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

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

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:

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

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

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

However, this is a 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) We create a sequence of vectors $\{\beta^0, \beta^1, \beta^2, \dots\}$ using the following recursive rule:

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

Here the step size η (eta) 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 [5]:

```
%%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: 1.48 s
```

In [6]:

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

Out[6]:

```
1.0
```

In [7]:

```
lg.score(X_train,y_train)
```

Out[7]:

```
0.9140625
```

In [8]:

```
lg.predict(X_test)
```

Out[8]:

```
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 [9]:

```
lg.coeff
```

Out[9]:

```
array([ 1.92156972e-03,  1.35014099e-02,  5.60837090e-03,  6.98616548e-02,
        1.63274570e-02,  8.11846822e-05, -3.05842439e-04, -5.82893624e-04,
       -2.34074392e-04,  1.52008899e-04,  8.23455085e-05,  1.19749200e-04,
        7.36438347e-04, -9.69461818e-04, -2.25609056e-02,  1.60975351e-06,
       -8.32345494e-05, -1.14621433e-04, -2.63019486e-05,  7.14323607e-06,
       -3.90033063e-06,  1.41429198e-02,  1.95665028e-03,  6.13141823e-02,
       -2.82368244e-02,  6.40567132e-05, -1.16724642e-03, -1.62789172e-03,
       -4.19315981e-04,  6.01178429e-05,  3.79852714e-06])
```

In [10]:

```
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[10]:
```
0.9824561403508771
```

In [11]:
```python
clf.score(X_train,y_train)
```

Out[11]:
```
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, $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K$, 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 $\mathbf{w}_1, \mathbf{w}_2, \dots, \mathbf{w}_K$ into columns, so that $\mathbf{w}_k = (w_{k0}, \dots, w_{kp})^\top \in \mathbb{R}^{p+1}$.

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

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

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

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

or element-wise written as

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

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

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

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

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

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):

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

Other notations for indicators: $1_{\{k\}}(y)$, $\mathbb{I}_{\{k\}}(y)$, $I_{\{k\}}(y)$

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 (in the notation of Matrix calculus):

$$\frac{\partial L}{\partial \mathbf{w}_k} = \left(\frac{\partial L}{\partial w_{k0}}, \frac{\partial L}{\partial w_{k1}}, \cdots \frac{\partial L}{\partial w_{kP}} \right)^T = \frac{1}{N} \sum_{i=1}^N \left(\frac{\exp(\tilde{x}^{(i)} \mathbf{w}_k)}{\sum_{m=1}^K \exp(\tilde{x}^{(i)} \mathbf{w}_m)} - 1_{\{y^{(i)}=k\}} \right) \tilde{x}^{(i)} \in \mathbb{R}^{p+1}.$$

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.

$$dW = \begin{pmatrix} | & | & | & | \\ \frac{\partial L}{\partial \mathbf{w}_1} & \frac{\partial L}{\partial \mathbf{w}_2} & \cdots & \frac{\partial L}{\partial \mathbf{w}_K} \\ | & | & | & | \end{pmatrix},$$

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 [12]:

```
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 [13]:

```
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]:

```
import pandas as pd
df = pd.DataFrame(X)
df
```

Out[26]:

	0	1	2	3	4	5	6	7	8	9	...	54	55	56	57	58	59	60	61	62
0	0.0	0.0	5.0	13.0	9.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	6.0	13.0	10.0	0.0	0.0
1	0.0	0.0	0.0	12.0	13.0	5.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	11.0	16.0	10.0	0.0
2	0.0	0.0	0.0	4.0	15.0	12.0	0.0	0.0	0.0	0.0	...	5.0	0.0	0.0	0.0	0.0	3.0	11.0	16.0	9.0
3	0.0	0.0	7.0	15.0	13.0	1.0	0.0	0.0	0.0	8.0	...	9.0	0.0	0.0	0.0	7.0	13.0	13.0	9.0	0.0
4	0.0	0.0	0.0	1.0	11.0	0.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	0.0	2.0	16.0	4.0	0.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
1792	0.0	0.0	4.0	10.0	13.0	6.0	0.0	0.0	0.0	1.0	...	4.0	0.0	0.0	0.0	2.0	14.0	15.0	9.0	0.0
1793	0.0	0.0	6.0	16.0	13.0	11.0	1.0	0.0	0.0	0.0	...	1.0	0.0	0.0	0.0	6.0	16.0	14.0	6.0	0.0
1794	0.0	0.0	1.0	11.0	15.0	1.0	0.0	0.0	0.0	0.0	...	0.0	0.0	0.0	0.0	2.0	9.0	13.0	6.0	0.0
1795	0.0	0.0	2.0	10.0	7.0	0.0	0.0	0.0	0.0	0.0	...	2.0	0.0	0.0	0.0	5.0	12.0	16.0	12.0	0.0
1796	0.0	0.0	10.0	14.0	8.0	1.0	0.0	0.0	0.0	2.0	...	8.0	0.0	0.0	1.0	8.0	12.0	14.0	12.0	1.0

1797 rows × 64 columns

In [27]:

```
X_train.shape
```

Out[27]:

```
(1617, 64)
```

In [15]:

```
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 [ ]:

```
lg.coeff
```

In [25]:

```
lg.coeff.shape #65 coefficients, 1 per row of data plus the constant term. 10 functions
```

Out[25]:

```
(65, 10)
```

In [28]:

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

Out[28]:

```
0.9722222222222222
```

In [18]:

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

Out[18]:

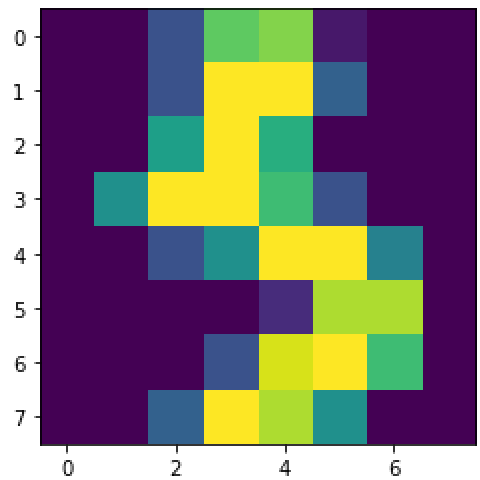
```
(array([  5,  71, 133, 149, 159], dtype=int64),)
```

In [36]:

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

Out[36]:

```
<matplotlib.image.AxesImage at 0x1970d4e3700>
```



In [37]:

```
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)
```

First row: seventeen 0s were classified as 0

Second row: ten 1s were classified as 2, one 1 was classified as 2

Exercise: Determine what the rest of the rows tell us!

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 $n = 1, 2, \cdots, n_E$
> $\quad\beta^0$ for the current epoch is β^{M+1} for the previous epoch.
> $\quad$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 (it supports a data matrix as the 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 η) during the iteration.

In [49]:

```
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 [38]:

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

Out[38]:

	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 [39]:

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

Out[39]:

	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

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender
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 [40]:

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

Out[40]:

	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
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	I
...	...	...	...	...	...	...	...	...	...	
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

45222 rows × 15 columns

In [41]:
```
df.drop(columns=['fnlwgt','native-country'], inplace=True) # drop some variables we are
df
```

Out[41]:

	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 [42]:
```
from sklearn.preprocessing import LabelEncoder
df_clean = df.apply(LabelEncoder().fit_transform) # transform the categorical variables
df_clean
```

Out[42]:

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

	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 [43]:
```
y = df_clean['income'].to_numpy()
X = df_clean.drop(columns = 'income').to_numpy()
```

In [44]:
```
X.shape
```

Out[44]:
```
(45222, 12)
```

In [45]:
```
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 [46]:
```
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[46]:
```
0.8275480875525094
```

In [50]:
```
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 [51]:
```
%%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:  0.5984037857262677
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:  0.5537905657746116
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
Wall time: 58.3 s
```

In [52]:
```
lg_gd.score(X_test,y_test)
```

Out[52]: 0.7950475348220207

In [53]:
```
%%time
lg_sgd.fit(X_train,y_train)
```

```
loss after 1 epochs and  1 iterations is:  0.6930446274473355
loss after 1 epochs and  51 iterations is:  0.687515883924888
loss after 1 epochs and  101 iterations is:  0.6819528818185219
loss after 1 epochs and  151 iterations is:  0.6769620866757833
loss after 1 epochs and  201 iterations is:  0.6723501613645855
loss after 1 epochs and  251 iterations is:  0.6682592623950855
loss after 1 epochs and  301 iterations is:  0.6642360552228926
loss after 1 epochs and  351 iterations is:  0.6601613906128302
loss after 1 epochs and  401 iterations is:  0.6566201750596257
loss after 1 epochs and  451 iterations is:  0.652730360019247
loss after 1 epochs and  501 iterations is:  0.6498211370468161
loss after 1 epochs and  551 iterations is:  0.6468680517108509
loss after 1 epochs and  601 iterations is:  0.6436935165745173
loss after 1 epochs and  651 iterations is:  0.6407866144425896
loss after 1 epochs and  701 iterations is:  0.6382603199009632
```

```
loss after 1 epochs and  751 iterations is:  0.636008759459535
loss after 1 epochs and  801 iterations is:  0.6336618919144462
loss after 1 epochs and  851 iterations is:  0.6315832416074683
loss after 1 epochs and  901 iterations is:  0.6295746711352063
loss after 1 epochs and  951 iterations is:  0.6272922362443611
loss after 1 epochs and  1001 iterations is:  0.625108782921287
loss after 2 epochs and  1051 iterations is:  0.6232631007398449
loss after 2 epochs and  1101 iterations is:  0.6214774002614993
loss after 2 epochs and  1151 iterations is:  0.6198723051316822
loss after 2 epochs and  1201 iterations is:  0.6180071671027185
loss after 2 epochs and  1251 iterations is:  0.6162725067634561
loss after 2 epochs and  1301 iterations is:  0.614760256673246З
loss after 2 epochs and  1351 iterations is:  0.6133615374721474
loss after 2 epochs and  1401 iterations is:  0.6118680987889064
loss after 2 epochs and  1451 iterations is:  0.6102182688235627
loss after 2 epochs and  1501 iterations is:  0.6087468329058434
loss after 2 epochs and  1551 iterations is:  0.6076169492968027
loss after 2 epochs and  1601 iterations is:  0.6064432686975922
loss after 2 epochs and  1651 iterations is:  0.6053274985542891
loss after 2 epochs and  1701 iterations is:  0.6043298159779605
loss after 2 epochs and  1751 iterations is:  0.603416200160567
loss after 2 epochs and  1801 iterations is:  0.6022930105473528
loss after 2 epochs and  1851 iterations is:  0.6011931776669032
loss after 2 epochs and  1901 iterations is:  0.6002560544920121
loss after 2 epochs and  1951 iterations is:  0.5992294830390025
loss after 2 epochs and  2001 iterations is:  0.5983608435220206
loss after 3 epochs and  2051 iterations is:  0.5976278378133407
loss after 3 epochs and  2101 iterations is:  0.5966847638437527
loss after 3 epochs and  2151 iterations is:  0.595931340946652
loss after 3 epochs and  2201 iterations is:  0.5951409662777318
loss after 3 epochs and  2251 iterations is:  0.5943440647133814
loss after 3 epochs and  2301 iterations is:  0.5935919277912972
loss after 3 epochs and  2351 iterations is:  0.5928974404858852
loss after 3 epochs and  2401 iterations is:  0.5921235137437189
loss after 3 epochs and  2451 iterations is:  0.5913350036355355
loss after 3 epochs and  2501 iterations is:  0.5907151209267606
loss after 3 epochs and  2551 iterations is:  0.5902001309313027
loss after 3 epochs and  2601 iterations is:  0.58962563131111
loss after 3 epochs and  2651 iterations is:  0.5891432751880878
loss after 3 epochs and  2701 iterations is:  0.5883979462883135
loss after 3 epochs and  2751 iterations is:  0.5877519340334483
loss after 3 epochs and  2801 iterations is:  0.5870992148901246
loss after 3 epochs and  2851 iterations is:  0.586422858289803
loss after 3 epochs and  2901 iterations is:  0.5858839093815912
loss after 3 epochs and  2951 iterations is:  0.5853570927309439
loss after 3 epochs and  3001 iterations is:  0.5848843142058823
loss after 3 epochs and  3051 iterations is:  0.5843596342014735
loss after 4 epochs and  3101 iterations is:  0.5838692630965587
loss after 4 epochs and  3151 iterations is:  0.583431356183656
loss after 4 epochs and  3201 iterations is:  0.58296558129837
loss after 4 epochs and  3251 iterations is:  0.5824615188960715
loss after 4 epochs and  3301 iterations is:  0.5820190711114429
loss after 4 epochs and  3351 iterations is:  0.5815755389573946
loss after 4 epochs and  3401 iterations is:  0.5810805940105698
loss after 4 epochs and  3451 iterations is:  0.5806770174138711
loss after 4 epochs and  3501 iterations is:  0.5802844161649041
loss after 4 epochs and  3551 iterations is:  0.5799207726267963
loss after 4 epochs and  3601 iterations is:  0.5795626343798965
loss after 4 epochs and  3651 iterations is:  0.5791168005222731
loss after 4 epochs and  3701 iterations is:  0.5787030719929944
loss after 4 epochs and  3751 iterations is:  0.578294996185845
loss after 4 epochs and  3801 iterations is:  0.5778271673208414
loss after 4 epochs and  3851 iterations is:  0.577401633683617
loss after 4 epochs and  3901 iterations is:  0.5770713723698572
loss after 4 epochs and  3951 iterations is:  0.5766735972744536
```

```
loss after 4 epochs and  4001 iterations is:  0.5763175142443229
loss after 4 epochs and  4051 iterations is:  0.5759630725165199
loss after 5 epochs and  4101 iterations is:  0.5756178797202284
loss after 5 epochs and  4151 iterations is:  0.5752832583167655
loss after 5 epochs and  4201 iterations is:  0.5749560309637735
loss after 5 epochs and  4251 iterations is:  0.5746007312061568
loss after 5 epochs and  4301 iterations is:  0.5742820974319747
loss after 5 epochs and  4351 iterations is:  0.5739398915897385
loss after 5 epochs and  4401 iterations is:  0.5736618087191695
loss after 5 epochs and  4451 iterations is:  0.5733228142049586
loss after 5 epochs and  4501 iterations is:  0.5730079951827857
loss after 5 epochs and  4551 iterations is:  0.5726795241634242
loss after 5 epochs and  4601 iterations is:  0.5723635339627089
loss after 5 epochs and  4651 iterations is:  0.5721060185131482
loss after 5 epochs and  4701 iterations is:  0.5717964692328318
loss after 5 epochs and  4751 iterations is:  0.5715394143844558
loss after 5 epochs and  4801 iterations is:  0.5712312202369444
loss after 5 epochs and  4851 iterations is:  0.5709702041000977
loss after 5 epochs and  4901 iterations is:  0.5707281537364952
loss after 5 epochs and  4951 iterations is:  0.5704702136225688
loss after 5 epochs and  5001 iterations is:  0.5702415529670348
loss after 5 epochs and  5051 iterations is:  0.5699502211282027
loss after 6 epochs and  5101 iterations is:  0.5696365453064862
loss after 6 epochs and  5151 iterations is:  0.5693782708141455
loss after 6 epochs and  5201 iterations is:  0.5691379914826566
loss after 6 epochs and  5251 iterations is:  0.5688756968701085
loss after 6 epochs and  5301 iterations is:  0.5686476743079197
loss after 6 epochs and  5351 iterations is:  0.5684520594682319
loss after 6 epochs and  5401 iterations is:  0.5681953966775559
loss after 6 epochs and  5451 iterations is:  0.5678968956318902
loss after 6 epochs and  5501 iterations is:  0.5676614216748869
loss after 6 epochs and  5551 iterations is:  0.5674291936361396
loss after 6 epochs and  5601 iterations is:  0.5672182281229884
loss after 6 epochs and  5651 iterations is:  0.5669559866258778
loss after 6 epochs and  5701 iterations is:  0.5667384417113279
loss after 6 epochs and  5751 iterations is:  0.5665212936449642
loss after 6 epochs and  5801 iterations is:  0.566278163340694
loss after 6 epochs and  5851 iterations is:  0.5660963178301629
loss after 6 epochs and  5901 iterations is:  0.5658636674086596
loss after 6 epochs and  5951 iterations is:  0.5656506015716293
loss after 6 epochs and  6001 iterations is:  0.5654436388280104
loss after 6 epochs and  6051 iterations is:  0.5652256019698314
loss after 6 epochs and  6101 iterations is:  0.5650312678110161
loss after 7 epochs and  6151 iterations is:  0.5647726581403855
loss after 7 epochs and  6201 iterations is:  0.5645949673481984
loss after 7 epochs and  6251 iterations is:  0.5644167147585311
loss after 7 epochs and  6301 iterations is:  0.5642247294279235
loss after 7 epochs and  6351 iterations is:  0.5640257089501122
loss after 7 epochs and  6401 iterations is:  0.5638218478513739
loss after 7 epochs and  6451 iterations is:  0.563629072781298
loss after 7 epochs and  6501 iterations is:  0.5634544908303338
loss after 7 epochs and  6551 iterations is:  0.5632747188295091
loss after 7 epochs and  6601 iterations is:  0.5631190422408741
loss after 7 epochs and  6651 iterations is:  0.5629674994365236
loss after 7 epochs and  6701 iterations is:  0.5627687107518358
loss after 7 epochs and  6751 iterations is:  0.5625760037920379
loss after 7 epochs and  6801 iterations is:  0.56240220098319
loss after 7 epochs and  6851 iterations is:  0.5622134196815317
loss after 7 epochs and  6901 iterations is:  0.5620564581182051
loss after 7 epochs and  6951 iterations is:  0.5618954469355816
loss after 7 epochs and  7001 iterations is:  0.5617124411245027
loss after 7 epochs and  7051 iterations is:  0.561519067976271
loss after 7 epochs and  7101 iterations is:  0.5613299349037445
loss after 8 epochs and  7151 iterations is:  0.5611455570777094
loss after 8 epochs and  7201 iterations is:  0.5609755112775617
```

```
loss after 8 epochs and  7251 iterations is:  0.5608093895370861
loss after 8 epochs and  7301 iterations is:  0.5606472552944759
loss after 8 epochs and  7351 iterations is:  0.5604887371324387
loss after 8 epochs and  7401 iterations is:  0.5603256767859882
loss after 8 epochs and  7451 iterations is:  0.5601825173115164
loss after 8 epochs and  7501 iterations is:  0.5600516789728943
loss after 8 epochs and  7551 iterations is:  0.5599074238350895
loss after 8 epochs and  7601 iterations is:  0.5597327518460828
loss after 8 epochs and  7651 iterations is:  0.5595939804635975
loss after 8 epochs and  7701 iterations is:  0.5594294564660448
loss after 8 epochs and  7751 iterations is:  0.5592949328533978
loss after 8 epochs and  7801 iterations is:  0.5591341087053963
loss after 8 epochs and  7851 iterations is:  0.559008098846671
loss after 8 epochs and  7901 iterations is:  0.5588525885314719
loss after 8 epochs and  7951 iterations is:  0.5587047632607972
loss after 8 epochs and  8001 iterations is:  0.5585626246786471
loss after 8 epochs and  8051 iterations is:  0.5584137005331373
loss after 8 epochs and  8101 iterations is:  0.5582710618090215
loss after 9 epochs and  8151 iterations is:  0.558144522989445
loss after 9 epochs and  8201 iterations is:  0.5580092346049592
loss after 9 epochs and  8251 iterations is:  0.5578840500986814
loss after 9 epochs and  8301 iterations is:  0.5577486398086239
loss after 9 epochs and  8351 iterations is:  0.5576221827195401
loss after 9 epochs and  8401 iterations is:  0.557491525002491
loss after 9 epochs and  8451 iterations is:  0.5573598727114001
loss after 9 epochs and  8501 iterations is:  0.5572029793113387
loss after 9 epochs and  8551 iterations is:  0.5570700575904961
loss after 9 epochs and  8601 iterations is:  0.5569275395124461
loss after 9 epochs and  8651 iterations is:  0.5568053474439338
loss after 9 epochs and  8701 iterations is:  0.5567008148221899
loss after 9 epochs and  8751 iterations is:  0.5565834376432277
loss after 9 epochs and  8801 iterations is:  0.5564756663140148
loss after 9 epochs and  8851 iterations is:  0.5563438817650964
loss after 9 epochs and  8901 iterations is:  0.5562299174551629
loss after 9 epochs and  8951 iterations is:  0.5560920117261633
loss after 9 epochs and  9001 iterations is:  0.5559778526504049
loss after 9 epochs and  9051 iterations is:  0.5558444310299169
loss after 9 epochs and  9101 iterations is:  0.5557391453223006
loss after 9 epochs and  9151 iterations is:  0.5556277192375163
loss after 10 epochs and  9201 iterations is:  0.5554954927310742
loss after 10 epochs and  9251 iterations is:  0.5553989583134692
loss after 10 epochs and  9301 iterations is:  0.5552858688207003
loss after 10 epochs and  9351 iterations is:  0.5551583332472495
loss after 10 epochs and  9401 iterations is:  0.555039779819155
loss after 10 epochs and  9451 iterations is:  0.5549147102836199
loss after 10 epochs and  9501 iterations is:  0.5548046970315949
loss after 10 epochs and  9551 iterations is:  0.5547107197535434
loss after 10 epochs and  9601 iterations is:  0.5545985284707791
loss after 10 epochs and  9651 iterations is:  0.5544990677326753
loss after 10 epochs and  9701 iterations is:  0.5543995973246864
loss after 10 epochs and  9751 iterations is:  0.5542829641944655
loss after 10 epochs and  9801 iterations is:  0.5541883885771578
loss after 10 epochs and  9851 iterations is:  0.5540968456101597
loss after 10 epochs and  9901 iterations is:  0.5539826130036599
loss after 10 epochs and  9951 iterations is:  0.5538807300878917
loss after 10 epochs and  10001 iterations is:  0.5537907246253846
loss after 10 epochs and  10051 iterations is:  0.5537166737990932
loss after 10 epochs and  10101 iterations is:  0.553603310129481
loss after 10 epochs and  10151 iterations is:  0.5534870979030011
loss after 11 epochs and  10201 iterations is:  0.5533936955932607
loss after 11 epochs and  10251 iterations is:  0.5532980209267326
loss after 11 epochs and  10301 iterations is:  0.5531830681911447
loss after 11 epochs and  10351 iterations is:  0.5530949801338361
loss after 11 epochs and  10401 iterations is:  0.5529990685921764
loss after 11 epochs and  10451 iterations is:  0.5528990198018474
```

```
loss after 11 epochs and  10501 iterations is:  0.5528110850786131
loss after 11 epochs and  10551 iterations is:  0.5527162082177198
loss after 11 epochs and  10601 iterations is:  0.552622191370355
loss after 11 epochs and  10651 iterations is:  0.5525282116585308
loss after 11 epochs and  10701 iterations is:  0.5524493648186258
loss after 11 epochs and  10751 iterations is:  0.5523497127123619
loss after 11 epochs and  10801 iterations is:  0.5522616091358604
loss after 11 epochs and  10851 iterations is:  0.5521523045058215
loss after 11 epochs and  10901 iterations is:  0.552064666879308
loss after 11 epochs and  10951 iterations is:  0.5519912750054478
loss after 11 epochs and  11001 iterations is:  0.5518883632731139
loss after 11 epochs and  11051 iterations is:  0.5518257174671004
loss after 11 epochs and  11101 iterations is:  0.5517415099926551
loss after 11 epochs and  11151 iterations is:  0.5516500122152859
loss after 12 epochs and  11201 iterations is:  0.5515694327208794
loss after 12 epochs and  11251 iterations is:  0.5514736514715759
loss after 12 epochs and  11301 iterations is:  0.5513912072581427
loss after 12 epochs and  11351 iterations is:  0.5513114777162561
loss after 12 epochs and  11401 iterations is:  0.5512270950616974
loss after 12 epochs and  11451 iterations is:  0.5511485003797753
loss after 12 epochs and  11501 iterations is:  0.5510704148119394
loss after 12 epochs and  11551 iterations is:  0.5509813482975748
loss after 12 epochs and  11601 iterations is:  0.550902459597465
loss after 12 epochs and  11651 iterations is:  0.5508227809469329
loss after 12 epochs and  11701 iterations is:  0.5507373792035465
loss after 12 epochs and  11751 iterations is:  0.5506628108363436
loss after 12 epochs and  11801 iterations is:  0.550605293876326
loss after 12 epochs and  11851 iterations is:  0.5505322949460433
loss after 12 epochs and  11901 iterations is:  0.5504575557770128
loss after 12 epochs and  11951 iterations is:  0.5503893197915377
loss after 12 epochs and  12001 iterations is:  0.5503056022183622
loss after 12 epochs and  12051 iterations is:  0.5502263450228532
loss after 12 epochs and  12101 iterations is:  0.5501337522086687
loss after 12 epochs and  12151 iterations is:  0.5500582028956167
loss after 12 epochs and  12201 iterations is:  0.5499738415395138
loss after 13 epochs and  12251 iterations is:  0.5499068642957373
loss after 13 epochs and  12301 iterations is:  0.549833565860933
loss after 13 epochs and  12351 iterations is:  0.5497695953032056
loss after 13 epochs and  12401 iterations is:  0.5496966083032404
loss after 13 epochs and  12451 iterations is:  0.549624673732
loss after 13 epochs and  12501 iterations is:  0.5495465706597754
loss after 13 epochs and  12551 iterations is:  0.5494709206579055
loss after 13 epochs and  12601 iterations is:  0.5493994877324923
loss after 13 epochs and  12651 iterations is:  0.549335383482365
loss after 13 epochs and  12701 iterations is:  0.5492719362029425
loss after 13 epochs and  12751 iterations is:  0.5492031879354226
loss after 13 epochs and  12801 iterations is:  0.5491553140230307
loss after 13 epochs and  12851 iterations is:  0.5490734452430494
loss after 13 epochs and  12901 iterations is:  0.5489969972027032
loss after 13 epochs and  12951 iterations is:  0.5489270927080351
loss after 13 epochs and  13001 iterations is:  0.548860503908648
loss after 13 epochs and  13051 iterations is:  0.5487914719855401
loss after 13 epochs and  13101 iterations is:  0.5487240366624102
loss after 13 epochs and  13151 iterations is:  0.54864595112778
loss after 13 epochs and  13201 iterations is:  0.548577357021481
loss after 14 epochs and  13251 iterations is:  0.5485037167579719
loss after 14 epochs and  13301 iterations is:  0.5484472387926145
loss after 14 epochs and  13351 iterations is:  0.5483919795531982
loss after 14 epochs and  13401 iterations is:  0.548319681665149
loss after 14 epochs and  13451 iterations is:  0.54826347439159
loss after 14 epochs and  13501 iterations is:  0.548198947848738
loss after 14 epochs and  13551 iterations is:  0.5481294871457383
loss after 14 epochs and  13601 iterations is:  0.5480909656537011
loss after 14 epochs and  13651 iterations is:  0.5480150117128122
loss after 14 epochs and  13701 iterations is:  0.5479499218548121
```

```
loss after 14 epochs and  13751 iterations is:  0.5478902548125008
loss after 14 epochs and  13801 iterations is:  0.5478245072017309
loss after 14 epochs and  13851 iterations is:  0.5477575113417384
loss after 14 epochs and  13901 iterations is:  0.5476955979990621
loss after 14 epochs and  13951 iterations is:  0.5476314155046504
loss after 14 epochs and  14001 iterations is:  0.5475637636414756
loss after 14 epochs and  14051 iterations is:  0.5475045558241283
loss after 14 epochs and  14101 iterations is:  0.5474474148055087
loss after 14 epochs and  14151 iterations is:  0.5473905105958964
loss after 14 epochs and  14201 iterations is:  0.5473205912951656
loss after 14 epochs and  14251 iterations is:  0.5472615492710918
loss after 15 epochs and  14301 iterations is:  0.5472073715223996
loss after 15 epochs and  14351 iterations is:  0.5471468323514811
loss after 15 epochs and  14401 iterations is:  0.5470928166072482
loss after 15 epochs and  14451 iterations is:  0.5470175057813201
loss after 15 epochs and  14501 iterations is:  0.5469657520196247
loss after 15 epochs and  14551 iterations is:  0.5469075242619512
loss after 15 epochs and  14601 iterations is:  0.5468551045460547
loss after 15 epochs and  14651 iterations is:  0.5467930687249141
loss after 15 epochs and  14701 iterations is:  0.5467360303894607
loss after 15 epochs and  14751 iterations is:  0.5466904839414759
loss after 15 epochs and  14801 iterations is:  0.5466366794496402
loss after 15 epochs and  14851 iterations is:  0.546574484761314
loss after 15 epochs and  14901 iterations is:  0.5465162291732748
loss after 15 epochs and  14951 iterations is:  0.5464654575651202
loss after 15 epochs and  15001 iterations is:  0.546416481866624
loss after 15 epochs and  15051 iterations is:  0.5463620774675362
loss after 15 epochs and  15101 iterations is:  0.5463065706074292
loss after 15 epochs and  15151 iterations is:  0.5462435031877249
loss after 15 epochs and  15201 iterations is:  0.5461835342612296
loss after 15 epochs and  15251 iterations is:  0.5461358135129599
Wall time: 1.6 s
```

In [54]:

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

Out[54]:

```
0.7952686270174663
```

Reference Reading Suggestions

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