1　ベイズ識別規則と評価基準

(1) 線形・2 次のベイズ識別規則の構成を付録のソースコード 1 に示した．

(2) (1) で構成した 2 つの識別規則をピマ・インディアンデータの発症しなかった人 223 名，発症した人 109 名のテストデータ Pima.te に対して適応すると，混同行列は以下のようになった．Linear_cm を線形のベイズ識別規則の混同行列，Quadratic_cm を 2 次のベイズ識別規則の混同行列として表している．

$$\text{Linear\_cm} = \begin{pmatrix} 198 & 25 \\ 46 & 63 \end{pmatrix}, \quad \text{Quadratic\_cm} = \begin{pmatrix} 198 & 25 \\ 39 & 70 \end{pmatrix}$$

そして上記の混同行列から，以下の指標を求めることができる．

表 1: ベイズ識別規則による性能指標

	偽陽性率	真陽性率	適合率	正確度	F 値
線形のベイズ識別規則	0.42	0.89	0.81	0.79	0.85
2 次のベイズ識別規則	0.36	0.89	0.84	0.81	0.86

上記の表 1 より，どちらの識別規則も真陽性率，適合率，正確度，F 値は比較的高いが，偽陽性率も高くなっていることが分かるため，あまり良い識別ができてるとは言えない．病気の有無の検査を考える際に，偽陽性の人たちは，可能な限り少なくさせた方が良いと考える．そのため今回の識別は，偽陽性の観点から良いとは言えない．識別境界を変化させることで，偽陽性率を向上させることができるが，誤り率は悪化することとなる．

(3) 線形・2 次のベイズ識別規則によるそれぞれの ROC 曲線のプロットを以下の図 1 に示す．

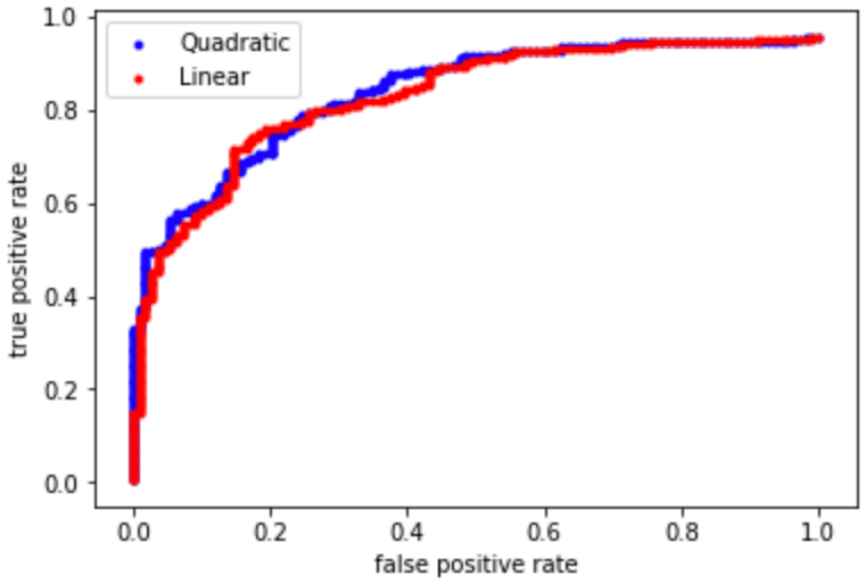

図 1: 線形・2 次のベイズ識別規則による ROC 曲線

また，得られた AUC はそれぞれ以下の通りである．

$$\text{Linear\_auc} = 0.87, \quad \text{Quadratic\_auc} = 0.88$$

どちらの識別規則も AUC の値は比較的高い数値を取っているため，AUC の値のみ見ると良い性能であるように見えるが，(2) より，偽陽性が高いことを考慮すると，今回のベイズ識別規則の性能は良いとは言えないと考える．

2 MNIST データ

(1) MNIST 手書き数字認証データを用いた kNN 法による識別を行い，ホールド・アウト法と 1 つ抜き法による誤り率のプロットを以下の図 2 に示す．ただし，計算時間を考慮して両者ともデータ数を 50 個に設定して実装している．

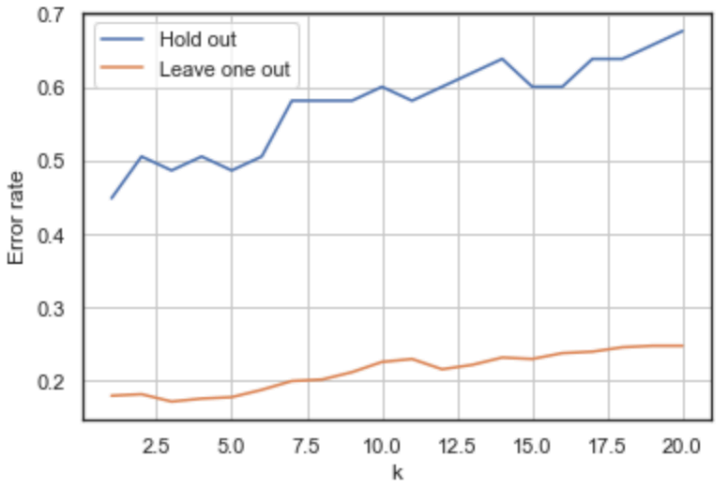

図 2: ホールド・アウト法と 1 つ抜き法による誤り率

(2) (1) より，ELT1 手書き数字データの誤り率のプロットとの違いは，以下のことが考えられる．

① ELT1 手書き数字データのプロットと比べ，ホールド・アウト法と 1 つ抜き法の誤り率の差が大きい．

② 明らかに MNIST 手書き数字認証データの方が，誤り率が大きい．

(3) 2 値化したデータによる $k = 5$ に固定した kNN 法を用いて，誤り率としきい値 T の関係を図 3（青線）に示した．
図 3 から分かる通り，今回最適なしきい値 T の求め方は，T を $0 \sim 255$ の 1 刻みで設定し，それぞれの T で誤り率を調べる，すなわち T による誤り率を毎回調べて，誤り率が最小である時の T を求める方法である．また同時に，T の選択の違いによる識別結果の影響を調べた．
図 3 の結果より，ある一定値から更に T を増加させると誤り率も増加することが分かった．従って，しきい値 T を単純に大きな数値に設定させるのではなく，最適な数値を求める必要があると言える．どちらかと言うと大きい数値よりも小さな数値で T を設定した方が精度が良いとも言える．

(4) sklearn.naive_bayes の BernoulliNB クラスを用いて，2 値化したデータのベルヌーイ分布によるベイズ識別規則を構成した．その構成したものをソースコード 2 に示す．また，(3) と同様にしきい値 T を変化させて，それぞれの誤り率を調べた結果を以下の図 3（橙色）に示す．

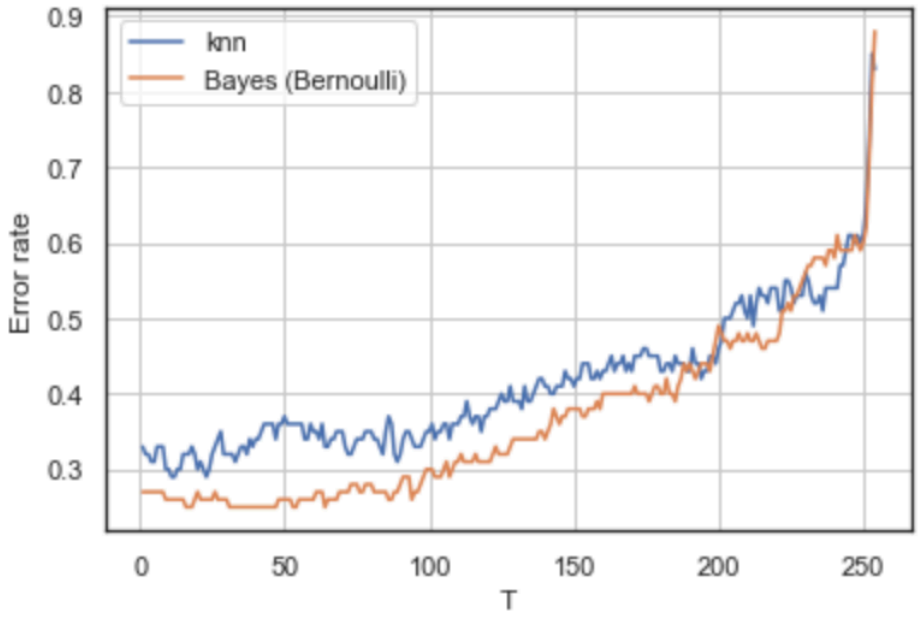

図 3: しきい値 T と誤り率の関係

図 3 より，ベイズ識別規則を用いた識別の方が，誤り率が低くなっていることが分かる．従って，2 値化したデータによるベルヌーイ分布に従う確率分布とした場合のベイズ識別規則の方が，良い識別規則であると言える．
しかし誤り率に着目すると，最適な T の時でも誤り率は大きいと言えるため精度を考えると，今回の場合は，どちらの手法もあまり良い手法では無いと考えられる．

3　ワインデータ

(1) 2 次元に射影したデータ LD1，LD2 の散布図と識別境界を以下の図 4 に示す．

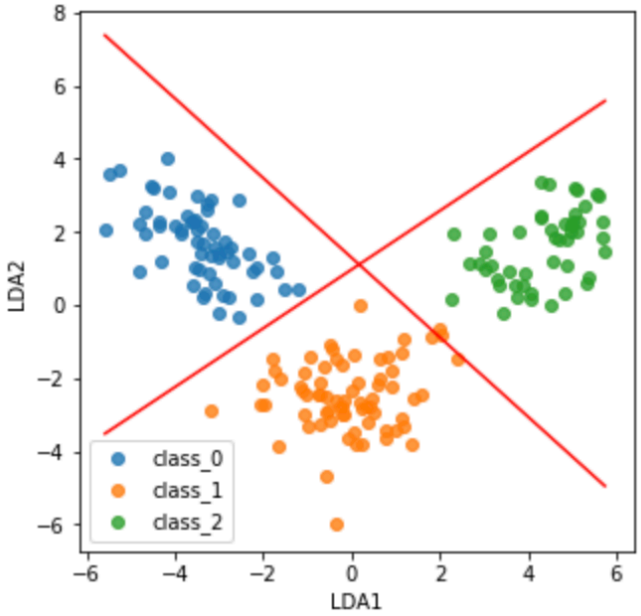

図 4: 2 次元に射影したデータの散布図と識別境界

また，混同行列 (cm) と予測精度は以下の結果となった．適合率と F 値はクラスごとにそれぞれ求めて，それらを平均したものである．

$$\mathrm{cm} = \begin{pmatrix} 59 & 0 & 0 \\ 0 & 71 & 0 \\ 0 & 0 & 48 \end{pmatrix}$$

表 2: ワインデータの線形判別分析による予測精度

正確度	1
汎化誤差	0
適合率	1
F 値	1

(2) 2 乗誤差最小基準による線形識別規則を構成し，求めた識別境界を以下の図 5 に示す．線形識別規則の構成はソースコード 3 に示す．

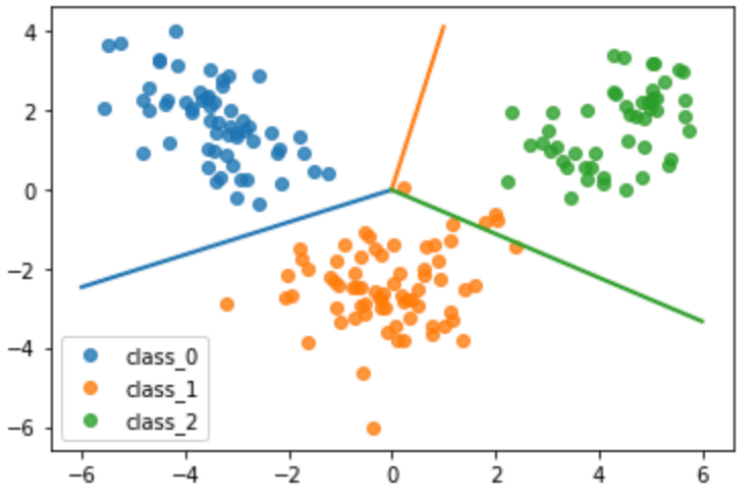

図 5: 2 乗誤差最小基準による識別境界

(3) 学習データとテストデータを 7:3 で分割し，乱数のシード値を”2020”として多項ロジット回帰分析を行い，混同行列 (Mult_LR_cm) と予測精度は以下の結果となった．(1) と同様に適合率と F 値はクラスごとにそれぞれ求めて，それらを平均したものである．また，多項ロジット回帰分析の実装はソースコード 4 である．

$$\text{Mult\_LR\_cm} = \begin{pmatrix} 18 & 0 & 0 \\ 0 & 23 & 1 \\ 0 & 0 & 12 \end{pmatrix}$$

表 3: ワインデータの多項ロジット回帰分析による予測精度

正確度	0.98
汎化誤差	0.019
適合率	0.97
F 値	0.98

線形判別分析と多項ロジット回帰分析の評価

表 2，表 3 より，どちらも非常に高い予測精度となっていることが分かる．両者を比較すると今回の場合は，線形判別分析の方が多項ロジット回帰分析よりも良い評価を得ることが可能であると言える．しかし，多項ロジット回帰分析も十分性能が良いと言えるため，多クラス分類を行う場合，非常に有効な手法のひとつであると考える．

4 判別分析とロジスティック回帰

(1) 学習データとテストデータを 7:3 で分割し，乱数のシード値を”2020”として識別を行った．また，学習データに対してそれぞれの手法で分析を行い，テストデータを予測させる実

装を行った．そして，識別の評価を行うために以下の指標をそれぞれ求めた．（以下表記は，LDA を線形判別分析，LR をロジスティック回帰，cm を混同行列としている．）

$$\text{LDA\_cm} = \begin{pmatrix} 2879 & 8 \\ 75 & 38 \end{pmatrix}, \quad \text{LR\_cm} = \begin{pmatrix} 2880 & 7 \\ 74 & 39 \end{pmatrix}$$

表 4: LDA と LR の性能指標

	正確度	汎化誤差	適合率	F 値
LDA	0.972	0.0277	0.975	0.986
LR	0.973	0.0270	0.975	0.986

(2) LDA と LR のそれぞれの ROC 曲線は以下の図 6，図 7 のようになった．

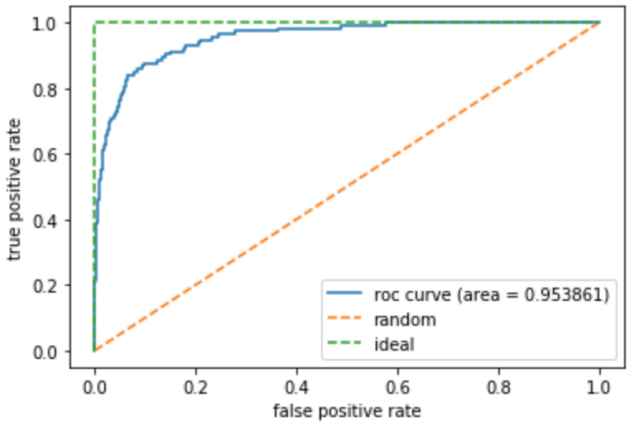

図 6: LDA の ROC 曲線

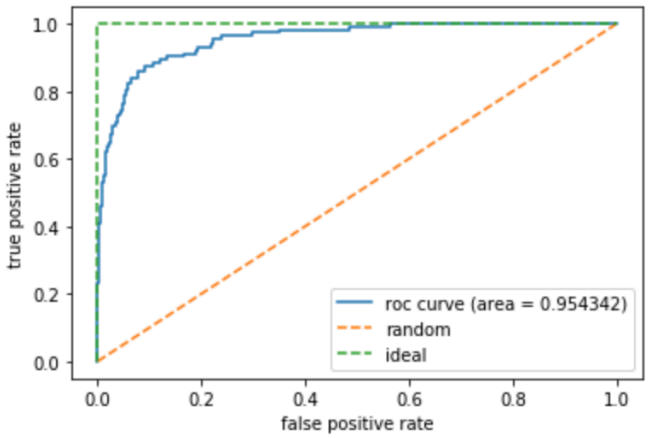

図 7: LR の ROC 曲線

また，ROC 曲線の AUC はそれぞれ以下の通りである．

$$\text{LDA\_auc} = 0.954, \ \text{LR\_auc} = 0.954$$

上記の表 4 と AUC の結果から，ロジスティックモデルを用いた手法の方が良い識別規則であると考える．

【理由】

AUC の結果から両者の AUC は，ほぼ同値であることが分かる．そのため，表 4 の指標で比較することになる．比較すると，ロジスティックモデルの方が汎化誤差が小さく，かつ正確度が高い値を取っていることが分かるため，ロジスティックモデルを用いた手法の方が良い識別規則であると言える．

今回どちらの手法が良いかの判断をしたが，両者の混同行列を見ると，偽陽性率が高くなっていることが分かるため識別について着目すると，あまり良い識別ができているとは言えない．

参考文献

[1] 多項ロジスティック回帰（LogisticRegression）を使用した機械学習
URL：https://hira03.hatenablog.com/entry/multinomial_logistic_regression
最終閲覧日：2020 年 7 月 21 日

[2] scikit-learn によるナイーブベイズ分類器 - Qiita
URL：https://qiita.com/ynakayama/items/ca3f5e9d762bbd50ad1f
最終閲覧日：2020 年 7 月 21 日

付録

ソースコード 1: 線形・2 次のベイズ識別規則

```
# 線形のベイズ識別規則
def bayes_normal_1(z, x1, x2 ,p1,p2):
    mu1 = np.mean(x1,axis=0)
    mu2 = np.mean(x2,axis=0)
    Sigma1 = np.cov(x1.T, bias=False)
    Sigma2 = np.cov(x2.T, bias=False)
    Spool = p1*Sigma1 + p2*Sigma2
    const = mu2.reshape(1, -1) @ np.linalg.inv(Spool)- mu1.reshape(1, -1) @
        np.linalg.inv(Spool)
    second_term = 2*const @ z.T
    F = mu1.reshape(1, -1) @ np.linalg.inv(Spool) @ mu1.reshape(1, -1).T \
    - mu2.reshape(1, -1) @ np.linalg.inv(Spool) @ mu2.reshape(1, -1).T \
     - 2*np.log(p1/p2)
    return(second_term+F)

# 2次のベイズ識別規則
def bayes_normal_2(z, x1, x2 ,p1,p2):
    mu1 = np.mean(x1,axis=0)
    mu2 = np.mean(x2,axis=0)
    Sigma1 = np.cov(x1.T, bias=False)
    Sigma2 = np.cov(x2.T, bias=False)
    S = np.linalg.inv(Sigma1) - np.linalg.inv(Sigma2)
    const = mu2.reshape(1, -1) @ np.linalg.inv(Sigma2)- mu1.reshape(1, -1) @
        np.linalg.inv(Sigma1)
    first_term = np.diag(z @ S @ z.T)
    second_term = 2*const @ z.T
    F = mu1.reshape(1, -1) @ np.linalg.inv(Sigma1) @ mu1.reshape(1, -1).T \
    - mu2.reshape(1, -1) @ np.linalg.inv(Sigma2) @ mu2.reshape(1, -1).T \
    + np.log( np.linalg.det(Sigma1)/np.linalg.det(Sigma2)) - 2*np.log(p1/p2)
    return(first_term+second_term+F)

import pandas as pd
import numpy as np

pima = pd.read_csv("pima.csv")
X = pima.loc[0:199,["npreg","glu","bp","skin","bmi","ped","age","type"]]
Y = X["type"].map({'No': 0, 'Yes': 1})
```

```
36
37 XA = X.loc[X['type'] == 'No',:].iloc[:, 0:7].to_numpy()
38 XB = X.loc[X['type'] == 'Yes',:].iloc[:, 0:7].to_numpy()
39
40 X_train = np.concatenate( (np.array(X['npreg']).reshape(-1,1), np.array(X['
       glu']).reshape(-1,1),np.array(X['bp']).reshape(-1,1),np.array(X['skin
       ']).reshape(-1,1), np.array(X['bmi']).reshape(-1,1),np.array(X['ped']).
       reshape(-1,1), np.array(X['age']).reshape(-1,1)), axis=1)
41
42 G1 = bayes_normal_1(X_train, XA, XB, 0.66, 0.34)
43 G2 = bayes_normal_2(X_train, XA, XB, 0.66, 0.34)
```

ソースコード 2: 2 値化したデータのベルヌーイ分布によるベイズ識別規則

```
1  from sklearn.naive_bayes import BernoulliNB
2  bernoulli_model = BernoulliNB()
3  accuracy_bernoulli = []
4
5  t_range = range(1, 255, 1)
6  for t in t_range:
7      Mnist_train_df_binarize = (Mnist100_train_df > t) * 1
8      Mnist_test_df_binarize = (Mnist100_test_df > t) * 1
9
10     bernoulli_model.fit(Mnist_train_df_binarize, Mnist100_train_label)
11     test_pred = bernoulli_model.predict(Mnist_test_df_binarize)
12     accuracy_bernoulli.append(metrics.accuracy_score(Mnist100_test_label,
           test_pred))
13
14 for k in range(0, 254):
15     accuracy_bernoulli[k] = 1 - accuracy_bernoulli[k]
```

ソースコード 3: 2 乗誤差最小基準による線形識別規則

```
1  from sklearn.datasets import load_wine
2  wine = load_wine()
3  X = wine.data
4  y = wine.target
5  target_names = wine.target_names
6
7  # LDA
8  lda = lda(n_components=2)
9  X_r = lda.fit(X, y).transform(X)
10
11 X = np.insert(X_r, 0, 1,axis=1)
12 T1 = np.concatenate([np.ones(60), np.zeros(118)],0)
13 T2 = np.concatenate([np.zeros(59), np.ones(59), np.zeros(60)],0)
14 T3 = np.concatenate([np.zeros(118), np.ones(60)],0)
15 T_mat = np.concatenate([T1,T2,T3],0).reshape(3,178).T
16 T_mat.shape
17
18 W_hat = np.linalg.inv(X.T @X) @ X.T @ T_mat
19 X_pred = (W_hat.T @ X.T).T
20 Y_pred = X_pred.argmax(axis=1) # 最小 2乗誤差基準による識別結果
21
```

```
for color, i, target_name in zip(colors, [0, 1, 2], target_names):
    plt.scatter(X_r[y == i, 0], X_r[y == i, 1], alpha=.8,
                label=target_name)

# 交点のx座標
x0 = ((W_hat[(0,2)]-W_hat[(0,0)])/(W_hat[(2,0)]-W_hat[(2,2)]) - (W_hat
    [(0,1)]-W_hat[(0,0)])/ \
    (W_hat[(2,0)]-W_hat[(2,1)]))/((W_hat[(1,1)]-W_hat[(1,0)])/ \
    (W_hat[(2,0)]-W_hat[(2,1)]) - (W_hat[(1,2)]-W_hat[(1,0)])/(W_hat
       [(2,0)]-W_hat[(2,2)]) )
# 交点のy座標
y0 = (W_hat[(1,2)]-W_hat[(1,0)])/(W_hat[(2,0)]-W_hat[(2,2)])*x0+(W_hat
    [(0,2)]-W_hat[(0,0)])/(W_hat[(2,0)]-W_hat[(2,2)])

# 1,2の識別境界の端点
x12 = -6
y12 = (W_hat[(1,1)]-W_hat[(1,0)])/(W_hat[(2,0)]-W_hat[(2,1)])*x12 + (W_hat
    [(0,1)]-W_hat[(0,0)])/ \
            (W_hat[(2,0)]-W_hat[(2,1)])
# 1,3の識別境界の点
x13 = 1
y13 = (W_hat[(1,2)]-W_hat[(1,0)])/(W_hat[(2,0)]-W_hat[(2,2)])*x13 + (W_hat
    [(0,2)]-W_hat[(0,0)])/ \
          (W_hat[(2,0)]-W_hat[(2,2)])
# 2,3の識別境界の端点
x23 = 6
y23 = (W_hat[(1,2)]-W_hat[(1,1)])/(W_hat[(2,1)]-W_hat[(2,2)])*x23 + (W_hat
    [(0,2)]-W_hat[(0,1)])/\
            (W_hat[(2,1)]-W_hat[(2,2)])

plt.plot([x12,x0],[y12, y0], lw=2)
plt.plot([x13,x0],[y13, y0], lw=2)
plt.plot([x0,x23],[y0, y23], lw=2)
plt.legend()
plt.show()
```

ソースコード 4: 多項ロジット回帰分析の実装

```
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split
from sklearn.datasets import load_wine
wine = load_wine()
X = wine.data
y = wine.target

X_tr, X_te, Y_tr, Y_te = train_test_split(X, y, test_size=0.3, random_state
    =2020)

scaler = StandardScaler()
X_train_sc = scaler.fit_transform(X_tr)
X_test_sc = scaler.fit_transform(X_te)
log_reg = LogisticRegression().fit(X_train_sc, Y_tr)
Y_pred = log_reg.predict(X_test_sc)
```