SSE-DP-2025-3

R と Python による統計的学習入門：

ISLR・ISLP 実習(日本語版)

国友直人・湯浅良太

統計数理研究所

2025 年 3 月

SSE-DP-2025-3

ISLR and ISLP via R and Python

: A Japanese Version

edited by

Naoto Kunitomo and Ryota Yuasa

Institute of Statistical Mathematics

March 2025

(Summary)

This report presents the results of an instructional material development exercise as part of the "Statistical Expert Training Program," titled *"Introduction to Statistical Learning with R and Python."* The exercise involved the Japanese translation of ISLR and ISLP, web-based resources developed by Professor Hastie's group at the Department of Statistics, Stanford University. The participants of this exercise were primarily volunteers from the statistics experts training program, including: **Naoto Kunitomo** (The Institute of Statistical Mathematics), **Yu Zhao** (Tokyo University of Science), **Hayato Nishi** (The University of Tokyo),**Yujie Xue** (The Institute of Statistical Mathematics),**Tomohiro Tajima** (Shiga University),**Tadashi Nakanishi** (Hokkaido University),**Ryota Yuasa** (Chiba University), and **Masahiro Mochizuki** (Waseda University). It should be noted that the content of this report does not reflect the views of the Institute of Statistical Mathematics.

R と Python による統計的学習入門：ISLR・ISLP 実習 (日本語版) *

国友直人 †・湯浅良太 ‡（編集）

2025 年 3 月

鍵言葉 (Key Words)：統計的学習 (Statistical Learning)、ISLR と ISRP の Lab 実例、R と Python による実習、ChatGPT と Google Colaboratory の利用

要約

統計数理研究所が推進している統計エキスパート養成事業の一環として 2024 年-2025 年の教材作成演習「R と Python による統計的学習入門」が企画された。演習では統計的学習の理論と応用に関心のある研修参加者が Stanford 大学統計学科 Trevor Hastie 教授を中心に（James,G., D.Witten, T. Hastie and R. Tibshirani、及び J. Taylor）が出版した 2 冊の教科書、ISLR（An Introduction to Statistical Learning with Applications in R, 2nd edition, 2021) と ISLP(An Introduction to Statistical Learning with Applications in Python, 2023) の内容を検討した。特に二つの書籍に関連した実修教材が Web 上で公開されているので、翻訳などの作業を行い、日本語で利用できる教材を作成した。ISLR および ISLP ではそれぞれ各章に R および Python による統計的学習によるデータ分析の方法が英語で説明されているので、その説明を ChatGPT など AI 翻訳、新たに利用可能となった Google Colaboratory などを活用するとともに、参加者が内容を検討、統計エキスパート養成事業など統計科学の教育上で有用と判断した内容を精査、日本語の教材としてまとめた。

*統計エキスパート養成事業における教材作成演習「R と Python による統計的学習入門」の報告書（Stanford 大学統計学科 Hastie 教授グループにより開発された Web 上の教材 ISLR, ISLP 演習の日本語訳）であり、同演習の参加メンバーは国友グループ研修参加者を中心とする有志（国友直人 (統計数理研究所), 趙宇 (東京理科大学), 西颯人 (東京大学), 薛玉傑 (統計数理研究所), 田島友祐 (滋賀大学), 中西正 (北海道大学), 湯浅良太 (千葉大学), 望月秦博 (早稲田大学) である。なおこの報告書の内容は統計数理研究所の見解を反映するものではない。

†統計数理研究所

‡千葉大学法政経学部

目次

はじめに

近年では統計的学習 (statistical learning) の理論と応用は統計学の中では特に発展が顕著な分野の一つになっている。統計数理研究所が推進している統計エキスパート養成プロジェクトでは 2024 年～2025 年に教材作成演習「R と Python による統計的学習入門」が企画された。

この教材作成演習では基本的な教材として指定しているスタンフォード大学統計学科 (Department of Statistics, Stanford University) の T. Hastie 教授を中心とする研究・教育グループ（James,G., D.Witten, T. Hastie and R. Tibshirani）による ISLR（An Introduction to Statistical Learning with Appications in R, 2nd edition, 2021)、さらに新たに J. Taylor 教授が参加した ISLP(An Introduction to Statistical Learning with Applications in Python, 2023) と云う二冊の書籍を教材として利用した。これらの書物は 2024 年時点においてスタンフォード大学など米国の主要な統計学科における学部上級生から大学院修士課程における統計的学習に関する一つの標準的教材となっている。二冊の書籍の各章では統計的方法の議論と共に、R および Python による PC を用いたデータ分析の方法が具体的に説明されているとともに、Hastie 教授などにより ISL の Web ページにおいて実習部分についての資料を英語ではあるが公開されている。これらの資料は 2025 年 1 月の時点では日本語で利用可能ではないが、現時点では日本語として自由に利用できる教材として広く利用できることが望ましい、と判断した。折しも近年になり DeepL, ChatGPT など新たに翻訳作業を効率化する手段や Google Colaboratory などの R プログラムや Python プログラムの実行を簡単化する方法なども利用可能になっているので、あわせて新しい方法の教育などにおける利用法の是非についても検討を加えた。作業を効率化する新たな手段を活用してみたところ、ChatGPT などによる AI 翻訳や Google Colaboratory などの計算の実行手段を活用することは有用であるが、2025 年 2 月の時点では出来上がった文案の内容をさらに統計エキスパート・プロジェクトの参加者が統計学的見地より検討を加えて必要性があるように判断した。そこで今後の教育的な見地から有用と判断した訳注などをコメントとして付け加えて日本語の教材としてまとめることとした。各章は担当者（目次に担当表を掲載する）が原案を作成、全体の統一性を確保するためにさらに原案の文言などを修正したが、各章の内容は担当した各人の責任

で作成した。無論ではあるが、この「統計エキスパート」養成プロジェクトの参加者は編集責任者を含め、この間の発展がが著しい計算機統計学や統計的学習論の専門家とは云えないことから日本語による表現・用語を含めて本稿はDP（ディスカッション・ペーパー）として暫定的な検討結果であることをあらかじめお断りしておく。このDPに関するコメント、改善箇所、誤りの指摘などを編集責任者の一人 (kunitomo アット ism.ac.jp) までお寄せいただければ幸いである。

なお念のためではあるが、Stanford大学におけるISLの開発責任者のTrevor Hastie教授からは公開資料を活用して日本語教材を作成することについてご了解をいただいたので、前書きの後に連絡メールを添付しておいた。Hastie教授が指摘しているようにISLR･ISLPのWebページ https://www.statlearning.com/ には本DPの原資料を含め様々な資料が掲載されている。原著に散見される誤植と思われる事項など修正したが、この日本語版に誤解や誤りがあるとすればむろん本稿の執筆者の責任である。収録したプログラムは2025年1月時点においてISLR, ISLPのWebに掲載されたversionであり、1月時点で各担当者が動作を確認したが、関連するプログラムの更新によるトラブルもありうることを付け加えておく。

例えばGoogle ColaboratoryによりISLP6章, 12章, 13章のコードを実行中に不安定な状況 (2025年3月20日頃) があったが、「ランタイムからセッションを再起動する」と問題なくそのまま動作した。この場合には念のためにJupyterではプログラムは正常に動作することを確認した。これはNumpyなどパッケージ更新のタイミング、あるいはグラフで日本語を使用しても文字化けが起こらないようにjapanize-matplotlibを用いたこと、などによる可能性はあるが理由は定かではない。

2025年3月時点において統計的学習の理論と応用に関して日本語で書かれたRとPythonによる統計学学習の分野についの学術的にも信頼できる書籍は多くない思われる。なお繰り返しになるが、教材演習の過程ではRによるデータ分析、Pythonによるデータ分析、「Google Colaboratory」、「DeepL」と「Chatgpt」による翻訳の性能評価、などデータ分析とAIを巡る最近の動向についても議論を重ねたが、近年のAI技術の発展は目覚ましいものがあるが、様々な課題があることも判明したので我々の作業も意味はあると判断した。

また編集者の一人と同様、R や Python の初心者、また最近の AI テクノロジーに疎い分野や世代に属する読者のために ISLR 第 2 章、ISLP 第 2 章などに R と Python に関する文献なども訳注としてあえて引用しておいた。関連する日本語資料の一部は GitHub（一定の期間の間はアドレス
https://github.com/hayato-n/stat_expert_ism
に掲示する予定）よりダウンロード可能とする予定である。計算プログラムは各章の担当者の責任において 2025 年月の時点で動作を確認したが)、他の環境での動作保証するものではないこと、さらにコメントは歓迎するが、内容の掲載についてはあくまで各担当者の判断によることをここであらためてお断りする。

この報告書は日本で統計的学習論や統計的データ分析に関心のある方々には手軽な「R や Python への導入」になっている。いままで R や Python を全く利用したことがなければ、R については例えば「R による統計データ分析入門」（小暮厚之, 朝倉書店）、R を利用していて分からないことがあればネットで検索、あるいは
https://cran.r-project.org/doc/contrib/manuals-jp/Mase-Rstatman.pdf
などが参考となる。Python 入門については https://utokyo-ipp.github.io/ あるいは https://utokyo-ipp.github.io/IPP_textbook.pdf などが参考になるだろう。さらにこの報告書の内容を伝統的な統計学の中身と対比されたい方には例えば、「データ分析のための統計学入門」(国友・小暮。吉田訳, 日本統計協会), 「統計学」(久保川達也・国友直人, 東京大学出版会), 「現代数理統計学」(竹村彰通, 学術図書出版社) などが参考になるだろう。

なお統計エキスパート養成プロジェクトの演習グループでは本稿に先立ち 2023 年〜2024 年には教材演習「統計的学習」を行い、Hastie-Tibshirani 講義録の日本語教材を作成、「SSE-DP-2023-3 統計的学習 (講義スライド) - Statistical Learning 国友直人・趙宇・湯浅良太 (訳者)」として
https://stat-expert.ism.ac.jp/wp/wp-content/uploads/2023/05/SSE-DP-2023-3.pdf
に掲示しておいたことに言及しておく。本稿は Hastie-Tibshirani 講義録の実習部分に対応しているのである。

前稿に引き続き、本稿が今後の日本における統計科学の高等教育の展開に役立つことを期待したい。

2025年3月
国友直人

Hastie教授からの手紙 (2024年11月24日)

Dear Professor Naoto

I think that is nice that you do that, and we are happy for you to translate the lab files to Japanese.

You should have somewhere on the page that offers these a note that says something (in Japanese) like:

“ These labs have been translated from the original english labs available from statlearning.com, with the permission of the authors. Any errors incurred in this translation are our responsibility ”

Also offering the translated labs for free is essential, since we offer them for free and would not be happy if someone charged for them.

best wishes,

Trevor

Part-I：Rによる統計的学習

ISLR 第 2 章 実習：R 入門

このラボでは、簡単な R コマンドを紹介する。新しい言語を学ぶ最良の方法は、実際にコマンドを試すことである。R は以下からダウンロードできる。

`http://cran.r-project.org/`

訳注 1：指示にしたがい R のインストール（"install R for the first time"がよい）が終わると画面上に現れる R（定期的に更新されるので最新の version がお薦め）をクリック、画面が立ち上がりプロンプト>が現れる。（以下のコマンドでは>を省略する。）初心者には R による統計分析の基本について例えば「R による統計データ分析入門」（小暮厚之, 朝倉書店）などが参考になる。なお R を利用していて分からないことがあればネットで検索する、あるいは

https://cran.r-project.org/doc/contrib/manuals-jp/Mase-Rstatman.pdf

を参照するとよいだろう

R を使用する際には、RStudio のような統合開発環境 (IDE) 内で実行することを勧める。RStudio は無料でダウンロード可能である。

`http://rstudio.com`

RStudio のウェブサイトでは、ソフトウェアのインストールが不要なクラウド版の R も提供している。

基本コマンド

R は関数を使用して操作を実行する。funcname という関数を実行するには、funcname(input1, input2)と入力するが、ここで、引数（input1 と input2）が R に実行方法を指示している。関数には任意の数の引数を指定でき、例えば、数値のベクトルを作成するには、c()（連結を意味する関数）を使用する。括弧内の数値は全て結合される。以下のコマンドは、1, 3, 2, 5 の数字を結合し、x というベクトルに保存する。ここで x と入力すると、ベクトルが表示される。

```
x <- c(1, 3, 2, 5)
x
## [1] 1 3 2 5
```

訳注 2：上の 1,3,2,5 はいずれも整数型, 整数は「integer 型」、実数は「double 型」として扱われ、整数型同士の足し算、引き算、掛け算の結果は integer 型であるが、割り算の結果は double 型になる。整数と実数を厳格に区別して使う場面は多くないため、numeric 型であることさえわかっていれば問題は生じない。

`>` はコマンドの一部ではなく、`R` が次のコマンド入力を待機していることを示すために表示されるものである。また、`<-` の代わりに `=` を使って値を保存することも可能である。

```
x = c(1, 6, 2)
x
## [1] 1 6 2
y = c(1, 4, 3)
```

上矢印を何度も押すと、以前に入力したコマンドが表示され、編集ができるので、よく似たコマンドを繰り返し使いたい場合に便利である。さらに、`?funcname` と入力すると、`funcname()` 関数に関する追加情報を含む新しいヘルプファイルウィンドウが開く。

`R` に 2 つの数値セットを足すように指示すると、それぞれのセットの最初の数同士、次の数同士が足し合わされる。ただし、`x` と `y` は同じ長さでなければならない。`length()`関数でデータの長さを確認できる。

```
length(x)
## [1] 3
length(y)
## [1] 3
x + y
## [1]  2 10  5
```

`ls()`関数により、これまでに保存したデータや関数など、すべてのオブジェクトの一覧を確認することができる。不要なものを削除するには、`rm()`関数を使えばよい。

```
ls()
## [1] "x" "y"
```

```
rm(x, y)
ls()
## character(0)
```

すべてのオブジェクトを一度に削除することも可能である:

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

`matrix()`関数を使用して数値の行列を作成できる。まず、`matrix()`関数について学ぶために、以下のように入力してみよう。

```
?matrix
## httpd ヘルプサーバーを起動... 完了
```

ヘルプ・ファイルによると、`matrix()`関数には複数の引数があるが、ここでは最初の 3 つに焦点を当ててみよう。データ（行列の要素）、行数、列数であるが、まず、簡単な行列を作成してみる。

```
x <- matrix(data = c(1, 2, 3, 4), nrow = 2, ncol = 2)
x
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

なお`data=`、`nrow=`、`ncol=`を省略しても同じように機能する。

```
x <- matrix(c(1, 2, 3, 4), 2, 2)
```

この場合も同じ結果が得られる。ただし、引数の名前を指定すると、関数の引数がヘルプ・ファイルに記載された順序通りでなくても認識されるため、便利である。デフォルトでは`R`は列ごとに行列を埋めるが、`byrow = TRUE`オプションを使用すると行ごとに行列が埋まる。

```
matrix(c(1, 2, 3, 4), 2, 2, byrow = TRUE)
##      [,1] [,2]
## [1,]    1    2
## [2,]    3    4
```

上記のコマンドでは、行列を変数 `x` などの値に割り当てているわけではない。この場合、行列は画面に表示されるだけで、将来の計算に保存されるわけではない。 `sqrt()`関数はベクトルや行列の各要素の平方根を返す。また、`x^2` とすると `x` の各要素を 2 乗するが、同様に任意のベキ（指数）も指定できる（小数や負の指数も可である）。

```
sqrt(x)
##          [,1]     [,2]
## [1,] 1.000000 1.732051
## [2,] 1.414214 2.000000
x^2
##      [,1] [,2]
## [1,]    1    9
## [2,]    4   16
```

`rnorm()`関数はランダムな正規分布の変数ベクトルを生成する。第 1 引数の `n` がサンプルサイズ、呼び出すたびに異なる結果が得られる。

訳注 3：乱数(random number)は初期値に依存するが、`R` ではデフォルトで自動的に毎回変わる。初期値を固定することは次に説明する `set.seed()` を利用すると可能となる。

ここで、`x` と `y` という 2 つの相関のある数値セットを作成し、`cor()`関数で相関を計算してみよう。

```
x <- rnorm(50)
y <- x + rnorm(50, mean = 50, sd = .1)
cor(x, y)
## [1] 0.9952277
```

デフォルトでは、rnorm()は平均 0、標準偏差 1 の標準正規分布を生成するが、上記のように mean と sd 引数で平均と標準偏差を変更できる。同じ乱数セットを再現したい場合は、set.seed()関数を使えば可能だが set.seed()は任意の整数引数、例えば 12345 などに設定できる。

```
set.seed(1303)
rnorm(50)
##  [1] -1.1439763145  1.3421293656  2.1853904757  0.5363925179  0.0631929665
##  [6]  0.5022344825 -0.0004167247  0.5658198405 -0.5725226890 -1.1102250073
## [11] -0.0486871234 -0.6956562176  0.8289174803  0.2066528551 -0.2356745091
## [16] -0.5563104914 -0.3647543571  0.8623550343 -0.6307715354  0.3136021252
## [21] -0.9314953177  0.8238676185  0.5233707021  0.7069214120  0.4202043256
## [26] -0.2690521547 -1.5103172999 -0.6902124766 -0.1434719524 -1.0135274099
## [31]  1.5732737361  0.0127465055  0.8726470499  0.4220661905 -0.0188157917
## [36]  2.6157489689 -0.6931401748 -0.2663217810 -0.7206364412  1.3677342065
## [41]  0.2640073322  0.6321868074 -1.3306509858  0.0268888182  1.0406363208
## [46]  1.3120237985 -0.0300020767 -0.2500257125  0.0234144857  1.6598706557
```

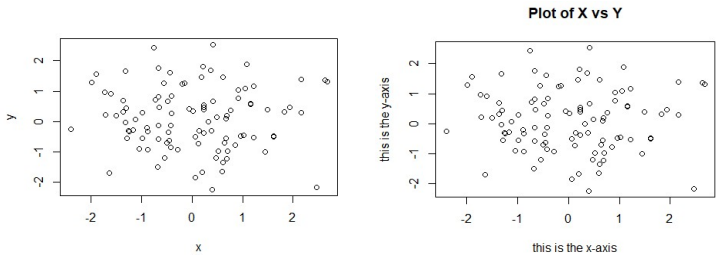
set.seed()を利用するとランダムな値を伴う計算に使用する際、結果を再現することができる。ただし、R の新バージョンがリリースされると、結果に若干の差異が生じることがありうる。 mean()と var()関数を使用すると、数値ベクトルの平均と分散が計算できる。分散の出力に sqrt()を適用すると標準偏差が得られる。あるいは標準偏差については sd()関数を使用することも可能である。

```
set.seed(3)
y <- rnorm(100)
mean(y)
## [1] 0.01103557
var(y)
## [1] 0.7328675
sqrt(var(y))
## [1] 0.8560768
sd(y)
## [1] 0.8560768
```

グラフィックス

`plot()`関数は、`R`でデータをプロットする主な方法である。例えば、`plot(x, y)`は`x`の数値と`y`の数値の散布図を作成する。`plot()`関数にはさまざまなオプションを指定でき、例えば、引数`xlab`を渡すと、x 軸にラベルが表示される。詳しい情報は`?plot`を参照してみると良い。

```
x <- rnorm(100)
y <- rnorm(100)
plot(x, y)
plot(x, y, xlab = "this is the x-axis",
    ylab = "this is the y-axis",
    main = "Plot of X vs Y")
```



訳注 4：図の説明を日本語にしたい場合には、例えば`plot(x, y,xlab="変数 X", ylab="変数 Y" ,main="散布図")`とすればよい。

`R`プロットの出力を保存することがよくあるが、保存するファイルタイプに応じてコマンドが異なる。例えば、PDF ファイルを作成するには`pdf()`関数、JPEG を作成するには`jpeg()`関数を使用する。

```
pdf("Figure.pdf")
plot(x, y, col = "green")
dev.off()
## png
##   2
```

`dev.off()`関数は、`R`にプロットの作成が終了したことを知らせる。あるいは、その代わりにプロット・ウィンドウをコピーして、Word 文書など適切なファイル形式に貼り付けることも可能である。

また`seq()`関数を使用して数値のシーケンスを作成できる。例えば、`seq(a, b)`は`a`から`b`までの整数を含むベクトルを生成する。その他にも多くのオプションがあり、たとえば`seq(0, 1, length = 10)`は`0`から`1`までの間に等間隔で`10`個の数値を生成する。整数の引数の場合、`3:11`と入力すると`seq(3, 11)`の省略形を意味する。

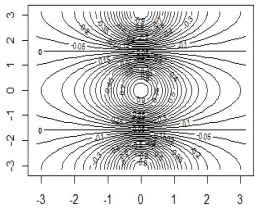
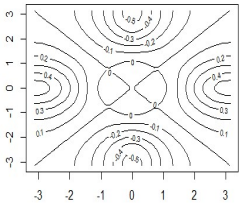
```
x <- seq(1, 10)
x
##  [1]  1  2  3  4  5  6  7  8  9 10
x <- 1:10
x
##  [1]  1  2  3  4  5  6  7  8  9 10
x <- seq(-pi, pi, length = 50)
```

次に、より高度なプロットを作成してみる。`contour()`関数は 3 次元データを表す等高線図を生成するが、これは地形図のようなものであり、3 つの引数をとる：

- `x`の値のベクトル（第 1 次元）
- `y`の値のベクトル（第 2 次元）
- 各`(x, y)`座標の`z`値（第 3 次元）に対応する行列

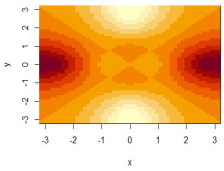
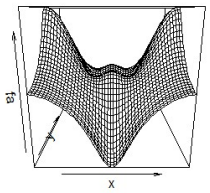
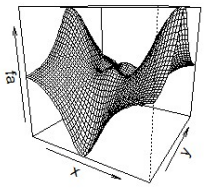
`plot()`関数と同様に、`contour()`関数にも出力を微調整するための追加の引数があるが、詳細は`?contour`で確認できる。

```
y <- x
f <- outer(x, y, function(x, y) cos(y) / (1 + x^2))
contour(x, y, f)
contour(x, y, f, nlevels = 45, add = T)
fa <- (f - t(f)) / 2
contour(x, y, fa, nlevels = 15)
```

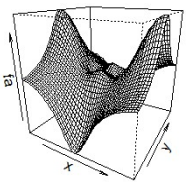
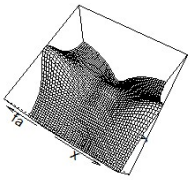
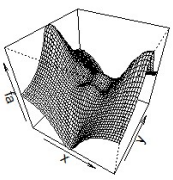



`image()`関数も`contour()`と同様に動作するが、色で`z`値を表すカラーコード・プロットを作成する。また、`persp()`は 3 次元プロットを作成する。引数`theta`と`phi`で視点の角度を制御できる。

```
image(x, y, fa)
persp(x, y, fa)
persp(x, y, fa, theta = 30)
```

```
persp(x, y, fa, theta = 30, phi = 20)
persp(x, y, fa, theta = 30, phi = 70)
persp(x, y, fa, theta = 30, phi = 40)
```

データのインデックス操作

データの一部だけを確認したいことがよくある。ここではデータが行列`A`に格納されていると仮定してみよう。

```
A <- matrix(1:16, 4, 4)
A
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
## [3,]    3    7   11   15
## [4,]    4    8   12   16
```

ここで以下のように入力すると、2 行 3 列目の要素が選択される。

```
A[2, 3]
## [1] 10
```

最初の数字は行、次の数字は列を表す。複数の行や列を選択する場合、インデックスにベクトルを使用することができる。

```
A[c(1, 3), c(2, 4)]
##      [,1] [,2]
## [1,]    5   13
## [2,]    7   15
A[1:3, 2:4]
##      [,1] [,2] [,3]
## [1,]    5    9   13
## [2,]    6   10   14
## [3,]    7   11   15
A[1:2, ]
##      [,1] [,2] [,3] [,4]
## [1,]    1    5    9   13
## [2,]    2    6   10   14
A[, 1:2]
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
```

```
## [3,]    3    7
## [4,]    4    8
```

最後の 2 つの例では、列のインデックスまたは行のインデックスが省略されている。これは`R`に対し、「すべての列」または「すべての行」を選択するよう指示することを意味する。また、`R`は行列の 1 行または 1 列をベクトルとして扱う。

```
A[1, ]
## [1]  1  5  9 13
```

インデックスに負の記号`-`を用いると、`R`では、指定した行や列以外を選択できる。

```
A[-c(1, 3), ]
##      [,1] [,2] [,3] [,4]
## [1,]    2    6   10   14
## [2,]    4    8   12   16
A[-c(1, 3), -c(1, 3, 4)]
## [1] 6 8
```

`dim()`関数は、行列の行数と列数を出力する。

```
dim(A)
## [1] 4 4
```

データの読み込み

ほとんどの分析において、最初のステップはデータセットを`R`にインポートすることになる。`read.table()`関数は、このための主要な方法の一つであるが、この関数の使い方の詳細については、ヘルプファイルに記載されている。データのエクスポートには`write.table()`関数を使用できる。

データセットを読み込む前に、R が適切なディレクトリを検索するように指定する必要がある。例えば、Windows システムでは、ファイルメニューの「ディレクトリの変更...」オプションを使用してディレクトリを選択することができる。ただし、この操作は使用するオペレーティングシステム（例：Windows、Mac、Unix）によって異なるため、ここでは詳細を省略する。

訳注 5：Windows の場合にはディレクトリを知りたい場合には>getwd(),また>setwd("C:/R")とすると C：/R に変更できる。C:/R 上の Auto.data ファイルは C:/R/Auto.data となる。また R を立ち上げてファイル,ディレクトリの変更を利用しても良い。エクセルファイルを直接読み込むにはファイルのコピーを利用して R 上で>Data=read.delim("clipbord")とすると Data$z(ファイル名)として利用可能となる。

まず、Auto データセットを読み込んでみよう。このデータは、第 3 章で説明されている ISLR2 ライブラリの一部である。read.table()関数を説明するために、ここでは、教科書のウェブサイトにあるテキストファイル Auto.data から読み込む。次のコマンドで Auto.data ファイルを R に読み込み、Auto というオブジェクトに格納する。データが読み込まれた後は、View()関数を使用してスプレッド・シートのようなウィンドウでデータを表示することができる。（この関数は時々少し扱いにくいことがあり、使用に問題が生じる場合には、代わりに head()関数を試してみるとよい。）head()関数はデータの最初の数行を表示するためにも使用できる。

訳注 6：必要なデータは https://www.statlearning.com/resources-second-edition の Data Sets からダウンロードすればよい。データは csv 形式 Auto.csv（エクセルで読める）とテキスト形式 Auto.data が用意されている。

```
Auto <- read.table("Auto.data")
View(Auto)
head(Auto)
##     V1        V2           V3         V4     V5           V6   V7     V8
## 1  mpg cylinders displacement horsepower weight acceleration year origin
## 2 18.0         8        307.0      130.0  3504.         12.0   70      1
## 3 15.0         8        350.0      165.0  3693.         11.5   70      1
## 4 18.0         8        318.0      150.0  3436.         11.0   70      1
## 5 16.0         8        304.0      150.0  3433.         12.0   70      1
## 6 17.0         8        302.0      140.0  3449.         10.5   70      1
##                          V9
## 1                      name
## 2 chevrolet chevelle malibu
```

```
## 3          buick skylark 320
## 4         plymouth satellite
## 5              amc rebel sst
## 6                ford torino
```

Auto.data は単なるテキストファイルであり、標準的なテキストエディタで開くこともできる。データを R に読み込む前に、テキストエディタや Excel などでデータセットを確認することをお勧めする。

実はこのデータセットは正しく読み込まれていない。R は変数名をデータの一部と見なしてしまい、最初の行に含めている。また、欠測値が?で示されている箇所もある。欠測値は現実のデータセットで一般的に見られる。この場合には read.table()関数で header = T（または header = TRUE）オプションを使用することで、ファイルの最初の行が変数名を含むことを R に指示し、na.strings オプションを使用して R が特定の文字（例えば?）を欠測データと見なすように指定することができる。

```
Auto <- read.table("Auto.data", header = T, na.strings = "?", stringsAsFactors
= T)
View(Auto)
```

stringsAsFactors = T 引数は、文字列を含む変数を質的変数として解釈し、各文字列がその質的変数の異なるレベルを表すことを R に指示する。Excel から R にデータを読み込む簡単な方法は、データを csv（カンマ区切り値）ファイルとして保存し、read.csv()関数を使用することである。

```
Auto <- read.csv("Auto.csv", na.strings = "?", stringsAsFactors = T)
View(Auto)
dim(Auto)
## [1] 392   9
Auto[1:4, ]
##   mpg cylinders displacement horsepower weight acceleration year origin
## 1  18         8          307        130   3504         12.0   70      1
## 2  15         8          350        165   3693         11.5   70      1
## 3  18         8          318        150   3436         11.0   70      1
## 4  16         8          304        150   3433         12.0   70      1
```

```
##                          name
## 1 chevrolet chevelle malibu
## 2         buick skylark 320
## 3        plymouth satellite
## 4             amc rebel sst
```

`dim()`関数を使用すると、データに 397 件の観測値（行）と 9 つの変数（列）があることがわかる。欠測データの処理方法はいくつかあるが、この場合、欠測観測が含まれているのはわずか 5 行なので、`na.omit()`関数を使用してこれらの行を単純に削除することにする。

```
Auto <- na.omit(Auto)
dim(Auto)
## [1] 392   9
```

データが正しく読み込まれたら、`names()`を使用して変数名を確認できる。

```
names(Auto)
## [1] "mpg"          "cylinders"    "displacement" "horsepower"   "weight"
## [6] "acceleration" "year"         "origin"       "name"
```

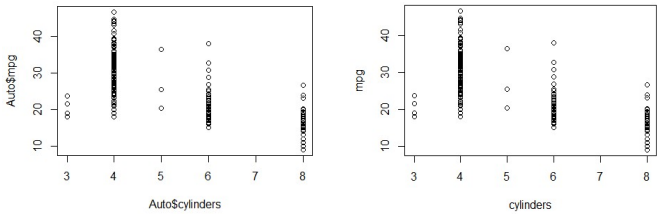
追加のグラフィカルおよび数値的な要約

`plot()`関数を使うと、定量的な変数の散布図を作成できる。しかし、変数名だけを入力するとエラーメッセージが表示されてしまう。これは、`R`が`Auto`データセットの中にある変数を参照する方法が分からないためである。

```
plot(cylinders, mpg)
## Error in eval(expr, envir, enclos): オブジェクト 'cylinders' がありません
```

変数を参照するには、データセットと変数名を記号`$`で結合して記述する必要がある。あるいは、`attach()`関数を使用して、このデータフレーム内の変数を名前で使用できるように`R`に指示することもできる。

```
plot(Auto$cylinders, Auto$mpg)
attach(Auto)
plot(cylinders, mpg)
```

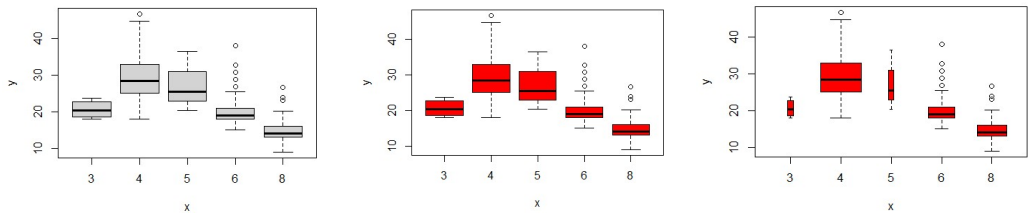


`cylinders`変数は数値ベクトルとして格納されているため、`R`はそれを定量的なものとして扱っている。しかし、`cylinders`の取り得る値が少ないため、定性的な変数として扱うほうが適切な場合もある。`as.factor()`関数により定量変数を定性変数に変換できる。

```
cylinders <- as.factor(cylinders)
```

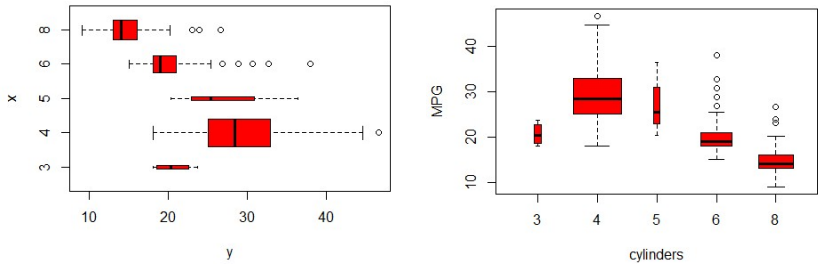
x 軸にプロットする変数が定性的な場合、`plot()`関数は自動的に箱ひげ図を作成する。また、プロットをカスタマイズするためにいくつかのオプションを指定することができる。

```
plot(cylinders, mpg)
plot(cylinders, mpg, col = "red")
plot(cylinders, mpg, col = "red", varwidth = T)
```



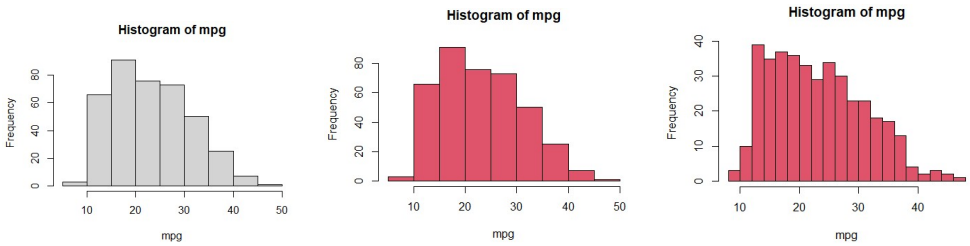

```
plot(cylinders, mpg, col = "red", varwidth = T,
    horizontal = T)
```

```
plot(cylinders, mpg, col = "red", varwidth = T,
    xlab = "cylinders", ylab = "MPG")
```



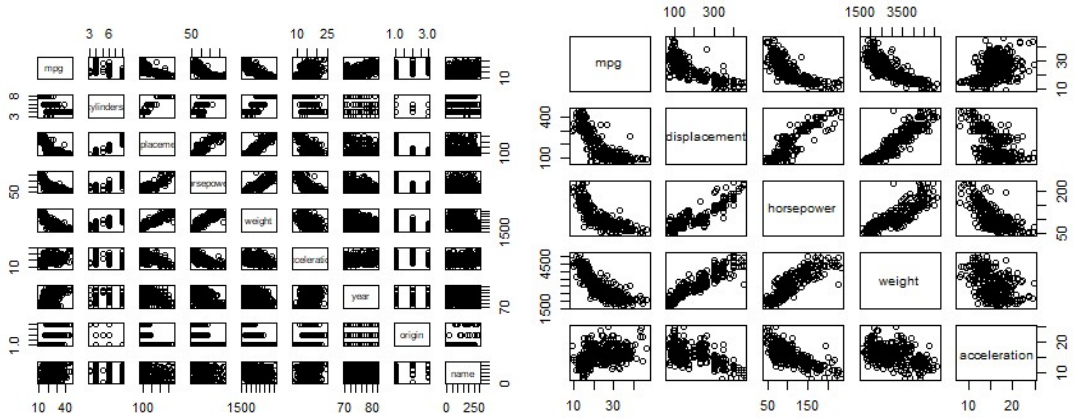
`hist()`関数を使用してヒストグラムをプロットできる。なお、`col = 2`とすることで、`col = "red"`と同じ効果が得られる。

```
hist(mpg)
hist(mpg, col = 2)
hist(mpg, col = 2, breaks = 15)
```



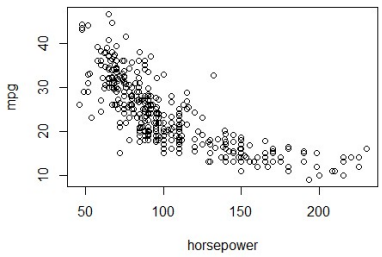
`pairs()`関数は散布図行列（すべての変数のペアについて散布図）を作成する。また、変数の一部だけについて散布図を生成することも可能である。

```
pairs(Auto)
pairs(
    ~ mpg + displacement + horsepower + weight + acceleration,
    data = Auto
  )
```



`plot()`関数と組み合わせて、`identify()`関数を使うと、プロット上の特定のポイントの変数値をインタラクティブに識別することができる。`identify()`には x 軸変数、y 軸変数、そしてポイントの値を確認したい変数を渡す。プロット上のポイントをクリックして Esc キーを押すと、指定した変数の値を表示してくれる。ここで `identify()`関数で表示される数字は、選択されたポイントの行番号である。

```
plot(horsepower, mpg)
identify(horsepower, mpg, name)
```




```
## integer(0)
```

`summary()`関数は、データセット内の各変数の数値的な要約を生成する。

```
summary(Auto)
```

```
##       mpg          cylinders      displacement     horsepower        weight
##  Min.   : 9.00   Min.   :3.000   Min.   : 68.0   Min.   : 46.0   Min.   :1613
```

```
##  1st Qu.:17.00   1st Qu.:4.000   1st Qu.:105.0   1st Qu.: 75.0   1st Qu.:2225
##  Median :22.75   Median :4.000   Median :151.0   Median : 93.5   Median :2804
##  Mean   :23.45   Mean   :5.472   Mean   :194.4   Mean   :104.5   Mean   :2978
##  3rd Qu.:29.00   3rd Qu.:8.000   3rd Qu.:275.8   3rd Qu.:126.0   3rd Qu.:3615
##  Max.   :46.60   Max.   :8.000   Max.   :455.0   Max.   :230.0   Max.   :5140
##
##   acceleration        year           origin                      name
##  Min.   : 8.00   Min.   :70.00   Min.   :1.000   amc matador       :  5
##  1st Qu.:13.78   1st Qu.:73.00   1st Qu.:1.000   ford pinto        :  5
##  Median :15.50   Median :76.00   Median :1.000   toyota corolla    :  5
##  Mean   :15.54   Mean   :75.98   Mean   :1.577   amc gremlin       :  4
##  3rd Qu.:17.02   3rd Qu.:79.00   3rd Qu.:2.000   amc hornet        :  4
##  Max.   :24.80   Max.   :82.00   Max.   :3.000   chevrolet chevette:  4
##                                                  (Other)           :365
```

定性的な変数（`name`など）に対しては、各カテゴリに属する観測値の数が表示される。さらに特定の変数のみを要約することもできる。

```
summary(mpg)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##    9.00   17.00   22.75   23.45   29.00   46.60
```

`R`の使用を終了する際には、`q()`と入力して終了する。`R`を終了する際に、現在のセッションで作成したすべてのオブジェクト（データセットなど）を次回のセッションでも利用できるよう保存するオプションが表示される。終了する前に、最近のセッションで入力したコマンドの記録を保存したい場合は、`savehistory()`関数を使えばよい。次回`R`に入った時に`loadhistory()`関数を使用してその履歴を読み込むことが可能となる。

ISLR 第 3 章 実習：線形回帰

ライブラリ

`library()` 関数は、基本の `R` ディストリビューションには含まれていない関数群やデータセット群をロードするために使用される。最小二乗線形回帰などの基本的な関数は標準で `R` に含まれているが、より特殊な機能には追加のライブラリが必要となる。ここでは、多くのデータセットと関数が含まれる `MASS` パッケージ、および本書に関連するデータセットが含まれる `ISLR2` パッケージを読み込んでみよう。

```
library(MASS)
library(ISLR2)
##
## 次のパッケージを付け加えます: 'ISLR2'

## 以下のオブジェクトは 'package:MASS' からマスクされています:

##
##     Boston
```

これらのライブラリの読み込み時にエラーメッセージが表示された場合、対応するライブラリがシステムにインストールされていない可能性がある。`MASS` のようなライブラリは `R` に標準で付属しているため、別途インストールすることは不要であるが、`ISLR2` のように最初にダウンロードする必要があるパッケージもある。たとえば、Windows システムでは、`R` の画面上でパッケージタブの「パッケージのインストール」オプションを選択し、任意のミラーサイトを選択すると、インストール可能なパッケージのリストが表示される。目的のパッケージを選択すると、`R` にダウンロードしてくれる。他の方法として、`R` コマンドラインから `install.packages("ISLR2")` と入力すればインストールされる。このインストールは最初の 1 回だけ必要で、その後は `R` セッションごとに `library()` 関数で読み込んでくれる。

訳注１：R から直接にパッケージをインストールしたい場合には、まず R 右上のパッケージから CRAN ミラーサイト(Japan)を指定、パッケージをダウンロード、その後に「**パッケージの読み込み**」を行う必要がある。

単回帰分析

ISLR2 ライブラリには、ボストンの 506 地域における住宅価格中央値 (medv) を記録した Boston データセットが含まれている。ここでは、rmvar（1 世帯あたりの部屋数の平均）、age（1940 年以前に建設された持ち家の割合）、lstat（低所得層の世帯の割合）など、12 の説明変数(予測子)を用いて価格 medv を予測してみよう。

```
head(Boston)
```

```
##      crim zn indus chas   nox    rm  age    dis rad tax ptratio lstat medv
## 1 0.00632 18  2.31    0 0.538 6.575 65.2 4.0900   1 296    15.3  4.98 24.0
## 2 0.02731  0  7.07    0 0.469 6.421 78.9 4.9671   2 242    17.8  9.14 21.6
## 3 0.02729  0  7.07    0 0.469 7.185 61.1 4.9671   2 242    17.8  4.03 34.7
## 4 0.03237  0  2.18    0 0.458 6.998 45.8 6.0622   3 222    18.7  2.94 33.4
## 5 0.06905  0  2.18    0 0.458 7.147 54.2 6.0622   3 222    18.7  5.33 36.2
## 6 0.02985  0  2.18    0 0.458 6.430 58.7 6.0622   3 222    18.7  5.21 28.7
```

このデータセットについて詳しく知りたい場合は、 ?Boston と入力すればよい。

まず、lm() 関数を使用して、medv を目的変数、lstat を説明変数として単回帰モデルをフィットしてみよう。基本的な構文は lm(y ~ x, data) で、y は目的変数、x は説明変数、data は変数が保持されているデータセットである。

```
lm.fit <- lm(medv ~ lstat)
```

```
## Error in eval(predvars, data, env): オブジェクト 'medv' がありません
```

エラーが発生したのは、R が medv と lstat の場所を認識していないためである。次の行で、R に変数が Boston にあることを伝える。Boston データ をアタッチすると、R が変数を認識するため、最初の行が正常に動作する。

```
lm.fit <- lm(medv ~ lstat, data = Boston)
attach(Boston)
lm.fit <- lm(medv ~ lstat)
```

`lm.fit` を入力すると、モデルに関する基本情報が出力される。詳細な情報を確認するには `summary(lm.fit)` を使用すればよい。これにより、係数のp値や標準誤差、R^2統計量、F統計量などが得られる。

```
lm.fit


##
## Call:
## lm(formula = medv ~ lstat)
##
## Coefficients:
## (Intercept)        lstat
##       34.55        -0.95
summary(lm.fit)
##
## Call:
## lm(formula = medv ~ lstat)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -15.168  -3.990  -1.318   2.034  24.500
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 34.55384    0.56263   61.41   <2e-16 ***
## lstat       -0.95005    0.03873  -24.53   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.216 on 504 degrees of freedom
## Multiple R-squared:  0.5441, Adjusted R-squared:  0.5432
## F-statistic: 601.6 on 1 and 504 DF,  p-value: < 2.2e-16
```

`names()` 関数を使用すると、`lm.fit` に格納されているその他の情報を確認できる。これらの値は、`lm.fit$coefficients` のように名前で取得できるが、`coef()` などの抽出関数(Extractor Function)を使う方が安全である。

```
names(lm.fit)
##  [1] "coefficients"  "residuals"     "effects"       "rank"
##  [5] "fitted.values" "assign"        "qr"            "df.residual"
##  [9] "xlevels"       "call"          "terms"         "model"
coef(lm.fit)
## (Intercept)       lstat
##  34.5538409  -0.9500494
```

係数推定値の信頼区間を得るには、`confint()` コマンドを利用する。コマンドラインで `confint(lm.fit)` を入力すると、信頼区間が表示される。

```
confint(lm.fit)
##                 2.5 %     97.5 %
## (Intercept) 33.448457 35.6592247
## lstat       -1.026148 -0.8739505
```

`predict()` 関数を使うと、特定の `lstat` 値に対する `medv` による推定(予測)に対して信頼区間と予測区間が生成される。

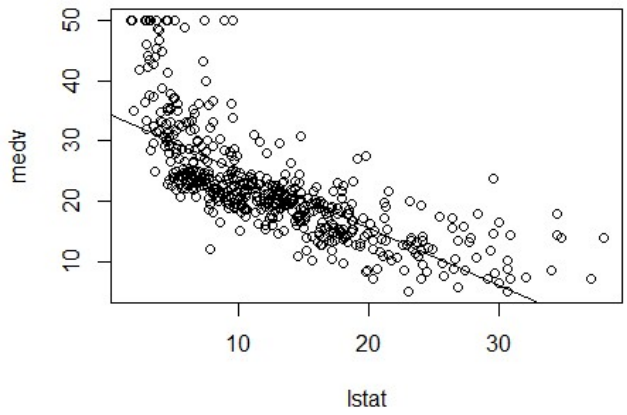
```
predict(lm.fit, data.frame(lstat = (c(5, 10, 15))),
    interval = "confidence")
##        fit      lwr      upr
## 1 29.80359 29.00741 30.59978
## 2 25.05335 24.47413 25.63256
## 3 20.30310 19.73159 20.87461
predict(lm.fit, data.frame(lstat = (c(5, 10, 15))),
    interval = "prediction")
##        fit       lwr      upr
## 1 29.80359 17.565675 42.04151
```

```
## 2 25.05335 12.827626 37.27907
## 3 20.30310  8.077742 32.52846
```

たとえば、`lstat` 値が 10 の場合の 95%信頼区間は (24.47,25.63) であり、95%予測区間は (12.828,37.28) となる。予想される通りに、信頼区間と予測区間は同じ点（`lstat` が 10 のときの予測値は 25.05）を中心にしているが、予測区間の方が大幅に広くなっている。

次に、`plot()` と `abline()` 関数を使って `medv` と `lstat` をプロットし、最小二乗回帰直線を描画してみよう。

```
plot(lstat, medv)
abline(lm.fit)
```

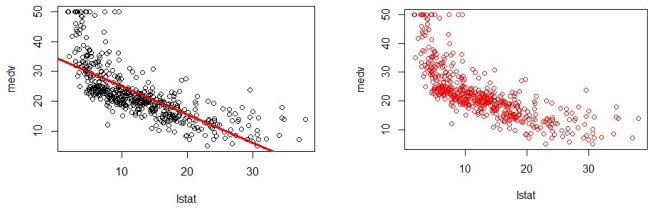


`lstat` と `medv` の関係には非線形性が存在する証拠が見つかるようであるが、この問題については、後で詳しく調べることにする。

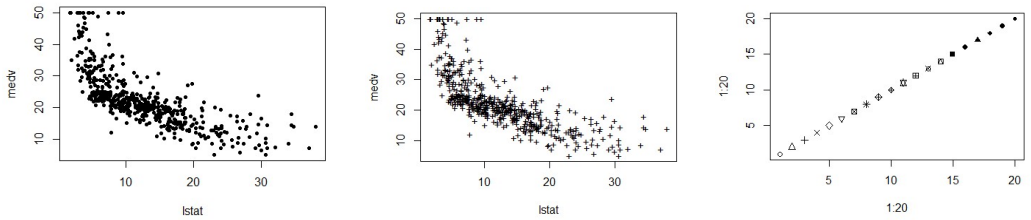
`abline()` 関数は最小二乗回帰直線以外にも任意の直線を描画に利用できる。傾きが `b`、切片が `a` の直線を描画するには `abline(a, b)` と入力すればよい。以下では、線や点のプロットの設定についていくつか実験を行ってみよう。 `lwd = 3` コマンドは回帰直線の幅を 3 倍にする。これは `plot()` や `lines()` 関数にも適用できる。また、`pch` オプションを使用すれば異なるプロット記号を利用できる。

```
plot(lstat, medv)
abline(lm.fit, lwd = 3)
abline(lm.fit, lwd = 3, col = "red")
```

```
plot(lstat, medv, col = "red")
```

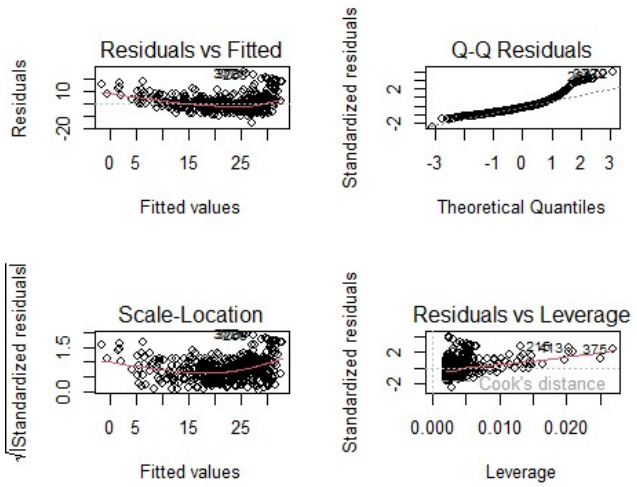



```
plot(lstat, medv, pch = 20)
plot(lstat, medv, pch = "+")
plot(1:20, 1:20, pch = 1:20)
```



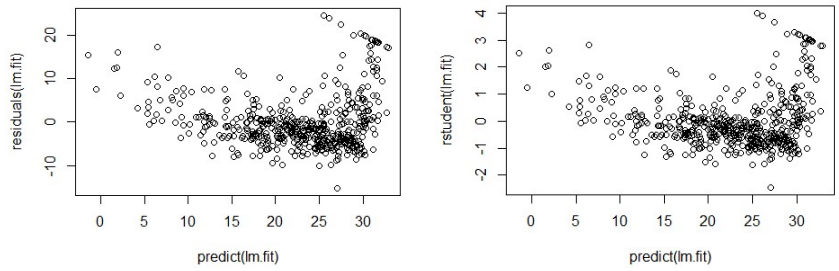
次にいくつかの診断プロットを利用してみよう。これらのうちいくつかは 3.3.3 節で説明されている。`plot()` 関数を `lm()` の出力に直接適用すると、4 つの診断プロットが自動的に生成される。このコマンドは通常 1 回に 1 つのプロットを生成し、Enter を押すと次のプロットが生成される。しかし、4 つのプロットを同時に表示する方が便利な場合もある。そのためには `par()` および `mfrow()` 関数を利用し、`R` に表示画面を分割して複数のプロットを同時に表示するように指示する必要がある。たとえば、`par(mfrow = c(2, 2))` とすればプロット領域を2×2のグリッドに分割できる。

```
par(mfrow = c(2, 2))
plot(lm.fit)
```



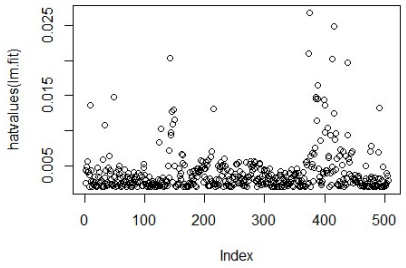
また、`residuals()` 関数を利用すると線形回帰フィットの残差を計算することもできる。`rstudent()` 関数によりスチューデント化残差を返し、この関数を使って予測値に対する残差のプロットが作成できる。

```
plot(predict(lm.fit), residuals(lm.fit))
plot(predict(lm.fit), rstudent(lm.fit))
```



残差プロットに基づくと、かなり非線形性の証拠を見つけられる。任意の説明変数に対して、`hatvalues()` 関数を使用してレバレッジ統計量（てこ比）を計算することができる。

```r
plot(hatvalues(lm.fit))
```




```r
which.max(hatvalues(lm.fit))
## 375
## 375
```

`which.max()` 関数は、ベクトルの中で最も大きい要素のインデックスを識別する。この場合、どの観測値が最大のレバレッジ統計量を持つかが示される。

重回帰分析

最小二乗法を用いて重回帰モデルをフィットするために、再び `lm()` 関数を利用する。3 つの説明変数(予測子) `x1`、`x2`、`x3` を利用した回帰モデルをフィットするには、次のような構文 `lm(y ~ x1 + x2 + x3)` を使用する。`summary()` 関数は、すべての予測子に対する回帰係数を出力する。

```r
lm.fit <- lm(medv ~ lstat + age, data = Boston)
summary(lm.fit)


##
## Call:
## lm(formula = medv ~ lstat + age, data = Boston)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -15.981  -3.978  -1.283   1.968  23.158
##
```

```
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept) 33.22276    0.73085  45.458  < 2e-16 ***
## lstat       -1.03207    0.04819 -21.416  < 2e-16 ***
## age          0.03454    0.01223   2.826  0.00491 **
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.173 on 503 degrees of freedom
## Multiple R-squared:  0.5513, Adjusted R-squared:  0.5495
## F-statistic:   309 on 2 and 503 DF,  p-value: < 2.2e-16
```

`Boston`データセットには 12 の変数が含まれているため、すべての説明変数(予測子)を使って回帰を実行する際にそれらをすべてのデータを入力するのはかなり面倒である。そこで代わりに、次のような省略形を利用できる。

```
lm.fit <- lm(medv ~ ., data = Boston)
summary(lm.fit)
```

```
##
## Call:
## lm(formula = medv ~ ., data = Boston)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.1304  -2.7673  -0.5814   1.9414  26.2526
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  41.617270   4.936039   8.431 3.79e-16 ***
## crim         -0.121389   0.033000  -3.678 0.000261 ***
## zn            0.046963   0.013879   3.384 0.000772 ***
## indus         0.013468   0.062145   0.217 0.828520
## chas          2.839993   0.870007   3.264 0.001173 **
## nox         -18.758022   3.851355  -4.870 1.50e-06 ***
## rm            3.658119   0.420246   8.705  < 2e-16 ***
## age           0.003611   0.013329   0.271 0.786595
```

```
## dis          -1.490754   0.201623  -7.394 6.17e-13 ***
## rad           0.289405   0.066908   4.325 1.84e-05 ***
## tax          -0.012682   0.003801  -3.337 0.000912 ***
## ptratio      -0.937533   0.132206  -7.091 4.63e-12 ***
## lstat        -0.552019   0.050659 -10.897  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.798 on 493 degrees of freedom
## Multiple R-squared:  0.7343, Adjusted R-squared:  0.7278
## F-statistic: 113.5 on 12 and 493 DF,  p-value: < 2.2e-16
```

ここで `summary` の個別の要素には名前でアクセスできる（利用可能な要素は `?summary.lm` で確認できる）。たとえば、`summary(lm.fit)$r.sq` はR^2を、`summary(lm.fit)$sigma` は残差標準誤差(RSE)を返す。`car` パッケージの一部である `vif()` 関数を使用すると、分散拡大係数 (VIF) を計算することができる。ほとんどの VIF はこのデータでは低から中程度となっている。なおこの `car` パッケージは `R` の基本インストールに含まれていないため、初回使用時には `install.packages()` 関数を使ってダウンロードする必要がある。

```
library(car)
```

```
## 要求されたパッケージ carData をロード中です
```

```
vif(lm.fit)
```

```
##     crim       zn    indus     chas      nox       rm      age      dis
## 1.767486 2.298459 3.987181 1.071168 4.369093 1.912532 3.088232 3.954037
##      rad      tax  ptratio    lstat
## 7.445301 9.002158 1.797060 2.870777
```

ここですべての説明変数を利用せず、1 つだけ除外して回帰を実行したい場合はどうすればよいだろうか？たとえば、上記の回帰出力で `age` の p値が高いので、この予測子を除外して回帰を実行することが考えられるだろう。そこで以下の構文では、`age` を除いたすべての予測子を使用して回帰を行ってみた。

```
lm.fit1 <- lm(medv ~ . - age, data = Boston)
summary(lm.fit1)
```

```
##
## Call:
## lm(formula = medv ~ . - age, data = Boston)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.1851  -2.7330  -0.6116   1.8555  26.3838
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept)  41.525128   4.919684   8.441 3.52e-16 ***
## crim         -0.121426   0.032969  -3.683 0.000256 ***
## zn            0.046512   0.013766   3.379 0.000785 ***
## indus         0.013451   0.062086   0.217 0.828577
## chas          2.852773   0.867912   3.287 0.001085 **
## nox         -18.485070   3.713714  -4.978 8.91e-07 ***
## rm            3.681070   0.411230   8.951  < 2e-16 ***
## dis          -1.506777   0.192570  -7.825 3.12e-14 ***
## rad           0.287940   0.066627   4.322 1.87e-05 ***
## tax          -0.012653   0.003796  -3.333 0.000923 ***
## ptratio      -0.934649   0.131653  -7.099 4.39e-12 ***
## lstat        -0.547409   0.047669 -11.483  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 4.794 on 494 degrees of freedom
## Multiple R-squared:  0.7343, Adjusted R-squared:  0.7284
## F-statistic: 124.1 on 11 and 494 DF,  p-value: < 2.2e-16
```

あるいはここで、`update()` 関数を使用することもできる。

```
lm.fit1 <- update(lm.fit, ~ . - age)
```

交互作用項

`lm()` 関数を使用して線形モデルに交互作用項を簡単に追加することができる。
`lstat:age` という構文は、`lstat` と `age` 間の交互作用項を含めることを指示してい

る。`lstat * age` という構文は、`lstat` と `age` に加え、`lstat` × `age` の交互作用項も同時に含めることを意味する。これは `lstat + age + lstat:age` の省略形である。また、変換された変数(予測子)を説明変数として渡すこともできる。

```
summary(lm(medv ~ lstat * age, data = Boston))
```

```
##
## Call:
## lm(formula = medv ~ lstat * age, data = Boston)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -15.806  -4.045  -1.333   2.085  27.552
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) 36.0885359  1.4698355  24.553  < 2e-16 ***
## lstat       -1.3921168  0.1674555  -8.313 8.78e-16 ***
## age         -0.0007209  0.0198792  -0.036   0.9711
## lstat:age    0.0041560  0.0018518   2.244   0.0252 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.149 on 502 degrees of freedom
## Multiple R-squared:  0.5557, Adjusted R-squared:  0.5531
## F-statistic: 209.3 on 3 and 502 DF,  p-value: < 2.2e-16
```

このようにすれば、交互作用項や変数選択を活用して回帰モデルを柔軟に構築することができる。

説明変数(予測子)の非線形変換

`lm()` 関数では、説明変数(予測子)の非線形変換も扱うことができる。例えば、説明変数(予測子) X がある場合、`I(X^2)` を使って X^2 を作成することができる。`I()` 関数が必要なのは、`^` が式オブジェクト内で特別な意味を持っているためで、`I()` で囲むことにより `X` を 2 乗する通常の方法を利用することができる。ここでは、`medv` を `lstat` および `lstat^2` に対して回帰を行ってみよう。

```
lm.fit2 <- lm(medv ~ lstat + I(lstat^2))
summary(lm.fit2)
##
## Call:
## lm(formula = medv ~ lstat + I(lstat^2))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -15.2834  -3.8313  -0.5295   2.3095  25.4148
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept) 42.862007   0.872084   49.15   <2e-16 ***
## lstat       -2.332821   0.123803  -18.84   <2e-16 ***
## I(lstat^2)   0.043547   0.003745   11.63   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.524 on 503 degrees of freedom
## Multiple R-squared:  0.6407, Adjusted R-squared:  0.6393
## F-statistic: 448.5 on 2 and 503 DF,  p-value: < 2.2e-16
```

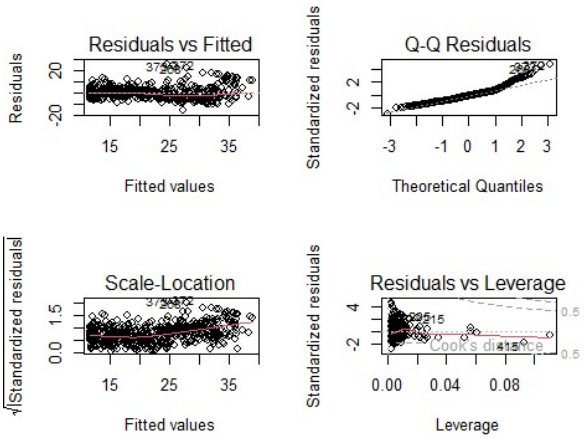
この 2 次項に関連するほぼゼロのp値は、この項を追加することでモデルが改善されることを示唆している。`anova()` 関数を使用して、2 次モデルが線形モデルよりどれだけ優れているかをさらに定量化することにしよう。

```
lm.fit <- lm(medv ~ lstat)
anova(lm.fit, lm.fit2)
## Analysis of Variance Table
##
## Model 1: medv ~ lstat
## Model 2: medv ~ lstat + I(lstat^2)
##   Res.Df   RSS Df Sum of Sq     F    Pr(>F)
## 1    504 19472
## 2    503 15347  1    4125.1 135.2 < 2.2e-16 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

ここで、モデル 1 は `lstat` だけを含む線形モデル、モデル 2 は `lstat` および `lstat^2` を含む線形モデルを表す。`anova()` 関数により 2 つのモデルを比較する仮説検定を行うことができる。帰無仮説は 2 つのモデルは同等にデータにフィットしているというもので、対立仮説は 2 次項を含むモデルが優れているというものである。この場合、F統計量は 135 で、関連するp値はほぼゼロとなっている。これは、`lstat` および `lstat^2` を含むモデルが `lstat` のみを含むモデルよりもはるかに優れていることを非常に明確に示している。これは、以前に `medv` と `lstat` の関係に非線形性がある証拠が観察されたので、ここで特に驚くべきことではない。

```
par(mfrow = c(2, 2))
plot(lm.fit2)
```



上記を実行すると、`lstat^2` 項がモデルに含まれると、残差に顕著なパターンがほとんど見られなくなることがわかる。

3 次関数をフィット作成するには、`I(X^3)` の形式の予測子を含めることにより可能となる。ただし、高次多項式に対してこの方法を利用していくと煩雑になる可能性がある。より良い方法は、`poly()` 関数を使用して `lm()` 内で多項式を作成することである。例えば、次のコマンドにより 5 次の多項式をフィットすることができる。

```
lm.fit5 <- lm(medv ~ poly(lstat, 5))
summary(lm.fit5)
```

```
##
## Call:
## lm(formula = medv ~ poly(lstat, 5))
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -13.5433  -3.1039  -0.7052   2.0844  27.1153
##
## Coefficients:
##                  Estimate Std. Error t value Pr(>|t|)
## (Intercept)       22.5328     0.2318  97.197  < 2e-16 ***
## poly(lstat, 5)1 -152.4595     5.2148 -29.236  < 2e-16 ***
## poly(lstat, 5)2   64.2272     5.2148  12.316  < 2e-16 ***
## poly(lstat, 5)3  -27.0511     5.2148  -5.187 3.10e-07 ***
## poly(lstat, 5)4   25.4517     5.2148   4.881 1.42e-06 ***
## poly(lstat, 5)5  -19.2524     5.2148  -3.692 0.000247 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 5.215 on 500 degrees of freedom
## Multiple R-squared:  0.6817, Adjusted R-squared:  0.6785
## F-statistic: 214.2 on 5 and 500 DF,  p-value: < 2.2e-16
```

これにより、5 次までの多項式項を追加することによりモデルのフィットの改善につながることが示唆されている。ただし、データをさらに調べると、5 次を超える多項式項は回帰モデルをフィットすると有意なp値が得られないことが分かっている。

デフォルトでは、`poly()` 関数は予測子を直交化する。つまり、この関数が出力する特徴量は引数のべき乗の変数ではない。

訳注 2：予測子（つまり説明変数）として多項式を直交化するとは、例えば 5 次多項式の変数を、共分散をゼロにするように変換することを意味している。

ただし、`poly()` 関数の出力に適用される線形モデルは、単純な多項式に適用される線形モデルとフィットは同一となる（係数の推定値、標準誤差、およびp値は異

なる場合がある）。単純な多項式を `poly()` 関数で利用するには、引数 `raw = TRUE` を使用する必要がある。 もちろん、予測子の多項式変換に限定されるわけではない。例えば、ここで対数変換を試みてみよう。

```
summary(lm(medv ~ log(rm), data = Boston))
```

```
##
## Call:
## lm(formula = medv ~ log(rm), data = Boston)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -19.487  -2.875  -0.104   2.837  39.816
##
## Coefficients:
##             Estimate Std. Error t value Pr(>|t|)
## (Intercept)  -76.488      5.028  -15.21   <2e-16 ***
## log(rm)       54.055      2.739   19.73   <2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 6.915 on 504 degrees of freedom
## Multiple R-squared:  0.4358, Adjusted R-squared:  0.4347
## F-statistic: 389.3 on 1 and 504 DF,  p-value: < 2.2e-16
```

質的な説明変数(予測子)

次に、`ISLR2` ライブラリの一部である `Carseats` データを調べてみよう。ここでは、400 箇所におけるチャイルドカーシートの販売量 (`Sales`) を、複数の説明変数(予測子)を用いて推定(予測)するのが問題である。

```
head(Carseats)
```

```
##   Sales CompPrice Income Advertising Population Price ShelveLoc Age Education
## 1  9.50       138     73          11        276   120       Bad  42        17
## 2 11.22       111     48          16        260    83      Good  65        10
## 3 10.06       113     35          10        269    80    Medium  59        12
```

```
## 4  7.40       117    100           4        466    97    Medium  55        14
## 5  4.15       141     64           3        340   128       Bad  38        13
## 6 10.81       124    113          13        501    72       Bad  78        16
##   Urban  US
## 1   Yes Yes
## 2   Yes Yes
## 3   Yes Yes
## 4   Yes Yes
## 5   Yes  No
## 6    No Yes
```

`Carseats` データには、`shelveloc` のような質的な予測子も含まれている。これは、各場所におけるカーシートの棚の場所の質（*Bad*、*Medium*、*Good* の 3 つの値）を示す指標である。このような質的な変数がある場合、`R` は自動的にダミー変数を生成する。以下では、いくつかの交互作用項を含む重回帰モデルをフィットしてみる。

```
lm.fit <- lm(Sales ~ . + Income:Advertising + Price:Age,
    data = Carseats)
summary(lm.fit)
```

```
##
## Call:
## lm(formula = Sales ~ . + Income:Advertising + Price:Age, data = Carseats)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -2.9208 -0.7503  0.0177  0.6754  3.3413
##
## Coefficients:
##                       Estimate Std. Error t value Pr(>|t|)
## (Intercept)         6.5755654  1.0087470   6.519 2.22e-10 ***
## CompPrice           0.0929371  0.0041183  22.567  < 2e-16 ***
## Income              0.0108940  0.0026044   4.183 3.57e-05 ***
## Advertising         0.0702462  0.0226091   3.107 0.002030 **
## Population          0.0001592  0.0003679   0.433 0.665330
## Price              -0.1008064  0.0074399 -13.549  < 2e-16 ***
```

```
## ShelveLocGood       4.8486762  0.1528378  31.724  < 2e-16 ***
## ShelveLocMedium     1.9532620  0.1257682  15.531  < 2e-16 ***
## Age                -0.0579466  0.0159506  -3.633 0.000318 ***
## Education          -0.0208525  0.0196131  -1.063 0.288361
## UrbanYes            0.1401597  0.1124019   1.247 0.213171
## USYes              -0.1575571  0.1489234  -1.058 0.290729
## Income:Advertising  0.0007510  0.0002784   2.698 0.007290 **
## Price:Age           0.0001068  0.0001333   0.801 0.423812
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 1.011 on 386 degrees of freedom
## Multiple R-squared:  0.8761, Adjusted R-squared:  0.8719
## F-statistic:   210 on 13 and 386 DF,  p-value: < 2.2e-16
```

`contrasts()` 関数は、`R` がダミー変数に使用するコーディングを返す。

```
attach(Carseats)
contrasts(ShelveLoc)
##        Good Medium
## Bad       0      0
## Good      1      0
## Medium    0      1
```

`shelveloc` が Good の場合に 1、そうでない場合に 0 をとるダミー変数 `ShelveLocGood` が自動的に生成される。同様に、変数 `ShelveLocMedium` も生成され、`shelveloc` が Medium の場合に 1、それ以外では 0 をとる。Bad の場合は 2 つのダミー変数がどちらもゼロになる。`ShelveLocGood` の回帰係数が正であることは、良い棚の場所が（悪い場所に対して）高い販売量と関連していることを示している。変数 `ShelveLocMedium` も正の係数だが、その値は Good より小さく、Medium の棚の場所が悪い棚の場所よりも販売量が高いが、良い棚の場所ほどではないことを示している。

関数の作成

`R` には多くの便利な関数が用意されているが、場合によっては利用可能な関数がない操作を行う必要が生じる。そのような場合には、自分で関数を作成することができる。以下では `LoadLibraries()` という名前の簡単な関数を例に挙げてみる。この関数により `ISLR2` と `MASS` ライブラリを読み込むことができる。

```
LoadLibraries
```

```
## Error in eval(expr, envir, enclos): オブジェクト 'LoadLibraries' がありません
```

```
LoadLibraries()
```

```
## Error in LoadLibraries(): 関数 "LoadLibraries" を見つけることができませんでした
```

`LoadLibraries()` という関数を作成する。ここで、`+` 記号は `R` によって自動的に表示されるものであり、手入力する必要はない。`{` 記号は、複数のコマンドが入力されることを `R` に通知する。`{` を入力して *Enter* を押すと、`R` は `+` 記号を表示する。その後、必要なだけコマンドを入力できる。各コマンドの入力後に *Enter* を押す。
最後に `}` 記号を入力すると、`R` はそれ以上のコマンドが入力されないと認識する。

```
LoadLibraries <- function() {
 library(ISLR2)
 library(MASS)
 print("The libraries have been loaded.")
}
```

ここで、`LoadLibraries` を入力すると,`R` はこの関数の中身を出力する.

```
LoadLibraries
```

```
## function() {
##  library(ISLR2)
##  library(MASS)
##  print("The libraries have been loaded.")
## }
```

この関数を呼び出すと、ライブラリが読み込まれ、メッセージが出力される。

```
LoadLibraries()
## [1] "The libraries have been loaded."
```

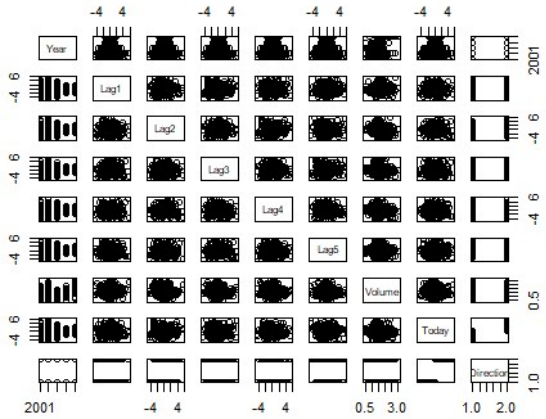
ISLR 第 4 章 実習：分類

4.7.1 株式市場データ

最初に、`ISLR2` ライブラリに含まれる `Smarket` データの数値的およびグラフ的な要約を確認しよう。このデータセットは、2001 年初頭から 2005 年末までの 1,250 日間にわたる S&P 500 株式指数のパーセントリターンを含んでいる。各日付について、前の 5 営業日のパーセントリターン（`Lag1` から `Lag5` まで）、前日の取引量（`Volume`、単位: 10 億株）、当日のパーセントリターン（`Today`）、および市場の方向性（`Up` または `Down`）が記録されている。目標は、他の特徴を使用して `Direction`（定性的な応答）を予測することである。

```
library(ISLR2)
names(Smarket)
## [1] "Year"      "Lag1"      "Lag2"      "Lag3"      "Lag4"      "Lag5"
## [7] "Volume"    "Today"     "Direction"
dim(Smarket)
## [1] 1250    9
summary(Smarket)
##       Year           Lag1                Lag2                Lag3
##  Min.   :2001   Min.   :-4.922000   Min.   :-4.922000   Min.   :-4.922000
##  1st Qu.:2002   1st Qu.:-0.639500   1st Qu.:-0.639500   1st Qu.:-0.640000
##  Median :2003   Median : 0.039000   Median : 0.039000   Median : 0.038500
##  Mean   :2003   Mean   : 0.003834   Mean   : 0.003919   Mean   : 0.001716
##  3rd Qu.:2004   3rd Qu.: 0.596750   3rd Qu.: 0.596750   3rd Qu.: 0.596750
##  Max.   :2005   Max.   : 5.733000   Max.   : 5.733000   Max.   : 5.733000
##       Lag4                Lag5              Volume           Today
##  Min.   :-4.922000   Min.   :-4.92200   Min.   :0.3561   Min.   :-4.922000
##  1st Qu.:-0.640000   1st Qu.:-0.64000   1st Qu.:1.2574   1st Qu.:-0.639500
##  Median : 0.038500   Median : 0.03850   Median :1.4229   Median : 0.038500
##  Mean   : 0.001636   Mean   : 0.00561   Mean   :1.4783   Mean   : 0.003138
##  3rd Qu.: 0.596750   3rd Qu.: 0.59700   3rd Qu.:1.6417   3rd Qu.: 0.596750
```

```
##  Max.   : 5.733000   Max.   : 5.73300   Max.   :3.1525   Max.   : 5.733000
##  Direction
##  Down:602
##  Up   :648
##
##
##
##
pairs(Smarket)
```



`cor()`関数は、データセット内の予測子間のすべてのペアワイズ相関を含む行列を生成する。ただし、`Direction`変数は定性的であるため、以下の最初のコマンドはエラーメッセージを出力してしまう。

```
cor(Smarket)
```

```
## Error in cor(Smarket): 'x' は数値でなければなりません
```

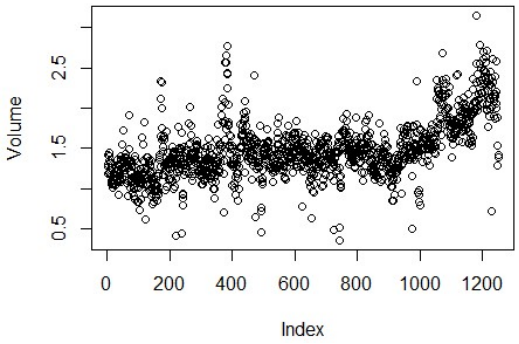
```
cor(Smarket[, -9])
```

```
##              Year         Lag1         Lag2         Lag3         Lag4
## Year   1.00000000  0.029699649  0.030596422  0.033194581  0.035688718
## Lag1   0.02969965  1.000000000 -0.026294328 -0.010803402 -0.002985911
## Lag2   0.03059642 -0.026294328  1.000000000 -0.025896670 -0.010853533
## Lag3   0.03319458 -0.010803402 -0.025896670  1.000000000 -0.024051036
## Lag4   0.03568872 -0.002985911 -0.010853533 -0.024051036  1.000000000
```

```
## Lag5   0.02978799 -0.005674606 -0.003557949 -0.018808338 -0.027083641
## Volume 0.53900647  0.040909908 -0.043383215 -0.041823686 -0.048414246
## Today  0.03009523 -0.026155045 -0.010250033 -0.002447647 -0.006899527
##                Lag5      Volume        Today
## Year    0.029787995  0.53900647  0.030095229
## Lag1   -0.005674606  0.04090991 -0.026155045
## Lag2   -0.003557949 -0.04338321 -0.010250033
## Lag3   -0.018808338 -0.04182369 -0.002447647
## Lag4   -0.027083641 -0.04841425 -0.006899527
## Lag5    1.000000000 -0.02200231 -0.034860083
## Volume -0.022002315  1.00000000  0.014591823
## Today  -0.034860083  0.01459182  1.000000000
```

予想通り、遅延変数と当日のリターンの間の相関はほぼゼロに近い。つまり、当日のリターンと過去の日のリターンとの間にほとんど相関はないようである。唯一の顕著な相関は、`Year` と `Volume` の間に見られる。データを時系列にプロットすると、`Volume` が時間とともに増加していることがわかる。つまり、1 日の平均取引量は 2001 年から 2005 年にかけて増加している。

```
attach(Smarket)
plot(Volume)
```



4.7.2 ロジスティック回帰

次に、変数 `Lag1` から `Lag5` および `Volume` を使用して `Direction` を予測するために、ロジスティック回帰モデルを適合させてみよう。`glm()`関数は、ロジスティック回

帰を含む多くの種類の一般化線形モデルを適合するために利用できる。この関数の構文は `lm()`関数と似ているが、`family = binomial`引数を渡すことで、ロジスティック回帰を実行するように指示する。

```
glm.fits <- glm(
    Direction ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 + Volume,
    data = Smarket, family = binomial
  )
summary(glm.fits)
```

```
##
## Call:
## glm(formula = Direction ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 +
##     Volume, family = binomial, data = Smarket)
##
## Coefficients:
##              Estimate Std. Error z value Pr(>|z|)
## (Intercept) -0.126000   0.240736  -0.523    0.601
## Lag1        -0.073074   0.050167  -1.457    0.145
## Lag2        -0.042301   0.050086  -0.845    0.398
## Lag3         0.011085   0.049939   0.222    0.824
## Lag4         0.009359   0.049974   0.187    0.851
## Lag5         0.010313   0.049511   0.208    0.835
## Volume       0.135441   0.158360   0.855    0.392
##
## (Dispersion parameter for binomial family taken to be 1)
##
##     Null deviance: 1731.2  on 1249  degrees of freedom
## Residual deviance: 1727.6  on 1243  degrees of freedom
## AIC: 1741.6
##
## Number of Fisher Scoring iterations: 3
```

ここで最も小さいp値は変数`Lag1`に関連している。この予測子（説明変数）の負の係数は、市場が前日にプラスのリターンを示した場合、今日市場が上昇する可能性が低いことを示唆している。ただし、0.15 という値は依然として比較的大き

いため、`Lag1`と`Direction`との間に明確な関連性があるという証拠があるとは言えない。

`coef()`関数を使用すると、この適合モデルの係数のみを取得できる。また、`summary()`関数を使用して、係数のp値など、モデルの特定の側面にアクセスすることができる。

```
coef(glm.fits)
```

```
##  (Intercept)         Lag1         Lag2         Lag3         Lag4         Lag5
## -0.126000257 -0.073073746 -0.042301344  0.011085108  0.009358938  0.010313068
##       Volume
##  0.135440659
```

```
summary(glm.fits)$coef
```

```
##                 Estimate Std. Error    z value  Pr(>|z|)
## (Intercept) -0.126000257 0.24073574 -0.5233966 0.6006983
## Lag1        -0.073073746 0.05016739 -1.4565986 0.1452272
## Lag2        -0.042301344 0.05008605 -0.8445733 0.3983491
## Lag3         0.011085108 0.04993854  0.2219750 0.8243333
## Lag4         0.009358938 0.04997413  0.1872757 0.8514445
## Lag5         0.010313068 0.04951146  0.2082966 0.8349974
## Volume       0.135440659 0.15835970  0.8552723 0.3924004
```

```
summary(glm.fits)$coef[, 4]
```

```
## (Intercept)        Lag1        Lag2        Lag3        Lag4        Lag5
##   0.6006983   0.1452272   0.3983491   0.8243333   0.8514445   0.8349974
##      Volume
##   0.3924004
```

`predict()`関数は、説明変数(予測変数)の値が与えられた場合に市場が上昇する確率を予測するために利用できる。`type = "response"`オプションは、ロジットではなく、$P(Y = 1|X)$の形式で確率を出力するように`R`に指示する。`predict()`関数にデータセットが指定されていない場合、確率はロジスティック回帰モデルを適合させる際に使用されたトレーニングデータに対して計算される。ここでは、最初の 10 個の確率のみを出力している。これらの値は市場が上昇する確率に対応していることは、`contrasts()`関数が`R`によって`Up`を 1 とするダミー変数が作成されたことを示しているため分かる。

```
glm.probs <- predict(glm.fits, type = "response")
glm.probs[1:10]

##         1         2         3         4         5         6         7         8
## 0.5070841 0.4814679 0.4811388 0.5152224 0.5107812 0.5069565 0.4926509 0.5092292
##         9        10
## 0.5176135 0.4888378
contrasts(Direction)
##      Up
## Down  0
## Up    1
```

特定の日に市場が上昇するか下降するかを予測するには、予測された確率をクラスラベル `Up` または `Down` に変換する必要がある。以下の 2 つのコマンドは、市場が上昇する予測確率が 0.5 より大きいか小さいかに基づいてクラス予測のベクトルを作成する。

```
glm.pred <- rep("Down", 1250)
glm.pred[glm.probs > .5] = "Up"
```

最初のコマンドは、1,250 個の `Down` 要素のベクトルを作成する。2 行目は、市場が上昇する予測確率が 0.5 を超えるすべての要素を `Up` に変換する。これらの予測を元に、`table()`関数を使用して混同行列(confusion matrix)を作成し、正しくまたは誤って分類された観測値の数を確認できる。

```
table(glm.pred, Direction)
##         Direction
## glm.pred Down  Up
##     Down  145 141
##     Up    457 507
(507 + 145) / 1250
## [1] 0.5216
mean(glm.pred == Direction)
## [1] 0.5216
```

混同行列の対角要素は正しい予測を示し、非対角要素は誤った予測を表す。したがって、モデルは市場が 507 日上昇し、145 日下降することを正確に予測している。合計で$507 + 145 = 652$の正しい予測が得られた。`mean()`関数を使用して、予測が正確だった日の割合を計算できる。この場合、ロジスティック回帰は市場の動きを 52.2%の確率で正確に予測した。

一見すると、ロジスティック回帰モデルはランダムな推測よりもほんの少し優れているように見える。しかし、この結果は誤解を招きやすい。なぜなら、1,250 件の観測値からなる同じデータセットでモデルを訓練しテストしたためである。言い換えれば、$100\% - 52.2\% = 47.8\%$はトレーニング誤差率を表している。これまで見てきたように、トレーニング誤差率は過度に楽観的になる傾向があり、テスト誤差率を過小評価する傾向がある。この設定でロジスティック回帰モデルの精度をより正確に評価するためには、データの一部を使用してモデルを適合させ、その後、除外されたデータに対してどの程度良い予測をするかを調べる必要がある。

この戦略を実装するために、まず 2001 年から 2004 年までの観測値に対応するベクトルを作成する。次に、このベクトルを使用して 2005 年の観測値からなる除外データセットを作成する。

```
train <- (Year < 2005)
Smarket.2005 <- Smarket[!train, ]
dim(Smarket.2005)
## [1] 252   9
Direction.2005 <- Direction[!train]
```

オブジェクト `train` は、データセット内の観測値に対応する 1,250 個の要素を持つベクトルである。2005 年以前の観測値に対応するベクトルの要素は `TRUE` に設定され、2005 年の観測値に対応するものは `FALSE` に設定される。オブジェクト `train` はブールベクトルである。その要素は `TRUE` または `FALSE` です。ブールベクトルを使用して、行列の行または列の部分集合を取得できる。例えば、コマンド `Smarket[train, ]`は、株式市場データセットの日付が 2005 年以前のものに対応するサブマトリックスを抽出する。これは、`train` の要素が `TRUE` である観測値に対応している。`!`記号を使用して、ブールベクトルのすべての要素を逆にすることができる。すなわち、`!train` は `train` と似たベクトルであるが、`train` で `TRUE` である要素が`!train` では `FALSE` になり、`train` で `FALSE` である要素が`!train` では `TRUE` になる。そのため、`Smarket[!train, ]`は、2005 年の日付を持つ観測値を含む株式市場

データのサブマトリックスを生成している。上記の出力は、252 件の観測値があることを示している。

次に、2005 年以前の日付に対応する観測値のサブセットを使用して、`subset` 引数を使用してロジスティック回帰モデルを適合させてみる。そしてその後、テストセット内の各日に対する市場上昇の予測確率を取得しよう—つまり、2005 年の日に対してである。

```
glm.fits <- glm(
    Direction ~ Lag1 + Lag2 + Lag3 + Lag4 + Lag5 + Volume,
    data = Smarket, family = binomial, subset = train
  )
glm.probs <- predict(glm.fits, Smarket.2005,
    type = "response")
```

ここで、トレーニングとテストを完全に別々のデータセットで実行したことに注意しよう。トレーニングは 2005 年以前の日付のみを使用して実行し、テストは 2005 年の日付のみを使用して実行している。最後に、2005 年の予測を計算し、それらを市場の実際の動きと比較する。

```
glm.pred <- rep("Down", 252)
glm.pred[glm.probs > .5] <- "Up"
table(glm.pred, Direction.2005)
##         Direction.2005
## glm.pred Down Up
##     Down   77 97
##     Up     34 44
mean(glm.pred == Direction.2005)
## [1] 0.4801587
mean(glm.pred != Direction.2005)
## [1] 0.5198413
```

`!=` の表記は「等しくない」ことを意味し、最後のコマンドはテストセットの誤分類率を計算している。結果はかなり残念なもので、テスト誤分類率は 52% となり、ランダムに予測するよりも悪い結果となった。当然ながら、この結果はそれ

ほど驚くべきものではない。一般的に、過去数日のリターンを用いて将来の市場の動向を予測できるとは期待しにくいからである。（もしそれが可能であれば、本書の著者たちは統計学の教科書を書くのではなく、投資で大儲けしていることだろう。）

前回のロジスティック回帰モデルでは、すべての予測変数に対して非常に低調なp値が得られた。最も小さいp値でさえそれほど小さくはなく、それは`Lag1`に対応していた。もしかすると、`Direction`を予測する上で有効とは思えない変数を除外することで、より効果的なモデルが得られるかもしれない。そもそも、応答変数と関連性のない説明変数(予測変数)を使用すると、テスト誤分類率が悪化する傾向にある（このような変数を追加すると分散が増加する一方で、バイアスの減少には寄与しないため）。したがって、こうした変数を削除することで、モデルの改善につながる可能性がある。以下では、元のロジスティック回帰モデルにおいて最も高い予測力を持っていたと思われる`Lag1`と`Lag2`のみを用いて、ロジスティック回帰を再びフィットしてみる。

```
glm.fits <- glm(Direction ~ Lag1 + Lag2, data = Smarket,
    family = binomial, subset = train)
glm.probs <- predict(glm.fits, Smarket.2005,
    type = "response")
glm.pred <- rep("Down", 252)
glm.pred[glm.probs > .5] <- "Up"
table(glm.pred, Direction.2005)
##         Direction.2005
## glm.pred Down  Up
##     Down   35  35
##     Up     76 106
mean(glm.pred == Direction.2005)
## [1] 0.5595238
106 / (106 + 76)
## [1] 0.5824176
```

この結果は、少し改善されているように見える。日々の市場の動きの56%が正確に予測されている。この場合、市場が毎日上昇すると予測するという、はるかに単純な戦略でも56%の確率で正しいことに注意する価値がある！したがって、全体の誤差率に関しては、ロジスティック回帰法は単純なアプローチに比べて優れ

ているわけではない。しかし、混同行列によると、ロジスティック回帰が市場の上昇を予測した日には、58%の精度を持つことが分かる。これは、市場の上昇を予測した日に購入し、市場の下降を予測した日には取引を控えるという取引戦略の可能性を示唆している。もちろん、この小さな改善が本物か、それとも単なる偶然によるものかを慎重に調査する必要はある。

特定の `Lag1` と `Lag2` の値に関連するリターンを予測したいとしよう。特に、`Lag1` と `Lag2` がそれぞれ 1.2 と 1.1 である日、および 1.5 と-0.8 である日に対する `Direction` を予測したいとしよう。 この予測を行うには、`predict()`関数を使用するとよい。

```
predict(glm.fits,
    newdata =
      data.frame(Lag1 = c(1.2, 1.5),  Lag2 = c(1.1, -0.8)),
    type = "response"
  )
##         1         2
## 0.4791462 0.4960939
```

4.7.3 線形判別分析(LDA)

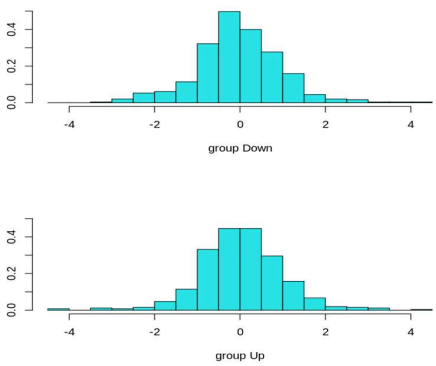
ここでは、`Smarket` データに対して LDA を実行しよう。`R` では、`MASS` ライブラリの `lda()`関数を使用して LDA モデルを適合できる。`lda()`関数の構文は `lm()`や `glm()`と同じであるが、`family` オプションがない点が異なっている。2005 年以前の観測値のみを使用してモデルをフィットしてみよう。

```
library(MASS)
##
## 次のパッケージを付け加えます: 'MASS'

## 以下のオブジェクトは 'package:ISLR2' からマスクされています:

##
##     Boston
```

```
lda.fit <- lda(Direction ~ Lag1 + Lag2, data = Smarket,
    subset = train)
lda.fit
```

```
## Call:
## lda(Direction ~ Lag1 + Lag2, data = Smarket, subset = train)
##
## Prior probabilities of groups:
##     Down       Up
## 0.491984 0.508016
##
## Group means:
##             Lag1        Lag2
## Down  0.04279022  0.03389409
## Up   -0.03954635 -0.03132544
##
## Coefficients of linear discriminants:
##             LD1
## Lag1 -0.6420190
## Lag2 -0.5135293
```

```
plot(lda.fit)
```



LDA の出力によれば、$\hat{\pi}_1 = 0.492, \hat{\pi}_2 = 0.508$となる。つまり、トレーニング観測値の 49.2%が市場が下降した日に対応している。また、各クラス内での各予測変数の平均（グループ平均）が提供されている。これらは、LDA によってμ_kの推定値として使用される。これにより、市場が上昇する日には直近 2 日間のリターン

が負である傾向があり、市場が下降する日には直近のリターンが正である傾向が示唆される。

線形判別係数 (coefficients of linear discriminants) の出力は、LDA の判別規則を形成するために `Lag1` と `Lag2` の線形結合を提供する。つまり、これは(4.24)式内で $X = x$の要素に掛けられる係数である。 例えば、$-0.642 \times$ `Lag1` -0.514 `Lag2` が大きければ、LDA 分類器は市場が上昇すると予測し、小さければ市場が下降すると予測する。

`plot()`関数は、トレーニング観測値に対して$-0.642 \times$ `Lag1` -0.514 `Lag2` を計算することで得られる線形判別をプロットする。`Up` と `Down` の観測値は個別に表示される。

`predict()`関数は、3 つの要素を含むリストを返す。最初の要素 `class` には、LDA による市場動向の予測が含まれる。2 番目の要素 `posterior` は行列で、そのk列目には、対応する観測値がクラスkに属する事後確率が(4.15 式)から計算されて格納されている。最後の要素 `x` には、前述した線形判別が含まれている。

```
lda.pred <- predict(lda.fit, Smarket.2005)
names(lda.pred)
## [1] "class"     "posterior" "x"
```

4.5 節で見たように LDA とロジスティック回帰の予測はほとんど同一になる。

```
lda.class <- lda.pred$class
table(lda.class, Direction.2005)
##          Direction.2005
## lda.class Down  Up
##      Down   35  35
##      Up     76 106
mean(lda.class == Direction.2005)
## [1] 0.5595238
```

事後確率として 50%の閾値を適用することで、`lda.pred$class` に含まれる予測を再現できる。

```
sum(lda.pred$posterior[, 1] >= .5)
```

```
## [1] 70
sum(lda.pred$posterior[, 1] < .5)
## [1] 182
```

モデルによって出力された事後確率は、市場が *下降*する確率に対応していることに注意しておこう。

```
lda.pred$posterior[1:20, 1]
##       999      1000      1001      1002      1003      1004      1005      1
006
## 0.4901792 0.4792185 0.4668185 0.4740011 0.4927877 0.4938562 0.4951016 0.4872
861
##      1007      1008      1009      1010      1011      1012      1013      1
014
## 0.4907013 0.4844026 0.4906963 0.5119988 0.4895152 0.4706761 0.4744593 0.4799
583
##      1015      1016      1017      1018
## 0.4935775 0.5030894 0.4978806 0.4886331
lda.class[1:20]
##  [1] Up   Up   Up   Up   Up   Up   Up   Up   Up   Up   Up   Down Up   Up   U
p
## [16] Up   Up   Down Up   Up
## Levels: Down Up
```

50%以外の事後確率の閾値を使用して予測を行いたい場合は、簡単に変更できる。例えば、市場がその日に確実に下降すると非常に確信できる場合—事後確率が少なくとも 90%である場合にのみ下降を予測するとする。

```
sum(lda.pred$posterior[, 1] > .9)
## [1] 0
```

2005 年には、その閾値を満たす日はなかった！実際、2005 年全体で市場が下降する最大の事後確率は 52.02%であった。

4.7.4 二次判別分析 (QDA)

次に、Smarketデータに対して QDA モデルを適合させよう。QDA は、同じく MASS ライブラリの qda()関数を使用することで R で実装できる。この構文は lda()と同一となる。

```
qda.fit <- qda(Direction ~ Lag1 + Lag2, data = Smarket,
    subset = train)
qda.fit

## Call:
## qda(Direction ~ Lag1 + Lag2, data = Smarket, subset = train)
##
## Prior probabilities of groups:
##     Down       Up
## 0.491984 0.508016
##
## Group means:
##             Lag1        Lag2
## Down  0.04279022  0.03389409
## Up   -0.03954635 -0.03132544
```

出力にはグループ平均が含まれるが、QDA 分類器が予測変数の 2 次関数を使用するため、線形判別の係数は含まれない。predict()関数の使い方は LDA と全く同じである。

```
qda.class <- predict(qda.fit, Smarket.2005)$class
table(qda.class, Direction.2005)
##          Direction.2005
## qda.class Down  Up
##      Down   30  20
##      Up     81 121
mean(qda.class == Direction.2005)
## [1] 0.5992063
```

興味深いことに、QDA の予測は 60%近い精度で正確である。しかも 2005 年のデータはモデルの適合には使用されていない。この精度は、株式市場データのモデル化が非常に難しいことで知られていることを考えると非常に印象的である。この結果は、QDA によって仮定される二次形式が、LDA やロジスティック回帰によって仮定される線形形式よりも真の関係をより正確に捉える可能性があることを示唆していると言える。しかし、このアプローチが一貫して市場を上回ると確信する前に、より大きなテストセットでこの方法の性能を評価することをお勧めしておこう！

4.7.5 ナイーブベイズ

次に、`Smarket` データに対してナイーブベイズモデルを適合させよう。ナイーブベイズは、`e1071` ライブラリの一部である `naiveBayes()` 関数を使用して `R` で実装されている。この構文は、`lda()` や `qda()` の構文と同じである。デフォルトでは、このナイーブベイズ分類器の実装は、各定量的特徴がガウス分布にしたがうとモデル化される。ただし、カーネル密度法を使用して分布を推定することもできる。

```
library(e1071)
nb.fit <- naiveBayes(Direction ~ Lag1 + Lag2, data = Smarket,
    subset = train)
nb.fit
```

```
##
## Naive Bayes Classifier for Discrete Predictors
##
## Call:
## naiveBayes.default(x = X, y = Y, laplace = laplace)
##
## A-priori probabilities:
## Y
##     Down       Up
## 0.491984 0.508016
##
## Conditional probabilities:
##       Lag1
## Y            [,1]     [,2]
##   Down  0.04279022 1.227446
```

```
##   Up   -0.03954635 1.231668
##
##       Lag2
## Y             [,1]     [,2]
##   Down  0.03389409 1.239191
##   Up   -0.03132544 1.220765
```

出力には、各クラスにおける各変数の推定平均と標準偏差が含まれている。たとえば、`Lag1` の平均は `Direction=Down` の場合 0.0428 で、標準偏差は 1.23 となる。この値は簡単に確認できる：

```
mean(Lag1[train][Direction[train] == "Down"])
## [1] 0.04279022
sd(Lag1[train][Direction[train] == "Down"])
## [1] 1.227446
```

`predict()` 関数はシンプルである。

```
nb.class <- predict(nb.fit, Smarket.2005)
table(nb.class, Direction.2005)
##         Direction.2005
## nb.class Down  Up
##     Down   28  20
##     Up     83 121
mean(nb.class == Direction.2005)
## [1] 0.5912698
```

ナイーブベイズはこのデータで非常に良い性能を発揮し、正確な予測を行う割合は 59% を超えている。この数値は QDA よりわずかに劣るが、LDA よりはかなり良い結果となっている。

`predict()` 関数は、各観測が特定のクラスに属する確率の推定値を生成することもできる。

```
nb.preds <- predict(nb.fit, Smarket.2005, type = "raw")
nb.preds[1:5, ]
```

```
##            Down        Up
## [1,] 0.4873164 0.5126836
## [2,] 0.4762492 0.5237508
## [3,] 0.4653377 0.5346623
## [4,] 0.4748652 0.5251348
## [5,] 0.4901890 0.5098110
```

4.7.6 K最近傍法(KNN)

ここでは、`class`ライブラリの一部である`knn()`関数を使用して KNN を実行する。この関数は、これまでに使用した他のモデルをフィットする関数とは異なり、モデルの適合と予測の 2 段階のアプローチではなく、単一のコマンドで予測を生成する。`knn()`関数には以下の 4 つの入力が必要となる。

- 訓練データに関連付けられた説明変数(予測変数)を含む行列（以下では`train.X`）。
- 予測を行いたいデータに関連付けられた予測変数を含む行列（以下では`test.X`）。
- 訓練観測値のクラスラベルを含むベクトル（以下では`train.Direction`）。
- Kの値。これは分類器が使用する最近傍点の数。

`cbind()`関数（*column bind* の略）を使用して、`Lag1`と`Lag2`変数を結合し、訓練データセット用とテストデータセット用の 2 つの行列を作成する。

```
library(class)
train.X <- cbind(Lag1, Lag2)[train, ]
test.X <- cbind(Lag1, Lag2)[!train, ]
train.Direction <- Direction[train]
```

次に、`knn()`関数を使用して 2005 年の日付に対する市場の動きを予測しよう。`knn()`を適用する前にランダムシードを設定しておく。これは、複数の観測が最近傍として同点になる場合、`R`がランダムに選択するためである。結果の再現性を確保するためにはシードを設定する必要がある。

```
set.seed(1)
knn.pred <- knn(train.X, test.X, train.Direction, k = 1)
table(knn.pred, Direction.2005)
##         Direction.2005
## knn.pred Down Up
##     Down   43 58
##     Up     68 83
(83 + 43) / 252
## [1] 0.5
```

$K = 1$を使用した結果はあまり良くなく、観測の 50%しか正しく予測されない。$K = 1$ではデータに対して柔軟すぎる適合がなされる可能性があるからだろう。以下では、$K = 3$を使用して分析を繰り返してみる。

```
knn.pred <- knn(train.X, test.X, train.Direction, k = 3)
table(knn.pred, Direction.2005)
##         Direction.2005
## knn.pred Down Up
##     Down   48 54
##     Up     63 87
mean(knn.pred == Direction.2005)
## [1] 0.5357143
```

結果はやや改善した。しかし、Kをさらに増やしても改善は見られない。このデータでは、これまで検討した方法の中で QDA が最も良い結果を提供するようである。

KNN は `Smarket` データでは良い結果を出せないが、他のケースでは驚くべき結果を提供することもある。例として、`ISLR2` ライブラリの一部である `Insurance` データセットに KNN アプローチを適用してみよう。このデータセットには、5,822 人の個人の人口統計的特徴を測定する 85 個の予測変数が含まれている。応答変数 `Purchase` は、特定の個人がキャラバン保険を購入するかどうかを示している。このデータセットでは、保険を購入した人はわずか 6% となっている。

```
dim(Caravan)
```

```
## [1] 5822   86
attach(Caravan)
summary(Purchase)
##   No  Yes
## 5474  348
348 / 5822
## [1] 0.05977327
```

KNN 分類器は、観測間の距離を基にクラスを予測するため、変数のスケールが重要である。大きなスケールの変数は、観測間の距離および KNN 分類器に大きな影響を与える。たとえば、`salary`（給与）と `age`（年齢）が含まれるデータセットでは、給与の差が 1,000 ドルであっても、年齢の差が 50 歳の場合、`salary` が KNN 分類結果を支配し、`age` はほとんど影響を与えない。 この問題に対処するための良い方法は、データを標準化して、すべての変数の平均を 0、標準偏差を 1 にすることである。これにより、すべての変数が比較可能なスケールになります。
`scale()`関数はこの目的で使用されている

この問題を処理する方法の一つは、データを*標準化*し、すべての変数の平均を 0、標準偏差を 1 にすること。こうすることで、すべての変数が比較可能なスケールになる。`scale()` 関数はこの処理を行う。

データを標準化する際、第 86 列は除外する。この列は質的変数 `Purchase` だからである。

```
standardized.X <- scale(Caravan[, -86])
var(Caravan[, 1])
## [1] 165.0378
var(Caravan[, 2])
## [1] 0.1647078
var(standardized.X[, 1])
## [1] 1
var(standardized.X[, 2])
## [1] 1
```

これで、`standardized.X` の各列は標準偏差が 1、平均が 0 になった。

観測値を最初の 1,000 個の観測からなるテストセットと、残りの観測からなる訓練セットに分割する。$K = 1$を使用して訓練データに KNN モデルを適合させ、テストデータでその性能を評価しよう。

```
test <- 1:1000
train.X <- standardized.X[-test, ]
test.X <- standardized.X[test, ]
train.Y <- Purchase[-test]
test.Y <- Purchase[test]
set.seed(1)
knn.pred <- knn(train.X, test.X, train.Y, k = 1)
mean(test.Y != knn.pred)
## [1] 0.118
mean(test.Y != "No")
## [1] 0.059
```

ベクトル `test` は数値型で、値は 1 から 1,000 までの範囲を持つ。`standardized.X[test, ]` を入力すると、インデックスが 1 から 1,000 の範囲にある観測値を含むデータの部分行列が得られる。一方で、`standardized.X[-test, ]` を入力すると、インデックスが 1 から 1,000 の範囲に 含まれない 観測値を持つ部分行列が得られる。

1,000 個のテスト観測値に対する KNN の誤分類率は 12% 未満である。一見すると、これは比較的良い結果のように思える。しかし、実際には保険を購入した顧客は全体の 6% にすぎないため、常に `No` と予測するだけで誤分類率を 6% まで下げることができる。

仮に、特定の個人に対して保険の販売を試みることに、無視できないコストがかかるとする。例えば、販売員が各潜在顧客を訪問しなければならない場合を考える。会社が無作為に顧客を選び、保険の販売を試みた場合、成功率はわずか 6% であり、この数値はコストを考慮すると低すぎる可能性がある。そのため、会社としては、保険を購入する可能性が高い顧客に対してのみ販売を試みたいと考える。この場合、全体の誤分類率は重要ではなく、実際に保険を購入すると正しく予測された顧客の割合が重要となる。

KNN（$K = 1$）を使用すると、保険を購入すると予測された顧客の中では、無作為な予測よりもはるかに良い結果が得られる。具体的には、保険を購入すると予測

された 77 人のうち、9 人（11.7%）が実際に保険を購入した。この数値は、無作為な予測で得られる購入率の 2 倍に相当する。

```
table(knn.pred, test.Y)
##         test.Y
## knn.pred  No Yes
##      No  873  50
##      Yes  68   9
9 / (68 + 9)
## [1] 0.1168831
```

$K = 3$では成功率は 19%に増加し、$K = 5$では 26.7%に達する。ランダム予測よりも 4 倍以上良い結果となった。

```
knn.pred <- knn(train.X, test.X, train.Y, k = 3)
table(knn.pred, test.Y)
##         test.Y
## knn.pred  No Yes
##      No  920  54
##      Yes  21   5
5 / 26
## [1] 0.1923077
knn.pred <- knn(train.X, test.X, train.Y, k = 5)
table(knn.pred, test.Y)
##         test.Y
## knn.pred  No Yes
##      No  930  55
##      Yes  11   4
4 / 15
## [1] 0.2666667
```

ただし、KNN で$K = 5$を使用すると、保険を購入すると予測された顧客はわずか 15 人となる。実際には、保険会社はさらに多くの潜在顧客を説得するためにリソースを投入する必要があるだろう。

比較のため、ロジスティック回帰モデルをデータにフィットすることもできる。0.5 を予測確率のカットオフとして使用すると、テスト観測のうち 7 件が保険を購入すると予測されますが、すべて間違っている。しかし、カットオフを 0.25 に変更すると、33 人が保険を購入すると予測され、そのうち約 33%が正解となる。この数値はランダム予測の 5 倍以上良い結果である。

```
glm.fits <- glm(Purchase ~ ., data = Caravan,
    family = binomial, subset = -test)
## Warning: glm.fit: 数値的に 0 か 1 である確率が生じました
glm.probs <- predict(glm.fits, Caravan[test, ],
    type = "response")
glm.pred <- rep("No", 1000)
glm.pred[glm.probs > .5] <- "Yes"
table(glm.pred, test.Y)
##         test.Y
## glm.pred  No Yes
##      No  934  59
##      Yes   7   0
glm.pred <- rep("No", 1000)
glm.pred[glm.probs > .25] <- "Yes"
table(glm.pred, test.Y)
##         test.Y
## glm.pred  No Yes
##      No  919  48
##      Yes  22  11
11 / (22 + 11)
## [1] 0.3333333
```

4.7.7 ポアソン回帰

最後に、`Bikeshare`データセットにポアソン回帰モデルをフィットしよう。このデータセットには、ワシントン DC での 1 時間あたりの自転車レンタル数（`bikers`）が記録されているが、このデータは`ISLR2`ライブラリに含まれている。

```
attach(Bikeshare)
dim(Bikeshare)
## [1] 8645   15
names(Bikeshare)
##  [1] "season"     "mnth"       "day"        "hr"         "holiday"
##  [6] "weekday"    "workingday" "weathersit" "temp"       "atemp"
## [11] "hum"        "windspeed"  "casual"     "registered" "bikers"
```

最初に、最小二乗法による線形回帰モデルをデータにフィットしてみよう。

```
mod.lm <- lm(
    bikers ~ mnth + hr + workingday + temp + weathersit,
    data = Bikeshare
  )
summary(mod.lm)

##
## Call:
## lm(formula = bikers ~ mnth + hr + workingday + temp + weathersit,
##     data = Bikeshare)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -299.00  -45.70   -6.23   41.08  425.29
##
## Coefficients:
##                           Estimate Std. Error t value Pr(>|t|)
## (Intercept)               -68.632      5.307 -12.932  < 2e-16 ***
## mnthFeb                     6.845      4.287   1.597 0.110398
## mnthMarch                  16.551      4.301   3.848 0.000120 ***
## mnthApril                  41.425      4.972   8.331  < 2e-16 ***
## mnthMay                    72.557      5.641  12.862  < 2e-16 ***
## mnthJune                   67.819      6.544  10.364  < 2e-16 ***
## mnthJuly                   45.324      7.081   6.401 1.63e-10 ***
## mnthAug                    53.243      6.640   8.019 1.21e-15 ***
```

```
## mnthSept                     66.678      5.925  11.254  < 2e-16 ***
## mnthOct                      75.834      4.950  15.319  < 2e-16 ***
## mnthNov                      60.310      4.610  13.083  < 2e-16 ***
## mnthDec                      46.458      4.271  10.878  < 2e-16 ***
## hr1                         -14.579      5.699  -2.558 0.010536 *
## hr2                         -21.579      5.733  -3.764 0.000168 ***
## hr3                         -31.141      5.778  -5.389 7.26e-08 ***
## hr4                         -36.908      5.802  -6.361 2.11e-10 ***
## hr5                         -24.135      5.737  -4.207 2.61e-05 ***
## hr6                          20.600      5.704   3.612 0.000306 ***
## hr7                         120.093      5.693  21.095  < 2e-16 ***
## hr8                         223.662      5.690  39.310  < 2e-16 ***
## hr9                         120.582      5.693  21.182  < 2e-16 ***
## hr10                         83.801      5.705  14.689  < 2e-16 ***
## hr11                        105.423      5.722  18.424  < 2e-16 ***
## hr12                        137.284      5.740  23.916  < 2e-16 ***
## hr13                        136.036      5.760  23.617  < 2e-16 ***
## hr14                        126.636      5.776  21.923  < 2e-16 ***
## hr15                        132.087      5.780  22.852  < 2e-16 ***
## hr16                        178.521      5.772  30.927  < 2e-16 ***
## hr17                        296.267      5.749  51.537  < 2e-16 ***
## hr18                        269.441      5.736  46.976  < 2e-16 ***
## hr19                        186.256      5.714  32.596  < 2e-16 ***
## hr20                        125.549      5.704  22.012  < 2e-16 ***
## hr21                         87.554      5.693  15.378  < 2e-16 ***
## hr22                         59.123      5.689  10.392  < 2e-16 ***
## hr23                         26.838      5.688   4.719 2.41e-06 ***
## workingday                    1.270      1.784   0.711 0.476810
## temp                        157.209     10.261  15.321  < 2e-16 ***
## weathersitcloudy/misty      -12.890      1.964  -6.562 5.60e-11 ***
## weathersitlight rain/snow   -66.494      2.965 -22.425  < 2e-16 ***
## weathersitheavy rain/snow -109.745     76.667  -1.431 0.152341
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 76.5 on 8605 degrees of freedom
## Multiple R-squared:  0.6745, Adjusted R-squared:  0.6731
## F-statistic: 457.3 on 39 and 8605 DF,  p-value: < 2.2e-16
```

このモデル(mod.lm)では、hr（時間）の第 1 レベル（0 時）と mnth（月）の第 1 レベル（1 月）が基準値として扱われるため、それらの係数推定値は提供されない。暗黙的にこれらの係数推定値は 0 であり、他のレベルはこれらの基準値との相対値で測定される。例えば、2 月の係数 6.845 は、他の変数が一定である場合、2 月には 1 月より平均して約 7 人多くのライダーがいることを示している。同様に、3 月には 1 月より約 16.5 人多くのライダーがいる。

4.6.1 節で利用した結果は、以下のように hr と mnth の変数の異なるコーディングを使用して得られた。

```
contrasts(Bikeshare$hr) = contr.sum(24)
contrasts(Bikeshare$mnth) = contr.sum(12)
mod.lm2 <- lm(
    bikers ~ mnth + hr + workingday + temp + weathersit,
    data = Bikeshare
  )
summary(mod.lm2)
```

```
##
## Call:
## lm(formula = bikers ~ mnth + hr + workingday + temp + weathersit,
##     data = Bikeshare)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
## -299.00  -45.70   -6.23   41.08  425.29
##
## Coefficients:
##                             Estimate Std. Error t value Pr(>|t|)
## (Intercept)                  73.5974     5.1322  14.340  < 2e-16 ***
## mnth1                       -46.0871     4.0855 -11.281  < 2e-16 ***
## mnth2                       -39.2419     3.5391 -11.088  < 2e-16 ***
## mnth3                       -29.5357     3.1552  -9.361  < 2e-16 ***
## mnth4                        -4.6622     2.7406  -1.701  0.08895 .
## mnth5                        26.4700     2.8508   9.285  < 2e-16 ***
## mnth6                        21.7317     3.4651   6.272 3.75e-10 ***
## mnth7                        -0.7626     3.9084  -0.195  0.84530
```

```
## mnth8                          7.1560     3.5347   2.024  0.04295 *
## mnth9                         20.5912     3.0456   6.761 1.46e-11 ***
## mnth10                        29.7472     2.6995  11.019  < 2e-16 ***
## mnth11                        14.2229     2.8604   4.972 6.74e-07 ***
## hr1                          -96.1420     3.9554 -24.307  < 2e-16 ***
## hr2                         -110.7213     3.9662 -27.916  < 2e-16 ***
## hr3                         -117.7212     4.0165 -29.310  < 2e-16 ***
## hr4                         -127.2828     4.0808 -31.191  < 2e-16 ***
## hr5                         -133.0495     4.1168 -32.319  < 2e-16 ***
## hr6                         -120.2775     4.0370 -29.794  < 2e-16 ***
## hr7                          -75.5424     3.9916 -18.925  < 2e-16 ***
## hr8                           23.9511     3.9686   6.035 1.65e-09 ***
## hr9                          127.5199     3.9500  32.284  < 2e-16 ***
## hr10                          24.4399     3.9360   6.209 5.57e-10 ***
## hr11                         -12.3407     3.9361  -3.135  0.00172 **
## hr12                          9.2814     3.9447   2.353  0.01865 *
## hr13                         41.1417     3.9571  10.397  < 2e-16 ***
## hr14                         39.8939     3.9750  10.036  < 2e-16 ***
## hr15                         30.4940     3.9910   7.641 2.39e-14 ***
## hr16                         35.9445     3.9949   8.998  < 2e-16 ***
## hr17                         82.3786     3.9883  20.655  < 2e-16 ***
## hr18                        200.1249     3.9638  50.488  < 2e-16 ***
## hr19                        173.2989     3.9561  43.806  < 2e-16 ***
## hr20                         90.1138     3.9400  22.872  < 2e-16 ***
## hr21                         29.4071     3.9362   7.471 8.74e-14 ***
## hr22                         -8.5883     3.9332  -2.184  0.02902 *
## hr23                        -37.0194     3.9344  -9.409  < 2e-16 ***
## workingday                    1.2696     1.7845   0.711  0.47681
## temp                        157.2094    10.2612  15.321  < 2e-16 ***
## weathersitcloudy/misty      -12.8903     1.9643  -6.562 5.60e-11 ***
## weathersitlight rain/snow   -66.4944     2.9652 -22.425  < 2e-16 ***
## weathersitheavy rain/snow -109.7446    76.6674  -1.431  0.15234
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 76.5 on 8605 degrees of freedom
## Multiple R-squared:  0.6745, Adjusted R-squared:  0.6731
## F-statistic: 457.3 on 39 and 8605 DF,  p-value: < 2.2e-16
```

2 つのコーディングの違いは、`mod.lm2` では `hr` と `mnth` の最後のレベルを除くすべてのレベルに係数推定値が報告されること。重要なのは、`mod.lm2` では `mnth` の最後のレベルの係数推定値が 0 ではなく、他のすべてのレベルの係数推定値の合計の負の値 になっている点。同様に、`hr` の最後のレベルの係数推定値も、他のすべてのレベルの係数推定値の合計の負の値になる。つまり、`mod.lm2` における `hr` と `mnth` の係数は常に 0 になるように調整されており、これらの係数は平均レベルからの偏差として解釈できる。例えば、1 月の係数が -46.087 である場合、他の変数が一定なら、1 月のライダー数は年間平均よりも 46 人少ないことを意味する。

コーディングの選択自体は重要ではなく、どのコーディングを使用しているかを理解し、それに基づいてモデルの出力を正しく解釈すれば問題ない。たとえば、コーディングに関係なく、線形モデルの予測値は同じであることが分かる:

```
sum((predict(mod.lm) - predict(mod.lm2))^2)
## [1] 1.586608e-18
```

差の二乗の合計はゼロです。`all.equal()`関数を使用して確認することもできる。

```
all.equal(predict(mod.lm), predict(mod.lm2))
## [1] TRUE
```

図 4.13 の左側を再現するために、`mnth` に関連する係数推定値を取得しよう。1 月から 11 月までの係数は `mod.lm2` オブジェクトから直接取得できるが、12 月の係数は他の月の合計の負として明示的に計算する必要がある。

```
coef.months <- c(coef(mod.lm2)[2:12],
    -sum(coef(mod.lm2)[2:12]))
```

プロットを作成する際に、x軸に月の名前を手動でラベル付けする。

```
plot(coef.months, xlab = "Month", ylab = "Coefficient",
    xaxt = "n", col = "blue", pch = 19, type = "o")
axis(side = 1, at = 1:12, labels = c("J", "F", "M", "A",
    "M", "J", "J", "A", "S", "O", "N", "D"))
```

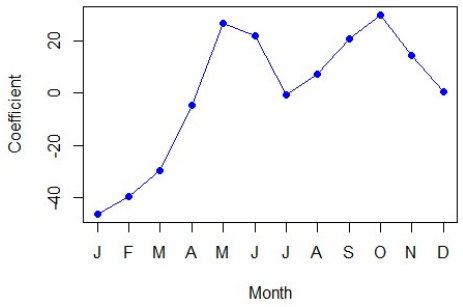
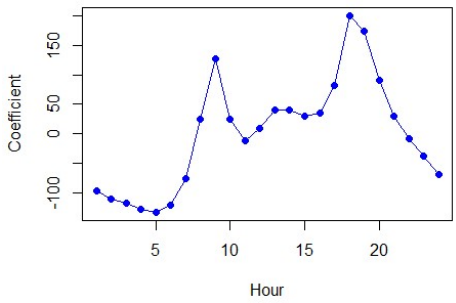


図 4.13 の右側を再現するプロセスも同様である。

```
coef.hours <- c(coef(mod.lm2)[13:35],
    -sum(coef(mod.lm2)[13:35]))
plot(coef.hours, xlab = "Hour", ylab = "Coefficient",
    col = "blue", pch = 19, type = "o")
```



次に、`Bikeshare`データにポアソン回帰モデルを適合させよう。変更点はほとんどなく、`glm()`関数の引数に`family = poisson`を指定して、ポアソン回帰モデルを適合させることだけとなる。

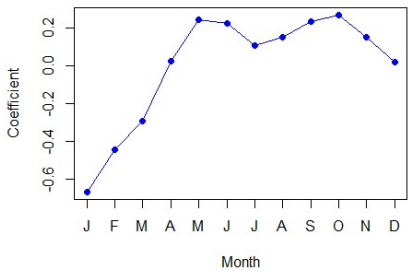
```
mod.pois <- glm(
    bikers ~ mnth + hr + workingday + temp + weathersit,
    data = Bikeshare, family = poisson
  )
summary(mod.pois)


##
## Call:
```

```
## glm(formula = bikers ~ mnth + hr + workingday + temp + weathersit,
##     family = poisson, data = Bikeshare)
##
## Coefficients:
##                             Estimate Std. Error  z value Pr(>|z|)
## (Intercept)                 4.118245   0.006021  683.964  < 2e-16 ***
## mnth1                      -0.670170   0.005907 -113.445  < 2e-16 ***
## mnth2                      -0.444124   0.004860  -91.379  < 2e-16 ***
## mnth3                      -0.293733   0.004144  -70.886  < 2e-16 ***
## mnth4                       0.021523   0.003125    6.888 5.66e-12 ***
## mnth5                       0.240471   0.002916   82.462  < 2e-16 ***
## mnth6                       0.223235   0.003554   62.818  < 2e-16 ***
## mnth7                       0.103617   0.004125   25.121  < 2e-16 ***
## mnth8                       0.151171   0.003662   41.281  < 2e-16 ***
## mnth9                       0.233493   0.003102   75.281  < 2e-16 ***
## mnth10                      0.267573   0.002785   96.091  < 2e-16 ***
## mnth11                      0.150264   0.003180   47.248  < 2e-16 ***
## hr1                        -0.754386   0.007879  -95.744  < 2e-16 ***
## hr2                        -1.225979   0.009953 -123.173  < 2e-16 ***
## hr3                        -1.563147   0.011869 -131.702  < 2e-16 ***
## hr4                        -2.198304   0.016424 -133.846  < 2e-16 ***
## hr5                        -2.830484   0.022538 -125.586  < 2e-16 ***
## hr6                        -1.814657   0.013464 -134.775  < 2e-16 ***
## hr7                        -0.429888   0.006896  -62.341  < 2e-16 ***
## hr8                         0.575181   0.004406  130.544  < 2e-16 ***
## hr9                         1.076927   0.003563  302.220  < 2e-16 ***
## hr10                        0.581769   0.004286  135.727  < 2e-16 ***
## hr11                        0.336852   0.004720   71.372  < 2e-16 ***
## hr12                        0.494121   0.004392  112.494  < 2e-16 ***
## hr13                        0.679642   0.004069  167.040  < 2e-16 ***
## hr14                        0.673565   0.004089  164.722  < 2e-16 ***
## hr15                        0.624910   0.004178  149.570  < 2e-16 ***
## hr16                        0.653763   0.004132  158.205  < 2e-16 ***
## hr17                        0.874301   0.003784  231.040  < 2e-16 ***
## hr18                        1.294635   0.003254  397.848  < 2e-16 ***
## hr19                        1.212281   0.003321  365.084  < 2e-16 ***
## hr20                        0.914022   0.003700  247.065  < 2e-16 ***
## hr21                        0.616201   0.004191  147.045  < 2e-16 ***
```

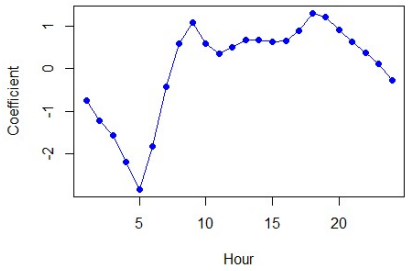
```
## hr22                          0.364181   0.004659   78.173  < 2e-16 ***
## hr23                          0.117493   0.005225   22.488  < 2e-16 ***
## workingday                    0.014665   0.001955    7.502 6.27e-14 ***
## temp                          0.785292   0.011475   68.434  < 2e-16 ***
## weathersitcloudy/misty       -0.075231   0.002179  -34.528  < 2e-16 ***
## weathersitlight rain/snow -0.575800   0.004058 -141.905  < 2e-16 ***
## weathersitheavy rain/snow -0.926287   0.166782   -5.554 2.79e-08 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
## (Dispersion parameter for poisson family taken to be 1)
##     Null deviance: 1052921  on 8644  degrees of freedom
## Residual deviance:  228041  on 8605  degrees of freedom
## AIC: 281159
## Number of Fisher Scoring iterations: 5
```

係数をプロットして、図 4.15 を再現しよう。

```
coef.mnth <- c(coef(mod.pois)[2:12],
    -sum(coef(mod.pois)[2:12]))
plot(coef.mnth, xlab = "Month", ylab = "Coefficient",
     xaxt = "n", col = "blue", pch = 19, type = "o")
axis(side = 1, at = 1:12, labels = c("J", "F", "M", "A", "M", "J", "J", "A", "S
", "O", "N", "D"))
```

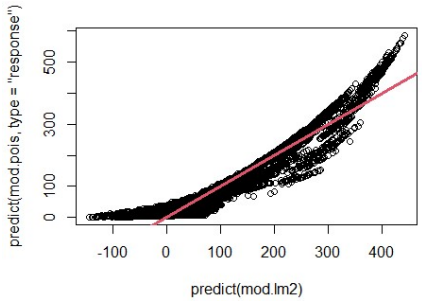



```
coef.hours <- c(coef(mod.pois)[13:35],
     -sum(coef(mod.pois)[13:35]))
plot(coef.hours, xlab = "Hour", ylab = "Coefficient",
    col = "blue", pch = 19, type = "o")
```



ポアソン回帰モデルの適合値（予測値）を取得するには、`predict()`関数を使用する。この際、引数`type = "response"`を指定して、`R`がデフォルトで出力する$\hat{\beta}_0 + \hat{\beta}_1 X_1 + \cdots + \hat{\beta}_p X_p$ではなく、$\exp\left(\hat{\beta}_0 + \hat{\beta}_1 X_1 + \cdots + \hat{\beta}_p X_p\right)$を出力するようにする。

```
plot(predict(mod.lm2), predict(mod.pois, type = "response"))
abline(0, 1, col = 2, lwd = 3)
```



ポアソン回帰モデルの予測は、線形モデルの予測と相関している。ただし、ポアソン回帰の予測値は非負であるため、ライダー数が非常に少ないまたは多い場合、ポアソン回帰の予測値は線形モデルの予測値よりも大きくなる傾向がある。

このセクションでは、`glm()`関数と`family = poisson`を使用してポアソン回帰を実行した。同様に、`family`引数の異なる選択肢を使用して他の種類の一般化線形モデル（GLM）を適合させることもできる。例えば、`family = Gamma`を使用するとガンマ回帰モデルをフィットすることができる。

ISLR 第 5 章 実習：リサンプリング法

このラボでは、本章で扱ったリサンプリング手法を検討する。このラボで使用するいくつかのコマンドは、実行に時間がかかる場合がある。

5.3.1 検証セット・アプローチ

`Auto` データセットに対してさまざまな線形モデルをフィッティングした際に生じるテスト誤差率を推定するために、検証セット・アプローチの使用を検討する。

始める前に、R の乱数生成器の *シード* を `set.seed()` 関数で設定する。これにより、この本の読者が以下に示される結果とまったく同じものを得られるようになる。クロスバリデーションのようなランダム性を含む分析を行う際には、シードを設定するのが一般的に良い習慣である。これにより、後で同じ結果を正確に再現できるようになる。

まず、`sample()` 関数を使用して観測データを 2 つのグループに分割する。元の 392 個の観測値のうち、ランダムに 196 個の観測値を選択し、それをトレーニングセットとする。

```
library(ISLR2)
set.seed(1)
train <- sample(392, 196)
```

（ここでは `sample` コマンドのショートカットを使用している。詳細は `?sample` を参照。）次に、`lm()` の `subset` オプションを使用し、トレーニングセットに対応する観測値のみを用いて線形回帰モデルをフィッティングする。

```
lm.fit <- lm(mpg ~ horsepower, data = Auto, subset = train)
```

次に、`predict()` 関数を使用して、全 392 個の観測値に対する応答を推定する。さらに、`mean()` 関数を用いて、検証セットに含まれる 196 個の観測値の MSE（平均二乗誤差）を計算する。なお、以下の `-train` インデックスは、トレーニングセットに含まれない観測値のみを選択することを意味する。

```
attach(Auto)
mean((mpg - predict(lm.fit, Auto))[-train]^2)
## [1] 23.26601
```

したがって、線形回帰モデルの推定テスト MSE は 23.27 となる。`poly()` 関数を使用することで、二次回帰および三次回帰のテスト誤差を推定できる。

```
lm.fit2 <- lm(mpg ~ poly(horsepower, 2), data = Auto,
    subset = train)
mean((mpg - predict(lm.fit2, Auto))[-train]^2)
## [1] 18.71646
lm.fit3 <- lm(mpg ~ poly(horsepower, 3), data = Auto,
    subset = train)
mean((mpg - predict(lm.fit3, Auto))[-train]^2)
## [1] 18.79401
```

これらの誤差率は、それぞれ 18.72 と 18.79 である。異なるトレーニングセットを選択すると、検証セットで得られる誤差も多少異なる結果になる。

```
set.seed(2)
train <- sample(392, 196)
lm.fit <- lm(mpg ~ horsepower, subset = train)
mean((mpg - predict(lm.fit, Auto))[-train]^2)
## [1] 25.72651
lm.fit2 <- lm(mpg ~ poly(horsepower, 2), data = Auto,
    subset = train)
mean((mpg - predict(lm.fit2, Auto))[-train]^2)
## [1] 20.43036
lm.fit3 <- lm(mpg ~ poly(horsepower, 3), data = Auto,
    subset = train)
mean((mpg - predict(lm.fit3, Auto))[-train]^2)
## [1] 20.38533
```

この方法で観測データをトレーニングセットと検証セットに分割すると、線形、二次、三次項を含むモデルの検証セット誤差率は、それぞれ 25.73、20.43、20.39 となる。

これらの結果は、以前の分析と一致している。つまり、`horsepower` の二次関数を用いて `mpg` を予測するモデルは、線形関数のみを使用するモデルよりも良いパフォーマンスを示す。一方で、三次関数を使用するモデルの優位性を示す証拠はほとんどない。

5.3.2 Leave-One-Out クロスバリデーション（LOOCV）

LOOCV の推定値は、`glm()` および `cv.glm()` 関数を使用して任意の一般化線形モデルに対して自動計算できる。第 4 章のラボでは、`glm()` 関数に `family = \"binomial\"` 引数を渡すことでロジスティック回帰を実行した。しかし、`glm()` を `family` 引数なしで使用すると、`lm()` 関数と同様に線形回帰を実行する。例えば、

```
glm.fit <- glm(mpg ~ horsepower, data = Auto)
coef(glm.fit)
## (Intercept)  horsepower
##  39.9358610  -0.1578447
```

及び

```
lm.fit <- lm(mpg ~ horsepower, data = Auto)
coef(lm.fit)
## (Intercept)  horsepower
##  39.9358610  -0.1578447
```

とすると、同一の線形回帰モデルが得られる。このラボでは、`cv.glm()` と併用できるため、`lm()` 関数ではなく `glm()` 関数を用いて線形回帰を実行する。

`cv.glm()` 関数は `boot` ライブラリの一部である。

```
library(boot)
glm.fit <- glm(mpg ~ horsepower, data = Auto)
cv.err <- cv.glm(Auto, glm.fit)
cv.err$delta
## [1] 24.23151 24.23114
```

`cv.glm()` 関数は、いくつかの要素を含むリストを出力する。`delta` ベクトルの 2 つの数値には、クロスバリデーションの結果が含まれる。この場合、両者の値は小数点 2 桁まで同じであり、(5.1) に示された LOOCV 統計量に対応する。以下では、これら 2 つの数値が異なる場合について説明する。この場合、テスト誤差に対するクロスバリデーションの推定値は約 24.23 となる。

次に、より複雑な多項式フィットに対してこの手順を繰り返えす。このプロセスを自動化するために、`for()` 関数を使用して *for ループ*を開始し、次数 $i = 1$ から $i = 10$ の多項式回帰を順次フィッティングし、それぞれのクロスバリデーション誤差を計算して、ベクトル `cv.error` の i 番目の要素に格納する。ここでは、ベクトルを初期化しよう。

```
cv.error <- rep(0, 10)
for (i in 1:10) {
  glm.fit <- glm(mpg ~ poly(horsepower, i), data = Auto)
  cv.error[i] <- cv.glm(Auto, glm.fit)$delta[1]
}
cv.error
##  [1] 24.23151 19.24821 19.33498 19.42443 19.03321 18.97864 18.83305 18.96115
##  [9] 19.06863 19.49093
```

図 5.4 と同様に、線形フィットと二次フィットの間で推定テスト MSE が大幅に減少することが確認できる。しかし、それ以上の高次の多項式を使用しても、明確な改善は見られない。

5.3.3 k 分割交差検証（k分割 CV）

`cv.glm()` 関数は、k分割 CV の実装にも使用できる。以下では、`Auto` データセットに対して、一般的な選択肢である $k = 10$ を使用しよう。再び乱数シードを設定

し、次数 1 から 10 の多項式フィットに対応するクロスバリデーション誤差を格納するためのベクトルを初期化する。

```
set.seed(17)
cv.error.10 <- rep(0, 10)
for (i in 1:10) {
  glm.fit <- glm(mpg ~ poly(horsepower, i), data = Auto)
  cv.error.10[i] <- cv.glm(Auto, glm.fit, K = 10)$delta[1]
}
cv.error.10
##  [1] 24.27207 19.26909 19.34805 19.29496 19.03198 18.89781 19.12061 19.14666
##  [9] 18.87013 20.95520
```

計算時間は LOOCV よりも短いことに注意する。（原則として、最小二乗線形モデルにおける LOOCV の計算時間は、式 (5.2) を利用できるため、k 分割 CV よりも速くなるはずである。しかし、残念ながら `cv.glm()` 関数はこの式を利用していない。）ここでは依然として、三次またはそれ以上の高次多項式を使用しても、単純な二次フィットよりもテスト誤差が低減するという明確な証拠は見られない。5.3.2 節で見たように、LOOCV を実行すると `delta` に関連する 2 つの数値は本質的に同一となる。一方、k 分割 CV を実行すると、`delta` に関連する 2 つの数値はわずかに異なる。最初の数値は標準的な k 分割 CV の推定値（式 (5.3) に対応）、2 番目の数値はバイアス補正バージョンである。このデータセットでは、両者の推定値は非常に近いものとなっている。

5.3.4 ブートストラップ

5.2 節での単純な例、および `Auto` データセットを使用した線形回帰モデルの精度推定に関する例を通じて、ブートストラップの使用方法を示しておこう。

興味のある統計量の精度推定

ブートストラップ手法の大きな利点の一つは、ほぼすべての状況に適用できることにある。複雑な数学的計算は必要なく、`R` でブートストラップ解析を実行するには 2 つのステップのみが必要となる。まず、関心のある統計量を計算する関数を作成する。次に、`boot` ライブラリの一部である `boot()` 関数を使用し、データセットから復元抽出を繰り返し行うことでブートストラップを実行する。

`ISLR2` パッケージの `Portfolio` データセットは、5.2 節で説明されている方法に従って生成された 100 組のリターンデータのシミュレーションである。このデータに対してブートストラップを適用する方法を示すために、まず `alpha.fn()` という関数を作成する必要がある。この関数は、(X, Y) データと、どの観測値を使用して α を推定するかを示すベクトルを入力として受け取り、選択された観測値に基づいて α の推定値を出力する。

```
alpha.fn <- function(data, index) {
  X <- data$X[index]
  Y <- data$Y[index]
  (var(Y) - cov(X, Y)) / (var(X) + var(Y) - 2 * cov(X, Y))
}
```

この関数は、引数 index で指定された観測値に (5.7) を適用し、α の推定値を*返す*（出力する）。例えば、次のコマンドは、すべての 100 個の観測値を使用して α を推定するように `R` に指示する。

```
alpha.fn(Portfolio, 1:100)
## [1] 0.5758321
```

次のコマンドでは、`sample()` 関数を使用して、1 から 100 の範囲から 100 個の観測値を復元抽出でランダムに選択する。これは、新しいブートストラップデータセットを構築し、そのデータセットに基づいて $\hat{\alpha}$ を再計算するのと同等となる。

```
set.seed(7)
alpha.fn(Portfolio, sample(100, 100, replace = T))
## [1] 0.5385326
```

このコマンドを何度も実行し、対応する αの推定値をすべて記録し、得られた標準偏差を計算することで、ブートストラップ解析を実装できる。しかし、`boot()` 関数を使用すれば、このプロセスを自動化できる。以下では、$R = 1{,}000$ 回のブートストラップを実行し、α を推定しよう。

```
boot(Portfolio, alpha.fn, R = 1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Portfolio, statistic = alpha.fn, R = 1000)
##
##
## Bootstrap Statistics :
##      original       bias    std. error
## t1* 0.5758321 0.0007959475  0.08969074
```

最終的な出力は、元のデータを使用した場合に$\hat{\alpha} = 0.5758$であることと、ブートストラップによる${\rm SE}(\hat{\alpha})$の推定値が 0.0897 であることを示している。

線形回帰モデルの精度推定

ブートストラップ手法は、統計的学習手法における係数推定値や予測のばらつきを評価するために使用できる。

ここでは、`Auto` データセットを用いた線形回帰モデルにおいて、`horsepower` を用いて `mpg` を予測する際の切片 β_0 と傾き β_1 の推定値のばらつきを評価するために、ブートストラップを適用する。また、ブートストラップによって得られた推定値と、3.1.2 節で説明した$\mathrm{SE}(\hat{\beta}_0)$および $\mathrm{SE}(\hat{\beta}_1)$の公式を用いた推定値とを比較する。

まず、`Auto` データセットと観測データのインデックスのセットを入力とし、線形回帰モデルの切片と傾きの推定値を返すシンプルな関数 `boot.fn()` を作成する。次に、この関数を 392 個のすべての観測データに適用し、第 3 章の通常の線形回帰係数推定式を用いて β_0 と β_1 の推定値を計算する。なお、この関数は 1 行のみのコードで定義されているため、関数の前後に `{,}` を記述する必要はない。

```
boot.fn <- function(data, index)
  coef(lm(mpg ~ horsepower, data = data, subset = index))
boot.fn(Auto, 1:392)
```

```
## (Intercept)  horsepower
##  39.9358610  -0.1578447
```

`boot.fn()` 関数は、観測データから復元抽出を行い、切片と傾きのブートストラップ推定値を作成するためにも使用できる。ここでは、その例を 2 つ示しておく。

```
set.seed(1)
boot.fn(Auto, sample(392, 392, replace = T))
## (Intercept)  horsepower
##  40.3404517  -0.1634868
boot.fn(Auto, sample(392, 392, replace = T))
## (Intercept)  horsepower
##  40.1186906  -0.1577063
```

次に、`boot()` 関数を使用して、切片と傾きの 1,000 回のブートストラップ推定値に対する標準誤差を計算する。

```
boot(Auto, boot.fn, 1000)
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Auto, statistic = boot.fn, R = 1000)
##
##
## Bootstrap Statistics :
##       original        bias    std. error
## t1* 39.9358610  0.0544513229 0.841289790
## t2* -0.1578447 -0.0006170901 0.007343073
```

この結果から、ブートストラップによる $\mathrm{SE}(\hat{\beta}_0)$の推定値は 0.84、$\mathrm{SE}(\hat{\beta}_1)$の推定値は 0.0073 であることがわかる。3.1.2 節で説明したように、線形モデルの回帰係数

の標準誤差は標準的な公式を用いて計算できる。これらの値は、`summary()` 関数を使用して取得できる。

```
summary(lm(mpg ~ horsepower, data = Auto))$coef
##               Estimate  Std. Error   t value      Pr(>|t|)
## (Intercept) 39.9358610 0.717498656  55.65984 1.220362e-187
## horsepower  -0.1578447 0.006445501 -24.48914  7.031989e-81
```

3.1.2 節の公式を用いて求めた $\hat{\beta}_0$ と $\hat{\beta}_1$ の標準誤差の推定値は、それぞれ切片で 0.717、傾きで 0.0064 である。興味深いことに、これらの値はブートストラップによる推定値とはやや異なっている。これはブートストラップに問題があることを示しているのだろうか？ 実際には、その逆である。ページ 66 の式 3.8 で示された標準的な公式は、いくつかの仮定に依存している。例えば、これらの公式は未知のパラメータ σ^2（ノイズの分散）に依存しており、σ^2 は残差平方和（RSS）を用いて推定する。しかし、標準誤差の公式自体は線形モデルが正しいことを前提としていないものの、σ^2 の推定値は線形モデルの正しさに依存する。92 ページの図 3.8 では、データに非線形関係が存在することが確認できる。そのため、線形モデルを適用すると残差が過大になり、それに伴い $\hat{\sigma}^2$ も過大評価される。さらに、標準的な公式は、x_i が固定されており、すべての変動は誤差 ϵ_i のみに起因するという仮定を置いている。しかし、この仮定は必ずしも現実的ではない。一方、ブートストラップ手法はこれらの仮定に依存せず、より正確な標準誤差の推定値を提供する可能性が高い。そのため、ブートストラップによる $\hat{\beta}_0$ と $\hat{\beta}_1$ の標準誤差の推定値は、`summary()` 関数を使用した場合よりも信頼性が高いと考えられる。

以下では、ブートストラップによる標準誤差の推定値と、二次モデルをデータにフィットさせた場合の標準的な線形回帰の推定値を計算する。このモデルはデータに適合しやすいため（図 3.8 参照）、ブートストラップの推定値と、$\mathrm{SE}(\hat{\beta}_0)$、$\mathrm{SE}(\hat{\beta}_1)$、$\mathrm{SE}(\hat{\beta}_2)$の標準推定値との対応がより良くなる。

```
boot.fn <- function(data, index)
  coef(
      lm(mpg ~ horsepower + I(horsepower^2),
        data = data, subset = index)
    )
set.seed(1)
boot(Auto, boot.fn, 1000)
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Auto, statistic = boot.fn, R = 1000)
##
##
## Bootstrap Statistics :
##         original        bias     std. error
## t1* 56.900099702  3.511640e-02 2.0300222526
## t2* -0.466189630 -7.080834e-04 0.0324241984
## t3*  0.001230536  2.840324e-06 0.0001172164
```

```
summary(
    lm(mpg ~ horsepower + I(horsepower^2), data = Auto)
  )$coef
```

```
##                     Estimate   Std. Error   t value      Pr(>|t|)
## (Intercept)     56.900099702 1.8004268063  31.60367 1.740911e-109
## horsepower      -0.466189630 0.0311246171 -14.97816  2.289429e-40
## I(horsepower^2)  0.001230536 0.0001220759  10.08009  2.196340e-21
```

ISLR 第 6 章 実習：線形モデルと正則化 (Linear Models and Regularization)

サブセット選択法

ベストサブセット選択

ここでは、`Hitters` データにベストサブセット選択法を適用しよう。前年の成績に基づいて野球選手の `Salary`（年俸）を予測したいとする。

まず、`Salary` 変数が一部の選手で欠落していることに注目しよう。`is.na()` 関数を使用すると、欠損観測値を特定できる。入力ベクトルと同じ長さのベクトルを返し、欠損している要素には `TRUE`、欠損していない要素には `FALSE` を返す。その後、`sum()` 関数を使用すればすべての欠損要素をカウントできる。

```
library(ISLR2)
names(Hitters)
##  [1] "AtBat"     "Hits"      "HmRun"     "Runs"      "RBI"       "Walks"
##  [7] "Years"     "CAtBat"    "CHits"     "CHmRun"    "CRuns"     "CRBI"
## [13] "CWalks"    "League"    "Division"  "PutOuts"   "Assists"   "Errors"
## [19] "Salary"    "NewLeague"
dim(Hitters)
## [1] 322  20
sum(is.na(Hitters$Salary))
## [1] 59
```

したがって、`Salary` が欠落している選手は 59 名であることがわかる。`na.omit()` 関数を使用すると、欠損値がある行をすべて削除できる。

```
Hitters <- na.omit(Hitters)
dim(Hitters)
```

```
## [1] 263  20
sum(is.na(Hitters))
## [1] 0
```

`regsubsets()` 関数（`leaps` ライブラリの一部）は、RSS を使用してベスト(最良)を定義し、指定された予測子の数を含む最良のモデルを特定することで、ベストサブセット選択を実行する。この構文は `lm()` と同様である。`summary()` コマンドは各モデルサイズの最適な変数セットを出力する。

訳注：パッケージから `leaps` を読み込む必要がある。内臓されてないパッケージは R の上側コマンドをクリック、ダウンロードしてからインストールする必要がある。

```
library(leaps)
regfit.full <- regsubsets(Salary ~ ., Hitters)
summary(regfit.full)
## Subset selection object
## Call: regsubsets.formula(Salary ~ ., Hitters)
## 19 Variables  (and intercept)
##            Forced in Forced out
## AtBat          FALSE      FALSE
```

…(訳者中略)

```
## NewLeagueN     FALSE      FALSE
## 1 subsets of each size up to 8
## Selection Algorithm: exhaustive
##          AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun CRuns CRB
I
## 1  ( 1 ) " "   " "  " "   " "  " " " " " "   " "   " "    " "   " "    " "   "*"
## 2  ( 1 ) " "   "*"  " "   " "  " " " " " "   " "   " "    " "   " "    " "   "*"
## 3  ( 1 ) " "   "*"  " "   " "  " " " " " "   " "   " "    " "   " "    " "   "*"
## 4  ( 1 ) " "   "*"  " "   " "  " " " " " "   " "   " "    " "   " "    " "   "*"
## 5  ( 1 ) "*"   "*"  " "   " "  " " " " " "   " "   " "    " "   " "    " "   "*"
## 6  ( 1 ) "*"   "*"  " "   " "  " " " " "*"   " "   " "    " "   " "    " "   "*"
## 7  ( 1 ) " "   "*"  " "   " "  " " " " "*"   " "   "*"    "*"   "*"    " "   " "
## 8  ( 1 ) "*"   "*"  " "   " "  " " " " "*"   " "   " "    " "   "*"    "*"   " "
##          CWalks LeagueN DivisionW PutOuts Assists Errors NewLeagueN
```

```
## 1  ( 1 ) " "    " "     " "       " "     " "     " "    " "
## 2  ( 1 ) " "    " "     " "       " "     " "     " "    " "
## 3  ( 1 ) " "    " "     " "       "*"     " "     " "    " "
## 4  ( 1 ) " "    " "     "*"       "*"     " "     " "    " "
## 5  ( 1 ) " "    " "     "*"       "*"     " "     " "    " "
## 6  ( 1 ) " "    " "     "*"       "*"     " "     " "    " "
## 7  ( 1 ) " "    " "     "*"       "*"     " "     " "    " "
## 8  ( 1 ) "*"    " "     "*"       "*"     " "     " "    " "
```

アスタリスクは、対応するモデルに含まれる変数を示している。例えば、この出力は、最適な 2 変数モデルには `Hits` と `CRBI` のみが含まれていることを示している。デフォルトでは、`regsubsets()` は 8 変数までの最適なモデルのみを報告するが、`nvmax` オプションを使用して任意の数の変数まで戻すことができる。ここでは最大 19 変数モデルをフィットしてみよう。

```
regfit.full <- regsubsets(Salary ~ ., data = Hitters,
    nvmax = 19)
reg.summary <- summary(regfit.full)
```

`summary()` 関数は、R^2、RSS、調整済み R^2、C_p、および BIC も返す。これらを調べて最適な全体モデルを選択できる。

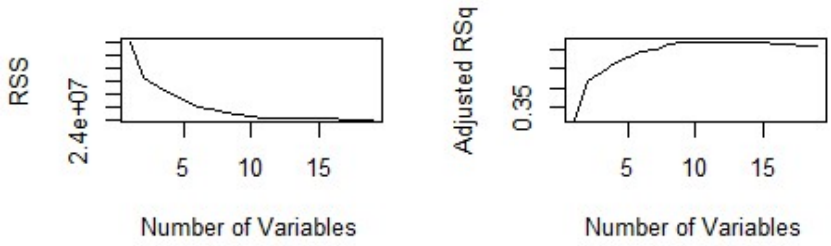
```
names(reg.summary)
## [1] "which"  "rsq"    "rss"    "adjr2"  "cp"     "bic"    "outmat" "obj"
```

例えば、1 変数のみを含むモデルの R^2 は 32% から始まり、すべての変数が含まれるモデルではほぼ 55% に増加することがわかる。予想通り、R^2 統計量は変数が増えるごとに単調に増加する。

```
reg.summary$rsq
##  [1] 0.3214501 0.4252237 0.4514294 0.4754067 0.4908036 0.5087146 0.5141227
##  [8] 0.5285569 0.5346124 0.5404950 0.5426153 0.5436302 0.5444570 0.5452164
## [15] 0.5454692 0.5457656 0.5459518 0.5460945 0.5461159
```

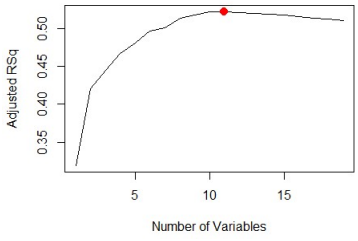
すべてのモデルの RSS、調整済み R^2、C_p および BIC を一度にプロットすることで、どのモデルを選択するかを判断するのに役立つ。`type = "l"` オプションは、プロットされたポイントを線で結ぶように `R` に指示する。

```
par(mfrow = c(2, 2))
plot(reg.summary$rss, xlab = "Number of Variables",
    ylab = "RSS", type = "l")
plot(reg.summary$adjr2, xlab = "Number of Variables",
    ylab = "Adjusted RSq", type = "l")
```



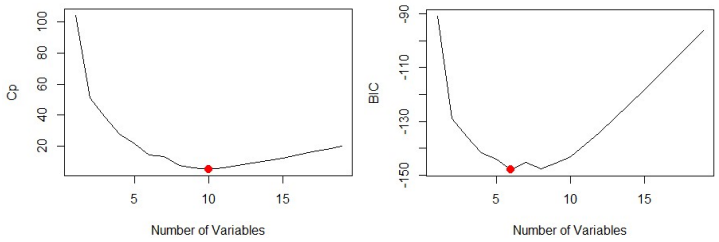
`points()` コマンドは、既に作成されたプロットにポイントを配置するためのもので、新しいプロットを作成する `plot()` コマンドとは異なる。`which.max()` 関数を使用してベクトルの最大点の位置を特定できる。次に、調整済み R^2 統計量が最大のモデルを示す赤い点をプロットする。

```
which.max(reg.summary$adjr2)
## [1] 11
plot(reg.summary$adjr2, xlab = "Number of Variables",
    ylab = "Adjusted RSq", type = "l")
points(11, reg.summary$adjr2[11], col = "red", cex = 2,
    pch = 20)
```



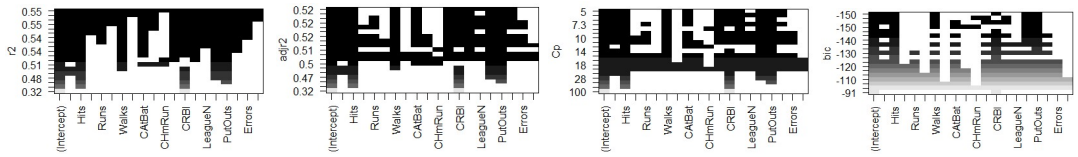
同様に、C_p と BIC 統計量をプロットし、最小の統計量のモデルを `which.min()` を使用して示しておく。

```
plot(reg.summary$cp, xlab = "Number of Variables",
    ylab = "Cp", type = "l")
which.min(reg.summary$cp)
## [1] 10
points(10, reg.summary$cp[10], col = "red", cex = 2,
    pch = 20)
which.min(reg.summary$bic)
## [1] 6
plot(reg.summary$bic, xlab = "Number of Variables",
    ylab = "BIC", type = "l")
points(6, reg.summary$bic[6], col = "red", cex = 2,
    pch = 20)
```



`regsubsets()`関数には組み込みの `plot()`コマンドがあり、指定された数の予測変数で最適なモデルを、BIC、C_p、調整済みR^2 または AIC に基づいて表示できる。関数の詳細については `?plot.regsubsets` と入力してみるとよい。

```
plot(regfit.full, scale = "r2")
plot(regfit.full, scale = "adjr2")
plot(regfit.full, scale = "Cp")
plot(regfit.full, scale = "bic")
```



各プロットの最上行には、最適なモデルで選択された変数ごとに黒い四角が表示されている。例えば、いくつかのモデルが BIC が−150に近い値を持つことがわかる。しかし、BIC が最も低いモデルは、`AtBat`、`Hits`、`Walks`、`CRBI`、`DivisionW`、および `PutOuts` の 6 つの変数のみを含むモデルである。このモデルに関連する係数推定値は、`coef()`関数を使用して確認できる。

```
coef(regfit.full, 6)
##  (Intercept)        AtBat         Hits        Walks         CRBI    Division
W
##   91.5117981   -1.8685892    7.6043976    3.6976468    0.6430169 -122.951533
8
##      PutOuts
##    0.2643076
```

前進・後退逐次選択法(Forward and Backward Stepwise Selection)

`regsubsets()` 関数に引数 `method = "forward"` または `method = "backward"` を指定することで、前進逐次選択法または後退逐次選択法を実行することができる。

```
regfit.fwd <- regsubsets(Salary ~ ., data = Hitters,
    nvmax = 19, method = "forward")
summary(regfit.fwd)
## Subset selection object
## Call: regsubsets.formula(Salary ~ ., data = Hitters, nvmax = 19, method = "f
orward")
## 19 Variables  (and intercept)
##            Forced in Forced out
## AtBat          FALSE      FALSE
```

…(訳者中略)

```
## NewLeagueN     FALSE      FALSE
## 1 subsets of each size up to 19
## Selection Algorithm: forward
##          AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun CRuns CRBI
## 1 ( 1 ) " "   " "  " "   " "  " " " " " "   " "   " "    " "   " "    " "   "*"
```

…(訳者中略)

```
## 19( 1 ) "*"   "*"  "*"   "*"  "*" "*"   "*"   "*"    "*"   "*"    "*"   "*"
##           CWalks LeagueN DivisionW PutOuts Assists Errors NewLeagueN
## 1  ( 1 )  " "    " "     " "       " "     " "     " "    " "
```

…(訳者中略)

```
## 19  ( 1 ) "*"    "*"     "*"       "*"     "*"     "*"    "*"
regfit.bwd <- regsubsets(Salary ~ ., data = Hitters,
    nvmax = 19, method = "backward")
summary(regfit.bwd)
## Subset selection object
## Call: regsubsets.formula(Salary ~ ., data = Hitters, nvmax = 19, method = "b
ackward")
## 19 Variables  (and intercept)
##            Forced in Forced out
## AtBat          FALSE      FALSE
```

…(訳者中略)

```
## NewLeagueN     FALSE      FALSE
## 1 subsets of each size up to 19
## Selection Algorithm: backward
##           AtBat Hits HmRun Runs RBI Walks Years CAtBat CHits CHmRun CRuns CR
BI
## 1 ( 1 )  " "   " "  " "   " "  " " " " " "   " "   " "    " "   " "    "*"   " "
```

…(訳者中略)

```
## 19 ( 1 ) "*"   "*"  "*"   "*"  "*" "*"   "*"   "*"    "*"   "*"    "*"   "*"
##           CWalks LeagueN DivisionW PutOuts Assists Errors NewLeagueN
## 1  ( 1 )  " "    " "     " "       " "     " "     " "    " "
```

```
…(訳者中略)
## 19  ( 1 ) "*"    "*"     "*"       "*"     "*"     "*"    "*"
```

例えば、前進逐次選択法では、最適な 1 変数モデルは `CRBI` のみを含み、最適な 2 変数モデルにはさらに `Hits` が含まれる。このデータでは、1 変数から 6 変数の最適なモデルは、ベストサブセット選択法および前進逐次選択法では同一であるが、最適な 7 変数モデルは異なる。

```
coef(regfit.full, 7)
## (Intercept)         Hits        Walks       CAtBat        CHits       CHmRun
##  79.4509472    1.2833513    3.2274264   -0.3752350    1.4957073    1.4420538
##   DivisionW      PutOuts
##-129.9866432    0.2366813
coef(regfit.fwd, 7)
## (Intercept)        AtBat         Hits        Walks         CRBI       CWalks
## 109.7873062   -1.9588851    7.4498772    4.9131401    0.8537622   -0.3053070
##   DivisionW      PutOuts
##-127.1223928    0.2533404
coef(regfit.bwd, 7)
## (Intercept)        AtBat         Hits        Walks        CRuns       CWalks
## 105.6487488   -1.9762838    6.7574914    6.0558691    1.1293095   -0.7163346
##   DivisionW      PutOuts
##-116.1692169    0.3028847
```

検証セット(Validation-Set)法と交差検証(Cross-Validation)によるモデル選択

ここまで、C_p、BIC および調整済み R^2 を用いて異なるサイズのモデルの中から選択する方法を見てきた。次に、検証セット法と交差検証法を使用してこれを行う方法を考える。

これらの方法で正確なテスト誤差を推定するには、トレーニング観測値のみを使用して、変数選択を含むすべてのモデルフィッティングを行う必要がある。したがって、指定されたサイズのどのモデルが最適かの決定はトレーニング観測値の

みで行う必要がある。この点は微妙ではあるが重要な問題である。もし全データセットを使ってベストサブセット選択を行うと、得られる検証セット誤差や交差検証誤差はテスト誤差の正確な推定にはならない。

検証セット法を使用するために、まず観測値をトレーニングセットとテストセットに分割する。ここでは、トレーニングセットに含まれる観測値には `TRUE`、それ以外には `FALSE` を返すランダムなベクトル `train` を作成する。また、`!` 演算子を使用して `TRUE` と `FALSE` を反転させ、テストセットの観測値には `TRUE` を、それ以外には `FALSE` を返す `test` ベクトルを作成する。また、ユーザーが同じトレーニングセット/テストセット分割を再現できるよう、乱数シードを設定する。

```
set.seed(1)
train <- sample(c(TRUE, FALSE), nrow(Hitters),
    replace = TRUE)
test <- (!train)
```

次に、`regsubsets()` 関数をトレーニングセットに適用し、ベストサブセット選択を実行する。

```
regfit.best <- regsubsets(Salary ~ .,
    data = Hitters[train, ], nvmax = 19)
```

トレーニングデータのサブセットにアクセスするために、`Hitters[train, ]` を使用してデータフレームを直接サブセット化している点に注目しよう。次に、各モデルサイズの最適モデルの検証セット誤差を計算する。まず、テストデータからモデル行列を作成する。

```
test.mat <- model.matrix(Salary ~ ., data = Hitters[test, ])
```

`model.matrix()` 関数は、多くの回帰パッケージでデータから「X」行列を構築するために使用される。次に、ループを実行し、各サイズ `i` に対して最適モデルの係数を `regfit.best` から抽出し、それらをテストモデル行列の適切な列に掛けて予測値を形成し、テスト MSE を計算する。

```
val.errors <- rep(NA, 19)
for (i in 1:19) {
 coefi <- coef(regfit.best, id = i)
 pred <- test.mat[, names(coefi)] %*% coefi
 val.errors[i] <- mean((Hitters$Salary[test] - pred)^2)
}
```

最適なモデルは 7 変数を含むモデルであることが分かる。

```
val.errors
##  [1] 164377.3 144405.5 152175.7 145198.4 137902.1 139175.7 126849.0 136191.4
##  [9] 132889.6 135434.9 136963.3 140694.9 140690.9 141951.2 141508.2 142164.4
## [17] 141767.4 142339.6 142238.2
which.min(val.errors)
## [1] 7
coef(regfit.best, 7)
## (Intercept)        AtBat         Hits        Walks        CRuns       CWalks
##  67.1085369   -2.1462987    7.0149547    8.0716640    1.2425113   -0.8337844
##   DivisionW      PutOuts
##-118.4364998    0.2526925
```

ここでの作業少し手間がかかるが、その一因としては、`regsubsets()`には`predict()`メソッドがないことが挙げられる。 この関数を再度使用するため、上記の手順をキャプチャして独自の `predict` メソッドを書いておこう。

```
 predict.regsubsets <- function(object, newdata, id, ...) {
  form <- as.formula(object$call[[2]])
  mat <- model.matrix(form, newdata)
  coefi <- coef(object, id = id)
  xvars <- names(coefi)
  mat[, xvars] %*% coefi
 }
```

この関数は、ほぼ上記で行った操作を再現している。唯一複雑な部分は、`regsubsets()`の呼び出しで使用された式をどのように抽出するかだろう。この関数をどのように使用するかは、以下でクロスバリデーションを行う際に示しておく。

最後に、データセット全体で最適な部分集合選択を行い、最良の 7 変数モデルを選択する。より正確な係数の推定値を得るためには、データセット全体を使用することが重要である。トレーニングセットから得られた変数を単に使用するのではなく、データセット全体で最良の 7 変数モデルを選択する理由は、データセット全体での最良の 7 変数モデルがトレーニングセットでの対応するモデルと異なる可能性があるためだろう。

```
regfit.best <- regsubsets(Salary ~ ., data = Hitters,
    nvmax = 19)
coef(regfit.best, 7)
## (Intercept)         Hits        Walks       CAtBat        CHits       CHmRun
##  79.4509472    1.2833513    3.2274264   -0.3752350    1.4957073    1.4420538
##   DivisionW      PutOuts
##-129.9866432    0.2366813
```

実際に、データセット全体での最良の 7 変数モデルは、トレーニングセットでの最良の 7 変数モデルとは異なる変数セットを持つことが分かる。

次に、クロスバリデーションを使用して異なるサイズのモデルを選択しよう。このアプローチはやや複雑で、各kトレーニングセット内で最適な部分集合選択を行う必要があるためだろう。それでも、`R`の巧妙なサブセット構文により、この作業は比較的容易に実行できる。まず、各観測を$k = 10$のフォールドの 1 つに割り当てるベクトルを作成し、結果を格納するための行列を作成してみよう。

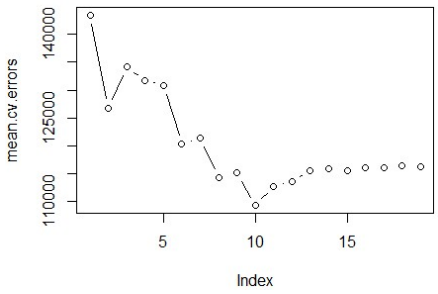
```
k <- 10
n <- nrow(Hitters)
set.seed(1)
folds <- sample(rep(1:k, length = n))
cv.errors <- matrix(NA, k, 19,
    dimnames = list(NULL, paste(1:19)))
```

次に、クロスバリデーションを実行する `for` ループを書こう。j番目のフォールドでは、`folds` の要素が `j` に等しいものがテストセットに含まれ、それ以外の要素がトレーニングセットに含まれる。各モデルサイズについて（新しい `predict()` メソッドを使用して）予測を行い、適切なサブセットでテストエラーを計算し、`cv.errors` 行列の適切なスロットに格納する。以下のコードでは、`best.fit` オブジェクトがクラス `regsubsets` であるため、`predict()` を呼び出す際に自動的に `predict.regsubsets()` 関数が使用されることに注意する必要がある。

```
for (j in 1:k) {
  best.fit <- regsubsets(Salary ~ .,
        data = Hitters[folds != j, ],
        nvmax = 19)
  for (i in 1:19) {
    pred <- predict(best.fit, Hitters[folds == j, ], id = i)
    cv.errors[j, i] <-
         mean((Hitters$Salary[folds == j] - pred)^2)
    }
 }
```

これにより 10×19 の行列が得られ、(j, i)番目の要素は最良のi変数モデルのj番目のクロスバリデーションフォールドのテスト MSE に対応する。`apply()` 関数を使用してこの行列の列を平均化し、i番目の要素がi変数モデルのクロスバリデーションエラーであるベクトルを取得してみよう。

```
mean.cv.errors <- apply(cv.errors, 2, mean)
mean.cv.errors
##        1        2        3        4        5        6        7        8
## 143439.8 126817.0 134214.2 131782.9 130765.6 120382.9 121443.1 114363.7
##        9       10       11       12       13       14       15       16
## 115163.1 109366.0 112738.5 113616.5 115557.6 115853.3 115630.6 116050.0
##       17       18       19
## 116117.0 116419.3 116299.1
par(mfrow = c(1, 1))
plot(mean.cv.errors, type = "b")
```



クロスバリデーションにより、10 変数モデルが選択されることが分かる。データセット全体で最適な部分集合選択を行い、10 変数モデルが得られる。

```
reg.best <- regsubsets(Salary ~ ., data = Hitters,
    nvmax = 19)
coef(reg.best, 10)
##  (Intercept)        AtBat         Hits        Walks       CAtBat        CRun
s
##  162.5354420   -2.1686501    6.9180175    5.7732246   -0.1300798    1.408249
0
##         CRBI       CWalks    DivisionW      PutOuts      Assists
##    0.7743122   -0.8308264 -112.3800575    0.2973726    0.2831680
```

リッジ回帰(Ridge Regression)とラッソ(Lasso)

リッジ回帰およびラッソを実行するために、`glmnet` パッケージを使用する。このパッケージの主な関数は `glmnet()` で、リッジ回帰モデル、ラッソ・モデルなどをフィットするために利用できる。なお、この関数は、これまでこの本で紹介した他のモデル適合関数とは少し異なる構文を持っている。特に、`x` 行列と `y` ベクトルを渡す必要があり、`y ~ x` 構文は使用しない。 ここでは、`Hitters` データを用いて `Salary` を予測するためにリッジ回帰とラッソを実行してみよう。なお、進める前に、6.5.1 節で説明したように、データから欠損値が取り除かれていることを確認しておこう。

```
x <- model.matrix(Salary ~ ., Hitters)[, -1]
y <- Hitters$Salary
```

`model.matrix()` 関数は特に `x` を作成する際に便利である。この関数は、19 個の予測変数に対応する行列を生成するだけでなく、質的変数を自動的にダミー変数に変換する。この後者の特性は重要である。なぜなら、`glmnet()` 関数は数値データや定量的な入力しか扱えないからである。

リッジ回帰

`glmnet()` 関数には、どの種類のモデルを適合させるかを決定する `alpha` 引数がある。`alpha=0` の場合、リッジ回帰モデルがフィットされ、`alpha=1` の場合はラッソ・モデルがフィットされる。ここでは、まずリッジ回帰モデルをフィットしてみよう。

```
library(glmnet)
## 要求されたパッケージ Matrix をロード中です
## Loaded glmnet 4.1-8
grid <- 10^seq(10, -2, length = 100)
ridge.mod <- glmnet(x, y, alpha = 0, lambda = grid)
```

デフォルトでは、`glmnet()` 関数は自動的に選択された λ 値の範囲でリッジ回帰を実行する。しかし、ここでは $\lambda = 10^{10}$ から $\lambda = 10^{-2}$ までの値を網羅するグリッドを使用して関数を実装しておき、切片のみを含む帰無モデルから最小二乗フィットによりあらゆるシナリオをカバーしておこう。また、元の grid 値に含まれていない特定の λ値に対してモデル適合を計算することも可能である。なお、デフォルトで `glmnet()` 関数は変数を標準化し、同じスケールに揃えることができる。このデフォルト設定を無効にするには、引数 `standardize = FALSE` を使用すればよい。

それぞれの λ値に関連付けられたリッジ回帰係数のベクトルは、`coef()` 関数でアクセスできる行列に格納される。この場合、行列は 20×100 のサイズで、20 行（各予測変数と切片 1 つ）と 100 列（λの各値）である。

```
dim(coef(ridge.mod))
## [1]  20 100
```

λの値が大きい場合、ℓ_2ノルムに基づく係数推定値が小さくなることが予想される。一方、λが小さい場合、係数推定値はより大きくなる。以下は、$\lambda = 11{,}498$ のときの係数とそのℓ_2ノルムである。

```
ridge.mod$lambda[50]
## [1] 11497.57
coef(ridge.mod)[, 50]
##   (Intercept)         AtBat          Hits         HmRun          Runs
## 407.356050200   0.036957182   0.138180344   0.524629976   0.230701523
##           RBI         Walks         Years        CAtBat         CHits
##   0.239841459   0.289618741   1.107702929   0.003131815   0.011653637
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##   0.087545670   0.023379882   0.024138320   0.025015421   0.085028114
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
##  -6.215440973   0.016482577   0.002612988  -0.020502690   0.301433531
sqrt(sum(coef(ridge.mod)[-1, 50]^2))
## [1] 6.360612
```

対照的に、$\lambda = 705$ のときの係数とそのℓ_2ノルムを示しておく。この小さい λ値に関連付けられた係数の ℓ_2ノルムがかなり大きいことに注目しておこう。

```
ridge.mod$lambda[60]
## [1] 705.4802
coef(ridge.mod)[, 60]
## (Intercept)        AtBat         Hits        HmRun         Runs          RBI
## 54.32519950   0.11211115   0.65622409   1.17980910   0.93769713   0.84718546
##       Walks        Years       CAtBat        CHits       CHmRun        CRuns
##  1.31987948   2.59640425   0.01083413   0.04674557   0.33777318   0.09355528
##        CRBI       CWalks      LeagueN    DivisionW      PutOuts      Assists
##  0.09780402   0.07189612  13.68370191 -54.65877750   0.11852289   0.01606037
##      Errors   NewLeagueN
## -0.70358655   8.61181213
sqrt(sum(coef(ridge.mod)[-1, 60]^2))
## [1] 57.11001
```

`predict()` 関数を使用すると、さまざまな目的で予測を行うことができる。たとえば、新しい λ値、ここでは 50 に対するリッジ回帰係数を取得することができる。

```
predict(ridge.mod, s = 50, type = "coefficients")[1:20, ]
##   (Intercept)         AtBat          Hits         HmRun          Runs
##  4.876610e+01 -3.580999e-01  1.969359e+00 -1.278248e+00  1.145892e+00
##           RBI         Walks         Years        CAtBat         CHits
##  8.038292e-01  2.716186e+00 -6.218319e+00  5.447837e-03  1.064895e-01
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##  6.244860e-01  2.214985e-01  2.186914e-01 -1.500245e-01  4.592589e+01
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -1.182011e+02  2.502322e-01  1.215665e-01 -3.278600e+00 -9.496680e+00
```

サンプルをトレーニングセットとテストセットに分割し、リッジ回帰およびラッソのテスト誤差を推定しよう。データセットをランダムに分割する一般的な方法は 2 つある。1 つ目は、`TRUE` と `FALSE` の要素からなるランダムなベクトルを作成し、`TRUE` に対応する観測値をトレーニングデータとして選択する方法である。2 つ目は、1 から n までの番号のサブセットをランダムに選択し、それをトレーニング観測値のインデックスとして使用する方法である。これらの 2 つの方法はどちらも同じように機能する。6.5.1 節では前者の方法を利用したが、ここでは後者の方法を示しておこう。

まず、ランダムシードを設定して、得られる結果が再現可能になるようにしておく。

```
set.seed(1)
train <- sample(1:nrow(x), nrow(x) / 2)
test <- (-train)
y.test <- y[test]
```

次に、トレーニングセットに対してリッジ回帰モデルをフィットさせ、$\lambda = 4$ を用いてテストセットの MSE を評価する。ここでも `predict()` 関数を使用する。この場合、`type="coefficients"` を `newx` 引数に置き換えることでテストセットに対する予測が得られる。

```
ridge.mod <- glmnet(x[train, ], y[train], alpha = 0,
    lambda = grid, thresh = 1e-12)
```

```
ridge.pred <- predict(ridge.mod, s = 4, newx = x[test, ])
mean((ridge.pred - y.test)^2)
## [1] 142199.2
```

テスト MSE は 142,199 となる。もし切片だけを持つモデルをフィットした場合、各テスト観測値をトレーニング観測値の平均で予測したであろう。その場合、次のようにしてテストセットの MSE を計算できる。

```
mean((mean(y[train]) - y.test)^2)
## [1] 224669.9
```

非常に大きなλ値を持つリッジ回帰モデルを適合させても同じ結果が得られる。なお、`1e10` は 10^{10} を意味する。

```
ridge.pred <- predict(ridge.mod, s = 1e10, newx = x[test, ])
mean((ridge.pred - y.test)^2)
## [1] 224669.8
```

このように、$\lambda = 4$ のリッジ回帰モデルをフィットすると、切片だけを持つモデルよりもテスト MSE が大幅に低くなる。

次に、$\lambda = 4$ でリッジ回帰を行うことにして、単に最小二乗回帰を行う場合と比べて利点があるかどうかを確認しよう。ここで最小二乗回帰は、$\lambda = 0$ のリッジ回帰と同じであることを思い出しておこう。

注： `glmnet()` で $\lambda = 0$ の場合に正確な最小二乗係数を得るには、`predict()` 関数を呼び出す際に引数 `exact = T` を使用する。そうでない場合、`predict()` 関数は `glmnet()` モデルを適合させる際に使用した λ値のグリッドを補間し、近似的な結果を返す。`exact = T` を使用すると、$\lambda = 0$ の場合の `glmnet()` の出力と `lm()` の出力の間に小数点以下第 3 位のわずかな差異が残るが、これは `glmnet()` の数値近似によるものである。

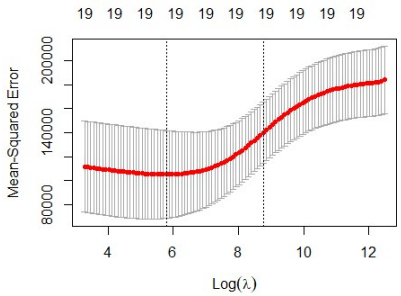
```
ridge.pred <- predict(ridge.mod, s = 0, newx = x[test, ],
    exact = T, x = x[train, ], y = y[train])
mean((ridge.pred - y.test)^2)
## [1] 168588.6
```

```
lm(y ~ x, subset = train)
##
## Call:
## lm(formula = y ~ x, subset = train)
##
## Coefficients:
##(Intercept)       xAtBat        xHits       xHmRun        xRuns         xRBI
##   274.0145      -0.3521      -1.6377       5.8145       1.5424       1.1243
##     xWalks       xYears      xCAtBat       xCHits      xCHmRun       xCRuns
##     3.7287     -16.3773      -0.6412       3.1632       3.4008      -0.9739
##      xCRBI      xCWalks     xLeagueN   xDivisionW     xPutOuts     xAssists
##    -0.6005       0.3379     119.1486    -144.0831       0.1976       0.6804
##    xErrors  xNewLeagueN
##    -4.7128     -71.0951
predict(ridge.mod, s = 0, exact = T, type = "coefficients",
    x = x[train, ], y = y[train])[1:20, ]
## (Intercept)        AtBat         Hits        HmRun         Runs          RBI
## 274.0200994   -0.3521900   -1.6371383    5.8146692    1.5423361    1.1241837
##       Walks        Years       CAtBat        CHits       CHmRun        CRuns
##   3.7288406  -16.3795195   -0.6411235    3.1629444    3.4005281   -0.9739405
##        CRBI       CWalks      LeagueN    DivisionW      PutOuts      Assists
##  -0.6003976    0.3378422 119.1434637 -144.0853061    0.1976300    0.6804200
##      Errors   NewLeagueN
##  -4.7127879  -71.0898914
```

一般的に、（ペナルティなしの）最小二乗モデルをフィットしたい場合は、`lm()` 関数を使用するのがよいだろう。`lm()` 関数は係数の標準誤差や p 値など、より有用な出力を提供してくれる。

任意に $\lambda = 4$ を選択する代わりに、チューニングパラメータ λを選択するためにクロスバリデーションを使用する方が望ましいだろう。このためには、組み込みのクロスバリデーション関数 `cv.glmnet()` を使用できる。デフォルトでは、この関数は 10 分割のクロスバリデーションを実行するが、引数 `nfolds` を使用して変更できる。クロスバリデーションフォールドの選択はランダムに行われるため、結果を再現可能にするために最初にランダムシードを設定しておく必要がある。

```
set.seed(1)
cv.out <- cv.glmnet(x[train, ], y[train], alpha = 0)
plot(cv.out)
```




```
bestlam <- cv.out$lambda.min
bestlam
## [1] 326.0828
```

これより、クロスバリデーションエラーが最小になるλの値は 326 であることが分かる。このλに関連するテスト MSE は次のようになる。

```
ridge.pred <- predict(ridge.mod, s = bestlam,
    newx = x[test, ])
mean((ridge.pred - y.test)^2)
## [1] 139856.6
```

これは、$\lambda = 4$ を使用した際に得られたテスト MSE よりもさらに改善されている。最後に、クロスバリデーションで選択されたλの値を利用して、完全なデータセットに対してリッジ回帰モデルをフィットして、係数推定値を確認しておこう。

```
out <- glmnet(x, y, alpha = 0)
predict(out, type = "coefficients", s = bestlam)[1:20, ]
## (Intercept)        AtBat         Hits        HmRun         Runs          RBI
## 15.44383120   0.07715547   0.85911582   0.60103106   1.06369007   0.87936105
```

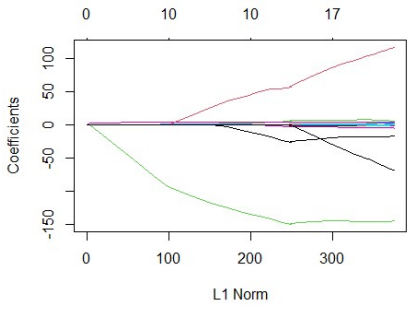
```
##       Walks       Years      CAtBat       CHits      CHmRun       CRuns
##  1.62444617  1.35254778  0.01134999  0.05746654  0.40680157  0.11456224
##        CRBI       CWalks      LeagueN    DivisionW      PutOuts      Assists
##  0.12116504  0.05299202 22.09143197 -79.04032656  0.16619903  0.02941950
##      Errors   NewLeagueN
## -1.36092945  9.12487765
```

予想どおり、係数はすべてゼロではなく、リッジ回帰では適切に変数選択を行えない！

ラッソ

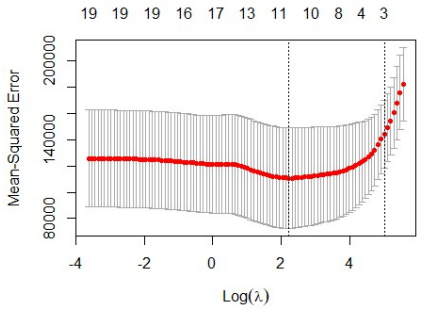
リッジ回帰において適切なλを選択することで、`Hitters`データセットにおいて最小二乗回帰や帰無モデルを上回る結果が得られることを確認した。次に、ラッソがリッジ回帰よりも精度が高い、あるいはより解釈しやすいモデルを提供できるかどうかを検討してみよう。ラッソモデルをフィットするには、再び`glmnet()`関数を使用しよう。ただし、今回は引数`alpha=1`を利用する。それ以外は、リッジモデルを適合させる場合と同様に進めればよい。

```
lasso.mod <- glmnet(x[train, ], y[train], alpha = 1,
    lambda = grid)
plot(lasso.mod)
## Warning in regularize.values(x, y, ties, missing(ties), na.rm = na.rm):
## ユニークな 'x' 値を得るため縮小します
```



係数プロットから、チューニングパラメータの選択に応じて、いくつかの係数が完全にゼロになることが分かる。次に、クロスバリデーションを実行して、関連するテストエラーを計算する。

```
set.seed(1)
cv.out <- cv.glmnet(x[train, ], y[train], alpha = 1)
plot(cv.out)
```




```
bestlam <- cv.out$lambda.min
lasso.pred <- predict(lasso.mod, s = bestlam,
    newx = x[test, ])
mean((lasso.pred - y.test)^2)
## [1] 143673.6
```

この値は、帰無モデルや最小二乗回帰のテストセット MSE よりも大幅に低く、クロスバリデーションで選択された λを用いたリッジ回帰のテスト MSE と非常に近い値である。

しかし、ラッソにはリッジ回帰に対する大きな利点がある。それは、得られる係数推定値がスパース（疎）であることによる。ここでは、19 個の係数推定値のうち 8 つが完全にゼロであることが確認できる。つまり、クロスバリデーションで選択された λを用いたラッソ・モデルには 11 個の変数しか含まれていない。

```
out <- glmnet(x, y, alpha = 1, lambda = grid)
lasso.coef <- predict(out, type = "coefficients",
    s = bestlam)[1:20, ]
lasso.coef
```

```
##   (Intercept)         AtBat          Hits         HmRun          Runs
##    1.27479059   -0.05497143    2.18034583    0.00000000    0.00000000
##           RBI         Walks         Years        CAtBat         CHits
##    0.00000000    2.29192406   -0.33806109    0.00000000    0.00000000
##        CHmRun         CRuns          CRBI        CWalks       LeagueN
##    0.02825013    0.21628385    0.41712537    0.00000000   20.28615023
##     DivisionW       PutOuts       Assists        Errors    NewLeagueN
## -116.16755870    0.23752385    0.00000000   -0.85629148    0.00000000
lasso.coef[lasso.coef != 0]
##   (Intercept)         AtBat          Hits         Walks         Years
##    1.27479059   -0.05497143    2.18034583    2.29192406   -0.33806109
##        CHmRun         CRuns          CRBI       LeagueN     DivisionW
##    0.02825013    0.21628385    0.41712537   20.28615023 -116.16755870
##       PutOuts        Errors
##    0.23752385   -0.85629148
```

主成分回帰（PCR）と部分最小二乗回帰（PLS）

主成分回帰 (PCR)

主成分回帰（PCR）は、`pls` ライブラリの `pcr()` 関数を使って実行できる。ここでは、`Hitters` データを使用して `Salary` を予測するために PCR を適用してみよう。まず、データから欠損値を除去していることを確認しておこう。

```
library(pls)
##
## 次のパッケージを付け加えます: 'pls'

## 以下のオブジェクトは 'package:stats' からマスクされています:

##
##     loadings
```

```
set.seed(2)
pcr.fit <- pcr(Salary ~ ., data = Hitters, scale = TRUE,
    validation = "CV")
```

`pcr()` 関数の構文は `lm()` と似ているが、いくつかの追加オプションがある。`scale = TRUE` を設定すると、各変数が標準化される。`validation = "CV"` を設定すると、各主成分数 M に対して 10 分割交差検証が行われる。`summary()` 関数で結果を確認しておこう。

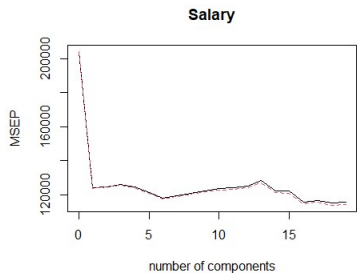
```
summary(pcr.fit)
## Data:    X dimension: 263 19
##  Y dimension: 263 1
## Fit method: svdpc
## Number of components considered: 19
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV             452    351.9    353.2    355.0    352.8    348.4    343.6
## adjCV          452    351.6    352.7    354.4    352.1    347.6    342.7
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## CV       345.5    347.7    349.6     351.4     352.1     353.5     358.2
## adjCV    344.7    346.7    348.5     350.1     350.7     352.0     356.5
##        14 comps  15 comps  16 comps  17 comps  18 comps  19 comps
## CV        349.7     349.4     339.9     341.6     339.2     339.6
## adjCV     348.0     347.7     338.2     339.7     337.2     337.6
##
## TRAINING: % variance explained
##       1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps
## X       38.31    60.16    70.84    79.03    84.29    88.63    92.26    94.96
## Salary  40.63    41.58    42.17    43.22    44.90    46.48    46.69    46.75
##       9 comps  10 comps  11 comps  12 comps  13 comps  14 comps  15 comps
## X       96.28     97.26     97.98     98.65     99.15     99.47     99.75
## Salary  46.86     47.76     47.82     47.85     48.10     50.40     50.55
```

```
##       16 comps  17 comps  18 comps  19 comps
## X        99.89     99.97     99.99    100.00
## Salary   53.01     53.85     54.61     54.61
```

各主成分数に対する CV スコアが表示されている。たとえば、CV の平均二乗誤差(MSE)を取得するには、`pcr()`で出力される平方根平均二乗誤差を 2 乗すればよい。

`validationplot()` 関数を使用し、`val.type = "MSEP"` を指定すると、MSE を視覚化することができる。

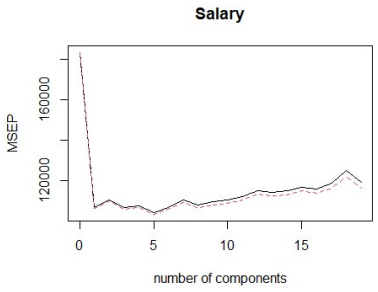
```
validationplot(pcr.fit, val.type = "MSEP")
```



この結果、$M = 18$ のときに最も低い CV 誤差が得られることが分かる。
PCR においてすべての成分を使用すると次元削減が行われないため、$M = 18$のときは単純な最小二乗法とほぼ同等である。 しかし、プロットから分かるように、モデルに 1 つの成分のみを含めた場合でも、交差検証誤差はほぼ同じである。
これは、少数の成分を使用するだけでも十分なモデルが構築できる可能性を示唆している。

次に、トレーニングデータで PCR を実行し、テストセットの性能を評価しよう。

```
set.seed(1)
pcr.fit <- pcr(Salary ~ ., data = Hitters, subset = train,
    scale = TRUE, validation = "CV")
validationplot(pcr.fit, val.type = "MSEP")
```



最小の CV 誤差は $M = 5$ のときに達成される。以下のコードでテスト MSE を計算できる。

```
pcr.pred <- predict(pcr.fit, x[test, ], ncomp = 5)
mean((pcr.pred - y.test)^2)
## [1] 142811.8
```

このテスト MSE は、リッジ回帰やラッソと競合する結果となる。ただし、PCR は変数選択を行わないため、モデルの解釈が難しくなる。

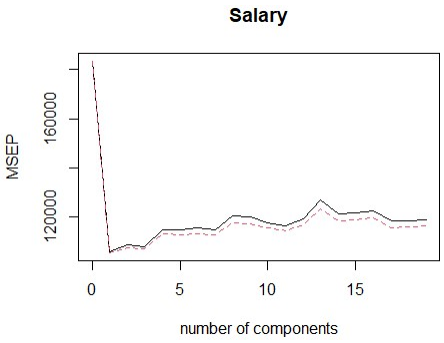
最後に、$M = 5$ を使用して全データセットに PCR をフィットしてみよう。

```
pcr.fit <- pcr(y ~ x, scale = TRUE, ncomp = 5)
summary(pcr.fit)
## Data:    X dimension: 263 19
##  Y dimension: 263 1
## Fit method: svdpc
## Number of components considered: 5
## TRAINING: % variance explained
##    1 comps  2 comps  3 comps  4 comps  5 comps
## X    38.31    60.16    70.84    79.03    84.29
## y    40.63    41.58    42.17    43.22    44.90
```

部分最小二乗法 (PLS)

部分最小二乗法 (PLS) は、`pls` ライブラリの `plsr()` 関数を使用して実装できる。構文は `pcr()` 関数と同一である。

```r
set.seed(1)
pls.fit <- plsr(Salary ~ ., data = Hitters, subset = train, scale = TRUE, valid
ation = "CV")
summary(pls.fit)
## Data:    X dimension: 131 19
##  Y dimension: 131 1
## Fit method: kernelpls
## Number of components considered: 19
##
## VALIDATION: RMSEP
## Cross-validated using 10 random segments.
##        (Intercept)  1 comps  2 comps  3 comps  4 comps  5 comps  6 comps
## CV           428.3    325.5    329.9    328.8    339.0    338.9    340.1
## adjCV        428.3    325.0    328.2    327.2    336.6    336.1    336.6
##        7 comps  8 comps  9 comps  10 comps  11 comps  12 comps  13 comps
## CV       339.0    347.1    346.4     343.4     341.5     345.4     356.4
## adjCV    336.2    343.4    342.8     340.2     338.3     341.8     351.1
##        14 comps  15 comps  16 comps  17 comps  18 comps  19 comps
## CV        348.4     349.1     350.0     344.2     344.5     345.0
## adjCV     344.2     345.0     345.9     340.4     340.6     341.1
##
## TRAINING: % variance explained
##       1 comps  2 comps  3 comps  4 comps  5 comps  6 comps  7 comps  8 comps
## X       39.13    48.80    60.09    75.07    78.58    81.12    88.21    90.71
## Salary  46.36    50.72    52.23    53.03    54.07    54.77    55.05    55.66
##       9 comps  10 comps  11 comps  12 comps  13 comps  14 comps  15 comps
## X       93.17     96.05     97.08     97.61     97.97     98.70     99.12
## Salary  55.95     56.12     56.47     56.68     57.37     57.76     58.08
##       16 comps  17 comps  18 comps  19 comps
## X        99.61     99.70     99.95    100.00
## Salary   58.17     58.49     58.56     58.62
validationplot(pls.fit, val.type = "MSEP")
```



最小の CV 誤差は $M = 1$ のときに達成されている。これに基づくテスト MSE を評価しておこう。

```
pls.pred <- predict(pls.fit, x[test, ], ncomp = 1)
mean((pls.pred - y.test)^2)
## [1] 151995.3
```

このテスト MSE はリッジ回帰、ラッソ、PCR に匹敵するが、わずかに大きくなる。最後に、全データセットに対して $M = 1$ を用いた PLS を実行してみよう。

```
pls.fit <- plsr(Salary ~ ., data = Hitters, scale = TRUE,
    ncomp = 1)
summary(pls.fit)
## Data:    X dimension: 263 19
##  Y dimension: 263 1
## Fit method: kernelpls
## Number of components considered: 1
## TRAINING: % variance explained
##         1 comps
## X         38.08
## Salary    43.05
```

PLS モデルの 1 成分が `Salary` の分散の 43.05% を説明していることが分かる。PCR は予測変数の分散を最大化する方向に焦点を当てているが、PLS は予測変数と応答の両方の分散を説明する方向を見つけるため、このような違いが生じる。

ISLR 第 7 章 非線形モデリング

このラボでは、この章の例で使用した `Wage` データを分析、多くの複雑な非線形フィッティング手法が `R` で簡単に実装できることを示す。まず、データを含む `ISLR2` ライブラリを読み込む。

```
library(ISLR2)
attach(Wage)
```

多項式回帰とステップ関数

図 7.1 の作成方法を確認します。まず、以下のコマンドを使用してモデルをフィットする。

```
fit <- lm(wage ~ poly(age, 4), data = Wage)
coef(summary(fit))
```

```
##                Estimate Std. Error    t value     Pr(>|t|)
## (Intercept)    111.70361  0.7287409 153.283015 0.000000e+00
## poly(age, 4)1  447.06785 39.9147851  11.200558 1.484604e-28
## poly(age, 4)2 -478.31581 39.9147851 -11.983424 2.355831e-32
## poly(age, 4)3  125.52169 39.9147851   3.144742 1.678622e-03
## poly(age, 4)4  -77.91118 39.9147851  -1.951938 5.103865e-02
```

この構文は、`lm()`関数を使用して `wage` を `age` の 4 次多項式 `poly(age, 4)`で推定(予測)する線形モデルをフィットする。`poly()`コマンドを用いると、`age` の累乗を長い式で書く必要がなくない。この関数は、**直交多項式**の基底となる列を持つ行列を返す。つまり、各列は `age`、`age^2`、`age^3`、および `age^4` の線形結合になっている。

しかし、`poly()`に `raw = TRUE` を指定すれば、`age`、`age^2`、`age^3`、および `age^4` を直接取得することも可能となるが、これは `poly()` 関数の `raw = TRUE` 引数を利用すれば実現できる。ただし後で確認するように、これはモデルに大きな影響を与える

わけではない。基底の選択によって係数の推定値は変わる、得られるフィットした値には影響を与えない。

```
fit2 <- lm(wage ~ poly(age, 4, raw = T), data = Wage)
coef(summary(fit2))
```

```
##                             Estimate   Std. Error   t value     Pr(>|t|)
## (Intercept)           -1.841542e+02 6.004038e+01 -3.067172 0.0021802539
## poly(age, 4, raw = T)1  2.124552e+01 5.886748e+00  3.609042 0.0003123618
## poly(age, 4, raw = T)2 -5.638593e-01 2.061083e-01 -2.735743 0.0062606446
## poly(age, 4, raw = T)3  6.810688e-03 3.065931e-03  2.221409 0.0263977518
## poly(age, 4, raw = T)4 -3.203830e-05 1.641359e-05 -1.951938 0.0510386498
```

他にもこのモデルをフィットするいくつかの等価な方法があり、以下でその例を示す：

```
fit2a <- lm(wage ~ age + I(age^2) + I(age^3) + I(age^4),
    data = Wage)
coef(fit2a)
```

```
##   (Intercept)           age      I(age^2)      I(age^3)      I(age^4)
## -1.841542e+02  2.124552e+01 -5.638593e-01  6.810688e-03 -3.203830e-05
```

この方法では、`I()`というラッパー関数を使用して`age^2`のような項を入れて多項式の基底関数を作成する。

```
fit2b <- lm(wage ~ cbind(age, age^2, age^3, age^4),
    data = Wage)
```

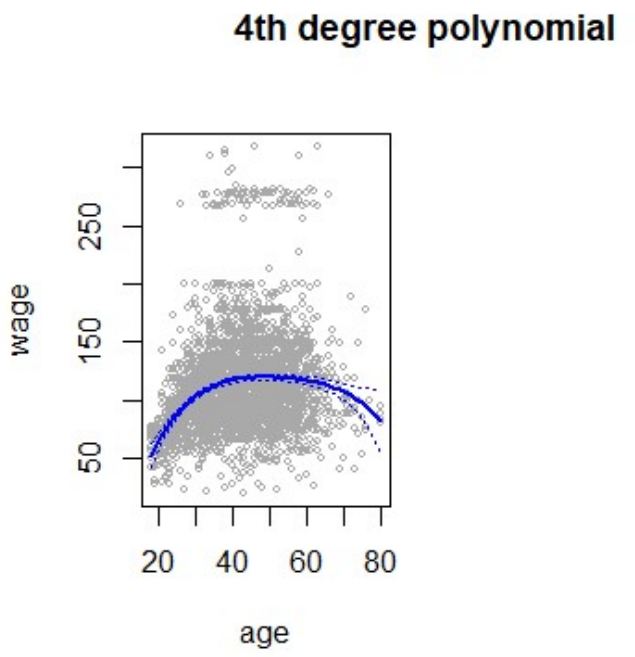
この方法では、`cbind()`関数を使用してベクトルの集合から行列を構築し、同様のことをより簡潔に実現している。

次に、予測したい`age`の値のグリッドを作成し、標準誤差を指定して`predict()`関数を呼び出そう。

```
agelims <- range(age)
age.grid <- seq(from = agelims[1], to = agelims[2])
preds <- predict(fit, newdata = list(age = age.grid),
    se = TRUE)
se.bands <- cbind(preds$fit + 2 * preds$se.fit,
    preds$fit - 2 * preds$se.fit)
```

最後に、データをプロットし、4 次多項式のフィットを追加します。

```
par(mfrow = c(1, 2), mar = c(4.5, 4.5, 1, 1),
    oma = c(0, 0, 4, 0))
plot(age, wage, xlim = agelims, cex = .5, col = "darkgrey")
title("4th degree polynomial", outer = T)
lines(age.grid, preds$fit, lwd = 2, col = "blue")
matlines(age.grid, se.bands, lwd = 1, col = "blue", lty = 3)
```



ここで、par()関数の mar および oma 引数を使用してプロットの余白を制御し、title()関数で両方のサブプロットをまたぐタイトルを作成する。

直交基底関数を生成するか否かはモデルの結果に意味のある影響は与えない。そのことを確認するために、いずれのケースでも得られるフィット値が同一であることを示しておこう。

```
preds2 <- predict(fit2, newdata = list(age = age.grid),
    se = TRUE)
max(abs(preds$fit - preds2$fit))
## [1] 6.842527e-11
```

多項式回帰を行う際には、使用する多項式の次数を決定する必要がある。その方法の一つとして仮説検定を利用してみる。ここでは線形モデルから 5 次の多項式モデルまで順にフィットし、wage と age の関係を説明するのに十分な最も単純なモデルを求めてみる。そのために anova()関数を使用する。anova()関数により帰無仮説をテストするのである。つまり、モデル M_1 がデータを説明するのに十分か否か、より複雑なモデル M_2 が必要か否かを検定する。なお anova()を使用するためには、M_1 と M_2 が入れ子となる（*ネストされた*）モデルである必要がある。

```
fit.1 <- lm(wage ~ age, data = Wage)
fit.2 <- lm(wage ~ poly(age, 2), data = Wage)
fit.3 <- lm(wage ~ poly(age, 3), data = Wage)
fit.4 <- lm(wage ~ poly(age, 4), data = Wage)
fit.5 <- lm(wage ~ poly(age, 5), data = Wage)
anova(fit.1, fit.2, fit.3, fit.4, fit.5)

## Analysis of Variance Table
##
## Model 1: wage ~ age
## Model 2: wage ~ poly(age, 2)
## Model 3: wage ~ poly(age, 3)
## Model 4: wage ~ poly(age, 4)
## Model 5: wage ~ poly(age, 5)
##   Res.Df     RSS Df Sum of Sq        F    Pr(>F)
## 1   2998 5022216
```

```
## 2   2997 4793430  1    228786 143.5931 < 2.2e-16 ***
## 3   2996 4777674  1     15756   9.8888  0.001679 **
## 4   2995 4771604  1      6070   3.8098  0.051046 .
## 5   2994 4770322  1      1283   0.8050  0.369682
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

線形モデル `Model 1` と二次モデル `Model 2` を比較する p 値は本質的にゼロ（$< 10^{-15}$）であり、線形フィットは不十分であることを示している。同様に、二次モデル `Model 2` と三次モデル `Model 3` を比較する p 値は非常に低い（0.0017）ため、二次フィットも不十分となる。三次モデル `Model 3` と四次モデル `Model 4` を比較する p 値は約 5%であり、五次モデル `Model 5` の p 値（0.37）は不要であることを示唆している。したがって、三次または四次多項式がデータに対して妥当なフィットを提供しているようである。

この場合、`anova()`関数を使用する代わりに、`poly()`が直交多項式を生成する特性を利用して、これらの p 値をより簡潔に得ることもできる。

```
coef(summary(fit.5))
```

```
##                Estimate Std. Error     t value     Pr(>|t|)
## (Intercept)    111.70361  0.7287647 153.2780243 0.000000e+00
## poly(age, 5)1  447.06785 39.9160847  11.2001930 1.491111e-28
## poly(age, 5)2 -478.31581 39.9160847 -11.9830341 2.367734e-32
## poly(age, 5)3  125.52169 39.9160847   3.1446392 1.679213e-03
## poly(age, 5)4  -77.91118 39.9160847  -1.9518743 5.104623e-02
## poly(age, 5)5  -35.81289 39.9160847  -0.8972045 3.696820e-01
```

p 値は同じであり、実際にはt統計量の二乗が `anova()`関数の F 統計量と等しいことが分かる。例えば：

```
(-11.983)^2
```

```
## [1] 143.5923
```

となる。ただし、ANOVA 手法は直交多項式を使用しない場合や、他の項をモデルに含む場合でも機能する。例えば、以下のように 3 つのモデルを比較することができる：

```
fit.1 <- lm(wage ~ education + age, data = Wage)
fit.2 <- lm(wage ~ education + poly(age, 2), data = Wage)
fit.3 <- lm(wage ~ education + poly(age, 3), data = Wage)
anova(fit.1, fit.2, fit.3)
```

```
## Analysis of Variance Table
##
## Model 1: wage ~ education + age
## Model 2: wage ~ education + poly(age, 2)
## Model 3: wage ~ education + poly(age, 3)
##   Res.Df     RSS Df Sum of Sq        F Pr(>F)
## 1   2994 3867992
## 2   2993 3725395  1    142597 114.6969 <2e-16 ***
## 3   2992 3719809  1      5587   4.4936 0.0341 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

なお、仮説検定や ANOVA を使用する代わりに、クロスバリデーション（第 5 章で議論）を使用して多項式の次数を選択することもできる。

次に、個人が年収 250,000 ドル以上を稼ぐかどうかを予測する課題を検討します。まず、適切な応答ベクトルを作成し、次に `family = "binomial"`を指定して `glm()`関数を使用し、多項式ロジスティック回帰モデルをフィットしてみる。

```
fit <- glm(I(wage > 250) ~ poly(age, 4), data = Wage,
    family = binomial)
```

ここで再び、ラッパー関数 `I()`を使用して、この 2 値応答変数をその場で作成しておく。式 `wage > 250`は `TRUE` と `FALSE` を含む論理変数に評価され、`glm()`によって `TRUE` は 1、`FALSE` は 0 に変換される。

次に、`predict()`関数を使用して予測を行う。

```
preds <- predict(fit, newdata = list(age = age.grid), se = T)
```

なお、信頼区間を計算することは線形回帰の場合より少し複雑になる。デフォルトの予測タイプは`type = "link"`であり、ここで使用されている。これは*ロジット*または対数オッズの予測値を返している。`glm()`モデルのデフォルトの予測タイプは`type = "link"`であり、ここでもそれを使用している。これは*ロジット*（log-odds）の予測値が得られることを意味している。つまり次の形のモデルをフィットしているのである。

$$\log\left(\frac{\Pr(Y=1|X)}{1-\Pr(Y=1|X)}\right) = X\beta$$

ここで得られる予測値は$X\hat{\beta}$の形をしており、標準誤差も同様に$X\hat{\beta}$に対するものになっている。

そこで$\Pr(Y=1|X)$の信頼区間を得るためには、以下の変換を用いる。

$$\Pr(Y=1|X) = \frac{\exp(X\beta)}{1+\exp(X\beta)}.$$

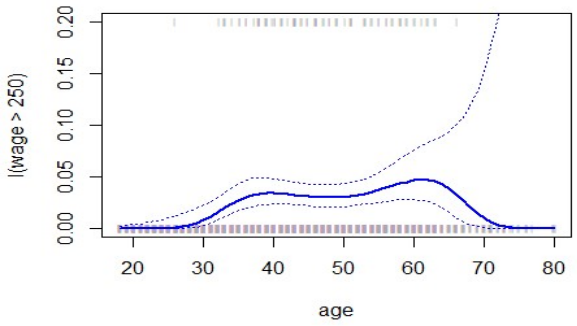
```
pfit <- exp(preds$fit) / (1 + exp(preds$fit))
se.bands.logit <- cbind(preds$fit + 2 * preds$se.fit,
    preds$fit - 2 * preds$se.fit)
se.bands <- exp(se.bands.logit) / (1 + exp(se.bands.logit))
```

なお、直接に確率を計算するために、`predict()`関数の`type = "response"`オプションを選択することも可能である。

```
preds <- predict(fit, newdata = list(age = age.grid),
    type = "response", se = T)
```

ただし、対応する信頼区間は負の確率を得る可能性があるため、理にかなったものとは言えないだろう。図 7.1 の右側のプロットは以下のコードで作成した。

```
plot(age, I(wage > 250), xlim = agelims, type = "n",
    ylim = c(0, .2))
points(jitter(age), I((wage > 250) / 5), cex = .5, pch = "|", col = "darkgrey")
lines(age.grid, pfit, lwd = 2, col = "blue")
matlines(age.grid, se.bands, lwd = 1, col = "blue", lty = 3)
```



ここでは、`wage > 250`に対応する`age`値をプロットの上部に灰色のマークで描画し、250 ドル以下の`wage`値を持つものはプロットの下部に灰色のマークで描画している。ただし同じ`age`値を持つ観測値が重ならないように、`jitter()`関数を使用して`age`値をわずかにランダムにずらしている。これはしばしば*ラグプロット*と呼ばれている。

7.2 節で説明したようにステップ関数をフィットするには`cut()`関数を使用する。

```
table(cut(age, 4))
##
## (17.9,33.5]   (33.5,49]   (49,64.5] (64.5,80.1]
##         750        1399         779          72
fit <- lm(wage ~ cut(age, 4), data = Wage)
coef(summary(fit))
##                         Estimate Std. Error   t value     Pr(>|t|)
## (Intercept)            94.158392   1.476069 63.789970 0.000000e+00
## cut(age, 4)(33.5,49]   24.053491   1.829431 13.148074 1.982315e-38
## cut(age, 4)(49,64.5]   23.664559   2.067958 11.443444 1.040750e-29
## cut(age, 4)(64.5,80.1]  7.640592   4.987424  1.531972 1.256350e-01
```

ここで、`cut()`は自動的にカットポイントを 33.5 歳、49 歳、64.5 歳に設定したが、`breaks`オプションを使用して独自のカットポイントを直接に指定することもできる。

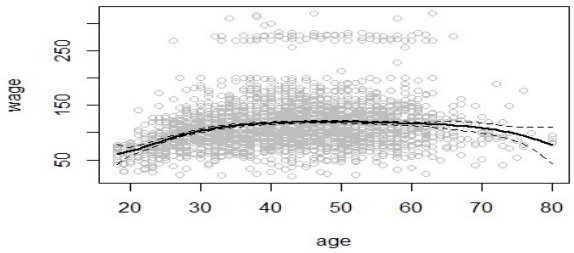
`cut()`関数は順序付きカテゴリ変数を返し、`lm()`関数は回帰で使用するダミー変数を生成する。`age < 33.5`カテゴリは省略されているため、切片係数（94,160）は

33.5 歳未満の平均給与を表し、他の係数は他の年齢グループの平均追加給与を表している。多項式フィットの場合と同様に予測やプロットを生成することができる。

スプライン

R で回帰スプラインをフィットするには、splines ライブラリを使用する。7.4 節で見たように、回帰スプラインは適切な基底関数の行列を構築することでフィットすることができます。bs() 関数は、指定されたノットのセットに基づいて、スプラインの基底関数の行列全体を生成する。デフォルト選択では三次スプラインが作成される。回帰スプラインを用いて wage を age にフィットするのは簡単である：

```
library(splines)
fit <- lm(wage ~ bs(age, knots = c(25, 40, 60)), data = Wage)
pred <- predict(fit, newdata = list(age = age.grid), se = T)
plot(age, wage, col = "gray")
lines(age.grid, pred$fit, lwd = 2)
lines(age.grid, pred$fit + 2 * pred$se, lty = "dashed")
lines(age.grid, pred$fit - 2 * pred$se, lty = "dashed")
```



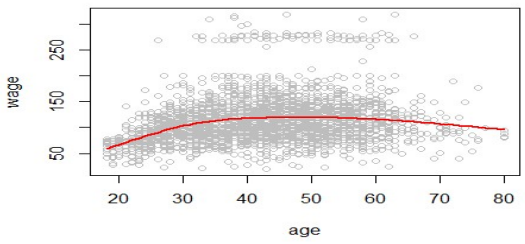
ここでは、ノットを 25 歳、40 歳、60 歳に事前指定した。この設定により 6 つの基底関数を持つスプラインが生成される（3 つのノットを持つ三次スプラインは 7 自由度を持つことを思い出してみよう）。また、df オプションを使用して、データの均一分位点でノットを配置するスプラインを生成することも可能である。

```
dim(bs(age, knots = c(25, 40, 60)))
## [1] 3000    6
dim(bs(age, df = 6))
## [1] 3000    6
attr(bs(age, df = 6), "knots")
## [1] 33.75 42.00 51.00
```

ここでは R は 33.8 歳、42.0 歳、51.0 歳の位置にノットを設定したが age の 25%、50%、75%分位点に対応している。また、bs()関数の degree 引数を使用して、デフォルトの 3 次スプライン以外の任意の次数のスプラインをフィットすることも可能である。

自然スプラインをフィットするには、ns()関数を使用する。ここでは 4 次の自然スプラインをフィットしてみよう。

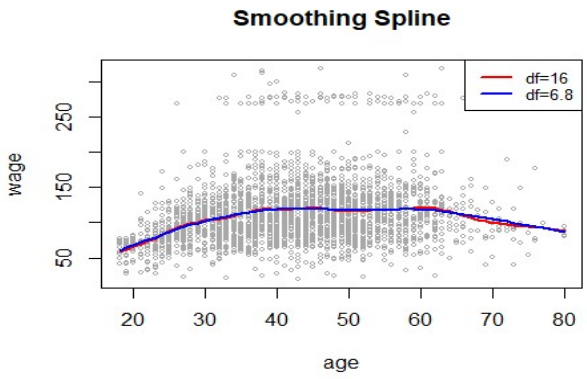
```
fit2 <- lm(wage ~ ns(age, df = 4), data = Wage)
pred2 <- predict(fit2, newdata = list(age = age.grid),
     se = T)
plot(age, wage, col = "gray")
lines(age.grid, pred2$fit, col = "red", lwd = 2)
```



ここでも、bs()関数と同様に、knots オプションを使用して結節点(ノット)を直接指定することもできる。平滑化(スムージング)スプラインをフィットするには、smooth.spline()関数を使用すればよいが、図 7.8 は以下のコードで作成した：

```
plot(age, wage, xlim = agelims, cex = .5, col = "darkgrey")
title("Smoothing Spline")
```

```
fit <- smooth.spline(age, wage, df = 16)
fit2 <- smooth.spline(age, wage, cv = TRUE)
## Warning in smooth.spline(age, wage, cv = TRUE): 一意でない 'x'
## 値によるクロスバリデーションには問題があるかも知れません
fit2$df
## [1] 6.794596
lines(fit, col = "red", lwd = 2)
lines(fit2, col = "blue", lwd = 2)
legend("topright", legend = c("df=16", "df=6.8"),
    col = c("red", "blue"), lty = 1, lwd = 2, cex = .8)
```

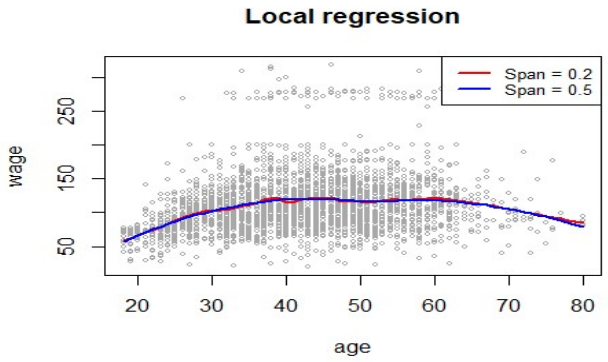


最初の`smooth.spline()`呼び出しでは`df = 16`を指定しているが関数は 16 自由度に対応するλの値を指定している。2 回目の`smooth.spline()`呼び出しでは、クロスバリデーションを使用して滑らかさレベルを選択、その結果、6.8 自由度に対応するλの値が得られる。

局所回帰を実行するには、`loess()`関数を使用する。

```
plot(age, wage, xlim = agelims, cex = .5, col = "darkgrey")
title("Local regression")
fit <- loess(wage ~ age, span = .2, data = Wage)
fit2 <- loess(wage ~ age, span = .5, data = Wage)
lines(age.grid, predict(fit, data.frame(age = age.grid)),
```

```
    col = "red", lwd = 2)
lines(age.grid, predict(fit2, data.frame(age = age.grid)),
    col = "blue", lwd = 2)
legend("topright", legend = c("Span = 0.2", "Span = 0.5"),
    col = c("red", "blue"), lty = 1, lwd = 2, cex = .8)
```



ここではスパン0.2および0.5を使用して局所線形回帰を実行した。これは、それぞれの近傍が観測値の 20%または 50%で構成されることを意味するが、スパンが大きいほど、フィットが滑らかになる。なお `locfit` ライブラリも局所回帰モデルをフィットするために利用できる。

一般化加法モデル（GAM）

次に、GAM を使用して `year` および `age` の自然スプライン関数で `wage` を予測し、`education` を質的変数として扱おう。以下のコードは式(7.16)を反映しているが、。このモデルでは、適切な基底関数を選択することで、単なる大規模な線形回帰モデルとしてフィットすることができる。

```
gam1 <- lm(wage ~ ns(year, 4) + ns(age, 5) + education,
    data = Wage)
```

次に、スムージングスプラインを使用して式(7.16)のモデルをフィットしよう。なお、より一般的な種類の GAM をフィットするには、`gam` ライブラリを使用するとよい。

`s()` 関数は `gam` ライブラリの一部であり、スムージングスプラインを使用することを示すために用いらる。ここでは、`lyear` の関数が **4 自由度**、`age` の関数が **5 自由度** を持つように指定している。`education` は質的変数であるため、そのままの形を利用し、4 つのダミー変数に変換している。これらの成分を用いて **GAM (Generalized Additive Model)** をフィットするために `gam()` 関数を使用しよう。すべての項は (7.16) 式のように同時にフィットし、それぞれが他の変数の影響を考慮しながら応答変数を説明する形になる。

```
library(gam)
```

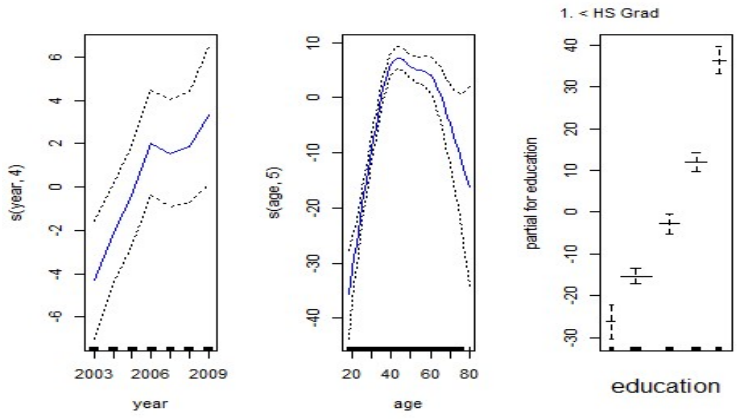
```
## 要求されたパッケージ foreach をロード中です
```

```
## Loaded gam 1.22-5
```

```
gam.m3 <- gam(wage ~ s(year, 4) + s(age, 5) + education,
    data = Wage)
```

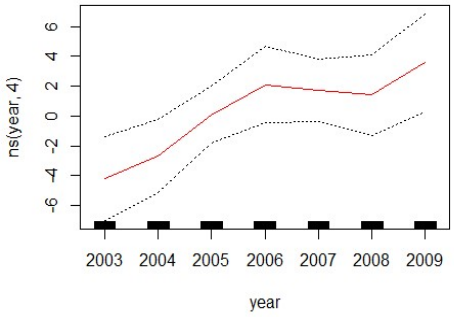
図 7.12 を作成するには単に `plot()`関数を呼び出せばよい。

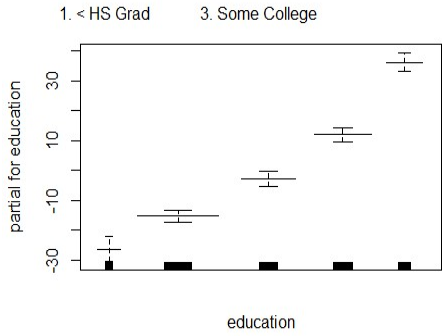
```
par(mfrow = c(1, 3))
plot(gam.m3, se = TRUE, col = "blue")
```

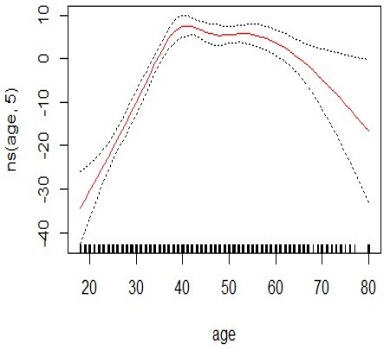


`plot()`関数は `gam.m3` が `Gam` クラスのオブジェクトであることを認識し、適切な `plot.Gam()`メソッドを呼び出す。同様に、`gam1` が `Gam` クラスではなく `lm` クラスである場合でも、`plot.Gam()`は使用できる。図 7.11 は以下のコマンドで作成した。

```
plot.Gam(gam1, se = TRUE, col = "red")
```







これらのプロットにより、`year`の関数がほぼ線形であることが分かる。次に、以下の 3 つのモデルの中でどれが最適かを決定するために、一連の ANOVA 検定を行おう： 1. `year`を含まない GAM (M_1)。 2. `year`の線形関数を用いる GAM (M_2)。 3. `year`のスプライン関数を用いる GAM (M_3)。

```
gam.m1 <- gam(wage ~ s(age, 5) + education, data = Wage)
gam.m2 <- gam(wage ~ year + s(age, 5) + education,
    data = Wage)
anova(gam.m1, gam.m2, gam.m3, test = "F")
## Analysis of Deviance Table
##
```

```
## Model 1: wage ~ s(age, 5) + education
## Model 2: wage ~ year + s(age, 5) + education
## Model 3: wage ~ s(year, 4) + s(age, 5) + education
##   Resid. Df Resid. Dev Df Deviance       F    Pr(>F)
## 1      2990    3711731
## 2      2989    3693842  1  17889.2 14.4771 0.0001447 ***
## 3      2986    3689770  3   4071.1  1.0982 0.3485661
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

結果として、`year`を含まないGAMよりも`year`の線形関数を使用するGAMの方が優れているという強い証拠が得られた（p値 = 0.00014）。ただし、`year`の非線形関数が必要であるという証拠は特にあるわけではない（p値 = 0.349）。したがって、ANOVAの結果に基づくと、M_2が最適になった。

ここで`summary()`関数を使用して、GAMのフィットの要約を出力しよう。

```
summary(gam.m3)
```

```
##
## Call: gam(formula = wage ~ s(year, 4) + s(age, 5) + education, data = Wage)
## Deviance Residuals:
##     Min      1Q  Median      3Q     Max
## -119.43  -19.70   -3.33   14.17  213.48
##
## (Dispersion Parameter for gaussian family taken to be 1235.69)
##
##     Null Deviance: 5222086 on 2999 degrees of freedom
## Residual Deviance: 3689770 on 2986 degrees of freedom
## AIC: 29887.75
##
## Number of Local Scoring Iterations: NA
##
## Anova for Parametric Effects
##              Df  Sum Sq Mean Sq F value    Pr(>F)
## s(year, 4)    1   27162   27162  21.981 2.877e-06 ***
```

```
## s(age, 5)     1  195338  195338 158.081 < 2.2e-16 ***
## education     4 1069726  267432 216.423 < 2.2e-16 ***
## Residuals  2986 3689770    1236
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Anova for Nonparametric Effects
##             Npar Df Npar F  Pr(F)
## (Intercept)
## s(year, 4)        3  1.086 0.3537
## s(age, 5)         4 32.380 <2e-16 ***
## education
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

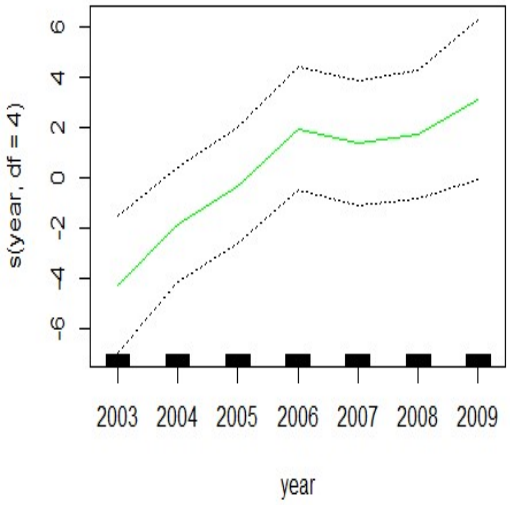
「**Anova for Parametric Effects**」 の p 値は、`year`、`age`、`education` のすべてが統計的に有意であることを明確に示している。ただし、これは変数が線形関係のみを仮定した場合でも有意であることを意味している。一方、「**Anova for Nonparametric Effects**」 の p 値は、`year` と `age` に関して、「**線形関係の帰無仮説**」と 「**非線形関係の対立仮説**」 を比較するものである。`year` の p 値が大きいことは、ANOVA テストの結果と一致しており、この変数には線形関数で十分であることを示唆している。一方で、`age` の p 値は非常に小さく、非線形項が必要であるという強い証拠を示している。

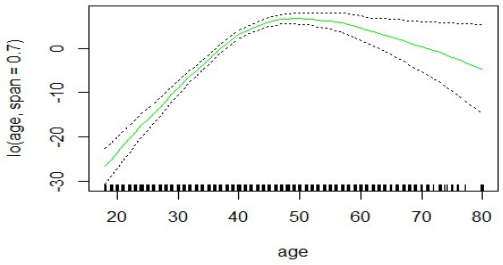
`Gam` クラスの `predict()`法を使用して予測を生成できる。以下はトレーニング・セット上で予測を行う例である：

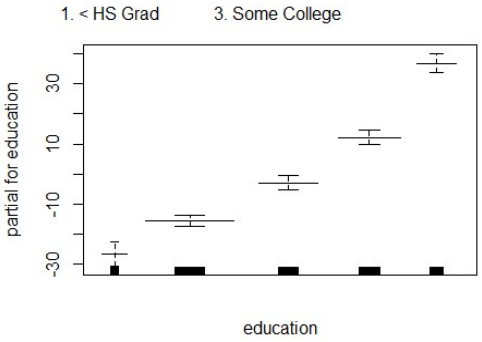
```
preds <- predict(gam.m2, newdata = Wage)
```

`lo()`関数を使用して、局所回帰を GAM の構成要素として使用することも可能である。以下は、`age` 項に局所回帰（スパン 0.7）を使用した例である：

```
gam.lo <- gam(
    wage ~ s(year, df = 4) + lo(age, span = 0.7) + education,
    data = Wage
  )
plot(gam.lo, se = TRUE, col = "green")
```







また、`lo()`関数を使用して、`gam()`関数を呼び出す前に交互作用を作成することもできる。例えば、以下のコードは`year`と`age`間の交互作用を局所回帰曲面としてモデル化してみよう。

```
gam.lo.i <- gam(wage ~ lo(year, age, span = 0.5) + education,
    data = Wage)


## Warning in lo.wam(x, z, wz, fit$smooth, which, fit$smooth.frame, bf.maxit, :
## liv too small.  (Discovered by lowesd)
```

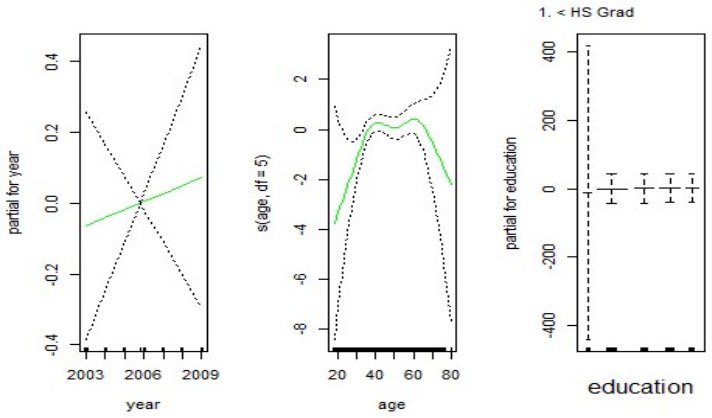
```
## Warning in lo.wam(x, z, wz, fit$smooth, which, fit$smooth.frame, bf.maxit, : lv
## too small.  (Discovered by lowesd)
## Warning in lo.wam(x, z, wz, fit$smooth, which, fit$smooth.frame, bf.maxit, :
## liv too small.  (Discovered by lowesd)
## Warning in lo.wam(x, z, wz, fit$smooth, which, fit$smooth.frame, bf.maxit, : lv
## too small.  (Discovered by lowesd)
```

なお、この結果の 2 次元曲面をプロットするには、次のように `akima` パッケージをインストールする必要がある。

```
library(akima)
plot(gam.lo.i)
## Error in gplot.matrix(x, y, se.y, xlab, ylab, residuals, rugplot, scale, : Y
ou need to install the package 'interp' from the R contributed libraries to use
 this plotting method for bivariate functions
```

ロジスティック回帰 GAM をフィットするには、2 値応答変数を作成する際に再び `I()`関数を使用し、`family=binomial` を指定する。

```
gam.lr <- gam(
    I(wage > 250) ~ year + s(age, df = 5) + education,
    family = binomial, data = Wage
  )
par(mfrow = c(1, 3))
plot(gam.lr, se = T, col = "green")
```

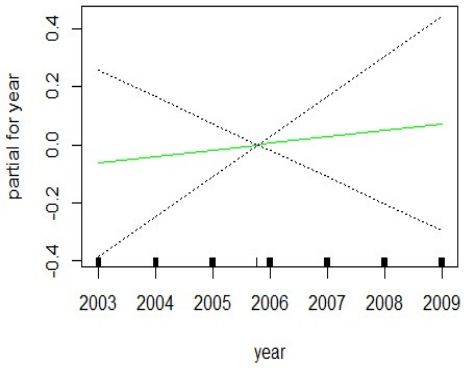


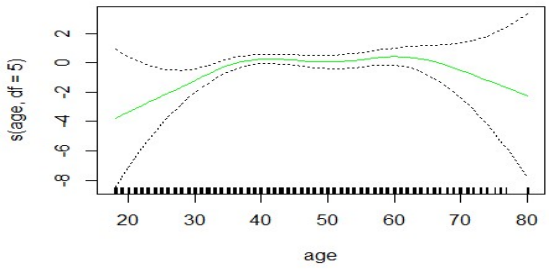
図より< HS カテゴリには高収入者がいないことが容易に確認できる。

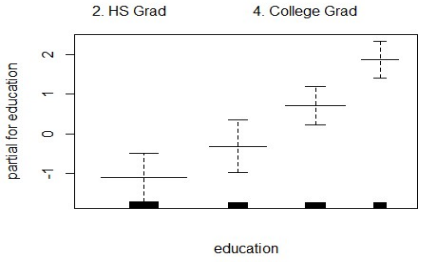
```
table(education, I(wage > 250))
##
## education             FALSE TRUE
##   1. < HS Grad          268    0
##   2. HS Grad            966    5
##   3. Some College       643    7
##   4. College Grad       663   22
##   5. Advanced Degree    381   45
```

したがって、このカテゴリを除外してロジスティック回帰 GAM をフィットしてみる。この方がより理にかなった結果が得られる。

```
gam.lr.s <- gam(
    I(wage > 250) ~ year + s(age, df = 5) + education,
    family = binomial, data = Wage,
    subset = (education != "1. < HS Grad")
  )
plot(gam.lr.s, se = T, col = "green")
```


partial for year
-0.4
-0.2
0.0
0.2
0.4
2003
2004
2005
2006
2007
2008
2009
year


s(age, df = 5)
-8
-6
-4
-2
0
2
20
30
40
50
60
70
80
age


2. HS Grad
4. College Grad
partial for education
-1
0
1
2
education

ISLR 第 8 章 決定木（Decision Trees）

分類木のフィッティング

`tree` ライブラリを使用して、分類木と回帰木を構築する。

```
library(tree)
```

まず、分類木を使用して `Carseats` データセットを分析しよう。このデータでは、`Sales` は連続変数であるため、まずこれを二値変数に変換する。`ifelse()`関数を使用して、`Sales` が8を超える場合に `Yes`、それ以外の場合に `No` となる変数 `High` を作成する。

```
library(ISLR2)
attach(Carseats)
High <- factor(ifelse(Sales <= 8, "No", "Yes"))
```

次に `data.frame()`関数を使用して、`High` を `Carseats` データの他の変数と統合する。

```
Carseats <- data.frame(Carseats, High)
```

ここで `tree()`関数を使用して分類木をフィッティングしよう。目的変数は `High` であり、`Sales` 以外のすべての変数を説明変数として使用する。`tree()`関数の構文は `lm()`関数と非常に似ている

```
tree.carseats <- tree(High ~ . - Sales, Carseats)
```

`summary()`関数を使用すると、ツリーの内部ノードとして使用される変数、終端ノードの数、および（学習データにおける）誤分類率が表示される。

```
summary(tree.carseats)
```

```
##
## Classification tree:
## tree(formula = High ~ . - Sales, data = Carseats)
## Variables actually used in tree construction:
## [1] "ShelveLoc"   "Price"       "Income"      "CompPrice"   "Population"
## [6] "Advertising" "Age"         "US"
## Number of terminal nodes:  27
## Residual mean deviance:  0.4575 = 170.7 / 373
## Misclassification error rate: 0.09 = 36 / 400
```

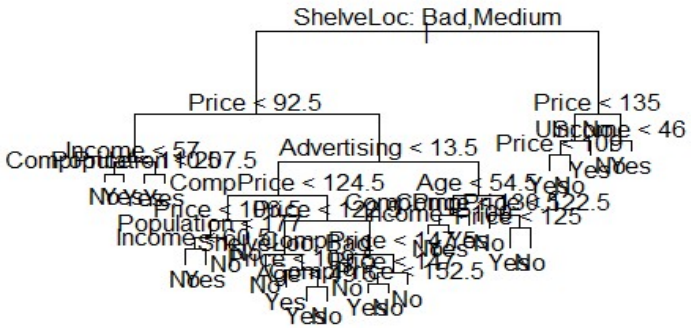
この出力から、学習誤分類率が 9% であることが分かる。分類木の場合には、`summary()`の出力に含まれる `deviance` は以下の式で表される：

$$-2\sum_{m}\sum_{k} n_{mk}\log\hat{p}_{mk}$$

ここで、n_{mk} はm番目の終端ノードに属するk番目のクラスの観測数であるが、この形は(8.7) で定義されたエントロピーと密接に関連している。`deviance` が小さいほど、学習データへのフィットが良いことを示す。また、*残差平均逸脱度（residual mean deviance）*は、`deviance` を$n - |T_0|$で割った値である。この場合には$400 - 27 = 373$となる。

ツリーの最も魅力的な特徴の一つは、グラフィカルに表示できることである。`plot()`関数を使用してツリー構造を表示し、`text()`関数を使用してノードラベルを表示する。ここで引数 `pretty = 0` を指定すると、カテゴリカル変数のカテゴリ名がそのまま表示される。

```
plot(tree.carseats)
text(tree.carseats, pretty = 0)
```



`Sales`を最もよく示す指標は棚の配置（ShelveLoc）であることが分かるす。最初の分岐で`Good`の場所と`Bad`・`Medium`の場所が分けられている。

ツリーオブジェクトの名前を入力すると、各分岐に対応する出力が表示される。`R`は、分岐条件（例：`Price < 92.5`）、その分岐の観測数、逸脱度（deviance）、その分岐の全体的な予測（`Yes`または`No`）、およびその分岐に属する`Yes`と`No`の割合を表示している。終端ノードにつながる分岐はアスタリスク（`*`）で示されている。

```
tree.carseats
```

```
## node), split, n, deviance, yval, (yprob)
##       * denotes terminal node
##
##   1) root 400 541.500 No ( 0.59000 0.41000 )
##     2) ShelveLoc: Bad,Medium 315 390.600 No ( 0.68889 0.31111 )
##       4) Price < 92.5 46  56.530 Yes ( 0.30435 0.69565 )
##         8) Income < 57 10  12.220 No ( 0.70000 0.30000 )
##          16) CompPrice < 110.5 5   0.000 No ( 1.00000 0.00000 ) *
##          17) CompPrice > 110.5 5   6.730 Yes ( 0.40000 0.60000 ) *
##         9) Income > 57 36  35.470 Yes ( 0.19444 0.80556 )
##          18) Population < 207.5 16  21.170 Yes ( 0.37500 0.62500 ) *
##          19) Population > 207.5 20   7.941 Yes ( 0.05000 0.95000 ) *
```

```
##       5) Price > 92.5 269 299.800 No ( 0.75465 0.24535 )
##        10) Advertising < 13.5 224 213.200 No ( 0.81696 0.18304 )
##          20) CompPrice < 124.5 96  44.890 No ( 0.93750 0.06250 )
##            40) Price < 106.5 38  33.150 No ( 0.84211 0.15789 )
##              80) Population < 177 12  16.300 No ( 0.58333 0.41667 )
##               160) Income < 60.5 6   0.000 No ( 1.00000 0.00000 ) *
##               161) Income > 60.5 6   5.407 Yes ( 0.16667 0.83333 ) *
##              81) Population > 177 26   8.477 No ( 0.96154 0.03846 ) *
##            41) Price > 106.5 58   0.000 No ( 1.00000 0.00000 ) *
##          21) CompPrice > 124.5 128 150.200 No ( 0.72656 0.27344 )
##            42) Price < 122.5 51  70.680 Yes ( 0.49020 0.50980 )
##              84) ShelveLoc: Bad 11   6.702 No ( 0.90909 0.09091 ) *
##              85) ShelveLoc: Medium 40  52.930 Yes ( 0.37500 0.62500 )
##               170) Price < 109.5 16   7.481 Yes ( 0.06250 0.93750 ) *
##               171) Price > 109.5 24  32.600 No ( 0.58333 0.41667 )
##                 342) Age < 49.5 13  16.050 Yes ( 0.30769 0.69231 ) *
##                 343) Age > 49.5 11   6.702 No ( 0.90909 0.09091 ) *
##            43) Price > 122.5 77  55.540 No ( 0.88312 0.11688 )
##              86) CompPrice < 147.5 58  17.400 No ( 0.96552 0.03448 ) *
##              87) CompPrice > 147.5 19  25.010 No ( 0.63158 0.36842 )
##               174) Price < 147 12  16.300 Yes ( 0.41667 0.58333 )
##                 348) CompPrice < 152.5 7   5.742 Yes ( 0.14286 0.85714 ) *
##                 349) CompPrice > 152.5 5   5.004 No ( 0.80000 0.20000 ) *
##               175) Price > 147 7   0.000 No ( 1.00000 0.00000 ) *
##        11) Advertising > 13.5 45  61.830 Yes ( 0.44444 0.55556 )
##          22) Age < 54.5 25  25.020 Yes ( 0.20000 0.80000 )
##            44) CompPrice < 130.5 14  18.250 Yes ( 0.35714 0.64286 )
##              88) Income < 100 9  12.370 No ( 0.55556 0.44444 ) *
##              89) Income > 100 5   0.000 Yes ( 0.00000 1.00000 ) *
##            45) CompPrice > 130.5 11   0.000 Yes ( 0.00000 1.00000 ) *
##          23) Age > 54.5 20  22.490 No ( 0.75000 0.25000 )
##            46) CompPrice < 122.5 10   0.000 No ( 1.00000 0.00000 ) *
##            47) CompPrice > 122.5 10  13.860 No ( 0.50000 0.50000 )
##              94) Price < 125 5   0.000 Yes ( 0.00000 1.00000 ) *
##              95) Price > 125 5   0.000 No ( 1.00000 0.00000 ) *
##     3) ShelveLoc: Good 85  90.330 Yes ( 0.22353 0.77647 )
##       6) Price < 135 68  49.260 Yes ( 0.11765 0.88235 )
##        12) US: No 17  22.070 Yes ( 0.35294 0.64706 )
```

```
##          24) Price < 109 8   0.000 Yes ( 0.00000 1.00000 ) *
##          25) Price > 109 9  11.460 No ( 0.66667 0.33333 ) *
##        13) US: Yes 51  16.880 Yes ( 0.03922 0.96078 ) *
##       7) Price > 135 17  22.070 No ( 0.64706 0.35294 )
##        14) Income < 46 6   0.000 No ( 1.00000 0.00000 ) *
##        15) Income > 46 11  15.160 Yes ( 0.45455 0.54545 ) *
```

分類木の性能を適切に評価するために、学習誤差だけでなくテスト誤差も推定する必要がある。そこで、データを学習データとテストデータに分割し、学習データを使ってツリーを構築し、テストデータでその性能を評価しよう。分類木の場合、`predict()`関数の引数`type = "class"`を指定すると、実際のクラス予測を返す。この方法ではテストデータの約 77% の観測値を正しく分類できる。

```
set.seed(2)
train <- sample(1:nrow(Carseats), 200)
Carseats.test <- Carseats[-train, ]
High.test <- High[-train]
tree.carseats <- tree(High ~ . - Sales, Carseats,
    subset = train)
tree.pred <- predict(tree.carseats, Carseats.test,
    type = "class")
table(tree.pred, High.test)
##          High.test
## tree.pred  No Yes
##       No  104  33
##       Yes  13  50
(104 + 50) / 200
## [1] 0.77
```

（なお、`predict()`関数を再実行すると、学習データの終端ノード内で`Yes`と`No`の割合が均等な場合、結果が少し変わる可能性がある。）

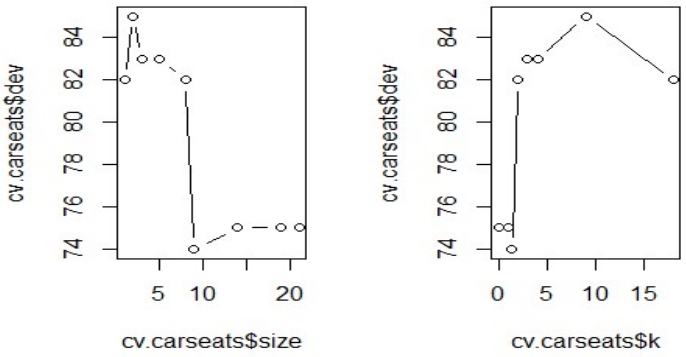
次に、ツリーの剪定が予測精度を向上させるかどうかを検討しよう。`cv.tree()`関数は、交差検証を実行して最適なツリーの複雑さを決定する。コスト複雑度剪定（cost complexity pruning）を使用して、考慮すべきツリーの系列を選択しよう。

ここで引数 `FUN = prune.misclass` を指定すると、交差検証と剪定プロセスを分類誤差率に基づいて行うようになる（デフォルトでは逸脱度 `deviance` が使用される）。

```
set.seed(7)
cv.carseats <- cv.tree(tree.carseats, FUN = prune.misclass)
names(cv.carseats)
## [1] "size"   "dev"    "k"      "method"
cv.carseats
## $size
## [1] 21 19 14  9  8  5  3  2  1
##
## $dev
## [1] 75 75 75 74 82 83 83 85 82
##
## $k
## [1] -Inf  0.0  1.0  1.4  2.0  3.0  4.0  9.0 18.0
##
## $method
## [1] "misclass"
##
## attr(,"class")
## [1] "prune"         "tree.sequence"
```

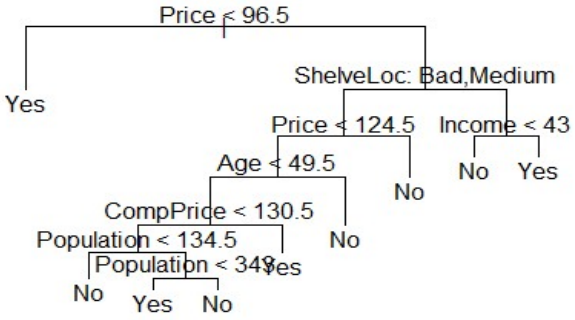
`cv.tree()`関数の出力では、各ツリーの終端ノード数（`size`）、対応する誤分類率、およびコスト複雑度パラメータ（α）が報告される。9 つの終端ノードを持つツリーでは、交差検証誤差が 74 に減少している。以下のコードで誤分類率を `size` および `k` の関数としてプロットできる。

```
par(mfrow = c(1, 2))
plot(cv.carseats$size, cv.carseats$dev, type = "b")
plot(cv.carseats$k, cv.carseats$dev, type = "b")
```



`prune.misclass()`関数を適用して、9 ノードのツリーに剪定しよう。

```
prune.carseats <- prune.misclass(tree.carseats, best = 9)
plot(prune.carseats)
text(prune.carseats, pretty = 0)
```



剪定後のツリーのテスト・データに対する性能を評価してみる。

```
tree.pred <- predict(prune.carseats, Carseats.test,
    type = "class")
table(tree.pred, High.test)
##          High.test
## tree.pred No Yes
##       No  97  25
##       Yes 20  58
(97 + 58) / 200
## [1] 0.775
```

結果として、77.5% のテスト観測値が正しく分類された。剪定によってツリーの解釈性が向上し、分類精度も若干向上したのである。

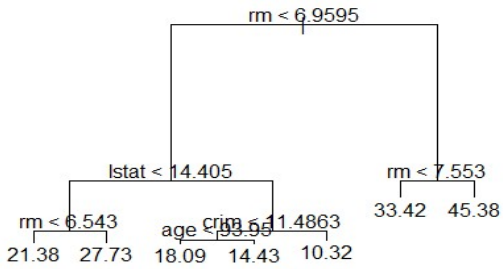
回帰木のフィッティング

次に、`Boston` データセットに回帰木をフィットしてみよう。まず、学習データを作成し、そのデータを使ってツリーを構築する。

```
set.seed(1)
train <- sample(1:nrow(Boston), nrow(Boston) / 2)
tree.boston <- tree(medv ~ ., Boston, subset = train)
summary(tree.boston)
##
## Regression tree:
## tree(formula = medv ~ ., data = Boston, subset = train)
## Variables actually used in tree construction:
## [1] "rm"    "lstat" "crim"  "age"
## Number of terminal nodes:  7
## Residual mean deviance:  10.38 = 2555 / 246
## Distribution of residuals:
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -10.1800  -1.7770  -0.1775   0.0000   1.9230  16.5800
```

`summary()`の出力から、ツリー構築に使用された変数が 4 つだけであることが分かる。ここで、回帰木の `deviance` は、ツリーの残差平方和（sum of squared errors）を表します。次に、ツリーをプロットしてみよう。

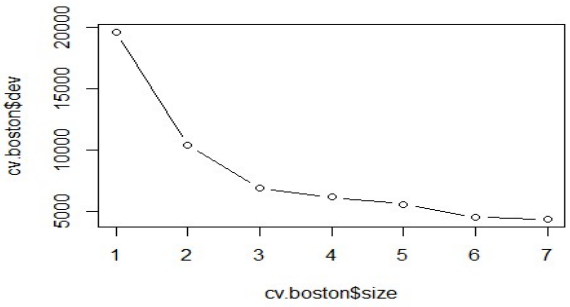
```
plot(tree.boston)
text(tree.boston, pretty = 0)
```



`lstat` 変数は、低所得者層の割合を測定し、`rm` 変数は平均部屋数を示します。このツリーは、`rm` が大きい、または `lstat` が低い場合に、住宅価格が高いことを示しています。例えば、`rm >= 7.553` の地域では、中央値の住宅価格は45,400と予測されます。

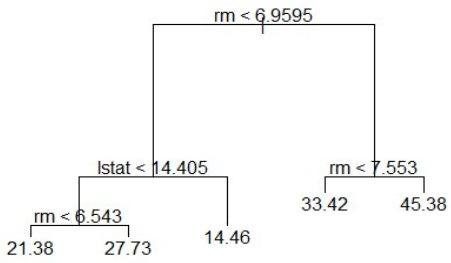
より大きなツリーをフィットすることも可能である。その場合、`control = tree.control(nobs = length(train), mindev = 0)` を `tree()`関数に渡す。次に、ツリーの剪定がパフォーマンスを向上させるかどうかを確認するため、`cv.tree()`関数を使用しよう。

```
cv.boston <- cv.tree(tree.boston)
plot(cv.boston$size, cv.boston$dev, type = "b")
```



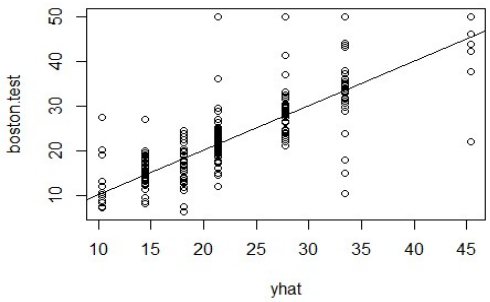
このケースでは、交差検証によって選択されたツリーのサイズは最大の所となった。ただし、さらにツリーを剪定したい場合には、`prune.tree()`関数を使用する。

```
prune.boston <- prune.tree(tree.boston, best = 5)
plot(prune.boston)
text(prune.boston, pretty = 0)
```



交差検証の結果に従い、剪定しないツリーを使用してテストデータの予測を行ってみる。

```
yhat <- predict(tree.boston, newdata = Boston[-train, ])
boston.test <- Boston[-train, "medv"]
plot(yhat, boston.test)
abline(0, 1)
```




```
mean((yhat - boston.test)^2)
```

```
## [1] 35.28688
```

この回帰木のテストデータにおける MSE（平均二乗誤差）は35.29となる。MSE の平方根は5.941であり、このモデルのテストデータの予測値が、中央値の住宅価格から平均5,941ドル程度の誤差を持つことを示している。

バギングとランダムフォレスト

ここでは、`randomForest` パッケージを使用して、`Boston` データに対してバギングとランダムフォレストをフィットしよう。

```
library(randomForest)
```

```
## randomForest 4.7-1.2
```

```
## Type rfNews() to see new features/changes/bug fixes.
```

```
set.seed(1)
bag.boston <- randomForest(medv ~ ., data = Boston,
    subset = train, mtry = 12, importance = TRUE)
bag.boston
```

```
##
```

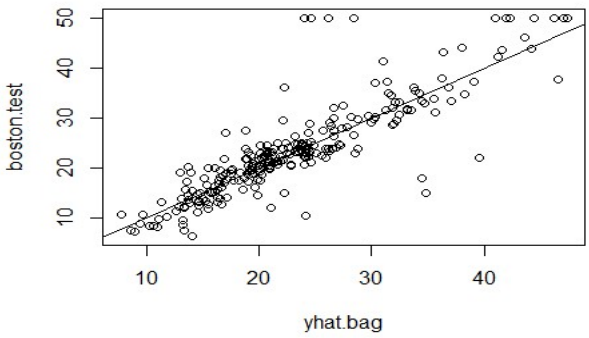
```
## Call:
```

```
##  randomForest(formula = medv ~ ., data = Boston, mtry = 12, importance = TRU
```

```
E,      subset = train)
##               Type of random forest: regression
##                     Number of trees: 500
## No. of variables tried at each split: 12
##
##           Mean of squared residuals: 11.40162
##                     % Var explained: 85.17
```

引数 `mtry = 12` を指定すると、全 12 個の説明変数が各分岐で考慮されるため、これはバギングになる。そこでテスト・データでの性能を確認しよう。

```
yhat.bag <- predict(bag.boston, newdata = Boston[-train, ])
plot(yhat.bag, boston.test)
abline(0, 1)
```




```
mean((yhat.bag - boston.test)^2)
## [1] 23.41916
```

バギングによる回帰木のテスト MSE は23.42であり、最適に剪定された単一の回帰木の MSE よりも約 3 分の 2 に低減している。また ntree 引数を使用すると、成長させるツリーの数を変更できる。

```
bag.boston <- randomForest(medv ~ ., data = Boston,
    subset = train, mtry = 12, ntree = 25)
yhat.bag <- predict(bag.boston, newdata = Boston[-train, ])
mean((yhat.bag - boston.test)^2)
## [1] 25.75055
```

ランダムフォレストを実行するには、`mtry`の値を小さくするだけでよい。デフォルト設定では、回帰木では$p/3$、分類木では$\sqrt{p}$の説明変数を使用している。ここでは`mtry = 6`を使用した。

```
set.seed(1)
rf.boston <- randomForest(medv ~ ., data = Boston,
    subset = train, mtry = 6, importance = TRUE)
yhat.rf <- predict(rf.boston, newdata = Boston[-train, ])
mean((yhat.rf - boston.test)^2)
## [1] 20.06644
```

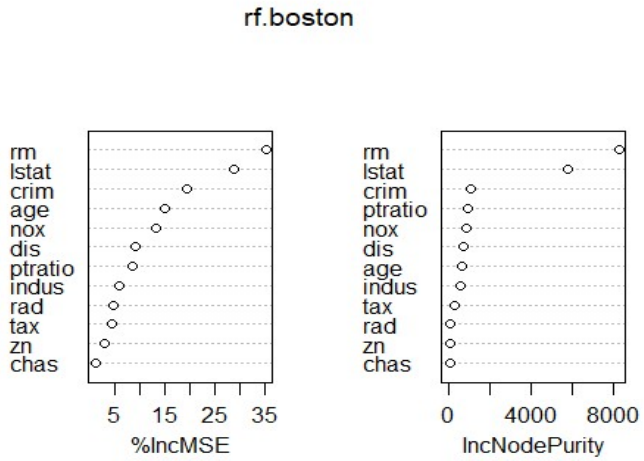
この場合、ランダムフォレストのテスト MSE は20.07であり、バギングよりも良い結果となった。なお、変数の重要度を確認するには、`importance()`関数を使用する。

```
importance(rf.boston)

##           %IncMSE IncNodePurity
## crim    19.435587    1070.42307
## zn       3.091630      82.19257
## indus    6.140529     590.09536
## chas     1.370310      36.70356
## nox     13.263466     859.97091
## rm      35.094741    8270.33906
## age     15.144821     634.31220
## dis      9.163776     684.87953
## rad      4.793720      83.18719
## tax      4.410714     292.20949
```

```
## ptratio  8.612780     902.20190
## lstat   28.725343    5813.04833
```

ここでは変数重要度として 2 つの指標が報告される。1 つ目は、特定の変数を入れ替えた際にアウト・オブ・バッグ（OOB）サンプルでの予測精度が平均的にどれだけ低下するかに基づく指標である。2 つ目は、その変数を基準に分割した際のノード不純度の総減少量を、すべてのツリーで平均した指標である（これは図 8.9 にプロットされている）。回帰木の場合、ノード不純度は学習データの残差平方和（RSS）で測定され、分類木の場合は逸脱度（deviance）で測定される。変数重要度をプロットするには、`varImpPlot()`関数を使用すればよい。

```
varImpPlot(rf.boston)
```



結果を見ると、ランダムフォレスト内のすべてのツリーで、`lstat`（低所得者割合）と `rm`（部屋数）が最も重要な変数であることがわかります。

勾配ブースティング

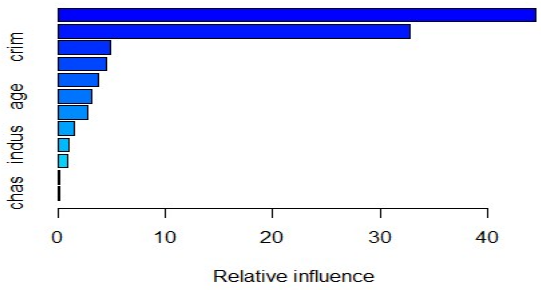
ここでは、`gbm` パッケージを使用して `Boston` データセットに対してブースティングを適用しよう。gbm() 関数を実行する際、distribution = “gaussian” を指定するのは、この問題が回帰問題だからである。もし二値分類問題であれば、distribution = “bernoulli” を指定しておけばよい。引数 n.trees = 5000 は、5000 本のツリーを作

成することを意味します。また、interaction.depth = 4 は、各ツリーの深さを最大 4 に制限している。

```
library(gbm)
## Loaded gbm 2.2.2
## This version of gbm is no longer under development. Consider transitioning t
o gbm3, https://github.com/gbm-developers/gbm3
set.seed(1)
boost.boston <- gbm(medv ~ ., data = Boston[train, ],
    distribution = "gaussian", n.trees = 5000,
    interaction.depth = 4)
```

`summary()`関数を用いると、変数の影響度をプロットできる。

```
summary(boost.boston)
```

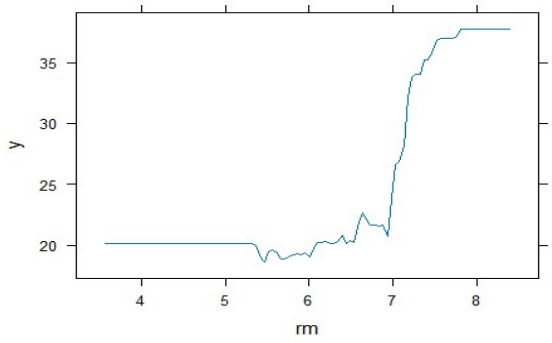



```
##             var     rel.inf
## rm           rm 44.48249588
## lstat     lstat 32.70281223
## crim       crim  4.85109954
## dis         dis  4.48693083
## nox         nox  3.75222394
## age         age  3.19769210
```

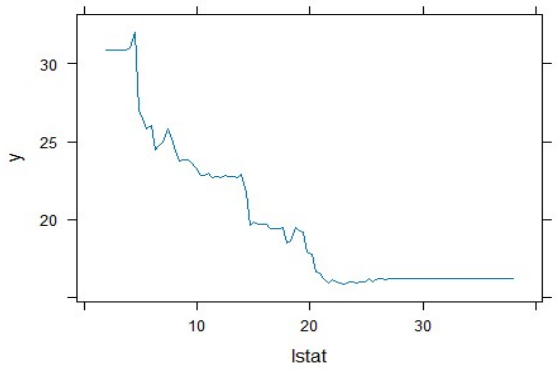
```
## ptratio ptratio  2.81354826
## tax         tax  1.54417603
## indus     indus  1.03384666
## rad         rad  0.87625748
## zn           zn  0.16220479
## chas       chas  0.09671228
```

次に、`rm`と`lstat`の部分依存プロットを作成する。

```
plot(boost.boston, i = "rm")
```




```
plot(boost.boston, i = "lstat")
```



テストデータに対してブースティング・モデルによる予測を行う。

```
yhat.boost <- predict(boost.boston,
    newdata = Boston[-train, ], n.trees = 5000)
mean((yhat.boost - boston.test)^2)
## [1] 18.39057
```

この場合、ブースティングのテスト MSE は18.39であり、ランダムフォレストやバギングよりもさらに良い結果が得られた。なお必要に応じて、(8.10) 式の収縮パラメータ λ の値を変更してブースティングを実行することも可能である。デフォルトの値は `0.001` ですが、簡単に変更は可能である。ここでは、$\lambda = 0.2$ を設定している。

```
boost.boston <- gbm(medv ~ ., data = Boston[train, ],
    distribution = "gaussian", n.trees = 5000,
    interaction.depth = 4, shrinkage = 0.2, verbose = F)
yhat.boost <- predict(boost.boston,
    newdata = Boston[-train, ], n.trees = 5000)
mean((yhat.boost - boston.test)^2)
## [1] 16.54778
```

この場合、$\lambda = 0.2$ とすると、$\lambda = 0.001$ を使用した場合よりもテスト MSE が低くなる。

ベイズ加法回帰木

この節では、`BART` パッケージの `gbart()` 関数を使用して、`Boston` 住宅データ・セットにベイズ加法回帰木（BART）モデルをフィットしてみよう。 `gbart()` 関数は定量的な目的変数（連続値）に適した設計になっている。二値の目的変数の場合は、`lbart()` や `pbart()` を利用すればよい。

`gbart()` 関数を実行するには、まず学習データとテストデータの説明変数を行列形式に変換する必要がある。ここでは、BART をデフォルト設定で実行してみる。

```r
library(BART)
```

```
## 要求されたパッケージ nlme をロード中です
```

```
## 要求されたパッケージ survival をロード中です
```

```r
x <- Boston[, 1:12]
y <- Boston[, "medv"]
xtrain <- x[train, ]
ytrain <- y[train]
xtest <- x[-train, ]
ytest <- y[-train]
set.seed(1)
bartfit <- gbart(xtrain, ytrain, x.test = xtest)
```

```
## *****Calling gbart: type=1
## *****Data:
## data:n,p,np: 253, 12, 253
## y1,yn: 0.213439, -5.486561
## x1,x[n*p]: 0.109590, 20.080000
## xp1,xp[np*p]: 0.027310, 7.880000
## *****Number of Trees: 200
## *****Number of Cut Points: 100 ... 100
## *****burn,nd,thin: 100,1000,1
## *****Prior:beta,alpha,tau,nu,lambda,offset: 2,0.95,0.795495,3,3.71636,21.7866
## *****sigma: 4.367914
## *****w (weights): 1.000000 ... 1.000000
## *****Dirichlet:sparse,theta,omega,a,b,rho,augment: 0,0,1,0.5,1,12,0
## *****printevery: 100
##
## MCMC
## done 0 (out of 1100)
## done 100 (out of 1100)
## done 200 (out of 1100)
## done 300 (out of 1100)
## done 400 (out of 1100)
## done 500 (out of 1100)
## done 600 (out of 1100)
```

```
## done 700 (out of 1100)
## done 800 (out of 1100)
## done 900 (out of 1100)
## done 1000 (out of 1100)
## time: 2s
## trcnt,tecnt: 1000,1000
```

テスト誤差を計算してみる。

```
yhat.bart <- bartfit$yhat.test.mean
mean((ytest - yhat.bart)^2)
## [1] 15.94718
```

BART によりランダム・フォレストやブースティングよりも低いテスト誤差値を示している。ここで変数の使用回数を確認しておこう。

```
ord <- order(bartfit$varcount.mean, decreasing = T)
bartfit$varcount.mean[ord]

##     nox   lstat     tax     rad      rm   indus    chas ptratio     age      zn
##  22.952  21.329  21.250  20.781  19.890  19.825  19.051  18.976  18.274  15.952
##     dis    crim
##  14.457  11.007
```

ISLR 第 9 章 実習：サポート・ベクター・マシン (Support Vector Machines)

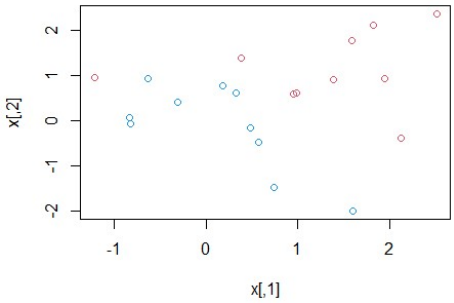
R の e1071 ライブラリを用いてサポートベクトル分類器(support vector classifier, SVM)を実装しよう。また、非常に大規模な線形問題には LiblineaR と云う便利なライブラリもある。

サポートベクトル分類器

e1071 ライブラリには、いくつかの統計学習手法の実装が含まれている。特に、引数 kernel = "linear"を使用すると、svm()関数を使用してサポートベクトル分類器をフィットできる。この関数は、サポートベクトル分類器に関して(9.14)や(9.25)とは若干異なる定式化を利用している。引数 cost により、マージン違反のコストを指定できる。cost が小さいとマージンが広くなり、多くのサポートベクトルがマージン上またはマージンを違反するようになる。一方、cost が大きい場合は、マージンが狭くなり、マージン上または違反するサポートベクトルが少なくなる。

次に、svm()関数を使用して、指定した cost パラメータのサポートベクトル分類器をフィットする。ここでは、2 次元の例を使用して、結果の決定境界をプロットできるようにする。まず、2 つのクラスに属する観測を生成し、クラスが線形分離可能かどうかを確認しよう。

```
set.seed(1)
x <- matrix(rnorm(20 * 2), ncol = 2)
y <- c(rep(-1, 10), rep(1, 10))
x[y == 1, ] <- x[y == 1, ] + 1
plot(x, col = (3 - y))
```



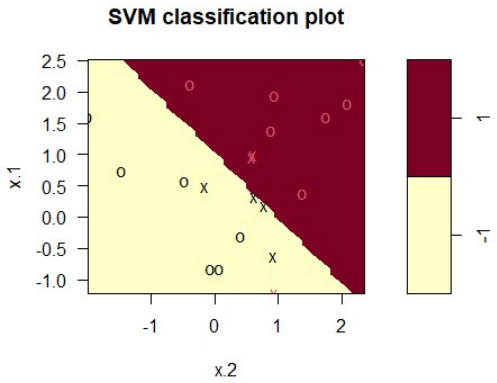
このデータは線形分離可能ではない。そこで、次にサポートベクトル分類器をフィットしてみる。分類（SVM ベースの回帰ではなく）を行うためには、応答をファクタ変数としてエンコードする必要がある。応答変数をファクタとしてコードしたデータフレームを作成する。

```
dat <- data.frame(x = x, y = as.factor(y))
library(e1071)
svmfit <- svm(y ~ ., data = dat, kernel = "linear",
    cost = 10, scale = FALSE)
```

引数 `scale = FALSE` は、各特徴量を平均 0、標準偏差 1 にスケーリングしないように `svm()`関数に指示する。アプリケーションに応じて、`scale = TRUE` を使用する方が良い場合もある。

サポートベクトル分類器による結果をプロットしてみよう：

```
plot(svmfit, dat)
```



SVM の `plot()`関数には、`svm()`の呼び出しの出力と、`svm()`に使用したデータが引数として必要となる。特徴空間のうち、-1クラスに割り当てられる領域は薄黄色で示され、$+1$クラスに割り当てられる領域は赤色で示されている。2 つのクラス間の決定境界は線形であり（`kernel = "linear"`を使用したため）、プロット関数の実装によりプロット上で境界が少しギザギザに見える（通常の`plot()`関数と異なり、ここでは 2 番目の特徴がx軸、1 番目の特徴がy軸にプロットされている）。サポートベクトルは×で、その他の観測値は○で表示されている。ここでは 7 つのサポートベクトルがあることがわかる。サポートベクトルの ID は次のように確認できる：

```
svmfit$index
## [1]  1  2  5  7 14 16 17
```

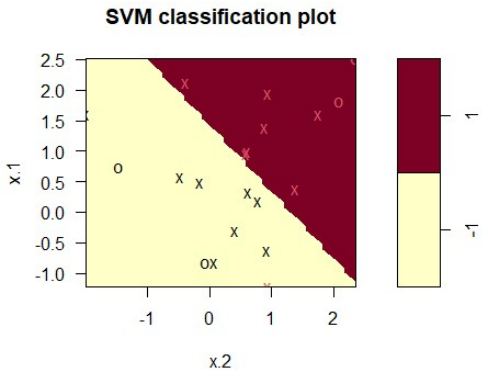
サポートベクトル分類器のフィットについての基本情報を得るために`summary()`コマンドを利用する：

```
summary(svmfit)

##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10, scale = FALSE)
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
##
## Number of Support Vectors:  7
##
##  ( 4 3 )
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

例えば、線形カーネルが使用され、`cost = 10`で、7 つのサポートベクトルがあり、うち 4 つが 1 つのクラス、3 つがもう一方のクラスに属することが分かる。

`cost`パラメータの値を小さくするとどうなるだろうか？

```
svmfit <- svm(y ~ ., data = dat, kernel = "linear",
    cost = 0.1, scale = FALSE)
plot(svmfit, dat)
```




```
svmfit$index
##  [1]  1  2  3  4  5  7  9 10 12 13 14 15 16 17 18 20
```

`cost`の値を小さくしたので、サポートベクトルの数が増え、マージンが広くなった。残念ながら、`svm()`関数は、サポートベクトル分類器をフィットした際に得られる線形決定境界の係数やマージンの幅を明示的に出力してくれない。

`e1071`ライブラリには、組み込み関数`tune()`があり、クロスバリデーションを行うことができる。デフォルトでは`tune()`は、関心のあるモデルセットに対して 10 分割クロスバリデーションを行う。この関数を使用するには、比較するモデルセットに関する情報を渡せばよい。次のコマンドは、線形カーネルを使用した SVM を`cost`パラメータのさまざまな値で比較の為に示している。

```
set.seed(1)
tune.out <- tune(svm, y ~ ., data = dat, kernel = "linear",
    ranges = list(cost = c(0.001, 0.01, 0.1, 1, 5, 10, 100)))
```

各モデルのクロスバリデーションエラーは`summary()`コマンドで簡単に確認できる：

```
summary(tune.out)
##
## Parameter tuning of 'svm':
## - sampling method: 10-fold cross validation
## - best parameters:
##  cost
##   0.1
## - best performance: 0.05
## - Detailed performance results:
##    cost error dispersion
## 1 1e-03  0.55  0.4377975
## 2 1e-02  0.55  0.4377975
## 3 1e-01  0.05  0.1581139
## 4 1e+00  0.15  0.2415229
## 5 5e+00  0.15  0.2415229
## 6 1e+01  0.15  0.2415229
## 7 1e+02  0.15  0.2415229
```

`cost = 0.1`が最も低いクロスバリデーションエラー率をもたらすことが分かる。`tune()`関数は得られた最適モデルを格納しており、次のようにアクセスできる：

```
bestmod <- tune.out$best.model
summary(bestmod)


##
## Call:
## best.tune(METHOD = svm, train.x = y ~ ., data = dat, ranges = list(cost = c(0.001,
##     0.01, 0.1, 1, 5, 10, 100)), kernel = "linear")
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  0.1
```

```
##
## Number of Support Vectors:  16
##
##  ( 8 8 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

`predict()`関数を使用して、任意の`cost`パラメータ値でテスト観測のクラスラベルを予測ができる。まず、テストデータセットを生成しよう。

```
xtest <- matrix(rnorm(20 * 2), ncol = 2)
ytest <- sample(c(-1, 1), 20, rep = TRUE)
xtest[ytest == 1, ] <- xtest[ytest == 1, ] + 1
testdat <- data.frame(x = xtest, y = as.factor(ytest))
```

次に、このテスト観測のクラスラベルを予測しよう。ここではクロスバリデーションで得られた最適モデルを使用して予測を行う。

```
ypred <- predict(bestmod, testdat)
table(predict = ypred, truth = testdat$y)
##        truth
## predict -1 1
##      -1  9 1
##      1   2 8
```

この`cost`の値で、テスト観測のうち 17 件が正しく分類された。`cost = 0.01`を使用した場合はどうなるだろうか？

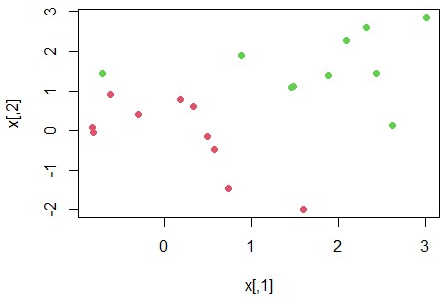
```
svmfit <- svm(y ~ ., data = dat, kernel = "linear",
    cost = .01, scale = FALSE)
```

```
ypred <- predict(svmfit, testdat)
table(predict = ypred, truth = testdat$y)
##        truth
## predict -1  1
##      -1 11  6
##      1   0  3
```

この場合、追加で 3 つの観測が誤分類されている。

次に、2 つのクラスが線形分離可能な場合を考えよう。この場合、`svm()`関数を使用して分離ハイパープレーンを見つけることができる。まず、シミュレーションデータの 2 つのクラスをさらに分離し、線形分離可能にしておこう：

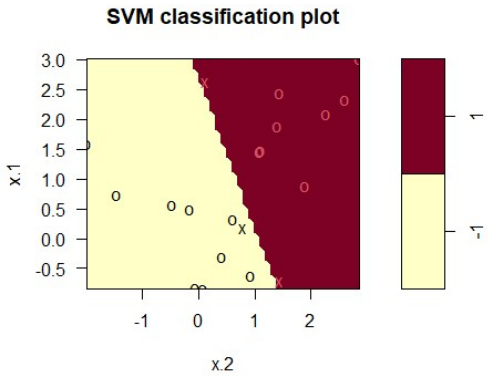
```
x[y == 1, ] <- x[y == 1, ] + 0.5
plot(x, col = (y + 5) / 2, pch = 19)
```



観測値はほぼ線形分離可能になった。非常に大きな`cost`値を使用してサポートベクトル分類器をフィットし、結果の超平面（ハイパープレーン）をプロットする。

```
dat <- data.frame(x = x, y = as.factor(y))
svmfit <- svm(y ~ ., data = dat, kernel = "linear",
    cost = 1e5)
summary(svmfit)

##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 1e+05)
```

```
##
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1e+05
##
## Number of Support Vectors:  3
##
##  ( 1 2 )
##
##
## Number of Classes:  2
##
## Levels:
##  -1 1


plot(svmfit, dat)
```

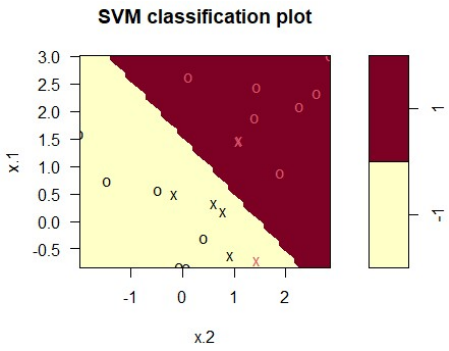


訓練誤差は発生せず、3 つのサポートベクトルが使用されている。しかし、図からもわかるように、マージンが非常に狭くなっている（サポートベクトルでない観測は、oで表示され、決定境界に非常に近い）。このモデルはテストデータに対しては性能が低下する可能性がある。次に、`cost`の値を小さくして試してみよう：

```r
svmfit <- svm(y ~ ., data = dat, kernel = "linear", cost = 1)
summary(svmfit)
```

```
##
## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 1)
##
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  1
##
## Number of Support Vectors:  7
##
##  ( 4 3 )
##
## Number of Classes:  2
##
## Levels:
##  -1 1
```

```r
plot(svmfit, dat)
```



`cost = 1` を使用すると、1 つの訓練観測が誤分類されたが、マージンがはるかに広くなり、7 つのサポートベクトルが使用されている。このモデルは、`cost = 1e5` のモデルよりもテストデータでの性能が向上する可能性がある。

サポートベクトルマシン

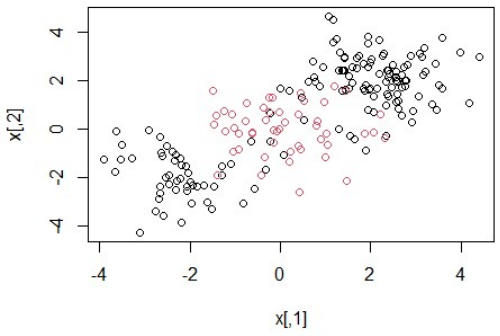
非線形カーネルを利用した SVM をフィットするために、再び `svm()`関数を使用するが、今回は `kernel` パラメータに異なる値を設定する。多項式(ポリノミアル)カーネルを使用する場合は `kernel = "polynomial"`、ラジアル・カーネルを使用する場合は `kernel = "radial"`を指定する。ポリノミアル・カーネルの場合、`degree` 引数で次数を指定し（これは(9.22)の d である）、ラジアルベーシスカーネルの場合は `gamma` でγの値を指定する（（9.24）式）。

次に、非線形クラス境界を持つデータを生成しよう。

```
set.seed(1)
x <- matrix(rnorm(200 * 2), ncol = 2)
x[1:100, ] <- x[1:100, ] + 2
x[101:150, ] <- x[101:150, ] - 2
y <- c(rep(1, 150), rep(2, 50))
dat <- data.frame(x = x, y = as.factor(y))
```

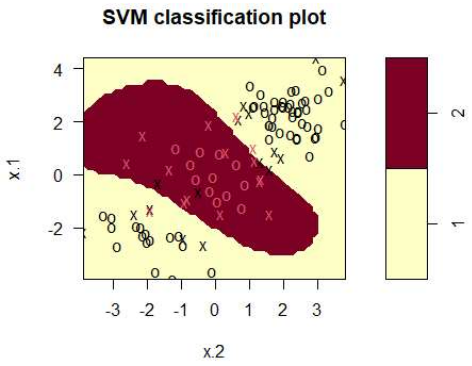
データをプロットすると、クラス境界が非線形であることは明らかだろう：

```
plot(x, col = y)
```



データはランダムに訓練セットとテストセットに分割される。その後、ラジアル・カーネルと$\gamma = 1$で訓練データをフィットする：

```
train <- sample(200, 100)
svmfit <- svm(y ~ ., data = dat[train, ], kernel = "radial",
    gamma = 1, cost = 1)
plot(svmfit, dat[train, ])
```



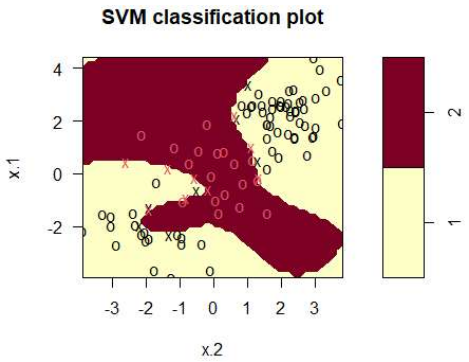
プロットから、結果として得られた SVM の決定境界がはっきりと非線形であることが分かる。`summary()`関数を使用して、SVM 適合についての情報を得ることができる。：

```
summary(svmfit)
```

```
## Call:
## svm(formula = y ~ ., data = dat[train, ], kernel = "radial", gamma = 1,
##     cost = 1)
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  radial
##        cost:  1
## Number of Support Vectors:  31
##  ( 16 15 )
## Number of Classes:  2
## Levels:
##  1 2
```

図から、多くの訓練エラーが発生していることが分かる。`cost`の値を増やすことで、訓練エラーの数を減らすことができるが、その代償として決定境界がより不規則になり、データの過剰フィットのリスクが高まる。

```
svmfit <- svm(y ~ ., data = dat[train, ], kernel = "radial",
    gamma = 1, cost = 1e5)
plot(svmfit, dat[train, ])
```



ラジアル・カーネルを使用する SVM の最適な γおよび`cost`の選択のために、クロスバリデーションを実行することもできる。

```
set.seed(1)
tune.out <- tune(svm, y ~ ., data = dat[train, ],
    kernel = "radial",
    ranges = list(
      cost = c(0.1, 1, 10, 100, 1000),
      gamma = c(0.5, 1, 2, 3, 4)
    )
  )
summary(tune.out)

##
## Parameter tuning of 'svm':
##
## - sampling method: 10-fold cross validation
## - best parameters:
```

```
##  cost gamma
##     1   0.5
## - best performance: 0.07
## - Detailed performance results:
##     cost gamma error dispersion
## 1  1e-01   0.5  0.26 0.15776213
## 2  1e+00   0.5  0.07 0.08232726
## 3  1e+01   0.5  0.07 0.08232726
## 4  1e+02   0.5  0.14 0.15055453
## 5  1e+03   0.5  0.11 0.07378648
## 6  1e-01   1.0  0.22 0.16193277
## 7  1e+00   1.0  0.07 0.08232726
## 8  1e+01   1.0  0.09 0.07378648
## 9  1e+02   1.0  0.12 0.12292726
## 10 1e+03   1.0  0.11 0.11005049
## 11 1e-01   2.0  0.27 0.15670212
## 12 1e+00   2.0  0.07 0.08232726
## 13 1e+01   2.0  0.11 0.07378648
## 14 1e+02   2.0  0.12 0.13165612
## 15 1e+03   2.0  0.16 0.13498971
## 16 1e-01   3.0  0.27 0.15670212
## 17 1e+00   3.0  0.07 0.08232726
## 18 1e+01   3.0  0.08 0.07888106
## 19 1e+02   3.0  0.13 0.14181365
## 20 1e+03   3.0  0.15 0.13540064
## 21 1e-01   4.0  0.27 0.15670212
## 22 1e+00   4.0  0.07 0.08232726
## 23 1e+01   4.0  0.09 0.07378648
## 24 1e+02   4.0  0.13 0.14181365
## 25 1e+03   4.0  0.15 0.13540064
```

従って、最適なパラメータの選択は`cost = 1`および`gamma = 0.5`となる。 このモデルを使用してテストセットの予測を`predict()`関数で行う。その為にデータフレーム`dat`の`-train`インデックスを使用してサブセット化する。

```
table(
    true = dat[-train, "y"],
    pred = predict(
```

```
      tune.out$best.model, newdata = dat[-train, ]
      )
    )
##     pred
## true  1  2
##    1 67 10
##    2  2 21
```

12%のテスト観測値がこの SVM で誤分類されたことが分かる。

ROC 曲線

`ROCR` パッケージを使用すると、図 9.10 および図 9.11 のような ROC 曲線を作成できる。各観測の数値スコア `pred` とクラスラベル `truth` のベクトルを与え、ROC 曲線をプロットする短い関数を作成しよう。

```
library(ROCR)
rocplot <- function(pred, truth, ...) {
  predob <- prediction(pred, truth)
  perf <- performance(predob, "tpr", "fpr")
  plot(perf, ...)
}
```

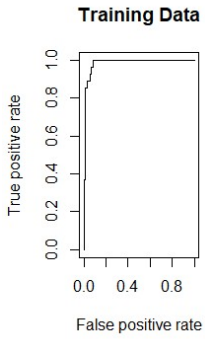
SVM とサポートベクトル分類器は各観測のクラスラベルを出力するが、各観測の*予測値*も取得することが可能で、これはクラスラベルを得るために使用された数値スコアとなる。例えば、サポートベクトル分類器の場合、観測 $X = (X_1, X_2, \ldots, X_p)^T$ の予測値は $\hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2 + \cdots + \hat{\beta}_p X_p$ の形を取る。非線形カーネルの SVM の場合、予測値を得る方程式は(9.23)で示されている。本質的に、予測値の符号は観測値がどちらのクラスに属するかを決定する。SVM モデルの適合で `decision.values = TRUE` を設定することで予測値が得られ、`predict()`関数で出力されるようになっている。

```
svmfit.opt <- svm(y ~ ., data = dat[train, ],
    kernel = "radial", gamma = 2, cost = 1,
```

```
    decision.values = T)
fitted <- attributes(
    predict(svmfit.opt, dat[train, ], decision.values = TRUE)
  )$decision.values
```

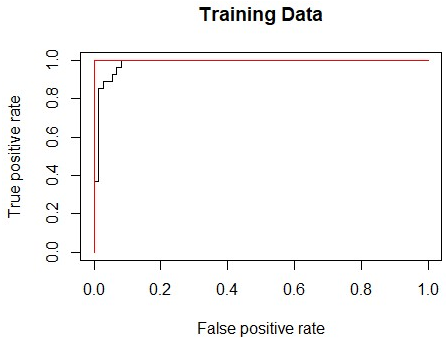
次に、ROC プロットを作成しよう。負の予測値を使用して、負の値がクラス 1 に、正の値がクラス 2 に対応するようにする。

```
par(mfrow = c(1, 2))
rocplot(-fitted, dat[train, "y"], main = "Training Data")
```



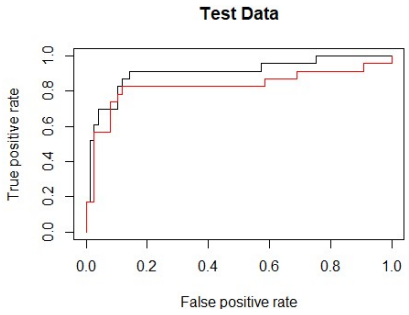
SVM は正確な予測を行っているようである。γを増加することで、より柔軟な適合を生成し、精度をさらに向上させることができる。

```
rocplot(-fitted, dat[train, "y"], main = "Training Data")
svmfit.flex <- svm(y ~ ., data = dat[train, ],
    kernel = "radial", gamma = 50, cost = 1,
    decision.values = T)
fitted <- attributes(
    predict(svmfit.flex, dat[train, ], decision.values = T)
  )$decision.values
rocplot(-fitted, dat[train, "y"], add = T, col = "red")
```



ただし、これらの ROC 曲線はすべて訓練データに基づいている。実際にはテストデータでの予測精度がより重要となる。テストデータの ROC 曲線を計算すると、$\gamma = 2$のモデルが最も正確な結果を与えていることが分かる。

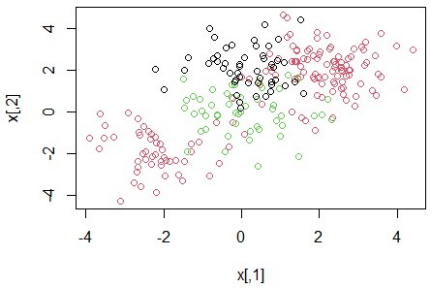
```
fitted <- attributes(
    predict(svmfit.opt, dat[-train, ], decision.values = T)
  )$decision.values
rocplot(-fitted, dat[-train, "y"], main = "Test Data")
fitted <- attributes(
    predict(svmfit.flex, dat[-train, ], decision.values = T)
  )$decision.values
rocplot(-fitted, dat[-train, "y"], add = T, col = "red")
```



複数クラスを持つ SVM

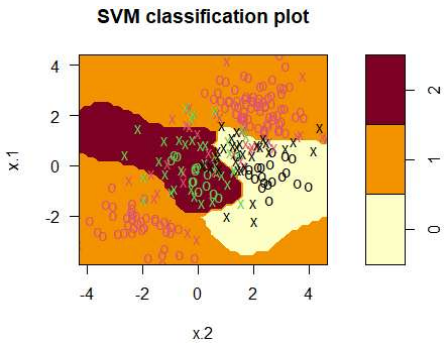
応答が 2 つ以上のレベルを含むファクタである場合、`svm()`関数は 1 対 1 のアプローチを用いて多クラス分類を行う。ここでは 3 番目のクラスの観測値を生成することで、この設定を探ってみよう。

```
set.seed(1)
x <- rbind(x, matrix(rnorm(50 * 2), ncol = 2))
y <- c(y, rep(0, 50))
x[y == 0, 2] <- x[y == 0, 2] + 2
dat <- data.frame(x = x, y = as.factor(y))
par(mfrow = c(1, 1))
plot(x, col = (y + 1))
```



次に、このデータに SVM をフィットする。

```
svmfit <- svm(y ~ ., data = dat, kernel = "radial",
    cost = 10, gamma = 1)
plot(svmfit, dat)
```



`e1071` ライブラリを使用すると、`svm()`に数値の応答ベクトルを渡すことで、サポートベクトル回帰を実行することもできる。

遺伝子発現データへの適用

ここで、`Khan`データ・セットを調べてみよう。このデータは、4 つの異なる種類の小型青色細胞腫瘍に対応する複数の組織サンプルで構成されており、各組織サンプルについて遺伝子発現測定が利用可能である。このデータセットには、トレーニング・データ`xtrain`と`ytrain`、およびテストデータ`xtest`と`ytest`が含まれている。

データの次元を確認しておこう。

```
library(ISLR2)
names(Khan)
## [1] "xtrain" "xtest"  "ytrain" "ytest"
dim(Khan$xtrain)
## [1]   63 2308
dim(Khan$xtest)
## [1]   20 2308
length(Khan$ytrain)
## [1] 63
length(Khan$ytest)
## [1] 20
```

このデータ・セットは、2,308 個の遺伝子の発現測定で構成され、トレーニング・セットとテスト・セットにはそれぞれ 63 個と 20 個の観測値が含まれている。

```
table(Khan$ytrain)
##
##  1  2  3  4
##  8 23 12 20
table(Khan$ytest)
##
## 1 2 3 4
## 3 6 6 5
```

サポートベクトル・アプローチを利用して、遺伝子発現測定を基に癌のサブタイプを予測してみる。このデータセットでは、観測数に対して特徴数が非常に多いため、多項式カーネルやラジアル・カーネルを使用することで生じる柔軟性は不要と考えられるため、線形カーネルを使用するのが適切となる。

```
dat <- data.frame(
    x = Khan$xtrain,
    y = as.factor(Khan$ytrain)
  )
out <- svm(y ~ ., data = dat, kernel = "linear",
    cost = 10)
summary(out)

## Call:
## svm(formula = y ~ ., data = dat, kernel = "linear", cost = 10)
## Parameters:
##    SVM-Type:  C-classification
##  SVM-Kernel:  linear
##        cost:  10
## Number of Support Vectors:  58
##  ( 20 20 11 7 )
## Number of Classes:  4
## Levels:
##  1 2 3 4
table(out$fitted, dat$y)
##
##      1  2  3  4
##   1  8  0  0  0
##   2  0 23  0  0
##   3  0  0 12  0
##   4  0  0  0 20
```

この例では*訓練誤差がない*ことが分かる。これは、多数の変数が観測数に比べて非常に多いため、クラスを完全に分離する超平面を見つけることが容易であるためである。しかし我々が最も関心を持っているのは、サポートベクトル分類器の訓練観測に対する性能ではなく、テスト観測に対する性能である。

```
dat.te <- data.frame(
    x = Khan$xtest,
    y = as.factor(Khan$ytest))
pred.te <- predict(out, newdata = dat.te)
table(pred.te, dat.te$y)
##
## pred.te 1 2 3 4
##       1 3 0 0 0
##       2 0 6 2 0
##       3 0 0 4 0
##       4 0 0 0 5
```

このデータで`cost = 10`を使用すると、テストセットで 2 つの誤分類が発生することが分かる。

ISLR 第 10 章 実習：深層学習 1 (Deep Learning, keras 版)

本節では、教科書(ISL)で議論された例をどのように学習させるかを説明する。 `keras` パッケージを利用して `tensorflow` パッケージと連携し、さらに効率的な `python` コードへとリンクする。 このコードは非常に高速であり、パッケージも整っている。 本節に関わる、優れた参考書として *Deep Learning with R*（F. Chollet and J.J. Allaire, *Deep Learning with R* (2018), Manning Publications, 訳注：邦訳「R と Keras によるディープラーニング」オライリー・ジャパン）を挙げておく。 実は本節のコードの大部分はこの書籍から採用したものである。

パッケージ `keras` をコンピュータ上で動作させることは手間がかかるかもしれない。書籍のウェブサイト <www.statlearning.com> では、セットアップの手順を詳しく解説している。 （`keras` のインストール手順を準備してくれた Balasubramanian Narasimhan に感謝する。） また、<keras.rstudio.com> でもガイドが提供されている。

訳注 1：`Anaconda` をインストールして、`Rstudio` で、`Tools`、`Global Options...`、`Python`、`Python interpreter` の `Select...` の順に進み、`Conda Environments` から Anaconda を選んでおく、という手順で実行できることが確認できた.

訳注 2：上記の Keras のセットアップの手順について、この章（深層学習 1）の最後に追加事項として載せている。 Anaconda をインストールしてもエラーが出る場合、追加事項の「2．conda がインストールされていない場合」にしたがう事でエラーがなくなることも確かめられた。

また本書の初版発行後、深層学習（ディープラーニング）実行の代替手段として `torch` パッケージが利用可能となった。 `torch` は `python` のインストールを必要としないが、現時点での実装は `keras` よりもかなり遅いように見受けられる。 `torch` を使用した実習例も書籍のウェブサイトで公開されている。 （`torch` 版の実習を準備してくれた Daniel Falbel および Sigrid Keydana に感謝する。）

訳注 3：R だけしか利用しないパッケージ torch による実習は実習・ 深層学習 2 として本稿・深層学習 1 に続いて掲載した。

訳注 4：Neural Networks, Deep Learning の実装に関する方法、例えば pooling(プーリング)や dropout(ドロップ・アウト)などについては例えば前述の「R と Keras によるディープラーニング」の他、「あたらしい機械学習の教科書」(伊藤真,翔泳社)などによる説明が参考になる。

訳注 5: コマンドの設定や C 環境の違いにより繰り返し計算の実行時間がかなりかかる場合がある。多分、初期値や乱数の設定などと思われる PC 環境の違いなどにより結果の数値が若干異なることがある。

打者(Hitters)データに対する単層ネットワーク

10.6 節で示されたモデルを学習しよう。データを準備し、訓練(トレーニング)セットとテストセットに分割する。

```
library(ISLR2)
```

```
## Warning: パッケージ 'ISLR2' はバージョン 4.4.3 の R の下で造られました
```

```
Gitters <- na.omit(Hitters)
n <- nrow(Gitters)
set.seed(13)
ntest <- trunc(n / 3)
testid <- sample(1:n, ntest)
```

線形モデルは馴染みがあるはずだが、ここでも改めて提示する。

```
lfit <- lm(Salary ~ ., data = Gitters[-testid, ])
lpred <- predict(lfit, Gitters[testid, ])
with(Gitters[testid, ], mean(abs(lpred - Salary)))
```

```
## [1] 254.6687
```

ここで `with()` コマンドの使用に注目する。第 1 引数にはデータフレームを指定し、第 2 引数にはデータフレームの要素を名前で参照した式を指定する。この場合、データフレームはテストデータに対応しており、ここではテストデータにおける平均絶対予測誤差を計算する。

次に、`glmnet` を用いて Lasso(ラッソ)回帰を学習する。このパッケージは `formula` を使用しないため、まず `x` と `y` を作成する。

```
x <- scale(model.matrix(Salary ~ . - 1, data = Gitters))
y <- Gitters$Salary
```

最初の行では `model.matrix()` を呼び出している。 これは `lm()` で使用されるのと同じ行列を生成する（`-1` を指定することで切片を省略している）。 この関数は因子型変数を自動的にダミー変数へ変換する。 `scale()` 関数は行列を標準化し、各列の平均を 0、分散を 1 にする。

```
library(glmnet)
## Warning: パッケージ 'glmnet' はバージョン 4.4.3 の R の下で造られました
## 要求されたパッケージ Matrix をロード中です
## Loaded glmnet 4.1-8
cvfit <- cv.glmnet(x[-testid, ], y[-testid],
    type.measure = "mae")
cpred <- predict(cvfit, x[testid, ], s = "lambda.min")
mean(abs(y[testid] - cpred))
## [1] 252.2994
```

ニューラルネットワークモデルを学習するために、まずネットワークの構造を定義する。

```
library(keras)
## Warning: パッケージ 'keras' はバージョン 4.4.3 の R の下で造られました
reticulate::install_miniconda() # conda がインストールされている場合は省略して良い
keras::install_keras(method = "conda", python_version = "3.10")
reticulate::use_condaenv(condaenv = "r-tensorflow")
modnn <- keras_model_sequential() %>%
  layer_dense(units = 50, activation = "relu",
        input_shape = ncol(x)) %>%
   layer_dropout(rate = 0.4) %>%
   layer_dense(units = 1)
```

ここでは、`modnn` というニューラルネットワークモデルオブジェクトを作成し、`keras_model_sequential()` を使用して順次、層を追加している。 パイプ演算子 `\%>\%` を用いることで、前の処理結果を次の関数の第 1 引数として渡し、その結果を返す。 これにより、ニューラルネットワークの層構造を可読性の高い形式で記述できる。

パイプ演算子の使用例として、単純なケースを示す。 先ほど、`x` を作成するために以下のコードを用いた。

```
x <- scale(model.matrix(Salary ~ . - 1, data = Gitters))
```

まず行列を作成し、次に各変数をセンタリングしている。 このような複合的な表現は構文解析を難しくすることがある。 同じ結果を、パイプ演算子を使って得ることもできる。

```
x <- model.matrix(Salary ~ . - 1, data = Gitters) %>% scale()
```

パイプ演算子を使用すると、一連の処理の流れを追いやすくなる。

ここで再びニューラルネットワークに戻る。オブジェクト `modnn` は、50 個のユニットを持つ単一の隠れ層を持ち、活性化関数として ReLU を使用する。
続くドロップアウト層では、確率的勾配降下法 (SGD) アルゴリズムの各イテレーションにおいて、前の層の 50 個のアクティベーションのうち 40% がランダムに 0 に設定される。
最後の出力層には 1 つのユニットがあり、活性化関数を持たないため、単一の数値を出力するモデルとなる。

次に、`modnn` に学習アルゴリズムの詳細を追加しよう。

ここでは、Keras の公式ドキュメントの例に従い、(10.23)の二乗誤差損失を最小化するように設定する。
また、訓練データにおける平均絶対誤差 (MAE) を追跡し、バリデーションデータが提供される場合にバリデーションデータに対するものも記録する。

```
modnn %>% compile(loss = "mse",
    optimizer = optimizer_rmsprop(),
    metrics = list("mean_absolute_error")
  )
```

上記のコードでは、パイプ演算子 `\%>\%` を使用し、`modnn` を `compile()` の第 1 引数として渡している。
`compile()` 関数は、`R` のオブジェクト `modnn` を直接変更するわけではなく、このモデルに関連する `python` インスタンスへ設定情報を伝える役割を果たす。

次に、モデルの学習を行う。
訓練データと、2 つの学習パラメータ `epochs`（エポック数）と `batch_size`（バッチサイズ）を指定する。
ここでは `batch_size = 32` とすることで、SGD の各ステップでランダムに 32 個の訓練データを選択し、勾配計算を行う。

10.4 節および 10.7 節で説明したように、1 エポックは n 個の観測データを 1 回処理するために必要な SGD ステップ数に対応する。
今回の訓練データのサンプル数は $n = 176$ であり、バッチサイズが 32 なので、1 エポックは $176/32 = 5.5$回の SGD ステップとなる。

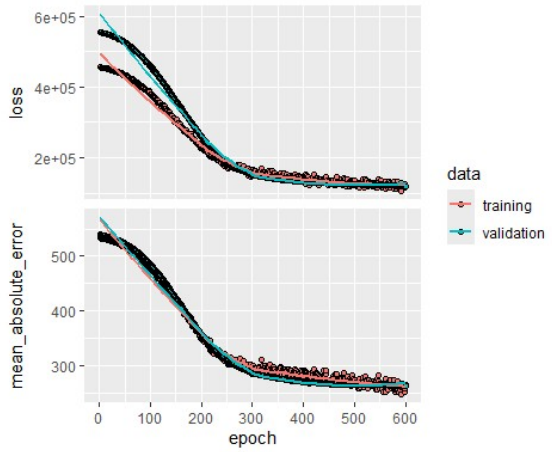
`fit()` 関数の `validation_data` 引数にテストデータを渡すと、訓練には使われず、モデルの学習の進捗状況を確認できる（ここでは MAE を記録）。ここでは実際にテストデータを与え、エポックが進むごとに訓練データとテストデータに対する平均絶対誤差が表示される。 なお学習の詳細オプションについては、`?fit.keras.engine.training.Model` を参照するとよい。

```
history <- modnn %>% fit(
#    x[-testid, ], y[-testid], epochs = 1500, batch_size = 32,
    x[-testid, ], y[-testid], epochs = 600, batch_size = 32,
    validation_data = list(x[testid, ], y[testid])
  )
## Epoch 1/600
## 6/6 - 0s - loss: 456637.2188 - mean_absolute_error: 533.4519 - val_loss: 555391.1875 - val_mean_
absolute_error: 539.3817 - 373ms/epoch - 62ms/step
訳者中略
## Epoch 599/600
## 6/6 - 0s - loss: 126506.1484 - mean_absolute_error: 258.8999 - val_loss: 120205.2188 - val_mean_
absolute_error: 263.1526 - 32ms/epoch - 5ms/step
## Epoch 600/600
## 6/6 - 0s - loss: 118580.7422 - mean_absolute_error: 265.6867 - val_loss: 120052.1172 - val_mean_
absolute_error: 262.9491 - 24ms/epoch - 4ms/step
```

(ここでは実行時間を管理しやすくするためにエポック数を 600 に減らしているが、もちろんユーザーが値を設定できる。)

`history` をプロットすることで、訓練データとテストデータにおける平均絶対誤差 (MAE) を表示できる。
最適な見た目にするために、`plot()` 関数を呼び出す前に `ggplot2` パッケージをインストールしておくとよい。
`ggplot2` をインストールしていなくても、以下のコードは動作するが、グラフの見た目がやや劣る可能性がある。

```
plot(history)
```



`fit()` コマンドを同じ `R` セッション内で再度実行すると、学習プロセスは前回の続きから再開される。
`fit()` コマンドを再実行し、その後 `plot()` コマンドを実行して、変化を確認してみよう。

最後に、学習済みモデルを用いて予測を行い、テストデータ上での性能を評価する。
SGD を使用しているため、学習のたびに結果がわずかに異なる。
残念ながら、`set.seed()` 関数を使用しても完全に同じ結果にはならない（`python` 内部での学習処理が影響するため）。
したがって、出力される結果は実行のたびに若干異なる。

```
npred <- predict(modnn, x[testid, ])
## 3/3 - 0s - 59ms/epoch - 20ms/step
```

```
mean(abs(y[testid] - npred))
## [1] 262.9491
```

MNIST 手書き数字データに対する多層ニューラルネットワーク

`keras` パッケージには、`MNIST` 手書き数字データを含むいくつかのサンプルデータセットが付属している。
最初のステップとして、`MNIST` データを読み込む。この目的のために、`dataset_mnist()` 関数が用意されている。

```
mnist <- dataset_mnist()
x_train <- mnist$train$x
g_train <- mnist$train$y
x_test <- mnist$test$x
g_test <- mnist$test$y
dim(x_train)
## [1] 60000    28    28
dim(x_test)
## [1] 10000    28    28
```

訓練データには 60,000 枚、テストデータには 10,000 枚の画像が含まれている。
画像のサイズは 28 × 28 ピクセルであり、3 次元配列として保存されているため、行列形式に変換する必要がある。また、クラスラベルを「One-Hot エンコーディング」（ダミー変数化）する必要がある。
幸いなことに、`keras` にはこれらの処理を簡単に行うための便利な関数が用意されている。

```
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
y_train <- to_categorical(g_train, 10)
y_test <- to_categorical(g_test, 10)
```

ニューラルネットワークは入力のスケールに対してやや敏感である。例えば、Ridge（リッジ）や Lasso（ラッソ）正則化もスケーリングの影響を受ける。
ここでの入力データは 8 ビットのグレースケール値であり、0 から 255 の範囲をとるため、[0,1] の範囲にリスケールする。（なお 8 ビットとは $2^8 = 256$ であり、通常 0 から始まるため、可能な値の範囲は 0 から 255 となる。）

```
x_train <- x_train / 255
x_test <- x_test / 255
```

これでニューラルネットワークの学習を行う準備が整った。

```
modelnn <- keras_model_sequential()
modelnn %>%
  layer_dense(units = 256, activation = "relu",
        input_shape = c(784)) %>%
  layer_dropout(rate = 0.4) %>%
  layer_dense(units = 128, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")
```

入力層：$28 \times 28 = 784$ のユニットを持つ。

第 1 隠れ層: 256 ユニットを持ち、ReLU 活性化関数を使う。 `layer_dense()`は、`modelnn` オブジェクトを入力として受け取り、修正された `modelnn` オブジェクトを返す。 その後、`layer_dropout()` によりドロップアウト正則化を適用する。

第 2 層隠れ層:128ユニットを持つ。 その後、再びドロップアウト層を適用される。

出力層:`"softmax"` 活性化関数(10.13)を用いて 10 クラス分類問題に対応する最終層になっている。 第 2 隠れ層からクラス確率への写像を定義している。

最後に、`summary()` を使ってモデルの概要を確認し、正しく構築されたかをチェックする。

```
summary(modelnn)
## Model: "sequential_1"
## ________________________________________________________________________
```

```
##  Layer (type)                        Output Shape                      Param #
## ================================================================================
##  dense_4 (Dense)                     (None, 256)                       200960
##  dropout_2 (Dropout)                 (None, 256)                       0
##  dense_3 (Dense)                     (None, 128)                       32896
##  dropout_1 (Dropout)                 (None, 128)                       0
##  dense_2 (Dense)                     (None, 10)                        1290
## ================================================================================
## Total params: 235146 (918.54 KB)
## Trainable params: 235146 (918.54 KB)
## Non-trainable params: 0 (0.00 Byte)
## ________________________________________________________________________________
```

各層のパラメータには定数項（バイアス項）も含まれるため、モデル全体のパラメータ数は 235,146 となる。
例えば、第 1 隠れ層では以下のように計算される：$(784 + 1) \times 256 = 200{,}960$

また、各層の名前（例: `dropout_1` や `dense_2`）には添え字がついている。これらの添え字はランダムに見えるが、実際、モデルを再構築するたびに変化する。これは `keras_model_sequential()` を呼び出すたびに `python` 内部でインクリメントされるためであり、特に問題はない。

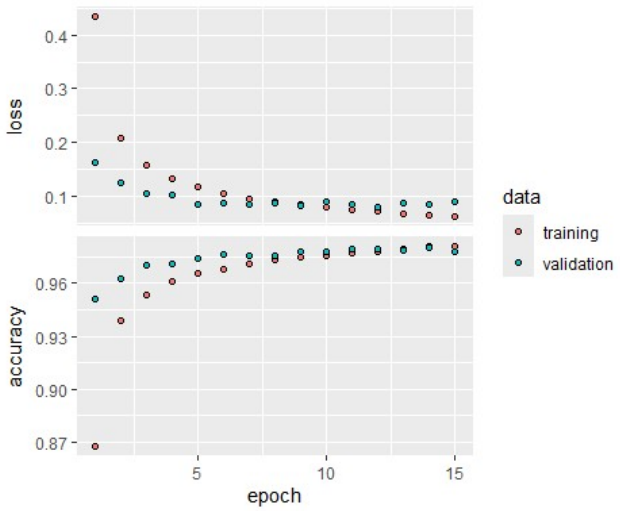
次に、学習アルゴリズムの詳細をモデルに追加する。
損失関数として交差エントロピー (10.14) を最小化する。

```
modelnn %>% compile(loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(), metrics = c("accuracy")
  )
```

準備が整った。最終的なステップは、訓練データを与えて、モデルを学習させることである。

```
system.time(
  history <- modelnn %>%
#     fit(x_train, y_train, epochs = 30, batch_size = 128,
      fit(x_train, y_train, epochs = 15, batch_size = 128,
        validation_split = 0.2)
)
```

```
## Epoch 1/15
## 375/375 - 2s - loss: 0.4338 - accuracy: 0.8680 - val_loss: 0.1629 - val_accuracy: 0.9512 - 2s/ep
och - 4ms/step
訳者中略
## Epoch 14/15
## 375/375 - 2s - loss: 0.0641 - accuracy: 0.9807 - val_loss: 0.0839 - val_accuracy: 0.9803 - 2s/ep
och - 4ms/step
## Epoch 15/15
## 375/375 - 1s - loss: 0.0629 - accuracy: 0.9810 - val_loss: 0.0902 - val_accuracy: 0.9781 - 1s/ep
och - 4ms/step
##    ユーザ システム       経過
##    48.69    13.87    21.94
plot(history, smooth = FALSE)
```



これはモデルの学習に関する進捗報告であり、エポックごとにグループ化されている。この情報は非常に有用である。大規模なデータセットでは、学習に時間がかかることがあるためである。このモデルの学習には、2.9 GHz の MacBook Pro（4 コア、32 GB RAM）で 144 秒を要した。ここでは 20%の検証分割を指定しているため、実際の訓練は訓練セットの 60,000 件の観測データの 80%で行われている。これは、10.9.1 節で行ったように、実際に検証データを与える代わりの方法である。オプション引数については、`?fit.keras.engine.training.Model` を参照せよ。SGD は、勾配を計算する際に 128 の観測データのバッチを使用しており、計算結果から 1 エポックが 375 の勾配ステップに対応していることがわかる。最後の `plot()` コマンドは、図 10.18 に似た図を生成する。

表 10.1 のテスト誤差を得るには、予測されたクラスラベルと真のクラスラベルを比較する簡単な関数 `accuracy()` をまず作成し、それを使って予測を評価する。

```
accuracy <- function(pred, truth)
  mean(drop(as.numeric(pred)) == drop(truth))
modelnn %>% predict(x_test) %>% k_argmax() %>% accuracy(g_test)
## 313/313 - 0s - 283ms/epoch - 903us/step
## [1] 0.9797
```

表 10.1 には、LDA（第 4 章）および多項ロジスティック回帰も報告されている。`glmnet` のようなパッケージは多項ロジスティック回帰を処理できるが、この大規模なデータセットでは非常に遅くなる。そのため、`keras` ソフトウェアを使用してモデルを学習させる方がはるかに速く、簡単である。ここでは、入力層と出力層だけを持ち、隠れ層は省略する。

```
modellr <- keras_model_sequential() %>%
  layer_dense(input_shape = 784, units = 10,
              activation = "softmax")
summary(modellr)
## Model: "sequential_2"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  dense_5 (Dense)                    (None, 10)                      7850
## =========================================================================
## Total params: 7850 (30.66 KB)
## Trainable params: 7850 (30.66 KB)
## Non-trainable params: 0 (0.00 Byte)
## _________________________________________________________________________
```

モデルは以前と同じように学習させる。

```
modellr %>% compile(loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(), metrics = c("accuracy"))
modellr %>% fit(x_train, y_train, epochs = 30,
     batch_size = 128, validation_split = 0.2)
```

```
## Epoch 1/30
## 375/375 - 1s - loss: 0.6774 - accuracy: 0.8316 - val_loss: 0.3616 - val_accuracy: 0.9023 - 724ms
/epoch - 2ms/step
訳者中略
## Epoch 29/30
## 375/375 - 0s - loss: 0.2501 - accuracy: 0.9319 - val_loss: 0.2630 - val_accuracy: 0.9299 - 494ms
/epoch - 1ms/step
## Epoch 30/30
## 375/375 - 1s - loss: 0.2496 - accuracy: 0.9327 - val_loss: 0.2618 - val_accuracy: 0.9309 - 510ms
/epoch - 1ms/step
modellr %>% predict(x_test) %>% k_argmax() %>% accuracy(g_test)
## 313/313 - 0s - 234ms/epoch - 749us/step
## [1] 0.9279
```

畳み込みニューラルネットワーク(Convolutional Neural Networks, CNN)

この節では、`keras`パッケージで利用可能な`CIFAR`データに対して CNN を学習させる。これは、`MNIST`データと似た形式になっている。

```
cifar100 <- dataset_cifar100()
names(cifar100)
## [1] "train" "test"
x_train <- cifar100$train$x
g_train <- cifar100$train$y
x_test <- cifar100$test$x
g_test <- cifar100$test$y
dim(x_train)
## [1] 50000    32    32     3
range(x_train[1,,, 1])
## [1]  13 255
```

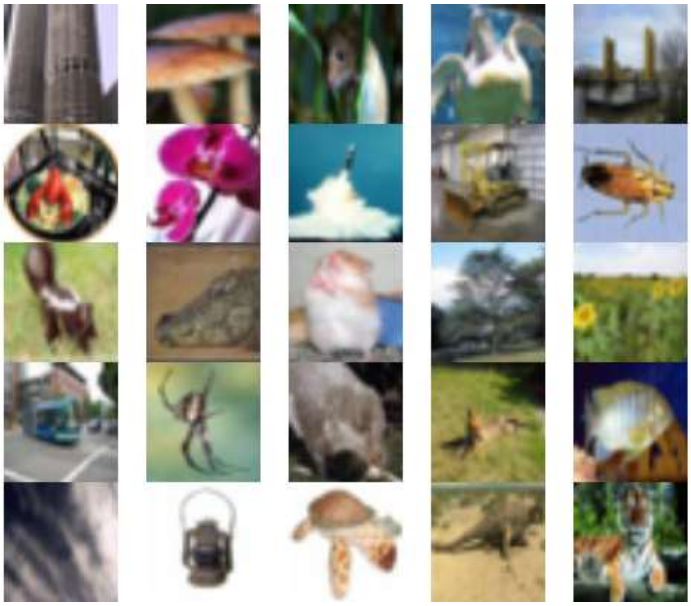
50,000 枚の訓練画像の配列は 4 次元を持ち、カラー画像は 3 つのチャンネルのセットとして表され、各チャンネルは32 × 32の 8 ビットピクセルで構成されている。

前処理として、`MNIST`データと同様に標準化を行うが、配列構造は維持する。応答変数は 100 列のバイナリ行列に One-Hot エンコードする。

```
x_train <- x_train / 255
x_test <- x_test / 255
y_train <- to_categorical(g_train, 100)
dim(y_train)
## [1] 50000   100
```

始める前に、`jpeg`パッケージを使って訓練画像のいくつかを確認する。類似のコードによって図 10.5（411 ページ）を生成した。

```
library(jpeg)
par(mar = c(0, 0, 0, 0), mfrow = c(5, 5))
index <- sample(seq(50000), 25)
for (i in index) plot(as.raster(x_train[i,,, ]))
```



`as.raster()`関数は、特徴マップをカラー画像としてプロットできるように変換する。なお、デモンストレーションのためにここでは中程度のサイズの CNN を指定するが、このモデルは図 10.8 に似た構造を持っている。

```
model <- keras_model_sequential() %>%
  layer_conv_2d(filters = 32, kernel_size = c(3, 3),
      padding = "same", activation = "relu",
      input_shape = c(32, 32, 3)) %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 64, kernel_size = c(3, 3),
      padding = "same", activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 128, kernel_size = c(3, 3),
      padding = "same", activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_conv_2d(filters = 256, kernel_size = c(3, 3),
      padding = "same", activation = "relu") %>%
  layer_max_pooling_2d(pool_size = c(2, 2)) %>%
  layer_flatten() %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 512, activation = "relu") %>%
  layer_dense(units = 100, activation = "softmax")
summary(model)
## Model: "sequential_3"
## _________________________________________________________________________
##  Layer (type)                       Output Shape                    Param #
## =========================================================================
##  conv2d_3 (Conv2D)                  (None, 32, 32, 32)              896
##  max_pooling2d_3 (MaxPooling2D)     (None, 16, 16, 32)              0
##  conv2d_2 (Conv2D)                  (None, 16, 16, 64)              18496
##  max_pooling2d_2 (MaxPooling2D)     (None, 8, 8, 64)                0
##  conv2d_1 (Conv2D)                  (None, 8, 8, 128)               73856
##  max_pooling2d_1 (MaxPooling2D)     (None, 4, 4, 128)               0
##  conv2d (Conv2D)                    (None, 4, 4, 256)               295168
##  max_pooling2d (MaxPooling2D)       (None, 2, 2, 256)               0
##  flatten (Flatten)                  (None, 1024)                    0
##  dropout_3 (Dropout)                (None, 1024)                    0
```

```
##  dense_7 (Dense)                  (None, 512)                   524800
##  dense_6 (Dense)                  (None, 100)                   51300
## ================================================================================
## Total params: 964516 (3.68 MB)
## Trainable params: 964516 (3.68 MB)
## Non-trainable params: 0 (0.00 Byte)
## ________________________________________________________________________________
```

`layer_conv_2D()`において、`padding = "same"`引数を使用したことに注目する。これにより、出力チャネルは入力チャネルと同じ次元を持つことが保証される。入力層が 3 チャネルであるのに対して、最初の隠れ層には 32 チャネルある。すべての層で各チャネルに3×3の畳み込みフィルタを使用する。各畳み込みの後には2×2のブロックに対する最大プーリング層が続く。サマリーを調べると、各最大プーリング操作後にチャネルの次元が半分になっていることがわかる。これらの操作の最後には、256 チャネルで2×2の次元を持つ層が得られる。これらは平坦化され、1,024 のサイズを持つ密な層に変換される。つまり、各2×2の行列は4次元ベクトルに変換され、横に並べられて 1 層にまとめられる。その後、ドロップアウト正則化層が続き、512 のサイズを持つ別の密な層が続き、最終的にソフトマックス出力層に到達する。

最後に、学習アルゴリズムを指定し、モデルを学習させる。

```
model %>% compile(loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(), metrics = c("accuracy"))
#history <- model %>% fit(x_train, y_train, epochs = 30,
history <- model %>% fit(x_train, y_train, epochs = 10,
    batch_size = 128, validation_split = 0.2)
## Epoch 1/10
## 313/313 - 9s - loss: 4.2129 - accuracy: 0.0507 - val_loss: 3.9445 - val_accuracy: 0.0917 - 9s/ep
och - 28ms/step
訳者中略
## Epoch 9/10
## 313/313 - 9s - loss: 2.2378 - accuracy: 0.4093 - val_loss: 2.4324 - val_accuracy: 0.3781 - 9s/ep
och - 29ms/step
## Epoch 10/10
## 313/313 - 9s - loss: 2.1061 - accuracy: 0.4409 - val_loss: 2.4460 - val_accuracy: 0.3812 - 9s/ep
och - 28ms/step
model %>% predict(x_test) %>% k_argmax() %>% accuracy(g_test)
## 313/313 - 1s - 1s/epoch - 5ms/step
```

```
## [1] 0.397
```

このモデルは実行に 10 分かかり、テストデータに対して 46%の精度を達成する。この結果は 100 クラスのデータにしては悪くない（ランダム分類器では 1%の精度しか得られない）が、ウェブを調べると約 75%の結果が見られる。通常、このような結果を得るには、アーキテクチャの細かい調整、正則化の微調整、そして時間が必要である。

事前学習済み CNN モデルの使用

次に、`imagenet` データベースに対して事前学習された CNN を使用して自然画像を分類する方法を示し、図 10.10 を作成した方法を示す。デジタル写真アルバムから 6 枚の jpeg 画像をコピーし、ディレクトリ `book_images` に保存した。（これらの画像は、<www.statlearning.com>のデータセクションから入手でき、**book_images.zip** をダウンロードしクリックすると、**book_images** ディレクトリが作成される。）まず、画像を読み込み、`keras` ソフトウェアで扱われる `imagenet` の仕様に合わせた配列形式に変換する。作業ディレクトリが、画像が保存されているフォルダに設定されていることを確認する必要がある。

訳注 6：上記の通りだが、ここで利用するファイルは https://www.statlearning.com/resources-second-edition の Data Sets の中から book_images.zip より手に入れることができる。なおデータファイルは R の current directory（ISLR 第 2 章の訳注 5 を参照）に入れておく必要がある。

```
img_dir <- "book_images"
image_names <- list.files(img_dir)
num_images <- length(image_names)
x <- array(dim = c(num_images, 224, 224, 3))
for (i in 1:num_images) {
  img_path <- paste(img_dir, image_names[i], sep = "/")
  img <- image_load(img_path, target_size = c(224, 224))
  x[i,,, ] <- image_to_array(img)
}
x <- imagenet_preprocess_input(x)
```

次に、学習済みのネットワークを読み込む。このモデルは 50 層を持ち、かなりの複雑さを持っている。

```
model <- application_resnet50(weights = "imagenet")
summary(model)
## Model: "resnet50"
## ____________________________________________________________________________
##  Layer (type)       Output Shape         Para   Connected to         Trainable
##                                          m #
## ============================================================================
##  input_1 (InputLay  [(None, 224, 224,    0      []                   Y
##  er)                3)]
##  conv1_pad (ZeroPa  (None, 230, 230, 3   0      ['input_1[0][0]']    Y
##  dding2D)           )
##  conv1_conv (Conv2  (None, 112, 112, 6   9472   ['conv1_pad[0][0]'   Y
##  D)                 4)                         ]
訳者中略
##  (Add)                                          [0][0]',
##                                                  'conv5_block3_3_b
##                                                 n[0][0]']
##  conv5_block3_out   (None, 7, 7, 2048)   0      ['conv5_block3_add   Y
##  (Activation)                                   [0][0]']
##  avg_pool (GlobalA  (None, 2048)         0      ['conv5_block3_out   Y
##  veragePooling2D)                               [0][0]']
##  predictions (Dens  (None, 1000)         2049   ['avg_pool[0][0]']   Y
##  e)                                     000
## ============================================================================
## Total params: 25636712 (97.80 MB)
## Trainable params: 25583592 (97.59 MB)
## Non-trainable params: 53120 (207.50 KB)
## ____________________________________________________________________________
```

最後に、6 枚の画像を分類し、それぞれの予測確率に基づいてトップ 3 のクラス選択を返す。

```
pred6 <- model %>% predict(x) %>%
  imagenet_decode_predictions(top = 3)
## 1/1 - 1s - 516ms/epoch - 516ms/step
names(pred6) <- image_names
print(pred6)
## $flamingo.jpg
##   class_name class_description       score
```

```
## 1  n02007558          flamingo 0.926349998
## 2  n02006656         spoonbill 0.071699299
## 3  n02002556       white_stork 0.001228207
## $hawk.jpg
##   class_name class_description     score
## 1  n03388043          fountain 0.2788656
## 2  n03532672              hook 0.1785547
## 3  n03804744              nail 0.1080726
## $hawk_cropped.jpeg
##   class_name class_description      score
## 1  n01608432              kite 0.72270960
## 2  n01622779    great_grey_owl 0.08182576
## 3  n01532829       house_finch 0.04218856
## $huey.jpg
##   class_name           class_description      score
## 1  n02097474             Tibetan_terrier 0.50929761
## 2  n02098413                       Lhasa 0.42209852
## 3  n02098105 soft-coated_wheaten_terrier 0.01695854
## $kitty.jpg
##   class_name    class_description      score
## 1  n02105641 Old_English_sheepdog 0.83265990
## 2  n02086240             Shih-Tzu 0.04513895
## 3  n03223299              doormat 0.03299779
## $weaver.jpg
##   class_name class_description      score
## 1  n01843065           jacamar 0.49795347
## 2  n01818515             macaw 0.22193296
## 3  n02494079   squirrel_monkey 0.04287859
```

IMDb 文章分類

次に、`IMDB` データセット（10.4 節）に対して文章分類を行う。このデータセットは `keras` パッケージの一部として利用可能である。辞書サイズは最も頻繁に使用される 10,000 個の単語、トークンに制限する。

```
max_features <- 10000
imdb <- dataset_imdb(num_words = max_features)
c(c(x_train, y_train), c(x_test, y_test)) %<-% imdb
```

3 行目はリストのリストをアンパックするためのショートカットである。`x_train` の各要素は 0 から 9999 の間の数値のベクトル（文章）であり、辞書内の単語を指している。例えば、最初の訓練文章は 419 ページのポジティブなレビューである。最初の 12 語のインデックスは以下の通りである。

```
x_train[[1]][1:12]
##  [1]    1   14   22   16   43  530  973 1622 1385   65  458 4468
```

単語を見るために、辞書に対する簡単なインターフェースとなる `decode_review()` という関数を作成する。

```
word_index <- dataset_imdb_word_index()
decode_review <- function(text, word_index) {
  word <- names(word_index)
  idx <- unlist(word_index, use.names = FALSE)
  word <- c("<PAD>", "<START>", "<UNK>", "<UNUSED>", word)
  idx <- c(0:3, idx + 3)
  words <- word[match(text, idx, 2)]
  paste(words, collapse = " ")
}
decode_review(x_train[[1]][1:12], word_index)
## [1] "<START> this film was just brilliant casting location scenery story dir
ection everyone's"
```

次に、文章のリストから各文章を「One-Hot」エンコードし、バイナリ行列を疎（スパース）行列形式で返す関数を作成する。

```
library(Matrix)
one_hot <- function(sequences, dimension) {
  seqlen <- sapply(sequences, length)
  n <- length(seqlen)
  rowind <- rep(1:n, seqlen)
  colind <- unlist(sequences)
  sparseMatrix(i = rowind, j = colind,
```

```
      dims = c(n, dimension))
}
```

疎（スパース）行列を構築するに際し、`1`は非ゼロの要素であることのみを表す。最後の行では、`sparseMatrix()`関数を呼び出し、各文章に対応する行インデックスと各文章内の単語に対応する列インデックスを与える。値は省略されるため、すべて 1 として扱われる。同じ文章に複数回出現する単語も、1 として記録される。

```
x_train_1h <- one_hot(x_train, 10000)
x_test_1h <- one_hot(x_test, 10000)
dim(x_train_1h)
## [1] 25000 10000
nnzero(x_train_1h) / (25000 * 10000)
## [1] 0.01316987
```

全要素の 1.3%のみが非ゼロであるため、これはかなりのメモリ節約に繋がる。次に、サイズが 2,000 の検証セットを作成し、23,000 を訓練用に残す。

```
set.seed(3)
ival <- sample(seq(along = y_train), 2000)
```

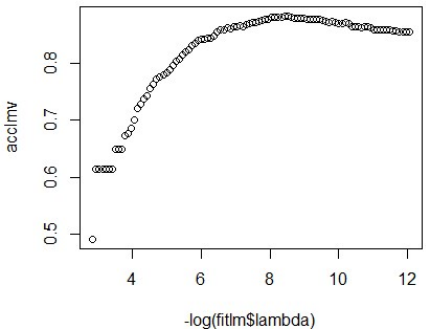
まず、`glmnet()`を使用して、訓練データに対してラッソ・ロジスティック回帰モデルを学習させ、検証データに対するパフォーマンスを評価する。最後に、縮小パラメータλの関数として精度`acclmv`をプロットする。同様にテストデータに対するパフォーマンスを計算し、これらは図 10.11 の左側のプロットを作成するために使用された。このコードは`x_train_1h`の疎（スパース）行列形式を活用し、約 5 秒で実行される。通常の密な形式では、約 5 分かかる。

```
library(glmnet)
fitlm <- glmnet(x_train_1h[-ival, ], y_train[-ival],
    family = "binomial", standardize = FALSE)
classlmv <- predict(fitlm, x_train_1h[ival, ]) > 0
acclmv <- apply(classlmv, 2, accuracy,  y_train[ival] > 0)
```

`accuracy()`関数は 10.9.2 節で作成したもので、予測行列 `classlmv` の各列に適用される。この行列は論理行列であり、`TRUE/FALSE` の値を持つため、2 番目の引数 `truth` も論理ベクトルとして与えられる。

プロットを作成する前に、プロットウィンドウを調整する。

```
par(mar = c(4, 4, 4, 4), mfrow = c(1, 1))
plot(-log(fitlm$lambda), acclmv)
```



次に、2 つの隠れ層を持つ全結合ニューラルネットワークを学習させる。各隠れ層は 16 ユニットを持ち、ReLU 活性化関数を使用する。

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16, activation = "relu",
      input_shape = c(10000)) %>%
  layer_dense(units = 16, activation = "relu") %>%
  layer_dense(units = 1, activation = "sigmoid")
model %>% compile(optimizer = "rmsprop",
    loss = "binary_crossentropy", metrics = c("accuracy"))
history <- model %>% fit(x_train_1h[-ival, ], y_train[-ival],
    epochs = 20, batch_size = 512,
    validation_data = list(x_train_1h[ival, ], y_train[ival]))
## Epoch 1/20
## 45/45 - 1s - loss: 0.4730 - accuracy: 0.8141 - val_loss: 0.3593 - val_accura
cy: 0.8640 - 601ms/epoch - 13ms/step
訳者中略
```

```
## Epoch 19/20
## 45/45 - 0s - loss: 0.0352 - accuracy: 0.9894 - val_loss: 0.5724 - val_accura
cy: 0.8660 - 244ms/epoch - 5ms/step
## Epoch 20/20
## 45/45 - 0s - loss: 0.0213 - accuracy: 0.9962 - val_loss: 0.6263 - val_accura
cy: 0.8545 - 251ms/epoch - 6ms/step
```

`history`オブジェクトには、各エポックでの訓練精度と検証精度を記録する`metrics`コンポーネントがある。図 10.11 には、各エポックでのテスト精度も含まれている。テスト精度を計算するためには、上記のコードブロックの最後の行を以下のように置き換え、再実行する。

```
history <- model %>% fit(
    x_train_1h[-ival, ], y_train[-ival], epochs = 20,
    batch_size = 512, validation_data = list(x_test_1h, y_test)
  )
## Epoch 1/20
## 45/45 - 0s - loss: 0.0286 - accuracy: 0.9913 - val_loss: 0.6885 - val_accuracy: 0.8532 - 496ms/e
poch - 11ms/step
訳者中略
## Epoch 19/20
## 45/45 - 0s - loss: 0.0012 - accuracy: 1.0000 - val_loss: 1.1228 - val_accuracy: 0.8512 - 431ms/e
poch - 10ms/step
## Epoch 20/20
## 45/45 - 0s - loss: 0.0091 - accuracy: 0.9972 - val_loss: 1.1314 - val_accuracy: 0.8516 - 386ms/e
poch - 9ms/step
```

再帰型ニューラルネットワーク(Recurrent Neural Networks, RNN)

この実習では、10.5 節で扱われたモデルを学習させる。

文章分類のための逐次モデル

ここでは、10.5.1 節で説明したように、`IMDb` 映画レビューのデータを用いて感情分析のためのシンプルな LSTM RNN を学習させる。データの入力方法については 10.9.5 節で既に説明したので、ここでは繰り返さない。

まず、文章の長さを計算しよう。

```
wc <- sapply(x_train, length)
median(wc)
## [1] 178
sum(wc <= 500) / length(wc)
## [1] 0.91568
```

91%以上の文章が 500 語未満であることがわかる。ここでの RNN は、すべての文章系列が同じ長さである必要がある。そのため、文章の長さを最後の$L = 500$語に制限し、短い文章の始めは空白でパディングする。

```
maxlen <- 500
x_train <- pad_sequences(x_train, maxlen = maxlen)
x_test <- pad_sequences(x_test, maxlen = maxlen)
dim(x_train)
## [1] 25000   500
dim(x_test)
## [1] 25000   500
x_train[1, 490:500]
##  [1]   16 4472  113  103   32   15   16 5345   19  178   32
```

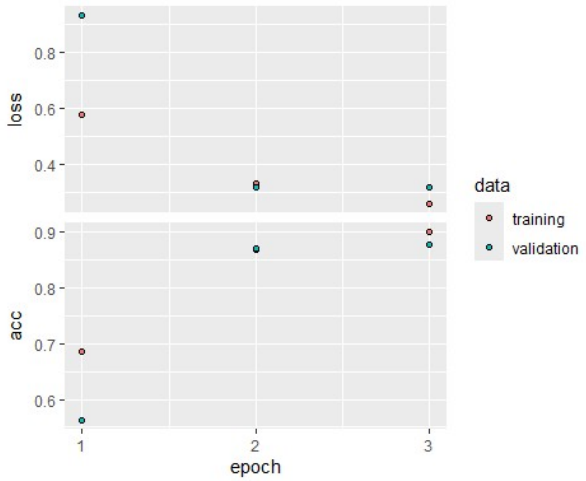
最後の表現は、最初の文章の最後のいくつかの単語を示している。この時点で、文章の各 500 語は、その単語が 10,000 語の辞書内でどの位置にあるかに対応する整数を使用して表現されている。RNN の最初の層は、32 のサイズを持つ埋め込み層であり、訓練中に学習される。この層は、各文章を$500 \times 10{,}000$の次元の行列として One-Hot エンコードし、これらの10,000次元を32次元に削減する。

```
model <- keras_model_sequential() %>%
  layer_embedding(input_dim = 10000, output_dim = 32) %>%
  layer_lstm(units = 32) %>%
  layer_dense(units = 1, activation = "sigmoid")
```

2 番目の層は 32 ユニットの LSTM で、出力層は 2 項分類タスクのためのシグモイドユニット 1 つである。

残りは、これまでに学習させた他のネットワークと似ている。ネットワークを学習する間、テストパフォーマンスを追跡し、87%の精度に達することがわかる。

```
model %>% compile(optimizer = "rmsprop",
    loss = "binary_crossentropy", metrics = c("acc"))
#history <- model %>% fit(x_train, y_train, epochs = 10,
history <- model %>% fit(x_train, y_train, epochs = 3,
    batch_size = 128, validation_data = list(x_test, y_test))
## Epoch 1/3
## 196/196 - 35s - loss: 0.5754 - acc: 0.6859 - val_loss: 0.9322 - val_acc: 0.5634 - 35s/epoch - 17
6ms/step
## Epoch 2/3
## 196/196 - 27s - loss: 0.3312 - acc: 0.8672 - val_loss: 0.3186 - val_acc: 0.8706 - 27s/epoch - 13
7ms/step
## Epoch 3/3
## 196/196 - 28s - loss: 0.2589 - acc: 0.8995 - val_loss: 0.3181 - val_acc: 0.8757 - 28s/epoch - 14
3ms/step
plot(history)
```




```
predy <- predict(model, x_test) > 0.5
## 782/782 - 21s - 21s/epoch - 27ms/step
mean(abs(y_test == as.numeric(predy)))
## [1] 0.87572
```

時系列予測

次に、10.5.2 節で説明されたモデルを使用して時系列予測を行う方法を示す。まずデータを設定し、各変数を標準化する。

```
library(ISLR2)
xdata <- data.matrix(
    NYSE[, c("DJ_return", "log_volume","log_volatility")]
  )
istrain <- NYSE[, "train"]
xdata <- scale(xdata)
```

変数 `istrain` は、訓練セットに含まれる各年に対して `TRUE`、テストセットに含まれる各年に対して `FALSE` をとる。

次に、3 つの時系列のラグ(遅れ)変数を作成する関数を書く。最初に、データ行列とラグLを入力として受け取り、ラグ変数の行列を返す関数を作成する。この関数は、単純にL行の `NA` を上に挿入し、下部を切り捨てる。

```
lagm <- function(x, k = 1) {
  n <- nrow(x)
  pad <- matrix(NA, k, ncol(x))
  rbind(pad, x[1:(n - k), ])
}
```

この関数を使用して、必要なすべてのラグ変数と応答変数を含むデータフレームを作成する。

```
arframe <- data.frame(log_volume = xdata[, "log_volume"],
   L1 = lagm(xdata, 1), L2 = lagm(xdata, 2),
   L3 = lagm(xdata, 3), L4 = lagm(xdata, 4),
   L5 = lagm(xdata, 5)
 )
```

このフレームの最初の 5 行を確認すると、ラグ変数に欠損値があることがわかる（上記の構築による）。これらの行を削除し、`istrain` も適宜調整する。

```
arframe <- arframe[-(1:5), ]
istrain <- istrain[-(1:5)]
```

次に、`lm()`を使用して訓練データで線形 AR モデルを学習させ、テストデータで予測を行う。

```
arfit <- lm(log_volume ~ ., data = arframe[istrain, ])
arpred <- predict(arfit, arframe[!istrain, ])
V0 <- var(arframe[!istrain, "log_volume"])
1 - mean((arpred - arframe[!istrain, "log_volume"])^2) / V0
## [1] 0.413223
```

最後の 2 行は、テストデータに対するR^2を計算する（(3.17)に定義されている通りである）。

このモデルに因子変数 `day_of_week` を含めて、再度学習させる。

```
arframed <-
    data.frame(day = NYSE[-(1:5), "day_of_week"], arframe)
arfitd <- lm(log_volume ~ ., data = arframed[istrain, ])
arpredd <- predict(arfitd, arframed[!istrain, ])
1 - mean((arpredd - arframe[!istrain, "log_volume"])^2) / V0
## [1] 0.4598616
```

RNN を学習させるために、これらのデータを再形成する必要がある。RNN では各観測について、$L = 5$の特徴ベクトルの系列$X = \{X_\ell\}_1^L$の形にする必要がある（式(10.20)、428 ページ参照）。これらは時系列のラグ変数で、1からLまで時間を遡ったものである。

```
n <- nrow(arframe)
xrnn <- data.matrix(arframe[, -1])
xrnn <- array(xrnn, c(n, 3, 5))
xrnn <- xrnn[,, 5:1]
xrnn <- aperm(xrnn, c(1, 3, 2))
dim(xrnn)
## [1] 6046    5    3
```

これを 4 つのステップで行った。最初のステップは、`arframe` から 3 つの予測変数のラグバージョンの$n \times 15$の行列を抽出する。2 番目のステップは、この行列を$n \times 3 \times 5$の配列に変換する。この変換は、単に次元属性を変更することで行う。新しい配列は列方向に埋められる。3 番目のステップは、ラグ変数の順序を逆にすることによって、インデックス1が最も遡った時間となり、インデックス5が最も近い時間となるようにする。最後のステップは、配列の座標を（部分転置のように）再配置して、`keras` の RNN モジュールで利用できる形式にする。

これで、12 ユニットの隠れ層を持つ RNN の準備が整った。

```
model <- keras_model_sequential() %>%
  layer_simple_rnn(units = 12,
      input_shape = list(5, 3),
      dropout = 0.1, recurrent_dropout = 0.1) %>%
  layer_dense(units = 1)
model %>% compile(optimizer = optimizer_rmsprop(),
    loss = "mse")
```

ここでは、隠れ層に入力されるユニットに対して 2 種類のドロップアウトを指定している。1 つはこの隠れ層に入力される入力系列用、もう 1 つはこの隠れ層に再帰的に入力される隠れユニット用である。出力層は、応答のためのユニット 1 つを持っている。

このモデルは、これまでのネットワークと同様の方法で学習させる。`fit` 関数にテストデータを検証データとして与え、その進行状況をモニタリングし、`history` 関数をプロットしてテストデータ上の進行状況を見ることができる。もちろん、これは早期打ち切りの基準として使用すべきではない。なぜなら、テスト性能にバイアスが現れるからである。

```
history <- model %>% fit(
    xrnn[istrain,, ], arframe[istrain, "log_volume"],
#    batch_size = 64, epochs = 200,
    batch_size = 64, epochs = 75,
    validation_data =
      list(xrnn[!istrain,, ], arframe[!istrain, "log_volume"])
  )
## Epoch 1/75
## 67/67 - 1s - loss: 1.0234 - val_loss: 0.9051 - 791ms/epoch - 12ms/step
訳者中略
## Epoch 74/75
## 67/67 - 0s - loss: 0.4476 - val_loss: 0.6279 - 193ms/epoch - 3ms/step
## Epoch 75/75
## 67/67 - 0s - loss: 0.4548 - val_loss: 0.6216 - 177ms/epoch - 3ms/step
kpred <- predict(model, xrnn[!istrain,, ])
## 56/56 - 0s - 133ms/epoch - 2ms/step
1 - mean((kpred - arframe[!istrain, "log_volume"])^2) / V0
```

```
## [1] 0.4103281
```

このモデルの訓練には約 1 分かかる。

次のように、上記の `keras_model_sequential()`コマンドを置き換えることもできる。

```
model <- keras_model_sequential() %>%
  layer_flatten(input_shape = c(5, 3)) %>%
  layer_dense(units = 1)
```

ここで、`layer_flatten()`は単に、入力系列を受け取り、それを予測変数の長いベクトルに変換する。この結果、線形 AR モデルが得られる。非線形 AR モデルを学習させるために、隠れ層を追加することができる。

しかし、既に `lm()`コマンドを使用して学習させた AR モデルのラグ変数の行列があるため、実際には平坦化を行わなくても非線形 AR モデルを学習させることができる。 `arframed` からモデル行列 `x` を抽出する。これには `day_of_week` 変数も含まれている。

```
x <- model.matrix(log_volume ~ . - 1, data = arframed)
colnames(x)
##  [1] "dayfri"            "daymon"            "daythur"
##  [4] "daytues"           "daywed"            "L1.DJ_return"
##  [7] "L1.log_volume"     "L1.log_volatility" "L2.DJ_return"
## [10] "L2.log_volume"     "L2.log_volatility" "L3.DJ_return"
## [13] "L3.log_volume"     "L3.log_volatility" "L4.DJ_return"
## [16] "L4.log_volume"     "L4.log_volatility" "L5.DJ_return"
## [19] "L5.log_volume"     "L5.log_volatility"
```

式の中の`-1` は、切片のための 1 の列が作成されるのを避ける。`day_of_week` 変数は 5 つのレベルを持つ因子（取引日は 5 日間）であり、`-1` により、ダミー変数は 4 つではなく 5 つになる。

非線形 AR モデルを学習させるための残りのステップは、今や馴染みがあるはずである。

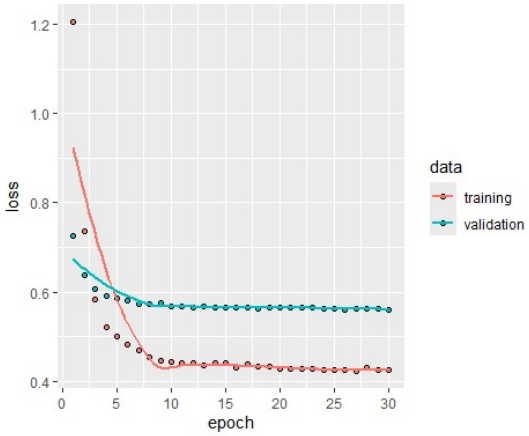
```
arnnd <- keras_model_sequential() %>%
  layer_dense(units = 32, activation = 'relu',
      input_shape = ncol(x)) %>%
  layer_dropout(rate = 0.5) %>%
  layer_dense(units = 1)
arnnd %>% compile(loss = "mse",
    optimizer = optimizer_rmsprop())
history <- arnnd %>% fit(
#    x[istrain, ], arframe[istrain, "log_volume"], epochs = 100,
    x[istrain, ], arframe[istrain, "log_volume"], epochs = 30,
    batch_size = 32, validation_data =
      list(x[!istrain, ], arframe[!istrain, "log_volume"])
  )
```

```
## Epoch 1/30
## 134/134 - 0s - loss: 1.2056 - val_loss: 0.7248 - 457ms/epoch - 3ms/step
```

訳者中略

```
## Epoch 29/30
## 134/134 - 0s - loss: 0.4251 - val_loss: 0.5638 - 251ms/epoch - 2ms/step
## Epoch 30/30
## 134/134 - 0s - loss: 0.4260 - val_loss: 0.5606 - 243ms/epoch - 2ms/step
```

```
plot(history)
```




```
npred <- predict(arnnd, x[!istrain, ])
```

```
## 56/56 - 0s - 73ms/epoch - 1ms/step
1 - mean((arframe[!istrain, "log_volume"] - npred)^2) / V0
## [1] 0.4681557
```

（訳者）追加事項：第 10 章のための Keras のインストール

はじめに

新しいバージョンの `keras` および `tensorflow` パッケージでは、インストールプロセスが大幅に簡素化されている。`Python3` の任意のインストールを使用できるが、`conda` を利用することを推奨する。これは `R` パッケージとの相性が良いためである。以下の 2 つの方法のうち、ご自身の環境に適したものを選択する必要がある。

1. 既存の conda インストールがある場合

すでに `conda` がインストールされていて、`r-tensorflow` という環境を作成済みの場合は、以下のコマンドを実行する。

```
install.packages("keras")
keras::install_keras(method = "conda", python_version = "3.10")
```

最初のコマンドで `R` の `keras` パッケージとその依存関係をインストールし、2 つ目のコマンドで `conda` 環境に `Python` の `keras` パッケージをインストールする。ここで `Python` のバージョンを指定することが重要である。 これは、最新の `Python` バージョンには対応する `tensorflow` パッケージが提供されていない可能性があるためである。

2．conda がインストールされていない場合

この場合は、`conda` 自体をインストールする必要がある。

```
install.packages("keras")
```

```
reticulate::install_miniconda()
keras::install_keras(method = "conda", python_version = "3.10")
```

この方法では、`keras`の依存パッケージである`reticulate`を利用して、デフォルトで`r-tensorflow`という環境に`conda`をインストールする。`Python`のバージョン指定についての注意点は、前述の方法と同様である。

Keras 実習コードの小さな変更点

`keras`の新バージョンに対応するため、Ch10-deeplearning-lab-keras 実習では以下の変更を加える必要がある。`library(keras)`の直後に、以下のコードを追加する。

```
library(keras)
reticulate::use_condaenv(condaenv = "r-tensorflow")
```

これは、事前学習済みモデル（pretrained models）を使用するセクションで必要になる。

訳注 7 ：Ch10-deeplearning-lab-keras 実習とは、本節「ISLR 第 10 章 実習：深層学習 1」のことである。この翻訳の本文は上記の変更を反映して行った。

ISLR 第 10 章 実習：深層学習 2 (Deep Learning, Torch 版)

本節では、教科書(ISL)で議論された例をどのように学習させるかを説明する。`luz` パッケージを使用する。このパッケージは `torch` パッケージへのインターフェースを提供しており、`torch` パッケージは `LibTorch` ライブラリの効率的な `C++` コードへとリンクする。

この Torch 版の実習は、これらのパッケージを作成した **Rstudio** のデータサイエンティストである Daniel Falbel 氏と Sigrid Keydana 氏によって作成された。元の `keras` 版(実習：深層学習 1)に比べた `Torch` 版の利点は、別途 `python` をインストールする必要がないことである。

訳注 1：Neural Networks, Deep Learning の実装に関する方法、例えば pooling(プーリング)や dropout(ドロップ・アウト)などについては例えば前述の「R と Keras によるディープラーニング」の他、「あたらしい機械学習の教科書」(伊藤真,翔泳社)などによる説明が参考になる。

打者(Hitters)データに対する単層ネットワーク

10.6 節 で示されたモデルを学習しよう。 データを準備し、訓練(トレーニング)セットとテストセットに分割する。

```
library(ISLR2)
```

```
## Warning: パッケージ 'ISLR2' はバージョン 4.4.3 の R の下で造られました
```

```
Gitters <- na.omit(Hitters)
n <- nrow(Gitters)
set.seed(13)
ntest <- trunc(n / 3)
testid <- sample(1:n, ntest)
```

線形モデルは馴染みがあるはずだが、ここでも改めて提示する。

```
lfit <- lm(Salary ~ ., data = Gitters[-testid, ])
lpred <- predict(lfit, Gitters[testid, ])
with(Gitters[testid, ], mean(abs(lpred - Salary)))
## [1] 254.6687
```

ここで `with()` コマンドの使用に注目する。 第 1 引数にはデータフレームを指定し、第 2 引数にはデータフレームの要素を名前で参照した式を指定する。 この場合、データフレームはテストデータに対応しており、ここではテストデータにおける平均絶対予測誤差を計算する。

次に、`glmnet` を用いて Lasso(ラッソ)回帰を学習する。 このパッケージは `formula` を使用しないため、まず `x` と `y` を作成する。

```
x <- scale(model.matrix(Salary ~ . - 1, data = Gitters))
y <- Gitters$Salary
```

最初の行では `model.matrix()` を呼び出している。これは `lm()` で使用されるのと同じ行列を生成する（`-1` を指定することで切片を省略している）。この関数は因子型変数を自動的にダミー変数へ変換する。`scale()` 関数は行列を標準化し、各列の平均を 0、分散を 1 にする。

```
library(glmnet)
## Warning: パッケージ 'glmnet' はバージョン 4.4.3 の R の下で造られました
## 要求されたパッケージ Matrix をロード中です
## Loaded glmnet 4.1-8
cvfit <- cv.glmnet(x[-testid, ], y[-testid],
    type.measure = "mae")
cpred <- predict(cvfit, x[testid, ], s = "lambda.min")
mean(abs(y[testid] - cpred))
## [1] 252.2994
```

ニューラルネットワークモデルを学習するために、まずネットワークの構造を定義する。

```
library(torch)
```

```
## Warning: パッケージ 'torch' はバージョン 4.4.3 の R の下で造られました
```

```
library(luz) # Torch に対する高レベルのインターフェース
```

```
## Warning: パッケージ 'luz' はバージョン 4.4.3 の R の下で造られました
```

```
library(torchvision) # データセットと画像変換のため
```

```
## Warning: パッケージ 'torchvision' はバージョン 4.4.3 の R の下で造られました
```

```
library(torchdatasets) # 使用する予定のデータセットのため
```

```
## Warning: パッケージ 'torchdatasets' はバージョン 4.4.3 の R の下で造られました
```

```
library(zeallot)
```

```
## Warning: パッケージ 'zeallot' はバージョン 4.4.3 の R の下で造られました
```

```
torch_manual_seed(13)
modnn <- nn_module(
  initialize = function(input_size) {
    self$hidden <- nn_linear(input_size, 50)
    self$activation <- nn_relu()
    self$dropout <- nn_dropout(0.4)
    self$output <- nn_linear(50, 1)
  },
  forward = function(x) {
    x %>%
      self$hidden() %>%
      self$activation() %>%
      self$dropout() %>%
      self$output()
  }
)
```

`modnn`というモデルを作成した。これは、まず`initialize()`と`forward()`関数を定義し、それらを`nn_module()`関数に渡すことで構築される。`initialize()`関数は、モデルが使用するサブモジュールの初期化を担っている。`forward`メソッドでは、モデ

ルが入力データに対して呼び出されたときの作業を実装する。この場合、`initialize()`で定義した層をその特定の順序で利用する。

`self`は、`nn_module()`のメソッド間で情報を共有するために使用されるリストのような特別なオブジェクトである。`initialize()`でオブジェクトを`self`に割り当てると、`forward()`でアクセスできるようになる。

パイプ（`pipe`）演算子`%>%`は、前の処理結果を次の関数の第 1 引数として渡し、その結果を返す。

パイプ演算子の使用例として、単純なケースを示す。先ほど、`x`を作成するために以下のコードを用いた。

```
x <- scale(model.matrix(Salary ~ . - 1, data = Gitters))
```

まず行列を作成し、次に各変数をセンタリングして標準化している。このような複合的な表現は構文解析を難しくすることがある。 同じ結果を、パイプ演算子を使って得ることもできる。

```
x <- model.matrix(Salary ~ . - 1, data = Gitters) %>% scale()
```

パイプ演算子を使用すると、一連の処理の流れを追いやすくなる。

ここで再びニューラルネットワークに戻る。オブジェクト`modnn`は、50 個のユニットを持つ単一の隠れ層を持ち、活性化関数として ReLU を使用する。
続くドロップアウト層では、確率的勾配降下法 (SGD) アルゴリズムの各イテレーションにおいて、前の層の 50 個のアクティベーションのうち 40% がランダムに 0 に設定される。
最後の出力層には 1 つのユニットがあり、活性化関数を持たないため、単一の数値を出力するモデルとなる。

次に、`modnn`に学習アルゴリズムの詳細を追加しよう。

ここでは、(10.23)の二乗誤差損失を最小化するように設定する。
また、訓練データにおける平均絶対誤差 (MAE) を追跡し、バリデーションデータが提供される場合にバリデーションデータに対するものも記録する。

```
modnn <- modnn %>%
  setup(
```

```
    loss = nn_mse_loss(),
    optimizer = optim_rmsprop,
    metrics = list(luz_metric_mae())
  ) %>%
  set_hparams(input_size = ncol(x))
```

上記のコードでは、パイプ演算子 `\%>\%` を使用し、`modnn` を `setup()`関数に渡している。 `setup()`関数は、これらの仕様を新しいモデルオブジェクトに組み込む。さらに、`set_hparams()`を使用して、`modnn` の `initialize()`メソッドに渡すべき引数を指定している。

次に、モデルの学習を行う。
訓練データと `epochs`（エポック数）を指定する。
デフォルトでは、SGD の各ステップでランダムに 32 個の訓練データを選択し、勾配計算を行う。

10.4 節 および 10.7 節 で説明したように、1 エポックは n 個の観測データを 1 回処理するために必要な SGD ステップ数に対応する。
今回の訓練データのサンプル数は $n = 176$ であり、1 エポックは $176/32 = 5.5$回の SGD ステップとなる。

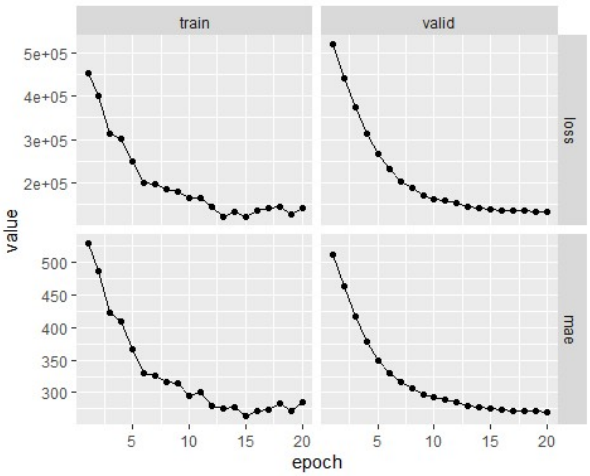
`fit()` 関数の `validation_data` 引数に引き渡されたデータは学習には使われず、モデルの学習の進捗状況を確認のために使用できる（ここでは 平均絶対誤差(MAE) を記録）。ここでは実際にテストデータを与え、エポックが進むごとに訓練データとテストデータに対する平均絶対誤差が表示される。 なお学習の詳細オプションについては、`?fit.luz_module_generator` を参照するとよい。

```
fitted <- modnn %>%
  fit(
    data = list(x[-testid, ], matrix(y[-testid], ncol = 1)),
    valid_data = list(x[testid, ], matrix(y[testid], ncol = 1)),
    epochs = 20 # 50
  )
```

(ここでは実行時間を管理しやすくするためにエポック数を 20 に減らしているが、もちろんユーザーが値を設定できる。)

`fitted` モデルをプロットして、訓練データとテストデータの平均絶対誤差を表示できる。

```
plot(fitted)
```



最後に、学習済みモデルを用いて予測を行い、テストデータ上での性能を評価する。SGD を使用しているため、学習のたびに結果がわずかに異なる。

```
npred <- predict(fitted, x[testid, ])
mean(abs(y[testid] - as.matrix(npred)))
## [1] 270.3537
```

なおこの `predict` メソッドは `torch_tensor` クラスのオブジェクトを返すため、`npred` オブジェクトを行列に変換する必要があった。

```
class(npred)
## [1] "torch_tensor" "R7"
```

MNIST 手書き数字データに対する多層ニューラルネットワーク

`torchvision` パッケージにはいくつかの例示用データセットが含まれており、その中に `MNIST` 数字データも含まれている。最初のステップとして、`MNIST` データを読み込む。この目的のために、`dataset_mnist()` 関数が用意されている。

この関数は、`dataset()`を返す。この `dataset()`は、データがどこに保存されているかやどのように整理されているかに関する仮定に依存することなく、`torch` で任意のデータセットを表現できるデータ構造である。通常、`torch` のデータセットはデータ取得プロセスも実装しており、例えばファイルのダウンロードやディスクへのキャッシュなどを行うことができる。

```
train_ds <- mnist_dataset(root = ".", train = TRUE, download = TRUE)
test_ds <- mnist_dataset(root = ".", train = FALSE, download = TRUE)

str(train_ds[1])
## List of 2
##  $ x: int [1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
##  $ y: int 6
str(test_ds[2])
## List of 2
##  $ x: int [1:28, 1:28] 0 0 0 0 0 0 0 0 0 0 ...
##  $ y: int 3
length(train_ds)
## [1] 60000
length(test_ds)
## [1] 10000
```

訓練データには 60,000 枚、テストデータには 10,000 枚の画像が含まれている。画像のサイズは 28×28 ピクセルであり、ピクセルの行列として保存されている。これらの画像をそれぞれベクトルに変換する必要がある。

ニューラルネットワークは入力のスケールに対してやや敏感である。
例えば、Ridge（リッジ）や Lasso（ラッソ）正則化もスケーリングの影響を受け

る。
ここでの入力データは 8 ビットのグレースケール値であり、0 から 255 の範囲をとるため、[0,1] の範囲にリスケールする。
（なお 8 ビットとは $2^8 = 256$ であり、通常 0 から始まるため、可能な値の範囲は 0 から 255 となる。）

これらの変換を適用するために、`train_ds` と `test_ds` を再定義し、`transform` 引数を渡して、各画像入力に変換を適用する。

```
transform <- function(x) {
  x %>%
    torch_tensor() %>%
    torch_flatten() %>%
    torch_div(255)
}
train_ds <- mnist_dataset(
  root = ".",
  train = TRUE,
  download = TRUE,
  transform = transform
)
test_ds <- mnist_dataset(
  root = ".",
  train = FALSE,
  download = TRUE,
  transform = transform
)
```

これでニューラルネットワークの学習を行う準備が整った。

```
modelnn <- nn_module(
  initialize = function() {
    self$linear1 <- nn_linear(in_features = 28*28, out_features = 256)
    self$linear2 <- nn_linear(in_features = 256, out_features = 128)
    self$linear3 <- nn_linear(in_features = 128, out_features = 10)
```

```
    self$drop1 <- nn_dropout(p = 0.4)
    self$drop2 <- nn_dropout(p = 0.3)

    self$activation <- nn_relu()
  },
  forward = function(x) {
    x %>%

      self$linear1() %>%
      self$activation() %>%
      self$drop1() %>%

      self$linear2() %>%
      self$activation() %>%
      self$drop2() %>%

      self$linear3()
  }
)
```

ここでは`nn_module()`の`initialize()`と`forward()`メソッドを定義した。

`initialize`では、モデルで使用されるすべての層を指定している。例えば、`nn_linear(784, 256)`は、$28 \times 28 = 784$の入力ユニットから256の隠れユニットへの密な層を定義する。モデルにはこれらが 3 つあり、それぞれ出力ユニットの数を減少させる。最後の層は 10 の出力ユニットを持ち、各ユニットは異なるクラスに対応し、10 クラスの分類問題となる。また、`nn_dropout()`を使用してドロップアウト層も定義した。これによりドロップアウト正則化が行われる。最後に、`nn_relu()`を使用して活性化層を定義している。

`forward()`では、これらの層が呼び出される順番を定義している。層は（線形、活性化、ドロップアウト）の順でブロックとして呼び出されるが、最後の層は活性化関数やドロップアウトを利用しない。

最後に、`print`を使ってモデルの概要を要約し、正しく定義されたことを確認する。

```
print(modelnn())
## An `nn_module` containing 235,146 parameters.
## ── Modules ──────────────────────────────────────────────────────────────
## • linear1: <nn_linear> #200,960 parameters
## • linear2: <nn_linear> #32,896 parameters
## • linear3: <nn_linear> #1,290 parameters
## • drop1: <nn_dropout> #0 parameters
## • drop2: <nn_dropout> #0 parameters
## • activation: <nn_relu> #0 parameters
```

各層のパラメータには定数項（バイアス項）も含まれるため、モデル全体のパラメータ数は 235,146 となる。
例えば、第 1 隠れ層では以下のように計算される：$(784 + 1) \times 256 = 200{,}960$

次に、学習アルゴリズムの詳細をモデルに追加する。交差エントロピー (10.14) を最小化することでモデルを学習する。

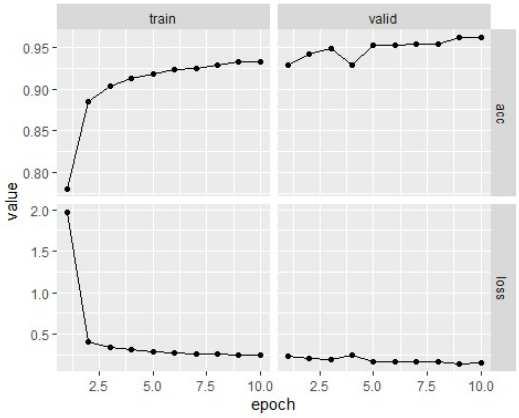
`torch` では、数値的安定性とメモリ効率のために交差エントロピー関数はロジットを基に定義されている。これにより、ターゲットは One-Hot エンコーディングされている必要はない。

```
modelnn <- modelnn %>%
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_rmsprop,
    metrics = list(luz_metric_accuracy())
  )
```

準備が整った。最終的なステップは、訓練データを与えて、モデルを学習させることである。

```
system.time(
   fitted <- modelnn %>%
      fit(
        data = train_ds,
```

```
        epochs = 10, #15,
        valid_data = 0.2,
        dataloader_options = list(batch_size = 256),
        verbose = TRUE
      )
 )
## Epoch 1/10
## Train metrics: Loss: 1.9673 - Acc: 0.7797
## Valid metrics: Loss: 0.2368 - Acc: 0.9281
訳者中略
## Epoch 9/10
## Train metrics: Loss: 0.2465 - Acc: 0.932
## Valid metrics: Loss: 0.1355 - Acc: 0.9613
## Epoch 10/10
## Train metrics: Loss: 0.2431 - Acc: 0.9331
## Valid metrics: Loss: 0.1454 - Acc: 0.9613
##    ユーザ システム      経過
##    99.29    15.70   100.40
plot(fitted)
```



これはモデルの学習に関する進捗報告であり、エポックごとにグループ化されている。この情報は非常に有用である。大規模なデータセットでは、学習に時間がかかることがあるためである。このモデルの学習には、2.7 GHz の MacBook Pro

（4 コア、16 GB RAM）で 215 秒を要した。ここでは 20%の検証分割を指定しているため、実際の訓練は訓練セットの 60,000 件の観測データの 80%で行われている。これは、10.9.1 節で行ったように、実際に検証データを与える代わりの方法である。オプション引数については、`?fit.luz_module_generator` を参照せよ。SGD は、勾配を計算する際に 256 の観測データのバッチを使用しており、計算結果から 1 エポックが 188 回の勾配ステップに対応していることがわかる。最後の `plot()` コマンドは、図 10.18 に似た図を生成する。

表 10.1 のテスト誤差を得るには、予測されたクラスラベルと真のクラスラベルを比較する簡単な関数 `accuracy()` をまず作成し、それを使って予測を評価する。

```
accuracy <- function(pred, truth) {
   mean(pred == truth) }


# test_ds のすべての観測から真のクラスを取得
truth <- sapply(seq_along(test_ds), function(x) test_ds[x][[2]])


fitted %>%
  predict(test_ds) %>%
  torch_argmax(dim = 2) %>%  # 予測されたクラスは、最も高い'logit'を持つもの
  as_array() %>% # R オブジェクトに変換
  accuracy(truth)
## [1] 0.9611
```

表 10.1 には、LDA（第 4 章）および多項ロジスティック回帰も報告されている。`glmnet` のようなパッケージは多項ロジスティック回帰を処理できるが、この大規模なデータセットでは非常に遅くなる。そのため、`luz` ソフトウェアを使用してモデルを学習させる方がはるかに速く、簡単である。ここでは、入力層と出力層だけを持ち、隠れ層は省略する。

```
modellr <- nn_module(
  initialize = function() {
    self$linear <- nn_linear(784, 10)
  },
  forward = function(x) {
```

```
    self$linear(x)
  }
)
print(modellr())
## An `nn_module` containing 7,850 parameters.
## ── Modules ──────────────────────────────────────────────────────────────────
## • linear: <nn_linear> #7,850 parameters
```

モデルは以前と同じように学習させる。

```
fit_modellr <- modellr %>%
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_rmsprop,
    metrics = list(luz_metric_accuracy())
  ) %>%
  fit(
    data = train_ds,
    epochs = 5,
    valid_data = 0.2,
    dataloader_options = list(batch_size = 128)
  )

fit_modellr %>%
  predict(test_ds) %>%
  torch_argmax(dim = 2) %>%  # 予測されたクラスは、最も高い'logit'を持つもの
  as_array() %>% # R オブジェクトに変換
  accuracy(truth)
## [1] 0.9163
# 代わりに'evaluate'関数を使って、test_ds での結果を得ることもできる
evaluate(fit_modellr, test_ds)
```

```
## A `luz_module_evaluation`
## ── Results ──────────────────────────────────────────────────────────────
## loss: 0.3111
## acc: 0.9163
```

畳み込みニューラルネットワーク(Convolutional Neural Networks, CNN)

この節では、`torchvision`パッケージで利用可能な`CIFAR`データに対して CNN を学習させる。これは、`MNIST`データと似た形式になっている。

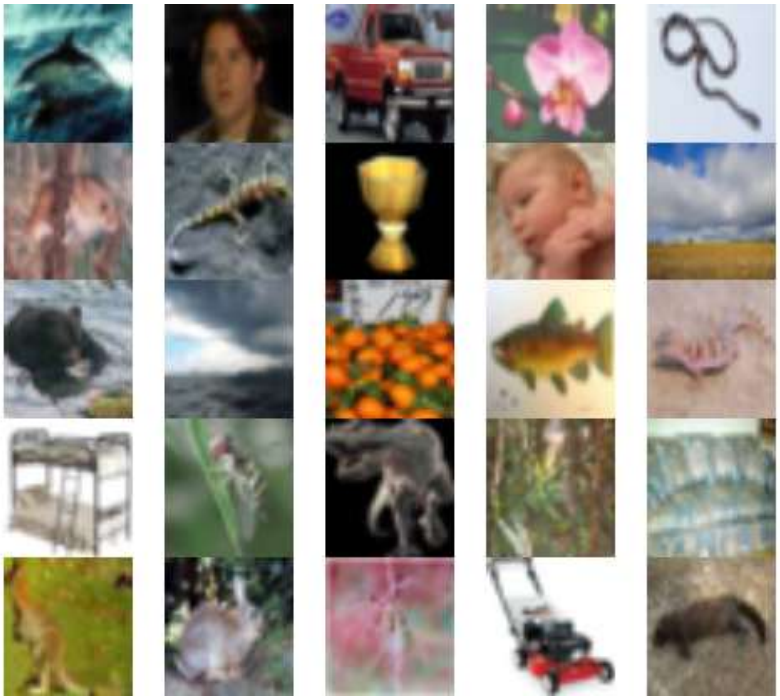
```
transform <- function(x) {
  transform_to_tensor(x)
}

train_ds <- cifar100_dataset(
  root = "./",
  train = TRUE,
  download = TRUE,
  transform = transform
)

test_ds <- cifar100_dataset(
  root = "./",
  train = FALSE,
  transform = transform
)

str(train_ds[1])
## List of 2
##  $ x:Float [1:3, 1:32, 1:32]
##  $ y: int 20
length(train_ds)
```

```
## [1] 50000
```

CIFAR データセットは 50,000 枚の訓練画像で構成されており、各画像は 3 次元テンソルで表されている。各カラー画像は、32 × 32の 8 ビットピクセルから成る 3 つのチャネルのセットとして表現されている。数字データの時と同様に標準化を行うが、配列構造は保持する。この処理は `transform` 引数を使って実現できる。

始める前に、いくつかの訓練画像を表示してみる。似たようなコードによって図 10.5（ページ 411）を生成した。

```
par(mar = c(0, 0, 0, 0), mfrow = c(5, 5))
index <- sample(seq(50000), 25)
for (i in index) plot(as.raster(as.array(train_ds[i][[1]]$permute(c(2,3,1)))))
```



`as.raster()`関数は、特徴マップをカラー画像としてプロットできるように変換する。なお、デモンストレーションのためにここでは中程度のサイズの CNN を指定するが、このモデルは図 10.8 に似た構造を持っている。

```
conv_block <- nn_module(
  initialize = function(in_channels, out_channels) {
    self$conv <- nn_conv2d(
      in_channels = in_channels,
      out_channels = out_channels,
      kernel_size = c(3,3),
      padding = "same"
    )
    self$relu <- nn_relu()
    self$pool <- nn_max_pool2d(kernel_size = c(2,2))
  },
  forward = function(x) {
    x %>%
      self$conv() %>%
      self$relu() %>%
      self$pool()
  }
)

model <- nn_module(
  initialize = function() {
    self$conv <- nn_sequential(
      conv_block(3, 32),
      conv_block(32, 64),
      conv_block(64, 128),
      conv_block(128, 256)
    )
    self$output <- nn_sequential(
      nn_dropout(0.5),
      nn_linear(2*2*256, 512),
      nn_relu(),
      nn_linear(512, 100)
    )
  },
```

```
  forward = function(x) {
    x %>%
      self$conv() %>%
      torch_flatten(start_dim = 2) %>%
      self$output()
  }
)
model()
## An `nn_module` containing 964,516 parameters.
## ── Modules ──────────────────────────────────────────────────────────────
## • conv: <nn_sequential> #388,416 parameters
## • output: <nn_sequential> #576,100 parameters
```

`layer_conv_2D()`において、`padding = "same"`引数を使用したことに注目する。これにより、出力チャネルは入力チャネルと同じ次元を持つことが保証される。入力層が 3 チャネルであるのに対して、最初の隠れ層には 32 チャネルある。すべての層で各チャネルに3×3の畳み込みフィルタを使用する。各畳み込みの後には2×2のブロックに対する最大プーリング層が続く。サマリーを調べると、各最大プーリング操作後にチャネルの次元が半分になっていることがわかる。これらの操作の最後には、256 チャネルで2×2の次元を持つ層が得られる。これらは平坦化され、1,024 のサイズを持つ密な層に変換される。つまり、各2×2の行列は4次元ベクトルに変換され、横に並べられて 1 層にまとめられる。その後、ドロップアウト正則化層が続き、512 のサイズを持つ別の密な層が続き、最終的に出力層に到達する。

最終的に、学習アルゴリズムを指定し、モデルを学習させる。

```
fitted <- model %>%
  setup(
    loss = nn_cross_entropy_loss(),
    optimizer = optim_rmsprop,
    metrics = list(luz_metric_accuracy())
  ) %>%
  set_opt_hparams(lr = 0.001) %>%
  fit(
    train_ds,
```

```
    epochs = 10, #30,
    valid_data = 0.2,
    dataloader_options = list(batch_size = 128)
  )

print(fitted)
## A `luz_module_fitted`
## ── Time ────────────────────────────────────────────────────────────────────
## • Total time: 6m 28.7s
## • Avg time per training epoch: 34.1s
## ── Results ─────────────────────────────────────────────────────────────────
## Metrics observed in the last epoch.
## ℹ Training:
## loss: 2.3863
## acc: 0.3737
## ── Model ───────────────────────────────────────────────────────────────────
## An `nn_module` containing 964,516 parameters.
## ── Modules ─────────────────────────────────────────────────────────────────
## • conv: <nn_sequential> #388,416 parameters
## • output: <nn_sequential> #576,100 parameters
evaluate(fitted, test_ds)
## A `luz_module_evaluation`
## ── Results ─────────────────────────────────────────────────────────────────
## loss: 2.4468
## acc: 0.3719
```

このモデルは実行に 10 分かかり、テストデータに対して 36%の精度を達成する。この結果は 100 クラスのデータにしては悪くない（ランダム分類器では 1%の精度しか得られない）が、ウェブを調べると約 75%の結果が見られる。通常、このような結果を得るには、アーキテクチャの細かい調整、正則化の微調整、そして時間が必要である。

事前学習済み CNN モデルの使用

次に、`imagenet` データベースに対して事前学習された CNN を使用して自然画像を分類する方法を示し、図 10.10 を作成した方法を示す。デジタル写真アルバムから 6 枚の jpeg 画像をコピーし、ディレクトリ `book_images` に保存した。（これらの画像は、<www.statlearning.com>のデータセクションから入手でき、**book_images.zip** をダウンロードしクリックすると、**book_images** ディレクトリが作成される。）まず、画像を読み込み、`Torch` ソフトウェアで扱われる `imagenet` の仕様に合わせた配列形式に変換する。作業ディレクトリが、画像が保存されているフォルダに設定されていることを確認する必要がある。

訳注 2：上記の通りだが、ここで利用するファイルは https://www.statlearning.com/resources-second-edition の Data Sets の中から book_images.zip より手に入れることができる。作業ディレクトリに画像が存在しないとエラーになり、例えば Error in `runtime_error()`:! unknown extension ” in path ’book_images/NA’などが表示される。またデータ・ファイルは R の current directory（ISLR 第 2 章の訳注を参照）に入れておく必要がある。

```
img_dir <- "book_images"
image_names <- list.files(img_dir)
num_images <- length(image_names)
x <- torch_empty(num_images, 3, 224, 224)
for (i in 1:num_images) {
   img_path <- file.path(img_dir, image_names[i])
   img <- img_path %>%
     base_loader() %>%
     transform_to_tensor() %>%
     transform_resize(c(224, 224)) %>%
     # imagenet の平均と標準偏差で正規化
     transform_normalize(
       mean = c(0.485, 0.456, 0.406),
       std = c(0.229, 0.224, 0.225)
     )
   x[i,,, ] <- img
}
```

次に、学習済みのネットワークを読み込む。このモデルは 18 層を持ち、かなりの複雑さを持っている。

```
model <- torchvision::model_resnet18(pretrained = TRUE)
model$eval() # モデルを評価モードにする
```

最後に、6 枚の画像を分類し、それぞれの予測確率に基づいてトップ 3 のクラス選択を返す。

```
preds <- model(x)

mapping <- jsonlite::read_json("https://s3.amazonaws.com/deep-learning-models/image-models/imagenet_class_index.json") %>%
  sapply(function(x) x[[2]])

top3 <- torch_topk(preds, dim = 2, k = 3)

top3_prob <- top3[[1]] %>%
  nnf_softmax(dim = 2) %>%
  torch_unbind() %>%
  lapply(as.numeric)

top3_class <- top3[[2]] %>%
  torch_unbind() %>%
  lapply(function(x) mapping[as.integer(x)])

result <- purrr::map2(top3_prob, top3_class, function(pr, cl) {
  names(pr) <- cl
  pr
})
names(result) <- image_names
print(result)
```

```
## $flamingo.jpg
##    flamingo   spoonbill white_stork
## 0.978211999 0.017045649 0.004742352
## $hawk.jpg
##         eel       agama common_newt
##   0.5391128   0.2527185   0.2081687
## $hawk_cropped.jpeg
##      kite       jay    magpie
## 0.6157812 0.2311861 0.1530326
## $huey.jpg
##           Lhasa Tibetan_terrier        Shih-Tzu
##      0.79760426      0.12013003      0.08226573
## $kitty.jpg
##        Saint_Bernard           guinea_pig Bernese_mountain_dog
##            0.3946672            0.3426990            0.2626339
## $weaver.jpg
## hummingbird    lorikeet   bee_eater
##   0.3633287   0.3577293   0.2789420
```

IMDb 文章分類

次に、`IMDB` データセット（10.4 節）に対して文章分類を行う。このデータセットは `torchdatasets` パッケージの一部として利用可能である。辞書サイズは最も頻繁に使用される 10,000 個の単語、トークンに制限する。

訳注 3：`tokenizers` パッケージをインストールする必要がある

```
set.seed(1)
max_features <- 10000
imdb_train <- imdb_dataset(
  root = ".",
  download = TRUE,
  split="train",
  num_words = max_features
```

```
)
imdb_test <- imdb_dataset(
  root = ".",
  download = TRUE,
  split="test",
  num_words = max_features
)
```

`imdb_train` の各要素は、1 から 10000 の間の数値（文章）で、辞書内の単語を指している。例えば、最初の訓練文章は 419 ページのポジティブなレビューである。最初の 12 語のインデックスは以下の通りである。

```
imdb_train[1]$x[1:12]
##  [1]    2   25  124   25   26   11 1113  149    6  211   54    4
```

単語を見るために、辞書に対する簡単なインターフェースとなる `decode_review()` という関数を作成する。

```
word_index <- imdb_train$vocabulary
decode_review <- function(text, word_index) {
   word <- names(word_index)
   idx <- unlist(word_index, use.names = FALSE)
   word <- c("<PAD>", "<START>", "<UNK>", word)
   words <- word[text]
   paste(words, collapse = " ")
}
decode_review(imdb_train[1]$x[1:12], word_index)
## [1] "<START> you know you are in trouble watching a comedy when the"
```

次に、文章のリストから各文章を「One-Hot」エンコードし、バイナリ行列を疎（スパース）行列形式で返す関数を作成する。

```
library(Matrix)
one_hot <- function(sequences, dimension) {
   seqlen <- sapply(sequences, length)
   n <- length(seqlen)
   rowind <- rep(1:n, seqlen)
   colind <- unlist(sequences)
   sparseMatrix(i = rowind, j = colind,
      dims = c(n, dimension))
}
```

疎（スパース）行列を構築するに際し、`1`は非ゼロの要素であることのみを表す。最後の行では、`sparseMatrix()`関数を呼び出し、各文章に対応する行インデックスと各文章内の単語に対応する列インデックスを与える。値は省略されるため、すべて 1 として扱われる。同じ文章に複数回出現する単語も、1 として記録される。

```
# すべての値をリストにまとめる
train <- seq_along(imdb_train) %>%
  lapply(function(i) imdb_train[i]) %>%
  purrr::transpose()
test <- seq_along(imdb_test) %>%
  lapply(function(i) imdb_test[i]) %>%
  purrr::transpose()

# num_words + padding + start + oov token = 10000 + 3
x_train_1h <- one_hot(train$x, 10000 + 3)
x_test_1h <- one_hot(test$x, 10000 + 3)
dim(x_train_1h)
## [1] 25000 10003
nnzero(x_train_1h) / (25000 * (10000 + 3))
## [1] 0.0131682
```

全要素の 1.3%のみが非ゼロであるため、これはかなりのメモリ節約に繋がる。次に、サイズが 2,000 の検証セットを作成し、23,000 を訓練用に残す。

```
set.seed(3)
ival <- sample(seq(along = train$y), 2000)
itrain <- seq_along(train$y)[-ival]
```

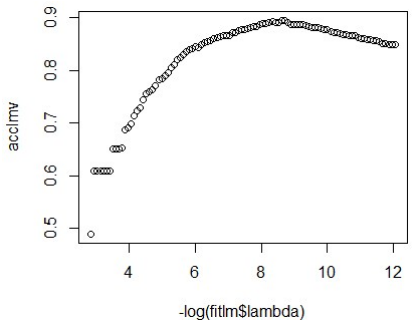
まず、`glmnet()`を使用して、訓練データに対してラッソ・ロジスティック回帰モデルを学習させ、検証データに対するパフォーマンスを評価する。最後に、縮小パラメータλの関数として精度`acclmv`をプロットする。同様にテストデータに対するパフォーマンスを計算し、これらは図 10.11 の左側のプロットを作成するために使用された。このコードは`x_train_1h`の疎（スパース）行列形式を活用し、約 5 秒で実行される。通常の密な形式では、約 5 分かかる。

```
library(glmnet)
y_train <- unlist(train$y)

fitlm <- glmnet(x_train_1h[itrain, ], unlist(y_train[itrain]),
    family = "binomial", standardize = FALSE)
classlmv <- predict(fitlm, x_train_1h[ival, ]) > 0
acclmv <- apply(classlmv, 2, accuracy,  unlist(y_train[ival]) > 0)
```

`accuracy()`関数は 10.9.2 節で作成したもので、予測行列`classlmv`の各列に適用される。この行列は論理行列であり、`TRUE/FALSE`の値を持つため、2 番目の引数`truth`も論理ベクトルとして与えられる。

プロットを作成する前に、プロットウィンドウを調整する。

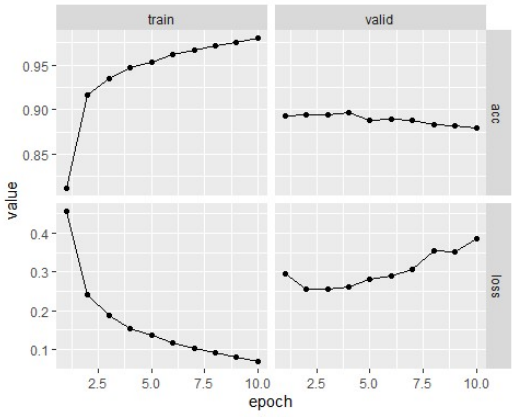
```
par(mar = c(4, 4, 4, 4), mfrow = c(1, 1))
plot(-log(fitlm$lambda), acclmv)
```



次に、2 つの隠れ層を持つ全結合ニューラルネットワークを学習させる。各隠れ層は 16 ユニットを持ち、ReLU 活性化関数を使用する。

```
model <- nn_module(
  initialize = function(input_size = 10000 + 3) {
    self$dense1 <- nn_linear(input_size, 16)
    self$relu <- nn_relu()
    self$dense2 <- nn_linear(16, 16)
    self$output <- nn_linear(16, 1)
  },
  forward = function(x) {
    x %>%
      self$dense1() %>%
      self$relu() %>%
      self$dense2() %>%
      self$relu() %>%
      self$output() %>%
      torch_flatten(start_dim = 1)
  }
)
model <- model %>%
  setup(
    loss = nn_bce_with_logits_loss(),
    optimizer = optim_rmsprop,
    metrics = list(luz_metric_binary_accuracy_with_logits())
  ) %>%
```

```
  set_opt_hparams(lr = 0.001)

fitted <- model %>%
  fit(
    # 訓練データと検証データを torch テンソルに変換
    list(
      torch_tensor(as.matrix(x_train_1h[itrain,]), dtype = torch_float()),
      torch_tensor(unlist(train$y[itrain]))
    ),
    valid_data = list(
      torch_tensor(as.matrix(x_train_1h[ival, ]), dtype = torch_float()),
      torch_tensor(unlist(train$y[ival]))
    ),
    dataloader_options = list(batch_size = 512),
    epochs = 10
  )

plot(fitted)
```



`fitted` オブジェクトには、各エポックでの訓練精度と検証精度を取得する `get_metrics` メソッドがある。図 10.11 には、各エポックでのテスト精度も含まれている。テスト精度を計算するためには、上記のコードブロックの最後を以下のように置き換え、再実行する。

```
fitted <- model %>%
  fit(
    list(
      torch_tensor(as.matrix(x_train_1h[itrain,]), dtype = torch_float()),
      torch_tensor(unlist(train$y[itrain]))
    ),
    valid_data = list(
      torch_tensor(as.matrix(x_test_1h), dtype = torch_float()),
      torch_tensor(unlist(test$y))
    ),
    dataloader_options = list(batch_size = 512),
    epochs = 10
  )
```

再帰型ニューラルネットワーク(Recurrent Neural Networks, RNN)

この実習では、10.5 節で扱われたモデルを学習させる。

文章分類のための逐次モデル

ここでは、10.5.1 節で説明したように、`IMDb` 映画レビューのデータを用いて感情分析のためのシンプルな LSTM RNN を学習させる。データの入力方法については 10.9.5 節で既に説明したので、ここでは繰り返さない。

まず、文章の長さを計算しよう。

```
wc <- sapply(seq_along(imdb_train), function(i) length(imdb_train[i]$x))
median(wc)
## [1] 178
sum(wc <= 500) / length(wc)
## [1] 0.916
```

91%以上の文章が 500 語未満であることがわかる。ここでの RNN は、すべての文章系列が同じ長さである必要がある。そのため、文章の長さを最後の$L = 500$語に制限し、短い文章の始めは空白でパディングする。このために `torchdatasets` の機能を使用する。

```
maxlen <- 500
num_words <- 10000
imdb_train <- imdb_dataset(root = ".", split = "train", num_words = num_words,
                           maxlen = maxlen)
imdb_test <- imdb_dataset(root = ".", split = "test", num_words = num_words,
                          maxlen = maxlen)

vocab <- c(rep(NA, imdb_train$index_from - 1), imdb_train$get_vocabulary())
tail(names(vocab)[imdb_train[1]$x])
## [1] "compensate" "you"        "the"        "rental"     ""
## [6] "d"
```

最後の表現は、最初の文章の最後のいくつかの単語を示している。この時点で、文章の各 500 語は、その単語が 10,000 語の辞書内でどの位置にあるかに対応する整数を使用して表現されている。RNN の最初の層は、32 のサイズを持つ埋め込み層であり、訓練中に学習される。この層は、各文章を$500 \times 10{,}000$の次元の行列として One-Hot エンコードし、これらの10,000次元を32次元に削減する。

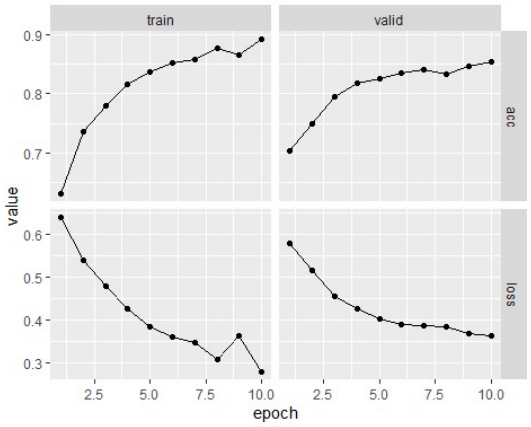
```
model <- nn_module(
  initialize = function() {
    self$embedding <- nn_embedding(10000 + 3, 32)
    self$lstm <- nn_lstm(input_size = 32, hidden_size = 32, batch_first = TRUE)
    self$dense <- nn_linear(32, 1)
  },
  forward = function(x) {
    c(output, c(hn, cn)) %<-% (x %>%
      self$embedding() %>%
      self$lstm())
    output[,-1,] %>%  # 最後の出力を取得
```

```
      self$dense() %>%
      torch_flatten(start_dim = 1)
  }
)
```

2 番目の層は 32 ユニットの LSTM で、出力層は 2 項分類タスクのためのシグモイドユニット 1 つである。

残りは、これまでに学習させた他のネットワークと似ている。ネットワークを学習する間、テストパフォーマンスを追跡し、87%の精度に達することがわかる。

```
model <- model %>%
  setup(
    loss = nn_bce_with_logits_loss(),
    optimizer = optim_rmsprop,
    metrics = list(luz_metric_binary_accuracy_with_logits())
  ) %>%
  set_opt_hparams(lr = 0.001)

fitted <- model %>% fit(
  imdb_train,
  epochs = 10,
  dataloader_options = list(batch_size = 128),
  valid_data = imdb_test
)
plot(fitted)
```




```
predy <- torch_sigmoid(predict(fitted, imdb_test)) > 0.5
evaluate(fitted, imdb_test, dataloader_options = list(batch_size = 512))
## A `luz_module_evaluation`
## ── Results ──────────────────────────────────────────────────────────────
## loss: 0.3621
## acc: 0.8532
```

時系列予測

次に、10.5.2 節で説明されたモデルを使用して時系列予測を行う方法を示す。まずデータを設定し、各変数を標準化する。

```
library(ISLR2)
xdata <- data.matrix(
 NYSE[, c("DJ_return", "log_volume","log_volatility")]
 )
istrain <- NYSE[, "train"]
xdata <- scale(xdata)
```

変数 `istrain` は、訓練セットに含まれる各年に対して `TRUE`、テストセットに含まれる各年に対して `FALSE` をとる。

次に、3 つの時系列のラグ(遅れ)変数を作成する関数を書く。最初に、データ行列とラグLを入力として受け取り、ラグ変数の行列を返す関数を作成する。この関数は、単純にL行の `NA` を上に挿入し、下部を切り捨てる。

```
lagm <- function(x, k = 1) {
   n <- nrow(x)
   pad <- matrix(NA, k, ncol(x))
   rbind(pad, x[1:(n - k), ])
}
```

この関数を使用して、必要なすべてのラグ変数と応答変数を含むデータフレームを作成する。

```
arframe <- data.frame(log_volume = xdata[, "log_volume"],
   L1 = lagm(xdata, 1), L2 = lagm(xdata, 2),
   L3 = lagm(xdata, 3), L4 = lagm(xdata, 4),
   L5 = lagm(xdata, 5)
 )
```

このフレームの最初の 5 行を確認すると、ラグ変数に欠損値があることがわかる（上記の構築による）。これらの行を削除し、`istrain` も適宜調整する。

```
arframe <- arframe[-(1:5), ]
istrain <- istrain[-(1:5)]
```

次に、`lm()`を使用して訓練データで線形 AR モデルを学習させ、テストデータで予測を行う。

```
arfit <- lm(log_volume ~ ., data = arframe[istrain, ])
arpred <- predict(arfit, arframe[!istrain, ])
V0 <- var(arframe[!istrain, "log_volume"])
1 - mean((arpred - arframe[!istrain, "log_volume"])^2) / V0
## [1] 0.413223
```

最後の 2 行は、テストデータに対するR^2を計算する（(3.17)に定義されている通りである）。このモデルに因子変数 `day_of_week` を含めて、再度学習させる。

```
arframed <-
    data.frame(day = NYSE[-(1:5), "day_of_week"], arframe)
arfitd <- lm(log_volume ~ ., data = arframed[istrain, ])
arpredd <- predict(arfitd, arframed[!istrain, ])
1 - mean((arpredd - arframe[!istrain, "log_volume"])^2) / V0
## [1] 0.4598616
```

RNN を学習させるために、これらのデータを再形成する必要がある。RNN では各観測について、$L = 5$の特徴ベクトルの系列$X = \{X_\ell\}_1^L$の形にする必要がある（式(10.20)、428 ページ参照）。これらは時系列のラグ変数で、1からLまで時間を遡ったものである。

```
n <- nrow(arframe)
xrnn <- data.matrix(arframe[, -1])
xrnn <- array(xrnn, c(n, 3, 5))
xrnn <- xrnn[,, 5:1]
xrnn <- aperm(xrnn, c(1, 3, 2))
dim(xrnn)
## [1] 6046    5    3
```

これを 4 つのステップで行った。最初のステップは、`arframe` から 3 つの予測変数のラグバージョンの$n \times 15$の行列を抽出する。2 番目のステップは、この行列を$n \times 3 \times 5$の配列に変換する。この変換は、単に次元属性を変更することで行う。新しい配列は列方向に埋められる。3 番目のステップは、ラグ変数の順序を逆にすることによって、インデックス1が最も遡った時間となり、インデックス5が最も近い時間となるようにする。最後のステップは、配列の座標を（部分転置のように）再配置して、`torch` の RNN モジュールで利用できる形式にする。

これで、12 ユニットの隠れ層を持つ RNN の準備が整った。

```
model <- nn_module(
  initialize = function() {
```

```
    self$rnn <- nn_rnn(3, 12, batch_first = TRUE)
    self$dense <- nn_linear(12, 1)
    self$dropout <- nn_dropout(0.2)
  },
  forward = function(x) {
    c(output, ...) %<-% (x %>%
      self$rnn())
    output[,-1,] %>%
      self$dropout() %>%
      self$dense() %>%
      torch_flatten(start_dim = 1)
  }
)

model <- model %>%
  setup(
    optimizer = optim_rmsprop,
    loss = nn_mse_loss()
  ) %>%
  set_opt_hparams(lr = 0.001)
```

出力層は、応答のためのユニット 1 つを持っている。

このモデルは、これまでのネットワークと同様の方法で学習させる。`fit` 関数にテストデータを検証データとして与え、その進行状況をモニタリングし、`history` 関数をプロットしてテストデータ上の進行状況を見ることができる。もちろん、これは早期打ち切りの基準として使用すべきではない。なぜなら、テスト性能にバイアスが現れるからである。

```
fitted <- model %>% fit(
    list(xrnn[istrain,, ], arframe[istrain, "log_volume"]),
    epochs = 30, # = 200,
    dataloader_options = list(batch_size = 64),
    valid_data =
      list(xrnn[!istrain,, ], arframe[!istrain, "log_volume"])
```

```
  )
kpred <- as.numeric(predict(fitted, xrnn[!istrain,, ]))
1 - mean((kpred - arframe[!istrain, "log_volume"])^2) / V0
## [1] 0.4028609
```

このモデルの訓練には約 1 分かかる。

次のように、上記の `nn_module()` コマンドを置き換えることもできる。

```
model <- nn_module(
  initialize = function() {
    self$dense <- nn_linear(15, 1)
  },
  forward = function(x) {
    x %>%
      torch_flatten(start_dim = 2) %>%
      self$dense()
  }
)
```

ここで、`torch_flatten()` は単に、入力系列を受け取り、それを予測変数の長いベクトルに変換する。この結果、線形 AR モデルが得られる。非線形 AR モデルを学習させるために、隠れ層を追加することができる。

しかし、既に `lm()` コマンドを使用して学習させた AR モデルのラグ変数の行列があるため、実際には平坦化を行わなくても非線形 AR モデルを学習させることができる。 `arframed` からモデル行列 `x` を抽出する。これには `day_of_week` 変数も含まれている。

```
x <- model.matrix(log_volume ~ . - 1, data = arframed)
colnames(x)
##  [1] "dayfri"           "daymon"           "daythur"
##  [4] "daytues"          "daywed"           "L1.DJ_return"
##  [7] "L1.log_volume"    "L1.log_volatility" "L2.DJ_return"
## [10] "L2.log_volume"    "L2.log_volatility" "L3.DJ_return"
```

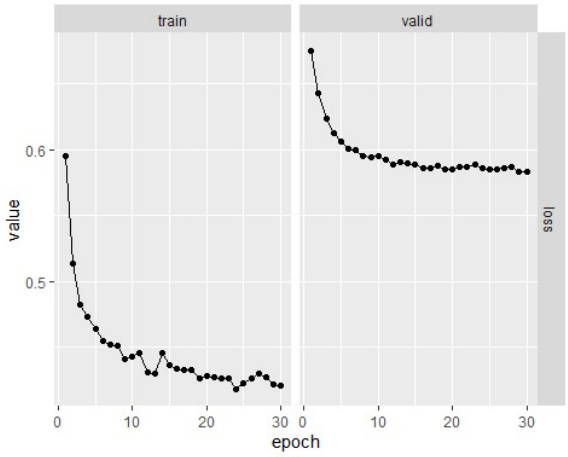
```
## [13] "L3.log_volume"     "L3.log_volatility" "L4.DJ_return"
## [16] "L4.log_volume"     "L4.log_volatility" "L5.DJ_return"
## [19] "L5.log_volume"     "L5.log_volatility"
```

式の中の`-1`は、切片のための 1 の列が作成されるのを避ける。`day_of_week`変数は 5 つのレベルを持つ因子（取引日は 5 日間）であり、`-1`により、ダミー変数は 4 つではなく 5 つになる。

非線形 AR モデルを学習させるための残りのステップは、今や馴染みがあるはずである。

```
arnnd <- nn_module(
  initialize = function() {
    self$dense <- nn_linear(20, 32)
    self$dropout <- nn_dropout(0.5)
    self$activation <- nn_relu()
    self$output <- nn_linear(32, 1)

  },
  forward = function(x) {
    x %>%
      torch_flatten(start_dim = 2) %>%
      self$dense() %>%
      self$activation() %>%
      self$dropout() %>%
      self$output() %>%
      torch_flatten(start_dim = 1)
  }
)
arnnd <- arnnd %>%
  setup(
    optimizer = optim_rmsprop,
    loss = nn_mse_loss()
  ) %>%
  set_opt_hparams(lr = 0.001)
```

```
fitted <- arnnd %>% fit(
    list(x[istrain,], arframe[istrain, "log_volume"]),
    epochs = 30,
    dataloader_options = list(batch_size = 64),
    valid_data =
      list(x[!istrain,], arframe[!istrain, "log_volume"])
  )
plot(fitted)
```




```
npred <- as.numeric(predict(fitted, x[!istrain, ]))
1 - mean((arframe[!istrain, "log_volume"] - npred)^2) / V0
## [1] 0.4669683
```

ISLR 第 11 章 実習：生存時間解析 (Survival Analysis)

このラボでは、3 つの異なるデータ・セットを使って生存時間解析の実習を行う。11.8.1 節では、11.3 節で初めて説明された `BrainCancer` データを分析する。11.8.2 節では、11.5.4 節の `Publication` データを検討する。最後に、11.8.3 節では、シミュレーションされたコールセンター・データセットを分析する。

脳腫瘍データ

最初に、`ISLR2` パッケージに含まれている `BrainCancer` データセットを利用しよう。

```
library(ISLR2)
```

```
## Warning: パッケージ 'ISLR2' はバージョン 4.4.3 の R の下で造られました
```

行は 88 人の患者を、列は 8 つの予測変数を表している。

```
names(BrainCancer)
```

```
## [1] "sex"       "diagnosis" "loc"       "ki"        "gtv"       "stereo"
## [7] "status"    "time"
```

まず、データを簡単に確認しておこう。

```
attach(BrainCancer)
table(sex)
```

```
## sex
## Female   Male
##     45     43
```

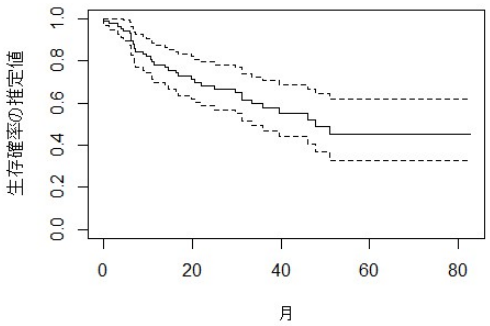
```
table(diagnosis)
```

```
## diagnosis
## Meningioma  LG glioma  HG glioma      Other
##         42          9         22         14
table(status)
## status
##  0  1
## 53 35
```

データ分析を始める前に、`status`変数がどのようにコード化されているかを確認することは重要である。ほとんどのソフトウェア（`R`を含む）では、`status = 1`がセンサリングされていない観測値、`status = 0`がセンサリングされた観測値を示すという慣習があるが、一部の科学者は逆のコード化を使用することもある。`BrainCancer`データセットでは、研究終了前に 35 人の患者が亡くなっている。

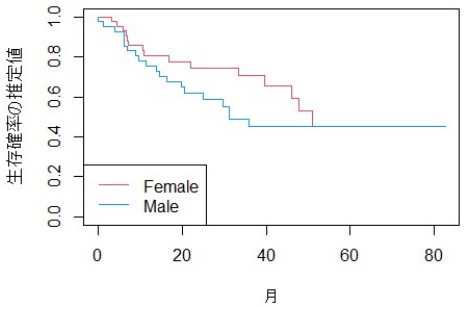
分析を始めるにあたり、`survival`ライブラリの`survfit()`関数を使って、図 11.2 に示されているカプラン・マイヤー生存曲線を再現しておこう。ここで、`time`はi番目のイベント（センサリングまたは死亡）の時間y_iに対応する。

```
library(survival)
fit.surv <- survfit(Surv(time, status) ~ 1)
plot(fit.surv, xlab = "月",
    ylab = "生存確率の推定値")
```



次に、性別(`sex`)で層別化したカプラン・マイヤー生存曲線を作成し、図 11.3 を再現してみる。

```
fit.sex <- survfit(Surv(time, status) ~ sex)
plot(fit.sex, xlab = "月",
    ylab = "生存確率の推定値", col = c(2,4))
legend("bottomleft", levels(sex), col = c(2,4), lty = 1)
```



11.4 節で説明したように、`survdiff()`関数を使用して、男性と女性の生存を比較するログ・ランク検定を実行する。

```
logrank.test <- survdiff(Surv(time, status) ~ sex)
logrank.test
## Call:
## survdiff(formula = Surv(time, status) ~ sex)
##
##             N Observed Expected (O-E)^2/E (O-E)^2/V
## sex=Female 45       15     18.5     0.676      1.44
## sex=Male   43       20     16.5     0.761      1.44
##
##  Chisq= 1.4  on 1 degrees of freedom, p= 0.2
```

結果のp値は 0.23 で、男女間の生存に差があるという証拠はない。

次に、`coxph()`関数を使ってコックス比例ハザードモデルをフィットしよう。最初に、`sex` のみを説明変数(予測変数)とするモデルを考える。

```
fit.cox <- coxph(Surv(time, status) ~ sex)
summary(fit.cox)
```

```
## Call:
## coxph(formula = Surv(time, status) ~ sex)
##   n= 88, number of events= 35
##           coef exp(coef) se(coef)     z Pr(>|z|)
## sexMale 0.4077    1.5033   0.3420 1.192    0.233
##         exp(coef) exp(-coef) lower .95 upper .95
## sexMale     1.503     0.6652     0.769     2.939
## Concordance= 0.565  (se = 0.045 )
## Likelihood ratio test= 1.44  on 1 df,   p=0.2
## Wald test            = 1.42  on 1 df,   p=0.2
## Score (logrank) test = 1.44  on 1 df,   p=0.2
```

尤度比検定、ウォルド検定、およびスコア検定の値は丸めている。ただし追加の桁を表示することも可能である。

```
summary(fit.cox)$logtest[1]
##     test
## 1.438822
summary(fit.cox)$waldtest[1]
## test
## 1.42
summary(fit.cox)$sctest[1]
##     test
## 1.440495
```

どの検定を使用しても、男女間の生存に有意な差がある証拠は見られない。

```
logrank.test$chisq
## [1] 1.440495
```

本章で学んだように、コックス・モデルのスコア検定はログ・ランク検定の統計量と完全に同等である！

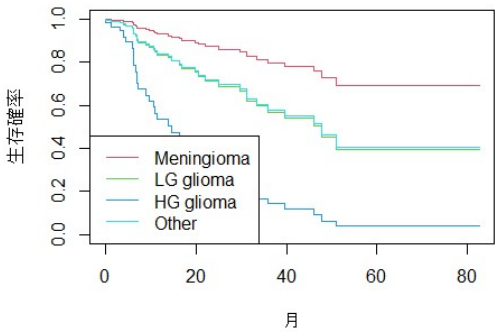
次に、追加の説明変数(予測変数)を利用したモデルをフィットしよう。

```
fit.all <- coxph(
Surv(time, status) ~ sex + diagnosis + loc + ki + gtv +
   stereo)
fit.all
## Call:
## coxph(formula = Surv(time, status) ~ sex + diagnosis + loc +
##     ki + gtv + stereo)
##                         coef exp(coef) se(coef)      z        p
## sexMale              0.18375   1.20171  0.36036  0.510  0.61012
## diagnosisLG glioma   0.91502   2.49683  0.63816  1.434  0.15161
## diagnosisHG glioma   2.15457   8.62414  0.45052  4.782 1.73e-06
## diagnosisOther       0.88570   2.42467  0.65787  1.346  0.17821
## locSupratentorial    0.44119   1.55456  0.70367  0.627  0.53066
## ki                  -0.05496   0.94653  0.01831 -3.001  0.00269
## gtv                  0.03429   1.03489  0.02233  1.536  0.12466
## stereoSRT            0.17778   1.19456  0.60158  0.296  0.76760
## Likelihood ratio test=41.37  on 8 df, p=1.776e-06
## n= 87, number of events= 35
##    (1 個の観測値が欠損のため削除されました)
```

`diagnosis`変数は、基準が髄膜腫（`meningioma`）に対応するようにコード化されている。結果は、高悪性度（HG）グリオーマに関連するリスクが髄膜腫のリスクの8倍以上（$e^{2.15} = 8.62$）であることを示している。言い換えれば、他の説明変数(予測変数)を調整した後でも、高悪性度グリオーマの患者は髄膜腫の患者に比べて生存率がはるかに悪いことになる。また、カルノフスキー指数（`ki`）の値が大きいほどリスクが低下、すなわち生存期間が長くなることも示されている。

最後に、他の予測変数を調整した上で、診断カテゴリごとの生存曲線をプロットする。このプロットを作成するために、他の予測変数の値を定量変数では平均、因子では最頻値に設定しておこう。まず、診断の各レベルに対応する4行のデータフレームを作成する。`survfit()`関数は、このデータフレームの各行に対応する曲線を生成し、`plot()`を1回呼び出すだけですべてを同じプロットに表示することができる。

```
modaldata <- data.frame(
     diagnosis = levels(diagnosis),
     sex = rep("Female", 4),
     loc = rep("Supratentorial", 4),
     ki = rep(mean(ki), 4),
     gtv = rep(mean(gtv), 4),
     stereo = rep("SRT", 4)
     )
survplots <- survfit(fit.all, newdata = modaldata)
plot(survplots, xlab = "月",
    ylab = "生存確率", col = 2:5)
legend("bottomleft", levels(diagnosis), col = 2:5, lty = 1)
```

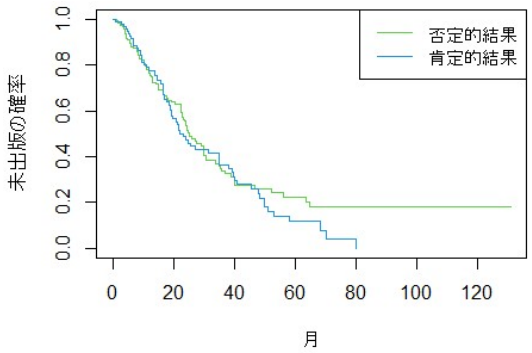


出版データ

11.5.4 節で紹介した `Publication` データは `ISLR2` ライブラリに含まれている。最初に、`posres` 変数（研究が肯定的結果か否定的結果かを記録）の層別化されたカプラン・マイヤー曲線をプロットして図 11.5 を再現しよう。

```
fit.posres <- survfit(
    Surv(time, status) ~ posres, data = Publication
  )
plot(fit.posres, xlab = "月",
```

```
    ylab = "未出版の確率", col = 3:4)
legend("topright", c("否定的結果", "肯定的結果"),
    col = 3:4, lty = 1)
```



前述のように、Cox 比例ハザード・モデルを `posres` 変数にフィットした結果、得られるp値は非常に大きく、肯定的結果と否定的結果を持つ研究間での出版までの時間に差があるという証拠はない。

```
fit.pub <- coxph(Surv(time, status) ~ posres,
    data = Publication)
fit.pub
## Call:
## coxph(formula = Surv(time, status) ~ posres, data = Publication)
##
##          coef exp(coef) se(coef)     z    p
## posres 0.1481    1.1596   0.1616 0.916 0.36
##
## Likelihood ratio test=0.83  on 1 df, p=0.3611
## n= 244, number of events= 156
```

予想通り、ログ・ランク検定も同じ結論を示している。

```
logrank.test <- survdiff(Surv(time, status) ~ posres,
    data = Publication)
logrank.test
## Call:
## survdiff(formula = Surv(time, status) ~ posres, data = Publication)
##             N Observed Expected (O-E)^2/E (O-E)^2/V
## posres=0 146       87     92.6     0.341     0.844
## posres=1  98       69     63.4     0.498     0.844
##  Chisq= 0.8  on 1 degrees of freedom, p= 0.4
```

しかし、モデルに他の予測変数を含めると、結果は劇的に変化する。ただし、ここでは資金調達メカニズム変数を除外している。

```
fit.pub2 <- coxph(Surv(time, status) ~ . - mech,
    data = Publication)
fit.pub2
## Call:
## coxph(formula = Surv(time, status) ~ . - mech, data = Publication)
##                 coef  exp(coef)   se(coef)      z       p
## posres     5.708e-01  1.770e+00  1.760e-01  3.244 0.00118
## multi     -4.086e-02  9.600e-01  2.512e-01 -0.163 0.87079
## clinend    5.462e-01  1.727e+00  2.620e-01  2.085 0.03710
## sampsize   4.678e-06  1.000e+00  1.472e-05  0.318 0.75070
## budget     4.385e-03  1.004e+00  2.465e-03  1.779 0.07518
## impact     5.832e-02  1.060e+00  6.676e-03  8.735 < 2e-16
## Likelihood ratio test=149.2  on 6 df, p=< 2.2e-16
## n= 244, number of events= 156
```

試験が臨床的エンドポイントに焦点を当てたかどうか、研究の影響、肯定的または否定的な結果を持つかどうかを含む、統計的に有意な変数がいくつかあることが分かる。

コールセンター・データ

この小節では、`coxed`ライブラリの`sim.survdata()`関数を使用して生存時間データをシミュレートする。このシミュレーションデータは、コールセンターに電話をかけた 2,000 人の顧客の待機時間（秒単位）を表すとする。この場合、顧客が電話を切るまでに応答されなかった場合にセンサリングが発生する。

3 つの共変量を考える：`Operators`（コールセンターの利用可能なオペレーター数、範囲は 5 から 15）、`Center`（A、B、または C）、および`Time`（朝、午後、または夜のいずれか）。これらの共変量のデータは、すべての可能性が均等に生じるように生成する。たとえば、朝、午後、夜の電話は同じ確率で発生し、オペレーター数は 5 から 15 の範囲で均等に分布すると仮定する。

```
set.seed(4)
N <- 2000
Operators <- sample(5:15, N, replace = T)
Center <- sample(c("A", "B", "C"), N, replace = T)
Time <- sample(c("Morn.", "After.", "Even."), N, replace = T)
X <- model.matrix( ~ Operators + Center + Time)[, -1]
```

計画行列`X`を見ると、変数がどのようにコード化されているかを確認することができる。

```
X[1:5, ]
##   Operators CenterB CenterC TimeEven. TimeMorn.
## 1        12       1       0         0         1
## 2        15       0       0         0         0
## 3         7       0       1         1         0
## 4         7       0       0         0         0
## 5        11       0       1         0         1
```

次に、係数とハザード関数を指定しよう。

```
true.beta <- c(0.04, -0.3, 0, 0.2, -0.2)
h.fn <- function(x) return(0.00001 * x)
```

ここでは、`Operators`に関連する係数を 0.04 に設定している。つまり、オペレーターが 1 人増えるごとに、コールが応答される「リスク」は$e^{0.04} = 1.041$倍になる（`Center`と`Time`共変量を考慮した場合）。これは理にかなっている：利用可能なオペレーター数が多いほど、待ち時間が短くなるはずである。`Center = B`に関連する係数は-0.3 で、`Center = A`を基準値として扱う。つまり、センターB でコールが応答されるリスクはセンターA のリスクの 0.74 倍であり、センターB での待ち時間が少し長いことを意味している。

次に、Cox 比例ハザードモデルの下でデータを生成しよう。`sim.survdata()`関数では、最大の故障時間（この場合は顧客の最長待機時間）を指定できる。ここでは 1,000 秒に設定する。

訳注：R の version が古いとパッケージ coxed を上手くインストールできない場合がありうるので注意を要する。

```
library(coxed)
queuing <- sim.survdata(N = N, T = 1000, X = X,
    beta = true.beta, hazard.fun = h.fn)
## Warning in FUN(X[[i]], ...): 9 additional observations right-censored becaus
e the user-supplied hazard function
##                                  is nonzero at the latest timepoint. To avo
id these extra censored observations, increase T
names(queuing)
## [1] "data"             "xdata"            "baseline"         "xb"
## [5] "exp.xb"           "betas"            "ind.survive"      "marg.effect"
## [9] "marg.effect.data"
```

「観測された」データは`queuing$data`に格納されており、`y`はイベント時間、`failed`はコールが応答されたかどうか（`failed = T`）または応答される前に顧客が電話を切ったかどうか（`failed = F`）を示す指標である。コールの約 90%が応答されたことが分かる。

```
head(queuing$data)
```

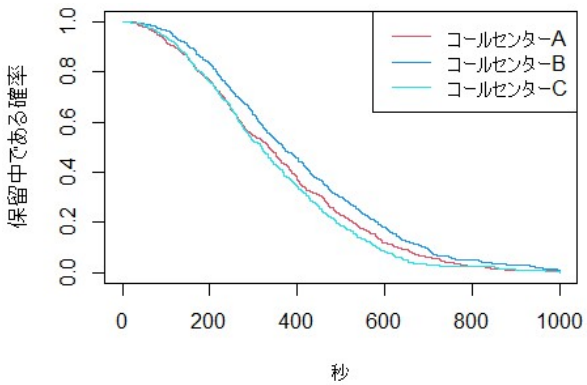
```
##   Operators CenterB CenterC TimeEven. TimeMorn.   y failed
## 1        12       1       0         0         1 344   TRUE
## 2        15       0       0         0         0 241   TRUE
## 3         7       0       1         1         0 187   TRUE
## 4         7       0       0         0         0 279   TRUE
## 5        11       0       1         0         1 954   TRUE
## 6         7       1       0         0         1 455   TRUE
```

```
mean(queuing$data$failed)
```

```
## [1] 0.89
```

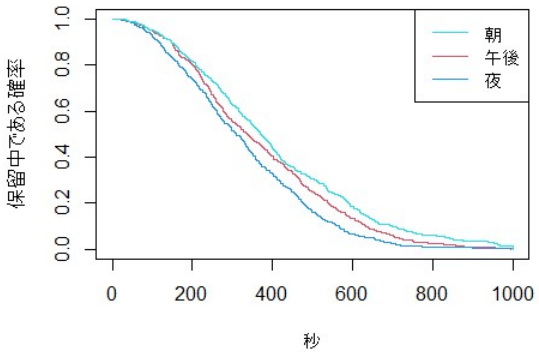
次に、カプラン・マイヤー生存曲線をプロットしよう。最初に変数 `Center` で層別化する。

```
fit.Center <- survfit(Surv(y, failed) ~ Center,
    data = queuing$data)
plot(fit.Center, xlab = "秒",
    ylab = "保留中である確率",
    col = c(2, 4, 5))
legend("topright",
     c("コールセンターA", "コールセンターB", "コールセンターC"),
     col = c(2, 4, 5), lty = 1)
```



次に変数 `Time` で層別化する。

```
fit.Time <- survfit(Surv(y, failed) ~ Time,
   data = queuing$data)
plot(fit.Time, xlab = "秒",
    ylab = "保留中である確率",
    col = c(2, 4, 5))
legend("topright", c("朝", "午後", "夜"),
    col = c(5, 2, 4), lty = 1)
```



センターB でのコールは、センターA および C でのコールよりも応答されるまでに時間がかかるようである。同様に、朝の待ち時間が最も長く、夜の待ち時間が最も短いようである。これらの差が統計的に有意かどうかを判断するためにログ・ランク検定を利用できる。

```
survdiff(Surv(y, failed) ~ Center, data = queuing$data)
## Call:
## survdiff(formula = Surv(y, failed) ~ Center, data = queuing$data)
##
##             N Observed Expected (O-E)^2/E (O-E)^2/V
## Center=A 683      603      579     0.971      1.45
## Center=B 667      600      701    14.641     24.64
## Center=C 650      577      499    12.062     17.05
##  Chisq= 28.3  on 2 degrees of freedom, p= 7e-07
survdiff(Surv(y, failed) ~ Time, data = queuing$data)
```

```
## Call:
## survdiff(formula = Surv(y, failed) ~ Time, data = queuing$data)
##                N Observed Expected (O-E)^2/E (O-E)^2/V
## Time=After. 688      616      619    0.0135     0.021
## Time=Even.  653      582      468   27.6353    38.353
## Time=Morn.  659      582      693   17.7381    29.893
##  Chisq= 46.8  on 2 degrees of freedom, p= 7e-11
```

センター間の差異および時刻間の差異は、いずれも非常に有意であることが分かる。

最後に、データに Cox 比例ハザード・モデルをフィットしてみる。

```
fit.queuing <- coxph(Surv(y, failed) ~ .,
    data = queuing$data)
fit.queuing
## Call:
## coxph(formula = Surv(y, failed) ~ ., data = queuing$data)
##                coef exp(coef) se(coef)      z        p
## Operators  0.04174   1.04263  0.00759  5.500  3.8e-08
## CenterB   -0.21879   0.80349  0.05793 -3.777 0.000159
## CenterC    0.07930   1.08253  0.05850  1.356 0.175256
## TimeEven.  0.20904   1.23249  0.05820  3.592 0.000328
## TimeMorn. -0.17352   0.84070  0.05811 -2.986 0.002828
## Likelihood ratio test=102.8  on 5 df, p=< 2.2e-16
## n= 2000, number of events= 1780
```

`Center = B`、`Time = Even.`、`Time = Morn.`のp値は非常に小さい。また、オペレーター数が増えるにつれて、コールが応答される瞬間的リスク（ハザード）が増加することは明らかである。データを自分で生成したため、`Operators`、`Center = B`、`Center = C`、`Time = Even.`、`Time = Morn.`の真の係数がそれぞれ 0.04、-0.3、0、0.2、および-0.2 であることが分かっている。したがって Cox モデルから得られる係数推定値はかなり正確なことが確認できる。

ISLR 第 12 章 実習：教師なし学習 (Unsupervised Learning)

主成分分析（PCA）

このラボでは、`R` の基本パッケージに含まれている `USArrests` データセットを使用して PCA を実行する。データセットの行は 50 州をアルファベット順に並べている。

```
states <- row.names(USArrests)
states
```

```
##  [1] "Alabama"        "Alaska"         "Arizona"        "Arkansas"
##  [5] "California"     "Colorado"       "Connecticut"    "Delaware"
##  [9] "Florida"        "Georgia"        "Hawaii"         "Idaho"
## [13] "Illinois"       "Indiana"        "Iowa"           "Kansas"
## [17] "Kentucky"       "Louisiana"      "Maine"          "Maryland"
## [21] "Massachusetts"  "Michigan"       "Minnesota"      "Mississippi"
## [25] "Missouri"       "Montana"        "Nebraska"       "Nevada"
## [29] "New Hampshire"  "New Jersey"     "New Mexico"     "New York"
## [33] "North Carolina" "North Dakota"   "Ohio"           "Oklahoma"
## [37] "Oregon"         "Pennsylvania"   "Rhode Island"   "South Carolina"
## [41] "South Dakota"   "Tennessee"      "Texas"          "Utah"
## [45] "Vermont"        "Virginia"       "Washington"     "West Virginia"
## [49] "Wisconsin"      "Wyoming"
```

データセットの列には 4 つの変数が含まれている。

```
names(USArrests)
```

```
## [1] "Murder"   "Assault"  "UrbanPop" "Rape"
```

まずデータを簡単に確認しよう。変数の平均値が大きく異なることに気が付くだろう。

```
apply(USArrests, 2, mean)
##   Murder  Assault UrbanPop     Rape
##    7.788  170.760   65.540   21.232
```

ここで`apply()`関数を使用して、データセットの各行または列に関数（この場合は`mean()`）を適用できる。2 番目の引数は、行の平均を計算する場合は 1、列の平均を計算する場合は 2 を指定すればよい。殺人(`Murder`)の約 3 倍のレイプ(`Rape`, 強姦)があり、レイプの 8 倍以上の暴行（`Assault`）があることが分かる。 また、`apply()`関数を使用して 4 つの変数の分散を確認することもできる。

```
apply(USArrests, 2, var)
##     Murder    Assault   UrbanPop       Rape
##   18.97047 6945.16571  209.51878   87.72916
```

予想通り、変数間で分散にも大きな違いがある。`UrbanPop`変数は各州で都市部に住む人口の割合を示しているが、これは各州で 10 万人あたりのレイプ数と比較するのは適当でない。変数を標準化せずに PCA を行うと、観測される主成分の多くが、平均値と分散が圧倒的に大きい`Assault`変数によって支配されることになる。そのため、PCA を実行する前に、変数を平均 0、標準偏差 1 に標準化することは重要となる。

次に、`prcomp()`関数を使用して主成分分析を実行しよう。この関数は`R`で PCA を行ういくつかの関数の 1 つである。

```
pr.out <- prcomp(USArrests, scale = TRUE)
```

デフォルトでは、`prcomp()`関数は変数を平均 0 に中心化する。`scale = TRUE`オプションを使用すると、変数が標準偏差 1 にスケーリングされる。`prcomp()`の出力には、いくつかの有益な情報が含まれている。

```
names(pr.out)
## [1] "sdev"     "rotation" "center"   "scale"    "x"
```

`center` と `scale` の要素は、PCA を実行する前にスケーリングに使用された変数の平均と標準偏差に対応する。

```
pr.out$center
##   Murder  Assault UrbanPop     Rape
##    7.788  170.760   65.540   21.232
pr.out$scale
##    Murder   Assault  UrbanPop      Rape
##  4.355510 83.337661 14.474763  9.366385
```

`rotation` 行列は主成分負荷量を与える。`pr.out$rotation` の各列は、対応する主成分負荷量ベクトルを含む。（この関数ではこれを回転行列と呼んでいる。$\bf X$行列を `pr.out$rotation` で行列積をとると、回転座標系におけるデータの座標を得られる。これが主成分スコアである。）

```
pr.out$rotation
##                  PC1        PC2        PC3         PC4
## Murder   -0.5358995 -0.4181809  0.3412327  0.64922780
## Assault  -0.5831836 -0.1879856  0.2681484 -0.74340748
## UrbanPop -0.2781909  0.8728062  0.3780158  0.13387773
## Rape     -0.5434321  0.1673186 -0.8177779  0.08902432
```

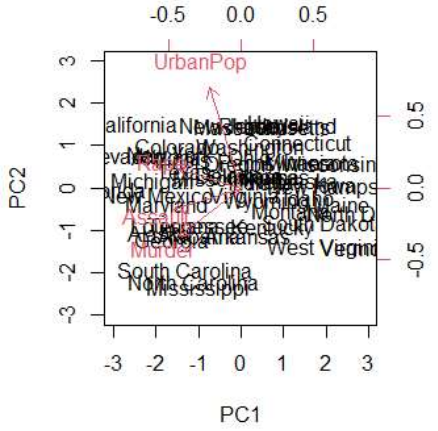
主成分が 4 つあることが分かった。これは、一般にn個の観測とp個の変数を持つデータセットでは $\min(n-1, p)$ 個の有益な主成分があることから予想できることである。

`prcomp()`関数を使用すると、主成分スコアベクトルを得るためにデータを主成分負荷量ベクトルで明示的に掛け合わせる必要はない。50×4 行列 `x` の列が主成分スコアベクトルになっている。すなわち、第k列が第k主成分スコアベクトルである。

```
dim(pr.out$x)
## [1] 50  4
```

次に、最初の 2 つの主成分をプロットしてみよう。

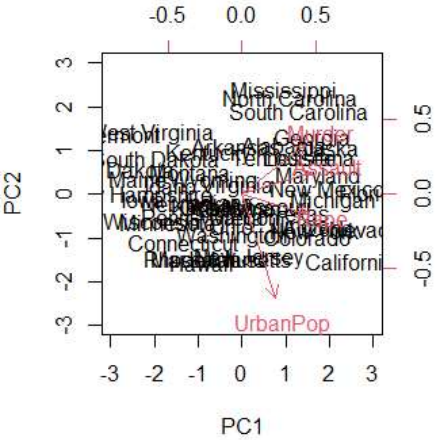
```
biplot(pr.out, scale = 0)
```



`biplot()`の `scale = 0` 引数により、矢印が負荷量を表すようにスケーリングされている。他の `scale` 値では、異なる解釈のバイプロットが得られる。

この図は図 12.1 の鏡像になっている。主成分は符号変更に対してのみ一意であることを思い出してみよう。いくつかの小さな変更を加えることで図 12.1 を再現することができる。

```
pr.out$rotation = -pr.out$rotation
pr.out$x = -pr.out$x
biplot(pr.out, scale = 0)
```



prcomp()関数は各主成分の標準偏差も出力してくれる。例えば、USArrestsデータセットでは、以下のようにすると標準偏差にアクセスできる。

```
pr.out$sdev
## [1] 1.5748783 0.9948694 0.5971291 0.4164494
```

各主成分で説明される分散はこれを二乗することで得られる。

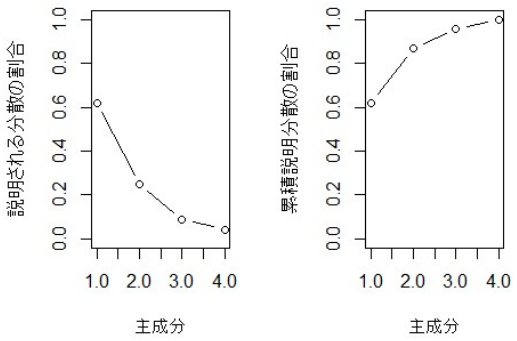
```
pr.var <- pr.out$sdev^2
pr.var
## [1] 2.4802416 0.9897652 0.3565632 0.1734301
```

各主成分で説明される分散の割合は、各主成分で説明される分散を 4 つの主成分すべてで説明される分散の合計で割ることで計算される。

```
pve <- pr.var / sum(pr.var)
pve
## [1] 0.62006039 0.24744129 0.08914080 0.04335752
```

第 1 主成分はデータの分散の 62.0,%を説明し、次の主成分は 24.7,%を説明している。以下も同様である。 各成分で説明される PVE（分散の説明割合）および累積 PVE は以下のようにするとプロットできる。

```
par(mfrow = c(1, 2))
plot(pve, xlab = "主成分",
    ylab = "説明される分散の割合", ylim = c(0, 1),
    type = "b")
plot(cumsum(pve), xlab = "主成分",
    ylab = "累積説明分散の割合",
    ylim = c(0, 1), type = "b")
```



この結果は図 12.3 に示されている。ここで関数 `cumsum()`は数値ベクトルの要素の累積和を計算してくれる。例えば：

```
a <- c(1, 2, 8, -3)
cumsum(a)
## [1]  1  3 11  8
```

行列補完(Matrix Completion)

ここでは、12.3 節で説明した `USArrests` データに対する分析を再現してみよう。データフレームの各列を平均 0、分散 1 にセンタリングおよびスケーリングした後、行列に変換する。

```
X <- data.matrix(scale(USArrests))
pcob <- prcomp(X)
summary(pcob)
## Importance of components:
##                           PC1    PC2     PC3     PC4
## Standard deviation     1.5749 0.9949 0.59713 0.41645
## Proportion of Variance 0.6201 0.2474 0.08914 0.04336
## Cumulative Proportion  0.6201 0.8675 0.95664 1.00000
```

最初の主成分が分散の 62%を説明していることが分かる。

12.2.2 節で見たように、中心化されたデータ行列$\boldsymbol{X}$における最適化問題(12.6)を解くことは、データの最初のM個の主成分を計算することと等価となる。 ここで特異値分解（SVD）は(12.6)を解くための一般的なアルゴリズムである。

```
sX <- svd(X)
names(sX)
## [1] "d" "u" "v"
round(sX$v, 3)
##        [,1]   [,2]   [,3]   [,4]
## [1,] -0.536 -0.418  0.341  0.649
## [2,] -0.583 -0.188  0.268 -0.743
## [3,] -0.278  0.873  0.378  0.134
## [4,] -0.543  0.167 -0.818  0.089
```

`svd()`関数は 3 つの要素、`u`、`d`、`v`を返す。行列`v`は主成分の負荷行列（符号反転を除いて）と等価である。

```
pcob$rotation
##                 PC1        PC2        PC3         PC4
## Murder   -0.5358995 -0.4181809  0.3412327  0.64922780
## Assault  -0.5831836 -0.1879856  0.2681484 -0.74340748
## UrbanPop -0.2781909  0.8728062  0.3780158  0.13387773
## Rape     -0.5434321  0.1673186 -0.8177779  0.08902432
```

行列`u`は*標準化*されたスコア行列と等価であり、標準偏差はベクトル`d`に格納されている。`svd()`の出力を利用するとスコアベクトルを再現することができる。 これらは`prcomp()`によって得られるスコアベクトルと同一である。

```
t(sX$d * t(sX$u))
##            [,1]       [,2]       [,3]        [,4]
##  [1,] -0.97566045 -1.12200121  0.43980366  0.154696581
訳者中略
## [49,]  2.05881199  0.60512507  0.13746933  0.182253407
## [50,]  0.62310061 -0.31778662  0.23824049 -0.164976866
pcob$x
```

```
##                       PC1         PC2         PC3          PC4
## Alabama        -0.97566045 -1.12200121  0.43980366  0.154696581
## Alaska         -1.93053788 -1.06242692 -2.01950027 -0.434175454
## Arizona        -1.74544285  0.73845954 -0.05423025 -0.826264240
## Arkansas        0.13999894 -1.10854226 -0.11342217 -0.180973554
## California     -2.49861285  1.52742672 -0.59254100 -0.338559240
## Colorado       -1.49934074  0.97762966 -1.08400162  0.001450164
## Connecticut     1.34499236  1.07798362  0.63679250 -0.117278736
## Delaware       -0.04722981  0.32208890  0.71141032 -0.873113315
## Florida        -2.98275967 -0.03883425  0.57103206 -0.095317042
## Georgia        -1.62280742 -1.26608838  0.33901818  1.065974459
## Hawaii          0.90348448  1.55467609 -0.05027151  0.893733198
## Idaho           1.62331903 -0.20885253 -0.25719021 -0.494087852
## Illinois       -1.36505197  0.67498834  0.67068647 -0.120794916
## Indiana         0.50038122  0.15003926 -0.22576277  0.420397595
## Iowa            2.23099579  0.10300828 -0.16291036  0.017379470
## Kansas          0.78887206  0.26744941 -0.02529648  0.204421034
## Kentucky        0.74331256 -0.94880748  0.02808429  0.663817237
## Louisiana      -1.54909076 -0.86230011  0.77560598  0.450157791
## Maine           2.37274014 -0.37260865  0.06502225 -0.327138529
## Maryland       -1.74564663 -0.42335704  0.15566968 -0.553450589
## Massachusetts   0.48128007  1.45967706  0.60337172 -0.177793902
## Michigan       -2.08725025  0.15383500 -0.38100046  0.101343128
## Minnesota       1.67566951  0.62590670 -0.15153200  0.066640316
## Mississippi    -0.98647919 -2.36973712  0.73336290  0.213342049
## Missouri       -0.68978426  0.26070794 -0.37365033  0.223554811
## Montana         1.17353751 -0.53147851 -0.24440796  0.122498555
## Nebraska        1.25291625  0.19200440 -0.17380930  0.015733156
## Nevada         -2.84550542  0.76780502 -1.15168793  0.311354436
## New Hampshire   2.35995585  0.01790055 -0.03648498 -0.032804291
## New Jersey     -0.17974128  1.43493745  0.75677041  0.240936580
## New Mexico     -1.96012351 -0.14141308 -0.18184598 -0.336121113
## New York       -1.66566662  0.81491072  0.63661186 -0.013348844
## North Carolina -1.11208808 -2.20561081  0.85489245 -0.944789648
## North Dakota    2.96215223 -0.59309738 -0.29824930 -0.251434626
## Ohio            0.22369436  0.73477837  0.03082616  0.469152817
## Oklahoma        0.30864928  0.28496113  0.01515592  0.010228476
## Oregon         -0.05852787  0.53596999 -0.93038718 -0.235390872
## Pennsylvania    0.87948680  0.56536050  0.39660218  0.355452378
## Rhode Island    0.85509072  1.47698328  1.35617705 -0.607402746
## South Carolina -1.30744986 -1.91397297  0.29751723 -0.130145378
## South Dakota    1.96779669 -0.81506822 -0.38538073 -0.108470512
## Tennessee      -0.98969377 -0.85160534 -0.18619262  0.646302674
```

```
## Texas          -1.34151838  0.40833518  0.48712332  0.636731051
## Utah            0.54503180  1.45671524 -0.29077592 -0.081486749
## Vermont         2.77325613 -1.38819435 -0.83280797 -0.143433697
## Virginia        0.09536670 -0.19772785 -0.01159482  0.209246429
## Washington      0.21472339  0.96037394 -0.61859067 -0.218628161
## West Virginia   2.08739306 -1.41052627 -0.10372163  0.130583080
## Wisconsin       2.05881199  0.60512507  0.13746933  0.182253407
## Wyoming         0.62310061 -0.31778662  0.23824049 -0.164976866
```

このラボの内容は`prcomp()`関数を使用しても可能であるが、ここでは`svd()`関数を使用した利用方法を示しておいた。

次に、50×4 のデータ行列からランダムに 20 個のエントリを除外しておこう。この操作を行うには、最初にランダムに 20 行（州）を選択し、次に各行の 4 つのエントリのうち 1 つをランダムに選択すればよい。この操作により、各行には少なくとも 3 つの観測値が含まれることが保証される。

```
nomit <- 20
set.seed(15)
ina <- sample(seq(50), nomit)
inb <- sample(1:4, nomit, replace = TRUE)
Xna <- X
index.na <- cbind(ina, inb)
Xna[index.na] <- NA
```

ここで、変数`ina`は 1 から 50 の間の整数 20 個を含む。これは欠損値が含まれる州を表している。また、変数`inb`は 1 から 4 の間の整数 20 個を含み、それぞれの州で欠損値が含まれる特徴を表す。インデックスを作成するために、`index.na`という 2 列の行列を作成しておく。この列は`ina`と`inb`です。行列のインデックスを使用して、欠損値を指定することができる。

次に、アルゴリズム 12.1 を実装するコードを記述しておこう。まず、行列を入力として受け取り、`svd()`関数を使用してその近似を返す関数を記述する。これはアルゴリズム 12.1 のステップ 2 として必要である。前述のように、`prcomp()`関数を使用することも可能であるが、ここでは例示のために`svd()`関数を利用する。

```
fit.svd <- function(X, M = 1) {
   svdob <- svd(X)
   with(svdob,
        u[, 1:M, drop = FALSE] %*%
        (d[1:M] * t(v[, 1:M, drop = FALSE]))
     )
}
```

ここでは、`return()`関数を明示的に呼び出して値を返すことはしていないが、計算結果は`R`によって自動的に返される。`with()`関数を使用して、`svdob`の要素を簡単にインデックスできるようにしている。`with()`を使用しない場合、`fit.svd()`関数内に記述することも可能だろう。

アルゴリズムのステップ 1 を実行するには、`Xhat`（アルゴリズム 12.1 の $\tilde{\boldsymbol{X}}$を初期化する必要がある。これは、欠損値を非欠損エントリの列平均で置き換えることで行えばよい。

```
Xhat <- Xna
xbar <- colMeans(Xna, na.rm = TRUE)
Xhat[index.na] <- xbar[inb]
```

ステップ 2 を開始する前に、反復の進捗状況を測定する準備を行う。

```
thresh <- 1e-7
rel_err <- 1
iter <- 0
ismiss <- is.na(Xna)
mssold <- mean((scale(Xna, xbar, FALSE)[!ismiss])^2)
mss0 <- mean(Xna[!ismiss]^2)
```

ここで、`ismiss`は`Xna`と同じ次元の新しい論理行列である。対応する行列要素が欠損している場合、その要素は`TRUE`となる。これにより、欠損および非欠損エントリの両方にアクセスすることができる。 非欠損要素の二乗平均を`mss0`に格納しよう。 `Xhat`の古いバージョンの非欠損要素の平均二乗誤差を`mssold`に格納する。

現在の `Xhat` バージョンの非欠損要素の平均二乗誤差を `mss` に格納し、アルゴリズム 12.1 のステップ 2 を、相対誤差`(mssold - mss) / mss0` が `thresh = 1e-7` を下回るまで反復する。

ステップ 2(a)では、`fit.svd()`を使用して `Xhat` を近似し、これを `Xapp` と呼ぼう。ステップ 2(b)では、`Xapp` を使用して、`Xna` で欠損している `Xhat` の要素を更新する。最後にステップ 2(c)で相対誤差を計算する。これら 3 つのステップは以下の `while()`ループに含まれている。

```
while(rel_err > thresh) {
    iter <- iter + 1
    # Step 2(a)
    Xapp <- fit.svd(Xhat, M = 1)
    # Step 2(b)
    Xhat[ismiss] <- Xapp[ismiss]
    # Step 2(c)
    mss <- mean(((Xna - Xapp)[!ismiss])^2)
    rel_err <- (mssold - mss) / mss0
    mssold <- mss
    cat("Iter:", iter, "MSS:", mss,
      "Rel. Err:", rel_err, "\n")
    }
## Iter: 1 MSS: 0.3821695 Rel. Err: 0.6194004
## Iter: 2 MSS: 0.3705046 Rel. Err: 0.01161265
## Iter: 3 MSS: 0.3692779 Rel. Err: 0.001221144
## Iter: 4 MSS: 0.3691229 Rel. Err: 0.0001543015
## Iter: 5 MSS: 0.3691008 Rel. Err: 2.199233e-05
## Iter: 6 MSS: 0.3690974 Rel. Err: 3.376005e-06
## Iter: 7 MSS: 0.3690969 Rel. Err: 5.465067e-07
## Iter: 8 MSS: 0.3690968 Rel. Err: 9.253082e-08
```

8 回の反復の後、相対誤差が `thresh = 1e-7` を下回り、アルゴリズムが終了した。この時、非欠損要素の平均二乗誤差は 0.369 である。

最後に、補完された 20 個の値と実際の値の相関を計算しておこう。

```
cor(Xapp[ismiss], X[ismiss])
## [1] 0.6535043
```

このラボでは、教育的な目的でアルゴリズム 12.1 を自分で実装した。しかし、自分のデータに行列補完を適用したい読者は、CRAN にある `softImpute` パッケージを使用することを勧めておこう。このパッケージは、このアルゴリズムの一般化を非常に効率的に実装している。

クラスタリング(Cluslering)

K-平均クラスタリング

`kmeans()`関数は `R` でK-平均法クラスタリングを実行する。ここでは、最初にデータに実際に 2 つのクラスタが存在するシンプルなシミュレーション例を利用してみる。最初の 25 個の観測値は、次の 25 個の観測値に対して平均値をシフトさせた。

```
set.seed(2)
x <- matrix(rnorm(50 * 2), ncol = 2)
x[1:25, 1] <- x[1:25, 1] + 3
x[1:25, 2] <- x[1:25, 2] - 4
```

次に、$K = 2$でK-平均法クラスタリングを実行しよう。

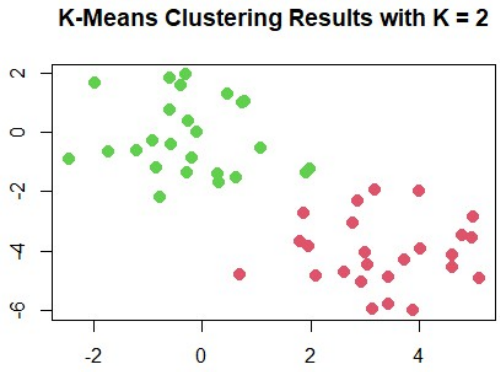
```
km.out <- kmeans(x, 2, nstart = 20)
```

50 個の観測値のクラスタ割り当ては `km.out$cluster` に格納されている。

```
km.out$cluster
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2
## [39] 2 2 2 2 2 2 2 2 2 2 2 2
```

K-平均法クラスタリングは、グループ情報を`kmeans()`に与えていなくても、観測値を完全に 2 つのクラスタに分けることができた。このデータをプロットし、各観測値をそのクラスタ割り当てに応じた色で表示してみる。

```
#par(mfrow = c(1, 2))
plot(x, col = (km.out$cluster + 1),
    main = "K-Means Clustering Results with K = 2",
    xlab = "", ylab = "", pch = 20, cex = 2)
```



観測値が 2 次元であるため、プロットが容易である。もし変数が 2 つ以上あれば、PCA を実行して最初の 2 つの主成分スコアベクトルをプロットすることもできる。

この例では、データを生成したために本当に 2 つのクラスタが存在することを知っていた。しかし、実際のデータでは、通常、真のクラスタ数を知ることはできない。なお、この例で$K = 3$でK-平均法クラスタリングを実行することも可能である。

```
set.seed(4)
km.out <- kmeans(x, 3, nstart = 20)
km.out
## K-means clustering with 3 clusters of sizes 17, 23, 10
## Cluster means:
##         [,1]        [,2]
## 1  3.7789567 -4.56200798
## 2 -0.3820397 -0.08740753
## 3  2.3001545 -2.69622023
```

```
## Clustering vector:
##  [1] 1 3 1 3 1 1 1 3 1 3 1 3 1 3 1 3 1 1 1 1 1 3 1 1 1 2 2 2 2 2 2 2 2 2 2
 2 2
## [39] 2 2 2 2 2 3 2 3 2 2 2 2
## Within cluster sum of squares by cluster:
## [1] 25.74089 52.67700 19.56137
##  (between_SS / total_SS =  79.3 %)
## Available components:
## [1] "cluster"      "centers"      "totss"        "withinss"     "tot.withins
s"
## [6] "betweenss"    "size"         "iter"         "ifault"
plot(x, col = (km.out$cluster + 1),
    main = "K-Means Clustering Results with K = 3",
    xlab = "", ylab = "", pch = 20, cex = 2)
```

$K = 3$では、K-平均法クラスタリングにより 2 つのクラスタを分割してしまった。

`R` で `kmeans()`関数を実行する際、初期クラスタ割り当てを複数回行うには、`nstart` 引数を使用する。`nstart` の値が 1 より大きい場合、アルゴリズム 12.2 のステップ 1 で複数のランダム割り当てを使用してK-平均法クラスタリングが実行され、`kmeans()`関数は最良の結果のみを記録する。ここでは、`nstart = 1` と `nstart = 20` を比較してみる。

```
set.seed(4)
km.out <- kmeans(x, 3, nstart = 1)
km.out$tot.withinss
```

```
## [1] 104.3319
km.out <- kmeans(x, 3, nstart = 20)
km.out$tot.withinss
## [1] 97.97927
```

`km.out$tot.withinss`はクラスタ内平方和の総和であり、K-平均法クラスタリング（式 12.17）で最小化を目指すものである。個々のクラスタ内平方和はベクトル`km.out$withinss`に格納されている。

常にK-平均法クラスタリングを`nstart`の値を 20 や 50 など大きな値に設定して実行することを強く推奨しておく。そうしないと、望ましくない局所的最適解が得られる可能性があるのである。

また、K-平均法クラスタリングを実行する際は、`set.seed()`関数を使用して乱数の種を設定することも重要である。この操作により、ステップ 1 での初期クラスタ割り当てが再現可能となり、K-平均法の結果が完全に再現可能になる。

階層型クラスタリング

`hclust()`関数は`R`で階層型クラスタリングを実装する。以下の例では、前のラボで使用したデータを使用して、ユークリッド距離を不類似度測定として用いた完全連結法、単一連結法、および平均連結法による階層型クラスタリングの樹形図をプロットしよう。

まず完全連結法を使用して観測値をクラスタリングする。`dist()`関数を使用して 50×50 の観測間ユークリッド距離行列を計算する。

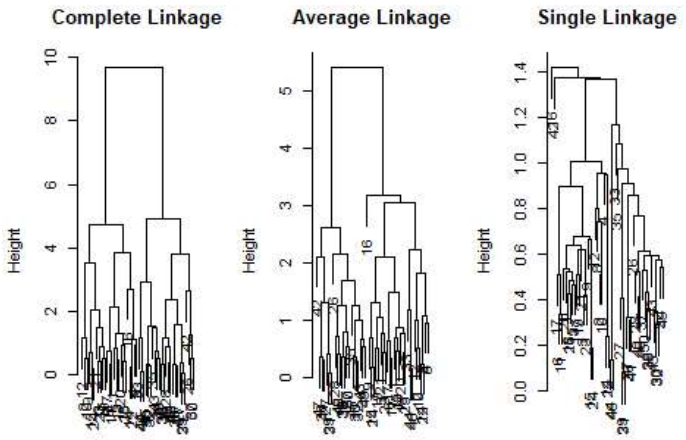
```
hc.complete <- hclust(dist(x), method = "complete")
```

同様にして、平均連結法や単一連結法を用いて階層型クラスタリングを実行することができる。

```
hc.average <- hclust(dist(x), method = "average")
hc.single <- hclust(dist(x), method = "single")
```

通常の`plot()`関数を使用して、得られた樹形図をプロットする。プロットの下部にある番号は各観測値を識別してくれる。

```
par(mfrow = c(1, 3))
plot(hc.complete, main = "Complete Linkage",
    xlab = "", sub = "", cex = .9)
plot(hc.average, main = "Average Linkage",
    xlab = "", sub = "", cex = .9)
plot(hc.single, main = "Single Linkage",
    xlab = "", sub = "", cex = .9)
```



樹形図の特定の切断に関連する各観測値のクラスタラベルを決定するには、`cutree()`関数を使用する。

```
cutree(hc.complete, 2)
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2
 2 2
## [39] 2 2 2 2 2 2 2 2 2 2 2 2
cutree(hc.average, 2)
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 1 2 2 2
 2 2
## [39] 2 2 2 2 2 1 2 1 2 2 2 2
cutree(hc.single, 2)
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
 1 1
## [39] 1 1 1 1 1 1 1 1 1 1 1 1
```

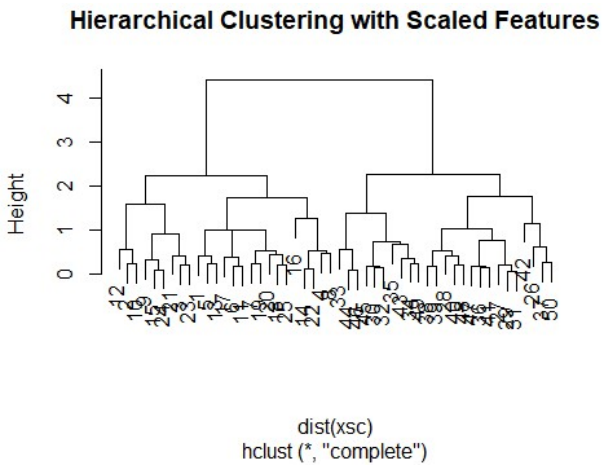
`cutree()`の 2 番目の引数は、取得したいクラスタ数を指定する。このデータでは、完全連結法と平均連結法は一般的に観測値を正しいグループに分ける。しかし、単一連結法では 1 つの観測値が独自のクラスタに属していると識別されている。より合理的な結果としては 4 つのクラスタを選択するとことが得られるが、それでも 2 つのシングルトン（独立した要素）が存在している。

```
cutree(hc.single, 4)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 1 1 1 1 1 1 1 1 1 3 3 3 3 3 3 3 3 3 3
 3 3
## [39] 3 3 3 4 3 3 3 3 3 3 3 3
```

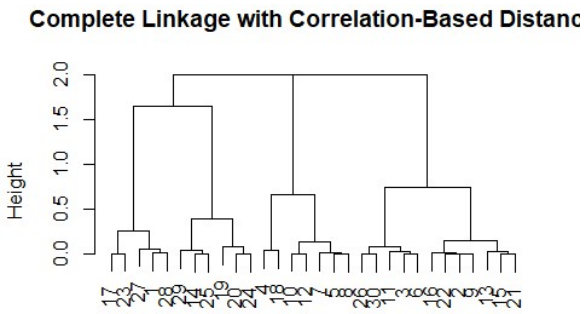
観測値の階層型クラスタリングを実行する前に変数をスケーリングするには、`scale()`関数を使用する。

```
xsc <- scale(x)
plot(hclust(dist(xsc), method = "complete"),
    main = "Hierarchical Clustering with Scaled Features")
```



相関に基づく距離は`as.dist()`関数を使用して計算できる。この関数は任意の対称行列を`hclust()`が距離行列として認識する形式に変換する。ただし、少なくとも 3 つの特徴を持つデータに対してのみ意味がある。2 つの特徴のみに基づく観測間の絶対相関は常に 1 だからである。そのため、3 次元のデータセットをクラスタリングしますが、このデータセットには真のクラスタは含まれていない。

```r
x <- matrix(rnorm(30 * 3), ncol = 3)
dd <- as.dist(1 - cor(t(x)))
plot(hclust(dd, method = "complete"),
    main = "Complete Linkage with Correlation-Based Distance",
    xlab = "", sub = "")
```



NCI60 データの例

教師なし学習の手法は、ゲノムデータの解析でよく使用されている。特に、PCA と階層型クラスタリングは人気のあるツールとなっている。 ここでは、`NCI` がん細胞株マイクロアレイデータを用いてこれらの手法を示しておこう。このデータは、64 のがん細胞株における 6,830 個の遺伝子発現測定値で構成されている。

```r
library(ISLR2)
nci.labs <- NCI60$labs
nci.data <- NCI60$data
```

各細胞株は、`nci.labs` で与えられたがんタイプでラベル付けされている。これらは教師なし手法である PCA やクラスタリングの実行には利用しないが、実行後にこれらのがんタイプが教師なし手法の結果とどの程度一致しているかを確認できる。

このデータは 64 行と 6,830 列からなる。

```r
dim(nci.data)
```

```
## [1]   64 6830
```

まず、細胞株のがんタイプを調べよう。

```
nci.labs[1:4]
## [1] "CNS"   "CNS"   "CNS"   "RENAL"
table(nci.labs)
## nci.labs
##      BREAST         CNS       COLON K562A-repro K562B-repro    LEUKEMIA
##           7           5           7           1           1           6
## MCF7A-repro MCF7D-repro    MELANOMA       NSCLC     OVARIAN    PROSTATE
##           1           1           8           9           6           2
##       RENAL     UNKNOWN
##           9           1
```

NCI60 データにおける PCA

まず、変数（遺伝子）を標準偏差 1 にスケーリングした後にデータに対して PCA を実行する。ただし、遺伝子をスケーリングしない方が良いと主張することも合理的と云えるだろう。

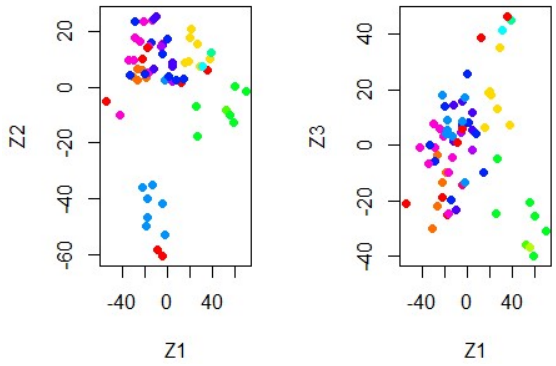
```
pr.out <- prcomp(nci.data, scale = TRUE)
```

次に、データを可視化するために最初のいくつかの主成分スコア・ベクトルをプロットする。特定のがんタイプに対応する観測値（細胞株）は同じ色でプロットしよう。このプロットにより、同じがんタイプ内の観測値がどの程度似ているかが分かる。まず、数値ベクトルの各要素に一意の色を割り当てる単純な関数を作成する。この関数を利用して、64 個の細胞株のそれぞれにがんタイプに基づいて色を割り当てる。

```
Cols <- function(vec) {
   cols <- rainbow(length(unique(vec)))
   return(cols[as.numeric(as.factor(vec))])
 }
```

rainbow()関数は引数として正の整数を取り、その数の異なる色を含むベクトルを返す。この関数で主成分スコアベクトルをプロットする。

```
par(mfrow = c(1, 2))
plot(pr.out$x[, 1:2], col = Cols(nci.labs), pch = 19,
    xlab = "Z1", ylab = "Z2")
plot(pr.out$x[, c(1, 3)], col = Cols(nci.labs), pch = 19,
    xlab = "Z1", ylab = "Z3")
```



結果のプロットは図 12.17 に示している。同じがんタイプに対応する細胞株は、最初のいくつかの主成分スコア・ベクトルで似た値を持つ傾向があることが分かる。これは、同じがんタイプの細胞株が非常に似た遺伝子発現レベルを持つ傾向があることを示している。

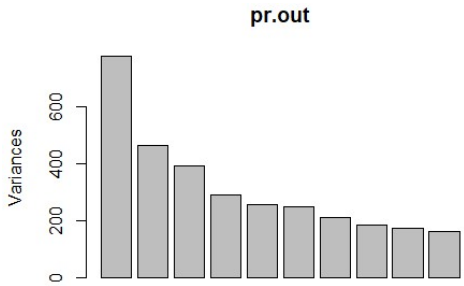
主成分の最初のいくつかが説明する分散の割合（PVE）の概要は、prcomp オブジェクトに対する summary()メソッドを使用して取得できる（出力は省略する）。

```
summary(pr.out)
## Importance of components:
##                            PC1      PC2      PC3      PC4      PC5      PC6
## Standard deviation     27.8535 21.48136 19.82046 17.03256 15.97181 15.72108
## Proportion of Variance  0.1136  0.06756  0.05752  0.04248  0.03735  0.03619
## Cumulative Proportion   0.1136  0.18115  0.23867  0.28115  0.31850  0.35468
##                            PC7      PC8      PC9     PC10     PC11     PC12
## Standard deviation     14.47145 13.54427 13.14400 12.73860 12.68672 12.15769
## Proportion of Variance  0.03066  0.02686  0.02529  0.02376  0.02357  0.02164
## Cumulative Proportion   0.38534  0.41220  0.43750  0.46126  0.48482  0.50646
##                           PC13     PC14     PC15     PC16     PC17     PC18
```

```
## Standard deviation     11.83019 11.62554 11.43779 11.00051 10.65666 10.48880
## Proportion of Variance  0.02049  0.01979  0.01915  0.01772  0.01663  0.01611
## Cumulative Proportion   0.52695  0.54674  0.56590  0.58361  0.60024  0.61635
##                            PC19    PC20     PC21    PC22    PC23    PC24
## Standard deviation     10.43518 10.3219 10.14608 10.0544 9.90265 9.64766
## Proportion of Variance  0.01594  0.0156  0.01507  0.0148 0.01436 0.01363
## Cumulative Proportion   0.63229  0.6479  0.66296  0.6778 0.69212 0.70575
##                            PC25    PC26    PC27   PC28    PC29    PC30    PC31
## Standard deviation     9.50764 9.33253 9.27320 9.0900 8.98117 8.75003 8.59962
## Proportion of Variance 0.01324 0.01275 0.01259 0.0121 0.01181 0.01121 0.01083
## Cumulative Proportion  0.71899 0.73174 0.74433 0.7564 0.76824 0.77945 0.79027
##                            PC32    PC33    PC34    PC35    PC36    PC37    PC38
## Standard deviation     8.44738 8.37305 8.21579 8.15731 7.97465 7.90446 7.82127
## Proportion of Variance 0.01045 0.01026 0.00988 0.00974 0.00931 0.00915 0.00896
## Cumulative Proportion  0.80072 0.81099 0.82087 0.83061 0.83992 0.84907 0.85803
##                            PC39    PC40    PC41   PC42    PC43   PC44    PC45
## Standard deviation     7.72156 7.58603 7.45619 7.3444 7.10449 7.0131 6.95839
## Proportion of Variance 0.00873 0.00843 0.00814 0.0079 0.00739 0.0072 0.00709
## Cumulative Proportion  0.86676 0.87518 0.88332 0.8912 0.89861 0.9058 0.91290
##                           PC46    PC47    PC48    PC49    PC50    PC51    PC52
## Standard deviation     6.8663 6.80744 6.64763 6.61607 6.40793 6.21984 6.20326
## Proportion of Variance 0.0069 0.00678 0.00647 0.00641 0.00601 0.00566 0.00563
## Cumulative Proportion  0.9198 0.92659 0.93306 0.93947 0.94548 0.95114 0.95678
##                            PC53    PC54    PC55    PC56    PC57   PC58    PC59
## Standard deviation     6.06706 5.91805 5.91233 5.73539 5.47261 5.2921 5.02117
## Proportion of Variance 0.00539 0.00513 0.00512 0.00482 0.00438 0.0041 0.00369
## Cumulative Proportion  0.96216 0.96729 0.97241 0.97723 0.98161 0.9857 0.98940
##                            PC60    PC61    PC62    PC63      PC64
## Standard deviation     4.68398 4.17567 4.08212 4.04124 1.951e-14
## Proportion of Variance 0.00321 0.00255 0.00244 0.00239 0.000e+00
## Cumulative Proportion  0.99262 0.99517 0.99761 1.00000 1.000e+00
```

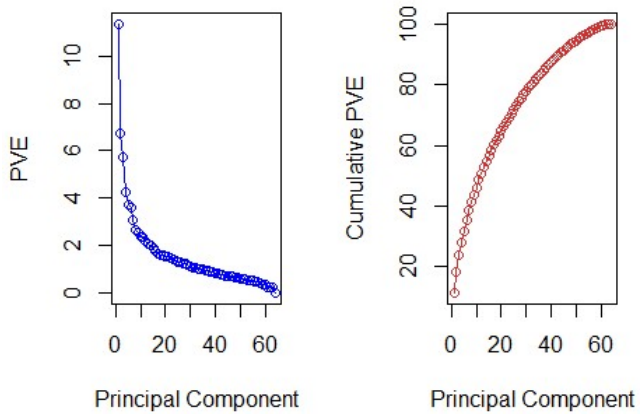
`plot()`関数を利用して、最初のいくつかの主成分が説明する分散をプロットすることもできる。

```
plot(pr.out)
```



棒グラフの各棒の高さは、対応する `pr.out$sdev` 要素を二乗することで得られる。ただし、各主成分の PVE（スクリープロット）と累積 PVE をプロットする方がより有益であり、これは次の作業で実現できる。

```
pve <- 100 * pr.out$sdev^2 / sum(pr.out$sdev^2)
par(mfrow = c(1, 2))
plot(pve,  type = "o", ylab = "PVE",
    xlab = "Principal Component", col = "blue")
plot(cumsum(pve), type = "o", ylab = "Cumulative PVE",
    xlab = "Principal Component", col = "brown3")
```



`pve` の要素は、`summary(pr.out)$importance[2, ]`から直接計算することもできまる。同様に `cumsum(pve)`の要素は `summary(pr.out)$importance[3, ]`によって得られる。これらの結果のプロットは図 12.18 に示されている。最初の 7 つの主成分がデータの分散のおよそ40%を説明することが分かる。これは大きな割合とは言えないが、スクリープロットを見ると、最初の 7 つの主成分のそれぞれがかなりの分散を説明する一方、それ以降の主成分による分散の説明は著しく減少している。こ

れは、7 つ目の主成分以降にエルボー（肘）があることを示唆しており、7 つ以上の主成分を調べることに大きな利点はないのかもしれない（ただし、7 つの主成分を調べること自体が難しい場合もある）。

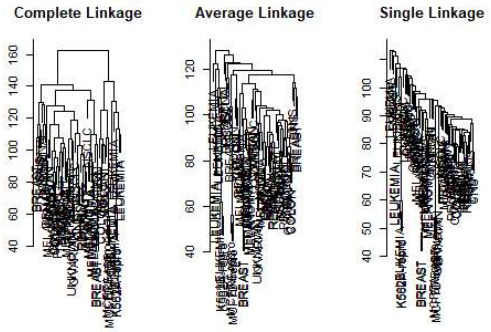
NCI60 データの観測値のクラスタリング

次に、`NCI` データの細胞株に階層型クラスタリングを行う。この作業は、観測値が異なるがんタイプにクラスタ化するかどうかを調べることを目的としている。まず、変数を平均 0、標準偏差 1 に標準化する。このステップは任意であるが、遺伝子を同じスケールに揃えたい場合にのみ実行する必要がある。

```
sd.data <- scale(nci.data)
```

次に、ユークリッド距離を不類似度尺度として利用、完全連結法、単一連結法、および平均連結法による観測値の階層型クラスタリングを実行する。

```
par(mfrow = c(1, 3))
data.dist <- dist(sd.data)
plot(hclust(data.dist), xlab = "", sub = "", ylab = "",
    labels = nci.labs, main = "Complete Linkage")
plot(hclust(data.dist, method = "average"),
    labels = nci.labs, main = "Average Linkage",
    xlab = "", sub = "", ylab = "")
plot(hclust(data.dist, method = "single"),
    labels = nci.labs,  main = "Single Linkage",
    xlab = "", sub = "", ylab = "")
```



結果は図 12.19 に示されているが、連結法の選択が結果に影響することが分かる。通常、単一連結法は*ヲ/きずり*クラスタを生成する傾向がある。つまり非常に大きなクラスタに観測値が一つずつ追加される傾向がある。一方、完全連結法と平均連結法では、より均衡の取れた魅力的なクラスタを生成する傾向がある。このため、通常、完全連結法と平均連結法が単一連結法よりも好まれる。

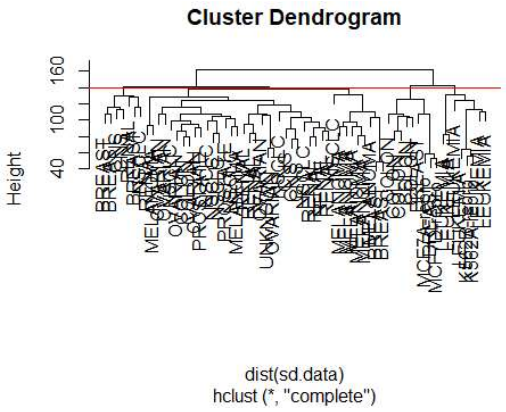
単一のがんタイプに属する細胞株がクラスタ内でまとまりやすいことが分かるが、クラスタリングは完全ではない。この後の分析では完全連結法を利用する。

次に、特定のクラスタ数（例：4）を得るように樹形図を切断してみよう。

```
hc.out <- hclust(dist(sd.data))
hc.clusters <- cutree(hc.out, 4)
table(hc.clusters, nci.labs)
##            nci.labs
## hc.clusters BREAST CNS COLON K562A-repro K562B-repro LEUKEMIA MCF7A-repro
##           1      2   3     2           0           0        0           0
##           2      3   2     0           0           0        0           0
##           3      0   0     0           1           1        6           0
##           4      2   0     5           0           0        0           1
##            nci.labs
## hc.clusters MCF7D-repro MELANOMA NSCLC OVARIAN PROSTATE RENAL UNKNOWN
##           1           0        8     8       6        2     8       1
##           2           0        0     1       0        0     1       0
##           3           0        0     0       0        0     0       0
##           4           1        0     0       0        0     0       0
```

いくつかの明確なパターンが見られる。すべての白血病細胞株はクラスタ3に属し、一方、乳がん細胞株は 3 つの異なるクラスタに分布している。これらの 4 つのクラスタを生成する切断を樹形図上にプロットされている。

```
par(mfrow = c(1, 1))
plot(hc.out, labels = nci.labs)
abline(h = 139, col = "red")
```



`abline()`関数は、既存のプロットに直線を描画する。引数 `h = 139` は高さ139の水平線を樹形図にプロットすることを意味する。この高さが 4 つの異なるクラスタを生成する。結果として得られたクラスタが `cutree(hc.out, 4)`を利用して得られるものと同じであることは簡単に確認できる。

`hclust` の出力を表示すると、有用だが簡潔な要約が得られる。

```
hc.out
## Call:
## hclust(d = dist(sd.data))
## Cluster method   : complete
## Distance         : euclidean
## Number of objects: 64
```

12.4.2 節で述べたように、K-平均法と階層的クラスタリングは、デンドログラムを適切にカットして同じクラスタ数を得たとしても、全く異なる結果をもたらすことがある。

では、`NCI` データに対する階層的クラスタリングの結果と、$K = 4$ の K-平均クラスタリングを比較するとどうなるだろうか？

```
set.seed(2)
km.out <- kmeans(sd.data, 4, nstart = 20)
km.clusters <- km.out$cluster
table(km.clusters, hc.clusters)
```

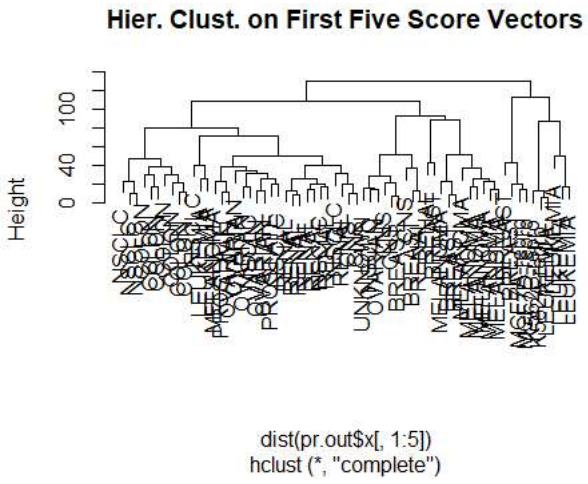
```
##            hc.clusters
## km.clusters  1  2  3  4
##           1 11  0  0  9
##           2 20  7  0  0
##           3  9  0  0  0
##           4  0  0  8  0
```

この結果を見ると、階層的クラスタリングとK-平均クラスタリングで得られた 4 つのクラスタは多少異なっている。

例えば、K-平均法のクラスタ 4 は、階層的クラスタリングのクラスタ 3 と完全に一致する。しかし、その他のクラスタには違いが見られる。具体的には、K-平均法のクラスタ 2 は、階層的クラスタリングのクラスタ 1 に分類された一部の観測値と、クラスタ 2 に分類されたすべての観測値を含んでいる。

データ全体に対して階層的クラスタリングを行うのではなく、最初のいくつかの主成分スコアベクトルに対して階層的クラスタリングを行うことも可能である。

```
hc.out <- hclust(dist(pr.out$x[, 1:5]))
plot(hc.out, labels = nci.labs,
    main = "Hier. Clust. on First Five Score Vectors")
```




```
table(cutree(hc.out, 4), nci.labs)
```

```
##    nci.labs
##     BREAST CNS COLON K562A-repro K562B-repro LEUKEMIA MCF7A-repro MCF7D-repro
##   1      0   2     7           0           0        2           0           0
##   2      5   3     0           0           0        0           0           0
##   3      0   0     0           1           1        4           0           0
##   4      2   0     0           0           0        0           1           1
##    nci.labs
##     MELANOMA NSCLC OVARIAN PROSTATE RENAL UNKNOWN
##   1        1     8       5        2     7       0
##   2        7     1       1        0     2       1
##   3        0     0       0        0     0       0
##   4        0     0       0        0     0       0
```

この結果は、データ全体に対して階層的クラスタリングを行った場合と異なる。場合によっては、最初のいくつかの主成分スコアベクトルに対してクラスタリングを実行することで、データ全体に対してクラスタリングを実行するよりも良い結果を得ることができる。 このような場合、主成分分析のステップはデータのノイズ除去の役割を果たしていると考えられる。
また、データ全体ではなく、最初のいくつかの主成分スコアベクトルに対してK-平均クラスタリングを適用することも可能である。

ISLR 第 13 章 実習：多重検定 (Multiple Testing)

仮説検定の復習

まず、`t.test()`関数を用いて幾つかの一標本t検定を実行してみよう。最初に、10 個の観測値を持つ 100 個の変数を作成する。最初の 50 個の変数は平均 0.5、分散 1 を持ち、残りは平均 0、分散 1 である。

```
set.seed(6)
x <- matrix(rnorm(10 * 100), 10, 100)
x[, 1:50] <- x[, 1:50] + 0.5
```

`t.test()`関数により一標本または二標本t検定を実行できる。デフォルトでは、一標本検定が実行される。最初に、$H_0: \mu_1 = 0$、すなわち最初の変数の平均がゼロであるという帰無仮説を検定してみよう。

```
t.test(x[, 1], mu = 0)
##  One Sample t-test
## data:  x[, 1]
## t = 2.0841, df = 9, p-value = 0.06682
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  -0.05171076  1.26242719
## sample estimates:
## mean of x
## 0.6053582
```

p値は 0.067 であり、$\alpha = 0.05$の水準では帰無仮説を棄却するには十分ではない。この場合、$\mu_1 = 0.5$なので帰無仮説は偽である。したがって、帰無仮説が偽である

にもかかわらず、棄却しなかったため、第二種過誤を犯している。 次に、$j = 1, \ldots, 100$について、$H_{0j}: \mu_j = 0$を検定する。100 個のp値を計算し、p値が 0.05 以下であればH_{0j}を棄却し、0.05 より大きければ棄却しないという判定を記録するベクトルを作成する。

```
p.values <- rep(0, 100)
for (i in 1:100)
  p.values[i] <- t.test(x[, i], mu = 0)$p.value
decision <- rep("Do not reject H0", 100)
decision[p.values <= .05] <- "Reject H0"
```

このシミュレーション・データセットでは、表 13.2 に似た2×2の表を作成できる。

```
table(decision,
    c(rep("H0 is False", 50), rep("H0 is True", 50))
  )
## decision           H0 is False H0 is True
##   Do not reject H0          40         47
##   Reject H0                 10          3
```

その結果、$\alpha = 0.05$の水準で、偽の帰無仮説 50 個のうち 10 個を棄却し、真の帰無仮説 50 個のうち 3 個を誤って棄却した。13.3 節の記法を使用すると、$W = 40$、$U = 47$、$S = 10$、$V = 3$である。なおこの表の行と列は表 13.2 と逆になっています。 $\alpha = 0.05$と設定しているため、真の帰無仮説のおよそ 5%を棄却すると予想される。上記の表では、真の帰無仮説 50 個のうち$V = 3$個を棄却したことを示している。

上記のシミュレーションでは、偽の帰無仮説について平均と標準偏差の比が $0.5/1 = 0.5$で、信号は非常に弱い。その結果、多数の第二種過誤が発生した。偽の帰無仮説について平均と標準偏差の比が 1 となるように信号を強くシミュレーションした場合、第二種過誤は 9 回に減少する。

```
x <- matrix(rnorm(10 * 100), 10, 100)
x[, 1:50] <- x[, 1:50] + 1
for (i in 1:100)
```

```
  p.values[i] <- t.test(x[, i], mu = 0)$p.value
decision <- rep("Do not reject H0", 100)
decision[p.values <= .05] <- "Reject H0"
table(decision,
    c(rep("H0 is False", 50), rep("H0 is True", 50))
  )
## decision           H0 is False H0 is True
##   Do not reject H0           9         49
##   Reject H0                 41          1
```

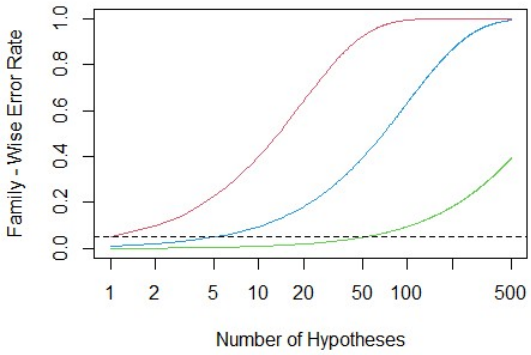
ファミリー全体の誤り率（FWER）

(13.5)から、m個の独立した仮説検定において、それぞれの帰無仮説が真である場合、FWER は$1-(1-\alpha)^m$に等しいことを思い出しておこう。この式を用いて、$m = 1, \ldots, 500$および$\alpha = 0.05, 0.01, 0.001$について FWER を計算する。

```
m <- 1:500
fwe1 <- 1 - (1 - 0.05)^m
fwe2 <- 1 - (1 - 0.01)^m
fwe3 <- 1 - (1 - 0.001)^m
```

これら 3 つのベクトルをプロットし、図 13.2 を再現する。赤、青、緑の線はそれぞれ$\alpha = 0.05, 0.01, 0.001$に対応している。

```
par(mfrow = c(1, 1))
plot(m, fwe1, type = "l", log = "x", ylim = c(0, 1), col = 2,
    ylab = "Family - Wise Error Rate",
    xlab = "Number of Hypotheses")
lines(m, fwe2, col = 4)
lines(m, fwe3, col = 3)
abline(h = 0.05, lty = 2)
```



中程度の m（例えば 50）でも、αが非常に低い値（例えば 0.001）に設定されない限り、FWER が 0.05 を超えることが分かる。しかし、αを非常に低い値に設定すると、第二種過誤が多発する可能性があり、つまり検出力が非常に低くなる。

ここで我々は、`Fund` データセットの最初の 5 人のマネージャーについて 1 標本t検定を行い、j番目のファンドマネージャーの平均リターンが 0 であるという帰無仮説$H_{0j}: \mu_j = 0$を検証してみよう。

```
library(ISLR2)
fund.mini <- Fund[, 1:5]
t.test(fund.mini[, 1], mu = 0)
##  One Sample t-test
## data:  fund.mini[, 1]
## t = 2.8604, df = 49, p-value = 0.006202
## alternative hypothesis: true mean is not equal to 0
## 95 percent confidence interval:
##  0.8923397 5.1076603
## sample estimates:
## mean of x
##         3
fund.pvalue <- rep(0, 5)
for (i in 1:5)
  fund.pvalue[i] <- t.test(fund.mini[, i], mu = 0)$p.value
fund.pvalue
## [1] 0.006202355 0.918271152 0.011600983 0.600539601 0.755781508
```

p値は、マネージャー1 とマネージャー3 で低く、他の 3 人のマネージャーでは高いことが分かる。 しかし、H_{01}とH_{03}を単純に棄却することはできないが、これは、行った多重検定を考慮しないためなのである。ここではその代わりに、FWER（家族誤差率）を制御するために、ボンフェローニ法とホルム法を実行する。

この作業を行うために、`p.adjust()`関数を利用する。この関数は、p値を入力として、多重検定に対して調整された新しいp値を出力する。ある仮説の調整済みp値がα以下である場合、その仮説は FWER がαを超えないことを維持しつつ棄却できる。言い換えれば、`p.adjust()`関数から得られる調整済みp値を希望の FWER と比較するだけで、各仮説を棄却するかどうかを判断できるのである。

例えば、ボンフェローニ法の場合、生のp値に仮説の総数mを掛けることで調整済みp値が得られる（ただし、調整済みp値は 1 を超えない）。

```
p.adjust(fund.pvalue, method = "bonferroni")
## [1] 0.03101178 1.00000000 0.05800491 1.00000000 1.00000000
pmin(fund.pvalue * 5, 1)
## [1] 0.03101178 1.00000000 0.05800491 1.00000000 1.00000000
```

したがって、ボンフェローニ法を利用すると、FWER を 0.05 に制御した上で、マネージャー1 の帰無仮説のみを棄却することができる。一方、ホルム法を使用すると、調整済みp値は、FWER が 0.05 の場合に、マネージャー1 とマネージャー3 の帰無仮説を棄却できることを示している。

```
p.adjust(fund.pvalue, method = "holm")
## [1] 0.03101178 1.00000000 0.04640393 1.00000000 1.00000000
```

前述の通り、マネージャー1 は特に良好なパフォーマンスを示しているように見え、マネージャー2 のパフォーマンスは低い。

```
apply(fund.mini, 2, mean)
## Manager1 Manager2 Manager3 Manager4 Manager5
##      3.0     -0.1      2.8      0.5      0.3
```

それではこれら 2 人のマネージャー間にパフォーマンスの有意な差がある証拠はあるだろうか？`t.test()`関数を利用して対応のあるt検定を行うと、p値は 0.038 となり、有意な差があることを示唆している。

```
t.test(fund.mini[, 1], fund.mini[, 2], paired = T)
##  Paired t-test
## data:  fund.mini[, 1] and fund.mini[, 2]
## t = 2.128, df = 49, p-value = 0.03839
## alternative hypothesis: true mean difference is not equal to 0
## 95 percent confidence interval:
##  0.1725378 6.0274622
## sample estimates:
## mean difference
##             3.1
```

ただし、この検定は、データを調べた結果、マネージャー1 と 2 が最高および最低の平均パフォーマンスを持つことに気付いた後に実施することを決定している。これは、13.3.2 節で述べたように、単一の仮説検定ではなく、暗黙的に$\binom{5}{2} = 5(5-1)/2 = 10$の仮説検定を行ったことを意味している。そのため、`TukeyHSD()`関数を利用して、Tukey 法を適用、多重検定を調整する必要がある。この関数は、回帰モデル（本質的にはすべての予測変数が質的である線形回帰）を入力とする。この場合、応答は各マネージャーが達成した月間超過リターンであり、予測変数は各リターンがどのマネージャーに対応しているかを示している。

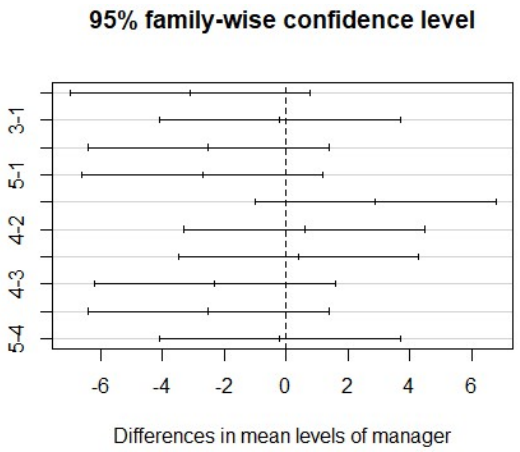
```
returns <- as.vector(as.matrix(fund.mini))
manager <- rep(c("1", "2", "3", "4", "5"), rep(50, 5))
a1 <- aov(returns ~ manager)
TukeyHSD(x = a1)
##   Tukey multiple comparisons of means
##     95% family-wise confidence level
## Fit: aov(formula = returns ~ manager)
## $manager
##     diff        lwr       upr     p adj
## 2-1 -3.1 -6.9865435 0.7865435 0.1861585
## 3-1 -0.2 -4.0865435 3.6865435 0.9999095
```

```
## 4-1 -2.5 -6.3865435 1.3865435 0.3948292
## 5-1 -2.7 -6.5865435 1.1865435 0.3151702
## 3-2  2.9 -0.9865435 6.7865435 0.2452611
## 4-2  0.6 -3.2865435 4.4865435 0.9932010
## 5-2  0.4 -3.4865435 4.2865435 0.9985924
## 4-3 -2.3 -6.1865435 1.5865435 0.4819994
## 5-3 -2.5 -6.3865435 1.3865435 0.3948292
## 5-4 -0.2 -4.0865435 3.6865435 0.9999095
```

`TukeyHSD()`関数は、各マネージャー間の差（`lwr`と`upr`）の信頼区間とp値を提供してくれ、これらの量はすべて多重検定に対して調整されている。

マネージャー1 と 2 の差に関するp値は 0.038 から 0.186 に増加しており、マネージャーのパフォーマンス間に明確な差がある証拠はなくなっている。`plot()`関数を利用して、ペア間比較の信頼区間がプロットしてみよう。

```
plot(TukeyHSD(x = a1))
```



偽発見率 (False Discovery Rate)

ここでは、`Fund`データセット内の 2,000 人すべてのファンドマネージャーに対して仮説検定を実行しよう。$H_{0j}: \mu_j = 0$（j番目のファンドマネージャーの平均リターンが 0 である）という仮説について 1 標本t検定を行おう。

```
fund.pvalues <- rep(0, 2000)
for (i in 1:2000)
  fund.pvalues[i] <- t.test(Fund[, i], mu = 0)$p.value
```

マネージャーの数が多すぎるため、FWER（家族誤差率）を制御しようとするのは現実的ではない。代わりに、FDR（偽発見率）、すなわち実際には偽陽性である棄却された帰無仮説の割合の期待値を制御することに焦点を当ててみよう。`p.adjust()`関数を利用して、ベンジャミニ-ホッホベルグ法を実行できる。

```
q.values.BH <- p.adjust(fund.pvalues, method = "BH")
q.values.BH[1:10]
##  [1] 0.08988921 0.99149100 0.12211561 0.92342997 0.95603587 0.07513802
##  [7] 0.07670150 0.07513802 0.07513802 0.07513802
```

ベンジャミニ-ホッホベルグ法によって出力されたq値は、特定の帰無仮説を棄却する最小の FDR 閾値として解釈できる。例えば、q値が 0.1 であれば、対応する帰無仮説を FDR が 10%以上で棄却できることを意味するが、FDR が 10%未満では棄却できない。

FDR を 10%に制御する場合、$H_{0j}: \mu_j = 0$を棄却できるファンドマネージャーの数はいくつだろうか？

```
sum(q.values.BH <= .1)
## [1] 146
```

2,000 人のファンドマネージャーのうち 146 人がq値 0.1 未満であることが分かった。したがって、146 人のファンドマネージャーが市場を上回っていると結論づけることができる（FDR が 10%で）。このうち約 15 人（146 の 10%）が偽発見である可能性がある。対照的に、FWER を$\alpha = 0.1$に制御するためにボンフェローニ法を用いた場合、どの帰無仮説も棄却できなかっただろう。

```
sum(fund.pvalues <= (0.1 / 2000))
## [1] 0
```

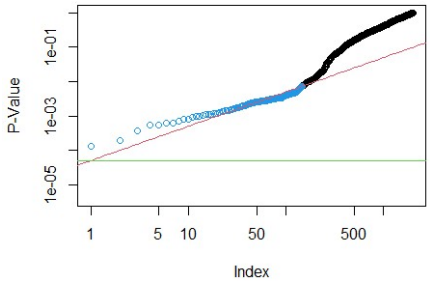
図 13.6 は、Fund データセットの順序付けられたp値、$p_{(1)} \leq p_{(2)} \leq \cdots \leq p_{(2000)}$、およびベンジャミニ-ホッホベルグ法による棄却の閾値を示している。ベンジャミニ-ホッホベルグ法は、最大のp値を探し、それが$p_{(j)} < qj/m$を満たす場合、そのp値を棄却する。ここでコードを書いてこの方法を再現し、仕組みを説明しよう。

まず、p値を順序付ける。その後、$p_{(j)} < qj/m$ を満たすすべてのp値を特定する（`wh.ps`）。最後に、`wh` は `wh.ps` 内の最大p値以下のp値をインデックス化する。 したがって、`wh` はベンジャミニ-ホッホベルグ法で棄却されたp値をインデックス化していることになる。

```
ps <- sort(fund.pvalues)
m <- length(fund.pvalues)
q <- 0.1
wh.ps <- which(ps < q * (1:m) / m)
if (length(wh.ps) >0) {
  wh <- 1:max(wh.ps)
 } else {
  wh <- numeric(0)
 }
```

次に、図 13.6 の中央パネルを再現してみよう。

```
plot(ps, log = "xy", ylim = c(4e-6, 1), ylab = "P-Value",
    xlab = "Index", main = "")
points(wh, ps[wh], col = 4)
abline(a = 0, b = (q / m), col = 2, untf = TRUE)
abline(h = 0.1 / 2000, col = 3)
```



リサンプリング手法

ここで 13.5 節において調べた `Khan` データセットを利用して仮説検定のリサンプリング手法を実装しよう。まず、トレーニングデータとテストデータを統合する。この操作により 83 人の患者に関する 2,308 個の遺伝子のデータが得られる。

```
attach(Khan)
x <- rbind(xtrain, xtest)
y <- c(as.numeric(ytrain), as.numeric(ytest))
dim(x)
## [1]   83 2308
table(y)
## y
##  1  2  3  4
## 11 29 18 25
```

がんは 4 つのクラスに分類されている。各遺伝子について、第 2 クラス（横紋筋肉腫）と第 4 クラス（バーキットリンパ腫）の平均発現量を比較しよう。11 番目の遺伝子について標準的な 2 標本t検定を行うと、検定統計量は-2.09、対応するp値は 0.0412 となり、2 つのがんタイプ間で平均発現量に違いがあることを示すには控えめな証拠となる。

```
x <- as.matrix(x)
x1 <- x[which(y == 2), ]
x2 <- x[which(y == 4), ]
n1 <- nrow(x1)
n2 <- nrow(x2)
t.out <- t.test(x1[, 11], x2[, 11], var.equal = TRUE)
TT <- t.out$statistic
TT
##         t
## -2.093633
t.out$p.value
## [1] 0.04118644
```

ただし、このp値は、2 つのグループ間に差がないという帰無仮説の下で、検定統計量が自由度$29 + 25 - 2 = 52$のt分布に従うという仮定に依存している。この理論的な帰無分布を利用する代わりに、54 人の患者を 29 人と 25 人の 2 つのグループにランダムに分割し、新しい検定統計量を計算してみよう。グループ間に差がないという帰無仮説の下では、この新しい検定統計量は元の検定統計量と同じ分布を持つはずである。

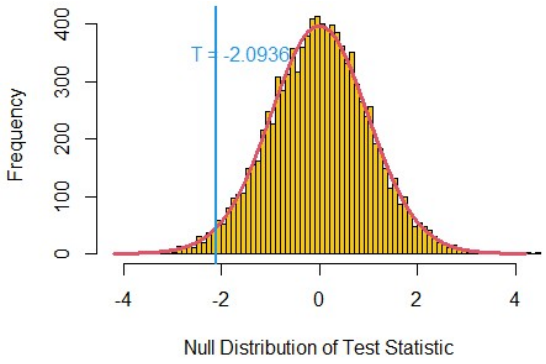
このプロセスを 1 万回繰り返すことで、検定統計量の帰無分布を近似する。観測された検定統計量がリサンプリングによって得られた検定統計量を超える頻度の割合を計算する。

```
set.seed(1)
B <- 10000
Tbs <- rep(NA, B)
for (b in 1:B) {
   dat <- sample(c(x1[, 11], x2[, 11]))
   Tbs[b] <- t.test(dat[1:n1], dat[(n1 + 1):(n1 + n2)],
        var.equal = TRUE
      )$statistic
}
mean((abs(Tbs) >= abs(TT)))
## [1] 0.0416
```

この割合、0.0416 は、リサンプリングに基づくp値となる。この値は、理論的な帰無分布を用いて得られたp値 0.0412 とほぼ同一である。リサンプリングに基づく検定統計量のヒストグラムをプロットし、図 13.7 を再現してみよう。

```
hist(Tbs, breaks = 100, xlim = c(-4.2, 4.2), main = "",
    xlab = "Null Distribution of Test Statistic", col = 7)
lines(seq(-4.2, 4.2, len = 1000),
    dt(seq(-4.2, 4.2, len = 1000),
      df = (n1 + n2 - 2)
    ) * 1000, col = 2, lwd = 3)
abline(v = TT, col = 4, lwd = 2)
```

```
text(TT + 0.5, 350, paste("T = ", round(TT, 4), sep = ""),
    col = 4)
```



リサンプリングに基づく帰無分布は、赤色で表示された理論的な帰無分布とほぼ一致している。

最後に、アルゴリズム 13.4 で説明されているリサンプリング FDR 手法を実装する。`Khan` データセットの全 2,308 個の遺伝子に対して FDR を計算するのは時間がかかる可能性がある。そのため、ここでランダムに選択した 100 個の遺伝子に対してこの手法を説明する。各遺伝子について、観測された検定統計量を計算し、1 万回リサンプリングされた検定統計量を生成する。

これには実行に数分かかる場合があります。急いでいる場合は、`B` を小さな値（例：`B = 500`）に設定すればよい。

```
m <- 100
B <- 1000
set.seed(1)
index <- sample(ncol(x1), m)
Ts <- rep(NA, m)
Ts.star <- matrix(NA, ncol = m, nrow = B)
for (j in 1:m) {
  k <- index[j]
  Ts[j] <- t.test(x1[, k], x2[, k],
         var.equal = TRUE
       )$statistic
  for (b in 1:B) {
```

```
    dat <- sample(c(x1[, k], x2[, k]))
    Ts.star[b, j] <- t.test(dat[1:n1],
         dat[(n1 + 1):(n1 + n2)], var.equal = TRUE
       )$statistic
  }
}
```

次に、アルゴリズム 13.4 で示されている閾値cの範囲に対して、棄却された帰無仮説の数R、偽陽性数の推定値$\hat{V}$,および推定 FDR を計算しよう。閾値cは 100 個の遺伝子の検定統計量の絶対値を用いて選択する。

```
cs <- sort(abs(Ts))
FDRs <- Rs <- Vs <- rep(NA, m)
for (j in 1:m) {
  R <- sum(abs(Ts) >= cs[j])
  V <- sum(abs(Ts.star) >= cs[j]) / B
  Rs[j] <- R
  Vs[j] <- V
  FDRs[j] <- V / R
}
```

指定された FDR で棄却される遺伝子を見つけることができる。例えば、FDR を 0.1 に制御した場合、100 個の帰無仮説のうち 15 個を棄却する。このうち 1〜2 個（15 の 10%）が偽発見であると予想される。FDR が 0.2 の場合、帰無仮説を 28 個棄却でき、そのうち約 6 個が偽発見であると予想される。`index` 変数は、分析をランダムに選択された 100 個の遺伝子に制限したために必要となる。

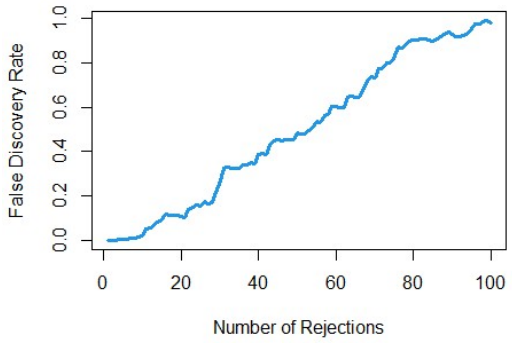
```
max(Rs[FDRs <= .1])
## [1] 15
sort(index[abs(Ts) >= min(cs[FDRs < .1])])
##  [1]   29  465  501  554  573  729  733 1301 1317 1640 1646 1706 1799 1942 2
159
max(Rs[FDRs <= .2])
## [1] 28
```

```
sort(index[abs(Ts) >= min(cs[FDRs < .2])])
```

```
##  [1]   29   40  287  361  369  465  501  554  573  679  729  733  990 1069 1
073
## [16] 1301 1317 1414 1639 1640 1646 1706 1799 1826 1942 1974 2087 2159
```

次の行は図 13.11 を生成する。この図は、図 13.9 に類似しているが、遺伝子のサブセットに基づいている。

```
plot(Rs, FDRs, xlab = "Number of Rejections", type = "l",
    ylab = "False Discovery Rate", col = 4, lwd = 3)
```



この章で述べたように、FDR 計算のリサンプリング手法のより効率的な実装が `R` の `samr` パッケージなどを用いれば利用可能である。