力学系による将来のパラメータ推定

前多啓一

2019 年 1 月 22 日

1　基本的な方針

注意 1.0.1 (方針)．以下の方針で予測を行う．与えられたデータは，n 個の観測ポイントにおける関数 $x : \mathbb{R} \to \mathbb{R}^n \quad t \mapsto (x_1, \cdots, x_n)$ の等間隔 τ の時間 $t_1, \cdots, t_m$ でのデータである．推定するのは，k 番目の特定の変数 x_k の将来での動きである．

- $(1, 2, \cdots, n)$ のなかから，L 個の数が入っているタプルを s 個選ぶ．
- l 番目のタプルから，次の値を最小化する $\psi_l : \mathbb{R}^L \to \mathbb{R}$ を推定する．(ガウス過程回帰)

$$\sum_{i=1}^{m-1} |x_k(t_{i+1}) - \psi_l(x_{l_1}(t_i), x_{l_2}(t_i), \cdots, x_{l_L}(t_i))|$$

- 各 ψ_l より 1 ステップの推定 $\tilde{x}_k^l(t+\tau) = \psi_k^l(x_{l_1}(t), \cdots, x_{l_L}(t))$ を計算する．
- 集めてできた推定の集合から，カーネル密度推定を行うことで，確率密度関数 $p(x)$ を推定する．
- 確率密度関数の歪度 γ を計算し，γ が 0.5 以下であれば採用し，$\tilde{x}_k(t+\tau) = \int xp(x)dx$ を推定として確定する．そうでなければ，以下のように推定値を修正する．交差検証によりインサンプルエラー δ_l を計算し，それに従って r 個のベストなサンプルを選び出す．

$$\tilde{x}_k(t+\tau) = \sum_{i=1}^{r} \omega_i \tilde{x}_k^{l_i}(t+\tau)$$

  ここで，$\omega_i = \dfrac{\exp(-\delta_i/\delta_1)}{\sum_j \exp(-\delta_j/\delta_1)}$ である．

定義 1.0.2 (カーネル密度推定)．$x_1, \cdots, x_n$ を確率密度関数 f をもつ独立同分布からの標本とする．カーネル関数 K，バンド幅 h のカーネル密度推定量とは，

$$\hat{f}_h(x) = \frac{1}{nh} \sum_{i=1}^{n} K\left(\frac{x - x_i}{h}\right)$$

基本的に，$K(x) = \dfrac{1}{\sqrt{2\pi}} e^{-x^2/2}$ を使う．また，最適なバンド幅として，以下の値がある．

$$h^* = \frac{c_1^{-2/5} c_2^{1/5} c_3^{-1/5}}{n^{1/5}},$$

where $c_1 = \displaystyle\int x^2 K(x) dx,\ c_2 = \int K(x)^2 dx,\ c_3 = \int (f''(x))^2 dx.$

これについてはカーネル密度推定が scipy に標準搭載されているのでそちらを援用.

2 ガウス過程回帰について

Bishop を参照 [2] しながら，ガウス過程回帰について復習する.

- 推定の仮定
  $y = \boldsymbol{w}^T \phi(\boldsymbol{x})$ とし，パラメータ $\boldsymbol{w}$ がガウス分布に従うと仮定する.
  すなわち，任意の $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n$ に対し，$\boldsymbol{y} = \Phi \boldsymbol{w}$ はガウス分布に従う．このことから，$\boldsymbol{y}$ は無限次元のガウス分布に従う，などとも言われる．ただし，$\Phi = (\phi(\boldsymbol{x}_i))_{i=1,\cdots,n}$ は計画行列である.
- 与えられるデータ (サンプル)
  $\boldsymbol{x}_1, \cdots, \boldsymbol{x}_n \in \mathbb{R}^n$ および $t_1, \cdots, t_n \in \mathbb{R}$ ただし，$t_n = y_n + \varepsilon_n$ であるとする．ε_n はノイズで，ガウス分布に従うとする.
- 推定するもの
  新しい入力 x_{n+1} が与えられたときの出力 t_{n+1} の確率分布を推定する．すなわち，

  $$p(t_{n+1}|\boldsymbol{x}_{n+1}, \boldsymbol{x}_1, \cdots, \boldsymbol{x}_n, t_1, \cdots, t_n) = N(t_{n+1}|m, \sigma^2)$$

  における m と σ^2 を推定する.

定理 2.0.1. 以下のようにカーネル関数のグラム行列を定義する.

$$K = (k(\boldsymbol{x}_i, \boldsymbol{x}_j))_{i,j}$$

さらに，以下のように置く.

$$\boldsymbol{t} = \begin{pmatrix} t_1 \\ \vdots \\ t_n \end{pmatrix}, \quad \boldsymbol{k} = \begin{pmatrix} k(\boldsymbol{x}_1, \boldsymbol{x}) \\ \vdots \\ k(\boldsymbol{x}_n, \boldsymbol{x}) \end{pmatrix}$$

最適な推定は，以下の通り.

$$\begin{aligned} m &= \boldsymbol{k}^T (K + \sigma_n^2 I)^{-1} \boldsymbol{t} \\ \sigma^2 &= k(\boldsymbol{x}, \boldsymbol{x}) - \boldsymbol{k}^T (K + \sigma_n^2 I) \boldsymbol{k} \end{aligned}$$

3 コード

以上を踏まえ，以下のようにコードを組んだ (参考 [1])

ソースコード 1 main.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('ggplot')
from scipy.stats import gaussian_kde

# 得られている情報の数
initial = 28
```

```
# 予測に使う数
n = 10
# 予測に使うデータの個数
m = 4
# 予測の回数
p = 30
# 予測に使うデータの深さ
q = 10
# カーネル関数の情報
a = 0.05
b = 0.05
beta = 100


# 予測する株の名前
terget = '1812 JT Equity'

similar = ['1801 JT Equity','1802 JT Equity','1803 JT Equity','1812 JT Equity','1820 JT Equity','1821 JT Equity','1824 JT Equity','1833 JT Equity','1860 JT Equity','1893 JT Equity']
# 推定のために 1/1000倍
df = pd.read_csv('./Data/Open.csv',usecols =similar)/1000

df_real = df[0:initial+p]
df = df[0:initial]

# リストを, ある値との距離順に並べる関数
def pointsort(arr,val):
    return [y[1] for y in sorted([(abs(x-val),x) for x in arr])]

# random に n 個の配列を作る
def pickup(arr, n):
    arr2 = arr[:]
    result = []
    for i in range(n):
        x = arr2[int(len(arr2) * np.random.rand())]
        result.append(x)
        arr2.remove(x)
    return result

# a を含まないランダムな配列を n 個作る
def pickup2(arr,a,n):
    arr2 = arr[:]
    arr2.remove(a)
    return pickup(arr2,n-1)

# 多変数のグラム行列を作る関数
def gram(f,x):
    result = []
    for i in x:
        smallresult = []
        for j in x:
            smallresult.append(f(i,j))
        result.append(smallresult)
    return np.array(result)

# ベクトルを作る関数
```

```
def k(f,new,x):
    result = []
    for i in x:
        result.append(f(new,i))
    return np.array(result)

# カーネル推定のクラス
class GaussianKernel(object):
    def __init__(self, params):
        assert np.shape(params) == (2,)
        self.__params = params

    def __call__(self, x1, x2):
        return self.__params[0] * np.exp(-0.5 * self.__params[1] * np.linalg.norm(x1 - x2)
             ** 2)

    # 以下の関数はparameter の推定のための用意
    def get_params(self):
        return np.copy(self.__params)

    def derivatives(self, x1, x2):
        delta_1 = -0.5 * sq_diff * delta_0 * self.__params[0]
        return (delta_0, delta_1)

    def delta0(self,x1,x2):
        sq_diff = np.linalg.norm(x1 - x2) ** 2
        return np.exp(-0.5 * self.__params[1] * sq_diff)

    def delta1(self,x1,x2):
        sq_diff = np.linalg.norm(x1 - x2) ** 2
        return -0.5 * sq_diff * self.delta0(x1,x2) * self.__params[0]

    def update_parameters(self, updates):
        assert np.shape(updates) == (2,)
        self.__params += updates

# GPR 用のクラス
class GaussianProcessRegression(object):
    def __init__(self, kernel, beta=1.):
        self.kernel = kernel
        self.beta = beta

    def fit(self, x, y):
        self.x = x
        self.y = y
        Gram = gram(self.kernel,x)
        self.covariance = Gram + np.identity(len(x)) / self.beta
        self.precision = np.linalg.inv(self.covariance)

    # Kernel function の parameter 推定 / 今回は綺麗に収束しないので, 要検討
    def fit_kernel(self, x, y, learning_rate=0.1, iter_max=100):
        for i in range(iter_max):
            params = self.kernel.get_params()
            self.fit(x, y)
            grad0 = gram(self.kernel.delta0,x)
            grad1 = gram(self.kernel.delta1,x)
            gradients = [grad0,grad1]
```

```
            updates = np.array(
                [-np.trace(self.precision.dot(grad)) + y.dot(self.precision.dot(grad).dot(
                    self.precision).dot(y)) for grad in gradients])
            print(updates)
            self.kernel.update_parameters(learning_rate * updates)
            if np.allclose(params, self.kernel.get_params()):
                break
        else:
            print("parameters may not have converged")

    def predict_dist(self, new):
        K = k(self.kernel, new, self.x)
        mean = K.dot(self.precision).dot(self.y)
        var = self.kernel(new, new) + 1 / self.beta - np.sum(K.dot(self.precision) * K)
        return mean.ravel(), np.sqrt(var.ravel())

# 期待値を算出する関数
def expectation(x,y):
    y = y / sum(y)
    return sum(x*y)

# kernel density estimation をしたのち，期待値を算出する
def kde_process(data_list):
    kde_model = gaussian_kde(data_list)
    y = kde_model(data_list)
    # x_grid = np.linspace(min(data_list), max(data_list), num=100)
    # estimy = kde_model(x_grid)
    # # データを正規化したヒストグラムを表示する用
    # weights = np.ones_like(data_list)/float(len(data_list))
    # # print("mean:", pandasy.mean())
    # plt.figure(figsize=(14,7))
    # plt.plot(x_grid, estimy)
    # plt.hist(data_list, alpha=0.3, bins=20, weights=weights)
    # plt.show()
    skew = pd.Series(y).skew()
    if abs(skew) < 0.1:
        return expectation(data_list,y)
    else:
        data_list2=pointsort(data_list,np.average(data_list))[0:int(len(data_list)/3)]
        return expectation(data_list2,kde_model(data_list2))

# データフレームdf の中から，terget の株の mean と sd を推定する関数
def onetimeestimation(df,terget,n,m,a,b,beta):
    # terget の情報（1 つだけずらして取得）
    y = df[terget].values[1:]
    y = y[:n-1]
    kernel = GaussianKernel(params=np.array([a, b]))
    result = np.array([])
    result_sd = np.array([])
    for i in range(n):
        x = df[pickup2(similar, terget, m)].values
        # 推定前のx(matrix)
        x = x[:n]
        x,x_test = x[:-1],x[-1]
        regression = GaussianProcessRegression(kernel=kernel, beta=beta)
        # regression.fit_kernel(x, y, learning_rate=0.1, iter_max=10000)
        regression.fit(x,y)
```

```
        pred_y, pred_y_sd = regression.predict_dist(x_test)
        result = np.append(result,pred_y)
        result_sd = np.append(result_sd,pred_y_sd)
    mean = kde_process(result)
    sd = np.average(result_sd)
    return mean,sd

# 同業種リストsimilar のすべての 1 ステップを推定する関数
def onetimeallestimation(df,similar,n,m,q,a,b,beta):
    result = []
    result_sd = []
    df = df[-q:]
    for terget in similar:
        result.append(onetimeestimation(df,terget,n,m,a,b,beta)[0])
        result_sd.append(onetimeestimation(df,terget,n,m,a,b,beta)[1])
    return result,result_sd

# 複数回の推定を行う
def manytimeestimation(df,similar,n,m,p,q,a,b,beta):
    error = df[0:0].copy()
    for i in range(p):
        mean,sd = onetimeallestimation(df,similar,n,m,q,a,b,beta)
        mean = pd.Series(mean,index=similar,name='pred'+str(i))
        sd = pd.Series(sd,index=similar,name=i)
        df = df.append(mean)
        error = error.append(sd)
    return df,error

# グラフの出力のための関数
def makegraph(estimation,sd,real):
    x_grid = np.array(range(len(estimation)))
    plt.figure(figsize=(14,7))
    plt.errorbar(x_grid,estimation,sd,fmt='ro-')
    plt.plot(x_grid,real,'go-')
    plt.show()

result, sd = manytimeestimation(df,similar,n,m,p,q,a,b,beta)
estimation = result[terget].values[initial:initial+p]
sd = sd[terget].values[0:p]
real = df_real[terget].values[initial:initial+p]

# print(onetimeestimation(df,terget,n,m,a,b,beta))
# print(onetimeallestimation(df,similar,10,3,10))
makegraph(estimation,sd,real)
```

4 結果

以下の同業種の株で推定を行った．

推定グループ (similar)：業種グループ:Engineering　業種サブグループ:Building&Construc-Misc

'1801 JT Equity','1802 JT Equity','1803 JT Equity','1812 JT Equity','1820 JT Equity',
'1821 JT Equity','1824 JT Equity','1833 JT Equity','1860 JT Equity','1893 JT Equity'

推定した株 (terget)：'1812 JT Equity'
推定に費やした回数 (n)：10 回
推定に使った株の数 (m)：3 ずつ
1 回のカーネル密度推定は以下のようになる.

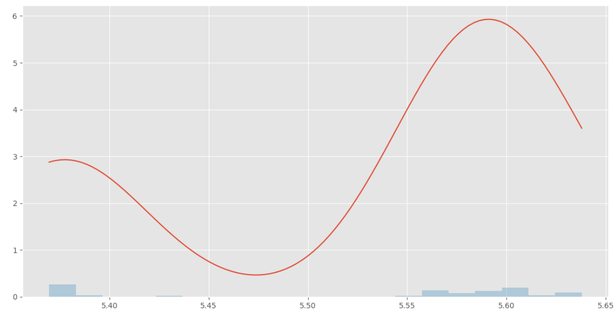

図 1　カーネル密度推定

以下は，10 回すべての株を推定し，将来予測を行なった結果である.

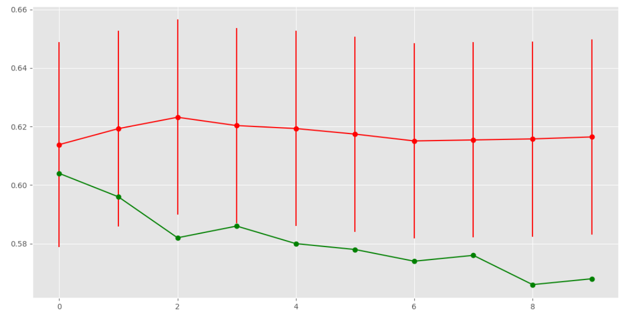

図 2　10 回予測

参考文献

[1] Qiita PRML 第 6 章 ガウス過程による回帰 Python 実装 `https://qiita.com/ctgk/items/4c4607edf15072cddc46`

[2] Christopher M. Bishop "Pattern Recognition and Machine Learning" 2013