Part-II：Pythonによる統計的学習

ISLP 第 2 章 ラボ：Python 入門

はじめに(Getting Started)

この書籍にある Labs を実行するためには、以下の２つが必要となる。

- `Python3` のインストール。これは Labs に使用されている `Python` の特定のバージョンである。

訳注 1：古いバージョンとして `Python2` があるが構文が少し異なり、Python3 とは完全には互換性がない。本書でのすべての Labs は `Python3` の構文を前提としている。

訳注 2：なお初心者への Python 入門としては 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。

- `Jupyter` へのアクセス。これはとても人気のある `Python` のインターフェースで、*notebook* と呼ばれるファイルを介してコードを実行する。

anaconda.com にある次の案内に従えば `Python3` のダウンロードとインストールを行うことができる。

You can download and install `Python3` by following the instructions available at anaconda.com.

`Jupyter` にアクセスする方法はいくつかあるので挙げておく。

- Google の `Colaboratory` サービスを利用する: colab.research.google.com/.
- `JupyterHub` を利用する： こちらから入手できる→ jupyter.org/hub.
- 自分でインストールした `jupyter` を利用する。 インストールの仕方は jupyter.org/install に記載されている。

読者の環境で使用する `Python` および `Jupyter` の最新情報を入手するためには、書籍のウェブサイト statlearning.com で `Python` の資料を確認してみよう。

Labs を実行するためには `ISLP` パッケージをインストールする必要がある。これは、我々が提供するデータセットと独自機能へのアクセスを提供してくれる。macOS もしくは Linux のターミナル内で、`pip install ISLP` を入力すれば、Labs の実行に必要なパッケージをインストールできる。`Python` の資料ページに `ISLP` の説明ウェブサイト(documentation)へのリンクがある。

訳注 3：日本語訳作成段階では、`ISLP` の documentation は intro-stat-learning.github.io/ISLP/ にある。また、Windows 向けのインストール方法についても記載があるので、Windows ユーザーは確認しておこう。

この Lab を実行するには、`Ch2-statlearn-lab.ipynb` を `Python` の資料ページからファイルをダウンロードするとよい。例えば次のコードをコマンドラインで実行する：`jupyter lab Ch2-statlearn-lab.ipynb`

Windows を利用している場合、`anaconda` にアクセスするために `start menu` と次のリンクを使用する。例えば、`ISLP` のインストールとこの Lab の実行のために、同じコードを `anaconda` シェルで実行することができる。

訳注 4：`anaconda` を Windows にインストールした場合には、`Anaconda Prompt` というソフトが多くの場合同時にインストールされると思われる（本文中で `anaconda` シェルと呼ばれているのはこれを意味する）。Linux ターミナル（Windows でいうコマンドプロンプトのような CUI ツール）で特定のコードを実行するように指示された際には、この `Anaconda Prompt` で同じコードを実行してみるとよいだろう。`pip` や `conda` で始まる Python のパッケージ管理に関するコードであれば、これでおおむねうまくいくはずである。

追加訳注 1：パッケージ ISLP は初めからインストールされていないので必要になれば `%%capture !pip install ISLP` とすればよい。

追加訳注２：Jupyter Notebook の場合には立ち上がったら New をクリック、Python 3 を選んで開始する。Jupyter では例えば ctrl+v がコピー・ペースト、Run が実行である。Google Colaboratory はネット上の「Colab にようこそ」に利用法が説明されているが、初回は「ノートブックを新規作成」を利用してみるとよいだろう。

基本コマンド(Basic Commands)

この Lab では、シンプルな `Python` コマンドをいくつか導入する。 `Python` に関する詳細については、docs.python.org/3/tutorial/にあるチュートリアルが参考になる。

ほとんどのプログラミングと同じく、`Python` は関数 *function* を処理の実行に使用する。 `fun` 関数を呼び出すためには、`fun(input1, input2)`と打ち込む。ここで、入力（もしくは引数 *argument*）である `input1` と `input2` が、関数がどのよ

うに実行されるべきかを Python に伝える。関数はいくつでも入力を持つことができる。例えば、print()関数はすべての引数についてその文字列表現をコンソールに出力する。

In [1]:

```
print('fit a model with', 11, 'variables')
```

Out[1]

```
fit a model with 11 variables
```

次のコマンドは print()関数に関する情報を返す。

In [2]:

```
print?
```

Python で２つの整数を足すのはとても直感的にできる。

In [3]:

```
3 + 5
```

Out[3]:

```
8
```

Python では、文字的なデータは文字列 *string* で扱う。例えば、"hello"と'hello'は文字列である。 これらは加算記号+を使って結合できる。

訳注 5：結合すると、２つの文字列が１つの文字列になるという意味である。

In [4]:

```
"hello" + " " + "world"
```

Out[4]:

```
'hello world'
```

文字列は実際には、配列 *sequence* の一種であり、これは順序付きリストに関する一般的な用語になる。配列には重要な３つの型があり、リスト・タプル・そして文字列である。

まずはリストを導入してみよう。

次のコマンドは、数値 3,4,5 を結合したものを、`x` というリスト *list* として保存するように `Python` に指示するものである。 `x` と入力すると、このリストが返ってくる。

In [5]:

```
x = [3, 4, 5]
x
```

Out[5]:

```
[3, 4, 5]
```

リストを作るために、かっこ`[]`が使われていることを覚えておくとよい。２つの数値のまとまりを足したいときがある？ このときには次のコードを試すのは良さそうに思えるが、実際には期待した結果は返ってこない。

In [6]:

```
y = [4, 9, 7]
x + y
```

Out[6]:

```
[3, 4, 5, 4, 9, 7]
```

この結果は、少し非直感的に思えるるだろう：なぜ `Python` はリストの中身を要素ごとに足さないのだろうか？

`Python`では、リストは内部に*任意の*オブジェクトを抱えることができ、足し算は*結合*になります。実際、結合は少し前に試した`"hello" + " " + "world"`と同じ挙動をする。

訳注 6：文字列はリストなので、リストを`+`でつないだときの挙動は、文字列を`+`でつないだときの挙動と同じになるべきだ、という意味である。

この結果は、`Python`が汎用のプログラミング言語である事実を反映している。`Python`の多くのデータ関連の機能は他のパッケージ、特に`numpy`と`pandas`を利用するために呼び出すことになる。

次のセクションでは、`numpy`パッケージを導入する。より詳しい内容を知るには docs.scipy.org/doc/numpy/user/quickstart.html を参照するとよい。

Python による数値計算

すでに述べたように、この本は`numpy`ライブラリ *library*（またはパッケージ *package*）に含まれる機能を使用する。パッケージはモジュールの集合体で、必ずしも`Python`の基本的なディストリビューションに含まれているわけではない。`numpy`という名前は、*numerical Python* の略である。

訳注 7：Python の環境は、必要なソフトウェアなどをセットにした「ディストリビューション」という形で配布されていることが多い。

`numpy`にアクセスするためには、まずこれを`import`する必要がある。

In [7]:

```
import numpy as np
```

上の行では、`numpy`モジュール *module* を`np`と命名している。これは参照を簡単にするための略称である。

訳注 8：Python では R と異なり、使用する関数やクラスをどのパッケージが提供しているのかを逐一明示するのが一般的である。例えば、`numpy`の`dot`関

数を使用する場合には、`numpy.dot` という形で関数を呼び出す。このとき、上記のように略称を指定しておけば、`np.dot` と短く呼び出すことができる。

`numpy` において、配列 *array* は数値の多次元集合に関する一般的な用語である。１次元配列（例えば、ベクトル）である `x` と `y` を定義するのに、`np.array()`を使用する。

In [8]:

```
x = np.array([3, 4, 5])
y = np.array([4, 9, 7])
```

あらかじめ `import numpy as np` コマンドを実行するのを忘れた場合、`np.array()` 関数の呼び出しがエラーになることに注意しよう。 構文 `np.array()`は、その呼び出された関数が（`np` という略称をつけた）`numpy` パッケージの一部であることを意味している。

`x` と `y` は `np.array()`で定義されているので、これらを足したときには先ほど期待したような結果を得ることができる。`numpy` を使用しないで変数を足そうとしたときに起きた前のセクションの結果と比較してみよう。

In [9]:

```
x + y
```

Out[9]:

```
array([ 7, 13, 12])
```

`numpy` では、行列は典型的には２次元の配列で、ベクトルは１次元の配列を意味する（`np.matrix()`で行列を構成することもできるが、本書の Labs では一貫して `np.array()`を使用する）。

訳注 9：`np.matrix` を使った場合、関数やメソッドの挙動が異なる場合があるので注意しよう。

２次元の配列は次のように構成できる。

In [10]:

```
x = np.array([[1, 2], [3, 4]])
x
```

Out[10]:

```
array([[1, 2],
       [3, 4]])
```

このときオブジェクト `x` はいくつかの属性 *attribute*（紐づけられたオブジェクト）を持っている。

訳注 10：Python を扱う上で出てくるすべての「モノ」は「オブジェクト」と見なしてよい。

`x` の属性を知るには `x.attribute` と入力する（`attribute` は属性の名称に置き換える必要がある。 例えば、`x` の属性 `ndim` には次のようにアクセスできる。

In [11]:

```
x.ndim
```

Out[11]:

```
2
```

この出力は、`x` が 2 次元配列であることを意味する。 同様に、`x.dtype` は `x` のデータ型 *data type* 属性であるが、これは `x` が 64 ビット整数からなること意味する。

In [12]:

```
x.dtype
```

Out[12]:

```
dtype('int64')
```

なぜ `x` は整数型なのだろうか？ それは関数 `np.array()`に整数だけを渡して `x` を作ったためである。 もし何かしらの小数を含む形で渡していたら、浮動小数点数 *floating point numbers*（例えば実数）の配列を取得していたことになる。

In [13]:

```
np.array([[1, 2], [3.0, 4]]).dtype
```

Out[13]:

```
dtype('float64')
```

`fun?`と入力すれば、`Python` は `fun` 関数に関する文書を（もしあれば）表示する。ここで `np.array()`に対して試してみよう。

In [14]:

```
np.array?
```

この文書から、`np.array()`の `dtype` 引数に（訳注：**`float`** を）渡すことで、浮動小数点の配列を作れることが分かる。

In [15]:

```
np.array([[1, 2], [3, 4]], float).dtype
```

Out[15]:

```
dtype('float64')
```

この配列 `x` は２次元の配列である。属性 `shape` を見ることで、行と列の数を確認できる。

In [16]:

```
x.shape
```

Out[16]:

```
(2, 2)
```

メソッド *method* (訳注：法と訳すことがある)とは、オブジェクトに紐づけられた関数である。 例えば、配列 `x` があるとしよう。このとき `sum()`メソッドを使えば、表記 `x.sum()`によって、そのすべての要素を足し合わせることができる。`x.sum()`と呼び出すと、自動的に `x` が `sum()`メソッドの最初の引数として渡される。

In [17]:

```
x = np.array([1, 2, 3, 4])
x.sum()
```

Out[17]:

```
10
```

`np.sum()`関数の引数に `x` を渡すことで、同様に `x` のすべての要素を足すこともできる。

In [18]:

```
x = np.array([1, 2, 3, 4])
np.sum(x)
```

Out[18]:

```
10
```

別の例として、`reshape()`メソッドは `x` と同じ要素だが異なる形

訳注 11：shape とは多次元配列のかたち（次元数）のことで、例えば行列の場合は行と列の個数のこと

を持つ新しい配列を返す。 これは、`reshape()`を呼び出すときに、`tuple`（この場合`(2, 3)`）を渡すことで実行できる。 このタプルは、2 行 3 列の 2 次元配列を作りたい、と指定している。

（リストのように、タプル *tuple* はオブジェクトの列を表す。なぜ列を作る方法が複数あるのだろうか？ タプルとリストにはいくつかの違いがあるが、たぶん最も重要な違いは、タプルの要素は変更できない一方で、リストの要素は変更できるという点だろう。）

訳注 **12**：リストは`[]`で、タプルは`()`で指定する。また配列の個々の要素に名前がついている辞書 **dictionary** という列もあり、これは`{key: value}`で指定する。辞書 `d` に対して、`d[key]`と指示すると、その `key` に対応する `value` が返ってくる。

以下では、文字`\n`は改行 *new line* を生成する。

In [19]:

```
x = np.array([1, 2, 3, 4, 5, 6])
print('beginning x:\n', x)
x_reshape = x.reshape((2, 3))
print('reshaped x:\n', x_reshape)
```

Out[19]:

```
beginning x:
 [1 2 3 4 5 6]
reshaped x:
 [[1 2 3]
 [4 5 6]]
```

上の出力では、`numpy` 配列が行 *row* の配列で指定されていることを明らかにしている。 これは、行優先順序 *row-major ordering* と呼ばれるが、その反対は列優先順序 *column-major ordering* と呼ばれる。

訳注 **13**：利用者目線で重要なのは、２次元配列 `A` について `A[i]`と呼び出したときに、`i` 番目の行が返ってくるのか、`i` 番目の列が返ってくるのか、というのが言語によって異なるという意味である。

`Python`（と `numpy`）は０ベースのインデックスを採用している。 このことから `x_reshape` の左上の要素にアクセスするときには、`x_reshape[0,0]`と打ち込むとことを意味する。

In [20]:

```
x_reshape[0, 0]
```

Out[20]:

```
1
```

同様に、`x_reshape[1,2]`は `x_reshape` の２行目３列目の要素を返す。

In [21]:

```
x_reshape[1, 2]
```

Out[21]:

```
6
```

同様に、`x[2]`は `x` の３番目の要素を返す。

ここで、`x_reshape` の左上の要素を変更してみよう。驚くべきことに、`x` の最初の要素も同様に変更されてしまうことが分かる！

In [22]:

```
print('x before we modify x_reshape:\n', x)
print('x_reshape before we modify x_reshape:\n', x_reshape)
x_reshape[0, 0] = 5
print('x_reshape after we modify its top left element:\n', x_reshape)
print('x after we modify top left element of x_reshape:\n', x)
```

Out[22]:

```
x before we modify x_reshape:
 [1 2 3 4 5 6]
x_reshape before we modify x_reshape:
 [[1 2 3]
 [4 5 6]]
x_reshape after we modify its top left element:
 [[5 2 3]
 [4 5 6]]
x after we modify top left element of x_reshape:
 [5 2 3 4 5 6]
```

`x_reshape` の修正は `x` の修正も意味する。これは、これら２つのオブジェクトがメモリ上の同じ空間を占有しているために起きる。

ここで配列の要素を変更できることが分かった。それではタプルも変更できるのだろうか？ 試してみると例外 *exception* あるいはエラーが出てきて、これはできないことが分かる。

In [23]:

```
my_tuple = (3, 4, 5)
my_tuple[0] = 2
```

Out[23]:

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-23-1bcb6270a2d5> in <cell line: 2>()
      1 my_tuple = (3, 4, 5)
----> 2 my_tuple[0] = 2

TypeError: 'tuple' object does not support item assignment
```

次に便利な配列の属性を簡単に紹介しよう。配列の `shape` 属性は、その次元（常にタプルです）を格納している。 `ndim` 属性は次元の個数を返し、`T` はその転置を作成する。

In [24]:

```
x_reshape.shape, x_reshape.ndim, x_reshape.T
```

Out[24]:

```
((2, 3),
 2,
 array([[5, 4],
        [2, 5],
        [3, 6]]))
```

３つの個々の出力`(2,3)`,`2`,と`array([[5, 4],[2, 5], [3,6]])`は、それ自体はタプルとしての出力であることに注意しよう。

配列に関数を適用したい場合がある。例えば、入力の平方根は`np.sqrt()`関数を使って計算できる。

In [25]:

```
np.sqrt(x)
```

Out[25]:

```
array([2.23606798, 1.41421356, 1.73205081, 2.        , 2.23606798,
       2.44948974])
```

要素の平方もできる。

In [26]:

```
x**2
```

Out[26]:

```
array([25,  4,  9, 16, 25, 36])
```

同じ記法で平方根も計算できる。これは冪指数を 2 の代わりに 1/2 とすればよい。

In [27]:

```
x**0.5
```

Out[27]:

```
array([2.23606798, 1.41421356, 1.73205081, 2.        , 2.23606798,
       2.44948974])
```

この書籍を通して、何度もランダム・データを生成することになる。`np.random.normal()`は正規乱数のベクトルを生成する関数である。 ヘルプ・ページを `np.random.normal?`と呼び出すことで、この関数について学ぶことができる。

このヘルプ・ページの最初の行には `normal(loc=0.0, scale=1.0, size=None)`とある。 このシグネチャ *signature* 行から、この関数の引数が `loc`, `scale`, `size` であると分かる。これらはキーワード *keyword* 引数であり、関数に渡されたときに（順序を問わずに）この名前で参照されることを意味する。 （`Python` は位置 *positional* 引数も使用する。位置引数はキーワードを使用する必要はない。例としては、`np.sum?`と入力してみよう。`a` は位置引数であると分かる。つまりこの関数は、受け取った最初の命名されていない引数を、合計するべき配列だと仮定しているわけである。対照的に、`axis` と `dtype` はキーワード引数で、`np.sum()`にこれらの引数が入力されたときの位置は問題にはならない。） デフォルトでは、この関数はこの関数は平均（`loc`）0 で標準偏差（`scale`）1 の正規乱数を生成する。さらに、引数 `size` が変更されない限り、単一の乱数が生成される。

ここで 50 個の独立な N(0,1)分布に従う乱数を生成してみよう。

In [28]:

```
x = np.random.normal(size=50)
x
```

Out[28]:

```
array([ 0.09313007, -1.00213876,  0.40980384,  0.26009698,  0.00581396,
        0.38917189,  0.26271527,  1.32891437,  0.87585302,  0.04583093,
       -0.52332176,  0.23624863,  1.24240248,  0.58554176,  0.73474189,
        0.60272886,  0.2134124 ,  0.73493102,  0.30037075,  0.55572978,
       -1.35475916, -1.23217513,  0.53453489,  0.52575205,  0.69550517,
       -0.64371278, -0.34127968,  1.40970744, -0.85776983,  0.86939974,
        0.92651936,  1.90393838, -0.45044323, -1.27094066,  0.17669491,
       -0.51955211, -1.05043372, -1.71488943,  1.19432317,  3.20117233,
        1.68067744,  0.15021894, -0.4928851 , -0.01746229, -0.46400596,
       -1.49475908, -2.18936919,  0.63661893, -3.22863971, -0.63108822])
```

独立な N(50,1)乱数を `x` の各要素に足すことで配列 `y` を作成しよう。

In [29]:

```
y = x + np.random.normal(loc=50, scale=1, size=50)
```

`np.corrcoef()`関数は、`x` と `y` の相関行列を計算する。非対角成分が `x` と `y` の相関である。

In [30]:

```
np.corrcoef(x, y)
```

Out[30]:

```
array([[1.        , 0.77137767],
       [0.77137767, 1.        ]])
```

もし自分で `Jupyter`notebook によりここまでのプログラムを追っているのであれば、以前のいくつかのコマンドを実行したときに、少し異なる結果が出てきたことに気付いたかもしれない。具体的には、`np.random.normal()`を呼び出すたびに、次に例示するような異なる答えが得られる。

IN [31]:

```
print(np.random.normal(scale=5, size=2))
print(np.random.normal(scale=5, size=2))
```

Out[31]:

```
[0.07186452 3.92111129]
[0.08295237 7.11480268]
```

プログラムが毎回まったく同じ結果を返すように保証するために、`np.random.default_rng()`関数に *random seed*（ランダムシード）をセットすることができる。この関数により任意のユーザ指定の整数を引数としてとることになる。もし乱数生成前にランダムシードを設定していれば、プログラムを再度実行しても同じ結果が得られる。`rng` オブジェクトは基本的に `np.random` にある

すべての乱数生成メソッドを持っている。正規乱数生成のためには `rng.normal()`を使えばよい。

In [32]:

```
rng = np.random.default_rng(1303)
print(rng.normal(scale=5, size=2))
rng2 = np.random.default_rng(1303)
print(rng2.normal(scale=5, size=2))
```

Out[32]:

```
[ 4.09482632 -1.07485605]
[ 4.09482632 -1.07485605]
```

この本では Labs を通して、`numpy`による乱数を含む計算を行うときにはいつでも `np.random.default_rng()`を使用する。 これにより原則として、読者は全く同じ結果を再現できる。 ただし、`numpy`の新しいバージョンが出たときには、`numpy`からの出力に小さな矛盾が生じる可能性はある。

関数 `np.mean()`, `np.var()`, `np.std()`は、配列の平均・分散・標準偏差を計算するために使用できる。 これらの関数は配列のメソッドとしても使うことができる。

In [33]:

```
rng = np.random.default_rng(3)
y = rng.standard_normal(10)
np.mean(y), y.mean()
```

Out[33]:

```
(-0.1126795190952861, -0.1126795190952861)
```

In [34]:

```
np.var(y), y.var(), np.mean((y - y.mean())**2)
```

Out[34]:

```
(2.7243406406465125, 2.7243406406465125, 2.7243406406465125)
```

ここでデフォルト設定では`np.var()`は n-1 ではなく、サンプルサイズ n で割り算することに注意しておく。 この点については`np.var?`を呼び出して引数`ddof`について確認しておこう。

訳注 **14**：デフォルト設定では不偏分散ではなく標本分散が計算されるということを意味する。

In [35]:

```
np.sqrt(np.var(y)), np.std(y)
```

Out[35]:

```
(1.6505576756498128, 1.6505576756498128)
```

関数`np.mean()`,`np.var()`,`np.std()`は行列の列と行に対して適用することもできる。このことを確認するために、N(0,1)に従う確率変数の 10x 3 行列を構築して、その行和計算を考えてみよう。

In [36]:

```
X = rng.standard_normal((10, 3))
X
```

Out[36]:

```
array([[ 0.22578661, -0.35263079, -0.28128742],
       [-0.66804635, -1.05515055, -0.39080098],
       [ 0.48194539, -0.23855361,  0.9577587 ],
       [-0.19980213,  0.02425957,  1.54582085],
       [ 0.54510552, -0.50522874, -0.18283897],
       [ 0.54052513,  1.93508803, -0.26962033],
       [-0.24355868,  1.0023136 , -0.88645994],
       [-0.29172023,  0.88253897,  0.58035002],
```

```
       [ 0.0915167 ,  0.67010435, -2.82816231],
       [ 1.02130682, -0.95964476, -1.66861984]])
```

配列は列優先順序となっているので、最初の軸（つまり `axis=0`）はその列を指している。 この引数をオブジェクト `X`

訳注 15：numpy 配列である。

の `mean()`メソッドに渡している。

In [37]:

```
X.mean(axis=0)
```

Out[37]:

```
array([ 0.15030588,  0.14030961, -0.34238602])
```

次は全く同じ結果を返す。

In [38]:

```
X.mean(0)
```

Out[38]:

```
array([ 0.15030588,  0.14030961, -0.34238602])
```

グラフィックス

`Python` では一般的に、`matplotlib` ライブラリをグラフィックスに使用する。 しかし `Python` はデータ分析を念頭に書かれたわけではないので、プロット記法は言葉から直感的に意味が分かるわけではない。 ここでは `matplotlib.pyplot` の `subplots()`関数を、データをプロットする figure（図）と axes（軸）の作成に使用しよう。 `Python` でのプロット方法に関する多くの例を見るには読者には matplotlib.org/stable/gallery/を開いてみることを勧めておこう。

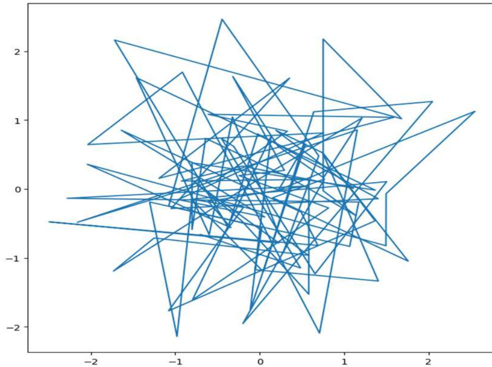
`matplotlib` では、プロットは１つの *figure*（図）と１つ以上の *axes*（軸）によって構成される。 figure は１つ以上のプロットが表示される白紙のキャンパス

と考えてよい：これがプロットウィンドウの全体となる。 *axes* はx-軸, y-軸のラベルやタイトルなどの、各プロットにおける重要な情報を含む。（`matplotlib` においては、*axes* という用語は *axis*（軸）の複数形ではない。プロットの *axes* は単に x 軸や y 軸だけでなくそれ以上の情報を持っている。）

`matplotlib` から `subplots()`をインポートするところから始めてみよう。 図を作るときにはこの関数をずっと使用する。 この関数は、長さ２のタプル：figure オブジェクトと、それに関連する axes オブジェクトを返す。 典型的には `figsize` をキーワード引数として渡すことになる。 axes ができたら、その `plot()`法を使って最初のプロットをしてみよう。 この方法については `ax.plot?` と打てば説明を読むことができる。

In [39]:

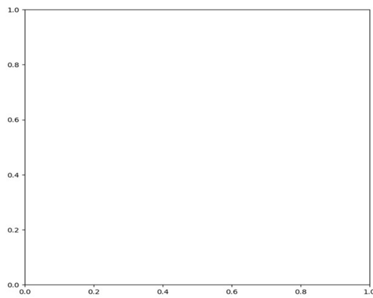
```
from matplotlib.pyplot import subplots
fig, ax = subplots(figsize=(8, 8))
x = rng.standard_normal(100)
y = rng.standard_normal(100)
ax.plot(x, y);
```



ここでいったん立ち止まって、*unpacked*（アンパックされた）長さ２のタプル（これは `subplots()`が返したもの）が２つの別の変数 `fig` と `ax` になっていることを確認しよう。 アンパッキングはふつう、次のような等価な（しかし少々冗長な）コードよりも好まれる。

In [40]:

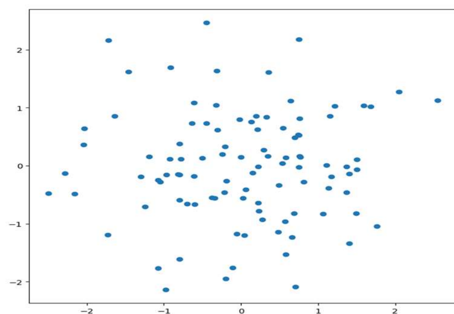
```
output = subplots(figsize=(8, 8))
fig = output[0]
ax = output[1]
```



前のセルがデフォルト設定であるラインプロットを生成していることが確認できる。 散布図を作成するには、`ax.plot`に円を示す追加の引数を与える。

In [41]:

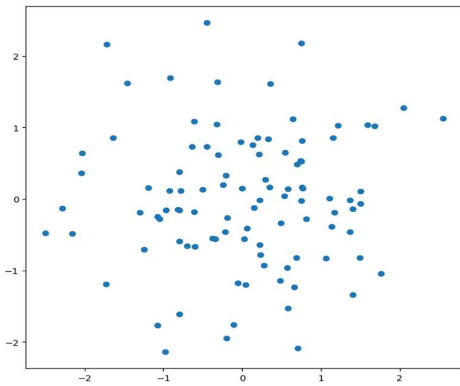
```
fig, ax = subplots(figsize=(8, 8))
ax.plot(x, y, 'o');
```



この追加引数に異なる値を使うと異なる色の線や線のスタイルを設定できる。別の方法として、`ax.scatter()`関数を散布図の作成に使うこともできる。

In [42]:

```
fig, ax = subplots(figsize=(8, 8))
ax.scatter(x, y, marker='o');
```



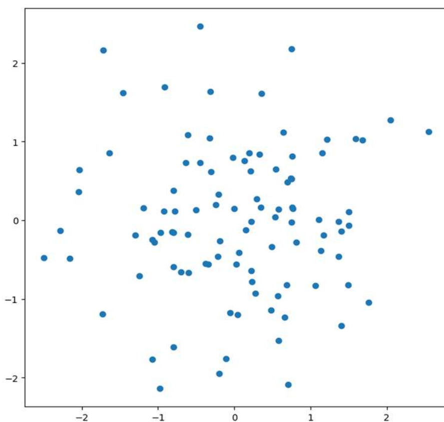
これまでのコード・ブロックでは、最後の行がセミコロンで終わっているのに気が付いただろうか？ これはノートブックに`ax.plot(x, y)`と表示されるのを防いでくれている。 しかし、プロットは生成されている。 ここで末尾のセミコロンを取り除いたら、次のような出力が得られる。

In [43]:

```
fig, ax = subplots(figsize=(8, 8))
ax.scatter(x, y, marker='o')
```

Out[43]:

```
<matplotlib.collections.PathCollection at 0x7d79947fd060>
```



以下では、議論に関係のない文章が出力される場合にはいつでも末尾のセミコロンを付与しておく。プロットにラベルを付けるために、`ax`のメソッド`set_xlabel()`, `set_ylabel()`, `set_title()`を使用する。

In [44]:

```
fig, ax = subplots(figsize=(8, 8))
ax.scatter(x, y, marker='o')
ax.set_xlabel("this is the x-axis")
ax.set_ylabel("this is the y-axis")
ax.set_title("Plot of X vs Y");
```

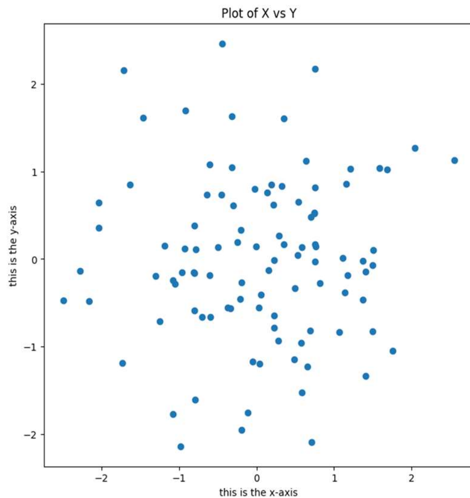
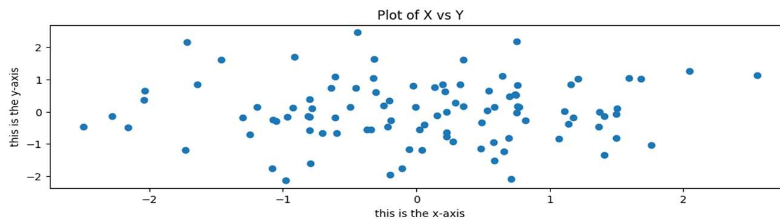


figure オブジェクトである `fig` にアクセスすることで、アスペクトの変更などを行い、それを再表示することが可能となる。ここではサイズを`(8, 8)`から`(12, 3)`に変更してみよう。

In [45]:

```
fig.set_size_inches(12,3)
fig
```

Out[46]:



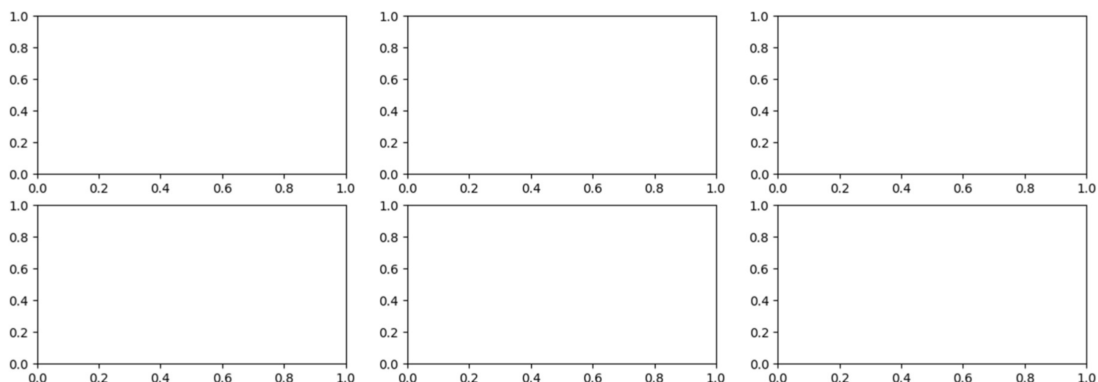
ときには、いくつかのプロットを 1 つの図に入れたいこともあるだろう。 このためには、追加の引数を `subplots()`に与えればよい。 以下では、`figsize` 引数で大きさを定められた figure の中に、プロットを配置できる 2x3 のグリッドを

作成してみよう。 そのような状況では、プロット内の axes に何らかの関係性があることが多く、 たとえばすべてのプロットが共通の x 軸を持っている場合もある。 キーワード引数 `sharex=True` を渡したとき、`subplots()`関数はこのような状況を自動的に取り扱ってくれる。 以下の `axes` オブジェクトは、figure 内の異なるプロットを示す配列となっている。

訳注 16：figure 内の異なるプロットに対応する axes をまとめた配列が、下のコードの `axes` オブジェクトになっているということを意味する。

In [46]:

```
fig, axes = subplots(nrows=2,
                     ncols=3,
                     figsize=(15, 5))
```

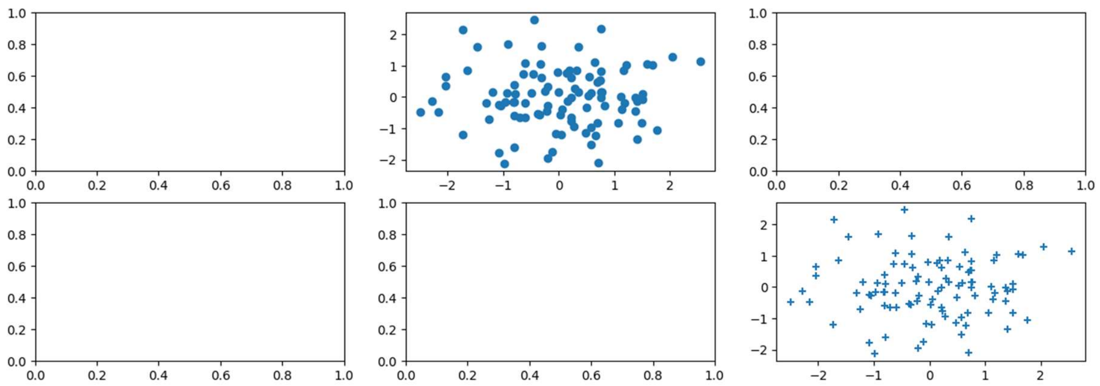


１行目の２列目に`'o'`を指定した散布図を、２行目の３列目に`'+'`を指定した散布図を作ってみる。

In [48]:

```
axes[0,1].plot(x, y, 'o')
axes[1,2].scatter(x, y, marker='+')
fig
```

Out[47]:



`subplots()`についてより詳しく知りたい場合は、`subplots?`と入力すればよい。`fig`の出力を保存するには、その`savefig()`法を呼び出す。引数`dpi`は dots per inch （インチ当たりドット数）であり、ピクセル規準で図がどれくらい大きくするかを指定するために使われる。

In [48]:

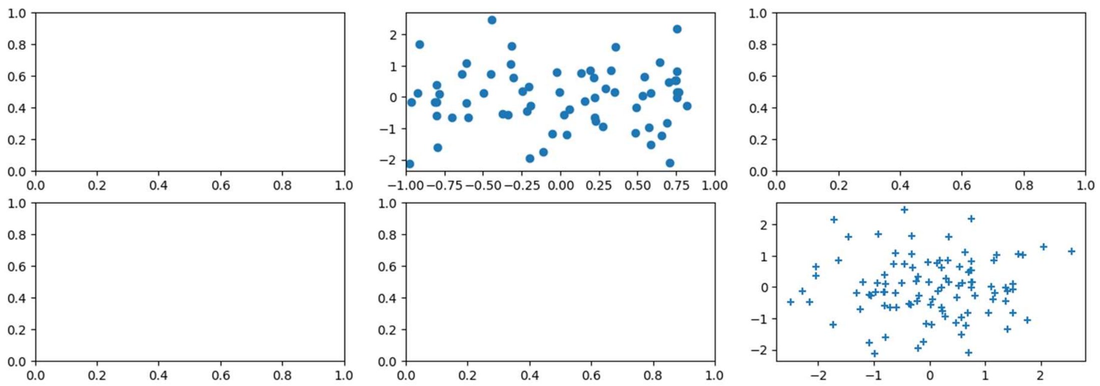
```
fig.savefig("Figure.png", dpi=400)
fig.savefig("Figure.pdf", dpi=200);
```

`fig`を 1 つずつ順に更新して行くことができる；例えば、x 軸の範囲を修正し、図を再度保存し、そして再表示することが可能である。

In [49]:

```
axes[0,1].set_xlim([-1,1])
fig.savefig("Figure_updated.jpg")
fig
```

Out[49]:



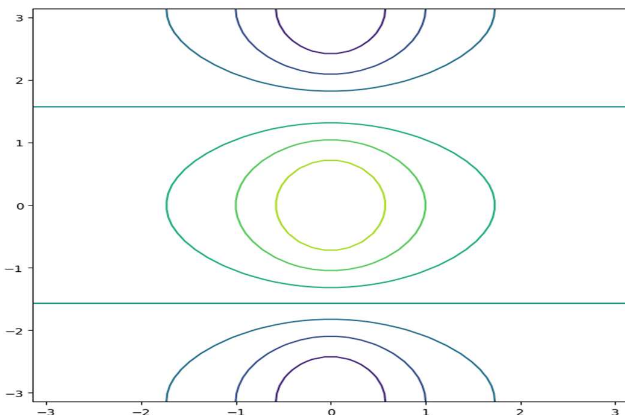
ここからはより洗練されたプロットを作ってみよう。３次元データを表現するときには、`ax.contour()`メソッドで地形図のような *contour plot*（等高線プロット）を作ることができる。これは３つの引数をとる。

- `x` 値のベクトル（１つ目の次元）
- `y` 値のベクトル（２つ目の次元）
- `(x,y)`座標のペアに対する `z` 値（３つ目の次元）に要素が対応する行列

`x` と `y` を生成するためにコマンド `np.linespace(a, b, n)`を使用する。これは `a` で始まり `b` で終わる `n` 個の数値を含むベクトルを返す。

In [50]:

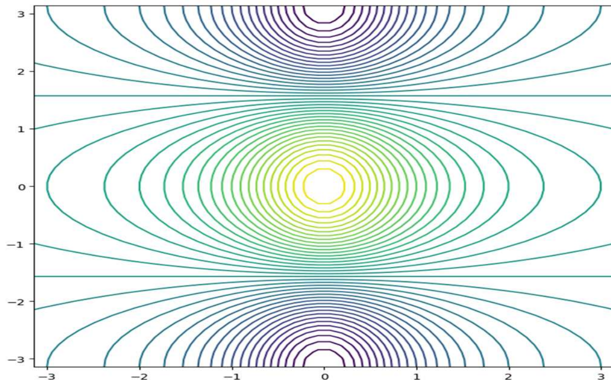
```
fig, ax = subplots(figsize=(8, 8))
x = np.linspace(-np.pi, np.pi, 50)
y = x
f = np.multiply.outer(np.cos(y), 1 / (1 + x**2))
ax.contour(x, y, f);
```



より多くのレベル(levels)を与えることで、解像度を上げることができる。

In [51]:

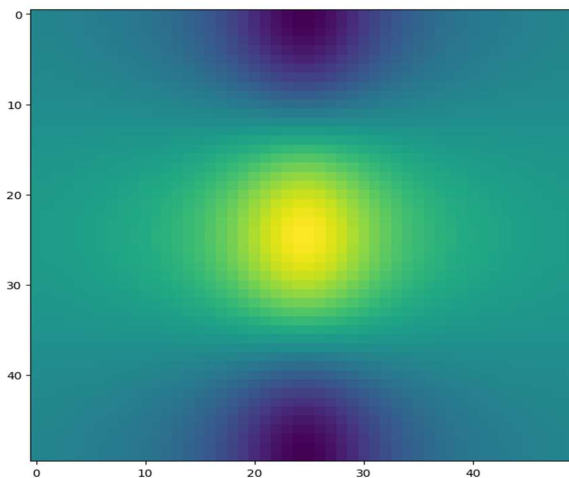
```
fig, ax = subplots(figsize=(8, 8))
ax.contour(x, y, f, levels=45);
```



`ax.contour()`関数の出力を改善するためには、`?plt.contour` と入力してヘルプファイルを閲覧してみよう。`ax.imshow()`法は`ax.contour()`に似ていますが、色が `z` 値に依存したカラーコードによるプロットが生成される点が異なる。 これは *heatmap*（ヒートマップ）として知られており、天気予報の気温をプロットするときなどに使用される。

In [52]:

```
fig, ax = subplots(figsize=(8, 8))
ax.imshow(f);
```



系列(Sequences)とスライス(Slice)

すでに見たように`np.linspace()`は数値の配列を作るのに利用される。

In [53]:

```
seq1 = np.linspace(0, 10, 11)
seq1
```

Out[53]:

```
array([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7.,  8.,  9., 10.])
```

`np.arange()`は間隔が`step`になっている数列を返す。`step`が指定されていない場合には、間隔のデフォルト値として 1 が使用される。ここで 0 で始まり 10 で終わる配列を生成してみよう。

In [54]:

```
seq2 = np.arange(0, 10)
seq2
```

Out[54]:

```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

なぜ 10 が出力されていないのだろうか？その理由は`Python`の *slice*（スライス）記法から分かる。スライス記法はリストやタプル、配列(array)などの配列(sequences)からデータを見つけるために使われる。

訳注 17：インデックスと言う。

ある文字列から 4 から 6 番目の要素を（末端を含む形で）取り出したいとしよう。`[3:6]`という記法によるインデックスで、この文字列のスライスをとることができる。

In [55]:

```
"hello world"[3:6]
```

Out[55]:

```
'lo '
```

ここで上のコードブロックにある `3:6` という記法は、`[]`の内部で使われている場合には `slice(3,6)`の略である。

In [56]:

```
"hello world"[slice(3,6)]
```

Out[56]:

```
'lo '
```

`slice(3,6)`と指定した場合には、文字列の 4 から 7 番目の文字が出力されることを期待するかもしれない（`Python` ではインデックスがゼロから始まることを思い出してみよう）が、代わりに 4 から 6 番目が出力される。 これは前の `np.arange(0, 10)`が 0 から 9 までの整数だけを出力する理由にもなっている。スライスを作るための有益な方法については `slice?`を参照してみると良い。

データの記述(Indexing Data)

まずは２次元の `numpy` 配列を作ろう。

In [57]:

```
A = np.array(np.arange(16)).reshape((4, 4))
A
```

Out[57]:

```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

`A[1,2]`と打てば、２行目３列目に対応する要素が抽出される（通常`Python`のインデックスは0始まりとなる。）

In [58]:

```
A[1,2]
```

Out[58]:

```
6
```

括弧開き記号`[`の直後の数値は行を、２番目の数値は列番号を表している。

行・列・部分行列(Indexing Rows, Columns, and Submatrices)

複数の行を一度に選ぶために、選択を指定するリストを渡すことができる。 例えば`[1,3]`は２行目と４行目を取り出せる。

In [59]:

```
A[[1,3]]
```

Out[59]:gyou

```
array([[ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

１列目と３列目を選ぶためには、`[0,2]`を四角い括弧（`[]`）の２番目の引数として渡す。 この場合最初の引数としては`:`を指定する必要がある。これによりすべての行を選択することを意味している。

In [60]:

```
A[:,[0,2]]
```

Out[60]:

```
array([[ 0,  2],
       [ 4,  6],
       [ 8, 10],
       [12, 14]])
```

それでは、２，４行目と１，３列目でできた部分行列を選びたいとしよう。これはややトリッキーなデータ作成であるがリストを行と列の抽出に使ってみる例としては自然だろう。

In [61]:

```
A[[1,3],[0,2]]
```

Out[61]:

```
array([ 4, 14])
```

おや、何が起こったのだろうか？ 次の結果と同じ一次元の配列が出力される。

In [62]:

```
np.array([A[1,0],A[3,2]])
```

Out[62]:

```
array([ 4, 14])
```

同じく、次のコードも２，４行目と１，３，４列目でできた部分行列を取り出すのに失敗する。

In [63]:

```
A[[1,3],[0,2,3]]
```

Out[63]:

```
---------------------------------------------------------------------
IndexError                                Traceback (most recent call last)
```

```
<ipython-input-64-ca8dff2976e8> in <cell line: 1>()
----> 1 A[[1,3],[0,2,3]]

IndexError: shape mismatch: indexing arrays could not be broadcast together
with shapes (2,) (3,)
```

何がうまくいっていないのだろうか？ ２つのインデックスリストを与えたとき、`numpy` は i,j ペアになっている一連のインデックスであると解釈している。 これがリストのペアが同じ長さでないといけない理由である。 しかし今は部分行列を見つけたいので、これは意図した挙動にはならない。

ここでの簡単な方法は次のようなものである。 まず `A` の行を部分的に指定して部分行列を作り、次にさらに列を部分的に指定することでその部分行列が作成される。

In [64]:

```
A[[1,3]][:,[0,2]]
```

Out[64]:

```
array([[ 4,  6],
       [12, 14]])
```

同じ結果が得られるもっと効果的な方法もある。

convenience function（利便性のための関数）である `np.ix_()`を使うと、中間的な *mesh*（メッシュ）オブジェクトを生成することにより、リストから部分行列を取り出せる。

In [65]:

```
idx = np.ix_([1,3],[0,2,3])
A[idx]
```

Out[65]:

```
array([[ 4,  6,  7],
       [12, 14, 15]])
```

さらに別の方法としてスライスを用いることで効果的に行列を部分指定できる。スライス `1:4:2` は配列の２番目と４番目の要素を取り出し、スライス `0:3:2` は１番目と３番目を取り出す（スライスの３番目の引数は間隔を指定している）。

In [66]:

```
A[1:4:2,0:3:2]
```

Out[66]:

```
array([[ 4,  6],
       [12, 14]])
```

リストではなくスライスを用いるとなぜ直接部分行列を取り出せるのだろうか？ これは、これらが異なる `Python` の型を持っており、`numpy` によって異なる取り扱いをされるためだからである。 スライスは文字列やリスト、タプルなどの任意の配列からオブジェクトを取り出すのに使えるが、インデックスに対するリストの利用はより限定的なのである。

ブール記法(Boolean Indexing)

`numpy` において、*Boolean*（ブーリアン）は `True`（真）または `False`（偽）（それぞれ 1 と 0 とも表現される）のいずれかの値をとる型である。 次の行は、Boolean として表現された、長さが `A` の最初の次元と等しい、0 で埋められたベクトルを生成する。

In [67]:

```
keep_rows = np.zeros(A.shape[0], bool)
keep_rows
```

Out[67]:

```
array([False, False, False, False])
```

このうち２つの要素に `True` をセットして見よう。

In [68]:

```
keep_rows[[1,3]] = True
keep_rows
```

Out[68]:

```
array([False,  True, False,  True])
```

keep_rows の要素は、整数としてみたときには、np.array([0,1,0,1])の値と同じになっている。なお以下では、==をこれらの等しさを確認するために利用する。２つの配列に適用した時には、==オペレーションは要素ごとに適用される。

In [69]:

```
np.all(keep_rows == np.array([0,1,0,1]))
```

Out[69]:

```
True
```

（ここでは関数 np.all()は配列のすべての要素が True であるかを確認した。似たような関数 np.any()は配列の要素に True があるかを確認するために使える。）

==によれば np.array([0,1,0,1]) と keep_rows は等しいが、これらは行の異なる集合を指している！ 前者は A の第１， ２， １， ２番目の行を取り出す。

In [70]:

```
A[np.array([0,1,0,1])]
```

Out[70]:

```
array([[0, 1, 2, 3],
       [4, 5, 6, 7],
       [0, 1, 2, 3],
       [4, 5, 6, 7]])
```

これに対して、`keep_rows`では`A`の２，４行目（つまり、Boolean が`True`に等しかった行）だけを取り出せる。

In [71]:

```
A[keep_rows]
```

Out[71]:

```
array([[ 4,  5,  6,  7],
       [12, 13, 14, 15]])
```

この例から、`numpy`においてはブーリアンと整数が異なる扱いをしていることが分かる。再度`np.ix_()`関数を使って、２，４番目の行と３，４番目の列を含むメッシュを作成してみよう。 今回は、リストではなくブーリアンに関数を適用する。

In [72]:

```
keep_cols = np.zeros(A.shape[1], bool)
keep_cols[[0, 2, 3]] = True
idx_bool = np.ix_(keep_rows, keep_cols)
A[idx_bool]
```

Out[72]:

```
array([[ 4,  6,  7],
       [12, 14, 15]])
```

`np.ix_()`の引数にブーリアンの配列でできたリストを混ぜることもできる。

In [73]:

```
idx_mixed = np.ix_([1,3], keep_cols)
A[idx_mixed]
```

Out[73]:

```
array([[ 4,  6,  7],
       [12, 14, 15]])
```

`numpy` におけるインデックス法については、先に挙げた `numpy` のチュートリアルを参照するとより詳しく知ることができる。

データ入力(Loading Data)

データセットはしばしば異なる型のデータを含んでおり、列や行に名前が紐づいていることもある。 このことから典型的にはデータフレームを用いて取り込むのがベストな方法と云える。 データフレームは、同じ長さの配列(array)--これらは列になる--でできた配列(sequence)と考えることができる。 異なる配列の見出しを結合して、行とすることができる。 データフレームオブジェクトを生成して取り扱うためには、`pandas` ライブラリを利用すればよい。

データの読み込み(Reading in a Data Set)

ほとんどの分析では、最初のステップでデータを `Python` に読み込む。 データセットを読み込もうとする前に、`Python` に対してファイルが置かれている場所を知らせる必要がある。 もしファイルがノートブックファイルと同じ場所にあれば、準備はできていることになっている。 そうでなければ、`os.chdir()` コマンドを使って*ディレクトリ変更*を行い(`os.chdir()` を呼ぶ前に `import os` を実行する必要がある)。

訳注 17: 例えば Jupyter を利用中に場所が分からなければコマンド (i) import os (ii) path=os.getcwd( ) (iii) print(path) とすると current directory が表示される。表示されている directory にファイルを置いておけばよい。Google Colaboratory (Colab)を利用する場合は少し複雑になる。まず Colab 上で (i) from google.colab import drive (ii) drive.mount(‘/content/drive’) として(初回は)指示に従い Drive の利用を許可、パスを指定する。次にファイルを Google Drive (Google メールに紐づけされている Google アプリの Google ドライブ上の MyDrive に Python と云う名のフォルダー)上に置く(Google ドライブ操作に慣れておく必要がある)。最後に Colab 上で (iii) iimport os (iv) os.chdir("/content/drive/MyDrive/Python/") とすればよい。

ここではまずは書籍ウェブトで入手できる `Auto.csv` を読むところから始めよう。これはコンマで分割されたファイルで、`pd.read_csv` を使って読むことができる。

In [74]:

```
import pandas as pd
Auto = pd.read_csv('Auto.csv')
Auto
```

Out[74]:

	Mpg	cylinders	displacement	horsepower	weight	acceleration	year	origin	Name
0	18.0	8	307.0	130	3504	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165	3693	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150	3436	11.0	70	1	plymouth satellite
3	16.0	8	304.0	150	3433	12.0	70	1	amc rebel sst
4	17.0	8	302.0	140	3449	10.5	70	1	ford torino
...	...	...	...	...	...	...	...	...	...
387	27.0	4	140.0	86	2790	15.6	82	1	ford mustang gl
388	44.0	4	97.0	52	2130	24.6	82	2	vw pickup
389	32.0	4	135.0	84	2295	11.6	82	1	dodge rampage
390	28.0	4	120.0	79	2625	18.6	82	1	ford ranger
391	31.0	4	119.0	82	2720	19.4	82	1	chevy s-10

392 rows × 9 columns

本書のウェブサイトには、空白文字で区切られたバージョンの `Auto.data` も置いてある。これは次のように読むことができる。

In [81]:

```
Auto = pd.read_csv('Auto.data', delim_whitespace=True)
```

`Auto.csv` と `Auto.data` はいずれも単純なテキストファイルである。`Python` で読み込む前に、テキストエディタや Microsoft Excel など別のソフトウェアで中身を閲覧してみると良いだろう。

そこでは変数 `horsepower` に対応する `Auto` の列を見てみよう。

In [82]:

```
Auto['horsepower']
```

Out[82]:

	Horsepower
0	130.0
1	165.0
2	150.0
3	150.0

4	140.0
...	...
392	86.00
393	52.00
394	84.00
395	79.00
396	82.00

397 rows × 1 columns

dtype: object

この列の `dtype` は `object` である。 これよりデータを読んだときに `horsepower` 列のすべての値は文字列として認識されたことが分かる。このことは各データの値を見れば分かる。

In [83]:

```
np.unique(Auto['horsepower'])
```

Out[83]:

```
array(['100.0', '102.0', '103.0', '105.0', '107.0', '108.0', '110.0',
       '112.0', '113.0', '115.0', '116.0', '120.0', '122.0', '125.0',
       '129.0', '130.0', '132.0', '133.0', '135.0', '137.0', '138.0',
       '139.0', '140.0', '142.0', '145.0', '148.0', '149.0', '150.0',
       '152.0', '153.0', '155.0', '158.0', '160.0', '165.0', '167.0',
       '170.0', '175.0', '180.0', '190.0', '193.0', '198.0', '200.0',
       '208.0', '210.0', '215.0', '220.0', '225.0', '230.0', '46.00',
       '48.00', '49.00', '52.00', '53.00', '54.00', '58.00', '60.00',
       '61.00', '62.00', '63.00', '64.00', '65.00', '66.00', '67.00',
       '68.00', '69.00', '70.00', '71.00', '72.00', '74.00', '75.00',
       '76.00', '77.00', '78.00', '79.00', '80.00', '81.00', '82.00',
       '83.00', '84.00', '85.00', '86.00', '87.00', '88.00', '89.00',
       '90.00', '91.00', '92.00', '93.00', '94.00', '95.00', '96.00',
       '97.00', '98.00', '?'], dtype=object)
```

問題は欠損としてエンコードされるべき値 `?があることが`分かる。問題を解消するためには、`pd.read_csv()`に引数 `na_values` を与える必要がある。 これで、ファイルの中の`?`はそれぞれ *not a number* を意味する値 `np.nan` に置き換えられる。

In [84]:

```
Auto = pd.read_csv('Auto.data',
                   na_values=['?'],
```

```
                    delim_whitespace=True)
Auto['horsepower'].sum()
```

Out[84]:

```
40952.0
```

`Auto.shape`属性から、データが397の観測値（もしくは行）と9つの変数（もしくは列）を持つことが分かる。

In [85]:

```
Auto.shape
```

Out[85]:

```
(397, 9)
```

データ欠損値を扱う方法はいくつかある。 この場合、5つの行だけが欠損値を含んでいるので、`Auto.dropna()`法を用いてこれらの行を単純に削除することを選ぶことも可能である。

In [86]:

```
Auto_new = Auto.dropna()
Auto_new.shape
```

Out[86]:

```
(392, 9)
```

行と列の取り出し方(Basics of Selecting Rows and Columns)

変数名の確認には`Auto.columns`を使用する。

In [87]:

```
Auto = Auto_new # overwrite the previous value
Auto.columns
```

Out[87]:

```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin', 'name'],
      dtype='object')
```

データフレームの行や列へのアクセスは、配列へのそれと似ていますが、全く同じというわけではない。`[]`法(メソッド)への最初の引数は常に配列の行に適用されたことを思い出してみよう。似たように、`[]`法にスライスを渡すと、その*行*がスライスによって指定されたデータフレームが生成される：

In [88]:

```
Auto[:3]
```

Out[88]:

	Mpg	Cylinders	displacement	horsepower	weight	acceleration	Year	Origin	Name
0	18.0	8	307.0	130.0	3504.0	12.0	70	1	chevrolet chevelle malibu
1	15.0	8	350.0	165.0	3693.0	11.5	70	1	buick skylark 320
2	18.0	8	318.0	150.0	3436.0	11.0	70	1	plymouth satellite

同じくブーリアンの配列を行の部分集合をとることに利用できる。

In [89]:

```
idx_80 = Auto['year'] > 80
Auto[idx_80]
```

Out[89]:（訳注：一部分を省略）

	Mpg	cylinders	Displacement	horsepower	weight	acceleration	year	origin	Name
338	27.2	4	135.0	84.0	2490.0	15.7	81	1	plymouth reliant
339	26.6	4	151.0	84.0	2635.0	16.4	81	1	buick skylark
340	25.8	4	156.0	92.0	2620.0	14.4	81	1	dodge aries wagon (sw)
341	23.5	6	173.0	110.0	2725.0	12.6	81	1	chevrolet citation
342	30.0	4	135.0	84.0	2385.0	12.9	81	1	plymouth reliant

343	39.1	4	79.0	58.0	1755.0	16.9	81	3	toyota starlet
344	39.0	4	86.0	64.0	1875.0	16.4	81	1	plymouth champ
345	35.1	4	81.0	60.0	1760.0	16.1	81	3	honda civic 1300
346	32.3	4	97.0	67.0	2065.0	17.8	81	3	Subaru
347	37.0	4	85.0	65.0	1975.0	19.4	81	3	datsun 210 mpg
348	37.7	4	89.0	62.0	2050.0	17.3	81	3	toyota tercel
.									
.									
.									
388	22.0	6	232.0	112.0	2835.0	14.7	82	1	ford granada l
389	32.0	4	144.0	96.0	2665.0	13.9	82	3	toyota celica gt
390	36.0	4	135.0	84.0	2370.0	13.0	82	1	dodge charger 2.2
391	27.0	4	151.0	90.0	2950.0	17.3	82	1	chevrolet camaro
392	27.0	4	140.0	86.0	2790.0	15.6	82	1	ford mustang gl
393	44.0	4	97.0	52.0	2130.0	24.6	82	2	vw pickup
394	32.0	4	135.0	84.0	2295.0	11.6	82	1	dodge rampage
395	28.0	4	120.0	79.0	2625.0	18.6	82	1	ford ranger
396	31.0	4	119.0	82.0	2720.0	19.4	82	1	chevy s-10

しかし、`[]`法に文字列のリストを渡した場合には、対応する*列*集合を持つデータ・フレームが得られる。

In [90]:

```
Auto[['mpg', 'horsepower']]
```

Out[90]:

	Mpg	horsepower
0	18.0	130.0
1	15.0	165.0
2	18.0	150.0
3	16.0	150.0
4	17.0	140.0
...	...	...
392	27.0	86.0
393	44.0	52.0
394	32.0	84.0
395	28.0	79.0
396	31.0	82.0

392 rows × 2 columns

データフレームを読み込んだ時には*インデックス*列を指定しないので列は 0 から 396 の整数でラベリングされている。

In [91]:

```
Auto.index
```

Out[91]:

```
Index([  0,   1,   2,   3,   4,   5,   6,   7,   8,   9,
       ...
       387, 388, 389, 390, 391, 392, 393, 394, 395, 396],
      dtype='int64', length=392)
```

`set_index()`法を使えば、`Auto['name']`の内容を使って行に名前をつけ直すことができる。

In [92]:

```
Auto_re = Auto.set_index('name')
Auto_re
```

Out[92]: （訳注：一部分を省略）

mpg
cylinders
displacement
horsepower
weight
acceleration
year
origin
name
chevrolet chevelle malibu
18.0
8
307.0
130.0
3504.0
12.0
70
1
...
...
.....
...
chevy s-10
31.0
4
119.0
82.0
2720.0
19.4
82
1
392 rows × 8 columns

In [93]:

```
Auto_re.columns
```

Out[93]:

```
Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'year', 'origin'],
      dtype='object')
```

ここで列 `name` がもうないことを確認できる。インデックスが `name` に設定されたので、`Auto` のメソッドである `loc[]`を用いれば、データフレームの行に `name` でアクセスできる：

In [94]:

```
rows = ['amc rebel sst', 'ford torino']
Auto_re.loc[rows]
```

Out[94]:
mpg
cylinders
displacement
horsepower
weight
acceleration
year
origin
name
amc rebel sst
16.0
8
304.0
150.0
3433.0
12.0
70
1
ford torino
17.0
8
302.0
140.0
3449.0
10.5
70
1

インデックス名の利用とは別の方法としては、`Auto` の 4,5 番目の行を `iloc[]`メソッドで取り出すこともできる：

In [95]:

```
Auto_re.iloc[[3,4]]
```

Out[95]:
mpg
cylinders
displacement
horsepower
weight
acceleration
year
origin
name
amc rebel sst
16.0
8
304.0
150.0
3433.0
12.0
70
1
ford torino
17.0
8
302.0
140.0
3449.0
10.5
70
1

この方法で、`Auto_re` の 1,3,4 列目を取り出すこともできる：

In [96]:

```
Auto_re.iloc[:,[0,2,3]]
```

Out[96]: （訳注：一部分を省略）
mpg
displacement
horsepower
name
chevrolet chevelle malibu
18.0
307.0
130.0
buick skylark 320
15.0
350.0
165.0
...
...
...
...
dodge rampage
32.0
135.0
84.0
ford ranger
28.0
120.0
79.0
chevy s-10
31.0
119.0
82.0

392 rows × 3 columns

`iloc[]`を一度呼び出すだけで、4,5 行目の 1,3,4 列目をとりだすことができる。

In [97]:

```
Auto_re.iloc[[3,4],[0,2,3]]
```

Out[97]:

mpg

displacement

horsepower

name

amc rebel sst

16.0

304.0

150.0

ford torino

17.0

302.0

140.0

インデックスの見出しはユニークである必要はない：例えばここでのデータフレームの中には `ford galaxie 500` という名前の車が複数ある。

In [98]:

```
Auto_re.loc['ford galaxie 500', ['mpg', 'origin']]
```

Out[98]:
mpg
origin
name
ford galaxie 500
15.0
1
ford galaxie 500
14.0
1
ford galaxie 500
14.0
1

行と列（続き）More on Selecting Rows and Columns

`yuear` が 80 より大きい -- つまり、1980 より後に作られた -- 車の部分集合に関する、`weight` と `origin` からなるデータフレームを作りたいとしよう。 このためには、まず行を指定するブーリアンの配列を作る。 `loc[]`法では、文字列と同じくブーリアンの見出しも利用できる：

In [99]:

```
idx_80 = Auto_re['year'] > 80
Auto_re.loc[idx_80, ['weight', 'origin']]
```

Out[99]: （訳注：一部分を省略）
weight
origin
name
plymouth reliant
2490.0
1
buick skylark
2635.0
1
dodge aries wagon (sw)
2620.0
1
chevrolet citation
2725.0
1
・
・
・
dodge rampage
2295.0
1
ford ranger
2625.0
1
chevy s-10
2720.0
1

より簡潔に書くには、無名関数 `lambda` を使えばよい：

In [100]:

```
Auto_re.loc[lambda df: df['year'] > 80, ['weight', 'origin']]
```

Out[100]: （訳注：一部分を省略）
weight
origin
name
plymouth reliant
2490.0

1
buick skylark
2635.0
1
dodge aries wagon (sw)
2620.0
1
・
・
・

chevrolet citation
vw pickup
2130.0
2
dodge rampage
2295.0
1
ford ranger
2625.0
1
chevy s-10
2720.0
1

`lambda`を呼び出すと、１つの引数（ここでは`df`）を持つ関数（ここでは`df['year']>80`を返す）が生成される。これはデータフレーム`Auto_re`の`loc[]`メソッド内で生成されているので、そのデータフレームが関数に与えられる引数となる。`lambda`の他の利用例として、1980年よりに作られ、30マイル/ガロン超を達成しているすべての車を取り出してみよう：

In [101]:

```
Auto_re.loc[lambda df: (df['year'] > 80) & (df['mpg'] > 30),
            ['weight', 'origin']
           ]
```

Out[101]: **(訳注：一部分を省略)**
weight
origin
name
toyota starlet
1755.0
3
・
・
・
vw pickup
2130.0
2
dodge rampage
2295.0

1
chevy s-10
2720.0
1

シンボル&は要素ごとの *and*（かつ）操作を計算することを意味する。 別の例として、`displacement`が 300 より小さいすべての `Ford` と `Datsun` の車を抽出したいとしよう。 このためには、各見出し `name` が文字列 `ford` か `dastun` を含んでいるかをデータフレームの `index` 属性が持つ `str.contains()`メソッドにより確認すればよい：

In [102]:

```
Auto_re.loc[lambda df: (df['displacement'] < 300)
                       & (df.index.str.contains('ford')
                       | df.index.str.contains('datsun')),
             ['weight', 'origin']
            ]
```

Out[102]: （訳注：一部分を省略）
weight
origin
name
ford maverick
2587.0
1
datsun pl510
2130.0
3
datsun pl510
2130.0
3
・
・
・
datsun 310 gx
1995.0
3
ford granada l
2835.0
1
ford mustang gl
2790.0
1
ford ranger
2625.0
1

ここでシンボル|は要素ごとの *or*（または）操作を計算することを意味する。

まとめると、データフレームの行と列を指定するための強力な操作を利用できる。 整数ベースの検索方法としては、`iloc[]`法を利用すればよい。 文字列やブーリアンによる選択には、`loc[]`メソッドを使う。 行をフィルタリングするよ

うな関数的な検索には、`loc[]`メソッドの行引数に関数（`lambda`など）を入れればよい。

For Loops について

`for`ループは多くの言語で一般的な、その中の値を変化させながらコードの塊を繰り返し評価するツールである。 たとえば、リストの要素をループして、それらの和を計算してみよう。

In [103]:

```
total = 0
for value in [3,2,19]:
    total += value
print('Total is: {0}'.format(total))
```

Out [103]:

```
Total is: 24
```

for 宣言した行の下にあるインデント

訳注 18：一定幅のスペースやタブによって段落全体を字下げすることを意味する。

されたコードは、`for`に指定された配列の中にある各値に対して実行される。 そのセルが終わるか、オリジナルの`for`宣言と同じレベルにインデントされたところが、ループの対象範囲になる。 上記において、合計を表示する最後の行は、ループが終了した後に 1 回だけ実行されたことが確認できる。 インデントを使いすることで、ループをネスト

訳注 19：ループの中にループを入れることを意味する。

することができる。

In [104]:

```
total = 0
for value in [2,3,19]:
    for weight in [3, 2, 1]:
```

```
        total += value * weight
print('Total is: {0}'.format(total))
```

Out [104]:

```
Total is: 144
```

ここでは、value と weight の各組合せに対して和をとった。 ここでは Python における*インクリメント*記法も利用した：a += b 表記は、a = a + b と同じ意味となる。 表記の利便性に加えて、明示的に生成する必要がない中間の値 a+b

訳注 20：総和の計算には部分和の計算は不要だということになる。

に関する負荷の大きい計算タスクのための時間を、この記法によって削ることができる。

おそらく、(value, weight)ペアに対する和はもっと一般的なタスクになると思われる。 たとえば、それぞれに確率 0.2, 0.3, 0.5 が割り当てられている確率変数 2, 3, 19 の平均値を計算するのには、加重和を計算することになる。 このようなタスクは、よく zip()関数を用いることで達成できる。この関数によって、複数のタプルを要素として持つ配列に対してループを行えばよい。

In [105]:

```
total = 0
for value, weight in zip([2,3,19],
                         [0.2,0.3,0.5]):
    total += weight * value
print('Weighted average is: {0}'.format(total))
```

Out [105]:

```
Weighted average is: 10.8
```

文字列(String Formatting)

上記のコードでは、合計を示す文字列も表示した。 しかし total は整数であって文字列ではない。 文字列の中に何らかの値を挿入するのは一般的なタスクであり、Python の強力な文字列フォーマット法の利用によってシンプルに実行で

きる。 多くのデータクリーニング作業は文字列の操作とプログラミングによる生成を含んでいる。

例として、データフレームの列に対してループを行い、各列について欠測値の割合を表示してみよう。 20%の要素が欠測（例えば`np.nan`がセットされている）である列によってできたデータフレーム`D`を生成してみよう。`rng.standard_normal()`を使って、平均 0 分散 1 の正規分布から`D`内の値を生成する。 そして一部を`rng.choice()`を使って上書きしてみる。

In [106]:

```
rng = np.random.default_rng(1)
A = rng.standard_normal((127, 5))
M = rng.choice([0, np.nan], p=[0.8,0.2], size=A.shape)
A += M
D = pd.DataFrame(A, columns=['food',
                             'bar',
                             'pickle',
                             'snack',
                             'popcorn'])
D[:3]
```

Out[106]:

	Food	bar	Pickle	Snack	popcorn
0	0.345584	0.821618	0.330437	-1.303157	NaN
1	NaN	-0.536953	0.581118	0.364572	0.294132
2	NaN	0.546713	NaN	-0.162910	-0.482119

In [107]:

```
for col in D.columns:
    template = 'Column "{0}" has {1:.2%} missing values'
    print(template.format(col,
          np.isnan(D[col]).mean()))
```

Out [107]:

```
Column "food" has 16.54% missing values
Column "bar" has 25.98% missing values
Column "pickle" has 29.13% missing values
```

```
Column "snack" has 21.26% missing values
Column "popcorn" has 22.83% missing values
```

`template.format()`法では２つの引数`{0}`と`{1:.2%}`が与えられることが期待されていて、後者はフォーマットに関する情報を含んでいることが分かる。 具体的には、２つ目の引数は 10 進数で２桁のパーセントで表示される。

docs.python.org/3/library/string.html には多くの有用でより複雑な例が掲載されている。

グラフと数値の追加事項

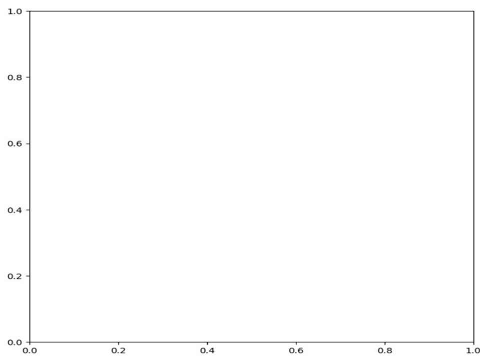
量的変数の表示には、`ax.plot()`関数や`ax.scatter()`関数を使用できる。 しかし、単純に変数名を打ち込むだけではエラーメッセージが表示されてしまう。 これは、対象となる変数を見つけるには`Auto`データセットを参照しないといけないということを`Python`が分からないために起きる。

In [108]:

```
fig, ax = subplots(figsize=(8, 8))
ax.plot(horsepower, mpg, 'o');
```

Out [108]:

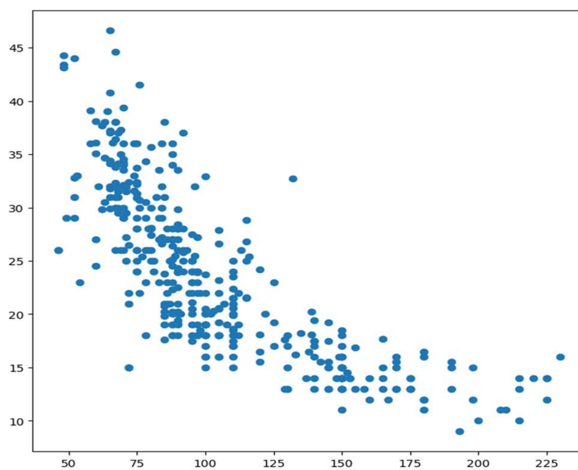
```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-108-1a2f9afa5cfe> in <cell line: 2>()
      1 fig, ax = subplots(figsize=(8, 8))
----> 2 ax.plot(horsepower, mpg, 'o');
NameError: name 'horsepower' is not defined
```



列に直接アクセスすることで、この問題を取り扱える：

In [109]:

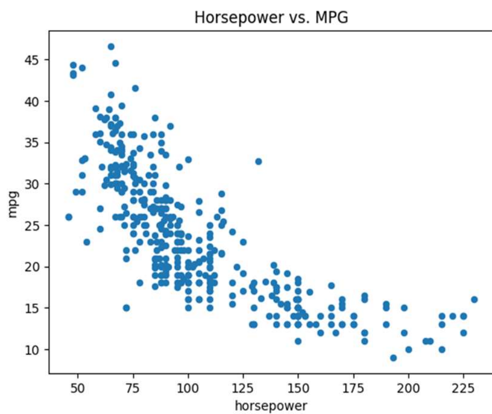
```
fig, ax = subplots(figsize=(8, 8))
ax.plot(Auto['horsepower'], Auto['mpg'], 'o');
```



別の方法としては、`plot()`法を`Auto.plot()`により呼び出すことで使用できる。 この方法を使うと、変数に名称でアクセスすることができる。 データフレームのグラフ描画メソッドは似たような axes オブジェクトを返す。 この操作を使って前にやったようにグラフを更新することもできる。

In [110]:

```
ax = Auto.plot.scatter('horsepower', 'mpg')
ax.set_title('Horsepower vs. MPG');
```



与えられた axes を含む figure を保存したい場合には、`figure` 属性にアクセスすることで関連する figure を見つけることができる：

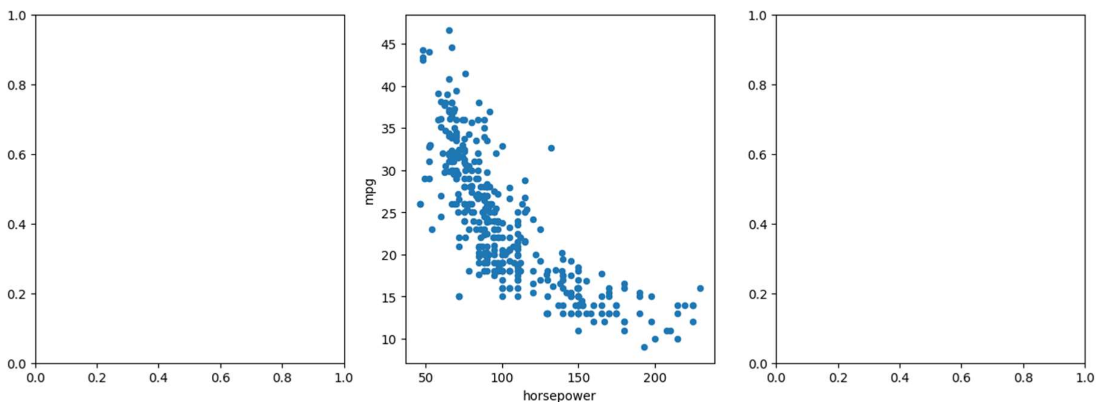
In [111]:

```
fig = ax.figure
fig.savefig('horsepower_mpg.png');
```

さらに特定の axes オブジェクトに対してプロットするデータフレームを指定することができる。 この場合、対応する `plot()` メソッドは引数として渡された axes を編集して返す。 1 次元のグラフのグリッドを要求した場合には、`axes` オブジェクトは同様の 1 次元となることに留意する必要がある。 ここで figure の中にある行の中央のプロットに散布図を配置してみる。

In [112]:

```
fig, axes = subplots(ncols=3, figsize=(15, 5))
Auto.plot.scatter('horsepower', 'mpg', ax=axes[1]);
```



データフレームの列に属性としてアクセスできることにも注意しよう：試しに `Auto.horsepower` と入力してみてみよう。

訳注 21：日本語の列名を設定した場合の挙動は次のようになる。

In [113]:

```
df_jp = pd.DataFrame({"en": [1,2,3], "日本語":[3,4,5]})
print(df_jp.en)
print("===================")
print(df_jp.日本語)
```

Out [113]:

```
0    1
1    2
2    3
Name: en, dtype: int64
===================
0    3
1    4
2    5
Name: 日本語, dtype: int64
```

ここで変数 `cylinders` について考えよう。 `Auto.cylinders.dtype` と入力すると、これが量的変数として扱われていることが分かる。 しかし、この変数に対してはとりうる値が少数しかないので、質的な変数として取り扱いたい場合がある。以下では、`cylinders` 列を `Auto.cylinders` のカテゴリカルなバージョンで置き換えてみよう。 関数 `pd.Serise()`は、`pandas` がよく時系列の応用に使われているという事実からその名前がつけられている。

In [114]:

```
Auto.cylinders = pd.Series(Auto.cylinders, dtype='category')
Auto.cylinders.dtype
```

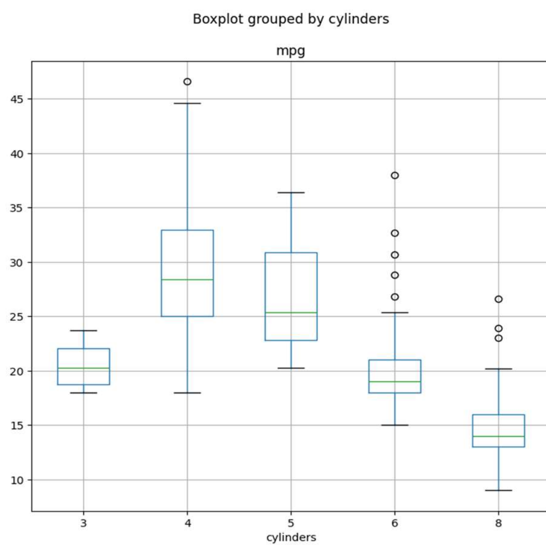
Out[114]:

```
CategoricalDtype(categories=[3, 4, 5, 6, 8], ordered=False, categories_dtype
=int64)
```

これで`cylinders`は質的変数になったので、`boxplot()`メソッドで表示できる。

In [115]:

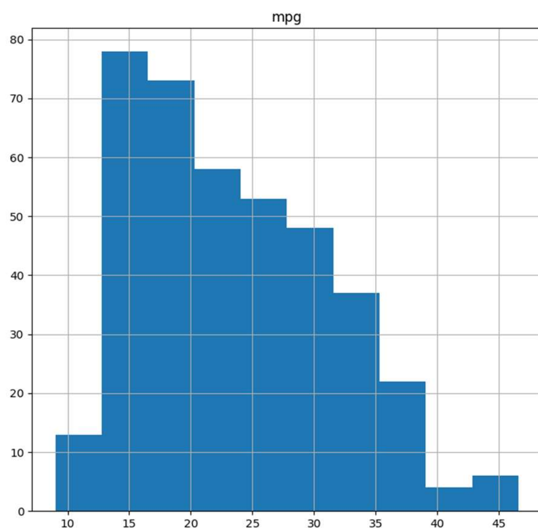
```
fig, ax = subplots(figsize=(8, 8))
Auto.boxplot('mpg', by='cylinders', ax=ax);
```



`hist()`法はヒストグラムを表示するために利用する。

In [116]:

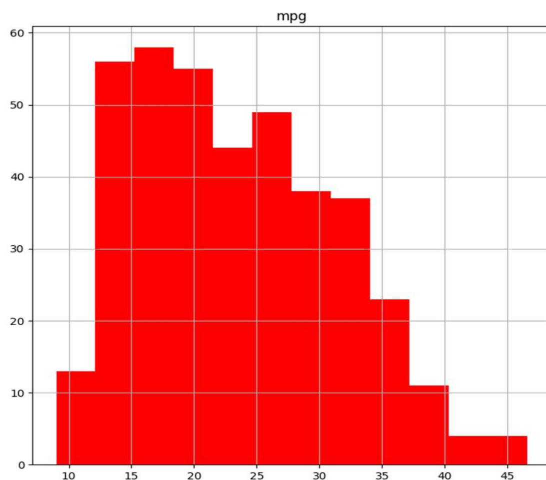
```
fig, ax = subplots(figsize=(8, 8))
Auto.hist('mpg', ax=ax);
```



棒の色やビン（訳注：ヒストグラムの棒のこと）の数は変更可能です。

In [117]:

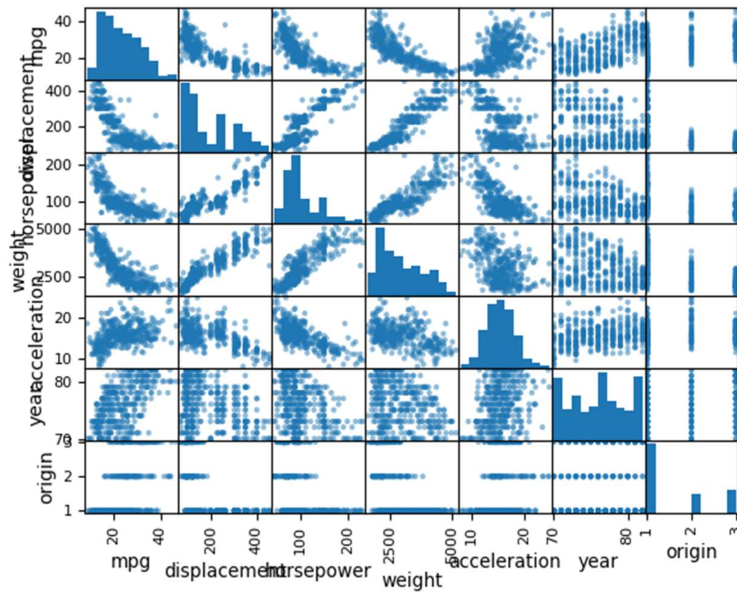
```
fig, ax = subplots(figsize=(8, 8))
Auto.hist('mpg', color='red', bins=12, ax=ax);
```



他のオプションに関しては、`Auto.hist?`により確認してみよう。
`pd.plotting.scatter_matrix()`関数を使ってデータフレームの列に対する任意の組み合わせの関係を可視化してくれる*散布図行列*を作成することができる。

In [118]:

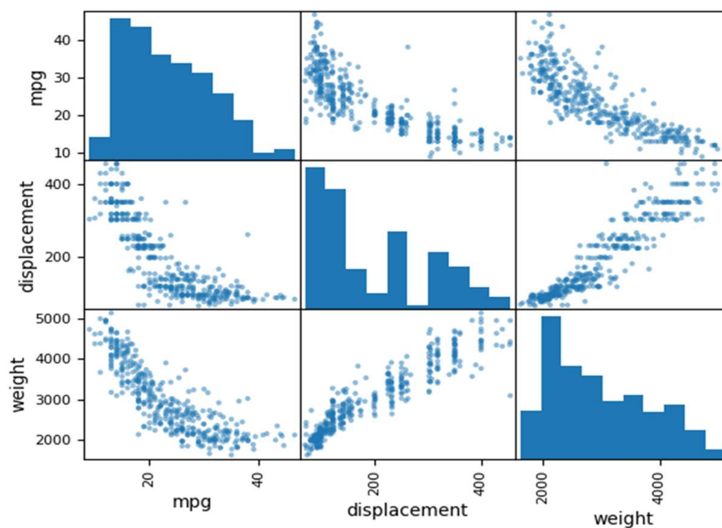
```
pd.plotting.scatter_matrix(Auto);
```



変数の部分集合に対して散布図を作成することもできる。

In [119]:

```
pd.plotting.scatter_matrix(Auto[['mpg',
                                 'displacement',
                                 'weight']]);
```



`describe()`法によりデータフレームの各列に対する数値的な要約

訳注 **22**：要約統計量を意味する。

を計算してくれる。。

In [120]:

```
Auto[['mpg', 'weight']].describe()
```

Out[120]:

	mpg	Weight
Count	392.000000	392.000000
Mean	23.445918	2977.584184
Std	7.805007	849.402560
Min	9.000000	1613.000000
25%	17.000000	2225.250000
50%	22.750000	2803.500000
75%	29.000000	3614.750000
Max	46.600000	5140.000000

単一の列に対しても同様に要約できる。

In [121]:

```
Auto['cylinders'].describe()
Auto['mpg'].describe()
```

Out[121]:

	mpg
Count	392.000000
Mean	23.445918
Std	7.805007
Min	9.000000
25%	17.000000
50%	22.750000
75%	29.000000
Max	46.600000

dtype: float64

`Jupyter`を終了するには、`File(訳注：Notebookの上の段) / Close and Halt.`とすればよい。

ISLP 第 3 章 線形回帰

パッケージのインポート(Importing packages)

まず標準的なライブラリー（パッケージ）をインポートして利用できるようにする。

In [1]:

```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
```

追加のインポート(New imports)

本セクションでは、新しい関数とライブラリを紹介する。コードやノートブックの上部に用いるライブラリのインポートを配置すると、最初の数行をスキャンするだけでどのライブラリが使用されているかが分かるため、コードが読みやすくなる。

In [2]:

```
import statsmodels.api as sm
```

様々な機能を持ったライブラリが存在し、取り組む課題によっては不要な機能も存在することも考えられる。その場合には、`import`によりライブラリの中で特定の機能だけをインポートすることができる。したがって、必要な機能だけを使うことができ、より簡潔なコードにすることができる。

ここでは、`statsmodels` パッケージからいくつか特定のオブジェクトをインポートする。

In [3]:

```
from statsmodels.stats.outliers_influence \
     import variance_inflation_factor as VIF
from statsmodels.stats.anova import anova_lm
```

上記のインポート ステートメントの 1 つは非常に長い行であるため、読みやすくするために改行 `\` を挿入するとよい。

この演習用に記述された `ISLP` パッケージ内のいくつかの関数も利用する。`as` の後に読み込んだ機能に呼び名をつけることができる。このことから、機能を利用する度に長い機能名を記述する必要がなくなるため、コード記述打鍵でなくコード自体も簡潔にすることができる。

In [4]:

```
from ISLP import load_data
from ISLP.models import (ModelSpec as MS,
                         summarize,
                         poly)
```

オブジェクトと名前(Objects and Namespaces)

関数 `dir()` は、指定したオブジェクトが持つ対象の変数や関数、オブジェクトなど属性をリストで返してくれる。

In [5]:

```
dir()
```

Out[5]:（訳注：一部分を省略）

```
['In',
 'MS',
 'Out',
```

```
 'VIF',
 '_',
 '__',
 '___',
 .

 .

 .

'subplots',
 'summarize']
```

`print()`などの組み込み関数への参照を含み`_builtins__`などの特定のオブジェクトも表示されている。

Python が利用できるオブジェクトには、`dir()`でアクセスできる独自の名前空間の概念が存在する。これには、オブジェクトの属性とそれに関連付けられたメソッドの両方が含まれる。たとえば、配列のリストには `sum` が表示されている。

In [6]:

```
A = np.array([3,5,11])
dir(A)
```

Out[6]: （訳注：一部分を省略）

```
['T',
 '__abs__',
 '__add__',
 '__and__',
 '__array__',
 '__array_finalize__',
 '__array_function__',
 .

 .

 .

 . 'tofile',
 'tolist',
 'tostring',
 'trace',
```

```
 'transpose',
 'var',
 'view']
```

この出力によると、オブジェクト`A.sum`が存在することを示している。このことは`A.sum?`と入力すると分かる（`?`をつけることで取扱説明書のような詳細を見ることができる）ように、配列`A`の合計を計算するために使用できる方法である。

In [7]:

```
A.sum()
```

Out[7]:

```
19
```

単回帰(Simple Linear Regression)

このセクションでは、`ISLP.models`の`ModelSpec()`変換を使用して、モデル・マトリックス（計画行列やデザイン行列とも呼ばれます）を構築する。

`ISLP`パッケージに含まれる`Boston`：住宅データセットを使用するが、`Boston`データセットには、ボストン周辺の506地区の`medv`（住宅価格の中央値）が記録されている。その他には、`rmvar`（住宅あたりの平均部屋数）、`age`（1940年以前に建てられた所有者居住ユニットの割合）、`lstat`（社会経済的地位の低い世帯の割合）などの13個の変数が記録されており、これらを使用して`medv`を予測する回帰モデルを構築していこう。回帰モデルの構築には、`Python`パッケージである`statsmodels`を使用していく。

`ISLP`パッケージには、シンプルな読み込み関数`load_data()`が含まれている。

```
# 訳注：ISLPがインストールされていなければインストールする必要がある。

%%capture
!pip install ISLP
```

In [8]:

```
Boston = load_data("Boston")
Boston.columns
```

Out[8]:

```
Index(['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax',
       'ptratio', 'lstat', 'medv'],
      dtype='object')
```

これらのデータについてさらに詳しく知りたければ、`Boston?`と入力すればよい。
まず、`sm.OLS()`関数を使用して、単純な線形回帰モデルを当てはめていこう。応答変数は `medv` であり、`lstat` のみを用いた単回帰になる。

In [9]:

```
X = pd.DataFrame({'intercept': np.ones(Boston.shape[0]),
                  'lstat': Boston['lstat']})
X[:4]
```

Out[9]:

	Intercept	lstat
0	1.0	4.98
1	1.0	9.14
2	1.0	4.03
3	1.0	2.94

応答変数を抽出し、モデルを構築する。

In [10]:

```
y = Boston['medv']
model = sm.OLS(y, X)
results = model.fit()
```

`sm.OLS()`はモデルを構築するのみであり、その後の `model.fit()`にて構築したモデルを用いてフィットしていく。

`ISLP` に含まれる `summarize()`関数は、パラメータ推定値、その標準誤差、t 統計量、p 値の簡単な表を生成してくれる。

この関数は、`fit` メソッドによって返されるオブジェクト `results` などの単一の引数を取り、それらの要約を返す。

In [11]:

```
summarize(results)
```

Out[11]:

	coef	std err	T	P>|t|
intercept	34.5538	0.563	61.415	0.0
Lstat	-0.9500	0.039	-24.528	0.0

適合モデルを操作するための他の方法を説明する前に、モデル行列 `X` を構築するためのより有用で一般的なフレームワークの概要を説明します。

変換の利用(Using Transformations: Fit and Transform)

上記のモデルには説明変数が 1 つだけあり、`X` の構築は簡単である。しかし、実際には、配列またはデータフレームを読み込み、複数の説明変数を使用してモデルを構築することがよくある。モデルを適合させる前に変数に変換を施したり、変数間の相互作用を指定したり、特定の変数を変数のセット（多項式など）に拡張したりする場合などがある。`sklearn` パッケージには、変換ができる機能があるが、変換は、いくつかのパラメータを引数として作成されるオブジェクトであり、`fit()`と `transform()`という２つの主な方法がある。

`ISLP` ライブラリに含まれる `ModelSpec()`関数を通じて、モデルを指定し、モデル・マトリックスを構築するための一般的なアプローチを確認していこう。`ModelSpec()`（本コードではインポートする際に、`as` を用いて `MS()`と表記できるようにしている）は、変換オブジェクトを作成し、次に `transform()`と `fit()`の 2 つの方法を使用して対応するモデル・マトリックスを構築していく。

最初に、`Boston` データフレームに含まれる説明変数 `lstat` を利用する、単純な回帰モデルについて見ていこう。変換は `design = MS(['lstat'])`によって作成する。次に、`fit()`メソッドは元の配列を受け取り、変換オブジェクトで指定されているように、その配列に対していくつかの初期計算を実行することができる。たとえば、中央揃えとスケーリングの平均と標準偏差を計算することが可能である。

`transform()`法は、適合された変換をデータの配列に適用し、モデル・マトリックスを生成する。

In [12]:

```
design = MS(['lstat'])
design = design.fit(Boston)
X = design.transform(Boston)
X[:4]
```

Out[12]:

	Intercept	lstat
0	1.0	4.98
1	1.0	9.14
2	1.0	4.03
3	1.0	2.94

この単純なケースでは、`fit()`メソッドはほとんど意味をなさない。`design`で指定された変数 `lstat` が `Boston` に存在するかどうかをチェックするだけである。次に、`transform()`は`intercept`と変数`lstat`の２つの列を持つモデル・マトリックスを構築する。

これらの２つの操作は、`fit_transform()`法と組み合わせることもできる。

In [13]:

```
design = MS(['lstat'])
X = design.fit_transform(Boston)
X[:4]
```

Out[13]:

	Intercept	lstat
0	1.0	4.98
1	1.0	9.14
2	1.0	4.03
3	1.0	2.94

前のコードチャンクと同様に、２つのステップが別々に実行された場合、`fit()`操作の結果として`design`オブジェクトが変更される。このパイプラインの効果は、相互作用と変換を含むより複雑なモデルを適合させるときに明らかになる。

フィットした回帰モデルに戻ろう。

オブジェクト`results`には、推論に使用できるいくつかの方法が存在する。適合の要点を示す関数 `summarize()`はすでに紹介した。モデルの網羅的な要約を得るには、`summary()`法を利用することができる。

In [16]:

```
results.summary()
```

Out[16]:

OLS Regression Results

Dep. Variable:	medv	R-squared:	0.544			
Model:	OLS	Adj. R-squared:	0.543			
Method:	Least Squares	F-statistic:	601.6			
Date:	Tue, 18 Feb 2025	Prob (F-statistic):	5.08e-88			
Time:	09:07:32	Log-Likelihood:	-1641.5			
No. Observations:	506	AIC:	3287.			
Df Residuals:	504	BIC:	3295.			
Df Model:	1					
Covariance Type:	nonrobust					
	coef	std err	t	P>|t|	[0.025	0.975]
Intercept	34.5538	0.563	61.415	0.000	33.448	35.659
Lstat	-0.9500	0.039	-24.528	0.000	-1.026	-0.874
Omnibus:	137.043	Durbin-Watson:	0.892			
Prob(Omnibus):	0.000	Jarque-Bera (JB):	291.373			
Skew:	1.453	Prob(JB):	5.36e-64			
Kurtosis:	5.319	Cond. No.	29.7			

Notes:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.

構築したモデルの係数は、`results` の `params` の属性として得ることもできる。

In [14]:

```
results.params
```

Out[14]:

```
intercept    34.553841
lstat        -0.950049
dtype: float64
```

`get_prediction()`法を使用すると、予測値を計算し、`lstat` の特定の値に対する `medv` の予測の信頼区間と予測区間を作成できる。

まず、予測を行う変数の値、今回であれば `lstat` のみを含む新しいデータフレームを作成する。次に、`design` の `transform()`メソッドを使用して、対応するモデル・マトリックスを作成する。

In [15]:

```
new_df = pd.DataFrame({'lstat':[5, 10, 15]})
newX = design.transform(new_df)
newX
```

Out[15]:

	Intercept	lstat
0	1.0	5
1	1.0	10
2	1.0	15

次に、`newX` での予測を計算し、`predicted_mean` 属性を抽出して表示しよう。

In [16]:

```
new_predictions = results.get_prediction(newX);
new_predictions.predicted_mean
```

Out[16]:

```
array([29.80359411, 25.05334734, 20.30310057])
```

予測値の信頼区間も作成できる。

In [17]:

```
new_predictions.conf_int(alpha=0.05)
```

Out[17]:

```
array([[29.00741194, 30.59977628],
       [24.47413202, 25.63256267],
       [19.73158815, 20.87461299]])
```

予測区間は `obs=True` を設定することで計算される。

In [18]:

```
new_predictions.conf_int(obs=True, alpha=0.05)
```

Out[18]:

```
array([[17.56567478, 42.04151344],
       [12.82762635, 37.27906833],
       [ 8.0777421 , 32.52845905]])
```

たとえば、`lstat` 値が 10 の場合の 95%信頼区間は(24.47, 25.63)、95%予測区間は(12.82, 37.28)となる。これは予想どおり、信頼区間と予測区間は同じポイント（`lstat` が 10 の場合の `medv` の予測値 25.05）を中心としているが、後者の方が大幅に広くなっている。

次に、`DataFrame.plot.scatter()`、`plot.scatter()`、`plot.slashslashscatter()`を使用して `medv` と `lstat` をプロットし、結果のプロットに回帰線を追加しよう。

関数の定義(Defining Functions)

`ISLP` パッケージには既存のプロットに線を追加する関数がありますが、この機会に最初の関数を定義してそれを実行していこう。

In [19]:

```
def abline(ax, b, m):
    "Add a line with slope m and intercept b to ax"
    xlim = ax.get_xlim()
    ylim = [m * xlim[0] + b, m * xlim[1] + b]
    ax.plot(xlim, ylim)
```

関数を定義するための構文を確認しよう。`def funcname(...)`とすることで `funcname` 関数を定義することができる。`abline()`関数には、引数 `ax, b, m` があり、`ax` は既存のプロットの軸オブジェクト、`b` は切片、`m` は目的の線の傾きに対応する。他のプロットオプションは、次のように追加のオプション引数を含めることで、`ax.plot` に渡すことができる。

In [20]:

```
def abline(ax, b, m, *args, **kwargs):
    "Add a line with slope m and intercept b to ax"
    xlim = ax.get_xlim()
    ylim = [m * xlim[0] + b, m * xlim[1] + b]
    ax.plot(xlim, ylim, *args, **kwargs)
```

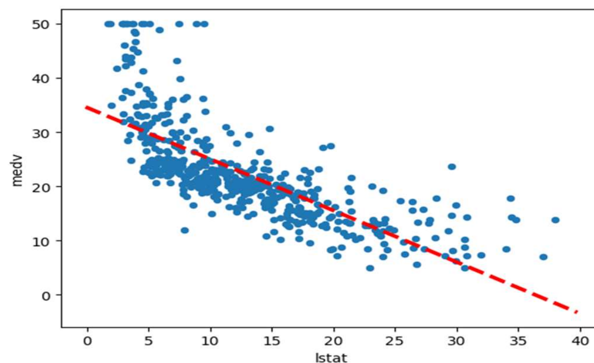
`*args` を追加すると、`abline` に任意の数の名前なし引数を指定でき、`*kwargs` を追加すると、`abline` に任意の数の名前付き引数（`linewidth=3` など）を指定することができる。

この関数では、上記の `ax.plot` にこれらの引数をそのまま渡すが、関数についてさらに詳しく知りたい場合は、docs.python.org/tutorial の関数定義に関するセクションを参照するとよい。

新しい関数を使用して、この回帰直線を `medv` と `lstat` のプロットに追加してみよう。

In [21]:

```
ax = Boston.plot.scatter('lstat', 'medv')
abline(ax,
       results.params[0],
       results.params[1],
       'r--',
       linewidth=3)
```



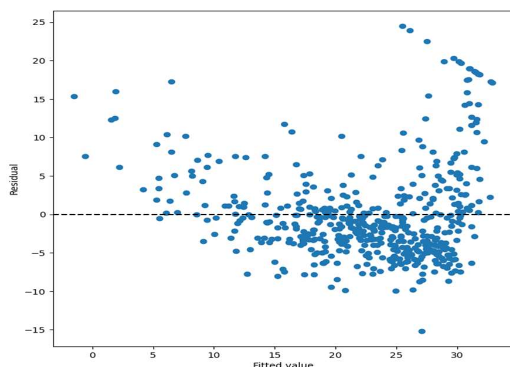
関数の引数をもとに考えると、`ax.plot()`の最後の呼び出し部分は`ax.plot(xlim, ylim, 'r--', linewidth=3)`となっている。ここでの`'r--'`は、引数を使用して赤い破線を生成し、幅３にする引数を追加している。また、`lstat` と `medv` の関係には非線形性があるという証拠があるが、この問題については、この演習部分の後半で説明していく。

前述のように、プロットに線を追加する関数は既に存在する。 --- `ax.axline()` --- が、その関数にあたるが、このような関数の書き方を知っていれば、より表現力豊かな表示を作成することが可能となる。

次に、適合値と適合の残差について可視化をしていこう。使用するメソッドは、`results` オブジェクトの属性として見つけることができる。回帰モデルを説明するさまざまな影響度は、`get_influence()`法で計算される。`subplots()`から最初の値として返される `fig` コンポーネントは使用しないので、２番目に返される値を以下の `ax` で取得するだけとなる。

In [22]:

```
ax = subplots(figsize=(8,8))[1]
ax.scatter(results.fittedvalues, results.resid)
ax.set_xlabel('Fitted value')
ax.set_ylabel('Residual')
ax.axhline(0, c='k', ls='--');
```



`ax.axhline()`法を使用して、参照用に `Residual` が0となる位置に水平線を追加する。これは、黒（`c='k'`）で破線の線種（`ls='--'`）であることを示している。

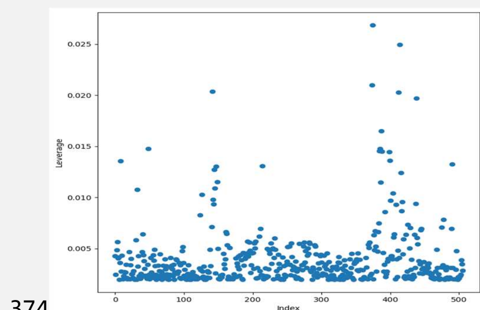
残差プロットに基づくと、非線形性を確認することができる。`get_influence()`メソッドによって返される値の `hat_matrix_diag` 属性を使用して、任意の数の説明変数のてこ比を計算できる。

In [23]:

```
infl = results.get_influence()
ax = subplots(figsize=(8,8))[1]
ax.scatter(np.arange(X.shape[0]), infl.hat_matrix_diag)
ax.set_xlabel('Index')
ax.set_ylabel('Leverage')
np.argmax(infl.hat_matrix_diag)
```

Out[23]:

374



`np.argmax()`関数は、配列の最大要素のインデックスを識別してくれる。これは、オプションで配列の軸上で計算されるが、この場合、配列全体を最大化して、どの観測値が影響力を持っているのかを判断することが可能となる。

重回帰(Multiple Linear Regression)

最小二乗法を使用して重線形回帰モデルをフィットするには、`ModelSpec()`変換を再度使用して、必要なモデルマトリックスと応答を構築すればよい。`ModelSpec()`への引数は非常に一般的なものにすることができるが、この場合は列名のリストで十分である。今回は、２つの変数`lstat`と`age`を使用したフィットを検討してみよう。

In [24]:

```
X = MS(['lstat', 'age']).fit_transform(Boston)
model1 = sm.OLS(y, X)
results1 = model1.fit()
summarize(results1)
```

Out[24]:

	coef	std err	t	P>|t|
intercept	33.2228	0.731	45.458	0.000
Lstat	-1.0321	0.048	-21.416	0.000
Age	0.0345	0.012	2.826	0.005

最初の行では`X`の構成をまとめている。

`Boston`データセットには１２個の変数が含まれているため、すべての変数を使用して回帰を実行するには、これらすべてを入力するのは面倒である。

その代わりに、`columns.drop()`や`columns.slashslashdrop()`などの省略形を使用することができる。

In [25]:

```
terms = Boston.columns.drop('medv')
terms
```

Out[25]:

```
Index(['crim', 'zn', 'indus', 'chas', 'nox', 'rm', 'age', 'dis', 'rad', 'tax',
       'ptratio', 'lstat'],
      dtype='object')
```

これで、同一のモデル・マトリックスのビルダーを使用して、`terms` 内のすべての変数を使用してモデルをフィットできるようになった。

In [26]:

```
X = MS(terms).fit_transform(Boston)
model = sm.OLS(y, X)
results = model.fit()
summarize(results)
```

Out[26]:

	Coef	std err	t	P>|t|
intercept	41.6173	4.936	8.431	0.000
Crim	-0.1214	0.033	-3.678	0.000
Zn	0.0470	0.014	3.384	0.001
indus	0.0135	0.062	0.217	0.829
Chas	2.8400	0.870	3.264	0.001
Nox	-18.7580	3.851	-4.870	0.000
Rm	3.6581	0.420	8.705	0.000
Age	0.0036	0.013	0.271	0.787
Dis	-1.4908	0.202	-7.394	0.000
Rad	0.2894	0.067	4.325	0.000
Tax	-0.0127	0.004	-3.337	0.001
ptratio	-0.9375	0.132	-7.091	0.000
Lstat	-0.5520	0.051	-10.897	0.000

1 つの変数を除くすべての変数を使用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、上記の回帰出力では、`age` のp値が高くなっている。そこで、この予測変数を除外して回帰を実行するとよい可能性があるため、`age` を除くすべての予測変数を使用した回帰を実施し確認していこう。

In [27]:

```
minus_age = Boston.columns.drop(['medv', 'age'])
Xma = MS(minus_age).fit_transform(Boston)
model1 = sm.OLS(y, Xma)
summarize(model1.fit())
```

Out[27]:

	coef	std err	t	P>|t|
intercept	41.5251	4.920	8.441	0.000
Crim	-0.1214	0.033	-3.683	0.000
Zn	0.0465	0.014	3.379	0.001
indus	0.0135	0.062	0.217	0.829
Chas	2.8528	0.868	3.287	0.001
Nox	-18.4851	3.714	-4.978	0.000
Rm	3.6811	0.411	8.951	0.000
Dis	-1.5068	0.193	-7.825	0.000
Rad	0.2879	0.067	4.322	0.000
Tax	-0.0127	0.004	-3.333	0.001
ptratio	-0.9346	0.132	-7.099	0.000
Lstat	-0.5474	0.048	-11.483	0.000

多変量適合度(Multivariate Goodness of Fi)

`results` で出力可能なものについては後に指定した名前でアクセスすることができる（`dir(results)`で利用可能なものが表示される）。例えば、`results.rsquared` は R^2を、`np.sqrt(results.scale)`は RSE を得ることができる。

`VIF` は、回帰モデルのモデル・マトリックスにおける共線性の影響を評価するのに役立つ場合がある。

List Comprehension の利用

計算をする際、オブジェクトを変換したい場合が生じる。以下では、`X` の行列要素に対して VIF を計算し、インデックスが `X` の列と一致するデータフレームを作成してみる。リスト内包の概念により、多くの場合、簡易的に記述することが可能となる。

リスト内包は、`Python` オブジェクトのリストを作成する際、シンプルですが、とても有用な方法である。そこで例を見てみよう。関数 `variance_inflation_factor()` を使用して、モデルマ・トリックス `X` 内の各変数の VIF を計算してみる。

In [28]:

```
vals = [VIF(X, i)
        for i in range(1, X.shape[1])]
vif = pd.DataFrame({'vif':vals},
                   index=X.columns[1:])
vif
```

Out[28]:

	vif
Crim	1.767486
Zn	2.298459
Indus	3.987181
Chas	1.071168
Nox	4.369093
Rm	1.912532
Age	3.088232
Dis	3.954037
Rad	7.445301
Tax	9.002158
ptratio	1.797060
Lstat	2.870777

`VIF()`関数は、データフレームまたは配列と、可変列インデックスの２つの引数を取る。上記のコードでは、`X` のすべての列に対して `VIF()`を呼び出す。ただし重要性が低いため、列０（切片）は除外している。

上記のオブジェクト `vals` は、次の for ループを使用して構築できる。

In [29]:

```
vals = []
for i in range(1, X.values.shape[1]):
    vals.append(VIF(X.values, i))
```

リストを理解することでこのような反復操作をより簡単な方法で実行できる。

交互項(Interaction Terms)

`ModelSpec()`を利用すると線形モデルに相互作用項を簡単に含めることができる。タプル`("lstat","age")`を含めると、モデル・マトリックス・ビルダーに`lstat`と`age`の間の相互作用項を含めるように指示される。

In [30]:

```
X = MS(['lstat',
        'age',
        ('lstat', 'age')]).fit_transform(Boston)
model2 = sm.OLS(y, X)
summarize(model2.fit())
```

Out[30]:

	coef	std err	t	P>|t|
intercept	36.0885	1.470	24.553	0.000
Lstat	-1.3921	0.167	-8.313	0.000
Age	-0.0007	0.020	-0.036	0.971
lstat:age	0.0042	0.002	2.244	0.025

予測の非線形変換(Non-linear Transformations of the Predictors)

モデル・マトリックス・ビルダーには、列名と相互作用以外の用語を含めることもできる。たとえば、`ISLP`で提供される`poly()`関数は、最初の引数の多項式関数を表す列がモデル・ マトリックスに追加を指定する。

In [31]:

```
X = MS([poly('lstat', degree=2), 'age']).fit_transform(Boston)
model3 = sm.OLS(y, X)
results3 = model3.fit()
summarize(results3)
```

Out[31]:

	coef	std err	t	P>|t|
intercept	17.7151	0.781	22.681	0.0
poly(lstat, degree=2)[0]	-179.2279	6.733	-26.620	0.0
poly(lstat, degree=2)[1]	72.9908	5.482	13.315	0.0
Age	0.0703	0.011	6.471	0.0

２次項に関連付けられた実質的にゼロの p 値（つまり、上記の 3 行目）は、モデルが改善されていることを示している。

デフォルトでは、`poly()`は、安定した最小二乗計算用に設計された*直交多項式*の列を持つモデル・マトリックスに含める基底マトリックスを作成する。

(実際には、`poly()` は、モデル・マトリックスの構築作業を行う主力のスタンドアロン関数 `Poly()` のラッパーである。)

または、上記の `poly()` 呼び出しに引数 `raw=True` を含めた場合、基底マトリックスは単に `lstat` と `lstat**2` で構成される。これらの基底はどちらも 2 次多項式を表すため、この場合は適合値は変化せず、多項式係数だけが変化する。また、デフォルトでは、`poly()` によって作成された列には切片列は含まれない。これは、`MS()` によって自動的に追加されるためである。

`anova_lm()` 関数を使用して、二次近似が線形近似よりも優れている程度をさらに定量化できる。

In [32]:

```
anova_lm(results1, results3)
```

Out[32]:

	df_resid	ssr	df_diff	ss_diff	F	Pr(>F)
0	503.0	19168.128609	0.0	NaN	NaN	NaN
1	502.0	14165.613251	1.0	5002.515357	177.278785	7.468491e-35

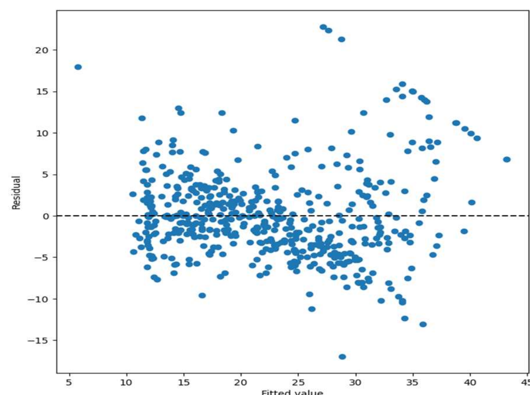
ここで、`results1` は予測子 `lstat` と `age` を含む線形サブモデルを表し、`results3` は `lstat` の 2 次項を含む上記のより大きなモデルに対応する。 `anova_lm()` 関数は、2 つのモデルを比較する仮説検定を実行する。帰無仮説は、より大きなモデルの 2 次項は不要であり、対立仮説は、より大きなモデルの方が優れているというものである。ここで、F 統計量は 177.28 で、関連する p 値は 0 であり、この場合、F 統計量は、`results3` の線形モデル要約の 2 次項の t 統計量の 2 乗となる。これは、これらのネストされたモデルが 1 自由度だけ異なるという事実の結果である。 これは、`lstat` の 2 次多項式が線形モデルを改善するというかなり明確な証拠を提供

してくれるが、これは驚くことではない。先ほど、`medv` と `lstat` の関係に非線形性の証拠があることを確認したばかりである。

関数 `anova_lm()` は、2 つ以上のモデルを入力として受け取ることができる。その場合、連続するすべてのモデル ペアを比較する必要がある。また最初の行に `NaN` があるのは、最初の行と比較する前のモデルがないためである。

In [33]:

```
ax = subplots(figsize=(8,8))[1]
ax.scatter(results3.fittedvalues, results3.resid)
ax.set_xlabel('Fitted value')
ax.set_ylabel('Residual')
ax.axhline(0, c='k', ls='--');
```



モデルに 2 次項が含まれている場合、残差にはほとんどパターンが見られないことが分かる。 3 次以上の多項式近似を作成するには、次数引数を `poly()` に変更するだけでよい。

質的予測量(Qualitative Predictors)

ここでは、`ISLP` パッケージに含まれる `Carseats` データを使用する。いくつかの予測子に基づいて、400 か所の `Sales` (チャイルド・カー・シートの売上) を予測してみよう。

In [35]:

```
Carseats = load_data('Carseats')
Carseats.columns
```

Out[35]:

```
Index(['Sales', 'CompPrice', 'Income', 'Advertising', 'Population', 'Price',
       'ShelveLoc', 'Age', 'Education', 'Urban', 'US'],
      dtype='object')
```

`Carseats` データには、`ShelveLoc` などの質的予測子が含まれている。これは、棚の位置の質、つまり店舗内でカーシートが展示されているスペースの指標である。予測子 `ShelveLoc` は、`Bad`、`Medium`、および `Good` の 3 つの値を取る。`ShelveLoc` などの質的変数が指定されると、`ModelSpec()` は自動的にダミー変数を生成する。これらの変数は、カテゴリ特性の *ワン・ホット・エンコーディング* と呼ばれることがよくある。列の合計は 1 になるため、切片との共線性を回避するために、最初の列は削除される。以下の例では、`Bad` が `ShelveLoc` の最初のレベルであるため、列 `ShelveLoc[Bad]` が削除されていることが分かる。以下では、いくつかの相互作用項を含む重回帰モデルをフィットしてみよう。

In [36]:

```
allvars = list(Carseats.columns.drop('Sales'))
y = Carseats['Sales']
final = allvars + [('Income', 'Advertising'),
                   ('Price', 'Age')]
X = MS(final).fit_transform(Carseats)
model = sm.OLS(y, X)
summarize(model.fit())
```

Out[36]:

	coef	std err	t	P>|t|
Intercept	6.5756	1.009	6.519	0.000
CompPrice	0.0929	0.004	22.567	0.000
Income	0.0109	0.003	4.183	0.000
Advertising	0.0702	0.023	3.107	0.002

Population	0.0002	0.000	0.433	0.665
Price	-0.1008	0.007	-13.549	0.000
ShelveLoc[Good]	4.8487	0.153	31.724	0.000
ShelveLoc[Medium]	1.9533	0.126	15.531	0.000
Age	-0.0579	0.016	-3.633	0.000
Education	-0.0209	0.020	-1.063	0.288
Urban[Yes]	0.1402	0.112	1.247	0.213
US[Yes]	-0.1576	0.149	-1.058	0.291
Income:Advertising	0.0008	0.000	2.698	0.007
Price:Age	0.0001	0.000	0.801	0.424

上記の最初の行では、`allvars` をリストにして、2 行下に相互作用項を追加できるようにした。

モデル・マトリックス・ビルダーは、棚の位置が良好な場合は 1、そうでない場合は 0 になる `ShelveLoc[Good]` ダミー変数を作成している。また、棚の位置が中程度の場合は 1、そうでない場合は 0 になる `ShelveLoc[Medium]` ダミー変数も作成している。

変数・棚の位置が悪い場合は、2 つのダミー変数のそれぞれが 0 になる。回帰出力の `ShelveLoc[Good]` の係数が正であるという事実は、棚の位置が良いと (悪い位置に比べて) 売上が高くなることを示している。 また、`ShelveLoc[Medium]` の正の係数は小さく、中程度の棚位置では悪い棚位置よりも売上が高くなるが、良い棚位置よりも売上が低くなることを示している。

ISLP 第 4 章：ロジスティック回帰、LDA、QDA、および KNN

株式市場データ

このラボでは、`ISLP` ライブラリに含まれる `Smarket` データを分析する。このデータセットは、2001 年初めから 2005 年末までの 1,250 日間にわたる S&P 500 株価指数のパーセンテージリターンで構成されている。各日付について、前の 5 営業日のパーセンテージリターン（`Lag1` から `Lag5` まで）が記録されている。また、`Volume`（前日の取引量、単位は 10 億株）、`Today`（その日のパーセンテージリターン）、および `Direction`（その日の市場が `Up` か `Down` か）も記録されている。

まず、必要なライブラリをインポートしよう。これらはすべて以前のラボで既にインポートしたことがある。

In [1]:

```
#pip install ISLP
```

In [2]:

```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
import statsmodels.api as sm
from ISLP import load_data
from ISLP.models import (ModelSpec as MS,
                         summarize)
```

また、このラボに必要な新しいインポートもまとめて行う。

In [3]:

```
from ISLP import confusion_table
from ISLP.models import contrast
from sklearn.discriminant_analysis import \
     (LinearDiscriminantAnalysis as LDA,
      QuadraticDiscriminantAnalysis as QDA)
from sklearn.naive_bayes import GaussianNB
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
```

これで、`Smarket`データをロードする準備が整った。

In [4]:

```
Smarket = load_data('Smarket')
Smarket
```

Out[4]:

	Year	Lag1	Lag2	Lag3	Lag4	Lag5	Volume	Today	Direction
0	2001	0.381	-0.192	-2.624	-1.055	5.010	1.19130	0.959	Up
1	2001	0.959	0.381	-0.192	-2.624	-1.055	1.29650	1.032	Up
2	2001	1.032	0.959	0.381	-0.192	-2.624	1.41120	-0.623	Down
3	2001	-0.623	1.032	0.959	0.381	-0.192	1.27600	0.614	Up
4	2001	0.614	-0.623	1.032	0.959	0.381	1.20570	0.213	Up
...	...	...	...	...	...	...	...	...	...
1245	2005	0.422	0.252	-0.024	-0.584	-0.285	1.88850	0.043	Up
1246	2005	0.043	0.422	0.252	-0.024	-0.584	1.28581	-0.955	Down
1247	2005	-0.955	0.043	0.422	0.252	-0.024	1.54047	0.130	Up
1248	2005	0.130	-0.955	0.043	0.422	0.252	1.42236	-0.298	Down
1249	2005	-0.298	0.130	-0.955	0.043	0.422	1.38254	-0.489	Down

1250 rows × 9 columns

このコマンドを実行すると、データの一部が表示され、変数名を確認できる。

In [5]:

```
Smarket.columns
```

Out[5]:

```
Index(['Year', 'Lag1', 'Lag2', 'Lag3', 'Lag4', 'Lag5', 'Volume', 'Today',
       'Direction'],
      dtype='object')
```

データフレームの`corr()`メソッドを使用して相関行列を計算しよう。この操作により、変数間のすべてのペアワイズ相関を含む行列が生成される。

ここで`pandas`に数値変数のみを使用するよう指示することで、`Direction`変数の相関は報告されないが、これは定性的な変数だからである。

In [6]:

```
Smarket.corr(numeric_only=True)
```

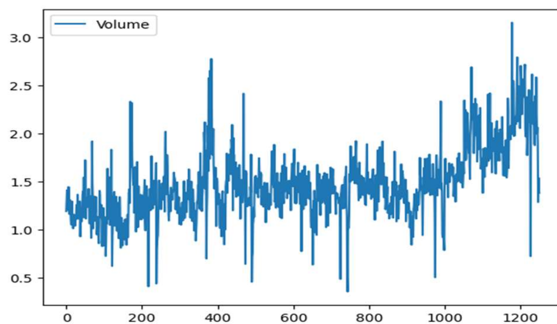
Out[6]:

	Year	Lag1	Lag2	Lag3	Lag4	Lag5	Volume	Today
Year	1.000000	0.029700	0.030596	0.033195	0.035689	0.029788	0.539006	0.030095
Lag1	0.029700	1.000000	-0.026294	-0.010803	-0.002986	-0.005675	0.040910	-0.026155
Lag2	0.030596	-0.026294	1.000000	-0.025897	-0.010854	-0.003558	-0.043383	-0.010250
Lag3	0.033195	-0.010803	-0.025897	1.000000	-0.024051	-0.018808	-0.041824	-0.002448
Lag4	0.035689	-0.002986	-0.010854	-0.024051	1.000000	-0.027084	-0.048414	-0.006900
Lag5	0.029788	-0.005675	-0.003558	-0.018808	-0.027084	1.000000	-0.022002	-0.034860
Volume	0.539006	0.040910	-0.043383	-0.041824	-0.048414	-0.022002	1.000000	0.014592
Today	0.030095	-0.026155	-0.010250	-0.002448	-0.006900	-0.034860	0.014592	1.000000

予想通り、ラグ付きリターン変数と当日のリターンとの間の相関はほぼゼロである。唯一の顕著な相関は`Year`と`Volume`の間にある。データをプロットすることで、`Volume`が時間とともに増加していることが分かる。つまり、2001 年から 2005 年にかけて、1 日の平均取引量が増加していることが分かった。

In [7]:

```
Smarket.plot(y='Volume');
```



ロジスティック回帰

次に、`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る。`sm.GLM()`関数は*一般化線形モデル*のフィットに利用されるが、この関数にはロジスティック回帰も含まれている。別の方法として、`sm.Logit()`関数ではロジスティック回帰モデルを直接にフィットする。`sm.GLM()`の構文は `sm.OLS()`と似ているが、`statsmodels` に他の種類の一般化線形モデルではなくロジスティック回帰を実行するように指示するため、`family=sm.families.Binomial()`引数を渡す必要がある。

In [8]:

```
allvars = Smarket.columns.drop(['Today', 'Direction', 'Year'])
design = MS(allvars)
X = design.fit_transform(Smarket)
y = Smarket.Direction == 'Up'
glm = sm.GLM(y,
             X,
             family=sm.families.Binomial())
results = glm.fit()
summarize(results)
```

Out[8]:

	coef	std err	Z	P>|z|
Intercept	-0.1260	0.241	-0.523	0.601
Lag1	-0.0731	0.050	-1.457	0.145
Lag2	-0.0423	0.050	-0.845	0.398

Lag3	0.0111	0.050	0.222	0.824
Lag4	0.0094	0.050	0.187	0.851
Lag5	0.0103	0.050	0.208	0.835
Volume	0.1354	0.158	0.855	0.392

ここで最も小さい p 値は `Lag1` に関連している。この予測変数の負の係数は、市場が前日に正のリターンを持っていた場合、今日は上昇する可能性が低いことを示唆している。しかし、0.15 という値は比較的大きいが、`Lag1` と `Direction` の間に明確な関連性があるという証拠ではない。

`results` の `params` 属性を利用してこの適合モデルの係数だけにアクセスしてみよう。

In [9]:

```
results.params
```

Out[9]:

```
intercept   -0.126000
Lag1        -0.073074
Lag2        -0.042301
Lag3         0.011085
Lag4         0.009359
Lag5         0.010313
Volume       0.135441
dtype: float64
```

同様に、係数の p 値にアクセスするためには `pvalues` 属性を利用すればよい。

In [10]:

```
results.pvalues
```

Out[10]:

```
intercept    0.600700
Lag1         0.145232
Lag2         0.398352
Lag3         0.824334
Lag4         0.851445
```

```
Lag5         0.834998
Volume       0.392404
dtype: float64
```

`results` の `predict()`メソッドを使用して、予測変数の値を考慮すると市場が上昇する確率を予測することができる。この方法では確率スケールで予測値を返してくる。`predict()`関数にデータセットが提供されていない場合、ロジスティック回帰モデルをフィットするために利用したトレーニングデータの確率が計算される。線形回帰と同様に、必要に応じてデザイン行列と一致するオプションの `exog` 引数を渡すことができる。ここでは最初の 10 個の確率のみを表示しておこう。

In [11]:

```
probs = results.predict()
probs[:10]
```

Out[11]:

```
array([0.50708413, 0.48146788, 0.48113883, 0.51522236, 0.51078116,
       0.50695646, 0.49265087, 0.50922916, 0.51761353, 0.48883778])
```

特定の日に市場が上昇するか下降するかを予測するために、これらの予測確率を `Up` または `Down` のクラスラベルに変換する必要がある。次の 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。

In [12]:

```
labels = np.array(['Down']*1250)
labels[probs>0.5] = "Up"
```

`ISLP` パッケージの `confusion_table()`関数ではこれらの予測を要約し、正しくまたは誤って分類された観測数を示してくれる。この関数は、`sklearn.metrics` モジュールの同様の関数から適応されたもので、生成された行列を転置し、行と列のラベルを含める。`confusion_table()`関数の第一引数は予測ラベル、第二引数は真のラベルである。

In [13]:

```
confusion_table(labels, Smarket.Direction)
```

Out[13]:

True Predicted	Down	Up
Down	145	141
Up	457	507

混同行列の対角要素は正しい予測であることを示し、非対角要素は誤った予測であることを示している。つまり、モデルは 507 日間の市場上昇と 145 日間の市場下降を正しく予測したことになる。合計で 507 + 145 = 652 回の正しい予測となった。`np.mean()`関数を使用して予測が正しかった日の割合を計算することができる。この場合、ロジスティック回帰により市場の動きを 52.2%の確率で正しく予測している。

In [14]:

```
(507+145)/1250, np.mean(labels == Smarket.Direction)
```

Out[14]:

```
(0.5216, 0.5216)
```

一見すると、ロジスティック回帰モデルはランダムに推測するよりも少し良く機能しているように見える。しかし、この結果は誤解を招くかもしれない。なぜなら、同じ 1,250 の観測値セットでモデルを訓練し、テストした結果だからである。言い換えれば、100-52.2=47.8%は訓練誤差率である。前述のように、訓練誤差率はしばしば過度に楽観的であり、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良く評価するためには、データの一部を使用してモデルをフィットし、保持したデータの予測精度を確認する必要がある。この方法ならモデルの性能を評価する際に、モデルを適合させるために使用したデータではなく、市場の動きが未知の将来の日に対する性能に興味があるため、より現実的な誤差率をもたらすだろう。

この戦略を実装するために、まず 2001 年から 2004 年までの観測に対応するブールベクトルを作成する。そして、このベクトルを使用して 2005 年の観測の保持データセットを作成する。

In [15]:

```
train = (Smarket.Year < 2005)
Smarket_train = Smarket.loc[train]
Smarket_test = Smarket.loc[~train]
Smarket_test.shape
```

Out[15]:

```
(252, 9)
```

`train` オブジェクトはデータセット内の観測に対応する 1,250 個の要素を持つベクトルである。2005 年より前の観測に対応するベクトルの要素は `True` に設定され、2005 年の観測に対応する要素は `False` に設定されている。したがって、`train` はブール配列であり、要素は `True` と `False` である。ブール配列を使用して、`loc` メソッドを使用してデータフレームの行または列のサブセットを取得できる。たとえば、`Smarket.loc[train]`コマンドは、2005 年より前の日付に対応する株式市場データのサブマトリックスの選択を意味する。~記号はブールベクトルのすべての要素を否定するために使用できる。つまり、`~train` は `train` に類似のベクトルだが、`train` 内で `True` である要素が`~train` で `False` に置き換えられ、その逆も同様である。したがって、`Smarket.loc[~train]`は `train` が `False` である観測のみを含むデータフレームの行のサブセットを返す。上記の出力は、252 個の観測があることを示している。

次に、2005 年より前の日付に対応する観測のサブセットのみを使用してロジスティック回帰モデルをフィットする。その後、テストセットの日付、つまり 2005 年の日々に対する株式市場の上昇確率を予測する。

In [16]:

```
X_train, X_test = X.loc[train], X.loc[~train]
y_train, y_test = y.loc[train], y.loc[~train]
glm_train = sm.GLM(y_train,
```

```
                    X_train,
                    family=sm.families.Binomial())
results = glm_train.fit()
probs = results.predict(exog=X_test)
```

このようにして、2 つの完全に別々のデータセットでモデルを訓練およびテストしてみた。訓練データは 2005 年より前の日付のみを使用し、テストデータは 2005 年の日付のみを使用した。

最後に、2005 年の予測をその期間の実際の市場の動きと比較してみる。最初にテストとトレーニングのラベルを保存しておこう（ここで `y_test` が二項変数であることを思い出しておこう）。

In [17]:

```
D = Smarket.Direction
L_train, L_test = D.loc[train], D.loc[~train]
```

次に、50%の閾値で適合確率をしきい値処理して予測ラベルを作成する。

In [18]:

```
labels = np.array(['Down']*252)
labels[probs>0.5] = 'Up'
confusion_table(labels, L_test)
```

Out[18]:

True Predicted	Down	Up
Down	77	97
Up	34	44

テスト精度は約 48%で、誤差率は約 52%であった。

In [19]:

```
np.mean(labels == L_test), np.mean(labels != L_test)
```

Out[19]:

```
(0.4801587301587302, 0.5198412698412699)
```

ここで`!=`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誤差率を計算している。この結果はかなり失望的である：テスト誤差率は 52%で、ランダムに予測するよりも精度が低くなっているが、この結果はそれほど驚くべきことではない。一般的に、前日のリターンを使用して将来の市場のパフォーマンスを予測できると期待することはほぼないと言えるからである（結局のところ、もしそれが可能であれば、この本の著者は統計学の教科書を書いている代わりに巨万の富を築いていただろう）。

ロジスティック回帰モデルでは、すべての予測変数に対する p 値は大きく、最も小さい p 値も `Lag1` に対応しているものの、それほど小さくはなかった。ここで `Direction` の予測に役立たないと思われる変数を削除することで、より効果的なモデルを得ることができるかもしれない。結局のところ、応答との関連がない予測変数を使用すると、テスト誤差率の悪化を引き起こす傾向があり（そのような予測変数はバイアスの減少に対応しない分散の増加を引き起こすため）、そのような予測変数を削除することで予測に改善がもたらされる可能性はある。そこで以下では、元のロジスティック回帰モデルで最も予測力が高いと思われる `Lag1` と `Lag2` のみを利用してロジスティック回帰を再びフィットしてみる。

In [20]:

```
model = MS(['Lag1', 'Lag2']).fit(Smarket)
X = model.transform(Smarket)
X_train, X_test = X.loc[train], X.loc[~train]
glm_train = sm.GLM(y_train,
                   X_train,
                   family=sm.families.Binomial())
results = glm_train.fit()
probs = results.predict(exog=X_test)
labels = np.array(['Down']*252)
labels[probs>0.5] = 'Up'
confusion_table(labels, L_test)
```

Out[20]:

True Predicted	Down	Up
Down	35	35
Up	76	106

全体の精度とロジスティック回帰が上昇を予測する日の精度を評価する。

In [21]:

```
(35+106)/252,106/(106+76)
```

Out[21]:

```
(0.5595238095238095, 0.5824175824175825)
```

今度の結果は少し良くなったように見える：日々の動きの 56%が正しく予測された。この場合、毎日市場が上昇すると予測する単純な戦略も 56%の確率で正しいことになる。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よりも優れているというわけではない。しかし、混同行列により、ロジスティック回帰が市場の上昇を予測する日には 58%の精度を持っていることを示している。これは、モデルが市場の上昇を予測する日に購入し、減少を予測する日に取引を回避するという可能な取引戦略を示唆していると言える。もちろん、この小さな改善が現実のものであるか、単なる偶然によるものであるかを慎重に調査する必要がある。

ここで特定の `Lag1` および `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` が 1.2、`Lag2` が 1.1 である日と、`Lag1` が 1.5、`Lag2` が-0.8 である日を予測したいとする。この予測を `predict()`関数を利用して行ってみると次のようになる。

In [22]:

```
newdata = pd.DataFrame({'Lag1':[1.2, 1.5],
                        'Lag2':[1.1, -0.8]});
newX = model.transform(newdata)
results.predict(newX)
```

Out[22]:

```
0    0.479146
1    0.496094
dtype: float64
```

線形判別分析

`LinearDiscriminantAnalysis()`関数（略して`LDA()`）を利用して、`Smarket`データに対して LDA を実行する。モデルの推定では 2005 年以前の観測値のみを使用してフィットする。

In [23]:

```
lda = LDA(store_covariance=True)
```

`LDA`推定量は自動的に切片を追加するため、`X_train`と`X_test`の両方で切片に対応する列を削除する必要がある。また、ブールしたベクトル`y_train`ではなく、直接ラベルを使用することもできる。

In [24]:

```
X_train, X_test = [M.drop(columns=['intercept'])
                   for M in [X_train, X_test]]
lda.fit(X_train, L_train)
```

Out[24]:

```
LinearDiscriminantAnalysis(store_covariance=True)
```

LinearDiscriminantAnalysis

```
LinearDiscriminantAnalysis(store_covariance=True)
```

ここでは 3.6.4 節で導入したリストの表記を利用する。上記の最初の行を見ると、右側が長さ 2 のリストであることが分かる。これは、`for M in [X_train, X_test]`

のコードが長さ 2 のリストを反復するためである。ここではリストをループしているが、リストの表記は任意の反復可能オブジェクトをループするときに機能する。次に、各反復要素に `drop()`メソッドを適用し、その結果をリストに収集する。左側は、この長さ 2 のリストを展開し、その要素を変数 `X_train` と `X_test` に割り当てるように `Python` に指示している。もちろん、これにより `X_train` と `X_test` の以前の値を上書きする。

モデルをフィットした後、`means_`属性を利用すると 2 つのクラスの平均を抽出できる。これらは、各クラス内の各予測変数の平均であり、LDA によってμ_kの推定値として使用される。このことは、市場が上昇する日は前の 2 日のリターンが負になる傾向があり、市場が下降する日は前日のリターンが正になる傾向があることが示唆されている。

In [25]:

```
lda.means_
```

Out[25]:

```
array([[ 0.04279022,  0.03389409],
       [-0.03954635, -0.03132544]])
```

推定された事前確率は `priors_`属性に格納される。`sklearn` パッケージは通常、`fit()`メソッドを利用するときに推定された量を示すために、この末尾の`_`を使用する。どのエントリがどのラベルに対応するかは `classes_`属性を見て確認できる。

In [26]:

```
lda.classes_
```

Out[26]:

```
array(['Down', 'Up'], dtype='<U4')
```

LDA の出力より、$\hat{\pi}_{Down} = 0.492$および$\hat{\pi}_{Up} = 0.508$となることが示されている。

In [27]:

```
lda.priors_
```

Out[27]:

```
array([0.49198397, 0.50801603])
```

線形判別ベクトルは `scalings_` 属性にある：

In [28]:

```
lda.scalings_
```

Out[28]:

```
array([[-0.64201904],
       [-0.51352928]])
```

これらの値は、LDA 決定ルールを形成するために使用される `Lag1` および `Lag2` の線形結合を与える。言い換えれば、これらは(4.24)の$X = x$の要素の乗数となる。 -0.64 ×`Lag1` - 0.51 ×`Lag2` が大きい場合、LDA 分類器は市場の上昇を予測し、小さい場合は市場の下降を予測している。

In [29]:

```
lda_pred = lda.predict(X_test)
```

分類方法の比較で観察したように（4.5 節）、LDA とロジスティック回帰の予測はほぼ同一になる。

In [30]:

```
confusion_table(lda_pred, L_test)
```

Out[30]:

True Predicted	Down	Up
Down	35	35
Up	76	106

また、トレーニングセットの各ポイントに対して各クラスの確率を推定することもできる。クラス 1 に属する事後確率に 50%の閾値を適用することで、`lda_pred` に含まれる予測を再び作成できる。

In [31]:

```
lda_prob = lda.predict_proba(X_test)
np.all(
       np.where(lda_prob[:,1] >= 0.5, 'Up','Down') == lda_pred
       )
```

Out[31]:

```
True
```

上記では、`np.where()`関数を使用して、`lda_prob` の 2 列目（'Up'の推定事後確率）が 0.5 より大きいインデックスに対して`'Up'`の値を持つ配列を作成した。2 つ以上のクラスがある問題では、事後確率が最も高いクラスとしてラベルが選択される：

In [32]:

```
np.all(
       [lda.classes_[i] for i in np.argmax(lda_prob, 1)] == lda_pred
       )
```

Out[32]:

```
True
```

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である。たとえば、市場がその日に下降する確率が非常に高いとき（たとえば、事後確率が少なくとも 90%）にのみ市場の下降を予測したいとしよう。`classes_`属性を確認した後、`lda_prob` の最初の列が `Down` ラベルに対応することが分かるので、上記のように 1 ではなく列インデックス 0 を利用する。

In [33]:

```
np.sum(lda_prob[:,0] > 0.9)
```

Out[33]:

```
0
```

2005 年にはその閾値に達する日はなかった。実際、2005 年全体での最大の下降確率は 52.02%であった。

上記の LDA 分類器は、`sklearn` ライブラリの最初の分類器である。いくつかの他のオブジェクトもこのライブラリから使用できる。これらのオブジェクトは共通の構造を持ち、クロスバリデーションなどのタスクを簡素化してくれる（Chapter~\ref{Ch5:resample}を参照）。具体的には、まずデータに言及せずに汎用分類器を作成する。この分類器は `fit()`メソッドによりデータにフィットし、予測は常に `predict()`メソッドにより生成される。この分類器を最初に利用してフィット、次に予測を生成するパターンは、`sklearn` で設計されている。この統一性により、クロスバリデーションで発生する異なるトレーニングセットにフィットするために分類器をクリーンにコピーすることが可能になる。この標準パターンにより、ワークフローの予測が可能になる。

二次判別分析

次に、`Smarket` データに QDA モデルをフィットしよう。QDA は、`sklearn` パッケージの `QuadraticDiscriminantAnalysis()`を介して実装されており、これを `QDA()`と略す。構文は `LDA()`と類似している。

In [34]:

```
qda = QDA(store_covariance=True)
qda.fit(X_train, L_train)
```

Out[34]:

```
QuadraticDiscriminantAnalysis(store_covariance=True)
```

QuadraticDiscriminantAnalysis

```
QuadraticDiscriminantAnalysis(store_covariance=True)
```

`QDA()`関数により再び`means_`と`priors_`を計算すると。

In [35]:

```
qda.means_, qda.priors_
```

Out[35]:

```
(array([[ 0.04279022,  0.03389409],
        [-0.03954635, -0.03132544]]),
 array([0.49198397, 0.50801603]))
```

`QDA()`分類器はクラスごとに 1 つの共分散を推定している。ここでは最初のクラスの推定共分散を示しておこう。

In [36]:

```
qda.covariance_[0]
```

Out[36]:

```
array([[ 1.50662277, -0.03924806],
       [-0.03924806,  1.53559498]])
```

この出力にはグループ平均が含まれているが、線形判別の係数は含まれていない。なぜなら、QDA 分類器は予測変数の線形関数ではなく、二次関数を含むからである。`predict()`関数は、LDA とまったく同じ方法で機能する。

In [37]:

```
qda_pred = qda.predict(X_test)
confusion_table(qda_pred, L_test)
```

Out[37]:

True Predicted	Down	Up
Down	30	20
Up	81	121

興味深いことに、QDA の予測は 2005 年のデータを使用してモデルをフィットしていないにもかかわらず、約 60%の精度を持っている。

In [38]:

```
np.mean(qda_pred == L_test)
```

Out[38]:

```
0.5992063492063492
```

このレベルの精度は、正確にモデル化するのが非常に難しいことが知られている株式市場データにとって非常に印象的である。これは、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。ただし、このアプローチが一貫して市場を上回るかどうかを確認する前に、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。

ナイーブベイズ

次に、`Smarket` データにナイーブベイズモデルをフィットする。構文は `LDA()`や `QDA()`と似ている。デフォルトでは、このナイーブベイズ分類器の実装である

`GaussianNB()`は、各定量的特徴にガウス分布を利用してモデル化している。ただし、カーネル密度法を利用して分布を推定することもできる。

In [39]:

```
NB = GaussianNB()
NB.fit(X_train, L_train)
```

Out[39]:

```
GaussianNB()
```

GaussianNB

```
GaussianNB()
```

クラスは `classes_`に格納される。

In [40]:

```
NB.classes_
```

Out[40]:

```
array(['Down', 'Up'], dtype='<U4')
```

クラスの事前確率は `class_prior_`属性に格納されている。

In [41]:

```
NB.class_prior_
```

Out[41]:

```
array([0.49198397, 0.50801603])
```

特徴のパラメータはtheta_とvar_属性にある。行数はクラスの数と等しく、列数は特徴の数と等しくなる。以下に示すように、Downクラスにおける特徴Lag1の平均は 0.043 である。

In [42]:

```
NB.theta_
```

Out[42]:

```
array([[ 0.04279022,  0.03389409],
       [-0.03954635, -0.03132544]])
```

分散は 1.503 である。

In [43]:

```
NB.var_
```

Out[43]:

```
array([[1.50355429, 1.53246749],
       [1.51401364, 1.48732877]])
```

これらの属性の名前を知る方法は？NB?（または?NB）を利用すればよい。また平均の計算は簡単に検証できる：

In [44]:

```
X_train[L_train == 'Down'].mean()
```

Out[44]:

```
Lag1    0.042790
Lag2    0.033894
dtype: float64
```

同様に分散も：

In [45]:

```
X_train[L_train == 'Down'].var(ddof=0)
```

Out[45]:

```
Lag1    1.503554
Lag2    1.532467
dtype: float64
```

`NB()`は`sklearn`ライブラリの分類器であるので予測を行うには上記の`LDA()`や`QDA()`と同じ構文を利用する。

In [46]:

```
nb_labels = NB.predict(X_test)
confusion_table(nb_labels, L_test)
```

Out[46]:

True Predicted	Down	Up
Down	29	20
Up	82	121

ナイーブベイズはこれらのデータではよく機能し、約 59%の精度で正確な予測を行っている。これは QDA にやや劣るが、LDA よりもはるかに優れている。`LDA`と同様に、`predict_proba()`メソッドでは各観測が特定のクラスに属する確率を推定する。

In [47]:

```
NB.predict_proba(X_test)[:5]
```

Out[47]:

```
array([[0.4873288 , 0.5126712 ],
       [0.47623584, 0.52376416],
       [0.46529531, 0.53470469],
       [0.47484469, 0.52515531],
       [0.49020587, 0.50979413]])
```

K-近傍法

次に、`KNeighborsClassifier()`関数を使用して KNN を実行しよう。この関数は、これまでに遭遇した他のモデルをフィットする関数と同様に機能する。

LDA および QDA の場合と同様に、`fit` メソッドを利用して分類器をフィットする。新しい予測は `fit()`から返されるオブジェクトの `predict` メソッドを利用して行う。

In [48]:

```
knn1 = KNeighborsClassifier(n_neighbors=1)
X_train, X_test = [np.asarray(X) for X in [X_train, X_test]]
knn1.fit(X_train, L_train)
knn1_pred = knn1.predict(X_test)
confusion_table(knn1_pred, L_test)
```

Out[48]:

True Predicted	Down	Up
Down	43	58
Up	68	83

$K = 1$を使用した結果はあまり良くなく、観測の 50%しか正しく予測していない。もちろん、$K = 1$はデータに対して過度に柔軟すぎるフィットしている可能性がある。

In [49]:

```
(83+43)/252, np.mean(knn1_pred == L_test)
```

Out[49]:

```
(0.5, 0.5)
```

以下では$K = 3$として分析を繰り返してみる。

In [50]:

```
knn3 = KNeighborsClassifier(n_neighbors=3)
knn3_pred = knn3.fit(X_train, L_train).predict(X_test)
np.mean(knn3_pred == L_test)
```

Out[50]:

```
0.5317460317460317
```

結果はわずかに改善された。しかし、K をさらに増やしてもこれ以上の改善は見られない。ここでのデータとこのトレーニング/テスト分割の結果では、これまでに検討した方法の中で QDA が最良の結果をもたらすようである。

KNN は `Smarket` データではうまく機能しなかったが、しばしば印象的な結果を提供してくれる。例として、`ISLP` ライブラリの一部である `Caravan` 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 人の個人の人口動態の特性を測定する 85 の予測変数が含まれている。応答変数は `Purchase` であり、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、わずか 6%の人々がキャラバン保険を購入している。

In [51]:

```
Caravan = load_data('Caravan')
Purchase = Caravan.Purchase
Purchase.value_counts()
```

Out[51]:

```
Purchase
No     5474
```

```
Yes     348
Name: count, dtype: int64
```

`value_counts()`メソッドは、`pd.Series`または`pd.DataFrame`を取り込み、それぞれのユニークな要素に対する対応するカウントを持つ`pd.Series`を返す。この場合、`Purchase`には`Yes`と`No`の値しかなく、各値がいくつあるかが返される。

In [52]:

```
348 / 5822
```

Out[52]:

```
0.05977327378907592
```

特徴量として`Purchase`以外のすべての列が含まれている。

In [53]:

```
feature_df = Caravan.drop(columns=['Purchase'])
```

KNN 分類器は、特定のテスト観測値のクラスを近くにある観測に基づき予測するので、変数スケールが重要となる。大きなスケールの変数は、観測間の*距離*に大きな影響を与え、小さなスケールの変数よりも KNN 分類器に大きな影響を与える。たとえば、`salary`（ドルで測定）と`age`（年で測定）という 2 つの変数を含むデータセットを考えてみよう。KNN にとって、給与の 1,000 ドルの違いは年齢の 50 年の違いに比べて非常に大きいことになる。その結果、`salary`が KNN 分類結果を駆動し、`age`はほとんど影響を与えないことになる。これは、給与の 1,000 ドルの違いが年齢の 50 年の違いに比べて非常に小さいという直感に反している。さらに、KNN 分類器にとってスケールの重要性は別の問題も引き起こす。給与を日本円で測定したり、年齢を分で測定したりすると、ドルと年で測定した場合とはまったく異なる分類結果が得られることになる。

この問題に対処する良い方法としては、データを*標準化*して、すべての変数に平均 0 と標準偏差 1 に基準化することである。この操作により、すべての変数が同じスケールになる。これは、`StandardScaler()`変換を利用して行うことができる。

In [54]:

```
scaler = StandardScaler(with_mean=True,
                        with_std=True,
                        copy=True)
```

引数 `with_mean` は平均を引くか否かを示し、`with_std` は列を標準偏差 1 にスケーリングするか否かを示している。なお引数 `copy=True` は、可能な場合でも計算をその場で行うのではなく、常にデータをコピーしていることを示している。

この変換は任意のデータに適用する前にフィットすることができる。以下の最初の行では、スケーリングのパラメータが計算され `scaler` に格納されている。2 行目では、実際に標準化された特徴セットが構築されている。

In [55]:

```
scaler.fit(feature_df)
X_std = scaler.transform(feature_df)
```

これで、`feature_std` の各列は標準偏差 1、平均 0 に基準化されている。

In [56]:

```
feature_std = pd.DataFrame(
                 X_std,
                 columns=feature_df.columns);
feature_std.std()
```

Out[56]:

```
MOSTYPE     1.000086
MAANTHUI    1.000086
MGEMOMV     1.000086
MGEMLEEF    1.000086
MOSHOOFD    1.000086
              ...
AZEILPL     1.000086
```

```
APLEZIER    1.000086
AFIETS      1.000086
AINBOED     1.000086
ABYSTAND    1.000086
Length: 85, dtype: float64
```

ここでは標準偏差が完全に 1 になっていないことに注意する必要がある。これは、いくつかの手続きが分散のために$\frac{1}{n}$を使用するのに対し（この場合 `scaler()`）、他の手続きが$\frac{1}{n-1}$を利用している為である（`std()`メソッド）。198 ページの脚注を参照すると良いが、この問題は変数がすべて同じスケールである限り、重要ではない。

`train_test_split()`関数を使用して、観測をテストセット（1,000 の観測を含む）と残りの観測を含むトレーニングセットに分割しよう。引数 `random_state=0` は、コードを再実行するたびに同じ分割が得られることを保証している。

In [57]:

```
(X_train,
 X_test,
 y_train,
 y_test) = train_test_split(np.asarray(feature_std),
                            Purchase,
                            test_size=1000,
                            random_state=0)
```

`?train_test_split` を利用すると、被キーワード引数が `lists`、`arrays`、`pandas dataframes` などであることが分かる。それらはすべて同じ長さ（`shape[0]`）を持ち、したがって*インデックス化することが可能となる*。この場合、データフレーム `feature_std` と応答変数 `Purchase` である。（なお `sklearn` のバグに対処するために `feature_std` を `ndarray` に変換したことに注意しておく。）

$K = 1$を使用してトレーニングデータに KNN モデルをフィットし、テストデータでそのパフォーマンスを評価する。

In [58]:

```
knn1 = KNeighborsClassifier(n_neighbors=1)
knn1_pred = knn1.fit(X_train, y_train).predict(X_test)
np.mean(y_test != knn1_pred), np.mean(y_test != "No")
```

Out[58]:

```
(0.111, 0.067)
```

KNN の 1,000 のテスト観測に対する誤差率は約 11%となる。一見すると、これはかなり良いように見える。しかし、顧客のうち 6%強しか保険を購入しないので、予測変数の値に関係なく常に `No` と予測することで、誤差率をほぼ 6%にまで下げることができる。この問題は ***null rate*** として知られている。

特定の個人に保険を販売しようとすることには何らかの非自明なコストがかかると仮定しよう。たとえば、営業担当者が各潜在顧客を訪問する必要があるかもしれない。会社がランダムに選択した顧客に保険を販売しようとすると、成功率はわずか 6%であり、関連するコストを考慮すると非常に低すぎるかもしれないのである。代わりに、会社は保険を購入する可能性が高い顧客にのみ販売を試みたいと考えることが多い。この場合には、全体の誤差率には関心はない。代わりに、保険を購入すると正しく予測される個人の割合には関心があるだろう。

In [59]:

```
confusion_table(knn1_pred, y_test)
```

Out[59]:

True Predicted	No	Yes
No	880	58
Yes	53	9

KNN の$K = 1$は、保険を購入すると予測された顧客の中ではランダムに推測するよりもはるかに優れていると言える。62 人の顧客のうち 9 人、つまり 14.5%が実際に保険を購入している。これはランダムに推測した場合の成功率の 2 倍となっている。

In [60]:

```
9/(53+9)
```

Out[60]:

```
0.14516129032258066
```

パラメータの調整

KNN の近傍数は*チューニングパラメータ*、または*ハイパーパラメータ*と呼ばれるが、どの値を利用するかは事前に不明である。したがって、これらのパラメータを変化させたときに分類器がテストデータでどのように機能するかを確認することが必要である。これは、2.3.8 節で説明した `for` ループを使用して実現できる。ここでは、近傍数を 1 から 5 まで変化させたときに、保険を購入すると予測されたグループ内での分類器の精度を確認するために `for` ループを利用してみよう。

In [61]:

```
for K in range(1, 6):
    knn = KNeighborsClassifier(n_neighbors=K)
    knn_pred = knn.fit(X_train, y_train).predict(X_test)
    C = confusion_table(knn_pred, y_test)
    templ = ('K={0:d}: # predicted to buy: {1:>2},' +
             '  # who did buy {2:d}, accuracy {3:.1%}')
    pred = C.loc['Yes'].sum()
    did_buy = C.loc['Yes', 'Yes']
    print(templ.format(
          K,
          pred,
          did_buy,
          did_buy / pred))
K=1: # predicted to buy: 62,  # who did buy 9, accuracy 14.5%
K=2: # predicted to buy:  6,  # who did buy 1, accuracy 16.7%
K=3: # predicted to buy: 20,  # who did buy 3, accuracy 15.0%
K=4: # predicted to buy:  4,  # who did buy 0, accuracy 0.0%
K=5: # predicted to buy:  7,  # who did buy 1, accuracy 14.3%
```

結果にはいくつかの変動が見られる--- `K=4` に対する数値は他の数字とは非常に異なっている。

ロジスティック回帰との比較

比較のために、データにロジスティック回帰モデルをフィットすることもできる。これは `sklearn` で行えるが、デフォルト設定ではロジスティック回帰の*リッジ回帰*バージョンに近いものがフィットされる。これは 6 章で紹介する。以下の引数 `C` を適切に設定することにより設定を変更できる。デフォルト値は 1 だが、非常に大きな数値に設定することで、上記で説明した通常の（正則化されていない）ロジスティック回帰推定量と同じ解に収束することが分かる。

`statsmodels` パッケージとは異なり、`sklearn` は推論よりも分類に重点を置いている。したがって、`statsmodels` で見た `summary` メソッドや、`summarize` で見た簡略化されたバージョンは、`sklearn` の分類器では一般的に利用できない。

In [62]:

```
logit = LogisticRegression(C=1e10, solver='liblinear')
logit.fit(X_train, y_train)
logit_pred = logit.predict_proba(X_test)
logit_labels = np.where(logit_pred[:,1] > .5, 'Yes', 'No')
confusion_table(logit_labels, y_test)
```

Out[62]:

True Predicted	No	Yes
No	931	67
Yes	2	0

上記では、デフォルト設定のソルバーではアルゴリズムが収束しないという警告を避けるために、引数 `solver='liblinear'`を利用した。

分類器の予測確率カットオフとして 0.5 を設定すると、問題が発生する。テスト観測値のうち 2 つしか保険を購入するとしか予測されない。しかし、0.5 のカットオフを使用する必要はない。代わりに、予測確率が 0.25 を超える場合に保険購入と予測すると、結果は大幅に改善される。この場合、29 人が保険を購入すると予測され、そのうち約 31%が正しい予測になる。これはランダムに推測する場合の 5 倍の成功率である。

In [63]:

```
logit_labels = np.where(logit_pred[:,1] > 0.25, 'Yes', 'No')
confusion_table(logit_labels, y_test)
```

Out[63]:

True Predicted	No	Yes
No	913	58
Yes	20	9

In [64]:

```
9/(20+9)
```

Out[64]:

```
0.3103448275862069
```

自転車シェアデータの線形およびポアソン回帰

ここでは、4.6 節で説明されているように、`Bikeshare` データに線形回帰モデルとポアソン回帰モデルをフィットしてみる。応答変数 `bikers` は、2010 年から 2012 年の期間にワシントン D.C.での 1 時間あたりの自転車レンタル数の測定値である。

In [65]:

```
Bike = load_data('Bikeshare')
```

このデータフレームの次元と変数名を確認してみよう。

In [66]:

```
Bike.shape, Bike.columns
```

Out[66]:

```
((8645, 15),
 Index(['season', 'mnth', 'day', 'hr', 'holiday', 'weekday', 'workingday',
```

```
        'weathersit', 'temp', 'atemp', 'hum', 'windspeed', 'casual',
        'registered', 'bikers'],
       dtype='object'))
```

線形回帰

データに線形回帰モデルをフィットすることから始めよう。

In [67]:

```
X = MS(['mnth',
        'hr',
        'workingday',
        'temp',
        'weathersit']).fit_transform(Bike)
Y = Bike['bikers']
M_lm = sm.OLS(Y, X).fit()
summarize(M_lm)
```

Out[67]:

	Coef	std err	t	P>|t|
Intercept	-68.6317	5.307	-12.932	0.000
mnth[Feb]	6.8452	4.287	1.597	0.110
mnth[March]	16.5514	4.301	3.848	0.000
mnth[April]	41.4249	4.972	8.331	0.000
mnth[May]	72.5571	5.641	12.862	0.000
mnth[June]	67.8187	6.544	10.364	0.000
mnth[July]	45.3245	7.081	6.401	0.000
mnth[Aug]	53.2430	6.640	8.019	0.000
mnth[Sept]	66.6783	5.925	11.254	0.000
mnth[Oct]	75.8343	4.950	15.319	0.000
mnth[Nov]	60.3100	4.610	13.083	0.000
mnth[Dec]	46.4577	4.271	10.878	0.000
hr[1]	-14.5793	5.699	-2.558	0.011
hr[2]	-21.5791	5.733	-3.764	0.000
hr[3]	-31.1408	5.778	-5.389	0.000
hr[4]	-36.9075	5.802	-6.361	0.000
hr[5]	-24.1355	5.737	-4.207	0.000

hr[6]	20.5997	5.704	3.612	0.000
hr[7]	120.0931	5.693	21.095	0.000
hr[8]	223.6619	5.690	39.310	0.000
hr[9]	120.5819	5.693	21.182	0.000
hr[10]	83.8013	5.705	14.689	0.000
hr[11]	105.4234	5.722	18.424	0.000
hr[12]	137.2837	5.740	23.916	0.000
hr[13]	136.0359	5.760	23.617	0.000
hr[14]	126.6361	5.776	21.923	0.000
hr[15]	132.0865	5.780	22.852	0.000
hr[16]	178.5206	5.772	30.927	0.000
hr[17]	296.2670	5.749	51.537	0.000
hr[18]	269.4409	5.736	46.976	0.000
hr[19]	186.2558	5.714	32.596	0.000
hr[20]	125.5492	5.704	22.012	0.000
hr[21]	87.5537	5.693	15.378	0.000
hr[22]	59.1226	5.689	10.392	0.000
hr[23]	26.8376	5.688	4.719	0.000
Workingday	1.2696	1.784	0.711	0.477
Temp	157.2094	10.261	15.321	0.000
weathersit[cloudy/misty]	-12.8903	1.964	-6.562	0.000
weathersit[heavy rain/snow]	-109.7446	76.667	-1.431	0.152
weathersit[light rain/snow]	-66.4944	2.965	-22.425	0.000

`hr` には 24 のレベルがあり、すべてで 40 行ある。`M_lm` では、`hr[0]`および`mnth[Jan]`の最初のレベルがベースライン値として扱われるため、こ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ゼロであり、他のすべてのレベルはこれらのベースラインと比較して測定されているのである。たとえば、2 月の係数が 6.845 であることは、他のすべての変数が一定である場合、1 月に比べて平均して約 7 人のライダーが多いことを意味している。同様に、3 月には 1 月に比べて約 16.5 人のライダーが多くなっている。

4.6.1 節で得られる結果は、以下のように変数 `hr` および `mnth` の若干異なるコーディングを設定したことによる。

In [68]:

```
hr_encode = contrast('hr', 'sum')
mnth_encode = contrast('mnth', 'sum')
```

再びフィットする：

In [69]:

```
X2 = MS([mnth_encode,
         hr_encode,
        'workingday',
        'temp',
        'weathersit']).fit_transform(Bike)
M2_lm = sm.OLS(Y, X2).fit()
S2 = summarize(M2_lm)
S2
```

Out[69]:

	Coef	std err	t	P>|t|
Intercept	73.5974	5.132	14.340	0.000
mnth[Jan]	-46.0871	4.085	-11.281	0.000
mnth[Feb]	-39.2419	3.539	-11.088	0.000
mnth[March]	-29.5357	3.155	-9.361	0.000
mnth[April]	-4.6622	2.741	-1.701	0.089
mnth[May]	26.4700	2.851	9.285	0.000
mnth[June]	21.7317	3.465	6.272	0.000
mnth[July]	-0.7626	3.908	-0.195	0.845
mnth[Aug]	7.1560	3.535	2.024	0.043
mnth[Sept]	20.5912	3.046	6.761	0.000
mnth[Oct]	29.7472	2.700	11.019	0.000
mnth[Nov]	14.2229	2.860	4.972	0.000
hr[0]	-96.1420	3.955	-24.307	0.000
hr[1]	-110.7213	3.966	-27.916	0.000
hr[2]	-117.7212	4.016	-29.310	0.000
hr[3]	-127.2828	4.081	-31.191	0.000
hr[4]	-133.0495	4.117	-32.319	0.000
hr[5]	-120.2775	4.037	-29.794	0.000
hr[6]	-75.5424	3.992	-18.925	0.000
hr[7]	23.9511	3.969	6.035	0.000
hr[8]	127.5199	3.950	32.284	0.000
hr[9]	24.4399	3.936	6.209	0.000
hr[10]	-12.3407	3.936	-3.135	0.002
hr[11]	9.2814	3.945	2.353	0.019
hr[12]	41.1417	3.957	10.397	0.000
hr[13]	39.8939	3.975	10.036	0.000
hr[14]	30.4940	3.991	7.641	0.000
hr[15]	35.9445	3.995	8.998	0.000
hr[16]	82.3786	3.988	20.655	0.000
hr[17]	200.1249	3.964	50.488	0.000

hr[18]	173.2989	3.956	43.806	0.000
hr[19]	90.1138	3.940	22.872	0.000
hr[20]	29.4071	3.936	7.471	0.000
hr[21]	-8.5883	3.933	-2.184	0.029
hr[22]	-37.0194	3.934	-9.409	0.000
Workingday	1.2696	1.784	0.711	0.477
Temp	157.2094	10.261	15.321	0.000
weathersit[cloudy/misty]	-12.8903	1.964	-6.562	0.000
weathersit[heavy rain/snow]	-109.7446	76.667	-1.431	0.152
weathersit[light rain/snow]	-66.4944	2.965	-22.425	0.000

2 つのコーディングの違いは何だろうか？ `M2_lm` では、`hr` のレベル `23` および `mnth` のレベル `Dec` 以外のすべてのレベルの係数推定値が報告されている。重要な点は、`M2_lm` では、`mnth` の最後のレベルの（報告されていない）係数推定値がゼロではないということである。代わりに、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。同様に、`M2_lm` では、`hr` の最後のレベルの係数推定値は、他のすべてのレベルの係数推定値の合計の負の値である。つまり、`M2_lm` の `hr` および `mnth` の係数は常にゼロに合計され、平均レベルとの差異として解釈できるのである。たとえば、1 月の係数-46.087 は、他のすべての変数が一定である場合、1 月には年間平均に比べて通常 46 人のライダーが少ないことを示している。

ここでコーディングの選択は、本当は重要ではない。モデルの出力をコーディングに照らして正しく解釈する限り重要とは言えないのである。たとえば、線形モデルの予測はコーディングに関係なく同じであることが分かる。

In [70]:

```
np.sum((M_lm.fittedvalues - M2_lm.fittedvalues)**2)
```

Out[70]:

```
1.5218600542236692e-20
```

二乗差の合計はゼロである。`np.allclose()`関数を使用しても同じ結果になることが分かる。

In [71]:

```
np.allclose(M_lm.fittedvalues, M2_lm.fittedvalues)
```

Out[71]:

```
True
```

本に掲載されている左図を再現するために、まず `mnth` に関連する係数推定値を取得する必要がある。1 月から 11 月の係数は `M2_lm` オブジェクトから直接取得できるが、12 月の係数は他のすべての月の係数の負の合計として明示的に計算する必要がある。そこでまず、`M2_lm` の係数から月に対応するすべての係数を抽出しよう。

In [72]:

```
coef_month = S2[S2.index.str.contains('mnth')]['coef']
coef_month
```

Out[72]:

```
mnth[Jan]     -46.0871
mnth[Feb]     -39.2419
mnth[March]   -29.5357
mnth[April]    -4.6622
mnth[May]      26.4700
mnth[June]     21.7317
mnth[July]     -0.7626
mnth[Aug]       7.1560
mnth[Sept]     20.5912
mnth[Oct]      29.7472
mnth[Nov]      14.2229
Name: coef, dtype: float64
```

次に、他のすべての月の合計の負の値として `Dec` を追加する。

In [73]:

```
months = Bike['mnth'].dtype.categories
coef_month = pd.concat([
                       coef_month,
                       pd.Series([-coef_month.sum()],
```

```
                                              index=['mnth[Dec]']
                                             )
                              ])
coef_month
```

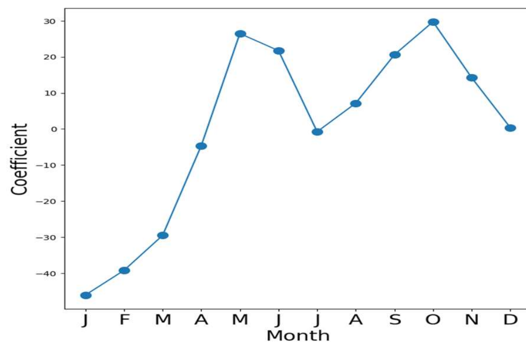
Out[73]:

```
mnth[Jan]     -46.0871
mnth[Feb]     -39.2419
mnth[March]   -29.5357
mnth[April]    -4.6622
mnth[May]      26.4700
mnth[June]     21.7317
mnth[July]     -0.7626
mnth[Aug]       7.1560
mnth[Sept]     20.5912
mnth[Oct]      29.7472
mnth[Nov]      14.2229
mnth[Dec]       0.3705
dtype: float64
```

最後に、プロットを見やすくする為に、インデックスのラベルの 6 文字目である各月の最初の文字だけを利用する。

In [74]:

```
fig_month, ax_month = subplots(figsize=(8, 8))
x_month = np.arange(coef_month.shape[0])
ax_month.plot(x_month, coef_month, marker='o', ms=10)
ax_month.set_xticks(x_month)
ax_month.set_xticklabels([l[5] for l in coef_month.index], fontsize=20)
ax_month.set_xlabel('Month', fontsize=20)
ax_month.set_ylabel('Coefficient', fontsize=20);
```



右図を再現するプロセスも同様に行う。

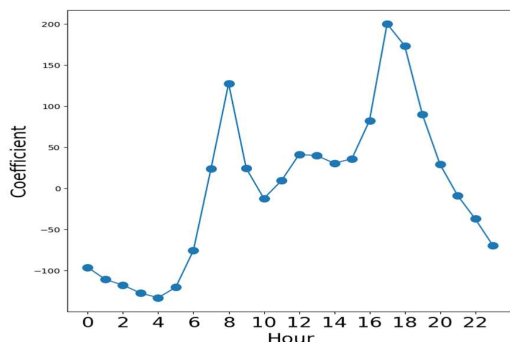
In [75]:

```
coef_hr = S2[S2.index.str.contains('hr')]['coef']
coef_hr = coef_hr.reindex(['hr[{0}]'.format(h) for h in range(23)])
coef_hr = pd.concat([coef_hr,
                     pd.Series([-coef_hr.sum()], index=['hr[23]'])
                    ])
```

次に、時間軸のプロットを作成する。

In [76]:

```
fig_hr, ax_hr = subplots(figsize=(8, 8))
x_hr = np.arange(coef_hr.shape[0])
ax_hr.plot(x_hr, coef_hr, marker='o', ms=10)
ax_hr.set_xticks(x_hr[::2])
ax_hr.set_xticklabels(range(24)[::2], fontsize=20)
ax_hr.set_xlabel('Hour', fontsize=20)
ax_hr.set_ylabel('Coefficient', fontsize=20);
```



ポアソン回帰

次に、`Bikeshare`データにポアソン回帰モデルをフィットしよう。ほとんどコードを変更する必要はないが、今回はポアソンファミリーを指定して`sm.GLM()`関数を利用する。

In [77]:

```
M_pois = sm.GLM(Y, X2, family=sm.families.Poisson()).fit()
```

`mnth`および`hr`に関連する係数をプロットして、4.15 図を再現してみよう。まず、これらの係数を前述のように得る。

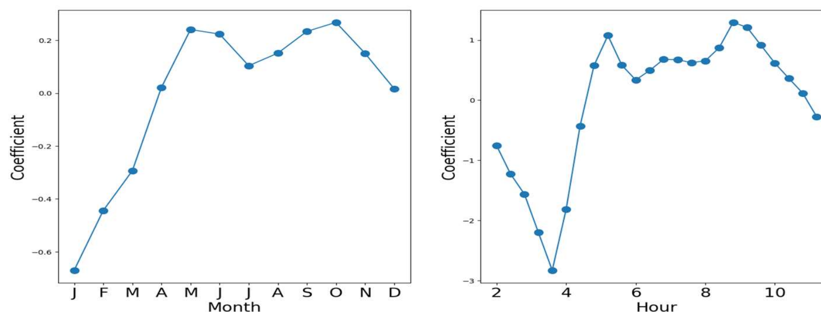
In [78]:

```
S_pois = summarize(M_pois)
coef_month = S_pois[S_pois.index.str.contains('mnth')]['coef']
coef_month = pd.concat([coef_month,
                        pd.Series([-coef_month.sum()],
                                  index=['mnth[Dec]'])])
coef_hr = S_pois[S_pois.index.str.contains('hr')]['coef']
coef_hr = pd.concat([coef_hr,
                     pd.Series([-coef_hr.sum()],
                     index=['hr[23]'])])
```

プロットは以前と同じである。

In [79]:

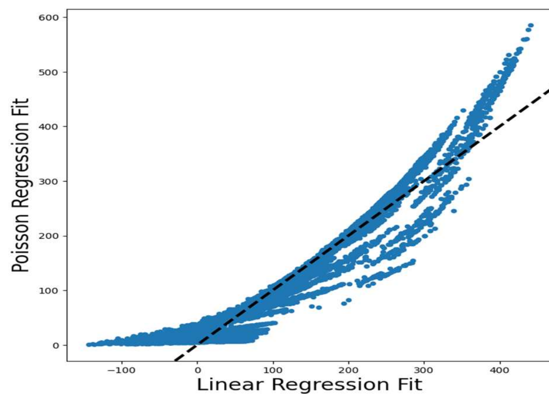
```
fig_pois, (ax_month, ax_hr) = subplots(1, 2, figsize=(16, 8))
ax_month.plot(x_month, coef_month, marker='o', ms=10)
ax_month.set_xticks(x_month)
ax_month.set_xticklabels([l[5] for l in coef_month.index], fontsize=20)
ax_month.set_xlabel('Month', fontsize=20)
ax_month.set_ylabel('Coefficient', fontsize=20)
ax_hr.plot(x_hr, coef_hr, marker='o', ms=10)
ax_hr.set_xticklabels(range(24)[::2], fontsize=20)
ax_hr.set_xlabel('Hour', fontsize=20)
ax_hr.set_ylabel('Coefficient', fontsize=20);
```



2 つのモデルのフィットした値(適合値)を比較しよう。適合値は、線形回帰とポアソン回帰の両方の `fit()`メソッドで返される `fittedvalues` 属性に格納されている。また線形予測子は `lin_pred` 属性として格納されている。

In [80]:

```
fig, ax = subplots(figsize=(8, 8))
ax.scatter(M2_lm.fittedvalues,
           M_pois.fittedvalues,
           s=20)
ax.set_xlabel('Linear Regression Fit', fontsize=20)
ax.set_ylabel('Poisson Regression Fit', fontsize=20)
ax.axline([0, 0], c='black', linewidth=3,
          linestyle='--', slope=1);
```



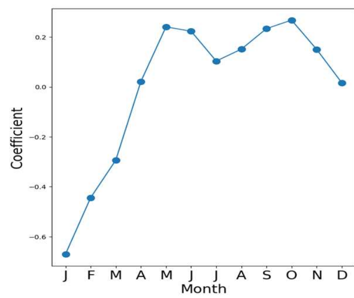
ポアソン回帰モデルによる予測は線形モデルからの予測と相関しているが、前者は非負値を取る。その結果、ポアソン回帰の予測は、非常に低いレベルまたは非常に高いレベルの乗客数に対して、線形モデルの予測よりも大きくなる傾向がある。

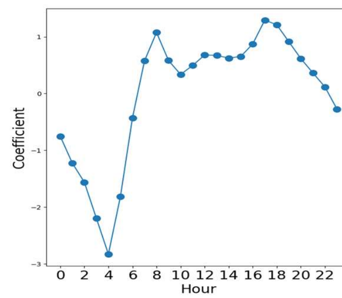
このセクションでは、`sm.GLM()`関数を使用して`family=sm.families.Poisson()`という引数でポアソン回帰モデルをフィットさせてみた。このラボの前半では、`sm.GLM()`関数を使用して`family=sm.families.Binomial()`という引数を指定してロジスティック回帰を実行した。他の GLM モデルをフィットするには`family`引数の他の選択肢を使用することもできる。たとえば、`family=sm.families.Gamma()`によりガンマ回帰モデルをフィットすることができる。

In [83]:

```
fig_pois, (ax_month, ax_hr) = subplots(1, 2, figsize=(16, 8))
ax_month.plot(x_month, coef_month, marker='o', ms=10)
ax_month.set_xticks(x_month)
ax_month.set_xticklabels([l[5] for l in coef_month.index], fontsize=20)
ax_month.set_xlabel('Month', fontsize=20)
ax_month.set_ylabel('Coefficient', fontsize=20)
ax_hr.plot(x_hr, coef_hr, marker='o', ms=10)

import matplotlib.ticker as ticker
locator = ticker.FixedLocator(range(24)[::2])
formatter = ticker.FixedFormatter(range(24)[::2])

ax_hr.xaxis.set_major_locator(locator)
```

```
ax_hr.xaxis.set_major_formatter(formatter)


ax_hr.set_xticklabels(range(24)[::2], fontsize=20)
ax_hr.set_xlabel('Hour', fontsize=20)
ax_hr.set_ylabel('Coefficient', fontsize=20);
```





ISLP 第 5 章 クロスバリデーションとブートストラップ

(Cross-Validation and the Bootstrap)

このラボでは、この章で取り上げたリサンプリング技術を探求する。このラボのいくつかのコマンドは、コンピュータ上で実行するのに時間がかかるかもしれない。他の章と同様に分析で使用するライブラリのインポートをトップレベルに置くことから始めよう。

In [1]:

```
import numpy as np
import statsmodels.api as sm
from ISLP import load_data
from ISLP.models import (ModelSpec as MS,
                         summarize,
                         poly)
from sklearn.model_selection import train_test_split
```

新しくいくつかのライブラリをインポートしよう。

In [3]:

```
from functools import partial
from sklearn.model_selection import \
     (cross_validate,
```

```
        KFold,
        ShuffleSplit)
from sklearn.base import clone
from ISLP.models import sklearn_sm
```

検証集合アプローチ(Validation Set Approach)

`Auto` データセットに様々な線形モデルを当てはめた結果のテスト・エラー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。

関数 `train_test_split()` を用いて、データを訓練セットと検証セットに分割する。392 個のオブザベーションがあるので、引数 `test_size=196` を使って、196 サイズの 2 つの等しいセットに分割する。一般的に、このようなランダムな要素を含む操作を行う際には、再現性を確保するために、シードを設定するのがよいだろう。ここでは `random_state=0` という引数でシードを設定している。

In [4]:

```
Auto = load_data('Auto')
Auto_train, Auto_valid = train_test_split(Auto,
                                          test_size=196,
                                          random_state=0)
```

これで、観測値(オブザベーション)または学習セット `Auto_train` に対応する観測値(オブザベーション)だけを用いて線形回帰をフィットできる。

In [5]:

```
hp_mm = MS(['horsepower'])
X_train = hp_mm.fit_transform(Auto_train)
y_train = Auto_train['mpg']
model = sm.OLS(y_train, X_train)
results = model.fit()
```

ここで、検証データセットを使って作成されたこのモデルのモデル行列を評価する `results` の `predict()` メソッドを使用する。また、モデルの検証 MSE も計算する。

In [6]:

```
X_valid = hp_mm.transform(Auto_valid)
y_valid = Auto_valid['mpg']
valid_pred = results.predict(X_valid)
np.mean((y_valid - valid_pred)**2)
```

Out[6]:

```
23.61661706966988
```

線形回帰フィットの検証 MSE の推定値は23.62となった。

高次の多項式回帰の誤差も推定することができる。まず、`evalMSE()`関数を用意し、モデルと訓練データとテストデータを受け取り、テストデータでの MSE を返す。

In [6]:

```
def evalMSE(terms,
            response,
            train,
            test):

   mm = MS(terms)
   X_train = mm.fit_transform(train)
   y_train = train[response]

   X_test = mm.transform(test)
   y_test = test[response]

   results = sm.OLS(y_train, X_train).fit()
   test_pred = results.predict(X_test)
```

```
    return np.mean((y_test - test_pred)**2)
```

この関数を使って、線形、2 次、3 次での検証 MSE を推定してみよう。ここでは `enumerate()`関数を使用する。この関数は、for ループを繰り返しながらオブジェクトの値とインデックスの両方を与えていく。

In [7]:

```
MSE = np.zeros(3)
for idx, degree in enumerate(range(1, 4)):
    MSE[idx] = evalMSE([poly('horsepower', degree)],
                       'mpg',
                       Auto_train,
                       Auto_valid)
MSE
```

Out[7]:

```
array([23.61661707, 18.76303135, 18.79694163])
```

これらの誤差はそれぞれ 23.62、18.76、18.80 である。もしトレーニング/検証の分割を変えた場合、検証セットでの誤差は多少異なることが予想される。

In [8]:

```
Auto_train, Auto_valid = train_test_split(Auto,
                                          test_size=196,
                                          random_state=3)
MSE = np.zeros(3)
for idx, degree in enumerate(range(1, 4)):
    MSE[idx] = evalMSE([poly('horsepower', degree)],
                       'mpg',
                       Auto_train,
```

```
                                Auto_valid)
MSE
```

Out[8]:

```
array([20.75540796, 16.94510676, 16.97437833])
```

観測値をトレーニング セットと検証セットに分割すると、線形、二次、三次の項を持つモデルの検証セットのエラー率はそれぞれ 20.76、16.95、16.97 であることが分かる。

これらの結果は、以前の調査結果と一致している。すなわち `horsepower` の二次関数を使用して `mpg` を予測するモデルは、`horsepower` の線形関数のみを含むモデルよりもパフォーマンスが優れており、`horsepower` の三次関数を使用しても改善の証拠はない。

交互検証(Cross-Validation)

理論的には、クロスバリデーション推定値は、どのような一般化線形モデルでも計算できる。しかし、Python でクロスバリデーションを行う最も簡単な方法は `sklearn` を使うことである。

これはデータ科学者がしばしば直面する問題でもある：「タスク A を実行する関数があり、それをタスク B を実行する関数に渡して、B(A(D))を計算する必要がある。A と B が自然に会話しない場合、ラッパーを使う必要がある。`ISLP` パッケージでは、`sklearn_sm()`というラッパーを提供しており、`statsmodels` によって適合されたモデルで `sklearn` の交差検証ツールを簡単に使うことができる。

クラス `sklearn_sm()` は `statsmodels` のモデルを第一引数にとる。さらに、オプションで 2 つの引数を取ることができる：`model_str` は計算式を指定するのに使用し、`model_args` はモデルをフィットする際に使用する追加引数の辞書を指定する。例えば、ロジスティック回帰モデルをフィットするには、`family` 引数を指定する必要がある。これは `model_args={'family':sm.families.Binomial()}`として渡される。

以下がラッパーの動作である：

In [9]:

```
hp_model = sklearn_sm(sm.OLS,
                      MS(['horsepower']))
X, Y = Auto.drop(columns=['mpg']), Auto['mpg']
cv_results = cross_validate(hp_model,
                            X,
                            Y,
                            cv=Auto.shape[0])
cv_err = np.mean(cv_results['test_score'])
cv_err
```

Out[9]:

```
24.23151351792924
```

`cross_validate()`の引数は以下の通りである: 適切な `fit()`、`predict()`、`score()` メソッドを持つオブジェクト、特徴量の配列 `X`、レスポンス `Y`。 また、`cross_validate()` に `cv` という引数を追加した。整数 K を指定すると、K 回のクロスバリデーションが行われる。整数 K を指定すると、K 回のクロスバリデーションになる。オブザベーションの総数に対応する値を指定すると、leave-one-out cross-validation (LOOCV) になる。cross_validate()`関数は、いくつかの成分を持つ辞書を生成する。ここでは、単純にクロスバリデーションされたテストスコア（MSE）を求めるが、これは 24.23 と推定される。

より複雑な多項式フィットに対してもこの手順を繰り返すことができる。このプロセスを自動化するために、1 次から 5 次までの多項式回帰を繰り返しフィットし、関連するクロスバリデーション誤差を計算し、i 番目の要素に格納する for ループを再び使う。 クロスバリデーション誤差を計算し、ベクトル `cv_error` の i 番目の要素に格納する。for ループの変数 `d` は多項式の次数に対応する。まず，ベクトルの初期化を行う． このコマンドの実行には数秒かかる可能性がある。

In [10]:

```
cv_error = np.zeros(5)
H = np.array(Auto['horsepower'])
M = sklearn_sm(sm.OLS)
for i, d in enumerate(range(1,6)):
```

```
    X = np.power.outer(H, np.arange(d+1))
    M_CV = cross_validate(M,
                          X,
                          Y,
                          cv=Auto.shape[0])
    cv_error[i] = np.mean(M_CV['test_score'])
cv_error
```

Out[10]:

```
array([24.23151352, 19.24821312, 19.33498406, 19.42443033, 19.03323827])
```

線形フィットと 2 次フィットの間で推定テスト MSE の急激な低下が見られるが、その後、高次の多項式を使用することによる明確な改善は見られない。

上記では `np.power()` 関数の `outer()` メソッドを紹介しました。`outer()`メソッドは `add()`、`min()`、`power()` のように 2 つの引数を持つ演算に適用される。これは引数として 2 つの配列を持ち、その 2 つの配列の各要素の組に演算が適用されるより大きな配列を形成する。

In [11]:

```
A = np.array([3, 5, 9])
B = np.array([2, 4])
np.add.outer(A, B)
```

Out[11]:

```
array([[ 5,  7],
       [ 7,  9],
       [11, 13]])
```

上の CV の例では K=n を使ったが、もちろん K<n を使うこともできる。コードは上記と非常によく似ている（そしてかなり高速である）。ここでは `KFold()` を使ってデータを K=10 個のランダムなグループに分割、`random_state` を用いてランダム

シードを設定し、ベクトル `cv_error` を初期化して、1 度から 5 度の多項式フィットに対応する CV 誤差を格納する。

In [12]:

```
cv_error = np.zeros(5)
cv = KFold(n_splits=10,
           shuffle=True,
           random_state=0) # use same splits for each degree
for i, d in enumerate(range(1,6)):
    X = np.power.outer(H, np.arange(d+1))
    M_CV = cross_validate(M,
                          X,
                          Y,
                          cv=cv)
    cv_error[i] = np.mean(M_CV['test_score'])
cv_error
```

Out[12]:

```
array([24.20766449, 19.18533142, 19.27626666, 19.47848403, 19.13720581])
```

計算時間は LOOCV よりはるかに短い。 (原理的には、最小 2 乗線形モデルに対する LOOCV の計算時間は K-fold CV の計算時間よりも速いはずである。これは、LOOCV の式が利用できるからである。 しかし、一般的な `cross_validate()` 関数はこの式を使用しない)。 次以上の多項式または高次の多項式を使用することが、単に 2 次フィットを使用するよりも低いテスト誤差につながるという証拠はまだほとんど見られない。

`cross_validate()`関数は柔軟で、さまざまな分割メカニズムを引数に取ることができる。例えば、 `ShuffleSplit()` 関数を使うことで、K-fold 交差検証のように簡単に検証セットのアプローチを実装することができる。

In [13]:

```
validation = ShuffleSplit(n_splits=1,
                          test_size=196,
                          random_state=0)
results = cross_validate(hp_model,
                         Auto.drop(['mpg'], axis=1),
                         Auto['mpg'],
                         cv=validation);
results['test_score']
```

Out[13]:

```
array([23.61661707])
```

テスト・エラーのばらつきは、次のようにして推定できる：

In [14]:

```
validation = ShuffleSplit(n_splits=10,
                          test_size=196,
                          random_state=0)
results = cross_validate(hp_model,
                         Auto.drop(['mpg'], axis=1),
                         Auto['mpg'],
                         cv=validation)
results['test_score'].mean(), results['test_score'].std()
```

Out[14]:

```
(23.802232661034168, 1.421845094109185)
```

この標準偏差は、テスト・スコアの平均値や個々のスコアのサンプリング変動の有効な推定値ではないことに注意する必要がある。ただしこの標準偏差は、異な

る無作為フォールドを選択することによって発生するモンテカルロ変動のアイデアを与えてくれている。

ブートストラップ(Bootstrap)

ブートストラップ法の簡単な例と `Auto` データセットでの線形回帰モデルの精度を推定する例で、ブートストラップの使い方を説明する。

統計量の精度推定(Estimating the Accuracy of a Statistic of Interest)

ブートストラップ・アプローチの大きな利点の 1 つは、ほとんどの状況に適用できることである。複雑な数学的計算は必要ない．Python ではブートストラップ が可能であるが、標準誤差を推定するためのブートストラップの使用は十分に簡単で、データがデータフレームに格納されている場合には以下にように関数を書くことができる。

ブートストラップを説明するために、まず簡単な例から始めよう。ここでの目的はパラメータ α のサンプリング分散を推定することである。 `alpha_func()`関数を作成するが、この関数は列 `X` と `Y` を持つと仮定したデータフレーム `D` と、 α を推定するためにどのオブザベーションを使用するかを示すベクトル `idx` を入力として受け取る。この関数により選択されたオブザベーションに基づいて α の推定値を出力する。

In [15]:

```
Portfolio = load_data('Portfolio')
def alpha_func(D, idx):
   cov_ = np.cov(D[['X','Y']].loc[idx], rowvar=False)
   return ((cov_[1,1] - cov_[0,1]) /
           (cov_[0,0]+cov_[1,1]-2*cov_[0,1]))
```

この関数では引数 `idx` でインデックス付けされたオブザベーションに最小分散公式を適用して、 α の推定値を返す。 例えば、以下のコマンドは 100 個のオブザベーションすべてを使って α を推定している。

In [16]:

```
alpha_func(Portfolio, range(100))
```

Out[16]:

```
0.57583207459283
```

次に、`range(100)` から 100 個の観測値（オブザベーション）をランダムに選ぶ。これは、新しいブートストラップ・データ集合を構築して、新しいデータ集合に基づいて$\hat{\alpha}$ を再計算することと等価である。

In [17]:

```
rng = np.random.default_rng(0)
alpha_func(Portfolio,
           rng.choice(100,
                      100,
                      replace=True))
```

Out[17]:

```
0.6074452469619004
```

このプロセスを一般化して、データフレームのみを引数に取る任意の関数に対してブートストラップ標準誤差を計算するための単純な関数 `boot_SE()` を作成することができる。

In [18]:

```
def boot_SE(func,
            D,
            n=None,
            B=1000,
            seed=0):
    rng = np.random.default_rng(seed)
```

```
    first_, second_ = 0, 0
    n = n or D.shape[0]
    for _ in range(B):
        idx = rng.choice(D.index,
                         n,
                         replace=True)
        value = func(D, idx)
        first_ += value
        second_ += value**2
    return np.sqrt(second_ / B - (first_ / B)**2)
```

`for _ in range(B)`のループ変数として`_`の使い方に注目しよう。これはカウンタの値が重要でない場合によく使われるもので、単純にループが`B`回実行されることを確認するだけである。

この関数を使って、B=1,000 ブートストラップ複製を使って、αの推定値の精度を評価しよう。

In [19]:

```
alpha_SE = boot_SE(alpha_func,
                   Portfolio,
                   B=1000,
                   seed=0)
alpha_SE
```

Out[19]:

```
0.09118176521277699
```

最後の出力は、SE($\hat{\alpha}$)のブートストラップ推定値が 0.0912 であることを示している。

線形回帰の精度推定(Estimating the Accuracy of a Linear Regression Model)

ブートストラップ・アプローチは，統計的学習からの係数推定値や予測値の変動性を評価するのに使うことができる。また ブートストラップ・アプローチは，統計的学習手法からの係数推定値と予測値のばらつきを評価するために使用できる。

ここでは、ブートストラップ・アプローチを用いて、`Auto`データセットにおいて`馬力`を用いて `mpg` を予測する線形回帰モデルの切片項と傾き項である β_0 と β_1 の推定値のばらつきを評価してみよう。ブートストラップを用いて得られた推定値を 3.1.2 節で説明した SE($\hat{\beta}_0$)と SE($\hat{\beta}_1$) の公式を用いて得られた推定値と比較してみる。

`boot_SE()`関数を使うには、関数（第一引数）を書かなければならない。その引数はデータフレーム `D` とインデックス `idx` だけである。しかし、ここでは、モデル式とデータで指定された特定の回帰モデルをブートストラップで評価したいので、いくつかの簡単なステップで行う方法を示す。

回帰モデルをブートストラップするための汎用関数 `boot_OLS()` を書くことから始める。`clone()`関数を使用して、新しいデータフレームに再フィットできる式のコピーを作成する。このことは、`poly()`（まもなく説明する）によって定義されるような派生特徴量は、再サンプリングされたデータフレームに再フィットされることを意味する。

In [20]:

```
def boot_OLS(model_matrix, response, D, idx):
    D_ = D.loc[idx]
    Y_ = D_[response]
    X_ = clone(model_matrix).fit_transform(D_)
    return sm.OLS(Y_, X_).fit().params
```

これは `boot_SE()` の第 1 引数として必要なものとは少し異なる。モデルを指定する最初の 2 つの引数はブートストラップ処理では変更されないので、それらをフリーズさせたいが、 `functools` モジュールの関数 `partial()` はまさにこれを行う。関数を引数として受け取り、その引数の一部を左から順にフリーズする。この関数を使って `boot_OLS()` の最初の 2 つのモデル式の引数をフリーズしてみる。

In [21]:

```
hp_func = partial(boot_OLS, MS(['horsepower']), 'mpg')
```

`hp_func?`と入力すると、この関数が `D` と `idx` の 2 つの引数を持っていることが分かる --- 最初の 2 つの引数を凍結した `boot_OLS()` のバージョンである --- したがって、`boot_SE()` の最初の引数として理想的である。

`hp_func()`関数は、オブザベーションの中からランダムにサンプリングして、切片項と傾き項のブートストラップ推定値を作成するために使用できる。まず、10 個のブートストラップ標本でその有用性が実証できる。

In [22]:

```
rng = np.random.default_rng(0)
np.array([hp_func(Auto,
          rng.choice(Auto.index,
                     392,
                     replace=True)) for _ in range(10)])
```

Out[22]:

```
array([[39.12226577, -0.1555926 ],
       [37.18648613, -0.13915813],
       [37.46989244, -0.14112749],
       [38.56723252, -0.14830116],
       [38.95495707, -0.15315141],
       [39.12563927, -0.15261044],
       [38.45763251, -0.14767251],
       [38.43372587, -0.15019447],
       [37.87581142, -0.1409544 ],
       [37.95949036, -0.1451333 ]])
```

次に、`boot_SE()` {}関数を用いて、切片項と傾きの 1,000 ブートストラップ推定値の標準誤差を計算する。

In [23]:

```
hp_se = boot_SE(hp_func,
                Auto,
                B=1000,
                seed=10)
hp_se
```

Out[23]:

```
intercept     0.731176
horsepower    0.006092
dtype: float64
```

ブートストラップ推定値は 0.85 である。 $\hat{\beta}_0$ のブートストラップ推定値は 0.85 であり、 $\hat{\beta}_1$ のブートストラップ推定値は 0.0074 である。 3.1.2 節で述べたように、線形モデルの回帰係数の標準誤差は標準公式を用いて計算できる。 これらは `ISLP.sm` の `summarize()` 関数を用いて求めることができる。

In [24]:

```
hp_model.fit(Auto, Auto['mpg'])
model_se = summarize(hp_model.results_)['std err']
model_se
```

Out[24]:

```
intercept     0.717
horsepower    0.006
Name: std err, dtype: float64
```

データに非線形関係があることが分かる。そのため、線形近似からの残差は膨らみ、$\hat{\sigma}^2$ も膨む。次に、標準式では (やや非現実的だが) x_i は固定されており、すべての変動は誤差 ε_i の変動から生じると想定されていた。ブートストラップ手法

はこれらの想定にまったく依存しないため、`sm.OLS` の結果よりも $\hat{\beta}_0$ と $\hat{\beta}_1$ の標準誤差のより正確な推定値が得られる可能性がある。

以下では、データに 2 次モデルを適合させた結果のブートストラップ標準誤差推定値と標準線形回帰推定値を計算する。このモデルはデータによくフィットするため、ブートストラップ推定値と標準推定値 $\mathrm{SE}(\hat{\beta}_0)$, $\mathrm{SE}(\hat{\beta}_1)$, $\mathrm{SE}(\hat{\beta}_2)$ の間にはより良い対応関係が見られる。

In [25]:

```
quad_model = MS([poly('horsepower', 2, raw=True)])
quad_func = partial(boot_OLS,
                    quad_model,
                    'mpg')
boot_SE(quad_func, Auto, B=1000)
```

Out[25]:

```
intercept                                  1.538641
poly(horsepower, degree=2, raw=True)[0]    0.024696
poly(horsepower, degree=2, raw=True)[1]    0.000090
dtype: float64
```

結果を `sm.OLS()` を使用して計算された標準誤差と比較してみる。

In [26]:

```
M = sm.OLS(Auto['mpg'],
           quad_model.fit_transform(Auto))
summarize(M.fit())['std err']
```

Out[26]:

```
intercept                                  1.800
poly(horsepower, degree=2, raw=True)[0]    0.031
```

```
poly(horsepower, degree=2, raw=True)[1]    0.000
Name: std err, dtype: float64
```

ISLP 第 6 章 線形モデルと正則化手法

このラボでは、本章で解説された多くの手法を実装する。

訳注 1：Google Colab でこのコードを実行した。Colab が不安定な場合(2025 年 3 月 20 日頃)には「ランタイムからセッションを再起動する」と問題ないことがあるが、Jupyter ではプログラムは正常に動作することを確認した。(Numpy などパッケージ更新のタイミングによるのではと思われる。)また、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。

In [1]:

```
# 訳注 2：

# %%capture は出力を非表示にするために、出力が長くなる箇所で用いている

# もちろん%%capture を消したりコメントアウトしたりすれば、出力が表示される

%%capture
!pip install japanize-matplotlib
```

いくつかのライブラリを最初にインポートする。

In [2]:

```
# 訳注 3：

%%capture
!pip install ISLP # ISLP は初めからインストールされてはいないのでインストールする必要

がある
```

In [3]:

```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
import japanize_matplotlib # グラフで日本語を使うために訳者が追加した
from statsmodels.api import OLS
import sklearn.model_selection as skm
import sklearn.linear_model as skl
from sklearn.preprocessing import StandardScaler
from ISLP import load_data
from ISLP.models import ModelSpec as MS
from functools import partial
```

さらに、このラボで必要なライブラリをインポートする。事前に `pip install l0bnb` を使用して `l0bnb` をインストールする必要がある。

In [4]:

```
# 訳注4：
%%capture
!pip install l0bnb
```

In [5]:

```
from sklearn.pipeline import Pipeline
from sklearn.decomposition import PCA
from sklearn.cross_decomposition import PLSRegression
from ISLP.models import \
     (Stepwise,
      sklearn_selected,
      sklearn_selection_path)
from l0bnb import fit_path
```

訳注**5**：`\`は長い行を分割して、見やすくするために用いられている。

部分集合選択法

ここでは、入力変数の部分集合にモデルを制限することで、モデルのパラメータ数を削減する方法を実装する。

変数増加法 (Forward Selection)

変数増加法を `Hitters` データに適用しよう。このデータでは、前年の成績に関連する統計量を基に野球選手の `Salary`（年俸）を予測する。

まず、`Salary` 変数が一部の選手で欠損していることに注意しておく。`np.isnan()`関数を使用して欠損観測値を特定できる。この関数は、入力ベクトルと同じ形状の配列を返し、欠損している要素には `True`、そうでない要素には `False` を返す。その後、`sum()`メソッドを使用して欠損要素の数をカウントできる。

In [6]:

```
Hitters = load_data('Hitters')
np.isnan(Hitters['Salary']).sum()
```

Out[6]:

```
59
```

59 人の選手で `Salary` が欠損していることがわかる。データフレームの `dropna()`メソッドは、（デフォルトでは）欠損値を持つすべての行を削除する（`Hitters.dropna?`を参照されたい）。

In [7]:

```
Hitters = Hitters.dropna();
Hitters.shape
```

Out[7]:

```
(263, 20)
```

次に、C_p（(6.2）式）を基準とした変数増加法で最良のモデルを選ぼう。このスコアは`sklearn`には組み込みの指標として存在しないため、自分で計算する関数を定義し、それをスコアリング関数として使用する。`sklearn`はデフォルトではスコアを最大化しようとするため、スコアリング関数では負のC_p統計量を計算する。

In [8]:

```
def nCp(sigma2, estimator, X, Y):
    "負の Cp 統計量"
    n, p = X.shape
    Yhat = estimator.predict(X)
    RSS = np.sum((Y - Yhat)**2)
    return -(RSS + 2 * p * sigma2) / n
```

ここで残差分散 σ^2を推定する必要がある。これは上記のスコアリング関数の最初の引数である。すべての変数を使用して最大のモデルをフィットし、その MSE に基づいてσ^2を推定する。

In [9]:

```
design = MS(Hitters.columns.drop('Salary')).fit(Hitters)
Y = np.array(Hitters['Salary']) # Salary を被説明変数として取り出す
X = design.transform(Hitters) # Salary 以外を design を用いて説明変数として取り出す
sigma2 = OLS(Y, X).fit().scale # OLS を用いた場合の誤差分散を計算する
```

関数`sklearn_selected()`は、スコアリング関数を引数に持ち、そのスコアリング関数が持つ引数としては、上記の`nCp()`の定義の最後の 3 つの引数(estimator, X, Y)だけを想定している。そこで`partial()`関数（5.3.3 節で初めて登場）を使用して、`nCp()`の最初の引数を σ^2の推定値で固定する。

In [10]:

```
neg_Cp = partial(nCp, sigma2)
```

これで、`neg_Cp()`をモデル選択のスコアリング関数として使用できる。

スコアと共に探索戦略を指定する必要がある。これは、`ISLP.models`パッケージ内の `Stepwise()`オブジェクトを利用して実行する。`Stepwise.first_peak()`メソッドは、評価スコアの改善が見られなくなるまで変数増加法を実行する。似たものとして、`Stepwise.fixed_steps()`メソッドは、ステップ数が固定されたステップワイズ検索を実行する。

In [11]:

```
strategy = Stepwise.first_peak(design,
                               direction='forward',
                               max_terms=len(design.terms))
```

次に、変数増加法を使用して、`Salary`を目的変数とする線形回帰モデルをフィットする。これは、`ISLP.models`パッケージの`sklearn_selected()`関数を利用して実行する。この関数は`statsmodels`のモデルと探索戦略を受け取り、その`fit`メソッドでモデルを選択する。`scoring`引数を指定しない場合、スコアはデフォルトで MSE となる。 その結果、すべての 19 変数が選択される。

In [12]:

```
hitters_MSE = sklearn_selected(OLS,
                               strategy)
hitters_MSE.fit(Hitters, Y)
hitters_MSE.selected_state_
```

Out[12]:

```
('Assists',
 'AtBat',
 'CAtBat',
 'CHits',
 'CHmRun',
 'CRBI',
 'CRuns',
 'CWalks',
 'Division',
 'Errors',
```

```
 'Hits',
 'HmRun',
 'League',
 'NewLeague',
 'PutOuts',
 'RBI',
 'Runs',
 'Walks',
 'Years')
```

`neg_Cp` を使用すると、想定通り、より小さなモデルが得られ、ちょうど 10 変数が選ばれる。

In [13]:

```
hitters_Cp = sklearn_selected(OLS,
                              strategy,
                              scoring=neg_Cp)
hitters_Cp.fit(Hitters, Y)
hitters_Cp.selected_state_
```

Out[13]:

```
('Assists',
 'AtBat',
 'CAtBat',
 'CRBI',
 'CRuns',
 'CWalks',
 'Division',
 'Hits',
 'PutOuts',
 'Walks')
```

検証集合(Validation Set)アプローチと交差検証(Cross-Validation)を使用したモデルの選択

C_pの代わりに、変数増加法でモデルを選択するために、交差検証を試してみることもできる。そのためには、変数増加法で見つかったモデルの完全なパスを保存

し、それぞれのモデルで予測を行う方法が必要となるが、これは`ISLP.models`の`sklearn_selection_path()`で実現できる。また、`ISLP.models`の`cross_val_predict()`関数は、パスに沿った各モデルの交差検証予測を計算し、それを使用して交差検証 MSE を評価できる。

ここでは、変数増加法で最後までパスをフィットするものとして、探索戦略を定義しよう。`sklearn_selection_path()`のパラメータの選択肢はいくつかありうるが、ここではデフォルトを使用する。デフォルトでは、各ステップで RSS の最大減少に基づいてモデルを選択する。

In [14]:

```
strategy = Stepwise.fixed_steps(design,
                                len(design.terms),
                                direction='forward')
full_path = sklearn_selection_path(OLS, strategy)
```

これより`Hitters`データに対して変数増加法によって最後までパスをフィットさせ、フィットされた値を計算する。

In [15]:

```
full_path.fit(Hitters, Y)
Yhat_in = full_path.predict(Hitters)
Yhat_in.shape
```

Out[15]:

```
(263, 20)
```

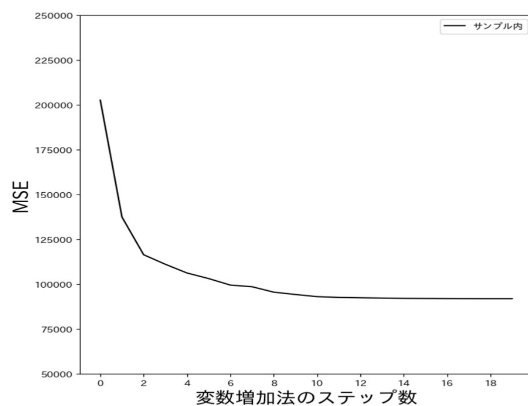
これにより、20 ステップ全体でのフィットされた値の配列が得られる。これには、ヌル(null)モデルの平均値も含まれており、サンプル内 MSE を評価するのに使用できる。期待通り、サンプル内 MSE はステップを進むごとに改善されており、モデルの選択には検証法または交差検証アプローチを使用する必要がある。後で交差検証と検証集合 MSE を比較するために、y 軸を 50,000 から 250,000 の範囲に固定しよう。これはリッジ回帰や Lasso、主成分回帰などの他の手法と同様である。

In [16]:

```
mse_fig, ax = subplots(figsize=(8,8))
insample_mse = ((Yhat_in - Y[:,None])**2).mean(0)
n_steps = insample_mse.shape[0]
ax.plot(np.arange(n_steps),
        insample_mse,
        'k', # 黒色
        label='サンプル内')
ax.set_ylabel('MSE', fontsize=20)
ax.set_xlabel('変数増加法のステップ数', fontsize=20)
ax.set_xticks(np.arange(n_steps)[::2])
ax.legend()
ax.set_ylim([50000,250000])
```

Out[16]:

```
(50000.0, 250000.0)
```



上記の `Y[:,None]` における `None` の表現に注目してみよう。これより、1 次元配列 `Y` に次元が追加され、2 次元配列 `Yhat_in` から引いたときにリサイクルされるようになる。

訳注 6：ただしリサイクルとは長さや大きさが異なるもので引き算などを行う際に長さや大きさが足りない分は要素が繰り返されることで、ここでは、263 × 20の `Yhat_in` に足りない分 `Y` が繰り返される。結果として、263 × 20の `Yhat_in` の各列から、長さが 263 のベクトル `Y` を引かれることになる。

これで、交差検証を使用してテストエラーをモデルのパスに沿って推定する準備が整った。ここでは変数選択を含む、モデルのフィットに関するすべての過程を訓練データのみを使用して行う必要がある。したがって、サイズを定めたもとでどのモデルが最良であるかどうかは、各訓練分割では訓練データのみを使用して決定する必要がある。この点は細かい点だが重要である。もし各ステップで最良部分集合を選ぶために全データセットを使用した場合、得られる検証集合誤差や交差検証誤差はテスト誤差の正確な推定値とはならない。

以下では、5 分割交差検証を用いて交差検証予測値を計算しよう。

In [17]:

```
K = 5
kfold = skm.KFold(K,
                  random_state=0,
                  shuffle=True)
Yhat_cv = skm.cross_val_predict(full_path,
                                Hitters,
                                Y,
                                cv=kfold)
Yhat_cv.shape
```

Out[17]:

```
(263, 20)
```

`skm.cross_val_predict()`を使用した場合、予測行列 `Yhat_cv` の形状は `Yhat_in` と同じだが、それぞれの行（サンプルインデックスに対応）は、その行を含まない訓練分割でフィットしたモデルからの予測である。

各モデルパスに沿って、各交差検証分割内での MSE を計算する。これを平均して平均 MSE を得るとともに、個別値を使用して平均の標準誤差の粗い推定値を計算

できる（ただし、5 つの誤差推定値は重複する訓練セットに基づくため独立していない）。したがって、各交差検証分割のテストインデックスを知る必要がある。これは、`kfold` の `split()`メソッドを使用して見つけることができる。上で `random_state` を固定したため、Y と行数が同じ任意の配列を分割するたびに、同じ訓練とテストのインデックスを得る事ができる。なお以下では訓練インデックスを無視する。

In [18]:

```
cv_mse = []
for train_idx, test_idx in kfold.split(Y):
    errors = (Yhat_cv[test_idx] - Y[test_idx,None])**2
    cv_mse.append(errors.mean(0)) # 列ごとの平均
cv_mse = np.array(cv_mse).T
cv_mse.shape
```

Out[18]:

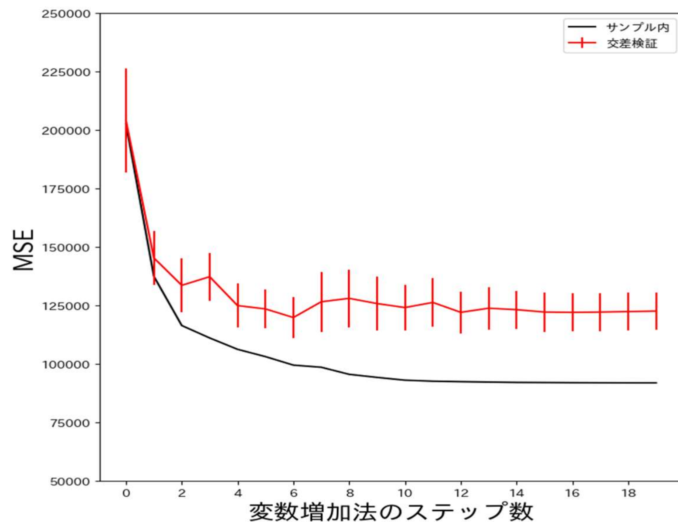
```
(20, 5)
```

交差検証誤差の推定値を MSE プロットに追加する。5 つの分割全体での平均誤差と、平均の標準誤差の推定値を含める。

In [19]:

```
ax.errorbar(np.arange(n_steps),
            cv_mse.mean(1),
            cv_mse.std(1) / np.sqrt(K),
            label='交差検証',
            c='r') # 赤色
ax.set_ylim([50000,250000])
ax.legend()
mse_fig
```

Out[19]:



検証集合アプローチを利用するには、`cv` を用いた部分を検証集合に変更するだけでよい。この場合、データをランダムに分割してテストセットと訓練セットを作成する。ここではテストサイズを 5 分割交差検証の各テストセットと同様に 20% に設定する。`skm.ShuffleSplit()`を用いてみよう。

In [20]:

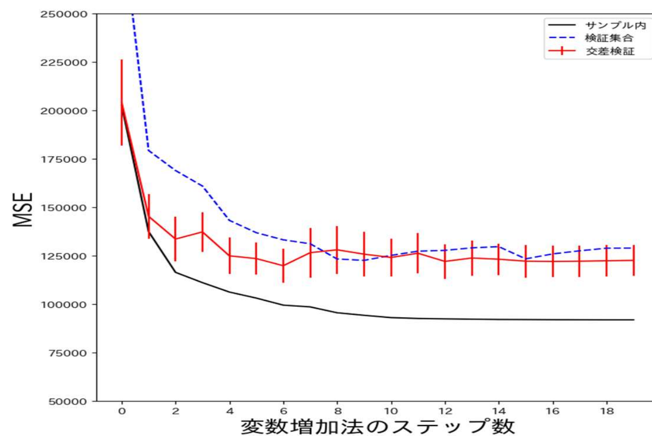
```
validation = skm.ShuffleSplit(n_splits=1,
                              test_size=0.2,
                              random_state=0)
for train_idx, test_idx in validation.split(Y):
    full_path.fit(Hitters.iloc[train_idx],
                  Y[train_idx])
    Yhat_val = full_path.predict(Hitters.iloc[test_idx])
    errors = (Yhat_val - Y[test_idx,None])**2
    validation_mse = errors.mean(0)
```

サンプル内 MSE の場合と同様に、検証集合アプローチでは標準誤差は計算できない。

In [21]:

```
ax.plot(np.arange(n_steps),
        validation_mse,
        'b--', # 青色、破線
        label='検証集合')
ax.set_xticks(np.arange(n_steps)[::2])
ax.set_ylim([50000,250000])
ax.legend()
mse_fig
```

Out[21]:



最良部分集合選択 (Best Subset Selection)

変数増加法は貪欲な選択手法である。各ステップで現在の集合に 1 つの追加変数を含める。`Hitters` データに最良部分集合選択を適用してみよう。これは、すべての部分集合のサイズに対して、最良の予測子集合を探索することで実現する。

`l0bnb` というパッケージを使用して、最良部分集合選択を行おう。このパッケージでは、部分集合を特定のサイズに制限する代わりに、サイズをペナルティとして使用して解のパスを生成する。この違いは細かい点だが、交差検証を行う際に結果に影響する。

In [22]:

```
D = design.fit_transform(Hitters)
D = D.drop('intercept', axis=1)
X = np.asarray(D)
```

ここでは、切片に対応する最初の列を除外したが、`l0bnb`は別に切片をフィットする。`fit_path()`関数を使用して、パスを見つけることができる。

In [23]:

```
path = fit_path(X,
                Y,
                max_nonzeros=X.shape[1])
```

Out[23]:

```
Preprocessing Data.
BnB Started.
Iteration: 1. Number of non-zeros:  1
Iteration: 2. Number of non-zeros:  2
Iteration: 3. Number of non-zeros:  2
Iteration: 4. Number of non-zeros:  2
Iteration: 5. Number of non-zeros:  3
Iteration: 6. Number of non-zeros:  3
Iteration: 7. Number of non-zeros:  4
Iteration: 8. Number of non-zeros:  9
Iteration: 9. Number of non-zeros:  9
Iteration: 10. Number of non-zeros:  9
Iteration: 11. Number of non-zeros:  9
Iteration: 12. Number of non-zeros:  9
Iteration: 13. Number of non-zeros:  9
Iteration: 14. Number of non-zeros:  9
Iteration: 15. Number of non-zeros:  9
Iteration: 16. Number of non-zeros:  9
Iteration: 17. Number of non-zeros:  9
Iteration: 18. Number of non-zeros:  17
Iteration: 19. Number of non-zeros:  19
```

`fit_path()`関数は、`B`としてフィットされた係数、`B0`として切片、および利用したパスアルゴリズムに関連する他のいくつかの属性を含むリストを返す。ただし、その詳細は本書の範囲を超えている。

In [24]:

```
path[3]
```

Out[24]:

```
{'B': array([0.        , 3.25484367, 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.67775265, 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        ]),
 'B0': -38.98216739555505,
 'lambda_0': 0.011416248027450187,
 'M': 0.5829861733382012,
 'Time_exceeded': False}
```

上記の例では、パスの第 4 ステップで、ペナルティパラメータ `lambda_0` の値が 0.114 であり、2 つの非ゼロ係数が`'B'`として得られている。検証集合でのこのようなフィットの列を、`lambda_0` の関数として予測に利用でき、また、交差検証による場合の結果を使うこともできる。

リッジ回帰と Lasso

`sklearn.linear_model`パッケージ（以下`skl`として略記する）を利用して、`Hitters`データにリッジ回帰と Lasso 正則化線形モデルをフィットしよう。前節の最良部分集合回帰で計算したモデル行列`X`（切片なし）を利用する。

リッジ回帰

リッジと Lasso をフィットするために、`skl.ElasticNet()`関数を用いる。リッジ回帰モデルのパスをフィットするには、`skl.ElasticNet.path()`関数を利用する。これはリッジおよび Lasso、または両方をフィットできる関数である。リッジ回帰は`l1_ratio=0`に対応する。変数が異なる単位で測定されている場合は、これらの適用の際には`X`の列を標準化すると良い。`skl.ElasticNet()`は正規化を行わないため、それを自分で行う必要がある。 なおデータを標準化した場合、もとのスケールで係数を得るためには、推定係数値を逆変換して戻す必要がある。 (6.5 式)および

(6.7 式)のパラメータλは、`sklearn`では`alphas`と呼ばれている。本章全体の一貫性を保つために、以下では`alphas`ではなく`lambdas`を使用する。

In [25]:

```
#訳注 7:
%%capture は訳者が追加
# 以下 Warning によって出力があまりに長くなる所は%%capture を使って出力を非表示にする
# もちろん%%capture を削除したり、コメントアウトしたりすれば出力が表示される
%%capture

Xs = X - X.mean(0)[None,:]
X_scale = X.std(0)
Xs = Xs / X_scale[None,:]
lambdas = 10**np.linspace(8, -2, 100) / Y.std()
soln_array = skl.ElasticNet.path(Xs,
                                 Y,
                                 l1_ratio=0.,
                                 alphas=lambdas)[1]
```

In [26]:

```
soln_array.shape
```

Out[26]:

```
(19, 100)
```

ここでは、正則化パスに沿った解に対応する係数の配列を抽出している。デフォルトでは、`skl.ElasticNet.path`メソッドは`l1_ratio=0`であるリッジ回帰の場合を除き、λの値の範囲を自動的に選択しフィットを行う。（これにはテクニカルな理由があり、リッジ以外では、すべての係数がゼロになるような最小のλの値を見つ

けることができるが、リッジ回帰の場合、この値は無限大になってしまうことによる。）

そこで、y の標準偏差でスケールされた$\lambda = 10^8$から $\lambda = 10^{-2}$の値のグリッド上で関数を実行する。これにより、切片のみを含むヌルモデルからフルモデルでの最小 2 乗フィットまで全範囲のシナリオを実質的にカバーすることができる。

各λの値ごとにリッジ回帰係数のベクトルが計算され、`soln_array` の各列を使用してアクセスできる。この場合、`soln_array` は 19×100 の行列で、19 行（予測子ごとに 1 行）と 100 列（λの値ごとに 1 列）がある。この行列を転置し、データフレームに変換して表示やプロットしやすくする。

In [27]:

```
soln_path = pd.DataFrame(soln_array.T,
                         columns=D.columns,
                         index=-np.log(lambdas))
soln_path.index.name = '-log(lambda)'
soln_path
```

Out[27]:

	AtBat	Hits	HmRun	Runs	RBI	Walks	Years	CAtBat	CHits
-log(lambda)									
-12.310855	0.000800	0.000889	0.000695	0.000851	0.000911	0.000900	0.000812	0.001067	0.001113
-12.078271	0.001010	0.001122	0.000878	0.001074	0.001150	0.001135	0.001025	0.001346	0.001404
-11.845686	0.001274	0.001416	0.001107	0.001355	0.001451	0.001433	0.001293	0.001698	0.001772
-11.613102	0.001608	0.001787	0.001397	0.001710	0.001831	0.001808	0.001632	0.002143	0.002236
-11.380518	0.002029	0.002255	0.001763	0.002158	0.002310	0.002281	0.002059	0.002704	0.002821
...	...	...	...	...	...	...	...	...	...
9.784658	-290.823989	336.929968	37.322686	-59.748520	-26.507086	134.855915	-17.216195	-387.775826	89.573601
10.017243	-290.879272	337.113713	37.431373	-59.916820	-26.606957	134.900549	-17.108041	-388.458404	89.000707
10.249827	-290.923382	337.260446	37.518064	-60.051166	-26.686604	134.936136	-17.022194	-388.997470	88.537380
10.482412	-290.958537	337.377455	37.587122	-60.158256	-26.750044	134.964477	-16.954081	-389.423414	88.164178
10.714996	-290.986528	337.470648	37.642077	-60.243522	-26.800522	134.987027	-16.900054	-389.760135	87.864551

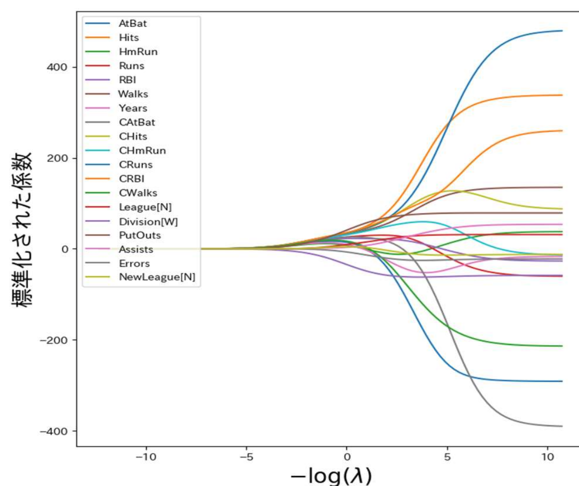
100 rows × 19 columns

CHmRun	CRuns	CRBI	CWalks	League[N]	Division[W]	PutOuts	Assists	Errors	NewLeague[N]
0.001064	0.001141	0.001149	0.000993	-0.000029	-0.000390	0.000609	0.000052	-0.000011	-0.000006
0.001343	0.001439	0.001450	0.001253	-0.000037	-0.000492	0.000769	0.000065	-0.000014	-0.000007
0.001694	0.001816	0.001830	0.001581	-0.000046	-0.000621	0.000970	0.000082	-0.000017	-0.000009
0.002138	0.002292	0.002309	0.001995	-0.000058	-0.000784	0.001224	0.000104	-0.000022	-0.000012
0.002698	0.002892	0.002914	0.002517	-0.000073	-0.000990	0.001544	0.000131	-0.000028	-0.000015
...	...	...	...	...	...	...	...	...	...
12.273926	476.079273	257.271255	-213.124780	31.258215	-58.457857	78.761266	53.622113	-22.208456	-12.402891
12.661459	477.031349	257.966790	-213.280891	31.256434	-58.448850	78.761240	53.645147	-22.198802	-12.391969
12.971603	477.791860	258.523025	-213.405740	31.254958	-58.441682	78.761230	53.663357	-22.191071	-12.383205
13.219329	478.398404	258.967059	-213.505412	31.253747	-58.435983	78.761230	53.677759	-22.184893	-12.376191
13.416889	478.881540	259.321007	-213.584869	31.252760	-58.431454	78.761235	53.689152	-22.179964	-12.370587

λによる係数の変化を可視化するために、正則化パスをプロットする。凡例の位置を制御するため、最初に`plot`メソッドでは`legend=False`を設定し、その後`ax`の`legend()`メソッドを利用して凡例を追加する。

In [28]:

```
path_fig, ax = subplots(figsize=(8,8))
soln_path.plot(ax=ax, legend=False)
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('標準化された係数', fontsize=20)
ax.legend(loc='upper left');
```



（横軸ラベルで `latex` フォーマットを使用して、ギリシャ文字のλを適切に表示させている。） 大きなλの値を使用すると、小さなλの値を使用する場合と比較して、係数の推定値がℓ_2ノルム（係数値の 2 乗の合計の平方根）で大幅に小さくなることが期待される。ここでは、$\lambda = 25.535$ の 40 番目のステップで係数を表示する。

In [29]:

```
beta_hat = soln_path.loc[soln_path.index[39]]
lambdas[39], beta_hat
```

Out[29]:

```
(25.53538897200662,
 AtBat          5.433750
 Hits           6.223582
 HmRun          4.585498
 Runs           5.880855
 RBI            6.195921
 Walks          6.277975
 Years          5.299767
 CAtBat         7.147501
 CHits          7.539495
 CHmRun         7.182344
 CRuns          7.728649
 CRBI           7.790702
 CWalks         6.592901
 League[N]      0.042445
 Division[W]   -3.107159
 PutOuts        4.605263
 Assists        0.378371
 Errors        -0.135196
 NewLeague[N]   0.150323
 Name: -3.240065292879872, dtype: float64)
```

標準化された係数のℓ_2ノルムを計算しよう。

In [30]:

```
np.linalg.norm(beta_hat)
```

Out[30]:

```
24.17061720144378
```

次に、$\lambda = 2.44\text{e-}01$ の場合の ℓ_2 ノルムを計算しよう。この小さなλ に基づく係数の ℓ_2 ノルムは、はるかに大きいことが分かる。

In [31]:

```
beta_hat = soln_path.loc[soln_path.index[59]]
lambdas[59], np.linalg.norm(beta_hat)
```

Out[31]:

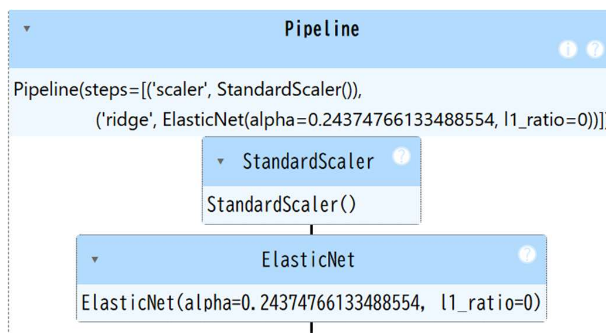
```
(0.24374766133488554, 160.42371017725912)
```

上記の例では、`X` を事前に標準化し、標準化されたデータ `Xs` を用いてリッジモデルをフィットした。`sklearn` の `Pipeline()` オブジェクトを使用することで、特徴量の標準化とリッジモデルのフィットを明確に分けることができる。

In [32]:

```
ridge = skl.ElasticNet(alpha=lambdas[59], l1_ratio=0)
scaler = StandardScaler(with_mean=True, with_std=True)
pipe = Pipeline(steps=[('scaler', scaler), ('ridge', ridge)])
pipe.fit(X, Y)
```

Out[32]:



パイプラインを使用した結果が、先に行った標準化データに対するフィットの結果と同じ ℓ_2 ノルムを持つことを確認しよう。

In [33]:

```
np.linalg.norm(ridge.coef_)
```

Out[33]:

```
160.42371017725904
```

ここでの注意点としては、上記の `pipe.fit(X, Y)` 操作は `ridge` オブジェクトを変更しており、例えば `coef_` などの属性が追加されていることである。

リッジ回帰のテスト誤差の推定

リッジ回帰の λ を事前に選択するのは困難であるため、検証法または交差検証を使用してチューニングパラメータを選択する必要がある。`Pipeline()` アプローチは、検証法（例えば `validation`）や k-分割交差検証に適用できる。

以下では、分割の `random_state` を固定して、再現性のある結果を得られるようにしよう。

In [34]:

```
validation = skm.ShuffleSplit(n_splits=1,
                              test_size=0.5,
                              random_state=0)
ridge.alpha = 0.01
results = skm.cross_validate(ridge,
                             X,
                             Y,
                             scoring='neg_mean_squared_error',
                             cv=validation)
-results['test_score']
```

Out[34]:

```
array([134214.00419204])
```

テスト MSE は 1.342e+05 であった。もし単に切片だけのモデルをフィットさせた場合、各テスト観測値をトレーニング観測値の平均で予測することになる。非常に大きなλでリッジ回帰モデルをフィットさせることで同じ結果を得ることができる。ここでの `1e10` は10^{10}を意味する。

In [35]:

```
ridge.alpha = 1e10
results = skm.cross_validate(ridge,
                             X,
                             Y,
                             scoring='neg_mean_squared_error',
                             cv=validation)
-results['test_score']
```

Out[35]:

```
array([231788.32155285])
```

$\lambda = 0.01$ を選ぶことは恣意的であるため、交差検証や検証集合アプローチを使用して最適なλ を選択しよう。 オブジェクト `GridSearchCV()` を使用すると、網羅的なグリッドサーチを行ってパラメータを選択できる。

まず、検証集合法を用いて λ を選択する。

In [36]:

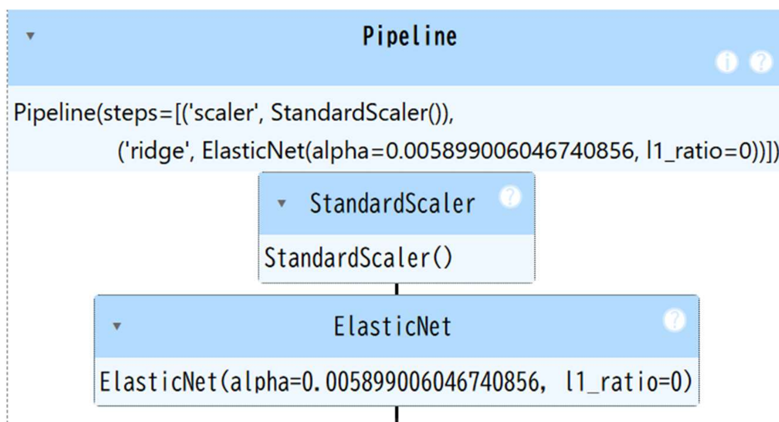
```
#訳注 8:%%capture は訳者追加、Warning で長くならないように先に使ったのと同様
%%capture

param_grid = {'ridge__alpha': lambdas}
```

```
grid = skm.GridSearchCV(pipe,
                        param_grid,
                        cv=validation,
                        scoring='neg_mean_squared_error')
grid.fit(X, Y)
grid.best_params_['ridge__alpha']
```

In [37]:

```
grid.best_estimator_
```

Out[37]:



代わりに、5 分割交差検証を利用することもできる。

In [38]:

```
#訳注９：%%capture は訳者追加、Warning で長くならないように先に使ったのと同様
%%capture

grid = skm.GridSearchCV(pipe,
                        param_grid,
                        cv=kfold,
```

```
                              scoring='neg_mean_squared_error')
grid.fit(X, Y)
```

In [39]:

```
grid.best_params_['ridge__alpha']
```

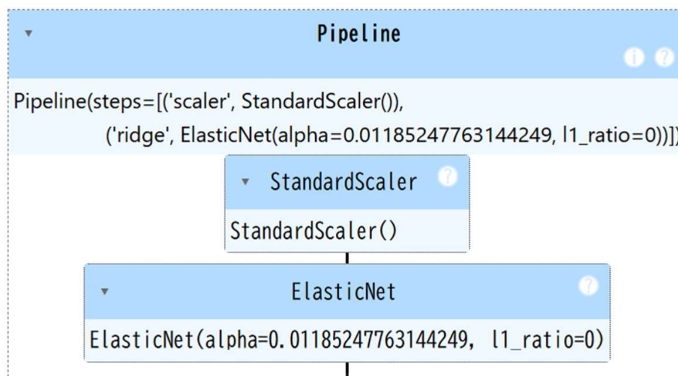
Out[39]:

```
0.01185247763144249
```

In [40]:

```
grid.best_estimator_
```

Out[40]:

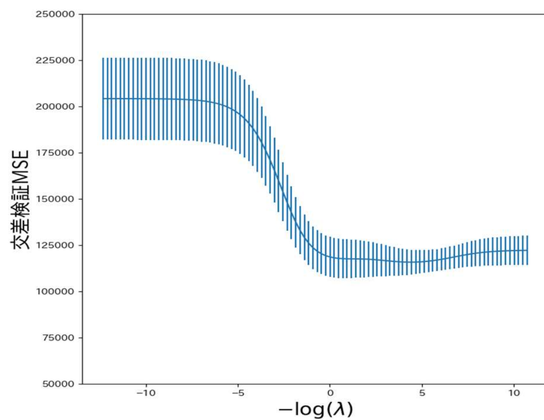


5 分割交差検証用に `kfold` オブジェクトを 271 ページで設定した。$-\log(\lambda)$の関数として交差検証 MSE をプロットすると、左から右に向かって縮小していく。

In [41]:

```
ridge_fig, ax = subplots(figsize=(8,8))
ax.errorbar(-np.log(lambdas),
            -grid.cv_results_['mean_test_score'],
```

```
            yerr=grid.cv_results_['std_test_score'] / np.sqrt(K))
ax.set_ylim([50000,250000])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('交差検証 MSE', fontsize=20);
```



異なる評価指標を使用してパラメータを選択することも可能である。`skl.ElasticNet()` のデフォルト指標はテスト R^2 である。ここでは交差検証で R^2 と MSE を使用して比較する。

In [42]:

```
#訳注 10:%%capture は訳者追加、Warning で長くならないように先に使ったのと同様
%%capture

grid_r2 = skm.GridSearchCV(pipe,
                           param_grid,
                           cv=kfold)
grid_r2.fit(X, Y)
```

In [43]:

```
grid_r2
```

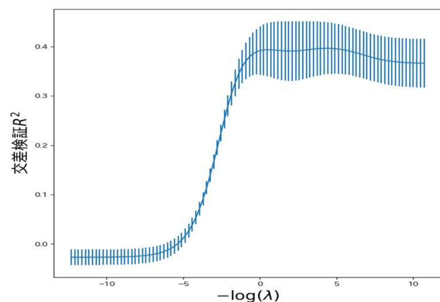
Out[43]:

```
GridSearchCV

GridSearchCV(cv=KFold(n_splits=5, random_state=0, shuffle=True),
         estimator=Pipeline(steps=[('scaler', StandardScaler()),
                                   ('ridge',
                                    ElasticNet(alpha=10000000000.0,
                                               l1_ratio=0))]),
         param_grid={'ridge__alpha': array([2.22093791e+05, 1.76005531e+05, 1.39481373e+05, 1.10536603e+05,
    8.75983676e+04, 6.94202082e+04, 5.50143278e+04, 4.35979140e+04,
    3.45506012e+04, 2.73807606...
    4.67486141e-03, 3.70474772e-03, 2.93594921e-03, 2.32668954e-03,
    1.84386167e-03, 1.46122884e-03, 1.15799887e-03, 9.17694298e-04,
    7.27257037e-04, 5.76338765e-04, 4.56738615e-04, 3.61957541e-04,
    2.86845161e-04, 2.27319885e-04, 1.80147121e-04, 1.42763513e-04,
    1.13137642e-04, 8.96596467e-05, 7.10537367e-05, 5.63088712e-05,
    4.46238174e-05, 3.53636122e-05, 2.80250579e-05, 2.22093791e-05])})

best_estimator_: Pipeline

Pipeline(steps=[('scaler', StandardScaler()),
                ('ridge', ElasticNet(alpha=0.011852477631442 49, l1_ratio=0))])

StandardScaler

StandardScaler()

ElasticNet

ElasticNet(alpha=0.01185247763144249, l1_ratio=0)
```

最後に、交差検証 R^2 をプロットする。

In [44]:

```
r2_fig, ax = subplots(figsize=(8,8))
ax.errorbar(-np.log(lambdas),
            grid_r2.cv_results_['mean_test_score'],
            yerr=grid_r2.cv_results_['std_test_score'] / np.sqrt(K))
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('交差検証$R^2$', fontsize=20);
```



解のパスに対する高速交差検証

リッジ回帰、Lasso、エラスティックネット(Elastic Net)は、λ値の列に沿って効率的にフィットし、解のパスまたは正則化パスと呼ばれるものを生成する。そのため、このようなパスをフィットさせ、交差検証を使用して適切なλを選択するための専用コードがある。同一の分割を用いても、以下の`pipeCV`と先述の`grid`の結果が*完全には*一致しないのは、`grid`では各分割で特徴量が標準化される一方で、`pipeCV`では一度だけ標準化が行われるためである。それにもかかわらず、標準化が分割間で比較的安定しているため、結果は類似している。

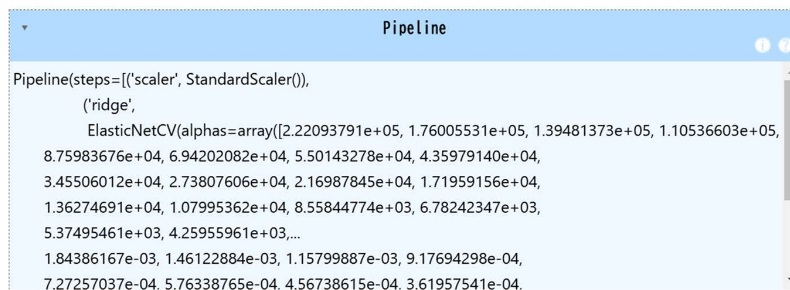
In [45]:

```
#訳注10:%%capture は訳者追加、Warning で長くならないように先に使ったのと同様

%%capture


ridgeCV = skl.ElasticNetCV(alphas=lambdas,
                           l1_ratio=0,
                           cv=kfold)
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('ridge', ridgeCV)])
pipeCV.fit(X, Y)
```

In [46]:

```
pipeCV
```

Out[46]:

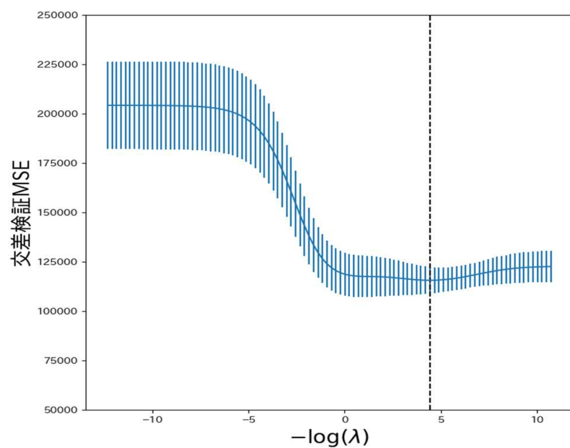


```
2.86845161e-04, 2.27319885e-04, 1.80147121e-04, 1.42763513e-04,
1.13137642e-04, 8.96596467e-05, 7.10537367e-05, 5.63088712e-05,
4.46238174e-05, 3.53636122e-05, 2.80250579e-05, 2.22093791e-05]),
             cv=KFold(n_splits=5, random_state=0, shuffle=True),
             l1_ratio=0))])

StandardScaler
StandardScaler()

ElasticNetCV
ElasticNetCV(alphas=array([2.22093791e+05, 1.76005531e+05, 1.39481373e+05, 1.10536603e+05,
       8.75983676e+04, 6.94202082e+04, 5.50143278e+04, 4.35979140e+04,
       3.45506012e+04, 2.73807606e+04, 2.16987845e+04, 1.71959156e+04,
       1.36274691e+04, 1.07995362e+04, 8.55844774e+03, 6.78242347e+03,
       5.37495461e+03, 4.25955961e+03, 3.37562814e+03, 2.67512757e+03,
       2.11999285e+03, 1.680058...
       1.84386167e-03, 1.46122884e-03, 1.15799887e-03, 9.17694298e-04,
       7.27257037e-04, 5.76338765e-04, 4.56738615e-04, 3.61957541e-04,
       2.86845161e-04, 2.27319885e-04, 1.80147121e-04, 1.42763513e-04,
       1.13137642e-04, 8.96596467e-05, 7.10537367e-05, 5.63088712e-05,
       4.46238174e-05, 3.53636122e-05, 2.80250579e-05, 2.22093791e-05]),
             cv=KFold(n_splits=5, random_state=0, shuffle=True), l1_ratio=0)
```

もう一度交差検証誤差をプロットして、`skm.GridSearchCV` を使用した場合と結果が類似することを確認しよう。

In [47]:

```
tuned_ridge = pipeCV.named_steps['ridge']
ridgeCV_fig, ax = subplots(figsize=(8,8))
ax.errorbar(-np.log(lambdas),
            tuned_ridge.mse_path_.mean(1),
            yerr=tuned_ridge.mse_path_.std(1) / np.sqrt(K))
ax.axvline(-np.log(tuned_ridge.alpha_), c='k', ls='--')
ax.set_ylim([50000,250000])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('交差検証 MSE', fontsize=20);
```



ここで、最小の交差検証誤差をもたらすλの値は 1.19e-02 であり、これは `tuned_ridge.alpha_` として得られる。このλに基づくテスト MSE はどれくらいだろうか？

In [48]:

```
np.min(tuned_ridge.mse_path_.mean(1))
```

Out[48]:

```
115526.70630987729
```

つまり$\lambda = 4$ を使用した場合に得られたテスト MSE よりもさらに改善されている。最後に、`tuned_ridge.coef_` には、このλ値で全データセットにフィットした係数が格納されている。

In [49]:

```
tuned_ridge.coef_
```

Out[49]:

```
array([-222.80877051,  238.77246614,    3.21103754,   -2.93050845,
          3.64888723,  108.90953869,  -50.81896152, -105.15731984,
```

```
       122.00714801,   57.1859509 ,  210.35170348,  118.05683748,
      -150.21959435,   30.36634231,  -61.62459095,   77.73832472,
        40.07350744,  -25.02151514,  -13.68429544])
```

予想通り、どの係数もゼロではない—リッジ回帰は変数選択を行わないから当然である！

交差検証によるリッジ回帰のテスト誤差の評価

交差検証を使用して λ を選択することで、3 章 で見たような線形回帰モデルをフィットした場合のように、単一の回帰推定量が得られる。したがって、そのテスト誤差を推定することは理にかなっていると言える。ここで問題になるのは、交差検証が λ を選択する際にすべてのデータを利用してしまっているため、テスト誤差を推定するためのデータが残っていないことである。妥協案として、データを初めに 2 つの非重複集合（訓練セットとテストセット）に分け、その訓練セットで交差検証を行い、テストセットでそのパフォーマンスを評価することが考えられる。ここではこの方法を検証集合が入れ子に定まった交差検証(cross-validation nested with the validation set approach)と呼ぶことにする。特に前提がなければ、検証にデータを半分使う理由はない。以下では、75%を訓練に、25%をテストに使用し、推定量は 5 分割交差検証で調整されたリッジ回帰を使用する。これは次のようなコードで実行できる：

In [50]:

```
outer_valid = skm.ShuffleSplit(n_splits=1,
                               test_size=0.25,
                               random_state=1)
inner_cv = skm.KFold(n_splits=5,
                     shuffle=True,
                     random_state=2)
ridgeCV = skl.ElasticNetCV(alphas=lambdas,
                           l1_ratio=0,
                           cv=inner_cv)
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('ridge', ridgeCV)]);
```

In [51]:

```
#訳注 11:%%capture は訳者追加、Warning で長くならないように先に使ったのと同様

%%capture

results = skm.cross_validate(pipeCV,
                             X,
                             Y,
                             cv=outer_valid,
                             scoring='neg_mean_squared_error')
-results['test_score']
```

Lasso 回帰

リッジ回帰はλの良い選択をすれば、最小 2 乗法やヌル(null)モデルよりも `Hitters` データセットで優れたパフォーマンスを発揮することが分かった。次に、Lasso 回帰がリッジ回帰よりも精度が高いか、あるいは解釈しやすいモデルを提供できるかどうかを検討する。Lasso モデルをフィットさせるためには、再び `ElasticNetCV()` 関数を利用するが、今回は `l1_ratio=1` を引数として指定する。このこと以外はリッジ回帰をフィットさせる場合と同じように進める。

In [52]:

```
lassoCV = skl.ElasticNetCV(n_alphas=100,
                           l1_ratio=1,
                           cv=kfold)
pipeCV = Pipeline(steps=[('scaler', scaler),
                         ('lasso', lassoCV)])
pipeCV.fit(X, Y)
tuned_lasso = pipeCV.named_steps['lasso']
tuned_lasso.alpha_
```

Out[52]:

```
3.1472370031649866
```

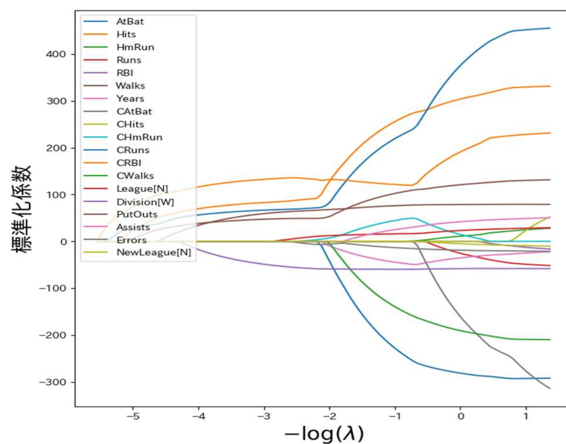
In [53]:

```
lambdas, soln_array = skl.Lasso.path(Xs,
                                     Y,
                                     l1_ratio=1,
                                     n_alphas=100)[:2]
soln_path = pd.DataFrame(soln_array.T,
                         columns=D.columns,
                         index=-np.log(lambdas))
```

標準化された係数のプロットから、チューニングパラメータの選択に応じて、いくつかの係数がゼロになることが分かる。

In [54]:

```
path_fig, ax = subplots(figsize=(8,8))
soln_path.plot(ax=ax, legend=False)
ax.legend(loc='upper left')
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('標準化係数', fontsize=20);
```



最小の交差検証誤差は、ヌル(null)モデルおよび最小二乗法のテストセット MSE よりも低く、リッジ回帰で交差検証によって選択された λ のテスト MSE（278 ページの 115526.71）と類似している。

In [55]:

```
np.min(tuned_lasso.mse_path_.mean(1))
```

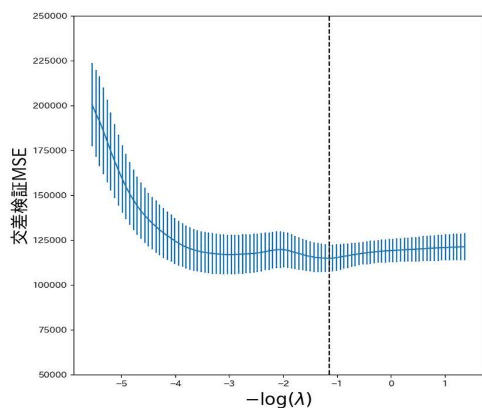
Out[55]:

```
114690.73118253677
```

再度、交差検証誤差のプロットを作成する。

In [56]:

```
lassoCV_fig, ax = subplots(figsize=(8,8))
ax.errorbar(-np.log(tuned_lasso.alphas_),
            tuned_lasso.mse_path_.mean(1),
            yerr=tuned_lasso.mse_path_.std(1) / np.sqrt(K))
ax.axvline(-np.log(tuned_lasso.alpha_), c='k', ls='--')
ax.set_ylim([50000,250000])
ax.set_xlabel('$-\log(\lambda)$', fontsize=20)
ax.set_ylabel('交差検証 MSE', fontsize=20);
```



ここで Lasso はリッジ回帰に対して大きな利点があることを指摘しておく。結果として得られる係数推定値がスパース（疎）であることである。ここでは、19 の係数推定値のうち 6 つがゼロであることが分かる。したがって、交差検証で選択された λ を使用した Lasso モデルは 13 の変数だけを含んでいる。

In [57]:

```
tuned_lasso.coef_
```

Out[57]:

```
array([-210.01008773,  243.4550306 ,    0.        ,    0.        ,
          0.        ,   97.69397357,  -41.52283116,   -0.        ,
          0.        ,   39.62298193,  205.75273856,  124.55456561,
       -126.29986768,   15.70262427,  -59.50157967,   75.24590036,
         21.62698014,  -12.04423675,   -0.        ])
```

リッジ回帰と同様に、交差検証による Lasso のテスト誤差を評価するために、まず訓練セットとテストセットに分割し、訓練セット内で交差検証を実行することができる。ただしこれは演習として残しておこう。

PCR と PLS 回帰

主成分回帰 (Principal Components Regression, PCR)

主成分回帰 (PCR) は、`sklearn.decomposition` モジュールの `PCA()` を使用して実行できる。ここでは、`Hitters` データを用いて `Salary` を予測するために PCR を適用する。データから欠損値が除去されていることを確認してみよう（6.5.1 節も参照されたい）。

ここでは、回帰モデルのフィットには `LinearRegression()` を利用する。ただしデフォルトで切片をフィットさせる点が、6.5.1 節で見た `OLS()` 関数とは異なる。

In [58]:

```
pca = PCA(n_components=2)
linreg = skl.LinearRegression()
pipe = Pipeline([('pca', pca),
                 ('linreg', linreg)])
pipe.fit(X, Y)
pipe.named_steps['linreg'].coef_
```

Out[58]:

```
array([0.09846131, 0.4758765 ])
```

PCA を実行する際、データが*標準化*されているかどうかによって結果が異なる。前述の例と同様に、これをパイプラインに追加のステップを含めることで実現できる。

In [59]:

```
pipe = Pipeline([('scaler', scaler),
                 ('pca', pca),
                 ('linreg', linreg)])
pipe.fit(X, Y)
pipe.named_steps['linreg'].coef_
```

Out[59]:

```
array([106.36859204,  21.60350456])
```

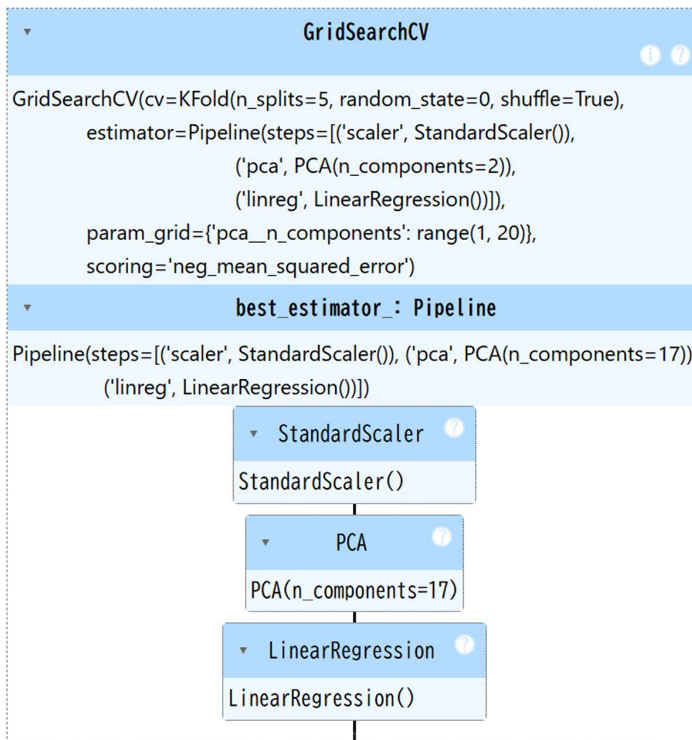
むろん交差検証 (CV) を利用して主成分の数を選ぶこともできる。`skm.GridSearCV` を利用し、`n_components` パラメータを変化させるように設定すればよい。

In [60]:

```
param_grid = {'pca__n_components': range(1, 20)}
grid = skm.GridSearchCV(pipe,
```

```
                                param_grid,
                                cv=kfold,
                                scoring='neg_mean_squared_error')
grid.fit(X, Y)
```

Out[60]:

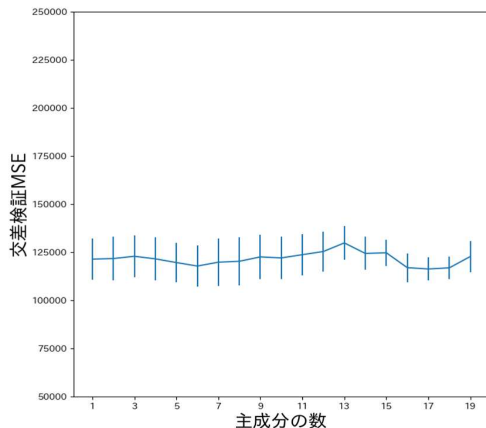


他の方法と同様に結果をプロットしてみよう。

In [61]:

```
pcr_fig, ax = subplots(figsize=(8,8))
n_comp = param_grid['pca__n_components']
ax.errorbar(n_comp,
            -grid.cv_results_['mean_test_score'],
            grid.cv_results_['std_test_score'] / np.sqrt(K))
ax.set_ylabel('交差検証 MSE', fontsize=20)
```

```
ax.set_xlabel('主成分の数', fontsize=20)
ax.set_xticks(n_comp[::2])
ax.set_ylim([50000,250000]);
```



最小の交差検証誤差は、17 個の主成分を使用したときに達成することが分かる。ただし、プロットからも分かるように、モデルに 1 つの主成分だけを含めた場合でも交差検証誤差はほぼ同じであることが分かる。このことから少数の主成分だけで十分なモデルが得られる可能性があることが示唆される。

CV スコアは、1 から 19 までの各主成分数に対して与えられる。`PCA()` メソッドは `n_components=0` で切片のみをフィットしようとするとエラーが生じる。そこでこの分割でヌル(null)モデルの MSE も計算する必要がある。

In [62]:

```
Xn = np.zeros((X.shape[0], 1))
cv_null = skm.cross_validate(linreg,
                             Xn,
                             Y,
                             cv=kfold,
                             scoring='neg_mean_squared_error')
-cv_null['test_score'].mean()
```

Out[62]:

```
204139.30692994667
```

`PCA` オブジェクトの `explained_variance_ratio_` 属性は、異なる数の主成分を使用した場合の、予測変数および応答に関する*説明された分散の割合*を与える。なおこの点については、12.2 章で詳しく説明されている。

In [63]:

```
pipe.named_steps['pca'].explained_variance_ratio_
```

Out[63]:

```
array([0.3831424 , 0.21841076])
```

ここで簡単に言えば、 M 個の主成分を使用して予測変数に関する情報をどれだけ捉えられるかを示している。例えば、M=1 では 38.31%の分散を捉え、M=2 ではさらに 21.84%を追加され、合計で 60.15%の分散を捉える。M=6 では 88.63%にも達する。それ以降は増分が減少し、最終的に全ての主成分 M=p=19 を使用すると 100%の分散が捉えられる。

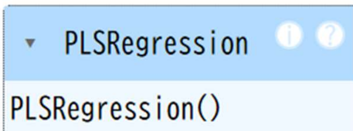
部分最小二乗法 (Partial Least Squares, PLS)

部分最小二乗法 (PLS) は、`PLSRegression()` 関数で実装されている。

In [64]:

```
pls = PLSRegression(n_components=2,
                    scale=True)
pls.fit(X, Y)
```

Out[64]:

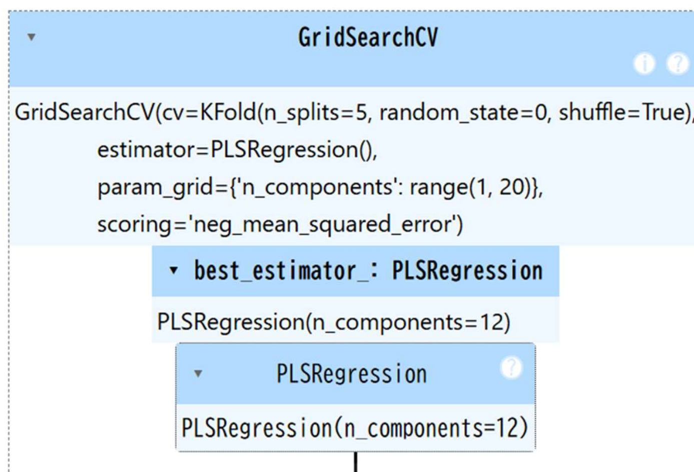


PCR と同様に、主成分の数を選ぶために交差検証 (CV) を利用する。

In [65]:

```
param_grid = {'n_components':range(1, 20)}
grid = skm.GridSearchCV(pls,
                          param_grid,
                        cv=kfold,
                        scoring='neg_mean_squared_error')
grid.fit(X, Y)
```

Out[65]:

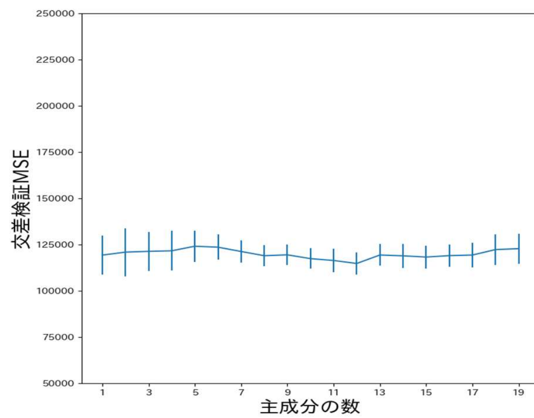


他の方法と同様に、MSE をプロットする。

In [66]:

```
pls_fig, ax = subplots(figsize=(8,8))
n_comp = param_grid['n_components']
ax.errorbar(n_comp,
            -grid.cv_results_['mean_test_score'],
            grid.cv_results_['std_test_score'] / np.sqrt(K))
ax.set_ylabel('交差検証 MSE', fontsize=20)
```

```
ax.set_xlabel('主成分の数', fontsize=20)

ax.set_xticks(n_comp[::2])
ax.set_ylim([50000,250000]);
```



交差検証誤差が最小となるのは、主成分が 12 個の場合であるが、少ない主成分数（2 や 3）との間には目立った差は見られない。

ISLP 第 7 章 ラボ：非線形モデル

このラボでは、この章で議論された非線形モデルによる分析の一部分を実際にデモしてみる。`Wage` データを使用して、多くの複雑な非線形フィッティングの手順が容易に `Python` で実装できることを示す。

通常通り、標準的なもののインポートから始めよう。

In [1]:

```
%%capture
!pip install ISLP

import numpy as np, pandas as pd
from matplotlib.pyplot import subplots
import statsmodels.api as sm
from ISLP import load_data
from ISLP.models import (summarize,
                         poly,
                         ModelSpec as MS)
from statsmodels.stats.anova import anova_lm
```

このラボで必要な新しいインポートを再度行う。多くは `ISLP` パッケージのために開発されたものである。

In [2]:

```
from pygam import (s as s_gam,
                   l as l_gam,
                   f as f_gam,
                   LinearGAM,
                   LogisticGAM)
from ISLP.transforms import (BSpline,
                             NaturalSpline)
```

```
from ISLP.models import bs, ns
from ISLP.pygam import (approx_lam,
                        degrees_of_freedom,
                        plot as plot_gam,
                        anova as anova_gam)
```

多項式回帰と階段関数

まず、Figure 7.1 の再現方法を示す。始めにデータをロードする。

In [3]:

```
Wage = load_data('Wage')
y = Wage['wage']
age = Wage['age']
```

このラボでの応答変数は `Wage['wage']`であり、これを上記の `y` に保存してある。3.6.6 節と同様に、`poly()`関数を使って 4 次の多項式を `age` にフィットするモデル行列を作成する。

In [4]:

```
poly_age = MS([poly('age', degree=4)]).fit(Wage)
M = sm.OLS(y, poly_age.transform(Wage)).fit()
summarize(M)
```

Out[4]:

	coef	std err	T	P>|t|
Intercept	111.7036	0.729	153.283	0.000
poly(age, degree=4)[0]	447.0679	39.915	11.201	0.000
poly(age, degree=4)[1]	-478.3158	39.915	-11.983	0.000
poly(age, degree=4)[2]	125.5217	39.915	3.145	0.002
poly(age, degree=4)[3]	-77.9112	39.915	-1.952	0.051

この多項式は、`poly()`関数を使用して構築される。この関数は、新しいデータポイントでの多項式の評価を簡単に実行できる特別な*変換器* `Poly()` を作成する。

（6.5.3 節にある `PCA()`などの特徴変換に関連する `sklearn` の用語を利用する）。`poly()` はヘルパー関数と呼ばれるものであり、そのような変換を設定するが、実際の計算は `Poly()` が行う。詳細については、書籍 page 129 の変換の議論を参照のこと。

上記のコードでは、初めに `fit()` メソッドを使用してデータフレーム `Wage` を再度評価し、必要なパラメーターを計算して保存する。これらのパラメーターは、以降の全ての `transform()` メソッドの評価で使用する。例えば、2 行目や下記で開発するプロット関数でも使用する。予測を行いたい `age`（年齢）に対する値のグリッドを作成しておく。

In [5]:

```
age_grid = np.linspace(age.min(),
                       age.max(),
                       100)
age_df = pd.DataFrame({'age': age_grid})
```

最後に、データをプロットし、4 次の多項式でのフィッティングを追加することを考えよう。以下ではいくつかの類似したプロットを作成するため、まず、すべての必要な要素を作成してプロットを生成する関数を記述する。この関数はモデルの仕様（ここでは変換によって指定された基底）と `age` の値のグリッドを入力として受け取る。関数はフィッティング曲線と 95%信頼区間を生成する。`basis` 引数を使用することで、異なる変換を使用して結果を生成し、プロットすることができ、後に示すスプラインなどの変換も可能になる。

In [6]:

```
def plot_wage_fit(age_df,
                  basis,
                  title):

    X = basis.transform(Wage)
    Xnew = basis.transform(age_df)
    M = sm.OLS(y, X).fit()
    preds = M.get_prediction(Xnew)
    bands = preds.conf_int(alpha=0.05)
```

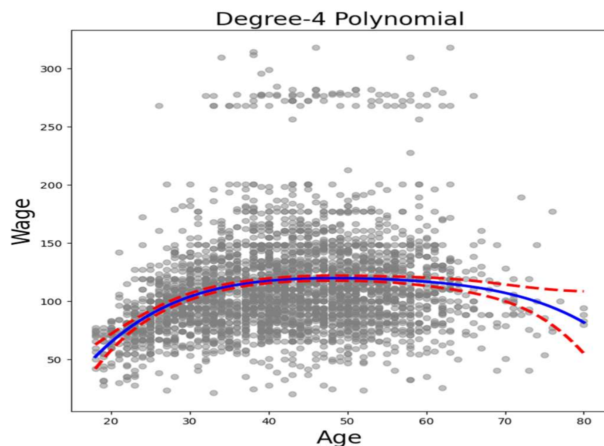
```
    fig, ax = subplots(figsize=(8,8))
    ax.scatter(age,
               y,
               facecolor='gray',
               alpha=0.5)
    for val, ls in zip([preds.predicted_mean,
                      bands[:,0],
                      bands[:,1]],
                     ['b','r--','r--']):
        ax.plot(age_df.values, val, ls, linewidth=3)
    ax.set_title(title, fontsize=20)
    ax.set_xlabel('Age', fontsize=20)
    ax.set_ylabel('Wage', fontsize=20);
    return ax
```

`ax.scatter()`には `alpha` という引数を含めており、ポイントに透明度を追加している。これにより、密度の視覚的な表示が可能になる。上記の `for` ループでは `zip()` 関数を使用していることに注目してみよう（2.3.8 節を参照）。プロットすべき 3 つのラインがあり、それぞれ異なる色と線の種類を使用している。ここで `zip()`を使用することで、これらをループ内で便利にまとめてイテレータとして扱うことができる。（`Python` の用語で、「イテレータ」とはループのように反復可能な有限個数のオブジェクトを意味する。）

この関数を使用して 4 次多項式のフィットをプロットしてみる。

In [7]:

```
plot_wage_fit(age_df,
              poly_age,
              'Degree-4 Polynomial');
```



多項式回帰では、あらかじめ使用する多項式の次数を決定する必要がある。時には直感的に 2 次や 3 次多項式を選んで非線形フィットを得ることもあるが、より体系的な方法でこの決定を行うことも可能である。その一つの方法として仮説検定を用いる方法があり、ここではその方法を示しておく。一次（線形）から五次多項式までの一連のモデルをフィットし、`wage`（賃金）と `age`（年齢）との関係を説明するのに十分な最もシンプルなモデルを見つけようとしよう。ここで ANOVA（分散分析）検定を実行する `anova_lm()`関数を利用する。

分散分析（ANOVA）では、モデル M_1 がデータを説明するのに十分であるという帰無仮説に対して、より複雑なモデル M_2 が必要であるという対立仮説を検定する。この判断は F 検定に基づいて行われる。検定を行うには、モデル M_1 と M_2 が入れ子になっている（*ネストされている*）必要がある。すなわち、M_1 の予測変数が張る空間が M_2 の予測変数が張る空間の部分空間でなければばならない。

訳注 1：”入れ子”とは M_1 に含まれるすべての説明変数が M_2 にも含まれる必要があることを意味する。

この場合、5 つの異なる多項式モデルをフィットし、より単純なモデルとより複雑なモデルを順に比較する。

In [8]:

```
models = [MS([poly('age', degree=d)])
          for d in range(1, 6)]
Xs = [model.fit_transform(Wage) for model in models]
anova_lm(*[sm.OLS(y, X_).fit()
           for X_ in Xs])
```

Out[8]:

	df_resid	ssr	df_diff	ss_diff	F	Pr(>F)
0	2998.0	5.022216e+06	0.0	NaN	NaN	NaN
1	2997.0	4.793430e+06	1.0	228786.010128	143.593107	2.363850e-32
2	2996.0	4.777674e+06	1.0	15755.693664	9.888756	1.679202e-03
3	2995.0	4.771604e+06	1.0	6070.152124	3.809813	5.104620e-02
4	2994.0	4.770322e+06	1.0	1282.563017	0.804976	3.696820e-01

上記の`anova_lm()`の行にある`*`に注意しておこう。この関数はキーワード引数ではなく、任意の数の引数（ここではフィットされたモデル）を取る。ここで行われているように、フィットされたモデルがリストとして提供される場合、そのリストの前に`*`を付ける必要がある。

線形モデル`models[0]`と二次モデル`models[1]`を比較する p 値は実質的にゼロであり、線形フィットでは十分でないことを示している。（ここで、0 から始まるインデックスは、次数を表す多項式の例としては混乱を招く可能性があるため、`models[1]`が線形ではなく二次であることに注意しておこう！）同様に、二次モデル`models[1]`と三次モデル`models[2]`を比較する p 値も非常に低く（0.0017）、二次フィットも不十分であることが分かる。三次モデルと四次モデル`models[2]`と`models[3]`を比較する p 値は約 5%であり、五次多項式`models[4]`は p 値が 0.37 のため、不要であると見なせる。したがって、三次または四次多項式がデータに対して合理的なフィットを提供すると考えられるが、次数がそれ以下またはそれ以上のモデルは正当化されないだろう。

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ことを利用して、これらの p 値をより簡潔に得ることも可能である。

In [9]:

```
summarize(M)
```

Out[9]:

	Coef	std err	t	P>|t|
intercept	111.7036	0.729	153.283	0.000
poly(age, degree=4)[0]	447.0679	39.915	11.201	0.000
poly(age, degree=4)[1]	-478.3158	39.915	-11.983	0.000
poly(age, degree=4)[2]	125.5217	39.915	3.145	0.002
poly(age, degree=4)[3]	-77.9112	39.915	-1.952	0.051

p 値が同一であることに注目してみよう。実際、t 統計量の 2 乗は`anova_lm()`関数の F 統計量と等しい。これは次のように計算して確認できる。

In [10]:

```
(-11.983)**2
```

Out[10]:

```
143.59228900000002
```

ただし、モデルがネストされている限り、直交多項式を使用したかどうかに関係なく ANOVA の手法は機能する。例えば、以下のように、`education`（教育）に線形項があり、`age`（年齢）に異なる次数の多項式が含まれている 3 つのモデルを`anova_lm()`を使って比較することができる。

In [11]:

```
models = [MS(['education', poly('age', degree=d)])
          for d in range(1, 4)]
XEs = [model.fit_transform(Wage)
       for model in models]
anova_lm(*[sm.OLS(y, X_).fit() for X_ in XEs])
```

Out[11]:

	df_resid	ssr	df_diff	ss_diff	F	Pr(>F)
0	2997.0	3.902335e+06	0.0	NaN	NaN	NaN
1	2996.0	3.759472e+06	1.0	142862.701185	113.991883	3.838075e-26
2	2995.0	3.753546e+06	1.0	5926.207070	4.728593	2.974318e-02

仮説検定や ANOVA を使用する代わりに、5 章で述べるように交差検証を用いて多項式の次数を選択することも可能である。

次に、個人が年間$250,000 以上を稼ぐか否かを予測することを考えよう。以前とほぼ同様の手順で進め、まず適切な応答ベクトルを作成し、その後、二項分布ファミリ

訳注 2：応答変数が二項分布に従うように設定するという意味である。

を用いて`glm()`関数を適用し、多項式ロジスティック回帰モデルをフィットしすればよい。

In [12]:

```
X = poly_age.transform(Wage)
high_earn = Wage['high_earn'] = y > 250 # shorthand
glm = sm.GLM(y > 250,
             X,
             family=sm.families.Binomial())
B = glm.fit()
summarize(B)
```

Out[12]:

	coef	std err	z	P>|z|
Intercept	-4.3012	0.345	-12.457	0.000
poly(age, degree=4)[0]	71.9642	26.133	2.754	0.006
poly(age, degree=4)[1]	-85.7729	35.929	-2.387	0.017
poly(age, degree=4)[2]	34.1626	19.697	1.734	0.083
poly(age, degree=4)[3]	-47.4008	24.105	-1.966	0.049

再び、`get_prediction()`メソッドを利用して予測を行う。

In [13]:

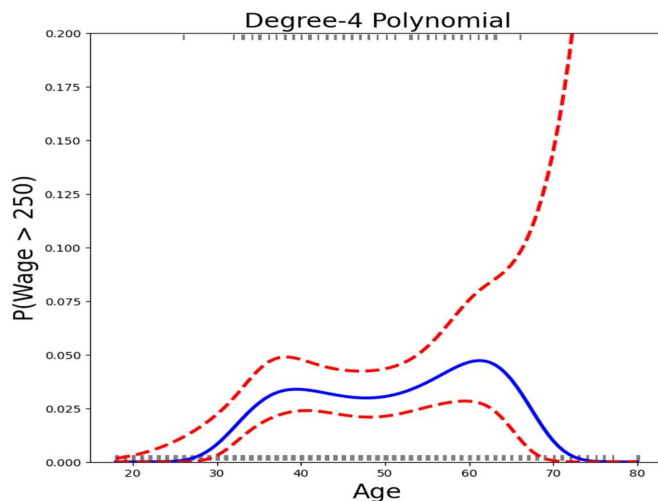
```
newX = poly_age.transform(age_df)
preds = B.get_prediction(newX)
bands = preds.conf_int(alpha=0.05)
```

次に、推定された関係をプロットする。

In [14]:

```
fig, ax = subplots(figsize=(8,8))
rng = np.random.default_rng(0)
ax.scatter(age +
           0.2 * rng.uniform(size=y.shape[0]),
           np.where(high_earn, 0.198, 0.002),
           fc='gray',
           marker='|')
```

```
for val, ls in zip([preds.predicted_mean,
                  bands[:,0],
                  bands[:,1]],
                 ['b','r--','r--']):
    ax.plot(age_df.values, val, ls, linewidth=3)
ax.set_title('Degree-4 Polynomial', fontsize=20)
ax.set_xlabel('Age', fontsize=20)
ax.set_ylim([0,0.2])
ax.set_ylabel('P(Wage > 250)', fontsize=20);
```



`wage` が 250 を超える観測値に対応する `age` の値はプロットの上部に灰色のマークとして描画され、`wage` が 250 未満の観測値はプロットの下部に灰色のマークとして表示されている。同じ `age` 値を持つ観測値が重ならないように、`age` 値に少量のノイズを追加してデータを分散している。この種のプロットは*ラグプロット*と呼ばれることがある。

ステップ関数をデータにフィットするには、7.2 節で説明したように、まず `pd.qcut()`関数を使用して、四分位数に基づいて `age` を離散化する。その後、`pd.get_dummies()`を使用して、このカテゴリ変数のモデル行列の列を作成する。この関数は通常の方法とは異なり、カテゴリごとにすべての列を含み、通常のアプローチとは違って 1 つの水準を省略しない。

訳注 3：通常、多重共線性を防ぐためにダミー変数を 1 つ落とすが、ここでは切片が入らないので、ダミーを落とさない。

In [15]:

```
cut_age = pd.qcut(age, 4)
summarize(sm.OLS(y, pd.get_dummies(cut_age)).fit())
```

Out[15]:

	coef	std err	t	P>|t|
(17.999, 33.75]	94.1584	1.478	63.692	0.0
(33.75, 42.0]	116.6608	1.470	79.385	0.0
(42.0, 51.0]	119.1887	1.416	84.147	0.0
(51.0, 80.0]	116.5717	1.559	74.751	0.0

ここで`pd.qcut()`は自動的に25%、50%、75%の分位数に基づいてカットポイントを選択し、4つの範囲が得られる。`4`という引数を使う代わりに、独自の分位数を指定することも可能である。分位数に基づかない区分けを行う場合には`pd.cut()`関数が利用できる。この場合には`pd.qcut()`（および`pd.cut()`）関数は順序付きのカテゴリ変数を返す。

回帰モデルは、この変数を回帰に使用するためのダミー変数のセットを生成する。このモデルでは`age`だけが変数であるため、94,158.40という値は33.75歳未満の人々の平均給与を表し、他の係数はそれぞれの年齢層における平均給与を示している。多項式フィットの場合と同様に、予測とプロットを生成することができる。

スプライン

回帰スプラインをフィットするには、`ISLP`パッケージの変換機能を利用する。実際のスプライン評価関数は`scipy.interpolate`パッケージにあり、それらを`Poly()`や`PCA()`と同様の形式でアップして変換機能として利用できる。

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る。`BSpline()`関数は、指定されたノット（区切り点）のセットに基づいて、スプラインの基底関数全体の行列を生成する。デフォルトでは、生成されるBスプラインは三次スプラインである。次数を変更するには、`degree`引数を利用すればよい。

In [16]:

```
bs_ = BSpline(internal_knots=[25,40,60], intercept=True).fit(age)
bs_age = bs_.transform(age)
bs_age.shape
```

Out[16]:

```
(3000, 7)
```

これは 7 列の行列となり、3 つの内部ノットを持つ三次スプライン基底として期待される形になる。同じ行列を `bs()`オブジェクトを使って構築することも可能で、これはモデル行列ビルダーに追加するのを容易にする（`poly()`と、その処理を行う`Poly()`の関係に似ている）。このことは既に 7.8.1 節で説明されている。

次に、`Wage` データに対して三次スプラインモデルをフィットする。

In [17]:

```
bs_age = MS([bs('age', internal_knots=[25,40,60])])
Xbs = bs_age.fit_transform(Wage)
M = sm.OLS(y, Xbs).fit()
summarize(M)
```

Out[17]:

	coef	std err	T	P>|t|
Intercept	60.4937	9.460	6.394	0.000
bs(age, internal_knots=[25, 40, 60])[0]	3.9805	12.538	0.317	0.751
bs(age, internal_knots=[25, 40, 60])[1]	44.6310	9.626	4.636	0.000
bs(age, internal_knots=[25, 40, 60])[2]	62.8388	10.755	5.843	0.000
bs(age, internal_knots=[25, 40, 60])[3]	55.9908	10.706	5.230	0.000
bs(age, internal_knots=[25, 40, 60])[4]	50.6881	14.402	3.520	0.000
bs(age, internal_knots=[25, 40, 60])[5]	16.6061	19.126	0.868	0.385

列名は少し煩雑であり、表示されたサマリーを切り詰める原因になる。

訳注：列名が長すぎると、結果に文字列の途中までしか表示されないことがある。

これらの列名は以下のように `name` 引数を使用して設定することができる。

In [18]:

```
bs_age = MS([bs('age',
                internal_knots=[25,40,60],
                name='bs(age)')])
Xbs = bs_age.fit_transform(Wage)
M = sm.OLS(y, Xbs).fit()
summarize(M)
```

Out[18]:

	coef	std err	t	P>|t|
Intercept	60.4937	9.460	6.394	0.000
bs(age)[0]	3.9805	12.538	0.317	0.751
bs(age)[1]	44.6310	9.626	4.636	0.000
bs(age)[2]	62.8388	10.755	5.843	0.000
bs(age)[3]	55.9908	10.706	5.230	0.000
bs(age)[4]	50.6881	14.402	3.520	0.000
bs(age)[5]	16.6061	19.126	0.868	0.385

7 つではなく、6 つのスプライン係数があることに注意しておこう。これは通常、モデルが全体に対する切片（インターセプト）を持つことから、デフォルトでは`bs()`が`intercept=False`を仮定しているためである。そこで`bs()`は指定されたノット（区切り点）でスプライン基底を生成し、その後、切片を考慮して基底関数の 1 つを破棄すればよい。

スプラインの複雑さを指定するために`df`（自由度）オプションを使用することもできる。上記のように、3 つのノットではスプライン基底には 6 つの列または自由度がある。実際のノットではなく`df=6`と指定すると、`bs()`はトレーニングデータの均一な分位数に基づいて 3 つのノットを選んだスプラインを生成する。この選ばれたノットを簡単に確認するには、直接`Bspline()`を使用すればよい。

In [19]:

```
BSpline(df=6).fit(age).internal_knots_
```

Out[19]:

```
array([33.75, 42.  , 51.  ])
```

自由度を 6 と指定すると、変換は年齢が 33.75、42.0、51.0 の地点にノットを配置する。これは `age` の 25%、50%、75%の分位数に対応している。

B スプラインを使用する場合、三次多項式（つまり `degree=3`）に限定する必要はない。例えば、`degree=0` を使用すると、上記の `pd.qcut()`の例のように、区分的に定数の関数が得られる。

In [20]:

```
bs_age0 = MS([bs('age',
                 df=3,
                 degree=0)]).fit(Wage)
Xbs0 = bs_age0.transform(Wage)
summarize(sm.OLS(y, Xbs0).fit())
```

Out[20]:

	Coef	std err	t	P>|t|
Intercept	94.1584	1.478	63.687	0.0
bs(age, df=3, degree=0)[0]	22.3490	2.152	10.388	0.0
bs(age, df=3, degree=0)[1]	24.8076	2.044	12.137	0.0
bs(age, df=3, degree=0)[2]	22.7814	2.087	10.917	0.0

このフィットは、25%、50%、75%の `age` 分位数で 4 つのビンを作成した `qcut()` を使用したセル[15]と比較すべきである。ここでゼロ次スプラインに対して `df=3` を指定したため、同じ 3 つの分位数にノットが配置されている。係数は異なるように見えるが、これは異なるコーディング方法によるものである。例えば、最初の係数はどちらの場合も同じで、最初のビンでの平均応答を表している。2 番目の係数については、$94.158 + 22.349 = 116.507 ～ 116.611 であり、後者はセル[15]における第 2 ビンでの平均値である。ここでは切片は 1 の列でコーディングされているため、第 2、第 3、第 4 の係数はそれぞれのビンの増分として表されている。ここでなぜ正確に一致しないのだろうか？これは、`qcut()`がビンのメンバーシップを決定する際に≤を使用し、`bs()`が<を使用するためである。

自然スプラインをフィットさせるために、対応するヘルパー`ns()`と共に`NaturalSpline()`変換を使用する。ここでは、（切片を除いて）5 つの自由度を持つ自然スプラインを適合させ、結果をプロットしている。

In [21]:

```
ns_age = MS([ns('age', df=5)]).fit(Wage)
M_ns = sm.OLS(y, ns_age.transform(Wage)).fit()
summarize(M_ns)
```

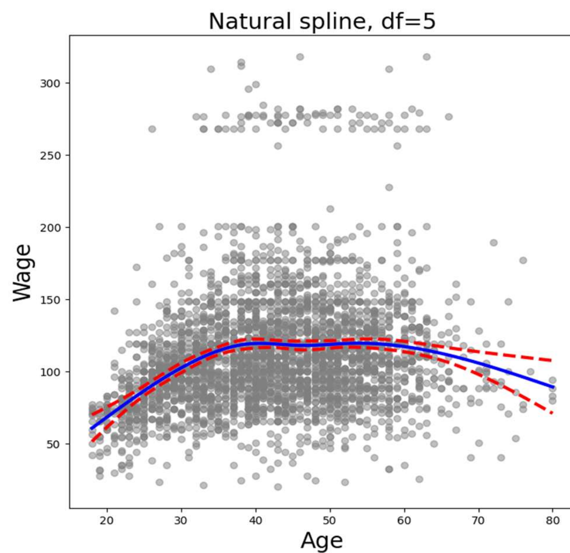
Out[21]:

	coef	std err	T	P>|t|
Intercept	60.4752	4.708	12.844	0.000
ns(age, df=5)[0]	61.5267	4.709	13.065	0.000
ns(age, df=5)[1]	55.6912	5.717	9.741	0.000
ns(age, df=5)[2]	46.8184	4.948	9.463	0.000
ns(age, df=5)[3]	83.2036	11.918	6.982	0.000
ns(age, df=5)[4]	6.8770	9.484	0.725	0.468

次に、プロット関数を使用して自然スプラインをプロットする。

In [22]:

```
plot_wage_fit(age_df,
              ns_age,
              'Natural spline, df=5');
```



平滑化スプライン(Smoothing Splines)と一般化加法モデル(GAMs)

平滑化スプラインは、二乗誤差損失と単一の特徴量を持つ GAM（Generalized Additive Model）の特殊なケースである。`Python` で GAM をフィッティングするために、`pygam` パッケージを使用しているが、`pip install pygam` でインストールできる。`LinearGAM()`推定器は二乗誤差損失を使用している。

GAM は、モデル行列の各列を特定のスムージング操作と関連付けることで指定される：`s` は平滑化スプライン、`l` は線形、`f` は因子またはカテゴリ変数を意味する。以下の `s` に渡される引数 `0` は、このスムージングが特徴行列の最初の列に適用されることを示している。以下では、1 つの列を持つ行列 `X_age` にそれを適用するが、引数 `lam` は、7.5.2 節で説明したペナルティパラメータ λ である。

In [23]:

```
X_age = np.asarray(age).reshape((-1,1))
gam = LinearGAM(s_gam(0, lam=0.6))
gam.fit(X_age, y)
```

Out[23]:

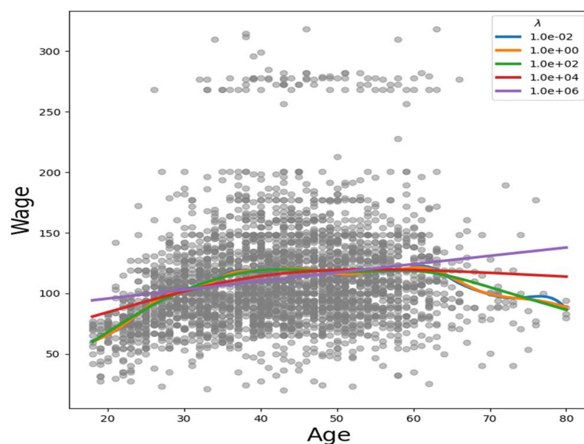
```
LinearGAM(callbacks=[Deviance(), Diffs()], fit_intercept=True,
   max_iter=100, scale=None, terms=s(0) + intercept, tol=0.0001,
   verbose=False)
```

`pygam` ライブラリは一般的に特徴量の行列を受けとるため、`age` をベクトル（一次元配列）ではなく、行列（二次元配列）に変形する。`reshape()`メソッドの呼び出しで`-1` を指定することで、その次元のサイズを残りの形状タプルのエントリに基づいて自動的に推定するよう `numpy` に指示する。

平滑化パラメータ `lam` によって適合がどのように変化するかを調べてみよう。`np.logspace()`関数は、`np.linspace()`に似ているが、対数スケール上で等間隔に点を配置する。以下では、`lam` を 10^{-2} から 10^{6} まで変化させている。

In [24]:

```
fig, ax = subplots(figsize=(8,8))
ax.scatter(age, y, facecolor='gray', alpha=0.5)
for lam in np.logspace(-2, 6, 5):
    gam = LinearGAM(s_gam(0, lam=lam)).fit(X_age, y)
    ax.plot(age_grid,
            gam.predict(age_grid),
            label='{:.1e}'.format(lam),
            linewidth=3)
ax.set_xlabel('Age', fontsize=20)
ax.set_ylabel('Wage', fontsize=20);
ax.legend(title='$\lambda$');
```



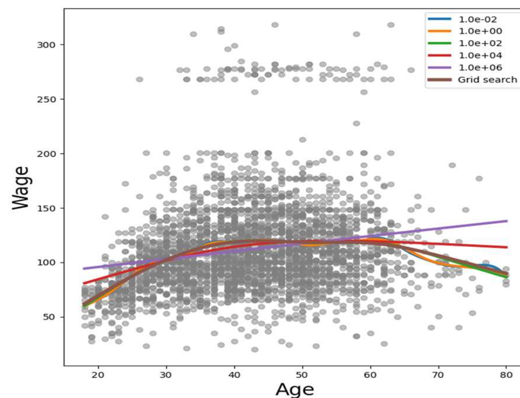
`pygam` パッケージでは最適な平滑化パラメーターを探すための探索機能も備えている。

In [25]:

```
gam_opt = gam.gridsearch(X_age, y)
ax.plot(age_grid,
        gam_opt.predict(age_grid),
        label='Grid search',
        linewidth=4)
ax.legend()
fig
```

```
100% (11 of 11) |########################| Elapsed Time: 0:00:01 Time:  0:00:01
```

Out[25]:



また、`ISLP.pygam`パッケージに含まれる関数を使用して、平滑化スプラインの自由度を固定することも可能である。以下では、約 4 の自由度を持つλの値を求めている。ここで注意すべき点は、これらの自由度には平滑化スプラインの罰則がついていない切片および線形項が含まれるため、少なくとも 2 つの自由度があるということである。

In [26]:

```
age_term = gam.terms[0]
lam_4 = approx_lam(X_age, age_term, 4)
age_term.lam = lam_4
degrees_of_freedom(X_age, age_term)
```

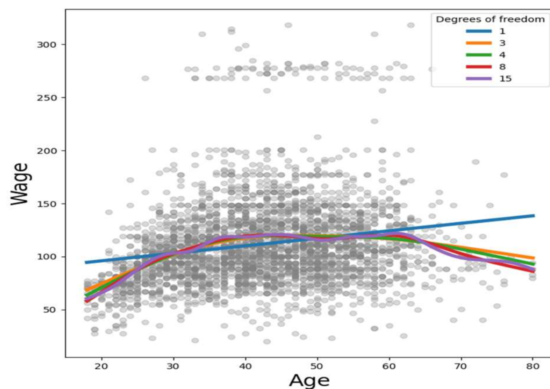
Out[26]:

```
4.000000100003869
```

上記と同様に、自由度を変化させてプロットしてみよう。これらの平滑化スプラインには常に切片項があるため、希望する自由度に 1 を加えた値を選択する。そのため、`df`が 1 の場合は単なる線形フィットになる。

In [27]:

```
fig, ax = subplots(figsize=(8,8))
ax.scatter(X_age,
           y,
           facecolor='gray',
           alpha=0.3)
for df in [1,3,4,8,15]:
    lam = approx_lam(X_age, age_term, df+1)
    age_term.lam = lam
    gam.fit(X_age, y)
    ax.plot(age_grid,
            gam.predict(age_grid),
            label='{:d}'.format(df),
            linewidth=4)
ax.set_xlabel('Age', fontsize=20)
ax.set_ylabel('Wage', fontsize=20);
ax.legend(title='Degrees of freedom');
```



加法モデルの拡張

一般化加法モデル（GAM）の強みは、線形モデルよりも柔軟に多変量回帰モデルを適合できる点にある。ここでは、2 つのアプローチを紹介しておこう。 1 つ目は、比較的手動で自然スプラインと区分定数関数を使用する方法、次に `pygam` パッケージと平滑化スプラインを用いる方法である。

以下のコードでは、手動で GAM をフィッティングして、`wage` を予測する。この例では、自然スプライン関数を `year` と `age` に適用し、`education` を質的な変数として扱う。これは式(7.16)のような形になる。GAM は適切な基底関数を使用する大規模な線形回帰モデルに過ぎないため、`sm.OLS()`関数を使って実行できる。

ここでは、部分依存性プロット（Partial Dependence Plot）を作成するときに、モデル行列の各要素に個別にアクセスできるようにしたいので、ある程度手動でモデル行列を構築する。

In [28]:

```
ns_age = NaturalSpline(df=4).fit(age)
ns_year = NaturalSpline(df=5).fit(Wage['year'])
Xs = [ns_age.transform(age),
      ns_year.transform(Wage['year']),
      pd.get_dummies(Wage['education']).values]
X_bh = np.hstack(Xs)
gam_bh = sm.OLS(y, X_bh).fit()
```

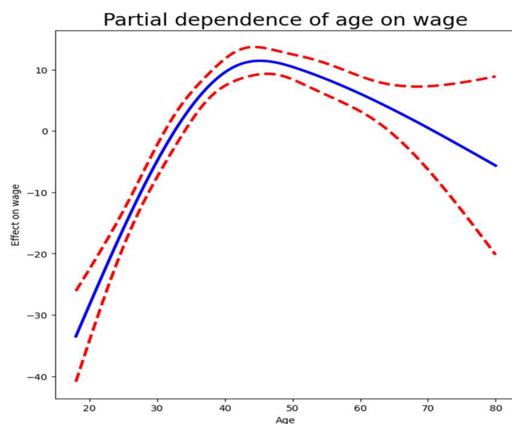
ここで `NaturalSpline()`関数がヘルパー関数 `ns()`を支えるように機能している。`education` というカテゴリカル変数の指示行列のすべての列を使用することを選択しているため、切片は不要である。最後に、3 つの要素行列を横方向に結合して、モデル行列 `X_bh` を作成した。

次に、推定した簡単な GAM の各項目に対して部分依存性プロットを作成する方法を示す。`age` と `year` のグリッドを使って予測することで、手動で作成できる。このとき、他の変数は固定し、1 つの変数のみを変動させた状態で新しい X 行列を使用して予測してみよう。

In [29]:

```
age_grid = np.linspace(age.min(),
                       age.max(),
                       100)
X_age_bh = X_bh.copy()[:100]
X_age_bh[:] = X_bh[:].mean(0)[None,:]
X_age_bh[:,:4] = ns_age.transform(age_grid)
preds = gam_bh.get_prediction(X_age_bh)
```

```
bounds_age = preds.conf_int(alpha=0.05)
partial_age = preds.predicted_mean
center = partial_age.mean()
partial_age -= center
bounds_age -= center
fig, ax = subplots(figsize=(8,8))
ax.plot(age_grid, partial_age, 'b', linewidth=3)
ax.plot(age_grid, bounds_age[:,0], 'r--', linewidth=3)
ax.plot(age_grid, bounds_age[:,1], 'r--', linewidth=3)
ax.set_xlabel('Age')
ax.set_ylabel('Effect on wage')
ax.set_title('Partial dependence of age on wage', fontsize=20);
```



上記で行った内容について、少し詳しく説明しておこう。アイデアとしては、新しい予測行列を作成し、age に属する列以外のすべてを一定値（訓練データの平均）に設定することである。そして、age の 4 つの列は、age_grid の 100 個の値で評価した自然スプライン基底で埋める。

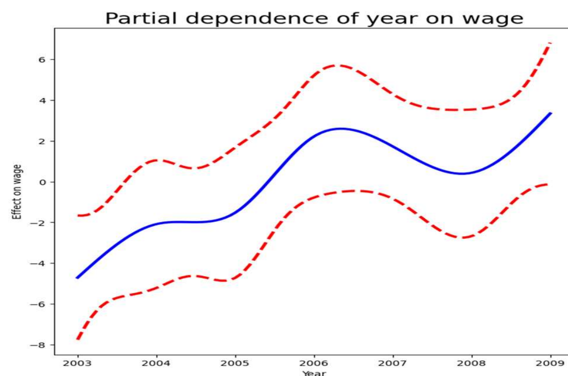
- age の長さ 100 のグリッドを作成し、X_age_bh という行列を作成した。これは 100 行で、列数は X_bh と同じである。
- この行列のすべての行を、元の行列の列ごとの平均値で置き換えた。
- その後、age を表す最初の 4 つの列だけを age_grid の値で計算された自然スプライン基底に置き換えた。

残りのステップは、すでに述べた方法と同一である。

`year` が `wage` に与える影響を見てみよう。このプロセスも同様である。

In [30]:

```
year_grid = np.linspace(2003, 2009, 100)
year_grid = np.linspace(Wage['year'].min(),
                        Wage['year'].max(),
                        100)
X_year_bh = X_bh.copy()[:100]
X_year_bh[:] = X_bh[:].mean(0)[None,:]
X_year_bh[:,4:9] = ns_year.transform(year_grid)
preds = gam_bh.get_prediction(X_year_bh)
bounds_year = preds.conf_int(alpha=0.05)
partial_year = preds.predicted_mean
center = partial_year.mean()
partial_year -= center
bounds_year -= center
fig, ax = subplots(figsize=(8,8))
ax.plot(year_grid, partial_year, 'b', linewidth=3)
ax.plot(year_grid, bounds_year[:,0], 'r--', linewidth=3)
ax.plot(year_grid, bounds_year[:,1], 'r--', linewidth=3)
ax.set_xlabel('Year')
ax.set_ylabel('Effect on wage')
ax.set_title('Partial dependence of year on wage', fontsize=20);
```



次に、自然スプラインではなく平滑化スプラインを用いてモデル（7.16）をフィット(適合)する。式（7.16）のすべての項は同時にフィットし、互いの影響を考慮しながら応答変数を説明している。`pygam` パッケージは行列しか受け付けないため、カテゴリカルな系列である `education` を配列表現に変換する必要がある。これは `education` の `cat.codes` 属性で取得できる。また、`year` には 7 つのユニークな値しかないため、7 つの基底関数だけを利用する。

In [31]:

```
gam_full = LinearGAM(s_gam(0) +
                     s_gam(1, n_splines=7) +
                     f_gam(2, lam=0))
Xgam = np.column_stack([age,
                        Wage['year'],
                        Wage['education'].cat.codes])
gam_full = gam_full.fit(Xgam, y)
```

2 つの `s_gam()` 項は平滑化スプラインによるフィットとなり、λのデフォルト値（`lam=0.6`）が使用されている。この値はある程度恣意的に決められたものである。カテゴリカルな項 `education` は `f_gam()` 項を使用して指定し、`lam=0` とすることで係数の縮小を防ぐ。

訳注 4：罰則をつけないことで、係数の推定値が 0 に向かって縮小されることを防いでいる。

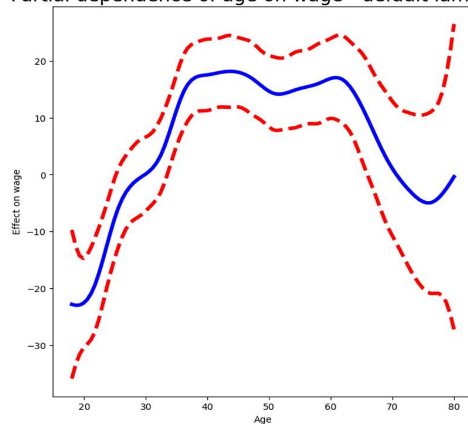
`age` に対する部分依存性プロットを作成し、これらの選択がもたらす影響を確認できる。

プロット用の値は `pygam` パッケージによって生成される。`ISLP.pygam` パッケージ内の `plot_gam()` 関数を使用することで、自然スプラインを使った前回の例よりも簡単に部分依存性プロットを作成できる。

In [32]:

```
fig, ax = subplots(figsize=(8,8))
plot_gam(gam_full, 0, ax=ax)
ax.set_xlabel('Age')
ax.set_ylabel('Effect on wage')
```

```
ax.set_title('Partial dependence of age on wage - default lam=0.6', fontsize=2
0);
```



関数がややギザギザしていることが分かる。`lam` の値を指定するよりも `df` を指定するほうが自然である。

訳注 5：基底関数が多すぎるように見えるという意味だと思われる。

`age` と `year` のそれぞれに対して 4 の自由度を指定して GAM を再度推定してみる。以下で 1 を加算しているのは、平滑化スプラインの切片を考慮したためである。

In [33]:

```
age_term = gam_full.terms[0]
age_term.lam = approx_lam(Xgam, age_term, df=4+1)
year_term = gam_full.terms[1]
year_term.lam = approx_lam(Xgam, year_term, df=4+1)
gam_full = gam_full.fit(Xgam, y)
```

上記で `age_term.lam` を更新すると、それは `gam_full.terms[0]` でも更新されることに注意しておこう！ `year_term.lam` についても同様である。

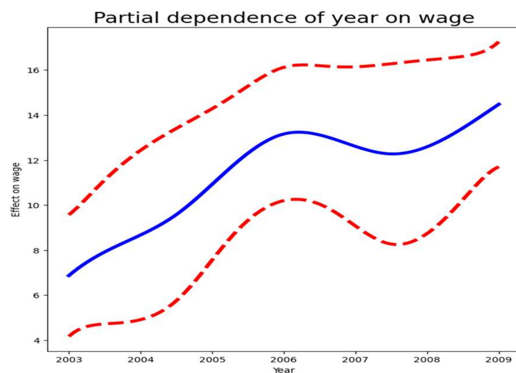
`age` に対するプロットを再度作成すると、ずっと滑らかになっていることが分かる。また、`year` に対するプロットも作成してみよう。

In [34]:

```
fig, ax = subplots(figsize=(8,8))
plot_gam(gam_full,
         1,
         ax=ax)
ax.set_xlabel('Year')
ax.set_ylabel('Effect on wage')
ax.set_title('Partial dependence of year on wage', fontsize=20)
```

Out[34]:

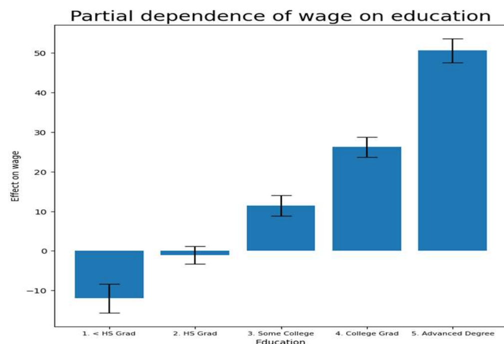
```
Text(0.5, 1.0, 'Partial dependence of year on wage')
```



最後に、カテゴリカルな `education` に対するプロットを作成してみる。この変数の各レベルに対して適合された定数のセットにより、部分依存性プロットは異なり、より適切な形式で表示される。

In [35]:

```
fig, ax = subplots(figsize=(8, 8))
ax = plot_gam(gam_full, 2)
ax.set_xlabel('Education')
ax.set_ylabel('Effect on wage')
ax.set_title('Partial dependence of wage on education',
             fontsize=20);
ax.set_xticklabels(Wage['education'].cat.categories, fontsize=8);
```



加法モデルへの ANOVA

ここで扱ったすべてのモデルにおいて、`year` の関数はほぼ線形に見える。これら 3 つのモデルのうちどれが最適かを判断するために、一連の ANOVA 検定を実施してみよう。

- `year` を除外した GAM (M_1)
- `year` を線形関数として扱った GAM (M_2)
- `year` をスプライン関数として扱った GAM (M_3)

In [36]:

```
gam_0 = LinearGAM(age_term + f_gam(2, lam=0))
gam_0.fit(Xgam, y)
gam_linear = LinearGAM(age_term +
                       l_gam(1, lam=0) +
                       f_gam(2, lam=0))
gam_linear.fit(Xgam, y)
```

Out[36]:

```
LinearGAM(callbacks=[Deviance(), Diffs()], fit_intercept=True,
   max_iter=100, scale=None, terms=s(0) + l(1) + f(2) + intercept,
   tol=0.0001, verbose=False)
```

上記の式に `age_term` を使用していることに注目しておこう。これは、自由度を 4 に調整するために、以前にこの項の `lam` の値を設定したためである。

`year` の影響を直接評価するために、上記で適合させた 3 つのモデルに対して ANOVA を実行する。

In [37]:

```
anova_gam(gam_0, gam_linear, gam_full)
```

Out[37]:

	Deviance	Df	deviance_diff	df_diff	F	pvalue
0	3.714362e+06	2991.004005	NaN	NaN	NaN	NaN
1	3.696746e+06	2990.005190	17616.542840	0.998815	14.265131	0.002314
2	3.693143e+06	2987.007254	3602.893655	2.997936	0.972007	0.435579

その結果、`year` を含まない GAM よりも `year` を線形関数として含む GAM のほうが適しているという強い証拠が得られた（p 値 = 0.002）。しかし、`year` を非線形関数として扱う必要があるという証拠は得られなかった（p 値 = 0.435）。
言い換えると、この ANOVA の結果に基づけば M_2 が適切だろう。

`age` に対しても同じプロセスを繰り返すことができる。その結果、`age` には非線形な項が必要であるという明確な証拠が得られる。

In [38]:

```
gam_0 = LinearGAM(year_term +
                  f_gam(2, lam=0))
gam_linear = LinearGAM(l_gam(0, lam=0) +
                       year_term +
                       f_gam(2, lam=0))
gam_0.fit(Xgam, y)
gam_linear.fit(Xgam, y)
anova_gam(gam_0, gam_linear, gam_full)
```

Out[38]:

	deviance	df	deviance_diff	df_diff	F	pvalue
0	3.975443e+06	2991.000589	NaN	NaN	NaN	NaN
1	3.850247e+06	2990.000704	125196.137317	0.999884	101.270106	1.681120e-07
2	3.693143e+06	2987.007254	157103.978302	2.993450	42.447812	5.669414e-07

GAM の推定には、（冗長ではあるが） `summary()` メソッドがある。

In [39]:

```
gam_full.summary()
LinearGAM

=============================================== ==========================================================
=======
Distribution:                          NormalDist Effective DoF:
12.9927
Link Function:                       IdentityLink Log Likelihood:                                       -24
117.907
Number of Samples:                           3000 AIC:                                                  482
63.7995
                                                 AICc:                                                    4
8263.94
                                                 GCV:                                                    12
46.1129
                                                 Scale:                                                  12
36.4024
                                                 Pseudo R-Squared:
 0.2928
==========================================================================================================
=======
Feature Function                  Lambda               Rank         EDoF         P > x        Sig.
Code
================================= ==================== ============ ============ ============ =====
=======
s(0)                              [465.0491]           20           5.1          1.11e-16     ***

s(1)                              [2.1564]             7            4.0          8.10e-03     **

f(2)                              [0]                  5            4.0          1.11e-16     ***

intercept                                              1            0.0          1.11e-16     ***

==========================================================================================================
=======
Significance codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1

WARNING: Fitting splines and a linear function to a feature introduces a model identifiability prob
lem
         which can cause p-values to appear significant when they are not.
```

```
WARNING: p-values calculated in this manner behave correctly for un-penalized models or models with
         known smoothing parameters, but when smoothing parameters have been estimated, the p-value
s
         are typically lower than they should be, meaning that the tests reject the null too readil
y.
<ipython-input-39-dd2b60753e24>:1: UserWarning: KNOWN BUG: p-values computed in this summary are li
kely much smaller than they should be.

Please do not make inferences based on these values!

Collaborate on a solution, and stay up to date at:
github.com/dswah/pyGAM/issues/163

  gam_full.summary()
```

`gam` オブジェクトでは、`lm` オブジェクトと同様に `predict()` メソッドを使用して予測を行うことができる。ここでは、トレーニングセットに対して予測を行おう。

In [40]:

```
Yhat = gam_full.predict(Xgam)
```

ロジスティック回帰の GAM を推定するためには、`pygam` の `LogisticGAM()` を利用する。

In [41]:

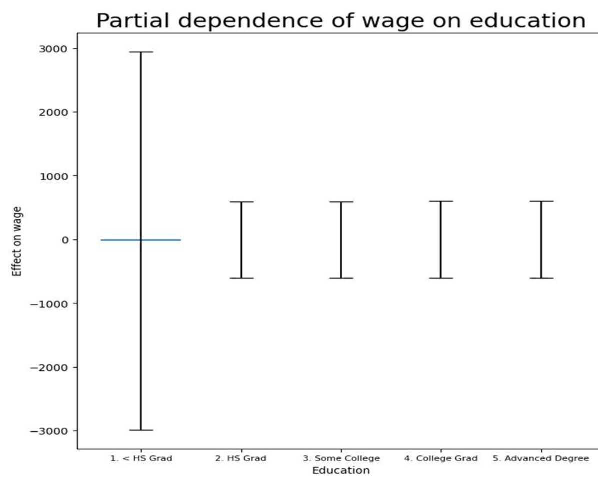
```
gam_logit = LogisticGAM(age_term +
                        l_gam(1, lam=0) +
                        f_gam(2, lam=0))
gam_logit.fit(Xgam, high_earn)
```

Out[41]:

```
LogisticGAM(callbacks=[Deviance(), Diffs(), Accuracy()],
   fit_intercept=True, max_iter=100,
   terms=s(0) + l(1) + f(2) + intercept, tol=0.0001, verbose=False)
```

In [42]:

```
fig, ax = subplots(figsize=(8, 8))
ax = plot_gam(gam_logit, 2)
ax.set_xlabel('Education')
ax.set_ylabel('Effect on wage')
ax.set_title('Partial dependence of wage on education',
             fontsize=20);
ax.set_xticklabels(Wage['education'].cat.categories, fontsize=8);
```



このモデルは非常に平坦であり、特に最初のカテゴリに対する誤差バーが大きくなっている。そこでデータをもう少し詳しく見てみよう。

In [43]:

```
pd.crosstab(Wage['high_earn'], Wage['education'])
```

Out[43]:
education
1. < HS Grad
2. HS Grad
3. Some College
4. College Grad
5. Advanced Degree
high_earn
False
268
966
643
663
381

```
True
0
5
7
22
45
```

`education` の最初のカテゴリには高所得者 (`high_earn`) が存在しないことが分かる。そのため、モデルの推定が難しくなっている。そこで、このカテゴリに該当する観測値をすべて除外してロジスティック回帰 GAM を推定してみよう。これにより、より妥当な結果が得られる。

分析を行うには、単にモデル行列をサブセット化することもできるが、その場合でも `Xgam` から該当する列が削除されるわけではない。どの列がこの特徴量に対応するかを推測することは可能であるが、再現性を考慮して、小さいサブセット上で再度モデル行列を作成してみよう。

In [44]:

```
only_hs = Wage['education'] == '1. < HS Grad'
Wage_ = Wage.loc[~only_hs]
Xgam_ = np.column_stack([Wage_['age'],
                         Wage_['year'],
                         Wage_['education'].cat.codes-1])
high_earn_ = Wage_['high_earn']
```

上記の 2 番目から最後の行では、`pygam` のバグを回避するために `education` のカテゴリコードから 1 を引いている。
この操作は単に教育レベルのラベルを付け直すだけなので、モデルの推定には影響しない。そこでモデルを推定してみる。

In [45]:

```
gam_logit_ = LogisticGAM(age_term +
                         year_term +
                         f_gam(2, lam=0))
gam_logit_.fit(Xgam_, high_earn_)
```

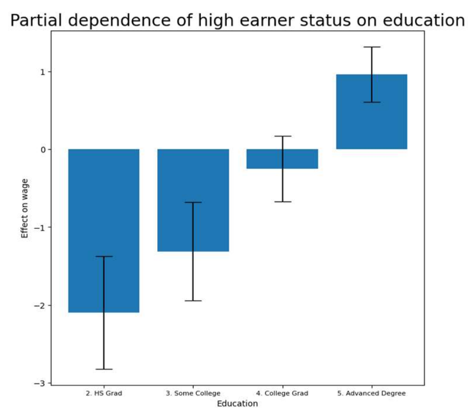
Out[45]:

```
LogisticGAM(callbacks=[Deviance(), Diffs(), Accuracy()],
   fit_intercept=True, max_iter=100,
   terms=s(0) + s(1) + f(2) + intercept, tol=0.0001, verbose=False)
```

これで先ほどの観測値を除外できたので、`education`、`year`、および `age` が高所得者(`high_earn`) のステータスに与える影響が確認できる。

In [46]:

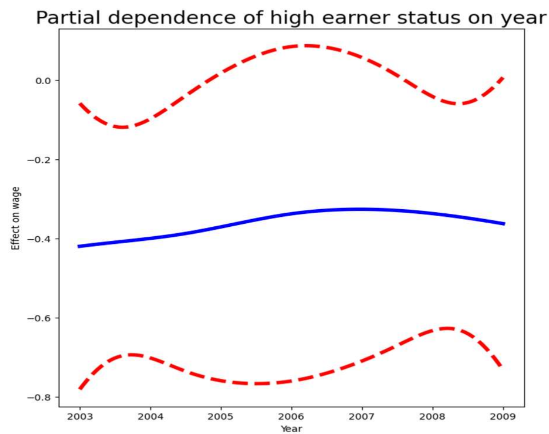
```
fig, ax = subplots(figsize=(8, 8))
ax = plot_gam(gam_logit_, 2)
ax.set_xlabel('Education')
ax.set_ylabel('Effect on wage')
ax.set_title('Partial dependence of high earner status on education', fontsize=
20);
ax.set_xticklabels(Wage['education'].cat.categories[1:],
                   fontsize=8);
```



In [47]:

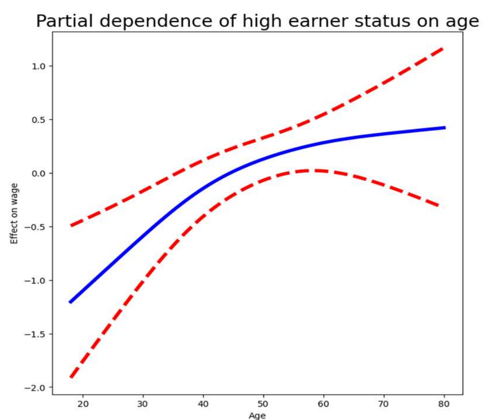
```
fig, ax = subplots(figsize=(8, 8))
ax = plot_gam(gam_logit_, 1)
ax.set_xlabel('Year')
ax.set_ylabel('Effect on wage')
```

```
ax.set_title('Partial dependence of high earner status on year',
             fontsize=20);
```



In [48]:

```
fig, ax = subplots(figsize=(8, 8))
ax = plot_gam(gam_logit_, 0)
ax.set_xlabel('Age')
ax.set_ylabel('Effect on wage')
ax.set_title('Partial dependence of high earner status on age', fontsize=20);
```



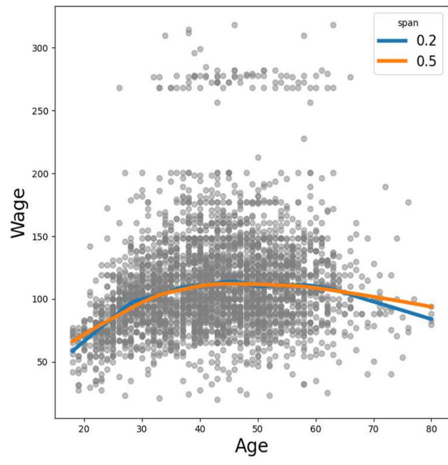
局所回帰(Local Regression)

`sm.nonparametric` の `lowess()` 関数を使用した局所回帰の利用方法を示しておく。一部の GAM の実装では、局所回帰を適用することができるが、`pygam` ではこれはサポートされていない。

ここでは、スパン (span) を 0.2 および 0.5 に設定したローカル線形回帰モデルをフィットする。この場合、それぞれの近傍は観測データの 20% または 50% で構成される。予想通り、スパンを 0.5 に設定した方が、0.2 よりも滑らかな結果が得られる。

In [49]:

```
lowess = sm.nonparametric.lowess
fig, ax = subplots(figsize=(8,8))
ax.scatter(age, y, facecolor='gray', alpha=0.5)
for span in [0.2, 0.5]:
    fitted = lowess(y,
                    age,
                    frac=span,
                    xvals=age_grid)
    ax.plot(age_grid,
            fitted,
            label='{:.1f}'.format(span),
            linewidth=4)
ax.set_xlabel('Age', fontsize=20)
ax.set_ylabel('Wage', fontsize=20);
ax.legend(title='span', fontsize=15);
```



span
0.2
0.5
Wage
Age
300
250
200
150
100
50
20
30
40
50
60
70
80

ISLP 第 8 章：決定木モデル

まず、必要な標準ライブラリをインポートする。

In [1]:

```
import numpy as np
import pandas as pd
from matplotlib.pyplot import subplots
import sklearn.model_selection as skm
from ISLP import load_data, confusion_table
from ISLP.models import ModelSpec as MS
```

また、このラボで必要な追加のライブラリもまとめてインポートしておく。

In [2]:

```
from sklearn.tree import (DecisionTreeClassifier as DTC,
                          DecisionTreeRegressor as DTR,
                          plot_tree,
                          export_text)
from sklearn.metrics import (accuracy_score,
                             log_loss)
from sklearn.ensemble import \
     (RandomForestRegressor as RF,
      GradientBoostingRegressor as GBR)
from ISLP.bart import BART
```

分類木の適合

最初に分類木を使用して 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 2 値変数に変換する。where()関数を使用して High という変数を作成し、Sales 変数が 8 を超える場合は Yes、それ以外の場合は No の値を取る。

In [3]:

```
Carseats = load_data('Carseats')
High = np.where(Carseats.Sales > 8,
                "Yes",
                "No")
```

次に、DecisionTreeClassifier()を使用して分類木を適合させ、Sales 以外のすべての変数を使用して High を予測する。このために、回帰モデルを適合させるときと同様にモデルマトリックスを作成する必要がある。

In [4]:

```
model = MS(Carseats.columns.drop('Sales'), intercept=False)
D = model.fit_transform(Carseats)
feature_names = list(D.columns)
X = np.asarray(D)
```

D をデータフレームから配列 X に変換した。これは以下のいくつかの分析で必要になる。また、後でプロットに注釈を付けるために feature_names も必要である。

分類器を指定するために必要なオプションがいくつかある。例えば、max_depth（木を成長させる深さ）、min_samples_split（分割するためにノードに必要な観測数の最小値）、および criterion（分割基準としてジニまたはクロスエントロピーを使用するか）などである。また、再現性のために random_state を設定する。なお、分割基準において同点の候補が複数ある場合は、ランダムに決定される。

In [5]:

```
clf = DTC(criterion='entropy',
          max_depth=3,
          random_state=0)
clf.fit(X, High)
```

Out[5]:

```
DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)
```

DecisionTreeClassifier

```
DecisionTreeClassifier(criterion='entropy', max_depth=3, random_state=0)
```

第 3 章の定性的特徴に関する議論では、線形回帰モデルでは、そのような特徴をダミー変数（ワンホットエンコーディング）の行列をモデルマトリックスに含めることで表現できると述べた。第 8 章の決定木に関するセクションで述べたように、定性的特徴を扱うより自然な方法がある。これはそのようなダミー変数を必要とせず、各分割はレベルを 2 つのグループに分割することに相当する。しかし、`sklearn` の決定木の実装はこのアプローチを活用せず、代わりにワンホットエンコードされたレベルを別々の変数として扱う。

In [6]:

```
accuracy_score(High, clf.predict(X))
```

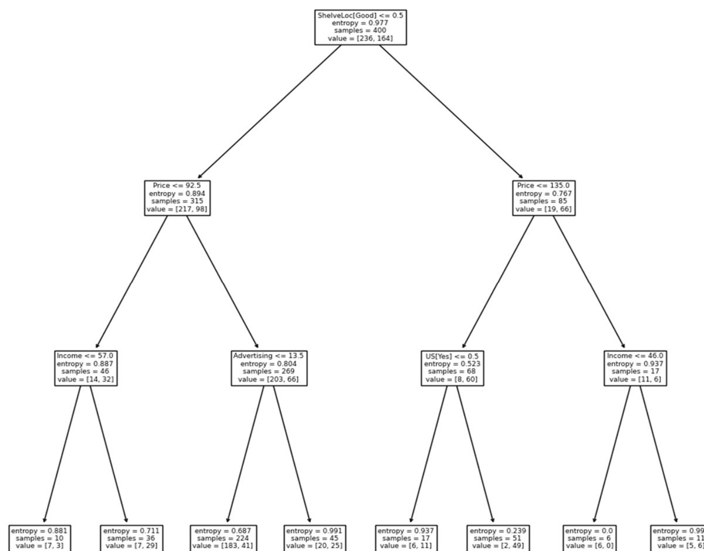
Out[6]:

```
0.79
```

木の最も魅力的な特性の 1 つは、グラフィカルに表示できることである。ここでは、`plot()`関数を使用して木の構造を表示する。

In [7]:

```
ax = subplots(figsize=(12, 12))[1]
plot_tree(clf,
          feature_names=feature_names,
          ax=ax);
```



`Sales`の最も重要な指標は`ShelveLoc`のようである。

`export_text()`を使用して、木のテキスト表現を表示できる。これは、各枝の分割基準（例：`Price <= 92.5`）を表示する。リーフノードでは、全体の予測（`Yes`または`No`）を示す。また、`show_weights=True`を指定することで、そのリーフに`Yes`と`No`の値を持つ観測数を確認できる。

In [8]:

```
print(export_text(clf,
                  feature_names=feature_names,
                  show_weights=True))
|--- ShelveLoc[Good] <= 0.50
|   |--- Price <= 92.50
```

```
|   |   |--- Income <= 57.00
|   |   |   |--- weights: [7.00, 3.00] class: No
|   |   |--- Income >  57.00
|   |   |   |--- weights: [7.00, 29.00] class: Yes
|   |--- Price >  92.50
|   |   |--- Advertising <= 13.50
|   |   |   |--- weights: [183.00, 41.00] class: No
|   |   |--- Advertising >  13.50
|   |   |   |--- weights: [20.00, 25.00] class: Yes
|--- ShelveLoc[Good] >  0.50
|   |--- Price <= 135.00
|   |   |--- US[Yes] <= 0.50
|   |   |   |--- weights: [6.00, 11.00] class: Yes
|   |   |--- US[Yes] >  0.50
|   |   |   |--- weights: [2.00, 49.00] class: Yes
|   |--- Price >  135.00
|   |   |--- Income <= 46.00
|   |   |   |--- weights: [6.00, 0.00] class: No
|   |   |--- Income >  46.00
|   |   |   |--- weights: [5.00, 6.00] class: Yes
```

これらのデータに対する分類木の性能を適切に評価するためには、訓練誤差を計算するだけでなく、テスト誤差を推定する必要がある。観測を訓練セットとテストセットに分割し、訓練セットを使用して木を構築し、テストデータに対する性能を評価する。このパターンは第 6 章の変数選択の章と似ており、線形モデルがここでは決定木に置き換えられている。検証のためのコードはほぼ同じである。このアプローチでは、テストデータセットの 68.5%の位置で正しい予測が得られる。

In [9]:

```
validation = skm.ShuffleSplit(n_splits=1,
                              test_size=200,
                              random_state=0)
results = skm.cross_validate(clf,
                             D,
                             High,
                             cv=validation)
results['test_score']
```

Out[9]:

```
array([0.685])
```

次に、木を剪定することで分類性能が向上するかどうかを検討しよう。まず、データを訓練セットとテストセットに分割する。クロスバリデーションを使用して訓練セットで木を剪定し、その後、テストセットで剪定された木の性能を評価する。

In [10]:

```
(X_train,
 X_test,
 High_train,
 High_test) = skm.train_test_split(X,
                                   High,
                                   test_size=0.5,
                                   random_state=0)
```

まず、訓練セットに完全な木を再適合させる。ここでは`max_depth`パラメータを設定しない。これはクロスバリデーションを通じて学習させるためである。

In [11]:

```
clf = DTC(criterion='entropy', random_state=0)
clf.fit(X_train, High_train)
accuracy_score(High_test, clf.predict(X_test))
```

Out[11]:

```
0.735
```

次に、`clf`の`cost_complexity_pruning_path()`メソッドを使用してコスト複雑度の値を抽出する。

In [12]:

```
ccp_path = clf.cost_complexity_pruning_path(X_train, High_train)
kfold = skm.KFold(10,
                  random_state=1,
                  shuffle=True)
```

これにより、不純物とαの値のセットが得られ、クロスバリデーションを通じて最適なものを抽出できる。

In [13]:

```
grid = skm.GridSearchCV(clf,
                        {'ccp_alpha': ccp_path.ccp_alphas},
                        refit=True,
                        cv=kfold,
                        scoring='accuracy')
grid.fit(X_train, High_train)
grid.best_score_
```

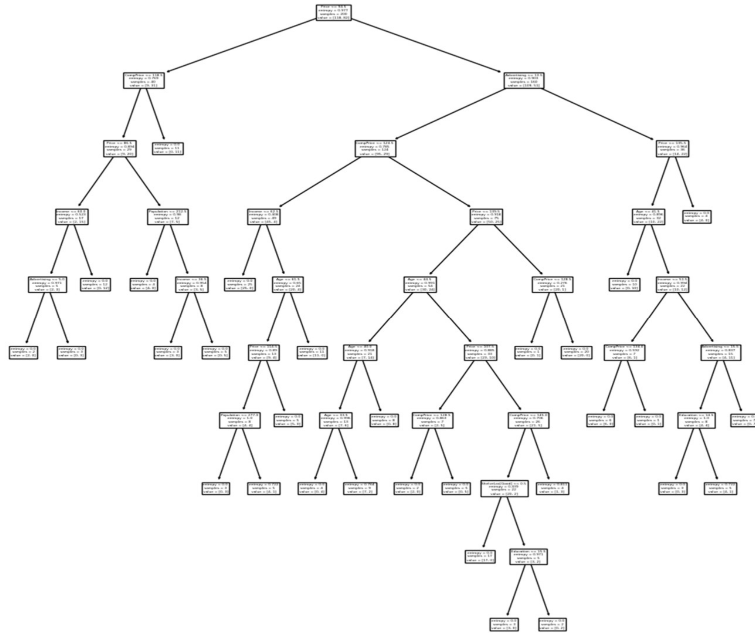
Out[13]:

```
0.685
```

剪定された木を見てみよう。

In [14]:

```
ax = subplots(figsize=(12, 12))[1]
best_ = grid.best_estimator_
plot_tree(best_,
          feature_names=feature_names,
          ax=ax);
```



これはかなり茂った木である。葉を数えることもでき、`best_`をクエリすることもできる。

In [15]:

```
best_.tree_.n_leaves
```

Out[15]:

```
30
```

30 個の終端ノードを持つ木が最も低いクロスバリデーション誤差率をもたらし、精度は 68.5%となる。この剪定された木がテストデータセットでどれだけうまく機能するかを確認しよう。再び`predict()`関数を適用する。

In [16]:

```
print(accuracy_score(High_test,
                     best_.predict(X_test)))
```

```
confusion = confusion_table(best_.predict(X_test),
                            High_test)
confusion
0.72
```

Out[16]:

True Predicted	No	Yes
No	94	32
Yes	24	50

テスト観測の 72.0%が正しく分類されている。これは完全な木（35 枚の葉を持つ）のエラーよりもわずかに悪い。したがって、クロスバリデーションはここではあまり役立たなかった。わずかに精度が低下する代わりに、5 つの葉が剪定されただけであった。これらの結果は、上記の乱数シードを変更すると変わる可能性がある。クロスバリデーションはモデル選択に対して偏りのないアプローチを提供するばらつきもある。

デフォルトの引数のみを使用すると、訓練誤差率は 21%となる。分類木の場合、`log_loss()`を使用してデビアンスの値にアクセスできる。

$$-2\sum_m \sum_k n_{mk} \log \hat{p}_{mk}$$

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。

In [17]:

```
resid_dev = np.sum(log_loss(High, clf.predict_proba(X)))
resid_dev
```

Out[17]:

```
4.775784074058023
```

回帰木の適合

ここでは、`Boston`データセットに回帰木を適合させてみる。手順は分類木の場合と類似している。

In [18]:

```
Boston = load_data("Boston")
model = MS(Boston.columns.drop('medv'), intercept=False)
D = model.fit_transform(Boston)
feature_names = list(D.columns)
X = np.asarray(D)
```

まず、データをトレーニングセットとテストセットに分割し、トレーニングデータに木を適合させる。ここではデータの 30%をテストセットに使用する。

In [19]:

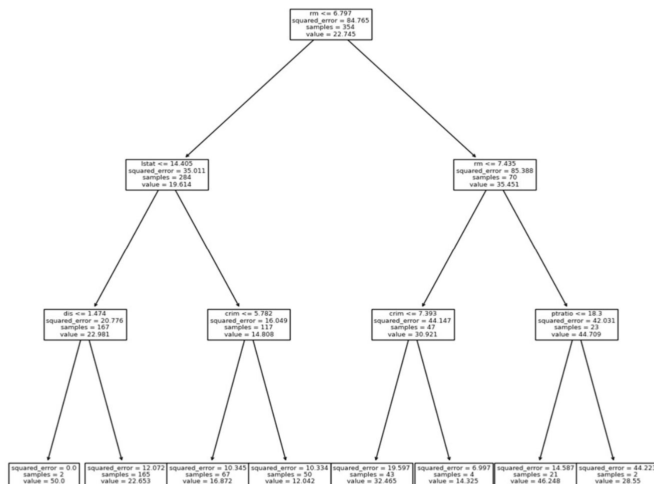
```
(X_train,
 X_test,
 y_train,
 y_test) = skm.train_test_split(X,
                                Boston['medv'],
                                test_size=0.3,
                                random_state=0)
```

トレーニングセットとテストセットを作成した後、回帰木を適合させる。

In [20]:

```
reg = DTR(max_depth=3)
reg.fit(X_train, y_train)
ax = subplots(figsize=(12,12))[1]
plot_tree(reg,
```

```
            feature_names=feature_names,
            ax=ax);
```



`lstat`変数は、低所得者層の割合を示している。決定木の分割結果から、`lstat`の値が低いほど住宅価格が高い傾向が見られる。この木は、低所得者層が多く（`lstat > 14.4`）、犯罪率が中程度（`crim > 5.8`）の郊外にある小型住宅（`rm < 6.8`）の中央値の住宅価格を 12,042 ドルと予測している。

次に、クロスバリデーション機能を使用して、木を剪定することで性能が向上するかどうかを確認しよう。

In [21]:

```
ccp_path = reg.cost_complexity_pruning_path(X_train, y_train)
kfold = skm.KFold(5,
                  shuffle=True,
                  random_state=10)
grid = skm.GridSearchCV(reg,
                        {'ccp_alpha': ccp_path.ccp_alphas},
                        refit=True,
                        cv=kfold,
```

```
                         scoring='neg_mean_squared_error')
G = grid.fit(X_train, y_train)
```

クロスバリデーションの結果に従って、剪定された木を使用してテストセットの予測を行う。

In [22]:

```
best_ = grid.best_estimator_
np.mean((y_test - best_.predict(X_test))**2)
```

Out[22]:

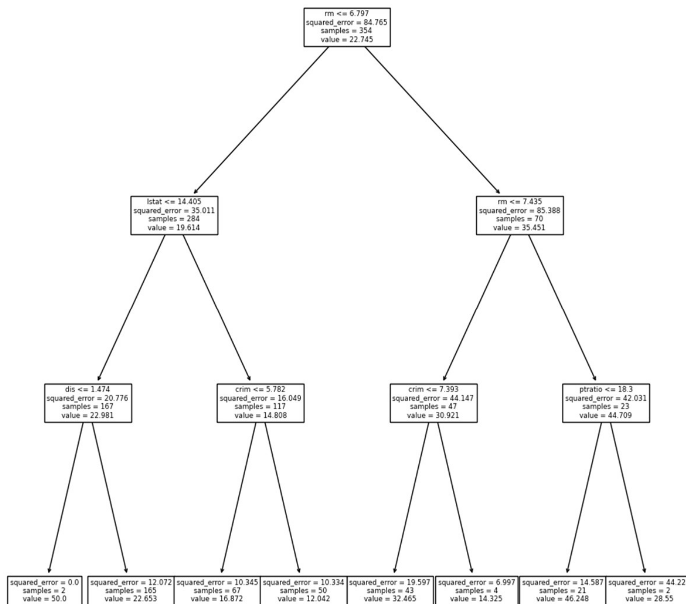
```
28.069857549754033
```

つまり、回帰木に関連するテストセット MSE は 28.07 となる。MSE の平方根は約 5.30 である。これは、このモデルが郊外の真の中央値の住宅価格から約 5300 ドル以内のテスト予測をもたらすことを示している。

最適な木をプロットして、その解釈可能性を確認しよう。

In [23]:

```
ax = subplots(figsize=(12,12))[1]
plot_tree(G.best_estimator_,
          feature_names=feature_names,
          ax=ax);
```



バギングとランダムフォレスト

ここでは、`sklearn.ensemble`パッケージの`RandomForestRegressor()`を使用して`Boston`データにバギングとランダムフォレストを適用する。バギングは、ランダムフォレストの特別なケースで、$m = p$である。したがって、`RandomForestRegressor()`関数はバギングとランダムフォレストの両方を実行できる。まずバギングから始める。

In [24]:

```
bag_boston = RF(max_features=X_train.shape[1], random_state=0)
bag_boston.fit(X_train, y_train)
```

Out[24]:

```
RandomForestRegressor(max_features=12, random_state=0)
```

RandomForestRegressor

```
RandomForestRegressor(max_features=12, random_state=0)
```

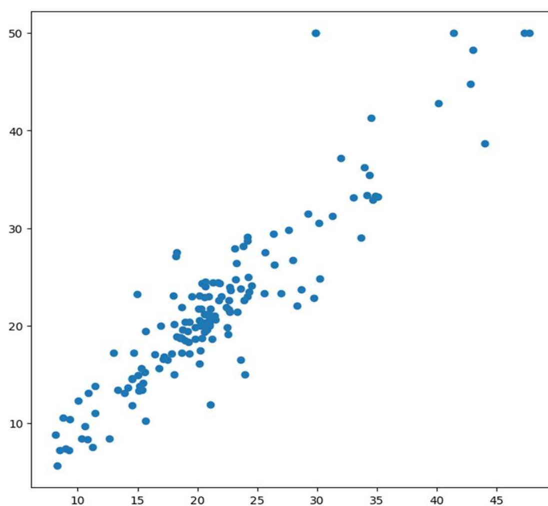
引数 `max_features` は、木の各分割に対してすべての 12 個の予測変数を考慮することを示している。つまり、バギングを行うことを意味する。このバギングモデルがテストセットでどれだけうまく機能するかを見てみよう。

In [25]:

```
ax = subplots(figsize=(8,8))[1]
y_hat_bag = bag_boston.predict(X_test)
ax.scatter(y_hat_bag, y_test)
np.mean((y_test - y_hat_bag)**2)
```

Out[25]:

```
14.634700151315787
```



バギング回帰木に関連するテストセット MSE は 14.63 で、最適に剪定された単一の木を使用した場合の約半分となる。デフォルトの 100 本の木から増やすことができる。

In [26]:

```
bag_boston = RF(max_features=X_train.shape[1],
                n_estimators=500,
                random_state=0).fit(X_train, y_train)
y_hat_bag = bag_boston.predict(X_test)
np.mean((y_test - y_hat_bag)**2)
```

Out[26]:

```
14.605662565263161
```

あまり変わらない。バギングとランダムフォレストは、木の数を増やしても過剰適合にはならず、数が少なすぎると適合不足になる可能性がある。

ランダムフォレストを成長させる方法はほぼ同じであるが、`max_features`引数の値を小さくする。デフォルトでは、`RandomForestRegressor()`は回帰木のランダムフォレストを構築するときにp個の変数を使用し（つまり、バギングにデフォルトする）、`RandomForestClassifier()`は分類木のランダムフォレストを構築するときに$\sqrt{p}$個の変数を使用する。ここでは`max_features=6`を使用する。

In [27]:

```
RF_boston = RF(max_features=6,
               random_state=0).fit(X_train, y_train)
y_hat_RF = RF_boston.predict(X_test)
np.mean((y_test - y_hat_RF)**2)
```

Out[27]:

```
20.04276446710527
```

テストセット MSE は 20.04 である。これは、この場合、ランダムフォレストがバギングよりもやや悪い結果を示したことを示している。適合されたモデルから`feature_importances_`値を抽出することで、各変数の重要性を確認できる。

In [28]:

```
feature_imp = pd.DataFrame(
    {'importance': RF_boston.feature_importances_},
    index=feature_names)
feature_imp.sort_values(by='importance', ascending=False)
```

Out[28]:

	Importance
Lstat	0.356203
Rm	0.332163
Ptratio	0.067270
Crim	0.055404
Indus	0.053851
Dis	0.041582
Nox	0.035225
Tax	0.025355
Age	0.021506
Rad	0.004784
Chas	0.004203
Zn	0.002454

これは、ノードの不純物の総減少を変数ごとの分割に対して平均した相対的な指標である（これは `Heart` データに適合したモデルに対して 8.9 図でプロットした）。

結果は、ランダムフォレストで検討されたすべての木において、コミュニティの富裕度（`lstat`）と住宅のサイズ（`rm`）が最も重要な 2 つの変数であることを示している。

ブースティング

ここでは、`sklearn.ensemble` の `GradientBoostingRegressor()`を使用して `Boston` データセットにブースティング回帰木を適合させよう。分類の場合は `GradientBoostingClassifier()`を使用する。引数 `n_estimators=5000` は 5000 本の木を使用することを示し、`max_depth=3` は各木の深さを制限する。引数 `learning_rate` は、ブースティングの説明で前述したλを指している。

In [29]:

```
boost_boston = GBR(n_estimators=5000,
                   learning_rate=0.001,
                   max_depth=3,
                   random_state=0)
boost_boston.fit(X_train, y_train)
```

Out[29]:

```
GradientBoostingRegressor(learning_rate=0.001, n_estimators=5000,
                          random_state=0)
```

GradientBoostingRegressor

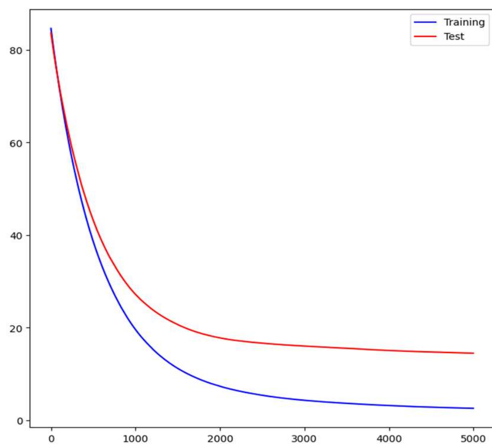
```
GradientBoostingRegressor(learning_rate=0.001, n_estimators=5000,
                          random_state=0)
```

`train_score_`属性を使用して、訓練誤差がどのように減少するかを確認できる。テスト誤差がどのように減少するかを把握するために、`staged_predict()`メソッドを使用して経路に沿った予測値が得られる。

In [30]:

```
test_error = np.zeros_like(boost_boston.train_score_)
for idx, y_ in enumerate(boost_boston.staged_predict(X_test)):
   test_error[idx] = np.mean((y_test - y_)**2)

plot_idx = np.arange(boost_boston.train_score_.shape[0])
ax = subplots(figsize=(8,8))[1]
ax.plot(plot_idx,
        boost_boston.train_score_,
        'b',
        label='Training')
ax.plot(plot_idx,
        test_error,
```

```
        'r',
        label='Test')
ax.legend();
```



次に、ブースティングモデルを使用してテストセットの `medv` を予測する。

In [31]:

```
y_hat_boost = boost_boston.predict(X_test);
np.mean((y_test - y_hat_boost)**2)
```

Out[31]:

```
14.481405918831591
```

得られたテスト MSE は 14.48 で、バギングのテスト MSE と似ている。ここで異なる収縮パラメータλを使用してブースティングを実行することもできる。デフォルト値は 0.001 であるが、これは簡単に変更できる。ここでは$\lambda = 0.2$を使用してみる。

In [32]:

```
boost_boston = GBR(n_estimators=5000,
                   learning_rate=0.2,
```

```
                    max_depth=3,
                    random_state=0)
boost_boston.fit(X_train, y_train)
y_hat_boost = boost_boston.predict(X_test);
np.mean((y_test - y_hat_boost)**2)
```

Out[32]:

```
14.501514553719565
```

この場合、$\lambda = 0.2$を使用すると、$\lambda = 0.001$を使用した場合とほぼ同じテスト MSE が得られる。

ベイジアン加法回帰木（BART）

このセクションでは、`ISLP.bart` パッケージにある BART の `Python` 実装を紹介する。`Boston` 住宅データセットにモデルを適合する。この `BART()`推定器は連続的な目的変数に対する回帰問題向けに設計されているが、ロジスティック回帰やプロビットモデルに適合する他の実装も利用可能である。

In [33]:

```
bart_boston = BART(random_state=0, burnin=5, ndraw=15)
bart_boston.fit(X_train, y_train)
```

Out[33]:

```
BART(burnin=5, ndraw=15, random_state=0)
```

BART

```
BART(burnin=5, ndraw=15, random_state=0)
```

このデータセットでは、テストセットとトレーニングセットに分割した結果、BART のテスト誤差はランダムフォレストのテスト誤差と類似している。

In [34]:

```
yhat_test = bart_boston.predict(X_test.astype(np.float32))
np.mean((y_test - yhat_test)**2)
```

Out[34]:

```
22.145009458109225
```

各変数が木の集合に登場した回数を確認できる。これは、ブースティングやランダムフォレストの変数重要度プロットに類似した結果となっている。

In [35]:

```
var_inclusion = pd.Series(bart_boston.variable_inclusion_.mean(0),
                                       index=D.columns)
var_inclusion
```

Out[35]:

```
crim       26.933333
zn         27.866667
indus      26.466667
chas       22.466667
nox        26.600000
rm         29.800000
age        22.733333
dis        26.466667
rad        23.666667
tax        24.133333
ptratio    24.266667
lstat      31.000000
dtype: float64
```

ISLP 第 9 章：サポートベクターマシン

このラボでは、`sklearn.svm` ライブラリを使用して サポートベクトル分類器（SVC）トベクトルマシン（SVM） の適用例を示す。

まず、必要な標準ライブラリをいくつかインポートしておく。

In [1]:

```
import numpy as np
from matplotlib.pyplot import subplots, cm
import sklearn.model_selection as skm
from ISLP import load_data, confusion_table
```

また、このラボで必要な追加のライブラリもまとめてインポートする。

In [2]:

```
from sklearn.svm import SVC
from ISLP.svm import plot as plot_svm
from sklearn.metrics import RocCurveDisplay
```

`RocCurveDisplay.from_estimator()`関数を使用して、いくつかの ROC プロットを作成する。簡略化のために `roc_curve` というショートカットを使用する。

In [3]:

```
roc_curve = RocCurveDisplay.from_estimator
```

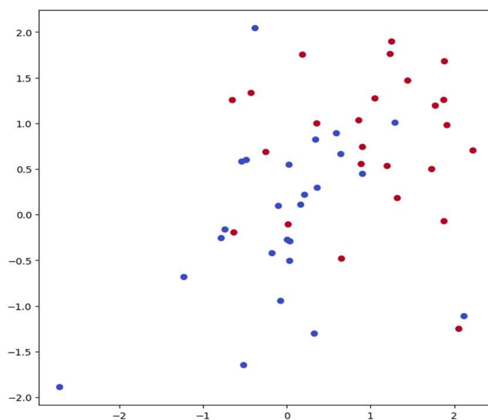
サポートベクトル分類器

ここでは、`sklearn` の `SupportVectorClassifier()`関数（略して `SVC()`）を使用して、パラメータ `C` の値に基づいてサポートベクトル分類器をデータにフィット（適合）する。`C` 引数はマージン違反のコストを指定するために使用する。`C` の値が小さいほど、分類マージンは広がり、多くのデータ点がマージン内またはマージンを超えて配置されるようになる。一方、`C` の値が大きいほど、マージンは狭まり、マージンを超えるサポートベクトルは少なくなる。

ここでは、2 次元の例を使用して `SVC()`の使用方法を示し、結果の決定境界をプロットしよう。まず、観測を生成し、それらのクラスが線形分離可能かどうかを確認する。

In [4]:

```
rng = np.random.default_rng(1)
X = rng.standard_normal((50, 2))
y = np.array([-1]*25+[1]*25)
X[y==1] += 1
fig, ax = subplots(figsize=(8,8))
ax.scatter(X[:,0],
           X[:,1],
           c=y,
           cmap=cm.coolwarm);
```



これらのデータは線形分離可能ではない。次に、分類器を学習させる。

In [5]:

```
svm_linear = SVC(C=10, kernel='linear')
svm_linear.fit(X, y)
```

Out[5]:

```
SVC(C=10, kernel='linear')
```

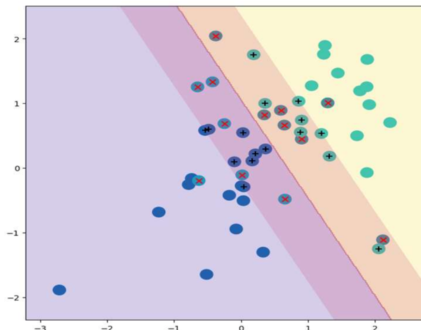
SVC

```
SVC(C=10, kernel='linear')
```

2 つの特徴量を持つサポートベクトル分類器は、*決定関数*の値をプロットすることで視覚化することができる。このための関数は `ISLP` パッケージに含まれており、`sklearn` のドキュメントに掲載されている同様の例を参考に作成した。

In [6]:

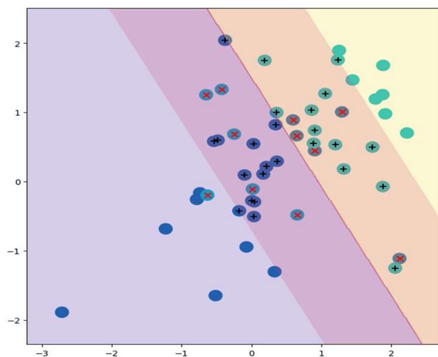
```
fig, ax = subplots(figsize=(8,8))
plot_svm(X,
         y,
         svm_linear,
         ax=ax)
```



2 つのクラス間の決定境界は線形である（`kernel='linear'`引数を使用したため）。サポートベクトルは`+`でマークされ、残りの観測は円でプロットされている。コストパラメータの値を小さくした場合はどうなるだろうか？

In [7]:

```
svm_linear_small = SVC(C=0.1, kernel='linear')
svm_linear_small.fit(X, y)
fig, ax = subplots(figsize=(8,8))
plot_svm(X,
         y,
         svm_linear_small,
         ax=ax)
```



コストパラメータの値を小さくすると、マージンが広くなり、多くのサポートベクトルが得られる。線形カーネルの場合、線形決定境界の係数を以下のように抽出できる。

In [8]:

```
svm_linear.coef_
```

Out[8]:

```
array([[1.17303943, 0.77348227]])
```

サポートベクトルマシンは`sklearn`による分類器であるため、通常のチューニング手法を使用できる。

In [9]:

```
kfold = skm.KFold(5,
                  random_state=0,
                  shuffle=True)
grid = skm.GridSearchCV(svm_linear,
                        {'C': [0.001, 0.01, 0.1, 1, 5, 10, 100]},
                        refit=True,
                        cv=kfold,
                        scoring='accuracy')
grid.fit(X, y)
grid.best_params_
```

Out[9]:

```
{'C': 1}
```

これらのモデルのクロスバリデーション誤差は`grid.cv_results_`で簡単に確認できる。出力の情報量が多いため、精度の結果のみを抽出する。

In [10]:

```
grid.cv_results_[('mean_test_score')]
```

Out[10]:

```
array([0.46, 0.46, 0.72, 0.74, 0.74, 0.74, 0.74])
```

クロスバリデーションの結果、`C=1`が最も良い精度（0.74）を示した。ただし、他のいくつかのC値でも同程度の精度が得られる。ここでは、`grid.best_estimator_`分類器を用いてテストデータのクラスラベルを予測しよう。まず、テストデータセットを生成する。

In [11]:

```
X_test = rng.standard_normal((20, 2))
y_test = np.array([-1]*10+[1]*10)
X_test[y_test==1] += 1
```

これよりテストデータのクラスラベルを予測する。クロスバリデーションで選択された最良のモデルを使用して予測を行う。

In [12]:

```
best_ = grid.best_estimator_
y_test_hat = best_.predict(X_test)
confusion_table(y_test_hat, y_test)
```

Out[12]:

True Predicted	-1	1
-1	8	4
1	2	6

この`C`の値を用いたモデルでは、テストデータの正解率が70%となった（混同行列から計算した）。`C=0.001`を使用した場合はどうだろうか？

In [13]:

```
svm_ = SVC(C=0.001, kernel='linear').fit(X, y)
y_test_hat = svm_.predict(X_test)
confusion_table(y_test_hat, y_test)
```

Out[13]:

True Predicted	-1	1
-1	2	0
1	8	10

この場合、テストデータの 60%が正しく分類された。

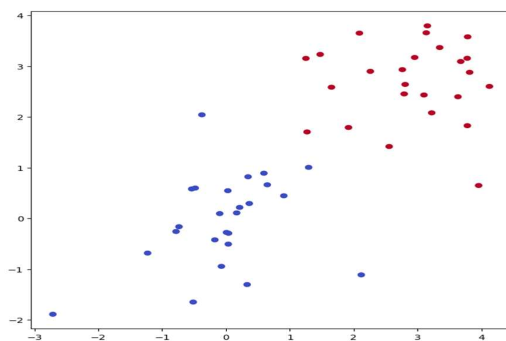
次に、2 つのクラスが線形分離可能な場合を考える。この状況では、`SVC()`推定器を用いて最適な分離超平面を求めることができる。まず、シミュレーションデータのクラス間の距離を広げ、線形分離が可能な状態にする。

In [14]:

```
X[y==1] += 1.9
fig, ax = subplots(figsize=(8,8))
ax.scatter(X[:,0], X[:,1], c=y, cmap=cm.coolwarm)
```

Out[14]:

```
<matplotlib.collections.PathCollection at 0x7fe9ddd2d2d0>
```



このデータの観測値はかろうじて線形分類可能だろう。

In [15]:

```
svm_ = SVC(C=1e5, kernel='linear').fit(X, y)
y_hat = svm_.predict(X)
confusion_table(y_hat, y)
```

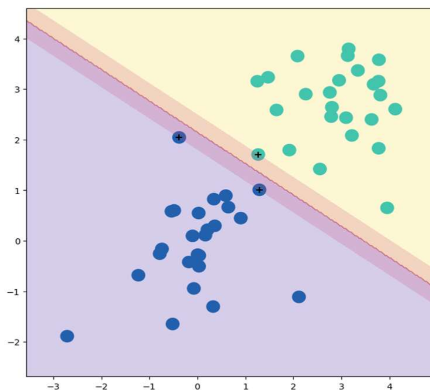
Out[15]:

True Predicted	-1	1
-1	25	0
1	0	25

非常に大きな`C`値を使用してサポートベクトル分類器を学習させ、決定境界をプロットしよう。この場合にはすべての観測が正しく分類されるようになった。

In [16]:

```
fig, ax = subplots(figsize=(8,8))
plot_svm(X, y, svm_, ax=ax)
```



実際に、訓練誤差はゼロであり、使用されたサポートベクトルは 3 個のみだった。大きな C 値を設定すると、これらの 3 つのサポートベクトルがマージン上に配置され、決定境界を定めることになる。しかし、わずか 3 つのデータポイントに依存する分類器の汎化性能には疑問が残るかもしれない。そこで、次に小さな C 値を試してみる。

In [17]:

```
svm_ = SVC(C=0.1, kernel='linear').fit(X, y)
y_hat = svm_.predict(X)
confusion_table(y_hat, y)
```

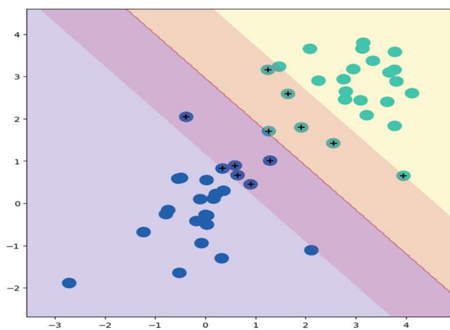
Out[17]:

True Predicted	-1	1
-1	25	0
1	0	25

`C=0.1` を使用すると、訓練データの誤分類は再び発生しないが、マージンが広がり、12 個のサポートベクトルが決定境界の形成に関与する。サポートベクトルの数が多いため、境界の安定性が向上する。このモデルは、`C=1e5` のモデルよりもテストデータで高い汎化性能を発揮する可能性があり（実際、大規模なテストセットでの簡単な実験によりこの傾向が確認できる）。

In [18]:

```
fig, ax = subplots(figsize=(8,8))
plot_svm(X, y, svm_, ax=ax)
```



サポートベクトルマシン

非線形カーネルを使用して SVM を学習させるために、再び `SVC()`推定器を使用しよう。ただし、今度はパラメータ `kernel` の値を変更する。多項式カーネルを使用する場合は `kernel="poly"`を使用し、ラジアルカーネルを使用する場合には

`kernel="rbf"`を使用する。前者の場合、`degree`引数を使用して多項式カーネルの次数を指定し（これは(9.22)のd）、後者の場合、`gamma`を使用してラジアルカーネルのγ値を指定する（(9.24)のγ）。

まず、非線形クラス境界を持つデータを生成する。

In [19]:

```
X = rng.standard_normal((200, 2))
X[:100] += 2
X[100:150] -= 2
y = np.array([1]*150+[2]*50)
```

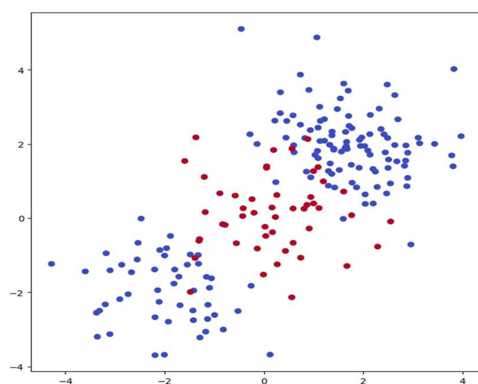
データをプロットすると、クラス境界が確かに非線形であることが明らかになる。

In [20]:

```
fig, ax = subplots(figsize=(8,8))
ax.scatter(X[:,0], X[:,1], c=y, cmap=cm.coolwarm)
```

Out[20]:

```
<matplotlib.collections.PathCollection at 0x7fe9be36b160>
```



データはランダムに訓練データとテストデータのグループに分割する。次に、訓練データを使用して、ラジアルカーネルと$\gamma = 1$の`SVC()`推定器を学習させる。

In [21]:

```
(X_train,
 X_test,
 y_train,
 y_test) = skm.train_test_split(X,
                                y,
                                test_size=0.5,
                                random_state=0)
svm_rbf = SVC(kernel="rbf", gamma=1, C=1)
svm_rbf.fit(X_train, y_train)
```

Out[21]:

```
SVC(C=1, gamma=1)
```

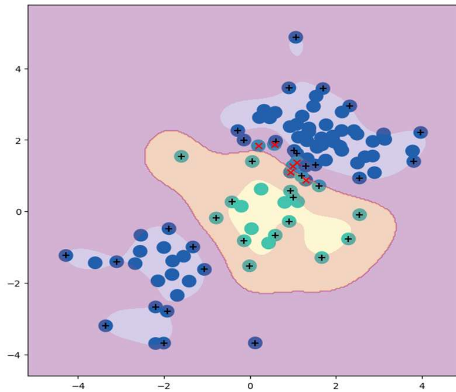
SVC

```
SVC(C=1, gamma=1)
```

データのプロットを見ると、学習された SVM の決定境界が明らかに非線形であることが分かる。

In [22]:

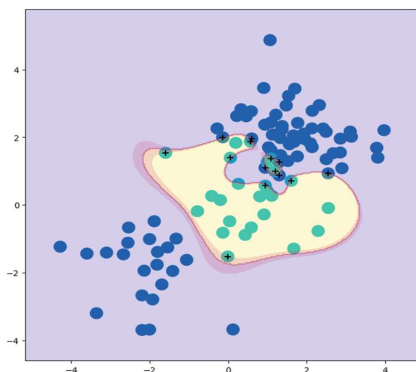
```
fig, ax = subplots(figsize=(8,8))
plot_svm(X_train,
         y_train,
         svm_rbf,
         ax=ax)
```



図から、この SVM モデルでは誤分類が比較的多いことが分かる。C の値を大きくすると誤分類を減らすことができるが、その代償として、より複雑で不規則な決定境界が形成され、データの過剰適合を引き起こすリスクが高まる。

In [23]:

```
svm_rbf = SVC(kernel="rbf", gamma=1, C=1e5)
svm_rbf.fit(X_train, y_train)
fig, ax = subplots(figsize=(8,8))
plot_svm(X_train,
         y_train,
         svm_rbf,
         ax=ax)
```



クロスバリデーションを利用するとラジアルカーネルを持つ SVM のγと C の最適な選択を行うことができる。

In [24]:

```
kfold = skm.KFold(5,
                  random_state=0,
                  shuffle=True)
grid = skm.GridSearchCV(svm_rbf,
                        {'C': [0.1, 1, 10, 100, 1000],
                         'gamma': [0.5, 1, 2, 3, 4]},
                        refit=True,
                        cv=kfold,
                        scoring='accuracy')
grid.fit(X_train, y_train)
grid.best_params_
```

Out[24]:

```
{'C': 1, 'gamma': 0.5}
```

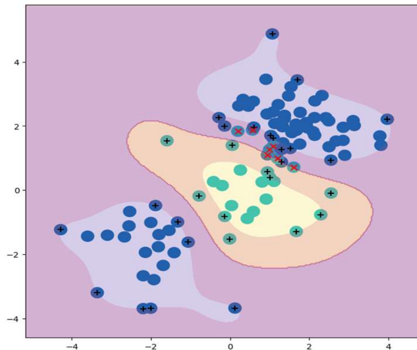
5 分割 CV の最適なパラメータの選択は、`C=1` および `gamma=0.5` であり、他のいくつかの値でも同じ精度を達成している。

In [25]:

```
best_svm = grid.best_estimator_
fig, ax = subplots(figsize=(8,8))
plot_svm(X_train,
         y_train,
         best_svm,
         ax=ax)

y_hat_test = best_svm.predict(X_test)
confusion_table(y_hat_test, y_test)
```

Out[25]:

True Predicted	1	2
1	69	6
2	6	19



この場合のパラメータによりテストデータの 12%が誤分類されていることが分かる。

ROC 曲線

SVM およびサポートベクトル分類器は、各観測値に対してクラスラベルを出力するだけでなく、数値スコア（適合値）を取得することも可能である。たとえば、サポートベクトル分類器において、観測値$X = (X_1, X_2, \ldots, X_p)^T$に対する適合値は$\hat{\beta}_0 + \hat{\beta}_1 X_1 + \hat{\beta}_2 X_2 + \cdots + \hat{\beta}_p X_p$で求めることができる。非線形カーネルを持つ SVM の場合、適合値は(9.23)で与えられる。適合値の符号により観測値が決定境界のどちら側に位置するかが決定される。したがって、特定の観測値に対する適合値とクラス予測の関係は単純であり、適合値が 0 を超える場合はあるクラスに、0 未満の場合は別のクラスに割り当てられる。さらに、閾値を 0 から正の値に変更すると、特定のクラスに対する分類のバイアスを調整できる。このように閾値を変化させることで、ROC 曲線のプロットに必要なデータを生成できる。これらの適合値は、学習済みの SVM 推定器の `decision_function()`メソッドを用いることで得られる。

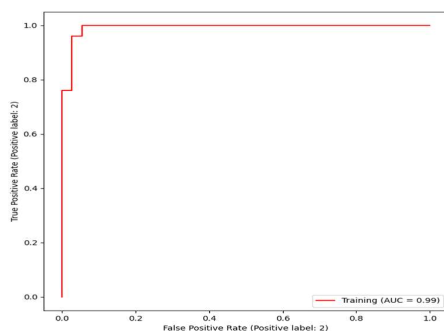
`ROCCurveDisplay.from_estimator()`関数（以下、`roc_curve`）により ROC 曲線がプロットされる。第一引数に学習済みの推定器を指定し、続いてモデルマトリックスXとラベル yを渡す。引数 `name` は凡例のラベルとして使用され、`color` は曲線の色を指定する。プロットは `ax` オブジェクト上に描画される。

In [26]:

```
fig, ax = subplots(figsize=(8,8))
roc_curve(best_svm,
          X_train,
          y_train,
          name='Training',
          color='r',
          ax=ax)
```

Out[26]:

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7fe9beb74670>
```



この例では、SVM の予測精度が高いことが示唆されている。γを大きくすると、モデルの柔軟性が向上し、さらなる精度向上が期待できよう。

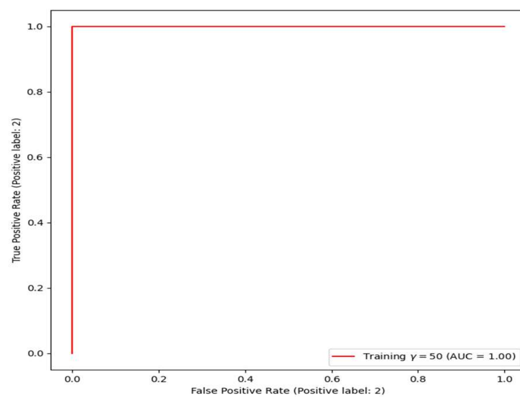
In [27]:

```
svm_flex = SVC(kernel="rbf",
              gamma=50,
              C=1)
svm_flex.fit(X_train, y_train)
fig, ax = subplots(figsize=(8,8))
roc_curve(svm_flex,
          X_train,
```

```
          y_train,
          name='Training $\gamma=50$',
          color='r',
          ax=ax)
```

Out[27]:

```
<sklearn.metrics._plot.roc_curve.RocCurveDisplay at 0x7fe9be62efb0>
```



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、モデルの汎化性能を評価するためにはテストデータにおける予測精度が重要である。テストデータに対して ROC 曲線を計算すると、$\gamma = 0.5$のモデルが最も高い分類精度を示していることが分かる。

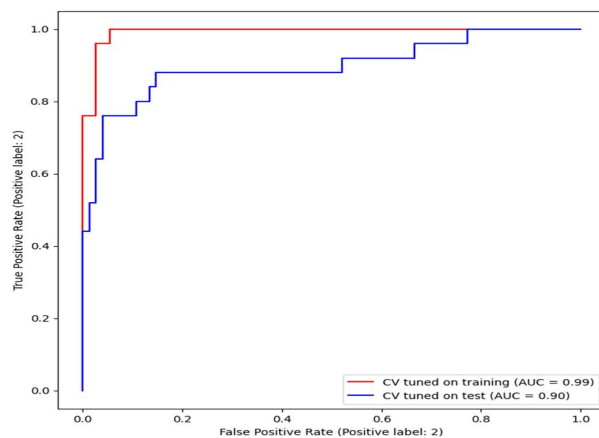
In [28]:

```
roc_curve(svm_flex,
          X_test,
          y_test,
          name='Test $\gamma=50$',
          color='b',
          ax=ax)
fig;
```

次にチューニングされた SVM を見てみよう。

In [29]:

```
fig, ax = subplots(figsize=(8,8))
for (X_, y_, c, name) in zip(
     (X_train, X_test),
     (y_train, y_test),
     ('r', 'b'),
     ('CV tuned on training',
      'CV tuned on test')):
    roc_curve(best_svm,
              X_,
              y_,
              name=name,
              ax=ax,
              color=c)
```



複数クラスを持つ SVM

応答変数が 3 つ以上のクラスを持つ場合、`SVC()`関数は `decision_function_shape='ovo'`（one-versus-one）または `decision_function_shape='ovr'`（one-versus-rest）を用いて多クラス分類を行うこ

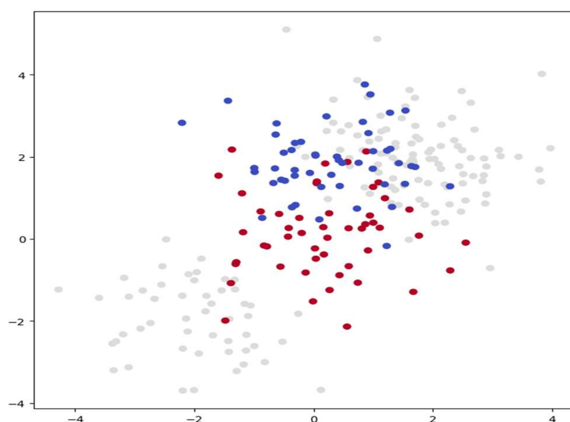
とができる。ここでは、新たに 3 番目のクラスのデータを生成し、その設定を簡単に確認しよう。

In [30]:

```
rng = np.random.default_rng(123)
X = np.vstack([X, rng.standard_normal((50, 2))])
y = np.hstack([y, [0]*50])
X[y==0,1] += 2
fig, ax = subplots(figsize=(8,8))
ax.scatter(X[:,0], X[:,1], c=y, cmap=cm.coolwarm)
```

Out[30]:

```
<matplotlib.collections.PathCollection at 0x7fe9dd8d8e50>
```

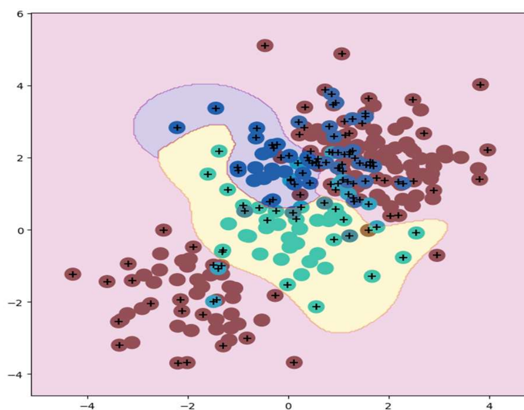


まずデータに SVM を学習させてみる。

In [31]:

```
svm_rbf_3 = SVC(kernel="rbf",
                C=10,
                gamma=1,
                decision_function_shape='ovo')
```

```
svm_rbf_3.fit(X, y)
fig, ax = subplots(figsize=(8,8))
plot_svm(X,
         y,
         svm_rbf_3,
         scatter_cmap=cm.tab10,
         ax=ax)
```



`sklearn.svm` ライブラリでは、連続値を予測するサポートベクトル回帰を `SupportVectorRegression()` 推定器で実行できる。

遺伝子発現データへの適用

ここでは、`Khan` データセットを調べてみよう。このデータセットは小円形青色細胞腫瘍の 4 つの異なるタイプに対応する多数の組織サンプルで構成されている。各組織サンプルには遺伝子発現測定値が利用可能である。データセットには、訓練データ `xtrain` と `ytrain`、およびテストデータ `xtest` と `ytest` が含まれている。

まずデータの次元を確認する。

In [32]:

```
Khan = load_data('Khan')
Khan['xtrain'].shape, Khan['xtest'].shape
```

Out[32]:

```
((63, 2308), (20, 2308))
```

このデータセットは 2,308 個の遺伝子の発現測定値で構成され、訓練データセットとテストデータセットはそれぞれ 63 と 20 の観測値で構成されている。遺伝子発現測定値を使用して癌のサブタイプを予測するためにサポートベクトルアプローチを利用しよう。このデータセットでは、観測数に対して特徴量の数が非常に多い。このことから多項式(ポリノミアル)またはラジアルカーネルを使用すると柔軟性が欠けるため、線形カーネルを利用すべきであることを示唆している。

In [33]:

```
khan_linear = SVC(kernel='linear', C=10)
khan_linear.fit(Khan['xtrain'], Khan['ytrain'])
confusion_table(khan_linear.predict(Khan['xtrain']),
                Khan['ytrain'])
```

Out[33]:

True Predicted	1	2	3	4
1	8	0	0	0
2	0	23	0	0
3	0	0	12	0
4	0	0	0	20

誤分類がまったくないことが分かる。実際、これは驚くべきことではない。なぜなら観測数に対して変数の数が非常に多い場合、クラスを完全に分離する超平面を見つけることは容易なのである。テストデータに対するサポートベクトル分類器の性能により関心が高い。

In [34]:

```
confusion_table(khan_linear.predict(Khan['xtest']),
                Khan['ytest'])
```

Out[34]:

True Predicted	1	2	3	4
1	3	0	0	0
2	0	6	2	0
3	0	0	4	0
4	0	0	0	5

このデータでは、`C=10` を使用した場合、テストセットで 2 つの誤分類が発生することが確認できる。

ISLP 第 10 章 深層学習(Deep Learning)

訳注１：Google Colaboratory を使って動作確認済みである (2025/02/26)。 第 10 章のみを Google Colaboratory で実行する場合は、`ISLP` と `torchinfo` をインストール必要がある。 例えば前者がインストールされてなければ 「１ ：`%%capture`
２： `!pip install ISLP` 」とすればよい。

訳注２：**プログラム上で必要に応じて（例えば In [62]: 付近）ISLP のホームページ https://www.statlearning.com/resources-python を参照し、Data Sets の中から適切なデータを入手し設定する必要がある。例えば Jyupyter を利用中に場所が分からなければコマンド (i) import os (ii) path=os.getcwd( ) (iii) print(path) とすると current directory が表示される。表示されている directory にファイルを置いておけばよい。Google Colaboratory (Colab)を利用する場合は少し複雑になる。まず Colab 上で (i) from google.colab import drive (ii) drive.mount('/content/drive') として(初回は)指示に従い Drive の利用を許可、パスを指定する。次にファイルを Google Drive（Google アプリの Google ドライブ上の MyDrive に Python と云う名のフォルダー）上に置く(Google メールに紐づけされている Google ドライブの操作に慣れておく必要がある）。最後に Colab 上で (iii) iimport os (iv) os.chdir("/content/drive/MyDrive/Python/") とすればよい。**

訳注 3: 深層学習における学習 or 推定は繰り返し最適化を行うので設定により時間がかかることがある。層の数や数値計算上のパラメターepoch などを変更すると計算時間は短くなることがある。

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「あたらしい機械学習の教科書」(伊藤真,翔泳社)などの説明が参考になる。

本章では、テキストで説明した例をフィットする方法を示す。`Python{}` の torch パッケージと、モデルの適合と評価を簡素化するユーティリティを提供する `pytorch_lightning` パッケージを使用する。このコードは、Apple の新しい M1 チップなどの特定のプロセッサでは非常に高速で動作することがあり得る。このパッケージはよく構造化されており、柔軟性があり、`Python{}` ユーザーには快適に感じられるはずである。良い参考書としては、pytorch.org/tutorials があり、多くのコードはそこから引用したものであり、`pytorch_lightning` のドキュメントも同様

である。執筆時点での正確な URL は、
https://pytorch.org/tutorials/beginner/basics/intro.html と https://pytorch-lightning.readthedocs.io/en/latest/

である。まず、以前と同様に標準的なインポートから始めよう。

In [1]:

```
import numpy as np, pandas as pd
from matplotlib.pyplot import subplots
from sklearn.linear_model import \
     (LinearRegression,
      LogisticRegression,
      Lasso)
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import KFold
from sklearn.pipeline import Pipeline
from ISLP import load_data
from ISLP.models import ModelSpec as MS
from sklearn.model_selection import \
     (train_test_split,
      GridSearchCV)
```

Torch インポート(Torch-Specific Imports)

`Torch` を動かすにはいくつかのライブラリをインポートする必要がある。（これらは `ISLP` に含まれていないため、別途インストールする必要がある。）まず、主要なライブラリと、順序構造を持つネットワークを指定するために使用される基本的なツールをインポートしておこう。

In [2]:

```
import torch
from torch import nn
from torch.optim import RMSprop
from torch.utils.data import TensorDataset
```

`torch`には他にもいくつかの補助(ヘルパー)パッケージがある。たとえば、`torchmetrics`パッケージは、モデルをフィットした時にパフォーマンスを評価するためのさまざまな量(メトリクス)を計算するためのユーティリティを提供してくれる。vtorchinfo `パッケージは、モデルの層(レイヤー)の有用なサマリーを提供してくれる。テスト画像をロードする際には、10.9.1 節の`read_image()` 関数を使用する。

もしまだ`torchvision`と`torchinfo`パッケージをインストールしていない場合は、`pip install torchinfo torchvision`を実行してインストールできる。それでは`torchinfo`からインポートを始めよう。

In [3]:

```
from torchmetrics import (MeanAbsoluteError,
                          R2Score)
from torchinfo import summary
```

`pytorch_lightning`パッケージは、`torch`の高レベルのインターフェースで、モデルの指定やフィットを簡素化する。これにより、必要な定型コードの量が減少し（`torch`単独で使用する場合と比較して）、モデルの作成やトレーニングがより簡単になる。

In [4]:

```
from pytorch_lightning import Trainer
from pytorch_lightning.loggers import CSVLogger
```

結果を再現するために、`seed_everything()`を利用しよう。また、可能な場合には`torch`に対して決定論的なアルゴリズムを使用するよう指示する。

In [5]:

```
from pytorch_lightning import seed_everything
seed_everything(0, workers=True)
torch.use_deterministic_algorithms(True, warn_only=True)
Seed set to 0
```

例として、torchvision に含まれているいくつかのデータセットを利用する。具体的には、画像分類のための事前学習済みネットワークと、前処理に使用するいくつかのトランスフォーム・パッケージである。

In [6]:

```
from torchvision.io import read_image
from torchvision.datasets import MNIST, CIFAR100
from torchvision.models import (resnet50,
                                ResNet50_Weights)
from torchvision.transforms import (Resize,
                                    Normalize,
                                    CenterCrop,
                                    ToTensor)
```

このラボのために、ISLP はいくつかのユーティリティを提供している。SimpleDataModule と SimpleModule は、pytorch_lightning で使用するプログラムのシンプルなバージョンで、torch モデルの適合を簡素化する高レベルのモジュールである。GPU での計算や並列データ処理など、より高度な機能もこのモジュールでは可能であるが、このラボではそれにはあまり焦点を当てないことにする。ErrorTracker により、バリデーションまたはテストの各段階でミニバッチごとにターゲットと予測を収集し、全体のバリデーションまたはテストデータセットに対してメトリクスを計算できるようにしている。

In [7]:

```
from ISLP.torch import (SimpleDataModule,
                        SimpleModule,
                        ErrorTracker,
                        rec_num_workers)
```

さらに、IMDb データベースをロードするためのいくつかのヘルパー関数と、整数をデータ・ベース内の特定のキーにマッピングするプログラムも含まれている。keras という深層学習モデルの適合用の別のパッケージから前処理された IMDb データの少し修正したコピーを含めている。これにより、かなりの前処理の手間が省け、モデルの設定とフィット合に集中することができる。

In [8]:

```
from ISLP.torch.imdb import (load_lookup,
                             load_tensor,
                             load_sparse,
                             load_sequential)
```

最後に、`torch` に直接関連しないいくつかのユーティリティのインポートを紹介しておく。`glob` モジュールの `glob()` 関数は、ワイルド・カード文字に一致するすべてのファイルを検索するために使用し、これを使って `ResNet50` モデルを自分の画像に適用する例で利用できる。また、`json` モジュールは、`ResNet50` の例で画像のラベルを特定するためにクラスを検索する JSON ファイルを読み込むために使用される。

In [9]:

```
from glob import glob
import json
```

打者の単一ネットワーク(Single Layer Network on Hitters Data)

まず、セクション 10.6 節で説明したモデルを `Hitters` データにフィットしよう。

In [10]:

```
Hitters = load_data('Hitters').dropna()
n = Hitters.shape[0]
```

2 つの線形モデル（最小二乗法とラッソ回帰）をフィットし、そのパフォーマンスをニューラル・ネットワーク・モデルのものと比較する。この比較では、バリデーション・データセットにおける平均絶対誤差（MAE）を利用する。

$$\mathrm{MAE}(y, \hat{y}) = (1/n) \sum_{i=1}^{n} | y(i) - \hat{y}(i) |$$

モデル行列と応答変数を設定する。

In [11]:

```
model = MS(Hitters.columns.drop('Salary'), intercept=False)
X = model.fit_transform(Hitters).to_numpy()
Y = Hitters['Salary'].to_numpy()
```

上記の `to_numpy()` 法は、`pandas` のデータフレームやシリーズを `numpy` 配列に変換する。この変換を行う理由は、ラッソ回帰モデルをフィットするために `sklearn` を使用する必要があり、`sklearn` は `numpy` 配列を要求するためである。また、比較を簡単にするために、`statsmodels` の第 3 章で述べた方法ではなく、`sklearn` の線形回帰法を利用する。ここでデータをテスト・セットとトレーニング・セットに分割し、`sklearn` が分割を行う際に使用するランダム状態を固定しておく。

In [12]:

```
(X_train,
 X_test,
 Y_train,
 Y_test) = train_test_split(X,
                            Y,
                            test_size=1/3,
                            random_state=1)
```

線形モデル(Linear Models)

線形モデルをフィット、テスト誤差を直接評価する。

In [13]:

```
hit_lm = LinearRegression().fit(X_train, Y_train)
Yhat_test = hit_lm.predict(X_test)
np.abs(Yhat_test - Y_test).mean()
```

Out[ ]:

```
259.71528833146243
```

次に、sklearn を使用して Lasso(ラッソ)回帰モデルをフィットする。モデルの選択と評価には平均絶対誤差（MAE）を使用するが、これは平均二乗誤差（MSE）ではない。6 章で利用したソルバーは平均二乗誤差のみを利用しているが、ここでは少し手間をかけてクロスバリデーション・グリッドを作成し、クロスバリデーションを直接実行する。

2 つのステップを含むパイプラインをエンコードする。まず、StandardScaler() トランスフォームを使用して特徴量を正規化し、その後、さらなる正規化なしでラッソ回帰モデルをフィットする。

In [14]:

```
scaler = StandardScaler(with_mean=True, with_std=True)
lasso = Lasso(warm_start=True, max_iter=30000)
standard_lasso = Pipeline(steps=[('scaler', scaler),
                                 ('lasso', lasso)])
```

ここでλの値のグリッドを作成する必要がある。一般的な方法としては lam_max から 0.01*lam_max まで、対数スケールで均等に 100 個の値を選択する。ここで、lam_max はすべての係数がゼロになる最小の λ の値である。この値は、任意の予測子と（中心化された）応答との内積の最大絶対値に等しい。ただしこの結果の導出は、本書の範囲を超えている。

In [15]:

```
X_s = scaler.fit_transform(X_train)
n = X_s.shape[0]
lam_max = np.fabs(X_s.T.dot(Y_train - Y_train.mean())).max() / n
param_grid = {'lasso__alpha': np.exp(np.linspace(0, np.log(0.01), 100))
             * lam_max}
```

変数のスケールが λ の選択に影響を与えるため、まずデータを変換する必要がある。これより λ の値の系列値を使用してクロス・バリデーションを実行する。

In [16]:

```
cv = KFold(10,
           shuffle=True,
           random_state=1)
grid = GridSearchCV(standard_lasso,
                    param_grid,
                    cv=cv,
                    scoring='neg_mean_absolute_error')
grid.fit(X_train, Y_train);
```

最良のクロス・バリデーションによる平均絶対誤差（MAE）を持つ Lasso（ラッソ）モデルを抽出し、クロスバリデーションで使用されなかった `X_test` と `Y_test` に対する性能を評価する。

In [17]:

```
trained_lasso = grid.best_estimator_
Yhat_test = trained_lasso.predict(X_test)
np.fabs(Yhat_test - Y_test).mean()
```

Out[ ]:

```
235.67548374780287
```

これは最小二乗法でフィットした線形モデルの結果と類似している。ただし、これらの結果は異なるトレイン/テストの分割によって大きく変動する可能性がある。そのため、読者にはコードブロック 12 で別のシードを試し、この時点までの後続のコードを再実行することを勧めておく。

ネットワークのクラスと継承(Specifying a Network: Classes and Inheritance)

ニューラルネットワークを適合させるために、まずネットワークを記述するモデル構造を設定する。それにフィットしたいモデルに特化した新しいクラスを定義

する必要がある。通常、`pytorch`では、ネットワークの一般的な表現をサブクラス化することで行う。ここでもこのアプローチを採用する。

この例はシンプルだが、次に続くより複雑な例にも役立つため、いくつかのステップをこまかく説明する。

In [18]:

```
class HittersModel(nn.Module):


    def __init__(self, input_size):
        super(HittersModel, self).__init__()
        self.flatten = nn.Flatten()
        self.sequential = nn.Sequential(
            nn.Linear(input_size, 50),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(50, 1))


    def forward(self, x):
        x = self.flatten(x)
        return torch.flatten(self.sequential(x))
```

`class`ステートメントは、`HittersModel`というクラスの宣言であり、`nn.Module`という基底クラスを継承している。この基底クラスは`torch`で広く使われており、ニューラル・ネットワーク・モデル内のマッピングを表現している。

`class`ステートメントの下は文字下げ（インデント）している。この場合、`__init__`と`forward`である。`__init__`法は、クラスの例が作成される際に呼ばれる（以下のセルで示すように）。方法内では`self`は常にクラスの例を示している。`__init__`法では、2 つのオブジェクトを`self`に属性として追加している：`flatten`と`sequential`。これらは`forward`法では、このモジュールが実装するマッピングを記述するために利用される。`__init__`法にはもう 1 行`super()`を呼び出す行がある。この関数は、サブクラス（つまり、`HittersModel`）が継承元のクラスの方法にアクセスできるようにしている。例えば、`nn.Module`クラスには独自の`__init__`法があり、上記で書いた`HittersModel.__init__()`法とは異なるのである。`super()`を使用することで、基底クラスのメソッドを呼び出すことができる。`torch`モデルでは、

モデルが適切に解釈するためにはこの `super()` 呼び出しを行う必要があるため、常にこうした設定をする必要がある。

`nn.Module` オブジェクトには、`__init__` と `forward` だけでなく、他にも多くの方法がある。これらは、継承のおかげで `HittersModel` を直接にアクセスできる。例としては、モデルをテストデータで評価したいときにドロップアウトを無効にするために使う `eval()` 法などである。

In [19]:

```
hit_model = HittersModel(X.shape[1])
```

`self.sequential` オブジェクトは、4 つのマップの合成である。最初のマップは、`Hitters` の 19 個の特徴量を 50 次元に変換する。このマップの重みと切片（通常はバイアスと呼ばれる）には、50 x 19 + 50 のパラメータが必要となる。この層は次に、ReLU 層にマッピングされ、続いて 40%のドロップアウト層、最後に 1 次元への線形マップがあり、これよりバイアスが生じる。総トレーニング可能パラメータ数は、50 x 19 + 50 + 50 + 1 = 1051 となる。

`torchinfo` パッケージには、ネットワークを通過する各テンソルのサイズを示す `summary()` 関数がある。入力のサイズを指定すると、ネットワークの層を通過する際の各テンソルのサイズを確認できる。

In [20]:

```
summary(hit_model,
        input_size=X_train.shape,
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

Out[ ]:

```
===================================================================================================
================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param
#
===================================================================================================
================
```

```
HittersModel                            [175, 19]                 [175]                     --
├─Flatten: 1-1                          [175, 19]                 [175, 19]                 --
├─Sequential: 1-2                       [175, 19]                 [175, 1]                  --
│    └─Linear: 2-1                      [175, 19]                 [175, 50]                 1,000
│    └─ReLU: 2-2                        [175, 50]                 [175, 50]                 --
│    └─Dropout: 2-3                     [175, 50]                 [175, 50]                 --
│    └─Linear: 2-4                      [175, 50]                 [175, 1]                  51
===================================================================================================
================
Total params: 1,051
Trainable params: 1,051
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 0.18
===================================================================================================
================
Input size (MB): 0.01
Forward/backward pass size (MB): 0.07
Params size (MB): 0.00
Estimated Total Size (MB): 0.09
===================================================================================================
================
```

出力の最後は少し省略しているが、ここで説明した内容はこれから利用する場合も同様である。

次に、トレーニングデータを `torch` がアクセスできる形式に変換する必要がある。`torch` の基本的なデータ型は `tensor` で、これは初期の章で扱った `ndarray` とよく似ている。また、`torch` は通常、64 ビット（*倍精度*）の浮動小数点数ではなく、32 ビット（*単精度*）の浮動小数点数で動作することに注意しておこう。そのため、データをテンソルに変換する前に `np.float32` に変換しておく。

次に、X と Y のテンソルを `TensorDataset()` を使用して `torch` が認識できる形式の `Dataset` に配置する。

In [21]:

```
X_train_t = torch.tensor(X_train.astype(np.float32))
Y_train_t = torch.tensor(Y_train.astype(np.float32))
hit_train = TensorDataset(X_train_t, Y_train_t)
```

テストデータについても同様の処理を行う。具体的には、テストデータを np.float32 型に変換し、その後 TensorDataset() を使って torch が認識できる形式に変換する。

In [22]:

```
X_test_t = torch.tensor(X_test.astype(np.float32))
Y_test_t = torch.tensor(Y_test.astype(np.float32))
hit_test = TensorDataset(X_test_t, Y_test_t)
```

最後に、このデータセットは DataLoader() に渡され、最終的にネットワークにデータが入力される。この処理は一見多くの無駄があるように思えるかもしれないが、データが異なるマシンに存在する場合や、データを GPU に渡さなければならない場合などには非常に役立つ。

標準的な使用のために、このタスクを簡単にするために ISLP パッケージには SimpleDataModule() というヘルパー関数が提供されている。この関数の引数の 1 つに num_workers があり、データをロードするために使用するプロセスの数を示している。Hitters のような小さなデータでは効果は少ないが、下記の MNIST や CIFAR100 の例ではかなりのメリットがある。

torch パッケージは実行中のプロセスを確認し、最大のワーカー数を決定する（これはコンピュータのハードウェアと使用可能なコア数に依存する）。この最大値を知るためには rec_num_workers() 関数が含まれているが、最大値は 16 であった。

In [23]:

```
max_num_workers = rec_num_workers()
```

pytorch_lightning の一般的なトレーニング設定では、トレーニング、検証、テストデータがそれぞれ異なるデータ入力として表す。各エポックで、モデルを学習するためのトレーニング・ステップと、誤差を追跡するための検証ステップを実行する。テストデータは通常、トレーニングが終了した後にモデルを評価するために使用する。

この場合、テストとトレーニングだけに分割したので、validation=hit_test の引数を使ってテストデータを検証データとして使用する。validation 引数は 0 と 1 の間の浮動小数点数、整数、または Dataset のいずれかで指定できる。浮動小数点数（または整数）が指定された場合、それを検証に使用する*トレーニング*観測の

割合（または数）として解釈する。もし`Dataset`が指定されれば、それは直接データ・ローダーに渡される。

In [24]:

```
hit_dm = SimpleDataModule(hit_train,
                          hit_test,
                          batch_size=32,
                          num_workers=min(4, max_num_workers),
                          validation=hit_test)
```

次に、トレーニングプロセス中に実行されるステップを制御する`pytorch_lightning`モジュールを提供する必要がある。`SimpleModule()`に対して、各エポックの終了時に損失関数の値や追加のメトリクスを記録する方法を提供する。これらの操作は、`SimpleModule.[training/test/validation]_step()`法によって制御されるが、これらの方法はこの例では特に変更しない。

In [25]:

```
hit_module = SimpleModule.regression(hit_model,
                           metrics={'mae':MeanAbsoluteError()})
```

`SimpleModule.regression()`法を使用することは、平方誤差損失（10.23）式を利用することを示している。また、ログに記録されるメトリクスとして、平均絶対誤差（mean absolute error）も追跡するようにしている。

結果は`CSVLogger()`を通じて記録されている。この場合、結果は`logs/hitters`ディレクトリ内のCSVファイルに保存される。フィッティングが完了した後、これにより結果は`pd.DataFrame()`として読み込み、以下で可視化することができる。`pytorch_lightning`には結果を記録するためのいくつかの方法があるが、ここではそれらを詳細に説明しないことにする。

In [26]:

```
hit_logger = CSVLogger('logs', name='hitters')
```

最後に、モデルをトレーニングし、結果を記録する準備が整ったが、
`pytorch_lightning` の `Trainer()` オブジェクトを利用した。`datamodule=hit_dm` 引数は、トレーニング/検証/テストログがどのように生成されるかをトレーナーに指示し、最初の引数 `hit_module` はネットワーク・アーキテクチャとトレーニング/検証/テストのステップを指定する。

`callbacks` 引数を使うことで、モデルのトレーニング中のさまざまなポイントで複数のタスクを実行できる。ここでは `ErrorTracker()` コールバックを使用して、トレーニング中に検証誤差を計算し、最終的にはテスト誤差を計算できるようにする。

これで 50 エポックの間モデルをフィットする。

In [27]:

```
hit_trainer = Trainer(deterministic=True,
                      max_epochs=50,
                      log_every_n_steps=5,
                      logger=hit_logger,
                      callbacks=[ErrorTracker()])
hit_trainer.fit(hit_module, datamodule=hit_dm)
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

  | Name  | Type         | Params
---------------------------------------
0 | model | HittersModel | 1.1 K
1 | loss  | MSELoss      | 0
---------------------------------------
1.1 K     Trainable params
0         Non-trainable params
1.1 K     Total params
0.004     Total estimated model params size (MB)
Epoch 0: 100%|██████████████████████████████████████████████████████████████| 6/6 [0
0:01<00:00,  4.03it/s, v_num=7]
Validation: |
         | 0/? [00:00<?, ?it/s]
Validation:   0%|
```

```
        | 0/3 [00:00<?, ?it/s]
Validation DataLoader 0:   0%|
        | 0/3 [00:00<?, ?it/s]
Validation DataLoader 0:  33%|███████████████████████
| 1/3 [00:00<00:00, 834.19it/s]
Validation DataLoader 0:  67%|██████████████████████████████████████████████
| 2/3 [00:00<00:00, 567.56it/s]
Validation DataLoader 0: 100%|██████████████████████████████████████████████████████████████████████
| 3/3 [00:00<00:00, 189.22it/s]
Epoch 1: 100%|█████████████████████████████████████████████████████████████████████████████| 6/6 [0
0:00<00:00, 228.62it/s, v_num=7]
Validation: |
        | 0/? [00:00<?, ?it/s]
Validation:   0%|
        | 0/3 [00:00<?, ?it/s]
Validation DataLoader 0:   0%|
        | 0/3 [00:00<?, ?it/s]
Validation DataLoader 0:  33%|███████████████████████
| 1/3 [00:00<00:00, 867.31it/s]
Validation DataLoader 0:  67%|██████████████████████████████████████████████
| 2/3 [00:00<00:00, 690.93it/s]
Validation DataLoader 0: 100%|██████████████████████████████████████████████████████████████████████
| 3/3 [00:00<00:00, 571.77it/s]
・・・

（一部の出力を省略）

・・・

Epoch 49: 100%|████████████████████████████████████████████████████████████████████████████| 6/6 [0
0:00<00:00, 229.69it/s, v_num=7]
Validation: |
        | 0/? [00:00<?, ?it/s]
Validation:   0%|
        | 0/3 [00:00<?, ?it/s]
Validation DataLoader 0:   0%|
        | 0/3 [00:00<?, ?it/s]
Validation DataLoader 0:  33%|███████████████████████
| 1/3 [00:00<00:00, 749.12it/s]
Validation DataLoader 0:  67%|██████████████████████████████████████████████
| 2/3 [00:00<00:00, 569.65it/s]
Validation DataLoader 0: 100%|██████████████████████████████████████████████████████████████████████
| 3/3 [00:00<00:00, 583.54it/s]
```

```
Epoch 49: 100%|████████████████████████████████████████████████████████████| 6/6 [0
0:00<00:00, 148.84it/s, v_num=7]
`Trainer.fit` stopped: `max_epochs=50` reached.
Epoch 49: 100%|████████████████████████████████████████████████████████████| 6/6 [0
0:00<00:00, 131.20it/s, v_num=7]
```

SGDの各ステップでは、アルゴリズムによりランダムに32のトレーニング観測値を選択して勾配の計算を行う。10.7節を思い出してみよう。1エポックは、nの観測値を処理するのに必要なSGDステップの回数である。トレーニング・セットにはn = 175があり、`hit_dm`の構築時に`batch_size`を32に指定したため、1エポックは175 / 32 = 5.5のSGDステップになる。

モデルをフィットした後、`test()`法を使用してテスト・データに対するパフォーマンスを評価する。

In [28]:

```
hit_trainer.test(hit_module, datamodule=hit_dm)
Testing DataLoader 0: 100%|████████████████████████████████████████████████
████████████████████| 3/3 [00:00<00:00, 318.37it/s]
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃        Test metric         ┃        DataLoader 0        ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│         test_loss          │      107904.6484375        │
│          test_mae          │     221.83148193359375     │
└────────────────────────────┴────────────────────────────┘
```

Out[ ]:

```
[{'test_loss': 107904.6484375, 'test_mae': 221.83148193359375}]
```

フィットの結果はCSVファイルに記録されており、`experiment.metrics_file_path`属性を使用して、この実行に関連する結果を確認できる。モデルがフィットするたびに、ロガーは`logs/hitters`ディレクトリ内の新しいサブディレクトリに結果を出力している。

次に、エポック数に対するMAE（平均絶対誤差）のプロットを作成する。まず、記録されたサマリーを得ておこう。

In [29]:

```
hit_results = pd.read_csv(hit_logger.experiment.metrics_file_path)
```

後の例でも似たようなプロットを作成するので、このプロットを生成する為のシンプルで汎用的な関数を作成する。

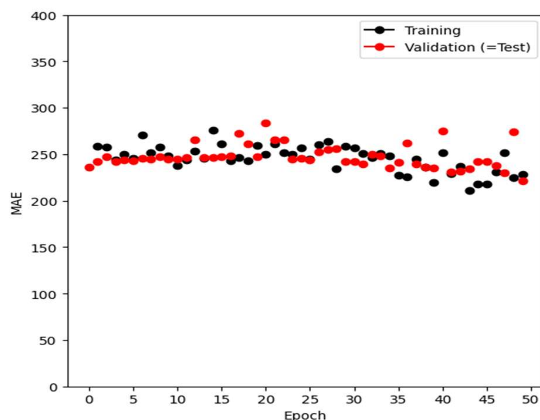
In [30]:

```
def summary_plot(results,
                 ax,
                 col='loss',
                 valid_legend='Validation',
                 training_legend='Training',
                 ylabel='Loss',
                 fontsize=20):
    for (column,
         color,
         label) in zip([f'train_{col}_epoch',
                        f'valid_{col}'],
                       ['black',
                        'red'],
                       [training_legend,
                        valid_legend]):
        results.plot(x='epoch',
                     y=column,
                     label=label,
                     marker='o',
                     color=color,
                     ax=ax)
    ax.set_xlabel('Epoch')
    ax.set_ylabel(ylabel)
    return ax
```

次に、軸を設定し、作成した関数を使用して MAE のプロットを生成する。

In [31]:

```
fig, ax = subplots(1, 1, figsize=(6, 6))
ax = summary_plot(hit_results,
                  ax,
                  col='mae',
                  ylabel='MAE',
                  valid_legend='Validation (=Test)')
ax.set_ylim([0, 400])
ax.set_xticks(np.linspace(0, 50, 11).astype(int));
```



最終的なモデルから直接に予測を行い、テスト・データでのパフォーマンスを評価する。

フィットする前に、`hit_model` の `eval()`法を呼び出す。このことにより、`torch` はモデルがフィットされたものとみなして、新しいデータに対して予測を行う準備する。ここでの最も重要な変更は、ドロップアウト層が無効になることである。つまり、新しいデータに対する予測時には、ランダムに重みが削除されことはない。

In [32]:

```
hit_model.eval()
preds = hit_module(X_test_t)
torch.abs(Y_test_t - preds).mean()
```

Out[ ]:

```
tensor(221.8315, grad_fn=<MeanBackward0>)
```

解除(Cleanup)

データ・モジュールを設定する際に、いくつかのワーカープロセスを開始してプロセスは実行され続ける。これらのプロセスが終了するには、`torch` オブジェクトへのすべての既存の参照を削除する必要がある。

In [33]:

```
del(Hitters,
    hit_model, hit_dm,
    hit_logger,
    hit_test, hit_train,
    X, Y,
    X_test, X_train,
    Y_test, Y_train,
    X_test_t, Y_test_t,
    hit_trainer, hit_module)
```

MNIST データ多層ネットワーク(Multilayer Network on the MNIST Digit Data)

`torchvision` パッケージには、`MNIST` の手書き数字データを含むいくつかの例示となるデータセットが付属している。最初のステップは、`MNIST()` 関数を使用して、トレーニングおよびテストデータセットを取得することである。この関数を初めて実行すると、データは自動的にダウンロードされ、`data/MNIST` ディレクトリに保存される。

In [34]:

```
(mnist_train,
 mnist_test) = [MNIST(root='data',
                      train=train,
                      download=True,
                      transform=ToTensor())
                for train in [True, False]]
mnist_train
```

Out[ ]:

```
Dataset MNIST
    Number of datapoints: 60000
    Root location: data
    Split: Train
    StandardTransform
Transform: ToTensor()
```

訓練データには 60,000 枚の画像があり、テストデータには 10,000 枚の画像がある。画像は 28x 28 のサイズで、ピクセルの行列として保存されている。これらの画像を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・モデルは入力データのスケールに敏感であり、リッジ回帰やラッソ回帰と同様にスケーリングが影響を与えることがある。ここでの入力は 8 ビットのグレースケール値で、範囲は 0 から 255 です。したがって、これらの値を単位区間[0, 1]に変換する必要がある。

(注: 8 ビットは 2^8 に相当し、256 となる。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。)

この変換と軸の並べ替えは、`torchvision.transforms` パッケージの `ToTensor()`変換によって行われる。

`Hitters` の例と同様に、訓練データとテストデータからデータ・モジュールを作成し、訓練画像の 20%を検証用に分ける。

In [35]:

```
mnist_dm = SimpleDataModule(mnist_train,
                            mnist_test,
                            validation=0.2,
                            num_workers=max_num_workers,
                            batch_size=256)
```

ネットワークに入力されるデータを見てみよう。テスト・データセットの最初の数バッチをループ処理し、2 バッチ後に停止する。

In [36]:

```
for idx, (X_ ,Y_) in enumerate(mnist_dm.train_dataloader()):
    print('X: ', X_.shape)
    print('Y: ', Y_.shape)
    if idx >= 1:
        break
X:  torch.Size([256, 1, 28, 28])
Y:  torch.Size([256])
X:  torch.Size([256, 1, 28, 28])
Y:  torch.Size([256])
```

各バッチの X は、`1x28x28` のサイズを持つ 256 枚の画像で構成されていることが分る。ここでの `1` は単一のチャネル（グレースケール）を示している。`CIFAR100` のような RGB 画像の場合、サイズの `1` は 3 つの RGB チャネルのために `3` に置き換わる。

これで、ニューラル・ネットワーク・モデルを設定する準備が整った。

In [37]:

```
class MNISTModel(nn.Module):
    def __init__(self):
        super(MNISTModel, self).__init__()
        self.layer1 = nn.Sequential(
            nn.Flatten(),
```

```
            nn.Linear(28*28, 256),
            nn.ReLU(),
            nn.Dropout(0.4))
        self.layer2 = nn.Sequential(
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.3))
        self._forward = nn.Sequential(
            self.layer1,
            self.layer2,
            nn.Linear(128, 10))
    def forward(self, x):
        return self._forward(x)
```

最初の層では、各 `1x28x28` の画像が平坦化され、次に 256 次元にマッピングされ、ReLU 活性化関数と 40%のドロップアウトが適用される。次の層では、最初の層の出力が 128 次元にマッピングされ、ReLU 活性化関数と 30%のドロップアウトが適用される。最後に、128 次元は `MNIST` データのクラス数である 10 次元にマッピングされる。

In [38]:

```
mnist_model = MNISTModel()
```

既存のバッチ `X_` を基に、モデルが期待されるサイズの出力を生成していることが確認できる。

In [39]:

```
mnist_model(X_).size()
```

Out[ ]:

```
torch.Size([256, 10])
```

モデルのサマリーを確認してみよう。`input_size`の代わりに、正しい形状のテンソルを渡すことができる。この場合、上記の最終的なバッチ`X_`を渡している。

In [40]:

```
summary(mnist_model,
        input_data=X_,
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

Out[ ]:

```
====================================================================================================
================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param
#
====================================================================================================
================
MNISTModel                               [256, 1, 28, 28]          [256, 10]                 --
├─Sequential: 1-1                        [256, 1, 28, 28]          [256, 10]                 --
│    └─Sequential: 2-1                   [256, 1, 28, 28]          [256, 256]                --
│    │    └─Flatten: 3-1                 [256, 1, 28, 28]          [256, 784]                --
│    │    └─Linear: 3-2                  [256, 784]                [256, 256]                200,96
0
│    │    └─ReLU: 3-3                    [256, 256]                [256, 256]                --
│    │    └─Dropout: 3-4                 [256, 256]                [256, 256]                --
│    └─Sequential: 2-2                   [256, 256]                [256, 128]                --
│    │    └─Linear: 3-5                  [256, 256]                [256, 128]                32,896
│    │    └─ReLU: 3-6                    [256, 128]                [256, 128]                --
│    │    └─Dropout: 3-7                 [256, 128]                [256, 128]                --
│    └─Linear: 2-3                       [256, 128]                [256, 10]                 1,290
====================================================================================================
================
Total params: 235,146
Trainable params: 235,146
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 60.20
====================================================================================================
================
Input size (MB): 0.80
```

```
Forward/backward pass size (MB): 0.81
Params size (MB): 0.94
Estimated Total Size (MB): 2.55
===================================================================================================
================
```

モデルとデータ・モジュールの両方を設定したので、このモデルのフィットの作業はほぼ `Hitters` の例と同じになる。回帰モデルとは異なり、ここでは交差エントロピー損失関数を使用する `SimpleModule.classification()`法を利用する。その為には問題のクラス数を指定する必要がある。

In [41]:

```
mnist_module = SimpleModule.classification(mnist_model,
                                           num_classes=10)
mnist_logger = CSVLogger('logs', name='MNIST')
```

これで準備が整った。最後のステップは、訓練データを与えてモデルをフィットすることである。以下では、実行時にブラウザで長時間の出力が表示されるのを避けるため、進行状況バーを無効にしておく。

In [42]:

```
mnist_trainer = Trainer(deterministic=True,
                        max_epochs=30,
                        logger=mnist_logger,
                        enable_progress_bar=False,
                        callbacks=[ErrorTracker()])
mnist_trainer.fit(mnist_module,
                  datamodule=mnist_dm)
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

  | Name  | Type             | Params
-------------------------------------------
```

```
0 | model | MNISTModel       | 235 K
1 | loss  | CrossEntropyLoss | 0
-------------------------------------------
235 K     Trainable params
0         Non-trainable params
235 K     Total params
0.941     Total estimated model params size (MB)
`Trainer.fit` stopped: `max_epochs=30` reached.
```

ここでは出力を抑制しているが、これによりモデル・フィットに関する進行状況報告で、エポックごとにグループ化される。これは非常に便利で、大きなデータセットではフィットに時間がかかることがある。このモデル・フィットには、Apple M1 Pro チップ（10 コア、16GB の RAM 搭載）の MacBook Pro で 245 秒かかった。

ここでは、訓練セットの 60,000 件の観測データのうち、80%（48,000 件）で訓練が実行され、残りの 20%（12,000 件）が検証データとして使用される。これは、`Hitters` データに対して行ったように実際の検証データを供給する代わりの方法である。

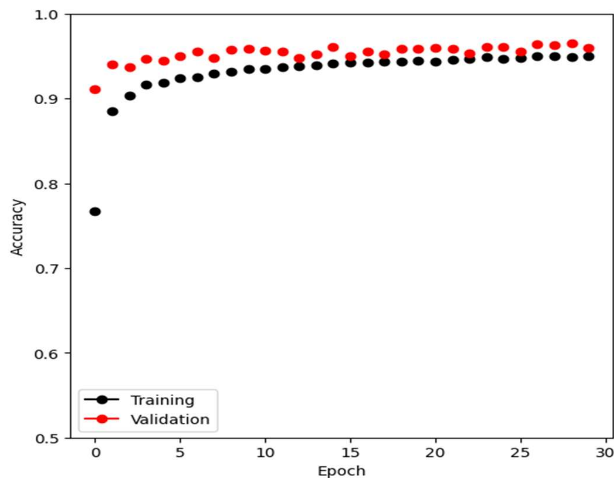
SGD は 256 件の観測データのバッチを使って勾配を計算します。算術的に計算すると、1 エポックは 188 回の勾配ステップに相当する。

`SimpleModule.classification()` にはデフォルトで精度メトリックが含まれ、他の分類メトリックは `torchmetrics` から追加できる。ここでは `summary_plot()` 関数を使用して、エポックごとの精度を表示する。

In [43]:

```
mnist_results = pd.read_csv(mnist_logger.experiment.metrics_file_path)
fig, ax = subplots(1, 1, figsize=(6, 6))
summary_plot(mnist_results,
             ax,
             col='accuracy',
             ylabel='Accuracy')
ax.set_ylim([0.5, 1])
```

```
ax.set_ylabel('Accuracy')
ax.set_xticks(np.linspace(0, 30, 7).astype(int));
```



再度、`test()` 法を用いて精度を評価しよう。このモデルはテストデータでおよそ97%の精度を達成している。

In [44]:

```
mnist_trainer.test(mnist_module,
                   datamodule=mnist_dm)
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃        Test metric         ┃        DataLoader 0        ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│       test_accuracy        │     0.9620000123977661     │
│         test_loss          │     0.15120187401771545    │
└────────────────────────────┴────────────────────────────┘
```

Out[ ]:

```
[{'test_loss': 0.15120187401771545, 'test_accuracy': 0.9620000123977661}]
```

表 10.1 では、LDA（4 章）と多項ロジスティック回帰による誤差率も報告している。LDA については、4 章を参照されたい。多項ロジスティック回帰をフィットさせるために `sklearn` の `LogisticRegression()` 関数を使用することもできるが、ここでは `torch` を利用してモデルをフィットする設定にしている。入力層と出力層だけを明示的、隠れ層は省略する！

In [45]:

```
class MNIST_MLR(nn.Module):
    def __init__(self):
        super(MNIST_MLR, self).__init__()
        self.linear = nn.Sequential(nn.Flatten(),
                                    nn.Linear(784, 10))
    def forward(self, x):
        return self.linear(x)

mlr_model = MNIST_MLR()
mlr_module = SimpleModule.classification(mlr_model,
                                         num_classes=10)
mlr_logger = CSVLogger('logs', name='MNIST_MLR')
```

In [46]:

```
mlr_trainer = Trainer(deterministic=True,
                      max_epochs=30,
                      enable_progress_bar=False,
                      callbacks=[ErrorTracker()])
mlr_trainer.fit(mlr_module, datamodule=mnist_dm)
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
/Users/jtaylo/anaconda3/envs/ISLP_v22_312/lib/python3.12/site-packages/pytorch_lightning/trainer/co
nnectors/logger_connector/logger_connector.py:75: Starting from v1.9.0, `tensorboardX` has been rem
oved as a dependency of the `pytorch_lightning` package, due to potential conflicts with other pack
ages in the ML ecosystem. For this reason, `logger=True` will use `CSVLogger` as the default logge
```

```
r, unless the `tensorboard` or `tensorboardX` packages are found. Please `pip install lightning[ext
ra]` or one of them to enable TensorBoard support by default

  | Name  | Type             | Params
-------------------------------------------
0 | model | MNIST_MLR        | 7.9 K
1 | loss  | CrossEntropyLoss | 0
-------------------------------------------
7.9 K     Trainable params
0         Non-trainable params
7.9 K     Total params
0.031     Total estimated model params size (MB)
`Trainer.fit` stopped: `max_epochs=30` reached.
```

モデルは前と同様にフィットし、テスト結果を計算する。

In [47]:

```
mlr_trainer.test(mlr_module,
                 datamodule=mnist_dm)
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃        Test metric        ┃       DataLoader 0        ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│       test_accuracy       │     0.916100025177002     │
│         test_loss         │    0.3469300866127014     │
└───────────────────────────┴───────────────────────────┘
```

Out[ ]:

```
[{'test_loss': 0.3469300866127014, 'test_accuracy': 0.916100025177002}]
```

このかなりシンプルなモデルでも、精度は 90%以上を達成する。`Hitters` の例と同様に、上で作成したいくつかのオブジェクトは削除する。

In [48]:

```
del(mnist_test,
    mnist_train,
```

```
    mnist_model,
    mnist_dm,
    mnist_trainer,
    mnist_module,
    mnist_results,
    mlr_model,
    mlr_module,
    mlr_trainer)
```

畳み込みニューラルネット(Convolutional Neural Networks)

この節では `torchvision` パッケージで利用可能な `CIFAR100` データに CNN をフィットする。 データは `MNIST` データと似た形式で配置する。

In [49]:

```
(cifar_train,
 cifar_test) = [CIFAR100(root="data",
                         train=train,
                         download=True)
             for train in [True, False]]
Files already downloaded and verified
Files already downloaded and verified
```

In [50]:

```
transform = ToTensor()
cifar_train_X = torch.stack([transform(x) for x in
                            cifar_train.data])
cifar_test_X = torch.stack([transform(x) for x in
                            cifar_test.data])
cifar_train = TensorDataset(cifar_train_X,
```

```
                            torch.tensor(cifar_train.targets))
cifar_test = TensorDataset(cifar_test_X,
                            torch.tensor(cifar_test.targets))
```

`CIFAR100` データセットは、50,000 枚のトレーニング画像で構成されていて、各画像は三次元テンソルで表されている。各三色画像は、32x 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。データは、数字と同様に標準化するが、配列構造はそのままにしておく。これを `ToTensor()` 変換を利用して実現する。

データ・モジュールの作成は、`MNIST` の例と類似している。

In [51]:

```
cifar_dm = SimpleDataModule(cifar_train,
                            cifar_test,
                            validation=0.2,
                            num_workers=max_num_workers,
                            batch_size=128)
```

再び、データ・ローダー内の典型的なバッチの形状を確認する。

In [52]:

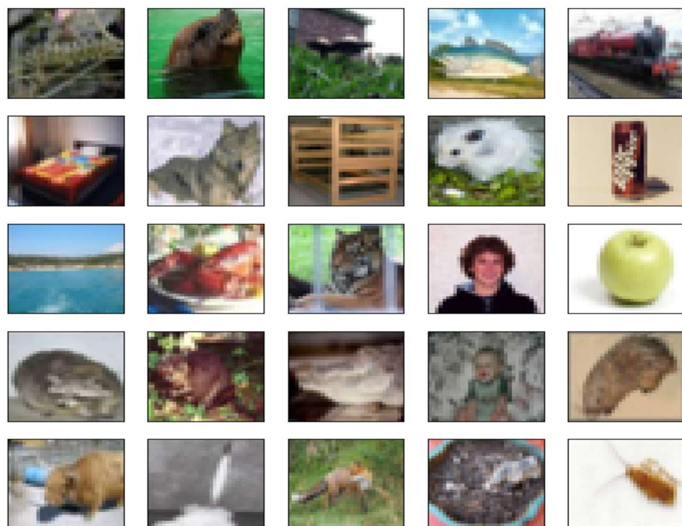
```
for idx, (X_ ,Y_) in enumerate(cifar_dm.train_dataloader()):
    print('X: ', X_.shape)
    print('Y: ', Y_.shape)
    if idx >= 1:
        break
X:  torch.Size([128, 3, 32, 32])
Y:  torch.Size([128])
X:  torch.Size([128, 3, 32, 32])
Y:  torch.Size([128])
```

始める前に、いくつかのトレーニング画像を見ておこう。似たようなコードにより 406 ページ の図 10.5 を作成できる。以下の例では、`TensorDataset` オブジェク

トが整数インデックス指定が可能なことを示している。ここでは、`cifar_train` をインデックス指定してトレーニングデータからランダムな画像を選んでいる。正しく表示するためには、`np.transpose()` を使って次元を再配置する必要がある。

In [53]:

```
fig, axes = subplots(5, 5, figsize=(10,10))
rng = np.random.default_rng(4)
indices = rng.choice(np.arange(len(cifar_train)), 25,
                     replace=False).reshape((5,5))
for i in range(5):
    for j in range(5):
        idx = indices[i,j]
        axes[i,j].imshow(np.transpose(cifar_train[idx][0],
                                      [1,2,0]),
                                      interpolation=None)
        axes[i,j].set_xticks([])
        axes[i,j].set_yticks([])
```



ここでは、`imshow()` 法が引数の形状から三次元の配列であり、最後の次元が 3 つの RGB カラー チャネルをインデックスしていることを認識している。

ここではデモンストレーションを行う目的で、図 10.8 と似た構造の適度なサイズの CNN を指定する。いくつかの層を使用し、それぞれが畳み込み、ReLU、最大プーリングのステップで構成されている。
まず、これらの層の 1 つを定義するモジュールを定義する。前の例と同様に、`nn.Module` の `__init__()` と `forward()` 法を書き直す。これでユーザー定義モジュールは、`nn.Linear()` や `nn.Dropout()` のように利用できるようになる。

In [54]:

```
class BuildingBlock(nn.Module):


    def __init__(self,
                 in_channels,
                 out_channels):


        super(BuildingBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels=in_channels,
                              out_channels=out_channels,
                              kernel_size=(3,3),
                              padding='same')
        self.activation = nn.ReLU()
        self.pool = nn.MaxPool2d(kernel_size=(2,2))


    def forward(self, x):
        return self.pool(self.activation(self.conv(x)))
```

ここで `nn.Conv2d()` に `padding = "same"` 引数を利用したことに注意おこう。これにより、出力チャネルが入力チャネルと同じ次元を持つことが保証される。最初の隠れ層には 32 チャネルがあり、入力層の 3 チャネルとは対照的となる。すべての層で、各チャネルに対して 3x 3 の畳み込みフィルターを利用する。各畳み込みの後には、2x 2 ブロックに対する最大プーリング層が続く。

`CIFAR100` データ用のディープラーニングモデルを構築する際には、いくつかの `BuildingBlock()` モジュールを順番に利用する。ここでのシンプルな例は、`torch` の強力さを示している。ユーザーは自分自身でモジュールを定義でき、それらを他のモジュールに組み合わせることができる。最終的には、すべてが汎用のトレーナーによってモデルをフィットできる。

In [55]:

```
class CIFARModel(nn.Module):

    def __init__(self):
        super(CIFARModel, self).__init__()
        sizes = [(3,32),
                 (32,64),
                 (64,128),
                 (128,256)]
        self.conv = nn.Sequential(*[BuildingBlock(in_, out_)
                                    for in_, out_ in sizes])

        self.output = nn.Sequential(nn.Dropout(0.5),
                                    nn.Linear(2*2*256, 512),
                                    nn.ReLU(),
                                    nn.Linear(512, 100))
    def forward(self, x):
        val = self.conv(x)
        val = torch.flatten(val, start_dim=1)
        return self.output(val)
```

モデルを構築し、そのサマリーを確認しよう。（以前に`X_`の例を作成した。）

In [56]:

```
cifar_model = CIFARModel()
summary(cifar_model,
        input_data=X_,
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

Out[ ]:

```
==================================================================================================
================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param
#
==================================================================================================
================
CIFARModel                               [128, 3, 32, 32]          [128, 100]                --
├─Sequential: 1-1                        [128, 3, 32, 32]          [128, 256, 2, 2]          --
│    └─BuildingBlock: 2-1                [128, 3, 32, 32]          [128, 32, 16, 16]         --
│    │    └─Conv2d: 3-1                  [128, 3, 32, 32]          [128, 32, 32, 32]         896
│    │    └─ReLU: 3-2                    [128, 32, 32, 32]         [128, 32, 32, 32]         --
│    │    └─MaxPool2d: 3-3               [128, 32, 32, 32]         [128, 32, 16, 16]         --
│    └─BuildingBlock: 2-2                [128, 32, 16, 16]         [128, 64, 8, 8]           --
│    │    └─Conv2d: 3-4                  [128, 32, 16, 16]         [128, 64, 16, 16]         18,496
│    │    └─ReLU: 3-5                    [128, 64, 16, 16]         [128, 64, 16, 16]         --
│    │    └─MaxPool2d: 3-6               [128, 64, 16, 16]         [128, 64, 8, 8]           --
│    └─BuildingBlock: 2-3                [128, 64, 8, 8]           [128, 128, 4, 4]          --
│    │    └─Conv2d: 3-7                  [128, 64, 8, 8]           [128, 128, 8, 8]          73,856
│    │    └─ReLU: 3-8                    [128, 128, 8, 8]          [128, 128, 8, 8]          --
│    │    └─MaxPool2d: 3-9               [128, 128, 8, 8]          [128, 128, 4, 4]          --
│    └─BuildingBlock: 2-4                [128, 128, 4, 4]          [128, 256, 2, 2]          --
│    │    └─Conv2d: 3-10                 [128, 128, 4, 4]          [128, 256, 4, 4]          295,16
8
│    │    └─ReLU: 3-11                   [128, 256, 4, 4]          [128, 256, 4, 4]          --
│    │    └─MaxPool2d: 3-12              [128, 256, 4, 4]          [128, 256, 2, 2]          --
├─Sequential: 1-2                        [128, 1024]               [128, 100]                --
│    └─Dropout: 2-5                      [128, 1024]               [128, 1024]               --
│    └─Linear: 2-6                       [128, 1024]               [128, 512]                524,80
0
│    └─ReLU: 2-7                         [128, 512]                [128, 512]                --
│    └─Linear: 2-8                       [128, 512]                [128, 100]                51,300
==================================================================================================
================
Total params: 964,516
Trainable params: 964,516
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 2.01
==================================================================================================
================
Input size (MB): 1.57
Forward/backward pass size (MB): 63.54
Params size (MB): 3.86
```

```
Estimated Total Size (MB): 68.97
==========================================================================================
================
```

トレーニング可能なパラメータの総数は 964,516 である。パラメータのサイズを調べることで、各最大プーリング操作後にチャネルが両方の次元で半分になることが分る。これらの操作の最後には、2 x 2 の次元を持つ 256 チャネルの層があり、その後、サイズ 1,024 の密な層にフラット化される。言い換えれば、各 2 x 2 の行列は 4 x 1 ベクトルに変換され、1 つの層に横並びに配置される。その後、ドロップアウト正則化層が続き、さらに 512 のサイズの密な層があり、最後に出力層が続く。

これまで、`SimpleModule()` でデフォルトの最適化を利用していたが、ここでのデータでは、実験により小さな学習率の方がデフォルトの 0.01 よりも良い結果を出すことが分っている。ここでは学習率 0.001 のカスタム最適化を利用する。それに加えて、ロギングとトレーニングはこれまでの例と同様のパターンで行う。最適化では `params` という引数を取り、これにより最適化は SGD（確率的勾配降下法）を利用するパラメータを認識する。

以前、モジュールのパラメータ設定はテンソルであることを見た。パラメータをオプティマイザーに渡す際には、単に配列を渡すだけではなく、グラフの構造の一部がテンソル自体にエンコードされているのである。

In [57]:

```
cifar_optimizer = RMSprop(cifar_model.parameters(), lr=0.001)
cifar_module = SimpleModule.classification(cifar_model,
                                    num_classes=100,
                                    optimizer=cifar_optimizer)
cifar_logger = CSVLogger('logs', name='CIFAR100')
```

In [58]:

```
cifar_trainer = Trainer(deterministic=True,
                        max_epochs=30,
                        logger=cifar_logger,
                        enable_progress_bar=False,
                        callbacks=[ErrorTracker()])
```

```
cifar_trainer.fit(cifar_module,
                  datamodule=cifar_dm)
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

  | Name  | Type             | Params
-------------------------------------------
0 | model | CIFARModel       | 964 K
1 | loss  | CrossEntropyLoss | 0
-------------------------------------------
964 K     Trainable params
0         Non-trainable params
964 K     Total params
3.858     Total estimated model params size (MB)
`Trainer.fit` stopped: `max_epochs=30` reached.
```

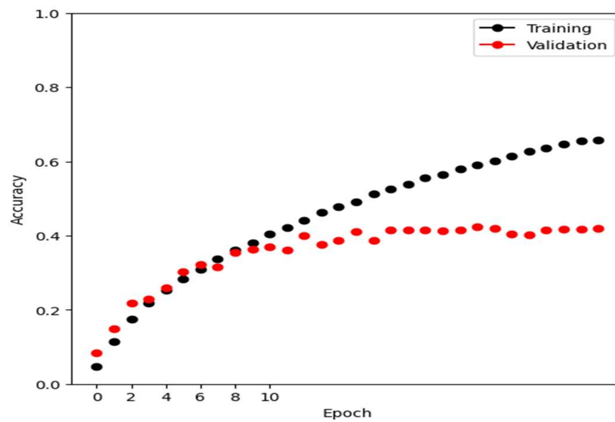
このモデルの実行には 10 分以上かかることがあり、テストデータに対して約42%の精度を達成する。100 クラスのデータに対しては悪くない結果ではあるが（ランダム分類器では 1%の精度しか出ない）、ウェブを検索すると約 75%の制度が出ている結果が見られる。このような結果を達成するには、通常、アーキテクチャの調整、正則化の微調整、そして時間が必要となる。

エポックごとの検証精度とトレーニング精度を見てみよう。

In [59]:

```
log_path = cifar_logger.experiment.metrics_file_path
cifar_results = pd.read_csv(log_path)
fig, ax = subplots(1, 1, figsize=(6, 6))
summary_plot(cifar_results,
             ax,
             col='accuracy',
             ylabel='Accuracy')
```

```
ax.set_xticks(np.linspace(0, 10, 6).astype(int))
ax.set_ylabel('Accuracy')
ax.set_ylim([0, 1]);
```



最後に、テストデータでモデルを評価する。

In [60]:

```
cifar_trainer.test(cifar_module,
                   datamodule=cifar_dm)
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃        Test metric        ┃       DataLoader 0        ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│       test_accuracy       │    0.4269999861717224     │
│         test_loss         │     2.45865797996521      │
└───────────────────────────┴───────────────────────────┘
```

Out[ ]:

```
[{'test_loss': 2.45865797996521, 'test_accuracy': 0.4269999861717224}]
```

ハードウエアの加速化(Hardware Acceleration)

ディープラーニングが機械学習で普及する中、ハードウェアメーカーは、勾配降下法のステップを高速化できる特別なライブラリを提供している。例えば、M1 チップを搭載した Mac OS デバイスでは、*Metal* プログラミングフレームワークが有効になっている場合があり、これにより `torch` の計算を高速化できる。この加速を利用する方法の例を示しておく。

主な変更点は、`Trainer()` の呼び出しと、データで評価されるメトリクスに関するものである。これらのメトリクスには、評価時にデータがどこにあるかを伝える必要があり、これをメトリクスを `to()` 法で呼び出すことで実現できる。

In [61]:

```
try:
    for name, metric in cifar_module.metrics.items():
        cifar_module.metrics[name] = metric.to('mps')
    cifar_trainer_mps = Trainer(accelerator='mps',
                                deterministic=True,
                                enable_progress_bar=False,
                                max_epochs=30)
    cifar_trainer_mps.fit(cifar_module,
                          datamodule=cifar_dm)
    cifar_trainer_mps.test(cifar_module,
                          datamodule=cifar_dm)
except:
    pass
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

  | Name  | Type             | Params
-------------------------------------------
0 | model | CIFARModel       | 964 K
1 | loss  | CrossEntropyLoss | 0
```

```
------------------------------------------
964 K     Trainable params
0         Non-trainable params
964 K     Total params
3.858     Total estimated model params size (MB)
`Trainer.fit` stopped: `max_epochs=30` reached.
```

このようにすることにより、各エポックの処理が約 2 倍または 3 倍の速度で加速される。このコードブロックは `try:` と `except:` 節を使って保護されているが、うまくいけば加速され、失敗すれば何も起こらない。

訓練された CNN モデルの利用(Using Pretrained CNN Models)

ここでは、`imagenet` データベースで事前学習された CNN を使用して自然画像を分類する方法を示し、図 10.10 をどのように作成したかを説明する。デジタルフォトアルバムから 6 枚の JPEG 画像を `book_images` ディレクトリにコピーした。これらの画像は、<www.statlearning.com> の ISLP 本のウェブサイトのデータセクションから入手できる。`book_images.zip` をダウンロードしてクリックすると `book_images` ディレクトリが作成される。

使用する事前学習ネットワークは `resnet50` と呼ばれ、仕様の詳細はウェブで確認できる。これらの画像を読み込み、`resnet50` の仕様に一致するように `torch` ソフトウェアが期待する配列形式に変換する。この変換では、リサイズ、クロップ、その後、3 つのチャネルそれぞれに対する事前定義された標準化が含まれている。この操作により画像を読み込み、前処理を行う。

In [62]:

```
resize = Resize((232,232), antialias=True)
crop = CenterCrop(224)
normalize = Normalize([0.485,0.456,0.406],
                      [0.229,0.224,0.225])
imgfiles = sorted([f for f in glob('book_images/*')])
imgs = torch.stack([torch.div(crop(resize(read_image(f))), 255)
```

```
                    for f in imgfiles])
imgs = normalize(imgs)
imgs.size()
```

Out[ ]:

```
torch.Size([6, 3, 224, 224])
```

ここでは、コードブロック~6 で読み込んだ重みを利用して、事前学習されたネットワークを設定する。このモデルは 50 層から成り、かなりの複雑さを持っている。

In [63]:

```
resnet_model = resnet50(weights=ResNet50_Weights.DEFAULT)
summary(resnet_model,
        input_data=imgs,
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

Out[ ]:

```
==========================================================================================
================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param
#
==========================================================================================
================
ResNet                                   [6, 3, 224, 224]          [6, 1000]                 --
├─Conv2d: 1-1                            [6, 3, 224, 224]          [6, 64, 112, 112]         9,408
├─BatchNorm2d: 1-2                       [6, 64, 112, 112]         [6, 64, 112, 112]         128
├─ReLU: 1-3                              [6, 64, 112, 112]         [6, 64, 112, 112]         --
├─MaxPool2d: 1-4                         [6, 64, 112, 112]         [6, 64, 56, 56]           --
├─Sequential: 1-5                        [6, 64, 56, 56]           [6, 256, 56, 56]          --
│    └─Bottleneck: 2-1                   [6, 64, 56, 56]           [6, 256, 56, 56]          --
│    │    └─Conv2d: 3-1                  [6, 64, 56, 56]           [6, 64, 56, 56]           4,096
│    │    └─BatchNorm2d: 3-2             [6, 64, 56, 56]           [6, 64, 56, 56]           128
│    │    └─ReLU: 3-3                    [6, 64, 56, 56]           [6, 64, 56, 56]           --
```

```
│    │    └─Conv2d: 3-4                [6, 64, 56, 56]           [6, 64, 56, 56]           36,864
│    │    └─BatchNorm2d: 3-5           [6, 64, 56, 56]           [6, 64, 56, 56]           128
│    │    └─ReLU: 3-6                  [6, 64, 56, 56]           [6, 64, 56, 56]           --
│    │    └─Conv2d: 3-7                [6, 64, 56, 56]           [6, 256, 56, 56]          16,384
│    │    └─BatchNorm2d: 3-8           [6, 256, 56, 56]          [6, 256, 56, 56]          512
│    │    └─Sequential: 3-9            [6, 64, 56, 56]           [6, 256, 56, 56]          16,896
│    │    └─ReLU: 3-10                 [6, 256, 56, 56]          [6, 256, 56, 56]          --
│    └─Bottleneck: 2-2                 [6, 256, 56, 56]          [6, 256, 56, 56]          --
│    │    └─Conv2d: 3-11               [6, 256, 56, 56]          [6, 64, 56, 56]           16,384
│    │    └─BatchNorm2d: 3-12          [6, 64, 56, 56]           [6, 64, 56, 56]           128
│    │    └─ReLU: 3-13                 [6, 64, 56, 56]           [6, 64, 56, 56]           --
│    │    └─Conv2d: 3-14               [6, 64, 56, 56]           [6, 64, 56, 56]           36,864
│    │    └─BatchNorm2d: 3-15          [6, 64, 56, 56]           [6, 64, 56, 56]           128
│    │    └─ReLU: 3-16                 [6, 64, 56, 56]           [6, 64, 56, 56]           --
│    │    └─Conv2d: 3-17               [6, 64, 56, 56]           [6, 256, 56, 56]          16,384
│    │    └─BatchNorm2d: 3-18          [6, 256, 56, 56]          [6, 256, 56, 56]          512
│    │    └─ReLU: 3-19                 [6, 256, 56, 56]          [6, 256, 56, 56]          --
│    └─Bottleneck: 2-3                 [6, 256, 56, 56]          [6, 256, 56, 56]          --
│    │    └─Conv2d: 3-20               [6, 256, 56, 56]          [6, 64, 56, 56]           16,384
│    │    └─BatchNorm2d: 3-21          [6, 64, 56, 56]           [6, 64, 56, 56]           128
│    │    └─ReLU: 3-22                 [6, 64, 56, 56]           [6, 64, 56, 56]           --
│    │    └─Conv2d: 3-23               [6, 64, 56, 56]           [6, 64, 56, 56]           36,864
│    │    └─BatchNorm2d: 3-24          [6, 64, 56, 56]           [6, 64, 56, 56]           128
│    │    └─ReLU: 3-25                 [6, 64, 56, 56]           [6, 64, 56, 56]           --
│    │    └─Conv2d: 3-26               [6, 64, 56, 56]           [6, 256, 56, 56]          16,384
│    │    └─BatchNorm2d: 3-27          [6, 256, 56, 56]          [6, 256, 56, 56]          512
│    │    └─ReLU: 3-28                 [6, 256, 56, 56]          [6, 256, 56, 56]          --
├─Sequential: 1-6                      [6, 256, 56, 56]          [6, 512, 28, 28]          --
│    └─Bottleneck: 2-4                 [6, 256, 56, 56]          [6, 512, 28, 28]          --
│    │    └─Conv2d: 3-29               [6, 256, 56, 56]          [6, 128, 56, 56]          32,768
│    │    └─BatchNorm2d: 3-30          [6, 128, 56, 56]          [6, 128, 56, 56]          256
│    │    └─ReLU: 3-31                 [6, 128, 56, 56]          [6, 128, 56, 56]          --
│    │    └─Conv2d: 3-32               [6, 128, 56, 56]          [6, 128, 28, 28]          147,45
    --
│    │    └─Conv2d: 3-143              [6, 512, 7, 7]            [6, 512, 7, 7]            2,359,
296
│    │    └─BatchNorm2d: 3-144         [6, 512, 7, 7]            [6, 512, 7, 7]            1,024
│    │    └─ReLU: 3-145                [6, 512, 7, 7]            [6, 512, 7, 7]            --
│    │    └─Conv2d: 3-146              [6, 512, 7, 7]            [6, 2048, 7, 7]           1,048,

. . .
```

```
（一部の出力を省略）

・・・

576
│    │    └─BatchNorm2d: 3-147          [6, 2048, 7, 7]          [6, 2048, 7, 7]          4,096
│    │    └─ReLU: 3-148                 [6, 2048, 7, 7]          [6, 2048, 7, 7]          --
├─AdaptiveAvgPool2d: 1-9                [6, 2048, 7, 7]          [6, 2048, 1, 1]          --
├─Linear: 1-10                          [6, 2048]                [6, 1000]                2,049,
000
===================================================================================================
================
Total params: 25,557,032
Trainable params: 25,557,032
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 24.54
===================================================================================================
================
Input size (MB): 3.61
Forward/backward pass size (MB): 1066.99
Params size (MB): 102.23
Estimated Total Size (MB): 1172.83
===================================================================================================
================
```

`eval()` モードに設定して、モデルが新しいデータに対して予測できる準備が整ったことを確認する。

In [64]:

```
resnet_model.eval()
```

Out[ ]:

```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
```

```
    (0): Bottleneck(
      (conv1): Conv2d(64, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
        (1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(256, 64, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(64, 256, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
  )
  (layer2): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(256, 128, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(128, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(128, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(256, 512, kernel_size=(1, 1), stride=(2, 2), bias=False)
```

```
        (1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
```

・・・

（一部の出力を省略）

・・・

```
  (layer4): Sequential(
    (0): Bottleneck(
      (conv1): Conv2d(1024, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (downsample): Sequential(
        (0): Conv2d(1024, 2048, kernel_size=(1, 1), stride=(2, 2), bias=False)
        (1): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      )
    )
    (1): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
    )
    (2): Bottleneck(
      (conv1): Conv2d(2048, 512, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn1): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv2): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (conv3): Conv2d(512, 2048, kernel_size=(1, 1), stride=(1, 1), bias=False)
      (bn3): BatchNorm2d(2048, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
```

```
    )
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(1, 1))
  (fc): Linear(in_features=2048, out_features=1000, bias=True)
)
```

上記の出力を確認すると、`resnet_model` を設定する際にここで `Bottleneck` を定義していることが分る。これは、既に利用した `BuildingBlock` モジュールに似ている。

次に、フィットしたネットワークを通じて、6 枚の画像を入力する。

In [65]:

```
img_preds = resnet_model(imgs)
```

まず、上位 3 つの選択肢に対する予測確率を見ておこう。最初に、`img_preds` のロジットにソフトマックスを適用して確率を計算する。ここで、`img_preds` テンソルを `detach()` 法で呼び出し、より馴染みのある `ndarray` に変換する必要があったことに注意しておく。

In [66]:

```
img_probs = np.exp(np.asarray(img_preds.detach()))
img_probs /= img_probs.sum(1)[:,None]
```

クラス・ラベルを確認するためには、`imagenet` に関連するインデックスファイルをダウンロードする必要がある。このファイルは、書籍のウェブサイトおよび s3.amazonaws.com/deep-learning-models/image-models/imagenet_class_index.json から入手可能である。

In [67]:

```
labs = json.load(open('imagenet_class_index.json'))
class_labels = pd.DataFrame([(int(k), v[1]) for k, v in
                            labs.items()],
                            columns=['idx', 'label'])
```

```
class_labels = class_labels.set_index('idx')
class_labels = class_labels.sort_index()
```

これから、上記のモデルによって推定された最も高い確率を持つ 3 つのラベルを各画像ファイルに対してデータフレームとして構築する。

In [68]:

```
for i, imgfile in enumerate(imgfiles):
    img_df = class_labels.copy()
    img_df['prob'] = img_probs[i]
    img_df = img_df.sort_values(by='prob', ascending=False)[:3]
    print(f'Image: {imgfile}')
    print(img_df.reset_index().drop(columns=['idx']))
Image: book_images/Cape_Weaver.jpg
      label      prob
0   jacamar  0.297499
1     macaw  0.068107
2  lorikeet  0.051105
Image: book_images/Flamingo.jpg
            label      prob
0        flamingo  0.609515
1       spoonbill  0.013586
2  American_egret  0.002132
Image: book_images/Hawk_Fountain.jpg
            label      prob
0            kite  0.184682
1           robin  0.084021
2  great_grey_owl  0.061274
Image: book_images/Hawk_cropped.jpg
            label      prob
0            kite  0.453833
1  great_grey_owl  0.015914
2             jay  0.012210
```

```
Image: book_images/Lhasa_Apso.jpg
             label      prob
0            Lhasa  0.260317
1         Shih-Tzu  0.097196
2  Tibetan_terrier  0.032820
Image: book_images/Sleeping_Cat.jpg
         label      prob
0  Persian_cat  0.163070
1        tabby  0.074143
2    tiger_cat  0.042578
```

モデルは `Flamingo.jpg` については結果にかなり自信があるが、他の画像については少し自信がないことが分る。

この節は、いつものようにクリーンアップで終了する。

In [69]:

```
del(cifar_test,
    cifar_train,
    cifar_dm,
    cifar_module,
    cifar_logger,
    cifar_optimizer,
    cifar_trainer)
IMDB Document Classification¶
```

IMDM Document Classification

ここでは、`IMDB` データセットに対する 10.4 節の感情分類（sentiment classification）のモデルを実装する。前述のコードブロック 8 で説明したように、`IMDB` データセットの前処理済みバージョンを `keras` パッケージで使用している。`keras` は異なるテンソルおよびディープラーニングライブラリである `tensorflow` を使用しているため、データを `torch` に適した形式に変換した。`keras` からの変換に使用したコードは `ISLP.torch._make_imdb` モジュールにあるが、これを実行するにはいくつかの

keras パッケージが必要となる。これらのデータは、サイズ 10,000 の辞書を利用している。

このラボでは、レビュー データの 3 つの異なる表現を保存している：

- load_tensor()：torch で使用できるスパーステンソル版。
- load_sparse()：sklearn で使用できるスパース行列版（ラッソフィットと比較するために使用）。
- load_sequential()：元のシーケンス表現のパディングされたバージョンで、各レビューの最後の 500 語に制限されている。

In [70]:

```
(imdb_seq_train,
 imdb_seq_test) = load_sequential(root='data/IMDB')
padded_sample = np.asarray(imdb_seq_train.tensors[0][0])
sample_review = padded_sample[padded_sample > 0][:12]
sample_review[:12]
```

Out[ ]:

```
array([   1,   14,   22,   16,   43,  530,  973, 1622, 1385,   65,  458,
       4468], dtype=int32)
```

データセット imdb_seq_train と imdb_seq_test はどちらも TensorDataset クラスの例である。これらを構成するテンソルは tensors 属性に格納されており、最初のテンソルが特徴量 X、2 番目が結果 Y である。ここで特徴量の最初の行を取り、それを padded_sample として保存した。このデータを作成する際の前処理では、シーケンスが十分に長くない場合、先頭に 0 でパディングが施されていたため、padded_sample > 0 の条件でパディングを取り除く。その後、サンプルレビューの最初の 12 単語を提供する。

これらの単語は、ISLP.torch.imdb モジュールの lookup 辞書にある。

In [71]:

```
lookup = load_lookup(root='data/IMDB')
' '.join(lookup[i] for i in sample_review)
```

Out[ ]:

```
"<START> this film was just brilliant casting location scenery story direction
everyone's"
```

最初のモデルでは、データセットの 10,000 の可能な単語それぞれに対してバイナリ特徴量を作成した。具体的には、単語 j がレビュー i に現れる場合、行列の i,j の位置に 1 を入力する。ほとんどのレビューは非常に短いため、このような特徴量行列は 98%以上がゼロで占められている。これらのデータは、`ISLP` ライブラリの `load_tensor()` を利用してアクセスする。

In [72]:

```
max_num_workers=10
(imdb_train,
 imdb_test) = load_tensor(root='data/IMDB')
imdb_dm = SimpleDataModule(imdb_train,
                           imdb_test,
                           validation=2000,
                           num_workers=min(6, max_num_workers),
                           batch_size=512)
```

最初に 2 層のモデルを利用する。

In [73]:

```
class IMDBModel(nn.Module):

    def __init__(self, input_size):
        super(IMDBModel, self).__init__()
        self.dense1 = nn.Linear(input_size, 16)
        self.activation = nn.ReLU()
        self.dense2 = nn.Linear(16, 16)
        self.output = nn.Linear(16, 1)
```

```
    def forward(self, x):
        val = x
        for _map in [self.dense1,
                     self.activation,
                     self.dense2,
                     self.activation,
                     self.output]:
            val = _map(val)
        return torch.flatten(val)
```

ここでモデルを推定してサマリーを確認する。

In [74]:

```
imdb_model = IMDBModel(imdb_test.tensors[0].size()[1])
summary(imdb_model,
        input_size=imdb_test.tensors[0].size(),
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

Out[ ]:

```
====================================================================================================
================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param
#
====================================================================================================
================
IMDBModel                                [25000, 10003]            [25000]                   --
├─Linear: 1-1                            [25000, 10003]            [25000, 16]               160,06
4
├─ReLU: 1-2                              [25000, 16]               [25000, 16]               --
├─Linear: 1-3                            [25000, 16]               [25000, 16]               272
├─ReLU: 1-4                              [25000, 16]               [25000, 16]               --
├─Linear: 1-5                            [25000, 16]               [25000, 1]                17
====================================================================================================
```

```
================
Total params: 160,353
Trainable params: 160,353
Non-trainable params: 0
Total mult-adds (Units.GIGABYTES): 4.01
====================================================================================================
================
Input size (MB): 1000.30
Forward/backward pass size (MB): 6.60
Params size (MB): 0.64
Estimated Total Size (MB): 1007.54
====================================================================================================
================
```

これらのデータには再び小さな学習率を使用するため、`SimpleModule` に `optimizer` を渡す。レビューはポジティブまたはネガティブな感情に分類されるため、`SimpleModule.binary_classification()` を利用する。ここで `binary_classification()` を `classification()` の代わりに使用する理由は、`torchmetrics.Accuracy()` の動作の微妙な違いと、ターゲットのデータ型に関連している。

In [75]:

```
imdb_optimizer = RMSprop(imdb_model.parameters(), lr=0.001)
imdb_module = SimpleModule.binary_classification(
                         imdb_model,
                         optimizer=imdb_optimizer)
```

データセットをデータ・モジュールに読み込み、`SimpleModule` を作成した後は、残りのステップはいつもと同一である。

In [76]:

```
imdb_logger = CSVLogger('logs', name='IMDB')
imdb_trainer = Trainer(deterministic=True,
                       max_epochs=30,
                       logger=imdb_logger,
                       enable_progress_bar=False,
                       callbacks=[ErrorTracker()])
```

```
imdb_trainer.fit(imdb_module,
                 datamodule=imdb_dm)
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

  | Name  | Type              | Params
--------------------------------------------
0 | model | IMDBModel         | 160 K
1 | loss  | BCEWithLogitsLoss | 0
--------------------------------------------
160 K     Trainable params
0         Non-trainable params
160 K     Total params
0.641     Total estimated model params size (MB)
/Users/jtaylo/anaconda3/envs/ISLP_v22_312/lib/python3.12/site-packages/pytorch_
lightning/loops/fit_loop.py:298: The number of training batches (45) is smaller
 than the logging interval Trainer(log_every_n_steps=50). Set a lower value for
 log_every_n_steps if you want to see logs for the training epoch.
`Trainer.fit` stopped: `max_epochs=30` reached.
```

テストエラーを評価すると、約 85%の精度が得られる。

In [77]:

```
test_results = imdb_trainer.test(imdb_module, datamodule=imdb_dm)
test_results
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃        Test metric        ┃       DataLoader 0        ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│       test_accuracy       │    0.8450000286102295     │
│         test_loss         │    1.2167928218841553     │
└───────────────────────────┴───────────────────────────┘
```

Out[ ]:

```
[{'test_loss': 1.2167928218841553, 'test_accuracy': 0.8450000286102295}]
```

Lasso との比較(Comparison to Lasso)

次に、`sklearn` の `LogisticRegression()` を使用して Lasso（ラッソ）ロジスティック回帰モデルをフィットする。`sklearn` は `torch` のスパーステンソルを認識しないため、`sklearn` が認識できる疎（スパース）行列を利用する。

In [78]:

```
((X_train, Y_train),
 (X_valid, Y_valid),
 (X_test, Y_test)) = load_sparse(validation=2000,
                                 random_state=0,
                                 root='data/IMDB')
```

10.9.1 節で行ったようにラッソ正則化パラメータ λ に 50 を設定する。

In [79]:

```
lam_max = np.abs(X_train.T * (Y_train - Y_train.mean())).max()
lam_val = lam_max * np.exp(np.linspace(np.log(1),
                                       np.log(1e-4), 50))
```

なお `LogisticRegression()` では、正則化パラメーターC は λ の逆数として指定される。ロジスティック回帰にはいくつかのソルバーがあるが、ここでは疎（スパース）入力形式に適した `liblinear` を利用する。

In [80]:

```
logit = LogisticRegression(penalty='l1',
                           C=1/lam_max,
                           solver='liblinear',
```

```
                            warm_start=True,
                            fit_intercept=True)
```

50 の値に対するパスを実行するのに約 40 秒かかる。

In [81]:

```
coefs = []
intercepts = []

for l in lam_val:
    logit.C = 1/l
    logit.fit(X_train, Y_train)
    coefs.append(logit.coef_.copy())
    intercepts.append(logit.intercept_)
```

係数と切片には余分な次元が含まれているため、`np.squeeze()` 関数を使用してこれを削除しておく。

In [82]:

```
coefs = np.squeeze(coefs)
intercepts = np.squeeze(intercepts)
```

これよりニューラル・ネットワーク・モデルの結果とラッソの結果を比較するプロットを作成する。

In [83]:

```
%%capture
fig, axes = subplots(1, 2, figsize=(16, 8), sharey=True)
for ((X_, Y_),
     data_,
     color) in zip([(X_train, Y_train),
                    (X_valid, Y_valid),
```

```
                    (X_test, Y_test)],
                    ['Training', 'Validation', 'Test'],
                    ['black', 'red', 'blue']):
    linpred_ = X_ * coefs.T + intercepts[None,:]
    label_ = np.array(linpred_ > 0)
    accuracy_ = np.array([np.mean(Y_ == l) for l in label_.T])
    axes[0].plot(-np.log(lam_val / X_train.shape[0]),
                 accuracy_,
                 '.--',
                 color=color,
                 markersize=13,
                 linewidth=2,
                 label=data_)
axes[0].legend()
axes[0].set_xlabel(r'$-\log(\lambda)$', fontsize=20)
axes[0].set_ylabel('Accuracy', fontsize=20)
```

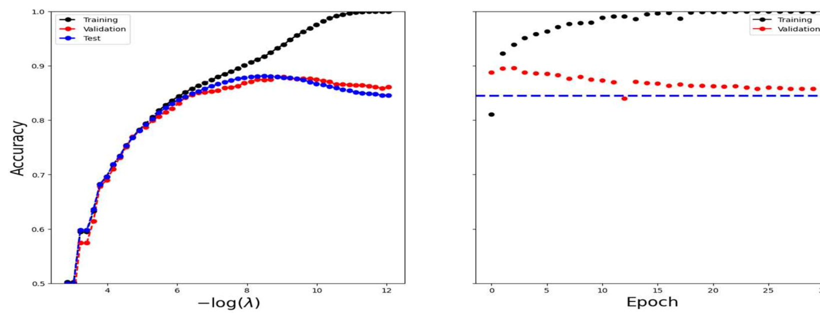
`%%capture` の利用に注意しておこう。これにより、部分的に完成した図の表示が抑制される。複雑な図を作成する際には便利であり、ステップを 2 つ以上のセルに分けて処理することができる。
次に、ラッソ精度のプロットを追加し、セルの最後にその名前を入力することで、作成した図を表示する。

In [84]:

```
imdb_results = pd.read_csv(imdb_logger.experiment.metrics_file_path)
summary_plot(imdb_results,
             axes[1],
             col='accuracy',
             ylabel='Accuracy')
axes[1].set_xticks(np.linspace(0, 30, 7).astype(int))
axes[1].set_ylabel('Accuracy', fontsize=20)
axes[1].set_xlabel('Epoch', fontsize=20)
axes[1].set_ylim([0.5, 1]);
axes[1].axhline(test_results[0]['test_accuracy'],
```

```
                 color='blue',
                 linestyle='--',
                 linewidth=3)
fig
```

Out[ ]:



グラフから、Lasso ロジスティック回帰の精度が約 0.88 でピークに達することが分る。これはニューラル・ネットワーク・モデルと同等になる。

再度、クリーンアップで終了する。

In [85]:

```
del(imdb_model,
    imdb_trainer,
    imdb_logger,
    imdb_dm,
    imdb_train,
    imdb_test)
```

RNN(Recurrent Neural Networks)

このラボでは、10.5 節で示されたモデルをフィットする。

逐次ドキュメント分類(Sequential Models for Document Classification)

ここでは、感情予測(sentiment prediction)のためにシンプルな LSTM RNN を `IMDb` の映画レビュー・データにフィットしてみよう。これは、10.5.1 節で説明した内容である。RNN では、ドキュメント内の単語の順番を考慮してシーケンスを使用している。データは、10.9.5 節 の冒頭で前処理されたものをロードした。前処理の詳細なスクリプトは `ISLP` ライブラリにあるが、特に、90%以上のドキュメントが 500 単語未満だったため、ドキュメントの長さを 500 に設定した。長いドキュメントには最後の 500 単語を使用し、短いドキュメントには前に空白でパディングを施した。

In [86]:

```
imdb_seq_dm = SimpleDataModule(imdb_seq_train,
                               imdb_seq_test,
                               validation=2000,
                               batch_size=300,
                               num_workers=min(6, max_num_workers)
                               )
```

RNN の最初の層は、サイズ 32 の埋め込み層で、これはトレーニング中に学習される。この層は、各ドキュメントを次元 500 x 10,003 の行列としてワンホットエンコードし、その後、これらの 10,003 次元を 32 次元にマッピングする。

(追加の 3 次元は、レビュー内でよく見られる単語以外のエントリに対応する。) 各単語は整数で表されるため、これは実質的にサイズ 10,003 x 32 の埋め込み行列を作成することによって実現される。ドキュメント内の 500 の整数は、この行列の適切な行をインデックスして、対応する 32 の実数にマッピングされる。

2 番目の層は 32 ユニットの LSTM で、出力層はバイナリ分類タスクのための単一のロジット・モデルである。以下の `forward()` 法の最後の行では、LSTM の最後の 32 次元の出力を取り、それを応答にマッピングしている。

In [87]:

```
class LSTMModel(nn.Module):
    def __init__(self, input_size):
        super(LSTMModel, self).__init__()
        self.embedding = nn.Embedding(input_size, 32)
        self.lstm = nn.LSTM(input_size=32,
                            hidden_size=32,
                            batch_first=True)
        self.dense = nn.Linear(32, 1)
    def forward(self, x):
        val, (h_n, c_n) = self.lstm(self.embedding(x))
        return torch.flatten(self.dense(val[:,-1]))
```

モデルを初期推定、コーパスの最初の 10 ドキュメントを使用してモデルのサマリーを確認する。

In [88]:

```
lstm_model = LSTMModel(X_test.shape[-1])
summary(lstm_model,
        input_data=imdb_seq_train.tensors[0][:10],
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

Out[ ]:

```
====================================================================================================
================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param
#
====================================================================================================
================
LSTMModel                                [10, 500]                 [10]                      --
├─Embedding: 1-1                         [10, 500]                 [10, 500, 32]             320,09
6
├─LSTM: 1-2                              [10, 500, 32]             [10, 500, 32]             8,448
├─Linear: 1-3                            [10, 32]                  [10, 1]                   33
```

```
================================================================================================
================
Total params: 328,577
Trainable params: 328,577
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 45.44
================================================================================================
================
Input size (MB): 50.00
Forward/backward pass size (MB): 2.56
Params size (MB): 1.31
Estimated Total Size (MB): 53.87
================================================================================================
================
```

サマリーでは 10,003 は抑制されていますが、パラメータ数においてそれが表示されている。これは 10,003 x 32 = 320,096 だからである。

In [89]:

```
lstm_module = SimpleModule.binary_classification(lstm_model)
lstm_logger = CSVLogger('logs', name='IMDB_LSTM')
```

In [90]:

```
lstm_trainer = Trainer(deterministic=True,
                       max_epochs=20,
                       logger=lstm_logger,
                       enable_progress_bar=False,
                       callbacks=[ErrorTracker()])
lstm_trainer.fit(lstm_module,
                 datamodule=imdb_seq_dm)
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs
```

```
  | Name  | Type              | Params
--------------------------------------------
0 | model | LSTMModel         | 328 K
1 | loss  | BCEWithLogitsLoss | 0
--------------------------------------------
328 K     Trainable params
0         Non-trainable params
328 K     Total params
1.314     Total estimated model params size (MB)
`Trainer.fit` stopped: `max_epochs=20` reached.
```

残りの部分は、これまでにフィットした他のネットワークに類似している。ネットワークをフィットする際にテストパフォーマンスを追跡し、およそ 85%の精度に達することが分る。

In [91]:

```
lstm_trainer.test(lstm_module, datamodule=imdb_seq_dm)
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃        Test metric         ┃        DataLoader 0        ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│       test_accuracy        │     0.8505200147628784     │
│         test_loss          │     0.7480539679527283     │
└────────────────────────────┴────────────────────────────┘
```

Out[ ]:

```
[{'test_loss': 0.7480539679527283, 'test_accuracy': 0.8505200147628784}]
```

再度、学習の進行状況を表示し、その後クリーンアップを行う。

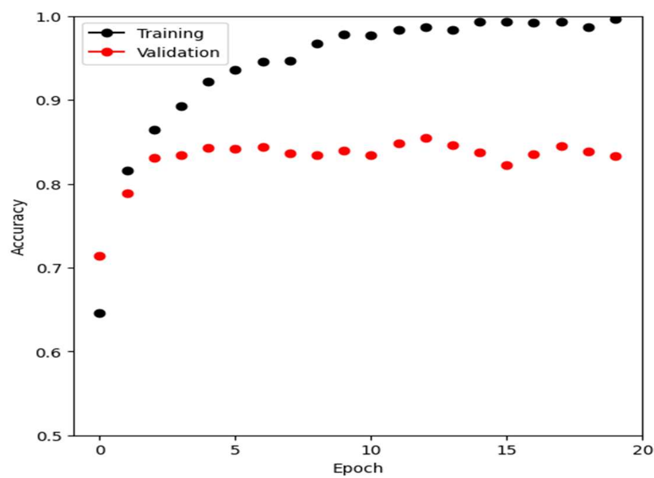
In [92]:

```
lstm_results = pd.read_csv(lstm_logger.experiment.metrics_file_path)
fig, ax = subplots(1, 1, figsize=(6, 6))
```

```
summary_plot(lstm_results,
             ax,
             col='accuracy',
             ylabel='Accuracy')
ax.set_xticks(np.linspace(0, 20, 5).astype(int))
ax.set_ylabel('Accuracy')
ax.set_ylim([0.5, 1])
```

Out[ ]:

```
(0.5, 1.0)
```



In [93]:

```
del(lstm_model,
    lstm_trainer,
    lstm_logger,
    imdb_seq_dm,
    imdb_seq_train,
    imdb_seq_test)
```

時系列予測(Time Series Prediction)

次に、時系列予測のために 10.5.2 節で示したモデルをフィットする方法を紹介しよう。まず、データをロードして標準化する。

In [94]:

```
NYSE = load_data('NYSE')
cols = ['DJ_return', 'log_volume', 'log_volatility']
X = pd.DataFrame(StandardScaler(
                     with_mean=True,
                     with_std=True).fit_transform(NYSE[cols]),
                 columns=NYSE[cols].columns,
                 index=NYSE.index)
```

次に、ラグ付きデータを設定し、`dropna()` 法を使用して欠損値を含む行を削除する。

In [95]:

```
for lag in range(1, 6):
    for col in cols:
        newcol = np.zeros(X.shape[0]) * np.nan
        newcol[lag:] = X[col].values[:-lag]
        X.insert(len(X.columns), "{0}_{1}".format(col, lag), newcol)
X.insert(len(X.columns), 'train', NYSE['train'])
X = X.dropna()
```

最後に、応答変数とトレーニング・データの変数を取り出し、現在時点の `DJ_return` と `log_volatility` を削除して、前日までのデータのみを使って予測を行う。

In [96]:

```
Y, train = X['log_volume'], X['train']
X = X.drop(columns=['train'] + cols)
X.columns
```

Out[ ]:

```
Index(['DJ_return_1', 'log_volume_1', 'log_volatility_1', 'DJ_return_2',
       'log_volume_2', 'log_volatility_2', 'DJ_return_3', 'log_volume_3',
       'log_volatility_3', 'DJ_return_4', 'log_volume_4', 'log_volatility_4',
       'DJ_return_5', 'log_volume_5', 'log_volatility_5'],
      dtype='object')
```

まず、単純な線形モデルをフィットし、`score()` 法を使用してテスト・データに対する R^2 を計算する。

In [97]:

```
M = LinearRegression()
M.fit(X[train], Y[train])
M.score(X[~train], Y[~train])
```

Out[ ]:

```
0.4128912938562521
```

このモデルを再びフィットし、`day_of_week` のようなカテゴリ変数も含める。`pandas` のカテゴリカル時系列に対して、`get_dummies()` 法を使用して指数が作成できる。

In [98]:

```
X_day = pd.concat([X,
                  pd.get_dummies(NYSE['day_of_week'])],
                  axis=1).dropna()
```

線形回帰モデルは再入力する必要はないことに注意しよう。線形回帰では `fit()` 法はデザイン行列と応答変数を直接に利用している。

In [99]:

```
M.fit(X_day[train], Y[train])
M.score(X_day[~train], Y[~train])
```

Out[ ]:

```
0.4595563133053273
```

このモデルでは、約 46%の R^2 を達成している。

RNN を適合させるためには、データを再構成する必要がある。RNN 層の `input_shape` 引数に示されたように、各特徴量の 5 つのラグ変数に利用が期待できる。まず、データフレームの列が、リシェイプされた行列が変数を正しくラグをとっているように整える。この操作は、`reindex()` 法を使用して行う。

入力形状 `(5,3)` の場合、各行は 3 つの変数のラグ変数を表します。`nn.RNN()` 層は、各観測の最初の行が最も早い時点を表すため、現在の順序を逆にする必要がある。そのためには、以下のように `range(5,0,-1)` をループして利用する。これは、`slice()` を使用して逐次的に変数を指標化する例である。一般的な表記は `start:end:step` である。

In [100]:

```
ordered_cols = []
for lag in range(5,0,-1):
    for col in cols:
        ordered_cols.append('{0}_{1}'.format(col, lag))
X = X.reindex(columns=ordered_cols)
X.columns
```

Out[ ]:

```
Index(['DJ_return_5', 'log_volume_5', 'log_volatility_5', 'DJ_return_4',
       'log_volume_4', 'log_volatility_4', 'DJ_return_3', 'log_volume_3',
       'log_volatility_3', 'DJ_return_2', 'log_volume_2', 'log_volatility_2',
       'DJ_return_1', 'log_volume_1', 'log_volatility_1'],
      dtype='object')
```

データをここで整理する。

In [101]:

```
X_rnn = X.to_numpy().reshape((-1,5,3))
X_rnn.shape
```

Out[ ]:

```
(6046, 5, 3)
```

最初のサイズを -1 と指定することで、`numpy.reshape()` は残りの引数に基づいてそのサイズを自動的に設定する。

これで、12 の隠れユニットと 10%のドロップアウトを使用する RNN の実行準備が整った。RNN を通過した後、`forward()` の中で最終的なタイムポイントを `val[:,-1]` として抽出する。これが 10%のドロップアウトを通過した後、線形層でフラット化される。

In [102]:

```
class NYSEModel(nn.Module):
    def __init__(self):
        super(NYSEModel, self).__init__()
        self.rnn = nn.RNN(3,
                          12,
                          batch_first=True)
        self.dense = nn.Linear(12, 1)
        self.dropout = nn.Dropout(0.1)
```

```
    def forward(self, x):
        val, h_n = self.rnn(x)
        val = self.dense(self.dropout(val[:,-1]))
        return torch.flatten(val)
nyse_model = NYSEModel()
```

モデルは、以前のネットワークと同様の方法でフィットさせる。`fit` 関数にはテストデータをバリデーションデータとして提供し、進行状況を監視し、履歴関数をプロットするときにテストデータ上の進捗を見ることができるようにする。当然、テストデータを基にした早期停止は避けるべきである。そうしないとテスト・パフォーマンスに偏りが生じてしまうことになる。

トレーニング・データセットは、`Hitters` の例と似た形で構成する。

In [103]:

```
datasets = []
for mask in [train, ~train]:
    X_rnn_t = torch.tensor(X_rnn[mask].astype(np.float32))
    Y_t = torch.tensor(Y[mask].astype(np.float32))
    datasets.append(TensorDataset(X_rnn_t, Y_t))
nyse_train, nyse_test = datasets
```

いつものようにサマリーを確認する。

In [104]:

```
summary(nyse_model,
        input_data=X_rnn_t,
        col_names=['input_size',
                   'output_size',
                   'num_params'])
```

Out[ ]:

```
==================================================================================================
================
Layer (type:depth-idx)                   Input Shape               Output Shape              Param
#
==================================================================================================
================
NYSEModel                                [1770, 5, 3]              [1770]                    --
├─RNN: 1-1                               [1770, 5, 3]              [1770, 5, 12]             204
├─Dropout: 1-2                           [1770, 12]                [1770, 12]                --
├─Linear: 1-3                            [1770, 12]                [1770, 1]                 13
==================================================================================================
================
Total params: 217
Trainable params: 217
Non-trainable params: 0
Total mult-adds (Units.MEGABYTES): 1.83
==================================================================================================
================
Input size (MB): 0.11
Forward/backward pass size (MB): 0.86
Params size (MB): 0.00
Estimated Total Size (MB): 0.97
==================================================================================================
================
```

再度、2 つのデータセットをデータ・モジュールに入れ、バッチサイズを 64 に設定する。

In [105]:

```
nyse_dm = SimpleDataModule(nyse_train,
                           nyse_test,
                           num_workers=min(4, max_num_workers),
                           validation=nyse_test,
                           batch_size=64)
```

このモデルを通していくつかのデータを実行し、サイズが正しく一致していることを確認する。

In [106]:

```
for idx, (x, y) in enumerate(nyse_dm.train_dataloader()):
    out = nyse_model(x)
    print(y.size(), out.size())
    if idx >= 2:
        break
torch.Size([64]) torch.Size([64])
torch.Size([64]) torch.Size([64])
torch.Size([64]) torch.Size([64])
```

前回の例に従って、回帰問題のための訓練をセットアップし、各エポックで R^2 を尺度に計算する。

In [107]:

```
nyse_optimizer = RMSprop(nyse_model.parameters(),
                         lr=0.001)
nyse_module = SimpleModule.regression(nyse_model,
                                      optimizer=nyse_optimizer,
                                      metrics={'r2':R2Score()})
```

モデルのフィットは、これまでの流れでお馴染みのものとなっているはずである。テスト・データに対する結果は、線形 AR モデルの結果と類似している。

In [108]:

```
nyse_trainer = Trainer(deterministic=True,
                       max_epochs=200,
                       enable_progress_bar=False,
                       callbacks=[ErrorTracker()])
nyse_trainer.fit(nyse_module,
                 datamodule=nyse_dm)
nyse_trainer.test(nyse_module,
                  datamodule=nyse_dm)
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
```

```
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

  | Name  | Type      | Params
------------------------------------
0 | model | NYSEModel | 217
1 | loss  | MSELoss   | 0
------------------------------------
217       Trainable params
0         Non-trainable params
217       Total params
0.001     Total estimated model params size (MB)
`Trainer.fit` stopped: `max_epochs=200` reached.
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃        Test metric        ┃       DataLoader 0        ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│         test_loss         │     0.616382360458374     │
│          test_r2          │    0.4150230884552002     │
└───────────────────────────┴───────────────────────────┘
```

Out[ ]:

```
[{'test_loss': 0.616382360458374, 'test_r2': 0.4150230884552002}]
```

`nn.RNN()` レイヤーを使わず、代わりに `nn.Flatten()` レイヤーを使用してモデルをフィットすることもできる。これにより、非線形 AR モデルが作成できる。さらに、隠れ層を除外すれば、実はこれが以前の線形 AR モデルと同等になる。

しかし、今回は `day_of_week` インジケーターを含む特徴セット `X_day` を使用して非線形 AR モデルをフィットしてみよう。そのためには、まずテストデータとトレーニングデータセットを作成し、それに対応するデータモジュールを作成する必要がある。これが少し手間に感じるかもしれないが、`torch` の一般的なパイプラインの一部である。

In [109]:

```
datasets = []
for mask in [train, ~train]:
    X_day_t = torch.tensor(
                   np.asarray(X_day[mask]).astype(np.float32))
    Y_t = torch.tensor(np.asarray(Y[mask]).astype(np.float32))
    datasets.append(TensorDataset(X_day_t, Y_t))
day_train, day_test = datasets
```

データ・モジュールの作成は、お馴染みのパターンに従う。

In [110]:

```
day_dm = SimpleDataModule(day_train,
                          day_test,
                          num_workers=min(4, max_num_workers),
                          validation=day_test,
                          batch_size=64)
```

`NonLinearARModel()` を構築し、20 の特徴量と 32 ユニットの隠れ層を入力として取る。残りのステップはお馴染みのものと同様である。

In [111]:

```
class NonLinearARModel(nn.Module):
    def __init__(self):
        super(NonLinearARModel, self).__init__()
        self._forward = nn.Sequential(nn.Flatten(),
                                      nn.Linear(20, 32),
                                      nn.ReLU(),
                                      nn.Dropout(0.5),
                                      nn.Linear(32, 1))
    def forward(self, x):
        return torch.flatten(self._forward(x))
```

In [112]:

```
nl_model = NonLinearARModel()
nl_optimizer = RMSprop(nl_model.parameters(),
                           lr=0.001)
nl_module = SimpleModule.regression(nl_model,
                                        optimizer=nl_optimizer,
                                        metrics={'r2':R2Score()})
```

通常のトレーニング・ステップを続けて、モデルをフィットし、テスト誤差を評価する。テストの R^2 によれば、`day_of_week` を含む線形 AR モデルに対してわずかな改善が見られる。

In [113]:

```
nl_trainer = Trainer(deterministic=True,
                     max_epochs=20,
                     enable_progress_bar=False,
                     callbacks=[ErrorTracker()])
nl_trainer.fit(nl_module, datamodule=day_dm)
nl_trainer.test(nl_module, datamodule=day_dm)
GPU available: True (mps), used: True
TPU available: False, using: 0 TPU cores
IPU available: False, using: 0 IPUs
HPU available: False, using: 0 HPUs

  | Name  | Type             | Params
-------------------------------------------
0 | model | NonLinearARModel | 705
1 | loss  | MSELoss          | 0
-------------------------------------------
705       Trainable params
0         Non-trainable params
705       Total params
```

```
0.003     Total estimated model params size (MB)
`Trainer.fit` stopped: `max_epochs=20` reached.
┏━━━━━━━━━━━━━━━━━━━━━━━━━━━┳━━━━━━━━━━━━━━━━━━━━━━━━━━━┓
┃        Test metric        ┃       DataLoader 0        ┃
┡━━━━━━━━━━━━━━━━━━━━━━━━━━━╇━━━━━━━━━━━━━━━━━━━━━━━━━━━┩
│         test_loss         │    0.5626183152198792     │
│          test_r2          │    0.4660477638244629     │
└───────────────────────────┴───────────────────────────┘
```

Out[ ]:

```
[{'test_loss': 0.5626183152198792, 'test_r2': 0.4660477638244629}]
```

ISLP 第 11 章 生存時間解析(Survival Analysis)

本章では、３つのデータセットについて生存時間解析を実施する。１つ目は `BrainCancer` データ、２つ目は `Publication` データ、３つ目はコールセンターの模擬データである。

まず初めに本章で扱うライブラリをインポートする。最初に使用するライブラリをまとめておくことでコードが読みやすくなるとともに、どのライブラリーを利用するか明瞭になる。

In [1]:

```
from matplotlib.pyplot import subplots
import numpy as np
import pandas as pd
from ISLP.models import ModelSpec as MS
from ISLP import load_data
```

生存時間解析を実施するために、新たに次のライブラリを読み込む。

In [2]:

```
from lifelines import \
     (KaplanMeierFitter,
      CoxPHFitter)
from lifelines.statistics import \
     (logrank_test,
      multivariate_logrank_test)
from ISLP.survival import sim_time
```

脳癌データ(Brain Cancer Data)

本セクションでは、`ISLP`に含まれている`BrainCancer`データを用いて生存時間解析を実施しよう。

In [3]:

```
BrainCancer = load_data('BrainCancer')
BrainCancer.columns
```

Out[3]:

```
Index(['sex', 'diagnosis', 'loc', 'ki', 'gtv', 'stereo', 'status', 'time'], dty
pe='object')
```

行は８８人の患者を示しており、８つの列には説明変数と応答変数が含まれる。
まず初めにデータセットについて簡単に見ていこう。

In [4]:

```
BrainCancer['sex'].value_counts()
```

Out[4]:

```
sex
Female    45
Male      43
Name: count, dtype: int64
```

In [5]:

```
BrainCancer['diagnosis'].value_counts()
```

Out[5]:

```
diagnosis
Meningioma    42
HG glioma     22
Other         14
LG glioma      9
Name: count, dtype: int64
```

In [6]:

```
BrainCancer['status'].value_counts()
```

Out[6]:

```
status
0    53
1    35
Name: count, dtype: int64
```

データ分析に取り掛かる際には各説明変数が何を示すものなのかを理解することが大事である。特に、ダミー変数の場合、要素がどの数値に割り当てられるのかを理解することで誤った解釈を防ぐことが可能となる。ここでは、`status`変数について確認しよう。これは、調査が打ち切られているかどうかを０と１の二値で表したダミー変数である。ソフトウェアやオープンデータによって、割り当てられる数値は逆の場合があるが、多くの場合`status`が１であれば打ち切られていない調査（死亡を含む場合も存在する）を示し、０であれば打ち切られた調査という慣例を用いることがある。`BrainCancer`のデータセットでは、調査終了までに生存している場合が０とし、死亡した場合を１としている。それぞれの値をカウントした結果から、３５人の患者が調査終了前に死亡しており、打ち切りとなっていることがわかる。

データ分析を始めるためにカプラン・マイヤー生存曲線を作成してみよう。生存分析に使用する主なパッケージは`lifelines`である。変数`time`はy_iに対応し、i 番目のイベント（打ち切りか死亡）までの時間である。`km.fit`の第 1 引数はイベント時刻で、第 2 引数は打ち切り変数である。`plot()`メソッドによって、信頼区間

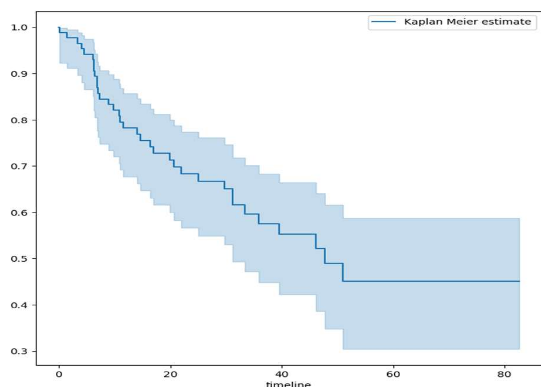
を含んだ生存曲線を作成する。デフォルトのままでは９０%信頼区間を示すが、`alpha`引数を変更することで希望する信頼区間を設定し可視化できる。

In [5]:

```
fig, ax = subplots(figsize=(8,8))
km = KaplanMeierFitter()
km_brain = km.fit(BrainCancer['time'], BrainCancer['status'])
km_brain.plot(label='Kaplan Meier estimate', ax=ax)
```

Out[5]:

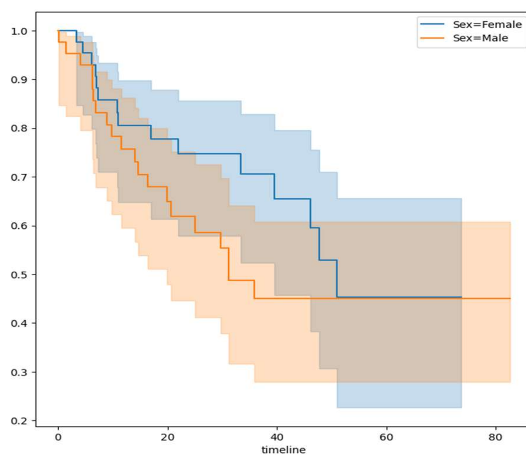
```
<Axes: xlabel='timeline'>
```



次に、`sex`を用いて層別化してカプラン・マイヤー生存曲線を作成しよう。層別化の際には、dataframe の`groupby()`メソッドを用いて、`for`によるループ処理を用いて性別毎に曲線を作成していこう。`in`の後ろに置いたデータ内、ここでは`BrainCancer`の`sex`に含まれる要素に対して、それぞれループ処理が行われる。今回であれば`Female`に該当するデータが抽出され可視化されたのち、`Male`に該当するデータが抽出され可視化される。ここでは注釈として`plot`を用いる際に`label='Sex=%s' % sex`とすることで文字列を補間し自動的にラベルを付けている。`Python`にはこのような操作を簡易にする記法が存在するので、使いながら慣れていくことで分析の質が向上することに繋がる。

In [6]:

```
fig, ax = subplots(figsize=(8,8))
by_sex = {}
for sex, df in BrainCancer.groupby('sex'):
    by_sex[sex] = df
    km_sex = km.fit(df['time'], df['status'])
    km_sex.plot(label='Sex=%s' % sex, ax=ax)
```



男性と女性の生存率を比較するために log-rank 検定を実行しよう。ここでは `lifelines.statistics` に含まれる `logrank_test()`関数を使用する。指定する引数として、最初はイベント時刻、２番目は対応する打ち切り指標を指定する。

In [7]:

```
logrank_test(by_sex['Male']['time'],
             by_sex['Female']['time'],
             by_sex['Male']['status'],
             by_sex['Female']['status'])
```

Out[7]:

t_0	-1
null_distribution	chi squared
degrees_of_freedom	1
test_name	logrank_test

	test_statistic	p	-log2(p)

0 1.44 0.23 2.12

その結果では p 値は 0.23 となり、男女間の生存率に差がないことを示している。次に、lifelines の CoxPHFitter()を用いて Cox 比例ハザードモデルをフィットしよう。 はじめに、唯一の予測因子として sex を使用するモデルを検討しよう。

In [8]:

```
coxph = CoxPHFitter # shorthand
sex_df = BrainCancer[['time', 'status', 'sex']]
model_df = MS(['time', 'status', 'sex'],
              intercept=False).fit_transform(sex_df)
cox_fit = coxph().fit(model_df,
                      'time',
                      'status')
cox_fit.summary[['coef', 'se(coef)', 'p']]
```

Out[8]:

covariate	coef	se(coef)	p
sex[Male]	0.407668	0.342004	0.233262

fit の最初の引数は、少なくともイベント時刻（この場合は２番目の引数で指定している time）と打ち切り変数（この場合は３番目の引数で指定している status）を含むデータフレームである必要がある。

ModelSpec に intercept=False という引数を使うことで、ここで考慮している Cox モデルに切片項を含ませないことができる。

summary()メソッドは多くの情報をまとめてくれるが、ここでは必要な要素を指定して出力している。 ここで導出したモデルと特徴量を持たないモデルを比較する尤度比検定は次のように求めることができる。

In [9]:

```
cox_fit.log_likelihood_ratio_test()
```

Out[9]:

null_distribution	chi squared
degrees_freedom	1
test_name	log-likelihood ratio test

	test_statistic	p	-log2(p)
0	1.44	0.23	2.12

どのテストを使った場合でも、男性と女性間で生存期間に差があるという明確な証拠は得られない。 Cox モデルからのスコア検定は対数順位検定統計量に等しい。

次に、追加の予測変数を用いたモデルを適合させてみよう。その前に、`diagnosis` には欠損値が存在することに注意しておく。 ここでは、欠損値処理として、欠損値となるデータを削除しておく。

In [10]:

```
cleaned = BrainCancer.dropna()
all_MS = MS(cleaned.columns, intercept=False)
all_df = all_MS.fit_transform(cleaned)
fit_all = coxph().fit(all_df,
                      'time',
                      'status')
fit_all.summary[['coef', 'se(coef)', 'p']]
```

Out[10]:
coef
se(coef)
p
covariate
sex[Male]
0.183748
0.360358
0.610119
diagnosis[LG glioma]
-1.239530
0.579555
0.032455
diagnosis[Meningioma]
-2.154566
0.450524
0.000002
diagnosis[Other]
-1.268870
0.617672
0.039949
loc[Supratentorial]
0.441195
0.703669
0.530665
ki

-0.054955
0.018314
0.002693
gtv
0.034293
0.022333
0.124661
stereo[SRT]
0.177778
0.601578
0.767597

`diagnosis` 変数は、高悪性度の神経膠腫（グリオーマ）がベースラインとなるようにコード化されている。 その結果、高悪性度の神経膠腫に関連するリスクは髄膜腫に関連するリスクの 8 倍以上（$e^{2.15}$=8.62）である。 言い換えれば、他の予測因子で調整した後では、高悪性度の神経膠腫の生存率は髄膜腫患者の生存率よりはるかに悪いことになる。 さらに、カルノフスキー指数である `ki` の値が大きいほど、リスクが低い、すなわち生存期間が長いことになる。

In [11]:

```
levels = cleaned['diagnosis'].unique()
def representative(series):
    if hasattr(series.dtype, 'categories'):
        return pd.Series.mode(series)
    else:
        return series.mean()
modal_data = cleaned.apply(representative, axis=0)
```

各列の平均をコピーしたものを `diagnosis` に含まれる要素ごとに作成しよう。

In [12]:

```
modal_df = pd.DataFrame(
              [modal_data.iloc[0] for _ in range(len(levels))])
modal_df['diagnosis'] = levels
modal_df
```

Out[12]:

	sex	diagnosis	loc	Ki	Gtv	stereo	status	Time
0	Female	Meningioma	Supratentorial	80.91954	8.687011	SRT	0.402299	27.188621

0	Female	HG glioma	Supratentorial	80.91954	8.687011	SRT	0.402299	27.188621
0	Female	LG glioma	Supratentorial	80.91954	8.687011	SRT	0.402299	27.188621
0	Female	Other	Supratentorial	80.91954	8.687011	SRT	0.402299	27.188621

次に、モデルの適合に使用されたモデル仕様 `all_MS` に基づいてモデル行列を構築し、行に `diagnosis` のレベルに応じた名前を付けていく。

In [13]:

```
modal_X = all_MS.transform(modal_df)
modal_X.index = levels
modal_X
```

Out[13]:

	sex[Male]	diagnosis[LG glioma]	diagnosis[Meningioma]	diagnosis[Other]	loc[Supratentorial]	ki	gtv	stereo[SRT]	Status	time
Meningioma	0.0	0.0	1.0	0.0	1.0	80.91954	8.687011	1.0	0.402299	27.188621
HG glioma	0.0	0.0	0.0	0.0	1.0	80.91954	8.687011	1.0	0.402299	27.188621
LG glioma	0.0	1.0	0.0	0.0	1.0	80.91954	8.687011	1.0	0.402299	27.188621
Other	0.0	0.0	0.0	1.0	1.0	80.91954	8.687011	1.0	0.402299	27.188621

このデータに `predict_survival_function()` メソッドを用いて推定生存関数を得ることが可能となる。

In [14]:

```
predicted_survival = fit_all.predict_survival_function(modal_X)
predicted_survival
```

Out[14]:

	Meningioma	HG glioma	LG glioma	Other
0.07	0.997947	0.982430	0.994881	0.995029
1.18	0.997947	0.982430	0.994881	0.995029
1.41	0.995679	0.963342	0.989245	0.989555
1.54	0.995679	0.963342	0.989245	0.989555
2.03	0.995679	0.963342	0.989245	0.989555
...	...	...	...	...

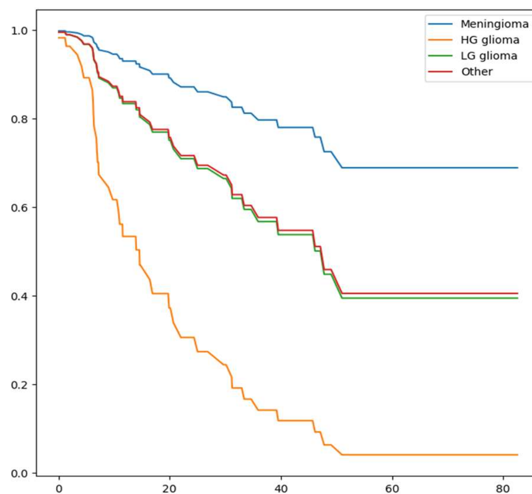
65.02	0.688772	0.040136	0.394181	0.404936
67.38	0.688772	0.040136	0.394181	0.404936
73.74	0.688772	0.040136	0.394181	0.404936
78.75	0.688772	0.040136	0.394181	0.404936
82.56	0.688772	0.040136	0.394181	0.404936

85 rows × 4 columns

この方法ではデータ・フレームを返す。データ・フレームには信頼区間などさまざまな生存曲線が含まれているが、ここでは図の乱雑さを避けるために信頼区間を表示しない。

In [15]:

```
fig, ax = subplots(figsize=(8, 8))
predicted_survival.plot(ax=ax);
```

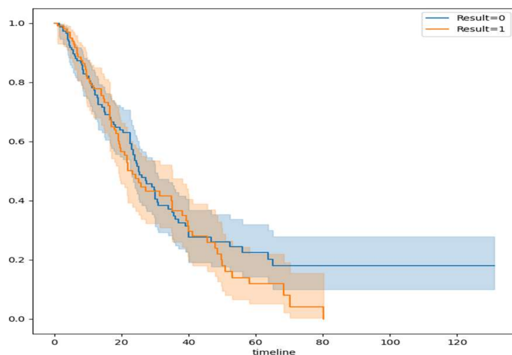


公刊データ(Publication Data)

本節では、`Publication` データを扱う。このデータも `ISLP` パッケージに含まれているものである。 まず `posres` 変数で層別化した Kaplan-Meier 曲線を確認しよう。

In [16]:

```
fig, ax = subplots(figsize=(8,8))
Publication = load_data('Publication')
by_result = {}
for result, df in Publication.groupby('posres'):
    by_result[result] = df
    km_result = km.fit(df['time'], df['status'])
    km_result.plot(label='Result=%d' % result, ax=ax)
```



Cox の比例ハザードモデルを `posres` 変数に当てはめると、p 値はかなり大きくなる。 これは肯定的な結果を得た研究と否定的な結果を得た研究の間で、論文発表までの時間に差があることを示すことにはならない。

In [19]:

```
posres_df = MS(['posres',
                'time',
                'status'],
                intercept=False).fit_transform(Publication)
posres_fit = coxph().fit(posres_df,
                         'time',
                         'status')
posres_fit.summary[['coef', 'se(coef)', 'p']]
```

Out[19]:
coef
se(coef)
p

covariate
posres
0.148076
0.161625
0.359579

しかし、他の予測因子をモデルに含めると、結果は次のようになる。 ここでは、資金調達メカニズム変数を除外している。

In [17]:

```
model = MS(Publication.columns.drop('mech'),
           intercept=False)
coxph().fit(model.fit_transform(Publication),
            'time',
            'status').summary[['coef', 'se(coef)', 'p']]
```

Out[17]:
coef
se(coef)
p
covariate
posres
0.570773
0.175960
1.179610e-03
multi
-0.040860
0.251194
8.707842e-01
clinend
0.546183
0.262000
3.709944e-02
sampsize
0.000005
0.000015
7.507005e-01
budget
0.004386
0.002465
7.515984e-02
impact
0.058318
0.006676
2.426306e-18

統計的に有意な変数が多数あることがわかる。具体的には、臨床エンドポイントに焦点をあてた試験かどうか、試験の影響度、試験結果が肯定的か否定的か、などである。

コールセンターデータ(Call Center Data)

このセクションでは、累積ハザードと生存率の関係を用いて生存データのシミュレーションを行う。 シミュレーション・データは、コールセンターに電話した 2,000 人の顧客の待ち時間（秒）である。 ここでは、電話に出る前に電話が切られてしまうと打ち切りが発生する。

共変量は３つある： １つは交換手(コールセンターのオペレーター)の数であり、5 から 15 までをとる）、次に`センター`（A、B、C のいずれか）、最後に`時間帯`（午前、午後、夕方のいずれか）である。 これらの共変量は、すべての要素の数が等しくなるようにデータを生成する。

In [18]:

```
rng = np.random.default_rng(10)
N = 2000
Operators = rng.choice(np.arange(5, 16),
                       N,
                       replace=True)
Center = rng.choice(['A', 'B', 'C'],
                    N,
                    replace=True)
Time = rng.choice(['Morn.', 'After.', 'Even.'],
                   N,
                   replace=True)
D = pd.DataFrame({'Operators': Operators,
                  'Center': pd.Categorical(Center),
                  'Time': pd.Categorical(Time)})
```

切片は省略し、モデル行列を作成する。

In [19]:

```
model = MS(['Operators',
            'Center',
            'Time'],
```

```
            intercept=False)
X = model.fit_transform(D)
```

モデル行列`X`を確認してみよう。デフォルトでは、カテゴリカル変数のレベルはソートされ、変数の 1-hot エンコーディングの最初の列は削除される。

In [20]:

```
X[:5]
```

Out[20]:

	Operators	Center[B]	Center[C]	Time[Even.]	Time[Morn.]
0	13	0.0	1.0	0.0	0.0
1	15	0.0	0.0	1.0	0.0
2	7	1.0	0.0	0.0	1.0
3	7	0.0	1.0	0.0	1.0
4	13	0.0	1.0	1.0	0.0

次に、係数とハザード関数を指定する。

In [23]:

```
true_beta = np.array([0.04, -0.3, 0, 0.2, -0.2])
true_linpred = X.dot(true_beta)
hazard = lambda t: 1e-5 * t
```

ここでは、`演算子`に関連する係数を 0.04 とした。 言い換えると、オペレーターが 1 人増えるごとに通話が応答されるリスクが $e^{0.04}$=1.041 倍になるため、オペレータの数が多ければ多いほど、待ち時間は短くなる。`Center==B`に関連する係数は-0.3 であり、`Center == A`がベースラインとして扱われる。 これはセンターB で電話に出るリスクは、センターA で電話に出るリスクの 0.74 倍であることを示しており、待ち時間はセンターB の方が少し長い。

この関数は、生存関数と累積ハザード S(t) = exp(-H(t)) の関係と、Cox モデルにおける累積ハザード関数の特定の形式を用いて、線形予測変数である`true_linpred`の値と累積ハザードに基づいてデータをシミュレートしてみる。我々は、累積ハザード関数を与える必要がある。

In [24]:

```
cum_hazard = lambda t: 1e-5 * t**2 / 2
```

この作業により Cox 比例アザーズ・モデルでデータを作成する準備ができた。 シミュレーションの待ち時間を最大 1000 秒として設定する。`sim_time()`関数は、線形予測変数、累積ハザード関数、乱数発生器を引数として設定する必要がある。

In [25]:

```
W = np.array([sim_time(l, cum_hazard, rng)
              for l in true_linpred])
D['Wait time'] = np.clip(W, 0, 1000)
```

ここで、打ち切り変数をシミュレートしよう。顧客が電話を切る(`Failed==0`)前に、90%の通話が応答された(`Failed==1`)と仮定する。

In [26]:

```
D['Failed'] = rng.choice([1, 0],
                         N,
                         p=[0.9, 0.1])
D[:5]
```

Out[26]:

	Operators	Center	Time	Wait time	Failed
0	13	C	After.	525.064979	1
1	15	A	Even.	254.677835	1
2	7	B	Morn.	487.739224	1
3	7	C	Morn.	308.580292	1
4	13	C	Even.	154.174608	1

In [27]:

```
D['Failed'].mean()
```

Out[27]:

```
0.9075
```

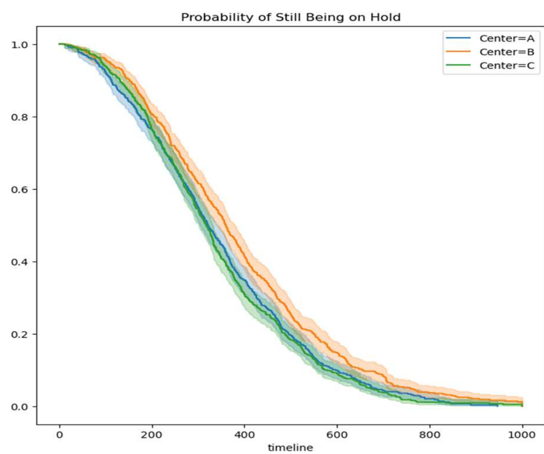
次にカプラン・マイヤー生存曲線を `Center` で層別化して可視化する。

In [28]:

```
fig, ax = subplots(figsize=(8,8))
by_center = {}
for center, df in D.groupby('Center'):
    by_center[center] = df
    km_center = km.fit(df['Wait time'], df['Failed'])
    km_center.plot(label='Center=%s' % center, ax=ax)
ax.set_title("Probability of Still Being on Hold")
```

Out[28]:

```
Text(0.5, 1.0, 'Probability of Still Being on Hold')
```



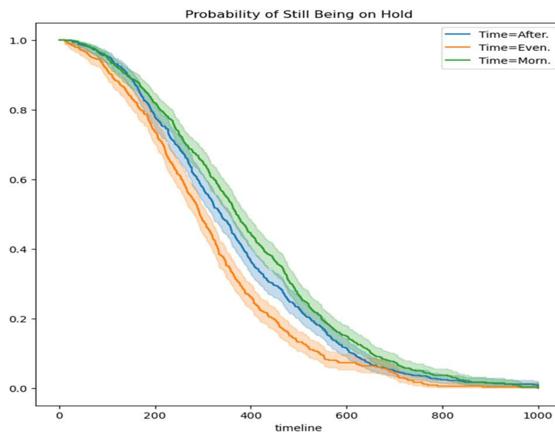
次に `Time` で層別化する。

In [29]:

```
fig, ax = subplots(figsize=(8,8))
by_time = {}
for time, df in D.groupby('Time'):
    by_time[time] = df
    km_time = km.fit(df['Wait time'], df['Failed'])
    km_time.plot(label='Time=%s' % time, ax=ax)
ax.set_title("Probability of Still Being on Hold")
```

Out[29]:

```
Text(0.5, 1.0, 'Probability of Still Being on Hold')
```



コールセンターB では、コールセンターA や C よりも、電話に出るのに時間がかかるようである。同様に、待ち時間は午前中が最も長く、夕方の時間帯が最も短い。`multivariate_logrank_test()`関数を用いると、これらの差が統計的に有意かどうかを対数順位検定で調べることができる。

In [30]:

```
multivariate_logrank_test(D['Wait time'],
                          D['Center'],
                          D['Failed'])
```

Out[30]:

t_0	-1
null_distribution	chi squared
degrees_of_freedom	2
test_name	multivariate_logrank_test

	test_statistic	p	-log2(p)
0	20.30	<0.005	14.65

次に、`Time` の影響について考えていこう。

In [31]:

```
multivariate_logrank_test(D['Wait time'],
                          D['Time'],
                          D['Failed'])
```

Out[31]:

t_0	-1
null_distribution	chi squared
degrees_of_freedom	2
test_name	multivariate_logrank_test

	test_statistic	p	-log2(p)
0	49.90	<0.005	35.99

2 水準を持つカテゴリー変数の場合と同様に、これらの結果は Cox 比例ハザードモデルからの尤度比検定と同様である。まず、`Center` の結果を見ていこう。

In [32]:

```
X = MS(['Wait time',
        'Failed',
        'Center'],
        intercept=False).fit_transform(D)
F = coxph().fit(X, 'Wait time', 'Failed')
F.log_likelihood_ratio_test()
```

Out[32]:

null_distribution	chi squared
degrees_freedom	2

test_name log-likelihood ratio test

	test_statistic	p	-log2(p)
0	20.58	<0.005	14.85

次に、`Time` の結果を見ていこう。

In [33]:

```
X = MS(['Wait time',
        'Failed',
        'Time'],
       intercept=False).fit_transform(D)
F = coxph().fit(X, 'Wait time', 'Failed')
F.log_likelihood_ratio_test()
```

Out[33]:

null_distribution chi squared
degrees_freedom 2
test_name log-likelihood ratio test

	test_statistic	P	-log2(p)
0	48.12	<0.005	34.71

センター間の差は、時間帯による差と同様に非常に有意であることがわかった。

最後に、Cox の比例ハザードモデルをデータにフィットする。

In [35]:

```
X = MS(D.columns,
       intercept=False).fit_transform(D)
fit_queuing = coxph().fit(
                  X,
                 'Wait time',
                 'Failed')
fit_queuing.summary[['coef', 'se(coef)', 'p']]
```

Out[35]:
coef
se(coef)

p
covariate
Operators
0.043934
0.007520
5.143589e-09
Center[B]
-0.236060
0.058113
4.864162e-05
Center[C]
0.012231
0.057518
8.316096e-01
Time[Even.]
0.268845
0.057797
3.294956e-06
Time[Morn.]
-0.148217
0.057334
9.733557e-03

センターB と夕方の p 値は非常に小さい。また、ハザード、つまり、電話に出るリスクは、オペレータの数とともに増加することも明らかである。

このデータはシミュレーションによって作成されたものなので、`オペレーター`、`センター=B`、`センター=C`、`時間=夕方`、`時間=午前`の真の係数は、それぞれ 0.04、-0.3、0、0.2、-0.2 であることがわかる。これと比較すると、フィットした Cox モデルからの係数推定はかなり正確であることが分かる。

ISLP 第 12 章 教師なし学習

このラボでは、いくつかのデータセットに対して PCA（主成分分析）とクラスタリングを実装する。 他のラボと同様に、最初にいくつかのライブラリをインポートする。 この作業により、コードが読みやすくなり、ノートブックの最初の数行でどのライブラリが使用されているかを把握できる。

訳注：Google Colab でこのコードを実行した。Colab が不安定な場合(2025 年 3 月 20 日頃)には「ランタイムからセッションを再起動する」と問題ないことがあるが、Jupyter ではプログラムは正常に動作することを確認した。(Numpy などパッケージ更新のタイミングによるのではと思われる。)また、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。

In [1]:

```
# 訳者追加
# %%capture は出力を非表示にするために、出力が長くなる箇所で用いている
# もちろん%%capture を消したりコメントアウトしたりすれば、出力が表示される
%%capture
!pip install japanize-matplotlib
```

In [2]:

```
# 訳者追加
%%capture
!pip install ISLP # ISLP は初めからインストールされてはいないのでインストールする。
```

In [3]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import japanize_matplotlib # グラフで日本語を使うために訳者が追加した
from statsmodels.datasets import get_rdataset
from sklearn.decomposition import PCA
from sklearn.preprocessing import StandardScaler
from ISLP import load_data
```

さらに、このラボで必要なライブラリをインポートする。

In [4]:

```
from sklearn.cluster import \
     (KMeans,
      AgglomerativeClustering)
from scipy.cluster.hierarchy import \
     (dendrogram,
      cut_tree)
from ISLP.cluster import compute_linkage
```

訳注：\は長い行を分割するために用いられている。

主成分分析（Principal Components Analysis, PCA）

このラボでは、`USArrests`というデータセットに対して PCA を行う。このデータセットは、`R`の計算環境に内蔵されている。`get_rdataset()`を使用してデータを取得する。この関数を使って、多くの標準的な`R`パッケージからデータを取得できる。データセットの行は、50 州をアルファベット順に並べたものである。

In [5]:

```
USArrests = get_rdataset('USArrests').data
USArrests
```

Out[5]:

rownames	Murder	Assault	UrbanPop	Rape
Alabama	13.2	236	58	21.2
Alaska	10.0	263	48	44.5
Arizona	8.1	294	80	31.0
Arkansas	8.8	190	50	19.5
California	9.0	276	91	40.6
Colorado	7.9	204	78	38.7
Connecticut	3.3	110	77	11.1
Delaware	5.9	238	72	15.8
Florida	15.4	335	80	31.9
Georgia	17.4	211	60	25.8
Hawaii	5.3	46	83	20.2
Idaho	2.6	120	54	14.2
Illinois	10.4	249	83	24.0
Indiana	7.2	113	65	21.0
Iowa	2.2	56	57	11.3
Kansas	6.0	115	66	18.0
Kentucky	9.7	109	52	16.3
Louisiana	15.4	249	66	22.2
Maine	2.1	83	51	7.8
Maryland	11.3	300	67	27.8
Massachusetts	4.4	149	85	16.3
Michigan	12.1	255	74	35.1
Minnesota	2.7	72	66	14.9
Mississippi	16.1	259	44	17.1
Missouri	9.0	178	70	28.2
Montana	6.0	109	53	16.4
Nebraska	4.3	102	62	16.5
Nevada	12.2	252	81	46.0
New Hampshire	2.1	57	56	9.5
New Jersey	7.4	159	89	18.8
New Mexico	11.4	285	70	32.1
New York	11.1	254	86	26.1
North Carolina	13.0	337	45	16.1
North Dakota	0.8	45	44	7.3
Ohio	7.3	120	75	21.4
Oklahoma	6.6	151	68	20.0
Oregon	4.9	159	67	29.3
Pennsylvania	6.3	106	72	14.9
Rhode Island	3.4	174	87	8.3
South Carolina	14.4	279	48	22.5
South Dakota	3.8	86	45	12.8
Tennessee	13.2	188	59	26.9
Texas	12.7	201	80	25.5
Utah	3.2	120	80	22.9
Vermont	2.2	48	32	11.2
Virginia	8.5	156	63	20.7
Washington	4.0	145	73	26.2
West Virginia	5.7	81	39	9.3
Wisconsin	2.6	53	66	10.8
Wyoming	6.8	161	60	15.6

データセットの列には、4 つの変数が含まれている。

In [6]:

```
USArrests.columns
```

Out[6]:

```
Index(['Murder', 'Assault', 'UrbanPop', 'Rape'], dtype='object')
```

まずは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と気づくだろう。

In [7]:

```
USArrests.mean()
```

Out[7]:

Murder	7.788
Assault	170.760
UrbanPop	65.540
Rape	21.232

dtype: float64

データフレームには、列ごとの要約統計を計算するための便利な方法がある。`var()`メソッドを使用して、4 つの変数の分散も調べてみよう。

In [8]:

```
USArrests.var()
```

Out[8]:

Murder	18.970465
Assault	6945.165714
UrbanPop	209.518776
Rape	87.729159

dtype: float64

予想通り、変数の分散も大きく異なる。`UrbanPop` 変数は、各州の都市地域に住む人口の割合を測定しており、これは 100,000 人あたりのレイプ数と比較できるものではない。 PCA は、データセットの大部分の分散を説明する合成変数を探索する。ただし PCA を実施する前に変数を標準化しなければ、主成分は主に `Assault`(暴行)変数によって支配されることになる。なぜなら、それが圧倒的に大きな分散を持っているからである。 したがって、変数が異なる単位で測定されている場合や、スケールが大きく異なる場合は、PCA を行う前に変数を標準化して、標準偏差が 1 になるようにする事が推奨される。通常、平均値もゼロに設定する。

この標準化は、上記でインポートした `StandardScaler()`変換を使って行うことができる。まずスケーラーを `fit` し、必要な平均値と標準偏差を計算した後、`transform` メソッドをデータに適用する。この章以前と同様に、`fit_transform()`メソッドを使ってこれらのステップを組み合わせる。

In [9]:

```
scaler = StandardScaler(with_std=True,
                        with_mean=True)
USArrests_scaled = scaler.fit_transform(USArrests)
```

データを標準化したので、次に sklearn.decomposition パッケージの PCA()変換を使って主成分分析を行う。

In [10]:

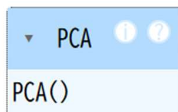
```
pcaUS = PCA()
```

デフォルトでは、PCA()変換は変数を中心化して平均をゼロにするが、スケーリングは行われない。変換 pcaUS を使って、fit()メソッドで返される PCA の scores を計算する。Fit メソッドを呼び出すと、pcaUS オブジェクトにはいくつかの有用な値も含まれる。

In [11]:

```
pcaUS.fit(USArrests_scaled)
```

Out[11]:



主成分分析の後、mean_属性は変数の平均を表す。ここでは、scaler()でデータを中心化して標準化したため、平均はすべて 0 になる。

In [12]:

```
pcaUS.mean_
```

Out[12]:

```
array([-7.10542736e-17,  1.38777878e-16, -4.39648318e-16,  8.59312621e-16])
```

フィットした後、スコアは`pcaUS`の`transfor_`メソッドを用いて計算できる。

In [13]:

```
scores = pcaUS.transform(USArrests_scaled)
```

これらのスコアは後でプロットする。`components_`属性は、主成分負荷（principal component loadings）を与える：`pcaUS.components_`の各行は、対応する主成分の負荷ベクトルである。

In [14]:

```
pcaUS.components_
```

Out[14]:

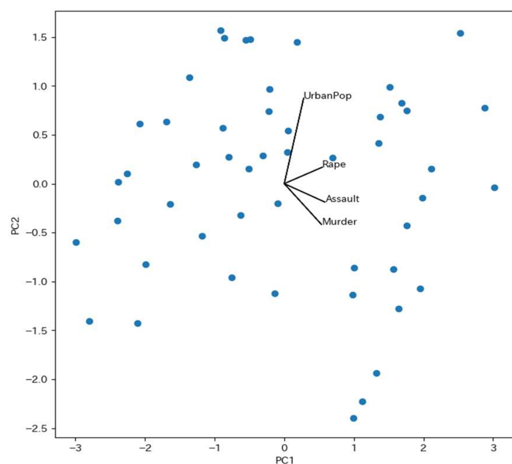
```
array([[ 0.53589947,  0.58318363,  0.27819087,  0.54343209],
       [-0.41818087, -0.1879856 ,  0.87280619,  0.16731864],
       [-0.34123273, -0.26814843, -0.37801579,  0.81777791],
       [-0.6492278 ,  0.74340748, -0.13387773, -0.08902432]])
```

`biplot`は PCA でよく使用される視覚化方法である。`sklearn`の標準的なものとしては組み込まれていないが、このようなプロットを作成するための Python パッケージはある。ここでは、シンプルな biplot を手動で作成してみる。

In [15]:

```
i, j = 0, 1 # どの主成分をプロットするか
fig, ax = plt.subplots(1, 1, figsize=(8, 8))
ax.scatter(scores[:,0], scores[:,1])
ax.set_xlabel('PC%d' % (i+1))
ax.set_ylabel('PC%d' % (j+1))
for k in range(pcaUS.components_.shape[1]):
    ax.arrow(0, 0, pcaUS.components_[i,k], pcaUS.components_[j,k])
```

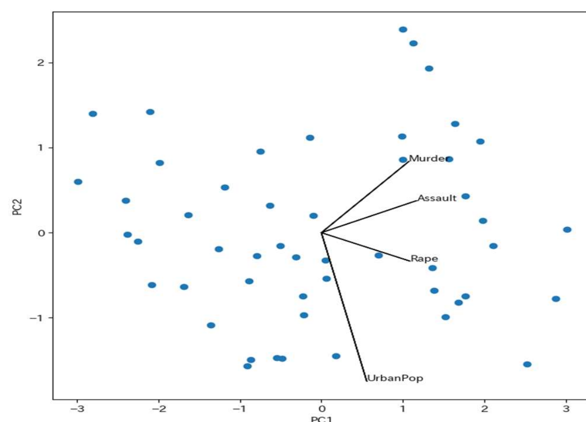
```
    ax.text(pcaUS.components_[i,k],
            pcaUS.components_[j,k],
            USArrests.columns[k])
```



この図は、図 12.1 の y 軸を反転させたものである。主成分は符号の変更を除いて一意であるため、2 番目のスコアと負荷の符号を反転させることで、その図を再現することができる。 また、矢印を長くして負荷を強調した。

In [16]:

```
scale_arrow = s_ = 2
scores[:,1] *= -1
pcaUS.components_[1] *= -1 # y 軸を反転
fig, ax = plt.subplots(1, 1, figsize=(8, 8))
ax.scatter(scores[:,0], scores[:,1])
ax.set_xlabel('PC%d' % (i+1))
ax.set_ylabel('PC%d' % (j+1))
for k in range(pcaUS.components_.shape[1]):
    ax.arrow(0, 0, s_*pcaUS.components_[i,k], s_*pcaUS.components_[j,k])
    ax.text(s_*pcaUS.components_[i,k],
            s_*pcaUS.components_[j,k],
            USArrests.columns[k])
```



主成分スコアの標準偏差は以下の通りである。

In [17]:

```
scores.std(0, ddof=1)
```

Out[17]:

```
array([1.5908673 , 1.00496987, 0.6031915 , 0.4206774 ])
```

各スコアの分散は、`pcaUS` オブジェクトの `explained_variance_` 属性から直接に取得できる。

In [18]:

```
pcaUS.explained_variance_
```

Out[18]:

```
array([2.53085875, 1.00996444, 0.36383998, 0.17696948])
```

各主成分によって説明される分散の割合（Proportion of Variance Explained, PVE）は、`explained_variance_ratio_` として格納されている。

In [19]:

```
pcaUS.explained_variance_ratio_
```

Out[19]:

```
array([0.62006039, 0.24744129, 0.0891408 , 0.04335752])
```

最初の主成分がデータの分散の 62.0%を説明し、次の主成分は分散の 24.7%を説明することが分かる。さらに、各主成分によって説明される PVE と累積 PVE をプロットできる。まず、分散の割合をプロットしてみよう。

In [20]:

```
%%capture
fig, axes = plt.subplots(1, 2, figsize=(15, 6))
ticks = np.arange(pcaUS.n_components_)+1
ax = axes[0]
ax.plot(ticks,
        pcaUS.explained_variance_ratio_,
        marker='o')
ax.set_xlabel('主成分')
ax.set_ylabel('説明された分散の割合')
ax.set_ylim([0,1])
ax.set_xticks(ticks)
```

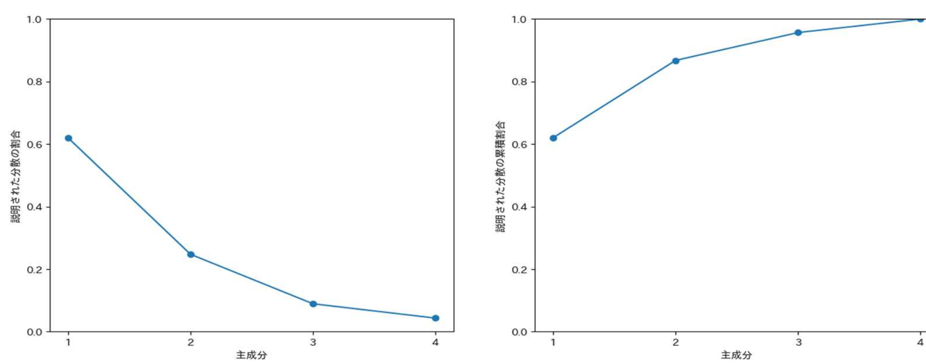
`%%capture` の使用に注目しておこう。これは、この段階では図が完成していないので、それを表示しないようにしている。

In [21]:

```
ax = axes[1]
ax.plot(ticks,
        pcaUS.explained_variance_ratio_.cumsum(),
        marker='o')
```

```
ax.set_xlabel('主成分')
ax.set_ylabel('説明された分散の累積割合')
ax.set_ylim([0, 1])
ax.set_xticks(ticks)
fig
```

Out[21]:



結果は、図 12.3 に示されたものと類似している。`cumsum()`メソッドにより数値ベクトルの要素の累積和が計算される。例えば：

In [22]:

```
a = np.array([1,2,8,-3])
np.cumsum(a)
```

Out[22]:

```
array([ 1,  3, 11,  8])
```

行列補完

次に、`USArrests`データに対する分析を再び行う。これは 12.3 節と同様である。

12.2.2 節では、センタリングされたデータ行列 **X** に対して最適化問題(12.6)を解くことが、データの最初の M 個の主成分を計算することと同等であることを見た。以下では、スケーリングおよびセンタリングされた `USArrests` データを **X** として使用する。なお*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。

In [23]:

```
X = USArrests_scaled
U, D, V = np.linalg.svd(X, full_matrices=False)
U.shape, D.shape, V.shape
```

Out[23]:

```
((50, 4), (4,), (4, 4))
```

`np.linalg.svd()`関数は、`U`、`D`、`V` の 3 つのコンポーネントを返す。行列 `V` は主成分負荷行列に相当する（符号は反転しうるが大きな問題ではない）。`full_matrices=False` オプションを使用することで、縦長の行列に対して `U` の形状を `X` の形状と一致させられる。

In [24]:

```
V
```

Out[24]:

```
array([[-0.53589947, -0.58318363, -0.27819087, -0.54343209],
       [-0.41818087, -0.1879856 ,  0.87280619,  0.16731864],
       [ 0.34123273,  0.26814843,  0.37801579, -0.81777791],
       [ 0.6492278 , -0.74340748,  0.13387773,  0.08902432]])
```

In [25]:

```
pcaUS.components_
```

Out[25]:

```
array([[ 0.53589947,  0.58318363,  0.27819087,  0.54343209],
       [ 0.41818087,  0.1879856 , -0.87280619, -0.16731864],
       [-0.34123273, -0.26814843, -0.37801579,  0.81777791],
       [-0.6492278 ,  0.74340748, -0.13387773, -0.08902432]])
```

行列`U`は、主成分スコア行列の*標準化された*バージョンに相当する（各列で平方和が 1 になるように標準化されている）。`U`の各列を`D`の対応する要素で乗算することで、主成分スコアを正確に復元できる（符号の反転は起きうるが、重要な問題ではない）。

In [26]:

```
(U * D[None,:])[:3]
```

Out[26]:

```
array([[-0.98556588, -1.13339238,  0.44426879,  0.15626714],
       [-1.95013775, -1.07321326, -2.04000333, -0.43858344],
       [-1.76316354,  0.74595678, -0.05478082, -0.83465292]])
```

In [27]:

```
scores[:3]
```

Out[27]:

```
array([[ 0.98556588,  1.13339238, -0.44426879, -0.15626714],
       [ 1.95013775,  1.07321326,  2.04000333,  0.43858344],
       [ 1.76316354, -0.74595678,  0.05478082,  0.83465292]])
```

このラボを`PCA()`を使用して実行することも可能であるが、ここではその利用法を説明するために`np.linalg.svd()`関数を用いた。

次に、50×4 データ行列からランダムに 20 個の要素を除く。これは、最初に 20 行（州）をランダムに選び、次に各行で 4 つのエントリのうちの 1 つをランダム

に選択することによって行う。これにより、各行には少なくとも 3 つの観測値が残る。

In [28]:

```
n_omit = 20
np.random.seed(15)
r_idx = np.random.choice(np.arange(X.shape[0]), # 欠損値がある州を表す
                         n_omit,
                         replace=False)
c_idx = np.random.choice(np.arange(X.shape[1]), # 欠損値がある特徴を表す
                         n_omit,
                         replace=True)
Xna = X.copy()
Xna[r_idx, c_idx] = np.nan
```

ここで、配列`r_idx`は 0 から 49 の中の 20 個の整数を含む。これは州(`X`の行)に関するもので、欠損値を含むように選ばれた州を表している。`c_idx`は 20 個の整数を含み、それぞれ 0 から 3 までの値である。これらは選択された州において欠損値が含まれる特徴（`X`の列）を表す。

次に、アルゴリズム 12.1 を実行するためのコードを書こう。最初に、行列を入力として受け取り、`svd()`関数を使用してその行列の近似を返す関数を書く。これはアルゴリズム 12.1 のステップ 2 で必要となる。

In [29]:

```
def low_rank(X, M=1):
    U, D, V = np.linalg.svd(X)
    L = U[:,:M] * D[None,:M]
    return L.dot(V[:M])
```

アルゴリズムのステップ 1 を実行するために、`Xhat`（これはアルゴリズム 12.1 の $\tilde{X}$）を初期化する(欠損値を何らかの形で補完する)。初期化として具体的には、欠損値を欠損していない要素の列平均で置き換える。列平均は、`np.nanmean()`を行

軸に対して実行することで計算され、`Xbar`に保存される。値を`Xhat`に代入する際、`Xna`の値を上書きしないようにコピーを作成しておく。

In [30]:

```
Xhat = Xna.copy()
Xbar = np.nanmean(Xhat, axis=0)
Xhat[r_idx, c_idx] = Xbar[c_idx]
```

ステップ 2 を開始する前に、繰り返しの進行度を測定する準備を整える。

In [31]:

```
thresh = 1e-7
rel_err = 1
count = 0
ismiss = np.isnan(Xna) # 欠損要素に対応する部分は True となる論理行列
mssold = np.mean(Xhat[~ismiss]**2) # Xhat の非欠損要素の 2 乗の平均。~は numpy の Tr
ue/False を反転させる。numpy のブール値を用いないと、~True は-2 を返す
mss0 = np.mean(Xna[~ismiss]**2) # X の非欠損要素の 2 乗の平均
```

ここで、`ismiss`は`Xna`と同じ次元を持つ論理行列である。ある要素が`True`の場合、それに対応する行列の要素が欠損していることを示している。`~ismiss`は~によってブールベクトル`ismiss`を反転させる。この操作により、欠損値と非欠損値の両方にアクセスできる。非欠損要素の 2 乗の平均は`mss0`に格納される。`Xhat`の古いバージョンの非欠損要素の 2 乗の平均を`mssold`に格納される（現在は`mss0`と一致）。`Xhat`の現在のバージョンの非欠損要素の平均 2 乗誤差が`mss`に格納され、相対誤差`(mssold - mss) / mss0`が`thresh = 1e-7`より小さくなるまでアルゴリズム 12.1 のステップ 2 を繰り返す。

アルゴリズム 12.1 では、ステップ 2 を繰り返して(12.14)が減少しなくなるまで続ける。ここで(12.14)が減少しているかどうかを確認するためには、`mssold - mss`だけを追跡すれば良いが、実際には`(mssold - mss) / mss0`を追跡するとよい。この作業により、アルゴリズム 12.1 が収束するために必要な繰り返し回数が、データ X に定数を掛けたかどうかに依存しないようにする。

アルゴリズム 12.1 のステップ 2(a)では、`low_rank()`を使用して `Xhat` を近似しているが、この近似を `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 つのステップは次の `while` ループに含まれている。

In [32]:

```
while rel_err > thresh:
    count += 1
    # ステップ 2(a)
    Xapp = low_rank(Xhat, M=1)
    # ステップ 2(b)
    Xhat[ismiss] = Xapp[ismiss]
    # ステップ 2(c)
    mss = np.mean(((Xna - Xapp)[~ismiss])**2)
    rel_err = (mssold - mss) / mss0
    mssold = mss
    print("繰り返し回数: {0}, MSS:{1:.3f}, 相対誤差 {2:.2e}"
          .format(count, mss, rel_err))
```

Out[32]:

```
繰り返し回数: 1, MSS:0.395, 相対誤差 5.99e-01
繰り返し回数: 2, MSS:0.382, 相対誤差 1.33e-02
繰り返し回数: 3, MSS:0.381, 相対誤差 1.44e-03
繰り返し回数: 4, MSS:0.381, 相対誤差 1.79e-04
繰り返し回数: 5, MSS:0.381, 相対誤差 2.58e-05
繰り返し回数: 6, MSS:0.381, 相対誤差 4.22e-06
繰り返し回数: 7, MSS:0.381, 相対誤差 7.65e-07
繰り返し回数: 8, MSS:0.381, 相対誤差 1.48e-07
繰り返し回数: 9, MSS:0.381, 相対誤差 2.95e-08
```

8 回の繰り返し後、相対誤差は `thresh = 1e-7` を下回り、アルゴリズムは終了した。この時点で、非欠損要素の平均 2 乗誤差は 0.381 に達している。

最後に、20 個の補完値と実際の値との相関を計算する。

In [33]:

```
np.corrcoef(Xapp[ismiss], X[ismiss])[0,1]
```

Out[33]:

```
0.7113567434297361
```

このラボでは、教育的な目的でアルゴリズム 12.1 を実装してみた。しかし、行列補完を自分のデータに適用したい読者は、より専門的な `Python` での実装を参照するとよいだろう。

クラスタリング (Clustering)

K-平均クラスタリング (K-Means Clustering)

`sklearn.cluster.KMeans()`は、`Python` で K-平均クラスタリングを実行する。まず、データに実際に 2 つのクラスタが存在する簡単なシミュレーション例を見てみよう。最初の 25 個の観測値は次の 25 個の観測値に対して平均的にシフトしている。

In [34]:

```
np.random.seed(0);
X = np.random.standard_normal((50,2));
X[:25,0] += 3;
X[:25,1] -= 4;
```

次に、K=2 で K-平均クラスタリングを実行する。

In [35]:

```
kmeans = KMeans(n_clusters=2,
                random_state=2,
                n_init=20).fit(X)
```

`random_state`を指定することで、結果を再現可能にする。50 個の観測値のクラスタ割り当ては、`kmeans.labels_`に格納されている。

In [36]:

```
kmeans.labels_
```

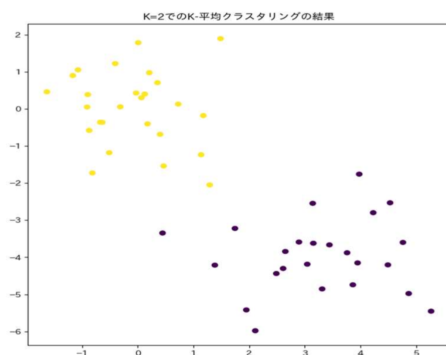
Out[36]:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
       0, 0, 0, 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], dtype=int32)
```

K-平均クラスタリングは、`KMeans()`にグループ情報を与えることなく、観測値を 2 つのクラスタに完全に分離した。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基づいて色分けすることができる。

In [37]:

```
fig, ax = plt.subplots(1, 1, figsize=(8,8))
ax.scatter(X[:,0], X[:,1], c=kmeans.labels_)
ax.set_title("K=2 での K-平均クラスタリングの結果");
```

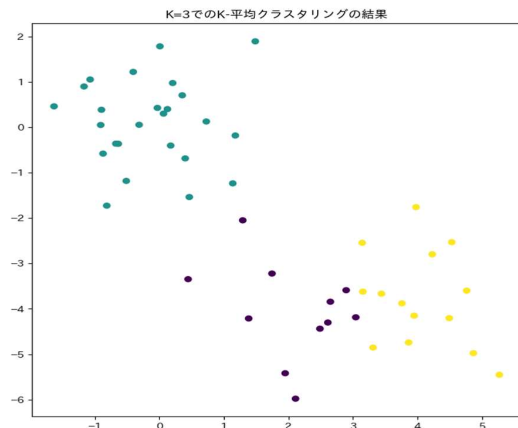


ここでは、観測値が 2 次元であるため、簡単にプロットできる。もし変数が 3 つ以上あった場合、代わりに PCA を実行し最初の 2 つの主成分スコアベクトルをプロットして、クラスタを表現することができる。

この例では、データを生成したので実際に 2 つのクラスタがあることが分っている。しかし、実際のデータでは、クラスタの数が本当はいくつなのか、またクラスタが厳密に存在するかどうかは分からない。代わりに、この例で K=3 で K-平均クラスタリングを実行してみよう。

In [38]:

```
kmeans = KMeans(n_clusters=3,
                random_state=3,
                n_init=20).fit(X)
fig, ax = plt.subplots(figsize=(8,8))
ax.scatter(X[:,0], X[:,1], c=kmeans.labels_)
ax.set_title("K=3 での K-平均クラスタリングの結果");
```



K=3 の場合、K-平均クラスタリングは 2 つのクラスタを分割する。`n_init` 引数を使用して、K-means を、初期クラスタを 20 回変えて実行した（デフォルトは 10 回である）。`n_init` が 1 より大きい場合、K-平均クラスタリングはアルゴリズム 12.2 のステップ 1 で複数のランダム割り当てを利用して実行され、`KMeans()`関数は最良の結果のみを報告する。ここでは、`n_init=1` と `n_init=20` を比較してみる。

In [39]:

```
kmeans1 = KMeans(n_clusters=3,
                random_state=3,
                n_init=1).fit(X)
kmeans20 = KMeans(n_clusters=3,
                  random_state=3,
                  n_init=20).fit(X);
kmeans1.inertia_, kmeans20.inertia_
```

Out[39]:

```
(76.85131986999251, 75.06261242745386)
```

`kmeans.inertia_`はクラスタ内平方和であり、これを K-平均クラスタリング(12.17)によって最小化を行う。

K-平均クラスタリングを実行する際は、常に`n_init`を 20 や 50 など大きな値に設定することを強く勧める。そうでないと、望ましくない局所最適解が得られる可能性がある。

K-平均クラスタリングを実行する際には、初期クラスタ割り当てを複数回行うだけでなく、`random_state`引数を使用してランダムシードを設定することも重要である。これにより、ステップ 1 での初期クラスタ割り当てが再現可能になり、K-means の出力が完全に再現可能となる。

階層的クラスタリング (Hierarchical Clustering)

`sklearn.clustering`パッケージの`AgglomerativeClustering()`クラスは、階層的クラスタリングを実装している。その名前が長いため、*Hierarchical Clustering* の略である`HClust`を使用しよう。この方法を使用しても返される型は変わらないので、インスタンスは引き続き`AgglomerativeClustering`クラスである。以下の例では、前のラボで使用したデータを用いて、完全連結法、単一連結法、および平均連結法を使用した階層的クラスタリングの樹形図(dendrogram)をプロットする。ここで非類似度としてはユークリッド距離を用いている。まず、完全連結法で観測値をクラスタリングしよう。

In [40]:

```
HClust = AgglomerativeClustering
hc_comp = HClust(distance_threshold=0,
                 n_clusters=None,
                 linkage='complete')
hc_comp.fit(X)
```

Out[40]:

```
AgglomerativeClustering
AgglomerativeClustering(distance_threshold=0, linkage='complete',
                        n_clusters=None)
```

これで樹形図全体が計算される。同じように、平均連結法や単一連結法を使用して階層的クラスタリングを実行することもできる：

In [41]:

```
hc_avg = HClust(distance_threshold=0,
                n_clusters=None,
                linkage='average');
hc_avg.fit(X)
hc_sing = HClust(distance_threshold=0,
                 n_clusters=None,
                 linkage='single');
hc_sing.fit(X);
```

事前に計算された距離行列を使用するには、追加の引数 `metric="precomputed"`を指定する必要がある。以下のコードでは、最初の 4 行で 50× 50 ペアワイズ距離行列を計算している。

In [42]:

```
D = np.zeros((X.shape[0], X.shape[0]));
for i in range(X.shape[0]):
    x_ = np.multiply.outer(np.ones(X.shape[0]), X[i])
```

```
    D[i] = np.sqrt(np.sum((X - x_)**2, 1));
hc_sing_pre = HClust(distance_threshold=0,
                     n_clusters=None,
                     metric='precomputed',
                     linkage='single')
hc_sing_pre.fit(D)
```

Out[42]:

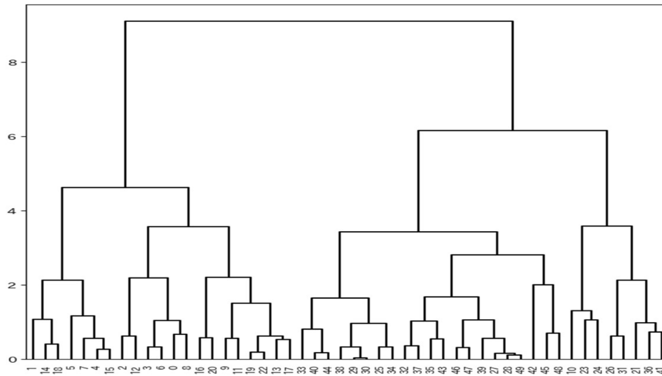
```
AgglomerativeClustering
AgglomerativeClustering(distance_threshold=0, linkage='single',
                        metric='precomputed', n_clusters=None)
```

`scipy.cluster.hierarchy` の `dendrogram()` を使用して樹形図をプロットしよう。ただし、`dendrogram()` はクラスタリングのリンク行列表現(*linkage-matrix representation*)を引数に持つ。これは `AgglomerativeClustering()` では提供されていないが、計算は可能である。この目的のために、`ISLP.cluster` パッケージの `compute_linkage()` 関数が提供されている。

これで樹形図をプロットが可能となるが、プロットの下部の数字は各観測値を特定している。`dendrogram()` 関数には、ツリーの異なる枝を色分けするデフォルトの方法があるが、これはツリーを特定の深さで事前に切り取ることを示している。このデフォルトの挙動を無効にするために、この閾値を無限大に設定する必要がある。これを多くの樹形図に適用したいので、これらの値を辞書 `cargs` に格納し、`**cargs` という記法でキーワード引数として渡すことにする。

In [43]:

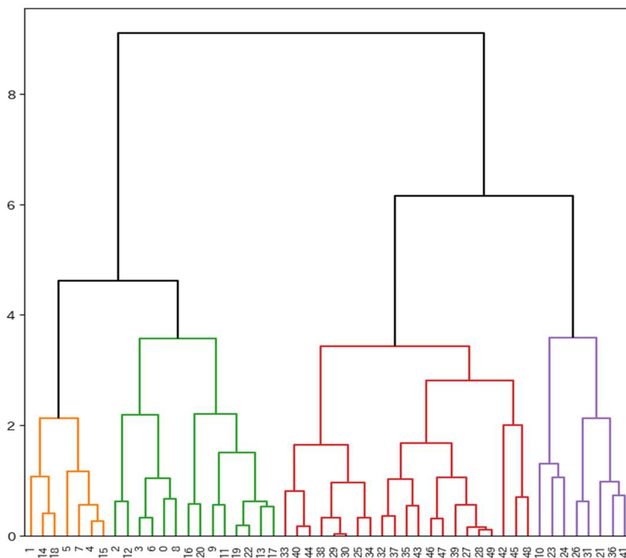
```
cargs = {'color_threshold':-np.inf,
         'above_threshold_color':'black'}
linkage_comp = compute_linkage(hc_comp)
fig, ax = plt.subplots(1, 1, figsize=(8, 8))
dendrogram(linkage_comp,
           ax=ax,
           **cargs);
```



ツリーの枝をカット閾値の上下で異なる色で分けたい場合がある。これを実現するために、`color_threshold` を変更しよう。ツリーを高さ 4 でカットし、4 以上で結合するリンクを黒で色付けする。

In [44]:

```
fig, ax = plt.subplots(1, 1, figsize=(8, 8))
dendrogram(linkage_comp,
           ax=ax,
           color_threshold=4,
           above_threshold_color='black');
```



樹形図の特定のカットに基づく各観測値のクラスタラベルを決定するために、`scipy.cluster.hierarchy`の`cut_tree()`関数を利用できる。

In [45]:

```
cut_tree(linkage_comp, n_clusters=4).T
```

Out[45]:

```
array([[0, 1, 0, 0, 1, 1, 0, 1, 0, 0, 2, 0, 0, 0, 1, 1, 0, 0, 1, 0, 0, 2,
        0, 2, 2, 3, 2, 3, 3, 3, 3, 2, 3, 3, 3, 3, 2, 3, 3, 3, 3, 2, 3, 3,
        3, 3, 3, 3, 3, 3]])
```

これも`HClust()`に引数`n_clusters`を渡すことで実現できる。ただし、各カットにはクラスタリングを再計算する必要がある。同様に、ツリーは`HClust()`に`distance_threshold`引数を、または`cut_tree()`に`height`引数を渡して閾値距離でカットすることができる。

In [46]:

```
cut_tree(linkage_comp, height=5)
```

Out[46]:

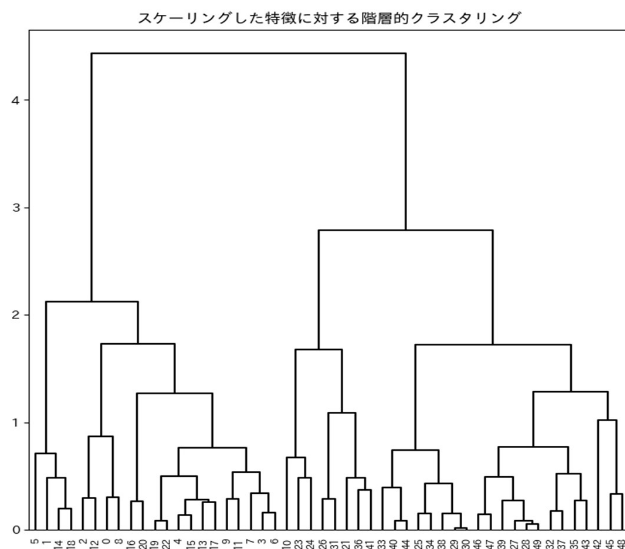
```
array([[0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [1],
       [0],
       [0],
```

```
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [0],
       [1],
       [0],
       [1],
       [1],
       [2],
       [1],
       [2],
       [2],
       [2],
       [2],
       [1],
       [2],
       [2],
       [2],
       [2],
       [1],
       [2],
       [2],
       [2],
       [2],
       [1],
       [2],
       [2],
       [2],
       [2],
       [2],
       [2],
       [2],
       [2]])
```

観測値の階層的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ために、PCA の例と同様に `StandardScaler()`を利用する。

In [47]:

```
scaler = StandardScaler()
X_scale = scaler.fit_transform(X)
hc_comp_scale = HClust(distance_threshold=0,
                       n_clusters=None,
                       linkage='complete').fit(X_scale)
linkage_comp_scale = compute_linkage(hc_comp_scale)
fig, ax = plt.subplots(1, 1, figsize=(8, 8))
dendrogram(linkage_comp_scale, ax=ax, **cargs)
ax.set_title("スケーリングした特徴に対する階層的クラスタリング");
```



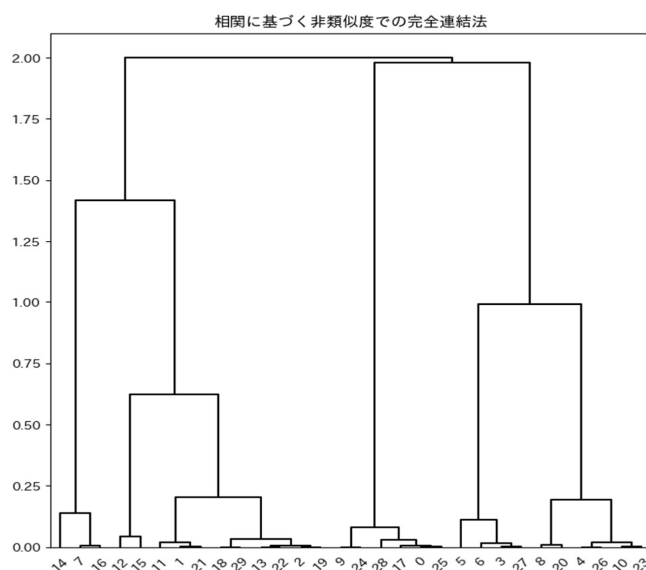
相関に基づく観測値間の距離をクラスタリングに利用することもできる。2 つの観測値間の相関は、それらの特徴量の値の類似性を測定している。（ここで各観測値が p 個の特徴を持ち、それぞれが単一の数値であると仮定する。2 つの観測値の類似性は、これらの p 個の数値ペアの相関を計算することによって測定する。）

n 個の観測値がある場合、n×n の相関行列は類似性（または親和性(affinity)）行列として使用でき、すなわち、相関行列を 1 から引いたものがクラスタリングに使用される非類似性行列になる。

相関を使用することは、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。なぜなら、2 つの特徴を持つ任意の観測値間の絶対相関は常に 1 だからである。したがって、3 次元のデータセットをクラスタリングできる。

In [48]:

```
X = np.random.standard_normal((30, 3))
corD = 1 - np.corrcoef(X)
hc_cor = HClust(linkage='complete',
                distance_threshold=0,
                n_clusters=None,
                metric='precomputed')
hc_cor.fit(corD)
linkage_cor = compute_linkage(hc_cor)
fig, ax = plt.subplots(1, 1, figsize=(8, 8))
dendrogram(linkage_cor, ax=ax, **cargs)
ax.set_title("相関に基づく非類似度での完全連結法");
```



NCI60 データの例

教師なし手法は、遺伝子データの解析でよく利用されている。特に、PCA と階層的クラスタリングは人気のあるツールである。ここではこれらの手法を `NCI60` がん細胞株マイクロアレイデータに適用して説明しておこう。このデータは 64 のがん細胞株における 6830 の遺伝子発現量から成っている。

In [49]:

```
NCI60 = load_data('NCI60')
nci_labs = NCI60['labels']
nci_data = NCI60['data']
```

各細胞株にはがんの種類がラベル付けされているが、PCA とクラスタリングを実行する際にはがんの種類を利用しない。これらの方法は教師なし手法であるためである。しかし、PCA とクラスタリングを実行した後、これらのがんの種類が教師なし手法の結果とどの程度一致するかを確認する。

このデータは 64 行、6830 列から成る。

In [50]:

```
nci_data.shape
```

Out[50]:

```
(64, 6830)
```

まず、細胞株のがんの種類を確認する。

In [51]:

```
nci_labs.value_counts()
```

Out[51]:

label	count
NSCLC	9
RENAL	9
MELANOMA	8
BREAST	7
COLON	7
LEUKEMIA	6
OVARIAN	6
CNS	5
PROSTATE	2
K562A-repro	1
K562B-repro	1
MCF7A-repro	1
MCF7D-repro	1
UNKNOWN	1

dtype: int64

NCI60 データに対する PCA

まず、変数（遺伝子）を標準偏差 1 にスケーリングした後、データに PCA を実行するが、ここでは遺伝子は同じ単位で測定されているため、スケーリングを行わない方が良いという主張もあり得る。

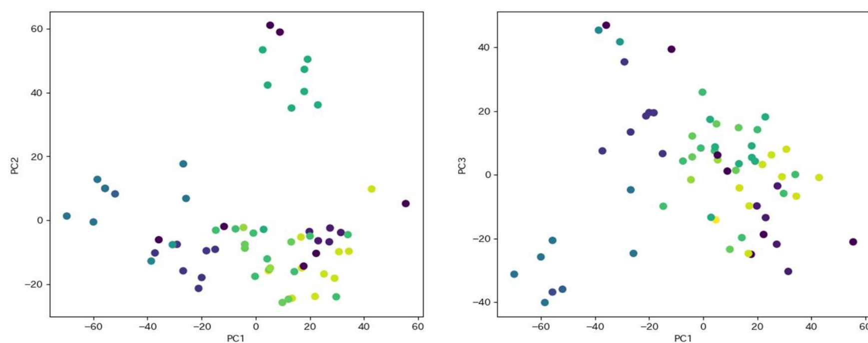
In [52]:

```
scaler = StandardScaler()
nci_scaled = scaler.fit_transform(nci_data)
nci_pca = PCA()
nci_scores = nci_pca.fit_transform(nci_scaled)
```

次に、最初のいくつかの主成分スコアベクトルをプロットして、データを可視化する。観測値（細胞株）は各がんの種類ごとに同じ色でプロットされ、がんの種類内の観測値がどれほど似ているかを確認できる。

In [53]:

```
cancer_types = list(np.unique(nci_labs))
nci_groups = np.array([cancer_types.index(lab)
                       for lab in nci_labs.values])
fig, axes = plt.subplots(1, 2, figsize=(15,6))
ax = axes[0]
ax.scatter(nci_scores[:,0],
           nci_scores[:,1],
           c=nci_groups,
           marker='o',
           s=50)
ax.set_xlabel('PC1'); ax.set_ylabel('PC2')
ax = axes[1]
ax.scatter(nci_scores[:,0],
           nci_scores[:,2],
           c=nci_groups,
           marker='o',
           s=50)
ax.set_xlabel('PC1'); ax.set_ylabel('PC3');
```

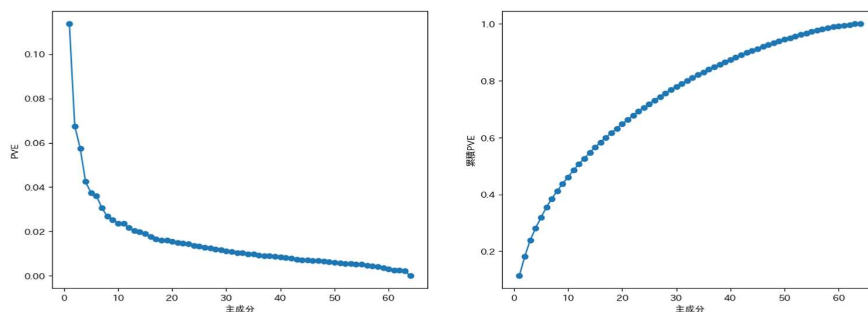


全体として、がんのある種類に対応する細胞株は、最初のいくつかの主成分スコアベクトルで似たような値を持つ傾向がある。これは、同じがんの種類の細胞株が遺伝子発現レベルにおいて非常に似ていることを示している。

また、主成分によって説明される分散の割合と、累積的な分散の割合をプロットすることができる。これは、`USArrests`データに対して行ったプロットと類似の作業である。

In [54]:

```
fig, axes = plt.subplots(1, 2, figsize=(15,6))
ax = axes[0]
ticks = np.arange(nci_pca.n_components_)+1
ax.plot(ticks,
        nci_pca.explained_variance_ratio_,
        marker='o')
ax.set_xlabel('主成分');
ax.set_ylabel('PVE')
ax = axes[1]
ax.plot(ticks,
        nci_pca.explained_variance_ratio_.cumsum(),
        marker='o');
ax.set_xlabel('主成分')
ax.set_ylabel('累積 PVE');
```



ここで、最初の 7 つの主成分がデータの約 40%の分散を説明していることが分かるが、分散全体の大きな割合ではない。しかし、スクリープロット(scree plot)を見ると、最初の 7 つの主成分はかなりの分散を説明している一方で、さらに多くの主成分によって説明される分散が急激に減少していることが分かる。つまり、7 番目あたりの主成分の後に肘(*elbow*)が現れる。これは、7 つより多い主成分を調

べてもあまり利益がない可能性があることを示唆している（ただし 7 つの主成分を調べるだけでも難しいと言えるかもしれない）。

NCI60 データの観測値のクラスタリング

次に、`NCI60` データの細胞株に対して階層的クラスタリングを実行する。使用する連結法は完全連結法(complete linkage)、単一連結法(single linkage)、および平均連結法(mean linkage)である。ここでの目的はまず観測値が異なるがんの種類にクラスタリングされるかどうかを確認することである。非類似度にはユークリッド距離を利用して、まず 3 つの樹形図を作成する短い関数を書いておこう。

In [55]:

```
def plot_nci(linkage, ax, cut=-np.inf):
    cargs = {'above_threshold_color':'black',
             'color_threshold':cut}
    hc = HClust(n_clusters=None,
                distance_threshold=0,
                linkage=linkage.lower()).fit(nci_scaled)
    linkage_ = compute_linkage(hc)
    dendrogram(linkage_,
               ax=ax,
               labels=np.asarray(nci_labs),
               leaf_font_size=10,
               **cargs)
    ax.set_title('%s Linkage' % linkage)
    return hc
```

結果をプロットしてみよう。

In [56]:

```
fig, axes = plt.subplots(3, 1, figsize=(15,30))
ax = axes[0]; hc_comp = plot_nci('Complete', ax)
ax = axes[1]; hc_avg = plot_nci('Average', ax)
ax = axes[2]; hc_sing = plot_nci('Single', ax)
```

連結法の選択が確かに得られる結果に影響を与えることが分る。通常、単一連結法はトレーリング(*trailing*)クラスタ、すなわち個々の観測値が大きなクラスタに1つずつ追加されやすくなる傾向がある。一方、完全連結法と平均連結法は、よりバランスの取れた魅力的なクラスタを作る傾向がある。このため、完全連結法と

平均連結法は一般的に単一連結法よりも好まれている。明らかに、同一のがんの種類内の細胞株はクラスタリングされる傾向があるが、完全ではない。これから行う分析では、完全連結法による階層的クラスタリングを利用する。樹形図を特定の高さで切断し、例えば 4 つのクラスタを得ることができる：

In [57]:

```
linkage_comp = compute_linkage(hc_comp)
comp_cut = cut_tree(linkage_comp, n_clusters=4).reshape(-1)
pd.crosstab(nci_labs['label'],
            pd.Series(comp_cut.reshape(-1), name='Complete'))
```

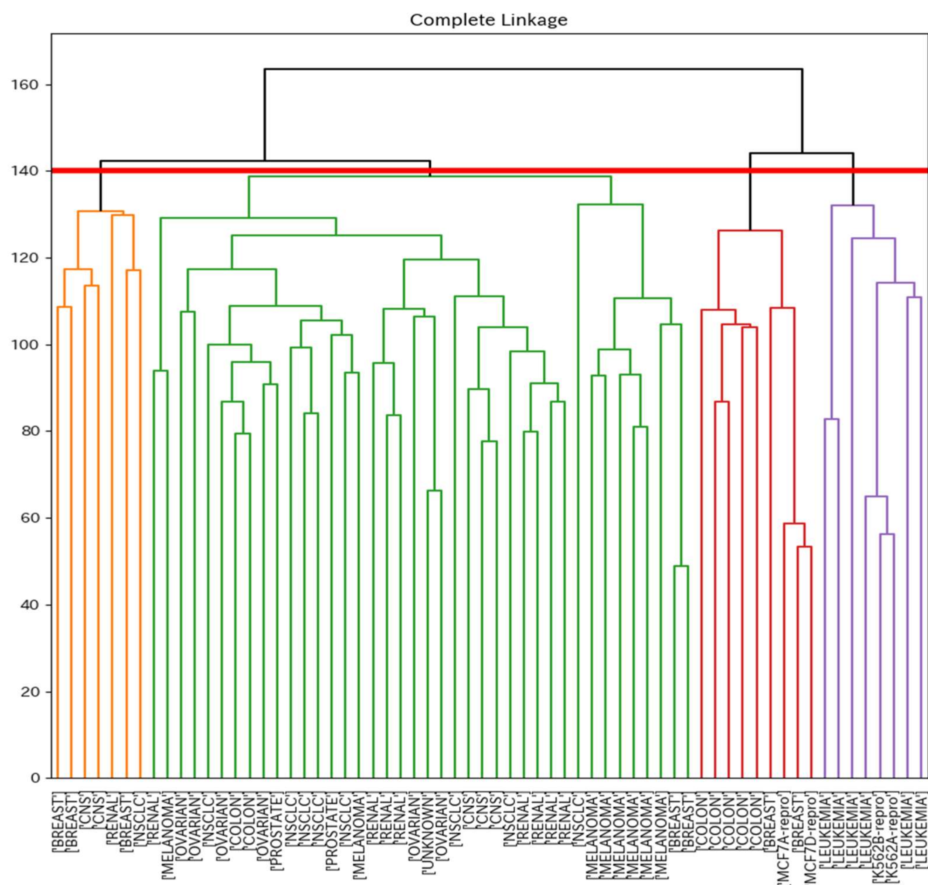
Out[57]:

Complete	0	1	2	3
label				
BREAST	2	3	0	2
CNS	3	2	0	0
COLON	2	0	0	5
K562A-repro	0	0	1	0
K562B-repro	0	0	1	0
LEUKEMIA	0	0	6	0
MCF7A-repro	0	0	0	1
MCF7D-repro	0	0	0	1
MELANOMA	8	0	0	0
NSCLC	8	1	0	0
OVARIAN	6	0	0	0
PROSTATE	2	0	0	0
RENAL	8	1	0	0
UNKNOWN	1	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 1 つのクラスタに分類され、乳がん細胞株は 3 つの異なるクラスタに分かれている。この 4 つのクラスタを得るために樹形図を切断したプロットを作成できる：

In [58]:

```
fig, ax = plt.subplots(figsize=(10,10))
plot_nci('Complete', ax, cut=140)
ax.axhline(140, c='r', linewidth=4);
```



`axhline()` 関数は、既存の軸上に水平線を描画する。引数 `140` は、樹形図上で高さ 140 に水平線を描画する。これは 4 つの異なるクラスタを得る高さである。結果として得られるクラスタが `comp_cut` で得られたものと同じであることは容易に確認できる。

前のセクション 12.4.2 節では、同じ数のクラスタを得るために樹形図を切断して行った階層的クラスタリングと K-平均クラスタリングは、非常に異なる結果をもたらす可能性があると述べた。それでは、この `NCI60` の階層的クラスタリングの結果は、K=4 で K-平均クラスタリングを行った場合と比較してどうだろうか？

In [59]:

```
nci_kmeans = KMeans(n_clusters=4,
                    random_state=0,
                    n_init=20).fit(nci_scaled)
```

```
pd.crosstab(pd.Series(comp_cut, name='HClust'),
            pd.Series(nci_kmeans.labels_, name='K-means'))
```

Out[59]:

K-means	0	1	2	3
HClust				
0	1	20	10	9
1	0	7	0	0
2	8	0	0	0
3	0	0	9	0

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、いくつか異なっていることが分かる。なお 2 つのクラスタリングにおけるラベルは任意であることに注意しておこう。つまり、クラスタの識別子を入れ替えても、クラスタリング自体は変わらない。ここでは、K-平均クラスタリングのクラスタ 3 は、階層的クラスタリングのクラスタ 2 と一致することが分かる。しかし、他のクラスタは異なっている。例えば、K-平均クラスタリングのクラスタ 0 は、階層的クラスタリングでクラスタ 0 に割り当てられた観測値の一部と、クラスタ 1 に割り当てられた観測値をすべて含んでいる。

データ行列全体に対して階層的クラスタリングを行う代わりに、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できる。そのときは最初のいくつかの成分を、データのノイズの少ないバージョンとして扱うことになる。

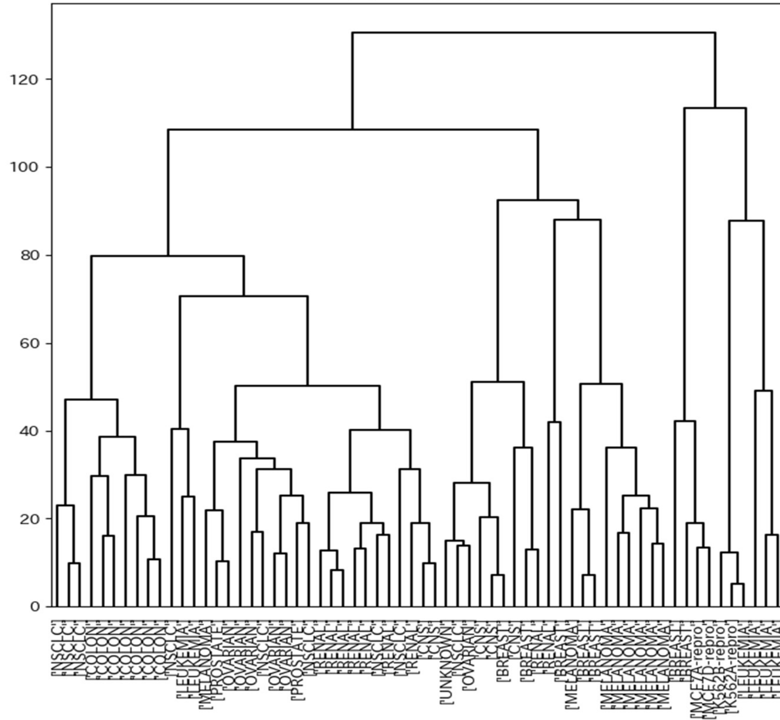
In [60]:

```
hc_pca = HClust(n_clusters=None,
                distance_threshold=0,
                linkage='complete'
                ).fit(nci_scores[:,:5])
linkage_pca = compute_linkage(hc_pca)
fig, ax = plt.subplots(figsize=(8,8))
dendrogram(linkage_pca,
           labels=np.asarray(nci_labs),
```

```
            leaf_font_size=10,
            ax=ax,
            **cargs)
ax.set_title("最初の 5 つのスコアベクトルを用いた階層的クラスタリング")
pca_labels = pd.Series(cut_tree(linkage_pca,
                                n_clusters=4).reshape(-1),
                       name='Complete-PCA')
pd.crosstab(nci_labs['label'], pca_labels)
```

Out[60]:

Complete-PCA	0	1	2	3
label				
BREAST	0	5	0	2
CNS	2	3	0	0
COLON	7	0	0	0
K562A-repro	0	0	1	0
K562B-repro	0	0	1	0
LEUKEMIA	2	0	4	0
MCF7A-repro	0	0	0	1
MCF7D-repro	0	0	0	1
MELANOMA	1	7	0	0
NSCLC	8	1	0	0
OVARIAN	5	1	0	0
PROSTATE	2	0	0	0
RENAL	7	2	0	0
UNKNOWN	0	1	0	0


最初の5つのスコアベクトルを用いた階層的クラスタリング
0
20
40
60
80
100
120
['NSCLC']
['NSCLC']
['NSCLC']
['COLON']
['COLON']
['COLON']
['COLON']
['COLON']
['COLON']
['COLON']
['NSCLC']
['LEUKEMIA']
['LEUKEMIA']
['MELANOMA']
['PROSTATE']
['OVARIAN']
['OVARIAN']
['OVARIAN']
['NSCLC']
['OVARIAN']
['OVARIAN']
['PROSTATE']
['NSCLC']
['RENAL']
['RENAL']
['RENAL']
['RENAL']
['RENAL']
['NSCLC']
['RENAL']
['NSCLC']
['RENAL']
['CNS']
['CNS']
['UNKNOWN']
['NSCLC']
['OVARIAN']
['CNS']
['CNS']
['BREAST']
['CNS']
['BREAST']
['RENAL']
['RENAL']
['BREAST']
['MELANOMA']
['BREAST']
['BREAST']
['MELANOMA']
['MELANOMA']
['MELANOMA']
['MELANOMA']
['MELANOMA']
['MELANOMA']
['BREAST']
['BREAST']
['MCF7A-repro']
['MCF7D-repro']
['K562B-repro']
['K562A-repro']
['LEUKEMIA']
['LEUKEMIA']
['LEUKEMIA']
['LEUKEMIA']

ISLP 第 13 章 多重検定

これまでのラボでも使ったライブラリーをインポートする。

訳注：Google Colab でこのコードを実行した。Colab が不安定な場合(2025 年 3 月 20 日頃)には「ランタイムからセッションを再起動する」と問題ないことがあるが、Jupyter ではプログラムは正常に動作することを確認した。(Numpy などパッケージ更新のタイミングによるのではと思われる。)また、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。

In [1]:

```
# 訳者追加

# %%capture は出力を非表示にするために、出力が長くなる箇所で用いている

# もちろん%%capture を消したりコメントアウトしたりすれば、出力が表示される

%%capture
!pip install japanize-matplotlib
```

In [2]:

```
# 訳者追加

%%capture

!pip install ISLP # ISLP は初めからインストールされてはいないのでインストールする
```

In [3]:

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
import japanize_matplotlib # グラフで日本語を使うために訳者が追加した

import statsmodels.api as sm
from ISLP import load_data
```

さらに、このラボで必要なライブラリーをインポートする。

In [4]:

```
from scipy.stats import \
    (ttest_1samp,
     ttest_rel,
     ttest_ind,
     t as t_dbn)
from statsmodels.stats.multicomp import \
     pairwise_tukeyhsd
from statsmodels.stats.multitest import \
     multipletests as mult_test
```

仮説検定の復習

まず、いくつかの 1 標本 t-検定を実行しよう。

100 個の変数を作成し、それぞれが 10 個の観測値を持つとする。最初の 50 個の変数は平均 0.5、分散 1 で、残りの 50 個は平均 0 、分散 1 としよう。

In [5]:

```
rng = np.random.default_rng(12)
X = rng.standard_normal((10, 100))
true_mean = np.array([0.5]*50 + [0]*50)
X += true_mean[None,:]
```

最初に、`scipy.stats` モジュールの `ttest_1samp()`を用いて$H_0: \mu_1 = 0$、つまり、最初の変数の平均がゼロであるという帰無仮説を検定する。

In [6]:

```
result = ttest_1samp(X[:,0], 0)
result.pvalue
```

Out[6]:

```
0.9307442156164141
```

p-値は 0.931 となり、$\alpha = 0.05$ の水準で帰無仮説を棄却するには十分に低くはない。この場合、$\mu_1 = 0.5$であるので帰無仮説は誤っている。したがって、帰無仮説が誤っているのに棄却しなかったためタイプ II エラーを犯している。

次に、$H_{0,j}: \mu_j = 0$を$j = 1, \dots, 100$について検定する。100 個の p-値を計算し、次に j 番目の p-値が 0.05 以下の場合に帰無仮説 $H_{0,j}$ を棄却、それより大きい場合は棄却しないという結果を記録するベクトルを作る。

In [7]:

```
p_values = np.empty(100)
for i in range(100):
   p_values[i] = ttest_1samp(X[:,i], 0).pvalue
decision = pd.cut(p_values,
                  [0, 0.05, 1],
                  labels=['H0 を棄却する',
                          'H0 を棄却しない'])
truth = pd.Categorical(true_mean == 0,
                       categories=[True, False],
                       ordered=True)
```

これはシミュレーションによるデータセットであるため、表 13.2 にような 2×2 表を作成できる。

In [8]:

```
pd.crosstab(decision,
            truth,
     rownames=['決定'], #rownames=['Decision'],
     colnames=['H0'])
```

Out[8]:

H0 決定	True	False
H0を棄却する	5	15
H0を棄却しない	45	35

したがって、$\alpha = 0.05$ の水準では 50 個の偽の帰無仮説のうち 15 個を棄却し、5 個の真の帰無仮説を誤って棄却している。13.3 節の記法を使用すると、V=5、S=15、U=45、W=35 となる。$\alpha = 0.05$ を設定しているため、真の帰無仮説の約 5%を棄却することが期待される。これは上の 2×2 表を見ると、50 個の真の帰無仮説のうち V=5 個を棄却したことを示している。

上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 だった。これはかなり弱い信号の場合に対応し、タイプ II エラーが多く発生した。代わりに、より強い信号を持つデータをシミュレーションで発生させ、偽の帰無仮説に対して平均と標準偏差の比が 1 になるようにしてみよう。この操作によりタイプ II エラーが 10 回に減少する。

In [9]:

```
true_mean = np.array([1]*50 + [0]*50)
X = rng.standard_normal((10, 100))
X += true_mean[None,:]
for i in range(100):
   p_values[i] = ttest_1samp(X[:,i], 0).pvalue
decision = pd.cut(p_values,
                  [0, 0.05, 1],
                  labels=['H0 を棄却する',
```

```
                             'H0 を棄却しない'])
truth = pd.Categorical(true_mean == 0,
                       categories=[True, False],
                       ordered=True)
pd.crosstab(decision,
            truth,
            rownames=['決定'],
            colnames=['H0'])
```

Out[9]:

H0	True	False
決定		
H0を棄却する	2	40
H0を棄却しない	48	10

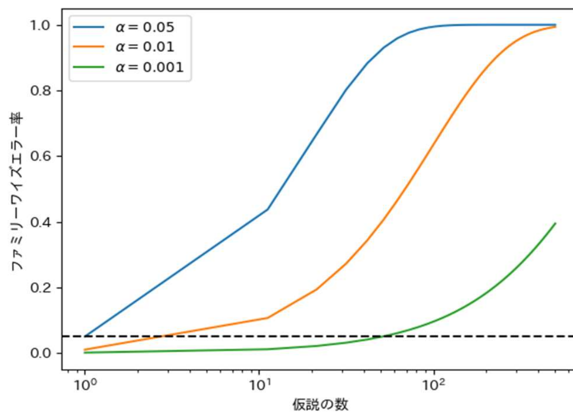
ファミリーワイズエラー率 (Family-Wise Error Rate, FWER)

(13.5)式を思い出そう。もし m 個の独立した仮説検定において帰無仮説がすべて正しい場合、FWER は$1-(1-\alpha)^m$に等しくなる。この式を使って、$\mathrm{m}=1,\ldots,500$ およびα =0.05, 0.01, 0.001 に対する FWER が計算してみる。これらのαの値に対する FWER をプロットして、図 13.2 再現してみる。

In [10]:

```
m = np.linspace(1, 501)
fig, ax = plt.subplots()
[ax.plot(m,
         1 - (1 - alpha)**m,
         label=r'$\alpha=%s$' % str(alpha))
         for alpha in [0.05, 0.01, 0.001]]
```

```
ax.set_xscale('log')
ax.set_xlabel('仮説の数')
ax.set_ylabel('ファミリーワイズエラー率')
ax.legend()
ax.axhline(0.05, c='k', ls='--');
```



前述のように、m が 50 など中程度の値であっても、αが非常に低い値（例えば 0.001）に設定されない限り、FWER は 0.05 を超える。もちろん、αをこのように低い値に設定する問題として、タイプ II エラーが多く発生する可能性がある。つまり、検出力が非常に低くなる。

次に、`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ゼロであるという帰無仮説 $H_{0,j}: \mu_j = 0$ を検定しよう。

In [11]:

```
Fund = load_data('Fund')
fund_mini = Fund.iloc[:,:5]
fund_mini_pvals = np.empty(5)
for i in range(5):
    fund_mini_pvals[i] = ttest_1samp(fund_mini.iloc[:,i], 0).pvalue
fund_mini_pvals
```

Out[11]:

```
array([0.00620236, 0.91827115, 0.01160098, 0.6005396 , 0.75578151])
```

マネージャー1 と 3 の p-値は低く、他の 3 人のマネージャーの p-値は高い。しかし、このまま $H_{0,1}$ と $H_{0,3}$ を単に棄却することはできない。なぜなら、そうすると多重検定を行ったことを考慮していないということになるからである。そで代わりに、ボンフェローニ法とホルム法を使用して FWER を制御しよう。

これを行うには `statsmodels` モジュールの `multipletests()`関数（省略して `mult_test()`）を利用する。p-値が与えられた場合、ホルム法やボンフェローニ法などの方法は、調整された p-値を出力する。これは（厳密には p-値ではないが）複数の検定に対して補正された、p-値のようなものと考えられる。与えられた仮説に対して調整された p-値がα以下であれば、その仮説を棄却することができ、FWER がαを超えないように保つことができる。言い換えれば、このような方法により `multipletests()`関数によって得られた調整された p-値を単に希望する FWER と比較することで、各仮説を棄却するかどうかを決定することが可能となる。後で、同じ関数を使用して FDR を制御する方法も見ていくことにする。

`mult_test()`関数は p-値と `method` 引数、オプションの `alpha` 引数を持つ。そして、決定（下の `reject`）と調整された p-値（`bonf`）を返す。

In [12]:

```
reject, bonf = mult_test(fund_mini_pvals, method = "bonferroni")[:2]
reject
```

Out[12]:

```
array([ True, False, False, False, False])
```

調整した p-値 `bonf` とは、単に `fund_mini_pvalues` を 5 倍して 1 以下に切り捨てた値である。

In [13]:

```
bonf, np.minimum(fund_mini_pvals * 5, 1)
```

Out[13]:

```
(array([0.03101178, 1.        , 0.05800491, 1.        , 1.        ]),
 array([0.03101178, 1.        , 0.05800491, 1.        , 1.        ]))
```

このように、ボンフェローニ法を使用することで、FWER を 0.05 に制御しながら、マネージャー1 のみを帰無仮説を棄却することになる。

一方、ホルム法を使用すると、調整された p-値は、マネージャー1 と 3 の帰無仮説を FWER が 0.05 で棄却することを示している。

In [14]:

```
mult_test(fund_mini_pvals, method = "holm", alpha=0.05)[:2]
```

Out[14]:

```
(array([ True, False,  True, False, False]),
 array([0.03101178, 1.        , 0.04640393, 1.        , 1.        ]))
```

前述のように、マネージャー1 は特に良い成績を収めているが、マネージャー2 の成績は悪いようである。

In [15]:

```
fund_mini.mean()
```

Out[15]:

	0
Manager1	3.0
Manager2	-0.1
Manager3	2.8
Manager4	0.5
Manager5	0.3

dtype: float64

これらの 2 人のマネージャー間に意味のあるパフォーマンスの違いがあるのだろうか。このことを確認するために、`scipy.stats` の `ttest_rel()`関数を使用して、対応のある t-検定を行うことにする。

In [16]:

```
ttest_rel(fund_mini['Manager1'],
          fund_mini['Manager2']).pvalue
```

Out[16]:

```
0.038391072368079586
```

検定の結果、p-値は 0.038 となり、統計的に有意な差が示唆される。しかし、この検定を実行する前にデータを調べると、マネージャー1 とマネージャー2 がそれぞれ最高および最低の平均パフォーマンスを持っていることに気が付かされる。ある意味では、ここでは 1 個だけの仮説検定でなく、実質的に ${}_5C_2 = 5(5\text{-}1)/2\text{=}10$ 個の仮説検定を行ったことになる（これは、13.3.2 節で説明した通りである）。ここで `statsmodels.stats.multicomp` の `pairwise_tukeyhsd()`関数を使用して、多重検定に対する補正を行うために Tukey 法を適用してみよう。この関数は、フィットした *ANOVA* 回帰モデルを入力として受け取る。これは本質的にすべての予測変数が質的である線形回帰を行うことに相当する。この場合、反応変数は各マネージャーによる月次超過リターンで、予測変数は各リターンに対応するマネージャーである。

In [17]:

```
returns = np.hstack([fund_mini.iloc[:,i] for i in range(5)])
managers = np.hstack([[i+1]*50 for i in range(5)])
tukey = pairwise_tukeyhsd(returns, managers)
print(tukey.summary())
```

Out [17]:

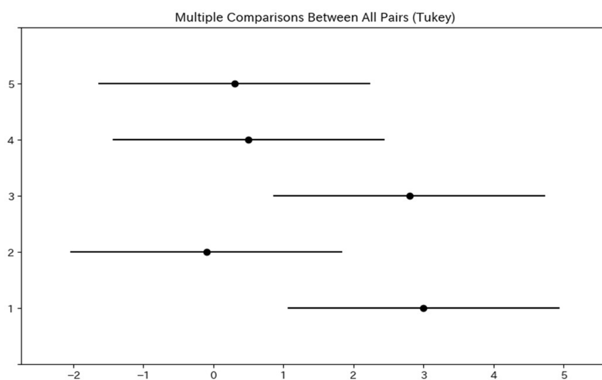
```
Multiple Comparison of Means - Tukey HSD, FWER=0.05
===================================================
group1 group2 meandiff p-adj   lower  upper  reject
---------------------------------------------------
     1      2     -3.1 0.1862 -6.9865 0.7865  False
     1      3     -0.2 0.9999 -4.0865 3.6865  False
```

```
     1      4     -2.5 0.3948 -6.3865 1.3865  False
     1      5     -2.7 0.3152 -6.5865 1.1865  False
     2      3      2.9 0.2453 -0.9865 6.7865  False
     2      4      0.6 0.9932 -3.2865 4.4865  False
     2      5      0.4 0.9986 -3.4865 4.2865  False
     3      4     -2.3  0.482 -6.1865 1.5865  False
     3      5     -2.5 0.3948 -6.3865 1.3865  False
     4      5     -0.2 0.9999 -4.0865 3.6865  False
---------------------------------------------------
```

`pairwise_tukeyhsd()`関数は、各マネージャーのペア間の差の信頼区間（`lower`および`upper`）と p-値を与える。これらの量はすべて多重検定に関して補正されている。マネージャー1 とマネージャー2 の差の p-値が 0.038 から 0.186 に増加したことに注目しておこう。つまり、マネージャー間のパフォーマンスの違いについてはもはや明確な証拠が得られなくなっていることになる。次に、`tukey`の`plot_simultaneous()`メソッドを使用して、対比較の信頼区間をプロットしよう。重ならない区間があれば、名目水準 0.05 で有意な差があることを示している。この場合、上の表で報告されているように、有意な差は存在しない。

In [18]:

```
fig, ax = plt.subplots(figsize=(8,8))
tukey.plot_simultaneous(ax=ax);
```



誤発見率 (False Discovery Rate, FDR)

次に、`Fund` データセットの 2,000 人のファンドマネージャー全員について仮説検定を行う。そこで$H_{0,j}: \mu_j = 0$の 1 標本 t-検定を行おう。これは、j 番目のファンドマネージャーの平均リターンがゼロであることを示す仮説である。

In [19]:

```
fund_pvalues = np.empty(2000)
for i, manager in enumerate(Fund.columns):
    fund_pvalues[i] = ttest_1samp(Fund[manager], 0).pvalue
```

ここでは多くのマネージャーがいるため、FWER を制御しようとするのは現実的ではない。そこで代わりに、FDR（誤発見率）を制御することにする。FDR は、棄却した帰無仮説のうち、実際に偽陽性であるものの期待割合である。これは `multipletests()`関数（省略して `mult_test()`）を使用して、Benjamini--Hochberg 法を実行できる。

In [20]:

```
fund_qvalues = mult_test(fund_pvalues, method = "fdr_bh")[1]
fund_qvalues[:10]
```

Out[20]:

```
array([0.08988921, 0.991491  , 0.12211561, 0.92342997, 0.95603587,
       0.07513802, 0.0767015 , 0.07513802, 0.07513802, 0.07513802])
```

Benjamini--Hochberg 法によって出力される q-値は、特定の帰無仮説を棄却するための最小の FDR 閾値として解釈できる。例えば、q-値が 0.1 の場合、それは FDR を 10%以上で制御する場合には帰無仮説を棄却できることを意味し、FDR を 10%未満に制御する場合では帰無仮説を棄却できないことを意味する。

FDR を 10%に制御した場合、どれだけにファンドマネージャーについて

$H_{0,j}: \mu_j = 0$を棄却できるだろうか？

In [21]:

```
(fund_qvalues <= 0.1).sum()
```

Out[21]:

```
146
```

2,000 人のファンドマネージャーのうち 146 人が q-値が 0.1 未満であることが分った。したがって、FDR が 10%のとき、146 人のファンドマネージャーが市場を上回ったと結論できる。なおこれらのファンドマネージャーのうち、約 15 人（146 人の 10%）は誤発見である可能性がある。

一方、もしボンフェローニ法を使用して FWER を$\alpha = 0.1$ のレベルで制御していた場合、どの帰無仮説も棄却できなかったことになる！

In [22]:

```
(fund_pvalues <= 0.1 / 2000).sum()
```

Out[22]:

```
0
```

図 13.6 には、`Fund` データセットの p-値を$p_{(1)} \leq p_{(2)} \leq \cdots \leq p_{(m)}$と並べたものが表示されており、Benjamini--Hochberg 法による棄却の閾値も示されている。Benjamini--Hochberg 法は、$p_{(j)} < qj/m$ を満たす最大の p-値を特定し、その p-値以下のすべての仮説を棄却する。以下のコードでは、Benjamini--Hochberg 法を実際に実装して、どのように機能するかを示す。まず、p-値を順に並べる。次に、$p_{(j)} < qj/m$ を満たす p-値を特定する（`sorted_set_`）。最後に、`selected_`は`sorted_[sorted_set_]`内の最大 p-値以下の p-値を示すブール配列を示す。したがって、`selected_`は Benjamini--Hochberg 法によって棄却された p-値を示す。

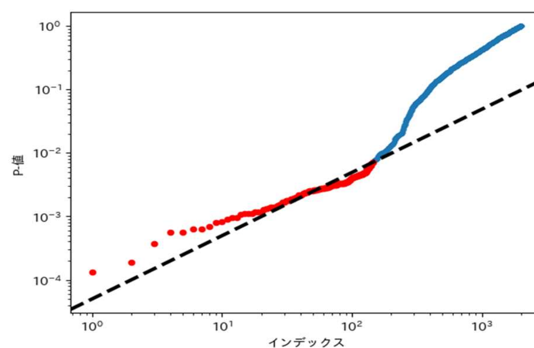
In [23]:

```
sorted_ = np.sort(fund_pvalues)
m = fund_pvalues.shape[0]
q = 0.1
sorted_set_ = np.where(sorted_ < q * np.linspace(1, m, m) / m)[0]
if sorted_set_.shape[0] > 0:
    selected_ = fund_pvalues < sorted_[sorted_set_].max()
    sorted_set_ = np.arange(sorted_set_.max())
else:
    selected_ = []
    sorted_set_ = []
```

ここで図 13.6 の中央パネルを再現しよう。

In [24]:

```
fig, ax = plt.subplots()
ax.scatter(np.arange(0, sorted_.shape[0]) + 1,
           sorted_, s=10)
ax.set_yscale('log')
ax.set_xscale('log')
ax.set_ylabel('P-値')
ax.set_xlabel('インデックス')
ax.scatter(sorted_set_+1, sorted_[sorted_set_], c='r', s=20)
ax.axline((0, 0), (1,q/m), c='k', ls='--', linewidth=3);
```



リサンプリングアプローチ

ここでは 13.5 節で検討した `Khan` データセットを使用して、仮説検定に対するリサンプリングアプローチを実行しよう。まず、トレーニングデータとテストデータを統合し、2,308 の遺伝子に関して 83 人の患者に関する観測値を得る。

In [25]:

```
Khan = load_data('Khan')
D = pd.concat([Khan['xtrain'], Khan['xtest']])
D['Y'] = pd.concat([Khan['ytrain'], Khan['ytest']])
D['Y'].value_counts()
```

Out[25]:

Y	count
2	29
4	25
3	18
1	11

dtype: int64

ここでは 4 つのがんのクラスがある。各遺伝子について、2 番目のクラス（横紋筋肉腫(rhabdomyosarcoma)）と 4 番目のクラス（バーキットリンパ腫(Burkitt's lymphoma)）の平均発現を比較しよう。`scipy.stats` の `ttest_ind()`を使用して 11 番目の遺伝子に対して標準的な二標本 t-検定を実行すると、検定統計量は-2.09、対応する p-値は 0.0412 であり、2 つのがんタイプ間で発現レベルに差があるという弱い証拠を示唆している。

In [26]:

```
D2 = D[lambda df:df['Y'] == 2]
D4 = D[lambda df:df['Y'] == 4]
gene_11 = 'G0011'
```

```
observedT, pvalue = ttest_ind(D2[gene_11],
                              D4[gene_11],
                              equal_var=True)
observedT, pvalue
```

Out[26]:

```
(-2.0936330736768185, 0.04118643782678394)
```

しかし、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、検定統計量が 29+25-2=52 の自由度を持つ 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の代わりに、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できる。帰無仮説が 2 つのグループ間に差がない場合、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 10,000 回繰り返すことで、検定統計量の帰無分布を近似できる。さらに観察された検定統計量がリサンプリングによって得られたテスト統計量を超える割合を計算する。

In [27]:

```
B = 10000
Tnull = np.empty(B)
D_ = np.hstack([D2[gene_11], D4[gene_11]])
n_ = D2[gene_11].shape[0]
D_null = D_.copy()
for b in range(B):
    rng.shuffle(D_null)
    ttest_ = ttest_ind(D_null[:n_],
                       D_null[n_:],
                       equal_var=True)
    Tnull[b] = ttest_.statistic
(np.abs(Tnull) > np.abs(observedT)).mean()
```

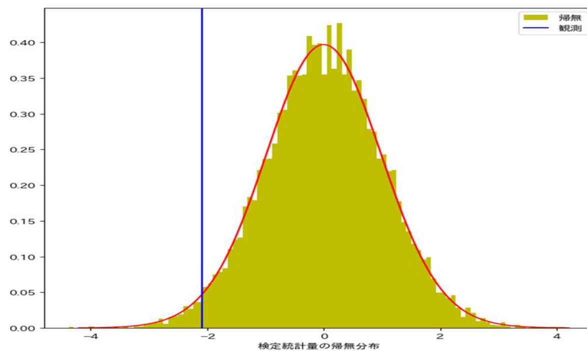
Out[27]:

```
0.0398
```

この割合、0.0398 が私たちのリサンプリングに基づいた p-値である。これは理論的な帰無分布を使用して得られた p-値 0.0412 とほぼ同じである。リサンプリングに基づいた検定統計量のヒストグラムをプロットすることで、図 13.7 が再現できる。

In [28]:

```
fig, ax = plt.subplots(figsize=(8,8))
ax.hist(Tnull,
        bins=100,
        density=True,
        facecolor='y',
        label='帰無')
xval = np.linspace(-4.2, 4.2, 1001)
ax.plot(xval,
        t_dbn.pdf(xval, D_.shape[0]-2),
        c='r')
ax.axvline(observedT,
           c='b',
           label='観測')
ax.legend()
ax.set_xlabel("検定統計量の帰無分布");
```



リサンプリングに基づいた帰無分布は、理論的な帰無分布とほぼ一致し、赤色で表示されている。

最後に、アルゴリズム(Algorithm) 13.4 で概説されたプラグインリサンプリング FDR アプローチを実行しよう。`Khan` データセットの全 2,308 遺伝子について FDR を計算するのには、コンピュータの速度によっては時間がかかる場合がある。したがって、このアプローチをランダムに選んだ 100 遺伝子で説明する。各遺伝子について、まず観測値に対する検定統計量を計算し、その後 10,000 回のリサンプリングによる検定統計量を生成するが、実行には数分かかる場合がある。急いでいる場合は、`B` を小さな値（例えば `B=500`）に設定することもできる。

In [29]:

```
m, B = 100, 10000
idx = rng.choice(Khan['xtest'].columns, m, replace=False)
T_vals = np.empty(m)
Tnull_vals = np.empty((m, B))

for j in range(m):
    col = idx[j]
    T_vals[j] = ttest_ind(D2[col],
                          D4[col],
                          equal_var=True).statistic
    D_ = np.hstack([D2[col], D4[col]])
    D_null = D_.copy()
    for b in range(B):
        rng.shuffle(D_null)
        ttest_ = ttest_ind(D_null[:n_],
```

```
                             D_null[n_:],
                             equal_var=True)
        Tnull_vals[j,b] = ttest_.statistic
```

次に、アルゴリズム(Algorithm) 13.4 での閾値 c の範囲について、棄却された帰無仮説の数 R、推定される偽陽性の数 $\widehat{V}$、および推定される FDR を計算する。閾値は 100 遺伝子からの検定統計量の絶対値を使用して選ぶ。

In [30]:

```
cutoffs = np.sort(np.abs(T_vals))
FDRs, Rs, Vs = np.empty((3, m))
for j in range(m):
   R = np.sum(np.abs(T_vals) >= cutoffs[j])
   V = np.sum(np.abs(Tnull_vals) >= cutoffs[j]) / B
   Rs[j] = R
   Vs[j] = V
   FDRs[j] = V / R
```

ここでは任意の FDR に対して、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 に制御すると、100 の帰無仮説のうち 15 個が棄却される。平均して、これらの遺伝子の約 1～2 個（すなわち、15 の 10%）が誤発見であると予想される。FDR を 0.2 にすると、28 の遺伝子について帰無仮説を棄却でき、そのうち約 6 個が誤発見であると予想される。

変数 `idx` は、100 個のランダムに選ばれた遺伝子のどれが含まれていたかを格納している。ここで、推定 FDR が 0.1 未満の遺伝子を見てみよう。

In [31]:

```
sorted(idx[np.abs(T_vals) >= cutoffs[FDRs < 0.1].min()])
```

Out[31]:

```
['G0097',
 'G0129',
```

```
 'G0182',
 'G0714',
 'G0812',
 'G0941',
 'G0982',
 'G1020',
 'G1022',
 'G1090',
 'G1320',
 'G1634',
 'G1697',
 'G1853',
 'G1854',
 'G1994',
 'G2017',
 'G2115',
 'G2193']
```

FDR 閾値を 0.2 にすると、さらに多くの遺伝子が選ばれ、誤発見である割合が高くなる。

In [32]:

```
sorted(idx[np.abs(T_vals) >= cutoffs[FDRs < 0.2].min()])
```

Out[32]:

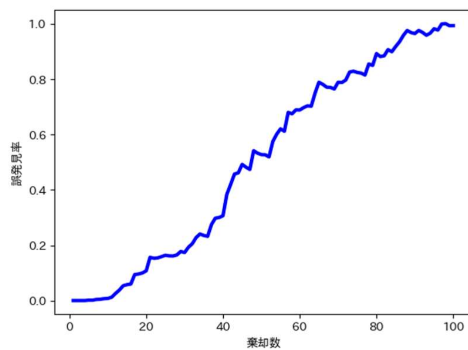
```
['G0097',
 'G0129',
 'G0158',
 'G0182',
 'G0242',
 'G0552',
 'G0679',
 'G0714',
 'G0751',
 'G0812',
 'G0908',
 'G0941',
 'G0982',
```

```
 'G1020',
 'G1022',
 'G1090',
 'G1240',
 'G1244',
 'G1320',
 'G1381',
 'G1514',
 'G1634',
 'G1697',
 'G1768',
 'G1853',
 'G1854',
 'G1907',
 'G1994',
 'G2017',
 'G2115',
 'G2193']
```

次のコードにより図 13.11 を生成する。これは図 13.9 に似ているが、遺伝子のサブセットに基づいている。

In [33]:

```
fig, ax = plt.subplots()
ax.plot(Rs, FDRs, 'b', linewidth=3)
ax.set_xlabel("棄却数")
ax.set_ylabel("誤発見率");
```



おわりに

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生の有志とメンターが ISLR と ISLP における実習内容を日本語の教材としての利用可能性を検討した。これら二冊の書籍ではそれぞれ各章に R および Python によるデータ分析の方法が説明されているが、ISL の web 資料として公開されていたので、ChatGPT や Google Colaboratory など新たに利用可能となった方法も活用、その後に各参加者が内容を検討、教育的な見地から有用と判断した内容を日本語の教材としてまとめて掲示する冊子とした。

近年になり統計的学習を巡る議論が活発化する中での統計科学の高等教育も影響され始めている。前回の報告書とともにこの報告書が今後の議論の参考になれば幸いである。

2025 年 3 月
国友直人