МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
ФГАОУ ВО «Крымский федеральный университет имени В. И. Вернадского»

ТАВРИЧЕСКАЯ АКАДЕМИЯ

Факультет математики и информатики

Анафиев А. С.

Учебно-методическое пособие
по дисциплине
«Системы и методы принятия решений»

ПРИМЕНЕНИЕ ЯЗЫКА «R» К РЕШЕНИЮ ЗАДАЧ КЛАССИФИКАЦИИ: ПРИМЕРЫ

Симферополь
2016

Анафиев А. С.

Учебно-методическое пособие по дисциплине «Системы и методы принятия решений». Применение языка «R» к решению задач классификации: примеры / А. С. Анафиев — ФГАОУ ВО «Крымский федеральный университет имени В. И. Вернадского». — Симферополь, 2016. — 40 с.

Пособие содержит основы языка программирования R, примеры программ решения задач классификации на языке R. Предназначено для обучающихся направления 01.03.02 Прикладная математика и информатика.

Рекомендовано к печати учебно-методическим советом Крымского федерального университета имени В. И. Вернадского
от 23 декабря 2016 г., протокол № 4

1. ЯЗЫК R

Язык R — язык для работы с данными: статистическая обработка данных, продвинутое математическое моделирование, визуализация данных. R является очень популярным в США и западной Европе, а в последнее время приобретает все большее распространение и в России. Его используют многие серьезные компании по всему миру.

Язык R — объектно-ориентированный язык программирования. Каждый объект принадлежит некоторому классу, который описывает данные и методы для работы с ними (данными). В зависимости от класса один и тот же метод может выводить различные результаты (например, функция `plot(x)` выводит различные результаты, когда `x` является регрессией и когда `x` является вектором). Теоретически всё что угодно может быть сохранено как объект.

Язык R можно бесплатно загрузить с официального сайта www.r-project.org под любую популярную операционною систему: Windows, Unix-системы и macOS.

После установки для запуска интерпретатора R достаточно выполнить в терминале команду:

```
$ R
```

результатом которой будет нечто вроде этого:

```
R version 3.2.4 (2016-03-10) -- "Very Secure Dishes"
Copyright (C) 2016 The R Foundation for Statistical
Computing
Platform: x86_64-apple-darwin13.4.0 (64-bit)

R -- это свободное ПО, и оно поставляется безо всяких
гарантий.
Вы вольны распространять его при соблюдении некоторых
условий.
Введите 'license()' для получения более подробной
информации.
```

```
R -- это проект, в котором сотрудничает множество
разработчиков.
Введите 'contributors()' для получения дополнительной
информации и 'citation()' для ознакомления с правилами
упоминания R и его пакетов в публикациях.

Введите 'demo()' для запуска демонстрационных программ,
'help()' -- для получения справки, 'help.start()' -- для
доступа к справке через браузер.
Введите 'q()', чтобы выйти из R.
```

Как видно из выше приведенного листинга для выхода необходимо использовать команду

```
> q()
```

для получения подробной информации (справки) о любой функции, необходимо выполнить команду:

```
> help(<имя функции>)
```

либо

```
> ?<имя функции>
```

Кроме функции `help()`, полезной, если мы не знаем точного названия функции, может оказаться команда

```
> help.search("vector")
```

результатом которой будет список команд, свойственных «векторам», с кратким описанием.

Для выполнения команд достаточно запустить среду R и выполнять в окне программы последовательность команд одну за другой. Однако, это не всегда удобно, более правильный подход — это создание *скриптов* (программ, которые загружаются и интерпретируются R). *Одной из основ правильной работы в R является создание скриптов даже в самых простых ситуациях*.

Загрузить некоторый скрипт можно при помощи команды:

```
> source("путь к файлу скрипта", echo = TRUE)
```

Скрипт R можно выполнить и не запуская саму среду, для этого в командной строке достаточно выполнить команду:

```
$ R -q --no-save < "имя файла-скрипта" > "имя файла-результата"
```

Одним из важных преимуществ R является наличие для него многочисленных расширений (пакетов) практически для решения любой задачи обработки данных, которые можно легко скачать с официального онлайн-репозитория — **CRAN** (http://cran.r-project.org/) и установить с помощью команды:

```
install.packages()
```

При установке R автоматически устанавливаются так называемые базовые пакеты, без которых система просто не работает (например, это такие пакеты, как `base`, `grDevices`), и некоторые «рекомендованные» пакеты (например, `cluster` (для решения задач кластерного анализа), `nlme` (для анализа нелинейных моделей) и д. р.).

1.1. Основные особенности языка R

R — регистрозависимый язык, т. е., например, символы «A» и «a» могут обозначать разные объекты.

Для присваивания используется символы «<-» или «->» (можно также использовать традиционное « = »).

```
2 + 3 -> x -> y                 # x = 5; y = 5
z <- x + y                      # z = 10
z = z * z                       # z = 100
z <- x + y -> t                 # z = x+y и t = x+y
sum <- (function(x, y) x + y) # Определить функцию sum
```

Аргументы функций передаются в круглых скобках через запятую.

```
func(x, y)      # вызов функции с двумя аргументами x и y
f(g(x, y), y)  # Суперпозиция функций f и g
```

Для получения списка аргументов функции следует использовать функцию:

```
> args(q)
function (save = "default", status = 0, runLast = TRUE)
NULL
```

Как видно по результату функции `args`, она выдаёт информацию не только о том, какие аргументы имеют значения по умолчанию, но и сами эти значения.

Некоторые аргументы могут иметь имена, благодаря чему их можно перечислять не по порядку, а по имени. Имена можно сокращать вплоть до одной буквы, но только если нет других аргументов, которые от такого сокращения станут неразличимы. При перечислении аргументов по порядку имена можно опускать:

```
> args(round)
function (x, digits = 0)
NULL

> round(pi) # Использовали значение по умолчанию для
“digits”
[1] 3

> round(pi, 3) # Прямой порядок аргументов
[1] 3.142

> round(pi, digits = 10) # с использованием имени аргумента
[1] 3.141593

> round(pi, d = 5) # с использованием сокращенного имени
[1] 3.14159

> round(digits = 5, pi) # вызов с другим порядком
аргументов
[1] 3.14159
```

Как вы успели заметить, для однострочных комментариев используется символ #:

```
# Комментарий
```

Команды разделяются точкой с запятой «;» или символом перевода на новую строку:

```
1:10 -> a; mean(a);
```

или

```
1:10 -> a
mean(a)
```

В самом простом случае R можно использовать как «продвинутый» калькулятор:

```
> # Использование R как калькулятора

> 1 + 2 + 3
[1] 6

> exp(1)^exp(1) # e в степени e
[1] 15.15426

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

Обратим внимание на единицу в квадратных скобках (`[1]`) — это индекс элемента вектора. Дело в том, что *в R любой результат с числами трактуется как вектор единичной длины*, так как скаляров в R, вообще говоря, нет. Кроме этого, нумерация элементов векторов начинается с 1, а не с 0, как во многих других языках программирования.

Сохранение числовых и строковых значений:

```
> number <- 10 # сохранение объект

> number # выводим объект
[1] 10
```

```
> (number <- 10) # Сохраняем и выводим объект
[1] 10

> string <- "Hello" # Сохраняем объект-строку

> string
[1] "Hello"
```

Кроме строк и чисел можно также создавать и сохранять *векторы*. Вектор создается с помощью функции `c()`, которая объединяет несколько однотипных элементов. Также, с помощью двоеточия «:» или функции `seq()` можно создать регулярную последовательность. Функция `rep()` позволяет повторять некоторый образец.

```
> v1 <- c(2, 3, 4, 6, 10)

> v1
[1]  2  3  4  6 10

> v1[3]  # Получить третий элемент вектора
[1] 4

> v1[3:5] # Получить третий, четвертый и пятый элементы
вектора
[1]  4  6 10

> v2 <- c(1:5) # Определить вектор как последовательность
от 1 до 5
> v2
[1] 1 2 3 4 5

> s <- rep("a", 4)

> s
[1] "a" "a" "a" "a"

> rep(1:4, 2)
[1] 1 2 3 4 1 2 3 4

> rep(1:4, each = 2)
[1] 1 1 2 2 3 3 4 4
```

```
> (v1 + v2) * v2 # Можно проводить простые операции над
векторами
[1]  3 10 21 40 75

> crossprod(v1, v2) # Вычисление скалярного произведения
     [,1]
[1,]   94

> v1[v1 > 4] # Получить все координаты вектора большие 4
[1]  6 10
```

Можно получать различные свойства векторов:

```
> length(v1) # Длина вектора
[1] 5

> mean(v1) # Среднее значение элементов вектора
[1] 5

> var(v1) # Дисперсия элементов вектора
[1] 10
```

Матрицы создаются с помощью команды `matrix()`:

```
> args(matrix)
function (data = NA, nrow = 1, ncol = 1, byrow = FALSE,
dimnames = NULL)
NULL

> matrix(data = 1:5, nrow = 5, ncol = 5, byrow = FALSE)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    1    1    1    1
[2,]    2    2    2    2    2
[3,]    3    3    3    3    3
[4,]    4    4    4    4    4
[5,]    5    5    5    5    5

> matrix(data = 1:5, nrow = 5, ncol = 5, byrow = TRUE)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    2    3    4    5
[2,]    1    2    3    4    5
[3,]    1    2    3    4    5
[4,]    1    2    3    4    5
[5,]    1    2    3    4    5
```

Можно создать матрицу с неопределенными значениями (для этого используется специальное обозначение NA):

```
> matrix(data = NA, nrow = 3, ncol = 3)
     [,1] [,2] [,3]
[1,]   NA   NA   NA
[2,]   NA   NA   NA
[3,]   NA   NA   NA
```

Матрицу можно также получить с помощью функций–комбинаторов `cbind` (объединяет столбцы) и `rbind` (объединяет строки).

```
> m <- cbind(v1, v2) # Создаем матрицу

> m
     v1 v2
[1,]  2  1
[2,]  3  2
[3,]  4  3
[4,]  6  4
[5,] 10  5

> typeof(m) # Получаем тип элементов матрицы
[1] "double"

> class(m) # Получаем класс объекта
[1] "matrix"

> is.matrix(m) # Проверяем, является ли m матрицей
[1] TRUE

> is.vector(m) # m не вектор
[1] FALSE

> dim(m) # Получаем размерность m
[1] 5 2
```

Примечание. Функция `cbind` (`rbind`) объединяет столбцы (строки) не только векторов, но и матриц и более сложных, подходящих для этого, структур данных:

```
> m1 <- cbind(m, m)

> m1
     v1 v2 v1 v2
[1,]  2  1  2  1
[2,]  3  2  3  2
[3,]  4  3  4  3
[4,]  6  4  6  4
[5,] 10  5 10  5


> m2 <- rbind(m, m)

> m2
      v1 v2
 [1,]  2  1
 [2,]  3  2
 [3,]  4  3
 [4,]  6  4
 [5,] 10  5
 [6,]  2  1
 [7,]  3  2
 [8,]  4  3
 [9,]  6  4
[10,] 10  5
```

Можно также определить свою структуру данных:

```
> mydat <- data.frame(m) # Создаём структуру данных

> names(mydat) # Даём имена каждой переменной
[1] "v1" "v2"

> str(mydat) #  Выводим структуру данных
'data.frame':  5 obs. of  2 variables:
 $ v1: num  2 3 4 6 10
 $ v2: num  1 2 3 4 5

> summary(mydat) # Статистика
       v1           v2
 Min.   : 2   Min.   :1
 1st Qu.: 3   1st Qu.:2
 Median : 4   Median :3
 Mean   : 5   Mean   :3
 3rd Qu.: 6   3rd Qu.:4
 Max.   :10   Max.   :5
```

1.2. Работа с графикой в R

Основной функцией для рисования объектов в R является функция `plot(x, y, …)`:

`x` — координаты точек графика, либо некоторая графическая структура, функция или объект, содержащий методы рисования.

`y` — y-координаты точек графика, если x — соответствующего типа.

… — остальные графические параметры. Перечислим некоторые из них:

- параметр `type` позволяет изменять внешний вид точек на графике и может принимать одно из следующих значений:
  - `"p"` — точки (*points*; используется по умолчанию);
  - `"l"` — линии (*lines*);
  - `"b"` — изображаются и точки, и линии (*both points and lines*);
  - `"o"` — точки изображаются поверх линий (*points over lines*);
  - `"h"` — гистограмма (*histogram*);
  - `"s"` — ступенчатая кривая (*steps*);
  - `"n"` — данные не отображаются (*no points*).
- параметры `xlab` и `ylab` задают название осей абсцисс и ординат, соответственно;
- параметр `main` задаёт заголовок графика.

Рассмотрим несколько примеров.

```
# Линия регрессии:
> require(stats) # для lowess, rpois, rnorm
  plot(cars)
  lines(lowess(cars))
```

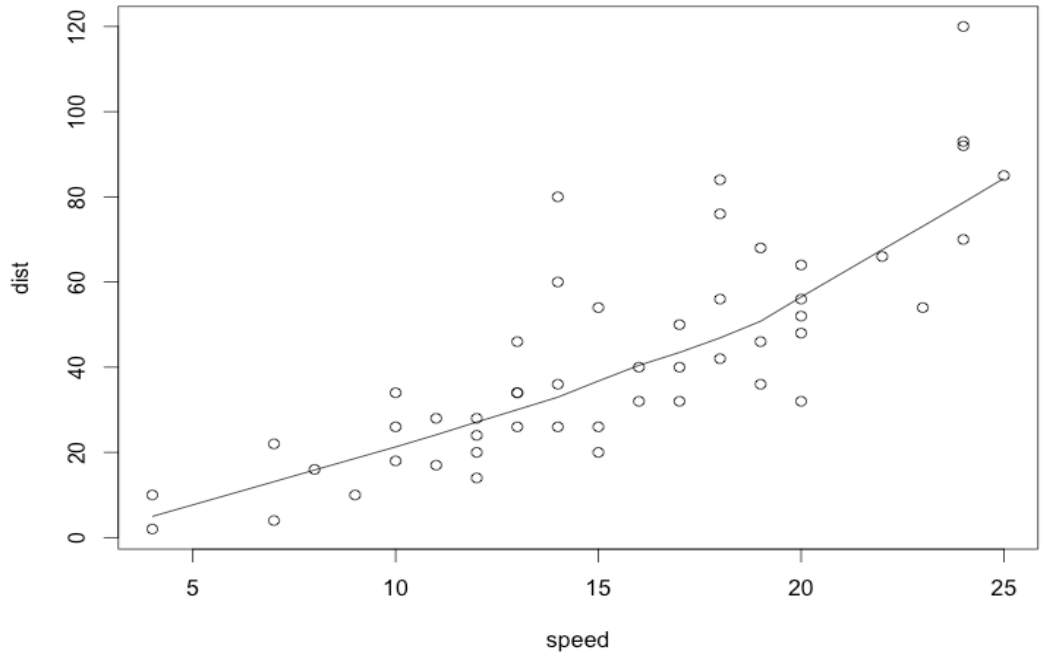

Рис 1.1. Линия регрессии.

```
# Синусоида:
> plot(sin, -pi, 2*pi)
```

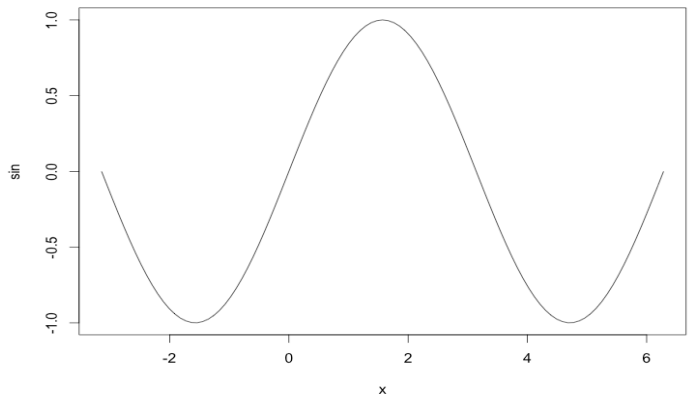

Рис 1.2. Синусоида.

```
## График дискретного распределения:
> plot(table(rpois(100, 5)), type = "h", col = "red",
       lwd = 10, main = "rpois(100, lambda = 5)")
```

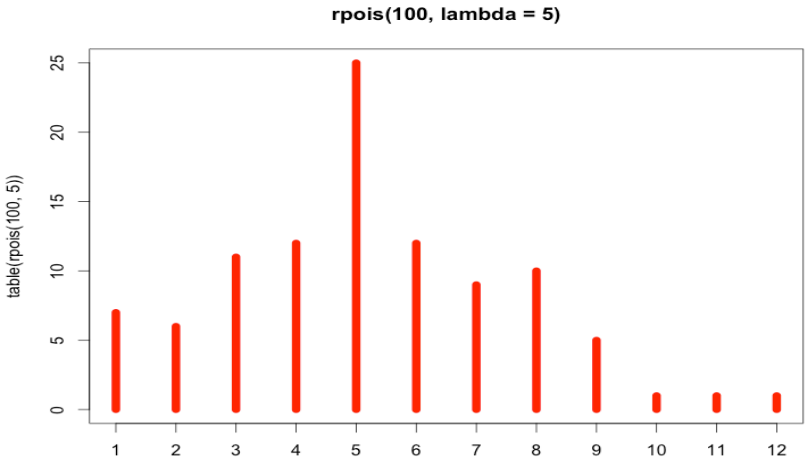

Рис 1.3. Дискретное распределение.

```
## Квантили:
> plot(x <- sort(rnorm(47)), type = "s",
       main = "plot(x, type = \"s\")")
  points(x, cex = .5, col = "dark red")
```

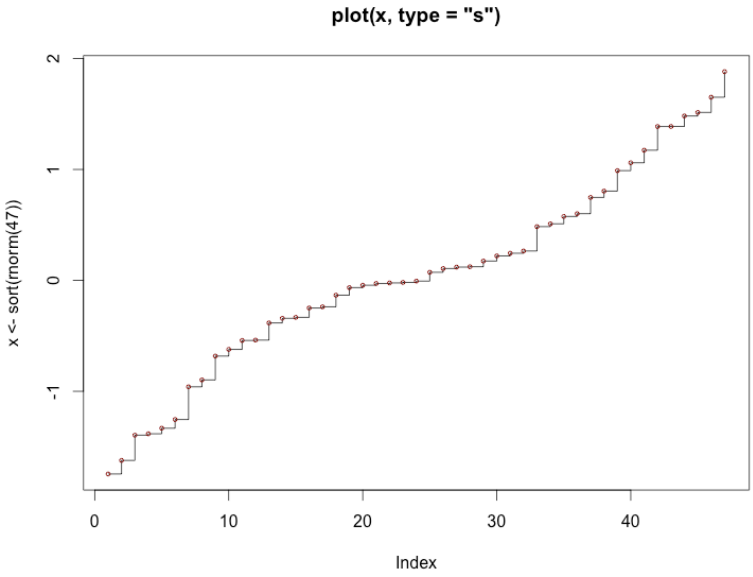

Рис 1.4. Квантили.

```
# Нейронная сеть:
...

library(neuralnet)
n <- names(train_)
f <- as.formula(paste("medv ~", paste(n[!n %in% "medv"],
collapse = " + ")))
nn <-
neuralnet(f,data=train_,hidden=c(5,3),linear.output=T)

plot(nn)
```

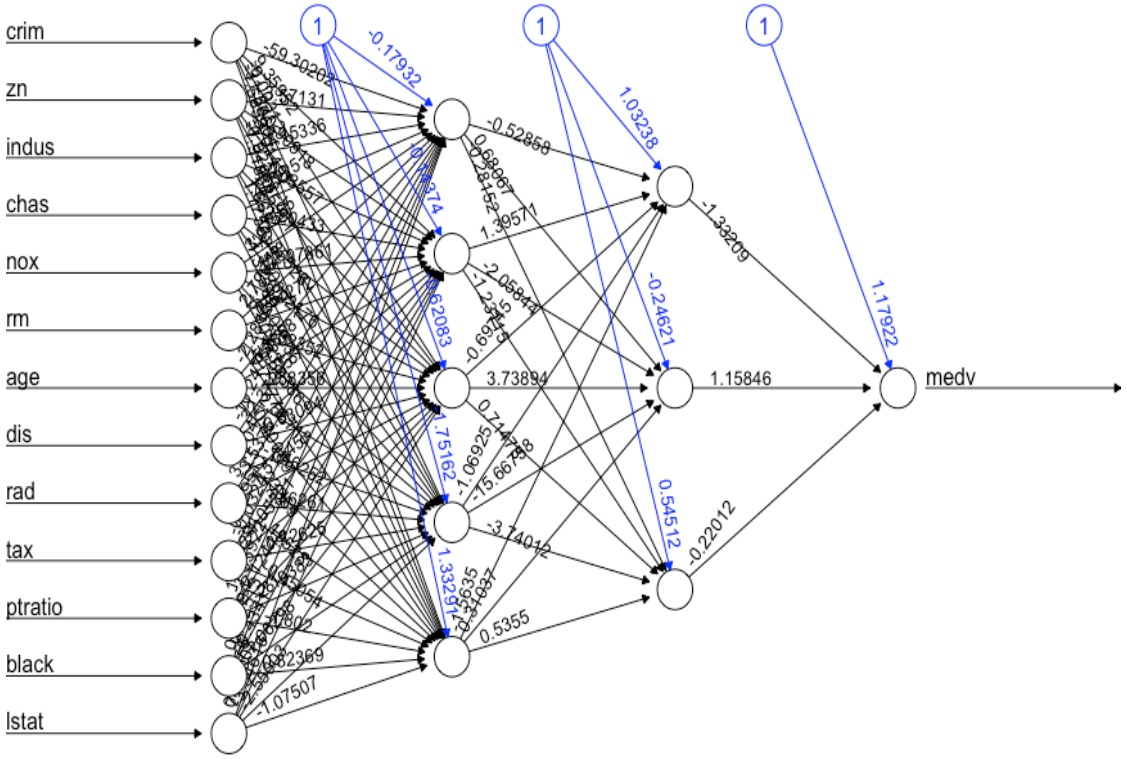

Рис 1.5. Нейронная сеть.

Как видно из примеров, с помощью функции `plot()` можно создавать большое количество разнообразных графиков.

2. ЗАДАЧА КЛАССИФИКАЦИИ

Пусть имеется некоторое *множество X объектов,* где каждый объект x представляется в виде n-мерного вектора значений признаков $(f_1(x), \dots, f_n(x))$, где $f_j(x)$ — значение j-го признака объекта x. Предположим, что существует неизвестная *целевая зависимость* y^*, которая произвольному объекту $x \in X$ ставит в соответствие некоторое имя класса $y \in Y$. Информация о неизвестной целевой зависимости y^* представлена в виде набора прецедентов (объект; имя–класса), который (набор) называется *обучающей выборкой* и обозначается $X^\ell = \{(x_i, y_i)\}_{i=1}^{\ell}$, где $x_i \in X$ — объект, для которого известен класс $y_i \in Y$, $i = \overline{1, \ell}$, которому он принадлежит.

Требуется построить алгоритм $a: X \to Y$, который бы восстанавливал неизвестную целевую зависимость y^* не только на объектах обучения $\{x_1, \dots, x_\ell\}$, но и на всем множестве X.

Сформулированная выше задача называется *задачей классификации*. Опишем несколько методов решения данной задачи.

Определение. *Моделью алгоритма* называется параметрическое семейство отображений

$$A = \{\phi: X \times \Theta \to Y\},$$

где ϕ — некоторое фиксированное отображение, множество Θ — множество параметров модели или, как еще говорят, пространство поиска.

Определение. *Эмпирическим риском* называется величина

$$Q\left(a, X^\ell\right) = \frac{1}{\ell} \sum_{i=1}^{\ell} L(a, x_i),$$

где $L(a, x_i)$ — неотрицательная *функция потерь*, характеризующая величину потери алгоритма a на объекте x.

Для задачи классификации, как правило, используют бинарную функцию потерь, которая равна 1, если алгоритм a допускает ошибку на объекте x, и 0 — иначе. В этом случае эмпирический риск — это частота ошибки алгоритма a на выборке X^ℓ.

Определение. *Алгоритмом* (*методом*) *обучения* называется отображение

$$\mu: (X \times Y)^\ell \to A,$$

которое произвольной конечной выборке X^ℓ ставит в соответствие алгоритм $a \in A$.

Классическим методом обучения μ решения задачи классификации является *минимизация эмпирического риска*:

$$\mu\left(A, X^\ell\right) = \arg\min_{a \in A} Q\left(a, X^\ell\right).$$

2.1. Метрические алгоритмы классификации

Предположим, что в пространстве объектов X задана некоторая функция расстояния (метрика) $\rho(x, x')$, характеризующая степень близости объектов, причём, необязательно метрика. Кроме этого, будем считать, что объекты из одного класса близки согласно функции ρ, а объекты разных классов — далеки. Данное предположение называют *гипотезой компактности*. Обратим внимание, что для вычисления ρ нам необязательно знать признаковое описание объектов.

Метрический классификатор — это классификатор, который принимает решение о принадлежности классам, помимо обучающей выборки, согласно функции расстояния ρ между объектами. В общем виде его можно записать следующим образом:

$$a(x) = \arg\max_{y \in Y} W_y\left(x, X^\ell\right),$$

где $W_y\left(x, X^\ell\right)$ — степень принадлежности объекта x классу y.

Рассмотрим несколько разновидностей метрических классификаторов.

Алгоритм ближайших соседей. В качестве функции $W_y\left(z, X^\ell\right)$=$W_y\left(z, X^\ell; k\right) = w_y(z, x_{z,1}, \dots, x_{z,k})$, $x_{z,j}$ — j-ый по близости (близость определяется согласно функции ρ) к объекту z объект обучающей выборки X^ℓ. Объект $x_{z,j}$ называется *j-ым соседом* объекта z. Тогда $y_{z,j}$ — ответ на j-ом соседе. Если в качестве функции w_y взять $w_y = \sum_{i=1}^{k}[y = y_{z,i}]$, где $\left[y = y_{z,i}\right] = \begin{cases} 1, & y = y_{z,i}; \\ 0, & y \neq y_{z,i}, \end{cases}$ то получим метод *k ближайших соседей (kNN)*. Другими словами, метод *kNN* относит классифицируемый объект к тому классу, к которому принадлежит б*о*льшая часть из ближайших *k* соседей.

Реализуем данный алгоритм на языке R. Для примера рассмотрим обучающую выборку «Ирисы Фишера», которая в R хранится в объекте `iris`, фрагмент которой приведен ниже:

```
    Sepal.Length Sepal.Width Petal.Length Petal.Width    Species
1            5.1         3.5          1.4         0.2     setosa
2            4.9         3.0          1.4         0.2     setosa
…
51           7.0         3.2          4.7         1.4 versicolor
52           6.4         3.2          4.5         1.5 versicolor
…
149          6.2         3.4          5.4         2.3  virginica
150          5.9         3.0          5.1         1.8  virginica
```

Данная выборка содержит 150 объектов–ирисов: по 50 объектов каждого из трёх классов (видов ирисов): «setosa» (щетинистые), «versicolor» (разноцветные) и «virginica» (виргиника). Каждый ирис представлен четырьмя признаками: «Sepal.Length» (длина чашелистика), «Sepal.Width» (ширина чашелистика), «Petal.Length» (длина лепестка) и «Petal.Width» (ширина лепестка). Последний столбец «Species» соответствует классу ирисов.

Для начала отобразим выборку, используя два последних признака:

```
colors <- c("setosa" = "red", "versicolor" = "green3",
            "virginica" = "blue")
plot(iris[, 3:4], pch = 21, bg = colors[iris$Species],
                            col = colors[iris$Species])
```

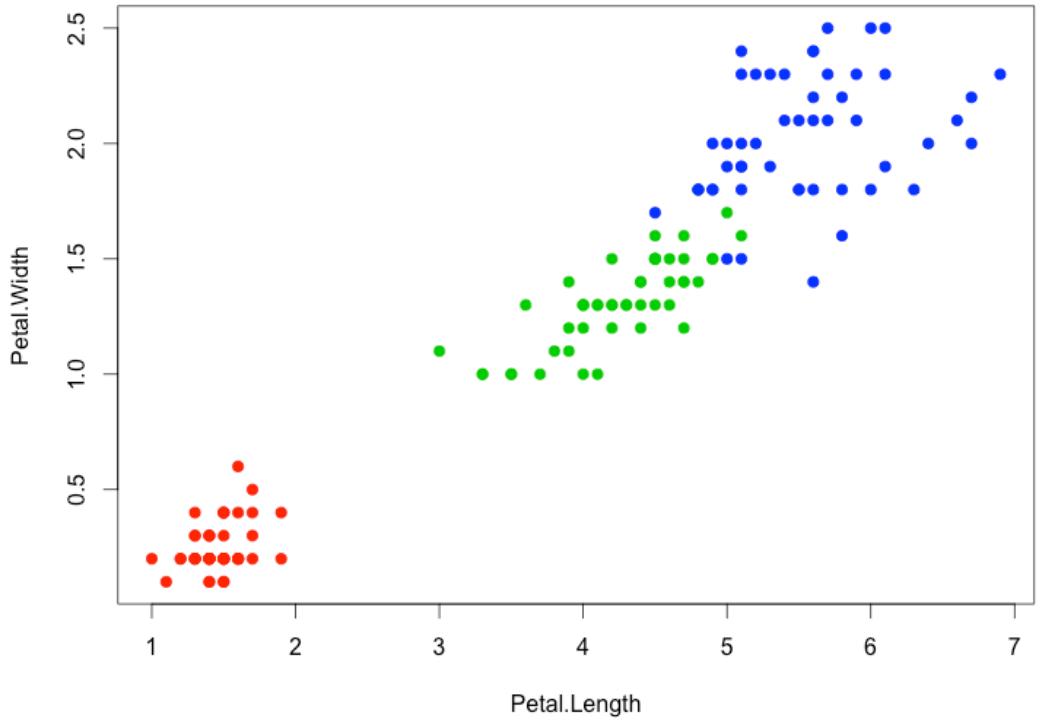

Рис 2.1. Выборка «Ирисы Фишера».

В качестве функции расстояния будем использовать обычное евклидово расстояние

```
##  Евклидово расстояние
euclideanDistance <- function(u, v)
{
    sqrt(sum((u - v)^2))
}
```

Далее приведен листинг для выше описанного алгоритма метода *kNN*.

```
##  Сортируем объекты согласно расстояния до объекта z
sortObjectsByDist <- function(xl, z, metricFunction =
euclideanDistance)
{
    l <- dim(xl)[1]
    n <- dim(xl)[2] - 1

    ##  Создаём матрицу расстояний
    distances <- matrix(NA, l, 2)

    for (i in 1:l)
    {
        distances[i, ] <- c(i, metricFunction(xl[i, 1:n], z))
    }

    ##  Сортируем
    orderedXl <- xl[order(distances[, 2]), ]

    return (orderedXl);
}


##  Применяем метод kNN
kNN <- function(xl, z, k)
{
    ##  Сортируем выборку согласно классифицируемого
объекта
    orderedXl <- sortObjectsByDist(xl, z)
    n <- dim(orderedXl)[2] - 1

    ##  Получаем классы первых k соседей
    classes <- orderedXl[1:k, n + 1]

    ##  Составляем таблицу встречаемости каждого класса
```

```
    counts <- table(classes)
    ##  Находим класс, который доминирует среди первых k
соседей
    class <- names(which.max(counts))

    return (class)
}

##  Рисуем выборку
colors <- c("setosa" = "red", "versicolor" = "green3",
"virginica" = "blue")
plot(iris[, 3:4], pch = 21, bg = colors[iris$Species], col
= colors[iris$Species], asp = 1)

##  Классификация одного заданного объекта
z <- c(2.7, 1)
xl <- iris[, 3:5]
class <- kNN(xl, z, k=6)
points(z[1], z[2], pch = 22, bg = colors[class], asp = 1)
```

Результатом работы данного скрипта будет следующий график:

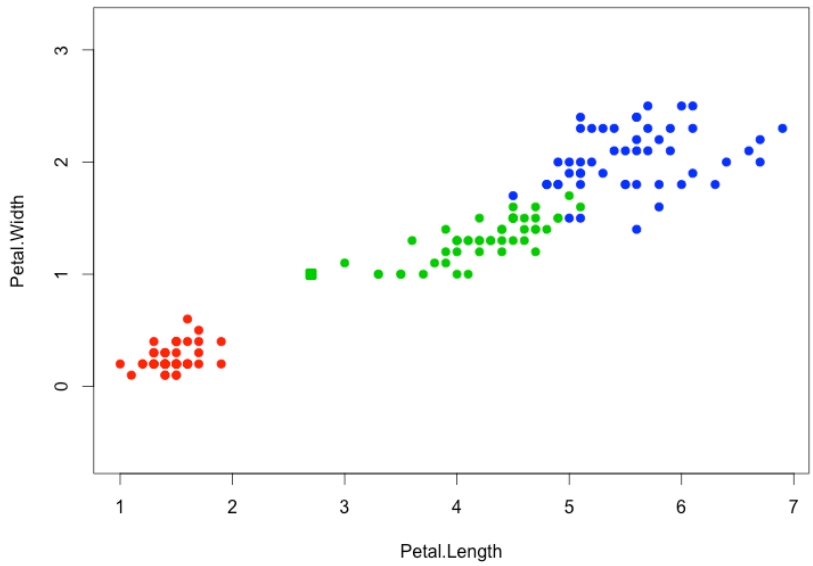

Рис 2.2. Классификация одного выделенного объекта (обозначен квадратиком).

На данном графике отображена обучающая выборка «Ирисы Фишера» (объекты–кружочки) и с помощью алгоритма 6NN проклассифицирован объект с координатами $(2.7, 1)$.

2.2. Байесовские алгоритмы классификации

Байесовские алгоритмы классификации основаны на предположении, что $X \times Y$ — вероятностное пространство с неизвестной плотностью распределения $p(x,y) = P(y)p(x|y)$, из которого случайно и независимо извлекаются ℓ наблюдений. Вероятность $P(y) = P_y$ появления объектов класса y называется *априорной вероятностью класса*, плотности распределения $p(x|y) = p_y(x)$ — функции *правдоподобия классов*.

Байесовский подход является классическим в теории распознавания образов и лежит в основе многих методов. Он опирается на теорему о том, что *если плотности распределения классов известны, то алгоритм классификации, имеющий минимальную вероятность ошибок, можно выписать в явном виде*.

Обозначим через λ_y – величину потери алгоритмом a при неправильной классификации объекта класса y.

Теорема. Если известны априорные вероятности классов P_y и функции правдоподобия $p_y(x)$, то минимум среднего риска

$$R(a) = \sum_{y \in Y} \sum_{s \in Y} \lambda_y P_y P(A_s|y), \quad A_s = \{x \in X \mid a(x) = s\},$$

достигается алгоритмом

$$a(x) = \arg\max_{y \in Y} \lambda_y P_y p_y(x).$$

Алгоритм $a(x)$ называется *оптимальным байесовским решающим правилом*. Однако, на практике зачастую плотности распределения классов неизвестны и их приходится восстанавливать

по обучающей выборке. В этом случае байесовский алгоритм перестает быть оптимальным. Поэтому, чем лучше удастся восстановить функции правдоподобия, тем ближе будет к оптимальному построенный алгоритм. Существуют множество способов восстановления плотностей распределения по обучающей выборке, откуда как следствие большое количество разновидностей байесовских алгоритмов классификаций.

2.2.1. Подстановочный алгоритм

Нормальный дискриминантный анализ — это один из вариантов байесовской классификации, в котором в качестве моделей восстанавливаемых плотностей рассматривают многомерные нормальные плотности:

$$N(x;\mu,\Sigma) = \frac{1}{\sqrt{(2\pi)^n|\Sigma|}} exp\left(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)\right), \quad x \in \mathbb{R}^n,$$

где $\mu \in \mathbb{R}^n$ — математическое ожидание (центр), $\Sigma \in \mathbb{R}^{n\times n}$ — ковариационная матрица. Предполагается, что матрица Σ симметричная, невырожденная, положительно определённая.

Восстанавливая параметры нормального распределения μ_y, Σ_y для каждого класса $y \in Y$ и подставляя в формулу оптимального байесовского классификатора восстановленные плотности, получим *подстановочный* (*plug-in*) алгоритм классификации либо *линейный дискриминант Фишера* (если предположить, что матрицы ковариации равны для всех классов).

Параметры нормального распределения оценивают согласно принципа максимума правдоподобия:

$$\mu_y = \frac{1}{\ell_y} \sum_{x_i : y_i = y} x_i, \qquad \Sigma_y = \frac{1}{\ell_y - 1} \sum_{x_i : y_i = y} (x_i - \mu_y)(x_i - \mu_y)^T.$$

Напишем программу, которая строит подстановочный (plug-in) алгоритм.

```
## Восстановление центра нормального распределения
estimateMu <- function(objects)
{
    ## mu = 1 / m  *  sum_{i=1}^m(objects_i)
    rows <- dim(objects)[1]
    cols <- dim(objects)[2]

    mu <- matrix(NA, 1, cols)
    for (col in 1:cols)
    {
        mu[1, col] = mean(objects[,col])
    }

    return(mu)
}

##  Восстановление ковариационной матрицы нормального
распределения
estimateCovarianceMatrix <- function(objects, mu)
{
    rows <- dim(objects)[1]
    cols <- dim(objects)[2]
    sigma <- matrix(0, cols, cols)

    for (i in 1:rows)
    {
        sigma <- sigma + (t(objects[i,] - mu) %*%
(objects[i,] - mu)) / (rows - 1)
    }

    return (sigma)
}

## Получение коэффициентов подстановочного алгоритма
getPlugInDiskriminantCoeffs <- function(mu1, sigma1, mu2,
sigma2)
{
    ## Line equation: a*x1^2 + b*x1*x2 + c*x2 + d*x1 + e*x2
+ f = 0
    invSigma1 <- solve(sigma1)
```

```
    invSigma2 <- solve(sigma2)

    f <- log(abs(det(sigma1))) - log(abs(det(sigma2))) +
mu1 %*% invSigma1 %*% t(mu1) - mu2 %*% invSigma2 %*%
t(mu2);

    alpha <- invSigma1 - invSigma2
    a <- alpha[1, 1]
    b <- 2 * alpha[1, 2]
    c <- alpha[2, 2]

    beta <- invSigma1 %*% t(mu1) - invSigma2 %*% t(mu2)
    d <- -2 * beta[1, 1]
    e <- -2 * beta[2, 1]

    return (c("x^2" = a, "xy" = b, "y^2" = c, "x" = d, "y"
= e, "1" = f))
}

## Количество объектов в каждом классе
ObjectsCountOfEachClass <- 100

## Подключаем библиотеку MASS для генерации многомерного
нормального распределения
library(MASS)

## Генерируем тестовые данные
Sigma1 <- matrix(c(10, 0, 0, 1), 2, 2)
Sigma2 <- matrix(c(1, 0, 0, 5), 2, 2)

Mu1 <- c(1, 0)
Mu2 <- c(15, 0)

xy1 <- mvrnorm(n=ObjectsCountOfEachClass, Mu1, Sigma1)
xy2 <- mvrnorm(n=ObjectsCountOfEachClass, Mu2, Sigma2)

## Собираем два класса в одну выборку
xl <- rbind(cbind(xy1, 1), cbind(xy2, 2))

## Рисуем обучающую выборку
colors <- c(rgb(0/255, 162/255, 232/255), rgb(0/255,
200/255, 0/255))
plot(xl[,1], xl[,2], pch = 21, bg = colors[xl[,3]], asp =
1)

## Оценивание
objectsOfFirstClass <- xl[xl[,3] == 1, 1:2]
objectsOfSecondClass <- xl[xl[,3] == 2, 1:2]
```

```
mu1 <- estimateMu(objectsOfFirstClass)
mu2 <- estimateMu(objectsOfSecondClass)

sigma1 <- estimateCovarianceMatrix(objectsOfFirstClass,
mu1)
sigma2 <- estimateCovarianceMatrix(objectsOfSecondClass,
mu2)

coeffs <- getPlugInDiskriminantCoeffs(mu1, sigma1, mu2,
sigma2)

## Рисуем дискриминантую функцию – красная линия
x <- y <- seq(-10, 20, len=100)
z <- outer(x, y, function(x, y) coeffs["x^2"]*x^2 +

coeffs["xy"]*x*y
          + coeffs["y^2"]*y^2 + coeffs["x"]*x
          + coeffs["y"]*y + coeffs["1"])

contour(x, y, z, levels=0, drawlabels=FALSE, lwd = 3, col =
"red", add = TRUE)
```

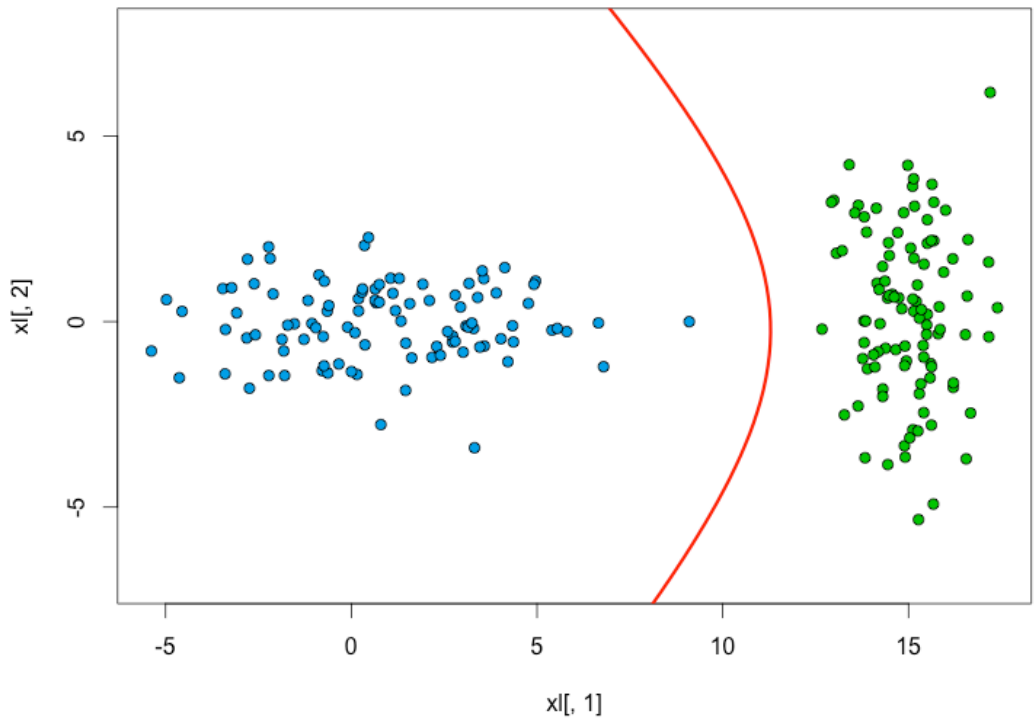

Рис 2.3. Байесовский нормальный квадратичный дискриминант.

Выбирая различные матрицы ковариации и центры для генерации тестовых данных, будем получать различные виды дискриминантной функции.

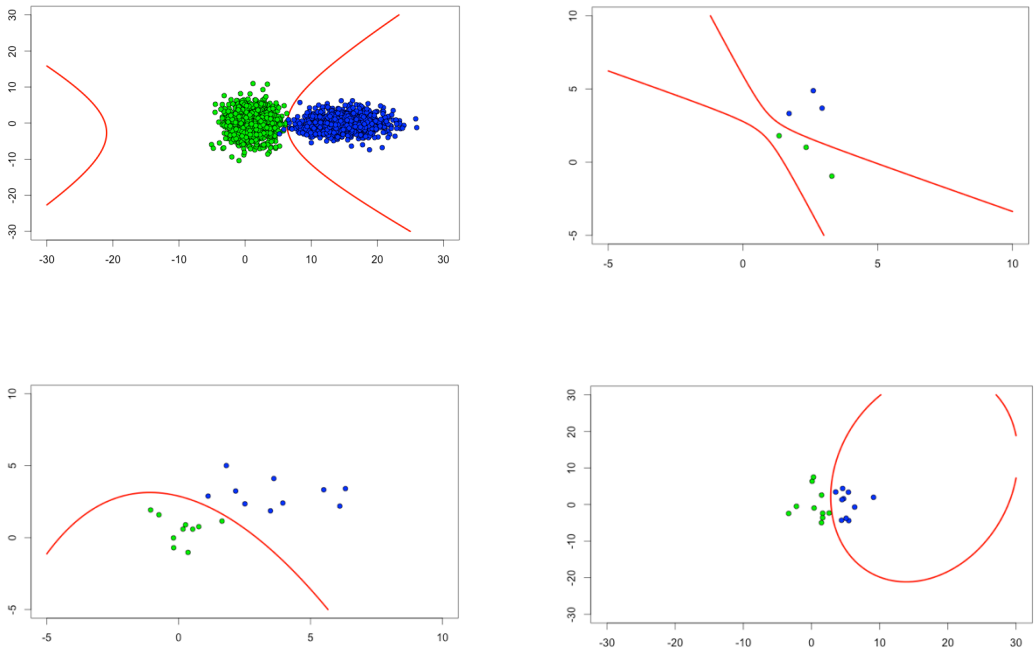

Рис 2.4. Примеры работы подстановочного алгоритма.

2.2.2. Линейный дискриминант Фишера — ЛДФ

Теперь рассмотрим линейный дискриминант Фишера (ЛДФ), который, в отличии от подстановочного алгоритма, при построении предполагает, что ковариационные матрицы классов равны, и для их восстановления нужно использовать все (всех классов) объекты обучающей выборки.

```
## Оценка ковариационной матрицы для ЛДФ
estimateFisherCovarianceMatrix <- function(objects1,
objects2, mu1, mu2)
{
    rows1 <- dim(objects1)[1]
    rows2 <- dim(objects2)[1]
    rows <- rows1 + rows2
    cols <- dim(objects1)[2]

    sigma <- matrix(0, cols, cols)

    for (i in 1:rows1)
    {
        sigma <- sigma + (t(objects1[i,] - mu1) %*%
(objects1[i,] - mu1)) / (rows + 2)
    }

    for (i in 1:rows2)
    {
        sigma <- sigma + (t(objects2[i,] - mu2) %*%
(objects2[i,] - mu2)) / (rows + 2)
    }

    return (sigma)
}

## Генерируем тестовые данные
Sigma1 <- matrix(c(2, 0, 0, 2), 2, 2)
Sigma2 <- matrix(c(2, 0, 0, 2), 2, 2)

Mu1 <- c(1, 0)
Mu2 <- c(15, 0)

xy1 <- mvrnorm(n=ObjectsCountOfEachClass, Mu1, Sigma1)
xy2 <- mvrnorm(n=ObjectsCountOfEachClass, Mu2, Sigma2)

## Собираем два класса в одну выборку
xl <- rbind(cbind(xy1, 1), cbind(xy2, 2))

## Рисуем обучающую выборку
colors <- c(rgb(0/255, 162/255, 232/255), rgb(0/255,
200/255, 0/255))
plot(xl[,1], xl[,2], pch = 21, bg = colors[xl[,3]], asp =
1)

## Оценивание
objectsOfFirstClass <- xl[xl[,3] == 1, 1:2]
objectsOfSecondClass <- xl[xl[,3] == 2, 1:2]
```

```
mu1 <- estimateMu(objectsOfFirstClass)
mu2 <- estimateMu(objectsOfSecondClass)

Sigma <-
estimateFisherCovarianceMatrix(objectsOfFirstClass,
objectsOfSecondClass, mu1, mu2)

## Получаем коэффициенты ЛДФ
inverseSigma <- solve(Sigma)
alpha <- inverseSigma %*% t(mu1 - mu2)
mu_st <- (mu1 + mu2) / 2
beta <- mu_st %*% alpha

## Рисуем ЛДФ
abline(beta / alpha[2,1], -alpha[1,1]/alpha[2,1], col =
"red", lwd = 3)
```

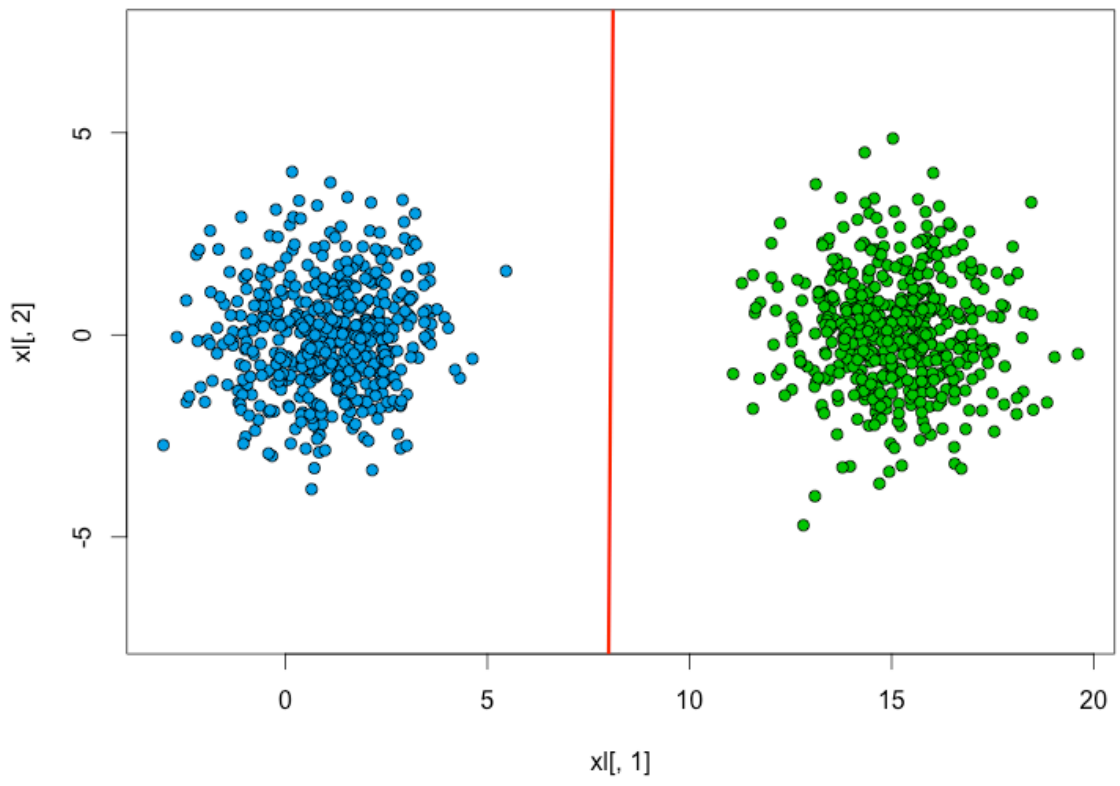

Рис 2.5. Линейный дискриминант Фишера.

2.3. Линейные алгоритмы классификации

Пусть $X = \mathbb{R}^n$ и $Y = \{-1; +1\}$. Алгоритм

$$a(x, w) = sign\,\langle w, x\rangle, \quad w \in \mathbb{R}^n,$$

является *линейным алгоритм классификации*.

Уравнение $\langle w, x\rangle = 0$ задаёт разделяющую классы гиперплоскость в пространстве $\mathbb{R}^n$.

Для подбора оптимального (минимизирующего эмпирический риск $Q(w, X^\ell)$) значения вектора весов w будем пользоваться *методом стохастического градиента* — итерационный процесс, на каждом шаге которого сдвигаемся в сторону противоположную вектору градиента $Q'(w, X^\ell)$ до тех пор, пока вектор весов w не перестанет изменяться, причем вычисления градиента производится не на всех объектах обучения, а выбирается случайный объект (отсюда и название метода «стохастический»), на основе которого и происходят вычисления. В зависимости от функции потерь, которая используется в функционале эмпирического риска, будем получать различные линейные алгоритмы классификации.

При использовании метода стохастического градиента необходимо нормализовать исходные данные:

```
## Нормализация обучающей выборки
trainingSampleNormalization <- function(xl)
{
    n <- dim(xl)[2] - 1
    for(i in 1:n)
    {
        xl[, i] <- (xl[, i] - mean(xl[, i])) / sd(xl[, i])
    }
    return (xl)
}
```

```
## Добавление колонки для из -1 для w0
trainingSamplePrepare <- function(xl)
{
    l <- dim(xl)[1]
    n <- dim(xl)[2] - 1
    xl <- cbind(xl[, 1:n], seq(from = -1, to = -1,
length.out = l), xl[, n + 1])
}
```

Алгоритм классификации ADALINE — *адаптивный линейный элемент*, в качестве функции потерь используется квадратичная функция потерь:

```
## Квадратичная функция потерь
lossQuad <- function(x)
{
    return ((x-1)^2)
}
```

Обучение ADALINE с помощью стохастического градиента:

```
## Стохастический градиент для ADALINE
sg.ADALINE <- function(xl, eta = 1, lambda = 1/6)
{
    l <- dim(xl)[1]
    n <- dim(xl)[2] - 1
    w <- c(1/2, 1/2, 1/2)
    iterCount <- 0

    ## initialize Q
    Q <- 0
    for (i in 1:l)
    {
        ## calculate the scalar product <w,x>
        wx <- sum(w * xl[i, 1:n])
        ## calculate a margin
        margin <- wx * xl[i, n + 1]

        Q <- Q + lossQuad(margin)
    }

    repeat
    {
        ## calculate the margins for all objects of the
training sample
        margins <- array(dim = l)
```

```
        for (i in 1:l)
        {
            xi <- xl[i, 1:n]
            yi <- xl[i, n + 1]

            margins[i] <- crossprod(w, xi) * yi
        }

        ## select the error objects
        errorIndexes <- which(margins <= 0)

        if (length(errorIndexes) > 0)
        {
            # select the random index from the errors
            i <- sample(errorIndexes, 1)
            iterCount <- iterCount + 1

            xi <- xl[i, 1:n]
            yi <- xl[i, n + 1]

            ## calculate the scalar product <w,xi>
            wx <- sum(w * xi)

            ## make a gradient step
            margin <- wx * yi

            ## calculate an error
            ex <- lossQuad(margin)
            eta <- 1 / sqrt(sum(xi * xi))
            w <- w - eta * (wx - yi) * xi

            ## Calculate a new Q
            Qprev <- Q
            Q <- (1 - lambda) * Q + lambda * ex
        }
        else
        {
            break
        }
    }

    return (w)
}
```

Персептрон Розенблатта — линейный классификатор, обучаемый с помощью стохастического градиента с правилом Хэбба и кусочно-линейной функции потерь:

```
## Функция потерь для правила Хэбба
lossPerceptron <- function(x)
{
    return (max(-x, 0))
}

## Стохастический градиент с правилом Хебба
sg.Hebb <- function(xl, eta = 0.1, lambda = 1/6)
{
    l <- dim(xl)[1]
    n <- dim(xl)[2] - 1
    w <- c(1/2, 1/2, 1/2)
    iterCount <- 0

    ## initialize Q
    Q <- 0
    for (i in 1:l)
    {
        ## calculate the scalar product <w,x>
        wx <- sum(w * xl[i, 1:n])
        ## calculate a margin
        margin <- wx * xl[i, n + 1]
        # Q <- Q + lossQuad(margin)
        Q <- Q + lossPerceptron(margin)
    }

    repeat
    {
        ## Поиск ошибочных объектов
        margins <- array(dim = l)

        for (i in 1:l)
        {
            xi <- xl[i, 1:n]
            yi <- xl[i, n + 1]

            margins[i] <- crossprod(w, xi) * yi
        }

        ## выбрать ошибочные объекты
        errorIndexes <- which(margins <= 0)
        if (length(errorIndexes) > 0)
        {
            # выбрать случайный ошибочный объект
            i <- sample(errorIndexes, 1)

            iterCount <- iterCount + 1

            xi <- xl[i, 1:n]
```

```
            yi <- xl[i, n + 1]

            w <- w + eta * yi * xi
        }
        else
            break;
    }

    return (w)
}
```

Логистическая регрессия — линейный байесовский классификатор, использующий логарифмическую функцию потерь, имеет ряд интересных особенностей, например, алгоритм способен помимо определения принадлежности объекта к классу определять и степень его принадлежности. Является одним из популярных алгоритмом классификации.

```
## Логарифмическая функция потерь
lossLog <- function(x)
{
    return (log2(1 + exp(-x)))
}

## Сигмоидная функция
sigmoidFunction <- function(z)
{
    return (1 / (1 + exp(-z)))
}

## Стохастический градиент для логистической регрессии
sg.LogRegression <- function(xl)
{
    l <- dim(xl)[1]
    n <- dim(xl)[2] - 1
    w <- c(1/2, 1/2, 1/2)
    iterCount <- 0
    lambda <- 1/l

    ## initialize Q
    Q <- 0
    for (i in 1:l)
    {
        ## calculate the scalar product <w,x>
        wx <- sum(w * xl[i, 1:n])
```

```
        ## calculate a margin
        margin <- wx * xl[i, n + 1]

        Q <- Q + lossSigmoid(margin)
    }

    repeat
    {
        # select the random index from the error objects
errorIndexes
        i <- sample(1:l, 1)

        iterCount <- iterCount + 1

        # i <- sample(1:l, 1)
        xi <- xl[i, 1:n]
        yi <- xl[i, n + 1]

        ## calculate the scalar product <w,xi>
        wx <- sum(w * xi)

        ## make a gradient step
        margin <- wx * yi

        ex <- lossSigmoid(margin)
        eta <- 0.3#1 / iterCount
        w <- w + eta * xi * yi * sigmoidFunction(-wx * yi)

        ## Calculate a new Q
        Qprev <- Q
        Q <- (1 - lambda) * Q + lambda * ex

        if (abs(Qprev - Q) / abs(max(Qprev, Q)) < 1e-5)
break
    }

    return (w)
}


#  Кол-во объектов в каждом классе
ObjectsCountOfEachClass <- 100


## Моделируем обучающие данные
library(MASS)

Sigma1 <- matrix(c(2, 0, 0, 10), 2, 2)
Sigma2 <- matrix(c(4, 1, 1, 2), 2, 2)
```

```
xy1 <- mvrnorm(n=ObjectsCountOfEachClass, c(0, 0), Sigma1)
xy2 <- mvrnorm(n=ObjectsCountOfEachClass, c(10, -10),
Sigma2)

xl <- rbind(cbind(xy1, -1), cbind(xy2, +1))

colors <- c(rgb(255/255, 255/255, 0/255), "white",
rgb(0/255, 200/255, 0/255))

## Нормализация данных
xlNorm <- trainingSampleNormalization(xl)
xlNorm <- trainingSamplePrepare(xlNorm)


## Отображение данных

## ADALINE
plot(xlNorm[, 1], xlNorm[, 2], pch = 21, bg = colors[xl[,3]
+ 2], asp = 1, xlab = "x1", ylab = "x2", main = "Линейные
классификаторы")
w <- sg.ADALINE(xlNorm)
abline(a = w[3] / w[2], b = -w[1] / w[2], lwd = 3, col =
"blue")


## Правило Хебба
w <- sg.Hebb(xlNorm)
abline(a = w[3] / w[2], b = -w[1] / w[2], lwd = 3, col =
"green3")


## Логистическая регрессия
w <- sg.LogRegression(xlNorm)
abline(a = w[3] / w[2], b = -w[1] / w[2], lwd = 3, col =
"red")


legend("bottomleft", c("ADALINE", "Правило Хэбба",
"Логистическая регрессия"), pch = c(15,15,15), col =
c("blue", "green3", "red"))
```

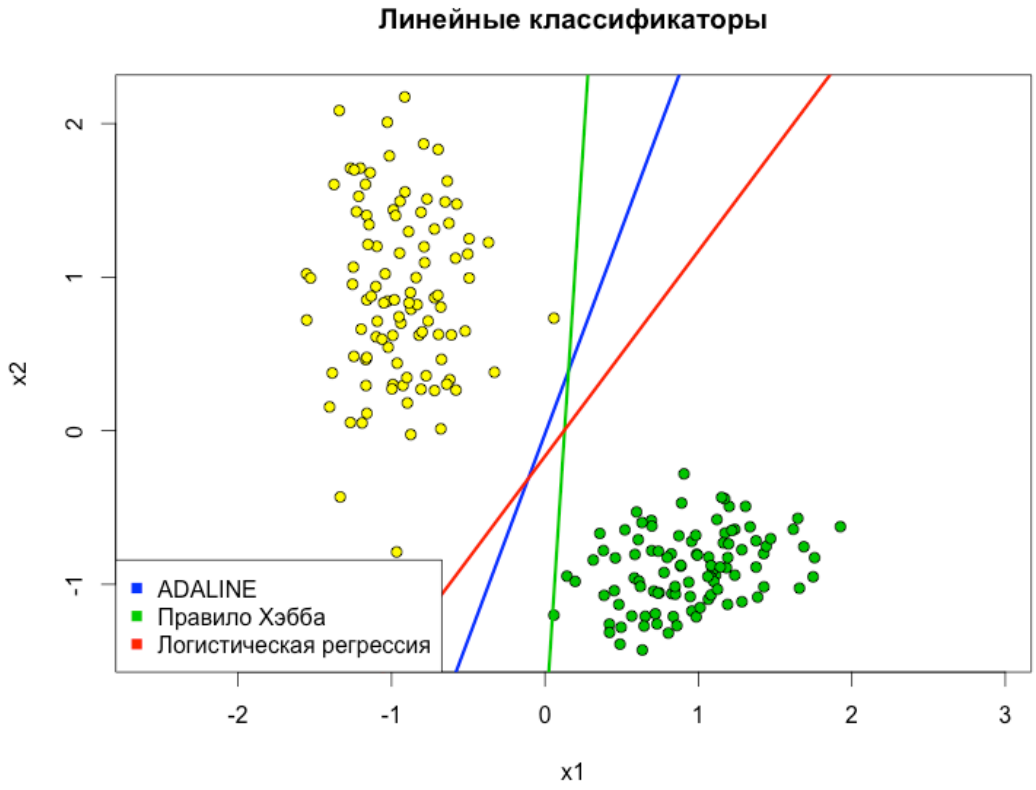

Рис 2.6. Линейные алгоритмы классификации.

Список литературы

1. Шипунов А. Б., Балдин Е. М. Анализ данных с R [Электронный ресурс] / А. Б. Шипунов, Е. М. Балдин / URL: http://www.soc.univ.kiev.ua/sites/default/files/course/materials/r1.pdf
2. Савельев А. А., Мухарамова С. С., Пилюгин А. Г. (2007) Основные понятия языка R. Учебно-методическое пособие. Казань: Казанский гос. ун-т, 29 с.
3. Зарядов И. С. (2010) Введение в статистический пакет R: типы переменных, структуры данных, чтение и запись информации, графика. М.: Издательство Российского университета дружбы народов, 207 с.
4. Воронцов К.В. Математические методы обучения по прецедентам (теория обучения машин). [Электронный ресурс]. — 2011. — 141 с. — URL: http://www.machinelearning.ru/wiki/images/6/6d/Voron-ML-1.pdf
5. Профессиональный информационно-аналитический ресурс, посвященный машинному обучению, распознаванию образов и интеллектуальному анализу данных — URL: http://machinelearning.ru

Учебно-методическое пособие по дисциплине «Системы и методы принятия решений». Применение языка «R» к решению задач классификации: примеры.

Автор-составитель: Анафиев Айдер Сератович

Редакция автора

Подписано в печать 23.12.2016. Формат 60×80/16.

Отпечатано в отделе редакционно-издательской деятельности
КФУ им. В. И. Вернадского
295007, Симферополь, пр-т Академика Вернадского, 4,
КФУ им. В. И. Вернадского