Lecture 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 lecture, we're going to learn **logistic regression** (https://en.wikipedia.org/wiki/Logistic_regression), which is a linear **classification** method and a direct generalization of linear regression. We will learn more classification models in the next lecture.

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 (https://en.wikipedia.org/wiki/Logistic_regression#:~:text=Logistic%20regression%20is%20a%20statistical,a%20form%20of%20b 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 have

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

MLE (Maximum Likelihood Estimation)

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

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

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 (https://en.wikipedia.org/wiki/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)

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

In vector form

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

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 (https://github.com/Jaewan-Yun/optimizer-visualization)](https://github.com/Jaewan-Yun/optimizer-visualization).

The simplest algorithm in optimization is [gradient descent (GD) (https://en.wikipedia.org/wiki/Gradient_descent#:~:text=Gradient%20descent%20is%20a%20first,function%20at%20the%20curre](https://en.wikipedia.org/wiki/Gradient_descent#:~:text=Gradient%20descent%20is%20a%20first,function%20at%20the%20curre)

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

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

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

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 durin
g
        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 our lecture
        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
 gradient
            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 our lecture
        beta = self.coeff # the estimated beta
        y_pred = np.round(self.sigmoid(np.dot(X,beta))).astype(int) # >0.5: ->1 else,
->0 -- note that we always use Numpy universal functions when possible
        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 our lecture
        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
-10) # avoid nan issues
        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 expressi
on -- check yourself. It's called Numpy broadcasting
        return np.mean(gradient_value, axis=0)
```

In [2]:
```python
from sklearn.datasets import load_breast_cancer
X, y = load_breast_cancer(return_X_y = True)
```

In [3]:
```python
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
=42)
```

In [4]:
```python
X_train.shape
```

Out[4]:
```
(512, 30)
```

In [7]:
```python
%%time
lg = myLogisticRegression_binary(learning_rate=1e-5)
lg.fit(X_train,y_train,n_iterations = 20000) # 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:  0.23894805957882748
loss after 2501 iterations is:  0.23311785521728873
loss after 3001 iterations is:  0.22909746536348713
loss after 3501 iterations is:  0.22615149966747045
loss after 4001 iterations is:  0.22388601401780292
loss after 4501 iterations is:  0.22207285161370105
loss after 5001 iterations is:  0.22057221113785214
loss after 5501 iterations is:  0.21929462157026375
loss after 6001 iterations is:  0.2181807831094825
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:  0.2140019158232 6
loss after 9001 iterations is:  0.21333156155309685
loss after 9501 iterations is:  0.21269603244872282
loss after 10001 iterations is:  0.21209051523044703
loss after 10501 iterations is:  0.21151124122819925
loss after 11001 iterations is:  0.2109552213025997
loss after 11501 iterations is:  0.2104200541495193
loss after 12001 iterations is:  0.20990378610015575
loss after 12501 iterations is:  0.20940480753853602
loss after 13001 iterations is:  0.2089217756675304
loss after 13501 iterations is:  0.20845355643721925
loss after 14001 iterations is:  0.20799918054355349
loss after 14501 iterations is:  0.20755780984807357
loss after 15001 iterations is:  0.20712871157651588
loss after 15501 iterations is:  0.20671123836543154
loss after 16001 iterations is:  0.20630481273382617
loss after 16501 iterations is:  0.20590891492308788
loss after 17001 iterations is:  0.2055230733150124
loss after 17501 iterations is:  0.20514685683332623
loss after 18001 iterations is:  0.20477986887872715
loss after 18501 iterations is:  0.20442174245513264
loss after 19001 iterations is:  0.2040721362254978
loss after 19501 iterations is:  0.20373073129634592
CPU times: user 10.5 s, sys: 4.42 s, total: 15 s
Wall time: 9.22 s
```

In [8]:
```python
lg.score(X_test,y_test)
```

Out[8]:
```
1.0
```

In [9]:
```python
lg.score(X_train,y_train)
```

Out[9]:
```
0.916015625
```

In [10]:
```python
lg.predict(X_test)
```

Out[10]:
```
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 [11]:
```python
y_test
```

Out[11]:
```
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 [12]:
```python
lg.coeff
```

Out[12]:
```
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 [13]:
```python
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/_log
istic.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[13]:
```
0.9824561403508771
```

In [14]:
```python
clf.score(X_train,y_train)
```

Out[14]:
```
0.955078125
```

It's very normal that our result is different with sklearn. In sklearn [logistic regression (https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression)](https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html#sklearn.linear_model.LogisticRegression), 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

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

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 (https://en.wikipedia.org/wiki/Softmax_function)](https://en.wikipedia.org/wiki/Softmax_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 call 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.

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

In [1]:

```
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 our lecture
        eta = self.learning_rate

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

        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 our lecture
        W = self.coeff # the estimated W
        y_pred = np.argmax(self.sigma(X,W), axis =1) # the category with largest probability
        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 our lecture
        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)
```

```python
    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 ca
n safely put it in the k-th column of 2D array dLdW
        return dLdW
```

In [2]:
```python
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
=42)
```

In [3]:
```python
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:  0.627154495740 4386
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:  0.23316150090969132
loss after 6001 iterations is:  0.22314954388974834
loss after 6501 iterations is:  0.21442400929215735
loss after 7001 iterations is:  0.2067320488693829
loss after 7501 iterations is:  0.19988447822601138
loss after 8001 iterations is:  0.19373672640885967
loss after 8501 iterations is:  0.1881762834198265
loss after 9001 iterations is:  0.1831141858457873
loss after 9501 iterations is:  0.17847909502105724
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:  0.15700606905300005
loss after 13001 iterations is:  0.15418864437799004
loss after 13501 iterations is:  0.1515309472374647
loss after 14001 iterations is:  0.14901781725362154
loss after 14501 iterations is:  0.14663603885543683
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:  0.1363365802952386
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 [4]:
```python
lg.score(X_test,y_test)
```

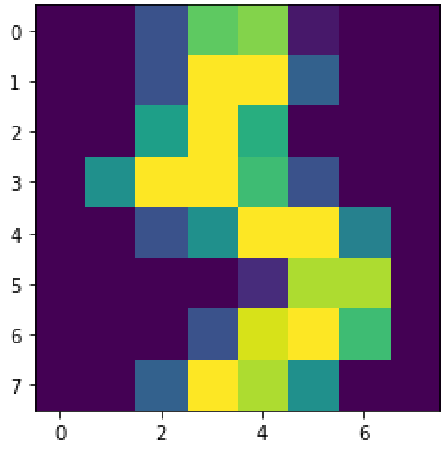
Out[4]: 0.9722222222222222

In [17]:
```python
np.where(lg.predict(X_test)!=y_test)
```

Out[17]: (array([5, 71, 133, 149, 159]),)

In [18]:
```python
import matplotlib.pyplot as plt
plt.imshow(X_test[149,].reshape(8,8))
```

Out[18]: `<matplotlib.image.AxesImage at 0x7fbbc62cd9d0>`



In [20]:
```python
print(lg.predict(X_test)[149],y_test[149])
```

```
5 3
```

For multi-class classification, the [confusion matrix (https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html)](https://scikit-learn.org/stable/modules/generated/sklearn.metrics.confusion_matrix.html) can provide as more details.

In [21]:
```python
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test,lg.predict(X_test))
```

Out[21]:
```
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]])
```

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 $M \le N/n_B$, number of epochs n_E
>
> For epoch $n = 1, 2, \cdots, n_E$
> β^0 for the current epoch is β^{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 [22]:

```python
import numpy as np

class myLogisticRegression_binary():
    """ Logistic Regression classifier -- this only works for the binary case. Here we provide the option of SGD in optimization.
    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_batch = 20):

        # 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 our lecture
        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 compute gradient
                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 reshuffle the data to create "randomness"
                for m in range(0,N,size_batch):   # m is the starting index of mini-batch
                    i = shuffle_index[m:m+size_batch] # index of samples in the mini-batch
                    dbeta = self.loss_gradient(beta,X[i,:],y[i]) # only use the data in mini-batch to compute gradient. Note the average is taken in the loss_gradient function
                    beta = beta - eta * dbeta # the formula of GD, but this time dbeta is different

                    if e % 1 == 0 and num_iter % 50 ==0: # print loss during the training process
                        print("loss after", e+1, "epochs and ", num_iter+1, "iterations is: ", self.loss(beta,X,y))

                    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 our lecture
        beta = self.coeff # the estimated beta
        y_pred = np.round(self.sigmoid(np.dot(X,beta))).astype(int) # >0.5: ->1 else,
->0 -- note that we always use Numpy universal functions when possible
        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 our lecture
        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
-10) # avoid nan issues
        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 expressi
on -- check yourself
        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 (https://www.stat.cmu.edu/~ryantibs/convexopt/lectures/stochastic-gd.pdf). 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 (https://archive.ics.uci.edu/ml/datasets/adult) to predict income. Note that it is a binary classification problem.

In [23]:
```python
import pandas as pd
df = pd.read_csv('adult.csv')
df
```

Out[23]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital gai
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	768
4	18	?	103497	Some-college	10	Never-married	?	Own-child	White	Female	
...	...	...	...	...	...	...	...	...	...	...	.
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	1502

48842 rows × 15 columns

In [24]:
```python
from numpy import nan
df = df.replace('?',nan) #dealing with missing values -- ? in original dataset
df.head()
```

Out[24]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	ca
0	25	Private	226802	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	
1	38	Private	89814	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	
2	28	Local-gov	336951	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	
3	44	Private	160323	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	
4	18	NaN	103497	Some-college	10	Never-married	NaN	Own-child	White	Female	0	

```
In [25]: df.dropna(inplace = True) # drop missing values
         df
```

Out[25]:

	age	workclass	fnlwgt	education	educational-num	marital-status	occupation	relationship	race	gender	capital gai
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	768
5	34	Private	198693	10th	6	Never-married	Other-service	Not-in-family	White	Male	
...	...	...	...	...	...	...	...	...	...	...	.
48837	27	Private	257302	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	
48838	40	Private	154374	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	
48839	58	Private	151910	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	
48840	22	Private	201490	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	
48841	52	Self-emp-inc	287927	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	1502

45222 rows × 15 columns

In [26]:
```python
df.drop(columns=['fnlwgt','native-country'], inplace=True) # drop some variables we are not interested
df
```

Out[26]:

	age	workclass	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capita los
0	25	Private	11th	7	Never-married	Machine-op-inspct	Own-child	Black	Male	0	
1	38	Private	HS-grad	9	Married-civ-spouse	Farming-fishing	Husband	White	Male	0	
2	28	Local-gov	Assoc-acdm	12	Married-civ-spouse	Protective-serv	Husband	White	Male	0	
3	44	Private	Some-college	10	Married-civ-spouse	Machine-op-inspct	Husband	Black	Male	7688	
5	34	Private	10th	6	Never-married	Other-service	Not-in-family	White	Male	0	
...	...	...	...	...	...	...	...	...	...	...	
48837	27	Private	Assoc-acdm	12	Married-civ-spouse	Tech-support	Wife	White	Female	0	
48838	40	Private	HS-grad	9	Married-civ-spouse	Machine-op-inspct	Husband	White	Male	0	
48839	58	Private	HS-grad	9	Widowed	Adm-clerical	Unmarried	White	Female	0	
48840	22	Private	HS-grad	9	Never-married	Adm-clerical	Own-child	White	Male	0	
48841	52	Self-emp-inc	HS-grad	9	Married-civ-spouse	Exec-managerial	Wife	White	Female	15024	

45222 rows × 13 columns

In [27]:
```python
from sklearn.preprocessing import LabelEncoder
df_clean = df.apply(LabelEncoder().fit_transform) # transform the categorical variabl
es into numerical
df_clean
```

Out[27]:

	age	workclass	education	educational-num	marital-status	occupation	relationship	race	gender	capital-gain	capital-loss
0	8	2	1	6	4	6	3	2	1	0	0
1	21	2	11	8	2	4	0	4	1	0	0
2	11	1	7	11	2	10	0	4	1	0	0
3	27	2	15	9	2	6	0	2	1	96	0
5	17	2	0	5	4	7	1	4	1	0	0
...	...	...	...	...	...	...	...	...	...	...	...
48837	10	2	7	11	2	12	5	4	0	0	0
48838	23	2	11	8	2	6	0	4	1	0	0
48839	41	2	11	8	6	0	4	4	0	0	0
48840	5	2	11	8	4	0	3	4	1	0	0
48841	35	3	11	8	2	3	5	4	0	110	0

45222 rows × 13 columns

Note that it is not best way to encode the data. Please see other solutions in [kaggle (https://www.kaggle.com/wenruliu/adult-income-dataset/notebooks)](https://www.kaggle.com/wenruliu/adult-income-dataset/notebooks).

In [28]:
```python
y = df_clean['income'].to_numpy()
X = df_clean.drop(columns = 'income').to_numpy()
```

In [29]:
```python
X.shape
```

Out[29]:
```
(45222, 12)
```

In [30]:
```python
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
=42)
```

In [31]:
```python
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/_log
istic.py:940: 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[31]:
```
0.8273269953570639
```

In [32]:
```python
lg_gd = myLogisticRegression_binary(learning_rate=1e-6, opt_method = 'GD')
lg_sgd = myLogisticRegression_binary(learning_rate=1e-6, opt_method = 'SGD', num_epoc
hs = 15, size_batch = 40)
```

In [33]:
```
%%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:  0.5984037857262678
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:  0.5600397185713462
loss after 8001 iterations is:  0.5585633633136624
loss after 8501 iterations is:  0.5572150485499265
loss after 9001 iterations is:  0.5559788415837271
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
CPU times: user 3min 30s, sys: 20 s, total: 3min 50s
Wall time: 1min 40s
```

In [34]:
```
lg_gd.score(X_test,y_test)
```

Out[34]:
```
0.7950475348220207
```

In [35]:
```
%%time
lg_sgd.fit(X_train,y_train)
```

```
loss after 1 epochs and  1 iterations is:  0.6930118979471646
loss after 1 epochs and  51 iterations is:  0.6876777123580536
loss after 1 epochs and  101 iterations is:  0.6826909304151703
loss after 1 epochs and  151 iterations is:  0.6778211429968567
loss after 1 epochs and  201 iterations is:  0.6730777278043059
loss after 1 epochs and  251 iterations is:  0.6689097392879488
loss after 1 epochs and  301 iterations is:  0.6646840056035999
loss after 1 epochs and  351 iterations is:  0.6608084361435315
loss after 1 epochs and  401 iterations is:  0.6571895666530512
loss after 1 epochs and  451 iterations is:  0.6536557515808685
loss after 1 epochs and  501 iterations is:  0.6502398527509298
loss after 1 epochs and  551 iterations is:  0.6470259037808793
loss after 1 epochs and  601 iterations is:  0.6440504864380129
loss after 1 epochs and  651 iterations is:  0.6414143680524861
loss after 1 epochs and  701 iterations is:  0.6386168188697363
loss after 1 epochs and  751 iterations is:  0.6359766163957282
loss after 1 epochs and  801 iterations is:  0.6335764631149665
loss after 1 epochs and  851 iterations is:  0.6311387054472773
loss after 1 epochs and  901 iterations is:  0.6288805734610214
loss after 1 epochs and  951 iterations is:  0.626735882437586
loss after 1 epochs and  1001 iterations is:  0.6250357968092135
loss after 2 epochs and  1051 iterations is:  0.6230640270655049
loss after 2 epochs and  1101 iterations is:  0.621493940030239
loss after 2 epochs and  1151 iterations is:  0.6194834220700702
loss after 2 epochs and  1201 iterations is:  0.6178010252962652
loss after 2 epochs and  1251 iterations is:  0.6162666464650577
loss after 2 epochs and  1301 iterations is:  0.6146381566162975
loss after 2 epochs and  1351 iterations is:  0.6132607610696661
loss after 2 epochs and  1401 iterations is:  0.6116814354839507
loss after 2 epochs and  1451 iterations is:  0.6102860555220877
loss after 2 epochs and  1501 iterations is:  0.6089742871451047
loss after 2 epochs and  1551 iterations is:  0.6078430548580128
loss after 2 epochs and  1601 iterations is:  0.6066669649572622
loss after 2 epochs and  1651 iterations is:  0.6055508471162475
loss after 2 epochs and  1701 iterations is:  0.60436611294241
loss after 2 epochs and  1751 iterations is:  0.6034497613732803
loss after 2 epochs and  1801 iterations is:  0.6025062329911987
loss after 2 epochs and  1851 iterations is:  0.601457826923542
loss after 2 epochs and  1901 iterations is:  0.6003971812261527
loss after 2 epochs and  1951 iterations is:  0.5993918953279724
loss after 2 epochs and  2001 iterations is:  0.5983424327236601
loss after 3 epochs and  2051 iterations is:  0.5975108509722612
loss after 3 epochs and  2101 iterations is:  0.5967476283045278
loss after 3 epochs and  2151 iterations is:  0.5959613191086646
loss after 3 epochs and  2201 iterations is:  0.595173691312486
loss after 3 epochs and  2251 iterations is:  0.5944204449959734
loss after 3 epochs and  2301 iterations is:  0.5935195803295548
loss after 3 epochs and  2351 iterations is:  0.5926884014234615
loss after 3 epochs and  2401 iterations is:  0.5921532635515561
loss after 3 epochs and  2451 iterations is:  0.591465923055286
loss after 3 epochs and  2501 iterations is:  0.5907128107413767
loss after 3 epochs and  2551 iterations is:  0.5900379178033799
loss after 3 epochs and  2601 iterations is:  0.5894047782406248
loss after 3 epochs and  2651 iterations is:  0.5887161852424456
loss after 3 epochs and  2701 iterations is:  0.5881139117031038
loss after 3 epochs and  2751 iterations is:  0.5874540732255548
loss after 3 epochs and  2801 iterations is:  0.58682603175035
loss after 3 epochs and  2851 iterations is:  0.586377388884109
loss after 3 epochs and  2901 iterations is:  0.585910169406304 9
loss after 3 epochs and  2951 iterations is:  0.5853663125933827
loss after 3 epochs and  3001 iterations is:  0.5848202538597242
loss after 3 epochs and  3051 iterations is:  0.5843280625024887
loss after 4 epochs and  3101 iterations is:  0.5838758431084522
loss after 4 epochs and  3151 iterations is:  0.5833850900129134
loss after 4 epochs and  3201 iterations is:  0.5828699166369437
loss after 4 epochs and  3251 iterations is:  0.5823444219898402
loss after 4 epochs and  3301 iterations is:  0.5818702099194496
loss after 4 epochs and  3351 iterations is:  0.5814575377786289
loss after 4 epochs and  3401 iterations is:  0.5809850299847537
loss after 4 epochs and  3451 iterations is:  0.5805730598917818
```

```
loss after 4 epochs and  3501 iterations is:  0.5801364398475553
loss after 4 epochs and  3551 iterations is:  0.5796998084665542
loss after 4 epochs and  3601 iterations is:  0.5792878940582911
loss after 4 epochs and  3651 iterations is:  0.5788455552134402
loss after 4 epochs and  3701 iterations is:  0.5784522233150032
loss after 4 epochs and  3751 iterations is:  0.5780706259037574
loss after 4 epochs and  3801 iterations is:  0.5776843734256253
loss after 4 epochs and  3851 iterations is:  0.5772829363413755
loss after 4 epochs and  3901 iterations is:  0.5769383577764936
loss after 4 epochs and  3951 iterations is:  0.5766058300149423
loss after 4 epochs and  4001 iterations is:  0.5762534036575593
loss after 4 epochs and  4051 iterations is:  0.5759567821426641
loss after 5 epochs and  4101 iterations is:  0.5755927895123201
loss after 5 epochs and  4151 iterations is:  0.575276755954271
loss after 5 epochs and  4201 iterations is:  0.5749264315630049
loss after 5 epochs and  4251 iterations is:  0.5745825502488443
loss after 5 epochs and  4301 iterations is:  0.5742605774724879
loss after 5 epochs and  4351 iterations is:  0.573922210783771
loss after 5 epochs and  4401 iterations is:  0.573634086908925
loss after 5 epochs and  4451 iterations is:  0.5733032609858596
loss after 5 epochs and  4501 iterations is:  0.5729617998380929
loss after 5 epochs and  4551 iterations is:  0.57268037451237
loss after 5 epochs and  4601 iterations is:  0.5723879557183079
loss after 5 epochs and  4651 iterations is:  0.5720984162114746
loss after 5 epochs and  4701 iterations is:  0.5718269492767001
loss after 5 epochs and  4751 iterations is:  0.5715750292525483
loss after 5 epochs and  4801 iterations is:  0.5712409570998339
loss after 5 epochs and  4851 iterations is:  0.5709564342246705
loss after 5 epochs and  4901 iterations is:  0.5707016203026634
loss after 5 epochs and  4951 iterations is:  0.5704618785106677
loss after 5 epochs and  5001 iterations is:  0.5701958141118837
loss after 5 epochs and  5051 iterations is:  0.5699123525938765
loss after 6 epochs and  5101 iterations is:  0.5696574021557947
loss after 6 epochs and  5151 iterations is:  0.5694073556678376
loss after 6 epochs and  5201 iterations is:  0.569141736987522
loss after 6 epochs and  5251 iterations is:  0.5688893968459046
loss after 6 epochs and  5301 iterations is:  0.5685898552130726
loss after 6 epochs and  5351 iterations is:  0.5683283653522587
loss after 6 epochs and  5401 iterations is:  0.5680899030332782
loss after 6 epochs and  5451 iterations is:  0.5677889235014187
loss after 6 epochs and  5501 iterations is:  0.5675630473669019
loss after 6 epochs and  5551 iterations is:  0.5673201703665066
loss after 6 epochs and  5601 iterations is:  0.5671153824103109
loss after 6 epochs and  5651 iterations is:  0.5669017211622869
loss after 6 epochs and  5701 iterations is:  0.5666655311615366
loss after 6 epochs and  5751 iterations is:  0.5664134548942581
loss after 6 epochs and  5801 iterations is:  0.566194914663283
loss after 6 epochs and  5851 iterations is:  0.565984299608547
loss after 6 epochs and  5901 iterations is:  0.5657904185463832
loss after 6 epochs and  5951 iterations is:  0.5655877003665757
loss after 6 epochs and  6001 iterations is:  0.5653692657723363
loss after 6 epochs and  6051 iterations is:  0.5652029511838159
loss after 6 epochs and  6101 iterations is:  0.5650628014798952
loss after 7 epochs and  6151 iterations is:  0.5648686771330182
loss after 7 epochs and  6201 iterations is:  0.5646856050153015
loss after 7 epochs and  6251 iterations is:  0.564472844233803
loss after 7 epochs and  6301 iterations is:  0.56429040741659
loss after 7 epochs and  6351 iterations is:  0.5640868649607729
loss after 7 epochs and  6401 iterations is:  0.5638921711310174
loss after 7 epochs and  6451 iterations is:  0.5636983256182466
loss after 7 epochs and  6501 iterations is:  0.5635170926531153
loss after 7 epochs and  6551 iterations is:  0.5633157113395132
loss after 7 epochs and  6601 iterations is:  0.5631278686437162
loss after 7 epochs and  6651 iterations is:  0.5629320639807348
loss after 7 epochs and  6701 iterations is:  0.5627234666010122
loss after 7 epochs and  6751 iterations is:  0.5625681413681102
loss after 7 epochs and  6801 iterations is:  0.5624208410158072
loss after 7 epochs and  6851 iterations is:  0.5622190763518221
loss after 7 epochs and  6901 iterations is:  0.5620112809893391
loss after 7 epochs and  6951 iterations is:  0.5618166290522824
```

```
loss after 7 epochs and  7001 iterations is:  0.5616393183351481
loss after 7 epochs and  7051 iterations is:  0.5614853401570282
loss after 7 epochs and  7101 iterations is:  0.5613397661780729
loss after 8 epochs and  7151 iterations is:  0.561163738217373
loss after 8 epochs and  7201 iterations is:  0.5609799158086491
loss after 8 epochs and  7251 iterations is:  0.5608394777307186
loss after 8 epochs and  7301 iterations is:  0.5606931297198858
loss after 8 epochs and  7351 iterations is:  0.5605581681658339
loss after 8 epochs and  7401 iterations is:  0.560401343827877
loss after 8 epochs and  7451 iterations is:  0.5602622034756315
loss after 8 epochs and  7501 iterations is:  0.5600800137277224
loss after 8 epochs and  7551 iterations is:  0.5599357420640172
loss after 8 epochs and  7601 iterations is:  0.5597909124659747
loss after 8 epochs and  7651 iterations is:  0.5596308718342891
loss after 8 epochs and  7701 iterations is:  0.5594784519459159
loss after 8 epochs and  7751 iterations is:  0.5593280843462587
loss after 8 epochs and  7801 iterations is:  0.5591851346540774
loss after 8 epochs and  7851 iterations is:  0.559026827512996
loss after 8 epochs and  7901 iterations is:  0.5588637175479478
loss after 8 epochs and  7951 iterations is:  0.5587054538220516
loss after 8 epochs and  8001 iterations is:  0.5585715926854635
loss after 8 epochs and  8051 iterations is:  0.5584369585304008
loss after 8 epochs and  8101 iterations is:  0.5582793950443813
loss after 9 epochs and  8151 iterations is:  0.5581563797423108
loss after 9 epochs and  8201 iterations is:  0.5579809820886362
loss after 9 epochs and  8251 iterations is:  0.5578526229638434
loss after 9 epochs and  8301 iterations is:  0.5577342942133428
loss after 9 epochs and  8351 iterations is:  0.5576052900882235
loss after 9 epochs and  8401 iterations is:  0.5574843195282672
loss after 9 epochs and  8451 iterations is:  0.5573162785216175
loss after 9 epochs and  8501 iterations is:  0.5571836200180155
loss after 9 epochs and  8551 iterations is:  0.5570509513263672
loss after 9 epochs and  8601 iterations is:  0.5569174011703156
loss after 9 epochs and  8651 iterations is:  0.556795306783237
loss after 9 epochs and  8701 iterations is:  0.5566740941547167
loss after 9 epochs and  8751 iterations is:  0.5565269651394383
loss after 9 epochs and  8801 iterations is:  0.5564089134966715
loss after 9 epochs and  8851 iterations is:  0.5563050464404546
loss after 9 epochs and  8901 iterations is:  0.5561849472372389
loss after 9 epochs and  8951 iterations is:  0.5560704134215242
loss after 9 epochs and  9001 iterations is:  0.5559747558833913
loss after 9 epochs and  9051 iterations is:  0.555863266740625
loss after 9 epochs and  9101 iterations is:  0.555743084286193
loss after 9 epochs and  9151 iterations is:  0.5556363443478388
loss after 10 epochs and  9201 iterations is:  0.5554946816401514
loss after 10 epochs and  9251 iterations is:  0.5553779404511259
loss after 10 epochs and  9301 iterations is:  0.5552783778278804
loss after 10 epochs and  9351 iterations is:  0.5551671401698989
loss after 10 epochs and  9401 iterations is:  0.5550692150106314
loss after 10 epochs and  9451 iterations is:  0.5549636125848847
loss after 10 epochs and  9501 iterations is:  0.5548656139145319
loss after 10 epochs and  9551 iterations is:  0.5547255297392687
loss after 10 epochs and  9601 iterations is:  0.5546279131199231
loss after 10 epochs and  9651 iterations is:  0.5545107843651654
loss after 10 epochs and  9701 iterations is:  0.5544058038823557
loss after 10 epochs and  9751 iterations is:  0.5542944995327926
loss after 10 epochs and  9801 iterations is:  0.5541900359369862
loss after 10 epochs and  9851 iterations is:  0.5541034582114494
loss after 10 epochs and  9901 iterations is:  0.5540187635355225
loss after 10 epochs and  9951 iterations is:  0.5539056513975256
loss after 10 epochs and  10001 iterations is:  0.5537955717095898
loss after 10 epochs and  10051 iterations is:  0.5536995896945599
loss after 10 epochs and  10101 iterations is:  0.5535908858384639
loss after 10 epochs and  10151 iterations is:  0.5534960679210433
loss after 11 epochs and  10201 iterations is:  0.553392123085896l
loss after 11 epochs and  10251 iterations is:  0.5532894180861134
loss after 11 epochs and  10301 iterations is:  0.5531917015084392
loss after 11 epochs and  10351 iterations is:  0.5531096518402113
loss after 11 epochs and  10401 iterations is:  0.5530013577023898
loss after 11 epochs and  10451 iterations is:  0.552910720494601 2
```

```
loss after 11 epochs and  10501 iterations is:  0.5527991492781624
loss after 11 epochs and  10551 iterations is:  0.5527120920005295
loss after 11 epochs and  10601 iterations is:  0.5526199365404822
loss after 11 epochs and  10651 iterations is:  0.5525123885031819
loss after 11 epochs and  10701 iterations is:  0.5524309674699276
loss after 11 epochs and  10751 iterations is:  0.5523395329530619
loss after 11 epochs and  10801 iterations is:  0.5522553085871922
loss after 11 epochs and  10851 iterations is:  0.5521646445515708
loss after 11 epochs and  10901 iterations is:  0.5520769922506568
loss after 11 epochs and  10951 iterations is:  0.5519960813619921
loss after 11 epochs and  11001 iterations is:  0.5519020539704856
loss after 11 epochs and  11051 iterations is:  0.5518055526205685
loss after 11 epochs and  11101 iterations is:  0.5517327831524174
loss after 11 epochs and  11151 iterations is:  0.5516462922853178
loss after 12 epochs and  11201 iterations is:  0.5515718902994766
loss after 12 epochs and  11251 iterations is:  0.5514952277551647
loss after 12 epochs and  11301 iterations is:  0.5514068178139546
loss after 12 epochs and  11351 iterations is:  0.5513189476304012
loss after 12 epochs and  11401 iterations is:  0.551221495657165
loss after 12 epochs and  11451 iterations is:  0.551153193656006
loss after 12 epochs and  11501 iterations is:  0.5510567464375132
loss after 12 epochs and  11551 iterations is:  0.550979103342692
loss after 12 epochs and  11601 iterations is:  0.5508891523439085
loss after 12 epochs and  11651 iterations is:  0.5508123140867541
loss after 12 epochs and  11701 iterations is:  0.5507301110290597
loss after 12 epochs and  11751 iterations is:  0.5506672728923228
loss after 12 epochs and  11801 iterations is:  0.5505774082299983
loss after 12 epochs and  11851 iterations is:  0.5505063611391656
loss after 12 epochs and  11901 iterations is:  0.5504275454796188
loss after 12 epochs and  11951 iterations is:  0.5503517312457632
loss after 12 epochs and  12001 iterations is:  0.5502730285819797
loss after 12 epochs and  12051 iterations is:  0.5501989350348456
loss after 12 epochs and  12101 iterations is:  0.5501256645541425
loss after 12 epochs and  12151 iterations is:  0.5500556702005096
loss after 12 epochs and  12201 iterations is:  0.5499792534398584
loss after 13 epochs and  12251 iterations is:  0.5498986679059644
loss after 13 epochs and  12301 iterations is:  0.5498211766515212
loss after 13 epochs and  12351 iterations is:  0.5497608820604237
loss after 13 epochs and  12401 iterations is:  0.5496787738052614
loss after 13 epochs and  12451 iterations is:  0.5496026147653735
loss after 13 epochs and  12501 iterations is:  0.5495153447835112
loss after 13 epochs and  12551 iterations is:  0.5494459138060909
loss after 13 epochs and  12601 iterations is:  0.5493676443554855
loss after 13 epochs and  12651 iterations is:  0.549309337565025
loss after 13 epochs and  12701 iterations is:  0.5492264916835013
loss after 13 epochs and  12751 iterations is:  0.5491563947149445
loss after 13 epochs and  12801 iterations is:  0.5490975356811367
loss after 13 epochs and  12851 iterations is:  0.5490397063917768
loss after 13 epochs and  12901 iterations is:  0.548971894453732
loss after 13 epochs and  12951 iterations is:  0.5489112396103043
loss after 13 epochs and  13001 iterations is:  0.5488421203092196
loss after 13 epochs and  13051 iterations is:  0.548779874267089
loss after 13 epochs and  13101 iterations is:  0.5487135417503742
loss after 13 epochs and  13151 iterations is:  0.5486445502443921
loss after 13 epochs and  13201 iterations is:  0.548576317216341
loss after 14 epochs and  13251 iterations is:  0.5485133151015091
loss after 14 epochs and  13301 iterations is:  0.5484471101343161
loss after 14 epochs and  13351 iterations is:  0.548370415071877
loss after 14 epochs and  13401 iterations is:  0.5483065842034808
loss after 14 epochs and  13451 iterations is:  0.5482412575346614
loss after 14 epochs and  13501 iterations is:  0.5481881703338498
loss after 14 epochs and  13551 iterations is:  0.5481234581243993
loss after 14 epochs and  13601 iterations is:  0.5480561131878258
loss after 14 epochs and  13651 iterations is:  0.5479896574383092
loss after 14 epochs and  13701 iterations is:  0.5479302251896877
loss after 14 epochs and  13751 iterations is:  0.5478746130395835
loss after 14 epochs and  13801 iterations is:  0.5478110786323709
loss after 14 epochs and  13851 iterations is:  0.5477469519397731
loss after 14 epochs and  13901 iterations is:  0.5476890917347207
loss after 14 epochs and  13951 iterations is:  0.5476234486110219
```

```
loss after 14 epochs and  14001 iterations is:  0.5475698378160184
loss after 14 epochs and  14051 iterations is:  0.5475169565301972
loss after 14 epochs and  14101 iterations is:  0.5474583392406789
loss after 14 epochs and  14151 iterations is:  0.5473973333377867
loss after 14 epochs and  14201 iterations is:  0.5473170316137347
loss after 14 epochs and  14251 iterations is:  0.5472617604041193
loss after 15 epochs and  14301 iterations is:  0.5471974194297892
loss after 15 epochs and  14351 iterations is:  0.5471320310387827
loss after 15 epochs and  14401 iterations is:  0.5470720978695461
loss after 15 epochs and  14451 iterations is:  0.5470164630118529
loss after 15 epochs and  14501 iterations is:  0.5469574132457932
loss after 15 epochs and  14551 iterations is:  0.5468904442989961
loss after 15 epochs and  14601 iterations is:  0.5468288530970149
loss after 15 epochs and  14651 iterations is:  0.546777323022216
loss after 15 epochs and  14701 iterations is:  0.5467232295925318
loss after 15 epochs and  14751 iterations is:  0.5466662951646292
loss after 15 epochs and  14801 iterations is:  0.546616832625842
loss after 15 epochs and  14851 iterations is:  0.5465605402857081
loss after 15 epochs and  14901 iterations is:  0.5464999767940053
loss after 15 epochs and  14951 iterations is:  0.5464521121851176
loss after 15 epochs and  15001 iterations is:  0.5463925693636691
loss after 15 epochs and  15051 iterations is:  0.5463316806195194
loss after 15 epochs and  15101 iterations is:  0.5462786141501177
loss after 15 epochs and  15151 iterations is:  0.5462177758395482
loss after 15 epochs and  15201 iterations is:  0.5461786968349149
loss after 15 epochs and  15251 iterations is:  0.5461399780558017
CPU times: user 6.12 s, sys: 1.29 s, total: 7.41 s
Wall time: 8.13 s
```

In [36]:
```
lg_sgd.score(X_test,y_test)
```

Out[36]:
```
0.7950475348220207
```

Reference Reading Suggestions

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