Section 11: Logistic Regression and Classification

In classification problem, the response variables y are discrete, representing different catagories.

Why not use linear regression for classification problem?

- The problem for range of y
- The inappropriate **MSE** loss function, especially for multi-class classification. It does not make sense to assume miss-classify 9 for 1 will yield a larger penalty than 7 for 1.
- There's no order in the y in **classification** -- they are just categories (imagine Iris flower, we can permute the label number as we like, while the permutation will definitely affect **regression** results)

Therefore for classification problem, we may want to:

- replace the mapping assumption between y and x
- replace the loss function in regression

In this section, we're going to learn **logistic regression**, which is a linear **classification** method and a direct generalization of linear regression. We will learn more classification models in the next section.

Binary Classification

For simplicity, we will first introduce the **binary classification case** -- y has only two categories, denoted as 0 and 1.

Model-setup of Logistic Regression (this is a classification model)

Assumption 1: Dependent on the variable x, the response variable y has different **probabilities** to take value in 0 or 1. Instead of predicting exact value of 0 or 1, we are actually predicting the **probabilities**.

Assumption 2: Logistic function assumption. Given x, what is the probability to observe $y = 1$?

$$P(y=1|\mathbf{x}) = f(\mathbf{x};\beta) = \frac{1}{1+\exp(-\tilde{x}\beta)} =: \sigma(\tilde{x}\beta).$$

where $\sigma(z) = \frac{1}{1+\exp{(-z)}}$ is called standard logistic function, or sigmoid function in deep learning. Recall that $\beta \in \mathbb{R}^{p+1}$ and $\tilde{x}$ is the "augmented" sample with first element one to incorporate intercept in the linear function.

Equivalent expression:

- Denote $p = P(y=1|\mathbf{x})$, then we can write in linear form (the LHS is called **odds ratio** in statistics)

$$\ln \frac{p}{1-p} = \tilde{x}\beta$$

- Since y only takes value in 0 or 1, we choose our exponents to be **indicators** of y. We have

$$P(y|\mathbf{x}, \beta) = f(\mathbf{x};\beta)^y \big(1 - f(\mathbf{x};\beta)\big)^{1-y}$$

- Note: for conditional probability, $|$ and ; are interchangable and mean the same thing.

MLE (Maximum Likelihood Estimation)

Assume the samples are independent. The overall probability to witness the whole training dataset

$$\begin{aligned} & P(\mathbf{y} \mid \mathbf{X}; \beta) \\ = & \prod_{i=1}^{N} P\left(y^{(i)} \mid \mathbf{x}^{(i)}; \beta\right) \\ = & \prod_{i=1}^{N} f\big(\mathbf{x}^{(i)};\beta\big)^{y^{(i)}} \Big(1 - f\big(\mathbf{x}^{(i)};\beta\big)\Big)^{\left(1-y^{(i)}\right)}. \end{aligned}$$

By maximizing the logarithm of likelihood function, then we derive the **loss function** to be minimized $$ L (\beta) = L (\beta; X,\mathbf{y}) = - \frac{1}{N}\sum_{i=1}^N \Bigl{y^{(i)} \ln\big(f(\mathbf{x}^{(i)};\beta) \big)

- (1 - y^{(i)}) \ln\big(1 - f(\mathbf{x}^{(i)};\beta) \big) \Bigr}. $$

The loss function also has clear probabilistic interpretations. Given i-th sample, the vector of true labels $(y^{(i)}, 1 - y^{(i)})$ can also be viewed as the probability distribution. Then the loss function is the mean of all cross entropy across samples, i.e. **"distance" between observed sample probability distribution and modelled probability distribution** via logistic model.

Remark: here we derive the loss function via MLE. Of course from the experience of linear regression, we know that we can also use MAP (bayesian approach), where the regularization term of β can be naturally introduced.

Algorithm

Take the gradient (left as exercise -- if you like)

$$\frac{\partial L(\beta)}{\partial \beta_k} = \frac{1}{N} \sum_{i=1}^{N} \big(\sigma(\tilde{x}^{(i)}\beta) - y^{(i)}\big)\tilde{x}_k^{(i)}.$$

In vector form:

$$\nabla_\beta\big(L(\beta)\big) = \sum_{i=1}^{N} \big(\sigma(\tilde{x}^{(i)}) - y^{(i)}\big)\tilde{x}^{(i)} = \frac{1}{N} \sum_{i=1}^{N} \big(f(\mathbf{x}^{(i)};\beta) - y^{(i)}\big)\tilde{x}^{(i)}$$

The last expression in the above line can be further represented as a row vector times a matrix:

$$\nabla_\beta\big(L(\beta)\big) = \frac{1}{N}\begin{pmatrix} f(\mathbf{x}^{(1)};\beta) - y^{(1)} & f(\mathbf{x}^{(2)};\beta) - y^{(2)} & \cdots & f(\mathbf{x}^{(N)};\beta) - y^{(N)} \end{pmatrix}\tilde{X}.$$

This is still the nonlinear function of β, indicating that we cannot derive something like "normal equations" in OLS. The solution here is numerical optimization.

The simplest algorithm in optimization is gradient descent (GD).

$$\beta^{k+1} = \beta^k - \eta \nabla L(\beta^k).$$

Here the step size η is also called **learning rate** in machine learning. Note that it is indeed the Euler's scheme to solve the ODE (for a 1-variable version, see Notes on Diffy Q's Section 1.7):

$$\dot{\beta} = -\nabla L(\beta).$$

By setting certain stopping criterion for GD, we think that we have approximated the optimized solution $\hat{\beta}$.

Making predictions and Evaluation of Performance

Now with the estimated $\hat{\beta}$ and given a new data x^{new}, we calculate the probability that $y^{new} = 1$ as $f(\mathbf{x}; \beta)$. If is greater than 0.5, we assign that $y^{new} = 1$.

For the test dataset, the **accuracy** is defined as ratio of number of correct predictions to the total number of samples.

Example Code

In [1]:

```
import numpy as np

class myLogisticRegression_binary():
    """ Logistic Regression classifier -- this only works for the binary case.
    Parameters:
    -----------
    learning_rate: float
        The step length that will be taken when following the negative gradient during
        training.
    """
    def __init__(self, learning_rate=.1):

        # learning rate can also be in the fit method
        self.learning_rate = learning_rate


    def fit(self, data, y, n_iterations = 1000):
        """
        don't forget the document string in methods
        """
        ones = np.ones((data.shape[0],1)) # column of ones
        X = np.concatenate((ones, data), axis = 1) # the augmented matrix, \tilde{X} in
        eta = self.learning_rate

        beta  = np.zeros(np.shape(X)[1]) # initialize beta, can be other choices

        for k in range(n_iterations):
            dbeta = self.loss_gradient(beta,X,y) # write another function to compute gr
            beta = beta - eta * dbeta # the formula of GD
            # this step is optional -- just for inspection purposes
            if k % 500 == 0: # pprint loss every 500 steps
                print("loss after", k+1, "iterations is: ", self.loss(beta,X,y))

        self.coeff = beta

    def predict(self, data):
        ones = np.ones((data.shape[0],1)) # column of ones
        X = np.concatenate((ones, data), axis = 1) # the augmented matrix, \tilde{X} in
        beta = self.coeff # the estimated beta
        y_pred = np.round(self.sigmoid(np.dot(X,beta))).astype(int) # >0.5: ->1 else,->
        return y_pred

    def score(self, data, y_true):
        ones = np.ones((data.shape[0],1)) # column of ones
        X = np.concatenate((ones, data), axis = 1) # the augmented matrix, \tilde{X} in
        y_pred = self.predict(data)
        acc = np.mean(y_pred == y_true) # number of correct predictions/N
        return acc

    def sigmoid(self, z):
        return 1.0 / (1.0 + np.exp(-z))

    def loss(self,beta,X,y):
        f_value = self.sigmoid(np.matmul(X,beta))
        loss_value = np.log(f_value + 1e-10) * y + (1.0 - y)* np.log(1 - f_value + 1e-1
        return -np.mean(loss_value)

    def loss_gradient(self,beta,X,y):
        f_value = self.sigmoid(np.matmul(X,beta))
        gradient_value = (f_value - y).reshape(-1,1)*X # this is the hardest expression
        return np.mean(gradient_value, axis=0)
```

In [2]:

```
from sklearn.datasets import load_breast_cancer
X, y = load_breast_cancer(return_X_y = True)
```

In [3]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=4
```

In [4]:

```
X_train.shape
```

Out[4]:

```
(512, 30)
```

In [19]:

```
%%time
lg = myLogisticRegression_binary(learning_rate=1e-5)
lg.fit(X_train,y_train,n_iterations = 10000) # what about increase n_iterations?
```

```
loss after 1 iterations is:  0.7704000919325609
loss after 501 iterations is:  0.3038878556607375
loss after 1001 iterations is:  0.26467267051646637
loss after 1501 iterations is:  0.24813479245950684
loss after 2001 iterations is:  0.2389480595788275
loss after 2501 iterations is:  0.2331178552172888
loss after 3001 iterations is:  0.22909746536348713
loss after 3501 iterations is:  0.22615149966747047
loss after 4001 iterations is:  0.22388601401780292
loss after 4501 iterations is:  0.22207285161370102
loss after 5001 iterations is:  0.22057221113785214
loss after 5501 iterations is:  0.21929462157026375
loss after 6001 iterations is:  0.21818078310948247
loss after 6501 iterations is:  0.21719023774728446
loss after 7001 iterations is:  0.21629469731306183
loss after 7501 iterations is:  0.21547395937965436
loss after 8001 iterations is:  0.21471332436186458
loss after 8501 iterations is:  0.21400191582326
loss after 9001 iterations is:  0.2133315615530969
loss after 9501 iterations is:  0.21269603244872282
Wall time: 1.52 s
```

In [20]:
```
lg.score(X_test,y_test)
```

Out[20]:
```
1.0
```

In [21]:
```
lg.score(X_train,y_train)
```

Out[21]:
```
0.9140625
```

In [22]:
```
lg.predict(X_test)
```

Out[22]:
```
array([1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1,
       0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1])
```

In [23]:
```
y_test
```

Out[23]:
```
array([1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1,
       0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1,
       1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1])
```

In [10]:
```
lg.coeff
```

Out[10]:
```
array([ 2.66882874e-03,  1.83927907e-02, -4.36239390e-03,  8.66487934e-02,
        1.49727981e-02,  5.50578751e-05, -6.71061493e-04, -1.14024608e-03,
       -4.51040831e-04,  1.06678514e-04,  8.62208844e-05,  1.72299429e-04,
        4.51211649e-04, -2.32558967e-03, -2.93966958e-02, -5.28628701e-06,
       -1.83325756e-04, -2.46370803e-04, -5.80574855e-05, -9.52561495e-06,
       -1.16457037e-05,  1.92488199e-02, -1.67105144e-02,  6.60427872e-02,
       -2.88220491e-02, -2.09664782e-05, -2.48568195e-03, -3.30713576e-03,
       -8.60537728e-04, -1.96407733e-04, -8.95680417e-05])
```

In [12]:
```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0)
clf.fit(X_train,y_train)
clf.score(X_test,y_test)
```

```
E:\ProgramData\Anaconda3\lib\site-packages\sklearn\linear_model\_logistic.py:762: Conver
genceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.

Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  n_iter_i = _check_optimize_result(
```

```
Out[12]: 0.9824561403508771
```

```
In [13]: clf.score(X_train,y_train)
```

```
Out[13]: 0.955078125
```

It's very normal that our result is different with sklearn. In sklearn logistic regression, by default the loss function is different (they regularization terms!).

Multi-class Classification

Note that your final project is a multi-class classification problem

Model

Let $\tilde{x} \in \mathbb{R}^{p+1}$ denotes the augmented row vector (one sample). We approximate the probabilities to take value in K classes as

$$f(\mathbf{x}; W) = \begin{pmatrix} P(y=1|\mathbf{x};\mathbf{w}) \\ P(y=2|\mathbf{x};\mathbf{w}) \\ \vdots \\ P(y=K|\mathbf{x};\mathbf{w}) \end{pmatrix} = \frac{1}{\sum_{k=1}^{K} \exp\left(\tilde{x}\mathbf{w}_k\right)} \begin{pmatrix} \exp(\tilde{x}\mathbf{w}_1) \\ \exp(\tilde{x}\mathbf{w}_2) \\ \vdots \\ \exp(\tilde{x}\mathbf{w}_K) \end{pmatrix}.$$

where we have K sets of parameters, $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_K$, and the sum factor normalizes the results to be a probability.

$\mathbf{W}$ is an $(p+1) \times K$ matrix containing all K sets of parameters, obtained by concatenating $\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_K$ into columns, so that $\mathbf{w}_k = (w_{k0}, \ldots, w_{kp})^\top \in \mathbb{R}^{p+1}$.

$$\mathbf{W} = \begin{pmatrix} | & | & | & | \\ \mathbf{w}_1 & \mathbf{w}_2 & \cdots & \mathbf{w}_K \\ | & | & | & | \end{pmatrix},$$

and $\tilde{X}\mathbf{W}$ is valid and useful in vectorized code.

Another Expression: Introduce the hidden variable $\mathbf{z} = (z_1, \ldots, z_K)$ and define

$$\mathbf{z} = \tilde{\mathbf{x}}\mathbf{W}$$

or element-wise written as

$$z_k = \tilde{\mathbf{x}}\mathbf{w_k}, k = 1, 2, \ldots, K$$

Then the **predicted probability distribution** can be denoted as

$$f(\mathbf{x}; W) = \sigma(z) \in \mathbb{R}^k$$

where $\sigma(\mathbf{z})$ is called the soft-max function which is defined as

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} \text{ for } i = 1, \dots, K \text{ and } \mathbf{z} = (z_1, \dots, z_K) \in \mathbb{R}^K$$

This is a valid probability distribution with K classes because you can check its element-wise sum is one and each component is positive.

This can be assumed as the (degenerate) simplest example of neural network that we're going to learn in later lectures, and that's why some people refer to multi-class logistic regression (also known as **soft-max logistic regression**) as **one-layer neural network**.

Loss function

Define the following **indicator function** (and again can be derived from MLE):

$$1_{\{y=k\}} = 1_{\{k\}}(y) = \delta_{yk} = \begin{cases} 1 & \text{when } y = k, \\ 0 & \text{otherwise.} \end{cases}$$

Loss function is again using the cross entropy:

$$\begin{aligned} L(\mathbf{W}; X, \mathbf{y}) &= -\frac{1}{N}\sum_{i=1}^N \sum_{k=1}^K \left\{ 1_{\{y^{(i)}=k\}} \ln P\left(y^{(i)} = k | \mathbf{x}^{(i)}; \mathbf{w}\right) \right\} \\ &= -\frac{1}{N}\sum_{i=1}^N \sum_{k=1}^K \left\{ 1_{\{y^{(i)}=k\}} \ln \left(\frac{\exp(\tilde{x}^{(i)}\mathbf{w}_k)}{\sum_{m=1}^K \exp\left(\tilde{x}^{(i)}\mathbf{w}_m\right)} \right) \right\}. \end{aligned}$$

Notice that for each term in the summation over N (i.e. fix sample i), only one term is non-zero in the sum of K elements due to the indicator function.

Gradient descent

After **careful calculation**, the gradient of L with respect the whole k-th set of weights is then:

$$\frac{\partial L}{\partial \mathbf{w}_k} = \frac{1}{N}\sum_{i=1}^N \left(\frac{\exp(\tilde{x}^{(i)}\mathbf{w}_k)}{\sum_{m=1}^K \exp(\tilde{x}^{(i)}\mathbf{w}_m)} - 1_{\{y^{(i)}=k\}} \right) \tilde{x}^{(i)} \in \mathbb{R}^{p+1}.$$

In writing the code, it's helpful to make this as the column vector, and stack all the K gradients together as a new matrix $\mathbf{dW} \in \mathbb{R}^{(p+1)\times K}$. This makes the update of matrix $\mathbf{W}$ very convenient in gradient descent.

Prediction

The largest estimated probability's class as this sample's predicted label.

$$\hat{y} = \arg\max_j P(y = j | \mathbf{x}),$$

In other words, we get the class j with the largest conditional probability, a.k.a. likelihood.

In [24]:

```
import numpy as np

class myLogisticRegression():
    """ Logistic Regression classifier -- this also works for the multiclass case.
    Parameters:
    -----------
    learning_rate: float
        The step length that will be taken when following the negative gradient during
        training.
    """
    def __init__(self, learning_rate=.1):

        # learning rate can also be in the fit method
        self.learning_rate = learning_rate


    def fit(self, data, y, n_iterations = 1000):
        """
        don't forget the document string in methods, here and all others!!!
        """
        self.K = max(y)+1 # specify number of classes in y
        ones = np.ones((data.shape[0],1)) # column of ones
        X = np.concatenate((ones, data), axis = 1) # the augmented matrix, \tilde{X} in
        eta = self.learning_rate

        W  = np.zeros((np.shape(X)[1],max(y)+1)) # initialize beta, can be other choice

        for k in range(n_iterations):
            dW = self.loss_gradient(W,X,y) # write another function to compute gradient
            W = W - eta * dW # the formula of GD
            # this step is optional -- just for inspection purposes
            if k % 500 == 0: # print loss every 500 steps
                print("loss after", k+1, "iterations is: ", self.loss(W,X,y))

        self.coeff = W

    def predict(self, data):
        ones = np.ones((data.shape[0],1)) # column of ones
        X = np.concatenate((ones, data), axis = 1) # the augmented matrix, \tilde{X} in
        W = self.coeff # the estimated W
        y_pred = np.argmax(self.sigma(X,W), axis =1) # the category with largest probab
        return y_pred

    def score(self, data, y_true):
        ones = np.ones((data.shape[0],1)) # column of ones
        X = np.concatenate((ones, data), axis = 1) # the augmented matrix, \tilde{X} in
        y_pred = self.predict(data)
        acc = np.mean(y_pred == y_true) # number of correct predictions/N
        return acc

    def sigma(self,X,W): #return the softmax probability
        s = np.exp(np.matmul(X,W))
        total = np.sum(s, axis=1).reshape(-1,1)
        return s/total

    def loss(self,W,X,y):
        f_value = self.sigma(X,W)
        K = self.K
        loss_vector = np.zeros(X.shape[0])
        for k in range(K):
            loss_vector += np.log(f_value+1e-10)[:,k] * (y == k) # avoid nan issues
        return -np.mean(loss_vector)

    def loss_gradient(self,W,X,y):
        f_value = self.sigma(X,W)
        K = self.K
        dLdW = np.zeros((X.shape[1],K))
        for k in range(K):
            dLdWk =(f_value[:,k] - (y==k)).reshape(-1,1)*X # Numpy broadcasting
            dLdW[:,k] = np.mean(dLdWk, axis=0)   # RHS is 1D Numpy array -- so you can
        return dLdW
```

In [25]:

```
from sklearn.datasets import load_digits
X,y = load_digits(return_X_y = True)
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=4
```

In [26]:

```
X_train.shape
```

Out[26]:

```
(1617, 64)
```

In [27]:

```
lg = myLogisticRegression(learning_rate=1e-4)
lg.fit(X_train,y_train,n_iterations = 20000) # what about change the parameters?
```

```
loss after 1 iterations is:  2.2975031101988965
loss after 501 iterations is:  0.9747646840265886
loss after 1001 iterations is:  0.6271544957404386
loss after 1501 iterations is:  0.48465074291917476
loss after 2001 iterations is:  0.4067886795416971
loss after 2501 iterations is:  0.3569853787369549
loss after 3001 iterations is:  0.3219498860091718
loss after 3501 iterations is:  0.2957112499207807
loss after 4001 iterations is:  0.27517638606506345
loss after 4501 iterations is:  0.2585728459578632
loss after 5001 iterations is:  0.24480630370680928
loss after 5501 iterations is:  0.23316150090969137
loss after 6001 iterations is:  0.22314954388974834
loss after 6501 iterations is:  0.21442400929215735
loss after 7001 iterations is:  0.20673204886938293
loss after 7501 iterations is:  0.19988447822601138
loss after 8001 iterations is:  0.19373672640885967
loss after 8501 iterations is:  0.18817628341982656
loss after 9001 iterations is:  0.1831141858457873
loss after 9501 iterations is:  0.17847909502105727
loss after 10001 iterations is:  0.17421308722718495
loss after 10501 iterations is:  0.17026860268039193
loss after 11001 iterations is:  0.16660619607205016
loss after 11501 iterations is:  0.16319285237565423
loss after 12001 iterations is:  0.16000070825015078
loss after 12501 iterations is:  0.15700606905300007
loss after 13001 iterations is:  0.15418864437799004
loss after 13501 iterations is:  0.15153094723746474
loss after 14001 iterations is:  0.14901781725362154
loss after 14501 iterations is:  0.14663603885543686
loss after 15001 iterations is:  0.14437403299967985
loss after 15501 iterations is:  0.1422216063268606
loss after 16001 iterations is:  0.14016974557619974
loss after 16501 iterations is:  0.13821044795587994
loss after 17001 iterations is:  0.13633658029523862
loss after 17501 iterations is:  0.1345417614013797
loss after 18001 iterations is:  0.13282026324911267
loss after 18501 iterations is:  0.13116692755309206
loss after 19001 iterations is:  0.12957709497826864
loss after 19501 iterations is:  0.12804654479263708
```

In [28]:

```
lg.score(X_test,y_test)
```

Out[28]:

```
0.9722222222222222
```

In [29]:

```
np.where(lg.predict(X_test)!=y_test)
```

Out[29]:

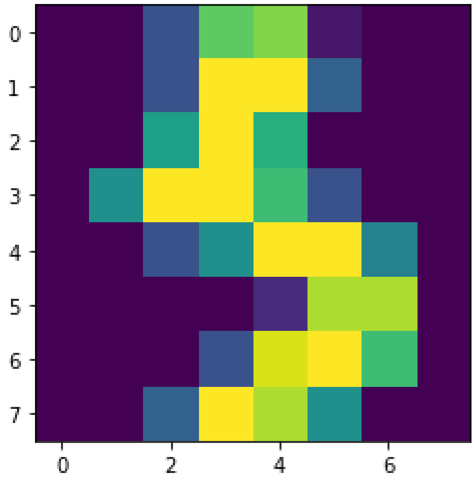
```
(array([  5,  71, 133, 149, 159], dtype=int64),)
```

In [31]:

```
import matplotlib.pyplot as plt
plt.imshow(X_test[149,].reshape(8,8))
```

Out[31]:

```
<matplotlib.image.AxesImage at 0x2637a38ceb0>
```



In [32]:

```
print(lg.predict(X_test)[149],y_test[149])
```

```
5 3
```

For multi-class classification, the confusion matrix can provide as more details.

In [33]:

```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test,lg.predict(X_test))
```

Out[33]:

```
array([[17,  0,  0,  0,  0,  0,  0,  0,  0,  0],
       [ 0, 10,  1,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0, 17,  0,  0,  0,  0,  0,  0,  0],
       [ 0,  0,  0, 16,  0,  1,  0,  0,  0,  0],
       [ 0,  0,  0,  0, 25,  0,  0,  0,  0,  0],
       [ 0,  0,  0,  0,  0, 21,  0,  0,  0,  1],
       [ 0,  0,  0,  0,  0,  0, 19,  0,  0,  0],
       [ 0,  0,  0,  0,  0,  0,  0, 18,  0,  1],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  8,  0],
       [ 0,  0,  0,  0,  0,  0,  0,  0,  1, 24]], dtype=int64)
```

Tricks in training: Stochastic Gradient Descent (SGD)

When you're doing the final project, it's very likely that you might lose patience -- training on the 60,000 MNIST data is VERY SLOW! (of course it's not an excuse to abandon the project lol)

To speed up the training process (most importantly the optimization algorithm), there are two directions of general strategies:

```
- find better algorithm whose convergence is faster (you take less steps
to arrive at the minimum)
- save the computational cost within each step
```

Of course there are trade-offs between these two directions.

Basic observation of SGD: Calculating the gradient in each step is TOO EXPENSIVE!

Recall that in general supervised learning,

$$\nabla_\beta L(\beta; X, Y) = \frac{1}{N}\sum_{i=1}^{N} \nabla_\beta l(\beta; x^{(i)}, y^{(i)})$$

It means that we need to implement 60,000 sum calculation in the single step!!!

"Wild" yet smart idea: Note that the RHS is in the form of "population average". The basic intuitive from statistics is that we can use "sample means" to replace "population average". If you're bold enough -- just randomly pick up ONE single sample and use this value to replace "population average"!

- Herustic expression of "pure stochastic" SGD:

  $$\beta^{k+1} = \beta^k - \eta \nabla_\beta l(\beta^k; x^{(r)}, y^{(r)}),$$

  where r denotes the index randomly picked during this step.

- (mini-batch SGD, or "standard" SGD):

  $$\beta^{k+1} = \beta^k - \eta \frac{1}{n_B}\sum_{k=1}^{n_B} \nabla_\beta l(\beta^k; x^{(k)}, y^{(k)}),$$

  where n_b denotes the size of mini-batch, and the average is taken over the n_b random samples.

In actual programming, we don't want to generate new random numbers in each step, nor want to "waste" some samples -- we desire all training data can be used during SGD. It is very useful to adopt the "epoch-batch" strategy (or called cyclic rule) through permutation of the data.

> Choose initial guess β^0, step size (learning rate) η,
> batch size n_B, number of inner iterations $M \leq N/n_B$, number of epochs n_E

```
For epoch $n = 1, 2, \cdots, n_E$
    $\beta^0$ for the current epoch is $\beta^{M+1}$ for the previous epoch.
    Randomly shuffle the training samples.
    For $m = 0, 1, 2, \cdots, M-1$
        $\beta^{m+1} = \beta^m - \frac{\eta}{n_B} \sum_{i=1}^{n_B} \nabla_\beta l(\beta^m; x^{(m*n_B+i)}, y^{(m*n_B+i)})$
```

If the gradient loss of your program is written in a highly vectorized way (support data matrix as input), then you can simply make the data matrix within the mini-batch as the input in each GD update. Below is the example based on our previous binary logistic regression codes.

In practice, you may also find it helpful to adjust the stepsize (learning rate) during the iteration.

In [11]:

```
import numpy as np

class myLogisticRegression_binary():
    """ Logistic Regression classifier -- this only works for the binary case. Here we
    Parameters:
    -----------
    learning_rate: float
        The step length that will be taken when following the negative gradient during
        training.
    """
    def __init__(self, learning_rate=.001, opt_method = 'SGD', num_epochs = 50, size_ba

        # learning rate can also be in the fit method
        self.learning_rate = learning_rate
        self.opt_method = opt_method
        self.num_epochs = num_epochs
        self.size_batch = size_batch


    def fit(self, data, y, n_iterations = 1000):
        """
        don't forget the document string in methods
        """
        ones = np.ones((data.shape[0],1)) # column of ones
        X = np.concatenate((ones, data), axis = 1) # the augmented matrix, \tilde{X} in
        eta = self.learning_rate

        beta  = np.zeros(np.shape(X)[1]) # initialize beta, can be other choices

        if self.opt_method == 'GD':
            for k in range(n_iterations):
                dbeta = self.loss_gradient(beta,X,y) # write another function to comput
                beta = beta - eta * dbeta # the formula of GD
                # this step is optional -- just for inspection purposes
                if k % 500 == 0: # pprint loss every 50 steps
                    print("loss after", k+1, "iterations is: ", self.loss(beta,X,y))

        if self.opt_method == 'SGD':
            N = X.shape[0]
            num_epochs = self.num_epochs
            size_batch = self.size_batch
            num_iter = 0
            for e in range(num_epochs):
                shuffle_index = np.random.permutation(N) # in each epoch, we first resh
                for m in range(0,N,size_batch):   # m is the starting index of mini-bat
                    i = shuffle_index[m:m+size_batch] # index of samples in the mini-ba
                    dbeta = self.loss_gradient(beta,X[i,:],y[i]) # only use the data in
                    beta = beta - eta * dbeta # the formula of GD, but this time dbeta

                    if e % 1 == 0 and num_iter % 50 ==0: # print loss during the traini
                        print("loss after", e+1, "epochs and ", num_iter+1, "iterations

                    num_iter = num_iter +1  # number of total iterations

        self.coeff = beta

    def predict(self, data):
        ones = np.ones((data.shape[0],1)) # column of ones
        X = np.concatenate((ones, data), axis = 1) # the augmented matrix, \tilde{X} in
        beta = self.coeff # the estimated beta
        y_pred = np.round(self.sigmoid(np.dot(X,beta))).astype(int) # >0.5: ->1 else,->
        return y_pred

    def score(self, data, y_true):
        ones = np.ones((data.shape[0],1)) # column of ones
        X = np.concatenate((ones, data), axis = 1) # the augmented matrix, \tilde{X} in
        y_pred = self.predict(data)
        acc = np.mean(y_pred == y_true) # number of correct predictions/N
        return acc

    def sigmoid(self, z):
        return 1.0 / (1.0 + np.exp(-z))

    def loss(self,beta,X,y):
        f_value = self.sigmoid(np.matmul(X,beta))
        loss_value = np.log(f_value + 1e-10) * y + (1.0 - y)* np.log(1 - f_value + 1e-1
        return -np.mean(loss_value)

    def loss_gradient(self,beta,X,y):
        f_value = self.sigmoid(np.matmul(X,beta))
        gradient_value = (f_value - y).reshape(-1,1)*X # this is the hardest expression
        return np.mean(gradient_value, axis=0)
```

You will find adapting the SGD codes above to multi-class logistic regression is very helpful in doing your final project! (although it's not basic requirement). Here is the very intuitive argument when SGD can boost the algorithms.

Suppose in the training dataset you have $N = 60,000$ samples. With GD, each iteration will cost 60,000 summations. Now consider using SGD. We have the mini-batch size of 30. Then each iteration will cost only 30 sums. For a complete epoch, you have 60,000 sums -- the same with GD, but you have already iterated for 2000 steps!

Of course you may argue that the "quality" of steps in GD is "far better" than SGD. Surely there is the trade-off, but pratically the inferior performace of SGD in convergence does not obscure its super efficiency over GD. In fact, SGD is the de facto optimization method in deep learning. (SGD and BP -- backward propogation to calculate the gradient are the two fundamental cornerstones in deep learning.)

Next, we compare GD and SGD with the UCI "adult" dataset to predict income. Note that it is a binary classification problem.

In [12]:

```
import pandas as pd
df = pd.read_csv('adult.csv')
df
```

Out[12]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	ge
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	I
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	I
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	I
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	I
4	18	?	103497	Some-college	10	Never-married	?	Own-child	White	Fe
...	...	...	...	...	...	...	...	...	...	
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Fe
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	I
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Fe
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	I
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Fe

48842 rows × 15 columns

In [13]:

```
from numpy import nan
df = df.replace('?',nan) #dealing with missing values -- ? in original dataset
df.head()
```

Out[13]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male
4	18	NaN	103497	Some-college	10	Never-married	NaN	Own-child	White	Female

In [14]:

```
df.dropna(inplace = True) # drop missing values
df
```

Out[14]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	ge
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	M
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	M
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	M
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	M
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	M
...	...	...	...	...	...	...	...	...	...	
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Fe
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	M
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Fe

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	ge
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	I
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Fe

45222 rows × 15 columns

```
In [15]: df.drop(columns=['fnlwgt','native-country'], inplace=True) # drop some variables we are
         df
```

Out[15]:

	age	workclass	education	educational-num	marital-status	occupation	relationship	race	gender	ca
0	25	Private	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	
1	38	Private	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	
2	28	Local-gov	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	
3	44	Private	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	
5	34	Private	10th	6	Never-married	Other-service	Not-in-family	White	Male	
...	...	...	...	...	...	...	...	...	...	
48837	27	Private	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	
48838	40	Private	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	
48839	58	Private	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	
48840	22	Private	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	
48841	52	Self-emp-inc	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	1

45222 rows × 13 columns

In [16]:

```
from sklearn.preprocessing import LabelEncoder
df_clean = df.apply(LabelEncoder().fit_transform) # transform the categorical variables
df_clean
```

Out[16]:

	age	workclass	education	educational-num	marital-status	occupation	relationship	race	gender	capit ga
0	8	2	1	6	4	6	3	2	1	
1	21	2	11	8	2	4	0	4	1	
2	11	1	7	11	2	10	0	4	1	
3	27	2	15	9	2	6	0	2	1	
5	17	2	0	5	4	7	1	4	1	
...	...	...	...	...	...	...	...	...	...	
48837	10	2	7	11	2	12	5	4	0	
48838	23	2	11	8	2	6	0	4	1	
48839	41	2	11	8	6	0	4	4	0	
48840	5	2	11	8	4	0	3	4	1	
48841	35	3	11	8	2	3	5	4	0	1

45222 rows × 13 columns

Note that it is not best way to encode the data. Please see other solutions in kaggle.

In [17]:

```
y = df_clean['income'].to_numpy()
X = df_clean.drop(columns = 'income').to_numpy()
```

In [18]:

```
X.shape
```

Out[18]:

```
(45222, 12)
```

In [19]:

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.1, random_state=4
```

In [20]:

```
from sklearn.linear_model import LogisticRegression
clf = LogisticRegression(random_state=0)
clf.fit(X_train,y_train)
clf.score(X_test,y_test)
```

```
/Users/cliffzhou/opt/anaconda3/lib/python3.7/site-packages/sklearn/linear_model/_logisti
c.py:764: ConvergenceWarning: lbfgs failed to converge (status=1):
STOP: TOTAL NO. of ITERATIONS REACHED LIMIT.
```

```
Increase the number of iterations (max_iter) or scale the data as shown in:
    https://scikit-learn.org/stable/modules/preprocessing.html
Please also refer to the documentation for alternative solver options:
    https://scikit-learn.org/stable/modules/linear_model.html#logistic-regression
  extra_warning_msg=_LOGISTIC_SOLVER_CONVERGENCE_MSG)
```

Out[20]:

```
0.8271059031616184
```

In [21]:

```
lg_gd = myLogisticRegression_binary(learning_rate=1e-6, opt_method = 'GD')
lg_sgd = myLogisticRegression_binary(learning_rate=1e-6, opt_method = 'SGD', num_epochs
```

In [22]:

```
%%time
lg_gd.fit(X_train,y_train,n_iterations = 15000)
```

```
loss after 1 iterations is:  0.6930358550277247
loss after 501 iterations is:  0.6503339171382144
loss after 1001 iterations is:  0.6250322404153786
loss after 1501 iterations is:  0.6091127195017652
loss after 2001 iterations is:  0.5984037857262676
loss after 2501 iterations is:  0.590712724359857
loss after 3001 iterations is:  0.5848586907302407
loss after 3501 iterations is:  0.5801861202580018
loss after 4001 iterations is:  0.5763181444418489
loss after 4501 iterations is:  0.5730292982409765
loss after 5001 iterations is:  0.570178434214311
loss after 5501 iterations is:  0.567672659406349
loss after 6001 iterations is:  0.565447582899603
loss after 6501 iterations is:  0.5634562915787977
loss after 7001 iterations is:  0.5616630677823014
loss after 7501 iterations is:  0.5600397185713463
loss after 8001 iterations is:  0.5585633633136624
loss after 8501 iterations is:  0.5572150485499265
loss after 9001 iterations is:  0.555978841583727
loss after 9501 iterations is:  0.5548412083490464
loss after 10001 iterations is:  0.5537905657746117
loss after 10501 iterations is:  0.5528169456723574
loss after 11001 iterations is:  0.551911733237817
loss after 11501 iterations is:  0.5510674578925445
loss after 12001 iterations is:  0.5502776225312458
loss after 12501 iterations is:  0.5495365620648746
loss after 13001 iterations is:  0.5488393250200673
loss after 13501 iterations is:  0.548181573717374
loss after 14001 iterations is:  0.5475594996778428
loss after 14501 iterations is:  0.5469697516624197
CPU times: user 2min 43s, sys: 41.6 s, total: 3min 24s
Wall time: 1min 10s
```

In [21]:

```
lg_gd.score(X_test,y_test)
```

Out[21]:

```
0.7950475348220207
```

In [23]:

```
%%time
lg_sgd.fit(X_train,y_train)
```

```
loss after 1 epochs and  1 iterations is:  0.693073548355826
loss after 1 epochs and  51 iterations is:  0.6879767230703683
loss after 1 epochs and  101 iterations is:  0.6831400319332097
loss after 1 epochs and  151 iterations is:  0.6782938333245944
loss after 1 epochs and  201 iterations is:  0.6738200798571305
loss after 1 epochs and  251 iterations is:  0.6691604580202567
loss after 1 epochs and  301 iterations is:  0.6645376116485925
loss after 1 epochs and  351 iterations is:  0.6606361478074845
loss after 1 epochs and  401 iterations is:  0.656892918315997
loss after 1 epochs and  451 iterations is:  0.6537144465440301
loss after 1 epochs and  501 iterations is:  0.6502217595774284
loss after 1 epochs and  551 iterations is:  0.6467550892994749
loss after 1 epochs and  601 iterations is:  0.6439438227455281
loss after 1 epochs and  651 iterations is:  0.6408120175812363
loss after 1 epochs and  701 iterations is:  0.6382790133577214
loss after 1 epochs and  751 iterations is:  0.6360192108554918
loss after 1 epochs and  801 iterations is:  0.6336400621705309
loss after 1 epochs and  851 iterations is:  0.6314887292426813
loss after 1 epochs and  901 iterations is:  0.629224902743593
loss after 1 epochs and  951 iterations is:  0.6270099987848637
loss after 1 epochs and  1001 iterations is:  0.6249520491847131
loss after 2 epochs and  1051 iterations is:  0.6229116758183082
loss after 2 epochs and  1101 iterations is:  0.6212831042876761
loss after 2 epochs and  1151 iterations is:  0.6193908196086252
loss after 2 epochs and  1201 iterations is:  0.6177241208299401
loss after 2 epochs and  1251 iterations is:  0.6162042055069931
loss after 2 epochs and  1301 iterations is:  0.6145709527872213
loss after 2 epochs and  1351 iterations is:  0.6131581155661675
loss after 2 epochs and  1401 iterations is:  0.6116949752628326
loss after 2 epochs and  1451 iterations is:  0.6102120497983966
loss after 2 epochs and  1501 iterations is:  0.6089500726386283
loss after 2 epochs and  1551 iterations is:  0.6078478604732187
loss after 2 epochs and  1601 iterations is:  0.6068213747664613
loss after 2 epochs and  1651 iterations is:  0.6057195026797241
loss after 2 epochs and  1701 iterations is:  0.6045041398426849
loss after 2 epochs and  1751 iterations is:  0.6033560994836489
loss after 2 epochs and  1801 iterations is:  0.6023100788963595
loss after 2 epochs and  1851 iterations is:  0.6012733545684351
loss after 2 epochs and  1901 iterations is:  0.6002829435720342
loss after 2 epochs and  1951 iterations is:  0.5993031358414305
loss after 2 epochs and  2001 iterations is:  0.598390842556989
loss after 3 epochs and  2051 iterations is:  0.5975160994820534
loss after 3 epochs and  2101 iterations is:  0.596573367780993
loss after 3 epochs and  2151 iterations is:  0.5956454569869442
loss after 3 epochs and  2201 iterations is:  0.5948309105933165
loss after 3 epochs and  2251 iterations is:  0.5940447516378796
loss after 3 epochs and  2301 iterations is:  0.5932651157978089
loss after 3 epochs and  2351 iterations is:  0.5926116567827485
loss after 3 epochs and  2401 iterations is:  0.5919698844718679
loss after 3 epochs and  2451 iterations is:  0.5913029640607724
loss after 3 epochs and  2501 iterations is:  0.5905203572018709
loss after 3 epochs and  2551 iterations is:  0.5897900662362989
loss after 3 epochs and  2601 iterations is:  0.5891565216224129
loss after 3 epochs and  2651 iterations is:  0.5885962244515681
loss after 3 epochs and  2701 iterations is:  0.5880255740278464
loss after 3 epochs and  2751 iterations is:  0.5874504177057764
loss after 3 epochs and  2801 iterations is:  0.5868418518117547
loss after 3 epochs and  2851 iterations is:  0.5863365352575426
loss after 3 epochs and  2901 iterations is:  0.5858141606138647
loss after 3 epochs and  2951 iterations is:  0.5852860512219659
loss after 3 epochs and  3001 iterations is:  0.5848763506130045
loss after 3 epochs and  3051 iterations is:  0.584378348048583
loss after 4 epochs and  3101 iterations is:  0.5838700292886737
loss after 4 epochs and  3151 iterations is:  0.5833477097123569
loss after 4 epochs and  3201 iterations is:  0.5828471603818938
loss after 4 epochs and  3251 iterations is:  0.5823958079033964
loss after 4 epochs and  3301 iterations is:  0.5819519201848536
loss after 4 epochs and  3351 iterations is:  0.5815524116951639
loss after 4 epochs and  3401 iterations is:  0.5810680213294548
loss after 4 epochs and  3451 iterations is:  0.5806107822767308
loss after 4 epochs and  3501 iterations is:  0.5801790814103686
loss after 4 epochs and  3551 iterations is:  0.5797598279909602
loss after 4 epochs and  3601 iterations is:  0.579355338262248
loss after 4 epochs and  3651 iterations is:  0.5789986047935279
loss after 4 epochs and  3701 iterations is:  0.578596885989125
loss after 4 epochs and  3751 iterations is:  0.578230202009241
loss after 4 epochs and  3801 iterations is:  0.5778467911902169
loss after 4 epochs and  3851 iterations is:  0.5774241917766875
loss after 4 epochs and  3901 iterations is:  0.5770542553213098
loss after 4 epochs and  3951 iterations is:  0.5766868421969693
loss after 4 epochs and  4001 iterations is:  0.576393692859817
loss after 4 epochs and  4051 iterations is:  0.5760186618157042
loss after 5 epochs and  4101 iterations is:  0.5756884977011725
loss after 5 epochs and  4151 iterations is:  0.5753351152463163
loss after 5 epochs and  4201 iterations is:  0.5750617169541532
loss after 5 epochs and  4251 iterations is:  0.574746251628778
loss after 5 epochs and  4301 iterations is:  0.5744542273758508
loss after 5 epochs and  4351 iterations is:  0.5741328526257887
loss after 5 epochs and  4401 iterations is:  0.5737993527662487
loss after 5 epochs and  4451 iterations is:  0.5734645448888471
loss after 5 epochs and  4501 iterations is:  0.5731236491492427
loss after 5 epochs and  4551 iterations is:  0.5727792271053392
loss after 5 epochs and  4601 iterations is:  0.5724779632628709
loss after 5 epochs and  4651 iterations is:  0.5721691465303221
loss after 5 epochs and  4701 iterations is:  0.5718303661332882
loss after 5 epochs and  4751 iterations is:  0.5715314154776104
loss after 5 epochs and  4801 iterations is:  0.571265513586129
loss after 5 epochs and  4851 iterations is:  0.5709722108892195
loss after 5 epochs and  4901 iterations is:  0.5706688991634383
loss after 5 epochs and  4951 iterations is:  0.5704726723870194
loss after 5 epochs and  5001 iterations is:  0.5701889054903357
loss after 5 epochs and  5051 iterations is:  0.5699401224006977
loss after 6 epochs and  5101 iterations is:  0.5696908738253946
loss after 6 epochs and  5151 iterations is:  0.5694297403182451
loss after 6 epochs and  5201 iterations is:  0.5691455780432414
loss after 6 epochs and  5251 iterations is:  0.5688834817681628
loss after 6 epochs and  5301 iterations is:  0.5686751536730287
loss after 6 epochs and  5351 iterations is:  0.5684235024314268
loss after 6 epochs and  5401 iterations is:  0.5682219228874507
loss after 6 epochs and  5451 iterations is:  0.5679499228979983
loss after 6 epochs and  5501 iterations is:  0.5676976496320318
loss after 6 epochs and  5551 iterations is:  0.567471916807015
loss after 6 epochs and  5601 iterations is:  0.5672117339884091
loss after 6 epochs and  5651 iterations is:  0.5670219415288704
loss after 6 epochs and  5701 iterations is:  0.5667685935233233
loss after 6 epochs and  5751 iterations is:  0.5665066561714743
loss after 6 epochs and  5801 iterations is:  0.5663263076259891
loss after 6 epochs and  5851 iterations is:  0.5661189776750595
loss after 6 epochs and  5901 iterations is:  0.5658790683665929
loss after 6 epochs and  5951 iterations is:  0.5656823459174787
loss after 6 epochs and  6001 iterations is:  0.5654745467782918
loss after 6 epochs and  6051 iterations is:  0.5652457598813172
loss after 6 epochs and  6101 iterations is:  0.5650369244770606
loss after 7 epochs and  6151 iterations is:  0.5647804685817911
loss after 7 epochs and  6201 iterations is:  0.5645982013150546
loss after 7 epochs and  6251 iterations is:  0.5643736586634753
loss after 7 epochs and  6301 iterations is:  0.5641268763428908
loss after 7 epochs and  6351 iterations is:  0.5639055400467494
loss after 7 epochs and  6401 iterations is:  0.5637039744266981
loss after 7 epochs and  6451 iterations is:  0.5635033416847027
loss after 7 epochs and  6501 iterations is:  0.5633386362618245
loss after 7 epochs and  6551 iterations is:  0.5631954256344996
loss after 7 epochs and  6601 iterations is:  0.563000284259968
loss after 7 epochs and  6651 iterations is:  0.5628218695868835
loss after 7 epochs and  6701 iterations is:  0.562663176055141
loss after 7 epochs and  6751 iterations is:  0.5624954851410792
loss after 7 epochs and  6801 iterations is:  0.5623123245823365
loss after 7 epochs and  6851 iterations is:  0.5621484592274941
loss after 7 epochs and  6901 iterations is:  0.5619590644077744
loss after 7 epochs and  6951 iterations is:  0.56178162318672
loss after 7 epochs and  7001 iterations is:  0.5616257153084493
loss after 7 epochs and  7051 iterations is:  0.5614771087509546
loss after 7 epochs and  7101 iterations is:  0.5613138750443128
loss after 8 epochs and  7151 iterations is:  0.561160378795533
loss after 8 epochs and  7201 iterations is:  0.5609925466344351
loss after 8 epochs and  7251 iterations is:  0.5608225169329373
loss after 8 epochs and  7301 iterations is:  0.5606909768529942
loss after 8 epochs and  7351 iterations is:  0.560520184634483
loss after 8 epochs and  7401 iterations is:  0.5603485541007089
loss after 8 epochs and  7451 iterations is:  0.5602044305689273
loss after 8 epochs and  7501 iterations is:  0.5600343805881499
loss after 8 epochs and  7551 iterations is:  0.5599015399985472
loss after 8 epochs and  7601 iterations is:  0.5597639983223104
loss after 8 epochs and  7651 iterations is:  0.5596236881184197
loss after 8 epochs and  7701 iterations is:  0.5594959587769343
loss after 8 epochs and  7751 iterations is:  0.5593578432444392
loss after 8 epochs and  7801 iterations is:  0.5591941255183607
loss after 8 epochs and  7851 iterations is:  0.5590145641416246
loss after 8 epochs and  7901 iterations is:  0.5588712099122911
loss after 8 epochs and  7951 iterations is:  0.558717416167886
loss after 8 epochs and  8001 iterations is:  0.5585914031133321
loss after 8 epochs and  8051 iterations is:  0.5584296437497707
loss after 8 epochs and  8101 iterations is:  0.5582985944392457
loss after 9 epochs and  8151 iterations is:  0.5581666895521233
loss after 9 epochs and  8201 iterations is:  0.5580250045406994
loss after 9 epochs and  8251 iterations is:  0.5579024319457301
loss after 9 epochs and  8301 iterations is:  0.5577693900214433
loss after 9 epochs and  8351 iterations is:  0.5576393571397954
loss after 9 epochs and  8401 iterations is:  0.5575075124035843
loss after 9 epochs and  8451 iterations is:  0.5573741971420589
loss after 9 epochs and  8501 iterations is:  0.5572315418083459
loss after 9 epochs and  8551 iterations is:  0.5571115461159141
loss after 9 epochs and  8601 iterations is:  0.5569929231944509
loss after 9 epochs and  8651 iterations is:  0.5568556423201667
loss after 9 epochs and  8701 iterations is:  0.5567342088214443
loss after 9 epochs and  8751 iterations is:  0.5565982458112914
loss after 9 epochs and  8801 iterations is:  0.556482551298681
loss after 9 epochs and  8851 iterations is:  0.556377297589127
loss after 9 epochs and  8901 iterations is:  0.5562522172070071
loss after 9 epochs and  8951 iterations is:  0.5561459247523323
loss after 9 epochs and  9001 iterations is:  0.5560239931162884
loss after 9 epochs and  9051 iterations is:  0.5558984328305816
loss after 9 epochs and  9101 iterations is:  0.5557698970586893
loss after 9 epochs and  9151 iterations is:  0.5556415580100784
loss after 10 epochs and  9201 iterations is:  0.5555056479247267
loss after 10 epochs and  9251 iterations is:  0.5553813972114405
loss after 10 epochs and  9301 iterations is:  0.5552759462681619
loss after 10 epochs and  9351 iterations is:  0.5551611332285509
loss after 10 epochs and  9401 iterations is:  0.5550609589372639
loss after 10 epochs and  9451 iterations is:  0.5549423482550363
loss after 10 epochs and  9501 iterations is:  0.5548294450445027
loss after 10 epochs and  9551 iterations is:  0.5547009676280864
loss after 10 epochs and  9601 iterations is:  0.5545928248240456
loss after 10 epochs and  9651 iterations is:  0.554491652927073
loss after 10 epochs and  9701 iterations is:  0.5544016886595462
loss after 10 epochs and  9751 iterations is:  0.5543196254392383
loss after 10 epochs and  9801 iterations is:  0.5541910197093372
loss after 10 epochs and  9851 iterations is:  0.5540700635897741
loss after 10 epochs and  9901 iterations is:  0.5539909906448339
loss after 10 epochs and  9951 iterations is:  0.5538887589244924
loss after 10 epochs and  10001 iterations is:  0.5537889237893255
loss after 10 epochs and  10051 iterations is:  0.5537049759373921
loss after 10 epochs and  10101 iterations is:  0.5536103035191939
loss after 10 epochs and  10151 iterations is:  0.5535040550680596
loss after 11 epochs and  10201 iterations is:  0.553409445081937
loss after 11 epochs and  10251 iterations is:  0.5533127029167191
loss after 11 epochs and  10301 iterations is:  0.5532304821051374
loss after 11 epochs and  10351 iterations is:  0.5531368581026818
loss after 11 epochs and  10401 iterations is:  0.5530530072376784
loss after 11 epochs and  10451 iterations is:  0.552953895198904
loss after 11 epochs and  10501 iterations is:  0.5528639187226405
loss after 11 epochs and  10551 iterations is:  0.5527469787405459
loss after 11 epochs and  10601 iterations is:  0.5526754165549996
loss after 11 epochs and  10651 iterations is:  0.5525853218082397
loss after 11 epochs and  10701 iterations is:  0.5524952962170938
loss after 11 epochs and  10751 iterations is:  0.5524171503102968
loss after 11 epochs and  10801 iterations is:  0.5523260494282531
loss after 11 epochs and  10851 iterations is:  0.5522417337398066
loss after 11 epochs and  10901 iterations is:  0.5521592077409282
loss after 11 epochs and  10951 iterations is:  0.5520556943718499
loss after 11 epochs and  11001 iterations is:  0.551959136054298
loss after 11 epochs and  11051 iterations is:  0.5518641365982015
loss after 11 epochs and  11101 iterations is:  0.551773433023146
loss after 11 epochs and  11151 iterations is:  0.5516809294559364
loss after 12 epochs and  11201 iterations is:  0.5515726887557071
loss after 12 epochs and  11251 iterations is:  0.5514786867422341
loss after 12 epochs and  11301 iterations is:  0.5514001279390471
loss after 12 epochs and  11351 iterations is:  0.5513369330400827
loss after 12 epochs and  11401 iterations is:  0.5512593403129692
loss after 12 epochs and  11451 iterations is:  0.5511835657971452
loss after 12 epochs and  11501 iterations is:  0.551100376313963
loss after 12 epochs and  11551 iterations is:  0.5510076797301039
loss after 12 epochs and  11601 iterations is:  0.5509296278704177
loss after 12 epochs and  11651 iterations is:  0.5508434290425478
loss after 12 epochs and  11701 iterations is:  0.5507509175726032
loss after 12 epochs and  11751 iterations is:  0.5506727801513763
loss after 12 epochs and  11801 iterations is:  0.5506079744097107
loss after 12 epochs and  11851 iterations is:  0.5505315105149775
loss after 12 epochs and  11901 iterations is:  0.5504462991254849
loss after 12 epochs and  11951 iterations is:  0.5503635596779508
loss after 12 epochs and  12001 iterations is:  0.5502908111831807
loss after 12 epochs and  12051 iterations is:  0.5502150627264949
loss after 12 epochs and  12101 iterations is:  0.5501365626718582
loss after 12 epochs and  12151 iterations is:  0.5500405190343785
loss after 12 epochs and  12201 iterations is:  0.5499730858850065
loss after 13 epochs and  12251 iterations is:  0.5498907693561292
loss after 13 epochs and  12301 iterations is:  0.5498214206877122
loss after 13 epochs and  12351 iterations is:  0.5497415932293183
loss after 13 epochs and  12401 iterations is:  0.5496670322231366
loss after 13 epochs and  12451 iterations is:  0.5495827368068428
loss after 13 epochs and  12501 iterations is:  0.5495294313794172
loss after 13 epochs and  12551 iterations is:  0.5494639082190642
loss after 13 epochs and  12601 iterations is:  0.5493860232639352
loss after 13 epochs and  12651 iterations is:  0.5493218719474573
loss after 13 epochs and  12701 iterations is:  0.5492430095336602
loss after 13 epochs and  12751 iterations is:  0.5491826885479171
loss after 13 epochs and  12801 iterations is:  0.5491127312840449
loss after 13 epochs and  12851 iterations is:  0.5490531660004251
loss after 13 epochs and  12901 iterations is:  0.5489918542309454
loss after 13 epochs and  12951 iterations is:  0.5489338051095347
loss after 13 epochs and  13001 iterations is:  0.5488772733380319
loss after 13 epochs and  13051 iterations is:  0.5487873235764463
loss after 13 epochs and  13101 iterations is:  0.5487070778825115
loss after 13 epochs and  13151 iterations is:  0.5486424223349826
loss after 13 epochs and  13201 iterations is:  0.5485679989701077
loss after 14 epochs and  13251 iterations is:  0.5485101753565144
loss after 14 epochs and  13301 iterations is:  0.5484547933990089
loss after 14 epochs and  13351 iterations is:  0.5483975753068677
loss after 14 epochs and  13401 iterations is:  0.5483358242570168
loss after 14 epochs and  13451 iterations is:  0.5482608613473307
loss after 14 epochs and  13501 iterations is:  0.5481918580556746
loss after 14 epochs and  13551 iterations is:  0.5481283433952536
loss after 14 epochs and  13601 iterations is:  0.5480528097776642
loss after 14 epochs and  13651 iterations is:  0.5479872429495423
loss after 14 epochs and  13701 iterations is:  0.5479271590782284
loss after 14 epochs and  13751 iterations is:  0.5478584027258118
loss after 14 epochs and  13801 iterations is:  0.5478015693266427
loss after 14 epochs and  13851 iterations is:  0.5477467527118464
loss after 14 epochs and  13901 iterations is:  0.5476967653017768
loss after 14 epochs and  13951 iterations is:  0.5476507252698889
loss after 14 epochs and  14001 iterations is:  0.547581162550019
loss after 14 epochs and  14051 iterations is:  0.5475087900496111
loss after 14 epochs and  14101 iterations is:  0.5474514743583333
loss after 14 epochs and  14151 iterations is:  0.5473794731018552
loss after 14 epochs and  14201 iterations is:  0.5473216388965041
loss after 14 epochs and  14251 iterations is:  0.5472619510410491
loss after 15 epochs and  14301 iterations is:  0.5472121869605444
loss after 15 epochs and  14351 iterations is:  0.5471516667916172
loss after 15 epochs and  14401 iterations is:  0.547093675659208
loss after 15 epochs and  14451 iterations is:  0.5470409489843366
loss after 15 epochs and  14501 iterations is:  0.5469835820745677
loss after 15 epochs and  14551 iterations is:  0.5469319152019122
loss after 15 epochs and  14601 iterations is:  0.5468704474441543
loss after 15 epochs and  14651 iterations is:  0.5468291603133646
loss after 15 epochs and  14701 iterations is:  0.546769425284483
loss after 15 epochs and  14751 iterations is:  0.5467182162171642
loss after 15 epochs and  14801 iterations is:  0.5466556323420779
loss after 15 epochs and  14851 iterations is:  0.5465911980875728
loss after 15 epochs and  14901 iterations is:  0.5465400847356786
loss after 15 epochs and  14951 iterations is:  0.5464880209636221
loss after 15 epochs and  15001 iterations is:  0.5464379994241739
loss after 15 epochs and  15051 iterations is:  0.5463670150011595
loss after 15 epochs and  15101 iterations is:  0.5463055231672244
loss after 15 epochs and  15151 iterations is:  0.5462490917029125
loss after 15 epochs and  15201 iterations is:  0.5461981939463999
loss after 15 epochs and  15251 iterations is:  0.5461416856768063
CPU times: user 7.32 s, sys: 3.19 s, total: 10.5 s
Wall time: 3.73 s
```

In [23]:

```
lg_sgd.score(X_test,y_test)
```

Out[23]:

```
0.7950475348220207
```

Reference Reading Suggestions

- ISLR: Chapter 4
- ESL: Chapter 4
- PML: Chapter 10