Lecture 11: Logistic Regression and Classification

In classification problem, the response variables y are discrete, representing different catagories. For simplicity, we will first introduce the **binary classification case** -- y has only two categories, denoted as 0 and 1.

Model-setup

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 $\sigma(z) = \frac{1}{1+\exp(-x)}$ 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
$$\ln \frac{p}{1-p} = \tilde{x}\beta$$
- Since y only takes value in 0 or 1, we have
$$P(y|\mathbf{x}, \beta) = f(\mathbf{x}; \beta)^y \big(1 - f(\mathbf{x}; \beta)\big)^{1-y}$$

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 true probability distribution and estimated probability distribution via logistic model.

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).

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

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

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

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 whole 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]:
```
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
=42)
```

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 = 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:  0.2646726705164665
loss after 1501 iterations is:  0.24813479245950681
loss after 2001 iterations is:  0.2389480595788275
loss after 2501 iterations is:  0.23311785521728876
loss after 3001 iterations is:  0.22909746536348718
loss after 3501 iterations is:  0.22615149966747047
loss after 4001 iterations is:  0.2238860140178029
loss after 4501 iterations is:  0.22207285161370108
loss after 5001 iterations is:  0.22057221113785216
loss after 5501 iterations is:  0.2192946215702638
loss after 6001 iterations is:  0.21818078310948252
loss after 6501 iterations is:  0.21719023774728446
loss after 7001 iterations is:  0.21629469731306186
loss after 7501 iterations is:  0.21547395937965436
loss after 8001 iterations is:  0.21471332436186458
loss after 8501 iterations is:  0.21400191582326006
loss after 9001 iterations is:  0.21333156155309688
loss after 9501 iterations is:  0.21269603244872282
loss after 10001 iterations is:  0.21209051523044697
loss after 10501 iterations is:  0.21151124122819925
loss after 11001 iterations is:  0.21095522130259972
loss after 11501 iterations is:  0.21042005414951925
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:  0.20799918054355343
loss after 14501 iterations is:  0.20755780984807357
loss after 15001 iterations is:  0.2071287115765159
loss after 15501 iterations is:  0.2067112383654315
loss after 16001 iterations is:  0.20630481273382617
loss after 16501 iterations is:  0.20590891492308783
loss after 17001 iterations is:  0.20552307331501238
loss after 17501 iterations is:  0.20514685683332629
loss after 18001 iterations is:  0.20477986887872715
loss after 18501 iterations is:  0.20442174245513262
loss after 19001 iterations is:  0.2040721362254978
loss after 19501 iterations is:  0.20373073129634595
CPU times: user 1.48 s, sys: 50.3 ms, total: 1.53 s
Wall time: 1.53 s
```

In [6]:
```
lg.score(X_test,y_test)
```

Out[6]:
```
1.0
```

In [7]:
```
lg.score(X_train,y_train)
```

Out[7]:
```
0.916015625
```

```
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 [9]: y_test

Out[9]: array([1, 0, 0, 1, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 1, 1, 1, 0, 0, 1,
               0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 1, 1,
               1, 1, 1, 1, 1, 1, 0, 0, 1, 1, 1, 1, 1])
```

```
In [10]: lg.coeff

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

```
In [11]: 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[11]: 0.9824561403508771
```

```
In [12]: clf.score(X_train,y_train)

Out[12]: 0.953125
```

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), they use different algorithms to solve the optimization problem, and even the model is different (adding regularization terms!)

Multi-class Case

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.

Loss function

Define the following indicator function:

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

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.

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

In [16]:
```
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 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
        """
        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 choi
ces

        for k in range(n_iterations):
            dW = self.loss_gradient(W,X,y) # write another function to compute gradie
nt
            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 prob
ability
        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):
        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 ca
n safely put it in the k-th column of 2D array dLdW
        return dLdW
```

In [14]:
```
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 [17]:
```
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:  0.23316150090969132
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 [18]:
```
lg.score(X_test,y_test)
```

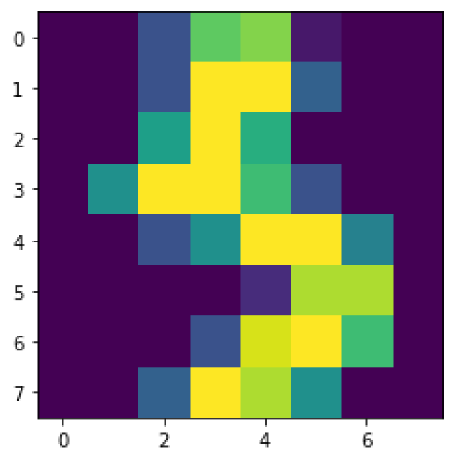
Out[18]:
```
0.9722222222222222
```

In [19]:
```
np.where(lg.predict(X_test)!=y_test)
```

Out[19]:
```
(array([  5,  71, 133, 149, 159]),)
```

In [20]:
```
import matplotlib.pyplot as plt
plt.imshow(X_test[149,].reshape(8,8))
```

Out[20]:
```
<matplotlib.image.AxesImage at 0x7f9832553b90>
```



In [21]:
```
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 [22]:
```
from sklearn.metrics import confusion_matrix
confusion_matrix(y_test,lg.predict(X_test))
```

Out[22]:
```
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 $\beta^0$, step size (learning rate) $\eta$,
batch size $n_B$, number of inner iterations $M \le N/n_B$, number of epochs $n_E$

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

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

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

In [23]:
```
import numpy as np

class myLogisticRegression_binary():
    """ Logistic Regression classifier -- this only works for the binary case. Here w
e provide the option of SGD in optimization.
    Parameters:
    -----------
    learning_rate: float
        The step length that will be taken when following the negative gradient durin
g
        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 comp
ute 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 re
shuffle the data to create "randomness"
                for m in range(0,N,size_batch):   # m is the starting index of mini-b
atch
                    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 fu
nction
                    beta = beta - eta * dbeta # the formula of GD, but this time dbet
a is different

                    if e % 1 == 0 and num_iter % 50 ==0: # print loss during the trai
ning process
                        print("loss after", e+1, "epochs and ", num_iter+1, "iteratio
ns 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 [24]:
```python
import pandas as pd
df = pd.read_csv('adult.csv')
df
```

Out[24]:

	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 [25]:
```python
from numpy import nan
df = df.replace('?',nan) #dealing with missing values -- ? in original dataset
df.head()
```

Out[25]:

	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 [26]:
```
df.dropna(inplace = True) # drop missing values
df
```

Out[26]:

	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 [27]:
```python
df.drop(columns=['fnlwgt','native-country'], inplace=True) # drop some variables we a
re not interested
df
```

Out[27]:

	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 [28]:
```python
from sklearn.preprocessing import LabelEncoder
df_clean = df.apply(LabelEncoder().fit_transform) # transform the categorical variabl
es into numerical
df_clean
```

Out[28]:

	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 [29]:
```python
y = df_clean['income'].to_numpy()
X = df_clean.drop(columns = 'income').to_numpy()
```

In [30]:
```python
X.shape
```

Out[30]:
```
(45222, 12)
```

In [31]:
```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 [32]:
```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[32]:
```
0.8273269953570639
```

In [34]:
```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 [35]: %%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 2min 14s, sys: 1.88 s, total: 2min 16s
Wall time: 34.5 s
```

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

Out[36]: 0.7950475348220207
```

In [37]:
```
%%time
lg_sgd.fit(X_train,y_train)
```

```
loss after 1 epochs and  1 iterations is:  0.6929943378097103
loss after 1 epochs and  51 iterations is:  0.6879157921439216
loss after 1 epochs and  101 iterations is:  0.6826583387379885
loss after 1 epochs and  151 iterations is:  0.6772727824166075
loss after 1 epochs and  201 iterations is:  0.6726785634817065
loss after 1 epochs and  251 iterations is:  0.6685869015583293
loss after 1 epochs and  301 iterations is:  0.6646473900980252
loss after 1 epochs and  351 iterations is:  0.6611693171938025
loss after 1 epochs and  401 iterations is:  0.657648969777927
loss after 1 epochs and  451 iterations is:  0.6543380353684581
loss after 1 epochs and  501 iterations is:  0.6506396745665634
loss after 1 epochs and  551 iterations is:  0.6475672520727679
loss after 1 epochs and  601 iterations is:  0.6444325842294624
loss after 1 epochs and  651 iterations is:  0.6414499846509609
loss after 1 epochs and  701 iterations is:  0.638868157805757
loss after 1 epochs and  751 iterations is:  0.6359064566095461
loss after 1 epochs and  801 iterations is:  0.6334408621127018
loss after 1 epochs and  851 iterations is:  0.6312278811574829
loss after 1 epochs and  901 iterations is:  0.6290642794693245
loss after 1 epochs and  951 iterations is:  0.6269938964370426
loss after 1 epochs and  1001 iterations is:  0.6250265021072323
loss after 2 epochs and  1051 iterations is:  0.623029627831275
loss after 2 epochs and  1101 iterations is:  0.6211426245735706
loss after 2 epochs and  1151 iterations is:  0.6193467723886272
loss after 2 epochs and  1201 iterations is:  0.6177270519857514
loss after 2 epochs and  1251 iterations is:  0.616107451679061
loss after 2 epochs and  1301 iterations is:  0.6145681630442259
loss after 2 epochs and  1351 iterations is:  0.6130998421319334
loss after 2 epochs and  1401 iterations is:  0.6116220085039598
loss after 2 epochs and  1451 iterations is:  0.6102084881583923
loss after 2 epochs and  1501 iterations is:  0.6090776658809949
loss after 2 epochs and  1551 iterations is:  0.6077928249914799
loss after 2 epochs and  1601 iterations is:  0.6066410887612778
loss after 2 epochs and  1651 iterations is:  0.605489187290032
loss after 2 epochs and  1701 iterations is:  0.6044380017823323
loss after 2 epochs and  1751 iterations is:  0.6033191282399746
loss after 2 epochs and  1801 iterations is:  0.6022723874689804
loss after 2 epochs and  1851 iterations is:  0.6012802739007284
loss after 2 epochs and  1901 iterations is:  0.6002815235883742
loss after 2 epochs and  1951 iterations is:  0.5993572571306436
loss after 2 epochs and  2001 iterations is:  0.5985389048325674
loss after 3 epochs and  2051 iterations is:  0.597531885791725
loss after 3 epochs and  2101 iterations is:  0.5966720291600264
loss after 3 epochs and  2151 iterations is:  0.5957612970107344
loss after 3 epochs and  2201 iterations is:  0.5950766279432791
loss after 3 epochs and  2251 iterations is:  0.5943559676358053
loss after 3 epochs and  2301 iterations is:  0.5935688486792756
loss after 3 epochs and  2351 iterations is:  0.5929201320587859
loss after 3 epochs and  2401 iterations is:  0.5922333698257596
loss after 3 epochs and  2451 iterations is:  0.591532399110286
loss after 3 epochs and  2501 iterations is:  0.5908821436439172
loss after 3 epochs and  2551 iterations is:  0.5902025650949886
loss after 3 epochs and  2601 iterations is:  0.5895305607797755
loss after 3 epochs and  2651 iterations is:  0.5889999803872425
loss after 3 epochs and  2701 iterations is:  0.5883547840173724
loss after 3 epochs and  2751 iterations is:  0.5876780979157352
loss after 3 epochs and  2801 iterations is:  0.5869976541045953
loss after 3 epochs and  2851 iterations is:  0.586354190320411
loss after 3 epochs and  2901 iterations is:  0.5858000785042473
loss after 3 epochs and  2951 iterations is:  0.5853681835590201
loss after 3 epochs and  3001 iterations is:  0.5849082656704451
loss after 3 epochs and  3051 iterations is:  0.5843228468712194
loss after 4 epochs and  3101 iterations is:  0.5838813200076072
loss after 4 epochs and  3151 iterations is:  0.5833637043570097
loss after 4 epochs and  3201 iterations is:  0.5829071213380829
loss after 4 epochs and  3251 iterations is:  0.5824148615973473
loss after 4 epochs and  3301 iterations is:  0.5819581899421266
loss after 4 epochs and  3351 iterations is:  0.5815316681823375
loss after 4 epochs and  3401 iterations is:  0.5811273217174444
loss after 4 epochs and  3451 iterations is:  0.5806500108258908
loss after 4 epochs and  3501 iterations is:  0.5802987640090631
loss after 4 epochs and  3551 iterations is:  0.5799207302452782
loss after 4 epochs and  3601 iterations is:  0.5794817774182126
loss after 4 epochs and  3651 iterations is:  0.5790229197301442
loss after 4 epochs and  3701 iterations is:  0.5786060170770326
loss after 4 epochs and  3751 iterations is:  0.5781610138754161
loss after 4 epochs and  3801 iterations is:  0.5777774569852753
loss after 4 epochs and  3851 iterations is:  0.5773108705786598
loss after 4 epochs and  3901 iterations is:  0.5769441047961352
loss after 4 epochs and  3951 iterations is:  0.5766059315933955
loss after 4 epochs and  4001 iterations is:  0.5763113930219275
loss after 4 epochs and  4051 iterations is:  0.5759388741544418
loss after 5 epochs and  4101 iterations is:  0.5756060993431625
loss after 5 epochs and  4151 iterations is:  0.575246553578359
loss after 5 epochs and  4201 iterations is:  0.5749124798807108
loss after 5 epochs and  4251 iterations is:  0.5745841461780776
loss after 5 epochs and  4301 iterations is:  0.5742173532508857
loss after 5 epochs and  4351 iterations is:  0.5738795327505383
loss after 5 epochs and  4401 iterations is:  0.5735656165110418
loss after 5 epochs and  4451 iterations is:  0.5732648623169405
loss after 5 epochs and  4501 iterations is:  0.5729635942857142
loss after 5 epochs and  4551 iterations is:  0.5726800641595508
loss after 5 epochs and  4601 iterations is:  0.5723953049698364
loss after 5 epochs and  4651 iterations is:  0.5721378941639194
loss after 5 epochs and  4701 iterations is:  0.5718801656792263
loss after 5 epochs and  4751 iterations is:  0.5715722626379177
loss after 5 epochs and  4801 iterations is:  0.5712414660643983
loss after 5 epochs and  4851 iterations is:  0.5709890468205382
loss after 5 epochs and  4901 iterations is:  0.5707011521495051
loss after 5 epochs and  4951 iterations is:  0.5704481533865665
loss after 5 epochs and  5001 iterations is:  0.5701890963151216
loss after 5 epochs and  5051 iterations is:  0.5698944838917015
loss after 6 epochs and  5101 iterations is:  0.5696446485968748
loss after 6 epochs and  5151 iterations is:  0.5694016597723256
loss after 6 epochs and  5201 iterations is:  0.5691625409290855
loss after 6 epochs and  5251 iterations is:  0.5688782405055512
loss after 6 epochs and  5301 iterations is:  0.5685978558901796
loss after 6 epochs and  5351 iterations is:  0.5683716699949697
loss after 6 epochs and  5401 iterations is:  0.5681131998984631
loss after 6 epochs and  5451 iterations is:  0.567877648390682
loss after 6 epochs and  5501 iterations is:  0.5676691209754645
loss after 6 epochs and  5551 iterations is:  0.5674055585744703
loss after 6 epochs and  5601 iterations is:  0.5671662715742651
loss after 6 epochs and  5651 iterations is:  0.5669546220810727
loss after 6 epochs and  5701 iterations is:  0.5667282388169786
loss after 6 epochs and  5751 iterations is:  0.5664845289698955
loss after 6 epochs and  5801 iterations is:  0.5662618263305396
loss after 6 epochs and  5851 iterations is:  0.5660502583056202
loss after 6 epochs and  5901 iterations is:  0.5658594471733267
loss after 6 epochs and  5951 iterations is:  0.5656598505829907
loss after 6 epochs and  6001 iterations is:  0.5654556703079168
loss after 6 epochs and  6051 iterations is:  0.56523953196772
loss after 6 epochs and  6101 iterations is:  0.5650148196411346
loss after 7 epochs and  6151 iterations is:  0.5648148458149036
loss after 7 epochs and  6201 iterations is:  0.5646040718128554
loss after 7 epochs and  6251 iterations is:  0.5643991606267291
loss after 7 epochs and  6301 iterations is:  0.5642087071952897
loss after 7 epochs and  6351 iterations is:  0.5639798886136619
loss after 7 epochs and  6401 iterations is:  0.5637697779922688
loss after 7 epochs and  6451 iterations is:  0.5635938700178585
loss after 7 epochs and  6501 iterations is:  0.5634007419011422
loss after 7 epochs and  6551 iterations is:  0.5632336573333413
loss after 7 epochs and  6601 iterations is:  0.5630560198278177
loss after 7 epochs and  6651 iterations is:  0.5628596155186233
loss after 7 epochs and  6701 iterations is:  0.5626725040341396
loss after 7 epochs and  6751 iterations is:  0.5624882137621866
loss after 7 epochs and  6801 iterations is:  0.5623161020195231
loss after 7 epochs and  6851 iterations is:  0.5621405959168572
loss after 7 epochs and  6901 iterations is:  0.5619819502922073
loss after 7 epochs and  6951 iterations is:  0.561807800793355
loss after 7 epochs and  7001 iterations is:  0.5616302762617381
loss after 7 epochs and  7051 iterations is:  0.5614702327544377
loss after 7 epochs and  7101 iterations is:  0.5613118533738295
loss after 8 epochs and  7151 iterations is:  0.5611911492022321
loss after 8 epochs and  7201 iterations is:  0.5610054312896275
loss after 8 epochs and  7251 iterations is:  0.5608324094471107
loss after 8 epochs and  7301 iterations is:  0.5606755791833046
loss after 8 epochs and  7351 iterations is:  0.5604966305723635
loss after 8 epochs and  7401 iterations is:  0.5603337762624834
loss after 8 epochs and  7451 iterations is:  0.5601687012512644
loss after 8 epochs and  7501 iterations is:  0.56003002480097
loss after 8 epochs and  7551 iterations is:  0.5598876430161809
loss after 8 epochs and  7601 iterations is:  0.5597290059454819
loss after 8 epochs and  7651 iterations is:  0.5595789965911967
loss after 8 epochs and  7701 iterations is:  0.559439370708113
loss after 8 epochs and  7751 iterations is:  0.5592754208065963
loss after 8 epochs and  7801 iterations is:  0.5591224263983647
loss after 8 epochs and  7851 iterations is:  0.5590040310766788
loss after 8 epochs and  7901 iterations is:  0.5588568847916731
loss after 8 epochs and  7951 iterations is:  0.5587250208401735
loss after 8 epochs and  8001 iterations is:  0.5585784612512226
loss after 8 epochs and  8051 iterations is:  0.558417907070479
loss after 8 epochs and  8101 iterations is:  0.5582779557593237
loss after 9 epochs and  8151 iterations is:  0.5581420237672297
loss after 9 epochs and  8201 iterations is:  0.5580445979884451
loss after 9 epochs and  8251 iterations is:  0.5579117251331686
loss after 9 epochs and  8301 iterations is:  0.5577439161296986
loss after 9 epochs and  8351 iterations is:  0.5576019804441052
loss after 9 epochs and  8401 iterations is:  0.5574714227119149
loss after 9 epochs and  8451 iterations is:  0.5573550480444752
loss after 9 epochs and  8501 iterations is:  0.5572138663408357
loss after 9 epochs and  8551 iterations is:  0.5570859635433245
loss after 9 epochs and  8601 iterations is:  0.556956692251072
loss after 9 epochs and  8651 iterations is:  0.5568205125144519
loss after 9 epochs and  8701 iterations is:  0.556692561786039
loss after 9 epochs and  8751 iterations is:  0.5565803103707045
loss after 9 epochs and  8801 iterations is:  0.5564547026608633
loss after 9 epochs and  8851 iterations is:  0.5563081168936544
loss after 9 epochs and  8901 iterations is:  0.5561928351103262
loss after 9 epochs and  8951 iterations is:  0.5560768647185363
loss after 9 epochs and  9001 iterations is:  0.5559662696282236
loss after 9 epochs and  9051 iterations is:  0.5558732332772022
loss after 9 epochs and  9101 iterations is:  0.5557542203984873
loss after 9 epochs and  9151 iterations is:  0.555625593118027
loss after 10 epochs and  9201 iterations is:  0.5554847786984974
loss after 10 epochs and  9251 iterations is:  0.5553888536294705
loss after 10 epochs and  9301 iterations is:  0.5552717784970336
loss after 10 epochs and  9351 iterations is:  0.555128509377872
loss after 10 epochs and  9401 iterations is:  0.5550285606525429
loss after 10 epochs and  9451 iterations is:  0.554923564124486
loss after 10 epochs and  9501 iterations is:  0.55482762399207
loss after 10 epochs and  9551 iterations is:  0.5547358717760673
loss after 10 epochs and  9601 iterations is:  0.5546270101342646
loss after 10 epochs and  9651 iterations is:  0.5545268467536437
loss after 10 epochs and  9701 iterations is:  0.5544299204787582
loss after 10 epochs and  9751 iterations is:  0.5543240574888451
loss after 10 epochs and  9801 iterations is:  0.5542216683885867
loss after 10 epochs and  9851 iterations is:  0.5541239732119122
loss after 10 epochs and  9901 iterations is:  0.5540103214092645
loss after 10 epochs and  9951 iterations is:  0.5539141403632816
loss after 10 epochs and  10001 iterations is:  0.553793756351588
loss after 10 epochs and  10051 iterations is:  0.5536848274180983
loss after 10 epochs and  10101 iterations is:  0.5535975796837489
loss after 10 epochs and  10151 iterations is:  0.5534920651444988
loss after 11 epochs and  10201 iterations is:  0.553381477596243
loss after 11 epochs and  10251 iterations is:  0.5532719602072351
loss after 11 epochs and  10301 iterations is:  0.553174946250866
loss after 11 epochs and  10351 iterations is:  0.5530700187891654
loss after 11 epochs and  10401 iterations is:  0.5529879371619767
loss after 11 epochs and  10451 iterations is:  0.5529028889790654
loss after 11 epochs and  10501 iterations is:  0.5528120075069572
loss after 11 epochs and  10551 iterations is:  0.5527227534626173
loss after 11 epochs and  10601 iterations is:  0.5526103700931397
loss after 11 epochs and  10651 iterations is:  0.5525429379119121
loss after 11 epochs and  10701 iterations is:  0.5524460537628306
loss after 11 epochs and  10751 iterations is:  0.5523581704601671
loss after 11 epochs and  10801 iterations is:  0.5522723133401622
loss after 11 epochs and  10851 iterations is:  0.5521947644454948
loss after 11 epochs and  10901 iterations is:  0.5520825027181196
loss after 11 epochs and  10951 iterations is:  0.5519933166035718
loss after 11 epochs and  11001 iterations is:  0.5519087915779376
loss after 11 epochs and  11051 iterations is:  0.5518136285923861
loss after 11 epochs and  11101 iterations is:  0.5517239891389172
loss after 11 epochs and  11151 iterations is:  0.5516374120142756
loss after 12 epochs and  11201 iterations is:  0.5515658737584687
loss after 12 epochs and  11251 iterations is:  0.5514762012990191
loss after 12 epochs and  11301 iterations is:  0.5513948221996366
loss after 12 epochs and  11351 iterations is:  0.5513189432305279
loss after 12 epochs and  11401 iterations is:  0.551221813621124
loss after 12 epochs and  11451 iterations is:  0.5511252154330428
loss after 12 epochs and  11501 iterations is:  0.5510311755495879
loss after 12 epochs and  11551 iterations is:  0.5509381113325423
loss after 12 epochs and  11601 iterations is:  0.5508610809845267
loss after 12 epochs and  11651 iterations is:  0.5507766641400444
loss after 12 epochs and  11701 iterations is:  0.5506998417700848
loss after 12 epochs and  11751 iterations is:  0.550638054602862
loss after 12 epochs and  11801 iterations is:  0.5505767388998226
loss after 12 epochs and  11851 iterations is:  0.550497291395151
loss after 12 epochs and  11901 iterations is:  0.5504046964469879
loss after 12 epochs and  11951 iterations is:  0.550331781834684
loss after 12 epochs and  12001 iterations is:  0.550260432566219
loss after 12 epochs and  12051 iterations is:  0.5501907117369469
loss after 12 epochs and  12101 iterations is:  0.5501195584752477
loss after 12 epochs and  12151 iterations is:  0.5500477900330109
loss after 12 epochs and  12201 iterations is:  0.5499768139505086
loss after 13 epochs and  12251 iterations is:  0.5499109132983397
loss after 13 epochs and  12301 iterations is:  0.5498309833565308
loss after 13 epochs and  12351 iterations is:  0.549771388818261
loss after 13 epochs and  12401 iterations is:  0.5496965511927396
loss after 13 epochs and  12451 iterations is:  0.5496140938400194
loss after 13 epochs and  12501 iterations is:  0.549539115576209
loss after 13 epochs and  12551 iterations is:  0.5494855492956467
loss after 13 epochs and  12601 iterations is:  0.5494094085657938
loss after 13 epochs and  12651 iterations is:  0.5493316028948573
loss after 13 epochs and  12701 iterations is:  0.5492563140350402
loss after 13 epochs and  12751 iterations is:  0.5491979481772212
loss after 13 epochs and  12801 iterations is:  0.5491447078250461
loss after 13 epochs and  12851 iterations is:  0.5490756472727727
loss after 13 epochs and  12901 iterations is:  0.5489993369554883
loss after 13 epochs and  12951 iterations is:  0.5489259736410982
loss after 13 epochs and  13001 iterations is:  0.5488520237105091
loss after 13 epochs and  13051 iterations is:  0.5487764619075166
loss after 13 epochs and  13101 iterations is:  0.5487068900319433
loss after 13 epochs and  13151 iterations is:  0.5486429723126187
loss after 13 epochs and  13201 iterations is:  0.548576954914066
loss after 14 epochs and  13251 iterations is:  0.5485066144257112
loss after 14 epochs and  13301 iterations is:  0.5484402220055412
loss after 14 epochs and  13351 iterations is:  0.5483861516730834
loss after 14 epochs and  13401 iterations is:  0.5483093245103873
loss after 14 epochs and  13451 iterations is:  0.5482484298640372
loss after 14 epochs and  13501 iterations is:  0.5481943437165917
loss after 14 epochs and  13551 iterations is:  0.5481246750116509
loss after 14 epochs and  13601 iterations is:  0.5480607208175462
loss after 14 epochs and  13651 iterations is:  0.5479883239196417
loss after 14 epochs and  13701 iterations is:  0.547932114906443
loss after 14 epochs and  13751 iterations is:  0.5478832774457331
loss after 14 epochs and  13801 iterations is:  0.5478191196664128
loss after 14 epochs and  13851 iterations is:  0.5477627692925748
loss after 14 epochs and  13901 iterations is:  0.5476951793177156
loss after 14 epochs and  13951 iterations is:  0.547636942391832
loss after 14 epochs and  14001 iterations is:  0.5475727038831864
loss after 14 epochs and  14051 iterations is:  0.5475029962872622
loss after 14 epochs and  14101 iterations is:  0.5474522011473978
loss after 14 epochs and  14151 iterations is:  0.5473888639126175
loss after 14 epochs and  14201 iterations is:  0.5473191568466856
loss after 14 epochs and  14251 iterations is:  0.5472595767863389
loss after 15 epochs and  14301 iterations is:  0.5471959816915696
loss after 15 epochs and  14351 iterations is:  0.547139600171737
loss after 15 epochs and  14401 iterations is:  0.547070942183188
loss after 15 epochs and  14451 iterations is:  0.5470157722029524
loss after 15 epochs and  14501 iterations is:  0.5469641316728932
loss after 15 epochs and  14551 iterations is:  0.5469066325243004
loss after 15 epochs and  14601 iterations is:  0.5468556034655901
loss after 15 epochs and  14651 iterations is:  0.5467853514796428
loss after 15 epochs and  14701 iterations is:  0.5467286750489505
loss after 15 epochs and  14751 iterations is:  0.5466789227751985
loss after 15 epochs and  14801 iterations is:  0.5466193858166553
loss after 15 epochs and  14851 iterations is:  0.5465650457604271
loss after 15 epochs and  14901 iterations is:  0.5465101745326028
loss after 15 epochs and  14951 iterations is:  0.5464607661008346
loss after 15 epochs and  15001 iterations is:  0.5464136330326027
loss after 15 epochs and  15051 iterations is:  0.5463591185069542
loss after 15 epochs and  15101 iterations is:  0.5463038550222042
loss after 15 epochs and  15151 iterations is:  0.5462387689962773
loss after 15 epochs and  15201 iterations is:  0.5461904968259284
loss after 15 epochs and  15251 iterations is:  0.5461404214621863
CPU times: user 6.8 s, sys: 631 ms, total: 7.43 s
Wall time: 1.86 s
```

In [38]:
```
lg_sgd.score(X_test,y_test)
```

Out[38]:
```
0.7952686270174663
```