情報工学実験4：データマイニング班
(week 5) 線形回帰モデルの多項式拡張、過学習とその回避

1. （復習）線形回帰モデルの実装
2. 入出力における線形と非線形
3. モデルの線形性
4. 多項式モデルによる線形回帰モデルの拡張
5. 実装演習1（多項式モデル導入）
6. 過学習と代表的な回避手段
    1. 実装演習2（ペナルティの導入）
    2. 実装演習3（交差確認）
7. 参考文献

実験ページ: http://ie.u-ryukyu.ac.jp/~tnal/2020/info4/dm/ - week5

目次

Linear Regression Model

review

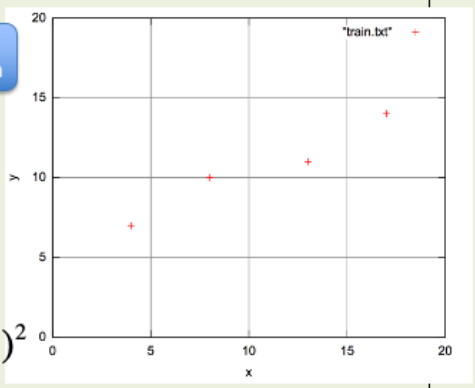
- Training datasets
  - (x,y) = (4,7), (8,10), (13,11), (17,14)
- Hypothesis

  Assumption 1
  Linear function

  $$h_\theta(x) = \theta_0 + \theta_1 x$$
- Parameters
  - θ0, θ1
- Cost function

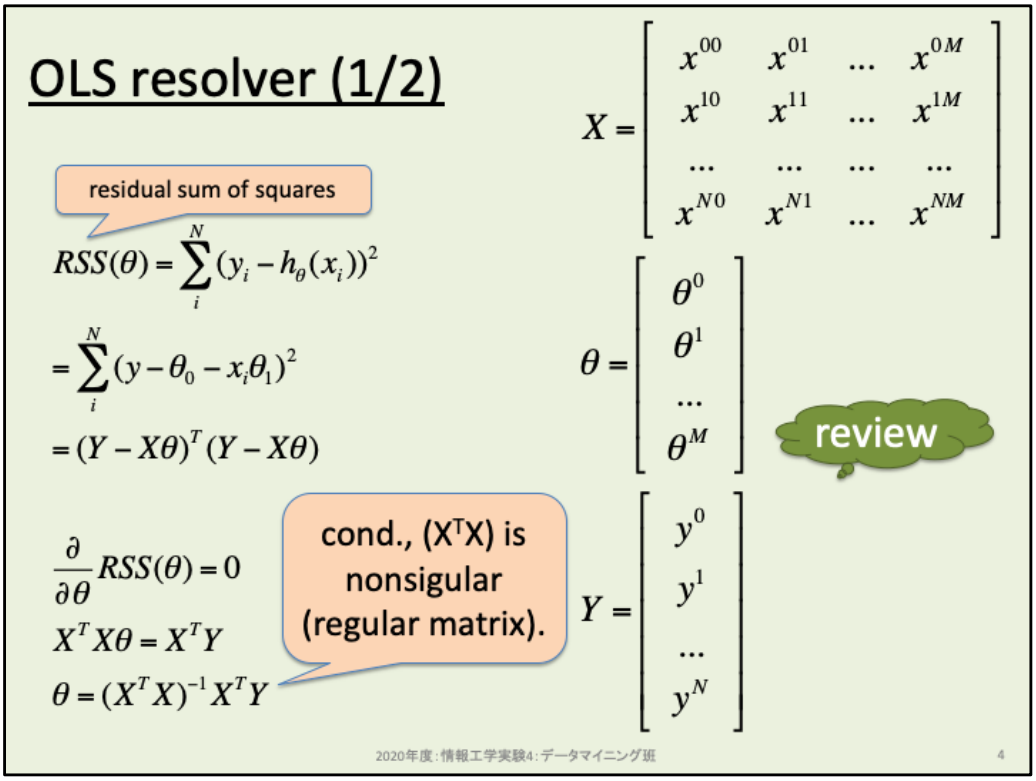
  Assumption 2
  Squared error

  $$J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$$



- Objective function (measurement of the goodness)

  $$\min_{\theta} J(\theta_0, \theta_1)$$

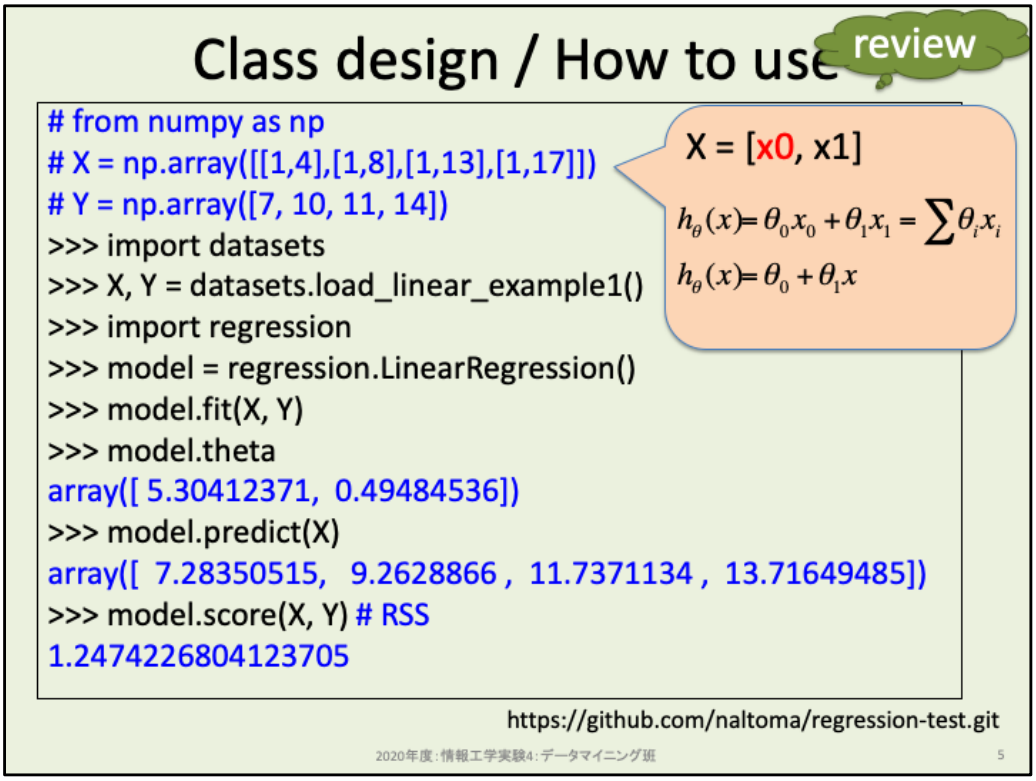


一般化するため、各サンプルxをM次元の特徴ベクトルとする。スライドで0～M個の特徴が並んでおりM+1個に見えるが、0番目のxはバイアス項として利用するための定数（全てのサンプルで1）とする。そのため、x00～x0Mがサンプル0番目の特徴ベクトルである。サンプルがN+1個用意されているものとし、これを行列Xとする。なお、サンプル数を特定していれば、サンプル数はN個（0～N-1なり、1～Nなり）で良い。ここでは単に特徴ベクトルの0～Mという表記に合わせただけである。

バイアス項θ0を含め、各特徴量に対するパラメータθをM+1個の縦ベクトルとして用意する。
サンプル数分の教師データyについても縦ベクトルYとして用意する。

行列X、ベクトルθ、ベクトルYを用いて残差平方和を書き直すとスライド左上のようになる。これを偏微分により0となる方程式に書き直し、θについて展開すると左下のようになる。従って、OLSではこの行列計算を求めることで最適なパラメータを求めることが可能だ。しかし前提として、(X^T・X)の逆行列が存在することが条件である点に注意を要する。

式の展開過程をより詳細に確認してみよう。



実装する際に、入力X、出力Yを np.array 形式で用意するものとする。これは scikit-learn を模倣した設計だ。実際のデータセットは datasets モジュールで用意するものとし、上記コード2行によりX, Yを受け取る。x0がバイアス用の定数(=1)なのは前述のとおりだ。そのため、例えばサンプル1番目の [1, 4] は、実際には1次元の特徴4だけが本来の入力であり、1はバイアス用の入力だ。この点は scikit-learn とは異なっている。Scikit-learnに限らず、このようなバイアス用の定数項はモデル側で準備してあるため、本来は特徴ベクトルに含める必要がない。今回は確認しやすくするためだけに、このような設計をしている。

その後、線形回帰モデルを regression モジュールで用意しておき、モデルを用意したあとで model.fit により適応(学習)させる。このfit関数の中身が先程求めた行列演算になる。また、学習で得られたパラメータは model.theta に保存されるものとしよう。学習後は model.predict により予測する個が可能であり、model.score で残差平方和によるスコアを返すようにしよう。

このように、システムをゼロから作る際にはどのように利用するかという視点から設計図を考えてみると良い。今回はこの設計図を元に、実装していこう。

その前に、便利なモジュールである numpy について軽く紹介する。

regression (ver.2: fit())

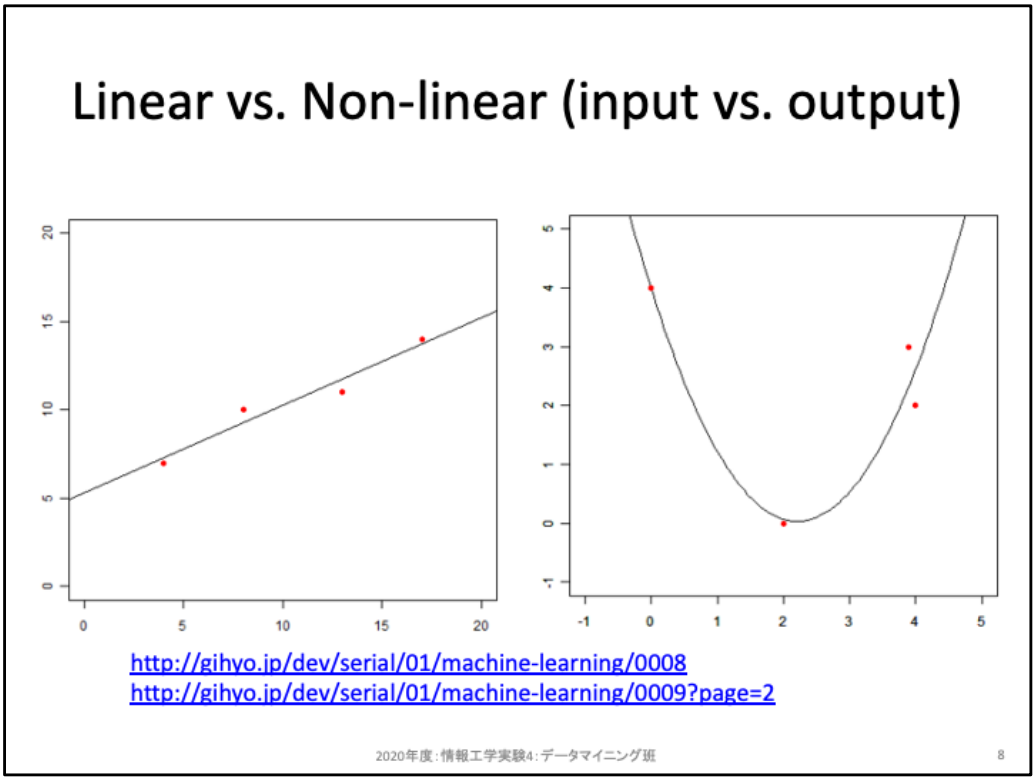
review

```
# testing
>>> import importlib
>>> importlib.reload(regression)
>>> model = regression.LinearRegression()
>>> model.fit(X, Y)
>>> model.theta
array([ 5.30412371,  0.49484536])
```

```
def fit(self, input, output):
    self.theta =
np.dot(np.dot(np.linalg.inv(np.dot(input.T,input)
),input.T),output)
```

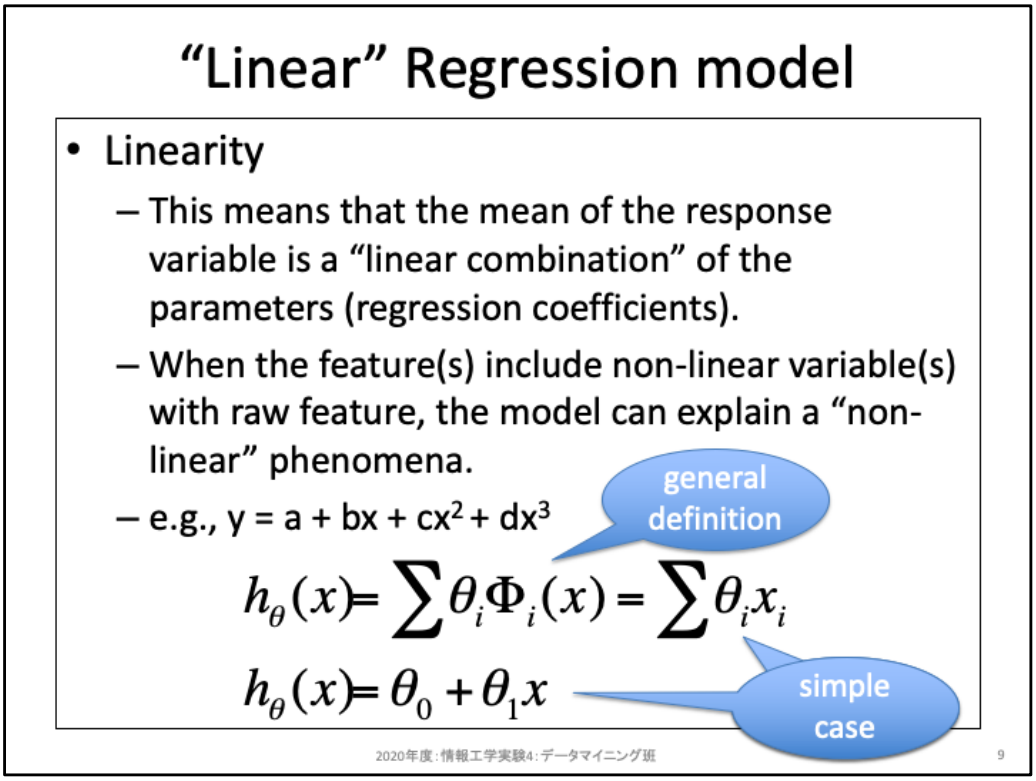
$$\theta = (X^T X)^{-1} X^T Y$$

目次



これらは入力を横軸、出力を縦軸として入出力関係を示した例である。赤い点がサンプルであり、黒線はモデル出力である。
左図は入出力関係が線形になっており、右図は非線形である、ということは分かるだろう。

http://gihyo.jp/dev/serial/01/machine-learning/0008
http://gihyo.jp/dev/serial/01/machine-learning/0009?page=2



線形回帰モデルにおける「線形」とは何だろうか。線形回帰モデルにおける線形等は、直線ということは意味していない。目的変数が説明変数群の線形和（linear combination）で表現されることだけを意味している。例えばスライドにあるように y = a + bx + cx^2 のような形であっても線形回帰モデルであり、これは入力xと出力yの関係は非線形になる。このように線形回帰モデルは説明変数の用意の仕方次第ではより柔軟な表現が可能である。

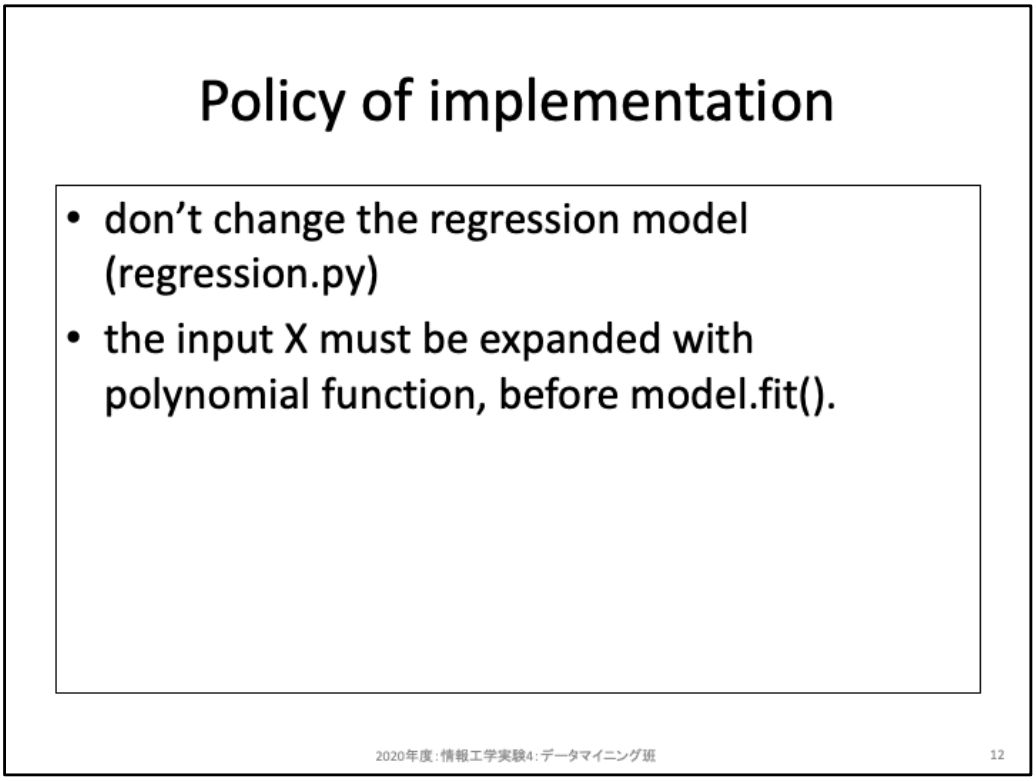
Polynomial expansion (polynomial regression)

- expansion of h(x) with 2nd degree of polynomial function.

$$h_\theta(x) = \sum \theta_i \Phi_i(x) = \theta_0 + \theta_1 x + \theta_2 x^2$$

説明変数を拡張する例として、ここでは2乗する項までを含めた式で仮説を用意してみよう。これを実装してみることにする。

目次



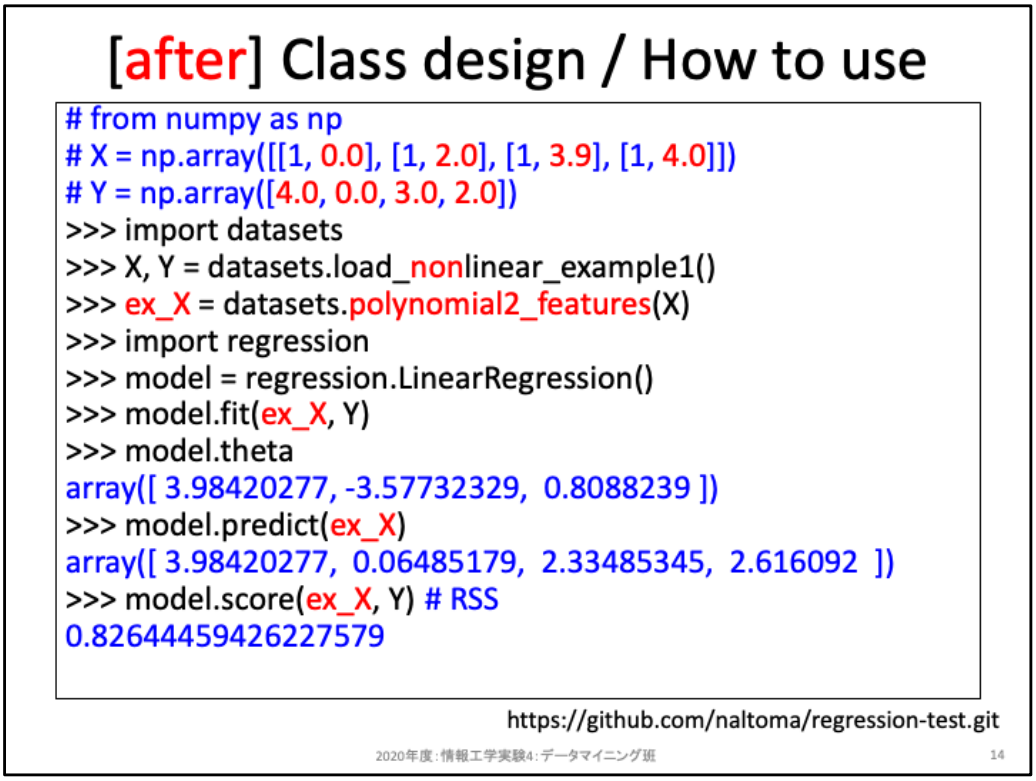
先程の仮説を実装する際に、2つの方針に則ってやろう。1つ目は既に作成した線形回帰モデル（class LinearRegression）には手を加えない。2つ目に、入力xをpolynominal関数で拡張したデータを用意し、それをmodel.fit()に適用する方針で実装しよう。

[before] Class design / How to use

review

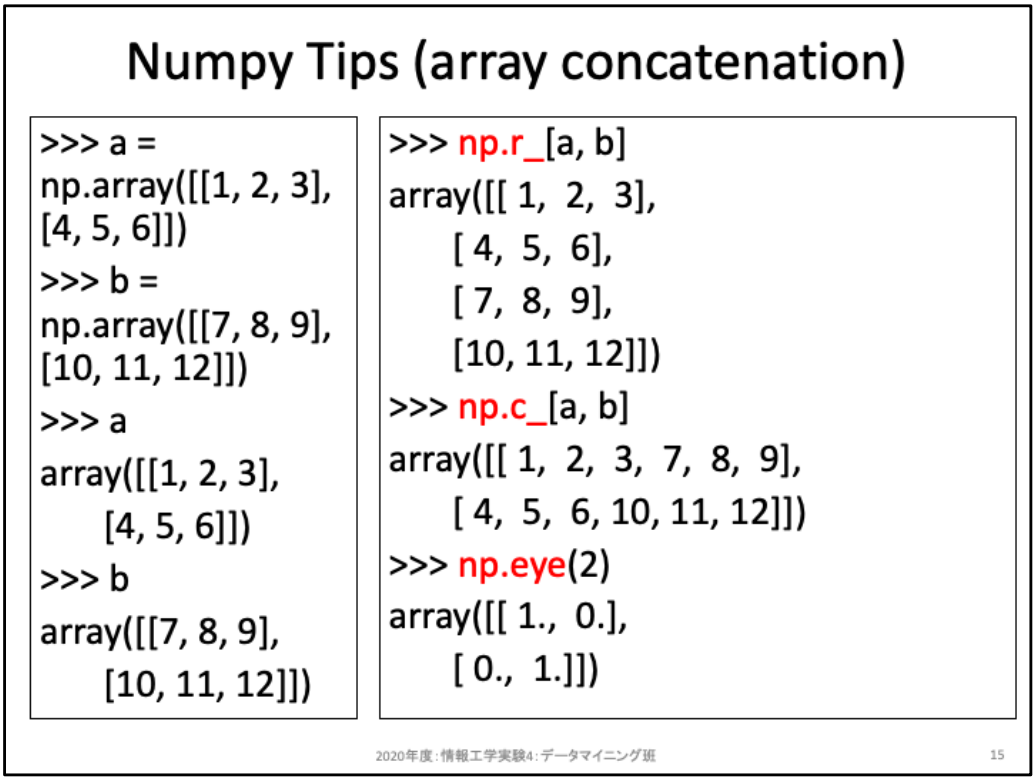
```
# from numpy as np
# X = np.array([[1,4],[1,8],[1,13],[1,17]])
# Y = np.array([7, 10, 11, 14])
>>> import datasets
>>> X, Y = datasets.load_linear_example1()
>>> import regression
>>> model = regression.LinearRegression()
>>> model.fit(X, Y)
>>> model.theta
array([ 5.30412371,  0.49484536])
>>> model.predict(X)
array([  7.28350515,   9.2628866 ,  11.7371134 ,  13.71649485])
>>> model.score(X, Y) # RSS
1.2474226804123705
```

https://github.com/naltoma/regression-test.git



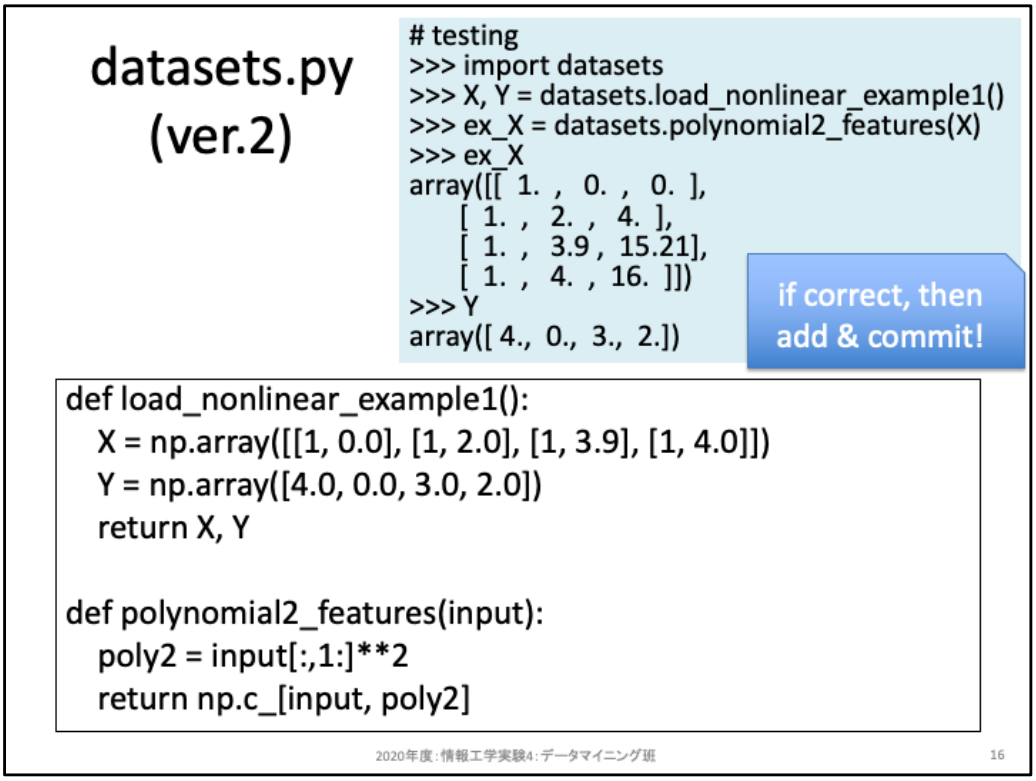
赤字の部分が変更部分である。
同じデータセットでは直線で近似できるため、異なるデータセットを datasets.load_nonlinear_example1() に用意しよう。
それが生データであり、これを polynominal2_features() により2乗の項まで含むデータに拡張する。

その後は拡張したデータを用いて fit, predict , score している。



データセットを拡張する際に、便利な記述を例示している。

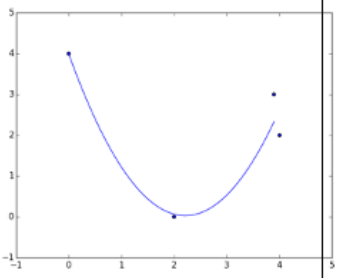
np.r_ は2つの行列を行方向に結合する。
np.c_ は列方向に結合する。
np.eye は対角行列を用意できる。



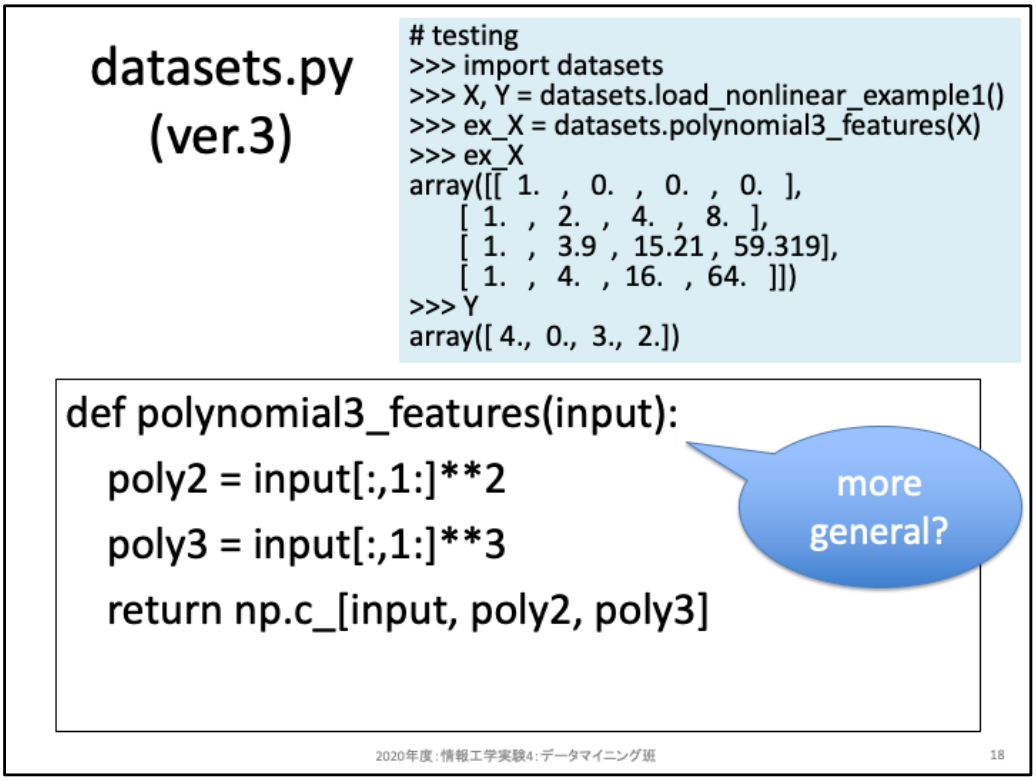
datasets.pyで、新しいデータセットと、2条の項を追加する関数を用意しよう。今回は単体テストまでは書かなくて良いが、動作確認はしよう。

illustrate the model.predict(): nonlinear_ex.py

```
import numpy as np
import datasets
import regression

X, Y = datasets.load_nonlinear_example1()
ex_X = datasets.polynomial2_features(X)
model = regression.LinearRegression()
model.fit(ex_X, Y)

samples = np.arange(0, 4, 0.1)
x_samples = np.c_[ np.ones(len(samples)), samples ]
ex_x_samples = datasets.polynomial2_features(x_samples)

import matplotlib.pyplot as plt
plt.scatter(X[:,1], Y)
plt.plot(samples, model.predict(ex_x_samples))
plt.show()
```



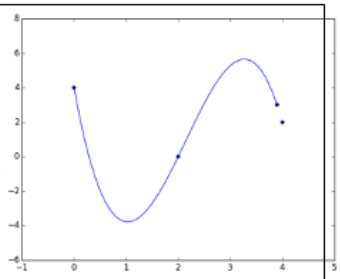
用意したデータセットを用いると、上記コードにより4点のサンプルと線形回帰モデルで学習した曲線を観測できるはずだ。Matplotlibの使い方、特にデータの指定方法を確認しよう。



先程は2乗の項を含める polynomial2_features を実装したが、今度は3乗の項まで拡張してみよう。この意図は、より複雑な事象を表現することを確認するためである。

illustrate the model.predict(): nonlinear_ex2.py

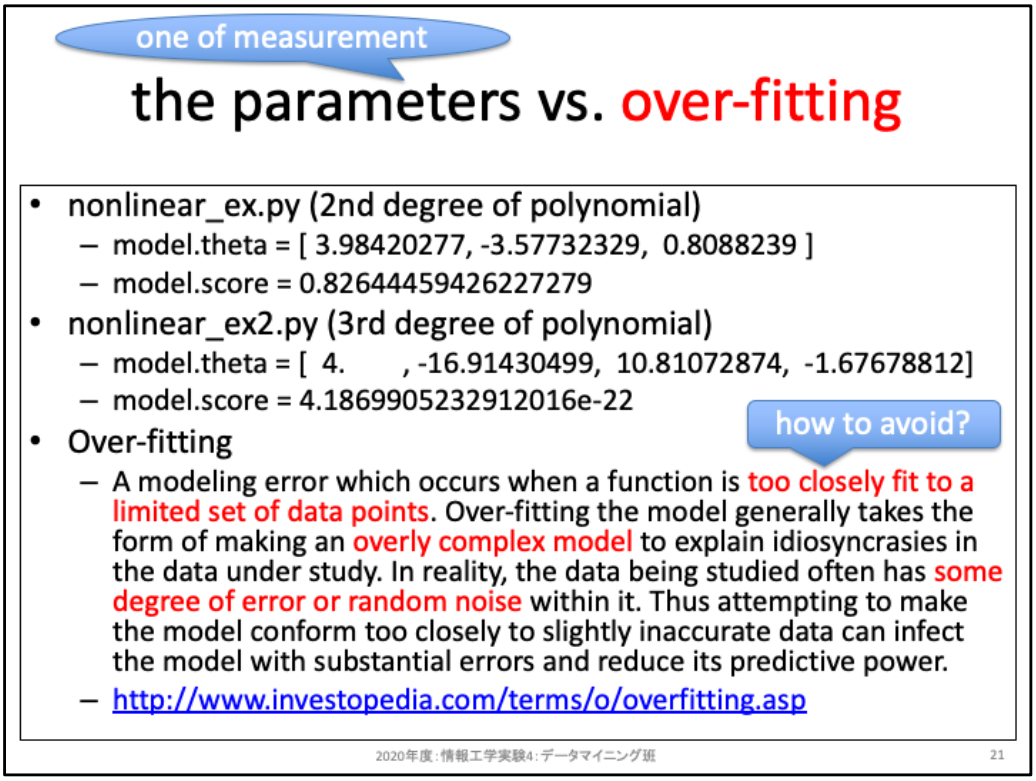
```
import numpy as np
import datasets
import regression

X, Y = datasets.load_nonlinear_example1()
ex_X = datasets.polynomial3_features(X)
model = regression.LinearRegression()
model.fit(ex_X, Y)

samples = np.arange(0, 4, 0.1)
x_samples = np.c_[ np.ones(len(samples)), samples ]
ex_x_samples = datasets.polynomial3_features(x_samples)

import matplotlib.pyplot as plt
plt.scatter(X[:,1], Y)
plt.plot(samples, model.predict(ex_x_samples))
plt.show()
```



polynomial3_featuresで3乗に拡張したデータセットに対して適用してみた。右図のように3次の関数を表現できていることが分かる。また、2次の場合と比べると、主観的にもこちらのほうがフィットしているようにみえるし、scoreの値も良いだろう。（確認してみよう）

さて、このような「モデルの表現能力の向上」は、今回はデータセットの拡張により実現したが、これはモデル内部でも同様のことが可能である。ここで問題として出てくるものが「どこまで表現能力を向上させるのがベストなのか」という視点だ。この点について考えるため、過学習とペナルティという概念を導入しよう。

目次



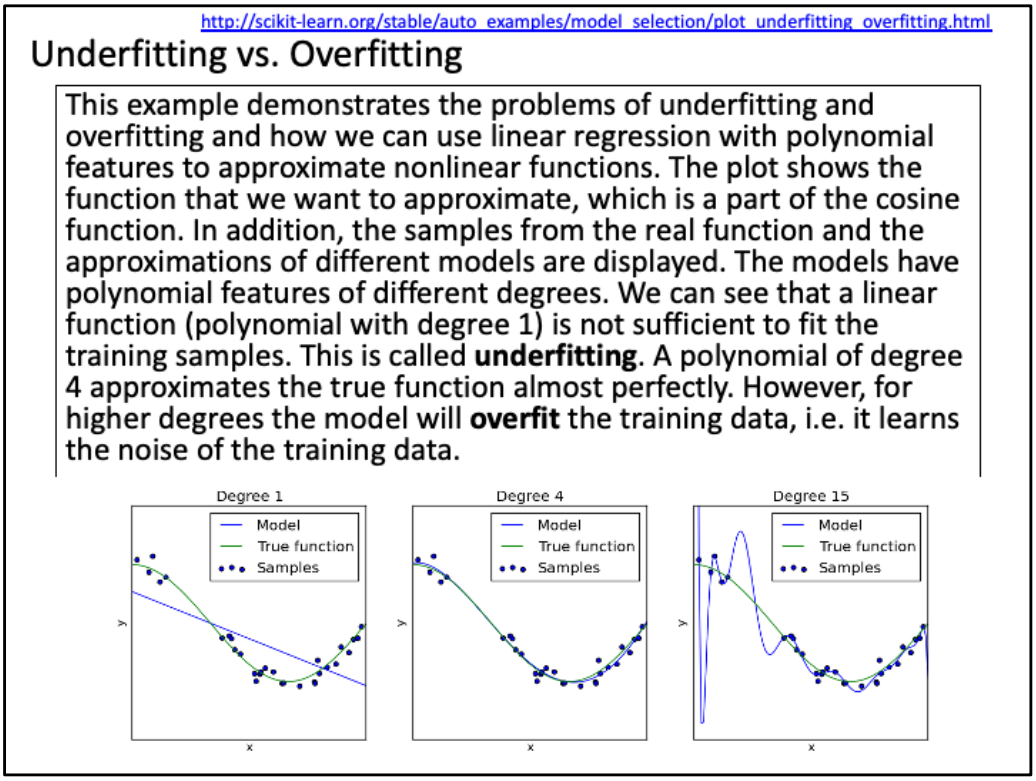
2次の項まで拡張した際に、モデルが学習により獲得したパラメータθと、その時のスコア。
3次の項まで拡張した際に、モデルが学習により獲得したパラメータθと、その時のスコア。それぞれを示している。

両者のスコアを比較すると圧倒的に3次のケースが良い。ところがここには大きな議論の余地がある。それは「データセットを過度に信用しすぎており、殆どのサンプルを綺麗に表現しすぎている」点だ。本来ならばデータセットを綺麗に表現できるモデルは望ましいはずだが、そもそもデータセットは全ての母集団とは透過ではない。母集団から部分的に選ばれたサンプル集合に過ぎず、その選ばれ方に偏りがある場合には母集団を表現するのではなくそのサンプル集合にしか適合していない不適切なモデルとなってしまう。また、基本的にあらゆるサンプル集合にはノイズば含まれると考えるべきだ。例えばTwitterにおけるlikeはどのような意味で成されているだろうか。これは人さまざまであり、解釈の幅が広い。また、誤ってクリックしてしまうこともあるだろう。このようにノイズを含むサンプル集合を過度に信用しすぎたモデルは、未知のサンプルに対しては誤差が大きくなるケースが多々ある。このような状況に陥ることを「過学習」と呼ぶ。

http://www.investopedia.com/terms/o/overfitting.asp

必ずではないが、過学習に陥っているモデルにおいてはパラメータが極端に大きな値を取るケースがママある。先程の2次モデルと3次モデルにおけるパラメータを見比べると、2次のモデルは1桁台に収まっているのに対し、3次のモデルは2桁のパラメータが出てきている。大きなパラメータがあることが正しいこともあるが、極端に大きすぎるパラメータは、多くの場合鋭敏な反応をしすぎていることが多い。そのため、パラメータの大きさに対するペナルティを与えることでこのような状況を抑える工夫をする。

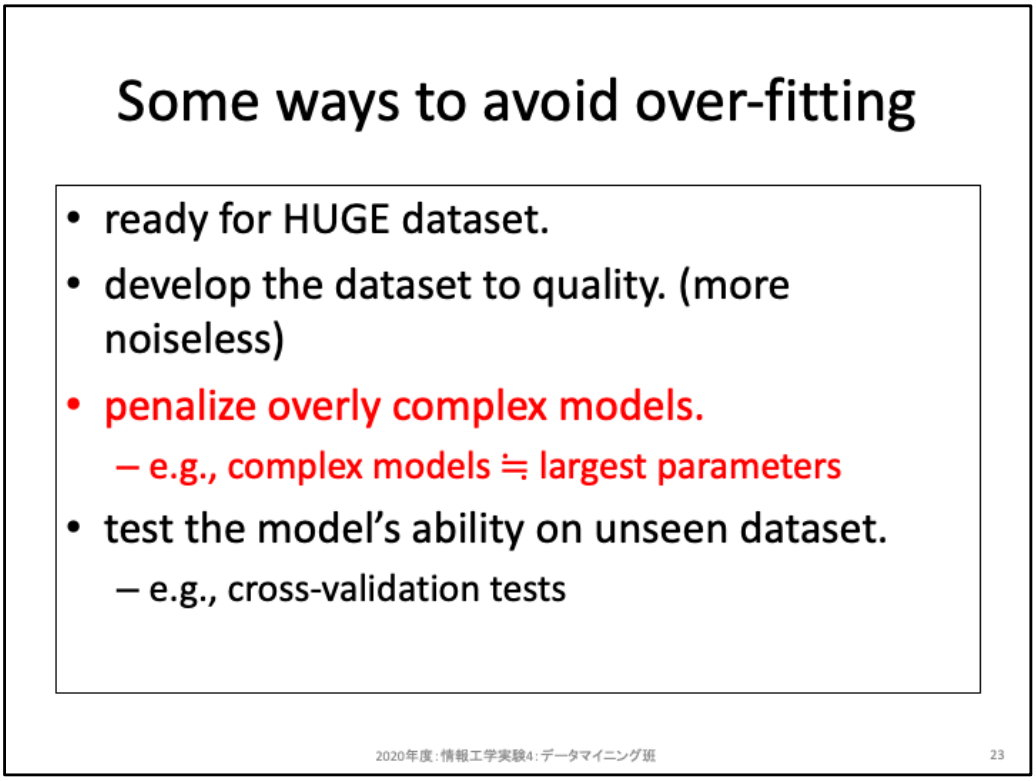
なお、先程の「偏ったサンプル」は reporting bias 等と呼ぶことがある。他にも様々なバイアスやノイズがある。詳細は下記の Fairness を参照するといいだろう。
https://developers.google.com/machine-learning/crash-course/fairness/video-lecture



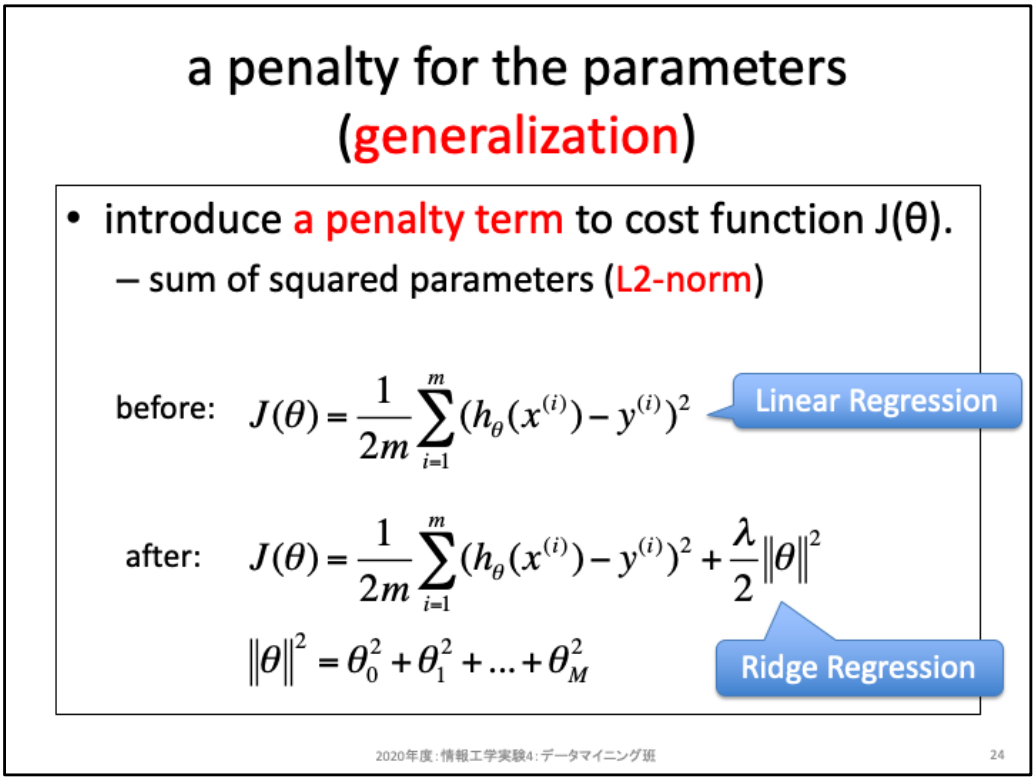
http://scikit-learn.org/stable/auto_examples/model_selection/plot_underfitting_overfitting.html

過学習に陥っている状況を入出力関係を可視化したグラフから観察してみよう。下図は3つの異なる次数を扱うモデルの結果を並べている。各図において黒点はサンプルであり、緑色は観測不可能な真の事象である。サンプルは何かしらのノイズを含むため、事象からは上下に少しずれた場所に位置することもあることが観察できる。また、母集団に対し等間隔でサンプルを取得しているわけではなく、中央から右側に偏っており、左端と中央の間はそもそもデータセットに含まれていないことも観測できる。

このような状況で、最もシンプルな1次の項だけを扱う線形回帰モデルで学習した結果が「Degree 1」である。大雑把に右下がりの傾向にあることは掴めているものの、真の事象に対して表現能力が不足している。このような状況を underfitting と呼ぶ。真ん中の Degree 4 は、サンプルからの誤差という点ではまだ不十分であり、上下の差が残っている（＝誤差が残っている）サンプルが観測できる状態だが、真の事象と見比べるとほぼ重なっている。このようなモデルを、バイアスやノイズを含むデータセットから学習できることが望ましい。これに対し右図の Degree 15 は、モデルの表現能力はとても高く、多くのサンプルをほぼ綺麗になぞるような曲線を獲得している。しかしながら、真の事象と見比べると大きく異なり、特にデータセットに含まれていなかった中央から左端の間については誤差が大きすぎる。また、サンプルがそれなりにあったはずの左端についても極端に下に下る予測を出力しており、誤差が大きすぎる。このような状況を「過学習（overfitting）」と呼ぶ。



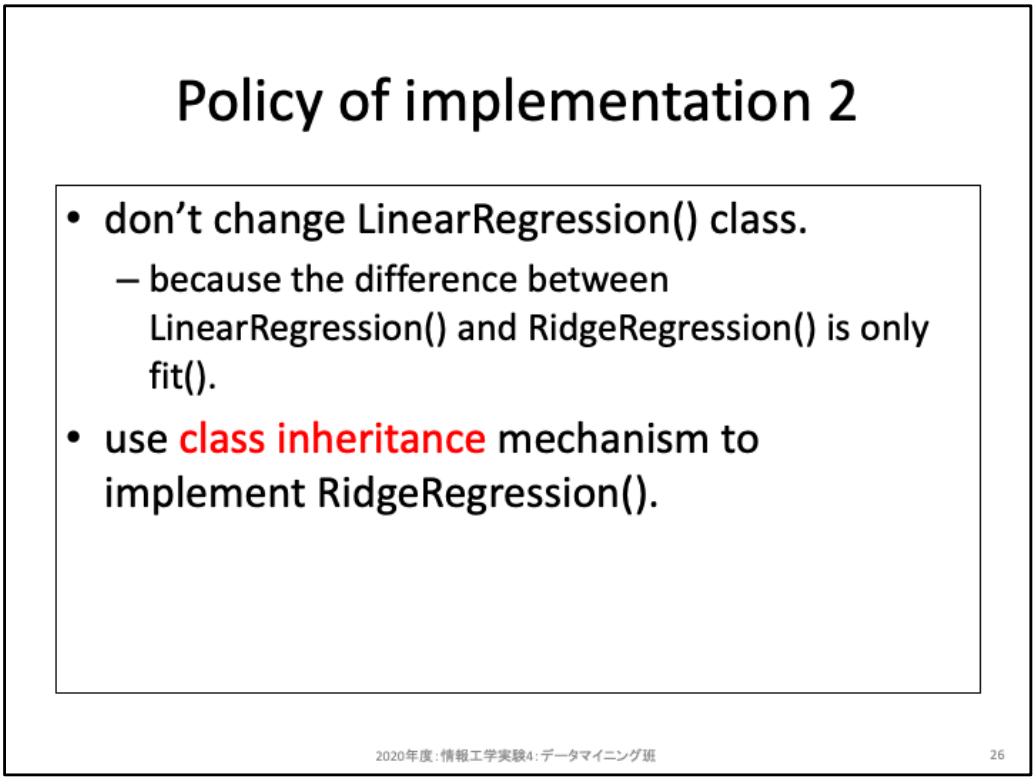
過学習を避けるための代表的なアプローチを示している。まず第一に、データセットを拡充できるならそうしよう。次に、データセットの質を改善できるならばそうしよう。3つ目は、モデルのパラメータに対してペナルティを与えることで、前のスライドで眺めたような「鋭敏な表現を抑える」工夫の導入だ。4つ目は、未知のサンプルに対してテストをすることだ。1つ目と2つ目はデータセット構築を頑張るというだけのため、次スライドでは3つ目と4つ目について具体例とともに考え方を概観していく。



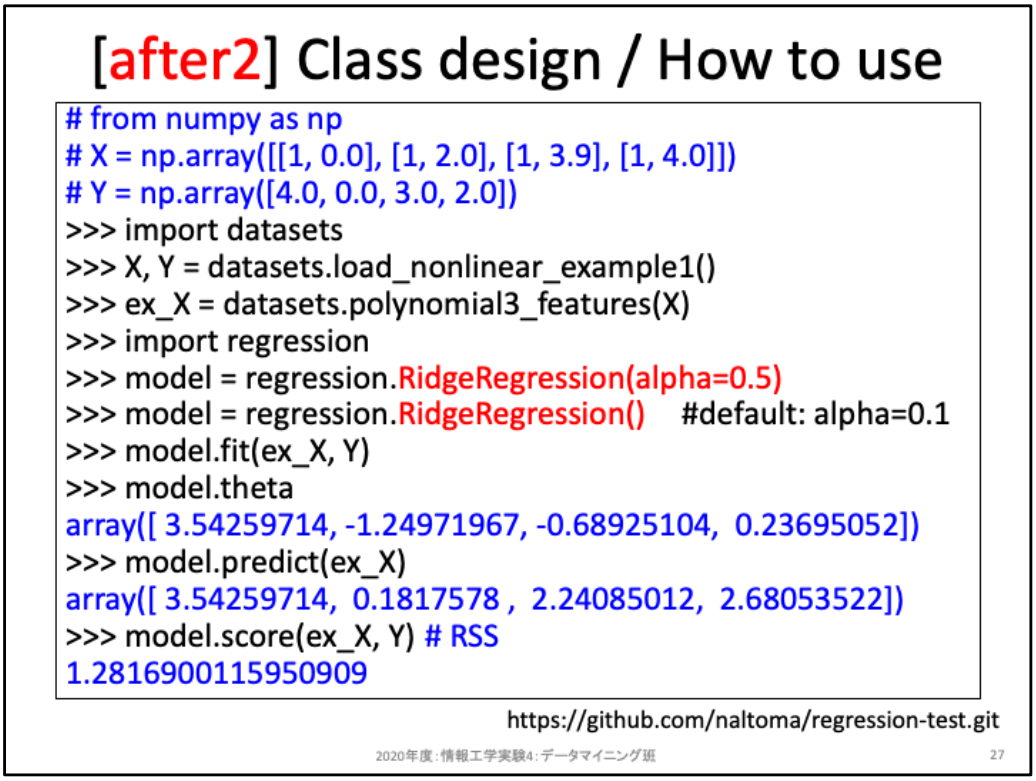
beforeは線形回帰モデルのコスト関数そのものを再掲している。これにペナルティ項としてL2-norm（L2ノルム）を追加したものがafterである。L2-normはパラメータの2乗和であり、各パラメータの大きさに対し指数関数的なコストを加える。このペナルティを加味した上で最適なパラメータを探すことになるため、極端に大きなパラメータを取ることが少なくなり、結果としてデータセットに対して過度に反応しすぎるモデルを構築せず、滑らかなモデルとなることが期待できる。

なお、λはそのペナルティ項と残差平方和との影響と鑑みて調整するハイパーパラメータである。また、線形回帰モデルにL2-normを加えたものはリッジ回帰モデル（Ridge Regression）と呼ばれている。

目次



リッジ回帰モデルを実装してみよう。この際、LinearRegression()は変更しないものとする。なぜなら、線形回帰モデルとリッジ回帰モデルの違いはfit関数（コスト関数）のみだからだ。それ以外は全く同一なので、LinearRegression()を継承してRidgeRegression（）を実装しよう。



リッジ回帰モデルを使う際の設計図を示している。変更点は赤字のみである。

引数のalphaはペナルティ項に対する重みであり、前述のλと考えよう。

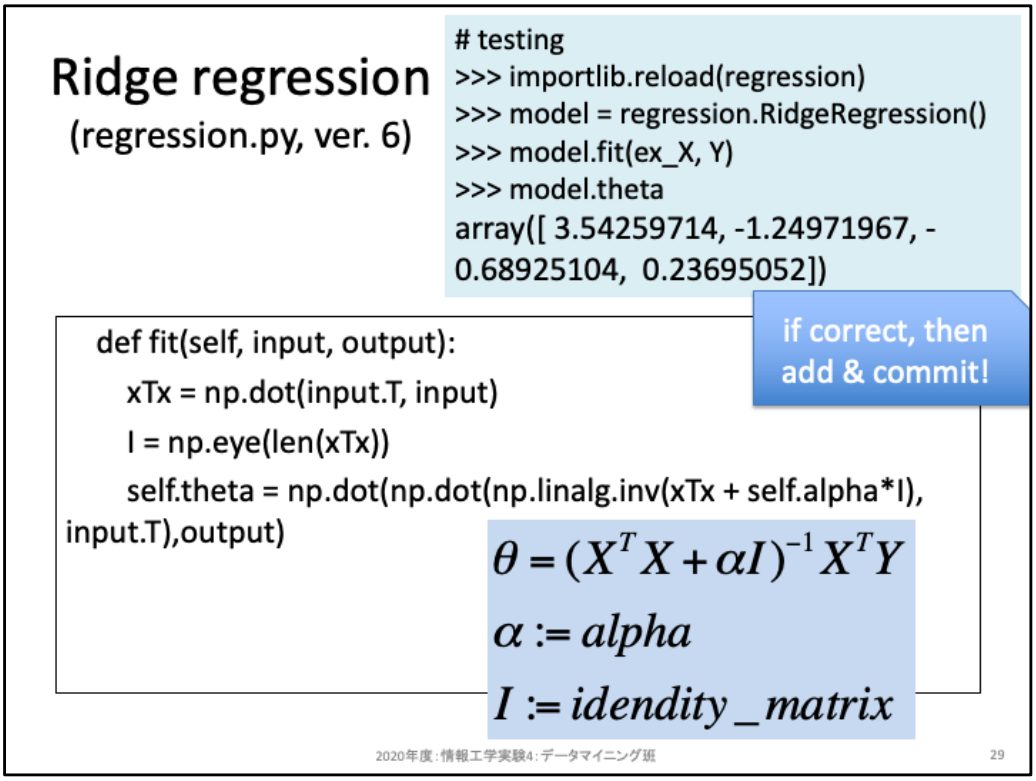
Ridge regression
(regression.py, ver. 5)

```
# testing
>>> import importlib
>>> importlib.reload(regression)
>>> model = regression.RidgeRegression()
>>> model.alpha
0.1
```

if correct, then add & commit!

```
class RidgeRegression(LinearRegression):
  alpha = None

  def __init__(self, alpha=0.1):
    self.alpha = alpha

  def fit(self, input, output):
    pass
```


ひとまずLinearRegressionを継承し、重みalphaを利用するための__init__関数を用意した。fit関数は次スライドで実装する。



L2-normを加えたコスト関数を最小二乗法で解くとスライド右下のようになる。何故こうなるのかは自身で導出してみよう。

Effect of the penalty

- alpha=0 # == LinearRegression()
  - theta: [4. -16.91430499 10.81072874 -1.67678812]
  - score: 7.4647921109305001e-22
- alpha=0.1
  - theta: [3.54259714, -1.24971967, -0.68925104, 0.23695052]
  - score: 1.2816900115950909
- alpha=0.5
  - theta: [2.52220383, -0.63725353, -0.63511135, 0.20043682]
  - score: 3.2271319080413789
- alpha=1.0
  - theta: [1.85895448, -0.43056141, -0.46106559, 0.1538384]
  - score: 5.5987129498416079
- alpha=10.0
  - theta: [0.33402625, -0.04968343, -0.04987846, 0.05004393]
  - score: 14.342958761816003

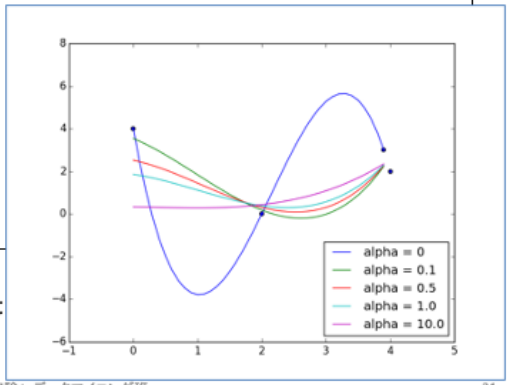
リッジ回帰モデルでalphaを変えて学習させた際に得られたパラメータとスコアを列挙した。alpha=0はペナルティ項自体が0になるため、実質的に線形回帰モデルと同等になる。

alphaを大きくしていくとスコアが悪くなるが、これはペナルティ項分のコストが加わるため必然である。元のスコアより改善するということは起こり得ない。単にスコアの良し悪しを比較して「リッジ回帰モデルより線形回帰モデルの方が良い」という判断は誤りの元なので注意しよう。

パラメータを観察すると、線形回帰モデルでは2桁を取っていたパラメータが、alpha = 0.1 では全て1桁未満になり、alpha=10.0では全てが少数点数となっていることが分かる。このようにパラメータの大きさを抑制し、滑らかなモデルを獲得するのがペナルティ項だ。

Exercise

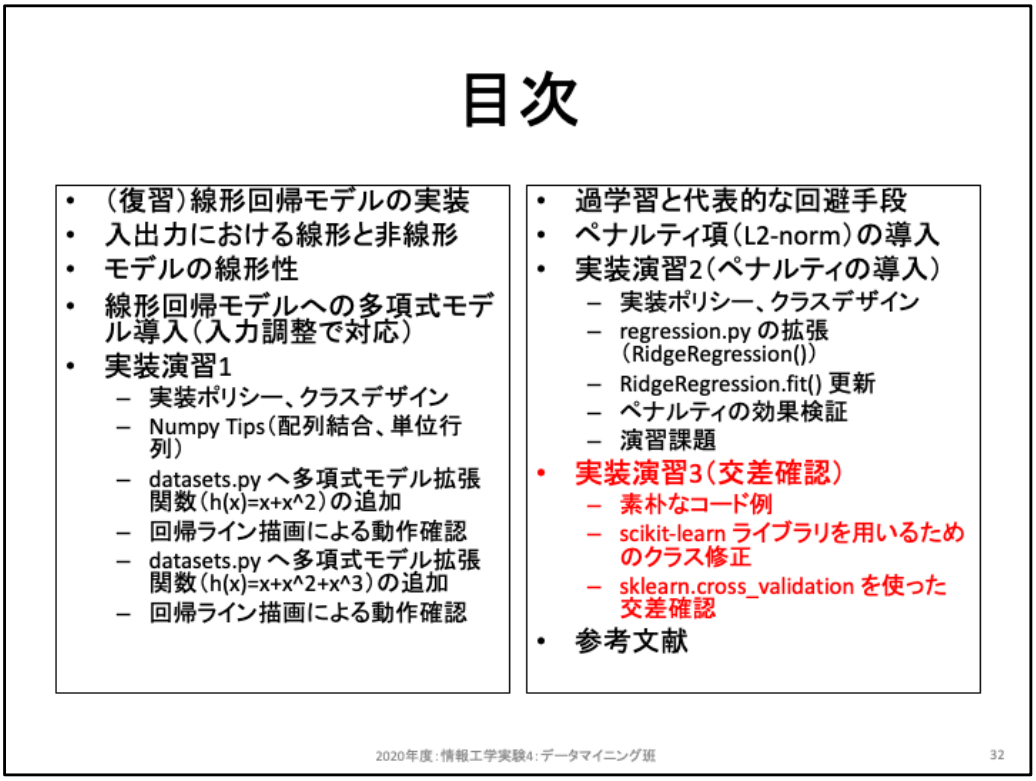
- preconditions
  - X, Y = datasets.load_nonlinear_example1()
  - ex_X = datasets.polynomial3_features(X)
- Illustrate the samples and regression lines on RidgeRegression() with alpha = {0, 0.1, 0.5, 1.0, 10.0}.



code example
https://github.com/naltoma/regression-test.git

さて、実際にどのぐらいなめらかになっているのかを可視化してみよう。右下図のように、alpha=0, 0.1, 0.5,,,と alphaを大きくしていくとなめらかな曲線になっていくことが観察できる。

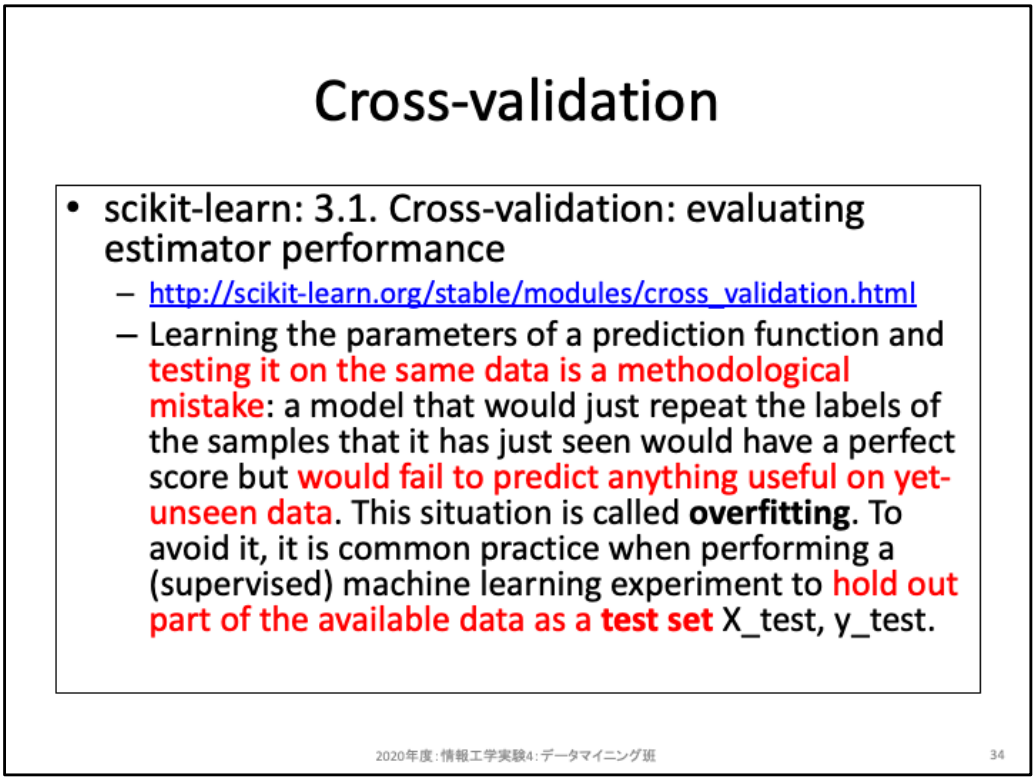
ここではコードは示さない。このような図を作成するコードを書け。



過学習を避けるためのもう一つの工夫である、テストの一種「交差確認（もしくは交差検証）」を概観していく。

Some ways to avoid over-fitting

- ready for HUGE dataset.
- develop the dataset to quality. (more noiseless)
- penalize overly complex models.
  - e.g., complex models ≒ largest parameters
- test the model's ability on unseen dataset.
  - e.g., cross-validation tests

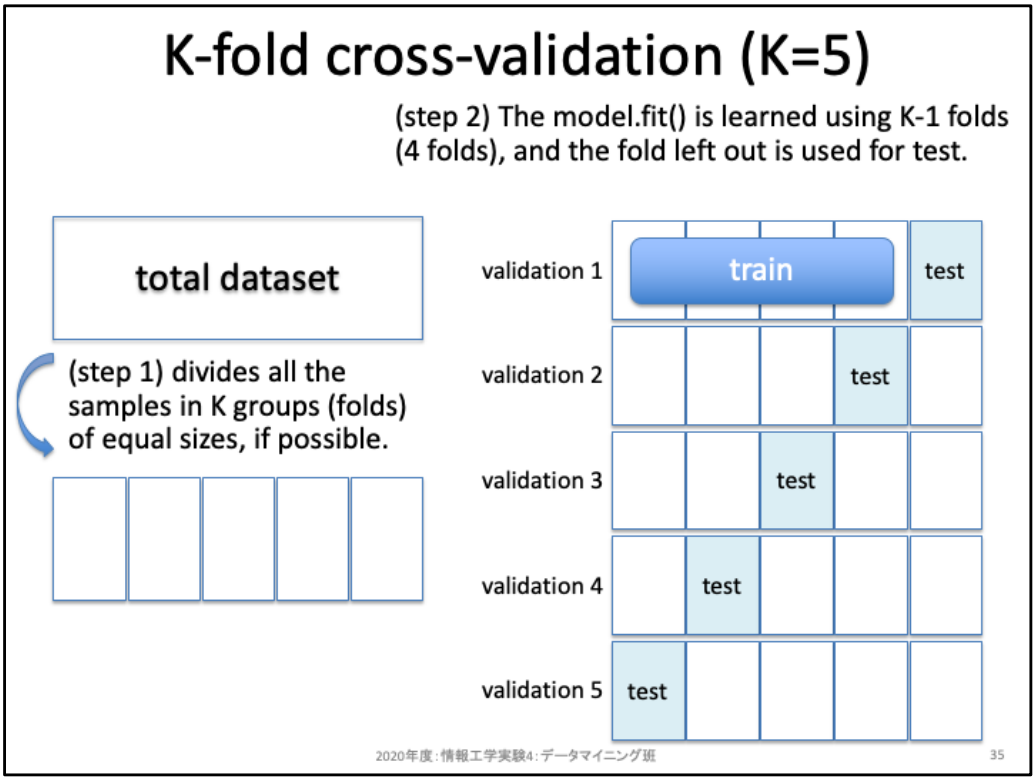


http://scikit-learn.org/stable/modules/cross_validation.html

交差検証とはモデルの妥当さを検証するための手法論の一つである。モデルの良し悪しを図る際に学習時のデータで検証することは、方法論としてそもそも不適切である。例えば、学習時のデータを全て記録させておき、そこに含まれるデータに対しては記録から参照して返すようなモデルを用意しておけば、常に学習データに対して100%の精度となる。これは何かを学習したことになるのだろうか。

このような問題を避けるため、データセットを学習用とテスト用に予め分けておき、学習時には学習データセットのみを与えてfitさせる。学習後のモデルを評価する際にはテストデータセットのみで評価する。

なお、このデータセットの分け方に対していくつか方法論がある。前述のように予め学習用・テスト用と分けて固定する方式をホールドアウト（hold out）と呼ぶ。次スライドでは、似ているが異なる方法論「交差検証」について眺めていこう。



交差検証においては、まずデータセットをk個の集合に分割する。kはユーザが指定する必要があり、スライドではk=5とした。例えばサンプル数が100個あるならば、20サンプルずつの5集合に分割できる。

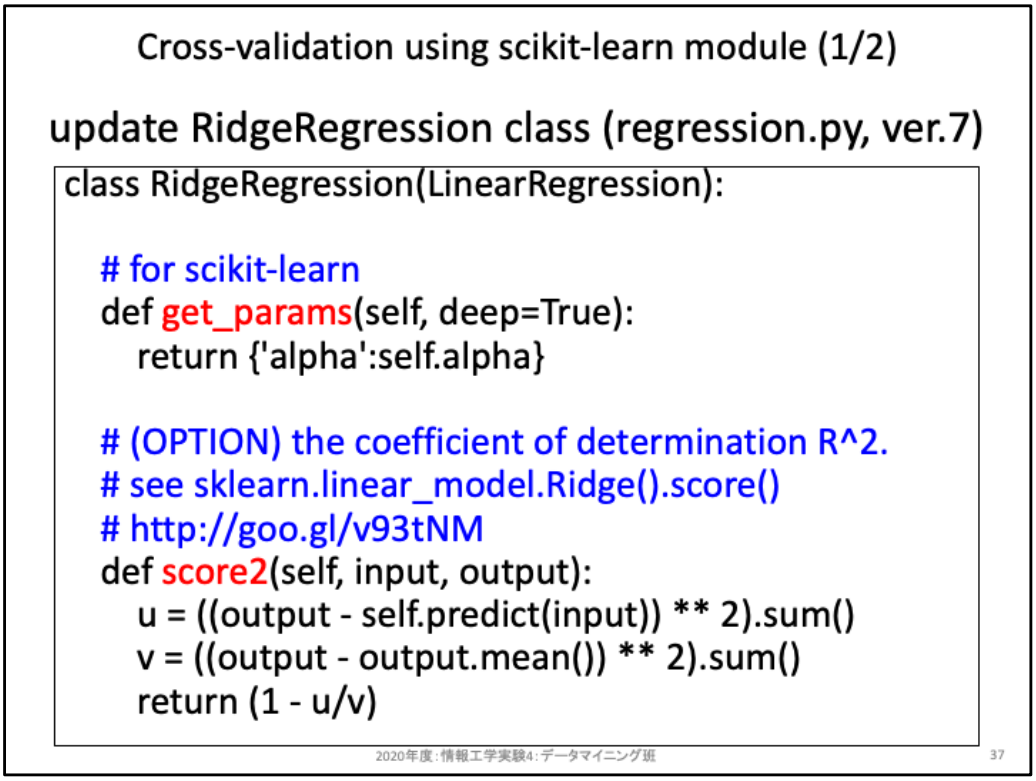
分割後は、1集合をテスト用に残しておき、残りのk-1集合で学習させる。これを全ての組み合わせでやることで、k=5ならば5回のモデル構築を行い、モデルごとにスコアを算出することができる。最終的な判断は全スコアの平均値や分散を観測するとよいだろう。もしモデルが過学習しているならば、平均的に悪いスコアになったり、一部のモデルが極端に良くなったり悪くなったりしていることが多い。

またスライドには示していないが、より一般的には「学習回数毎に、学習データとテストデータに対する評価を比較する」ことをやることも多い。両者を比較することにより、双方に対してスコアが改善している間は過学習に陥っていないことの判断材料の一つになる。逆に、学習データに対しては改善し続けているが、ある回数を境にテストデータに対して頭打ちになったり、スコアが下がり始めたのなら、その付近で過学習に陥り始めているという判断材料となる。

Example of K=2 cross-validation: crossvalidation.py

```
from sklearn import datasets
boston = datasets.load_boston()
x = boston.data
y = boston.target
half = int(len(x)/2)

import regression
model = regression.RidgeRegression(alpha=0.1)
# case 1: learn on the first half, test on the last half
model.fit(x[:half], y[:half])
score = model.score(x[half:], y[half:])
# case 2: learn on the last half, test on the first half
model.fit(x[half:], y[half:])
score += model.score(x[:half], y[:half])
print("RidgeRegression(alpha=0.1) score =", score)
#-> RidgeRegression(alpha=0.1) score = 78656.6246552
#-> RidgeRegression(alpha=1.0) score = 42334.1689238
```


k=2の交差検証（2分割交差検証と呼ぶ）のコード例を示す。ここではデータセットを単にスライス処理で用意した。ただN分割に増やしていくと、自前で書くのは面倒だ。そこでscikit-learnのライブラリを利用することにしよう。



scikit-learnの交差検証を使うにあたり、get_params関数を実装する必要がある。これはモデルで利用するハイパーパラメータを取得するためのものだ。

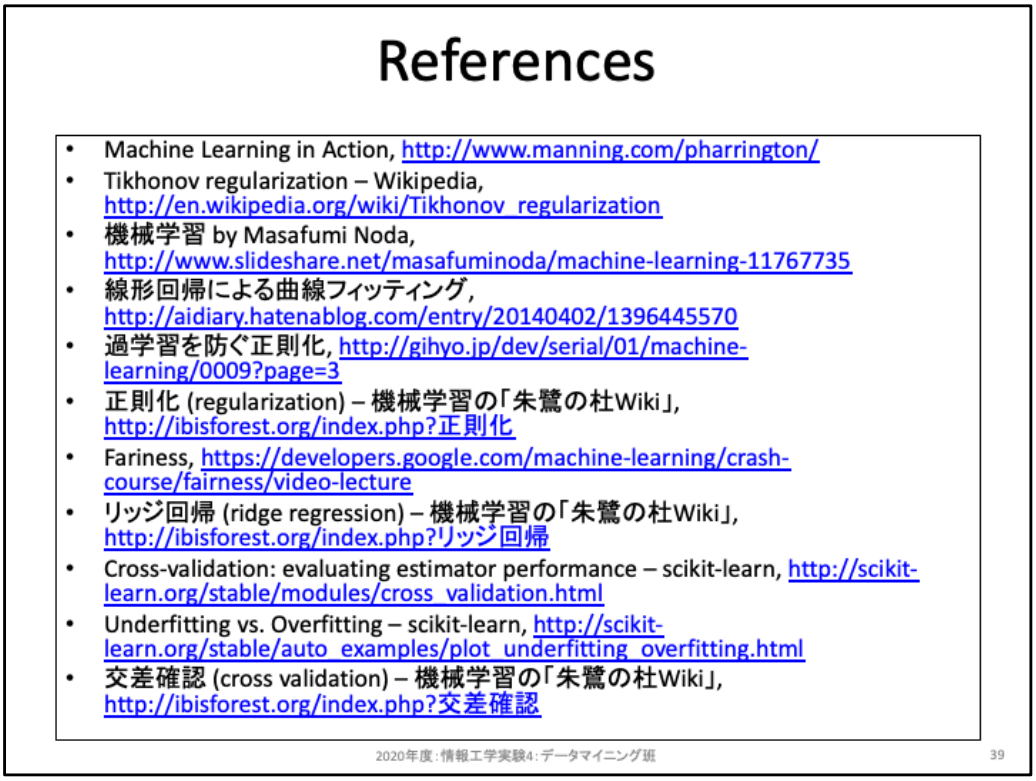
score2関数はなくても構わない。ここではスコアにも様々な算出方法があるという例示のため、scikit-learnのリッジ回帰モデルで採用されているスコア関数を実装した例を示している。

Cross-validation using scikit-learn module (2/2)

crossvalidation_sklearn.py

```
from sklearn import datasets
boston = datasets.load_boston()
x = boston.data
y = boston.target

import numpy as np
ones = np.ones((len(x),1))
ex_x = np.c_[ones, x]

import regression
alpha = 0.1
model = regression.RidgeRegression(alpha=alpha)

#from sklearn import cross_validation #will be removed in 0.20
from sklearn.model_selection import cross_val_score
scores = cross_val_score(model, ex_x, y, cv=10, n_jobs=-1)
print("*** Ridge(alpha=%0.2f) ***" % alpha)
print("scores=", scores)
print("mean score = %f (+/- %0.2f)" % (scores.mean(), scores.std()*2))
```


交差検証は cross_val_score モジュールを利用して実装する。上述のようにモジュールを読み込み、用意したモデルに何分割で検証するのかといった引数を指定して実行するだけで検証を終えることができる。パラメータmodelは用意したモデルそのもの。ex_xとyがデータセット。cvはデータセットの分割数。n_jobsはCPUやコア数が複数ある場合に、最大利用可能数まで使いたいという指定が「-1」だ。例えばp.35の validation 1と validation 2 は、CPUやメモリが空いているならば同時に実行して構わないはずだ。このような並行処理を指定しているのが n_jobs である。



Machine Learning in Action, http://www.manning.com/pharrington/
Tikhonov regularization – Wikipedia, http://en.wikipedia.org/wiki/Tikhonov_regularization
機械学習 by Masafumi Noda, http://www.slideshare.net/masafuminoda/machine-learning-11767735
線形回帰による曲線フィッティング, http://aidiary.hatenablog.com/entry/20140402/1396445570
過学習を防ぐ正則化, http://gihyo.jp/dev/serial/01/machine-learning/0009?page=3
正則化 (regularization) – 機械学習の「朱鷺の杜Wiki」, http://ibisforest.org/index.php?正則化
Fariness, https://developers.google.com/machine-learning/crash-course/fairness/video-lecture
リッジ回帰 (ridge regression) – 機械学習の「朱鷺の杜Wiki」, http://ibisforest.org/index.php?リッジ回帰
Cross-validation: evaluating estimator performance – scikit-learn, http://scikit-learn.org/stable/modules/cross_validation.html
Underfitting vs. Overfitting – scikit-learn, http://scikit-learn.org/stable/auto_examples/plot_underfitting_overfitting.html
交差確認 (cross validation) – 機械学習の「朱鷺の杜Wiki」, http://ibisforest.org/index.php?交差確認