機械学習・要点まとめ

第１章 線形回帰モデル

▣ 線形回帰モデル

以下の線形結合において、最小二乗法によりパラメータ w_i を推定

$$\hat{y} = w^T x + w_0 = \sum_{j=1}^{m} w_j x_j + w_0 \quad (m：パラメータ数、説明変数の数)$$

回帰係数 $\hat{w} = (X^{(train)T} X^{(train)})^{-1} X^{(train)T} y^{(train)}$

予測値 $\hat{y} = X(X^{(train)T} X^{(train)})^{-1} X^{(train)T} y^{(train)}$

$n \times (m+1)$ (n：データ数)

▣ 線形回帰モデルのハンズオン (regression.ipynb)

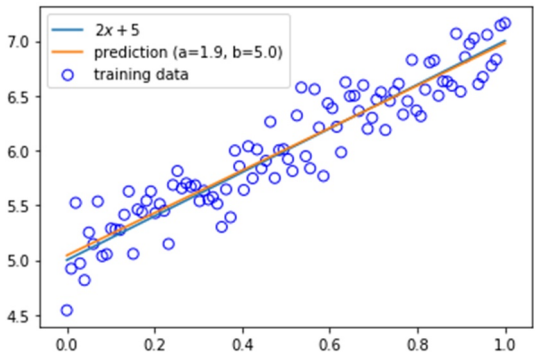
- numpy の場合： $\hat{w_1} = \mathrm{Cov}[x,y]/\mathrm{Var}[x]$, $\hat{w_0} = \mu_y - \hat{w_1}\mu_x$ (Cov：共分散、Var：分散、μ：平均) により、パラメータを推定。単回帰 (1 変数) と重回帰 (複数の変数) の場合で回帰を実装。
- sklearn の場合： linear_model の LinearRegression を用いる。fit 関数でパラメータを推定、predict 関数で予測。

```
#numpy 実装の回帰
def train(xs, ys):
    cov = np.cov(xs, ys, ddof=0)
    a = cov[0, 1] / cov[0, 0]
    b = np.mean(ys) - a * np.mean(xs)
    return cov, a, b

cov, a, b = train(xs, ys)
print("cov: {}".format(cov))
print("coef: {}".format(a))
print("intercept: {}".format(b))
```

```
cov: [[0.08501684 0.16464466] [0.16464466 0.36099457]]
coef: 1.9366124417390687
intercept: 5.042076659869998
```

[考察]

★ 変数間の共分散、回帰直線の係数と切片で回帰分析がされる。



第２章 非線形回帰モデル

▣ 非線形回帰モデル

- **基底展開法**： 回帰関数として、基底関数と呼ばれる既知の非線形関数とパラーメータベクトルの線形結合で表現。未知パラメータは、線形回帰モデルと同様に最小二乗法や最尤法により推定。

機械学習・要点まとめ

$\hat{y}_i = w^T \phi(x_i) + w_0 = \sum_{j=1}^{m} w_j \varphi_j(x_i) + w_0$

説明変数　$\boldsymbol{x_i} = (x_{i1}, x_{i2}, \cdots, x_{im}) \in R^m$

非線形関数ベクトル　$\Phi(x_i) = (\varphi_1(x_i), \varphi_2(x_i), \cdots, \varphi_k(x_i)) \in R^k$

非線形関数の計画行列　$\Phi^{(train)} = (\Phi(x_1), \Phi(x_2), \cdots, \Phi(x_n)) \in R^{n \times k}$

最尤法による予測　$\hat{y} = \Phi \left(\Phi^{(train)T} \Phi^{(train)} \right)^{-1} \Phi^{(train)T} y^{(train)}$

- よく使われる基底関数：

  多項式関数 $\varphi_j = x^j$　　ガウス型基底関数 $\varphi_j(x) = \exp\left\{ (x - \mu_j)^2 / 2h_j \right\}$

- **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。多くの基底関数を用意してしまうと過学習が起こるため、適切な基底関数を用意（交差検証 CV 等で選択）。
- **正則化法（罰則化法）**：「モデルの複雑さに伴って、その値が大きくなる正則化項（罰則項）を課した関数」を最小化。

$$S_\gamma = \underset{n \times k}{(y - \Phi w)^T (y - \Phi w)} + \gamma R(w) \quad (\gamma > 0)$$

基底関数の数（k）が増加するとパラメータが増加し、残差は減少（モデルが複雑化）　　モデルの複雑さに伴う罰則

▣ 非線形回帰モデルのハンズオン（nonlinear_regression.ipynb）

•numpy の場合：sin 曲線を 3 次までの多項式の線形結合式で、パラメータを近似させている。

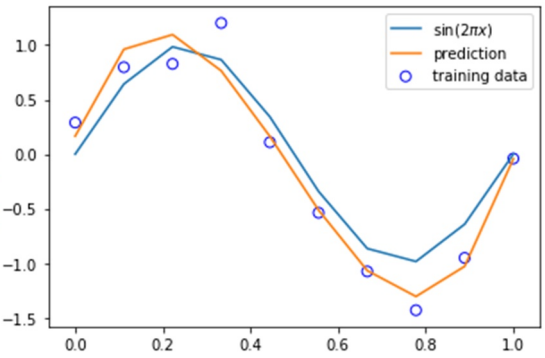
```
def polynomial_features(xs, degree=3):
    """多項式特徴ベクトルに変換
         X = [[1, x1, x1^2, x1^3], [1, x2, x2^2, x2^3],
                        ...                [1, xn, xn^2, xn^3]]"""
    X = np.ones((len(xs), degree+1))
    X_t = X.T #(10, 4)
    for i in range(1, degree+1):
        X_t[i] = X_t[i-1] * xs
    return X_t.T
```

```
Phi = polynomial_features(xs)     # 多項式の値の行列 Φ を計算
# 行列 Φ の逆行列と Φ の転置行列の積を計算
Phi_inv = np.dot(np.linalg.inv(np.dot(Phi.T, Phi)), Phi.T)
w = np.dot(Phi_inv, ys)     # 更に、目的変数 ys との積を求めて回帰係数wを計算
```

【考察】

★ 回帰の精度はあまり高くなく、回帰曲線が元の sin 曲線とずれを生じている。

★ 多項式を基底関数とする線形結合で sin 曲線を回帰している。回帰式の形が線形の場合と同じなので理解がしやすい。



- sklearn の場合：[0,1] の一様乱数に対する非線形な多項式関数（4 次式）の値にランダムなノイズを乗せた 2 次元データをプロット発生させた周期的に変動する 2 次元データに対して、Kernel

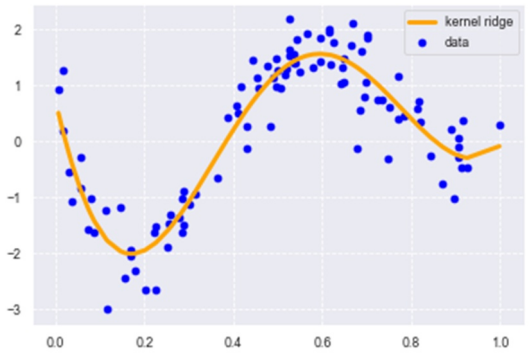
機械学習・要点まとめ

Ridge Regression は非線形なガウス型基底関数 rbf により、l2 制約付き最小二乗法で学習させて回帰分析し、非線形回帰曲線をプロット。

```
from sklearn.kernel_ridge import KernelRidge

clf = KernelRidge(alpha=0.0002, kernel='rbf')
clf.fit(data, target)
p_kridge = clf.predict(data)
```

[考察]

★比較的上手く回帰出来ているように見える。L2 制約付き最小二乗法による学習で過学習を防止している。



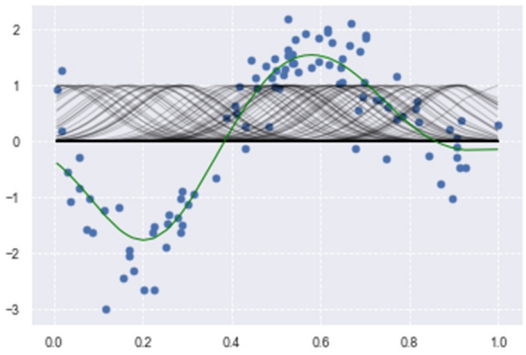
- sklearn の場合： KernelRidge を使わずに、RBF カーネルによる l2 制約付き最小二乗方法で学習させて回帰分析し、非線形回帰曲線と基底関数をプロット。

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

kx = rbf_kernel(X=data, Y=data, gamma=50)
#KX = rbf_kernel(X, x)

#clf = LinearRegression()
clf = Ridge(alpha=30)
clf.fit(kx, target)
p_ridge = clf.predict(kx)

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

```
0.8442145484419565
```

[考察]

★決定係数は 0.844 で、線形モデルから大幅に改善している。L2 制約付き最小二乗法による学習で過学習を防止している。



- sklearn の場合： LinearRegression で線形回帰と、2 次から 10 次までの多項式による非線形回帰で学習させて、予測値をプロット。

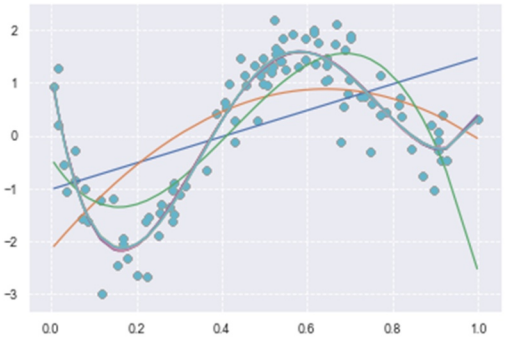
```
#PolynomialFeatures(degree=1)
deg = [1,2,3,4,5,6,7,8,9,10]
```

機械学習・要点まとめ

```
for d in deg:
    regr = Pipeline([
        ('poly', PolynomialFeatures(degree=d)),
        ('linear', LinearRegression())
    ])
    regr.fit(data, target)
    # make predictions
    p_poly = regr.predict(data)
```

【考察】

★3 次多項式曲線が最も低バイアスで低バリアンスに見える。



- sklearn の場合： サポートベクター回帰（SVR）のガウシアンカーネルで回帰分析し、予測曲線をプロット。

```
from sklearn import preprocessing, linear_model, svm
# SVR-rbf
clf_svr = svm.SVR(kernel='rbf', C=1e3, gamma=0.1, epsilon=0.1)
clf_svr.fit(data, target)
y_rbf = clf_svr.fit(data, target).predict(data)
```

【考察】

★本条件では、回帰の性能はあまり高くないようである。

※ SVR について参照

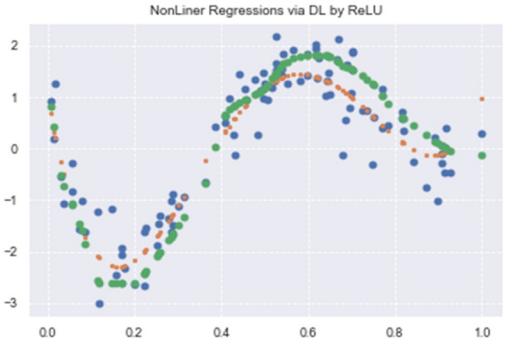
https://qiita.com/koshian2/items/baa51826147c3d538652

- sklearn の場合： Keras と Tensorflow により、深層学習（全結合の 10 層、活性化関数 ReLu）で予測し、ノイズ付与前の元のデータの真の関数値と比較してプロット。

```
Epoch 100/100 85/90 [============================>..] - ETA: 0s - loss: 0.2151
Epoch 00100: saving model to ./out/checkpoints/weights.100-0.34.hdf5 90/90
[==============================] - 3s 28ms/sample - loss: 0.2459 - val_loss: 0.3358
```

【考察】

★全結合 10 層（ReLU 関数）の深層モデルで、予測精度が高いように見える。



機械学習・要点まとめ

第３章 ロジスティック回帰モデル

▣ ロジスティック回帰モデル

- **分類問題（クラス分類）**： **対数オッズ** log（p/1-p）を線形回帰で予測し、それを**正規化**して確率として出力を得ることで、結果としてクラス分類を実現。対数オッズの逆関数が**シグモイド関数**（活性化関数） ⇒ 活性化関数（＝**ロジット変換**）により、クラス i に属する確率 p が求まる。
- 目的関数は、**対数尤度**（回帰モデル構築時に最大化したい**尤度関数**の**対数**をとったもの、その負値が**交差エントロピー**関数）。**交差エントロピー**関数の最小化は、**対数尤度**の最大化と同値。

説明変数　$x = (x_1, x_2, \cdots, x_m)^T \in R^m$

目的変数　$y \in \{0, 1\}$

パラメータ　$w = (w_1, w_2, \cdots, w_m)^T \in R^m$

線形結合　$z = w^T x + w_0 = \sum_{j=1}^{m} w_j x_j + w_0$

出力　$\hat{y} = P(Y=1 \mid x) = \sigma(z = \sum_{j=1}^{m} w_j x_j + w_0) = \frac{1}{1+exp(-az)}$　σ：シグモイド関数

シグモイド関数の出力を Y=1 になる確率に対応（Y は $\hat{y} \geqq 0.5$ なら 1、$\hat{y} < 0.5$ なら 0 と予測）

- **ロジスティック回帰モデルの最尤推定**： 対数尤度関数 E を最大とするパラメータを探索

y_1～y_n のデータが得られた際の**尤度関数** $L(\mathbf{w})$（確率 P はベルヌーイ分布に従うものとする）：

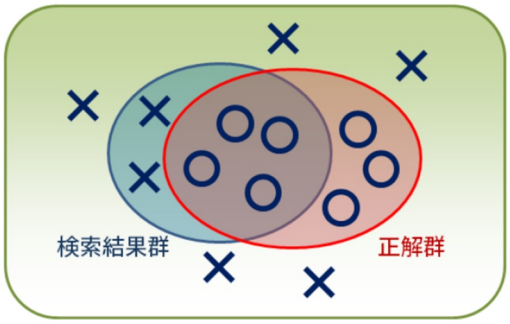
$L(\mathbf{w}) = P(y_1, y_2, \cdots, y_n \mid w_0, w_1, \cdots, w_m) = \prod_{i=1}^{n} p_i^{y_i}(1-p_i)^{1-y_i} = \prod_{i=1}^{n} \sigma(w^T x_i)^{y_i}(1-\sigma(w^T x_i))^{1-y_i}$

交差エントロピー　$E(w) = -\log L(w) = \sum_{i=1}^{n}\{y_i \log p_i + (1-y_i)\log(1-p_i)\}$

$\frac{\partial E(w)}{\partial w} = -\sum_{i=1}^{n}(y_i - p_i)\, x_i$　より、

勾配降下法　$w^{(k+1)} = w^{(k)} - \eta\frac{\partial E(w)}{\partial w} = w^{(k)} + \eta\sum_{i=1}^{n}(y_i - p_i)\, x_i$

▣ 適合率と再現率

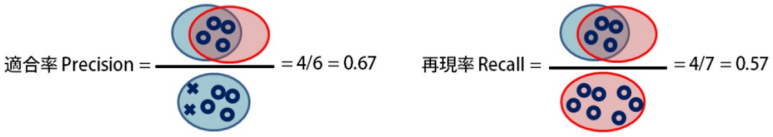


混同行列：真陽性（TP）、偽陽性（FP）、偽陰性（FN）、真陰性（TN）

正解率 accuracy＝（TP＋TN）／全体＝9／14＝0.64

適合率 precision＝TP／（TP＋FP）　⇔　偽陽性（FP）

再現率 recall＝TP／（TP＋FN）　⇔　偽陰性（FN）



▣ ロジスティック回帰モデルのハンズオン（logistic_regression.ipynb）

- numpy の場合： 正規分布に従う乱数に基づいて中心をずらしてそれぞれ点群を発生させて訓練データを生成し、中心位置に合わせて 2 クラス {0, 1} にロジスティック回帰で分類。

```
# 【学習】

# バイアス w0 に対する入力項１を１軸方向に前側に付け加える
```

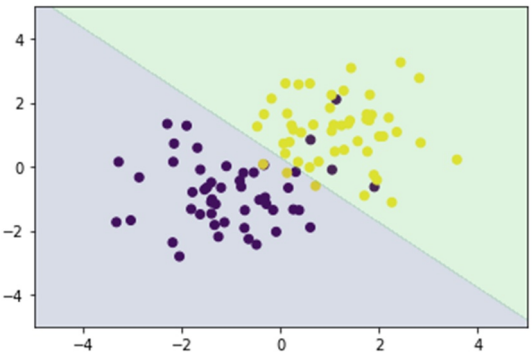
機械学習・要点まとめ

```
def add_one(x):
    return np.concatenate([np.ones(len(x))[:, None], x], axis=1)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def sgd(X_train, max_iter, eta):
    w = np.zeros(X_train.shape[1])
    for _ in range(max_iter):
        w_prev = np.copy(w)
        sigma = sigmoid(np.dot(X_train, w))
        grad = np.dot(X_train.T, (sigma - y_train))     # 微分の計算
        w -= eta * grad
        if np.allclose(w, w_prev):
            return w
    return w

X_train = add_one(x_train)
max_iter=100
eta = 0.01
w = sgd(X_train, max_iter, eta)
```

```
# 【予測 〔結果をプロット〕】

xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T
# 識別等高線を描くために、メッシュの各点が〔0,1〕のどちらのクラスに分類されるかを判定
X_test = add_one(xx)
proba = sigmoid(np.dot(X_test, w))
y_pred = (proba > 0.5).astype(np.int)

# 入力データの各点をクラス別に色分けしてプロットすると共に、クラスの識別等高線をプロット
plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
plt.contourf(xx0, xx1, proba.reshape(100, 100), alpha=0.2,
    levels=np.linspace(0, 1, 3))
```

【考察】

★上手く線形回帰で分類されているが、幾つかの点は間違っている。線形では難しい対象のようである。



- sklearn の場合： titanic データで生死を分けた多くの説明変数から、その原因をロジスティック回帰で分類。不要なデータの削除・欠損値の補間、運賃〔1 変数〕から生死を判別。

```
from sklearn.linear_model import LogisticRegression
# ロジスティック回帰モデルのインスタンスを生成
model=LogisticRegression()
# 学習
```

機械学習・要点まとめ

```
model.fit(data1, label1)
# 運賃＝$61 の生死を予測
model.predict([[61]])   # Fare = $61 ---> not survived
```

```
array([0], dtype=int64)
```

```
# 運賃＝$62 の生死の確率を予測
model.predict_proba([[62]])   # Fare = $62 ---> survived
```

```
array([[0.49978123, 0.50021877]])
```

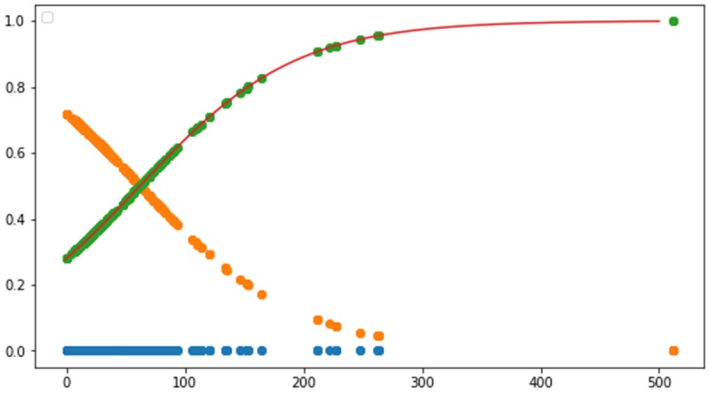
```
X_test_value = model.decision_function(data1)
# # 決定関数値（絶対値が大きいほど識別境界から離れている）
# X_test_value = model.decision_function(X_test)
# # 決定関数値をシグモイド関数で確率に変換
# X_test_prob = normal_sigmoid(X_test_value)

# 重み係数 w1 とバイアス w0 をプリント出力
print (model.intercept_)
print (model.coef_)
```

```
[-0.94131796]
[[0.01519666]]
```

```
# 各データ（運賃）に対する生死の確率と運賃に対するシグモイド関数の値をプロット

w_0 = model.intercept_[0]
w_1 = model.coef_[0,0]

def sigmoid(x):
    return 1 / (1+np.exp(-(w_1*x+w_0)))

x_range = np.linspace(-1, 500, 3000)
plt.figure(figsize=(9,5))
plt.legend(loc=2)
plt.plot(data1,np.zeros(len(data1)), 'o')
plt.plot(data1, model.predict_proba(data1), 'o')
plt.plot(x_range, sigmoid(x_range), '-')
```

【考察】

★運賃が$62 以上で survived、$61 以下で not survived で、そこが識別境界（右図の２つの曲線が交差している点、確率 0.5 に相当）となった。



- sklearn の場合： 2 変数から生死を判別。性別 Sex を整数にマッピング（Gender）し、更に、Gender と社会的地位（Pclass）を 1 つの変数 Pclass_Gender を作成。その上で年齢 AgeFill と Pclass_Gender の 2 変数と生死の関係とその境界線をプロット。

機械学習・要点まとめ

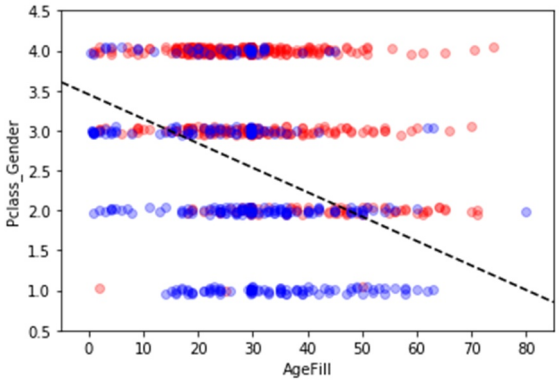
```
#AgeFill の欠損値を埋めたので
#titanic_df = titanic_df.drop(['Age'], axis=1)
# 'Sex'の'female': 0, 'male': 1とした、新たな説明変数'Gender'を生成
titanic_df['Gender'] = titanic_df['Sex'].map({'female': 0, 'male': 1}).astype(int)
# 'Pclass'と'Gender'を加算した新たな説明変数'Pclass_Gender'を生成（特徴エンジニアリング）
titanic_df['Pclass_Gender'] = titanic_df['Pclass'] + titanic_df['Gender']
# 説明変数から'Pclass', 'Sex', 'Gender','Age'を削除
titanic_df = titanic_df.drop(['Pclass', 'Sex', 'Gender','Age'], axis=1)
titanic_df.head()
```

	Survived	SibSp	Parch	Fare	Embarked	AgeFill	Pclass_Gender
0	0	1	0	7.2500	S	22.0	4
1	1	1	0	71.283	C	38.0	1
2	1	0	0	7.9250	S	26.0	3
3	1	1	0	53.1000	S	35.0	1
4	0	0	0	8.0500	S	35.0	4

```
#生死フラグのみのリストを作成
label2 =  titanic_df.loc[:,["Survived"]].values
# ロジスティック回帰モデルのインスタンスを生成
model2 = LogisticRegression()
# 学習
model2.fit(data2, label2)
# Age=10, Pclass_Gender=1 の生死を予測
model2.predict([[10,1]])    # Age=10, Pclass_Gender=1 ---> survived
array([1], dtype=int64)
# Age=10, Pclass_Gender=1 の生き残る確率を予測
model2.predict_proba([[10,1]])
array([[0.03754749, 0.96245251]])

# 'AgeFill'を横軸、'Pclass_Gender'を縦軸にし、全データの生死をプロット
#  ⇒ 'AgeFill'が小さく（年齢が若い）、'Pclass_Gender'が小さい（地位が高く女性）
#      ほど生き残る確率が高そう！
#  ⇒ 生死の分かれ目の境界のラインを引く
```

【考察】

★右図で境界線の右上部分が not survived （赤点）、左下部分が survived （青点）が多くなっていることがわかる。



- sklearn の場合： 1）運賃 Fare の 1 変数だけ、2）年齢 AgeFill と Pclass_Gender の 2 変数のそれぞれと生死の関係に対する混同行列を計算、それをヒートマップにしてプロット。

```
from sklearn.model_selection import train_test_split
# 'Fare'だけのデータを、訓練データ（80%）とテストデータ（20%）に分割･･･データ1
traindata1, testdata1, trainlabel1, testlabel1 = train_test_split
                                        (data1, label1, test_size=0.2)
# 'AgeFill'と'Pclass_Gender'だけのデータを、訓練データ（80%）とテストデータ（20%）に分割･･･データ2
```

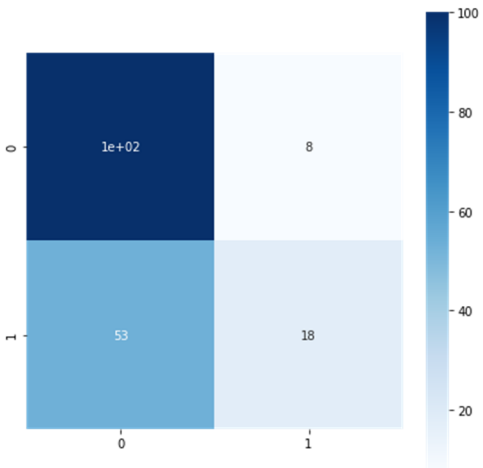
機械学習・要点まとめ

```
traindata2, testdata2, trainlabel2, testlabel2 = train_test_split
                                    (data2, label2, test_size=0.2)
# ロジスティック回帰モデルの各インスタンスを生成
eval_model1=LogisticRegression()
eval_model2=LogisticRegression()
# 各モデルでそれぞれ学習
predictor_eval1=eval_model1.fit(traindata1, trainlabel1).predict(testdata1)
predictor_eval2=eval_model2.fit(traindata2, trainlabel2).predict(testdata2)
# データ1（Fare のみ）に対するモデルを評価 ⇒ 訓練データの正解率＝0.671
eval_model1.score(traindata1, trainlabel1)
# データ1（Fare のみ）に対するモデルを評価 ⇒ テストデータの正解率＝0.648
eval_model1.score(testdata1,testlabel1)
# データ2（'AgeFill'と'Pclass_Gender'）に対するモデルを評価 ⇒ 訓練データの正解率＝0.774
eval_model2.score(traindata2, trainlabel2)
# データ2（'AgeFill'と'Pclass_Gender'）に対するモデルを評価 ⇒ テストデータの正解率＝0.760
eval_model2.score(testdata2,testlabel2)
# sklearn の metrics により、各データのテストデータに対する識別レポートを出力
from sklearn import metrics
print(metrics.classification_report(testlabel1, predictor_eval1))
print(metrics.classification_report(testlabel2, predictor_eval2))
```

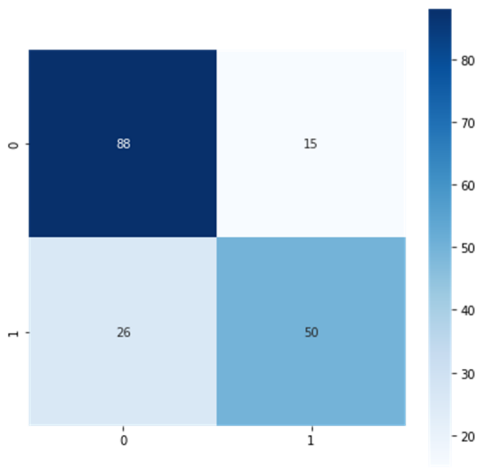
```
              precision recall    f1-score   support
       0      0.65      0.93      0.77       108
       1      0.69      0.25      0.37        71
accuracy                         0.66       179
macro avg     0.67      0.59      0.57       179
weighted avg  0.67      0.66      0.61       179

              precision recall   f1-score   support
       0      0.77      0.85      0.81       103
       1      0.77      0.66      0.71        76
accuracy                         0.77       179
macro avg     0.77      0.76      0.76       179
weighted avg  0.77      0.77      0.77       179
```

```
# sklearn の metrics により、各データのテストデータに対する混同行列を計算
from sklearn.metrics import confusion_matrix
confusion_matrix1=confusion_matrix(testlabel1, predictor_eval1)
confusion_matrix2=confusion_matrix(testlabel2, predictor_eval2)
```

Fare の混同行列



AgeFill と Pclass_Gender の混同行列



機械学習・要点まとめ

第４章 主成分分析

▣ 主成分分析

- **主成分分析**（PCA；Principal Component Analysis）： 教師なし学習で、多変量データの持つ構造をより少数個の指標に圧縮しつつそれに伴う情報の損失はなるべく小さくするため、変換後の分散を最大にするように探索する。

  学習データ　$\boldsymbol{x_i} = (x_{i1},\ x_{i2}, \cdots,\ x_{im}) \in \mathrm{R}^m$

  平均（ベクトル）　$\bar{x} = \frac{1}{n}\sum_{i=1}^{n} x_i$

  データ行列　$\bar{X} = (x_1 - \bar{x},\ \cdots,\ x_n - \bar{x})^{\mathrm{T}}$

  分散共分散行列　$\Sigma = \mathrm{Var}(\bar{X}) = \frac{1}{n}\bar{X}^{\mathrm{T}}\bar{X}$

  線形変換後のベクトル　$s_j = (s_{1j},\ s_{2j}, \cdots, s_{nj})^{\mathrm{T}} = \bar{X}\boldsymbol{a_j}$　（$\bar{X}$: n×m, $\boldsymbol{a_j}$:m×1）　j は射影後のインデックス

  $$\mathrm{Var}(s_j) = \frac{1}{n}{s_j}^{\mathrm{T}}s_j = \frac{1}{n}(\bar{X}\boldsymbol{a_j})^{\mathrm{T}}(\bar{X}\boldsymbol{a_j}) = \frac{1}{n}\boldsymbol{a_j}^{\mathrm{T}}\bar{X}^{\mathrm{T}}\bar{X}\boldsymbol{a_j} = \boldsymbol{a_j}^{\mathrm{T}}\mathrm{Var}(\bar{X})\boldsymbol{a_j}$$

  線形変換後の変数の分散が最大となる射影軸は、以下のラグランジュ関数を最大にする：

  $$\mathrm{E}(\boldsymbol{a_j}) = \boldsymbol{a_j}^{\mathrm{T}}\,\mathrm{Var}(\bar{X})\,\boldsymbol{a_j} - \lambda(\boldsymbol{a_j}^{\mathrm{T}}\boldsymbol{a_j} - 1)$$

  $$\frac{\partial E(\boldsymbol{a_j})}{\partial \boldsymbol{a_j}} = 2\mathrm{Var}(\bar{X})\boldsymbol{a_j} - 2\lambda\boldsymbol{a_j} = 0 \ \Rightarrow\ \mathrm{Var}(\bar{X})\boldsymbol{a_j} = \lambda\boldsymbol{a_j}$$

  $$\mathrm{Var}(s_j) = \boldsymbol{a_j}^{\mathrm{T}}\mathrm{Var}(\bar{X})\boldsymbol{a_j} = \lambda_{\mathrm{j}}\boldsymbol{a_j}^{\mathrm{T}}\boldsymbol{a_j} = \lambda_{\mathrm{j}}$$　射影先の分散は固有値と一致

- **寄与率**： 各成分の重要度が測れる

  第1～元次元分の主成分の分散は固有値と等しく、元のデータの分散と一致

  元データの総分散　$\mathrm{V_{total}} = \sum_{i=1}^{m}\lambda_{\mathrm{i}}$

  寄与率　c_k：　第 k 主成分の分散の全分散に対する割合（第 k 主成分が持つ情報量の割合）

  累積寄与率　r_k：　第1～k 主成分まで圧縮した際の情報損失量の割合

  $$c_k = \frac{\lambda\mathrm{k}}{\sum_{i=1}^{m}\lambda\mathrm{i}} \qquad r_k = \frac{\sum_{j=1}^{k}\lambda\mathrm{j}}{\sum_{i=1}^{m}\lambda\mathrm{i}}$$

- オートエンコーダと主成分分析の関係･･･オートエンコーダの中間層の活性化関数を**恒等関数**にすると、オートエンコーダと主成分分析は等価な数式で表すことができ同じ結果を返す。

▣ 主成分分析のハンズオン（pca.ipynb）

- numpy の場合：　2 変量正規分布に従う乱数により点群を発生（2 次元の期待値 mean と分散共分散行列 cov が入力）させて、主成分分析を行う。

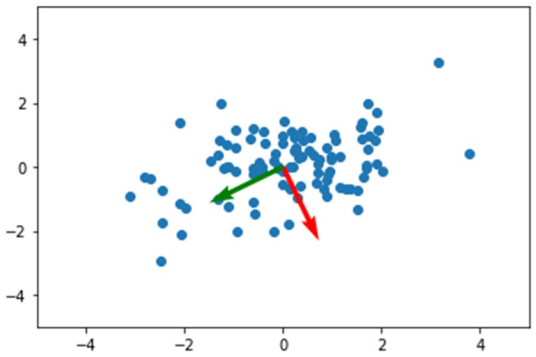
```
# 【学習】

n_components=2

# データ（点群）の平均と不偏共分散行列を出力
def get_moments(X):
    mean = X.mean(axis=0)   # 平均
    stan_cov = np.dot((X - mean).T, X - mean) / (len(X) - 1)  # 不偏共分散行列
    return mean, stan_cov
```

機械学習・要点まとめ

```
# 固有値の大きい順に n_components 個の固有ベクトル成分を返す
def get_components(eigenvectors, n_components):
#     W = eigenvectors[:, -n_components:]
#     return W.T[::-1]
    W = eigenvectors[:, ::-1][:, :n_components]
    return W.T

# 2つの主成分に対す固有ベクトルをプロット
def plt_result(X, first, second):
    plt.scatter(X[:, 0], X[:, 1])
    plt.xlim(-5, 5)
    plt.ylim(-5, 5)
    # 第1主成分
    plt.quiver(0, 0, first[0], first[1], width=0.01, scale=6, color='red')
    # 第2主成分
    plt.quiver(0, 0, second[0], second[1], width=0.01, scale=6, color='green')

# 分散共分散行列を標準化
meean, stan_cov = get_moments(X)
# 固有値分解による固有値と固有ベクトルを計算
eigenvalues, eigenvectors = np.linalg.eigh(stan_cov)
# 固有値の大きい順に固有ベクトル成分を返す
components = get_components(eigenvectors, n_components)
# PCA の主成分(2 成分)の固有ベクトルをプロット
plt_result(X, eigenvectors[0, :], eigenvectors[1, :])
```

[考察]

★最も分散が大きくなる方向に第 1 主成分が取られ、それと垂直方向に第 2 主成分が取られていることが判る。



- numpy の場合： 上記で求めた主成分の方向にデータ全体を変換(射影)する。

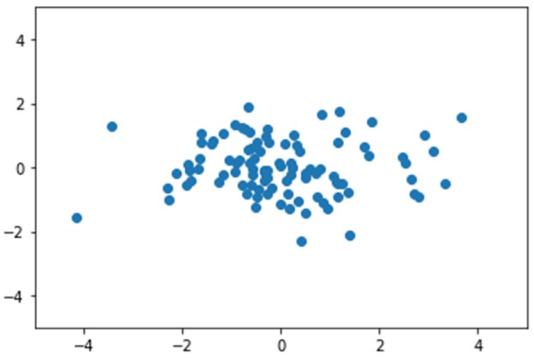
```
# PCA の2つの主成分に対する固有ベクトルの方向にデータを変換
#def transform_by_pca(X, pca): → 5/11/2020 コーディング修正
def transform_by_pca(X, components):
    mean = X.mean(axis=0)
    return np.dot(X-mean, components)
# PCA の2つの主成分に対する固有ベクトルの方向にデータを変換し、プロット
Z = transform_by_pca(X, components.T)
plt.scatter(Z[:, 0], Z[:, 1])
plt.xlim(-5, 5)
plt.ylim(-5, 5)
```

機械学習・要点まとめ

【考察】

★確かに、主成分の方向が、新たな X 軸と Y 軸になっていることがわかる。

★元のデータを m 次元に変換（射影）するときは、行列 W を $W=[w_1,w_2,\cdots,w_m]$とし、データ点 x を $z=W^Tx$ によって変換(射影)する。よって、データ X に対しては $Z=X^TW$ によって変換する。



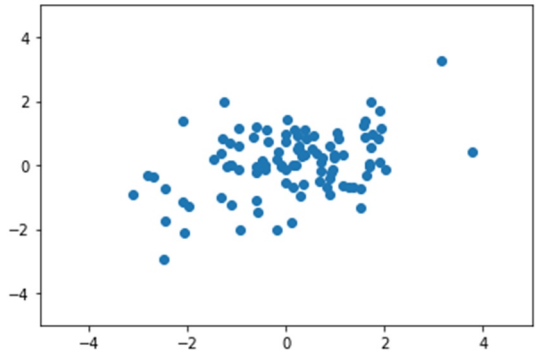
- numpy の場合： 元の座標系に逆変換する。

```
mean = X.mean(axis=0)
#X_ = np.dot(Z, components.T) + mean → 5/11/2020 コーディング修正
X_ = np.dot(Z, components) + mean
```

【考察】

★確かに、元の座標系に戻り、点群が元と同じ位置にあることがわかる。

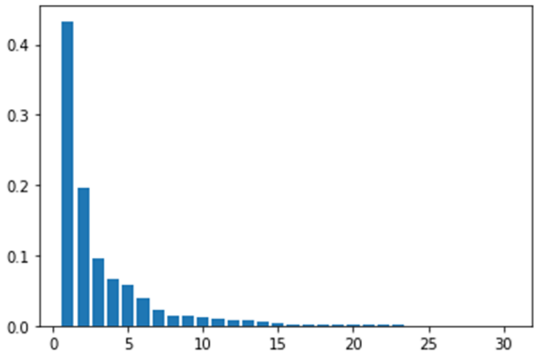
★射影されたデータ点 z を元のデータ空間へ逆変換するときは$\bar{x}=(W^T)^{-1}z=Wz$ によって変換する。よって、射影されたデータ Z に対しては$\bar{X}$ $=ZW^T$によって変換する。

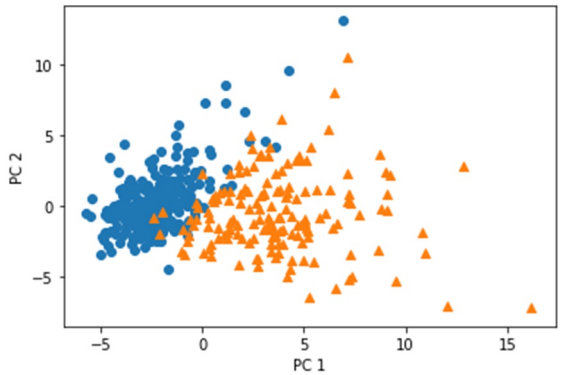


- sklearn の場合： がん診断データを分析（ロジスティック回帰、PCA）する。

```
# 良性 B＝0 / 悪性 M＝1 として、目的変数を抽出
y = cancer_df.diagnosis.apply(lambda d: 1 if d == 'M' else 0)
# 説明変数'radius_mean'の抽出
X = cancer_df.loc[:, 'radius_mean':]
# 学習用とテスト用でデータを分離
X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=0)

# 標準化
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# ロジスティック回帰で学習
logistic = LogisticRegressionCV(cv=10, random_state=0)
logistic.fit(X_train_scaled, y_train)

# 検証： Train score: 0.988， Test score: 0.972，★検証スコア 97%で分類
'''
   [混同行列]：          Predicted
                        Negative  Positive
    Actual Negative     TN 89%    FP  1%
           Positive     FN  3%    TP 50%
'''
# sklearn の主成分分析で説明変数を 32 から 30 に削減、各主成分の寄与率を棒グラフで表示
pca = PCA(n_components=30)
pca.fit(X_train_scaled)
```

```
plt.bar([n for n in range(1, len(pca.explained_variance_ratio_)+1)], pca.e
xplained_variance_ratio_)

# sklearn の主成分分析で説明変数（次元数）を 2 まで圧縮
pca = PCA(n_components=2)
X_train_pca = pca.fit_transform(X_train_scaled)

# 寄与率
print('explained variance ratio: {}'.format(pca.explained_variance_ratio_))
# explained variance ratio: [ 0.43315126  0.19586506]

# 散布図にプロット
--------------------
plt.scatter(x=b[0], y=b[1], marker='o') # 良性は○でマーク
plt.scatter(x=m[0], y=m[1], marker='^') # 悪性は△でマーク
plt.xlabel('PC 1') # 第 1 主成分を x 軸
plt.ylabel('PC 2') # 第 2 主成分を y 軸

X_train_pca shape: (426, 2)
explained variance ratio: [0.43315126 0.19586506]
```





[考察]

★ 左図が寄与率のグラフで、右図は上位 2 成分の主成分でプロットしたもの。2 成分までのそれぞれの寄与率は 0.43315126, 0.19586506（2 主成分の累積寄与率は約 63%）で、青色（〇）が良性、橙色（△）が悪性を示す。良性と悪性がかなりよく分離されているがその境界は曖昧で、線形回帰では完全に分離するのは難しい。

第 5 章 k 近傍法

▣ k 近傍法

- 分類問題で適用： 最近傍のデータを k 個取ってきて、それらが最も多く所属するクラスに識別
  ⇒ k（ハイパーパラメータ）により結果が変わる、k を大きくすると決定境界は滑らかになる

▣ k 近傍法のハンズオン（np_knn.ipynb）

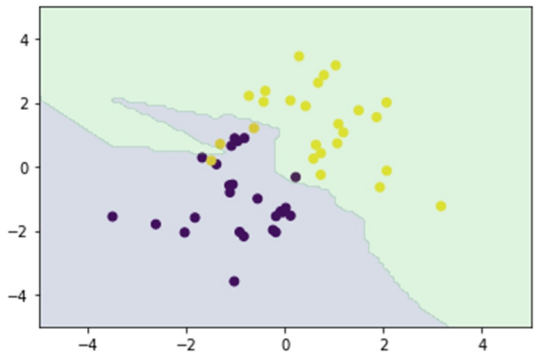
- numpy の場合： 正規分布に従う乱数に基づいて中心をずらしてそれぞれ点群を発生させ、中心位置に合わせて k 近傍法で 2 クラス {0, 1} に分類する。

```
# 【予測】
def distance(x1, x2):
```

```
    return np.sum((x1 - x2)**2, axis=1)

# テストデータに対し、最近傍の n 個の訓練データの点を見つけ、その最頻値のクラスを割り当てる
def knc_predict(n_neighbors, x_train, y_train, X_test):
    y_pred = np.empty(len(X_test), dtype=y_train.dtype)
    for i, x in enumerate(X_test):
        distances = distance(x, X_train)
        nearest_index = distances.argsort()[:n_neighbors]
                # stats.mode:最頻値とその個数を求める
        mode, _ = stats.mode(y_train[nearest_index])
        y_pred[i] = mode
    return y_pred

# 訓練データのクラスで色分けしたメッシュ状の各点のクラスに合わせ等高線（クラス境界線）をプロット
def plt_resut(x_train, y_train, y_pred):
    xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
    xx = np.array([xx0, xx1]).reshape(2, -1).T
    plt.scatter(x_train[:, 0], x_train[:, 1], c=y_train)
    plt.contourf(xx0, xx1, y_pred.reshape(100, 100).astype
                (dtype=np.float), alpha=0.2, levels=np.linspace(0, 1, 3))
# メッシュ各点がどちらのクラスに分類されるか判定し、クラスに合わせて等高線（クラス境界線）をプロット
n_neighbors = 3
xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
X_test = np.array([xx0, xx1]).reshape(2, -1).T
y_pred = knc_predict(n_neighbors, X_train, ys_train, X_test)
plt_resut(X_train, ys_train, y_pred)
```

【考察】

★ 予測するデータ点との、距離が最も近い k 個の、訓練データのラベルの最頻値を割り当てる。従って、各変数のスケールは合わせる必要があると思われる。



第６章 k-means（k 平均クラスタリング）

▣ k-means

- 教師なし学習、分類問題で適用： 最近傍のデータを k 個取ってきて、それらが最も多く所属するクラスに識別 ⇒ k（ハイパーパラメータ）により結果が変わる、k を大きくすると決定境界は滑らかになる

【学習】

①各クラスタ中心の初期値を設定する。

②各データ点に対して、各クラスタ中心との距離を計算し、最も距離が近いクラスタを割り当てる

③各クラスタの平均ベクトル（中心）を計算する

④収束するまで②・③の処理を繰り返す

▣ k-means のハンズオン（kmeans.ipynb）

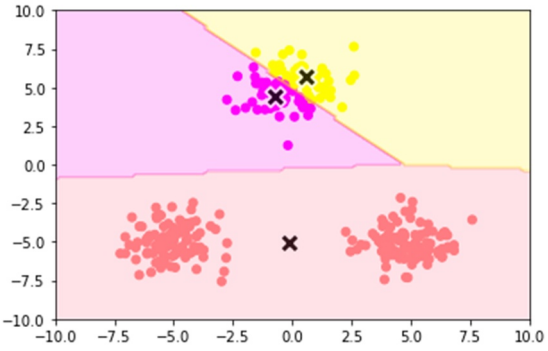
機械学習・要点まとめ

- numpy の場合： 離れた 3 点を中心に、それぞれ 100 個ずつの点群を正規分布に従う乱数によって生成させ、それに対して k 平均法でクラスタリングを行いプロット。

```
# 2 点間の距離を計算
def distance(x1, x2):
    return np.sum((x1 - x2)**2, axis=1)

n_clusters = 3
iter_max = 100

# 3つのクラスタの中心をランダムに選択して初期化
centers = X_train[np.random.choice(len(X_train), n_clusters, replace=False)]

for _ in range(iter_max):
    prev_centers = np.copy(centers)
    D = np.zeros((len(X_train), n_clusters))
    # 各データ点に対して、各クラスタ中心との距離を計算
    for i, x in enumerate(X_train):
        D[i] = distance(x, centers)
    # 各データ点に、最も距離が近いクラスタを割り当てる
    cluster_index = np.argmin(D, axis=1)
    for k in range(n_clusters):    # 各クラスタの中心を再計算
        # 各クラスタ毎に、各データ点がクラスタに属した場合、インデックスに True をセット
        index_k = cluster_index == k
# クラスタの中心を再計算（インデックスが True の場合だけ平均の対象）
        centers[k] = np.mean(X_train[index_k], axis=0)
    # 収束判定（中心の位置が変化しなくなった場合に終了）
    if np.allclose(prev_centers, centers):
        break
```

【考察】

★ クラスタの中心の初期値が近い場合、K-means では上手くクラスタリングができないことがあるようだ。

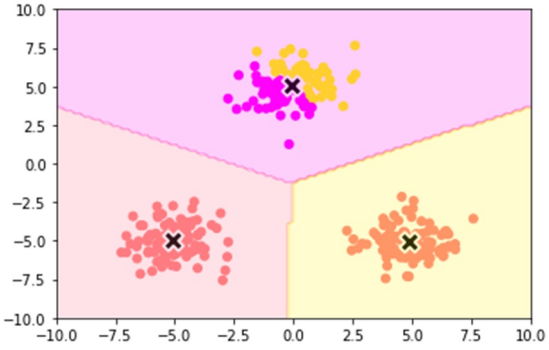


- sklearn の場合： 同じデータに対し、sklearn を用いて k-means 法でクラスタリングしプロット。

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=3, random_state=0).fit(X_train)
```

【考察】

★ sklearn.cluster の KMeans を用いた場合、適切な中心の初期値がセットされているらしく、結果、上手くクラスタリングができている。



- sklearn の場合： 同じデータに対し、sklearn を用いて k-means 法でクラスタリングしプロット。

```
import pandas as pd
from sklearn import cluster, preprocessing, datasets
from sklearn.cluster import KMeans

# ワインのターゲット名は、'class_0', 'class_1', 'class_2' の３種類
 ------------------------------------
model = KMeans(n_clusters=3)
labels = model.fit_predict(X)      # X は wine データ
df = pd.DataFrame({'labels': labels})

def species_label(theta):
    if theta == 0:
        return wine.target_names[0]     # 'class_0'
    if theta == 1:
        return wine.target_names[1]     # 'class_1'
    if theta == 2:
        return wine.target_names[2]     # 'class_2'
df['species'] = [species_label(theta) for theta in wine.target]
# pd.crosstab()：クロス集計  第一引数 index に結果の行見出しとなる列、
# 第二引数 columns に結果の列見出しとなる列を指定
pd.crosstab(df['labels'], df['species'])
```

species	class_0	class_1	class_2
labels			
0	0	50	19
1	46	1	0
2	13	20	29

【考察】

★ k-means (k=3) のクラスタリングの結果、グループ labels 0 は class_1 (50/69) が最大で、グループ labels1 は class_0 (46/47) が最大。ただし、グループ labels 2 は class_2 (29/62) が多いが、他の class も多く混じっていて、labels 2 は上手くクラスタリングできていない。

第７章 サポートベクターマシン

▣ サポートベクターマシン (線形・非線形共通)

【学習】

特徴空間上で線形なモデル$y(\boldsymbol{x})=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})+b$ を用い、その正負によって２値分類を行うことを考える。サポートベクトターマシンではマージンの最大化を行うが、それは以下の最適化問題を解くことと同じ。ただし、訓練データを$X=[\boldsymbol{x}_1,\boldsymbol{x}_2,\ldots,\boldsymbol{x}_n]^{\mathrm{T}}$, $\boldsymbol{t}=[t_1,t_2,\ldots,t_n]^{\mathrm{T}}$ $(t_i=\{-1,+1\})$ とする。

$$\min_{\boldsymbol{w},b} \ \frac{1}{2}\|\boldsymbol{w}\|^2 \ , \quad \text{subject to} \quad t_i(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_i)+b)\geq 1 \ \ (i=1,2,\cdots,n)$$

＜線形の場合は、$\boldsymbol{\phi}(\boldsymbol{x}_i)=\boldsymbol{x}_i$ で、以降、同様に読み替えること＞

ラグランジュ乗数法を使うと、上の最適化問題はラグランジュ乗数$\boldsymbol{a}$ (≥0)を用いて、以下の目的関数を最小化する問題となる。

機械学習・要点まとめ

$L(\boldsymbol{w}, b, \boldsymbol{a})=\frac{1}{2}\|\boldsymbol{w}\|^2-\sum_{i=1}^{n} a_i\, t_i(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_i)+b-1)$ … (1)

目的関数が最小となるのは、$\boldsymbol{w}$,bに関して偏微分した値が 0 となるときなので、

$\frac{\partial L}{\partial \boldsymbol{w}}=\boldsymbol{w}-\sum_{i=1}^{n} a_i\, t_i\boldsymbol{\phi}(\boldsymbol{x}_i)=0$ ， $\frac{\partial L}{\partial b}=\sum_{i=1}^{n} a_i\, t_i=0$

これを式(1) に代入することで、最適化問題は結局以下の目的関数の最大化となる。

$\tilde{L}(\boldsymbol{a})=\sum_{i=1}^{n} a_i-\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j t_i t_j\boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_j)=\boldsymbol{a}^{\mathrm{T}}\mathbf{1}-\frac{1}{2}\boldsymbol{a}^{\mathrm{T}}H\boldsymbol{a}$ （1：全成分が 1 のベクトル）

ただし、行列Hのi行j列成分は$H_{ij}=t_i t_j\boldsymbol{\phi}(\boldsymbol{x}i)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}j)=t_i t_j k(\boldsymbol{x}_i,\boldsymbol{x}_j)$である。

また制約条件は、$\boldsymbol{a}^{\mathrm{T}}\boldsymbol{t}=0$ $(\frac{1}{2}\|\boldsymbol{a}^T\boldsymbol{t}\|^2=0)$である。

この最適化問題を最急降下法で解く。目的関数と制約条件を$\boldsymbol{a}$で微分すると、

$\frac{d\tilde{L}}{d\boldsymbol{a}}=\mathbf{1}-H\boldsymbol{a}$ ， $\frac{d}{d\boldsymbol{a}}(\frac{1}{2}\|\boldsymbol{a}^T\boldsymbol{t}\|^2)=(\boldsymbol{a}^T\boldsymbol{t})\boldsymbol{t}$

従って、$\boldsymbol{a}$を以下の二式で更新する。

$\boldsymbol{a}\leftarrow\boldsymbol{a}+\eta_1(\mathbf{1}-H\boldsymbol{a})$ ， $\boldsymbol{a}\leftarrow\boldsymbol{a}-\eta_2(\boldsymbol{a}^{\mathrm{T}}\boldsymbol{t})\boldsymbol{t}$

【予測】

新しいデータ点$\boldsymbol{x}$に対しては、$y(\boldsymbol{x})=\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x})+b=\sum_{i=1}^{n} a_i\, t_i k(\boldsymbol{x},\boldsymbol{x}_i)+b$ の正負によって分類する。
ここで、最適化の結果得られた$a_i(i=1,2,...,n)$の中で$a_i=0$ に対応するデータ点は予測に影響を与えないので、$a_i>0$ に対応するデータ点（**サポートベクトル**）のみ保持しておく。bはサポートベクトルのインデックスの集合をSとすると、$b=\frac{1}{S}\sum_{s\in S}(t_s-\sum_{i=1}^{n} a_i\, t_i k(\boldsymbol{x},\boldsymbol{x}_i))$によって求める。

▣ ソフトマージン SVM

【学習】

分離不可能な場合は学習できないが、データ点がマージン内部に入ることや誤分類を許容することでその問題を回避する。
スラック変数$\xi_i\geq 0$ を導入し、マージン内部に入った点や誤分類された点に対しては、$\xi_i=|1-t_i y(\boldsymbol{x}_i)|$とし、これらを許容する代わり、ペナルティを与えるように、最適化問題を以下のように修正する。

$\min_{\boldsymbol{w},b}\ \frac{1}{2}\|\boldsymbol{w}\|^2+C\sum_{i=1}^{n}\xi_i$

subject to $ti(\boldsymbol{w}^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_i)+b)\geq 1-\xi i$ $(i=1,2,\cdots,n)$

ただし、パラメータCはマージンの大きさと誤差の許容度のトレードオフを決めるパラメータである。
この最適化問題にラグランジュ乗数法などを用いると、結局最大化する目的関数はハードマージン SVM と同じになる。

$\tilde{L}(\boldsymbol{a})=\sum_{i=1}^{n} a_i-\frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n} a_i a_j t_i t_j\boldsymbol{\phi}(\boldsymbol{x}_i)^{\mathrm{T}}\boldsymbol{\phi}(\boldsymbol{x}_j)$

ただし、制約条件が$a_i\geq 0$ の代わりに $0\leq a_i\leq C$ $(i=1,2,\cdots,n)$ となる。（ハードマージン SVM と同じ $\sum_{i=1}^{n} a_i\, t_i=0$ も制約条件）

- Cについて： Cが大きいと完全に分離することに過度になり、汎化性能が小さくなるので、一般に過学習を防ぐためにはCを小さくする方がよい。

機械学習・要点まとめ

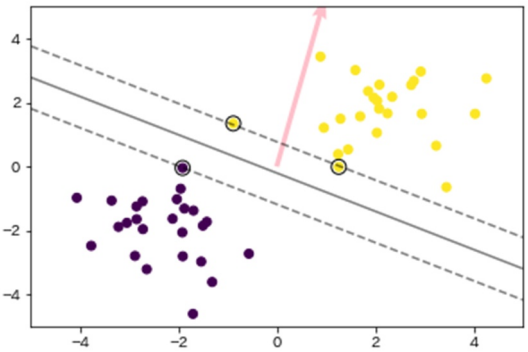
▣ サポートベクターマシンのハンズオン (np_svm.ipynb)

- 線形分離可能な場合： 正規分布に従う乱数に基づいて中心をずらしてそれぞれ点群を発生させ、SVM で 2 クラスに分類する。

```
# 【学習】
t = np.where(ys_train == 1.0, 1.0, -1.0)

n_samples = len(X_train)
# 線形カーネル
K = X_train.dot(X_train.T)
eta1 = 0.01
eta2 = 0.001
n_iter = 500

H = np.outer(t, t) * K
a = np.ones(n_samples)
for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.where(a > 0, a, 0)

# 【予測】
index = a > 1e-6
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]

term2 = K[index][:, index].dot(support_vector_a * support_vector_t)
b = (support_vector_t - term2).mean()
xx0, xx1 = np.meshgrid(np.linspace(-5, 5, 100), np.linspace(-5, 5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

X_test = xx
y_project = np.ones(len(X_test)) * b
for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * sv.dot(X_test[i])
y_pred = np.sign(y_project)
```

【考察】

★ マージンの内部にデータ点が存在せず、マージンの最大化が実現されている。



＜マージンと決定境界を可視化＞

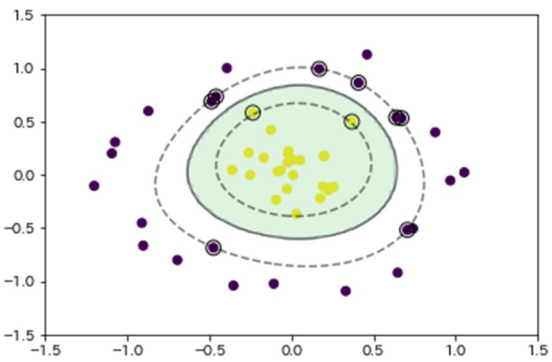
- **線形分離不可能**な場合： 同心円状に正規分布に従う乱数に基づいて点群を発生させ、非線形な rbf SVM で 2 クラスに分類する。

機械学習・要点まとめ

```
# 【学習】
def rbf(u, v):
        sigma = 0.8
        return np.exp(-0.5 * ((u - v)**2).sum() / sigma**2)

X_train = x_train
t = np.where(y_train == 1.0, 1.0, -1.0)

n_samples = len(X_train)
# RBF カーネル
K = np.zeros((n_samples, n_samples))
for i in range(n_samples):
    for j in range(n_samples):
        K[i, j] = rbf(X_train[i], X_train[j])

eta1 = 0.01
eta2 = 0.001
n_iter = 5000
H = np.outer(t, t) * K
a = np.ones(n_samples)
for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.where(a > 0, a, 0)

# 【予測】
index = a > 1e-6
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]

term2 = K[index][:, index].dot(support_vector_a * support_vector_t)
b = (support_vector_t - term2).mean()
xx0, xx1 = np.meshgrid(np.linspace(-1.5, 1.5, 100), np.linspace(-1.5, 1.5, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T
X_test = xx
y_project = np.ones(len(X_test)) * b
for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * rbf(X_test[i], sv)
y_pred = np.sign(y_project)
```

【考察】

★線形の場合と同様に、マージンの内部にデータ点が存在せず、マージンの最大化が実現されている。



機械学習・要点まとめ

- **ソフトマージン SVM**：正規分布に従う乱数に基づいて中心をずらしてそれぞれ点群を、重なりを許容して発生させ、ソフトマージン SVM で 2 クラスに分類する。

```
# 【学習】
X_train = x_train
t = np.where(y_train == 1.0, 1.0, -1.0)
n_samples = len(X_train)
# 線形カーネル
K = X_train.dot(X_train.T)

C = 1
eta1 = 0.01
eta2 = 0.001
n_iter = 1000

H = np.outer(t, t) * K
a = np.ones(n_samples)
for _ in range(n_iter):
    grad = 1 - H.dot(a)
    a += eta1 * grad
    a -= eta2 * a.dot(t) * t
    a = np.clip(a, 0, C)

# 【予測】
index = a > 1e-8
support_vectors = X_train[index]
support_vector_t = t[index]
support_vector_a = a[index]

term2 = K[index][:, index].dot(support_vector_a * support_vector_t)
b = (support_vector_t - term2).mean()

xx0, xx1 = np.meshgrid(np.linspace(-4, 4, 100), np.linspace(-4, 4, 100))
xx = np.array([xx0, xx1]).reshape(2, -1).T

X_test = xx
y_project = np.ones(len(X_test)) * b
for i in range(len(X_test)):
    for a, sv_t, sv in zip(support_vector_a, support_vector_t, support_vectors):
        y_project[i] += a * sv_t * sv.dot(X_test[i])
y_pred = np.sign(y_project)
```

【考察】

★ マージンの内部にデータ点の存在を許容しつつ、マージンの最大化を実現することになる。

★ 許容量は、ハイパーパラメータで制御できる。この値Cが大きいと完全に分離することに過度になり、汎化性能が小さくなるので、一般に過学習を防ぐためにはCを小さくする方が良いらしい。