Наглядная статистика. Используем R!

А. Б. Шипунов, Е. М. Балдин, П. А. Волкова, А. И. Коробейников, С. А. Назарова, С. В. Петров, В. Г. Суфиянов

13 июля 2014 г.

Оглавление

Предисловие

Эта книга написана для тех, кто хочет научиться обрабатывать данные. Такая задача возникает очень часто, особенно тогда, когда нужно выяснить ранее неизвестный факт. Например: есть ли эффект от нового лекарства? Или: различаются ли рейтинги двух политиков? Или: как будет меняться курс доллара на следующей неделе?

Многие люди думают, что этот неизвестный факт можно выяснить, если просто немного подумать над данными. К сожалению, часто это совершенно не так. Например, по опросу 262 человек, выходящих с избирательных участков, выяснилось, что 52% проголосовало за кандидата А, а 48% — за кандидата Б (естественно, мы упрощаем ситуацию). Значит ли это, что кандидат А победил? Подумав, многие сначала скажут «Да», а через некоторое время, возможно, «Кто его знает». Но есть простой (с точки зрения современных компьютерных программ) «тест пропорций», который позволяет не только ответить на вопрос (в данном случае «Нет»), но и вычислить, сколько надо было опросить человек, чтобы можно было бы ответить на такой вопрос. В описанном случае это примерно 5000 человек (см. объяснение в конце главы про одномерные данные)!

В общем, если бы люди знали, что можно сделать методами анализа данных, ошибок и неясностей в нашей жизни стало бы гораздо меньше. К сожалению, ситуация в этой области далека от благополучия. Тем из нас, кто заканчивал институты, часто читали курс «Теория вероятностей и математическая статистика», однако кроме ужаса и/или тоски от длинных математических формул, набитых греческими буквами, большинство ничего из этих курсов не помнит. А ведь на теории вероятностей основаны большинство методов анализа данных! С другой стороны, ведь совсем не обязательно знать радиофизику для того, чтобы слушать любимую радиостанцию по радиоприемнику. Значит, для того чтобы анализировать данные в практических целях, не обязательно свободно владеть математической статистикой и теорией вероятностей. Эту проблему давно уже почувствовали многие английские и американские авторы — названиями типа «Статистика без слез» пестрят книжные полки магазинов, посвященные книгам по анализу данных.

Тут, правда, следует быть осторожным как авторам, так и читателям таких книг: многие методы анализа данных имеют, если можно так

выразиться, двойное дно. Их (эти методы) можно применять, глубоко не вникая в сущность используемой там математики, получать результаты и обсуждать эти результаты в отчетах. Однако в один далеко не прекрасный день может выясниться, что данный метод совершенно не подходил для ваших данных, и поэтому полученные результаты и результатами-то назвать нельзя... В общем, будьте бдительны, внимательно читайте про все *ограничения* методов анализа, а при чтении примеров досконально сравнивайте их со своими данными.

Про примеры: мы постарались привести как можно больше примеров, как простых, так и сложных, и по возможности из разных областей жизни, поскольку читателями этой книги могут быть люди самых разных профессий. Еще мы попробовали снизить объем теоретического материала, потому что мы знаем — очень многие учатся только на примерах. Поскольку книга посвящена такой компьютерной программе, которая «работает на текстовом коде», логично было поместить эти самые коды в текстовый файл, а сам файл сделать общедоступным. Так мы и поступили — приведенные в книге примеры можно найти на веб-странице по адресу `http://ashipunov.info/shipunov/software/r/`. Там же находятся разные полезные ссылки и те файлы данных, которые не поставляются вместе с программой.

О структуре книги: первая глава, по сути, целиком теоретическая. Если лень читать общие рассуждения, можно сразу переходить ко второй главе. Однако в первой главе есть много такой информации, которая позволит в будущем не «наступать на грабли». В общем, решайте сами. Во второй главе самые важные — разделы, начиная с «Как скачать и установить R», в которых объясняется, как работать с программой R. Если не усвоить этих разделов, все остальное чтение будет почти бесполезным. Советуем внимательно прочитать и обязательно *проработать все примеры* из этого раздела. Последующие главы составляют ядро книги, там рассказывается про самые распространенные методы анализа данных. Глава «Статистическая разведка», в которой обсуждается общий порядок статистического анализа, подытоживает книгу; в ней еще раз рассказывается про методы, обсуждавшиеся в предыдущих главах. В приложениях к книге содержится много полезной информации: там рассказано о графических интерфейсах к R, приведен простой практический пример работы, описаны основы программирования в R, приведены выдержки из перевода официальной документации. По сути, каждое приложение — это отдельный небольшой справочник, который можно использовать более или менее независимо от остальной книги.

Конечно, множество статистических методов, в том числе и довольно популярных, в книгу не вошли. Мы почти не касаемся статистических моделей, ничего не пишем о контрастах, не рассказываем о стандартных распределениях (за исключением нормального), эффектах, кри-

вых выживания, байесовых методах, факторном анализе, геостатистике, не объясняем, как делать многофакторный и блочный дисперсионный анализ, планировать эксперимент и т. д., и т. п. Наша цель — научить основам статистического анализа. А если читатель хорошо освоит основы, то любой продвинутый метод он сможет одолеть без особого труда, опираясь на литературу, встроенную справку и Интернет.

Несколько технических замечаний: все десятичные дроби в книге представлены в виде чисел с разделителем-точкой (типа `10.4`), а не запятой (типа `10,4`). Это сделано потому, что программа R по умолчанию «понимает» только первый вариант дробей. И еще: многие приведенные в книге примеры можно (и нужно!) повторить самостоятельно. Такие примеры напечатаны `машинописным шрифтом` и начинаются со значка «больше» — «`>`». Если пример не умещается на одной строке, все последующие его строки начинаются со знака «плюс» — «`+`» (не набирайте эти знаки, когда будете выполнять примеры!). Если в книге идет речь о загрузке файлов данных, то предполагается, что все они находятся в поддиректории `data` в текущей директории. Если вы будете скачивать файлы данных с упомянутого выше сайта, не забудьте создать эту поддиректорию и скопировать туда файлы данных.

Глава 1

Что такое данные и зачем их обрабатывать?

В этой главе рассказывается о самых общих понятиях анализа данных.

1.1. Откуда берутся данные

«Без пруда не выловишь и рыбку из него»,— говорит народная мудрость. Действительно, если хочешь анализировать данные, надо их сначала получить. Способов получения данных много, а самые главные — *наблюдения* и *эксперименты*.

Наблюдением будем называть 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.

Вот, например, встала задача исследовать, чем питается какое-то редкое животное. Оптимальная стратегия наблюдения здесь состоит в установке скрытых камер во всех местах, где это животное обитает. После этого останется только обработать снятое, чтобы определить вид пищи. Очень часто, однако, оптимальное решение совершенно невыполнимо, и тогда пытаются обойтись, скажем, наблюдением за животным в зоопарке. Ясно, что в последнем случае на объект оказывается воздействие, и немалое. В самом деле, животное поймали, привезли в совершенно нетипичные для него условия, да и корм, скорее всего, будет непохож на тот, каким оно питалось на родине. В общем, если наблюдения в зоопарке поставлены грамотно, то выяснено будет не то, чем вообще питается данное животное, а то, чем оно питается при содержании в определенном зоопарке. К сожалению, многие (и исследователи, и те, кто потом читает их отчеты) часто не видят разницы между этими двумя утверждениями.

Вернемся к примеру из предисловия. Предположим, мы опрашиваем выходящих с избирательных участков. Часть людей, конечно, вообще окажется отвечать. Часть ответит что-нибудь, не относящееся к делу. Часть вполне может намеренно или случайно исказить свой ответ. Часть ответит правду. И все это серьезным образом зависит от наблюдателя — человека, проводящего опрос.

Даже упомянутые выше скрытые камеры приведут к определенному воздействию: они же скрытые, но не невидимые и невесомые. Нет никакой гарантии, что наше животное или его добыча не отреагирует на них. А кто будет ставить камеры? Если это люди, то чем больше камер поставить, тем сильнее будет воздействие на окружающую среду. Сбрасывать с вертолета? Надеемся, что вам понятно, к чему это может привести.

В общем, из сказанного должно быть понятно, что наблюдение «в чистом виде» более или менее неосуществимо, поскольку всегда будет внесено какое-нибудь воздействие. Поэтому для того, чтобы адекватно работать с данными наблюдений, надо всегда четко представлять, как они проводились. Если воздействие было значительным, то надо представлять (хотя бы теоретически), какие оно могло повлечь изменения, а в отчете обязательно указать на те ограничения, которые были вызваны способом наблюдения. Не следует без необходимости применять экстраполяцию: если мы увидели, что А делает Б, нельзя писать «А всегда делает Б» и даже «А обычно делает Б». Можно лишь писать нечто вроде «в наших наблюдениях А делал Б, это позволяет с некоторой вероятностью предположить, что он может делать Б».

У эксперимента свои проблемы. Наиболее общие из них — это точный учет воздействия и наличие контроля. Например, мы исследуем действие нового лекарства. Классический эксперимент состоит в том, что выбираются две группы больных (*как* выбрать такие группы, *сколько* должно быть человек и прочее, рассмотрено дальше). Всем больным сообщают, что проводится исследование нового лекарства, но его дают только больным первой группы, остальные получают так называемое *плацебо*, внешне неотличимое от настоящего лекарства, но не содержащее ничего лекарственного. Зачем это делается? Дело в том, что если больной будет знать, что ему дают «ненастоящее» лекарство, то это скажется на эффективности лечения, потому что результат зависит не только от того, что больной пьет, но и от того, что он чувствует. Иными словами, психологическое состояние больного — это дополнительный фактор воздействия, от которого в эксперименте лучше избавиться. Очень часто не только больным, но и их врачам не сообщают, кому дают плацебо, а кому — настоящее лекарство (двойной слепой метод). Это позволяет гарантировать, что и психологическое состояние врача не повлияет на исход лечения.

Группа, которой дают плацебо (она называется *контроль*), нужна для того, чтобы отделить эффект, который может произвести лекарство, от эффекта какого-нибудь постороннего внешнего фактора. Известно, например, что уменьшение длины светового дня осенью и зимой провоцирует многие нервные заболевания. Если наше исследование придется на это время и у нас не будет контроля, то увеличение частоты заболеваний мы вполне можем принять за результат применения лекарства.

1.2. Генеральная совокупность и выборка

«Статистика знает все»,— писали Ильф и Петров в «Двенадцати стульях», имея в виду то, что обычно называют статистикой,— сбор всевозможной информации обо всем на свете. Чем полнее собрана информация, тем, как считается, лучше. Однако лучше ли?

Возьмем простой пример. Допустим, фирма-производитель решила выяснить, какой из двух сортов производимого мороженого предпочитают покупатели. Проблем бы не было, если бы все мороженое продавалось в одном магазине. На самом же деле продавцов очень много: это оптовые рынки и гипермаркеты, средние и малые магазины, киоски, отдельные мороженщики с тележками, те, кто торгует в пригородных поездах, и т. п. Можно попробовать учесть доход от продажи каждого из двух сортов. Если они сто́ят одинаково, то бо́льшая сумма дохода должна отразить больший спрос. Представим, однако, что спрос одинаков, но по каким-то причинам мороженое первого сорта тает быстрее. Тогда потерь при его транспортировке будет в среднем больше, продавцы будут покупать его несколько чаще, и получится, что доход от продажи первого сорта будет несколько выше, чем от второго. Это рассуждение, конечно, упрощает реальную ситуацию, но подумайте, сколько других неучтенных факторов стоит на пути такого способа подсчета! Анализ товарных чеков получше, однако многие конечные продавцы таких чеков не имеют и поэтому в анализ не попадут. А нам-то необходимо как раз учесть спрос покупателей, а не промежуточных продавцов.

Можно поступить иначе — раздать всем конечным продавцам анкеты, в которых попросить указать, сколько какого мороженого продано; а чтобы анкеты были обязательно заполнены, вести с этими продавцами дела только при наличии заполненных анкет. Только ведь никто не будет контролировать, *как* продавцы заполняют анкеты... Вот и получит фирма большую, подробную сводную таблицу о продажах мороженого, которая ровным счетом ничего отражать не будет.

Как же поступить? Здесь на помощь приходит идея *выборочных исследований*. Всех продавцов не проконтролируешь, но ведь нескольких-

то можно! Надо выбрать из общего множества несколько торговых точек (*как* выбирать — это особая наука, об этом ниже) и проконтролировать тамошние продажи силами самой фирмы или такими нанятыми людьми, которым можно доверять. В итоге мы получим результат, который является частью общей картины. Теперь самый главный вопрос: можно ли этот результат распространить на всю совокупность продаж? Оказывается, можно, поскольку на основе теории вероятностей уже много лет назад была создана *теория выборочных исследований*. Ее-то и называют чаще всего математической статистикой, или просто статистикой.

Пример с мороженым показывает важную вещь: выборочные исследования могут быть (и часто бывают) значительно более точными (в смысле соответствия реальности), чем сплошные.

Еще один хороший пример на эту же тему есть в результатах сплошной переписи населения России 1897 г. Если рассмотреть численность населения по возрастам, то получается, что максимальные численности (пики) имеют возрасты, кратные 5 и в особенности кратные 10. Понятно, как это получилось. Бóльшая часть населения в те времена была неграмотна и свой возраст помнила только приблизительно, с точностью до пяти или до десяти лет. Чтобы все-таки узнать, каково было распределение по возрастам на самом деле, нужно не увеличивать объем данных, а наоборот, создать выборку из нескольких процентов населения и провести комплексное исследование, основанное на перекрестном анализе нескольких источников: документов, свидетельств и личных показаний. Это даст гораздо более точную картину, нежели сплошная перепись.

Естественно, сам процесс создания выборки может являться источником ошибок. Их принято называть «ошибками репрезентативности». Однако правильная организация выборки позволяет их избежать. А поскольку с выборкой можно проводить гораздо более сложные исследования, чем со всеми данными (их называют *генеральной совокупностью*, или *популяцией*), те ошибки (ошибки точности), которые возникают при сплошном исследовании, в выборочном исследовании можно исключить.

1.3. Как получать данные

В предыдущих разделах неоднократно упоминалось, что от правильного подбора выборки 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,

а рандомизация — для того, чтобы избежать отклонений, вызванных посторонними причинами.

Принцип повторностей предполагает, что один и тот же эффект будет исследован несколько раз. Собственно говоря, для этого мы в предыдущих примерах опрашивали *множество* избирателей, ловили в заповеднике *много* животных, подбирали группы из *нескольких десятков* больных и контролировали *различных* продавцов мороженого. Нужда в повторностях возникает оттого, что все объекты (даже только что изготовленные на фабрике изделия) пусть в мелочах, но отличаются друг от друга. Эти отличия способны затуманить общую картину, если мы станем изучать объекты поодиночке. И наоборот, если мы берем несколько объектов сразу, их различия часто «взаимно уничтожаются».

Не стоит думать, что создать повторности — простое дело. К сожалению, часто именно небрежное отношение к повторностям сводит на нет результаты вроде бы безупречных исследований. Главное правило — *повторности должны быть независимы друг от друга.* Это значит, например, что нельзя в качестве повторностей рассматривать данные, полученные в последовательные промежутки времени с одного и того же объекта или с одного и того же места. Предположим, что мы хотим определить размер лягушек какого-то вида. Для этого с интервалом в 15 минут (чтобы лягушки успокоились) ловим сачком по одной лягушке. Как только наберется двадцать лягушек, мы их меряем и вычисляем средний размер. Однако такое исследование не будет удовлетворять правилу независимости, потому что каждый отлов окажет влияние на последующее поведение лягушек (например, к концу лова будут попадаться самые смелые, или, наоборот, самые глупые). Еще хуже использовать в качестве повторностей последовательные наблюдения за объектом. Например, в некотором опыте выясняли скорость зрительной реакции, показывая человеку на доли секунды предмет, а затем спрашивая, что это было. Всего исследовали 10 человек, причем каждому показывали предмет пять раз. Авторы опыта посчитали, что у них было 50 повторностей, однако на самом деле — только десять. Это произошло потому, что каждый следующий показ не был независим от предыдущего (человек мог, например, научиться лучше распознавать предмет).

Надо быть осторожным не только с данными, собранными в последовательные промежутки времени, но и просто с данными, собранными с одного и того же места. Например, если мы определяем качество телевизоров, сходящих с конвейера, не годится в качестве выборки брать несколько штук подряд — с большой вероятностью они изготовлены в более близких условиях, чем телевизоры, взятые порознь, и, стало быть, их характеристики не независимы друг от друга.

Второй важный вопрос о повторностях: сколько надо собрать данных. Есть громадная литература по этому поводу, но ответа, в общем, два: (1) чем больше, тем лучше и (2) 30. Выглядящее несколько юмористически «правило 30» освящено десятилетиями опытной работы. Считается, что выборки, меньшие 30, следует называть малыми, а бóльшие — большими. Отсюда то значение, которое придают числу тридцать в анализе данных. Бывает так, что и тридцати собрать нельзя, однако огорчаться этому не стóит, поскольку многие процедуры анализа данных способны работать с очень малыми выборками, в том числе из пяти и даже из трех повторностей. Следует, однако, иметь в виду, что чем меньше повторностей, тем менее надежными будут выводы. Существуют, кроме того, специальные методы, которые позволяют посчитать, сколько надо собрать данных, для того чтобы с определенной вероятностью высказать некоторое утверждение. Это так называемые «тесты мощности» (см. пример в главе про одномерные данные).

Рандомизация — еще одно условие создания выборки, и также «с подвохом». 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. Предположим, нам поручено случайным образом отобрать сто деревьев в лесу, чтобы впоследствии померить степень накопления тяжелых металлов в листьях. Как мы будем выбирать деревья? Если просто ходить по лесу и собирать листья с разных деревьев, с большой вероятностью они не будут собранными случайно, потому что вольно или невольно мы будем собирать листья, чем-то привлекшие внимание (необычностью, окраской, доступностью). Этот метод, стало быть, не годится. Возьмем метод посложнее — для этого нужна карта леса с размеченными координатами. Мы выбираем случайным образом два числа, например 123 м к западу и 15 м к югу от точки, находящейся примерно посередине леса, затем высчитываем это расстояние на местности и выбираем дерево, которое ближе всего к нужному месту. Будет ли такое дерево выбрано случайно? Оказывается, нет. Ведь деревья разных пород растут неодинаково, поэтому у деревьев, растущих теснее (например, у елок), шанс быть выбранными окажется больше, чем у разреженно растущих дубов. Важным условием рандомизации, таким образом, является то, что *каждый объект должен иметь абсолютно те же самые шансы быть выбранным, что и все прочие объекты.*

Как же быть? Надо просто перенумеровать все деревья, а затем выбрать сто номеров по жребию. Но это только звучит просто, а попробуйте так сделать! А если надо сравнить 20 различных лесов?.. В общем, требование рандомизации часто оборачивается весьма серьезными затратами на проведение исследования. Естественно поэтому, что нередко

рандомизацию осуществляют лишь частично. Например, в нашем случае можно случайно выбрать направление, протянуть в этом направлении бечевку через весь лес, а затем посчитать, скольких деревьев касается бечевка, и выбрать каждое энное (пятое, пятнадцатое...) дерево, так чтобы всего в выборке оказалось 100 деревьев. Заметьте, что в данном случае метод рандомизации состоит в том, чтобы внести в исследуемую среду *такой порядок, которого там заведомо нет.* Конечно, у этого последнего метода есть недостатки, а какие — **попробуйте догадаться** сами (ответ см. в конце главы).

Теперь вы знаете достаточно, чтобы ответить на еще один вопрос. В одной лаборатории изучали эффективность действия ядохимикатов на жуков-долгоносиков (их еще называют «слоники»). Для этого химикат наносили на фильтровальную бумагу, а бумагу помещали в стеклянную чашку с крышкой (чашку Петри). Жуков выбирали из банки, в которой их разводили для опытов, очень простым способом: банку с жуками открывали, и первого выползшего на край жука пересаживали в чашку с ядохимикатом. Затем засекали, сколько пройдет времени от посадки жука в банку до его гибели. Потом брали другого жука и так повторяли 30 раз. Потом меняли ядохимикат и начинали опыт сначала. Но однажды один умный человек заметил, что в этом эксперименте самым сильным всегда оказывался тот химикат, который был взят для исследования первым. Как вы думаете, в чем тут дело? Какие нарушения принципов повторности и рандомизации были допущены? Как надо было поставить этот опыт? (См. ответ в конце главы).

Для рандомизации, конечно, существует предел. Если мы хотим выяснить возрастной состав посетителей какого-то магазина, не нужно во имя рандомизации опрашивать прохожих с улицы. Нужно четко представлять себе генеральную совокупность, с которой идет работа, и не выходить за ее границы. Помните пример с питанием животного? Если генеральная совокупность — это животные данного вида, содержащиеся *в зоопарках*, нет смысла добавлять к исследованию данные о питании этих животных *в домашних условиях.* Если же такие данные просто необходимо добавить (например, потому что данных из зоопарков очень мало), то тогда генеральная совокупность будет называться «множество животных данного вида, содержащихся *в неволе*».

Интересный вариант рандомизации используют, когда в эксперименте исследуются одновременно несколько взаимодействий. Например, мы хотим выяснить эффективность разных типов средств против обледенения тротуаров. Для этого логично выбрать (случайным образом) несколько разных (по возрасту застройки, плотности населения, расположению) участков города и внутри каждого участка случайным образом распределить разные типы этих средств. Потом можно, например, фиксировать (в баллах или как-нибудь еще) состояние тротуаров

каждый день после нанесения средства, можно также повторить опыт при разной погоде. Такой подход называется «блочный дизайн». Блоками здесь являются разные участки города, а повторность обеспечивается тем, что в каждом блоке повторяются одни и те же воздействия. При этом даже не обязательно повторять однотипные воздействия по нескольку раз внутри блоков, важно выбрать побольше отличающихся друг от друга блоков. Можно считать разными блоками и разные погодные условия, и тогда у нас получится «вложенный блочный дизайн»: в каждый погодный блок войдет несколько «городских» блоков, и уже внутри этих блоков будут повторены все возможные воздействия (типы средств).

В области рандомизации лежит еще одно коренное различие между наблюдением и экспериментом. Допустим, мы изучаем эффективность действия какого-то лекарства. Вместо того, чтобы подбирать две группы больных, использовать плацебо и т. п., можно просто порыться в архивах и подобрать соответствующие примеры (30 случаев применения лекарства и 30 случаев неприменения), а затем проанализировать разницу между группами (например, число смертей в первый год после окончания лечения). Однако такой подход сопряжен с опасностью того, что на наши выводы окажет влияние какой-то (или какие-то) неучтенный фактор, выяснить наличие которого из архивов невозможно. Мы просто не можем гарантировать, что соблюдали рандомизацию, анализируя архивные данные. К примеру, первая группа (случайно!) окажется состоящей почти целиком из пожилых людей, а вторая — из людей среднего возраста. Ясно, что это окажет воздействие на выводы. Поэтому в общем случае эксперимент всегда предпочтительней наблюдения.

1.4. Что ищут в данных

Прочитав предыдущие разделы, читатель, наверное, уже не раз задавался вопросом: «Если так все сложно, зачем он вообще, этот анализ данных? Неужели и *так* не видно, что в один магазин ходит больше народу, одно лекарство лучше другого и т. п.?» В общем, *так* бывает видно довольно часто, но обычно тогда, когда либо (1) данных и/или исследуемых факторов очень мало, либо (2) разница между ними очень резка. В этих случаях действительно запускать всю громоздкую машину анализа данных не сто́ит. Однако гораздо чаще встречаются случаи, когда названные выше условия не выполняются. Давно, например, доказано, что средний человек может одновременно удержать в памяти лишь 5–9 объектов. Стало быть, анализировать в уме данные, которые насчитывают больше 10 компонентов, уже нельзя. А значит, не обой-

тись без каких-нибудь, пусть и самых примитивных (типа вычисления процентов и средних величин), методов анализа данных.

Бывает и так, что внешне очевидные результаты не имеют под собой настоящего основания. Вот, например, одно из исследований насекомых-вредителей. Агрономы определяли, насколько сильно вредят кукурузе гусеницы кукурузного мотылька. Получились вполне приемлемые результаты: разница в урожае между пораженными и непораженными растениями почти вдвое. Казалось, что и обрабатывать ничего не надо — «и так все ясно». Однако нашелся вдумчивый исследователь, который заметил, что пораженные растения, различающиеся по степени поражения, не различаются по урожайности. Здесь, очевидно, что-то не так: если гусеницы вредят растению, то чем сильнее они вредят, тем меньше должен быть урожай. Стало быть, на какой-то стадии исследования произошла ошибка. Скорее всего, дело было так: для того чтобы измерять урожайность, среди здоровых растений отбирали самые здоровые (во всех смыслах этого слова), ну а среди больных старались подобрать самые хилые. Вот эта ошибка репрезентативности и привела к тому, что возникли такие «хорошие» результаты. Обратите внимание, что только анализ взаимосвязи «поражение—урожай» (на языке анализа данных он называется «регрессионный анализ», см. главу про двумерные данные) привел к выяснению истинной причины. А кукурузный мотылек, оказывается, почти и не вредит кукурузе...

Итак, анализ данных необходим всегда, когда результат неочевиден, и часто даже тогда, когда он *кажется очевидным*. Теперь разберемся, к каким последствиям может привести анализ, что он умеет.

1. Во-первых, анализ данных умеет давать общие характеристики для больших выборок. Эти характеристики могут отражать так называемую центральную тенденцию, то есть число (или ряд чисел), вокруг которых, как пули вокруг десятки в мишени, «разбросаны» данные. Всем известно, как считать среднее значение, но существует еще немало полезных характеристик «на ту же тему». Другая характеристика — это разброс, который отражает не *вокруг чего* «разбросаны» данные, а *насколько сильно* они разбросаны.

2. Во-вторых, можно проводить сравнения между разными выборками. Например, можно выяснить, в какой из групп больных инфарктом миокарда частота смертей в первый год после лечения выше — у тех, к кому применяли коронарное шунтирование, или у тех, к кому применяли только медикаментозные способы лечения. «На взгляд» этой разницы может и не быть, а если она и есть, то где гарантия того, что эти различия не вызваны случайными при-

чинами, не имеющими отношения к лечению? Скажем, заболел человек острым аппендицитом и умер после операции: к лечению инфаркта это может не иметь никакого отношения. Сравнение данных при помощи *статистических тестов* позволяет выяснить, насколько велика вероятность, что различия между группами вызваны случайными причинами. Заметьте, что гарантий анализ данных тоже не дает, зато позволяет оценить (численным образом) шансы. Анализ данных позволяет оценить и упомянутые выше общие характеристики.

3. Третий тип результата, который можно получить, анализируя данные,— это сведения о взаимосвязях. Изучение взаимосвязей — наверное, самый серьезный и самый развитый раздел анализа данных. Существует множество методик выяснения и, главное, проверки «качества» связей. В дальнейшем нам понадобятся сведения о том, какие бывают взаимосвязи. Есть так называемые *соответствия*, например когда два явления чаще встречаются вместе, нежели по отдельности (как гром и молния). Соответствия нетрудно найти, но силу их измерить трудно. Следующий тип взаимосвязей — это *корреляции*. Корреляции показывают *силу* взаимосвязи, но не могут определить ее *направления*. Другими словами, если выяснилась корреляция между качанием деревьев и ветром, то нельзя решить, дует ли ветер оттого, что деревья качаются, или наоборот. Наконец, есть *зависимости*, для которых можно измерить и силу, и направление, и оценить, насколько вероятно то, что они — результат случайных причин. Кстати говоря, последнее можно, как водится в анализе данных, сделать и для корреляций, и даже для соответствий. Еще одно свойство зависимостей состоит в том, что можно *предсказать*, как будет «вести» себя зависимая переменная в каких-нибудь до сих пор не опробованных условиях. Например, можно прогнозировать колебания спроса, устойчивость балок при землетрясении, интенсивность поступления больных и т. п.

4. И наконец, анализ данных можно использовать для установления *структуры*. Это самый сложный тип анализа, поскольку для выяснения структуры обычно используются *сразу несколько характеристик*. Есть и специальное название для такой работы — «многомерная статистика». Самое главное, на что способен многомерный анализ,— это создание и проверка качества *классификации* объектов. В умелых руках хорошая классификация очень полезна. Вот, например, мебельной фабрике потребовалось выяснить, какую мебель как лучше перевозить: в разобранном или в собранном виде. Рекомендации по перевозке зависят от уймы

причин (сложность сборки, хрупкость, стоимость, наличие стеклянных частей, наличие ящиков и полок и т. д.). Одновременно оценить эти факторы может лишь очень умелый человек. Однако существуют методы анализа, которые с легкостью разделят мебель на две группы, а заодно и проверят качество классификации, например ее соответствие сложившейся практике перевозок.

Существует и другой подход к результатам анализа данных. В нем все методы делятся на *предсказательные и описательные*. К первой группе методов относится все, что можно статистически оценить, то есть 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 находятся способы их вероятностной оценки, и поэтому первая группа все время увеличивается.

* * *

Ответ к задаче про случайный выбор деревьев в лесу. В этом случае шанс быть выбранными у елок выше, чем у дубов. Кроме того, лес может иметь какую-то структуру именно в выбранном направлении, и поэтому одной такой «диагонали» будет недостаточно для того, чтобы отобразить весь лес. Чтобы улучшить данный метод, надо провести несколько «диагоналей», а расстояния между выбираемыми деревьями по возможности увеличить.

Ответ к задаче про выбор жуков. Дело в том, что первыми вылезают самые активные особи, а чем активнее особь, тем быстрее она набирает на лапки смертельную дозу ядохимиката и, стало быть, быстрее гибнет. Это и было нарушением принципа рандомизации. Кроме того, нарушался принцип повторности: в чашку последовательно сажали жука за жуком, что не могло не повлиять на исход опыта. Для того чтобы поставить опыт правильно, надо было сначала подготовить (30 $\times$ количество ядохимикатов) чашек, столько же листочков с бумагой, *случайным образом* распределить ядохимикаты по чашкам, а затем перемешать жуков в банке, достать соответствующее количество и рассадить по чашкам.

Глава 2

Как обрабатывать данные

В принципе, обрабатывать данные можно и без компьютера. Так поступали в те годы, когда компьютерная техника была недоступна. Однако многие статистические расчеты настолько тяжеловесны, что уже в XIX веке стали придумывать способы их автоматизации.

2.1. Неспециализированные программы

Почти в любом компьютере с предустановленной системой есть программа-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 (по-английски «exact») подсчет многих статистик слишком сложен, порой даже для продвинутых компьютерных программ, поэтому используются оценочные («estimated») 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, в то время как обычно данные как раз «идут» сериями (колонками, векторами). Чтобы работать с сериями более эффективно (и не только для этого), были придуманы электронные таблицы. Объяснять, как они устроены, наверное, не нужно. Сила электронных таблиц прежде всего в том, что они помогают визуализировать данные.

Глядя в электронную таблицу, можно сразу понять, как выглядят данные в целом, и «на глазок» оценить их основные параметры. Это очень полезно. Кроме визуализации, программы электронных таблиц снабжены развитым инструментарием для ввода и преобразования данных — автодополнением, автокопированием, сортировкой и т. п. Однако

большинство таких программ имеет своеобразное «родимое пятно» — они создавались в основном для офисного применения и были изначально ориентированы на бухгалтерские задачи. Для обработки данных нужен гораздо более специализированный инструмент. Конечно, развитые программы электронных таблиц, такие как MS Excel, Gnumeric или OpenOffice.org Calc, имеют, среди прочего, набор статистических функций. Но поскольку это — не основной компонент, на статистику в этих программах традиционно обращается мало внимания. Набор статистических тестов невелик, многие методы, особенно многомерные, отсутствуют, реализация (то есть как именно идут внутри программы сами вычисления) часто далека от оптимальной, нет специализированной системы отчетов, много неудобных ограничений, возможны ошибки, которые будут исправляться не слишком быстро, опять-таки потому, что статистика — не первоочередная функция электронных таблиц.

Кроме того, базовый принцип электронных таблиц — визуализация данных — имеет и свои оборотные стороны. Что, если данные не помещаются в окне? В этом случае их надо будет прокручивать или скрывать часть ячеек. И то, и другое оказывает пользователю медвежью услугу, потому что он рассчитывал на облегчение восприятия данных, в то время как программа, скорее, затрудняет его. Или другой пример — надо провести операции с тремя несмежными столбцами. Сделать это через выделение нельзя, потому что выделение всегда одного типа (да и буфер обмена очень часто всего один), приходится делать много движений мышкой с большим риском ошибиться. И уж совсем никуда не годится графическая система, если надо сочетать методы обработки каким-нибудь сложным образом.

Выход — в пользовании специализированными статистическими программами.

2.2. Специализированные статистические программы

2.2.1. Оконно-кнопочные системы

Есть две группы специализированных статистических программ. Первые не особенно отличаются внешне от электронных таблиц, однако снабжены значительно большим арсеналом доступных статистических приемов. Кроме того, у них традиционно мощная графическая часть (возможных графиков больше, и управление ими более гибкое), а часто и подсистема подготовки отчетов. Многие такие программы имеют значительно меньше ограничений, чем электронные таблицы.

Очень распространена в России относящаяся к этой группе система STATISTICA. Как сказано выше, она отличается мощной графической частью, то есть имеет множество возможных вариантов графического вывода, которые при этом довольно гибко настраиваются, так что количество «стандартных» графиков можно смело увеличить в несколько раз. Другим, и очень серьезным, преимуществом STATISTICA является наличие переведенной на русский язык системы помощи, свободно доступной в Интернете. Эта система может служить заодно и руководством по статистике, поскольку там освещены и общие вопросы. Издано немало книг, посвященных STATISTICA. Надо, однако, заметить, что популярным этот пакет является в основном только в России. Весьма редко можно встретить ссылки на обработку данных этой системой в статьях из ведущих научных (в основном англоязычных) журналов, и это несмотря на то, что система делается в Америке. Поэтому легко можно представить проблемы с обменом данными. Кроме того, как и у всех визуальных систем, однажды проведенное исследование нелегко повторить, если, скажем, появились новые данные. 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. Алгоритмы вычисления в STATISTICA, естественно, закрыты, поэтому иногда приходится проводить целое исследование, чтобы выяснить, что на самом деле в данном случае делает программа. К тому же к системе в свое время было немало претензий по поводу «быстрых и грязных» алгоритмов работы, и есть подозрение, что ситуация не слишком изменилась.

Другой программой, популярной в свое время на российском рынке, является система STADIA. Написанная русскоязычными авторами, она отличается продуманным интерфейсом и очень хорошей системой помощи. К сожалению, это тоже закрытая программа. Немного похожа на STADIA программа PAST. Изначально она предназначалась для специализированной обработки данных в геологии, но затем функции значительно расширились, и в сейчас в PAST представлены практически все широко распространенные средства анализа данных. Графическая часть PAST небогата, но достаточна для базового исследования. Следует отметить, что, в отличие от двух предыдущих программ, PAST распространяется бесплатно.

SPSS и MiniTab широко используются за рубежом, однако в России эти системы не слишком распространены. Общий интерфейс их схож со STATISTICA, хотя имеется и множество своих особенностей, например в подсистемах генерации отчетов. Нужно упомянуть также StatGraphics, который был доступен в России еще со времен господства MS-DOS, а в настоящее время приобрел развитый графический интерфейс и стал похож на остальные программы этой группы.

2.2.2. Статистические среды

Эта группа программ использует в основном интерфейс командной строки. Пользователь вводит команды, система на них отвечает. Звучит это просто, однако сами эти программы — одни из самых сложных систем обработки. Вообще говоря, командный интерфейс имеет немало недостатков. Например, пользователь лишен возможности выбрать тип обработки из списка (меню), вместо этого он должен помнить, какие типы обработки доступны. Кроме того, ввод команд схож (а иногда и неотличим) от «настоящего» программирования, так что для работы с подобными системами нужны некоторые навыки программиста (или достаточно смелости, для того чтобы эти навыки приобрести по ходу дела). Зато пользователь получает *полный контроль над системой*: он может комбинировать любые типы анализа, записывать процедуры в скрипты, которые можно запустить в любое время, модифицировать вывод графиков, сохранять их в любые графические форматы, легко писать расширения для системы, а если она к тому же еще имеет открытый код, то и модифицировать саму систему (или, по крайней мере, легко выяснять, как именно работают вычислительные алгоритмы).

Одна из наиболее продвинутых систем этого плана — это 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. Написано множество книг, описывающих работу с SAS и некоторые ее алгоритмы. Вместе с тем система сохраняет множество рудиментов 70-х годов, и пользоваться ей поначалу не очень легко даже человеку, знакомому с командной строкой и программированием. А стоимость самой системы просто запредельная — многие тысячи долларов!

2.3. 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 C. 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 (версия для UNIX стоит порядка $6500). 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, например обращением с глобальными и локальными переменными, а также работой с памятью. Фактически они создали не аналог S-Plus, а новую «ветку» на «дереве S» (многие вещи, которые отличают R от S-Plus, связаны с влиянием языка Scheme). Проект вначале развивался довольно медленно, но когда в нем появилось достаточно возможностей, в том числе уникальная по легкости система написания дополнений (пакетов), все большее количество людей стало переходить на R с S-Plus. Когда же, наконец, были устранены свойственные первым версиям проблемы работы с памятью, на R стали переходить и «любители» других статистических пакетов (прежде всего тех, которые имеют интерфейс командной строки: SAS, Stata, SYSTAT). Количество книг, написанных про R, за последние годы выросло в несколько раз, а количество пакетов уже приближается к трем с половиной тысячам!

2.4. Применение, преимущества и недостатки R

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, более всего его применяют для статистического анализа — от вычисления средних величин до вейвлет-преобразований и временны́х рядов. Географически R распространен тоже очень широко. Трудно найти американский или западноевропейский университет, где бы не работали с R. Очень многие серьезные компании (скажем, Boeing) устанавливают R для работы.

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 современно-

го статистического анализа, который бы не был сейчас представлен в R. Свободный код — это не просто бесплатность программы (хотя в сравнении с коммерческими пакетами, продающимися за совершенно безумные деньги, это, конечно, преимущество, да еще какое!), но и возможность разобраться, как именно происходит анализ, а если в коде встретилась ошибка — самостоятельно исправить ее и сделать исправление доступным для всех.

У R есть и немало недостатков. 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. Если функция известна, то узнать, что она делает, очень легко, обычно достаточно набрать команду `help(`*название функции*`)`. Увидеть код функции тоже легко, для этого надо просто набрать ее название без скобок или (лучше) ввести команду `getAnywhere(`*название функции*`)`. А вот что делать, если «задали» провести, скажем, дисперсионный анализ, а функция неизвестна? (См. ответ в конце главы.)

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. Вот, например, простая задача: требуется превратить выборку, состоящую из цифр от 1 до 9, в таблицу из трех колонок (допустим, это были данные за три дня, и каждый день делалось три измерения). Чтобы сделать это в программе с визуальным интерфейсом, скажем в STATISTICA, требуется: (1) учредить две новые переменные, (2–3) скопировать дважды кусок выборки в буфер, (4–5) скопировать его в одну и другую переменную и (6) уничтожить лишние строки. В R это делается одной командой:

```
> b <- matrix(1:9, ncol=3)
```

Второй недостаток R — относительная медлительность. Некоторые функции, особенно использующие циклы, и виды объектов, особенно списки и таблицы данных, «работают» в десятки раз медленнее, чем их аналоги в коммерческих пакетах. Но этот недостаток преодолевается, хотя и медленно. Новые версии R «умеют» делать параллельные вычисления, создаются оптимизированные варианты подпрограмм, работающие много быстрее, память в R используется все эффективнее, а вместо циклов рекомендуется применять векторизованные вычисления (см. приложение о программировании в R).

Что же касается трудности обучения, то мы надеемся, что эта книга послужит одним из средств его преодоления.

2.5. Как скачать и установить R

Поскольку R — свободная система, то его можно скачать и установить без всякой оплаты. Есть несколько способов, которые зависят от того, какая у вас операционная система.

Linux. Если у вас какой-нибудь из распространенных дистрибутивов, то R наверняка входит в репозиторий «прилагающихся» к вашей системе пакетов. Единственное, что нужно учесть,— обновление пакетов часто отстает от выхода версий R (как правило, четыре раза в год).

Версия нумеруется тремя числами, первые два — это главная версия, обновляется два раза в год. С каждой главной версией в R вносятся изменения, часто довольно значительные. Как правило, это множество новых команд, исправленные алгоритмы выполнения старых, и разумеется, исправления ошибок. Кроме того, бывает так, что страдает обратная совместимость — новые версии начинают иначе истолковывать команды, а иногда команды или их опции перестают действовать. Естественно, разработчики стараются минимизировать такие изменения. По своему опыту можем сказать, что написанные на R программы пяти-семилетней давности работают без проблем. В общем, мораль такая: обновляйте R, но при этом всегда читайте список изменений версии. На каждую главную версию выходят, как правило, две «минорные» версии (нулевая и первая). Первая минорная версия обычно ничего нового, кроме исправления ошибок, не содержит.

Если вы хотите всегда иметь самую свежую версию, вам надо будет скачивать R из так называемого CRAN (Comprehensive R Archive Network). У этого сайта довольно много зеркал (копий), так что выбирайте подходящее. Скачать можно как исходный код для последующей компиляции, так и собранный бинарный пакет (последнее, естественно, не всегда возможно). Кстати говоря, исходный код R компилируется очень просто.

Mac OS X. Для того чтобы начать работать с R в Mac OS X, надо сначала убедиться, что у вас последняя версия этой операционной системы. Последние версии R выходят только под последние версии Mac OS. Установочный пакет скачивается с CRAN (не забудьте также скачать оттуда Tcl/Tk). Имейте в виду, что графическая оболочка R для Mac (R Mac GUI) обновляется быстрее, чем сам R, поэтому разработчики предусмотрели возможность скачивания и установки оболочки отдельно. Установка происходит практически так же, как установка любой программы под Mac. Надо еще

отметить, что R завоевал себе популярность не в последнюю очередь тем, что он запускался под Mac, в то время как S-Plus под эту платформу был недоступен.

Windows. Для того чтобы запускать R, можно иметь любые версии этой операционной системы, начиная с Windows 95. Как и в предыдущем случае, установочный пакет скачивается с CRAN. Опять-таки, установка напоминает обычную для Windows. Есть программа-установщик, которая задает несколько вопросов, от ответов на которые ничего серьезного не зависит. После инсталляции появляется ярлык, щелкнув на который, можно запускать R.

Здесь 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 не критичны. Единственное, что надо при этом иметь в виду,— рабочая папка должна быть открыта для записи, иначе вам некуда будет записывать результаты работы. Еще одна вещь, важная для всех операционных систем: R (в отличие от S-Plus) держит все свои вычисления в оперативной памяти, поэтому если в процессе работы, скажем, выключится питание, результаты сессии, не записанные явным образом в файл, пропадут. (Тут надо заметить, что существует пакет `SOAR`, который изменяет это поведение.)

2.6. Как начать работать в R

2.6.1. Запуск

Каждая операционная система имеет свои особенности работы. Но в целом можно сказать, что под все три упомянутые выше 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).

Терминальный способ прост: достаточно в командной строке терминала (пользователи Windows часто называют его «черным окошком» или «окном DOS») набрать

```
$ R
```

,—

и появится вводный экран системы. Под Windows это немного сложнее — необходимо вызывать программу `Rgui.exe`, причем для этого надо либо «находиться» в той же папке, где находится эта программа, либо путь к этой папке должен быть прописан в переменной `PATH`. Кроме того, для того чтобы в русскоязычной Windows вводный экран был

читаем, надо сменить в окне терминала кодировку (командой «`chcp 1251`») и установить соответствующий шрифт (скажем, Lucida Console). Если в UNIX терминал запущен без графической среды (X11), то все изображения будут «скидываться» в один многостраничный PDF-файл `Rplots.pdf`. Под Mac это будет происходить, даже если X11 запущен, так что полноценно использовать R под Mac можно только в GUI-варианте. Терминальный запуск под Windows таких ограничений не имеет.

В дальнейшем договоримся, что под «сессией R» мы будем иметь в виду терминальный запуск под X11 в Linux и GUI-запуск в Windows и Mac. GUI под эти операционные системы построены так, что они все равно запускают терминал-подобное окно — общение с R возможно только в режиме диалога, «команда — ответ». Полноценного GUI с R не поставляется, хотя существуют многочисленные попытки создать такую систему. Одна из лучших, на наш взгляд, — это R Commander, советуем также обратить внимание на RStudio; о них мы еще поговорим в приложении. Но, вообще-то, система из меню и окошек не способна заменить полноценного интерфейса командной строки, особенно в случае таких сложных систем, как R. Интересно, что S-Plus имеет очень приличный GUI, но если вы откроете любой учебник по этой системе, вы увидите, что авторы настоятельно рекомендуют пользоваться командной строкой.

Особенность R GUI под Windows — ее можно запустить в много- (MDI) и однооконном (SDI) режимах. Настоятельно рекомендуем вам второй, тем более что все остальные реализации R только его и «умеют».

Когда вы запустили R, первым делом появляется вводный экран. Если у вас русифицированная система, то вы увидите надпись по-русски. Если по какой-то причине вам надо видеть все по-английски, установите переменную `LANGUAGE` — создайте файл `Renviron.site`, в который внесите строчку

```
LANGUAGE=en
```

Этот файл должен находиться в так называемой «домашней» директории R. Более подробно про это можно узнать, если прочитать помощь по команде `Startup()`. Под Linux вызвать R таким способом еще проще — надо внести в строку вызова опцию (ту же самую строчку).

2.6.2. Первые шаги

Перед тем как начать работать, надо понять, как выйти. Для этого надо ввести одну команду, нажать `Enter` и ответить на один вопрос:

```
q()
```

```
Save workspace image? [y/n/c]: n
```

Уже такой простой пример демонстрирует, что в R любая команда имеет аргумент в круглых скобках. Если аргумент не указан, скобки все равно нужны. Если вы забудете это сделать, то вместо выхода из R получите определение функции:

```
> q
function (save = "default", status = 0, runLast = TRUE)
.Internal(quit(save, status, runLast))
<environment: namespace:base>
```

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 либо (если вы работаете в Linux) текст помощи в основном окне программы (выход из этого режима — по клавише «q»). Если внимательно прочитать текст, становится ясно, что выйти из R можно, и не отвечая на вопрос, если ввести `q("no")`.

Зачем же нужен этот вопрос? Или, другими словами, что будет, если ответить положительно? В этом случае в рабочую папку R (ту, из которой он вызван) запишутся два файла: бинарный `.RData` и текстовый `.Rhistory`. Первый содержит все объекты, созданные вами за время сессии. Второй — полную историю введенных команд.

Когда вы работаете в R, предыдущую команду легко вызвать, нажав клавишу-стрелку «вверх». Но если вы сохраните файл `.Rhistory`, ваши команды будут доступны и в следующей сессии — при условии, что вы вызовете R из той же самой папки. И наоборот, если вы случайно сохраните рабочую среду (эти два файла), то при следующем старте они загрузятся автоматически. Иногда такое поведение R становится причиной различных недоумений, так что будьте внимательны!

2.7. R и работа с данными: вид снаружи

2.7.1. 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()`, которая не меняет объект, а выдает результат «наружу». Если же у вас уже есть таблица данных, можно использовать команды `fix()` или `edit()`, которые в данном случае вызовут тот же Excel-подобный редактор.

Функция `edit()`, вызванная для другого типа объекта, запустит его редактирование в текстовом редакторе (под Windows это обычно Notepad или Блокнот), и вы сможете отредактировать объект там. Если вы хотите поменять редактор, напишите, например:

```
options(editor="c:\\Program Files\\CrazyPad\\crazypad.exe")
```

Однако лучше всего научиться загружать в R файлы, созданные при помощи других программ, скажем, при помощи Excel. Имейте в виду, что такие данные имеют очень разный формат, и написать по этому вопросу сколько-нибудь компактное руководство трудно. Но мы все же попробуем.

В целом данные, которые надо обрабатывать, бывают двух типов — текстовые и бинарные. Не вдаваясь в детали, примем, что текстовые — это такие, которые можно прочитать и отредактировать в любом текстовом редакторе («Блокнот», Notepad, TextEdit, Vi). Для того чтобы отредактировать бинарные данные, нужна, как правило, программа, которая эти данные когда-то вывела. 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 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:/programs/R/R-2.14.1"
```

Допустим, что это вовсе не та папка, которая вам нужна. Поменять рабочую папку можно командой:

```
setwd("e:\\wrk\\temp")
> getwd()
[1] "e:/wrk/temp"
```

Обратите внимание на работу с обратными слэшами под Windows: вместо одного слэша надо указывать два, только тогда R их поймет. Под Linux и Mac OS X все проще: там по умолчанию используются прямые слэши «/», с которыми проблем никаких нет. Кстати, под Windows тоже можно использовать прямые слэши, например:

```
setwd("e:/wrk/temp")
> getwd()
[1] "e:/wrk/temp"
```

Дальше надо проверить, есть ли в нужной поддиректории нужный файл:

```
> dir("data")
[1] "mydata.txt"
```

Теперь можно, наконец, загружать данные. (Предполагается, что в текущей директории у вас есть папка `data`, а в ней — файл `mydata.txt`. Если это не так, то для того, чтобы запустить этот пример, нужно скачать файл с упомянутого в предисловии сайта, создать папку `data` и поместить туда файл). Данные загружает команда `read.table()`:

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

Это ровно то, что надо, за тем исключением, что перед нами «рояль в кустах» — авторы заранее знали структуру данных, а именно то, что у столбцов есть имена (`head=TRUE`), а разделителем является точка с запятой (`sep=";"`). Функция `read.table()` очень хороша, но не настолько умна, чтобы определять формат данных «на лету». Поэтому вам придется выяснить нужные сведения заранее, скажем, в том же самом текстовом редакторе. Есть и способ выяснить это через R, для этого используется команда `file.show("data/mydata.txt")`. Она выводит свои данные таким же образом, как и `help()`,— отдельным окном или в отдельном режиме основного окна. Вот что вы должны увидеть:

```
a;b;c
1;2;3
4;5;6
7;8;9
```

Вернемся к предыдущему примеру. В R многие команды, в том числе и `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, поэтому если в ваших данных вместо точек с запятыми — пробелы, можно аргумент `sep` не указывать. Естественно, бывает безумное множество различных частных случаев, и как бы мы ни старались, все не описать. Отметим, однако, еще несколько важных вещей:

1. Русский текст в файлах обычно читается без проблем. Если все же возникли проблемы, то лучше перевести все в кодировку UTF-8 (при помощи любой доступной утилиты перекодирования, скажем `iconv`) и добавить соответствующую опцию:

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

2. Иногда нужно, чтобы R прочитал, кроме имен столбцов, еще и имена строк. Для этого используется такой прием:

   ```
   > read.table("data/mydata2.txt", sep=";", head=TRUE)
         a b c
   one   1 2 3
   two   4 5 6
   three 7 8 9
   ```

   В файле `mydata2.txt` в первой строке было три колонки, а в остальных строках — по четыре. **Проверьте** это при помощи `file.show()`.

3. 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. Сокращать можно, но с осторожностью, поэтому дальше в тексте мы всегда будем писать `TRUE` или `FALSE`, хотя на практике многие используют просто `T` или `F`.

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

1) данные набирают в какой-нибудь «внешней» программе;
2) записывают их в текстовый файл с разделителями (см. выше);
3) загружают как объект в R и работают с этим объектом, возможно, внося изменения;
4) если были изменения, командой `write.table()` (про нее см. следующий подраздел) записывают обратно в файл;
5) импортируют как текстовый файл в исходную программу и работают дальше.

Такой способ выглядит сложновато, но 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, часто сводится к совету по образцу известного анекдота — «выключим газ, выльем воду и вернемся к условию предыдущей задачи». То есть надо найти способ, как преобразовать (с минимальными потерями значимой информации, естественно) бинарные данные в текстовые таблицы. Проблем на этом пути возникает обычно не слишком много.

Второй путь — найти способ прочитать данные в R без преобразования. В R есть пакет `foreign`, который может читать бинарные данные, выводимые пакетами MiniTab, S, SAS, SPSS, Stata, Systat, а также формат `DBF`. Чтобы узнать про это подробнее, надо загрузить пакет (командой `library(foreign)`) и вызвать общую справку по его командам (например, командой `help(package=foreign)` или просто посмотреть справку по пакету через браузер, который откроется по команде `help.start()`).

Что касается других распространенных форматов, скажем, форматов MS Excel, здесь дело хуже. Есть не меньше пяти разных способов, как загружать в R эти файлы, но все они имеют ограничения. Изо всех способов нам наиболее привлекательным представляется обмен с R через буфер. Если у вас открыт Excel, то можно скопировать в буфер любое количество ячеек, а потом загрузить их в R командой `read.table("clipboard")`.

Это просто и, главное, работает с любой Excel-подобной программой, в том числе с «новым» Excel, Gnumeric и OpenOffice.org / LibreOffice Calc.

Добавим еще несколько деталей:

1. R может загружать изображения. Для этого есть сразу несколько пакетов, наиболее разработанный из них — `pixmap`. R может также загружать карты в формате ArcInfo и др. (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()` (она работает на всех системах).

2. 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"
```

(Здесь есть несколько доселе не описанных команд. Для сохранения и загрузки бинарных файлов служат команды `save()` и `load()`, для удаления объекта — команда `rm()`. Для того чтобы показать вам, что объект удален, мы использовали команду-проверку `exists()`. Все, что написано на строчке после символа «#»,— это комментарий. Комментарии R пропускает, не читая).

3. 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`). Рассматривать в подробностях мы их здесь не будем.

4. Наконец, R может записывать таблицы и другие результаты обработки данных и, разумеется, графики. Об этом мы поговорим ниже.

2.7.2. 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()` (именно так, без аргументов).

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. (Мы задали параметр `split=TRUE`, для того чтобы вывести данные еще и на экран.) Сама команда записана не будет, а если вы хотите, чтобы она была записана, придется писать что-то вроде:

```
> print("2+2")
[1] "2+2"
> 2+2
[1] 4
```

Другими словами, придется повторять каждую команду два раза. Для сохранения истории команд служит, как мы уже писали выше, команда `savehistory()`, а для сохранения всех созданных объектов — `save.image()`. Последняя может оказаться полезной для сохранения промежуточных результатов работы, если вы не уверены в стабильности работы компьютера.

2.7.3. R как калькулятор

Самый простой способ использования R — это арифметические вычисления. Например, выражение «`log(((sqrt(sum(c(2,2))))^2)*2.5)`» вычисляется так:

1. Из двух двоек создается вектор: `c(2,2)`.
2. Подсчитывается сумма его членов: `2+2=4`.
3. Извлекается квадратный корень: `sqrt(4)=2`.
4. Он возводится в квадрат: `2^2=4`.
5. Результат умножается на 2.5: `4*2.5=10`.
6. Вычисляется десятичный логарифм: `log(10)=1`.

Как видите, круглые скобки можно вкладывать друг в друга. R раскрывает их, вычисляя значение «изнутри наружу», а если скобки отсутствуют, то следует правилам «старшинства» арифметических операций, похожим на те, которым нас учили в школе. Например, в выражении `2+3*5` сначала будет вычислено произведение (`3*5=15`), а потом сумма (`2+15=17`). **Проверьте** это в R самостоятельно.

2.7.4. Графики

Одним из основных достоинств статистического пакета служит разнообразие типов графиков, которые он может построить. R в этом смысле — один из рекордсменов. В базовом наборе есть несколько десятков типов графиков, еще больше в рекомендуемом пакете `lattice`, и еще больше — в пакетах с CRAN, из которых не меньше половины строят как минимум один оригинальный график (а половина — это больше полутора тысяч!). Таким образом, по прикидкам получается, что разнообразных типов графики в R никак не меньше тысячи. При этом они все еще достаточно хорошо настраиваются, то есть пользователь легко может разнообразить эту исходную тысячу на свой вкус. Здесь мы постараемся, однако, не описывать разнообразие R-графики, а остановимся на нескольких фундаментальных принципах, понимание которых должно существенно облегчить новичку построение графиков в R.

Рассмотрим такой пример (рис. 1):

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

Тут много такого, о чем речи пока не шло. Самое главное — то, что первая команда рисует график «с нуля», тогда как вторая — добавляет к уже нарисованному графику детали. Это и есть два типа графических команд, используемых в базовом графическом наборе R. Теперь немного подробнее. `plot()` — основная графическая команда, причем команда «умная» (а правильнее сказать — `generic`, или «общая»). Это значит, что 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* (подробнее о векторах см. ниже), а для «одиночного» вектора предусмотрен график, где по оси абсцисс — индексы (то есть номера каждого элемента вектора по порядку), а по оси ординат — сами эти элементы. Если в аргументе команды будет что-то другое, будет, скорее всего, построен иной график. Вот пример (рис. 2):

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

Здесь тоже есть команды обоих типов, но немного иначе оформленные. Как видите, не беда, что мы забыли дать заголовок в команде `plot()`, его всегда можно добавить потом, командой `title()`. `cars` — это встроенная в R таблица данных, которая использована здесь по прямому назначению, для демонстрации возможностей программы. Прочитать, что такое `cars`, можно, вызвав справку обычным образом (`?cars`).

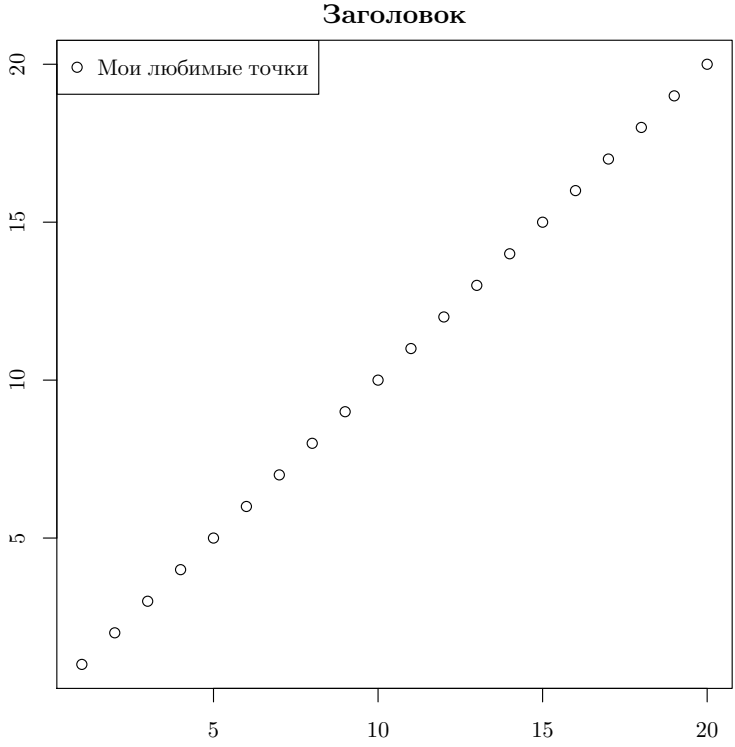


Рис. 1. Пример графика с заголовком и легендой

Для нас сейчас важно, что это — не вектор, а таблица из двух колонок — `speed` и `distance` (скорость и тормозная дистанция). Функция `plot()` автоматически нарисовала коррелограмму (scatterplot), где по оси `x` откладывается значение одной переменной (колонки), а по оси `y` — другой, и еще присвоила осям имена этих колонок. Любопытным советуем **проверить**, что нарисует `plot()`, если ему «подложить» таблицу с тремя колонками, скажем встроенную таблицу `trees`.

Как мы уже писали, очень многие пакеты расширяют графические функции R. Вот, например, что будет, если мы применим к тем же данным (`1:20`) функцию `qplot()` из пакета `ggplot2` (рис. 3):

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

(Мы уже писали выше, что команда `library()` загружает нужный пакет. Но пакета `ggplot2` в вашей системе могло не быть! Если R не смог загрузить пакет, то нужно его скачать из Интернета и установить. Делается это через меню или командой `install.packages("ggplot2")`. Дан-

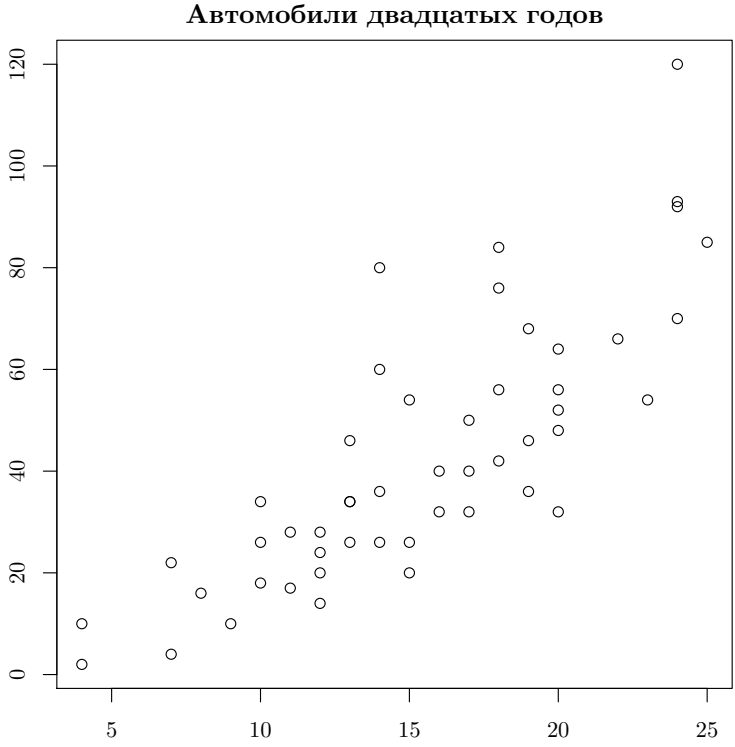


Рис. 2. Пример графика, отражающего встроенные данные `cars`

ные `1:20` мы повторили дважды потому, что `qplot()` работает немного иначе, чем `plot()`).

Получилось почти то же самое, но оформление выглядит сложнее. Пакет `ggplot2` и создавался для того, чтобы разнообразить слишком аскетичное, по мнению автора (это Hadley Wickham), оформление графиков в R по умолчанию.

2.7.5. Графические устройства

Это второй очень важный момент для понимания устройства графики в R. Когда вы вводите команду `plot()`, 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. Если вы дадите команду `dev.off()`, то R закроет графическое окно (что, впрочем, можно сделать, просто щелкнув по кнопке закрытия окна). Экранных устройств в R предусмотрено несколько, в каждой операционной системе — свое

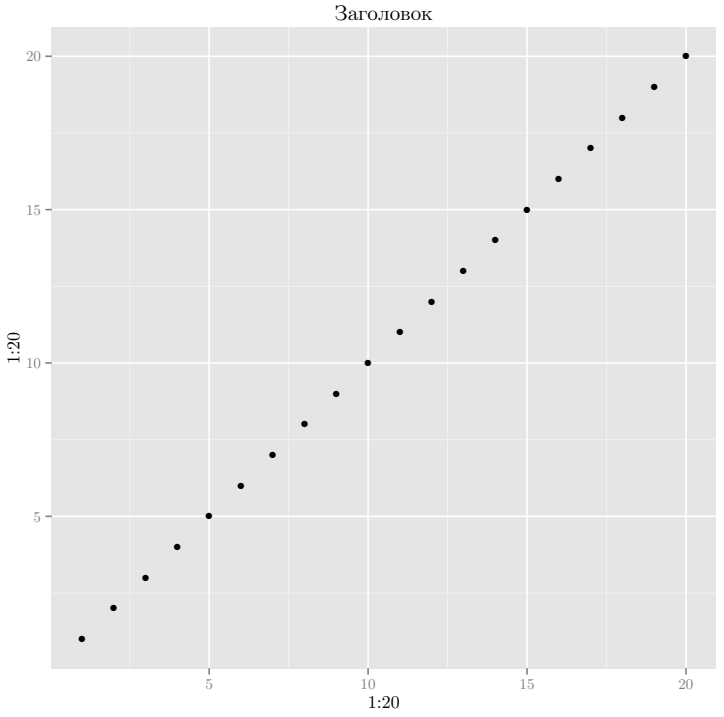


Рис. 3. Пример графика с заголовком, полученного при помощи команды `qplot()` из пакета `ggplot2`

(а в Mac OS X даже два). Но все это не так важно, пока вы не захотите строить графики и сохранять их в файлы автоматически. Тогда вам 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 (удобно, например, для веб-дизайна). Такого параметра у экранных устройств нет. Как только вводится команда `dev.off()`, устройство закрывается, и на диске появляется файл `1-20.png` (на самом деле файл появляется еще при открытии устройства, но сначала в нем ничего не записано). `png()` — одно из самых распространенных устройств при записи

файлов. Недостатком его является, разумеется, растровость. R поддерживает и векторные форматы, например PDF. Здесь, однако, могут возникнуть специфические для русскоязычного пользователя трудности со шрифтами. Остановимся на этом чуть подробнее. 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 того, если вы работаете под Windows, то R должен «знать» путь к программе `gswin32c.exe` (другими словами, нужно, чтобы путь к этой программе был записан в системную переменную `PATH`), иначе встраивание шрифтов не сработает. К счастью, если Ghostscript был установлен нормально, то проблем возникнуть не должно.

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

2.7.6. Графические опции

Как уже говорилось, графика в R настраивается в очень широких пределах. 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 (рис. 4):

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

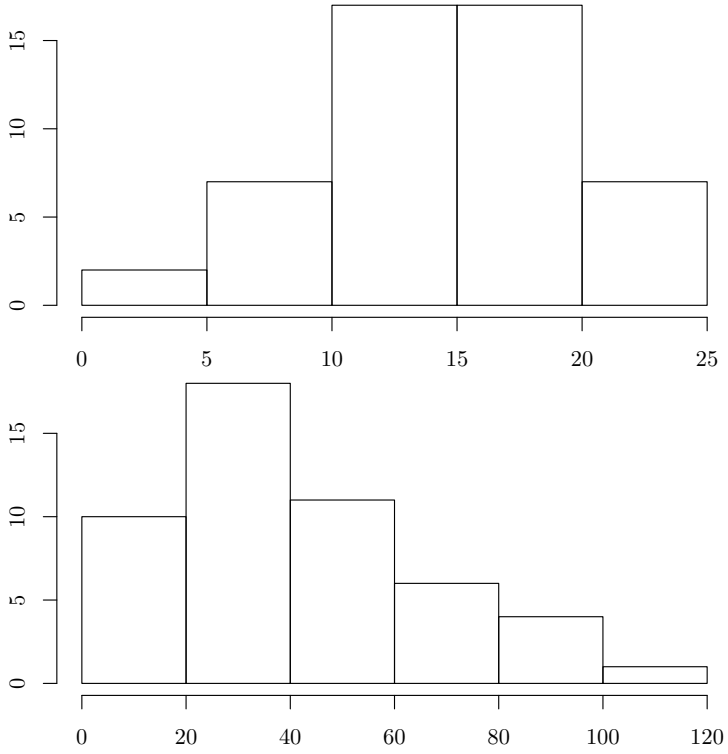


Рис. 4. Две гистограммы на одном графике

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. Чтобы не печатать каждый раз команду `par()` со всеми ее аргументами, мы «запомнили» старое значение в объекте `old.par`, а в конце вернули состояние к запомненному. Команда `hist()` строит график-гистограмму (подробнее о ней рассказано в главе про одномерные данные).

2.7.7. Интерактивная графика

Интерактивная графика позволяет выяснить, где именно на графике расположены нужные вам 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. Еще несколько лет назад мы бы написали,

что здесь вам вместо R следует воспользоваться другими аналитическими инструментами, но R развивается так быстро, что все эти методы теперь доступны, и даже в нескольких вариантах. Приведем лишь один пример. Вот так можно добавлять подписи в указанную мышкой область графика (пока вы еще не начали проверять — 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.

* * *

Ответ на вопрос про то, как найти функцию R, зная только то, что она должна делать. Чтобы, не выходя из R, узнать, как сделать, скажем, дисперсионный анализ, есть несколько способов. Первый — это команда «??» (два вопросительных знака):

```
> ??anova

Help files with alias or concept or title matching ‘anova’
using fuzzy matching:

...
stats::anova            Anova Tables
stats::anova.glm        Analysis of Deviance for Generalized
                        Linear Model Fits
stats::anova.lm         ANOVA for Linear Model Fits
...
stats::stat.anova       GLM Anova Statistics
...

Type ’?PKG::FOO’ to inspect entries ’PKG::FOO’, or
’TYPE?PKG::FOO’ for entries like ’PKG::FOO-TYPE’.
```

Надо только знать, что дисперсионный анализ по-английски называется «Analysis of Variance», или «ANOVA». Похожего результата можно добиться, если сначала запустить интерактивную помощь

(`help.start()`), а там перейти на поиск и задать в поле поиска то же самое слово `anova`.

Ну а если это не помогло, надо искать в Интернете. Сделать это можно прямо изнутри R:

```
 > RSiteSearch("anova")

A search query has been submitted to
                    http://search.r-project.org
The results page should open in your browser shortly
```

В браузере должно открыться окно с результатами запроса.

Глава 3

Типы данных

Чтобы обрабатывать данные, надо не просто их получить. Надо еще перевести их на язык цифр, ведь математика, на которой основан анализ данных, оперирует по большей части именно с числами. Сделать это можно самыми разными способами, иногда — удачно, иногда — с натяжками.

Со времен Галилея, который говорил, что «следует измерять то, что измеримо, и делать измеримым то, что таковым не является», европейская наука накопила громадный опыт в переводе окружающих явлений в измерения. Возникла и специальная наука об измерении — метрология. Ко многим достижениям метрологии мы уже настолько привыкли, что совершенно их не замечаем. Если, например, ставится опыт с расширением металлического бруска при нагревании, само собой подразумевается, что для этого нам нужны термометр и линейка. Эти два прибора (да, именно прибора!) как раз и являются устройствами, переводящими температуру и расстояние на язык цифр.

3.1. Градусы, часы и километры: интервальные данные

Очень важно, что температура и расстояние изменяются 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. Поэтому такие данные и называются `интервальными`. Интервальные данные чаще всего сравнивают с хорошо известной нам из курса математики числовой прямой, на которой расположены так называемые действительные (или, как их еще называют, вещественные) числа. Можно еще вспомнить о рациональных числах — то есть таких числах, которые можно представить в виде дроби. И те, и другие очень близки по своей сути к интервальным данным.

Не всегда, однако, интервальные данные изменяются плавно и непрерывно, от (как говорят математики) плюс бесконечности к минус бесконечности. Пример перед глазами: температура соответствует не прямой, а лучу, потому что со стороны отрицательной (слева) она ограничена абсолютным нулем, ниже которого температуры просто не бывает. Но на остальном протяжении этого луча показатели температуры можно уподобить действительным числам. Еще интереснее измерять углы. Угол изменяется непрерывно, но вот после 359° следует 0° — вместо прямой имеем отрезок без отрицательных значений. Есть даже особый раздел статистики, так называемая *круговая статистика* (directional, or circular statistics), которая работает с углами.

А вот другая ситуация. Допустим, мы считаем посетителей магазина. Понятно, что если в один день в магазин зашло 947 человек, а в другой — 832, то очень легко представить промежуточные значения. К тому же очевидно, что в первый день в магазине было больше народа. Однако если взять два «соседних» числа, например 832 и 831, то промежуточное значение представить нельзя, потому что люди на части не делятся. Получается, что такие данные соответствуют не действительным, а скорее натуральным числам. У этих чисел тоже есть отношение порядка, но вот промежуточное значение есть не всегда. И отрицательных значений у них нет. Перед нами — другой тип интервальных данных, не непрерывный, а *дискретный*.

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:

1) *распределение данных* близко к *нормальному* (незнакомые термины можно посмотреть в словаре);

2) выборка — большая (обычно не менее 30 наблюдений);

3) данные — интервальные непрерывные.

Если *хотя бы одно* из этих условий не выполняется, данные считаются непараметрическими и обрабатываются *непараметрическими методами*. Несомненным достоинством непараметрических методов является, как ни банально это звучит, их способность работать с непараметрическими (то есть «неидеальными») данными. Зато параметрические методы имеют бóльшую мощность (то есть при прочих равных вероятность не заметить существующую закономерность выше). 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 — об этом написано ниже). 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)
```

`x` — это имя объекта R, «`<-`» — функция присвоения, `c()` — функция создания вектора (от англ. *concatenate*, собрать). Собственно, R и работает в основном с объектами и функциями. У объекта может быть 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 так называемых 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()`», с которыми мы поработаем ниже. Называть объекты можно в принципе как угодно, но лучше придерживаться некоторых правил:

1. Использовать для названий только латинские буквы, цифры и точку (имена объектов не должны начинаться с точки или цифры).

2. Помнить, что R чувствителен к регистру, `X` и `x` — это разные имена.

3. 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()`.

3.2. «Садись, двойка»: 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 («ordered factor» — см. ниже). Если же стоит задача создать шкальные дан-

ные 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. Однако иногда сбор непрерывных данных требует использования труднодоступного оборудования и сложных методик (например, если вы решите исследовать окраску цветков как непрерывную переменную, вам понадобится спектрофотометр для измерения длины волны отраженного света — количественного выражения видимого цвета). В этом случае можно выйти из положения путем последующего перекодирования данных на стадии их обработки. Например, цвет можно закодировать в значениях красного, зеленого и синего каналов компьютерной цветовой шкалы RGB.

Вот еще один пример перекодирования. Предположим, вы изучаете высоту зданий в различных городах земного шара. Можно в графе «город» написать его название (номинальные данные). 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.

3.3. Красный, желтый, зеленый: 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, например логистическую регрессию (см. главу о двумерных данных) или бинарные коэффициенты сходства (см. главу о многомерных данных).

Для обозначения номинальных данных в R есть несколько способов, разной степени «правильности». Во-первых, можно создать текстовый (character) 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" ...
```

Обратите внимание на функцию `str()`! Это очень важная функция, мы бы рекомендовали выучить ее одной из первых. На первых порах пользователь R не всегда понимает, с каким типом объекта (вектором, таблицей, списком и т. п.) он имеет дело. Разрешить сомнения помогает `str()`.

Предположим теперь, что `sex` — это описание пола сотрудников небольшой организации. Вот как R выводит содержимое этого вектора:

```
> sex
```

```
[1] "male"   "female" "male"   "male"   "female" "male"   "male"
```

Кстати, пора раскрыть загадку единицы в квадратных скобках — это просто номер элемента вектора. Вот как его можно использовать (да-да, квадратные скобки — это тоже команда, можно это **проверить**, набрав помощь `?"["`):

```
> sex[1]
[1] "male"
```

«Умные», то есть объект-ориентированные, команды R кое-что понимают про объект `sex`, например команда `table()`:

```
> table(sex)
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 (рис. 5):

```
> 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
```

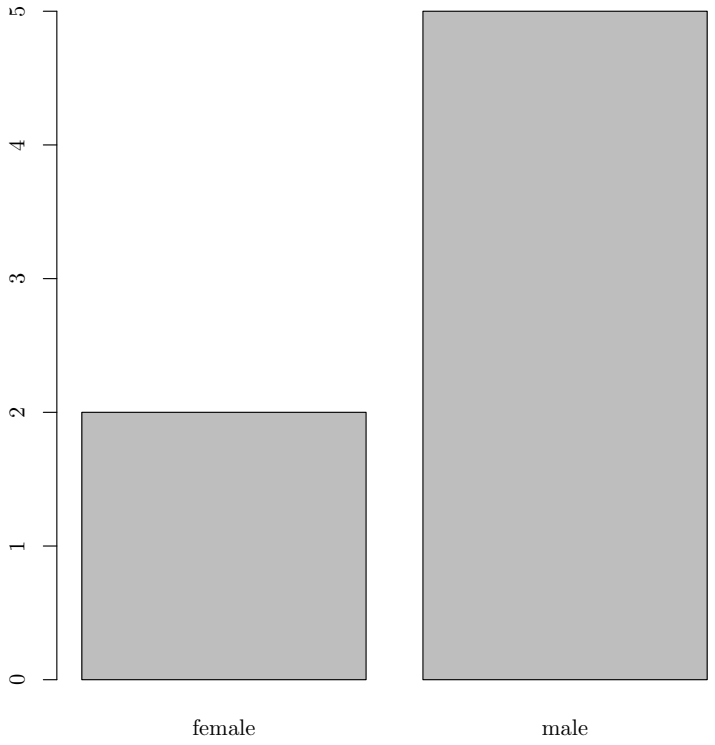


Рис. 5. Вот так команда `plot()` рисует фактор

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. Еще несколько свойств факторов надо знать заранее. Во-первых, подмножество фактора — это фактор с тем же количеством уровней, даже если их в подмножестве не осталось:

```
> sex.f[5:6]
[1] female male
Levels: female male
> sex.f[6:7]
[1] male male
Levels: female male
```

«Избавиться» от лишнего уровня можно, применив специальный аргумент или выполнив преобразование данных «туда и обратно»:

```
> sex.f[6:7, drop=TRUE]
```

```
[1] male male
Levels: male
> factor(as.character(sex.f[6:7]))
[1] male male
Levels: male
```

Во-вторых, 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:

```
> w <- c(69, 68, 93, 87, 59, 82, 72)
```

И мы хотим построить такой график, на котором были бы видны одновременно рост, вес и пол. Вот как это можно сделать (рис. 6):

```
> 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))
```

Тут, разумеется, нужно кое-что объяснить. `pch` и `col` — эти параметры предназначены для определения соответственно 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. Еще надо отметить, что изображение пола при помощи значка и цвета избыточно, для «нормального» графика хватит и одного из этих способов.

В-третьих, 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 видим, 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:

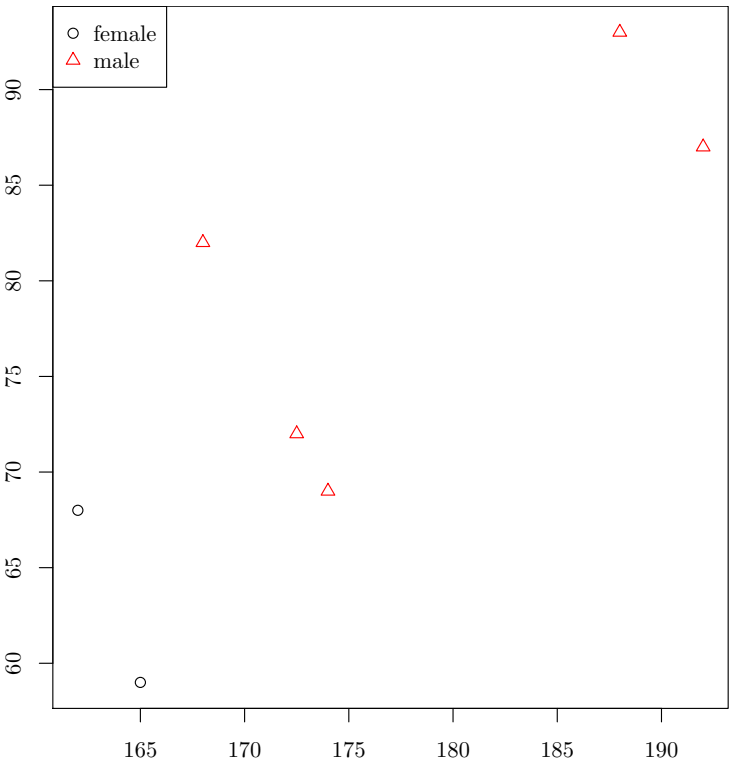


Рис. 6. График, показывающий одновременно три переменные

```
> 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 нечисло, 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)`.

3.4. 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 из них распространенные — это, наверное, графики-пироги и столбчатые диаграммы. Почти любая компьютерная программа, имеющая дело с таблицами данных, умеет строить такие графики. Однако надо заметить, что столбчатые диаграммы и в особенности «пироги» — неудачный способ представления информации. Многочисленные эксперименты доказали, что читаются такие графики гораздо хуже остальных. Самое печальное, что главная задача графиков — показать, где цифры различаются, а где сходны, практически не выполняется. В экспериментах людям предлагали несколько минут смотреть на такие графики, а затем просили расположить группы, отраженные на графике, в порядке значений подсчитанного признака. Оказалось, что это нелегко сделать. Вот пример. На рисунке 7 — столбчатый график результатов гадания на ромашках:

```
> romashka.t <- read.table("data/romashka.txt", sep="\t")
> romashka <- romashka.t$V2
> names(romashka) <- romashka.t$V1
> oldpar <- par(mar = c(7, 4, 4, 2) + 0.1)
```

```
> romashka.plot <- barplot(romashka, names.arg="")
> text(romashka.plot, par("usr")[3]-0.25, srt=45, adj=1,
+ xpd=TRUE, labels=names(romashka))
> par(oldpar)
```

(Пришлось исхитриться, чтобы поместить длинные надписи под столбиками.)

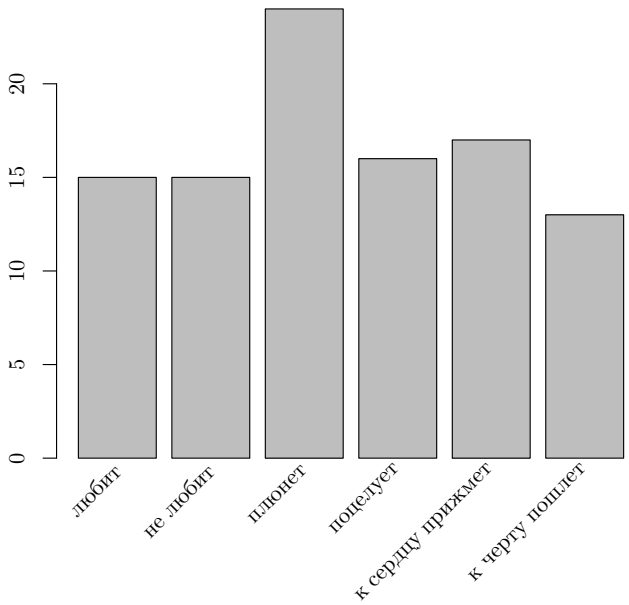


Рис. 7. Результаты гадания на ромашках (проценты исходов), показанные с помощью столбчатой диаграммы

Попробуйте проделать тот же эксперимент: посмотреть на график 3–5 минут, а затем закрыть книгу и расположить возможные исходы гадания в порядке убывания, от самого большого значения — к самому маленькому. (Ответ **проверьте** в конце главы.)

Создатели R прекрасно знали об этих проблемах, и поэтому в первых версиях вообще не было команды для рисования «пирогов». На замену им, а также на замену менее сомнительным столбчатым диаграммам в R есть так называемые точечные графики (dotplots). Ниже приведен пример такого графика для тех же самых данных по гаданию (рис. 8):

```
> dotchart(romashka)
```

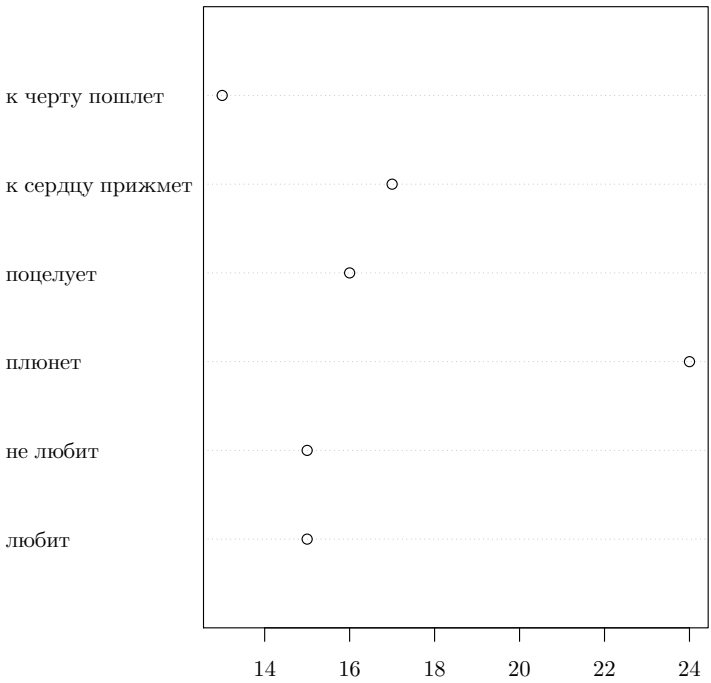


Рис. 8. Результаты гадания на ромашках (проценты исходов), показанные с помощью точечного графика

Надеемся, что большинство читателей согласятся с нами в том, что точечные диаграммы читаются легче «пирогов». Именно поэтому профессионалы рекомендуют точечные графики и боксплоты, а не «пироги» и столбчатые диаграммы.

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 (см. ниже) 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. (О самом тесте речь пойдет в следующей главе.)

3.5. 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 возник-

ло 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 мы обнаружили нечто, логически не совместимое с тем признаком, который надо фиксировать. Скажем, мы меряем длины клюва у обитателей скворечников, и тут нам попалась белка. У нее нет клюва, значит, и длины его тоже нет. Про эти «философские тонкости» хорошо помнить, хотя большинство компьютерных программ, занимающихся анализом данных, не различают этих вариантов.

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. Теперь о том, как надо работать с полученным вектором h. Если мы просто попробуем посчитать среднее значение (функция mean()), то получим:

```
> mean(h)
[1] NA
```

И это «идеологически правильно», поскольку функция может поразному обрабатывать NA, и по умолчанию она просто сигнализирует о том, что с данными что-то не так. 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
> h.old
[1]  8 10 NA NA  8 NA  8
```

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, предоставляющий графический интерфейс для «борьбы» с пропущенными данными, `MissingDataGUI`.

3.6. 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. К сожалению, 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, о которых мы еще поговорим ниже.

3.7. Меняем данные: основные принципы преобразования

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 (как на рис. 12) 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, 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, может дать нормальное распределение. Линеаризует зависимости и выравнивает дисперсии.
- Логит: `log(p/(1-p))`. Чаще всего применяется к пропорциям. Линеаризует так называемую сигмовидную кривую. Кроме логит-преобразования, 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 <- 1:10
> b <- seq(100, 1000, 100)
> d <- data.frame(a, b)
> d
    a    b
1   1  100
2   2  200
3   3  300
4   4  400
5   5  500
6   6  600
7   7  700
8   8  800
9   9  900
10 10 1000
> scale(d)
                a          b
 [1,] -1.4863011 -1.4863011
 [2,] -1.1560120 -1.1560120
 [3,] -0.8257228 -0.8257228
 [4,] -0.4954337 -0.4954337
 [5,] -0.1651446 -0.1651446
 [6,]  0.1651446  0.1651446
 [7,]  0.4954337  0.4954337
```

```
 [8,]  0.8257228  0.8257228
 [9,]  1.1560120  1.1560120
[10,]  1.4863011  1.4863011
```

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.

3.8. Матрицы, списки и таблицы данных

3.8.1. Матрицы

Матрицы — очень распространенная форма представления данных, организованных в форме таблицы. Про матрицы в R, в общем, нужно знать две важные вещи — во-первых, что они могут быть разной размерности и, во-вторых, что матриц как таковых в R, по сути, нет.

Начнем с последнего. Матрица в R — 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 (напомним, структуру любого объекта можно посмотреть при помощи очень важной команды `str()`) объектов `m` и `ma` не слишком различается, различается, по сути, лишь их вывод на экран компьютера. 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. Читателю предоставляется **догадаться**, почему матрица `mb` отличается от матрицы `ma` (ответ см. в конце главы).

Мы указали 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
, , 1

     [,1] [,2]
[1,]    1    3
[2,]    2    4

, , 2

     [,1] [,2]
[1,]    5    7
[2,]    6    8
```

`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. Аналогично можно создать и четырехмерную матрицу (как встроенные данные `Titatic`). Многомерные матрицы в R принято называть «arrays».

3.8.2. Списки

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 функции в R выдают «наружу» именно списки.

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

[[2]]
[1] 1 2 3

[[3]]
[1] TRUE

[[4]]
[1] NA

[[5]]
[[5]][[1]]
[1] "r"

[[5]][[2]]
[1] 4
```

Видно, что список — это своего рода ассорти. Вектор (и, естественно, матрица) может состоять из элементов одного и того же типа, а вот список — из чего угодно, в том числе (как видно из примера) и из других списков.

Теперь поговорим про индексирование, или выбор элементов списка. Элементы вектора выбираются, как мы помним, при помощи функции — квадратной скобки:

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

Элементы матрицы выбираются так же, только используются несколько аргументов (для двумерных матриц это номер строки и номер столбца — именно в такой последовательности):

```
>  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, и строки 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
```

3.8.3. Таблицы данных

И 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 (особенно начинающие пользователи R). 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 одного типа (а вот сами колонки могут быть разного типа). Проиллюстрируем это на примере созданных ранее векторов:

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

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), ]
     weight height 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. Подумайте, как применить команду `order()` для того, чтобы отсортировать колонки получившейся матрицы по алфавиту (см. ответ в конце главы).

* * *

Заключая разговор о типах данных, следует отметить, что эти типы вовсе не так резко отграничены друг от друга. Поэтому если вам 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.

* * *

Ответ к задаче про матрицы. Создавая матрицу `ma`, мы задали `byrow=TRUE`, то есть указали, что элементы вектора будут соединяться в матрицу построчно. Если бы мы указали `byrow=FALSE`, то получили бы точно такую же матрицу, как `mb`:

```
> ma <- matrix(m, ncol=2, byrow=FALSE)
> ma
     [,1] [,2]
[1,]    1    3
[2,]    2    4
```

Ответ к задаче про сортировку. Для того чтобы что-то сделать с колонками, надо использовать квадратные скобки с запятой в центре, а команды помещать справа от запятой. Логично использовать ту же команду `order()`:

```
> d.sorted <- d[order(d$sex, d$height), ]
> d.sorted[, order(names(d.sorted))]

     height    sex size weight
Женя    162 female    S     68
Катя    165 female    S     59
Вася    168   male    M     82
Жора    172.5 male    L     72
Коля    174   male    L     69
Петя    188   male   XL     93
Саша    192   male  XXL     87
```

Заметьте, что нельзя просто написать `order()` после запятой, эта команда должна выдать новый порядок колонок. Поэтому мы «скормили» ей имена этих колонок, которые для таблицы данных всегда можно получить, вызвав команду `names()`. Кстати, вместо `order()` здесь подошла бы и команда `sort()`, потому что нам надо отсортировать всего лишь один вектор.

Данные гадания на ромашке. Вот они (проценты от общего количества исходов):

```
> rev(sort(romashka))
          плюнет к сердцу прижмет          поцелует
                      24                        17                        16
        не любит                  любит   к черту пошлет
                      15                        15                        13
```

(Мы использовали `rev()`, потому что `sort()` сортирует по возрастанию.)

Как видно, гадание было не очень-то удачное...

Глава 4

Великое в малом: одномерные данные

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

4.1. 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 (мы еще поговорим об этом ниже). 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, тем, что Саша — экспедитор, а Катя — глава фирмы.

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

Получается, что из-за высокой Катиной зарплаты среднее гораздо хуже отражает «типичную», центральную зарплату, чем медиана. Отчего же так получается? Дело в том, что медиана вычисляется совершенно иначе, чем среднее.

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 — 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. Если вы читали предыдущий абзац внимательно, то, наверное, уже поняли, что медиана — это просто третий квартиль (50%). Первый и пятый квартили — это соответственно минимум и максимум, а второй и четвертый квартили используют для робастного вычисления разброса (см. ниже). Можно понятие «квартиль» расширить и ввести специальный термин для значения, отсекающего любой процент упорядоченного распределения (не обязательно по четвертям),— это называется «*квантиль*». Квантили используются, например, при анализе данных на нормальность (см. ниже).

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 способ использует тот факт, что таблицы данных — это списки из колонок. Для строк такой прием не сработает, надо будет запустить `apply()`. (Если вам пришел в голову четвертый способ, то напоминаем, что циклические конструкции типа `for` в R без необходимости не приветствуются).

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 (или межквартильный разброс), 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. Мы проверим это ниже.

В наших данных по зарплате — всего 7 цифр. А как понять, есть ли какие-то «выдающиеся» цифры, типа Катиной зарплаты, в данных большого, «тысячного» размера? Для этого есть графические функции. Самая простая — так называемый «ящик-с-усами», или боксплот. 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 (а такие программы написаны!), то надо использовать `replace=FALSE`, потому что из колоды нельзя достать опять ту же самую карту. Кстати говоря, из того, что значения случайные, следует, что результаты последующих вычислений могут отличаться, если

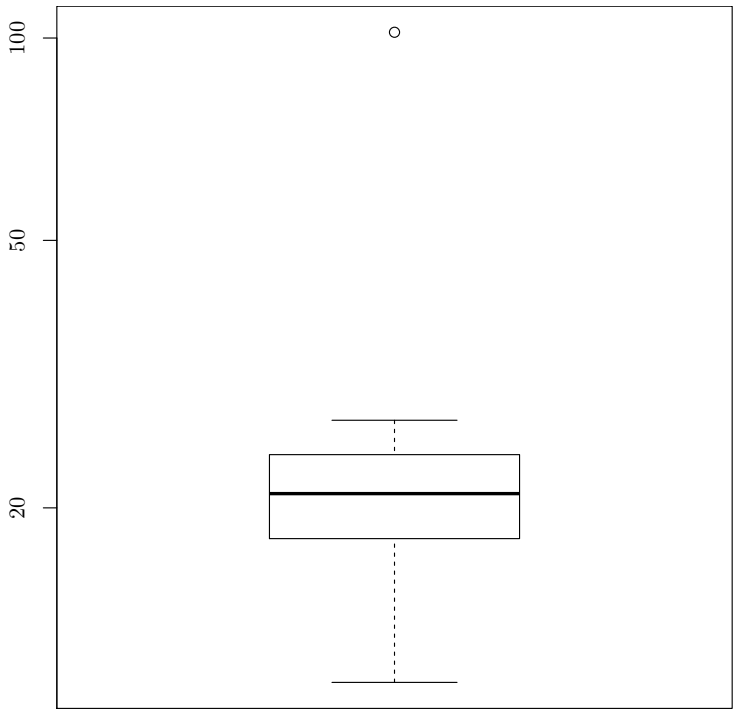


Рис. 9. «Ящик-с-усами», или боксплот

их воспроизвести еще раз, поэтому ваш график может выглядеть чуть иначе.

Но вернемся к боксплоту. Как видно, Катина зарплата 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, ограничен сверху 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. Боксплоты были специально придуманы известным статистиком Джоном Тьюки, для того чтобы быстро, эффективно и устойчиво отражать основные робастные характеристики выборки. R может рисовать несколько боксплотов сразу (то есть эта команда *векторизована*, см. результат на рис. 10):

```
> boxplot(trees)
```

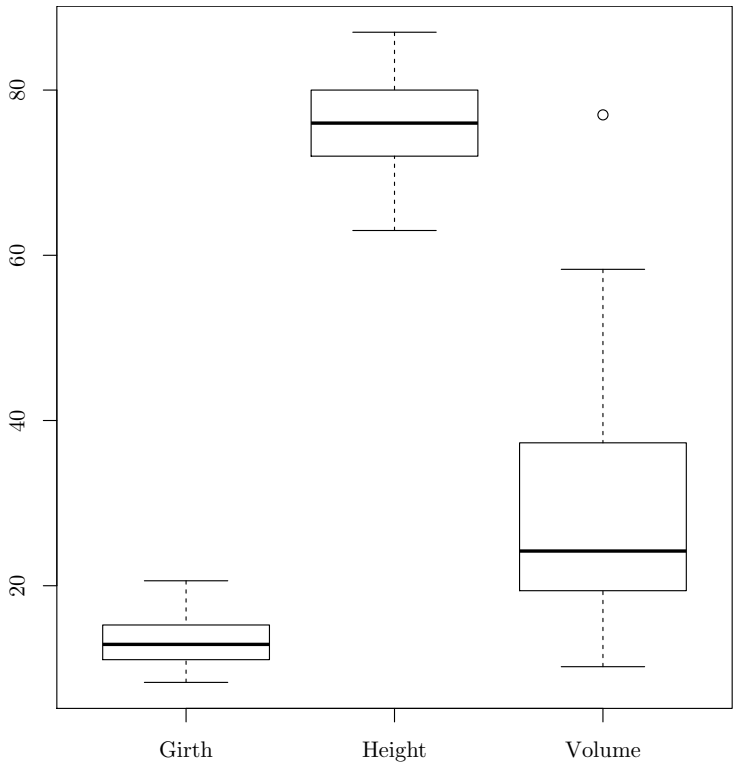


Рис. 10. Три боксплота, каждый отражает одну колонку из таблицы данных

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

Другой способ графического изображения — это гистограмма, то есть линии столбиков, высота которых соответствует встречаемости данных, попавших в определенный диапазон (рис. 11):

```
> 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))
(10.9,15.5]   (15.5,20]   (20,24.6] (24.6,29.1] (29.1,33.7]
```

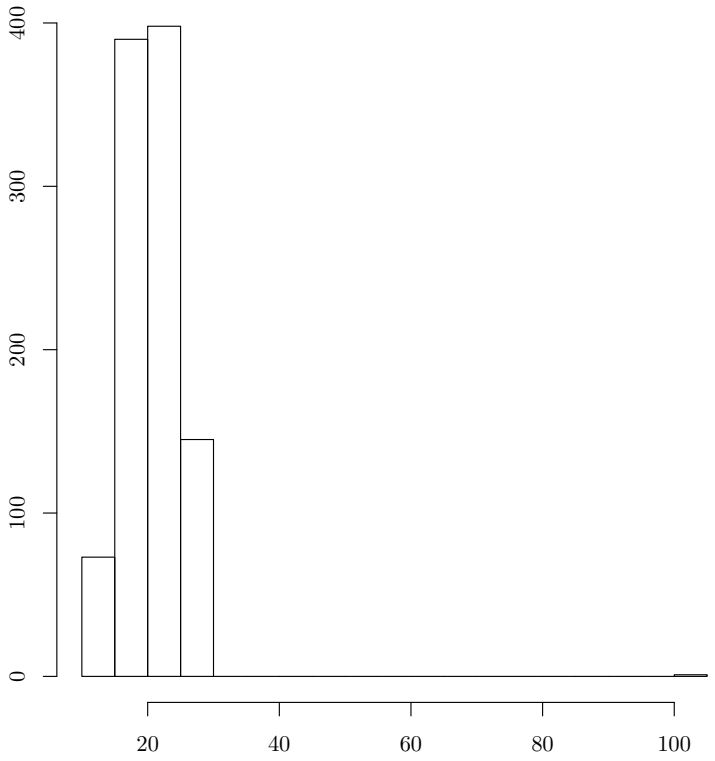


Рис. 11. Гистограмма зарплат 1007 гипотетических сотрудников

```
         76          391          295          244            0
(33.7,38.3]
          0
(38.3,42.8] (42.8,47.4] (47.4,51.9] (51.9,56.5] (56.5,61.1]
          0            0            0            0            0
(61.1,65.6]
          0
(65.6,70.2] (70.2,74.7] (74.7,79.3] (79.3,83.9] (83.9,88.4]
          0            0            0            0            0
  (88.4,93]   (93,97.5]  (97.5,102]
          0            0            1
```

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

```
> stem(salary, scale=2)
  The decimal point is 1 digit(s) to the right of the |
```

```
  1 | 19
  2 | 1157
  3 |
  4 |
  5 |
  6 |
  7 |
  8 |
  9 |
 10 | 2
```

Это очень просто — значения данных изображаются не точками, а цифрами, соответствующими самим этим значениям. Таким образом, видно, что в интервале от 10 до 20 есть две зарплаты (11 и 19), в интервале от 20 до 30 — четыре и т. д.

Другая функция тоже близка к гистограмме, но требует гораздо более изощренных вычислений. Это график плотности распределения (рис. 12):

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

(Мы использовали «добавляющую» графическую функцию `rug()`, 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 боксплотов 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 построить график-улей, потребуется загрузить (а возможно, еще и установить сначала) пакет `beeswarm`. После этого можно поглядеть на сам «улей» (рис. 13):

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

Мы здесь не просто построили график-улей, но еще и добавили туда боксплот, чтобы стали видны квартили и медиана. Для этого нам понадобился аргумент `add=TRUE`.

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

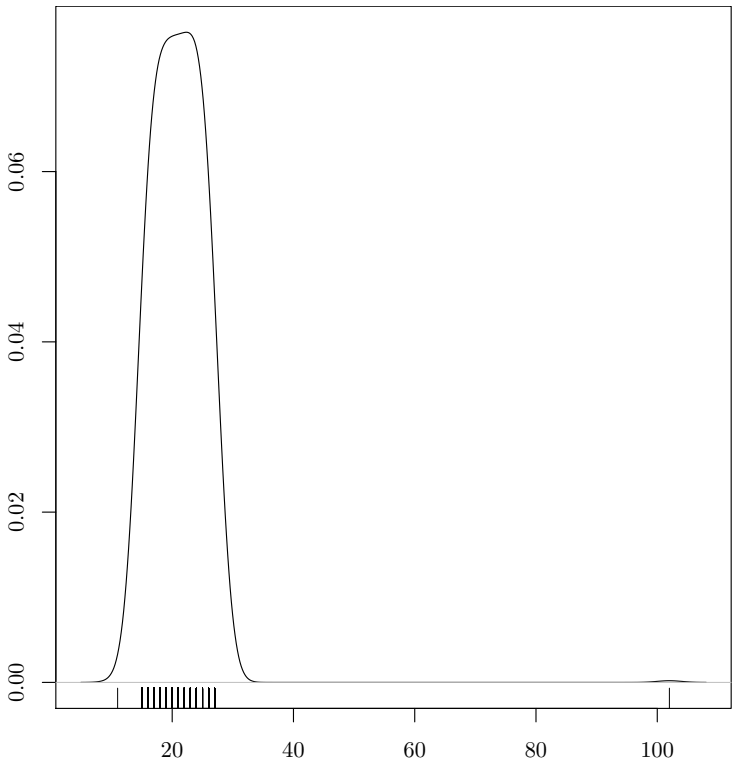


Рис. 12. Плотность распределения зарплат 1007 гипотетических сотрудников

```
> lapply(list(salary, salary2), summary)
[[1]]
   Min. 1st Qu.  Median    Mean 3rd Qu.
  11.00   20.00   21.00   32.29   26.00
   Max.
 102.00

[[2]]
   Min. 1st Qu.  Median    Mean 3rd Qu.
  11.00   18.00   21.00   21.09   24.00
   Max.
 102.00
```

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

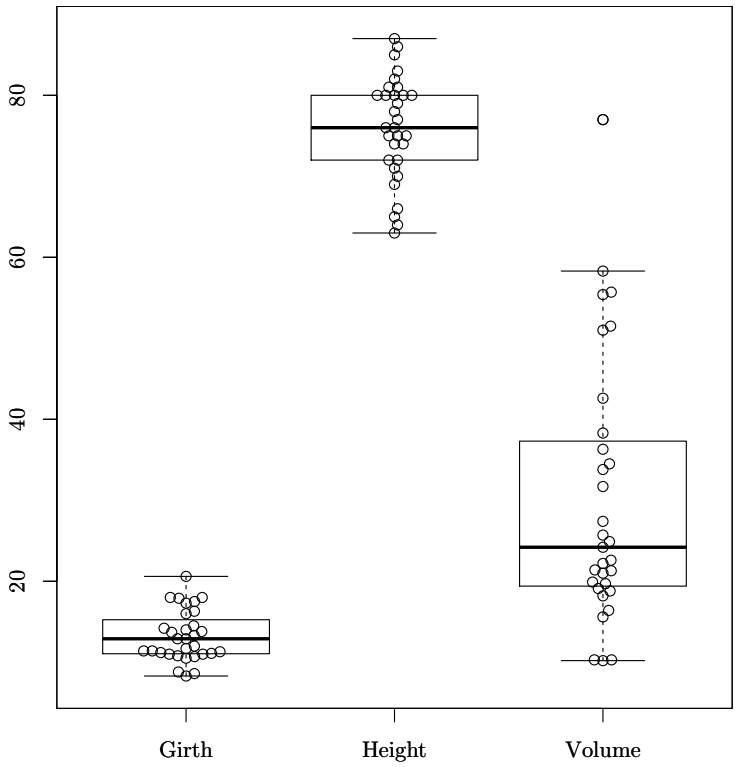


Рис. 13. График-«улей» с наложенными боксплотами для трех характеристик деревьев

добавлением случайно взятых «зарплат» вид распределения не должен был существенно поменяться.

Функция `summary()` — общая, и по законам объект-ориентированного подхода она возвращает разные значения для объектов разного типа. Вы только что увидели, она работает для числовых векторов. Для списков она работает немного иначе. Вывод может быть, например, таким (на примере встроенных данных `attenu` о 23 землетрясениях в Калифорнии):

```
> summary(attenu)

     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
     accel
 Min.   :0.00300
 1st Qu.:0.04425
 Median :0.11300
 Mean   :0.15422
 3rd Qu.:0.21925
 Max.   :0.81000
```

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 мы для быстроты применили `sapply()` — вариант `lapply()` с упрощенным выводом, и `colMeans()`, которая просто вычисляет среднее для каждой колонки. Сразу отметим, что функций, подобных `colMeans()`, в R несколько. Например, очень 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()`.

4.2. Ошибочные данные

Способность функции `summary()` указывать пропущенные данные, максимумы и минимумы служит очень хорошим подспорьем на самом раннем этапе анализа данных — проверке качества. Предположим, у нас есть данные, набранные с ошибками, и они находятся в директории `data` внутри текущей директории:

```
> dir("data")
[1] "errors.txt" ...
```

```
> err <- read.table("data/errors.txt", h=TRUE, sep="\t")
> 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
```

Обработка начинается с проверки наличия нужного файла. 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.

4.3. Одномерные статистические тесты

Закончив разбираться с описательными статистиками, перейдем к простейшим статистическим тестам (подробнее тесты рассмотрены в следующей главе). 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:

```
> t.test(salary, mu=mean(salary))
One Sample t-test
data:  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. Подробнее о статистических гипотезах можно прочесть в следующей главе.

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 трех с половиной тысяч до 61 тысячи рублей.

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

```
> 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 (попробуйте провести этот тест самостоятельно):

```
> shapiro.test(salary)
> shapiro.test(salary2)
```

Но что же он показывает? 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 столько случайных чисел, распределенных по нормальному закону, сколько указано в его аргументе. Это аналог функции `sample()`. Раз мы получили высокое p-значение, то это свидетельствует о том, что альтернативная гипотеза в данном случае: «распределение не соответствует нормальному». Кроме того, чтобы результаты при вторичном воспроизведении были теми же, использована функция `set.seed()`, регулирующая встроенный в R генератор случайных чисел так, чтобы числа в следующей команде были созданы по одному и тому же «закону».

Таким образом, распределение данных в `salary` и `salary2` отличается от нормального.

Другой популярный способ проверить, насколько распределение похоже на нормальное,— графический. Вот как это делается (рис. 14):

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

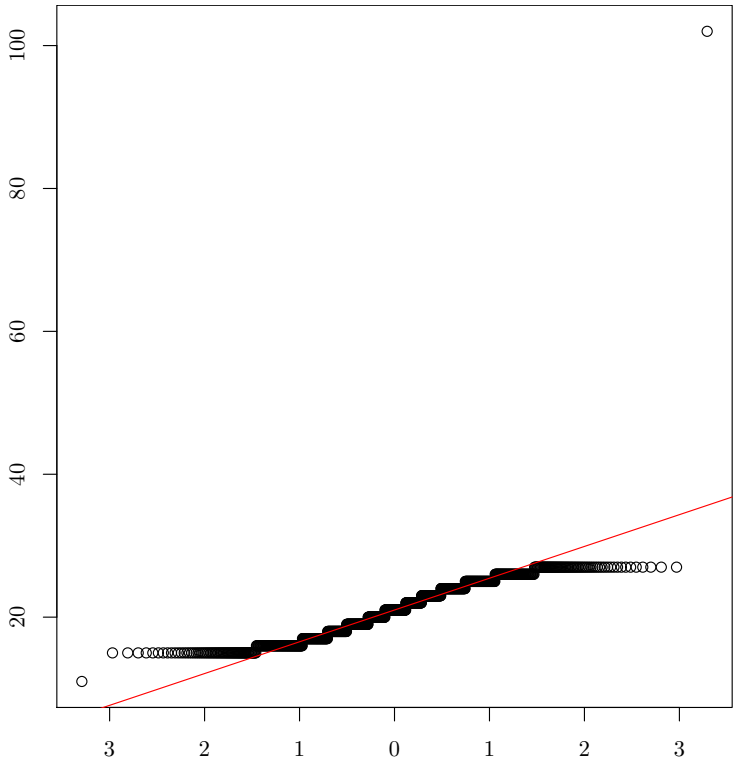


Рис. 14. Графическая проверка нормальности распределения

Для каждого элемента вычисляется, какое место он занимает в сортированных данных (так называемый «выборочный квантиль») и какое место 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.

4.4. 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 (об этом подробнее написано в главе про двумерные данные). 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. Проверить это можно, заглянув в справку, а можно и **экспериментально** (как? — см. ответ в конце главы). 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 1: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 для первой 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 test

data:  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 (**проверьте** самостоятельно, как она работает):

```
> 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.

4.5. 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%. Следует внимательно относиться к вычислению долей и не забывать про исходные данные. В 1960-е годы в районном отделе по сельскому хозяйству было обнаружено, что падеж лошадей в одном сельском поселении составил 50%. Пришлось составить компетентную комиссию для проверки причин такого ужасного явления. Прибыв на место, комиссия обнаружила, что в данном поселении было всего две лошади, в том числе и сдохшая недавно старая кляча, которая и послужила причиной формирования и выезда на место комиссии!

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 not
equal 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

текста пропорций), а также `mcnemar.test()` (для теста Мак-Немара, который проводится тогда, когда выборки связаны друг с другом). Узнать, как их использовать, можно, прочитав справку (и особенно примеры!) по обеим функциям.

* * *

Ответ к задаче про функцию `normality()`. Чтобы узнать, откуда взять значения p-value, надо сначала вспомнить, что в R почти все, что выводится на экран,— это результат «печати» различных списков при помощи невидимой команды `print()`. А из списка легко «достать» нужное либо по имени, либо по номеру (если, скажем, имени у элементов нет — так иногда бывает). Сначала выясним, из чего состоит список-вывод функции `shapiro.test()`:

```
> str(shapiro.test(rnorm(100)))
List of 4
 $ statistic: Named num 0.992
  ..- attr(*, "names")= chr "W"
 $ p.value  : num 0.842
 $ method   : chr "Shapiro-Wilk normality test"
 $ data.name: chr "rnorm(100)"
 - attr(*, "class")= chr "htest"
```

В этом списке из 4 элементов есть элемент под названием `p.value`, что нам и требовалось. Проверим на всякий случай, то ли это:

```
> set.seed(1683)
> shapiro.test(rnorm(100))$p.value
[1] 0.8424077
```

Именно то, что надо. Осталось только вставить это название в нашу функцию.

Ответ к задаче про выборы из предисловия.

Для того чтобы рассчитать p-value для нашей пропорции (48% проголосовавших за кандидата B), можно воспользоваться описанным выше `prop.test()`:

```
> prop.test(0.48*262, 262)

1-sample proportions test with continuity correction

data:  0.48 * 262 out of 262, null probability 0.5
X-squared = 0.343, df = 1, p-value = 0.5581
```

```
alternative hypothesis: true p is not equal to 0.5
95 percent confidence interval:
 0.4183606 0.5422355
sample estimates:
   p
0.48
```

Получается, что отвергнуть нулевую гипотезу о равенстве пропорций мы не можем, слишком велико p-value (гораздо больше «положенных» 0.05 и даже «либеральных» 0.1). Процент же проголосовавших может лежать в интервале от 42% до 54%! Итак, по результатам опроса нельзя сказать, что кандидат A победил.

А вот так можно рассчитать количество человек, которых надо опросить, чтобы быть уверенным в том, что 48% и 52% и вправду отражают реальную ситуацию (генеральную совокупность):

```
> 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`). Чуть более подробно про мощность (power) и ее связь с другими характеристиками теста можно прочитать в следующей главе.

Глава 5

Анализ связей: двумерные данные

Здесь речь пойдет о том, как работать с двумя выборками. Если у нас два набора цифр, то первое, что приходит в голову,— сравнить их. Для этого нам потребуются статистические тесты.

5.1. 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.

5.1.1. Статистические гипотезы

Из первой главы мы знаем, что статистическая выборка должна быть репрезентативной (то есть адекватно характеризовать генеральную совокупность, или, как ее еще называют, популяцию). Но как же мы можем знать, репрезентативна ли выборка, если мы не исследовали всю генеральную совокупность? Этот логический тупик называют парадоксом выборки. 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 наших 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.

Вполне возможно, что 1001-ый (или 1 000 001-ый) факт опровергнет эту теорию. 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 (чувствуете разницу?). Если же мы не можем опровергнуть нулевую гипотезу, то мы вынуждены принять ее.

5.1.2. 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 возможно четыре исхода (табл. 5.1).

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

Таблица 5.1. Статистические ошибки

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 (см. первую главу), однозначного ответа на этот вопрос нет. Более или менее общепринято, что пороговым значением надо считать 0.05 (то есть альтернативная гипотеза отвергается, если вероятность ошибки при ее принятии больше или равна 5%). В медицине ценой ошибки нередко являются человеческие жизни, поэтому там пороговое значение часто принимают равным 0.01 или даже 0.001 (то есть решение о существовании закономерности принимается, если вероятность ошибки ничтожна).

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.

Для полной ясности перерисуем нашу таблицу для конкретного случая — исследования зависимости между двумя признаками (табл. 5.2).

5.2. Есть ли различие, или 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,`

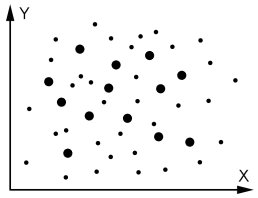
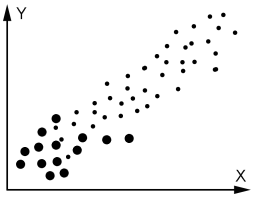
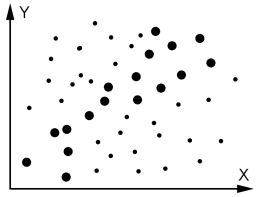
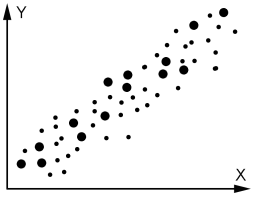
Популяция / Выборка	Верна H_0	Верна H_1
Принимаем H_0		
Принимаем H_1		

Таблица 5.2. Статистические ошибки при исследовании зависимости между двумя признаками. Мелкие точки отражают объекты генеральной совокупности (популяции), крупные точки — объекты из нашей выборки

`3, 4, 5, 6, 7, 8, 9)` и `c(5, 5, 5, 5, 5, 5, 5, 5, 5)` различаться не будут.

Как вы помните, 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 рост мужчин 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 (two sample 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 между сравниваемыми 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 (рис. 15). 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 предположе-

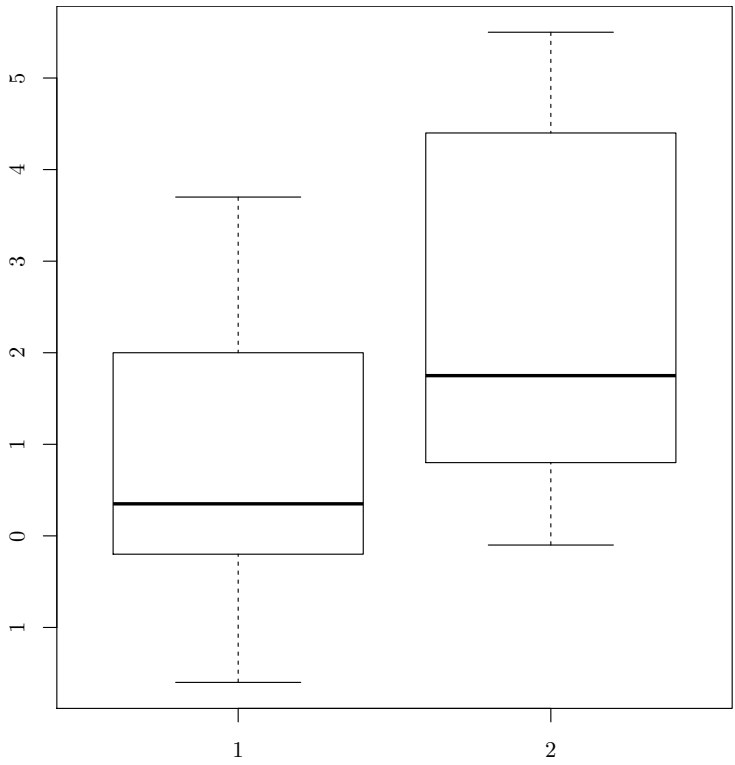


Рис. 15. Среднее приращение продолжительности сна после начала приема разных лекарств в двух группах по отношению к контрольной

нии, 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 *ad hoc*, то есть без знания какой-либо дополнительной информации. Существуют и *post hoc* тесты (скажем, обсуждаемый ниже «Tukey Honest Significant Differences test»), но их немного.

Для сравнения двух выборок существуют, разумеется, и непараметрические тесты. Один из них, *тест знаков*, настолько прост, что его нет в 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.

Задача. Придумайте, при помощи какого R-выражения сделать такой тест, и протестируйте две выборки, которые мы привели в начале раздела.

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. Впрочем, попробуйте выяснить самостоятельно (это **задача**!), насколько близки к нормальным помесячные распределения значений концентраций озона (см. ответ в конце главы).

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 correction

data:  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.

Различия между выборками хорошо иллюстрировать при помощи графика «ящик-с-усами», например (рис. 16):

```
> boxplot(Ozone ~ Month, data = airquality,
+ subset = Month %in% c(5, 8))
```

(Обратите внимание на то, что здесь для `boxplot()` мы использовали ту же самую формулу модели.)

Часто считают, что если «ящики» перекрываются более чем на треть своей высоты, то выборки достоверно не различаются.

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 μ (см. предыдущую главу).

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 A за первую и вторую четверть? Какой класс учился в первой четверти лучше — A или B? Ответ см. в конце главы.

Задача. В супермаркете, среди прочих, работают два кассира. Для того чтобы проанализировать эффективность их работы, несколько раз в день в середине недели подсчитывали очереди к каждой из них. Данные записаны в файле `kass.txt`. Какой кассир работает лучше?

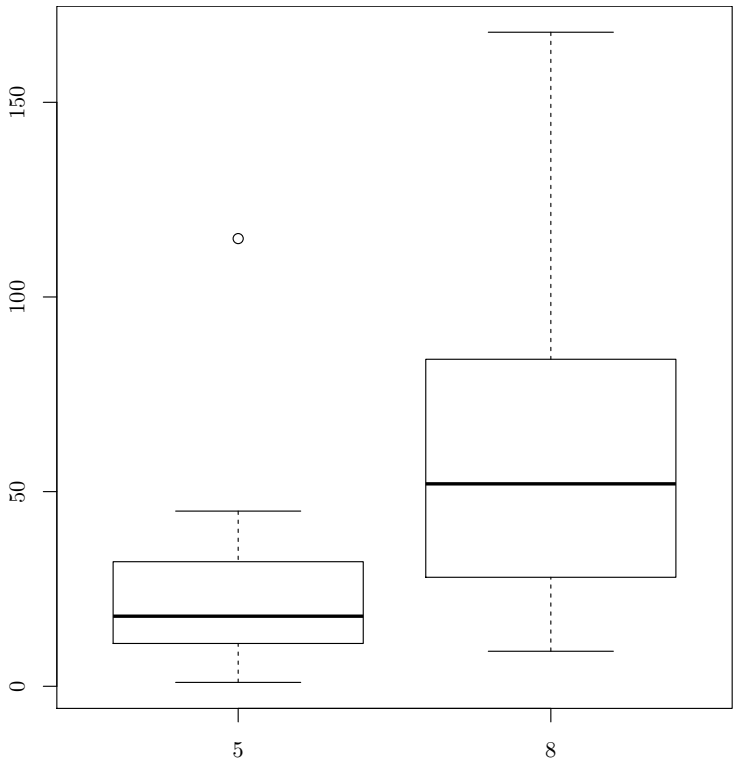


Рис. 16. Распределение озона в воздухе в мае и июне

5.3. 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
```

Функция `mosaicplot` позволяет получить графическое изображение таблицы сопряженности (рис. 17):

```
> titanic <- apply(Titanic, c(1, 4), sum)
> titanic
> titanic
      Survived
Class   No Yes
  1st  122 203
  2nd  167 118
  3rd  528 178
  Crew 673 212
> mosaicplot(titanic, col = c("red", "green"), main = "",
+ cex.axis=1)
```

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 test

data:  x
X-squared = 138.2898, df = 9, p-value < 2.2e-16
```

(Того 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()` (рис. 18):

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

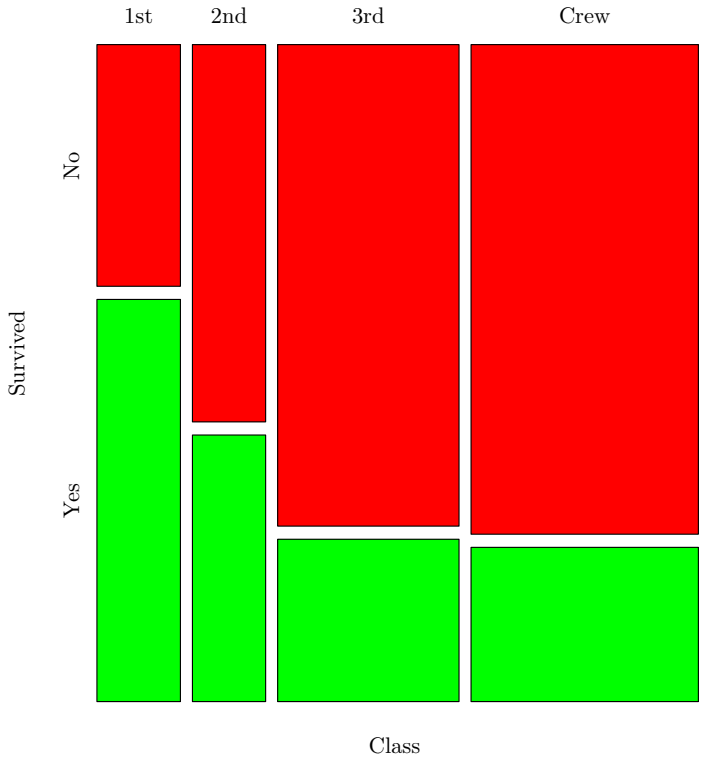


Рис. 17. Доля выживших на «Титанике» пассажиров и членов команды в зависимости от класса их билета: мозаичный график

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 совсем не 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 как следует вспомнить, кто что ел на банкете, и тем самым выяснить причину отравления. Получившиеся данные имеют следующий формат:

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

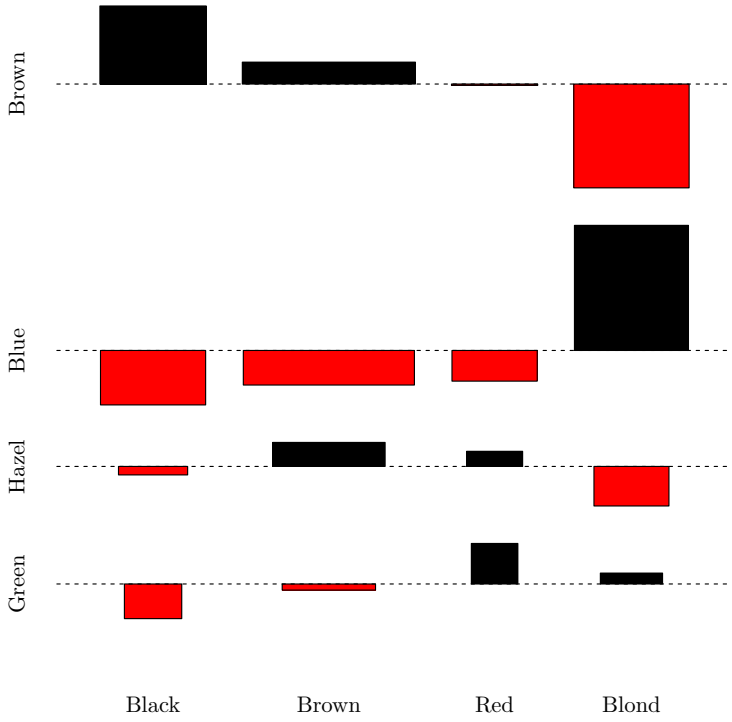


Рис. 18. Связь между цветом волос и цветом глаз: график сопряженности

```
> 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"
```

Небольшая хитрость (цикл `for`) позволила нам не писать тест тринадцать раз подряд (хотя copy-paste никто не отменял, и можно было бы сделать так, как сделано в разделе «Прогноз» главы про временные ряды). Без функции `print()` цикл бы ничего не напечатал, а `paste()` помогла оформить результат. Ну а результат таков, что есть два значимых соответствия — с цезарским салатом (`CAESAR`) и с помидорами (`TOMATO`). Посмотрим, что нам покажет график ассоциаций (рис. 19):

```
> assocplot(table(ILL=tox$ILL, CAESAR=tox$CAESAR))
```

Практически такая же картина и по тем, кто ел помидоры. Виновник найден! Почти. Ведь вряд ли испорченными были сразу оба блюда. Надо теперь попробовать разобраться, что же было главной причиной. Для этого вам придется заглянуть ниже, туда, где мы знакомимся с логистической регрессией.

* * *

Кроме методов работы с таблицами сопряженности (таких, например, как тест хи-квадрат), номинальные и шкальные данные можно

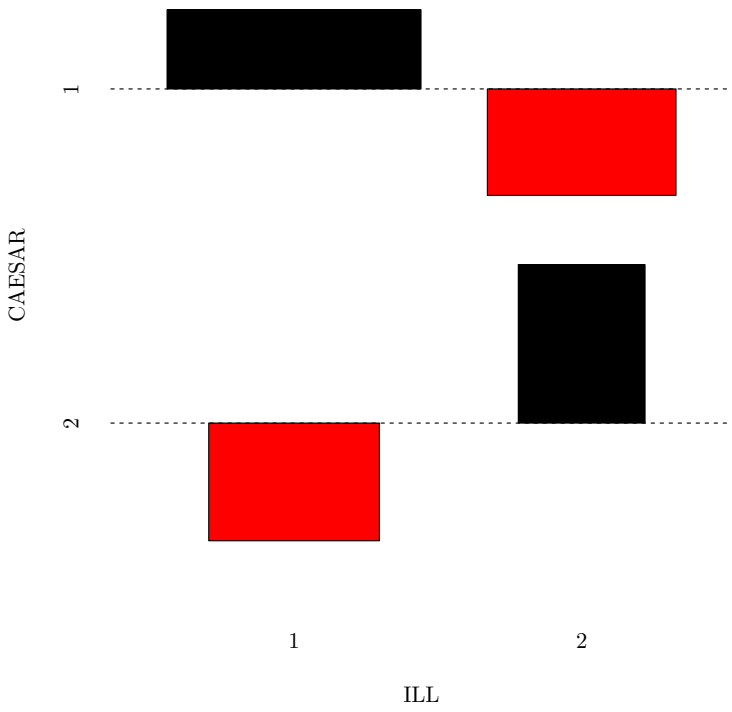


Рис. 19. Соответствие между отравившимися и теми, кто ели цезарский салат

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("data/pokorm_03.dat", h=TRUE, sep=";")
> library(concord)
> cohen.kappa(as.matrix(pok))
Kappa test for nominally classified data
```

```
2 categories - 2 methods
kappa (Cohen) = 0.718855 , Z = 8.56608 , p = 0
kappa (Siegel) = 0.67419 , Z = 7.11279 , p = 5.68656e-13
kappa (2*PA-1) = 0.761658
```

(Мы загрузили пакет `concord`, потому что именно в нем находится нужная нам функция `cohen.kappa()`.)

Каппа довольно близка к единице (0.718855), вероятность нулевой гипотезы — нулевая. Это значит, что результаты исследования можно считать согласными 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? Ответ см. в конце главы.

5.4. Есть ли взаимосвязь, или 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 *коэффициент детерминации*: 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. Примером такой связи может служить связь между степенью возбужденности человека и качеством решения им математических задач.

Очень слабо возбужденный человек (скажем, засыпающий) и очень сильно возбужденный (во время футбольного матча) будет решать задачи гораздо хуже, чем умеренно возбужденный человек. Поэтому, перед тем как оценить взаимосвязь численно (вычислить коэффициент корреляции), 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 переменными А 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
Volume 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 в в этом 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) ρ. Он менее подвержен влиянию случайных «выбросов» в данных, чем коэффициент Пирсона. Для подсчета ρ 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()` (рис. 20):

```
> 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 температуры»).

Еще один интересный способ представления корреляционной матрицы предоставляется пакетом `ellipse`. В этом случае значения коэффициентов корреляции рисуются в виде эллипсов. Чем ближе значение коэффициента 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()`:

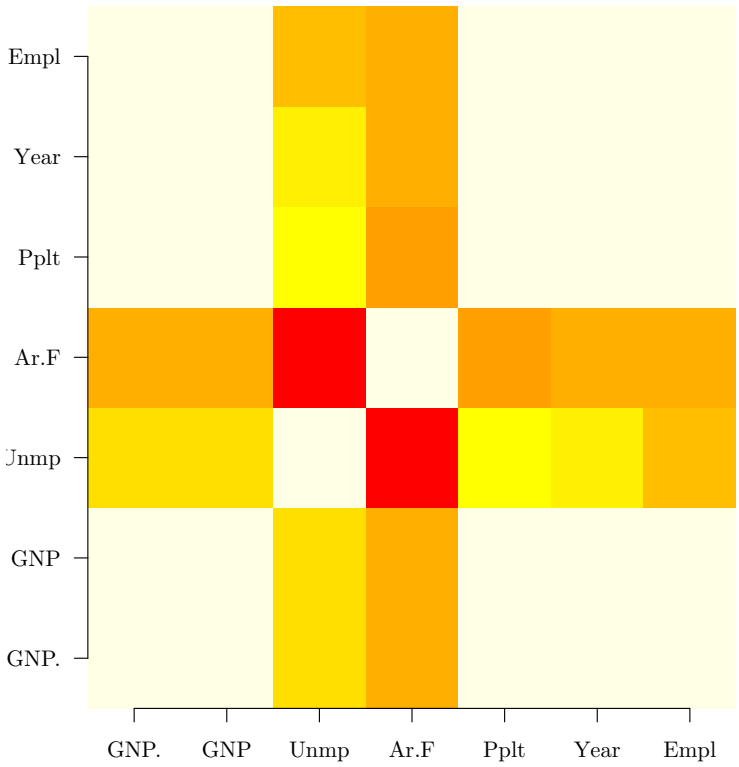


Рис. 20. Графическое представление корреляционной матрицы

```
> with(trees, cor.test(Girth, Height))

Pearson's product-moment correlation

data:  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:
      cor
0.5192801
```

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.

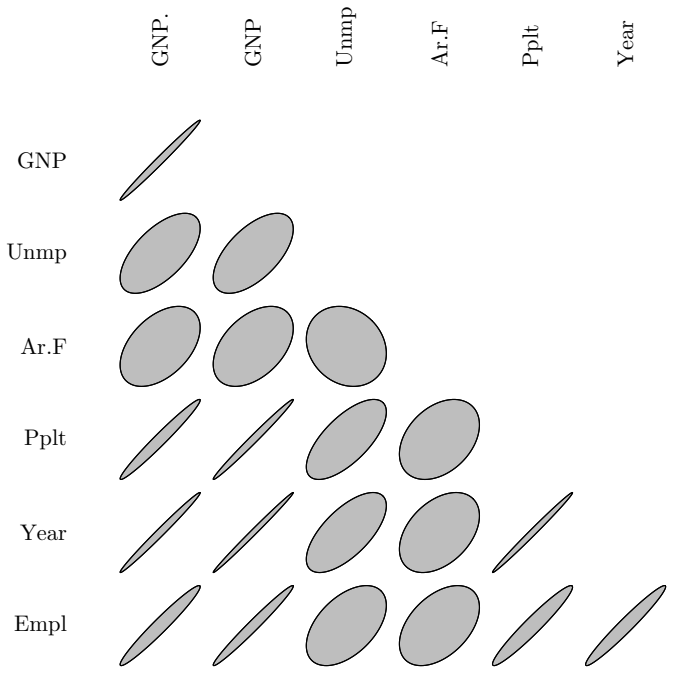


Рис. 21. Коэффициенты корреляции в виде эллипсов

5.5. 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 σ^2, который не 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:

$$R^2 = 1 - \sigma_m^2/\sigma_y^2,$$

где σ_y^2 — 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")
```

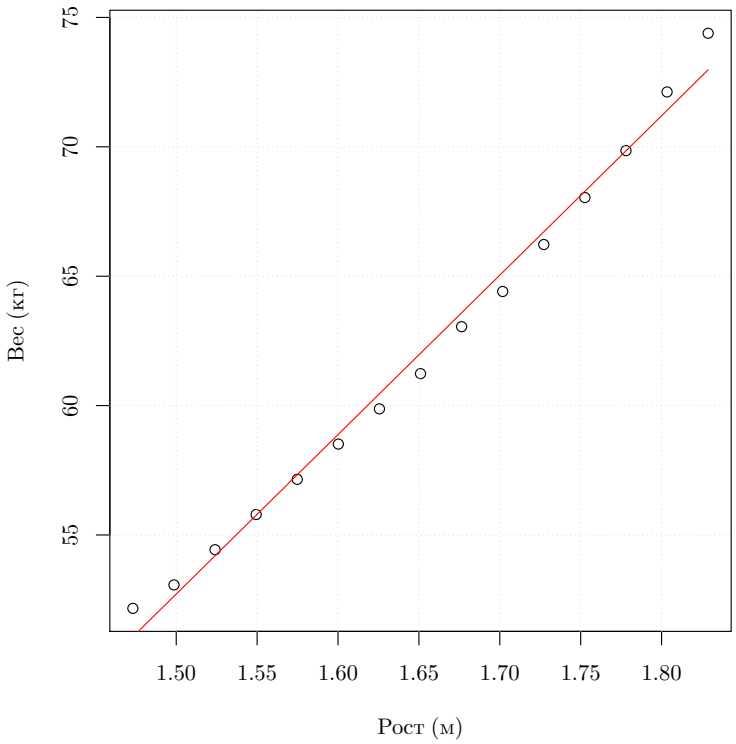


Рис. 22. Зависимость между ростом и весом

Для просмотра результатов линейной аппроксимации можно использовать 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 freedom
Multiple 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:

1. Получена линейная регрессионная модель вида

   `Вес(мод) = -39.697 + 61.61 * Рост`,

   то есть увеличение роста на 10 см соответствует увеличению веса примерно на 6 кг.

2. Наибольшее положительное отклонение истинного значения отклика от модельного составляет 1.4137 кг, наибольшее отрицательное -0.7862 кг.

3. Почти половина остатков находится в пределах от первой квартили (`1Q` $= -0.5141$ кг) до третьей (`3Q` $= 0.3364$ кг).

4. Все коэффициенты значимы на уровне `p-value < 0.001`; это показывают `***` в строке `Coefficients`) и сами значения p-value `Pr(>|t|)`: `1.71e-09` для b_0 (`«(Intercept)»`) и `1.09e-14` (`«height»`) для b_1.

5. Среднеквадратичное отклонение (`Adjusted R-squared`) для данной модели составляет $R^2 = 0.9903$. Его значение близко к 1, что говорит о высокой значимости.

6. 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.

7. Если на основе этого анализа будет составлен отчет, то надо будет указать еще и значения степеней свободы `df`: 1 и 13 (по колонкам и 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("data/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

Residual standard error: 0.01929 on 151 degrees of freedom
Multiple R-squared: 0.152,Adjusted R-squared: 0.1463
F-statistic: 27.06 on 1 and 151 DF,  p-value: 6.331e-07

Call:
lm(formula = KAND.2/IZBIR ~ JAVKA)

Residuals:
      Min        1Q    Median        3Q       Max
-0.036542 -0.014191 -0.000162  0.011782  0.050139

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.093487   0.007807  11.974  < 2e-16 ***
JAVKA       -0.066597   0.011911  -5.591 1.03e-07 ***
---
```

```
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.0169 on 151 degrees of freedom
Multiple R-squared: 0.1715,Adjusted R-squared: 0.166
F-statistic: 31.26 on 1 and 151 DF,  p-value: 1.027e-07

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

Residual standard error: 0.03684 on 151 degrees of freedom
Multiple R-squared: 0.945,Adjusted R-squared: 0.9446
F-statistic:  2595 on 1 and 151 DF,  p-value: < 2.2e-16
```

Коэффициенты регрессии у третьего кандидата сильно отличаются. Да и R^2 гораздо выше. Посмотрим, как это выглядит графически (рис. 23):

```
> plot(KAND.3/IZBIR ~ JAVKA, xlim=c(0,1), ylim=c(0,1),
+ xlab="Явка", ylab="Доля проголосовавших за кандидата")
> points(KAND.1/IZBIR ~ JAVKA, pch=2)
> points(KAND.2/IZBIR ~ JAVKA, pch=3)
> abline(lm.3)
> abline(lm.2, lty=2)
> abline(lm.1, lty=3)
> legend("topleft", lty=c(3,2,1),
+ legend=c("Кандидат 1","Кандидат 2","Кандидат 3"))
> detach(vybory)
```

Итак, третий кандидат имеет электорат, который значительно отличается по своему поведению от электоратов двух первых кандидатов. Разумеется, хочется проверить это статистически, а не только визуально. Для того чтобы сравнивать регрессии, существует специальный

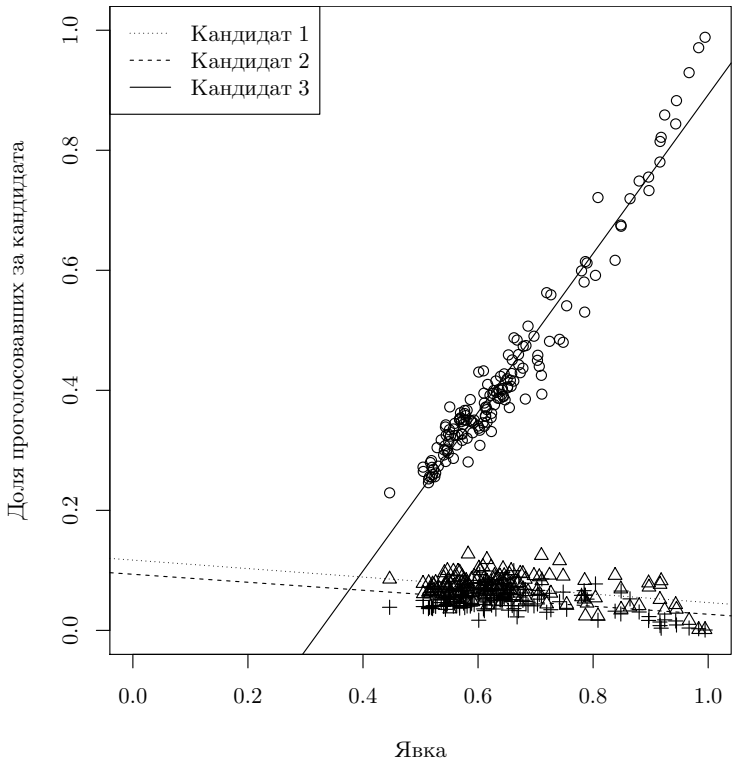


Рис. 23. Зависимость результатов трех кандидатов от явки на выборы

метод, анализ ковариаций (ANCOVA). Подробное рассмотрение этого метода выходит за рамки настоящей книги, но мы сейчас 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)

Call:
lm(formula = dolja ~ javka * kand, data = vybory2)

Residuals:
      Min        1Q    Median        3Q       Max
-0.116483 -0.015470 -0.000699  0.014825  0.102810

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 ***
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.02591 on 453 degrees of freedom
Multiple R-squared: 0.9824,Adjusted R-squared: 0.9822
F-statistic:  5057 on 5 and 453 DF,  p-value: < 2.2e-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: сначала большинство 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.

5.6. Вероятность успеха, или Логистическая регрессия

Номинальные данные очень трудно обрабатывать статистически. Тест пропорций да 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

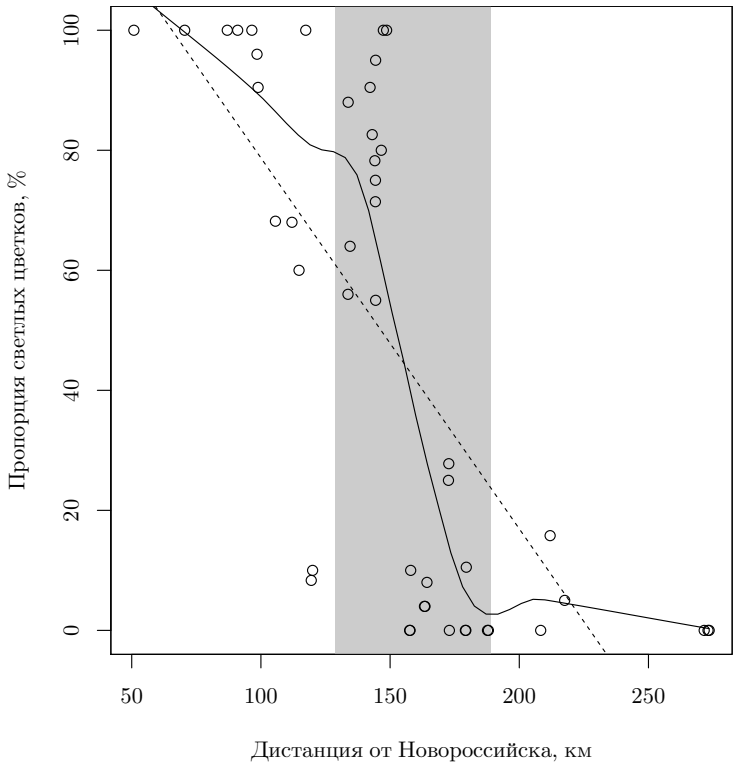


Рис. 24. Изменение пропорций светлых цветков в популяциях первоцвета

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")
> 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 тут подходят 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 математические 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.

Попробуйте теперь решить такую **задачу**. В файле `pokaz.txt` находятся результаты эксперимента с показом предметов на доли секунды, описанного в первой главе. Первая колонка содержит номер испытуемого (их было десять), вторая — номер испытания (пять на человека,

испытания проводились последовательно), а третья — результат в бинарной форме: 0 — не угадал(а), 1 — угадал(а). Выясните, есть ли связь между номером испытания и результатом.

В выдаче функции `summary.glm()` стоит обратить внимание на предпоследнюю строчку, AIC (Akaike's Information Criterion) — информационный критерий Акаике. Это критерий, который позволяет сравнивать модели между собой (несколько подробнее про AIC рассказано в главе про временные ряды). Чем меньше AIC, тем лучше модель. Покажем это на примере. Вернемся к нашим данным про «виновника» отравления. Помидоры или салат? Вот так можно попробовать решить проблему:

```
> tox.logit <- glm(formula=I(2-ILL) ~ CAESAR + TOMATO,
+ family=binomial, data=tox)
> tox.logit2 <- update(tox.logit, . ~ . - TOMATO)
> tox.logit3 <- update(tox.logit, . ~ . - CAESAR)
```

Сначала мы построили логистическую модель. Поскольку логистическая модель «хочет» в качестве отклика (слева от тильды) получить 0 или 1, то мы просто вычли значения переменной `ILL` из 2, при этом заболевшие стали закодированы как 0, а здоровые — как 1. Функция `I()` позволяет нам избежать интерпретации минуса как части формулы модели, то есть возвращает минусу обычные арифметические свойства. Затем при помощи `update()` мы видоизменили исходную модель, убрав из нее помидоры, а потом убрали из исходной модели салат (точки в формуле модели означают, что используются все воздействия и все отклики). Теперь посмотрим на AIC:

```
> tox.logit$aic
[1] 47.40782
> tox.logit2$aic
[1] 45.94004
> tox.logit3$aic
[1] 53.11957
```

Что же, похоже, что настоящим виновником отравления является салат, ведь модель с салатом обладает наименьшим AIC. Почему же тест хи-квадрат указал нам еще и на помидоры? Дело, скорее всего, в том, что между салатом и помидорами существовало *взаимодействие* (interaction). Говоря человеческим языком, те, кто брали салат, обычно брали к нему и помидоры.

5.7. 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 только 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. Аналогичные рассуждения могут быть применены и к экстремальным видам спорта. Например, вероятность того, что парашют не раскроется при прыжке, довольно мала (допустим, 1/100), и странно бы было ожидать, что парашют не раскроется как раз тогда, когда человек прыгает впервые. При этом любой десантник, имеющий опыт нескольких сотен прыжков, может рассказать несколько захватывающих историй о том, как ему пришлось использовать запасной парашют.

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 различается только одна пара выборок, и в случае, если различаются все выборки. Чтобы узнать, *какие именно* выборки отличаются, надо будет проводить специальные тесты (о них рассказано ниже).

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()) (рис. 25):

```
> 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 соответствуют 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. Глядя на боксплот, можно предположить, что это — блондины. Проверим предположение при помощи множественного t-теста (pairwise t-test):

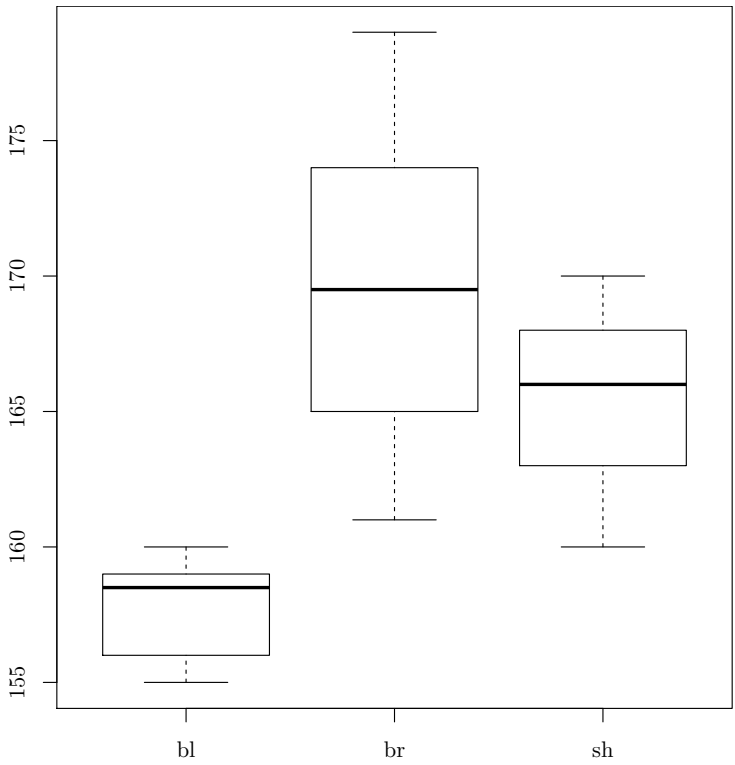


Рис. 25. Различается ли вес между тремя выборками людей с разным цветом волос? (Данные получены искусственно)

```
> pairwise.t.test(data$ROST, data$CVET)

Pairwise comparisons using t tests with pooled SD
data:  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. Есть и еще один похожий тест. Это «Tukey Honest Significant Differences test» (у него есть и специальный график, см. рис. 26):

```
> rost.cvet <- aov(lm(data$ROST ~ data$CVET))
> (rost.cvet.hsd <- TukeyHSD(rost.cvet))
```

```
  Tukey multiple comparisons of means
    95% family-wise confidence level

Fit: aov(formula = lm(data$ROST ~ data$CVET))

$`data$CVET`
           diff       lwr       upr    p adj
br-bl 11.933333  9.678935 14.187732 0.00e+00
sh-bl  7.533333  5.278935  9.787732 0.00e+00
sh-br -4.400000 -6.654399 -2.145601 3.43e-05

> plot(rost.cvet.hsd)
```

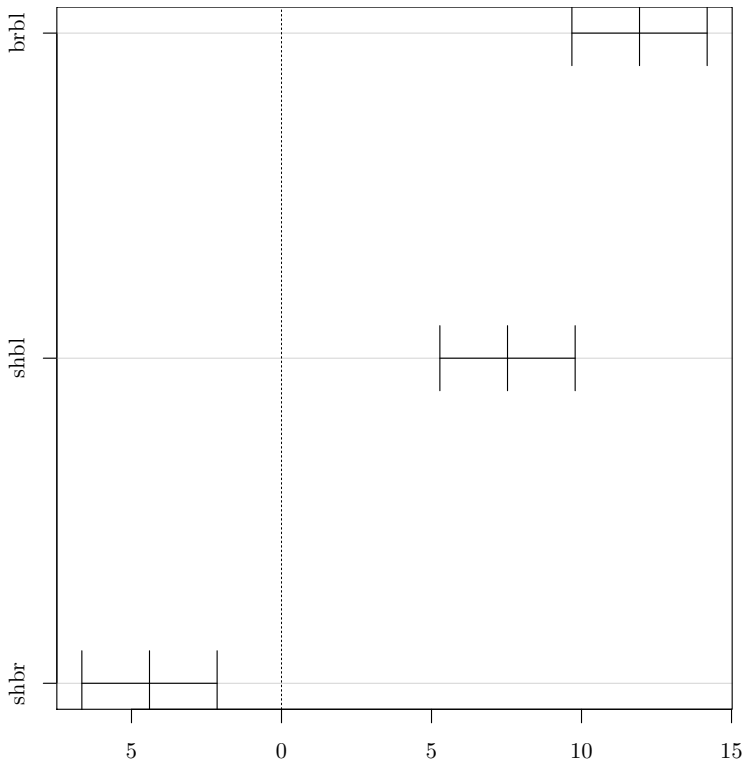


Рис. 26. Результаты *post hoc* теста Тьюки. Обе разницы между средними для роста блондинов и двух остальных групп лежат внутри доверительного интервала (справа от пунктирной линии)

Результаты обоих тестов, разумеется, похожи.

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). Можно попробовать его на наших искусственных данных:

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

Kruskal-Wallis rank sum test

data:  data$ROST by data$CVET
Kruskal-Wallis chi-squared = 63.9153, df = 2,
p-value = 1.321e-14
```

Результат аналогичен «классическому» ANOVA, а p-значение чуть больше, как и положено для непараметрического теста.

Задача про вес. Попробуйте самостоятельно выяснить, отличаются ли наши группы еще и по весу? Если да, то отличаются ли от остальных по-прежнему блондины?

Задача про зависимость роста и веса. А зависит ли вес от роста в наших искусственных данных? Если да, то *какая* эта зависимость?

* * *

Ответ к задаче про тест знаков. Достаточно написать:

```
> a <- c(1, 2, 3, 4, 5, 6, 7, 8, 9)
> b <- c(5, 5, 5, 5, 5, 5, 5, 5, 5)
> dif <- a-b
> pol.dif <- dif[dif > 0]
> binom.test(length(pol.dif), length(dif))

Exact binomial test

data:  length(pol.dif) and length(dif)
number of successes = 4, number of trials = 9, p-value = 1
alternative hypothesis: true probability of success is not equal to 0.5
95 percent confidence interval:
 0.1369957 0.7879915
sample estimates:
probability of success
             0.4444444
```

То есть отличий от половины нет, и тест знаков считает такие выборки одинаковыми! Чего и можно было, в общем, ожидать — ведь мы уже писали, что проверяются не выборки как таковые, а лишь их центральные значения.

Ответ к задаче про озон. Чтобы узнать, нормально ли распределение, применим созданную в предыдущей главе функцию `normality3()`:

```
> ozone.month <- airquality[,c("Ozone","Month")]
> ozone.month.list <- unstack(ozone.month)
> normality3(ozone.month.list)
$`5`
[1] "NOT NORMAL"

$`6`
[1] "NORMAL"

$`7`
[1] "NORMAL"

$`8`
[1] "NORMAL"

$`9`
[1] "NOT NORMAL"
```

Здесь мы применили функцию `unstack()`, которая «разобрала» данные по месяцам. Разумеется, это можно было сделать и при помощи уже известных вам функций:

```
> normality3(ozone.month[ozone.month$Month==5,][1])
$Ozone
[1] "NOT NORMAL"

> normality3(ozone.month[ozone.month$Month==6,][1])
$Ozone
[1] "NORMAL"
```

и т. д.

Мы использовали здесь две пары квадратных скобок, для того чтобы данные не превратились в вектор, а остались таблицей данных, ведь наша функция `normality3()` «заточена» именно под таблицы данных. Если же эти две пары квадратных скобок вам не нравятся, можно поступить еще проще (хотя и муторнее) и вернуться к оригинальному `shapiro.test()`:

```
> shapiro.test(ozone.month[ozone.month$Month==5,"Ozone"])

Shapiro-Wilk normality test

data:  ozone.month[ozone.month$Month == 5, "Ozone"]
W = 0.714, p-value = 8.294e-06
```

(Загляните в предыдущую главу, чтобы посмотреть, какая в этом тесте альтернативная гипотеза.)

Как видите, в R одну и ту же задачу всегда можно решить множеством способов. И это прекрасно!

Ответ к задаче про оценки двух классов. 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 в предыдущей главе 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 A результаты первой и второй четвертей похожи, а вот класс B учился в первой четверти лучше A. 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

Warning messages:
...

> 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

Warning message:
In wilcox.test.default(klassy$B1, klassy$A1, alt = "greater") :
  cannot compute exact p-value with ties
```

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 B лучше A, а не просто отличаются ли их результаты.

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

Ответ к задаче про кассиров. Попробуем сначала выяснить, можно ли использовать параметрические методы:

```
> kass <- read.table("data/kass.txt", h=TRUE)
> head(kass)
  KASS.1 KASS.2
1      3     12
2     12     12
3     13      9
4      5      6
5      4      2
6     11      9
```

```
> normality3(kass)
$KASS.1
[1] "NORMAL"
$KASS.2
[1] "NORMAL"
```

Отлично! Как насчет средних?

```
> (kass.m <- sapply(kass, mean))
  KASS.1   KASS.2
8.380952 7.809524
```

Похоже, что у первого кассира очереди побольше. Проверим это:

```
> with(kass, t.test(KASS.1, KASS.2))

Welch Two Sample t-test

data:  KASS.1 and KASS.2
t = 0.4358, df = 39.923, p-value = 0.6653
alternative hypothesis:
            true difference in means is not equal to 0
95 percent confidence interval:
 -2.078962  3.221819
sample estimates:
mean of x mean of y
 8.380952  7.809524
```

Увы, достоверной разницы нет. Попробуем отобразить это графически (рис. 27) и проверить, выполняется ли «правило трех сигм»:

```
> (kass.sd <- sapply(kass, sd))
  KASS.1   KASS.2
4.341384 4.154745
> library(gplots)
> barplot2(kass.m, plot.ci=TRUE,
+ ci.l=kass.m, ci.u=(kass.m + (3 * kass.sd)),
+ names.arg=c("Первый кассир","Второй кассир"),
+ ylab="Очередь")
```

Столбчатые диаграммы с разбросами — это распространенный, хотя и не самый эффективный, способ изображения данных. Для того чтобы нарисовать разброс на столбике, мы загрузили пакет `gplots` и вызвали оттуда функцию `barplot2()`. Кроме того, мы использовали прием

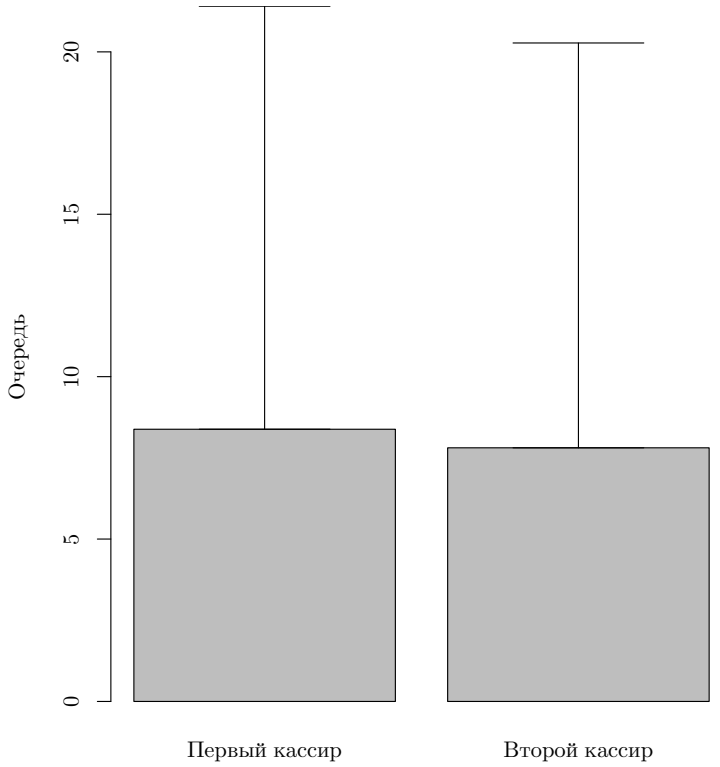


Рис. 27. Столбчатые диаграммы с разбросами в три стандартных отклонения отображают изменчивость очередей к двум кассирам в супермаркете

с внешними скобками, чтобы результат функции `sapply()` шел и на экран, и в объект `kass.sd`.

«Правило трех сигм» нас не подвело — разбросы шириной в три стандартных отклонения сильно перекрываются. Значит, выборки достоверно не отличаются.

Ответ к задаче про проращивание зараженных грибом семян. Загрузим данные, посмотрим на их структуру и применим тест хи-квадрат:

```
> pr <- read.table("data/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 correction

data:  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 correction

data:  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 correction

data:  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 от контроля, а третьего (CID63) — нет.

В ответе использован оператор `%in%`, который позволяет выбрать из того, что слева, то, что есть справа. Это сэкономило нам немного времени. Можно было написать и `table(pr[pr$CID==0 & pr$CID==63, ])`, где `&` означает логическое «и». Есть, разумеется, и другие способы сделать то же самое.

Нужно заметить, что поскольку мы три раза повторили сравнение с одной и той же группой (контролем), нужно пороговое p-значение уменьшить в три раза (это называется «*поправка Бонферрони* для множественных сравнений»). Кстати, для подобных поправок в R есть специальная функция, `p.adjust()`. В нашем случае первые два значения все равно останутся значимыми.

Ответ к задаче про вес. Да и нет:

```
> anova(lm(data$VES ~ data$CVET))
```

```
Analysis of Variance Table

Response: data$VES
          Df Sum Sq Mean Sq F value    Pr(>F)
data$CVET  2 1043.5  521.73  31.598 4.837e-11 ***
Residuals 87 1436.5   16.51
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1
> pairwise.t.test(data$VES, data$CVET)

Pairwise comparisons using t tests with pooled SD

data:  data$VES and data$CVET

   bl      br
br 1.5e-10 -
sh 0.15    7.5e-08

P value adjustment method: holm
```

Отличия между группами есть, но отличаются по весу как раз брюнеты, а не блондины.

Ответ к задаче о показе предметов. Поступим так же, как и в примере про программистов. Сначала загрузим данные и присоединим их, чтобы было легче работать с переменными:

```
> pokaz <- read.table("data/pokaz.txt")
> attach(pokaz)
```

Затем проверим модель:

```
> pokaz.logit <- glm(formula=V3 ~ V2, family=binomial)
> summary(pokaz.logit)

Call:
glm(formula = V3 ~ V2, family = binomial)

Deviance Residuals:
    Min       1Q   Median       3Q      Max
-2.4029  -0.8701   0.4299   0.7825   1.5197

Coefficients:
            Estimate Std. Error z value Pr(>|z|)
```

```
(Intercept)  -1.6776     0.7923  -2.117  0.03423 *
V2            0.9015     0.2922   3.085  0.00203 **
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

(Dispersion parameter for binomial family taken to be 1)

    Null deviance: 62.687  on 49  degrees of freedom
Residual deviance: 49.738  on 48  degrees of freedom
AIC: 53.738

Number of Fisher Scoring iterations: 4
```

(Вызывая переменные, мы учли тот факт, что R присваивает колонкам данных без заголовков имена `V1`, `V2`, `V3` и т. д.)

Модель значима! Так что наша идея, высказанная в первой главе (о том, что человек обучается в ходе попыток), имеет под собой определенные основания. А вот так можно построить график для этого эксперимента (рис. 28):

```
> popytki <- seq(1, 5, 0.081) # ровно 50 значений от 1 до 5
> pokaz.p <- predict(pokaz.logit, list(V2=popytki),
+ type="response")
> plot(V3 ~ jitter(V2, amount=.1))
> lines(popytki, pokaz.p)
> detach(pokaz)
```

Мы использовали `predict()`, чтобы рассчитать вероятности успеха в по-разному «расположенных» попытках, а на графике, чтобы точки не накладывались друг на друга, добавили немного случайных отклонений при помощи функции `jitter()`. Не забудьте отсоединить данные командой `detach()`.

Ответ к задаче про зависимость веса от роста. Есть значимая корреляция, но зависимость слабая (рис. 29):

```
> cor.test(data$VES, data$ROST)

Pearson’s product-moment correlation

data:  data$VES and data$ROST
t = 4.8868, df = 88, p-value = 4.565e-06
alternative hypothesis: true correlation is not equal to 0
95 percent confidence interval:
```

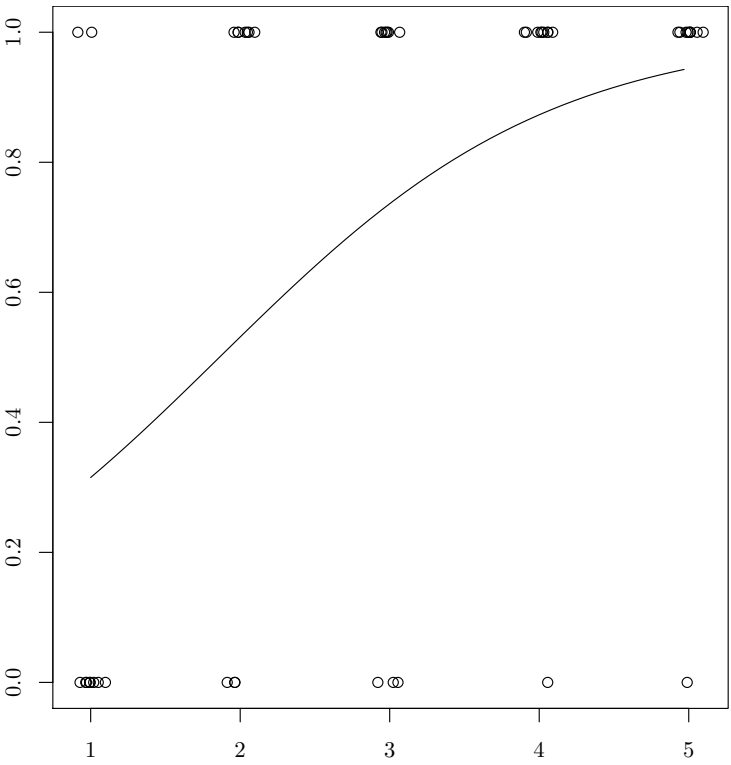


Рис. 28. Графическое изображение логистической модели (данные о показе предметов)

```
 0.2818862 0.6106708
sample estimates:
      cor
0.4620071

> ves.rost <- lm(data$VES ~ data$ROST)
> summary(ves.rost)

Call:
lm(formula = data$VES ~ data$ROST)

Residuals:
     Min       1Q   Median       3Q      Max
-10.2435  -3.4745  -0.5642   2.9358  12.3203

Coefficients:
```

```
            Estimate Std. Error t value Pr(>|t|)
(Intercept) 10.69580   13.37252   0.800    0.426
data$ROST    0.39742    0.08132   4.887 4.57e-06 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 4.708 on 88 degrees of freedom
Multiple R-squared: 0.2135,Adjusted R-squared: 0.2045
F-statistic: 23.88 on 1 and 88 DF,  p-value: 4.565e-06

> plot(data$VES ~ data$ROST)
> abline(ves.rost)
```

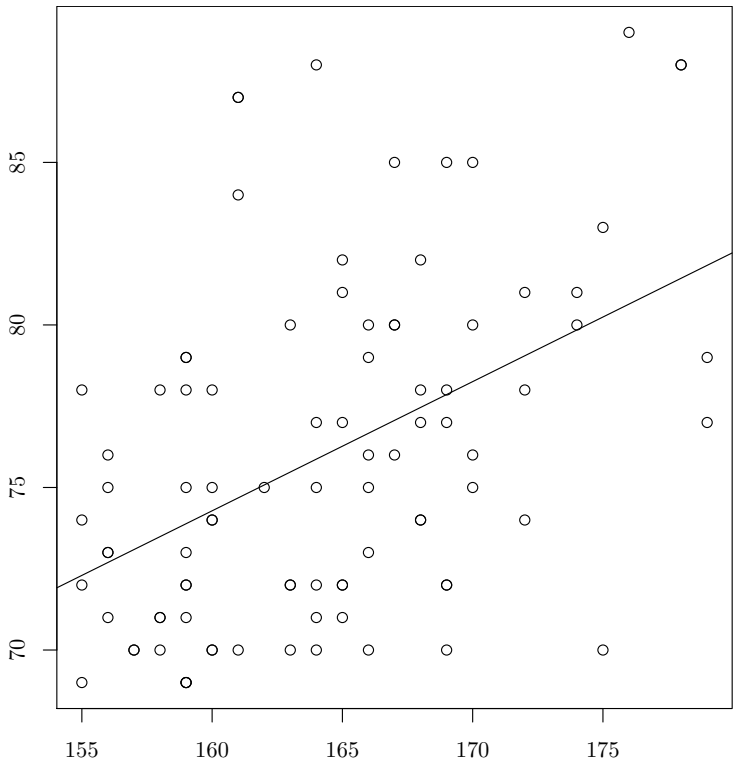


Рис. 29. Слабая зависимость

Вывод о том, что зависимость слабая, сделан на основании низкого R^2 и незначимого первого коэффициента. Кроме того, очень много остатков (больше 70% длины интервала между минимумом и максимумом) находится снаружи от первого и третьего квартилей. Да и график показывает серьезный разброс.

Глава 6

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 этими группами нерезкая, но для удобства мы станем рассматривать их именно в этом порядке.

6.1. 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.

6.1.1. 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` (рис. 30):

```
> library(rgl)
> plot3d(iris$Sepal.Length, iris$Sepal.Width, iris$Petal.Length,
+ col=as.numeric(iris$Species), size=3)
```

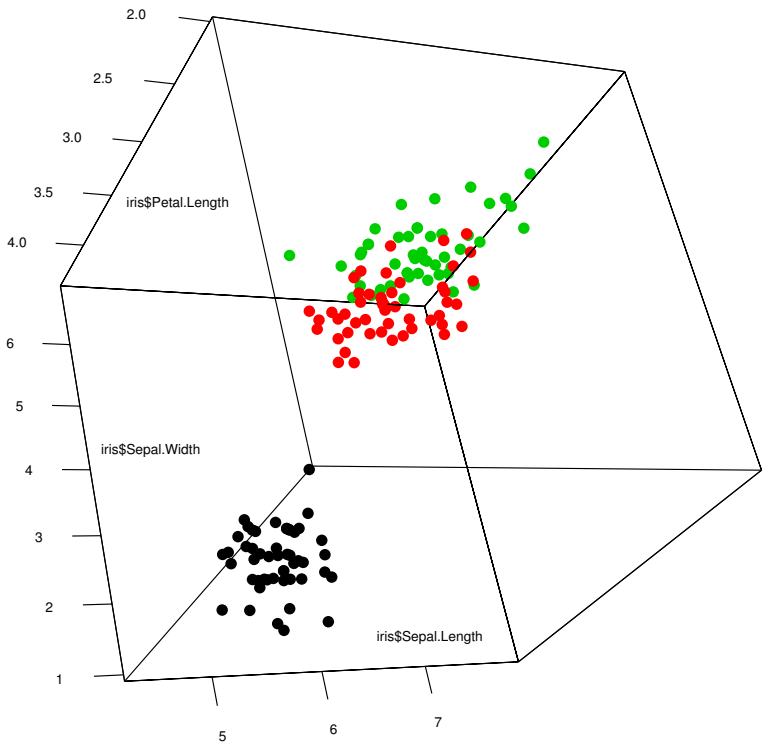


Рис. 30. Пример трехмерного графика `RGL`

Размер появившегося окна и проекцию можно (и нужно) менять при помощи мышки.

Сразу видно, что один из видов (*Iris setosa*) хорошо отличается от двух других по признаку длины лепестков (`Petal.Length`). Кста-

ти, в том, что мы изобразили именно **4** признака, не было оговорки, ведь вид ириса — тоже признак, закодированный в данном случае цветом. Можно обойтись и без `RGL`, тогда вам может потребоваться пакет `scatterplot3d`, содержащий одноименную функцию. Но, вообще говоря, трехмерными графиками лучше не злоупотреблять — очень они 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)
```

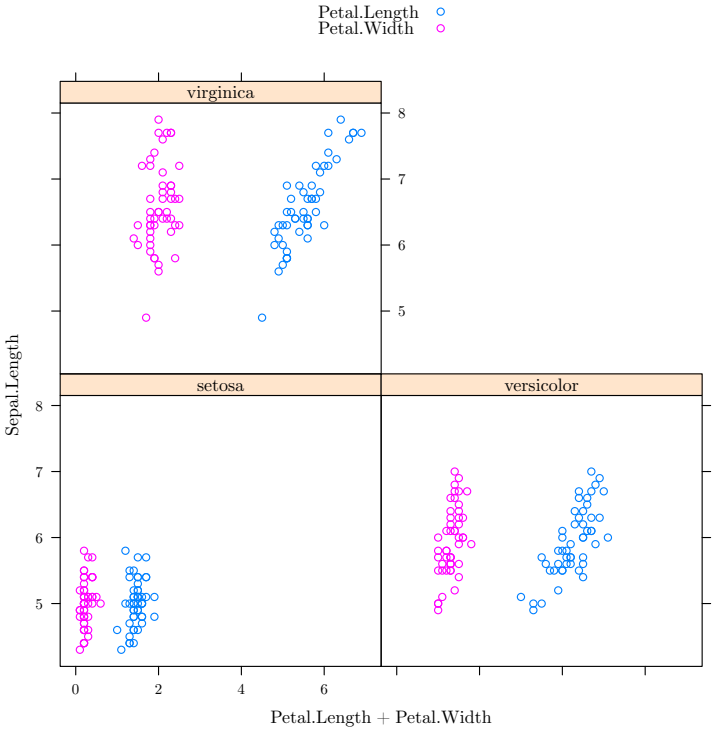


Рис. 31. Пример Trellis-графика

Получилась картинка зависимости длины чашелистиков как от длины, так и от ширины лепестков для каждого из трех видов.

Можно обойтись и без `lattice`. Несколько подобных графиков доступны и в базовой поставке R. Например, `coplot()` очень похож на `xyplot()` (рис. 32):

```
> coplot(dolja ~ javka | kand, data=vybory2)
```

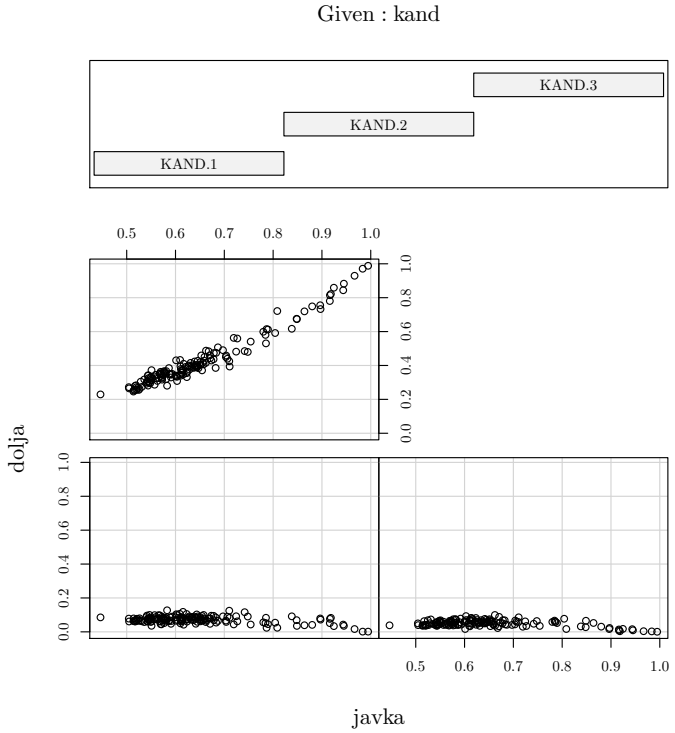


Рис. 32. Данные о выборах из раздела о регрессионном анализе, представленные с помощью функции `coplot()`

Мы визуализировали данные о выборах из раздела про регрессионный анализ. Эти данные тоже можно назвать многомерными, ведь каждый кандидат, по сути,— это отдельная переменная.

Матричный график рисуется при помощи функции `pairs()` (рис. 33):

```
> pairs(iris[1:4], pch=21, bg=c("red", "green3", "blue")
+ [unclass(iris$Species)])
```

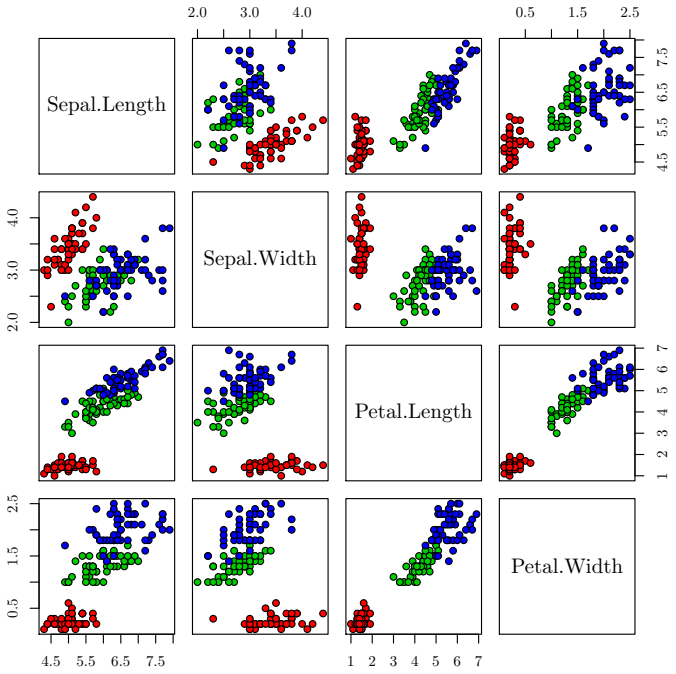


Рис. 33. Матричный график

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.

6.1.2. Пиктограммы

Для визуализации многомерных данных можно использовать разнообразные графики-пиктограммы, где каждый элемент представляет один объект наблюдений, а его параметры характеризуют значения признаков объекта.

Один из классических графиков-пиктограмм — звезды (рис. 34):

```
> stars(mtcars[1:9,1:7], cex=1.2)
```

Здесь каждая пиктограмма — это один тип автомобиля, а длины лучей соответствуют значениям разных характеристик этих машин. Легко

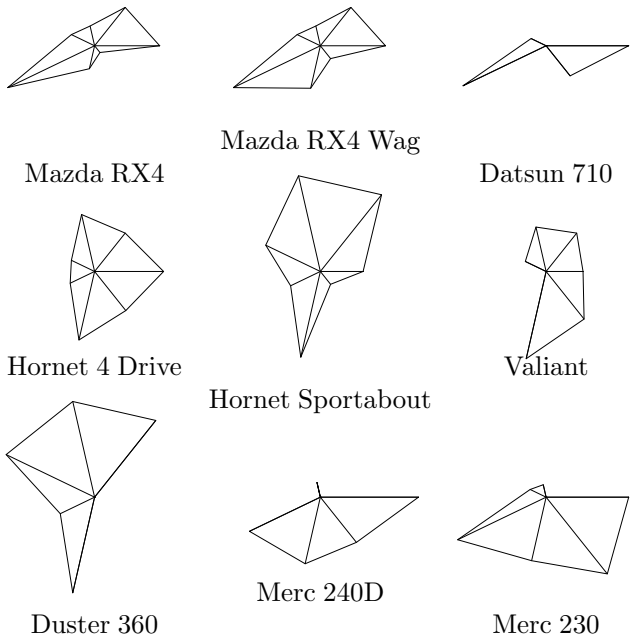


Рис. 34. Звезды изображают разные марки автомобилей

видеть, например, обе «Мазды» схожи с друг с другом больше, чем с остальными марками.

А вот более экзотический график-пиктограмма, так называемые «лица Чернова». Идея графика состоит в том, что лица люди различают очень хорошо (рис. 35):

```
> library(TeachingDemos)
> faces(mtcars[1:9,1:7])
```

Здесь каждое лицо соответствует одному исследуемому объекту (типу автомобиля), а черты лица характеризуют значения признаков объекта. Сходство «Мазд» хорошо заметно и на этом графике.

«Идеологически» к пиктограммам близок график особого типа — *график параллельных координат*. Его можно построить при помощи функции `parcoord()` из пакета `MASS`. Чтобы понять, что он делает, лучше всего сразу посмотреть на результат на рис. 36 (для графика использованы данные об измерениях растений):

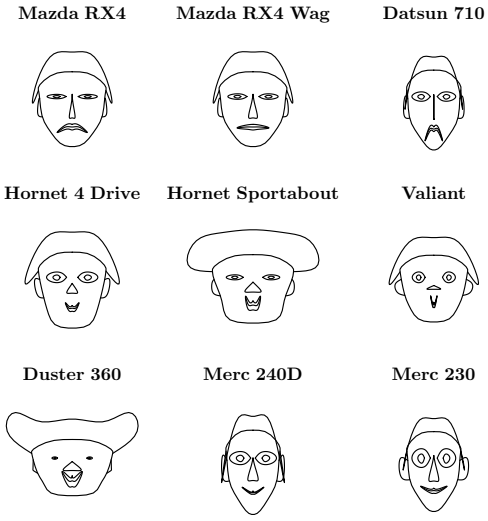


Рис. 35. Лица Чернова изображают разные марки автомобилей

```
> measurements <- read.table("data/eq-s.txt", h=T, sep=";")
> library(MASS)
> parcoord(measurements[,-1],
+ col=rep(rainbow(54), table(measurements[,1])))
```

С одной стороны, это очень простой график, нечто вроде многомерного боксплота (или гистограммы). Каждый столбец данных (у нас их всего девять, но первый — это номер местообитания) представлен в виде одной оси, на которой равномерно отмечаются все возможные значения данного признака для каждой строчки данных (в нашем случае — для каждого отдельного растения). Потом для каждого растения эти точки соединяются линиями. В дополнение мы раскрасили эти линии в цвета местообитаний (всего у нас было 54 местообитания). Однако на такой график можно смотреть очень долго и каждый раз находить там интересные детали, рассказывающие о структуре данных. Например, видно, что некоторые признаки измерены дискретно, даже невзирая на то, что они непрерывные (например, `DL.TR.V`); видно, что некоторые популяции попадают примерно в одно и то же место на каждом при-

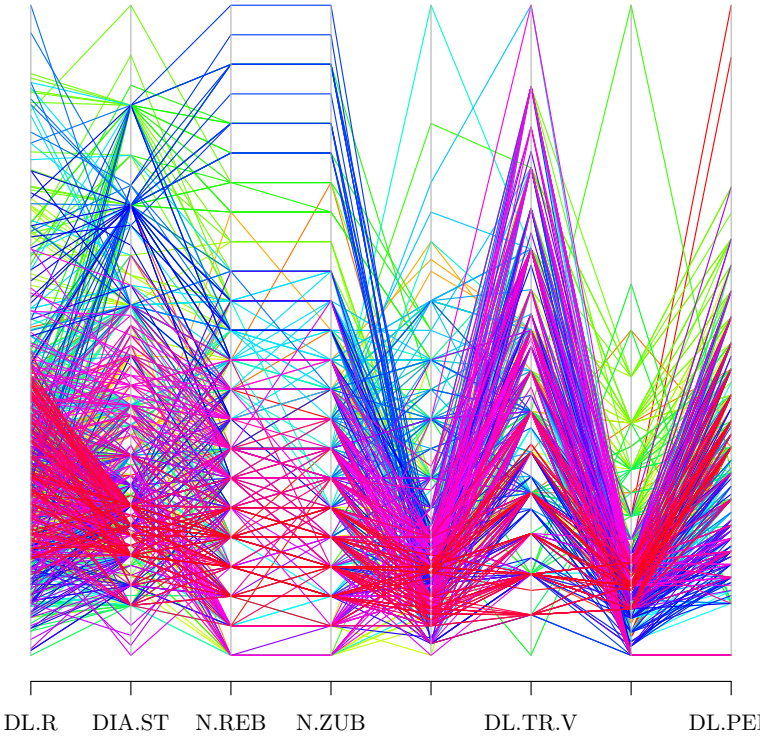


Рис. 36. График параллельных координат

знаке; видно, что у некоторых признаков (типа `DL.BAZ`) изменчивость смещена в сторону меньших значений, и многое другое. Именно поэтому график параллельных координат (как и «heatmap», описанный в разделе о корреляциях) почти обязательно находится в арсенале любой программы, занимающейся «data mining».

6.2. Тени многомерных облаков: анализ главных компонент

Перейдем теперь к методам сокращения размерности. Самый распространенный из них — это анализ главных компонент. Суть его в том, что наши 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 в 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,— как вы уже догадались, это будет вторая главная компонента. Эти две компоненты образуют плоскость, на которую и 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, сколько 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 «факторами», и это порождает некоторую путаницу с похожим на анализ главных 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 (рис. 37):

```
> 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. Вместо графика можно получить то же самое в текстовом виде, написав:

```
> summary(iris.pca)
Importance of components:
                          Comp.1    Comp.2     Comp.3      Comp.4
Standard deviation     1.7026571 0.9528572 0.38180950 0.143445939
Proportion of Variance 0.7296245 0.2285076 0.03668922 0.005178709
Cumulative Proportion  0.7296245 0.9581321 0.99482129 1.000000000
```

Отметьте, что первые две компоненты вместе объясняют почти 96% разброса (как говорят, их вклады, «proportions of variance» = 73% и 23% соответственно).

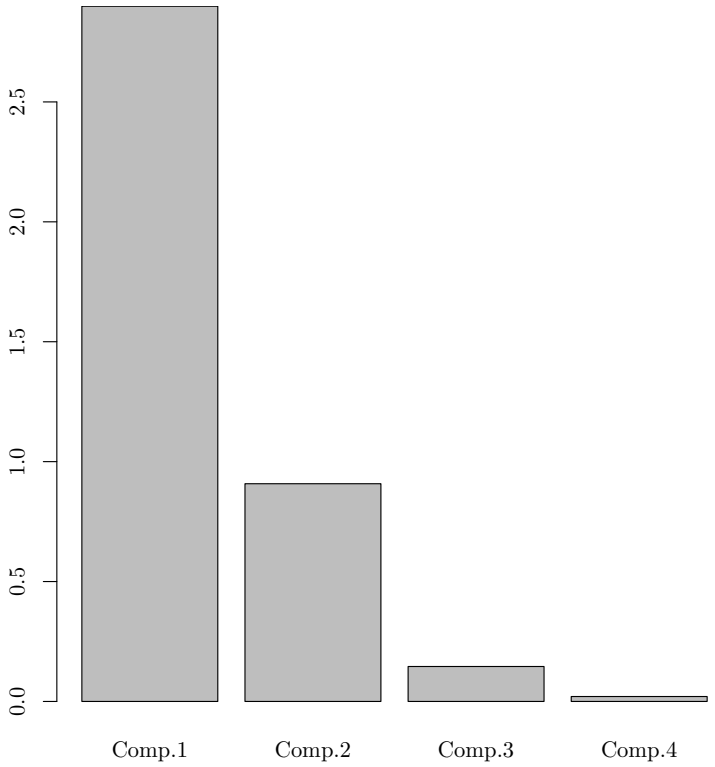


Рис. 37. График, отражающий вклады каждой компоненты в разброс данных

Перейдем к собственно визуализации (рис. 38):

```
> 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()` (рис. 39):

```
> biplot(iris.pca)
```

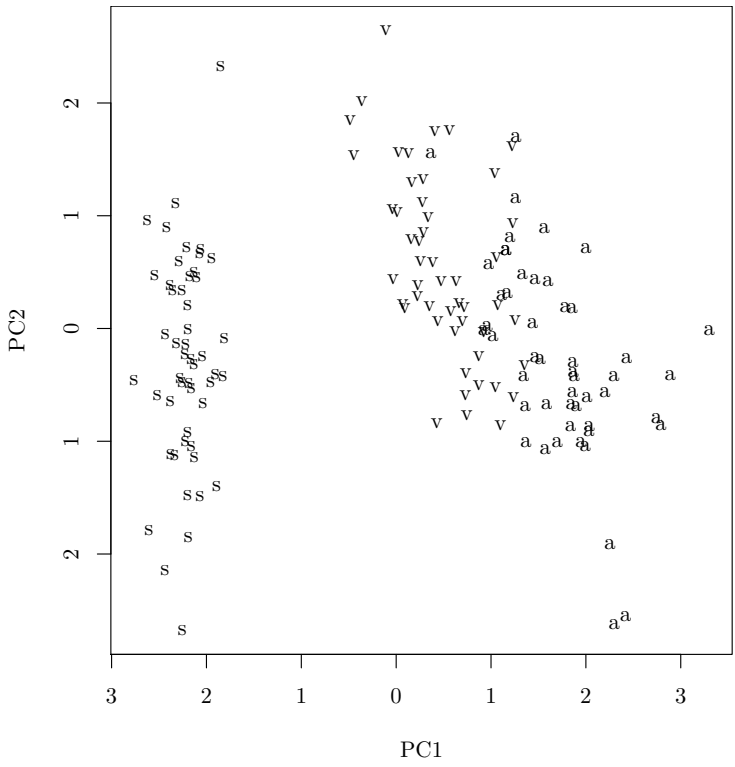


Рис. 38. Разнообразие ирисов на графике первых двух главных компонент

Парный график дает возможность понять, насколько силен вклад каждого из четырех исходных признаков в первые две компоненты. В данном случае хорошо видно, что признаки длины и ширины лепестков (в отличие от признаков длины и ширины чашелистиков) вносят гораздо больший вклад в первую компоненту, которая, собственно, и «различает» виды. К сожалению, графиком, выдаваемым при помощи `biplot()`, довольно трудно «управлять». Поэтому часто предпочтительнее функция `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
```

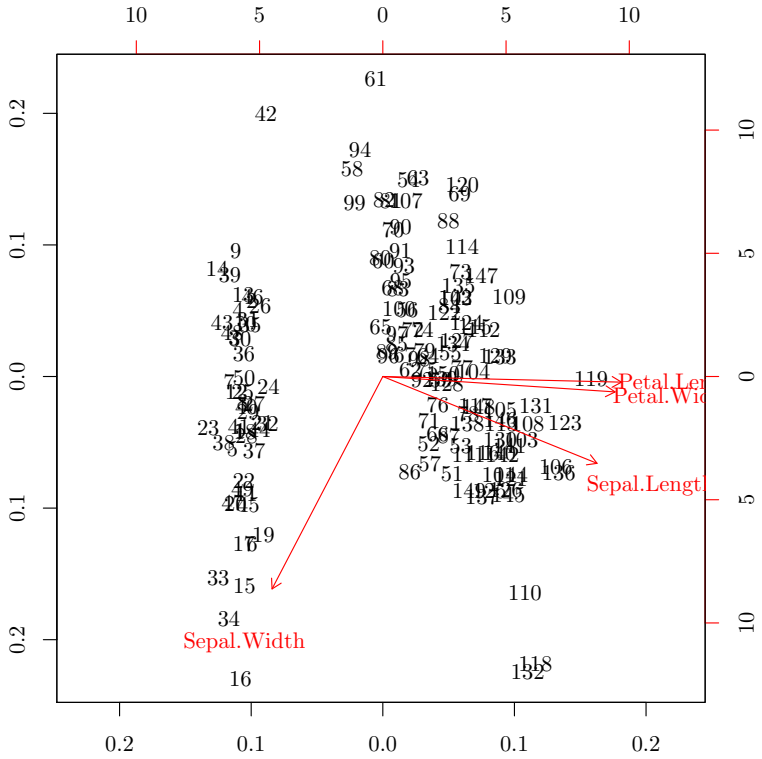


Рис. 39. Парный график показывает вклад каждого признака в первые две компоненты

```
                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
```

Собственно, показывает она то же самое, что и `biplot()`, но подробнее: 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` (рис. 40):

```
> library(ade4)
> iris.dudi <- dudi.pca(iris[,1:4], scannf=FALSE)
```

```
> s.class(iris.dudi$li, iris[,5])
```

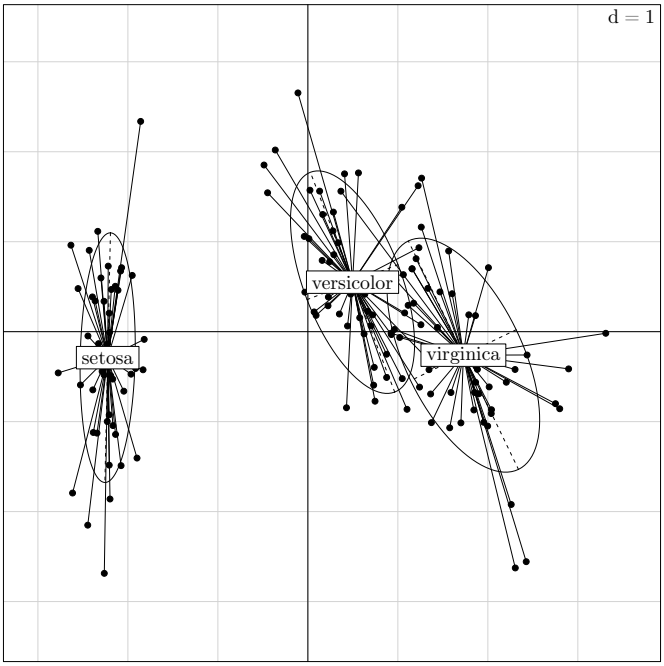


Рис. 40. Разнообразие ирисов на графике первых двух главных компонент (пакет `ade4`)

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.

6.3. Классификация без обучения, или Кластерный анализ

Другим способом снижения размерности является классификация без обучения (упорядочение, или ординация), проводимая на основании заранее 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 эвклидово и квартальное (манхеттеновское), применимые в основном к непрерывным переменным (объяснение см. на рис. 41). 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.

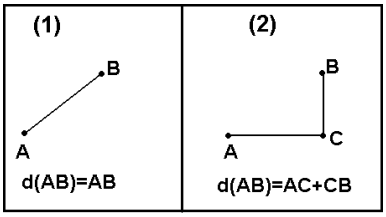


Рис. 41. Эвклидово (1) и манхеттеновское (2) расстояния

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»). Суть метода 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 это не 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 (рис. 42):

```
> 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 главных компонент, что неудивительно — ведь внутренняя структура данных (которую нам и надо найти в процессе «data mining») не изменилась. Кроме `cmdscale()`, советуем обратить внимание на непараметрический вариант этого метода, осуществляемый при помощи функции `isoMDS()`.

Результат многомерного шкалирования тоже можно украсить. Вот как мы попытались выделить получившиеся группы (рис. 43):

```
> library(KernSmooth)
> est <- bkde2D(iris.c[,1:2], bandwidth=c(.7,1.5))
> 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"))
> contour(est$x1, est$x2, est$fhat, add=TRUE,
+ drawlabels=FALSE, lty=3)
```

Функция `bkde2D()` из пакета `KernSmooth` предсказывает для каждой точки графика плотность распределения данных (в нашем случае — букв), а функция `contour()` рисует нечто вроде карты, где пиками являются места с максимальной плотностью букв. Чтобы график выглядел красиво, параметры `bkde2d()` были подобраны вручную.

Другой вариант работы с матрицей различий — это кластерный анализ. Существует множество его разновидностей. Наиболее употреби-

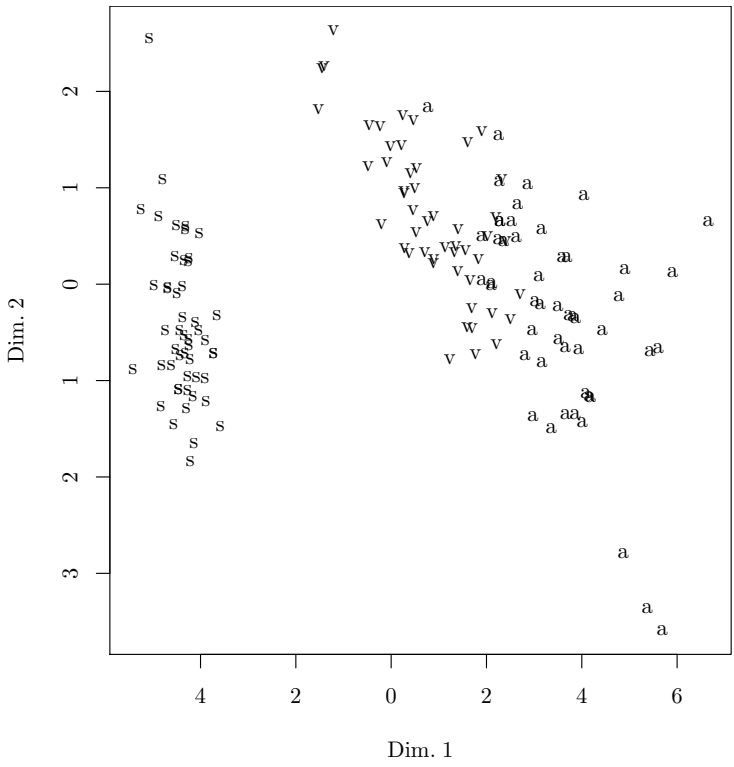


Рис. 42. Результат многомерного шкалирования данных об ирисах

тельными 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")
> plot(iriss.h, labels=abbreviate(iriss[,5],1,
+ method="both.sides"), main="")
```

Поскольку на выходе получается «дерево», мы взяли для его построения каждую пятую строку данных, иначе ветки сидели бы слишком плотно (это, кстати, недостаток иерархической кластеризации как метода визуализации данных). Метод Уорда (`ward`) 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*) разделяют-

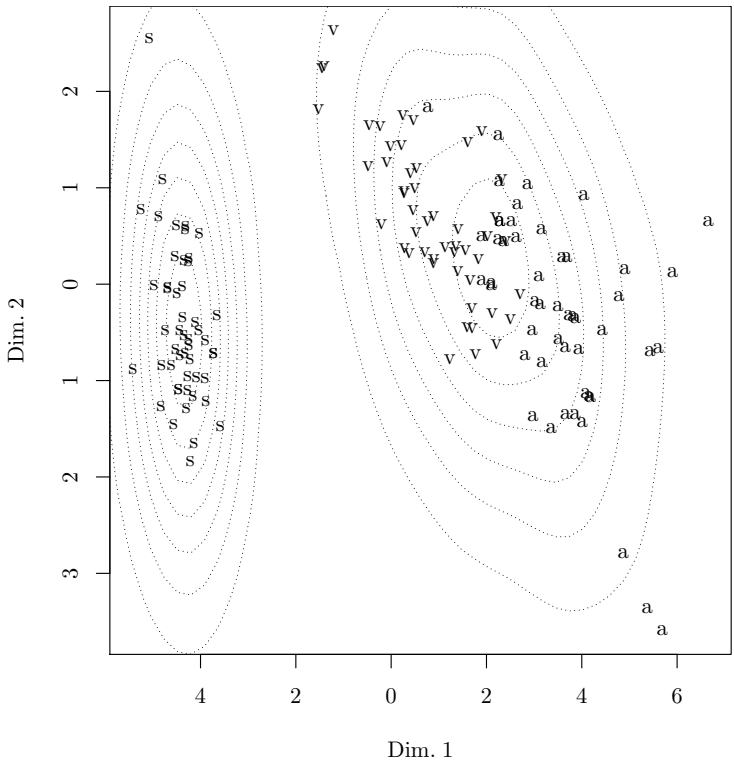


Рис. 43. Визуализация результата многомерного шкалирования при помощи функции плотности

ся на более низком уровне (`Height`≈ 12), то есть сходство между ними сильнее, чем каждого из них с третьим видом.

Задача. Попробуйте построить иерархическую классификацию сортов пива. Данные содержатся в файле `pivo-s.txt`, а расшифровка признаков — в файле `pivo-c.txt`. Сведения собирались в 2001 году в Москве (это надо учесть, если ваш опыт говорит о чем-то ином).

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)
```

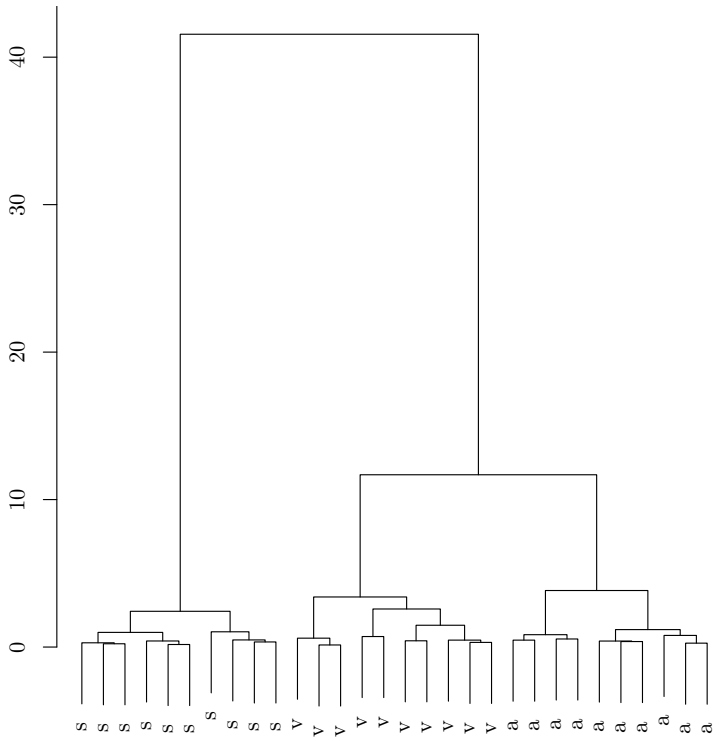


Рис. 44. Дендрограмма, отражающая иерархическую классификацию ирисов

```
> irisst <- t(iriss[,1:4])
> colnames(irisst) <- paste(abbreviate(iriss[,5], 3),
+ colnames(irisst))
> iriss.pv <- pvclust(irisst, method.dist="manhattan",
+ method.hclust="ward", nboot=100)
Bootstrap (r = 0.5)... Done.
Bootstrap (r = 0.5)... Done.
Bootstrap (r = 0.5)... Done.
Bootstrap (r = 0.75)... Done.
Bootstrap (r = 0.75)... Done.
...
> plot(iriss.pv, col.pv=c(1,0,0), main="")
```

Над каждым узлом печатаются p-значения (au), связанные с устойчивостью кластеров в процессе репликации исходных данных. Значения, близкие к 100, считаются «хорошими». В нашем случае мы видим как раз «хорошую» устойчивость получившихся основных трех класте-

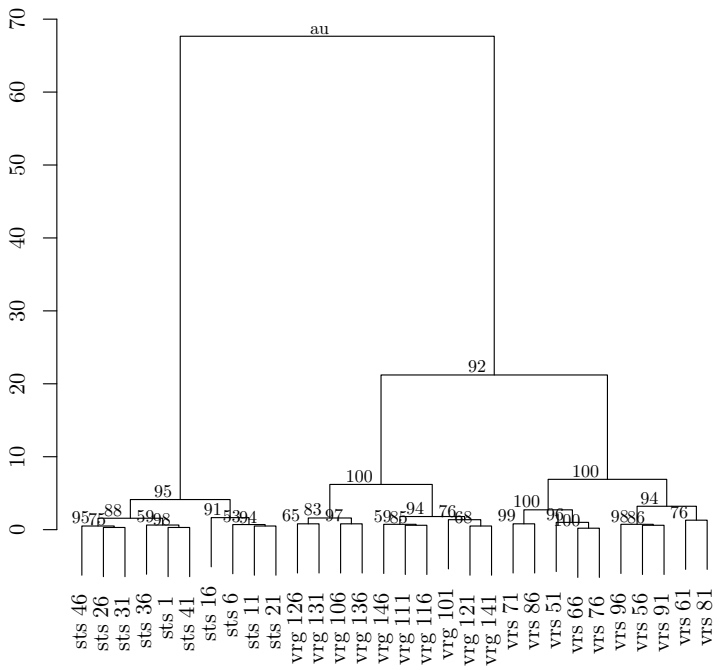


Рис. 45. Дендрограмма со значениями поддержки кластеров (пакет `pvclust`)

ров (разных видов ирисов) и неплохую устойчивость кластера, состоящего из двух видов на «v».

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 отличающихся видов хвощей. Попробуем проверить, сможет ли такой метод распознать эти виды, зная лишь признаки. Используем функцию `kmeans()`:

```
> eq <- read.table("data/eq.txt", h=TRUE)
> eq.k <- kmeans(eq[,-1], 2)
> table(eq.k$cluster, eq$SPECIES)

    arvense fluviatile
  1      37          5
  2       1         41
```

Получилось! Ошибка классификации довольно низкая. Интересно будет потом посмотреть, на каких именно растениях метод ошибся и почему. Отметьте, кстати, что функция `kmeans()` (как и следующие две) берет на входе не матрицу расстояний, а оригинальные данные.

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` (рис. 46):

```
> 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 каждой строчки 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 принадлежит к первому кластеру, тяготеет и к третьему. Недостатком этого метода является 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.

* * *

Как вы уже поняли, 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, которые могут работать с таблицами сопряженности? Оказывается, такие не просто существуют, но широко используются в разных областях, например в экологии.

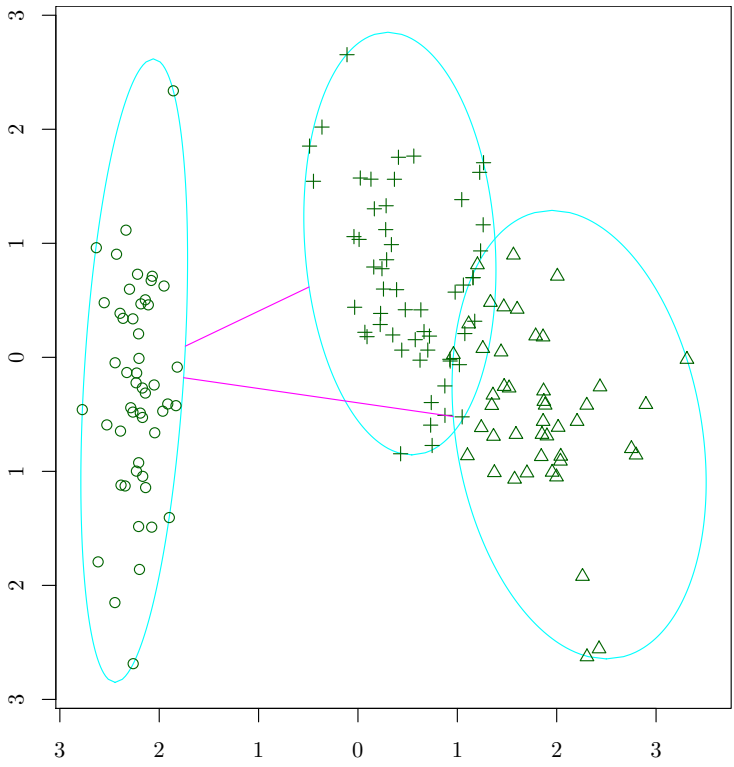


Рис. 46. Результат кластеризации функцией `fanny()`

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()` (рис. 47):

```
> 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))
```

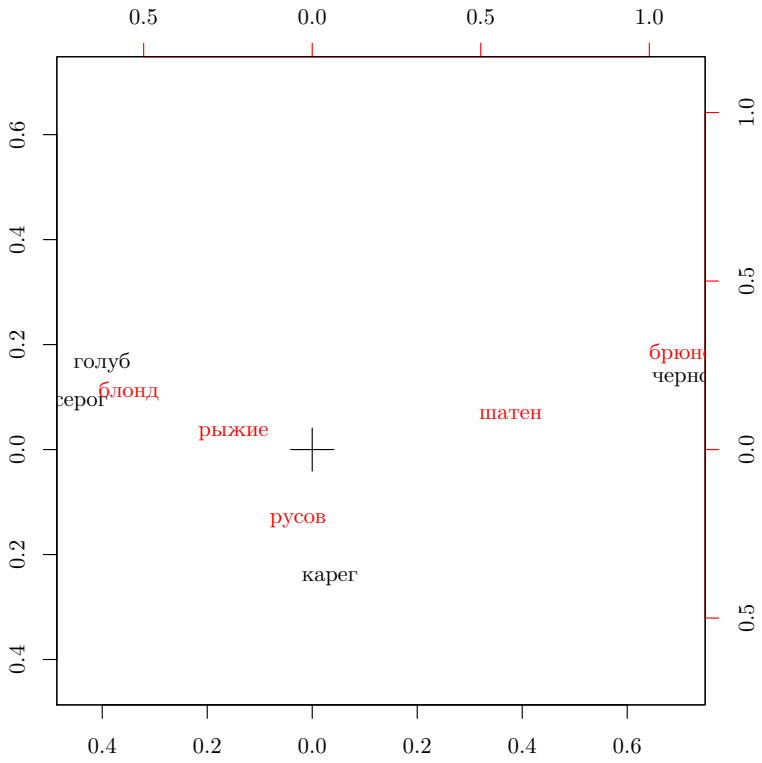


Рис. 47. График взаимосвязей, полученный из таблицы сопряженности

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 две 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, например, данные по 10 озерам, где указаны их названия и то, какие виды рыб в 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, прозрачность воды, придонную температуру летом), то можно будет визуализовать

сразу три набора признаков! Пакет `vegan` содержит несколько таких функций, например `cca()` и `decorana()`.

6.4. Классификация с обучением, или Дискриминантный анализ

Перейдем теперь к методам, которые лишь частично могут называться «визуализацией». В зарубежной литературе именно их принято 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 (тренировочной 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. Можно более точно подсчитать ошибки классификации. Для этого мы написали функцию `misclass()`:

```
> misclass <- function(pred, obs) {
+ tbl <- table(pred, obs)
+ sum <- colSums(tbl)
+ dia <- diag(tbl)
+ msc <- (sum - dia)/sum * 100
+ m.m <- mean(msc)
+ cat("Classification table:", "\n")
+ print(tbl)
+ cat("Misclassification errors:", "\n")
+ print(round(msc, 1))
+ }
> misclass(iris.ldap, iris.unknown[,5])
Classification table:
            obs
pred         setosa versicolor virginica
  setosa         40          0         0
  versicolor      0         40         7
  virginica       0          0        33
Misclassification errors:
    setosa versicolor  virginica
       0.0        0.0       17.5
```

Как видим, ошибка классификации — 17.5%.

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) (см. главу про анализ связей), и само значение «Wilks», поскольку это — «likelihood 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 (рис. 48):

```
> 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"))
```

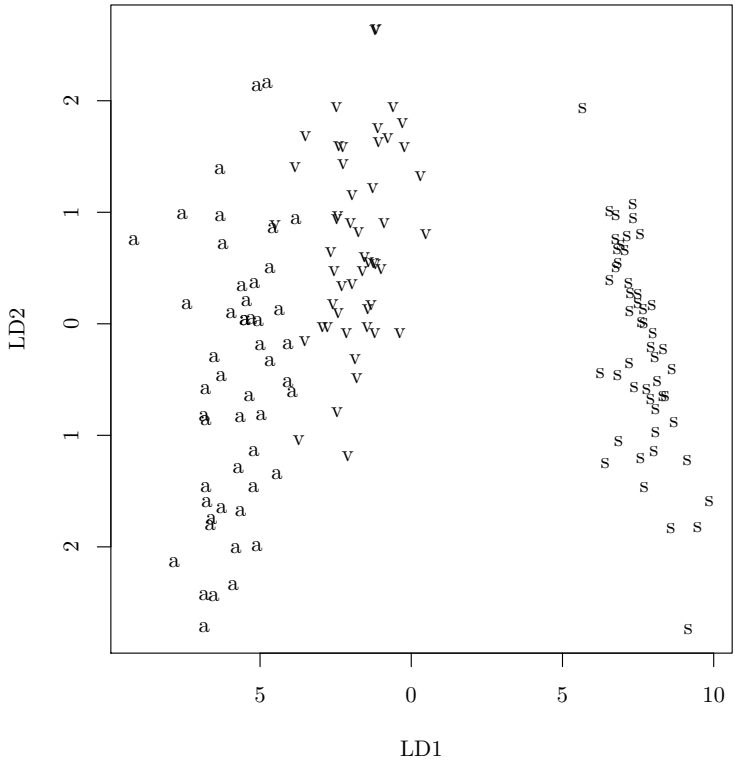


Рис. 48. Графическое представление результатов дискриминантного анализа

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 (рис. 49):

```
> library(tree)
> iris.tree <- tree(iris[,5] ~ ., iris[,-5])
> plot(iris.tree)
> text(iris.tree)
```

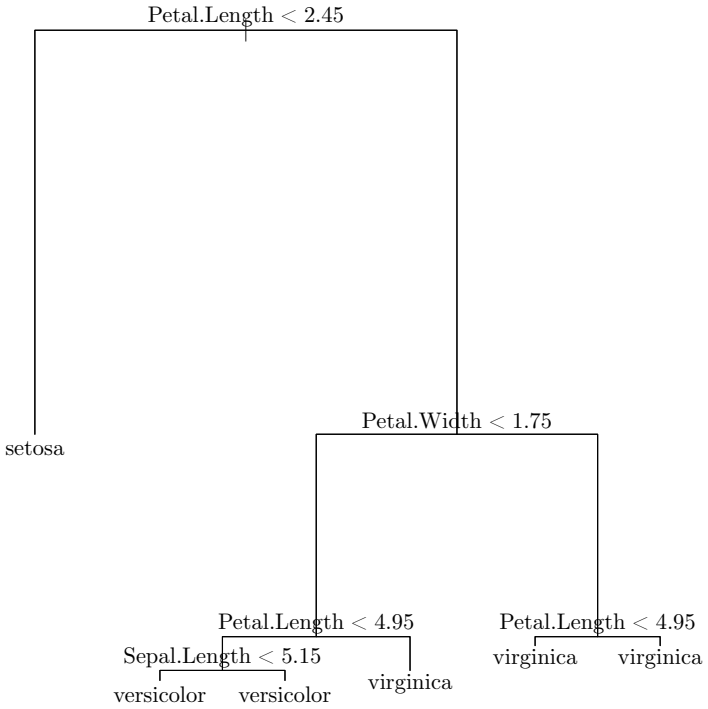


Рис. 49. Дерево классификации для данных об ирисах

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` относятся 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`.

Задача. Попробуйте выяснить, какие именно значения признаков различают два вида растений, которые мы использовали как пример к функции `kmeans()` (данные `eq.txt`, расшифровка признаков — в файле `eq-c.txt`).

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
  setosa         40          0         0
  versicolor      0         39         9
  virginica       0          1        31
```

Заметна значительно более высокая эффективность этого метода по 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 (рис. 50):

```
> 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».

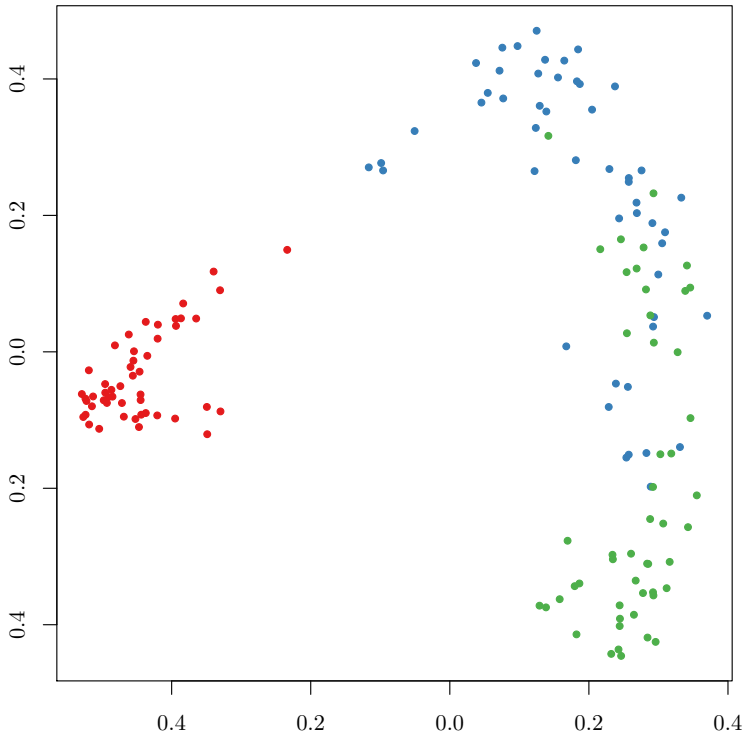


Рис. 50. Визуализация данных при помощи «Random Forest»

```
> 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.

В заключение приведем краткий обзор применимости основных из обсуждавшихся в этой главе методов многомерного анализа данных (рис. 51). При чтении диаграммы стоит помнить о том, что исходные данные можно трансформировать (например, создать из таблицы дан-

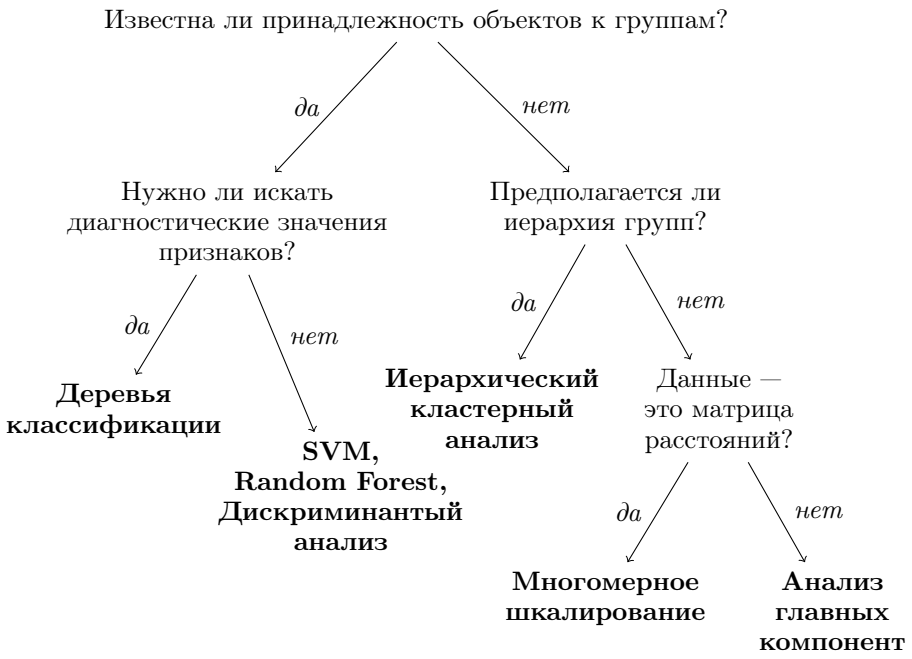


Рис. 51. Как выбрать подходящий метод многомерного анализа

ных матрицу расстояний), и тогда станут доступны и другие методы анализа.

* * *

Ответ к задаче про иерархическую классификацию пива. Чтобы построить иерархическую классификацию, надо вначале вычислить матрицу расстояний, а уже потом сгруппировать объекты и изобразить это при помощи дендрограммы. Сначала посмотрим на данные:

```
> pivo <- read.table("data/pivo-s.txt", h=T)
> head(pivo)
                    C.1 C.2 C.3 C.4 C.5 C.6 C.7 C.8 C.9 C.10
baltika.3             0   1   0   0   0   0   1   0   1    1
baltika.6             0   1   1   1   1   0   1   0   1    1
baltika.9             0   1   1   1   1   0   1   0   1    1
ochak.klassicheskoe   0   1   0   0   0   0   0   0   1    1
ochak.temnoe          0   1   1   1   1   0   0   0   1    1
sib.kor.prazdnichnoe  1   1   1   1   1   1   0   0   0    0
...
```

Данные, как видите, бинарные, и, значит, надо применять специфический метод вычисления расстояний. (Кстати говоря, бинарные данные часто используются там, где нужно классифицировать объекты, описанные качественными характеристиками). Воспользуемся функцией `vegdist()` из пакета `vegan`, которая умеет использовать популярный для бинарных данных метод Жаккара (`method="jaccard"`). Наверное, до этого момента вы не знали про такой метод и могли решить задачу несколько иначе, например, используя метод `binary` из стандартной функции `dist()`. Вот так мы получили дендрограмму, изображенную на рис. 52: 0

```
> library(vegan)
> pivo.d <- vegdist(pivo, "jaccard")
> plot(hclust(pivo.d, "ward"), main="", xlab="", sub="")
```

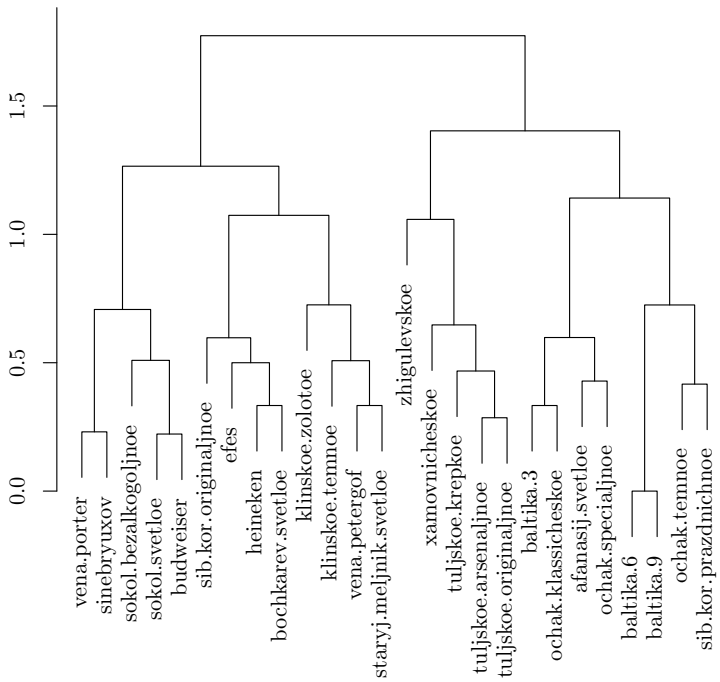


Рис. 52. Иерархическая классификация сортов пива

Получились две большие группы, одна с «Жигулевским» и «Балтикой», другая с «Клинским» и «Старым мельником».

Ответ к задаче про два вида растений. Попробуем построить дерево классификации (рис. 53):

```
> eq <- read.table("data/eq.txt", h=TRUE)
> eq.tree <- tree(eq[,1] ~ ., eq[,-1])
> plot(eq.tree); text(eq.tree)
```

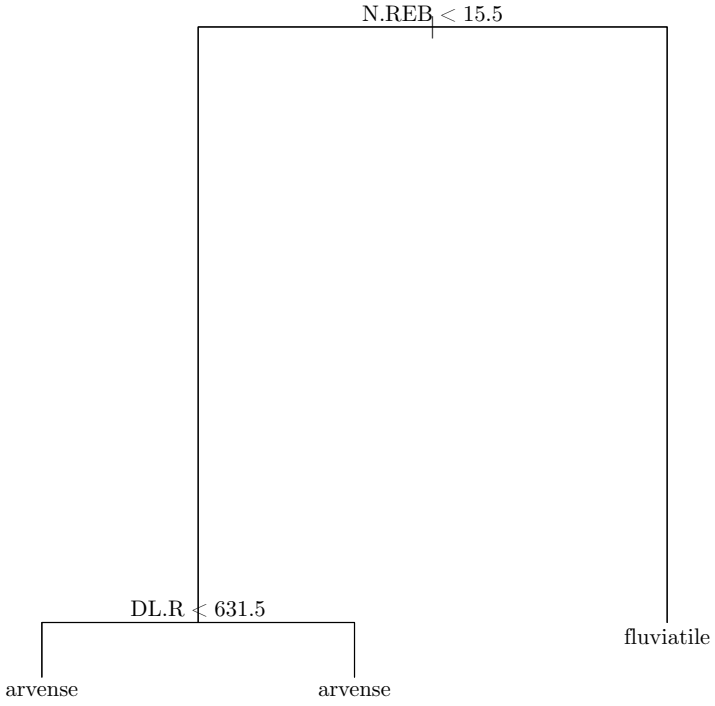


Рис. 53. Дерево классификации показывает, что два вида хвощей различаются по признаку `N.REB` (это количество ребер стебля)

Глава 7

Узнаем будущее: анализ временных рядов

В этой главе мы рассмотрим только самые основные принципы работы с временны́ми рядами. За более подробными сведениями рекомендуем обратиться к специальной литературе.

7.1. 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 страны, 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 *временны́м рядом* (time series).

В любом временном ряду можно выделить две компоненты:

1) неслучайную (детерминированную) компоненту;

2) 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.

7.2. Тренд и период колебаний

Если явление, результатом которого является изучаемый ряд, зависит от времени года (или времени суток, или дня недели, или иного фиксированного периода календаря), то из неслучайной компоненты может быть выделена еще одна компонента — *сезонные* колебания явления. 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 временно́м 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}, бо́льших 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 котором автокорреляция наиболее высокая, а следовательно, связь между текущим 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 между основным 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».

7.3. Построение временного ряда

Анализ временно́го 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 (степенно́е 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 временны́х рядов. Основным типом временны́х данных является «ts», который представляет собой временной ряд, состоящий из значений, разделенных одинаковыми интервалами времени. Временны́е 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 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$dates)
 Factor w/ 5 levels "2011-01-01","2011-01-02",..: 1 2 3 4 5
> dates.1 <- as.Date(dates.df$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 временны́м 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()` (рис. 54):

```
> plot(z,
+ plot.type="single", # поместить все ряды на одном графике
+ lty=1:3)            # типы линий временных рядов
```

Методы для анализа временны́х 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` в директории `data` записаны результаты длившихся трое суток непрерывных наблюдений над хищным растением росянкой. Листья этого растения постоянно открываются и закрываются «в надежде» поймать и затем переварить мелкое насекомое. Файл содержит результаты наблюдений над четвертым листом второго растения, поэтому он так называется. Состояние листа отмечали каж-

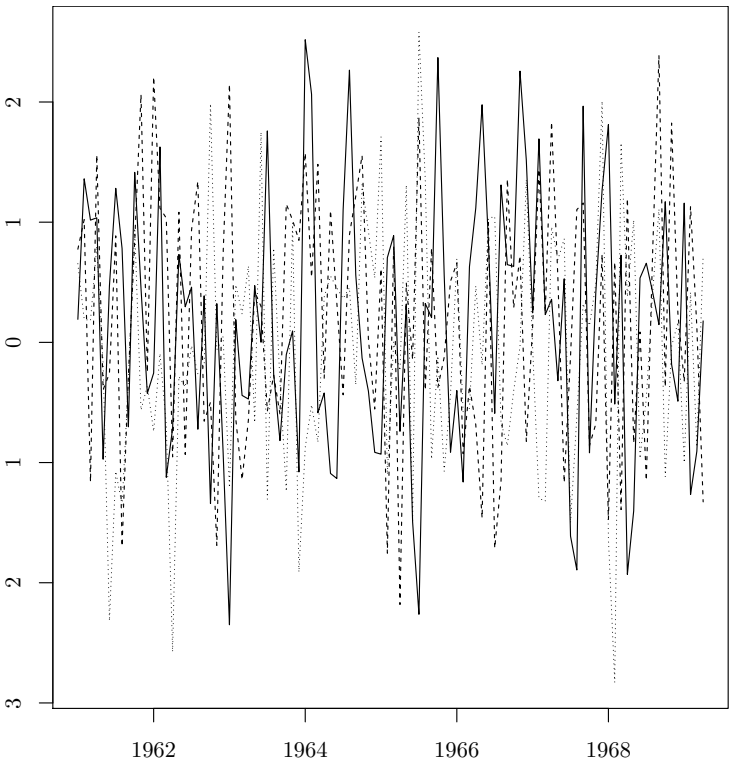


Рис. 54. График трех временны́х рядов с общими осями времени

дые 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("data/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=";")
```

Сразу стóит посмотреть, все ли в порядке:

```
> str(leaf)
'data.frame': 80 obs. of  7 variables:
 $ K.UVL  : int  2 1 1 2 1 1 1 1 1 1 ...
 $ FORM   : int  1 1 1 2 2 2 2 2 2 2 ...
 $ ZAGN   : int  2 2 2 3 3 3 2 2 2 2 ...
 $ OTOGN  : int  2 1 1 1 1 1 1 1 1 1 ...
 $ SVEZH  : int  1 0 1 1 1 1 1 1 1 1 ...
 $ POLUPER: int  0 1 0 0 0 0 0 0 0 0 ...
 $ OSTATKI: int  1 1 1 0 0 0 1 1 1 1 ...
> summary(leaf)
     K.UVL            FORM          ZAGN
 Min.   :1.000   Min.   :1.0   Min.   :1.0
 1st Qu.:1.000   1st Qu.:1.0   1st Qu.:2.0
 Median :1.000   Median :2.0   Median :2.0
 Mean   :1.325   Mean   :1.7   Mean   :2.5
 3rd Qu.:2.000   3rd Qu.:2.0   3rd Qu.:3.0
 Max.   :2.000   Max.   :2.0   Max.   :4.0

    OTOGN          SVEZH          POLUPER
 Min.   :0.00   Min.   :0.0000  Min.   :0.000
 1st Qu.:0.00   1st Qu.:0.0000  1st Qu.:1.000
 Median :1.00   Median :0.0000  Median :1.000
 Mean   :0.75   Mean   :0.1625  Mean   :0.925
 3rd Qu.:1.00   3rd Qu.:0.0000  3rd Qu.:1.000
 Max.   :2.00   Max.   :1.0000  Max.   :2.000

    OSTATKI
 Min.   :0.0000
 1st Qu.:0.0000
 Median :1.0000
 Mean   :0.6125
 3rd Qu.:1.0000
 Max.   :2.0000
```

Все загрузилось правильно, видимых ошибок и выбросов нет. Теперь 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 (рис. 55):

```
> (acf(forma, main=""))

Autocorrelations of series ‘forma’, by lag

0.0000 0.0278 0.0556 0.0833 0.1111 0.1389 0.1667 0.1944 0.2222
 1.000  0.614  0.287  0.079  0.074  0.068 -0.038 -0.085 -0.132
 0.2500 0.2778
 -0.137 -0.024
0.3056 0.3333 0.3611 0.3889 0.4167 0.4444 0.4722 0.5000 0.5278
 0.030  0.025 -0.082 -0.087  0.027  0.021 -0.043 -0.090 -0.137
```

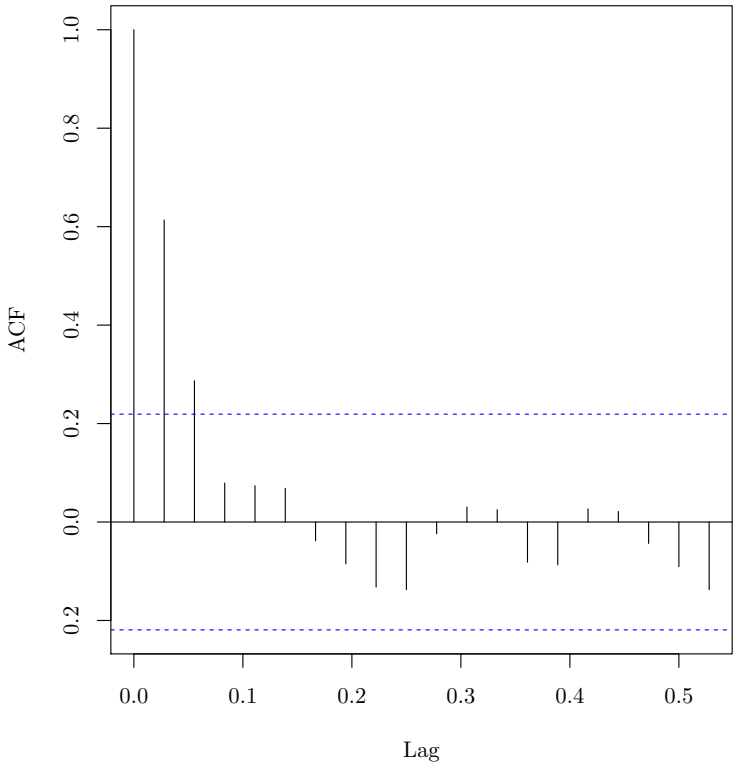


Рис. 55. График автокорреляций для состояния формы листа росянки

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 автокорреляции без лага или с очень маленьким лагом. По сути, это показывает, что в пределах 0.05 суток (у нас период наблюдений — сутки), то есть около 1 часа, следующее состояние листа будет таким же, как и текущее, а вот на бóльших интервалах таких предсказаний сделать нельзя. То, что волнообразный график пиков как бы затухает, говорит о том, что в наших данных возможен тренд. Проверим это (рис. 56):

```
> plot(stl(forma, s.window="periodic")$time.series, main="")
```

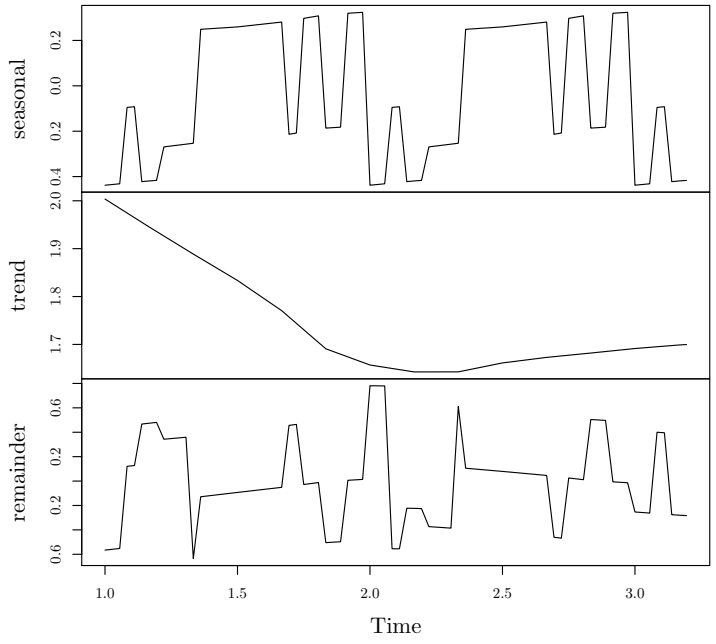


Рис. 56. График сезонной декомпозиции для состояния формы листа росянки. Возможный тренд изображен на среднем графике

Действительно, наблюдается тенденция к уменьшению значения формы с течением времени. Мы выяснили это при помощи функции `stl()`

(названа по имени метода, STL — «Seasonal Decomposition of Time Series by Loess»), которая вычленяет из временного ряда три компоненты: сезонную (в данном случае суточную), тренд и случайную, при помощи сглаживания данных методом LOESS.

Задача. Попробуйте понять, имеет ли другой признак того же листа (`K.UVL`, коэффициент увлажнения, отражающий степень «мокрости» листа) такую же периодичность и тренд.

7.4. 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:

1) данных о подключениях за декабрь 2004 года;

2) 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 временны́х ряда показывают экспоненциальный рост, поэтому выведем их в полулогарифмических координатах (рис. 57):

```
> 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)
```

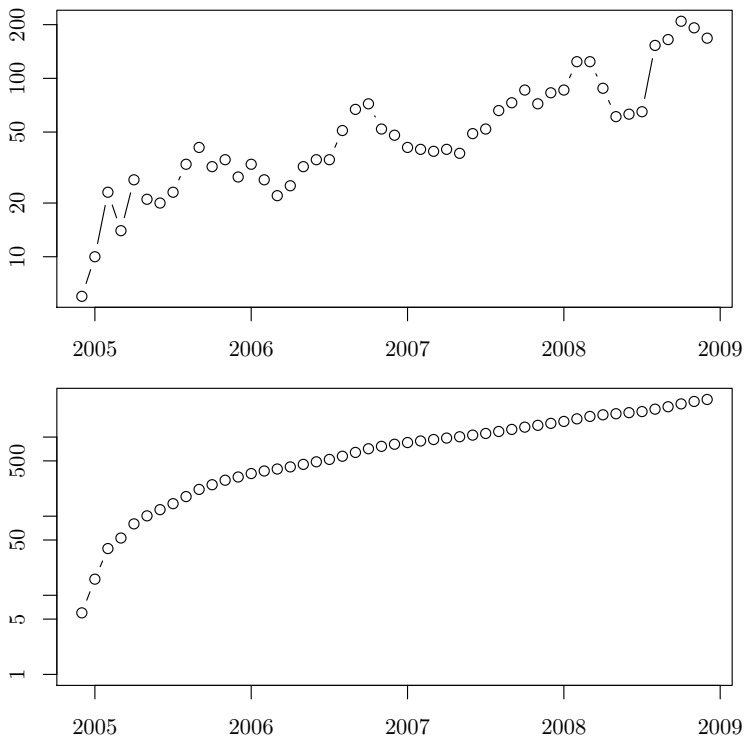


Рис. 57. Изменения помесячного и общего количества пользователей

Линейный рост временны́х 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 Integrated 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))
> model02 <- arima(cum.polzovateli, order=c(0,0,2))
> model03 <- arima(cum.polzovateli, order=c(0,0,3))
> model04 <- arima(cum.polzovateli, order=c(0,0,4))
> model05 <- arima(cum.polzovateli, order=c(0,0,5))
> model06 <- arima(cum.polzovateli, order=c(0,0,6))
> model07 <- arima(cum.polzovateli, order=c(0,0,7))
```

```
> model08 <- arima(cum.polzovateli, order=c(0,0,8))
> model09 <- arima(cum.polzovateli, order=c(0,0,9))
> model010 <- arima(cum.polzovateli, order=c(0,0,10))
> model011 <- arima(cum.polzovateli, order=c(0,0,11))
> model012 <- arima(cum.polzovateli, order=c(0,0,12))
> model013 <- arima(cum.polzovateli, order=c(0,0,13))
> model014 <- arima(cum.polzovateli, order=c(0,0,14))
```

(Можно было бы написать здесь цикл с оператором `for`, как мы делали для данных об отравлении в главе про двумерные данные, но нам было лень. Так что вот вам, дорогой читатель, **задача**: напишите такой цикл. Важная подсказка: нужно использовать функцию `assign()`.)

Теперь сравним модели. «Лучшая» модель будет соответствовать минимуму AIC (рис. 58):

```
> 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:

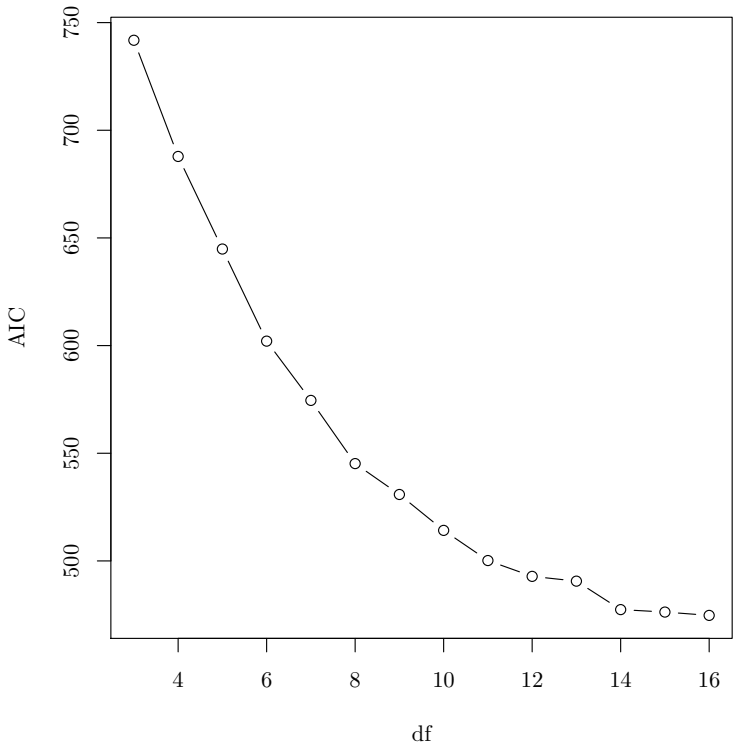


Рис. 58. Поиск наилучшей модели по минимуму значения AIC

```
> model2120 <- arima(cum.polzovateli, order=c(2,0,12))
> model2121 <- arima(cum.polzovateli, order=c(2,1,12))
> model2122 <- arima(cum.polzovateli, order=c(2,2,12))
> model2123 <- arima(cum.polzovateli, order=c(2,3,12))
> model2124 <- arima(cum.polzovateli, order=c(2,4,12))
> 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 (рис. 59):

```
> 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")
```

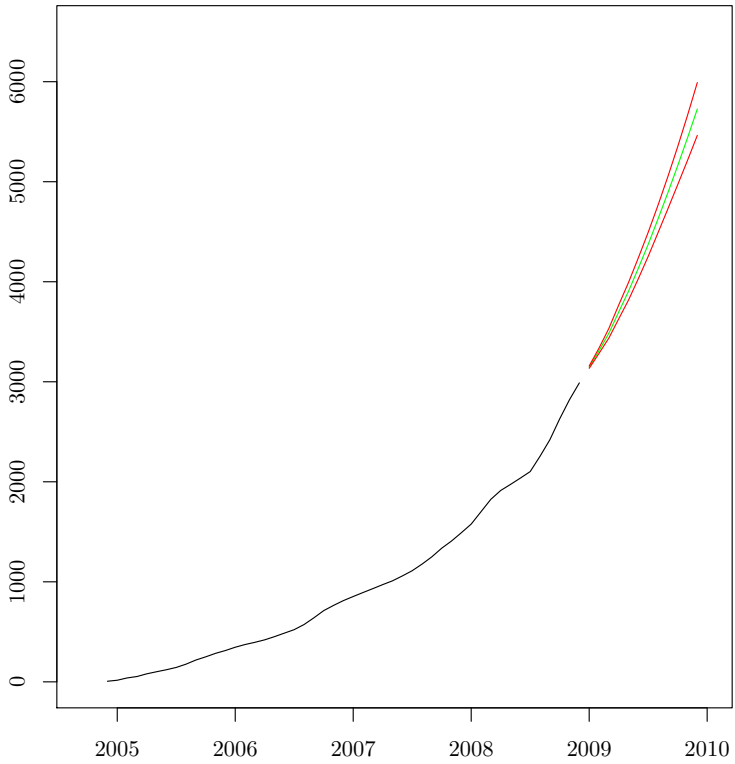


Рис. 59. Прогноз изменения общего числа абонентов на 2009 год

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  Dec
2009 3134 3283 3437 3628 3817 4031 4253 4490 4728 4969 5213 5463
```

Что же, теперь можно заняться обещанным в предисловии «предсказанием курса доллара на следующую неделю». Итак, **задача**: в файле `dollar.txt` содержатся значения курса доллара Центрального Банка с 1 июля по 9 августа 2011 года, всего за 11 недель. Попробуйте предсказать курс доллара на две недели вперед. Чтобы проверить эффективность предсказания, возьмите для модели данные по 26 июля, а предскажите последние две недели.

* * *

Ответ к задаче про лист росянки. Применим тот же подход, который мы использовали для признака формы листа:

```
> uvl <- ts(leaf$K.UVL, frequency=36)
> str(uvl)
 Time-Series [1:80] from 1 to 3.19: 2 1 1 2 1 1 1 1 1 1 ...
> acf(uvl)
> plot(stl(uvl, s.window="periodic")$time.series)
```

(Графики мы предлагаем вам построить **самостоятельно**.)

Видно, что на этот раз присутствует некая периодичность, с интервалом около 0.2 суток (примерно 5 часов). А вот выраженного тренда нет.

Ответ к задаче про цикл. Вот как можно его сделать:

```
 > for (m in 1:14)
 + {
 + assign(paste("model0",m,sep=""), arima(cum.polzovateli,
 + order=c(0,0,m)))
 + }
```

Функция `assign()` заменяет стрелочку (`<-`) и при этом позволяет программно менять название объекта — адреса присвоения.

Ответ к задаче о курсе доллара. Поступим ровно так же, как в примере о пользователях провайдера, то есть сначала превратим данные во временной ряд:

```
> dollar1 <- read.table("data/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 и 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 передавать 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 (рис. 60):

```
> 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))
```

Итак, наша модель не смогла предсказать резкое падение курса 29 июля, 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 (**проверьте** это самостоятельно) показывает, что значимых автокорреляций в нашем ряду мало.

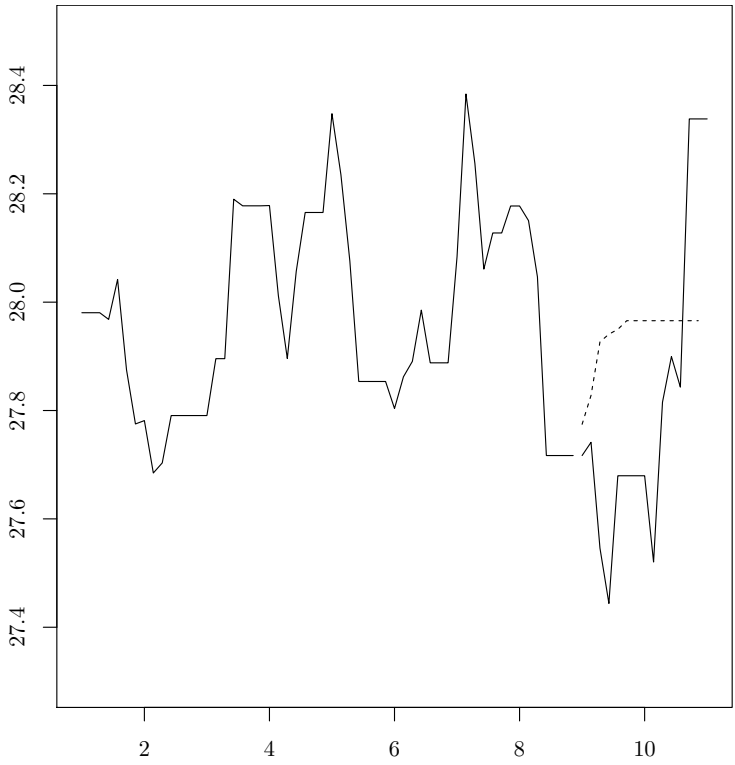


Рис. 60. Предсказание курса доллара (предсказанные значения обозначены пунктирной линией). По оси `x` отложены недели, прошедшие с 1 июня 2011 года

Глава 8

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?

8.1. Первичная обработка данных

Вначале надо выяснить несколько вопросов:

1. Какой тип у данных, каким способом (способами) они представлены?
2. Однородны ли данные — в каких единицах измерены показатели?
3. Параметрические или непараметрические у нас данные — можно ли предположить нормальное распределение данных?
4. 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: бóльшая часть таких ошибок — это результат неправильного оформления и/или ввода данных.

8.2. Окончательная обработка данных

Для того чтобы помочь читателю разобраться в многочисленных способах анализа, мы составили следующую таблицу (табл. 8.1). Она

устроена очень просто: надо ответить всего лишь на три вопроса, и способ обработки найдется в соответствующей ячейке. Главный вопрос — это тип данных (см. соответствующую главу). 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 сто́ит или не сто́ит применять парные методы сравнения (см. главу о двумерных данных).

Найдя в таблице нужный метод, загляните в указатель и перейдите на страницу с более подробным описанием функции. А можно просто ввести команду `help()`.

8.3. 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.

Можно пойти и еще дальше. Что, если создать файл, который будет содержать код R, перемешанный с текстовыми комментариями, и потом «скормить» этот файл R, так чтобы фрагменты кода заменились на результат их исполнения? Идея эта называется «literate programming» (грамотное программирование) и принадлежит Дональду Кнуту, создателю TeX. В случае R такая система используется для автоматической генерации отчетов — особенности, которая делает R поистине незаме-

Данные			Одна группа	Две группы: различия	Две группы: связи	Три и более групп: связи	Три и более групп: общая картина
Количественные	Параметрические	Независимые	summary()	t.test()	cor.test(..., method="pe")	oneway.test(), paiwise.t.test(), anova(), lm()	lda(), manova()
		Зависимые		t.test(..., paired = TRUE)		–	–
	Непараметрические	Независимые		wilcox.test()	cor.test(..., method="sp")	kruskal.test()	pca(), tree(), cor(), hclust(), isoMDS(), cmdscale()
		Зависимые		wilcox.test(..., paired = TRUE)		–	–
Номинальные или шкальные	Непараметрические	Независимые		chisq.test(), prop.test(), binom.test()	glm(..., "binomial")	–	cor(), dist(), hclust(), isoMDS(), corresp()
		Зависимые		mcnemar.test()	–	–	–

Таблица 8.1. Варианты статистического анализа и соответствующие функции R

нимым. Для создания подобного отчета надо вначале набрать простой LaTeX-файл и назвать его, например, `test-Sweave.Rnw`:

```
\documentclass[a4paper,12pt]{article}
\usepackage[T2A]{fontenc}
\usepackage[utf8]{inputenc}
\usepackage[english,russian]{babel}
\usepackage[noae]{Sweave}

\begin{document} % Тут начинается отчет

\textsf{R} как калькулятор:

<<echo=TRUE,print=TRUE>>=
1 + 1
1 + pi
sin(pi/2)
@

Картинка:

<<fig=TRUE>>=
plot(1:20)
@
\end{document}
```

Затем этот файл необходимо обработать в R:

```
> Sweave("test-Sweave.Rnw")
Writing to file test-Sweave.tex
Processing code chunks ...
 1 : echo print term verbatim
 2 : echo term verbatim eps pdf

You can now run LaTeX on 'test-Sweave.tex'
```

При этом создается готовый LaTeX-файл `test-Sweave.tex`. И наконец, при помощи LaTeX и `dvips` или pdfLaTeX можно получить результирующий PDF-файл, который изображен на рис. 61.

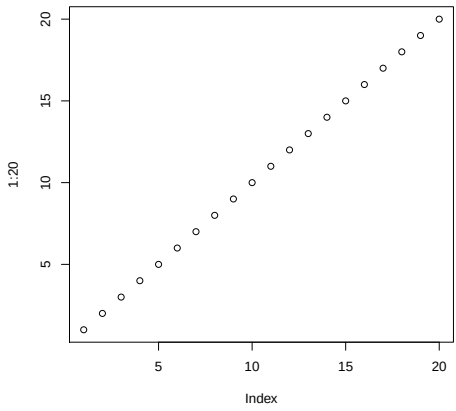
Такой отчет можно расширять, шлифовать, изменять исходные данные, и при этом усилия по оформлению сводятся к минимуму.

R как калькулятор:

```
> 1 + 1
[1] 2
> 1 + pi
[1] 4.141593
> sin(pi/2)
[1] 1
```

Картинка:

```
> plot(1:20)
```



1

Рис. 61. Пример отчета, полученного с помощью команды `Sweave()`

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 работать с ODF (формат OpenOffice.org).

И все-таки полностью полагаться на автоматику никогда не сто́ит. Обязательно проверьте, например, полученные графики — достаточен ли размер графических файлов, нет ли проблем с цветами. И не забудьте 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()
```

Удачного вам анализа данных!

Приложение A

Пример работы в R

Это приложение написано для тех, кто не хочет читать длинные тексты, а хочет как можно скорее научиться обрабатывать данные в R. Для этого надо «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)`). Мы не стали показывать здесь то, что выводит R, и не приводим получающиеся графики: **все это вам нужно будет получить и проверить самостоятельно**.

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
```

Начинаем...

Создаем на жестком диске рабочую директорию, создаем в ней директорию `data`; копируем в последнюю файл данных в текстовом формате с расширением `*.txt` и разделителем-табуляцией (делается из файла Excel или другой подобной прграммы командой меню `Save as... / Сохранить как...`). Пусть ваш файл называется `zhuki.txt`.

Внимание! Проверьте, чтобы десятичный разделитель был «.», то есть точкой, а не запятой: половина — это 0.5, а не 0,5! Если десятичный разделитель — запятая, то можно заменить его на точку в любом текстовом редакторе, например в Блокноте/Notepad.

Открываем программу R. Указываем директорию, где находится ваш 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 — «/».

Для проверки местоположения печатаем команду

```
> dir("data")
```

...и нажимаем `Enter` (эту клавишу нужно нажимать каждый раз после ввода команды).

Эта команда должна вывести, среди прочего, имя вашего файла (`zhuki.txt`).

Читаем файл данных (создаем в памяти программы объект под названием `data`, который представляет собой копию вашего файла данных). В строке ввода набираем:

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

Внимание! Будьте осторожны со скобками, знаками препинания и регистром букв! Если десятичный разделитель — запятая и вы почему-либо не смогли заменить его на точку, то есть способ загрузить и такие данные:

```
> data.2 <- read.table("data/zhuki_zap.txt", h=TRUE, dec=",")
```

Кстати, похожую таблицу данных можно создать при помощи R следующими командами:

```
> data.3 <- data.frame(POL=sample(0:1, 100, replace=TRUE),
+ CVET=sample(1:3, 100, replace=TRUE),
+ VES=sample(20:120, 100, replace=TRUE),
+ ROST=sample(50:150, 100, replace=TRUE))
```

Первая команда создает данные (у нас указана повторность 100) методом случайной выборки, а вторая записывает их в файл с заданным названием. Цифры и графики будут немного другие (выборка-то случайная!), но на наши упражнения это повлиять не должно.

Посмотрим на ваш файл данных (если файл большой, то можно использовать специальную команду `head(data)`):

```
> data
```

Внимание! Внутри R вносить изменения в данные не очень удобно. Разумно вносить их в текстовый 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)
```

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 (больше 10 мм) самцов:

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

Кстати, эту команду проще не вводить заново, а получить путем редактирования предыдущей команды (обычное дело в R). Для вызова предыдущей команды воспользуйтесь «стрелкой вверх» на клавиатуре. Использованные знаки «==» и «&» — это логические выражения «таких, что» и «и». Именно они служат критериями отбора. Кроме того, для отбора данных обязательно нужны квадратные скобки, а если данные табличные (как у нас), 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 был добавлен только к копии вашего файла данных, который находится в памяти программы. Чтобы сохранить измененный файл данных под именем `zhuki_new.txt` в вашей директории на жестком диске компьютера, нужно написать:

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

Охарактеризуем выборку...

Посмотрим сначала на основные характеристики каждого признака (не имеет смысла для категориальных данных):

```
> summary(data)
```

Конечно, команду `summary()`, как и многие прочие, можно применять как к всему файлу данных, так и к любому отдельному признаку:

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

А можно вычислять эти характеристики по отдельности. Минимум и максимум:

```
> min(data$VES)
> max(data$VES)
```

... медиана:

```
> median(data$VES)
```

... среднее арифметическое только для веса и для каждого из признаков:

```
> mean(data$VES)
```

и

```
> colMeans(data)
```

соответственно.

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 не в числе объектов, а в процентах, приняв за 100% общее число объектов:

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

И еще округлим 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 напоследок 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)
```

(Строки — разные цвета, столбцы — самцы и самки.)

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)
```

Обе команды «делают» один график (поэтому мы написали их в одну строчку через точку с запятой). 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)
```

и

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

Можно изменить вид значка, который обозначает объект (см. номера значков на рис. 62)). Похожую таблицу можно вызвать при помощи команды `example(points)` и нажать несколько раз `Enter`, пока «демонстрация» не окончится.

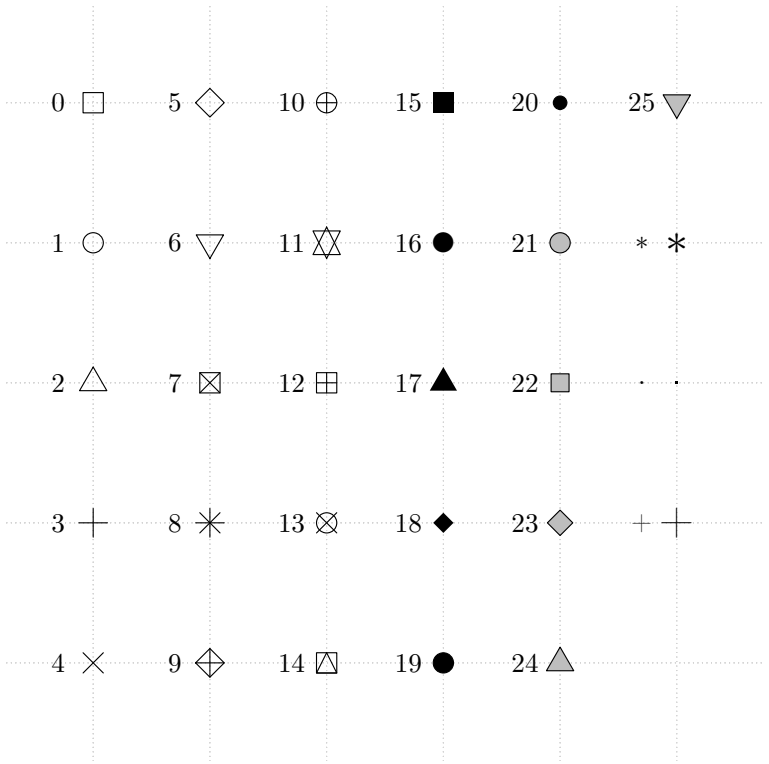


Рис. 62. Номера значков, используемых в стандартных графиках R

Вот, например, обозначим объекты значком 2-го типа (пустым треугольником):

```
> 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)
```

Другой, более сложный вариант — с использованием встроенных в R шрифтов Hershey (в порядке исключения приводим его на рис. 63):

```
> plot(data$ROST, data$VES, type="n", xlab="Рост", ylab="Вес")
> text(data$ROST, data$VES,
+ labels=ifelse(data$POL, "\\MA", "\\VE"),
+ vfont=c("serif","plain"), cex=1.5)
```

... и еще разными цветами:

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

Наконец, в случае с разноцветными значками нужно добавить условные обозначения (легенду):

```
> legend(50, 100, c("male", "female"), pch=c(0,1), col=c(1,2))
```

Числами указано положение (координаты на графике) верхнего левого угла легенды: первая цифра (`50`) — абсцисса, вторая цифра (`100`) — ордината.

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

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

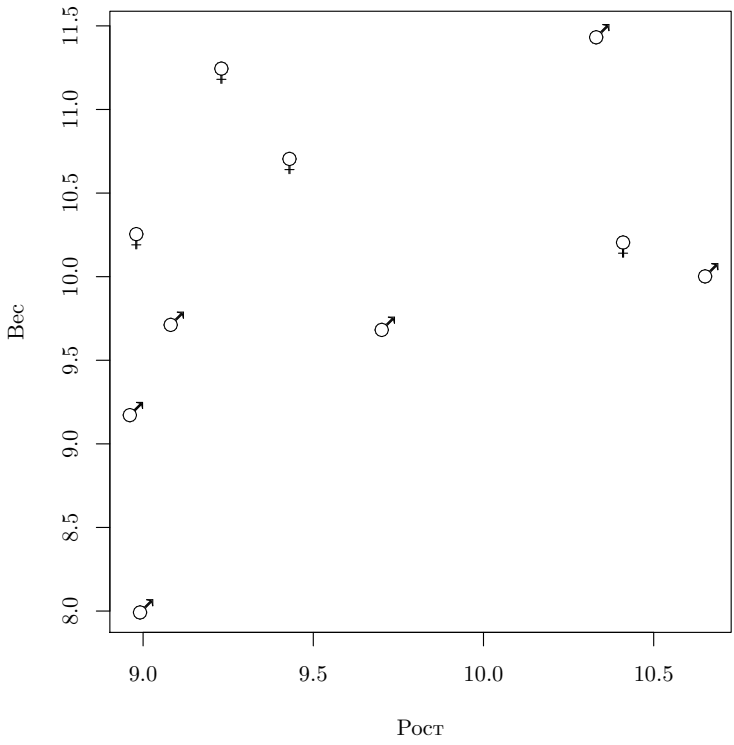


Рис. 63. Распределение самцов и самок жуков по росту и весу (для символов использованы шрифты Hershey)

Не забудьте напечатать `dev.off()`!

Чтобы получить график без посторонних надписей, следует дать команду `plot(..., main="", xlab="", ylab="")`, где многоточие обозначает все остальные возможные аргументы.

Сохранять график можно, и не выводя его на экран, сразу в графический файл, опять-таки с помощью *двух* дополнительных команд:

```
> png("data.png")
# [... тут печатаем команды для построения графика ...]
> 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, боксплот (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))
```

Заканчиваем...

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")`. Более того, можно запустить данный файл, не заходя в R,— для этого в командной строке надо набрать

```
$ Rscript zhuki.r
```

Внимание! **Всегда** сохраняйте то, что вы делали в R!

Выходим из программы через меню или с помощью команды `q("no")`.

Приложение Б

Графический интерфейс (GUI) для R

Прежде чем начать рассказ о GUI, хотелось бы сказать пару слов в пользу консольного интерфейса. Анализ данных — это творческий процесс, и ничто не продвигает его лучше, чем неспешный ввод с клавиатуры. Естественно, этому занятию должен предшествовать интервал времени, целиком и полностью посвященный чтению документации, книг и статей по тематике проблемы. Меню и кнопки отвлекают, создавая иллюзию простоты творческого процесса, требуя нажать их немедленно и посмотреть, что получится. Как правило, не получается ничего, то есть все равно приходится брать в руки книгу и думать.

Консольный интерфейс в R идеален. Он предоставляет пользователю историю команд, дополнение по `Tab` (то есть выдает при нажатии на эту клавишу список возможных продолжений команд, объектов и даже имен файлов), сохраняет информацию и объекты между сессиями (если пользователь этого захочет, естественно). Нужно поработать на удаленном компьютере? С консолью нет никаких проблем. А если воспользоваться программой `screen`, то можно не бояться разорванных сессий и случайно закрытых терминалов.

К счастью, практически все графические оболочки для R учитывают это и, как правило, «расширяют» базовые возможности пакета такими дополнениями, как интегрированный редактор кода с подсветкой, отладчик, редактор массивов данных и т. п.

Б.1. R Commander

R Commander, или Rcmdr (рис. 64),— кросс-платформенный графический интерфейс к R, написанный на Tcl/Tk. Его домашнюю страницу можно найти по адресу `http://socserv.mcmaster.ca/jfox/Misc/Rcmdr/`.

Автор R Commander Джон Фокс (John Fox) признается, что в случае программ для статистического анализа он не является фанатом интерфейса, состоящего из меню и диалогов. С его точки зрения, R Commander полезен в основном для образовательных целей при введении в R, а также в отдельных, очень редких случаях — для быстрого анализа.

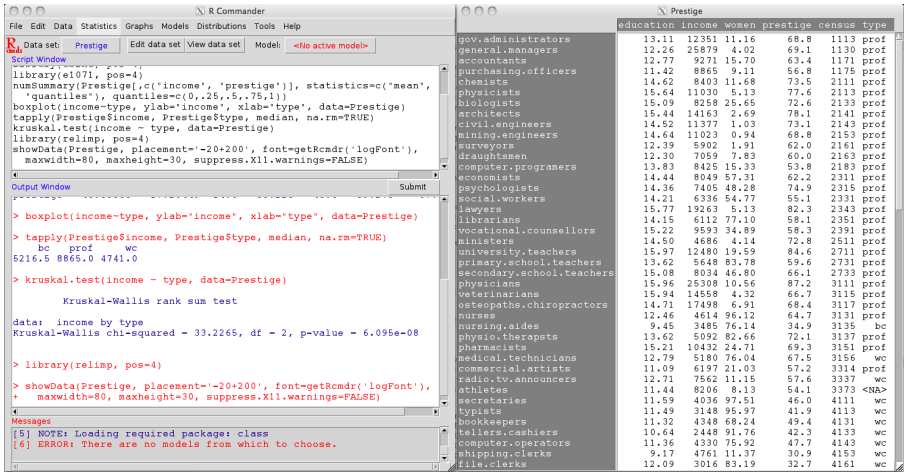


Рис. 64. Окно *R Commander*

Одной из основополагающих целей, преследуемых при создании интерфейса R Commander, был мягкий перевод пользователя в консоль, где можно автоматизировать свои действия более глобально.

Пакет распространяется под лицензией GPLv2, и поэтому доступен во всех стандартных дистрибутивах Linux. Например, для установки в Debian/Ubuntu достаточно выполнить команду:

```
$ sudo aptitude install r-cran-rcmdr
```

Пакет присутствует также и на CRAN, поэтому его установку можно произвести и собственными силами R:

```
> install.packages("Rcmdr", dependencies=TRUE)
```

Обратите внимание на значение опции `dependencies`: R Commander зависит от довольно большого числа пакетов. После установки в сессии R следует выполнить команду

```
> library(Rcmdr)
```

При запуске R Commander открывается снабженное довольно «развесистым» меню окно, разделенное на **Окно скриптов** (Script Window), **Окно вывода** (Output Window), а также информационное окно **Сообщения** (Messages). Многие действия в R Commander можно выполнять через меню, которое достаточно легко настраивается, например с помощью редактирования текстового файла `Rcmdr-menus.txt`. В **Окне скриптов** можно вводить команды, то есть консоль никуда не

делась. Графики появляются в отдельных окнах, как и в случае «чистого» R.

R Commander имеет русский перевод, который активизируется автоматически, если система русифицирована.

Основная проблема русифицированного R Commander (если вы работаете в Linux) — это кириллические шрифты, которые выбраны по умолчанию. Поэтому если вы предпочитаете русский интерфейс, то перед загрузкой R Commander ему с помощью команды `options` следует передать примерно следующее:

```
 > options(Rcmdr=list(default.font=
"-rfx-fixed-medium-r-normal-*-20*", suppress.X11.warnings=TRUE))
```

Растровый шрифт семейства `rfx` от Дмитрия Болховитянова (20 в конце строки — это размер по умолчанию), установленный в качестве `default.font`, находится обычно в составе пакета `xfonts-bolkhov` вашего дистрибутива Linux. Второй параметр, `suppress.X11.warnings`, подавляет надоедливые сообщения при создании новых графических окон.

Выйти из R Commander можно через меню `Файл -> Выйти`, при этом можно одновременно закрыть и сессию R. Если же сессия R осталась открытой, то повторный запуск R Commander выполняется с помощью команды

```
 > Commander()
```

Для вводного ознакомления с возможностями R Commander следует прочитать текст `Getting-Started-with-the-Rcmdr.pdf`, получить доступ к которому можно через меню `Помощь -> Введение в R Commander`.

Если по какой-то причине было принято решение анализировать данные исключительно с помощью R Commander, то можно сделать так, чтобы при запуске R эта графическая оболочка загружалась автоматически. Для этого в файл пользовательских настроек `~/.Rprofile` достаточно добавить следующие строки:

```
old <- getOption("defaultPackages")
options(defaultPackages = c(old, "Rcmdr"))
```

Глобальная переменная `defaultPackages` содержит информацию о модулях, автоматически загружаемых при старте R.

Б.2. RStudio

RStudio (`http://www.rstudio.com/ide/`) — это относительно новая оболочка для R, доступная под свободной лицензией. Среда работает

под основными операционными системами (Windows, Linux, Mac OS X), и, кроме этого, предоставляет доступ к R через Web-интерфейс.

После запуска появляется окно (рис. 65), удобно разделенное на несколько частей: исходный код редактируемого файла, консоль, содержимое рабочего пространства, история команд, и часть, предоставляющая доступ к списку пакетов, рисунков и файлов в текущем рабочем каталоге и справке (содержимое отдельных частей окна можно поменять через меню `Tools -> Options -> Pane Layout`).

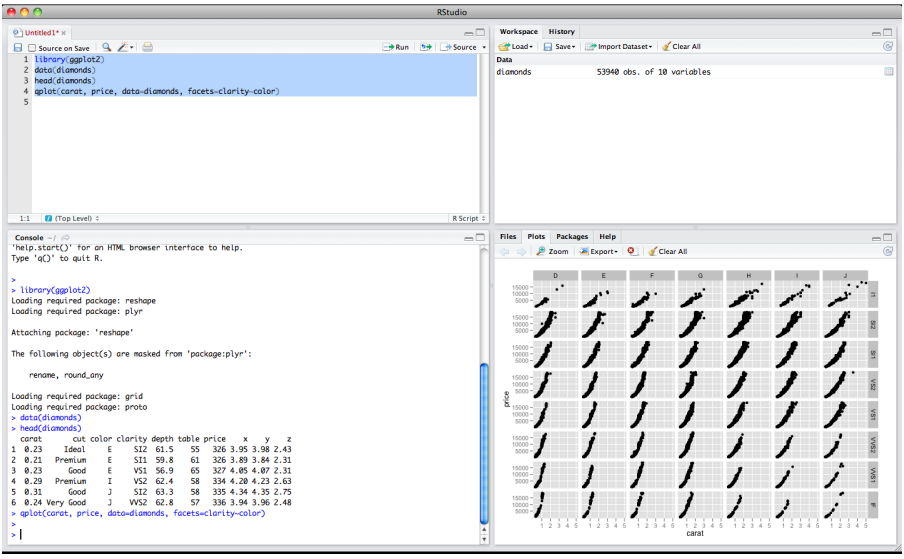


Рис. 65. Окно RStudio

В качестве отдельных преимуществ оболочки следует отметить редактор кода с автоматической подсветкой, автодополнением, простейшими преобразования кода (например, выделение нескольких строк в отдельную функцию), поддержку редактирования не только R кода, но и документов `Sweave`.

Авторы среды пошли немного дальше простого предоставления функций R из-под оболочки: в комплекте с RStudio поставляется пакет `manipulate`, позволяющий *интерактивно* менять параметры графиков внутри среды (рис. 66). Так, например, следующий код

```
> library(manipulate)
> manipulate(plot(cars, xlim = c(0, x.max), type = type,
+ ann = label),
+ x.max = slider(10, 25, step=5, initial = 25),
+ type = picker("Points" = "p", "Line" = "l", "Step" = "s"),
+ label = checkbox(TRUE, "Draw Labels"))
```

выведет окно настроек, позволяющее менять значение переменных `x.max`, `type` и `label` и при этом сразу же видеть результат такого изменения на графике.

Б.3. RKWard

RKWard (`http://rkward.sourceforge.net`) — это довольно удобный, основанный на KDE интерфейс к R (рис. 67). Разработчики RKWard старались совместить мощь R с простотой использования, подобной предоставляемой коммерческими статистическими пакетами (такими, как SPSS или STATISTICA).

Для начинающих пользователей RKWard предоставляет широкие возможности по выбору многих стандартных процедур статистического анализа по принципу «выдели и щелкни»: достаточно активировать соответствующий пункт меню.

Для продвинутого пользователя RKWard предлагает удобный редактор кода с подсветкой, автоматической расстановкой отступов, автодополнением — теми вещами, без которых в настоящее время не обходится ни одна среда программирования. Привычная консоль R также наличествует, она доступна в любое время на вкладке **R Console**.

Авторы RKWard взяли курс на как можно более полную интеграцию функций R в графическую среду: присутствует **браузер текущего окружения** (`environment`) и **редактор данных**. Есть **менеджер пакетов**, умеющий не только устанавливать их, но и следить за обновлениями; обеспечивается прозрачная интеграция со справочной системой. Кроме того, RKWard умеет перехватывать создание графических окон и добавлять к ним очень удобные функции типа сохранения содержимого в файл одного из стандартных форматов, поддерживаемых R (PDF, EPS, JPEG, PNG).

Интерфейс RKWard чрезвычайно гибок: пользователь может расширять его за счет написания собственных модулей (кстати, все встроенные средства анализа, доступные сразу же после запуска из меню,— это такие же модули, но только созданные авторами RKWard).

Б.4. Revolution-R

Ключевые особенности Revolution-R (его домашняя страница — `http://www.revolutionanalytics.com`) — это, разумеется, графический интерфейс (рис. 68), полная интеграция с Microsoft Visual Studio, наличие полноценного пошагового отладчика и оптимизированных подпрограмм математических функций `BLAS` и `LAPACK`, позволяющих автоматически использовать преимущества многопроцессорных систем.

Рис. 66. Вот так в RStudio работает пакет `manipulate`

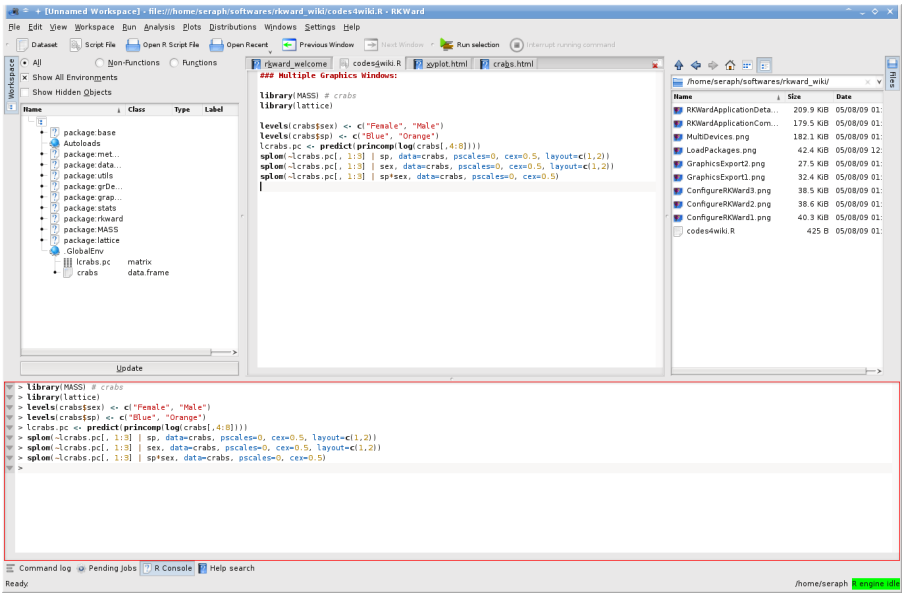


Рис. 67. Окно RKWard

Как читатель уже понял, основная платформа, на которую «нацеливается» Revolution-R, — это Windows. Версия под Linux тоже существует, но пошаговый отладчик в комплекте отсутствует, что сразу же снижает ценность пакета.

Revolution-R существует в двух редакциях: бесплатной *Community Edition* и очень платной *Enterprise Edition* (цена на последнюю начинается с $1000 за рабочее место). Главное отличие бесплатной версии заключается в отсутствии какой-либо интеграции с Visual Studio (а, следовательно, и в отсутствии отладчика). Впрочем, оптимизированные математические подпрограммы доступны всюду, поэтому получать преимущества от многопроцессорных вычислений можно совершенно бесплатно.

Стоит отметить, что существует программа по предоставлению бесплатных лицензий (но без технической поддержки) на *Enterprise* редакцию при условии использования для научных исследований. Для получения такой лицензии на веб-сайте программы следует отправить запрос с использованием «академического» (например, университетского) адреса электронной почты.

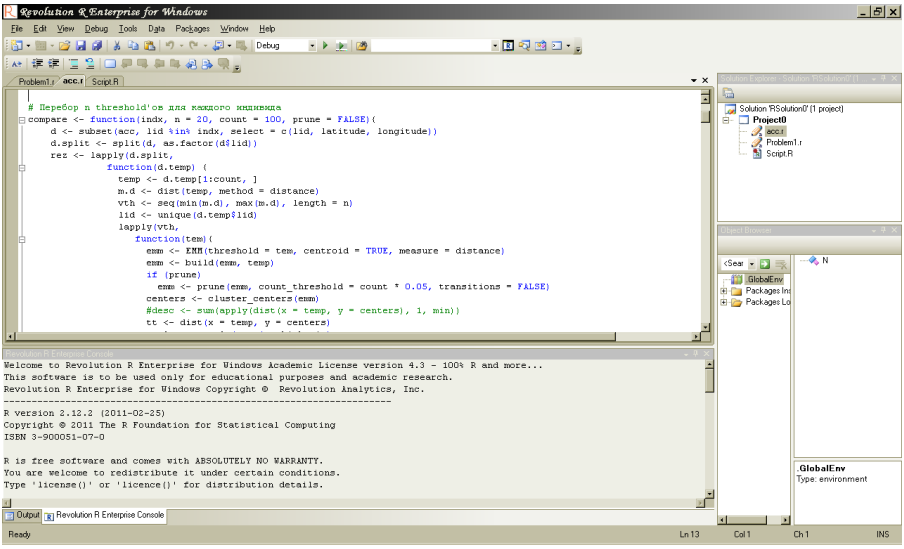


Рис. 68. Окно Revolution-R

Б.5. JGR

Предпочитаете, чтобы графический интерфейс отрисовывался средствами Java (рис. 69)? Тогда JGR (Java GUI для R) — это то, что вы искали. JGR (произносится как «ягуар») был впервые представлен публике в 2004 году, но развивается и поддерживается до сих пор. Пакет JGR распространяется под открытой лицензией GPLv2, и его домашняя страница доступна по адресу `http://jgr.markushelbig.org/JGR.html`.

JGR создавался для Mac OS X «со всеми вытекающими», но под Windows и Linux он тоже работает. Для использования JGR под Linux необходимо инсталлировать Sun Java Development Kit (JDK):

```
$ apt-get install sun-java6-jdk
$ sudo update-java-alternatives -s java-6-sun
$ sudo R CMD javareconf
```

После установки и настройки Java-окружения следует запустить сессию R и выполнить следующие действия:

```
> install.packages('JGR')
> library(JGR)
> JGR()
Starting JGR run script. ...
```

Готово — JGR запущен.

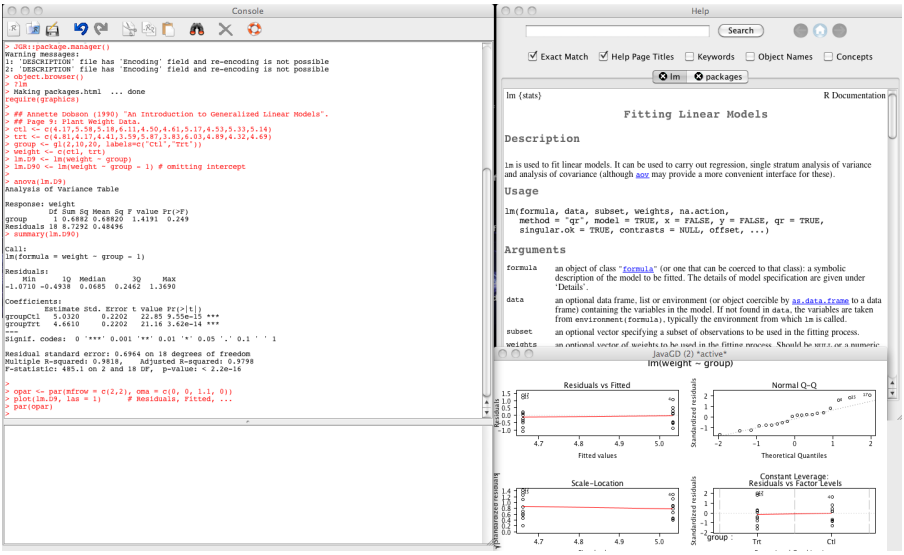


Рис. 69. Окно JGR

В JGR есть встроенный текстовый редактор, в нем есть подсветка синтаксиса и `Tab`-завершение команд. Есть и гипертекстовая помощь, простенькая электронная таблица, возможность управлять объектами, в том числе и с помощью мыши, имеется и графический интерфейс для установки и загрузки R-пакетов. С учетом того, что этот GUI может работать везде, где есть Java, на него стоит обратить внимание.

Б.6. Rattle

Rattle (`http://rattle.togaware.com`) — сокращение, обозначающее «R Analytical Tool To Learn Easily» (легкая в освоении среда анализа R). Программа активно развивается. Rattle — это среда для «разглядывания» данных человеком (рис. 70), она предназначена для интеллектуального анализа данных (data mining), иными словами, для выявления скрытых закономерностей или взаимосвязей между переменными в больших массивах необработанных данных.

Для установки Rattle под Linux необходимо наличие пакетов `ggobi` (программа визуализации данных) и `libglade2-dev`:

```
$ apt-get install ggobi libglade2-dev
```

После этого в консоли R следует выполнить команду

```
> install.packages("rattle", dependencies=TRUE)
```

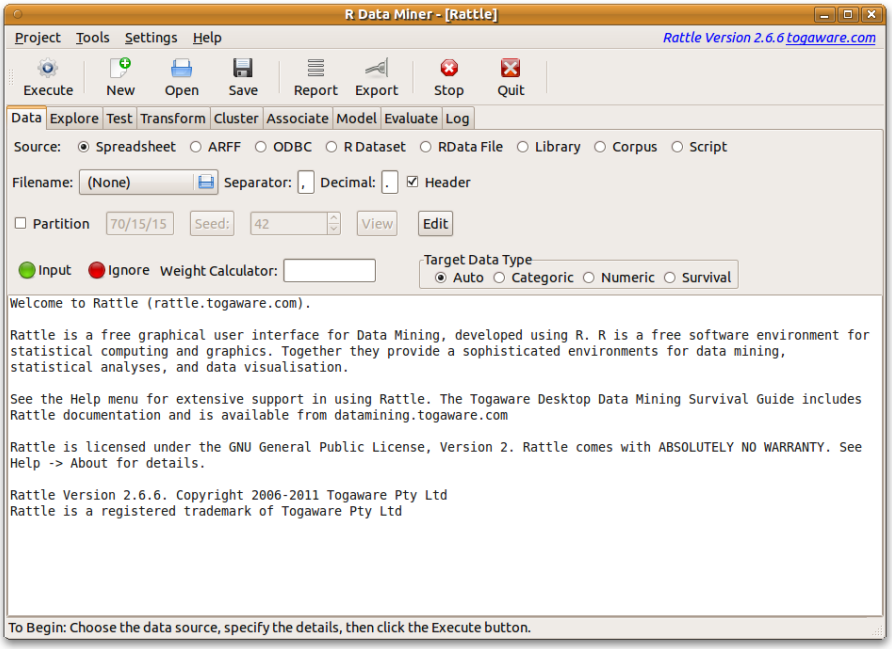


Рис. 70. Стартовое окно Rattle

и откинуться на спинку кресла. Из-за большого количества зависимостей установка занимает много времени.

Запуск GUI производится как обычно:

```
> library(rattle)
> rattle()
```

Б.7. rpanel

Пакет `rpanel` — это простая и одновременно высокоуровневая надстройка над R, написанная в том числе на языке Tcl/Tk. Это не GUI в строгом смысле слова, а скорее средство (toolkit) для того, чтобы самостоятельно создать простые GUI, которые потом можно использовать для множества целей. Самое, наверное, важное преимущество `rpanel` — вы можете сделать графическое приложение под конкретную задачу всего за несколько минут!

Если пакет `rpanel` отсутствует в системе, то надо его скачать и установить с помощью команды `install.packages("rpanel")`. После этого можно загрузить саму библиотеку:

```
> library(rpanel)
```

Логика использования пакета довольно проста:

1. С помощью функции `rp.control` нужно создать объект `panel` и объявить в нем нужные нам переменные.

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

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

Разберем простейший пример, в котором функция выводит на экран одно из двух: или гистограмму полученных данных, или «ящик-с-усами» (боксплот) в зависимости от выбранного значения в элементе `rp.radiogroup` (рис. 71).

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

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

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

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

В заключение объявим функцию, которая будет выполняться в ответ на взаимодействие с переключателем:

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

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

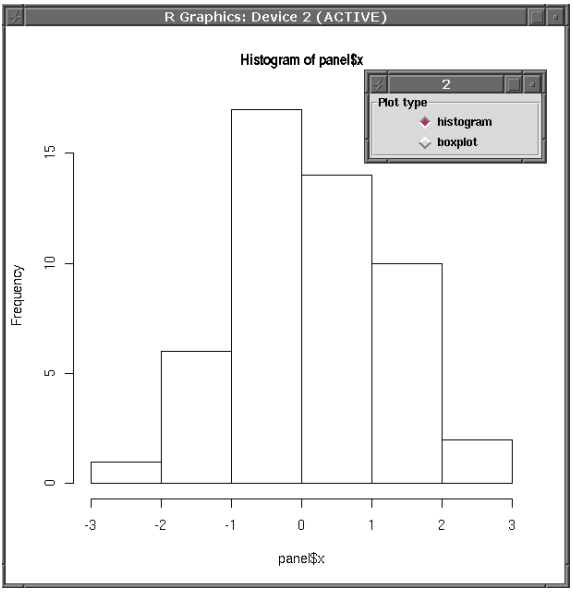


Рис. 71. Пример простого переключателя

Б.8. ESS и другие IDE

Теперь поговорим не о GUI в узком смысле слова, а о так называемых средствах разработки, IDE (integrated desktop environments), под которыми обычно понимаются сильно расширенные текстовые редакторы со встроенными средствами компиляции, анализа исходного кода и т. д.

Тут прежде всего надо вспомнить ESS (`http://ess.r-project.org`). ESS — это специализированная интерактивная среда или мода для редактора Emacs. ESS является сокращением от фразы «Emacs Speaks Statistics», которую можно перевести как «Emacs говорит на языке статистики». Пакет ESS поддерживает не только систему статистического анализа R, но и многие другие статистические среды. При установке пакета ESS (так, как это принято в вашейсистеме) и добавлении строчки для инициализации ESS в `/.emacs`

```
(require 'ess-site)
```

редактору Emacs становятся доступны две дополнительные моды:

- **Мода ESS**, которая предназначена для редактирования исходных файлов с использованием команд R (включая дополнительную поддержку для редактирования `Rnw`-файлов), и

- **Мода iESS** — замена обычной R-консоли.

Интерпретатор R запускается с помощью команды «R»: `Alt+X`, а затем `R` и `Enter`. При старте спрашивается о местоположении рабочей директории для открываемой сессии R, а затем отображается стандартное R-приглашение. Можно параллельно запустить насколько сессий, а также задействовать удаленный компьютер (команда **ess-remote**). Работа в этой моде почти ничем не отличается от консоли, за исключением наличия стандартных возможностей редактирования текста в Emacs и дополнительного вспомогательного меню **iESS**. Из него можно получить доступ к таким интересным возможностям, как, например, редактирование объектов R. Emacs — это прежде всего текстовый редактор, поэтому подобные особенности делают анализ данных комфортнее.

ESS-мода инициализируется автоматически при загрузке файлов с расширением R. Emacs предоставляет пользователю подсветку синтаксиса, выравнивание исходного кода по нажатии `Tab`, выполнение отдельных строк, фрагментов и всего файла в интерпретаторе R. Дополнительное меню ESS позволяет получить доступ ко всем этим возможностям. Редактирование `Rnw`-файлов (смесь LaTeX- и R-команд для быстрого составления отчетов с помощью `Sweave`) чуть более мудреное: мода ESS активируется, только когда курсор оказывается внутри фрагментов с R-командами.

Так что если у вас есть навыки работы в Emacs, то ESS — это верный выбор. Но если вы предпочитаете другой редактор, то расстраиваться не стоит. Почти все «уважающие себя» текстовые редакторы или среды разработки в той или иной степени поддерживают R. Кроме уже упомянутого Emacs, к таковым относятся и Vim, и jEdit, и Bluefish, и SciTE. «Большие» IDE, например Eclipse и Komodo, тоже имеет соответствующие модули (соответственно, StatET, `http://www.walware.de/goto/statet`, и SciViews-K, `http://www.sciviews.org/SciViews-K/index.html`). Под Windows есть замечательный редактор Tinn-R.

Приложение B

Основы программирования в R

Это приложение — для «продвинутого» читателя, который уже имеет определенный опыт программирования и хочет освоить азы программирования в R.

B.1. 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 — более сложная сущность. Так например, тип объекта `vector` определяется типом элементов, из которых он состоит.

B.1.1. 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, может быть 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 C, 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». Строки заключаются в одинарные или двойные кавычки. В случае использования двойных кавычек любую двойную кавычку внутри строки необходимо предварять символом «\». Аналогично следует поступать с одинарными кавычками внутри строки, заключенной в одинарные кавычки. При выводе на экран всегда используются двойные кавычки.

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)
```

B.1.2. Список

Вектор содержит элементы исключительно одного типа. Это делает его простым, но в то же время несколько ограничивает сферу его при-

менения. 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()` (см. ниже).

B.1.3. 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 (массив) 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 специальной функции `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
```

B.1.4. 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
```

B.1.5. 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 (конечно, его длина обязательно должна совпадать с длиной векторов данных, а сами имена не должны повторяться).

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
```

B.1.6. Выражение

Язык R предоставляет доступ к конструкциям самого языка. Так, функция, как будет описано далее, 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`.

B.2. 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
```

Хотя дальнейшие примеры и будут, как правило, описаны в терминах операции извлечения, сразу же стоит отметить, что все нижеописанное применимо и к операции замены (если явно не сказано обратного).

B.2.1. 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]  1  2  3  4  5 NA NA NA NA 10
```

B.2.2. 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.

B.2.3. 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
```

B.2.4. 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
```

B.2.5. Оператор $

Оператор $ служит для извлечения отдельного элемента из списка. Синтаксис очень прост: слева от знака $ стоит список, элемент которого следует 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:

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

B.2.6. 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.

B.2.7. 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 мо-

жет 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 ошибкам в 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.

B.2.8. 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 (осторожно!):

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

Заменяет 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))
```

B.3. 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, мы рекомендуем 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. Отметим еще использование конструкции `Recall()` во втором примере. 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:

1. Аргументы, переданные в виде пары *имя* = *значение*, имена которых совпадают явно.

2. Аргументы, переданные в виде пары *имя* = *значение*, имена которых совпали по уникальному префиксу.

3. 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:

1. Назначаются аргументы `arg1` и `aa`, так как их имена совпадают целиком.

2. Значение аргументу `aa` уже было присвоено на предыдущем шаге, поэтому `ab` может быть назначен по совпадению префикса `a`, ставшего уникальным.

3. Вследствие совпадения имени для переданного аргумента `ac` происходит добавление аргумента к списку (. . .). `arg2` получает значение по умолчанию. После этого этапа именованных аргументов не остается.

4. 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 отличаться от 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.

B.4. 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 (например, C), они используются существенно реже по причинам, которые будут разобраны ниже.

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. Если длина условия больше 1, то используется только *первый элемент* вектора 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, но не их содержимого целиком. Ожидаемого поведения можно достичь при помощи 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`.

B.5. 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 командами и операторами этого языка и командами R. Некоторые из них приведены в табл. B.1.

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

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 `-`

Таблица В.1. Некоторые (примерные) соответствия между операторами и командами SQL и функциями R

таблицы на основании общей колонки («ключа» в терминологии баз данных).

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("data/eq-l.txt", h=T, sep=";")
> measurements <- read.table("data/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
6     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 (см. главу про многомерные данные), 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).

B.6. Правила переписывания. Векторизация

Встроенные операции языка R *векторизованы*, то есть выполняются покомпонентно. 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):

1. Длина результата совпадает с длиной операнда наибольшей длины.

2. 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 покомпонентно над операндами одинаковой длины.

3. 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 object
  length
```

Большинство встроенных функций языка R так или иначе векторизованы, то есть выдают «естественный» результат при передаче в качестве 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, очевидно, человеком, хорошо знакомым с языком типа C, но малознакомым с R):

```
> p <- 1:20
> lik <- 0
> for (i in 1: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 образец кода 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 переписы-

вании кода с других языков программирования) код следует изменить для того, чтобы он стал векторизованным. Например, код

```
> v <- NULL
> v2 <- 1:10
> for (i in 1: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. Среда R действительно прячет детали выделения и освобождения памяти от пользователя, но это вовсе не значит, что о них не надо знать и их не надо учитывать. В данном случае 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 векторизовать при 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`. Векторы `vtrue` и `vfalse` переписываются до этой длины. 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 1: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 строк и 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`.

B.7. 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, оно возникает 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)
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 exit

1: 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 ар-

гумента 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. Мы не будем останавливаться детально на работе с этим пакетом, всю информацию можно получить из встроенной справки:

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

B.8. Элементы объектно-ориентированного программирования в R

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 функции можно при помощи `methods`:

```
 > methods(summary)
[1]  summary.aov        summary.aovlist    summary.aspell*
[4]  summary.connection summary.data.frame summary.Date
[7]  summary.default    summary.ecdf*      summary.factor
[10] summary.glm        summary.infl       summary.lm
[13] summary.loess*     summary.manova     summary.matrix

...

Non-visible functions are asterisked
```

Сразу же отметим, что файлы справки для универсальной функции и функции-метода, вообще говоря, *отличаются*, поэтому, например, справку по методу `summary()` для класса `Date` следует вызывать, используя имя метода: `?summary.Date`.

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

Приложение Г

Выдержки из документации R

Это приложение поможет глубже разобраться в структуре R. Вдумчивый и заинтересованный читатель найдет здесь много полезного, особенно из областей создания функций и графиков. В качестве источника мы взяли английский текст «Introduction to R» — часть официальной документации, распространяемой на сайте программы, сократив некоторые разделы, в том числе те, которые уже были освещены в предыдущих главах и приложениях.

Г.1. Среда R

R представляет собой набор программных средств для работы с данными, вычислений и графического отображения. Среди прочего возможны:

- 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 непосредственно на компьютере или при выводе на печать;
- хорошо развитый, простой и эффективный язык программирования (под названием «S»), который включает условия, циклы, определенные пользователем рекурсивные функции и возможности ввода-вывода. (Большинство поддерживаемых системой функций сами написаны на языке S.)

Термины «окружение»/«среда» введены, чтобы подчеркнуть наличие полностью спланированной и непротиворечивой системы, а не постепенно возникшего конгломерата весьма специфических и негибких утилит, как это часто бывает с другим программным обеспечением анализа данных.

R — это средство разработки новых методов интерактивного анализа данных. Он быстро разрабатывается и расширяется большой коллекцией пакетов.

Г.2. R и S

R можно рассматривать как реализацию языка S, который был разработан в компании «Bell Laboratories» Риком Бекером, Джоном Чамбером и Алланом Вилксом и который лежит в основе S-Plus.

Документация S/S-Plus, как правило, тоже может быть использована с R, если помнить о различиях между реализациями S.

Г.3. R и статистика

В нашем описании среды R не упоминается о статистике, но многие люди используют R в качестве статистической системы. Мы предпочитаем думать о нем как об окружении, в котором были реализованы многие классические и современные статистические методы. Некоторые из них встроены в базовое окружение R, но многие поставляется как пакеты. Есть около 25 пакетов, поставляемых с R (так называемые «стандартный» и «рекомендованный» наборы пакетов), гораздо больше можно получить через CRAN (`http://CRAN.R-project.org`) и из других источников. Более подробно информацию о пакетах мы рассмотрим ниже.

Существует важное различие между философией в S (и, следовательно, R) и других основных статистических систем. В S статистический анализ, как правило, выполняется в виде серии шагов, с сохранением промежуточных результатов в объектах. Так, если SAS и SPSS выведут обильные результаты процедуры регрессионного или дискриминантного анализа, R выведет минимум результатов и сохранит вывод в объект для последующего использования.

Г.4. Получение помощи

R имеет встроенную команду `help()` по аналогии с командой `man` в UNIX. Чтобы получить дополнительную информацию о какой-либо конкретной функции, например `solve()`, нужно ввести `help(solve)`. Альтернативный вариант — `?solve`.

Для наименования, обозначенного специальными символами, аргумент должен быть помещен в двойные или одиночные кавычки, что превращает его в строку символов. Это также необходимо для некото-

рых слов и знаков, имеющих синтаксическое значение, включая `if` и `for`. Пример — `help("[[")`.

Любая форма кавычек может быть использована, чтобы экранировать другие кавычки, как в строке `"It's important"`. Мы будем придерживаться по возможности использования двойных кавычек.

На большинстве установленных R-систем доступна справка в формате HTML, достаточно выполнить команду `help.start()` — и будет запущен веб-браузер, что позволит просматривать страницы помощи в режиме гипертекста. В UNIX последующие `help()` запросы будут направляться в HTML-вариант системы помощи. Ссылка «Поисковая система и ключевые слова», доступная на странице, загружаемой после выполнения `help.start()`, является особенно полезной, поскольку содержит высокоуровневое оглавление, служащее для поиска доступных функций.

Команда `help.search()` позволяет искать подсказку различными способами: выполните `?help.search` для того, чтобы узнать подробности. Примеры по теме страницы помощи можно, как правило, запустить командой `example(тема)`.

Windows-версии R имеют и другие дополнительные системы помощи — воспользуйтесь `?help`, для того чтобы узнать подробности.

Г.5. Команды R

Технически R является языком выражений с очень простым синтаксисом. Он учитывает регистр, как и большинство других программ UNIX. Так, «`A`» и «`a`» — это различные символы и будут обозначать различные переменные. Набор символов, которые могут быть использованы как имена в R, зависит от операционной системы и страны, в которых будет запущен R (технически говоря, от используемой локали). Обычно разрешены все алфавитно-цифровые символы вместе с «`.`» и «`_`», с ограничением, что имя должно начинаться с «`.`» или буквы, а если оно начинается с «`.`», то второй символ не должен быть цифрой.

Простые команды состоят из выражений либо присвоений. Если выражение вводится как команда, оно вычисляется, выводится (если специально не сделано невидимым), и результат теряется. Присвоение также вычисляет выражение и передает значение переменной, но результат автоматически не выводится.

Команды разделяются либо точкой с запятой («`;`»), либо переводом строки. Простые команды могут быть сгруппированы в единое составное выражение фигурными скобками («`{`» и «`}`»). Комментарии могут быть практически где угодно, начинаются с символа решетки («`#`»), при этом все до конца строки является комментарием.

Если команда не завершена в конце строки, R выдаст особое приглашение, по умолчанию

```
+
```

во второй и последующих строках и продолжит ожидать ввода, пока команда не будет синтаксически завершена. Это приглашение может быть изменено пользователем.

Командные строки, набираемые в консоли, ограничены 1024 байтами (не символами!).

Г.6. Повтор и коррекция предыдущих команд

Во многих версиях UNIX и Windows R предусматривает механизм восстановления и повторного выполнения предыдущей команды. Вертикальные стрелки на клавиатуре можно использовать для прокрутки вперед и назад по истории команд. Когда команда найдена таким способом, курсор может перемещаться внутри команды с помощью горизонтальных стрелок, можно удалять символы при помощи клавиши `<DEL>` или добавлять при помощи остальных клавиш.

Возможности повтора и редактирования в UNIX являются гибко настраиваемыми. Вы можете узнать, как это сделать, прочитав системное руководство по `readline` (под Linux это можно сделать, набрав в консоли «`man readline`»).

Г.7. Сохранение данных и удаление объектов

Записи, которые создает R и которыми он манипулирует, известны как «объекты». Это могут быть переменные, массивы чисел, строки символов, функции или более сложные конструкции, построенные из этих компонентов.

В ходе сессии R создаются и хранятся именованные объекты. Команда

```
> objects()
```

так же, как и `ls()`, может быть использована для отображения названий объектов, которые в настоящее время хранятся в R. Набор объектов, который в настоящее время хранится, называется «рабочее пространство». Чтобы удалить объекты, есть функция `rm()`:

```
> rm(x, y, z)
```

Если нужно удалить все созданные в течение сессии объекты (например, для того чтобы не выходить и опять входить в R), то можно выполнить команду (осторожно, не удалите нужные вам объекты!)

```
> rm(list=ls())
```

Все объекты, созданные в ходе сессий R, могут быть сохранены в файл для использования в последующих сессиях. В конце каждой сессии R вам предоставляется возможность сохранить все имеющиеся в настоящее время объекты. Если вы подтвердите, что хотите этого, объекты записываются в файл `.RData` в текущем каталоге, а строки команд, использованных в сессии, сохраняются в файл `.Rhistory`.

Если R будет запущен позже из этого каталога, то рабочее пространство будет перезагружено из файла `.RData`. Одновременно загрузится связанная история команд из `.Rhistory`.

Мы рекомендуем вам использовать отдельный рабочий каталог для каждого из анализов, проведенных в R.

Если команды хранятся во внешнем файле, например `commands.r` в поддиректории `work`, они могут быть выполнены в любой момент с помощью команды `source("work/commands.r")`.

Функция `sink("record.lis")` будет направлять весь последующий вывод консоли во внешний файл, `record.lis`. Команда `sink()` восстановит вывод в консоль.

Г.8. Внешнее произведение двух матриц

Важной операцией на матрицах является внешнее произведение. Если `a` и `b` — две числовые матрицы, то их внешнее произведение — это матрица, чей вектор размерностей состоит из последовательности элементов двух исходных векторов размерности (порядок важен!), а вектор данных получают, рассчитывая все возможные произведения элементов вектора данных `a` с элементами вектора данных `b`. Внешнее произведение выполняет специальный оператор `%o%`:

```
> a=b=array(1:24, dim=c(3,4,2))
> ab <- a %o% b
```

Альтернатива:

```
> ab <- outer(a, b, "*")
```

Функция умножения может быть заменена произвольной функцией двух переменных. Например, для того чтобы вычислить функцию

f(x; y) = cos(y)/(1 + x^2) на регулярной сетке значений с x и y координатами, определенными векторами x и y соответственно, мы могли бы действовать следующим образом:

```
> x=y=1:10
> f <- function(x, y) cos(y)/(1 + x^2)
> z <- outer(x, y, f)
```

Кстати, внешнее произведение двух обычных векторов является матрицей с двойным индексом (то есть матрицей с рангом как минимум 1). Заметьте, что оператор внешнего произведения не коммутативен (от перемены мест множителей результат меняется).

Г.9. c()

Следует отметить, что в то время как cbind() и rbind() являются функциями объединения, которые учитывают параметр dim, базовая функция c() не учитывает, а наоборот — снимает со всех численных объектов параметры dim и dimnames. Это иногда удобно.

Естественным способом привести матрицу обратно к простому векторному объекту является использование as.vector():

```
> vec <- as.vector(trees)
```

Однако аналогичный результат может быть достигнут и путем использования c() с только одним аргументом:

```
> vec <- c(trees)
```

Г.10. Присоединение

Конструкция с использованием $, такая как accountants$statef, не так уж и удобна. Полезной возможностью будет каким-то образом сделать компоненты списка или таблицы данных временно видимыми как переменные под их собственными именами, без необходимости каждый раз указывать название списка.

Предположим, lentils является таблицей данных с тремя переменными (столбцами) lentils$u, lentils$v, lentils$w. Подключаем ее командой attach():

```
> lentils <- data.frame(u=1:10, v=11:20, w=21:30)
> attach(lentils)
```

Тем самым мы помещаем таблицу данных в путь поиска на позицию 2 и подразумеваем отсутствие каких-либо объектов с именами `u`, `v` или `w` в позиции 1. С этого момента такие присвоения, как `u <- v + w` не заменяют компонент `u` таблицы данных, но маскируют ее другой переменной `u` рабочего каталога на позиции 1 пути поиска. Чтобы изменить переменные в самой таблице данных, можно прибегнуть к `$`-обозначениям:

```
> lentils$u <- v + w
```

Однако новое значение компонента `u` не будет видно до тех пор, пока таблица данных не будет отсоединена и присоединена снова.

Чтобы отсоединить таблицу данных, используют функцию `detach()`. Говоря точнее, это выражение отсоединяет от пути поиска записи, находящиеся в настоящее время в позиции 2. Таким образом, в данном контексте переменные `u`, `v` и `w` будут больше не видны. Записи на позициях больше чем 2 можно отсоединить путем указания их номера в `detach`, но гораздо безопаснее всегда использовать название, например `detach(lentils)` или `detach("lentils")`

Г.11. `scan()`

Предположим, что векторы данных имеют одинаковую длину и считываются параллельно. Далее предположим, что есть три вектора, первый символьного типа и оставшиеся два численного типа, а файл называется `input.dat`. Первым шагом является использование `scan()` для считывания трех векторов в виде списка следующим образом:

```
> inp <- scan("data/input.dat", list("", 0, 0))
```

Второй аргумент является фиктивным списком, который задает тип считываемых трех векторов. В результате получен список, компонентами которого являются три считанных вектора. Для разделения данных на три отдельных вектора используем назначение типа:

```
> label <- inp[[1]]; x <- inp[[2]]; y <- inp[[3]]
```

Удобнее, чтобы фиктивный список имел поименованные компоненты, в этом случае имена могут использоваться для доступа к считанным векторам. Например:

```
> inp <- scan("data/input.dat", list(id="", x=0, y=0))
```

Если вы хотите получить доступ к переменным по отдельности, они могут быть реорганизованы в переменные текущей таблицы данных:

```
> label <- inp$id
> x <- inp$x
> y <- inp$y
```

Или список может быть присоединен в позицию 2 пути поиска.

Если второй аргумент является одиночным значением, а не списком, считывается единый вектор, все компоненты которого должны быть одного типа, как и фиктивное значение.

```
> X <- matrix(scan("data/input.dat", list("",0,0)), ncol=3,
+ byrow=TRUE)
```

Г.12. R как набор статистических таблиц

В R есть весьма представительный набор статистических таблиц (табл. Г.1). Любой функции, перечисленной в таблице, можно задать префикс `d` названию для получения плотности, `p` для кумулятивной плотности распределения, `q` для квантильной функции и `r` для генерации случайной выборки. Первый (основной) аргумент в указанных функциях: `x` для `dxxx`, `q` для `pxxx`, `p` для `qxxx` и `n` для `rxxx` (за исключением `rhyper` и `rwilcox`, для которых задается `nn`). Все `pxxx` и `qxxx` функции имеют логические аргументы `lower.tail` и `log.p`, `dxxx` имеет параметр `log`.

Г.13. Область действия

Этот раздел содержит больше технических деталей, чем другие части этого документа, поскольку здесь описывается одно из основных различий между S-Plus и R.

Символы, которые встречаются в теле функции, могут быть разделены на три класса: формальные параметры, локальные переменные и свободные переменные. Формальные параметры функции — это те, которые входят в список аргументов функции. Их значение определяется в процессе связывания фактических аргументов функции с формальными аргументами. Локальные — это переменные, значения которых определяются вычислением выражений в теле функции. Переменные, которые не являются формальными параметрами или локальными переменными, называются свободными переменными. Если свободной переменной присвоить локальную переменную, то она тоже станет локальной. Рассмотрим следующее определение функции:

Распределение	Функция	Дополнительные аргументы
Бета	`beta`	`shape1, shape2, ncp`
Биноминальное	`binom`	`size, prob`
Коши-Лоренца	`cauchy`	`location, scale`
Хи-квадрат	`chisq`	`df, ncp`
Экспоненциальное	`exp`	`rate`
F (Фишера)	`f`	`df1, df2, ncp`
Гамма	`gamma`	`shape, scale`
Геометрическое	`geom`	`prob`
Гипергеометрическое	`hyper`	`m, n, k`
Лог-нормальное	`lnorm`	`meanlog, sdlog`
Логистическое	`logis`	`location, scale`
Негативное биномиальное	`nbinom`	`size, prob`
Нормальное	`norm`	`mean, sd`
Пуассона	`pois`	`lambda`
t (Стьюдента)	`t`	`df, ncp`
Равномерное	`unif`	`min, max`
Вейбулла	`weibull`	`shape, scale`
Уилкоксона	`wilcox`	`m, n`

Таблица Г.1. Распределения, встроенные в R

```
> f <- function(x) {
+ y <- 2*x
+ print(x)
+ print(y)
+ print(z)
+ }
```

В этой функции `x` — формальный параметр, `y` является локальной переменной, а `z` является свободной переменной.

В R связывание свободной переменной решается сначала поиском в области, в которой функция была создана. Это называется «область действия». Сначала определим функцию `cube()`:

```
> cube <- function(n) {
+ sq <- function() n*n
+ n*sq()
+ }
```

Переменная n в функции sq() — это не аргумент данной функции. Поэтому она выступает в качестве свободной переменной, и для определения значения, которое должно быть ей назначено, должны быть использованы правила области действия. Согласно статической модели области действия (S-Plus), значение определяется глобальной переменной с названием n. В области действия (R) она является параметром функции cube(), поскольку n *активна* в момент определения функции sq(). Разница между вычислением в R и вычислением в S-Plus заключается в том, что S-Plus ищет глобальную переменную с названием n, а R сначала ищет переменную с названием n в окружении, созданном после определения cube().

```
# Сначала вычислим в S-PLUS
S> cube(2)
Error in sq(): Object "n" not found
Dumped
S> n <- 3
S> cube(2)

[1] 18

# Та же функция вычисляется в R
R> cube(2)

[1] 8
```

Области действия могут использоваться, чтобы изменять состояние функций. В следующем примере мы покажем, как R может быть использован для имитации банковского счета. Для функционирования банковского счета необходимо иметь баланс или итог, функции для снятия, функцию для создания депозитов и функции для определения текущего баланса. Это достигается путем создания трех функций внутри счета, а затем возвращения списка, содержащего их. Когда счет заводится, он принимает численный аргумент total и возвращает список, содержащий все три функции. Поскольку эти функции определяются в окружении, которое содержит total, они будут иметь доступ к его значению.

Специальный оператор присваивания, «<<-», используется для изменения значения, связанного с total. Этот оператор «смотрит назад»

в поисках такого окружения, которое содержит символ `total`, и когда он находит такое окружение, то заменяет значение в этом окружении значением с правой стороны. Если достигнуто глобальное окружение или окружение верхнего уровня, а символ `total` так и не найден, то такая переменная создается, и ей присваивается значение. В большинстве случаев `<<-` создает глобальную переменную и присваивает значение справа налево. Только тогда, когда `<<-` был использован в функции, которая была возвращена как значение другой функции, происходит описываемое здесь специальное поведение. Вот пример:

```
 > open.account <- function(total) {
 + list(
 + deposit = function(amount) {
 + if(amount <= 0)
 + stop("Deposits must be positive!\n")
 + total <<- total + amount
 + cat(amount, "deposited. Your balance is", total, "\n\n")
 + },
 + withdraw = function(amount) {
 +if(amount > total)
 + stop("You don't have that much money!\n")
 +total <<- total - amount
 + cat(amount, "withdrawn. Your balance is", total, "\n\n")
 + },
 + balance = function() {
 + cat("Your balance is", total, "\n\n")
 + }
 + )
 + }

 > ross <- open.account(100)
 > robert <- open.account(200)
 > ross$withdraw(30)
30 withdrawn. Your balance is 70

 > ross$balance()
Your balance is 70

 > robert$balance()
Your balance is 200

 > ross$deposit(50)
50 deposited. Your balance is 120
```

```
 > ross$balance()
Your balance is 120
```

Г.14. Настройка окружения

Пользователи могут настраивать свои окружения несколькими способами. Существует системный файл настроек запуска (инициализации), каждый рабочий каталог также может иметь свои собственные специальные файлы инициализации. Наконец, могут быть использованы специальные функции `.First()` и `.Last()`.

Место расположения системного файла инициализации берется из значения переменной среды `R_PROFILE`. Если эта переменная не установлена, используется файл `Rprofile.site` в подкаталоге `etc` домашней директории R. Этот файл должен содержать команды, которые вы хотите исполнять каждый раз, когда запускаете R в вашей системе. Второй, индивидуальный профайл с именем `.Rprofile` может быть размещен в любом каталоге. Если R запускается в этом каталоге, то файл будет считан. Этот файл предоставляет индивидуальным пользователям контроль над их рабочим окружением и дает в различных рабочих каталогах возможность отличающихся процедур запуска. Если файла `.Rprofile` нет в каталоге запуска, тогда R ищет файл `.Rprofile` в пользовательском домашнем каталоге и использует его (если он существует).

Любая функция с именем `.First()` в любом из этих двух профайлов или в `.RData` имеет особый статус. Она автоматически выполнится в начале R-сессии и может использоваться для инициализации среды. Например, определение в примере ниже изменяет приглашение системы на `$` и устанавливает другие «полезности», которые будут использоваться по умолчанию.

```
.First <- function() {
# $ как приглашение
options(prompt="$ ", continue="+\t")
# формат чисел и область печати
options(digits=5, length=999)
# графический драйвер
x11()
# рисовать символом "+"
par(pch = "+")
# мои собственные функции
source(file.path(Sys.getenv("HOME"), "R", "mystuff.R"))
# присоединить пакет
```

```
library(MASS)
}
```

Вот последовательность, в которой исполняются описанные выше файлы:

1. `Rprofile.site`;
2. `.Rprofile`;
3. `.RData`;
4. `.First()`.

Определения в последующих файлах маскируют определения в предыдущих файлах.

Если определена функция `.Last()`, она выполняется в самом конце сессии:

```
.Last <- function() {
# небольшая мера безопасности
graphics.off()
cat(paste(date(),"\nAdios\n"))
}
```

Г.15. Графические функции

Графические средства являются важной и очень гибкой частью среды R. Можно использовать эти возможности для широкого спектра предопределенных статистических графиков, а также для того, чтобы создавать совершенно новые виды графиков.

Графические возможности могут быть использованы в интерактивном и пакетном режимах, но в большинстве случаев интерактивное использование более продуктивно. В момент запуска R включает драйвер графического устройства, который открывает специальное окно для отображения интерактивной графики. Хотя это делается автоматически, полезно знать, что в UNIX используется команда `X11()`, а в Windows — команда `windows()`.

Как только драйвер устройства запущен, рисующие команды R могут быть использованы для производства различных графических отображений и для создания совершенно новых типов изображений.

Команды рисования делятся на три основные группы:

- Функции рисования *высокого уровня* создают новый рисунок на графическом устройстве, возможно, вместе с осями, метками, названиями и т. д.

- Функции рисования *низкого уровня* добавляют дополнительную информацию к существующим рисункам, такую как дополнительные точки, линии и метки.

- *Интерактивные* графические функции позволяют интерактивно добавить либо получить информацию о существующем рисунке, используя графический манипулятор, такой как, например, мышь.

Кроме того, R ведет список графических параметров, которыми можно манипулировать, чтобы изменить свой рисунок.

Этот раздел описывает только то, что известно как «базовая» графика. Отдельная подсистема графики из пакета `grid` сосуществует с базовой — это более мощная, но более сложная в использовании система. Существует рекомендуемый пакет `lattice`, который построен на `grid` и обеспечивает способы для получения составных графиков.

Г.15.1. `plot()`

Функции рисования высокого уровня призваны создать законченный рисунок по данным, заданным как аргументы функции. При необходимости автоматически создаются оси, метки и заголовки. Команды рисования высокого уровня всегда начинают новый рисунок, стирая текущий в случае необходимости.

Одна из наиболее часто используемых функций рисования в R — это функция `plot()`. Это общая функция: вид производимого рисунка зависит от типа или класса первого аргумента.

```
> plot(x, y)
```

Если `x` и `y` являются векторами, `plot(x, y)` выдает коррелограмму `y` по `x`. Тот же эффект может быть произведен заданием одного аргумента (вторая форма) либо как списка, содержащего два элемента `x` и `y`, либо как матрицы из двух колонок:

```
> plot(x)
```

Если `x` является временным рядом, выводится график временного ряда. Если `x` является вектором чисел, выводится график значений вектора по индексу вектора. Если `x` — это комплексный вектор, выводится график мнимой части векторных элементов по реальным частям.

```
> plot(f)
> plot(f, y)
```

Если `f` является фактором, а `y` является вектором чисел, то первый вариант создаст столбчатую диаграмму по `f`; а второй вариант выведет ящики-с-усами `y` для каждого уровня `f`.

```
> plot(df)
> plot(~ expr)
> plot(y ~ expr)
```

Если `df` — это таблица данных, `y` — любой объект, `expr` — список имен объектов, разделенных «+» (например, `a + b + c`), то в первых двух вариантах выводятся графики распределения по переменным в таблице данных (первый вариант) или по числу имен объектов (второй вариант). Третий вариант рисует `y` по каждому объекту, указанному в `expr`.

Г.15.2. Отображение многомерных данных

R предусматривает весьма полезные функции, представляющие многомерные данные. Если `X` — матрица чисел или таблица данных, то команда

```
> pairs(X)
```

выводит матрицу попарных коррелограмм для всех переменных, столбцами `X`, то есть каждый столбец `X` отображается по всем другим столбцам `X` и результирующие $n \times (n-1)$ графиков будут выведены в виде матрицы, с расположенными симметрично относительно диагонали одинаковыми графиками.

Если рассматриваются три или четыре переменных, более полезной может быть функция `coplot()`. Если `a` и `b` — векторы чисел и `c` — вектор чисел или фактор (все одинаковой длины), тогда команда

```
> coplot(a ~ b | c)
```

производит ряд коррелограмм `a` по `b` для заданных значений `c`. Если `c` представляет собой фактор, это просто означает, что `a` выводится по `b` для каждой градации `c`. Когда `c` — численный вектор, то он делится на ряд непрерывных интервалов и для каждого интервала `a` отображается по `b` для значений `c` из данного интервала. Количество и положение интервалов можно контролировать с помощью аргумента `given.values=` функции `coplot()`. Функция `co.intervals()` полезна для выбора интервалов. Вы также можете использовать две переменные, например:

```
> coplot(a ~ b | c + d)
```

рисует коррелограмму `a` на `b` для каждого совместного интервала группировки по `c` и `d`.

Обе функции, `coplot()` и `pairs()`, принимают аргумент `panel=`, который может быть использован для настройки типа графика в каждой из панелей. По умолчанию применяется `points()` (коррелограмма), но путем указания других низкоуровневых графических функций как значения `panel=` можно выводить нужный тип графиков. Примером `panel`-функции, полезной для `coplots()`, является `panel.smooth()`.

Г.15.3. Другие графические функции высокого уровня

Вот некоторые примеры:

```
> qqnorm(x)
> qqline(x)
> qqplot(x, y)
```

Это графики сравнения распределений. Первый вариант рисует численный вектор `x` в сравнении с ожидаемым нормальным распределением квантилей, а второй добавляет прямую линию к такому графику, проводя ее через квартили. Третий вариант выводит квантили `x` по `y` для сравнения их распределений.

```
> hist(x)
> hist(x, nclass=n)
> hist(x, breaks=b, ...)
```

выводят гистограммы вектора чисел `x`. Как правило, выбирается разумное количество интервалов группировки, но можно их задать прямо аргументом `nclass`. Кроме того, точки разбиения можно указать точно аргументом `breaks`. Если указан аргумент `probability=TRUE`, столбцы вместо сумм представляют относительные частоты.

```
> dotchart(x, ...)
```

создает точечную диаграмму из данных, приведенных в `x`. В точечной диаграмме ось `y` дает метки данных из `x`, а ось `x` отображает их величину. Это позволяет, например, легко отобрать все записи данных, значения которых лежат в определенных диапазонах.

```
> image(x, y, z, ...)
> contour(x, y, z, ...)
> persp(x, y, z, ...)
```

рисуют графики трех переменных. График `image()` рисует сетку прямоугольников, используя различные цвета для отображения величины `z`, график `contour()` рисует горизонтали, представляющие уровни `z`, а график `persp()` рисует 3D-поверхность.

Г.15.4. Параметры функций высокого уровня

Есть ряд параметров, которые могут быть переданы высокоуровневым графическим функциям, например

```
..., add=TRUE
```

принуждает функцию действовать в качестве низкоуровневой графической функции, накладывая свой график на текущий (это работает только для некоторых функций).

```
..., axes=FALSE
```

подавляет вывод осей, что бывает полезно для того, чтобы добавить собственные оси с помощью функции `axis()`. По умолчанию `axes=TRUE`, что означает отображение осей.

```
..., log="x"
..., log="y"
..., log="xy"
```

Приводит `x`, `y` или обе оси к логарифмическому масштабу. Это сработает для многих, но не всех видов графиков.

Аргумент

```
..., type
```

контролирует тип выводимого графика, в частности

```
..., type="p"
```

выводит отдельные точки (принято по умолчанию),

```
..., type="l"
```

рисует линии,

```
..., type="b"
```

выводит точки, соединенные линиями вместе,

```
..., type="o"
```

рисует точки, перекрытые линиями,

```
..., type="h"
```

рисует вертикальные линии от точек до нулевого уровня оси,

```
..., type="s"
..., type="S"
```

рисует ступенчатую функцию. В первом варианте точка наверху, а во втором — внизу.

```
..., type="n"
```

не рисует ничего. Однако оси по-прежнему выводятся (это задано по умолчанию), и система координат создается в соответствии с данными. Идеально подходит для создания участков последовательностью низкоуровневых графических функций.

```
..., xlab="строка"
..., ylab="строка"
```

Подписи осей `x` и `y`.

```
..., main="строка"
```

Название всего графика большим шрифтом, расположено в верхней части рисунка.

```
..., sub="строка"
```

Подзаголовок меньшим шрифтом, расположенный чуть ниже оси `x`.

Г.15.5. Низкоуровневые графические команды

Иногда высокоуровневые графические функции не дают именно такой график, какой вам нужен. Можно использовать низкоуровневые графические команды, чтобы добавить дополнительную информацию (например, точки, линии или текст) к текущему рисунку.

Некоторые из наиболее полезных низкоуровневых функций:

```
> points(x, y)
> lines(x, y)
```

добавляют точки или связывающие линии к текущему графику. Для `plot()` аргументы типа `type=` также могут быть переданы в эти функции (по умолчанию `"p"` для `points()` и `"l"` для `lines()`).

```
> text(x, y, labels, ...)
```

добавляет текст в рисунок в точке, заданной `x` и `y`. Часто `labels` — целочисленный или символьный вектор, в этом случае `labels[i]` выводится в точке `(x[i], y[i])`. Значение по умолчанию — `1:length(x)`.

Эта функция часто используется в таком сочетании:

```
> plot(x, y, type="n")
> text(x, y, names)
```

Графический параметр `type="n"` подавляет точки, но создает оси, а функция `text()` поставляет специальные символы, которые заданы именами в символьном векторе для каждой из точек.

```
> abline(a, b)
> abline(h=y)
> abline(v=x)
> abline(lm.obj)
```

добавляют в текущий рисунок линии наклона `b`, отсекающие отрезок `a`.

«`h=y`» может быть использовано для задания `y`-координаты высоты горизонтальной линии, проходящей через рисунок, а «`v=x`» аналогичным образом задает вертикальную линию. «`lm.obj`» может быть списком с компонентами длины 2 (например, результатом функции подгонки модели), которые используются в качестве отсекающего отрезка и наклона (именно в таком порядке).

```
> polygon(x, y, ...)
```

строит многоугольник, определенный вершинами из `(x, y)`, можно дополнительно затенить его штриховкой или закрасить его, если графическое устройство позволяет закраску изображений.

```
> legend(x, y, legend, ...)
```

добавляет легенду к текущему графику в указанной позиции. Начертание символов, стиль линий, цвета и т. п. определяются метками в символьном векторе описания. Должен быть задан по меньшей мере один аргумент `v` (вектор той же длины, как описание) с соответствующими значениями единиц измерения, а именно:

```
> legend(..., fill=v)
```

так определяются цвета для заполнения прямоугольников,

```
> legend(..., col=v)
```

так — цвета, которыми рисуются точки или линии.

```
> legend(..., lty=v)
```

Стиль линии

```
> legend(..., lwd=v)
```

Ширина линии

```
> legend(..., pch=v)
```

Выводимые символы (символьный вектор)

```
> title(main, sub)
```

добавляет заголовок `main` в начало текущего рисунка и подзаголовок `sub` внизу.

```
> axis(side, ...)
```

добавляет оси к текущему рисунку со стороны, заданной первым аргументом (от 1 до 4, считая по часовой стрелке снизу). Другие аргументы контролируют расположение оси внутри или рядом с рисунком и позиции меток. Для добавления в дальнейшем пользовательских осей полезно вызывать `plot()` с аргументом `axes=FALSE`.

Низкоуровневые графические функции, как правило, требуют некоторой позиционной информации (например, `x`- и `y`-координаты), чтобы определить место для нового элемента рисунка. Координаты определяются в предыдущей высокоуровневой графической команде и выбираются на основе предоставленной информации.

Если нужны `x` и `y`, то достаточно задать один аргумент — список с элементами по имени `x` и `y`. Матрица с двумя столбцами также подходит для ввода. Таким путем функции типа `locator()` (см. ниже) могут использоваться для интерактивного определения позиции на рисунке.

Г.15.6. Математические формулы

В некоторых случаях бывает полезно добавить математические символы и формулы на график. Это можно сделать в R не символьной строкой в `text`, `mtext`, `axis` или `title`, а описанием выражения. Например, следующий код рисует формулу для биномиальной функции распределения:

```
> text(x, y, expression(paste(bgroup("(", atop(n, x), ")"),
+ p^x, q^{n-x})))
```

Более подробную информацию, включая полный перечень доступных возможностей, можно получить в R с помощью команд:

```
> help(plotmath)
> example(plotmath)
> demo(plotmath)
```

Г.15.7. Интерактивная графика

R также обеспечивает функции, которые позволяют пользователям получать или добавлять информацию на график с помощью мыши. Простейшей из них является функция `locator()`.

```
> locator(n, type)
```

ожидает щелчка левой кнопкой мыши. Это продолжается до тех пор, пока не будет выбрано `n` (по умолчанию 512) точек или нажата любая другая кнопка мыши. Аргумент `type` применим для вывода отмеченных точек и действует так же, как на высокоуровневые графические команды; по умолчанию вывод отсутствует. `locator()` возвращает координаты отдельных точек как список с двумя компонентами `x` и `y`.

`locator()` обычно вызывают без аргументов. Это особенно полезно для интерактивного выбора позиции для графических элементов, таких как описания и метки, когда трудно рассчитать заранее, куда надо пометить графический элемент. Например, чтобы поставить некоторый информативный текст вблизи точки-выброса, может быть полезна команда

```
> text(locator(1), "Outlier", adj=0)
```

(Команда `locator()` будет проигнорирована, если текущее устройство, например `postscript`, не поддерживает интерактивного взаимодействия.)

```
> identify(x, y, labels)
```

позволяет пользователю выделить любую из точек с определенными координатами `x` и `y` путем вывода поблизости соответствующего компонента `labels` (или индекс точки, если `labels` отсутствует). Возвращает индексы выбранных точек, когда нажимают другую кнопку мыши.

Иногда мы хотим идентифицировать именно точки на графике, а не их координаты. Например, мы можем пожелать, чтобы пользователь выбрал некоторые интересующие наблюдения на графическом дисплее, а затем обработать эти наблюдения разными способами. Задав ряд `(x, y)` координат двумя численными векторами `x` и `y`, мы могли бы использовать функцию `identify()` следующим образом:

```
> plot(x, y)
> identify(x, y)
```

Функция `identify()` сама не рисует, она просто позволяет пользователю переместить курсор мыши и нажать левую кнопку мыши вблизи точки. Если возле указателя мыши есть точка, она будет помечена своим индексом (то есть своей позицией в `x/y` векторах), выведенным поблизости.

Можно также использовать строки (например, название случая) в качестве подписи, используя аргумент `labels` для `identify()`, или вообще отключить выделение аргументом `plot = FALSE`. Когда работа с графиком прекращается (см. выше), `identify()` возвращает индексы выбранных точек; вы можете использовать эти индексы для извлечения отдельных точек из первоначальных векторов `x` и `y`.

Г.15.8. `par()`

При создании графики, в частности для выступлений или публикаций, R по умолчанию не всегда выводит именно то, что требуется. Однако можно настроить практически каждый аспект вывода, используя *графические параметры*. R поддерживает список из большого числа графических параметров, которые контролируют среди многих других такие вещи, как стиль линии, цвет, расположение рисунка и текста. Каждый графический параметр имеет имя (например, «`col`», который контролирует цвет) и значение (например, номер цвета).

Отдельный список графических параметров поддерживается для каждого активного устройства, и каждое устройство, когда оно инициализировано, по умолчанию имеет набор параметров. Графические параметры можно задать двумя способами: либо постоянно, затронув все графические функции, связанные с текущим устройством, либо временно, затрагивая только один вызов графической функции.

Функция `par()` используется для доступа к списку параметров и изменению графических параметров для текущего графического устройства.

```
> par()
```

без параметров возвращает список всех графических параметров и их значения для текущего устройства.

```
> par(c("col", "lty"))
```

Если аргумент — символьный вектор, команда возвращает только названные графические параметры.

```
> par(col=4, lty=2)
```

с именованными аргументами (или одним общим списком аргументов) устанавливает значения для соответствующих графических параметров и возвращает первоначальные значения параметров как список.

При настройке параметров графики функцией `par()` изменение значений параметров постоянно в том смысле, что все будущие вызовы графических функций (от текущего устройства) будут зависеть от нового значения.

Заметим, что вызовы `par()` всегда влияют на значения графических параметров глобально, даже когда `par()` вызывается внутри функции. Это часто нежелательное поведение — как правило, мы хотим установить некоторые параметры графики для некоторых рисунков, а затем восстановить исходные значения, чтобы не повлиять на пользовательскую сессию R. Вот как можно восстановить некоторые параметры, временно сохраняя результат `par()`:

```
> oldpar <- par(col=4, lty=2)
[... строим график ...]
> par(oldpar)
```

Чтобы сохранить и восстановить вообще все настраиваемые графические параметры, используем

```
> oldpar <- par(no.readonly=TRUE)
[... строим график ...]
> par(oldpar)
```

Графические параметры часто могут быть переданы в графические функции как именованные аргументы. Это действует так же, как передача аргументов в функции `par()`, за исключением того, что изменения действуют только в течение срока действия вызова функции. Например:

```
> plot(x, y, pch="+")
```

производит коррелограмму, используя знак «+» и не изменяя символ печати по умолчанию для будущих графиков.

Г.15.9. Список графических параметров

В следующих подразделах детализируются многие из широко используемых графических параметров. В справочной документации сведения по функции `par()` представлены кратко, здесь дан несколько более подробный вариант.

Графические параметры будут представлены в следующей форме:

```
..., имя = значение
```

(Заметим, что `axes` — это не графический параметр, а аргумент для нескольких методов `plot`; см. документацию к `xaxt` и `yaxt`.)

Графики в R состоят из точек, линий, текста и полигонов (заполненных областей). Графические параметры существуют для того, чтобы контролировать, как эти графические элементы будут нарисованы, а именно:

```
..., pch="+"
```

Символ, который будет использоваться для рисования точек. Умолчание различно для различных графических устройств, однако это, как правило, кружок. Выводимые точки, как правило, немного выше или ниже соответствующей позиции, за исключением ситуации, когда как символ вывода используется `"."`.

```
..., pch=4
```

Когда `pch` задается как целое число между 0 и 25 включительно, получается специальный символ. Чтобы увидеть, какие это символы, используйте команду

```
> legend(locator(1), as.character(0:25), pch = 0:25)
```

Символы с 21 по 25 могут показаться дублирующими предыдущие символы, но у них разные цвета: см. помощь по `points()` и приведенные примеры.

Кроме того, `pch` может быть символом или числом в диапазоне `32:255`, представляющем символ в текущем шрифте.

```
..., lty=2
```

Это — тип линии. Альтернативные стили линии не поддерживаются всеми графическими устройствами (и различаются у разных устройств), но тип 1 — это всегда сплошная линия, линия типа 0 всегда невидима, линии типов 2 и более являются точечными или штриховыми, или сочетаниями обоих типов.

```
..., lwd=2
```

Ширина линии. Желаемая ширина линий, кратная «стандартной» толщине линии. Влияет на линии осей, а также линий, выводимых командой `lines()` и другими. Не все устройства это поддерживают, многие имеют ограничения на ширину.

```
..., col=2
```

Это цвета, которые будут использоваться для точек, линий, текста, заполненных областей и изображений. Номер из текущей палитры (см. `?palette`), или название цвета.

```
..., col.axis=
..., col.lab=
..., col.main=
..., col.sub=
```

Это цвета, которые будут использоваться соответственно для аннотации осей, `x` и `y` меток, заголовка и подзаголовка.

```
..., font=2
```

Параметр, который определяет, какой шрифт использовать для всего текста на графике. Если возможно, устройство вывода организует это так, что 1 соответствует простому тексту, 2 — выделенному, 3 — курсиву, 4 — выделенному курсиву, а 5 — символьному шрифту (включая греческие буквы).

```
..., font.axis=
..., font.lab=
..., font.main=
..., font.sub=
```

Шрифты, которые будут использоваться соответственно для аннотации осей, `x` и `y` меток, заголовка и подзаголовка.

```
..., adj=-0.1
```

Юстировка текста относительно позиции вывода. 0 означает юстировать влево, 1 означает юстировать вправо и 0.5 — центрировать горизонтально относительно позиции вывода. Фактическое значение означает долю текста, которая появляется с левой стороны от позиции вывода. Значение -0.1 оставляет между текстом и позицией вывода пробел в 10% от ширины текста.

```
..., cex=1.5
```

Масштаб символов. Значение — это желаемый размер символов текста (включая выводимые символы) по отношению к размеру шрифта по умолчанию.

```
..., cex.axis=
..., cex.lab=
..., cex.main=
..., cex.sub=
```

Масштаб символов, который будет использоваться для аннотации оси, `x` и `y` меток, заголовков и подзаголовков.

Оси и шкалы

Многие высокоуровневые рисунки R имеют оси, вы также можете самостоятельно конструировать оси низкоуровневыми графическими функциями `axis()`. Оси имеют три основных компонента: линию оси (стиль линии, который контролируется графическим параметром `lty`), шкалу (которая обозначает деление линии оси единицами измерения) и метки шкалы (которые обозначают единицы измерения). Две последние компоненты можно настроить следующими параметрами графики:

```
..., lab=c(5, 7, 12)
```

Первые два числа — это желаемое количество интервалов по `x` и `y` осям соответственно. Третье — желаемая длина меток оси, в символах (включая запятую).

```
..., las=1
```

Ориентация меток оси. 0 означает «всегда параллельно оси», 1 означает «всегда горизонтально» и 2 — «всегда перпендикулярно оси».

```
..., mgp=c(3, 1, 0)
```

Это — позиции компонентов оси. Первый компонент — это расстояние от метки оси до позиции оси, в текстовых строках. Второй компонент – это расстояние до меток шкалы, и окончательный компонент – это расстояние от позиции оси до линии оси (обычно ноль). Положительные числа «уводят» за пределы региона рисования, отрицательные числа — внутрь региона.

```
..., tck=0.01
```

Длина меток шкалы как доля размера региона рисования. Когда `tck` невелик (менее 0.5), метки шкалы на осях `x` и `y` одинакового размера. Значение 1 дает сетку линий. Отрицательные значения дают метки за пределами региона рисования. Используйте `tck = 0.01` и `mgp = c(1, -1.5, 0)` для внутренних меток шкалы.

```
..., xaxs="r"
..., yaxs="i"
```

Стили для осей `x` и `y` соответственно. Со стилями `"i"` (внутренний) и `"r"` (по умолчанию) метки шкалы всегда находятся внутри диапазона данных, однако стиль `"r"` оставляет небольшое пространство на краях. (S имеет и другие стили, которые не были реализованы в R).

Г.15.10. Края рисунка

Одиночный рисунок в R называется «изображение» и включает регион рисования, окруженный полями (возможно, содержащими метки осей, заголовки и т. д.).

Графические параметры, контролирующие формат изображения:

```
..., mai=c(1, 0.5, 0.5, 0)
```

Определяет ширину соответственно нижнего, левого, верхнего и правого полей; измеряется в дюймах.

```
..., mar=c(4, 2, 2, 1)
```

Аналогично `mai`, за исключением того, что единица измерения — это текстовые строки.

`mar` и `mai` эквивалентны в том смысле, что настройка одного вызывает изменение значения другого. Часто значения по умолчанию выбраны для этого параметра слишком большими; правый край требуется редко, и верхнее поле не нужно, если нет названия. Нижнее и левое поля должны быть достаточными для размещения оси и меток масштаба.

Кроме того, размер по умолчанию выбирается без учета размера устройства вывода: например, использование устройства `postscript()` с параметром `height=4` приведет к рисунку, 50% которого составляют поля, если используются `mai` или `mar` по умолчанию.

Когда выводятся составные изображения (см. ниже), поля уменьшаются, однако этого может быть недостаточно, когда на одной странице много изображений.

Г.15.11. Составные изображения

R позволяет создать массив из $n \times m$ изображений на одной странице. Каждое изображение имеет свои поля, а массив изображений может быть окружен наружными полями.

Графические параметры, связанные с составными изображениями,— это:

```
..., mfcol=c(3, 2)
..., mfrow=c(2, 4)
```

Они устанавливают размер массива составных изображений. Первый аргумент у них — это количество строк, а второй — число столбцов. Единственная разница между ними в том, что установление `mfcol` выводит изображения по колонкам, а `mfrow` заполняет массив построчно.

Установка любого из этих показателей может сократить базовый размер символов и текста (контролируется при помощи `par("cex")`) и `pointsize` устройства вывода. В формате с двумя строками и столбцами базовый размер уменьшается на 0.83, если есть три или больше строки или столбца, коэффициент уменьшения равен 0.66.

```
..., mfg=c(2, 2, 3, 2)
```

Это — позиция текущего изображения в составном окружении. Первые два номера — это строка и столбец текущего составного изображения; последние два числа — это строки и столбцы в составном массиве изображений. Используйте этот параметр для перехода между изображениями в массиве. Для отображения разноразмерных изображений на одной и той же странице можно даже использовать отличающиеся от правильных значений значения для последних двух аргументов.

```
..., fig=c(4, 9, 1, 4)/10
```

Позиция текущего изображения на странице. Значения параметров — это позиции левого, правого, верхнего и нижнего краев соответственно, в процентах от размера страницы, измеряемого от нижнего левого угла. Пример соответствует изображению в нижнем правом углу страницы. Используйте этот параметр для произвольного позиционирования изображений на странице. Если вы хотите добавить изображение к текущей странице, используйте `new=TRUE`.

```
..., oma=c(2, 0, 3, 0)
..., omi=c(0, 0, 0.8, 0)
```

Размер внешних полей. Как `mar` и `mai`, первое измеряется в текстовых строках, а второе — в дюймах, начиная с нижнего поля и по часовой стрелке.

Внешнее поле особенно полезно для заметок на полях и т. д. Текст может быть добавлен к внешнему полю функцией `mtext()` с аргументом `outer=TRUE`. По умолчанию внешних полей не существует.

Более сложные составные изображения могут быть выведены функциями `split.screen()` и `layout()`, а также с использованием пакетов `grid` и `lattice`.

Г.15.12. Устройства вывода

R может генерировать графику (разного качества) почти для любого типа дисплея или устройства печати, но сначала он должен быть проинформирован, с каким типом устройства нужно взаимодействовать. Это делается путем запуска «драйвера устройства вывода». Смысл драйвера устройства вывода состоит в преобразовании графических команд R («нарисовать линию», например) к такой форме, которую конкретное устройство вывода может понять.

Драйвер устройства вывода запускают, вызывая функцию драйвера устройства. Для каждого устройства графического вывода существует единственная функция драйвера устройства (введите `help(Devices)`) для получения полного списка. Например, ввод команды `postscript()` перенаправляет весь последующий графический вывод на принтер, преобразуя его в формат PostScript. Некоторые широко используемые драйверы устройств вывода:

`X11()` служит для использования с X11 — оконной системой UNIX-подобных ОС;

`windows()` нужна для использования в Windows;

`quartz()` используется в Mac OS X;

`postscript()` работает для печати на принтерах PostScript, а также создания графических файлов PostScript;

`pdf()` выводит PDF-файл, который может быть включен в другие PDF-файлы;

`png()` выводит растровый файл PNG;

`jpeg()` выводит растровый файл JPEG (этот формат лучше использовать только для рисунков-изображений).

Когда вы закончите работать с устройством вывода, убедитесь, что остановили драйвер устройства вывода командой `dev.off()`. Это гарантирует, что устройство остановится правильным образом, например в случае печатного устройства это гарантирует, что каждая страница будет отправлена на принтер.

Г.15.13. Несколько устройств вывода одновременно

При профессиональном использования R зачастую бывает полезно иметь несколько графических устройств вывода, используемых одновременно. Конечно, в любой отдельный момент времени только одно графическое устройство может принимать графические команды: оно обозначается как «текущее» устройство. Когда несколько устройств открыты одновременно, они образуют пронумерованную последовательность с именами, определяющими тип устройства вывода в каждой из позиций.

Каждый новый вызов функции драйвера устройства вывода открывает новое графическое устройство, что расширяет на одну позицию список устройств вывода. Это устройство становится текущим устройством вывода.

```
> dev.list()
```

Эта команда возвращает количество и наименование всех активных устройств вывода. Устройство в позиции 1 списка всегда является null-устройством, которое вообще не принимает графических команд.

```
> dev.next()
> dev.prev()
```

возвращает номер и название, соответственно, последующего или предыдущего графического устройства.

```
> dev.set(which=k)
```

может быть использована для изменения текущего графического устройства на устройство в позиции `k` списка устройств вывода. Возвращает номер и метку устройства.

```
> dev.off(k)
```

закрывает графическое устройство в позиции `k` списка устройств вывода. Для некоторых устройств, таких как `postscript`, это будет либо немедленно распечатывать файл, либо корректно завершать файл для последующей печати, в зависимости от того, как было инициировано устройство.

```
> dev.copy(device, ..., which=k)
> dev.print(device, ..., which=k)
```

делает копию устройства `k`. Здесь устройство — это функция драйвера устройства, например `postscript` (без скобок), в случае необходимости с дополнительными аргументами, назначенными посредством «`...`».

`dev.print()` работает аналогично `dev.copy()`, но копируемое устройство немедленно закрывается.

```
> graphics.off()
```

закрывает все графические устройства, кроме `null`-устройства.

Г.16. Пакеты

Все функции и данные R хранятся в пакетах. Их содержимое становится доступным, лишь когда пакет загружается. Это делается как для производительности, так и для помощи разработчикам, которые предохранены от конфликтов имен.

Чтобы узнать, какие пакеты установлены в вашей инсталляции, наберите команду `library()` без аргументов. Для загрузки конкретного пакета (например, пакета `boot`) используйте такую команду:

```
> library(boot)
```

Пользователи, подключенные к Интернету, могут использовать функции `install.packages()` и `update.packages()` для установки и обновления пакетов.

Чтобы узнать, какие пакеты уже загружены, используйте

```
> search()
```

для отображения списка. Некоторые пакеты могут быть загружены, но не доступны в этом списке: они будут включены в список, выводимый

```
> loadedNamespaces()
```

Чтобы увидеть список всех доступных тем в системе помощи, касающихся установленных пакетов, используйте `help.start()`. Эта команда запускает HTML-систему помощи, затем надо перейти на список пакетов в разделе «Ссылки».

Г.16.1. Стандартные и сторонние пакеты

Стандартные пакеты содержат базовые функции, которые позволяют R работать, а также наборы данных и стандартные статистические и графические функции, которые описаны выше. Они должны быть автоматически доступны в любой инсталляции R.

Существуют тысячи сторонних пакетов для R, написанных различными авторами. Некоторые из этих пакетов реализуют специализированные статистические методы, другие предоставляют доступ к данным или оборудованию, третьи призваны дополнять руководства. Некоторые из них (рекомендованные, «recommended») распространяются с каждым бинарным дистрибутивом R. Большинство из них доступны для загрузки с CRAN (`http://CRAN.R-project.org` и его зеркала) и других хранилищ, например `http://www.bioconductor.org`.

Г.16.2. Пространство имен пакета

Пакеты должны иметь пространства имен. Пространство имен делает три вещи: оно позволяет автору пакета скрыть функции и данные, предназначенные только для внутреннего использования, предотвращает функции пакета от поломки, когда пользователь (или автор другого пакета) выбирает название, совпадающее с названием из пакета, а также предусматривает способ сослаться на объект в рамках конкретного пакета.

Например, `t()` — это функция транспонирования в R, но пользователи могут определять свои собственные функции с именем «`t`». Пространство имен предохраняет переопределения пользователя и прерывает все функции, которые пытаются транспонировать матрицу.

Есть два оператора, которые работают с пространством имен. Двойное двоеточие — оператор «`::`» — выбирает определения из определенного пространства имен. В примере выше функция транспонирования всегда будет доступна как `base::t()`, поскольку оно определяется в

базовом пакете. Таким способом можно получить только функции, которые экспортируются из пакета.

Тройное двоеточие, оператор «`:::`», можно увидеть в некоторых местах кода R — он действует подобно оператору двойного двоеточия, но обеспечивает доступ к маскированным объектам. Пользователи чаще используют функцию `getAnywhere()`, которая ищет сразу в нескольких пакетах.

Пакеты зачастую взаимосвязаны, и загрузка одного может потребовать другие, которые будут автоматически загружены. Операторы двоеточия, описанные выше, вызывают автоматическую загрузку соответствующего пакета. Когда пакеты загружаются автоматически, они не добавляются в список поиска.

Приложение Д

Краткий словарь языка R

Здесь мы собрали примерно пятьдесят самых-самых, на наш взгляд, необходимых команд, операторов и обозначений. Список мы постарались сделать как можно короче, для того чтобы эти команды было легче запомнить.

`?` 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

Приложение E

Краткий словарь терминов

Этот очень краткий словарь поможет найти для самых распространенных терминов и понятий статистики английский эквивалент, соответствующую команду R и краткое пояснение. Его можно использовать как своего рода обратный словарь команд, который может оказаться полезным тогда, когда известно, что нужно делать, но неизвестно, какую команду R использовать.

p-значение – p-value – вероятность получить оцениваемую характеристику при условии, что нулевая гипотеза верна; если это значение ниже определенного порога, то нулевую гипотезу надо отвергнуть (о статистических гипотезах рассказано в главе о двумерных данных).

Автокорреляция – autocorrelation – `acf()` – корреляция между последовательными значениями временного ряда.

Авторегрессия интегрированного скользящего среднего – Autoregressive Integrated Moving Average, ARIMA – `arima()` – метод построения модели временного ряда для составления прогноза.

Анализ главных компонент – principal component analisys – `princomp()`, `prcomp()` – метод многомерного анализа, «проецирующий» многомерное облако на плоскость компонент.

Биномиальный тест – binomial test – `binom.test()` – позволяет проверить, различаются ли пропорции.

График-гистограмма – histogram – `hist()` – диаграмма для отображения частоты встречаемости значений в выборке.

График-коррелограмма, диаграмма рассеяния – scatterplot – `plot(x,y)` – график, показывающий соотношение двух признаков в выборке.

График нормального распределения – normal distribution graph – `plot(density(rnorm(1000000)))` – «колокол», «шляпа» (рис. 72).

График сравнения квантилей – quantile comparison plot – `qqplot()` – график, показывающий соотношение квантилей в двух выборках

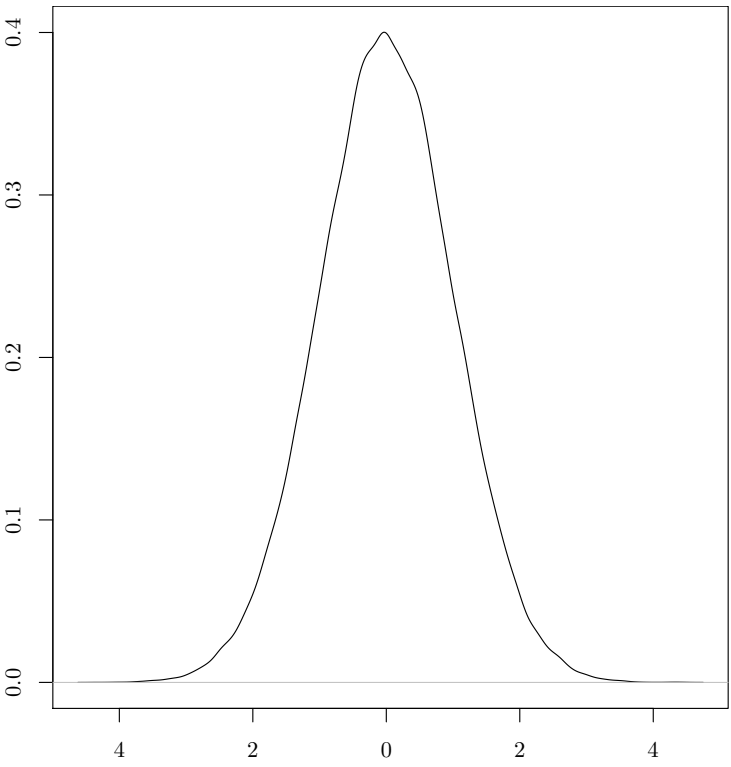


Рис. 72. График нормального распределения

(используется, например, при сравнении выборочного распределения с нормальным).

График «стебель-с-листьями», текстовая гистограмма – stem-anf-leaf plot – `stem()` – псевдографик, показывающий частоту встречаемости значений в выборке.

График «ящик-с-усами», боксплот – boxplot – `boxplot()` – диаграмма для отображения основных характеристик одной или нескольких выборок.

Дисперсионный анализ – analysis of variance, ANOVA – `anova()` – семейство параметрических тестов, которые используются для сравнения нескольких выборок.

Дисперсия – variance – `var()` – усредненная разница между средним значением и всеми остальными значениями выборки.

Доверительный интервал – confidence interval – промежуток, в котором может находиться значение какого-нибудь параметра (средней, медианы и т. п.).

Информационный критерий Акаике – Akaike's Information Criterion, AIC – `AIC()` – критерий, оценивающий оптимальность модели; наиболее оптимальная модель обычно соответствует минимальному AIC.

Квантиль – quantile – `quantile()` – возвращает значения квантилей (по умолчанию 0, 25, 50,75 и 100%) для выборки.

Кластерный анализ – cluster analisys – `hclust()` – метод визуализации сходства/различия объектов в выборке в виде дерева (дендрограммы).

Корреляционная матрица – correlation matrix – `cor()` – возвращает коэффициенты корреляции для всех пар выборок.

Корреляционный анализ – correlation analysis – `cor.test()` – группа методов, служащих для определения силы соответствия между двумя выборками.

Линейный дискриминантный анализ – linear discriminant analysis – `lda()` – метод многомерного анализа, позволяет создать классификацию на основе тренировочной выборки.

Линейный регрессионный анализ – linear regression – `lm()` – исследует характер линейной связи между объектами.

Матрица различий, Матрица расстояний (например, в кластерном анализе, в многомерном шкалировании) – distance matrix – `dist()`, `daisy()`, `vegdist()` – вычисляет расстояние (различие) между объектами по набору признаков по заданному алгоритму.

Медиана – median – `median()` – значение, отсекающее половину упорядоченной выборки.

Межквартильный разброс – interquartile range – `IQR()` – расстояние между вторым и четвертым квартилями, устойчивый (робастный) метод измерения разброса данных.

Многомерное шкалирование, анализ главных координат – multidimensional scaling, MDS – `cmdscale()` – строит подобие географической карты на основе матрицы расстояний.

Непараметрический – non-parametric – не связанный предположениями о типе распределения, пригодный для анализа произвольно распределенных данных.

Нормальное распределение – normal distribution – `rnorm()` – главное распределение статистики, возникает, например, если долго стрелять в мишень, а потом подсчитать все расстояния до «десятки», основа параметрических методов.

Параметрический – parametric – соответствующий нормальному распределению, пригодный для анализа нормального распределенных данных.

Распределение – distribution – «внешний вид» данных; *теоретическое распределение* указывает, как должны выглядеть данные; *распределение выборки* — как данные выглядят на самом деле.

Среднее арифметическое – arithmetic mean, mean, average – `mean()` – сумма всех значений выборки, деленная на ее объем.

Стандартная ошибка – standard error, SE – `sd(x)/sqrt(length(x))` – стандартное отклонение, нормализованное по объему выборки.

Стандартное отклонение, среднеквадратическое отклонение – standard deviation – `sd()` – корень квадратный из дисперсии.

Столбчатая диаграмма – bar plot – `barplot()` – диаграмма для отображения нескольких числовых значений (например, подсчетов).

Тест (критерий) Вилкоксона (Уилкоксона) – Wilcoxon test – `wilcox.test()` – используется для сравнения медиан одной или двух выборок, непараметрический аналог t-теста.

Тест (критерий) Колмогорова-Смирнова – Kolmogorov-Smirnov test – `ks.test()` – используется для сравнения двух распределений, в том числе сравнения распределения выборки с нормальным.

Тест (критерий) Краскала-Уоллиса – Kruskal-Wallis test – `kruskal.test()` – используется для сравнения нескольких выборок, непараметрический аналог дисперсионного анализа.

Тест (критерий) Манна-Уитни – Mann-Whitney test – `wilcox.test()` – см. тест (критерий) Уилкоксона.

Тест (критерий) Стьюдента, t-тест – t-test – `t.test()` – семейство параметрических тестов, которые используются для сравнения средних одной или двух выборок.

Тест (критерий) Фишера, F-тест – F-test – `var.test()` – параметрический тест, который используется для сравнения дисперсий в двух выборках.

Тест (критерий) хи-квадрат, критерий согласия, тест Пирсона – Chi-squared test – `chisq.test()` – проверяет, есть ли соответствие между строками и столбцами в таблице сопряженности.

Тест (критерий) Шапиро-Уилкса – Shapiro-Wilk test – `shapiro.test()` – тест для проверки гипотезы о нормальности распределения выборки.

Тест пропорций – proportional test – `prop.test()` – проверяет, в частности, одинаковы ли пропорции.

Типы данных – data types – различные типы данных, которые используют в статистическом анализе:

Интервальные – measurement:

Непрерывные – continuous;

Дискретные – meristic, discrete, discontinuous;

Шкальные – ranked, ordered;

Номинальные – categorical, nominal.

Формулы моделей – model formulae – `formula()` – способ краткого описания модели для статистического анализа:

`отклик ~ воздействие`: формула анализа регрессии;

`отклик ~ воздействие1 + воздействие2`: формула множественной регрессии;

`отклик ~ фактор`: формула однофакторного дисперсионного анализа;

`отклик ~ фактор1 + фактор2`: формула многофакторного дисперсионного анализа;

`отклик ~ воздействие * фактор`: формула анализа ковариаций, раскрывается в формулу «`отклик ~ воздействие + воздействие : фактор`».

Знаки, используемые в формулах моделей:

`.` все возможные факторы и воздействия (*предикторы* модели);

`+` добавляет фактор или воздействие;

`-` убирает фактор или воздействие;

`:` взаимодействие;

`*` все логически возможные комбинации факторов и воздействий;

`/` вложение, «`фактор1 / фактор2`» значит, что `фактор2` вложен в `фактор1` (скажем, улица «вложена» в округ, округ в город);

`|` условие, «`фактор1 | фактор2`» значит «разбить `фактор1` по уровням `фактор2`»;

`I()` вернуть всему, что в скобках, обычный арифметический смысл.

Литература

К сожалению, русскоязычных книг по статистике, где статистические проблемы освещались бы простым языком, довольно мало. Надеемся, что наш труд поможет в какой-то мере исправить ситуацию.

Волкова П. А., Шипунов А. Б. Статистическая обработка данных в учебно-исследовательских работах. — М.: Форум, 2012. — 96 с.

Гланц С. Медико-биологическая статистика. — М.: Практика, 1998. — 459 с.

Кендэл М. Временные ряды. — М.: Финансы и статистика, 1981. — 191 с.

Кимбл Г. Как правильно пользоваться статистикой. — М.: Финансы и статистика, 1982. — 294 с.

Любищев А. А. Дисперсионный анализ в биологии. — М.: Изд-во Моск. ун-та, 1986. — 200 с.

Плавинский С. Л. Введение в биостатистику для медиков. — М.: Акварель, 2011. — 584 с.

Петри А., Сэбин К. Наглядная статистика в медицине. — М.: Гэотар-Мед, 2002. — 144 с.

Тьюки Дж. Анализ результатов наблюдений. Разведочный анализ. — М.: Мир, 1981. — 695 с.

Тюрин Ю. Н., Макаров А. Л. Анализ данных на компьютере. — М.: ИНФРА-М, 1995. — 284 с.

Факторный, дискриминантный и кластерный анализ. — М.: Финансы и статистика, 1989. — 215 с.

Cleveland W. S. The elements of graphing data. — USA: Wandsworth Advanced Books and Software, 1985. — 323 p.

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.

Gordon A. D. Classification. — USA: Chapman & Hall/CRC, 1999. — 256 p.

Marriott F. H. C. The interpretation of multiple observations. — USA, England: Academic Press, 1974. — 117 p.

McKillup S. Statistics explained. An introductory guide for life scientists. — England: Cambridge University Press, 2005. — 267 p.

R Development Core Team. R: A language and environment for statistical computing. — R Foundation for Statistical Computing, Vienna, Austria.

Rowntree D. Statistics without tears. — England: Clays, 2000. — 195 p.

Sokal R. R., Rolf F. J. Biometry. The principles and practice of statistics in biological research. — USA: W.H. Freeman and Company, 1995. — 887 p.

van Emden H. Statistics for terrified biologists. — USA: Blackwell Publishing, 2008. — 343 p.

Venables W. N., Ripley B. D. Modern applied statistics with S. 4th ed. — USA: Springer, 2002. — 495 p.

Предметный указатель

Об авторах

Шипунов Алексей Борисович — биолог, кандидат биол. наук, доцент кафедры биологии Университета Майнота (США), русский переводчик R.

Балдин Евгений Михайлович — физик, кандидат физ.-мат. наук, Институт ядерной физики им. Г. И. Будкера, преподает в Новосибирском государственном университете, ведет колонку в журнале «Linux Format».

Волкова Полина Андреевна — биолог, кандидат биол. наук, учитель биологии в профильных классах Московской гимназии на Юго-Западе № 1543.

Коробейников Антон Иванович — математик, кандидат физ.-мат. наук, ассистент кафедры статистического моделирования математико-механического факультета Санкт-Петербургского Государственного Университета.

Назарова София Александровна — биолог, Санкт-Петербургский государственный университет, Российский государственный педагогический университет им. А. И. Герцена.

Петров Сергей Валерьевич — кандидат медицинских наук, магистр технических наук, доцент Гродненского государственного университета имени Янки Купалы.

Суфиянов Вадим Гарайханович — математик, кандидат физ.-мат. наук, доцент кафедры механики и прикладной информатики факультета прикладной математики Ижевского Государственного Технического Университета.