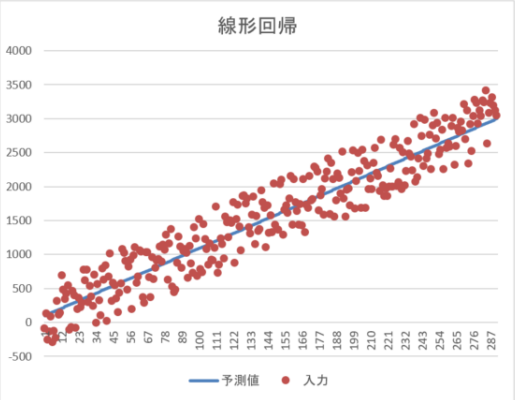
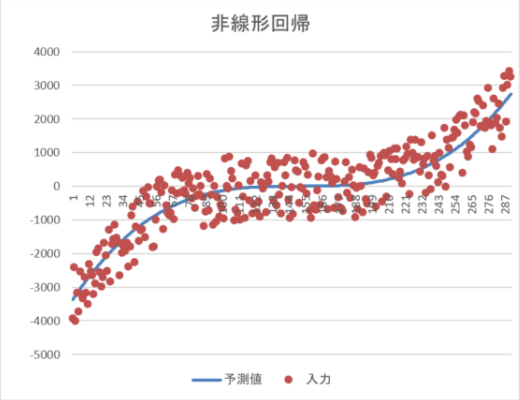
1　非線形回帰

・非線形、つまり一次式でなく高次式で表したいようなデータの形をモデリングしたい
・例えば以下分布の右側





・モデルの形は高次式だけど変数を基底関数 $(\phi_i(x_i))$ に置き換えただけ (基底展開法という)
・非線形関数 (基底関数) とパラメータベクトルの線形結合式となる (以下)

$$y_i = w_0 + \sum_{j=1}^{n} w_j \phi_j(x_i) + \varepsilon_i$$

・未知パラメータ (w) は線形回帰と同じく最小二乗法や最尤法により推定する

・ん?どこかで見たような・・・ラプラス変換に近い・・・波形合成は

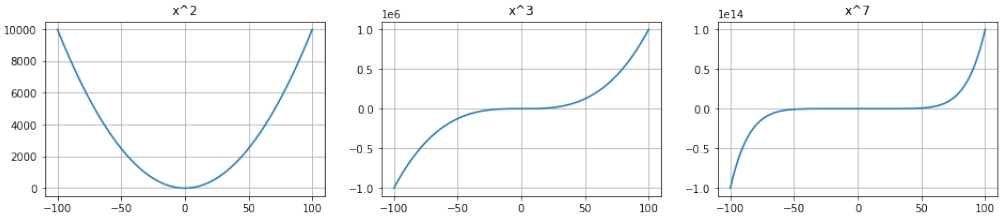
$$L(s) = \int_0^\infty e^{-st} f(t)\, dx$$

2 基底関数

よく使われるものが 3 つ列挙されている

・多項式関数

$$\phi_j = x^j$$



・ガウス型基底関数

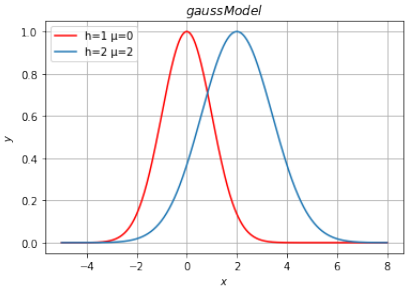
$\phi_j = exp\{\frac{(x-\mu_j)^2}{2h_j}\}$ と紹介されているが $\phi_j = exp\{-\frac{(x-\mu_j)^2}{2h_j}\}$ が正解?

h は幅を決める、μ が中心の位置を決める、となる

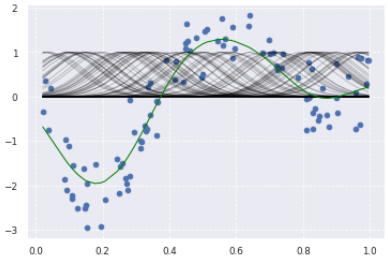
sklearn のガウス基底関数は以下の定義

$\phi_j = exp\{-\frac{\|x-x^{'}\|^2}{2\sigma^2}\}$

修正後の波形が以下なので多分その理解でよい (いわゆる正規分布っぽい形)



演習コードより fit の様子が理解できる (ガウス型基底関数とパラメータの線形結合)



・スプライン関数/B スプライン関数

N 個の制御点間を N-1 次多項式でなめらかに接続する関数

制御点付近を前後近似してなめらかに結合

B スプラインは制御点を通さなくても良いとされている関数

3 予測モデル式

・線形回帰とほぼ同じだが入力値 x_i が関数 $\phi(x_i)$ に置き換わったという事になる

説明変数

$\mathbf{x}_i = (x_{i1}, x_{i2}, \ldots, x_{im}) \in \mathbb{R}^m$(m は説明変数の数)

非線形関数ベクトル

$\boldsymbol{\phi}(\mathbf{x}_i) = (\phi_i(\mathbf{x}_i), \phi_2(\mathbf{x}_i), \ldots, \phi_k(\mathbf{x}_i)) \in \mathbb{R}^k$(k は基底関数の数)

非線形関数の計画行列

$\boldsymbol{\Phi}^{(train)} = (\boldsymbol{\phi}(\mathbf{x}_1), \boldsymbol{\phi}(\mathbf{x}_2), \ldots, \boldsymbol{\phi}(\mathbf{x}_n)) \in \mathbb{R}^k$

予測値の式は以下

$\hat{\mathbf{y}} = \boldsymbol{\Phi}(\boldsymbol{\Phi}^{(train)T}\boldsymbol{\Phi}^{(train)})^{-1}\boldsymbol{\Phi}^{(train)T}\mathbf{y}^{(train)}$

4 未学習 (underfitting) と過学習 (overfitting)

・学習データに対して十分小さな誤差が得られないモデル → 未学習

(対策) 表現力のモデル (高次関数等) を利用する

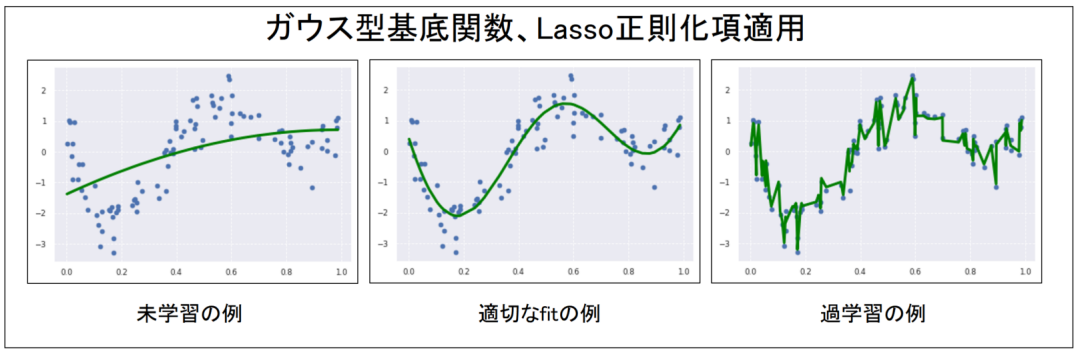
・小さな誤差は得られたけど、テスト集合誤差との差が大きいモデル → 過学習

⇒ 過学習なモデルは教師データに overfit しており汎化性能 (未知データの予測性能) が低い

(対策 1) 学習データの数を増やす (ただしそうそう融通がきくモノでもないだろう)

(対策 2) 不要な基底関数 (変数) を削除して表現力を抑止

(対策 3) 正則化法 (後述) を利用して表現力を抑止



5　過学習対策

・不要な基底関数を削除

基底関数の数、位置やバンド幅によりモデルの複雑さが変化する

解きたい問題に対して多くの基底関数を用意してしまうと、過学習が起こるので

適切な基底関数を用意する (クロスバリデーションなどで選択)

・正則化項 (罰則化法)

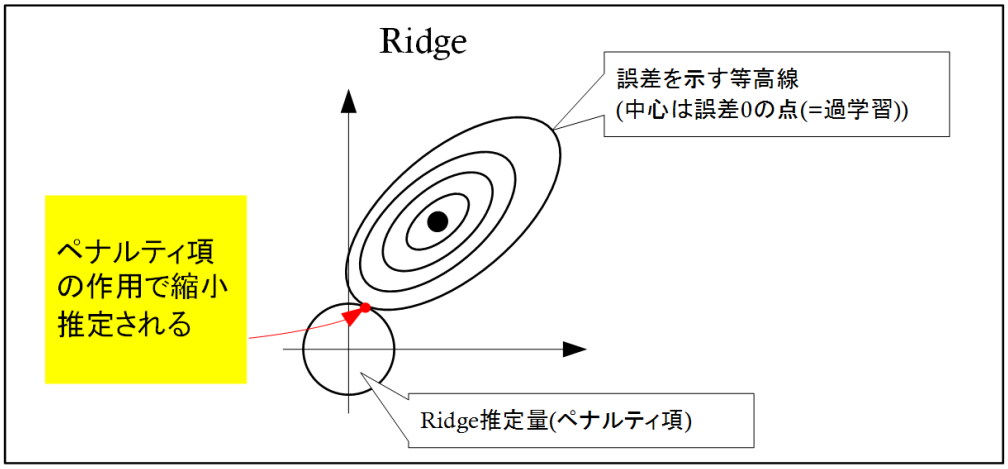
モデルの複雑さに伴って値が大きくなる様な正則化項 ($\gamma R(\mathbf{w})$) を誤差式に加える (ペナルティ)

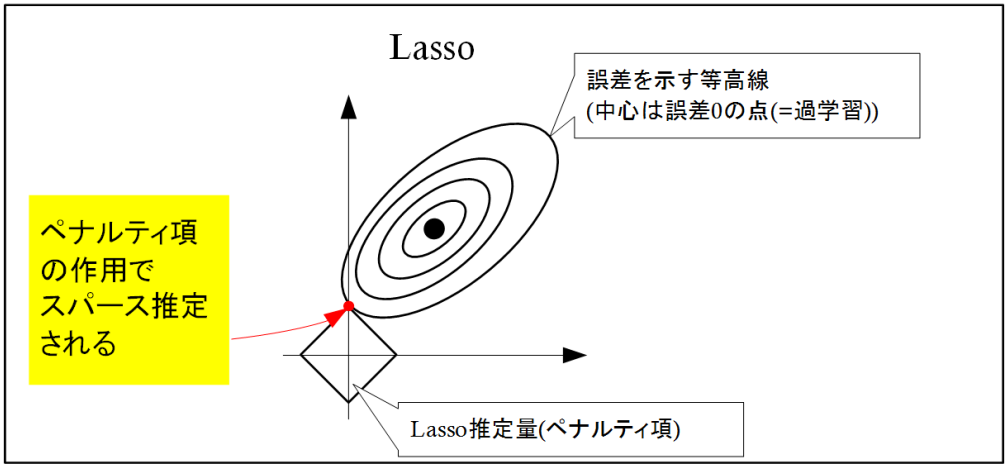
$$\mathbf{S}_\gamma = (\mathbf{y} - \boldsymbol{\phi}\mathbf{w})^T(\mathbf{y} - \boldsymbol{\phi}\mathbf{w}) + \gamma R(\mathbf{w})$$

一般的に Ridge と Lasso がある

Ridge：ペナルティをユークリッド距離 (二次) で与える → 縮小性 (満遍なくパラメータ縮小)

Lasso：ペナルティをマンハッタン距離 (一次) で与える → スパース性 (不要パラメータを 0 に)





6　ホールドアウト、クロスバリデーション、グリッドサーチ

・以下コードの様に理解 (これは実習コードに添付する)
・厳密にはビデオで説明の CV ではないかもしれないが、ご容赦を
(ランダム割り付けかつ再現性担保、また TrainTest 比率 8:2 に対して 10 通りなので
それなりに満遍なく振り分けされていると思っている)

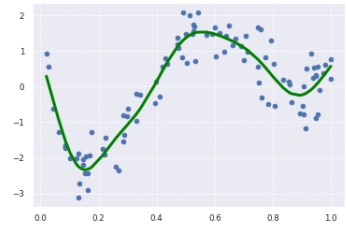
```
#一応・・・CVかつグリッドサーチなコードを作って性能を追いかけておく
#対象はLasso推定とする
#何度かパラメータいじりつつ追い込み

from sklearn.metrics.pairwise import rbf_kernel
from sklearn.linear_model import Lasso

#gammaとalphaをいじって性能を出してみる
gamma_vals=[7,8,9,10,11,12,13,14]  #表現力上げすぎると過学習気味になるのでこのくらいで
alpha_vals=[0.00010,0.00015,0.0002] #alphaが小さすぎると収束しにくくなるのでこのくらいで
rs_vals=[0,10,20,30,40,50,60,70,80,90]  #random_stateの値は適当に振る(これが1個だとホールドアウト法と同じ)

bestscore = [0,0,0]
max_iter = 1000000

#グリッドサーチの部分
for gval in gamma_vals:
  for aval in alpha_vals:
    nowScore=0
    #CVの部分
    for rsval in rs_vals:
      #trainデータとtestデータの比率は8:2にした(trainとtest)
      #ランダム振り分け(だけど再現性は担保)
      kx = rbf_kernel(X=data, Y=data, gamma=gval)
      d_train, d_test, t_train, t_test = train_test_split(kx, target, random_state=rsval,test_size=0.2)
      lasso_clf = Lasso(alpha=aval, max_iter=max_iter)
      lasso_clf.fit(d_train, t_train)

      #テストデータにて評価
      nowScore += lasso_clf.score(d_test,t_test)

    #最良パラメータを保持
    score = nowScore/len(rs_vals)
    print("gamma={} alpha={} clf={}".format(gval,aval,score))
    if bestscore[0] < score:
      bestscore[0] = score
      bestscore[1] = aval
      bestscore[2] = gval
```

7　実習

・変更箇所 (Lasso 推定量が機能してないのでハイパーパラメータをチューニング)

```
#Lasso

from sklearn.metrics.pairwise import rbf_kernel
from sklearn.linear_model import Lasso

kx = rbf_kernel(X=data, Y=data, gamma=50)   #gammaも一桁多くしておく(基底関数の数)
#KX = rbf_kernel(X, x)

#lasso_clf = LinearRegression()
#lasso_clf = Lasso(alpha=10000, max_iter=1000)  #なんだか・・・・alpha値が大きすぎて正則化が効きすぎ、すべてのパラメータが0に帰着している
lasso_clf = Lasso(alpha=0.001, max_iter=100000) #このくらいが良さげ??iterが1000だと収束しないって警告しているので適当に増やした

lasso_clf.fit(kx, target)

p_lasso = lasso_clf.predict(kx)

plt.scatter(data, target)

#ちょっとOverfit気味かも?最後に交差検証かつグリッドサーチにて追い込みしておく
#plt.plot(data, p, color='green', marker='o', linestyle='-', linewidth=0.1, markersize=3)
plt.plot(data, p_lasso, color='green', linestyle='-', linewidth=3, markersize=3)

print(lasso_clf.score(kx, target))
```

```
0.8717751343330086
```

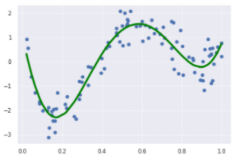


・追加箇所 (Lasso でグリッドサーチ CV を追加)

```
#一応・・・CVかつグリッドサーチなコードを作って性能を追いかけておく
#対象はLasso推定とする
#何度かパラメータいじりつつ追い込み

from sklearn.metrics.pairwise import rbf_kernel
from sklearn.linear_model import Lasso

#gammaとalphaをいじって性能を出してみる
gamma_vals=[7,8,9,10,11,12,13,14]  #表現力上げすぎると過学習気味になるのでこのくらいで
alpha_vals=[0.00010,0.00015,0.0002] #alphaが小さすぎると収束しにくくなるのでこのくらいで
rs_vals=[0,10,20,30,40,50,60,70,80,90]  #random_stateの値は適当に振る(これが1個だとホールドアウト法と同じ)

bestscore = [0,0,0]
max_iter = 1000000

#グリッドサーチの部分
for gval in gamma_vals:
  for aval in alpha_vals:
    nowScore=0
    #CVの部分
    for rsval in rs_vals:
      #trainデータとtestデータの比率は8:2にした(trainとtest)
      #ランダム振り分け(だけど再現性は担保)
      kx = rbf_kernel(X=data, Y=data, gamma=gval)
      d_train, d_test, t_train, t_test = train_test_split(kx, target, random_state=rsval,test_size=0.2)
      lasso_clf = Lasso(alpha=aval, max_iter=max_iter)
      lasso_clf.fit(d_train, t_train)

      #テストデータにて評価
      nowScore += lasso_clf.score(d_test,t_test)

    #最良パラメータを保持
    score = nowScore/len(rs_vals)
    print("gamma={} alpha={} clf={}".format(gval,aval,score))
    if bestscore[0] < score:
      bestscore[0] = score
      bestscore[1] = aval
      bestscore[2] = gval

#確定したパラメータにて全データをfitさせて具合を見る
print("----------------------------------------------")
print("gamma={} alpha={} clf={}".format(bestscore[2],bestscore[1],bestscore[0]))
kx = rbf_kernel(X=data, Y=data, gamma=bestscore[2])
lasso_clf = Lasso(alpha=bestscore[1], max_iter=max_iter)
lasso_clf.fit(kx, target)
score = lasso_clf.score(kx, target)
print(score)
p_lasso = lasso_clf.predict(kx)
plt.scatter(data, target)
plt.plot(data, p_lasso, color='green', linestyle='-', linewidth=3, markersize=3)

print(lasso_clf.coef_)
```

結果は以下

```
gamma=14 alpha=0.0002 clf=0.7962343976071781
----------------------------------------------
gamma=8 alpha=0.0001 clf=0.8007397804470477
0.8675846229406591
[ -0.          -0.          -0.          -2.2575533  -14.748422
  -0.39253667  -0.3694471   -4.3716054   -0.          -0.
  -0.          -0.          -0.          -0.          -0.
  -0.          -0.          -0.          -0.          -0.
  -0.          -0.          -0.           0.           0.
   0.           0.           0.           0.           0.
   0.           0.           0.          -0.          -0.
  -0.          -0.          -0.          -0.          -0.
  -0.          -0.          -0.          -0.          -0.
  -0.          -0.          -0.40833384  -0.16622257  -1.4981425
  -2.265881   -1.9983274   -3.6410892   -3.9792113   -0.
  -0.          -0.          -0.          -0.          -0.
  -0.          -0.          -0.          -0.           0.
   0.           0.           0.           0.           0.
   0.           0.           0.           0.           0.
   0.           0.           0.24517687   0.           0.
   0.           0.           0.           0.           0.
   0.           0.           0.           0.           0.
  -0.          -0.          -0.          -0.          -0.
  -0.          -0.          -0.         -19.954954    -0.03368528]
```

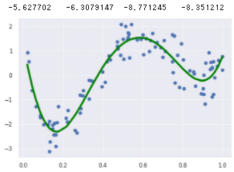


・追加箇所 (Ridge でグリッドサーチ CV を追加)

```
#一応・・・CVかつグリッドサーチなコードを作って性能を追いかけておく
#対象はRidge推定とする
#何度かパラメータいじりつつ追い込み

from sklearn.metrics.pairwise import rbf_kernel
from sklearn.linear_model import Ridge

#gammaとalphaをいじって性能を出してみる
gamma_vals=[6,7,8,9,10,11,12,13,14]  #表現力上げすぎると過学習気味になるのでこのくらいで
alpha_vals=[0.0004,0.0006,0.0008,0.002,0.004,0.006] #alphaが小さすぎると収束しにくくなるのでこのくらいで
rs_vals=[0,10,20,30,40,50,60,70,80,90]  #random_stateの値は適当に振る(これが1個だとホールドアウト法と同じ)
bestscore = [0,0,0]

#グリッドサーチの部分
for gval in gamma_vals:
  for aval in alpha_vals:
    nowScore=0
    #CVの部分
    for rsval in rs_vals:
      #trainデータとtestデータの比率は8:2にした(trainとtest)
      #ランダム振り分け(だけど再現性は担保)
      kx = rbf_kernel(X=data, Y=data, gamma=gval)
      d_train, d_test, t_train, t_test = train_test_split(kx, target, random_state=rsval,test_size=0.2)
      ridge_clf = Ridge(alpha=aval)
      ridge_clf.fit(d_train, t_train)

      nowScore += ridge_clf.score(d_test,t_test)

    score = nowScore/len(rs_vals)
    print("gamma={} alpha={} clf={}".format(gval,aval,score))
    if bestscore[0] < score:
      bestscore[0] = score
      bestscore[1] = aval
      bestscore[2] = gval

#確定したパラメータにて全データをfitさせて具合を見る
#性能はLassoと大差なし(追い込んだ範疇では)
print("----------------------------------------------")
print("gamma={} alpha={} clf={}".format(bestscore[2],bestscore[1],bestscore[0]))
kx = rbf_kernel(X=data, Y=data, gamma=bestscore[2])
ridge_clf = Ridge(alpha=bestscore[1])
ridge_clf.fit(kx, target)
score = ridge_clf.score(kx, target)
print(score)
p_ridge = ridge_clf.predict(kx)
plt.scatter(data, target)
plt.plot(data, p_ridge, color='green', linestyle='-', linewidth=3, markersize=3)

print(ridge_clf.coef_)
```

結果は以下

```
gamma=14 alpha=0.006 clf=0.796870899168711
----------------------------------------------
gamma=6 alpha=0.0004 clf=0.8037150111170289
0.8689522329613967
[[-1.7210898  -1.7939197  -2.641679   -3.014607   -3.5894883  -3.392647
  -3.5712926  -3.512422   -2.978064   -2.8076665  -2.696084   -2.8633997
  -2.370618   -2.345079   -1.9763461  -2.1347811  -1.7751248  -1.7152865
  -1.643876   -1.2658402   0.09369346  0.5522729   0.79590064  1.6235577
   1.8895519   2.3092868   2.4114997   2.3353364   2.53606     2.596175
   2.6975713   2.6385517   2.412571    1.3882681   1.560715    1.0389895
   0.71713805  0.4233543   0.12782964 -1.0216578  -1.005301   -1.1018608
  -1.5133779  -1.655468   -2.0138793  -1.9615527  -2.1841063  -2.5139518
  -2.5013447  -2.455747   -2.5574367  -2.6590219  -2.808254   -2.9552917
  -2.9333465  -2.7649004  -2.713835   -2.5772202  -2.187972   -2.018893
  -1.6557192  -1.3418303  -1.0865762  -0.26005012 -0.01023729  0.22443543
   0.39138013  1.6279385   2.0162287   2.1232598   1.8736951   2.0746384
   2.6890829   2.697162    2.879693    2.9480004   2.896578    3.0271542
   2.798359    2.6646152   2.6455586   1.3228853   0.779862    0.39995766
   0.1618402   0.28490424 -0.43190813 -1.0385355  -2.1340294  -2.126783
  -2.653896   -2.947329   -2.8445418  -3.1782913  -3.373766   -3.7390661
  -5.627702   -6.3079147  -8.771245   -8.351212  ]]
```



コードは本ドキュメントと同じレポジトリに提出する

https://github.com/toruuno/report_ml/blob/master/skl_nonregression.ipynb