Central University of Finance and Economics

中央财经大学

数据挖掘课程

代码文档

吴宇翀

2017310836

WUYUCHONG.COM

指导老师：马景义

2020 年 6 月 10 日

目录

1 简介

此文档为两个算法的代码文档，包括了 **Logisitic 回归**和**神经网络**两个模型算法，文件编码：UTF-8

1. model 文件夹：存放两个算法模型的类
2. source 文件夹：存放依赖函数
3. main.py：包含测试用例 Demo
4. main.ipynb：包含测试用例 Demo

代码同时开源托管在 github: 点击访问

2 更新

在作业第一次提交的基础上，又做了以下改进：

- 为 Logit 算法添加了新功能：引入自动的可变的学习率，在迭代刚开始时较大，以便于损失函数的迅速变化；在迭代中后期则尽量小，以保证精确度。
- 为神经网络算法添加了新功能：可通过 *figure* 函数直接画出损失函数的迭代图。

3 Logistic 回归算法

我们使用**梯度下降**的优化方法构建 logit 模型。

3.1 模型求解步骤

如果 p 是一个事件的概率，这个事件的发生比率就是 p/(1-p)。逻辑回归就是建立模型预测这一比率的对数:

$$\log\left(\frac{p}{1-p}\right) = \beta_0 + \beta_1 x_1 + \cdots + \beta_P x_P$$

即：

$$p = \frac{1}{1 + \exp\left[-\left(\beta_0 + \beta_1 x_1 + \cdots + \beta_P x_P\right)\right]}$$

假设我们有 n 个独立的训练样本 {(x1, y1) ,(x2, y2),..., (xn, yn)}，y={0, 1}。那每一个观察到的样本 (xi, yi) 出现的概率是：

$$P\left(\mathrm{y}_i, \mathrm{x}_i\right) = P\left(\mathrm{y}_i = 1|\mathrm{x}_i\right)^{y_i} \left(1 - P\left(\mathrm{y}_i = 1|\mathrm{x}_i\right)\right)^{1-y_i}$$

那我们的整个样本集，也就是 n 个独立的样本出现的似然函数为:

$$L(\theta) = \prod P\left(\mathrm{y}_i = 1|\mathrm{x}_i\right)^{y_i} \left(1 - P\left(\mathrm{y}_i = 1|\mathrm{x}_i\right)\right)^{1-y_i}$$

那么，损失函数（cost function）就是最大似然函数**取对数**。[1]

用 $L(\theta)$ 对 θ 求导，得到：

$$\frac{\partial L(\theta)}{\partial \theta} = \sum_{i=1}^{n} y_i x_i - \sum_{i}^{n} \frac{e^{\theta^T x_i}}{1 + e^{\theta_{x_i}}} x_i = \sum_{i=1}^{n} \left(y_i - \sigma\left(\theta^T x_i\right)\right) x_i$$

令该导数为 0，无法解析求解。使用梯度下降法 [1] ，那么：

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial L(\theta)}{\partial \theta} = \theta^t - \alpha \sum_{i=1}^{n} \left(y_i - \sigma\left(\theta^T x_i\right)\right) x_j$$

在进行训练之前，我们对各个变量的数据进行标准化处理。

3.2 代码实现 - 逐步讲解（Step by Step）

3.2.1 包和数据导入

```
import numpy as np
import math
from sklearn import datasets
```

我们读取经典的 iris 数据集。[2]

```
iris = datasets.load_iris()
X = iris['data']
y = iris['target']
X = X[y!=2]
y = y[y!=2]
X[0:5]
```

```
## array([[5.1, 3.5, 1.4, 0.2],
##        [4.9, 3. , 1.4, 0.2],
##        [4.7, 3.2, 1.3, 0.2],
##        [4.6, 3.1, 1.5, 0.2],
```

[1] 最大似然法就是求模型中使得似然函数最大的系数取值 *

[2] 只展示前 5 行

```
##        [5. , 3.6, 1.4, 0.2]])
```

```
y[0:5]
```

```
## array([0, 0, 0, 0, 0])
```

3.2.2 定义 sigmoid 函数

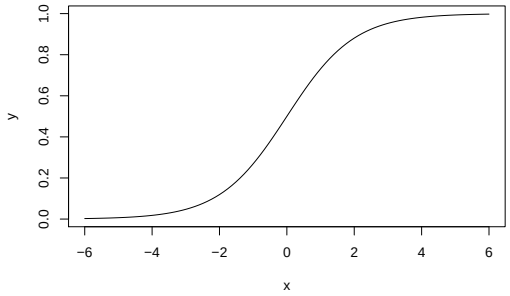

图 1: sigmoid function

适用于向量的 sigmoid 函数

```
def sigmoidVector(Xi,thetas):
    params = - np.sum(Xi * thetas)
    outcome = 1 /(1 + math.exp(params))
    return outcome
```

适用于矩阵的 sigmoid 函数

```
def sigmoidMatrix(Xb,thetas):
    params = - Xb.dot(thetas)
    outcome = np.zeros(params.shape[0])
    for i in range(len(outcome)):
        outcome[i] = 1 /(1 + math.exp(params[i]))
    return outcome
```

带阈值判别的 sigmoid 函数

- 阈值（threshold）：用于给出概率后进行分类，默认为 50%

```
def sigmoidThreshold(Xb,thetas):
    params = - Xb.dot(thetas)
    outcome = np.zeros(params.shape[0])
    for i in range(len(outcome)):
        outcome[i] = 1 /(1 + math.exp(params[i]))
```

```
        if outcome[i] >= 0.5:
            outcome[i] = 1
        else:
            outcome[i] = 0
    return outcome
```

3.2.3 定义损失函数

损失函数（cost function）就是最大似然函数**取对数**

```
def costFunc(Xb,y):
    sum = 0.0
    for i in range(m):
        yPre = sigmoidVector(Xb[i,], thetas)
        if yPre == 1 or yPre == 0:
            return float(-2**31)
        sum += y[i] * math.log(yPre) + (1 - y[i]) * math.log(1-yPre)
    return -1/m * sum
```

3.2.4 用梯度下降法进行训练

初始化：

- 学习率（alpha）：用于调整每次迭代的对损失函数的影响大小
- 准确度（accuracy）：作为终止迭代的评判指标

```
thetas = None
m = 0
alpha = 0.01
accuracy = 0.001
```

在第一列插入 1，构成 Xb 矩阵

```
thetas = np.full(X.shape[1]+1,0.5)
m = X.shape[0]
a = np.full((m, 1), 1)
Xb = np.column_stack((a, X))
n = X.shape[1] + 1
```

使用梯度下降法进行迭代：

```
i = 1
while True:
    before = costFunc(Xb, y)
    c = sigmoidMatrix(Xb, thetas) - y
    for j in range(n):
        thetas[j] = thetas[j] - alpha * np.sum(c * Xb[:,j])
    after = costFunc(Xb, y)
    if after == before or math.fabs(after - before) < accuracy:
        print(" 迭代完成，损失函数值:",after)
        break
    print(" 迭代次数:",i)
    print(" 损失函数变化",(after - before))
    i += 1
```

```
## 迭代次数: 1
## 损失函数变化 5.025121118513411
## 迭代次数: 2
## 损失函数变化 -3.292032434097819
## 迭代次数: 3
## 损失函数变化 -0.9713865369569756
## 迭代次数: 4
## 损失函数变化 2.649715136143148
## 迭代次数: 5
## 损失函数变化 -6.0552442422478245
## 迭代次数: 6
## 损失函数变化 -0.08147412308098556
## 迭代次数: 7
## 损失函数变化 -0.05224409412488061
## 迭代完成，损失函数值: 0.00953683385119681
```

迭代完成后得到的参数：

```
thetas
```

```
## array([ 0.05586408, -0.68733466, -1.75042736,  2.95078531,  1.58964498])
```

3.2.5 预测

进行样本内预测：

```
a = np.full((len(X),1),1)
Xb = np.column_stack((a,X))
sigmoidThreshold(Xb, thetas)
```

```
## array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
##        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
##        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

3.2.6 改进

相比作业预提交，我们添加了一个新的功能，通过新上线的 *auto_fit* 函数，我们可以不用再指定一个固定的学习率，交给程序自动调整学习率。

为了使得学习过程中保持精确的同时，迭代次数尽可能的减少，我们引入一种**可变的学习率**。这种学习率在迭代刚开始时较大，以便于损失函数的迅速变化；在迭代中后期则尽量小，以保证精确度。[2]

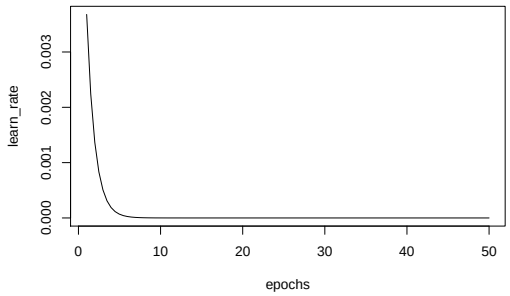

图 2: 递减的学习率函数

3.3 代码实现 - 类封装

可以调整的参数包括：

- 学习率（alpha）：用于调整每次迭代的对损失函数的影响大小
- 准确度（accuracy）：作为终止迭代的评判指标
- 阈值（threshold）：用于给出概率后进行分类，默认为 50%

可以调用的函数包括：

- fit() ：用于固定学习率的拟合
- auto_fit() ：用于自动下降学习率的拟合
- predict() ：用于输出预测结果

```
# ----------------- 导入基本模块 -----------------
import numpy as np
import math

# ------------ 导入 source 中定义的函数 ------------
from source.sigmoidVector import sigmoidVector
from source.sigmoidMatrix import sigmoidMatrix
from source.sigmoidThreshold import sigmoidThreshold

# ----------------- 定义 base 类 -----------------
class Regression(object):
    def __init__(self, X, y, threshold = 0.5):
        self.thetas = None
        self.X = X
        self.y = y

# ---------------- 定义逻辑回归类 ----------------
class LogisticRegression(Regression):
    def __init__(self, X, y, threshold = 0.5):
        Regression.__init__(self, X, y, threshold = 0.5) # 继承 Regression 类
        self.m = 0
        self.threshold = threshold
        self.epoch = 1

    def fit(self, alpha = 0.01, accuracy = 0.001):
        self.thetas = np.full(self.X.shape[1] + 1,0.5)
        self.m = self.X.shape[0]
        a = np.full((self.m, 1), 1)
        Xb = np.column_stack((a,self.X))
        n  = self.X.shape[1]+1

        while True:
            before = self.costFunc(Xb, self.y)
            c = sigmoidMatrix(Xb, self.thetas) - self.y
            for j in range(n):
                self.thetas[j] = self.thetas[j] - alpha * np.sum(c * Xb[:,j])
            after = self.costFunc(Xb, self.y)
            if after == before or math.fabs(after - before) < accuracy:
                break
```

```
            self.epoch += 1

    def auto_alpha(self):
        if self.epoch < 100:
            return 1/math.exp(self.epoch)/100
        else:
            return 1/math.exp(100)

    def auto_fit(self, accuracy = 0.001):
        self.thetas = np.full(self.X.shape[1] + 1,0.5)
        self.m = self.X.shape[0]
        a = np.full((self.m, 1), 1)
        Xb = np.column_stack((a,self.X))
        n  = self.X.shape[1]+1

        while True:
            before = self.costFunc(Xb, self.y)
            c = sigmoidMatrix(Xb, self.thetas) - self.y
            for j in range(n):
                self.thetas[j] = self.thetas[j] - self.auto_alpha() * np.sum(c * Xb[:,j])
            after = self.costFunc(Xb, self.y)
            if after == before or math.fabs(after - before) < accuracy:
                break
            self.epoch += 1

    def costFunc(self, Xb, y):
        sum = 0.0
        for i in range(self.m):
            yPre = sigmoidVector(Xb[i,], self.thetas)
            if yPre == 1 or yPre == 0:
                return float(-2**31)
            sum += y[i] * math.log(yPre) + (1 - y[i]) * math.log(1 - yPre)
        return -1/self.m * sum

    def predict(self):
        a = np.full((len(self.X), 1), 1)
        Xb = np.column_stack((a, self.X))
        return sigmoidThreshold(Xb, self.thetas, self.threshold)
```

3.4 测试用例

我们使用经典的 iris 数据集作为测试用例，为了验证算法正确性，我们进行样本内预测。

```
iris = datasets.load_iris()
X = iris['data']
y = iris['target']
X = X[y!=2]
y = y[y!=2]
```

3.4.1 固定学习率

```
iris = datasets.load_iris()
X = iris['data']
y = iris['target']
X = X[y!=2]
y = y[y!=2]

Logstic = LogisticRegression(X, y, threshold = 0.5)
Logstic.fit(alpha = 0.01, accuracy = 0.001)
print("epoch:",Logstic.epoch)
```

```
## epoch: 8
```

```
print("theta:",Logstic.thetas)
```

```
## theta: [ 0.05586408 -0.68733466 -1.75042736  2.95078531  1.58964498]
```

```
y_predict = Logstic.predict()
y_predict
```

```
## array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
##        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
##        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

3.4.2 自动下降学习率

```
Logstic2 = LogisticRegression(X, y, threshold = 0.5)
Logstic2.auto_fit(accuracy = 0.001)
print("epoch:",Logstic2.epoch)
```

```
## epoch: 6
```

```
print("theta:",Logstic2.thetas)
```

```
## theta: [ 0.34284946 -0.23444331 -0.07464194  0.42403653  0.52057474]
```

```
y_predict = Logstic2.predict()
y_predict
```

```
## array([0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
##        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0.,
##        0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 0., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.,
##        1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1., 1.])
```

可以发现，相比于固定学习率，采用自动下降学习率之后，在相同的精度要求下，迭代的次数从 8 次下降到了 6 次，有着速度优势。

4 神经网络算法

我们使用**随机梯度下降**的优化方法构建一个较为简单的神经网络模型。[3] [3]

4.1 模型求解步骤

4.1.1 神经元

神经元是神经网络的基本单元。[4] 神经元先获得输入，然后执行某些数学运算后，再产生一个输出。[5]

对于一个二输入神经元，输出步骤为：先将两个输入乘以权重，把两个结果想加，再加上一个偏置，最后将它们经过激活函数处理得到输出。[4]

$$y = f(x1 \times weight_1 a + x2 \times weight_1 b + b)$$

[3] 考虑到神经网络是较为复杂的一类模型，在此我们只构造它的初始版本，限制较多，不具备广泛实用性。

[4] 神经元（Neurons）权重（weight）偏置（bias）激活函数（activation function）

我们选择 sigmoid 函数作为神经网络的激活函数。

$$S(x) = \frac{1}{1 + e^{-x}}$$

$$S'(x) = \frac{e^{-x}}{(1 + e^{-x})^2} = S(x)(1 - S(x))$$

4.1.2 神经网络

我们搭建一个具有 2 个输入、一个包含 2 个神经元的隐藏层（h1 和 h2）、包含 1 个神经元的输出层（o1）的简单神经网络。[5] [6]

4.1.3 前馈

把神经元的输入向前传递获得输出的过程称为前馈[6]

4.1.4 损失

在训练神经网络之前，我们需要有一个标准定义，以便我们进行改进。我们采用均方误差（MSE）来定义损失（loss）：

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})$$

我们在训练中每次只取一个样本，那么损失函数为：

$$\begin{aligned}\text{MSE} &= \frac{1}{1}\sum_{i=1}^{1}\left(y_{true} - y_{pred}\right)^2 \\ &= \left(y_{true} - y_{pred}\right)^2 \\ &= \left(1 - y_{pred}\right)^2\end{aligned}$$

4.1.5 训练神经网络

训练神经网络就是将损失最小化，预测结果越好，损失就越低。

由于预测值是由一系列网络权重和偏置计算出来的，所以损失函数实际上是包含多个权重、偏置的多元函数：

[5] A neural network with: - 2 inputs - a hidden layer with 2 neurons (h1, h2) - an output layer with 1 neuron (o1)

[6] 前馈（feedforward）

$$L\left(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3\right)$$

由链式求导法则（以 weight_1a 为例）：

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

这种向后计算偏导数的系统称为反向传导。[7]

4.1.6 随机梯度下降优化方法

我们使用随机梯度下降（SGD）的优化算法 [7] 来逐步改变网络的权重 w 和偏置 b，使损失函数会缓慢降低，从而改进我们的神经网络。以 weight_1a 为例：

$$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$

4.1.7 权重的初始化

神经网络中结点的各权重 (weight) 和偏置 (bias) 的初始化均服从标准正态分布

$$Weight_i, Bias_i \sim N(0, 1)$$

4.2 代码实现 - 逐步讲解（Step by Step）

4.2.1 包和数据导入

```
import numpy as np
import math
```

我们读取经典的 iris 数据集，取其前两个自变量。[8]

```
iris = datasets.load_iris()
X = iris['data']
y = iris['target']
X = X[y!=2][:,0:2]
```

[7] 反向传导（backpropagation）

[8] 只展示前 5 行

```
y = y[y!=2]
X[0:5]
```

```
## array([[5.1, 3.5],
##        [4.9, 3. ],
##        [4.7, 3.2],
##        [4.6, 3.1],
##        [5. , 3.6]])
```

```
y[0:5]
```

```
## array([0, 0, 0, 0, 0])
```

4.2.2 定义 sigmoid 函数

$$f(x) = \frac{1}{1 + e^{-x}}$$

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

4.2.3 定义 sigmoid 的导函数

$$\frac{\mathrm{d}f}{\mathrm{d}x} = f(x) * (1 - f(x))$$

```
def sigmoidDerivative(x):
    fx = sigmoid(x)
    return fx * (1 - fx)
```

4.2.4 定义均方误差损失函数：

$$\mathrm{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})$$

```
def mse(y_true, y_pred):
    return ((y_true - y_pred) ** 2).mean()
```

4.2.5 权重和截距的初始化

$$L\left(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3\right)$$

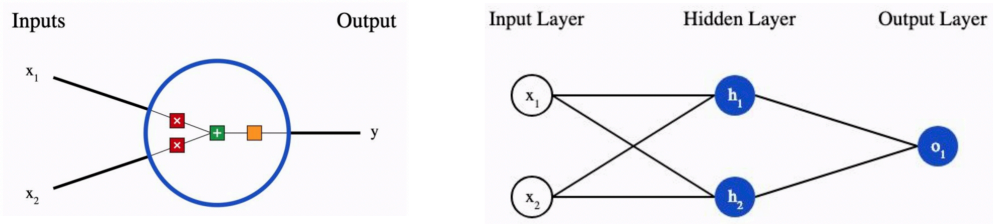

图 3: 搭建的神经网络示意图

```
weight_1a = np.random.normal()
weight_1b = np.random.normal()
weight_2a = np.random.normal()
weight_2b = np.random.normal()
weight_o1 = np.random.normal()
weight_o2 = np.random.normal()

bias_1 = np.random.normal()
bias_2 = np.random.normal()
bias_o = np.random.normal()
```

4.2.6 前馈

将输入向前传递获得输出。

```
def feedforward(x):
    h1 = sigmoid(weight_1a * x[0] + weight_1b * x[1] + bias_1)
    h2 = sigmoid(weight_2a * x[0] + weight_2b * x[1] + bias_2)
    o1 = sigmoid(weight_o1 * h1 + weight_o2 * h2 + bias_o)
    return o1
```

4.2.7 设置学习率和迭代次数

```
learn_rate = 0.1
epochs = 1000
record = np.array((None, None))
```

4.2.8 训练神经网络

```
record = np.array([None, None])
for epoch in range(epochs):
    for x, y_true in zip(X, y):
        # 初次前馈
        sum_h1 = weight_1a * x[0] + weight_1b * x[1] + bias_1
        h1 = sigmoid(sum_h1)

        sum_h2 = weight_2a * x[0] + weight_2b * x[1] + bias_2
        h2 = sigmoid(sum_h2)

        sum_o1 = weight_o1 * h1 + weight_o2 * h2 + bias_o
        o1 = sigmoid(sum_o1)
        y_pred = o1

        # 计算导数
        L_ypred = -2 * (y_true - y_pred)

        # 输出层
        ypred_weight_o1 = h1 * sigmoidDerivative(sum_o1)
        ypred_weight_o2 = h2 * sigmoidDerivative(sum_o1)
        ypred_bias_o = sigmoidDerivative(sum_o1)

        ypred_h1 = weight_o1 * sigmoidDerivative(sum_o1)
        ypred_h2 = weight_o2 * sigmoidDerivative(sum_o1)

        # 隐藏层 1
        h1_weight_1a = x[0] * sigmoidDerivative(sum_h1)
        h1_weight_1b = x[1] * sigmoidDerivative(sum_h1)
        h1_bias_1 = sigmoidDerivative(sum_h1)

        # 隐藏层 2
        h2_weight_2a = x[0] * sigmoidDerivative(sum_h2)
        h2_weight_2b = x[1] * sigmoidDerivative(sum_h2)
        h2_bias_2 = sigmoidDerivative(sum_h2)

        # 迭代权重和偏差
        # 隐藏层 1
```

```
        weight_1a -= learn_rate * L_ypred * ypred_h1 * h1_weight_1a
        weight_1b -= learn_rate * L_ypred * ypred_h1 * h1_weight_1b
        bias_1 -= learn_rate * L_ypred * ypred_h1 * h1_bias_1

        # 隐藏层 2
        weight_2a -= learn_rate * L_ypred * ypred_h2 * h2_weight_2a
        weight_2b -= learn_rate * L_ypred * ypred_h2 * h2_weight_2b
        bias_2 -= learn_rate * L_ypred * ypred_h2 * h2_bias_2

        # 输出层
        weight_o1 -= learn_rate * L_ypred * ypred_weight_o1
        weight_o2 -= learn_rate * L_ypred * ypred_weight_o2
        bias_o -= learn_rate * L_ypred * ypred_bias_o

    # 预测和计算损失函数，并保存
    y_preds = np.apply_along_axis(feedforward, 1, X)
    loss = mse(y, y_preds)
    new = np.array([epoch, loss])
    record = np.vstack([record,new])
```

4.2.9 查看损失函数的变化

```
record[1:10,1]
```

```
## array([0.3025091646995126, 0.309669036464365, 0.3108323621374483,
##        0.3096646011407177, 0.306638357657948, 0.30202924488354843,
##        0.29604572521415135, 0.28875907538207657, 0.280338931076833],
##       dtype=object)
```

我们画出损失函数的变化图像，可以看出，随着迭代次数的增加，均方误差先快速减小，后趋向于稳定。

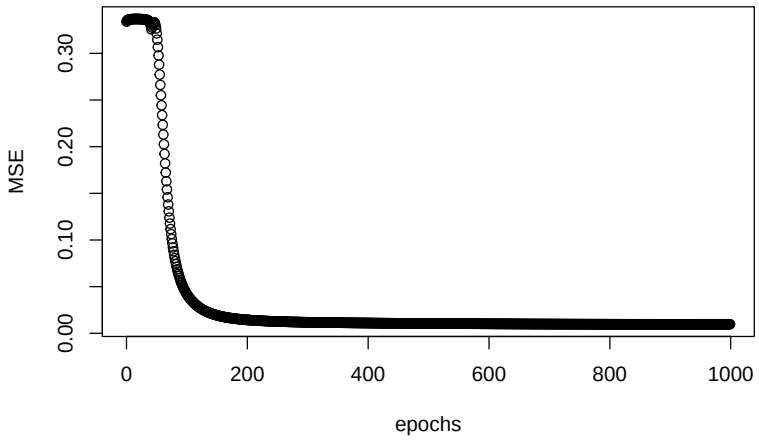

图 4: 损失函数的变化图像

4.2.10 预测

为了验证算法正确性，我们进行样本内预测。

预测第 1 个样本:

```
X[0]
```

```
## array([5.1, 3.5])
```

```
y[0]
```

```
## 0
```

```
y_preds = feedforward(X[0])
y_preds
```

```
## 0.0025859167404758162
```

预测第 100 个样本:

```
X[99]
```

```
## array([5.7, 2.8])
```

```
y[99]
```

```
## 1
```

```
y_preds = feedforward(X[99])
y_preds
```

```
## 0.978945733363624
```

4.2.11 改进

相比作业预提交，我们添加了一个新的功能，通过新上线的 *figure* 函数，我们可以直接画出损失函数的迭代图。

4.3 代码实现 - 类封装

可以调整的参数包括：

- 迭代次数（epochs）：理论上迭代次数越多越精确，耗时越长，默认为 100 次
- 学习率（learn_rate）：默认为 0.1，可进一步调优

可以调用的函数包括：

- train() ：用于训练神经网络模型
- predict() ：用于输出预测结果
- figure() ：用于画出损失函数迭代图

```
# ----------------- 导入基本模块 -----------------
import numpy as np
import math

# ------------ 导入 source 中定义的函数 ------------
from source.sigmoid import sigmoid
from source.sigmoidDerivative import sigmoidDerivative
from source.mse import mse

# ---------------- 定义神经网络类 ----------------
class NeuralNetwork:
    def __init__(self, X, y, learn_rate = 0.1, epochs = 100):
        # 学习率和迭代次数
        self.learn_rate = learn_rate
        self.epochs = epochs
        # 数据
        self.X = X
        self.y = y
        # 权重和截距的初始化
        self.weight_1a = np.random.normal()
        self.weight_1b = np.random.normal()
        self.weight_2a = np.random.normal()
        self.weight_2b = np.random.normal()
```

```
        self.weight_o1 = np.random.normal()
        self.weight_o2 = np.random.normal()
        self.bias_1 = np.random.normal()
        self.bias_2 = np.random.normal()
        self.bias_o = np.random.normal()
        # 记录损失函数值
        self.record = np.array([None, None])

    # 用于画 MSE 的图像
    def figure(self):
        x = self.record[1:,0]
        y = self.record[1:,1]
        plt.title("Variation of the Loss Function")
        plt.xlabel("epochs")
        plt.ylabel("MSE")
        plt.plot(x,y,"ob")
        plt.show()

    # 定义前馈
    def feedforward(self, x):
        h1 = sigmoid(self.weight_1a * x[0] + self.weight_1b * x[1] + self.bias_1)
        h2 = sigmoid(self.weight_2a * x[0] + self.weight_2b * x[1] + self.bias_2)
        o1 = sigmoid(self.weight_o1 * h1 + self.weight_o2 * h2 + self.bias_o)
        return o1

    # 定义预测函数
    def predict(self):
        y_preds = np.apply_along_axis(self.feedforward, 1, self.X)
        return y_preds

    # 定义训练函数
    def train(self):
        for epoch in range(self.epochs):
            for x, y_true in zip(self.X, self.y):
                # 初次前馈
                sum_h1 = self.weight_1a * x[0] + self.weight_1b * x[1] + self.bias_1
                h1 = sigmoid(sum_h1)

                sum_h2 = self.weight_2a * x[0] + self.weight_2b * x[1] + self.bias_2
```

```
        h2 = sigmoid(sum_h2)

        sum_o1 = self.weight_o1 * h1 + self.weight_o2 * h2 + self.bias_o
        o1 = sigmoid(sum_o1)
        y_pred = o1

        # 计算导数
        L_ypred = -2 * (y_true - y_pred)

        # 输出层
        ypred_weight_o1 = h1 * sigmoidDerivative(sum_o1)
        ypred_weight_o2 = h2 * sigmoidDerivative(sum_o1)
        ypred_bias_o = sigmoidDerivative(sum_o1)

        ypred_h1 = self.weight_o1 * sigmoidDerivative(sum_o1)
        ypred_h2 = self.weight_o2 * sigmoidDerivative(sum_o1)

        # 隐藏层 1
        h1_weight_1a = x[0] * sigmoidDerivative(sum_h1)
        h1_weight_1b = x[1] * sigmoidDerivative(sum_h1)
        h1_bias_1 = sigmoidDerivative(sum_h1)

        # 隐藏层 2
        h2_weight_2a = x[0] * sigmoidDerivative(sum_h2)
        h2_weight_2b = x[1] * sigmoidDerivative(sum_h2)
        h2_bias_2 = sigmoidDerivative(sum_h2)

        # # 迭代权重和偏差
        # 隐藏层 1
        self.weight_1a -= self.learn_rate * L_ypred * ypred_h1 * h1_weight_1a
        self.weight_1b -= self.learn_rate * L_ypred * ypred_h1 * h1_weight_1b
        self.bias_1 -= self.learn_rate * L_ypred * ypred_h1 * h1_bias_1

        # 隐藏层 2
        self.weight_2a -= self.learn_rate * L_ypred * ypred_h2 * h2_weight_2a
        self.weight_2b -= self.learn_rate * L_ypred * ypred_h2 * h2_weight_2b
        self.bias_2 -= self.learn_rate * L_ypred * ypred_h2 * h2_bias_2

        # 输出层
```

```
                self.weight_o1 -= self.learn_rate * L_ypred * ypred_weight_o1
                self.weight_o2 -= self.learn_rate * L_ypred * ypred_weight_o2
                self.bias_o -= self.learn_rate * L_ypred * ypred_bias_o

            # 计算损失函数，并保存
            y_preds = np.apply_along_axis(self.feedforward, 1, self.X)
            loss = mse(self.y, y_preds)
            new = np.array([epoch, loss])
            self.record = np.vstack([self.record, new])
```

4.4 测试用例

我们使用经典的 iris 数据集作为测试用例。为了验证算法正确性，我们进行样本内预测。

```
iris = datasets.load_iris()
X = iris['data']
y = iris['target']
X = X[y!=2][:,0:2]
y = y[y!=2]

np.random.seed(1)
model = NeuralNetwork(X, y, learn_rate = 0.1, epochs = 1000)
model.train()
model.predict()
```

```
## array([0.00612189, 0.0630929 , 0.00633335, 0.00679723, 0.00507907,
##        0.00506943, 0.00485548, 0.00650541, 0.00825689, 0.02099575,
##        0.00625819, 0.00527099, 0.02816319, 0.00547294, 0.00604891,
##        0.00482305, 0.00506943, 0.00612189, 0.00855735, 0.0048618 ,
##        0.03348889, 0.00501548, 0.00471269, 0.01310494, 0.00527099,
##        0.15951826, 0.00650541, 0.00723052, 0.0109153 , 0.00633335,
##        0.01237574, 0.03348889, 0.00477186, 0.00482469, 0.02099575,
##        0.01628059, 0.02447208, 0.00491453, 0.00615798, 0.0079752 ,
##        0.00551203, 0.94783952, 0.00493599, 0.00551203, 0.0048618 ,
##        0.02816319, 0.0048618 , 0.00559769, 0.00560638, 0.00898936,
##        0.98109248, 0.97997156, 0.98104199, 0.98000492, 0.98081001,
##        0.97862178, 0.97727701, 0.97327073, 0.98088105, 0.96609978,
##        0.97936396, 0.97762857, 0.98022802, 0.98026693, 0.97322983,
##        0.98093453, 0.96338667, 0.98001792, 0.98030098, 0.98000188,
```

```
##         0.96480269, 0.98048591, 0.98057894, 0.98048591, 0.98076196,
##         0.98088815, 0.98090211, 0.98094353, 0.97988157, 0.98002613,
##         0.97994905, 0.97994905, 0.98001792, 0.98044387, 0.91186996,
##         0.92324553, 0.98093453, 0.98042374, 0.96338667, 0.97972117,
##         0.97913902, 0.97973749, 0.98024183, 0.97809794, 0.97892728,
##         0.97121131, 0.97640171, 0.9805073 , 0.97525214, 0.97862178])
```

在此数据集下，我们将迭代次数设置在 **100** 次，训练后发现在有较小的几率判别效果不佳。

但在迭代次数设置为 **1000** 次后，基本没有出现上述情况。

参考文献

[1] 汪宝彬, 汪玉霞. 随机梯度下降法的一些性质 [J]. 数学杂志, 2011, 31(6): 1041–1044.

[2] 李丰. 统计计算课程 [EB/OL]. https://feng.li/teaching/statcomp/.

[3] ZHOU V. Machine Learning for Beginners: An Introduction to Neural Networks[EB/OL]. https://victorzhou.com/blog/intro-to-neural-networks/.

[4] CHUA L O, YANG L. Cellular neural networks:theory[C]//IEEE International Workshop on Cellular Neural Networks & Their Applications. 1988.

[5] 庄镇泉, 王东生. 神经网络与神经计算机: 第二讲神经网络的学习算法 [J]. 电子技术应用, 1990(5): 38–41.

[6] 阎平凡. 人工神经网络与模拟进化计算 (第 2 版)[M]. 2005.

[7] 王功鹏, 段萌, 牛常勇. 基于卷积神经网络的随机梯度下降算法 [J]. 计算机工程与设计, 2018, 39(2): 441–445.