

# project1:naive bayes and logistic regression

## Navie bayes

在sklearn中实现了三类朴素贝叶斯：**GaussianNB**(高斯朴素贝叶斯)、**MultinomialNB**(多项式朴素贝叶斯)、**BernoulliNB**(伯努利朴素贝叶斯)，他们的区别是似然函数不同

### Bernoulli Naive Bayes

$$P(x_i | y) = P(i | y)x_i + (1 - P(i | y))(1 - x_i)$$

适合服从根据多元伯努利分布分布的数据

### Gaussian Naive Bayes

$$P(x_i | y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

适合处理连续变量

### Multinomial Naive Bayes

$$\hat{\theta}_{yi} = \frac{N_{yi} + \alpha}{N_y + \alpha n}$$

适合处理离散变量

本次尝试用这三种分类器来进行分类从而对比

import

```
from sklearn.naive_bayes import BernoulliNB
from sklearn.naive_bayes import GaussianNB
from sklearn.naive_bayes import MultinomialNB
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
import csv
import time
```

load data

```
X_train_file = csv.reader(open('../data/X_train.csv'))
Y_train_file = csv.reader(open('../data/Y_train.csv'))
X_test_file = csv.reader(open('../data/X_test.csv'))
Y_test_file = csv.reader(open('../data/Y_test.csv'))
X_train = []
Y_train = []
X_test = []
Y_test = []

def loadData():
    i = 0
    for content in X_train_file:
        i += 1
        if (i == 1):
            continue
        content = list(map(int, content))
        if len(content) != 0:
            X_train.append(content)
    i = 0;
```

```

for content in Y_train_file:
    i += 1
    if (i == 1):
        continue
    content = list(map(int, content))
    if len(content) != 0:
        Y_train.append(content)
i = 0;
for content in X_test_file:
    i += 1
    if (i == 1):
        continue
    content = list(map(int, content))
    if len(content) != 0:
        X_test.append(content)

i = 0;
for content in Y_test_file:
    i += 1
    if (i == 1):
        continue
    content = list(map(int, content))
    if len(content) != 0:
        Y_test.append(content)

```

## 主函数

```

startTime = time.time()
loadData()
endLoadTime = time.time()
bernNB()/ MultiNB()/GausNB()
endTime =time.time()
print("load file time: %.2f\ncalculating time: %.2f\ntotal time: %.2f"%(endLoadTime-startTime, endTime-endLoadTime, endTime-startTime))

```

## BernoulliNB

```

def bernNB():
    BernNB = BernoulliNB(binarize=True)
    BernNB.fit(X_train, Y_train)
    print(BernNB)
    y_expect = Y_test
    y_pred = BernNB.predict(X_test)
    print(accuracy_score(y_expect, y_pred))

```

## 主函数输出

```

BernoulliNB(binarize=True)
0.7825072170014127
load file time: 1.23
calculating time: 0.56
total time: 1.79

```

## MultinomialNB

```

def MultiNB():
    MultiNB = MultinomialNB()
    MultiNB.fit(X_train, Y_train)
    print(MultiNB)
    y_expect = Y_test
    y_pred = MultiNB.predict(X_test)
    print(accuracy_score(y_expect, y_pred))

```

## 主函数输出

```
MultinomialNB()  
0.785148332412014  
load file time: 1.25  
calculating time: 0.51  
total time: 1.76
```

## GaussianNB

```
def GausNB():  
    GausNB = GaussianNB()  
    GausNB.fit(X_train, Y_train)  
    print(GausNB)  
    y_expect = Y_test  
    y_pred = GausNB.predict(X_test)  
    print(accuracy_score(y_expect, y_pred))
```

```
GaussianNB()  
0.7952828450340889  
load file time: 1.23  
calculating time: 0.53  
total time: 1.77
```

因为加载数据的代码一致，应该比较calculating time, 得出性能应该是 MultinomialNB > GaussianNB>BernoulliNB

因为输入的数据是离散值并且没有符合伯努利分布，所以使用MultinomialNB最好，实际结果和理论分析一致

## logistic regression

### 数学推导

假设函数  $h_w(X) = \frac{1}{1+e^{-W^T X}}$

参数  $w$

$$\begin{aligned}h_w(X) &= g(W^T X) \\ g(z) &= \frac{1}{1+e^{-z}} \\ z &= W^T X\end{aligned}$$

SGD for logistic regression 损失函数

$$J(w) = -y^{(i)} \log(g(W^T x)) - (1 - y^{(i)}) \log(1 - g(W^T x))$$

$$\begin{aligned}g(z)' &= \frac{e^{-z}}{(1+e^{-z})^2} \\ &= \frac{1}{1+e^{-z}} \frac{e^{-z}}{1+e^{-z}} \\ &= \frac{1}{1+e^{-z}} \frac{1+e^{-z}-1}{1+e^{-z}} \\ &= g(z)(1-g(z))\end{aligned}$$

所以

$$\begin{aligned}
\frac{\partial}{\partial w} J(w) &= - \left[ y^{(i)} \frac{1}{g(W^T x^{(i)})} + (1 - y^{(i)}) \frac{1}{1 - g(W^T x^{(i)})} \right] \frac{\partial}{\partial w} g(W^T x^{(i)}) \\
&= - \times \frac{y^{(i)} (1 - g(W^T x^{(i)})) + (1 - y^{(i)}) g(W^T x^{(i)})}{g(W^T x^{(i)}) (1 - g(W^T x^{(i)}))} \\
&\times g(W^T x^{(i)}) \left( 1 - g(W^T x^{(i)}) \right) \frac{\partial}{\partial w} (W^T x^{(i)}) \\
&= - (y^{(i)} - g(W^T x^{(i)})) x^{(i)} \\
&= (h_w(x^{(i)}) - y^{(i)}) x^{(i)}
\end{aligned}$$

更新函数为

$$\begin{aligned}
w_j &:= w_j - \eta (h_w(x^{(i)}) - y^{(i)}) x_j^{(i)} \\
&\text{(与线性回归参数更新公式的形式相同, 但 } h(x^{(i)}) \text{ 不同。)} \\
h_w(X) &= g(W^T X)
\end{aligned}$$

## 辅助构造函数

### 1. 激活函数/sigmoid函数：

```
def sigmoid(z):
    a = 1/(1+np.exp(-z))
    return a
```

就这么easy, sigmoid的公式就是 $1/(1+e^{-x})$ , 这里用 `np.exp()` 就可以轻松构建。

### 2. 参数初始化函数（给参数都初始化为0）：

```
def initialize_with_zeros(dim):
    w = np.zeros((dim,1))
    b = 0
    return w,b
```

`W`是一个列向量, 传入维度`dim`, 返回`shape`为 `(dim,1)` 的`W`, `b`就是一个数。这里用到的方法是 `np.zeros(shape)`。

**3.propagate函数：**这里再次解释一下这个propagate, 它包含了forward-propagate和backward-propagate, 即正向传播和反向传播。正向传播求的是cost, 反向传播是从cost的表达式倒推`W`和`b`的偏导数。

`w` -- 权重, `shape` : `(num_px * num_px * 3, 1)`  
`b` -- 偏置项, 一个标量  
`X` -- 数据集, `shape` : `(num_px * num_px * 3, m)`, `m`为样本数  
`Y` -- 真实标签, `shape` : `(1,m)`

```
def propagate(w, b, X, Y):

    # Sample number m:
    m = X.shape[1]

    # Forward propagation:
    A = sigmoid(np.dot(w.T, X) + b) # use sigmoid func
    cost = -(np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A))) / m

    # Back propagation:
    dZ = A - Y
    dw = (np.dot(X, dZ.T)) / m
    db = (np.sum(dZ)) / m

    # return value
```

```

grads = {"dw": dw,
        "db": db}

return grads, cost

```

numpy中矩阵的点乘,也就是内积运算,是用 `np.dot(A,B)`,它要求前一个矩阵的列数等于后一个矩阵的行数。但矩阵也可以进行元素相乘 (element product),就是两个相同形状的矩阵对于元素相乘得到一个新的相同形状的矩阵,可以直接用  $A*B$  或者用 `np.multiply(A, B)`。矩阵求log用 `np.log()`,对矩阵元素求和用 `np.sum()`。

#### 4. optimize函数：

```

def optimize(w, b, X, Y, num_iterations, learning_rate, costs, startStep, print_cost=False):
    for i in range(num_iterations):
        # use propagate func to calculate the cost and gradient every iteration
        grads, cost = propagate(w, b, X, Y)
        dw = grads["dw"]
        db = grads["db"]

        # update parameters
        w = w - learning_rate * dw
        b = b - learning_rate * db

        # every num_iterations iterations, save the cost
        if i % num_iterations == 0:
            costs.append(cost)

        # print cost every num_iterations to keep track of the progress of the model
        if print_cost and i % (num_iterations / 10) == 0:
            print("Cost after iteration %i: %f" % (i+startStep, cost))
    # After iterating, place the final parameters in the dictionary and return:
    params = {"w": w,
             "b": b}
    grads = {"dw": dw,
            "db": db}
    return params, grads, costs

```

#### 5.predict函数：

按照预测的结果和0.5比来决定结果是1还是0.因此,我们可以设立规则:0.5~1的A对于预测值1,小于0.5的对应预测值0。

```

def predict(w, b, X):
    m = X.shape[1]
    Y_prediction = np.zeros((1, m))

    A = sigmoid(np.dot(w.T, X) + b)
    for i in range(m):
        if A[0, i] > 0.5:
            Y_prediction[0, i] = 1
        else:
            Y_prediction[0, i] = 0

    return Y_prediction

```

#### 6. logistic\_model 函数

不断调用optimize函数训练模型并实时输出模型训练情况 (accuracy, costs)

```

def logistic_model(x_train, y_train, x_test, y_test, learning_rate=0.1, num_iterations=2000, print_cost=False):
    # To obtain characteristic dimensions, initialization parameters:
    dim = X_train.shape[0]
    w, b = initialize_with_zeros(dim)
    step = 100
    # Define a Costs array to store the cost after each several iterations, so that we can draw a graph to see the change trend of cost:
    costs = []
    # Define the accuracy array to store accuracy after several iterations, so that a graph can be drawn to see the variation trend of accu
    accuracys_train = []
    accuracys_test = []
    for i in range(int(num_iterations / step)):
        # Gradient descent, model parameters can be calculated iteratively:
        params, grads, costs = optimize(w, b, x_train, y_train, step, learning_rate, costs, i*step, print_cost)
        w = params['w']

```

```

b = params['b']

# Use the parameters learned to make predictions:
prediction_train = predict(W, b, x_train)

# Calculation accuracy, respectively in training set and test set:
accuracy_train = 1 - np.mean(np.abs(prediction_train - y_train))
print("Accuracy on train set:", accuracy_train)
accuracys_train.append(accuracy_train)
# To facilitate analysis and inspection, we store all the parameters and hyperparameters obtained into a dictionary and return them:
d = {"costs": costs,
     "y_prediction_train": prediction_train,
     "w": W,
     "b": b,
     "learning_rate": learning_rate,
     "num_iterations": num_iterations,
     "train_acy": accuracy_train,
     "accuracys_train": accuracys_train
    }
# Use the parameters learned to make predictions:
prediction_test = predict(W, b, x_test)

# Calculation accuracy, respectively in training set and test set:
accuracy_test = 1 - np.mean(np.abs(prediction_test - y_test))
print("Accuracy on test set:", accuracy_test)

return d

```

## 7. 导入数据,归一化

使用panda导入，并使用reshape将数据归一化

```

def load_data():
    __filepath__ = ['../data/X_train.csv', '../data/Y_train.csv', '../data/X_test.csv', '../data/Y_test.csv']
    __X_train__ = pd.read_csv(__filepath__[0]).values
    __X_train__ = preprocessing.scale(__X_train__)
    __Y_train__ = pd.read_csv(__filepath__[1])['label']
    __Y_train__ = __Y_train__.values
    __X_test__ = pd.read_csv(__filepath__[2]).values
    __X_test__ = preprocessing.scale(__X_test__)
    __Y_test__ = pd.read_csv(__filepath__[3])['label'].values
    __X_train__ = __X_train__.reshape(__X_train__.shape[0], -1).T
    __Y_train__ = __Y_train__.reshape(__Y_train__.shape[0], -1).T
    __X_test__ = __X_test__.reshape(__X_test__.shape[0], -1).T
    __Y_test__ = __Y_test__.reshape(__Y_test__.shape[0], -1).T
    return __X_train__, __Y_train__, __X_test__, __Y_test__

```

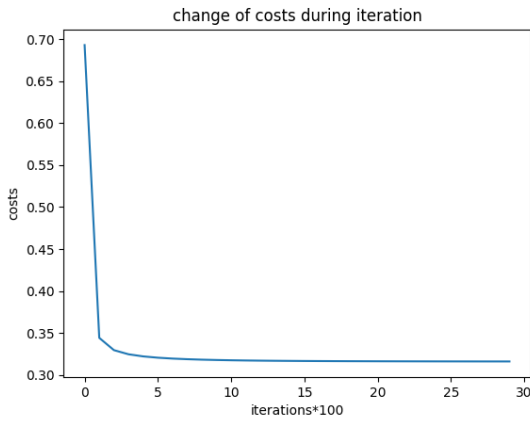
## 8. 主函数运行

设置学习率为0.15，迭代次数为3000次，并画出每次迭代的损失函数的变化情况

```

startTime = time.time()
X_train, Y_train, X_test, Y_test = load_data()
endLoadTime = time.time()
d = logistic_model(X_train, Y_train, X_test, Y_test, num_iterations=3000,
                  learning_rate=0.15, print_cost=True)
endTime=time.time()
print("load file time: %.2f\ncalculating time: %.2f \ntotal time: %.2f" % (endLoadTime-startTime, endTime-endLoadTime, endTime-startTime))
plt.ylabel('costs')
plt.xlabel('iterations*100')
plt.plot(d['costs'])
plt.show()

```



可以看见超过了85%的正确率

```
/Users/kangyixiao/EchoFile/coding/SE125_ML/lab1/venv/bin/python /Users/kangyixiao/EchoFile/coding/SE125_ML/lab1/src/logi/logi.py
Cost after iteration 0: 0.693147
Cost after iteration 10: 0.496506
Cost after iteration 20: 0.436814
Cost after iteration 30: 0.405058
Cost after iteration 40: 0.385495
Cost after iteration 50: 0.372463
Cost after iteration 60: 0.363283
Cost after iteration 70: 0.356528
Cost after iteration 80: 0.351380
Cost after iteration 90: 0.347343
Accuracy on train set: 0.8454900033782746
Cost after iteration 100: 0.344100
Cost after iteration 110: 0.341442
Cost after iteration 120: 0.339225
Cost after iteration 130: 0.337349
Cost after iteration 140: 0.335742
Cost after iteration 150: 0.334349
Cost after iteration 160: 0.333130
Cost after iteration 170: 0.332054
Cost after iteration 180: 0.331097
Cost after iteration 190: 0.330240
Accuracy on train set: 0.8484997389515064
Cost after iteration 200: 0.329468
Cost after iteration 210: 0.328769
Cost after iteration 220: 0.328132
Cost after iteration 230: 0.327550
Cost after iteration 240: 0.327015
Cost after iteration 250: 0.326522
Cost after iteration 260: 0.326065
Cost after iteration 270: 0.325642
Cost after iteration 280: 0.325248
Cost after iteration 290: 0.324880
Accuracy on train set: 0.8504038573753877
Cost after iteration 300: 0.324536
Cost after iteration 310: 0.324213
Cost after iteration 320: 0.323909
Cost after iteration 330: 0.323624
Cost after iteration 340: 0.323354
Cost after iteration 350: 0.323099
Cost after iteration 360: 0.322858
Cost after iteration 370: 0.322629
Cost after iteration 380: 0.322412
Cost after iteration 390: 0.322206
Accuracy on train set: 0.8513559165873283
Cost after iteration 400: 0.322009
Cost after iteration 410: 0.321822
Cost after iteration 420: 0.321643
Cost after iteration 430: 0.321472
Cost after iteration 440: 0.321309
Cost after iteration 450: 0.321153
Cost after iteration 460: 0.321003
```

```
Cost after iteration 470: 0.320860
Cost after iteration 480: 0.320722
Cost after iteration 490: 0.320590
Accuracy on train set: 0.8522772642117871
Cost after iteration 500: 0.320463
Cost after iteration 510: 0.320341
Cost after iteration 520: 0.320223
Cost after iteration 530: 0.320110
Cost after iteration 540: 0.320001
Cost after iteration 550: 0.319896
Cost after iteration 560: 0.319795
Cost after iteration 570: 0.319697
Cost after iteration 580: 0.319603
Cost after iteration 590: 0.319511
Accuracy on train set: 0.852338687386751
Cost after iteration 600: 0.319423
Cost after iteration 610: 0.319338
Cost after iteration 620: 0.319256
Cost after iteration 630: 0.319176
Cost after iteration 640: 0.319099
Cost after iteration 650: 0.319024
Cost after iteration 660: 0.318952
Cost after iteration 670: 0.318882
Cost after iteration 680: 0.318814
Cost after iteration 690: 0.318748
Accuracy on train set: 0.8525843800866066
Cost after iteration 700: 0.318684
Cost after iteration 710: 0.318622
Cost after iteration 720: 0.318562
Cost after iteration 730: 0.318503
Cost after iteration 740: 0.318446
Cost after iteration 750: 0.318391
Cost after iteration 760: 0.318338
Cost after iteration 770: 0.318285
Cost after iteration 780: 0.318235
Cost after iteration 790: 0.318186
Accuracy on train set: 0.853045053898836
Cost after iteration 800: 0.318138
Cost after iteration 810: 0.318091
Cost after iteration 820: 0.318046
Cost after iteration 830: 0.318002
Cost after iteration 840: 0.317959
Cost after iteration 850: 0.317917
Cost after iteration 860: 0.317876
Cost after iteration 870: 0.317837
Cost after iteration 880: 0.317798
Cost after iteration 890: 0.317760
Accuracy on train set: 0.8531064770737999
Cost after iteration 900: 0.317723
Cost after iteration 910: 0.317688
Cost after iteration 920: 0.317653
Cost after iteration 930: 0.317619
Cost after iteration 940: 0.317586
Cost after iteration 950: 0.317553
Cost after iteration 960: 0.317522
Cost after iteration 970: 0.317491
Cost after iteration 980: 0.317461
Cost after iteration 990: 0.317431
Accuracy on train set: 0.8534135929486195
Cost after iteration 1000: 0.317403
Cost after iteration 1010: 0.317375
Cost after iteration 1020: 0.317347
Cost after iteration 1030: 0.317321
Cost after iteration 1040: 0.317295
Cost after iteration 1050: 0.317269
Cost after iteration 1060: 0.317244
Cost after iteration 1070: 0.317220
Cost after iteration 1080: 0.317196
Cost after iteration 1090: 0.317173
Accuracy on train set: 0.8535057277110654
Cost after iteration 1100: 0.317150
Cost after iteration 1110: 0.317128
Cost after iteration 1120: 0.317106
Cost after iteration 1130: 0.317085
Cost after iteration 1140: 0.317064
Cost after iteration 1150: 0.317044
Cost after iteration 1160: 0.317024
Cost after iteration 1170: 0.317004
Cost after iteration 1180: 0.316985
Cost after iteration 1190: 0.316967
Accuracy on train set: 0.8537207088234391
```



```

Cost after iteration 1200: 0.316948
Cost after iteration 1210: 0.316930
Cost after iteration 1220: 0.316913
Cost after iteration 1230: 0.316896
Cost after iteration 1240: 0.316879
Cost after iteration 1250: 0.316863
Cost after iteration 1260: 0.316846
Cost after iteration 1270: 0.316831
Cost after iteration 1280: 0.316815
Cost after iteration 1290: 0.316800
Accuracy on train set: 0.8536285740609932
Cost after iteration 1300: 0.316785
Cost after iteration 1310: 0.316771
Cost after iteration 1320: 0.316756
Cost after iteration 1330: 0.316742
Cost after iteration 1340: 0.316729
Cost after iteration 1350: 0.316715
Cost after iteration 1360: 0.316702
Cost after iteration 1370: 0.316689
Cost after iteration 1380: 0.316677
Cost after iteration 1390: 0.316664
Accuracy on train set: 0.853751420410921
Cost after iteration 1400: 0.316652
Cost after iteration 1410: 0.316640
Cost after iteration 1420: 0.316628
Cost after iteration 1430: 0.316617
Cost after iteration 1440: 0.316606
Cost after iteration 1450: 0.316595
Cost after iteration 1460: 0.316584
Cost after iteration 1470: 0.316573
Cost after iteration 1480: 0.316563
Cost after iteration 1490: 0.316552
Accuracy on train set: 0.8538742667608489
Cost after iteration 1500: 0.316542
Cost after iteration 1510: 0.316532
Cost after iteration 1520: 0.316523
Cost after iteration 1530: 0.316513
Cost after iteration 1540: 0.316504
Cost after iteration 1550: 0.316494
Cost after iteration 1560: 0.316485
Cost after iteration 1570: 0.316476
Cost after iteration 1580: 0.316468
Cost after iteration 1590: 0.316459
Accuracy on train set: 0.853843555173367
Cost after iteration 1600: 0.316451
Cost after iteration 1610: 0.316442
Cost after iteration 1620: 0.316434
Cost after iteration 1630: 0.316426
Cost after iteration 1640: 0.316418
Cost after iteration 1650: 0.316411
Cost after iteration 1660: 0.316403
Cost after iteration 1670: 0.316396
Cost after iteration 1680: 0.316388
Cost after iteration 1690: 0.316381
Accuracy on train set: 0.8540278246982587
Cost after iteration 1700: 0.316374
Cost after iteration 1710: 0.316367
Cost after iteration 1720: 0.316360
Cost after iteration 1730: 0.316353
Cost after iteration 1740: 0.316347
Cost after iteration 1750: 0.316340
Cost after iteration 1760: 0.316334
Cost after iteration 1770: 0.316328
Cost after iteration 1780: 0.316321
Cost after iteration 1790: 0.316315
Accuracy on train set: 0.8540278246982587
Cost after iteration 1800: 0.316309
Cost after iteration 1810: 0.316303
Cost after iteration 1820: 0.316297
Cost after iteration 1830: 0.316292
Cost after iteration 1840: 0.316286
Cost after iteration 1850: 0.316281
Cost after iteration 1860: 0.316275
Cost after iteration 1870: 0.316270
Cost after iteration 1880: 0.316264
Cost after iteration 1890: 0.316259
Accuracy on train set: 0.8538742667608489
Cost after iteration 1900: 0.316254
Cost after iteration 1910: 0.316249
Cost after iteration 1920: 0.316244
Cost after iteration 1930: 0.316239

```

```

Cost after iteration 1940: 0.316234
Cost after iteration 1950: 0.316230
Cost after iteration 1960: 0.316225
Cost after iteration 1970: 0.316220
Cost after iteration 1980: 0.316216
Cost after iteration 1990: 0.316211
Accuracy on train set: 0.8537207088234391
Cost after iteration 2000: 0.316207
Cost after iteration 2010: 0.316202
Cost after iteration 2020: 0.316198
Cost after iteration 2030: 0.316194
Cost after iteration 2040: 0.316190
Cost after iteration 2050: 0.316186
Cost after iteration 2060: 0.316182
Cost after iteration 2070: 0.316178
Cost after iteration 2080: 0.316174
Cost after iteration 2090: 0.316170
Accuracy on train set: 0.8535057277110654
Cost after iteration 2100: 0.316166
Cost after iteration 2110: 0.316162
Cost after iteration 2120: 0.316159
Cost after iteration 2130: 0.316155
Cost after iteration 2140: 0.316151
Cost after iteration 2150: 0.316148
Cost after iteration 2160: 0.316144
Cost after iteration 2170: 0.316141
Cost after iteration 2180: 0.316138
Cost after iteration 2190: 0.316134
Accuracy on train set: 0.8536285740609932
Cost after iteration 2200: 0.316131
Cost after iteration 2210: 0.316128
Cost after iteration 2220: 0.316124
Cost after iteration 2230: 0.316121
Cost after iteration 2240: 0.316118
Cost after iteration 2250: 0.316115
Cost after iteration 2260: 0.316112
Cost after iteration 2270: 0.316109
Cost after iteration 2280: 0.316106
Cost after iteration 2290: 0.316103
Accuracy on train set: 0.8536899972359571
Cost after iteration 2300: 0.316100
Cost after iteration 2310: 0.316097
Cost after iteration 2320: 0.316094
Cost after iteration 2330: 0.316092
Cost after iteration 2340: 0.316089
Cost after iteration 2350: 0.316086
Cost after iteration 2360: 0.316084
Cost after iteration 2370: 0.316081
Cost after iteration 2380: 0.316078
Cost after iteration 2390: 0.316076
Accuracy on train set: 0.8536899972359571
Cost after iteration 2400: 0.316073
Cost after iteration 2410: 0.316071
Cost after iteration 2420: 0.316068
Cost after iteration 2430: 0.316066
Cost after iteration 2440: 0.316063
Cost after iteration 2450: 0.316061
Cost after iteration 2460: 0.316059
Cost after iteration 2470: 0.316056
Cost after iteration 2480: 0.316054
Cost after iteration 2490: 0.316052
Accuracy on train set: 0.8536899972359571
Cost after iteration 2500: 0.316050
Cost after iteration 2510: 0.316047
Cost after iteration 2520: 0.316045
Cost after iteration 2530: 0.316043
Cost after iteration 2540: 0.316041
Cost after iteration 2550: 0.316039
Cost after iteration 2560: 0.316037
Cost after iteration 2570: 0.316035
Cost after iteration 2580: 0.316033
Cost after iteration 2590: 0.316031
Accuracy on train set: 0.8536592856484752
Cost after iteration 2600: 0.316029
Cost after iteration 2610: 0.316027
Cost after iteration 2620: 0.316025
Cost after iteration 2630: 0.316023
Cost after iteration 2640: 0.316021
Cost after iteration 2650: 0.316019
Cost after iteration 2660: 0.316017
Cost after iteration 2670: 0.316016

```

```

Cost after iteration 2680: 0.316014
Cost after iteration 2690: 0.316012
Accuracy on train set: 0.8536285740609932
Cost after iteration 2700: 0.316010
Cost after iteration 2710: 0.316009
Cost after iteration 2720: 0.316007
Cost after iteration 2730: 0.316005
Cost after iteration 2740: 0.316003
Cost after iteration 2750: 0.316002
Cost after iteration 2760: 0.316000
Cost after iteration 2770: 0.315999
Cost after iteration 2780: 0.315997
Cost after iteration 2790: 0.315995
Accuracy on train set: 0.8536285740609932
Cost after iteration 2800: 0.315994
Cost after iteration 2810: 0.315992
Cost after iteration 2820: 0.315991
Cost after iteration 2830: 0.315989
Cost after iteration 2840: 0.315988
Cost after iteration 2850: 0.315986
Cost after iteration 2860: 0.315985
Cost after iteration 2870: 0.315983
Cost after iteration 2880: 0.315982
Cost after iteration 2890: 0.315980
Accuracy on train set: 0.8535978624735112
Cost after iteration 2900: 0.315979
Cost after iteration 2910: 0.315978
Cost after iteration 2920: 0.315976
Cost after iteration 2930: 0.315975
Cost after iteration 2940: 0.315974
Cost after iteration 2950: 0.315972
Cost after iteration 2960: 0.315971
Cost after iteration 2970: 0.315970
Cost after iteration 2980: 0.315968
Cost after iteration 2990: 0.315967
Accuracy on train set: 0.8534750161235835
Accuracy on test set: 0.8524046434494196
load file time: 0.49
calculating time: 8.72
total time: 9.21

```

在上一个sklearn中使用logistics regression来进行对比

```

def logisticRegression():
    log_reg = LogisticRegression()
    log_reg.fit(X_train, Y_train)
    y_expect = Y_test
    y_pred = log_reg.predict(X_test)
    print(accuracy_score(y_expect, y_pred))

```

```

0.7978011178674529
load file time: 1.15
calculating time: 0.57
total time: 1.72

```

```

Process finished with exit code 0

```

发现正确率由自己实现的高，但是我的实现时间比较长，因为在我的实现中有大量的I/O 操作。

## 总结

朴素贝叶斯的时间比较短，而logistics regression的时间由迭代次数决定，如果需要较高的精度，就需要较长的时间，因为在后来会收敛所以可以观察大致多少步可以结束。同时自己实现的方式和sklearn的logistic 的性能差异较大，可能是数据优化和IO输出导致的。