] ~ ., iris[,-5])
> plot(iris.tree)
> text(iris.tree)
```




Здесь мы опять использовали всю выборку в качестве тренировочной (чтобы посмотреть на пример частичной тренировочной выборки, можно набрать `?predict.tree`). Получился график, очень похожий на так называемые определительные таблицы, по которым биологи определяют виды организмов. Из графика легко понять, что к `setosa` от носятся все ирисы, у которых длина лепестков меньше 2.45 (читать график надо влево), а из оставшихся ирисов те, у которых ширина лепестков меньше 1.75, а длина — меньше 4.95, относятся к `versicolor`.Все остальные ирисы относятся к `virginica`.

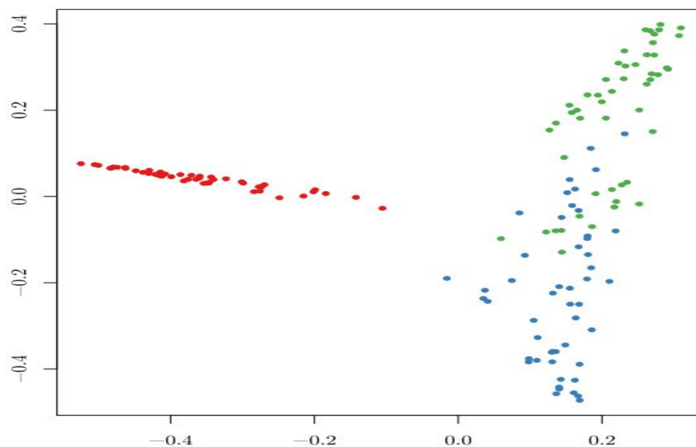
Деревья классификации реализованы и в другом пакете — `rpart`.

Еще один, набирающий все большую популярность метод тоже идеологически близок к деревьям классификации. Он называется «Random Forest», поскольку основой метода является производство большого количества классификационных «деревьев».

```
> library(randomForest)
> set.seed(17)
> iris.rf<-randomForest(iris.train[,5]~.,data=iris.train[,1:4])
> iris.rfp <- predict(iris.rf, iris.unknown[,1:4])
> table(iris.rfp, iris.unknown[,5])
> iris.rfp   setosa versicolor virginica
iris.rfp setosa versicolor virginica
setosa       40      0            0
versicolor  0      39            9
virginica   0       1           31
```

Заметна значительно более высокая эффективность этого методапо сравнению с линейным дискриминантным анализом. Кроме того, Random Forest позволяет выяснить значимость («importance») каждого признака, а также дистанции между всеми объектами тренировочной выборки («proximity»), которые затем можно использовать для кластеризации или многомерного шкалирования. Наконец, этот метод позволяет производить «чистую визуализацию» данных, то есть он может работать как метод классификации без обучения:

```
> set.seed(17)
> iris.urf <- randomForest(iris[,-5])
> MDSplot(iris.urf, iris[,5])
```




Великое множество методов «data mining», разумеется, нельзя охватить в одной главе. Однако нельзя не упомянуть еще об одном современном методе классификации с обучением, основанном на идее вычисления параметров гиперплоскости, разделяющей различные группы в многомерном пространстве признаков, «Support Vector Machines».

```
> library(e1071)
> iris.svm <- svm(Species ~ ., data = iris.train)
> iris.svmp <- predict(iris.svm, iris[,1:4])
> table(iris.svmp, iris[,5])
iris.svmp   setosa versicolor virginica
setosa      50          0          0
versicolor  0          50         14
virginica   0           0         36
```

Этот метод изначально разрабатывался для случая бинарной классификации, однако в R его можно использовать (как мы здесь видим)и для большего числа групп. Работает он эффективнее дискриминантного анализа, хотя и не так точно, как Random Forest.

# 9. Анализ временныхрядов

В этой главе рассмотрим только самые основные принципы работы с временными рядами. За более подробными сведениями следует обратиться к специальной литературе.

### ***Что такое временные ряды***

Во многих областях деятельности людей замеры показателей проводятся не один раз, а повторяются через некоторые интервалы времени. Иногда этот интервал равен многим годам, как при переписи населения, иногда – дням, часам, минутам и даже секундам, но интервал между измерениями во временном ряду есть всегда. Его называют *интервалом выборки* (sampling interval). А образующийся в результате выборки ряд данных называют *временным рядом* (time series).

В любом временном ряду можно выделить две компоненты:

- *неслучайную (детерминированную) компоненту;*
- *случайную компоненту.*

Неслучайная компонента обычно наиболее интересна, так как она дает возможность проверить гипотезы о производящем временной ряд явлении. Математическая модель неслучайной компоненты может быть использована для прогноза поведения временного ряда в будущем.

### *Тренд и период колебаний*

Если явление, результатом которого является изучаемый ряд, за висит от времени года (или времени суток, или дня недели, или ино го фиксированного периода календаря), то из неслучайной компонентыможет быть выделена еще одна компонента — *сезонные* колебания явле ния. Ее следует отличать от циклической компоненты, не привязанной к какому-либо естественному календарному циклу.

Под *трендом* (тенденцией) понимают неслучайную и непериодическую компоненту ряда. Первый вопрос, с которым сталкивается исследователь, анализирующий временной ряд, – существует ли в нем тренд? При наличии во временном ряде тренда и периодической компоненты значения любого последующего значения ряда зависят от предыдущих. Силу и знак этой связи можно измерить уже знакомым вам инструментом — коэффициентом корреляции. Корреляционная зависимость между последовательными значениями временного ряда называется *автокорреляцией*.

Коэффициент автокорреляции первого порядка определяет зависимость между соседними значениями ряда $t_n$ и $t_{n-1}$, больших порядков – между более отдаленными значениями. *Лаг* (сдвиг) автокорреляции – это количество периодов временного ряда, между которыми определяется коэффициент автокорреляции. Последовательность коэффициентов автокорреляции первого, второго и других порядков называется *автокорреляционной функцией* временного ряда.

Анализ автокорреляционной функции позволяет найти лаг, при ко тором автокорреляция наиболее высокая, а следовательно, связь междутекущим и предыдущими уровнями временного ряда наиболее тесная. Если значимым оказался только первый коэффициент автокорреляции(коэффициент автокорреляции первого порядка), временной ряд, скорее всего, содержит только тенденцию (тренд). Если значимым оказался коэффициент автокорреляции, соответствующий лагу $n$, то ряд содержит циклические колебания с периодичностью в $n$ моментов времени. Если ни один из коэффициентов автокорреляции не является значимым, то можно сказать, что, либо ряд не содержит тенденции (тренда)и циклических колебаний, либо ряд содержит нелинейную тенденцию, которую линейный коэффициент корреляции выявить не способен.

Взаимная корреляция (*кросс-корреляция*) отражает, есть ли связь между рядами. В этом случае расчет происходит так же, как и для автокорреляции, только коэффициент корреляции рассчитывается междуосновным рядом и рядом, связь с которым основного ряда нужно определить. Лаг (сдвиг) при этом может быть и отрицательной величиной, поскольку цель расчета взаимной корреляции — выяснение того, какой из двух рядов «ведущий».

### *Построение временного ряда*

Анализ временного ряда часто строится вокруг объяснения выявленного тренда и циклических колебаний значений ряда в рамках некоторой статистической модели.

Найденная модель позволяет: прогнозировать будущие значения ряда (forecasting), генерировать искусственный временной ряд, все статистические характеристики которого эквивалентны исходному (simulation), и заполнять пробелы в исходном временном ряду наиболее вероятными значениями.

Нужно отличать *экстраполяцию* временного ряда (прогноз будущих значений ряда) от *интерполяции* (заполнение пробелов между имеющимися данными ряда). Не всегда модели ряда, пригодные для интерполяции, можно использовать для прогноза. Например, полином (степенное уравнение с коэффициентами) очень хорошо сглаживает исходный ряд значений и позволяет получить оценку показателя, который описывает данный ряд в промежутках между значениями ряда. Но если мы попытаемся продлить полином за стартовое значение ряда, то получим совершенно случайный результат. Вместе с тем обычный линейный тренд, хотя и не столь изощренно следует изгибам внутри ряда,дает устойчивый прогноз развития ряда в будущем.

Разные участки ряда могут описывать различные статистические модели, в этом случае говорят, что ряд *нестационарный*. Нестационарный временной ряд во многих случаях удается превратить в стационарный путем преобразования данных.

В базовые возможности R входят средства для представления и анализа временных рядов. Основным типом временных данных является «ts», который представляет собой временной ряд, состоящий из значений, разделенных одинаковыми интервалами времени. Временные ряды могут быть образованы и неравномерно отстоящими друг от друга значениями. В этом случае следует воспользоваться специальными типами данных – zoo и its, которые становятся доступными после загрузки пакетов с теми же именами.

Часто необходимо обрабатывать календарные даты. По умолчанию read.table() считывает все нечисловые даты (например, «12/15/04» или «2004-12-15») как текстовые строки или факторы. Поэтому после загрузки таких данных при помощи read.table() нужно обязательно применить функцию as.Date(). Она «понимает» описание шаблона даты и преобразует строки символов в тип данных Date. В последних версиях R она работает и с факторами:

```
> dates.df <- data.frame(dates=c("2011-01-01","2011-01-02",
+ "2011-01-03","2011-01-04","2011-01-05"))
> str(dates.df$dates)
chr [1:5] "2011-01-01" "2011-01-02" "2011-01-03" ...
> dates.1 <- as.Date(dates.df$dates, "%Y-%m-%d")
> str(dates.1)
Date[1:5], format: "2011-01-01" "2011-01-02" "2011-01-03"
"2011-01-04" ...
```

А вот как создаются временные ряды типа ts:

```
> ts(1:10,   # ряд данных
+ frequency = 4,  # поквартально
+ start = c(1959, 2)) # начинаем во втором квартале 1959 года
   Qtr1   Qtr2 Qtr3   Qtr4
1959        1     2     3
1960 4      5     6     7
1961 8      9     10
```

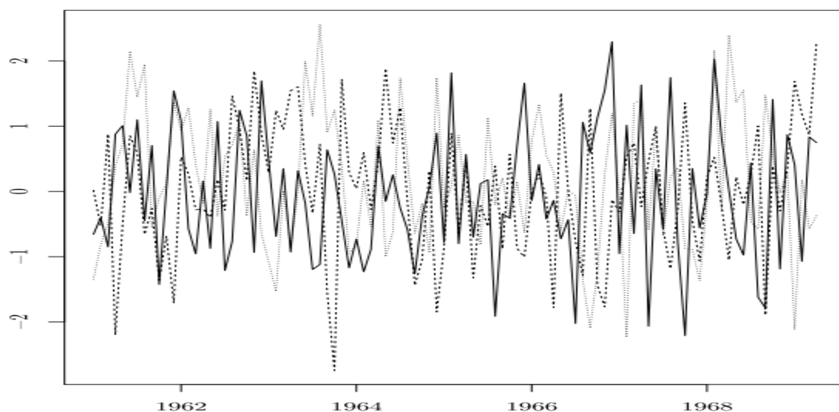
Можно конвертировать сразу матрицу, тогда каждая колонка матрицы станет отдельным временным рядом:

# матрица данных из трех столбцов

```
> z <- ts(matrix(rnorm(300), 100, 3),
+ start=c(1961, 1), # начинаем в 1-й месяц 1961 года
+ frequency=12)   # помесячно
class(z)
[1] "mts" "ts"   # тип данных - многомерный временной ряд
```

Ряды отображаются графически с помощью стандартной функции plot():

```
> plot(z,
+ plot.type="single", # поместить все ряды на одном графике
+ lty=1:3) # типы линий временных рядов
```




Методы для анализа временных рядов и их моделирования включают ARIMA-модели, реализованные в функциях arima(), AR() и VAR(), структурные модели в StructTS(), функции автокорреляции и частной автокорреляции в acf() и pacf(), классическую декомпозицию временного ряда в decompose(), STL-декомпозицию в stl(), скользящее среднее и авторегрессивный фильтр в filter().

Покажем на примере, как применить некоторые из этих функций. В текстовом файле leaf2-4.txt записаны результаты длившихся трое суток непрерывных наблюдений над хищным растением росянкой. Листья этого растения постоянно открываются и за крываются «в надежде» поймать и затем переварить мелкое насекомое. Файл содержит результаты наблюдений над четвертым листом второго растения, поэтому он так называется. Состояние листа отмечали каждые 40 минут, всего в сутки делали 36 наблюдений. Попробуем сделать временной ряд из колонки FORM, в которой закодированы изменения формы пластинки листа (1 — практически плоская, 2 — вогнутая); это шкальные данные, поскольку можно представить себе форму = 1.5. Сначала посмотрим, как устроен файл данных, при помощи команды file.show("leaf2-4.txt"):

```
K.UVL;FORM;ZAGN;OTOGN;SVEZH;POLUPER;OSTATKI 2;1;2;2;1;0;1
1;1;2;1;0;1;1
1;1;2;1;1;0;1
2;2;3;1;1;0;0
1;2;3;1;1;0;0
...
```

Теперь можно загружать его:

```
> leaf < read.table("data/leaf2-4.txt", head=TRUE,
+ as.is=TRUE, sep=";")
```

Чтобы посмотреть, все ли в порядке, можно воспользоваться функцией str:

```
> str(leaf)
```

Преобразуем колонку FORM во временной ряд:

```
> forma <- ts(leaf$FORM, frequency=36)
```

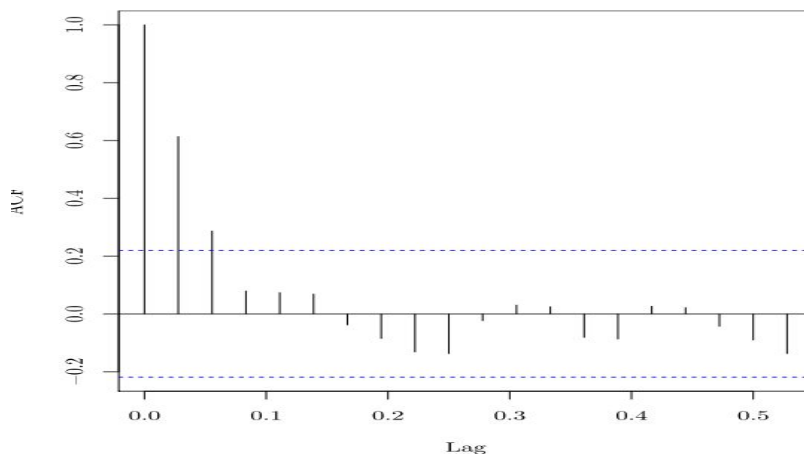
Проверим:

```
> str(forma)
Time-Series [1:80] from 1 to 3.19: 1 1 1 2 2 2 2 2 2 2 ...
```

Все правильно, наблюдения велись чуть больше трех (3.19) суток. Ну вот, а теперь попробуем понять, насколько наши данные периодичны, и есть ли в них тренд:

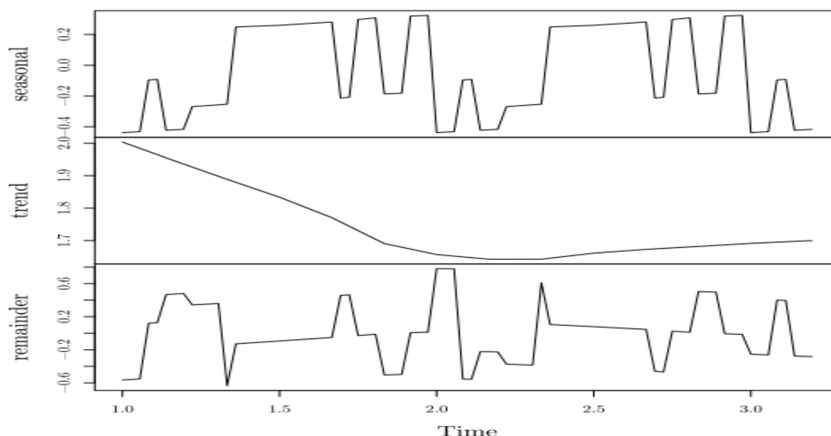
```
> (acf(forma, main=""))
```

Autocorrelations of series 'forma', by lag




Эта команда («auto-correlation function», ACF) выводит коэффициенты автокорреляции и рисует график автокорреляции, на котором в нашем случае можно увидеть, что значимой периодичности нет – все пики лежат внутри обозначенного пунктиром доверительного интервала, за исключением самых первых пиков, которые соответствуют авто корреляции без лага или с очень маленьким лагом. По сути, это пока зывает, что в пределах 0.05 суток (у нас период наблюдений – сутки) следующее состояние листа будет таким же, как и текущее, а вот на б**о**льших интервалах таких предсказаний сделать нельзя. То, что волнообразный график пиков как бы затухает, говорит о том, что в наших данных возможен тренд. Проверим это:

```
> plot(stl(forma, s.window="periodic")$time.series, main="")
```




Действительно, наблюдается тенденция к уменьшению значения фор мы с течением времени. Мы выяснили это при помощи функции `stl()` (названа по имени метода, STL — «Seasonal Decomposition of Time Se ries by Loess»), которая вычленяет из временного ряда три компоненты:сезонную (в данном случае суточную), тренд и случайную, при помощи сглаживания данных методом LOESS.

***Прогноз***

Научившись базовым манипуляциям с временны́ми рядами, мы можем попробовать решить задачу построения модели временного ряда. Построенная модель позволит нам проследить развитие наблюдаемого процесса в будущем. Кроме того, мы сможем рассмотреть применение более сложных функций анализа временны́х рядов, а заодно

познакомить читателя с общими принципами сравнительного анализа статистических моделей.

Наш пример будет заключаться в прогнозе числа абонентов интернет провайдера. Исходные данные состоят из:

- *данных о подключениях за декабрь 2004 года;*
- *помесячных данных о подключениях в 2005–2008 годах.*

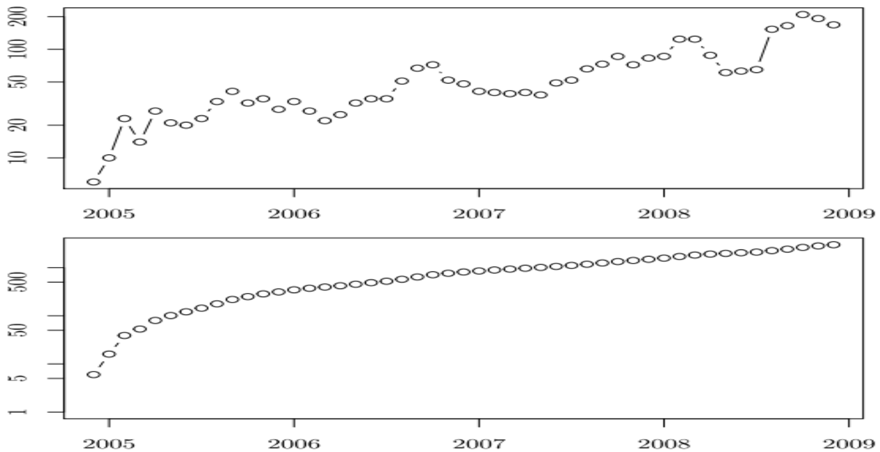
```
> polzovateli <- ts(read.table("data/data.txt")$V3,
+ start=c(2004,12), frequency=12)
```

Общее количество абонентов на каждый месяц отчетного периода составило:

```
> cum.polzovateli <- ts(cumsum(polzovateli),
+ start=c(2004,12), frequency=12)
```

Отобразим данные помесячного подключения и изменения количества пользователей графически. Оба временных ряда показывают экспоненциальный рост, поэтому выведем их в полулогарифмических координатах:

```
> oldpar <- par(mfrow=c(2,1))
> plot(polzovateli, type="b", log="y", xlab="")
> plot(cum.polzovateli, type="b", ylim=c(1,3000), log="y")
> par(oldpar)
```




Линейный рост временных рядов в полулогарифмических координатах подтверждает предположение об экспоненциальном росте во времени и количества подключений в месяц, и общего количества пользователей.

Попробуем теперь построить модель временного ряда общего числа подключений распространенным методом ARIMA («Autoregressive In tegrated Moving Average», авторегрессия интегрированного скользящего среднего). Нам надо выбрать подходящее значение параметра `order`, для этого мы будем перебирать различные значения каждого из трех его компонентов:

```
> model01 <- arima(cum.polzovateli, order=c(0,0,1))
> . . .
> model014 <- arima(cum.polzovateli, order=c(0,0,14))
```

Можно написать здесь цикл с оператором `for` (используя функцию `assign()`:

```
> for (m in 1:14)
+ {
+ assign(paste("model0",m,sep=""), arima(cum.polzovateli,
```

```
+ order=c(0,0,m)))
+ }
```

Теперь сравним модели. «Лучшая» модель будет соответствоватьминимуму AIC:

```
> plot(AIC(model01,model02,model03,model04, model05, model06,
+ model07, model08, model09, model010, model011, model012,
+ model013, model014), type="b")
```

На графике можно увидеть, что по ходу кривой первый минимум наблюдается в районе компонента, соответствующего коэффициенту = 12, а дальше ход вычислений становится нестабильным.

Теперь выберем лаг авторегресии (первый элемент `order`):

```
> model012 <- arima(cum.polzovateli, order=c(0,0,12))
> model112 <- arima(cum.polzovateli, order=c(1,0,12))
> model212 <- arima(cum.polzovateli, order=c(2,0,12))
> model312 <- arima(cum.polzovateli, order=c(3,0,12))
> model412 <- arima(cum.polzovateli, order=c(4,0,12))
```

Здесь тоже можно применить AIC:

```
> AIC(model012, model112, model212, model312, model412)
   df   AIC
model012 14 477.4036
model112 15 438.3171
model212 16 435.7380
model312 17 438.1186
model412 18 439.0120
```

AIC минимален, когда лаг авторегресии равен 2, поэтому принимаем это значение для дальнейшего анализа.

Аналогично выберем второй компонент.

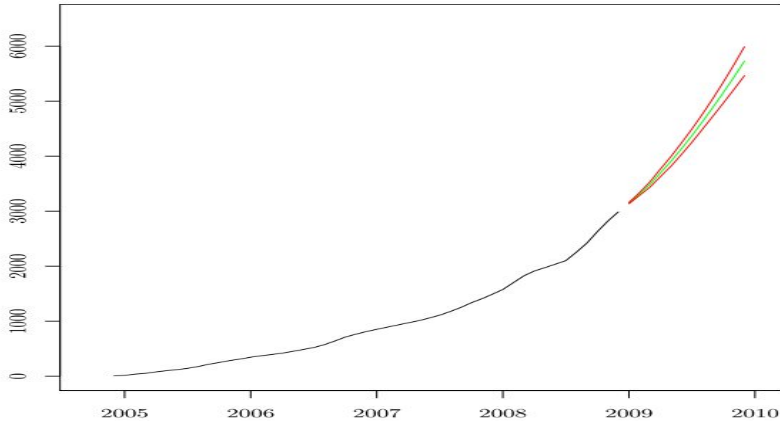
```
> model2120 <- arima(cum.polzovateli, order=c(2,0,12))
> model2121 <- arima(cum.polzovateli, order=c(2,1,12))
> . . .
> model2125 <- arima(cum.polzovateli, order=c(2,5,12))
AIC(model2120, model2121 ,model2122, model2123, model2124)
df    AIC
model2120  16 435.7380
model2121  15 421.6246
model2122  15 405.5416
model2123  15 399.1918
model2124  15 407.6942
```

Видно, что оптимальной моделью является `model2123`.

Ну а теперь, зная оптимальную модель, построим прогноз изменения общего числа абонентов на 2009 год:

```
> plot(cum.polzovateli, # Накопленное число пользователей
+ xlim=c(2004.7,2010), ylim=c(0,6500))
# Границы включают и данные прогноза. Линия прогноза:
> lines(predict(model2123, n.ahead=12, se.fit = TRUE)$pred,
+ col="green")
# Верхняя граница прогноза:
> lines(predict(model2123, n.ahead=12, se.fit = TRUE)$se +
```

```
+ predict(model2123, n.ahead=12, se.fit = TRUE)$pred,
+ col="red")
# Нижняя граница прогноза:
> lines(-predict(model2123, n.ahead=12, se.fit = TRUE)$se +
+ predict(model2123, n.ahead=12, se.fit = TRUE)$pred,
+ col="red")
```




Максимальное и минимальное ожидаемое количество абонентов по месяцам 2009 года составит:

```
> round(predict(model2123,
+ n.ahead=12,     # период прогноза
+ se.fit = TRUE)$se +  # берем из объекта только ошибку
+ predict(model2123,    # складываем ошибку и данные прогноза
+ n.ahead=12,
+ se.fit = TRUE)$pred)  # берем только данные самого прогноза

Jan  Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec
2009 3158 3337 3533 3767 3992 4239 4494 4765 5050 5349 5663 5991

> round(-predict(model2123, n.ahead=12, se.fit = TRUE)$se +
+ predict(model2123, n.ahead=12, se.fit = TRUE)$pred)

Jan    Feb  Mar  Apr  May  Jun  Jul  Aug  Sep  Oct  Nov  Dec2009
3134 3283 3437 3628 3817 4031 4253 4490 4728 4969 5213 5463
```

Теперь можно заняться «предсказанием курса доллара на следующую неделю».

Итак, в файле `dollar.txt` содержатся значения курса доллара Центрального Банка за 11 недель. Попробуем предсказать курс доллара на две недели вперед. Чтобы проверить эффективность предсказания, возьмём для модели данные по 26 июля, а предскажем последние две недели.

Поступим ровно так же, как впримере о пользователях провайдера, то есть сначала превратим данные во временной ряд:

```
> dollar1 <- read.table("dollar.txt", dec=",")$V3
> dollar <- ts(dollar1[1:56], frequency=7)
```

Десятичным разделителем была запятая! Еще мы учли, что курс доллара имеет недельную периодичность, и организовали данные понедельно.

Затем проверим разные коэффициенты модели ARIMA:

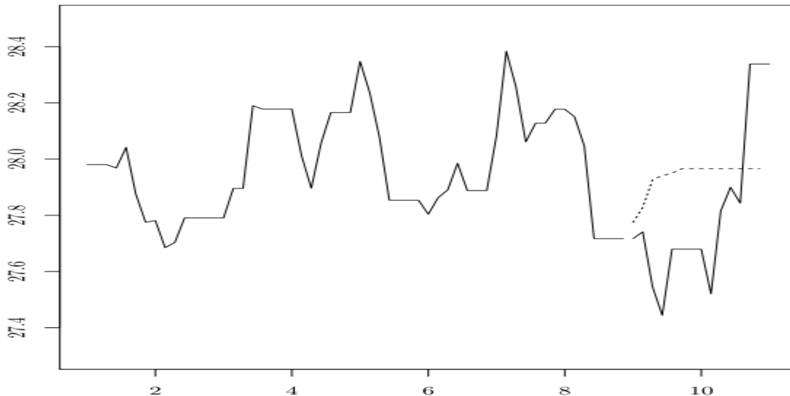
```
> for (m in 1:7)
+ {
+ mm <- paste("model0", m, sep="")
+ assign(mm, arima(dollar, order=c(0,0,m)))
+ cat(paste(mm, ": ", AIC(get(mm)), "\n", sep=""))
+ }
model01: -67.8129312683548
model02: -80.9252194390681
model03: -82.7498433251648
model04: -82.4022042909309
model05: -84.5913013237983
model06: -83.0836200480987
model07: -82.2056122336345
> for (m in 0:5)
+ {
+ mm <- paste("model", m, "05", sep="")
+ assign(mm, arima(dollar, order=c(m,0,5)))
+ cat(paste(mm, ": ", AIC(get(mm)), "\n", sep=""))
+ }
model005: -84.5913013237983
model105: -82.772885907062
model205: -81.8467316861654
model305: -79.9942752423287
model405: -83.3710277055304
model505: -80.7835758224462
> for (m in 0:5)
+ {
+ mm <- paste("model0", m, "5", sep="")
+ assign(mm, arima(dollar, order=c(0,m,5)))
+ cat(paste(mm, ": ", AIC(get(mm)), "\n", sep=""))
+ }
model005: -84.5913013237983
model015: -78.0533071948488
model025: -69.4383206785473
model035: -58.0629434523778
model045: -36.1736715036462
model055: -18.2117126978254
```

Для ускорения процесса был придуман цикл, который не только меняет значения коэффициентов и имена моделей, но еще вычисляет и выводит AIC для каждой из них. Обратите внимание на функцию `get()`, ее использование в чем-то противоположно `assign()`: если имеется имя `name`, то `get(name)` будет искать объект с именем `name` и пере давать его наружу именно как *объект*, а не как строку текста. Функция `cat()` использовалась для печати «наружу», без нее цикл бы ничего не вывел.

Итого, модель `model005` имеет минимальный AIC, и, стало быть, нам лучше выбрать для предсказания именно ее. Построим график предсказания, добавив туда еще и реальные значения курса доллара за последние две недели, с 27 июля по 9 августа

```
> plot(dollar, xlim=c(1,11), ylim=c(27.3,28.5))
```

```
> lines(predict(model005, n.ahead=14, se.fit = TRUE)$pred,
+ lty=2)
> lines(ts(dollar1[56:70], start=9, frequency=7))
```




Предсказанные значения обозначены пунктирной линией. Наша модель не смогла предсказать резкое падение, но тем не менее указала на некоторое увеличение курса в следующие две недели. На самом деле результаты даже лучше, чем можно было ожидать, поскольку график автокорреляций показывает, что значимых автокорреляций в нашем ряду мало.

## 10. Статистическая разведка

Вот и окончилось наше изложение основных методов статистики и их использования в R. Теперь настало время применить знания на практике. Но перед тем, как сделать этот шаг, неплохо представить полученные знания в «концентрированном» виде. Итак, как нужно анализировать данные?

***Первичная обработка данных***

Вначале надо выяснить несколько вопросов:

- *Какой тип у данных, каким способом (способами) они представлены.*
- *Однородны ли данные — в каких единицах измерены показатели.*
- *Параметрические или непараметрические у нас данные – можно ли предположить нормальное распределение данных.*
- *Нужна ли «чистка» данных – есть ли пропущенные данные, выбросы, опечатки.*

Если в таблице есть, кроме цифр, буквы, то нужно тщательно проследить, нет ли опечаток. Иногда опечатки бывают почти невидимыми, скажем, русская буква «с» и английская «c» по виду неразличимы, а вот программа обработки данных посчитает их разными. К счастью, базовые команды `read.table()` и `summary()` позволяют «выловить» подавляющее большинство подобных проблем. Кстати, если при попытке загрузки данных в R возникает ошибка, не спешите винить программу: бо**́**льшая часть таких ошибок – это результат неправильного оформления и/или ввода данных.

***Окончательная обработка данных***

Для того чтобы разобраться в многочисленных способах анализа, обратимся к следующей таблице:

<table>
<tr><th colspan="3">Данные</th><th>Одна группа</th><th>Две группы: различия</th><th>Две группы: связи</th><th>Три и более групп: связи</th><th>Три и более групп: общая картина</th></tr>
<tr><td rowspan="4">Количественные</td><td rowspan="2">Параметрические</td><td>Независимые</td><td rowspan="6">summary()</td><td>t.test()</td><td rowspan="2">cor.test(..., method="pe")</td><td>oneway.test(), paiwise.t.test(), anova(), lm()</td><td>lda(), manova()</td></tr>
<tr><td>Зависимые</td><td>t.test(..., paired = TRUE)</td><td>–</td><td>–</td></tr>
<tr><td rowspan="2">Непараметрические</td><td>Независимые</td><td>wilcox.test()</td><td rowspan="2">cor.test(..., method="sp")</td><td>kruskal.test()</td><td>pca(), tree(), cor(), hclust(), isoMDS(), cmdscale()</td></tr>
<tr><td>Зависимые</td><td>wilcox.test (..., paired = TRUE)</td><td>–</td><td>–</td></tr>
<tr><td rowspan="2">Номинальные или шкальные</td><td rowspan="2">Непараметрические</td><td>Независимые</td><td>chisq.test(), prop.test(), binom.test()</td><td>glm(..., "binomial")</td><td>–</td><td>cor(), dist(), hclust(), isoMDS(), corresp()</td></tr>
<tr><td>Зависимые</td><td>mcnemar.test()</td><td>–</td><td>–</td><td>–</td></tr>
</table>

Она устроена очень просто: надо ответить всего лишь на три вопроса, и способ обработки найдется в соответствующей ячейке. Главный вопрос — это тип данных. Если данные количественные, то подойдет верхняя половина таблицы, если качественные или порядковые — нижняя половина. Затем надо понять, можно ли посчитать ваши данные распределенными нормально, то есть параметрическими, или по крайней мере без опаски закрыть глаза на некоторое уклонение от нормальности. И наконец, нужно понять, зависят ли разные колонки данных друг от друга, то есть ст**о**ит или не ст**о**ит применять парные методы сравнения.

Найдя в таблице нужный метод, загляните в указатель и перейдите на страницу с более подробным описанием функции. А можно просто ввести команду `help()`.

### *Отчет*

Любое полноценное исследование оканчивается отчетом – презентацией, статьей или публикацией в Интернете. В R есть множество автоматизированных средств подготовки отчетов.

Таблицы, созданные в R, можно сохранить в форматах LaTeX или HTML при помощи пакета `xtable`. Естественно, хочется пойти дальше и сохранять в каком-нибудь из этих форматов вообще всю R-сессию. Для HTML такое возможно, если использовать пакет `R2HTML`:

```
> library(R2HTML)
> dir.create("example")
> HTMLStart("example")
> 2+2
> plot(1:20)
> HTMLplot()
```

```
> HTMLStop()
```

В рабочей директории будет создана поддиректория example, и туда будут записаны HTML-файлы, содержащие полный отчет о текущей сессии, в том числе и созданный график.

Есть и другие системы генерации отчетов, например пакет brew, который позволяет создавать автоматические отчеты в текстовой форме (разумеется, без графиков), и пакет odfWeave, который может работатьс ODF (формат OpenOffice.org).

И все-таки полностью полагаться на автоматику никогда не стóит. Обязательно проверьте, например, полученные графики — достаточен ли размер графических файлов, нет ли проблем с цветами. И не за будьте запомнить в файл историю ваших команд. Это очень поможет, если придется опять вернуться к обработке этих данных. А если вы захотите сослаться в своем отчете на тот замечательный инструмент, который помог вам обработать данные, не забудьте вставить в список литературы то, что выводит строка

```
> citation()
```

## 11. Самое необходимое

Здесь собрано примерно пятьдесят самых необходимых команд, операторов и обозначений.

- ? Помощь
- <- Присвоить то, что справа, тому, кто слева
- [ Выбрать часть объекта
- $ Вызвать элемент списка по имени
- anova() Дисперсионный анализ
- apply() Применить функцию к объекту
- as.character() Конвертировать в текст
- as.numeric() Конвертировать в числа
- boxplot() Ящик-с-усами, боксплот
- c() Соединить в вектор
- cbind() Соединить в матрицу по колонкам
- chisq.test() Тест хи-квадрат
- cor() Корреляция между многими переменными
- colSums() Просуммировать каждую колонку
- cor.test() Корреляционный тест
- data.frame() Сделать таблицу данных
- dotchart() Точечный график (замена «пирогу»)
- example() Вызвать пример работы команды
- file.show() Показать файл
- function() Сделать новую функцию
- head() Показать первые строчки таблицы help() Помощь
- hist() Гистограмма
- legend() Дополнение к графику: легенда
- length() Длина переменной
- lines() Дополнение к графику: линии
- lm() Линейная модель
- log() Натуральный логарифм
- max() Наибольшее значение
- mean() Среднее
- median() Медиана
- min() Наименьшее значение

- NA Пропущенное значение
- names() Вывести имена элементов
- nrow() Сколько строк?
- order() Сортировать
- plot() График
- points() Дополнение к графику: точки
- predict() Предсказать значения
- q() Выйти из R
- qqnorm(); qqline() Проверка на нормальность распределения
- rbind() Соединить в матрицу по строкам
- read.table() Чтение файла данных
- rep() Создать последовательность одинаковых элементов
- sample() Выбрать случайным образом
- savehistory() Сохранить историю команд
- scale() Нормировать переменные
- sd() Стандартное отклонение
- source() Загрузить скрипт
- str() Структура переменной
- summary() Главные описательные статистики (сводная статистика)
- t() Повернуть (транспонировать) таблицу
- t.test() Тест Стьюдента
- table() Кросс-табуляция
- tapply() Применить функцию и кросс-табуляцию к объекту
- text() Дополнение к графику: нанести на график текст
- wilcox.test() Тест Вилкоксона или Манна-Уитни
- write.table() Записать таблицу данных на диск

## 12. Литература


1. А.Б. Шипунов, Е.М. Балдин, П.А. Волкова и др.: Наглядная статистика. Используем R! ДМК Пресс, 2014, 298 с. (ISBN: 978-5-97060-094-8)
2. Роберт И. Кабаков: R в действии. Анализ и визуализация данных на языке R М.: ДМК Пресс, 2014. – 588 с. (ISBN 978-5-947060-077-1)
3. Дуглас Люк: Анализ сетей (графов) в среде R. Руководство пользователя. Цветное издание. (перевод с английского, ISBN: 978-5-97060-428-1 )
4. Джеймс Г., Уиттон Д., Хасти Т., Тибширани Р.: Введение в статистическое обучение с примерами на языке R. (ISBN: 978-5-97060-293-5 )
5. Мастицкий С.Э., Шитиков В.К.: Статистический анализ и визуализация данных с помощью R (ISBN: 978-5-97060-301-7)
6. Шитиков В.К., Мастицкий С.Э. Классификация, регрессия и другие алгоритмы Data Mining с использованием R Тольятти, Лондон, 2017, 351 с.
7. Митин И.В., Русаков В.С. Анализ и обработка экспериментальных данных. М. издательство НЭВЦ ФИПТ, 1998, 48 с.
8. Волкова П. А., Шипунов А. Б. Статистическая обработка данных в учебно-исследовательских работах. М.: Форум, 2012, 96 с.
9. Кимбл Г. Как правильно пользоваться статистикой. М.: Финансы и статистика, 1982, 294 с.

10. Тьюки Дж. Анализ результатов наблюдений. Разведочный анализ. — М.: Мир, 1981, 695 с.
11. Тюрин Ю. Н., Макаров А. Л. Анализ данных на компьютере. — М.: ИНФРА-М, 1995, 284 с.
12. Факторный, дискриминантный и кластерный анализ. — М.: Финансы и статистика, 1989, 215 с.
13. Crawley M. R Book., England: John Whiley & Sons, 2007. — 942 p. Dalgaard P. Introductory statistics with R. 2 ed. — USA: Springer Science, Business Media, 2008, 363 p.
14. McKillup S. Statistics explained. An introductory guide for life scientists. — England: Cambridge University Press, 2005, 267 p.
15. Rowntree D. Statistics without tears. England: Clays, 2000, 195 p.

Онлайн курсы:

1. Stepic: Анализ данных в R
2. Stepic: Анализ данных в R. Часть 2
3. Stepic: Основы программирования на R
4. swirl: R пакет для интерактивного обучения на хабре
5. Coursera: R Programming
6. Data Camp: Introduction to R
7. Udacity: Data Analysis with R. Visually Analyze and Summarize Data Sets
8. swirl teaches you R programming and data science interactively