中山大学计算机学院

人工智能

本科生实验报告

（2022 学年春季学期）

课程名称：Artificial Intelligence

教学班级	计科 2 班	专业（方向）	计算机科学与技术
学号	22336126	姓名	李漾

一、 实验题目

购房预测分类任务

二、 实验内容

1. 算法原理

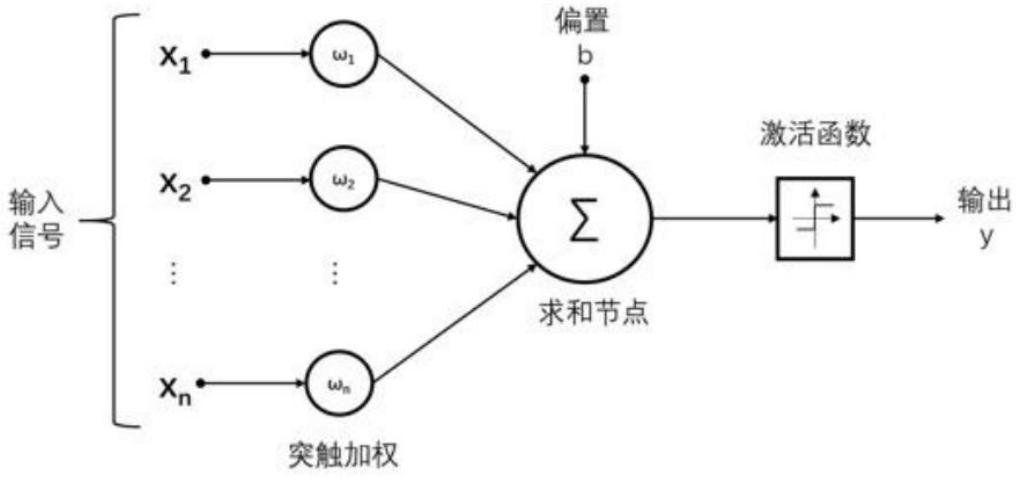
（1）神经网络介绍

• 神经网络概述

• 定义：神经网络采用了仿生学的思想，通过模拟生物神经网络的结构和功能来实现建模。人工神经网络从信息处理的角度对生物神经系统进行抽象，建立某种简单模型，按不同的连接方式组成不同的网络，在工程与学术界也常简称为神经网络。

• 单层感知机

1957 年 Frank Rosenblatt 提出了一种简单的人工神经网络，被称之为感知机。早期的感知机结构和 MCP 模型相似，由一个输入层和一个输出层构成，因此也被称为“单层感知机”。感知机的输入层负责接收实数值的输入向量，输出层则为 1 或-1 两个值。单层感知机可作为一种二分类线性分类模型，结构如图所示。



单层感知机的基本公式为：

$$y(x) = sign(w\,x + b)$$

• 多层感知机

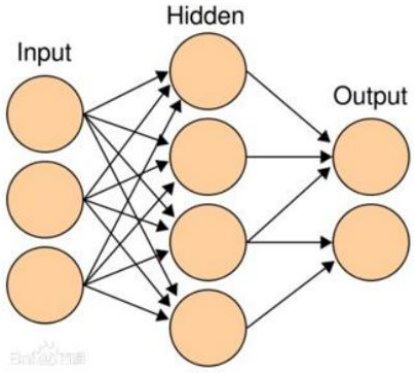
由于无法模拟诸如异或以及其他复杂函数的功能，使得单层感知机的应用较为单一。

一个简单的想法是，如果能在感知机模型中增加若干隐藏层，增强神经网络的非线性表达能力，就会让神经网络具有更强拟合能力。因此，由多个隐藏层构成的多层感知机被提出。

如图所示，多层感知机由输入层、输出层和至少一层的隐藏层构成。网络中各个隐藏层中神经元可接收相邻前序隐藏层中所有神经元传递而来的信息，经过加工处理后将信息输出给相邻后续隐藏层中所有神经元

在多层感知机中，相邻层所包含的神经元之间通常使用“全连接”方式进行连接。

所谓“全连接”是指两个相邻层之间的神经元相互成对连接，但同一层内神经元之间没有连接。多层感知机可以模拟复杂非线性函数功能，所模拟函数的复杂性取决于网络隐藏层数目和各层中神经元数目



$$H = g(XW^{(1)} + b^{(1)})$$
$$\hat{y} = g(HW^{(2)} + b^{(2)})$$

激活函数

- 两层神经网络不再使用sign作为函数g，而是使用一些平滑的函数作为函数g，该函数也被称作激活函数
- 常用的激活函数有：
  - Sigmoid: $f(z) = \frac{1}{1+e^{-z}}$
  - ReLU: $f(z) = \max(0, z)$
  - Softmax: $f(z) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$

损失函数

- 作用：为了衡量网络表现是否良好，并为之后的网络参数优化提供指导。
- 常见的用在分类任务上的损失函数：
  - 均方误差(MSE): $L_{MSE} = \frac{1}{2n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$
  - 交叉熵: $L_{CE} = -\sum_{i}^{n} y^{(i)} \log \hat{y}^{(i)} + \left(1 - y^{(i)}\right) \log(1 - \hat{y}^{(i)})$

• 梯度下降

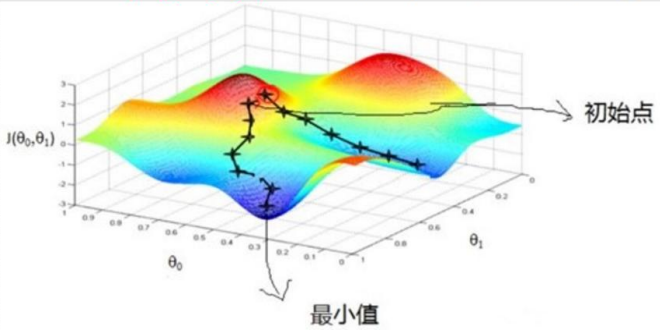
梯度定义：梯度是一个向量，表示某一函数在该点出的方向导数沿着该方向取得最大值。

也就是说该点处沿着梯度的方向变化最快，变化率最大人工智能实验

沿着梯度方向容易找到函数最大值

• 沿着梯度方向的反方向，容易找到函数最小值

梯度下降的一般公式为：$\theta = \theta - \eta\nabla_\theta L(\theta)$，其中，$\eta$是学习率，$\nabla_\theta$是对$\theta$的梯度，$\theta$是参数



2. 关键代码展示（带注释）

逻辑回归模型：

初始化：

lr=0.1：学习率，用于控制参数更新的步长，默认为 0.1。

num_iterations=10000：迭代次数，用于控制算法训练的迭代次数，默认为 10000。

self.weights：模型的权重，初始化为 None。

self.bias：模型的偏置，初始化为 None。

self.loss：用于存储每次迭代的损失值的列表，初始化为空。

归一化：

用于将线性模型的输出转换为 0 到 1 之间的概率值。

模型训练：

采用输入特征矩阵 X 和目标变量 y。

首先初始化模型的权重和偏置为零向量和零，然后进行迭代训练。

在每次迭代中，计算线性模型的输出，将其传递给 sigmoid 函数得到预测概率值。

计算损失函数，并保存到 self.loss 列表中。

计算损失函数对权重和偏置的梯度，并根据学习率更新权重和偏置。
预测：
输入：特征矩阵 X。
计算线性模型的输出：linear_model = np.dot(X, self.weights) + self.bias。
将线性模型的输出传递给 sigmoid 函数，得到预测的类别概率值：y_predicted = self.sigmoid(linear_model)。
根据概率值判断类别，大于 0.5 的设置为 1，小于等于 0.5 的设置为 0，得到预测的类别标签数组。
返回预测的类别标签数组。

```
class LogisticRegression:
    def __init__(self, lr=0.1, num_iterations=10000):
        self.lr = lr
        self.num_iterations = num_iterations
        self.weights = None
        self.bias = None
        self.loss = []

    def sigmoid(self, z):
        return 1 / (1 + np.exp(-z))

    def fit(self, X, y):
        n_samples, n_features = X.shape
        self.weights = np.zeros(n_features)
        self.bias = 0

        for _ in range(self.num_iterations):
            linear_model = np.dot(X, self.weights) + self.bias
            y_predicted = self.sigmoid(linear_model)

            dw = (1 / n_samples) * np.dot(X.T, (y_predicted - y))
            db = (1 / n_samples) * np.sum(y_predicted - y)

            self.weights -= self.lr * dw
            self.bias -= self.lr * db

            # 计算损失并保存
            loss = - (1 / n_samples) * np.sum(y * np.log(y_predicted + 1e-8) + (1 - y) * np.log(1 - y_predicted + 1e-8))
            self.loss.append(loss)

    def predict(self, X):
        linear_model = np.dot(X, self.weights) + self.bias
        y_predicted = self.sigmoid(linear_model)
        y_predicted_cls = [1 if i > 0.5 else 0 for i in y_predicted]
        return np.array(y_predicted_cls)
```

感知机模型：
初始化：
input_size 表示输入特征的维度。
learning_rate 是学习率，控制模型参数在每次迭代中的更新步长，默认为 0.001。
max_iters 是最大迭代次数，用于控制模型训练的停止条件，默认为 10000。
self.loss 是用于存储每次迭代的损失值的列表。
self.weights 是感知机模型的参数，包括权重和偏置，初始化为一个长度为 input_size + 1 的随机向量，权重范围在 [-1, 1]。

前向传播：模型根据当前权重和偏置计算预测值。
在反向传播：模型根据预测值和真实标签更新权重和偏置，以减小损失函数。
拟合方法：模型对训练集进行多次迭代，以调整权重和偏置，以降低损失函数。
模型训练：
使用训练集对感知机模型进行训练，通过多次迭代来逐步优化模型参数。

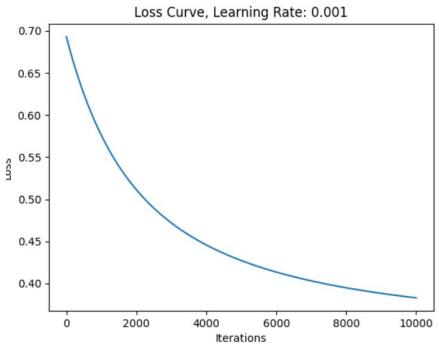
每 100 次迭代输出一次损失值，以便观察模型的训练情况。

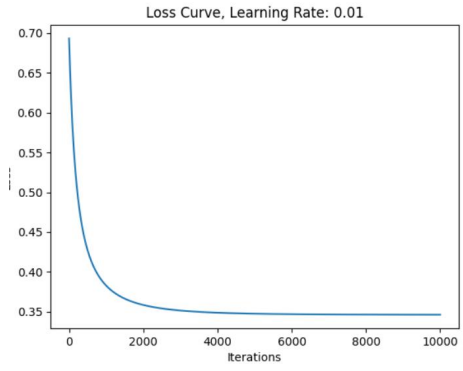
```
class Perceptron:
    def __init__(self, input_size, learning_rate=0.001, max_iters=10000):  # 减小学习率，增加迭代次数
        self.input_size = input_size
        self.learning_rate = learning_rate
        self.max_iters = max_iters
        self.loss = []

        # 初始化权重和偏置
        self.weights = np.random.rand(self.input_size + 1) * 2 - 1  # 权重范围在 [-1, 1]

    def forward(self, X):
        # 前向传播
        return np.dot(X, self.weights[1:]) + self.weights[0]

    def backward(self, X, y, y_pred):
        # 反向传播
        error = y - y_pred
        self.weights[1:] += self.learning_rate * np.dot(X.T, error)
        self.weights[0] += self.learning_rate * np.sum(error)

    def fit(self, X, y):
        for i in range(self.max_iters):
            # 前向传播
            y_pred = self.forward(X)

            # 计算损失
            loss = np.mean(np.square(y - y_pred))
            self.loss.append(loss)

            # 反向传播并更新权重和偏置
            self.backward(X, y, y_pred)

            # 每100次迭代打印一次损失
            if i % 100 == 0:
                print(f"Iteration {i}, Loss: {loss}")
```

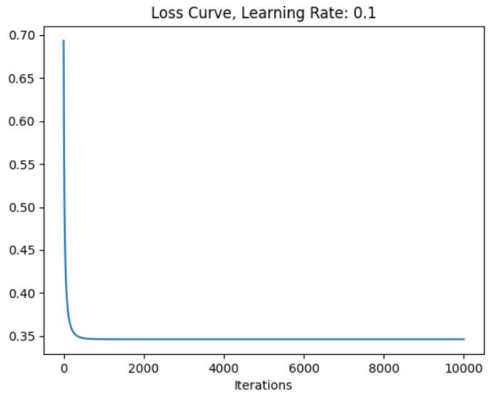
3. 创新点&优化（如果有）

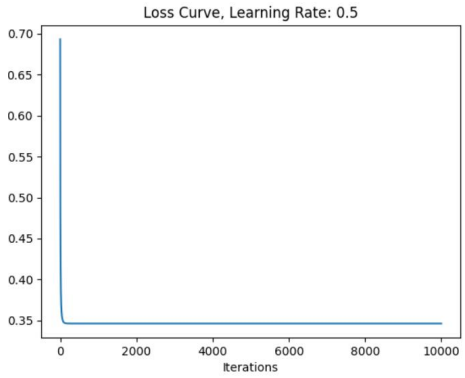
在逻辑回归模型中：

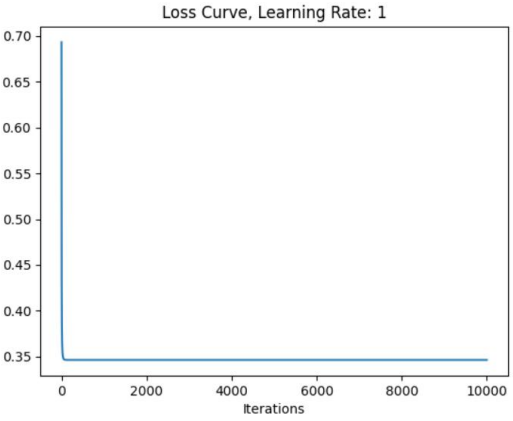
可视化不同学习率的影响

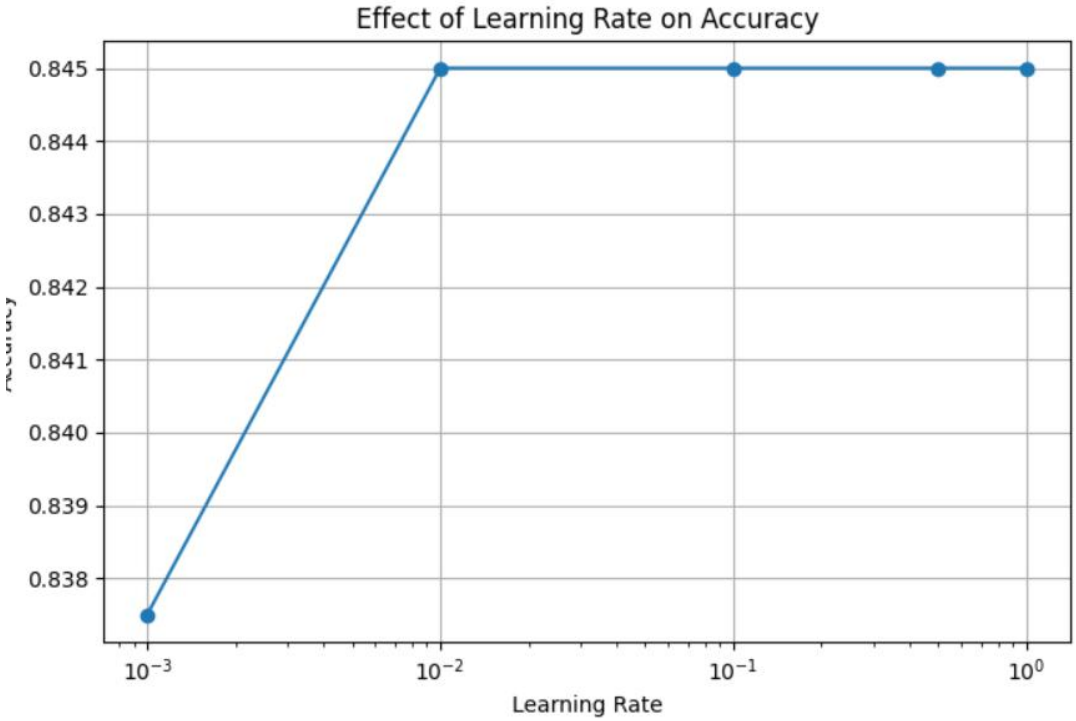





Loss Curve, Learning Rate: 0.1
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0
2000
4000
6000
8000
10000
Iterations


Loss Curve, Learning Rate: 0.5
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0
2000
4000
6000
8000
10000
Iterations

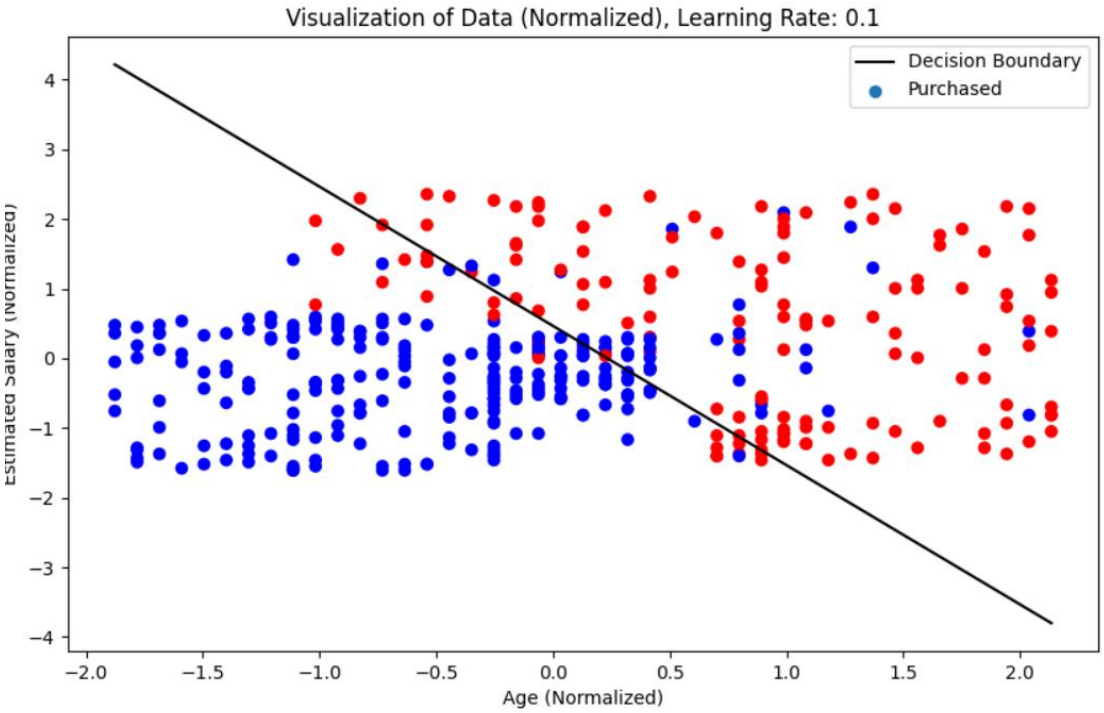

Loss Curve, Learning Rate: 1
0.70
0.65
0.60
0.55
0.50
0.45
0.40
0.35
0
2000
4000
6000
8000
10000
Iterations

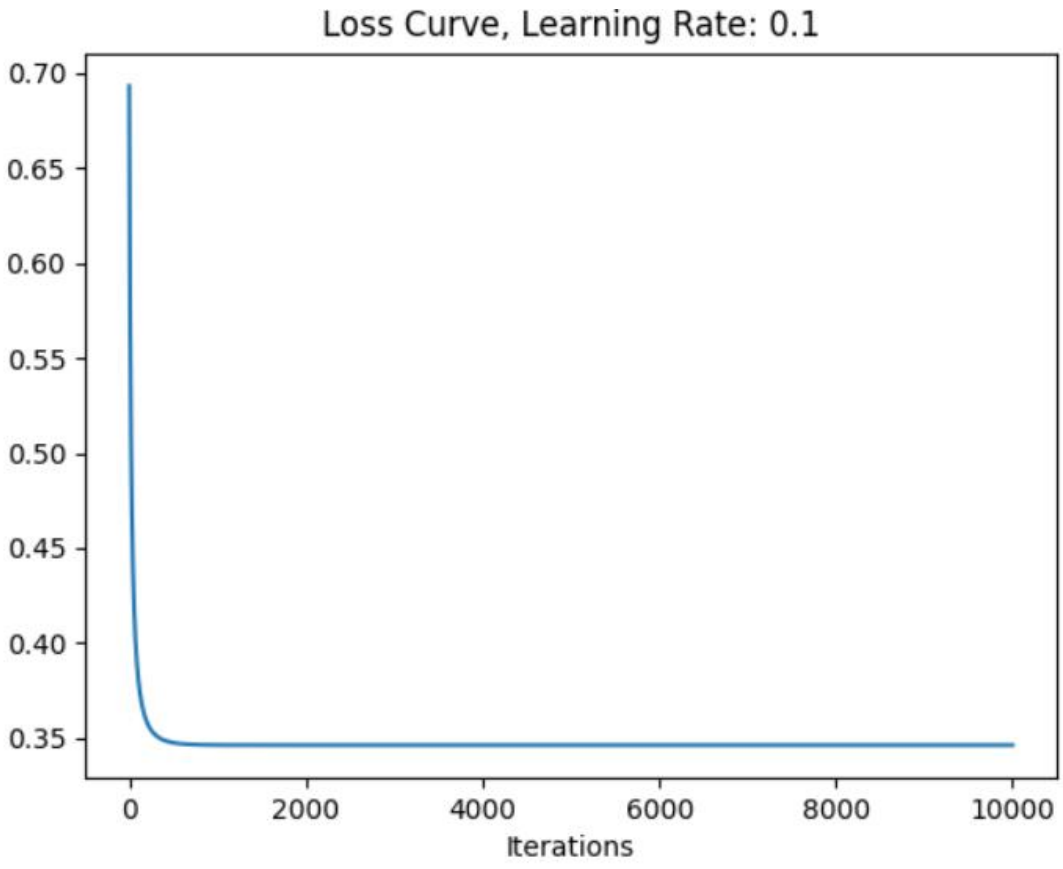

Effect of Learning Rate on Accuracy
0.845
0.844
0.843
0.842
0.841
0.840
0.839
0.838
10^{-3}
10^{-2}
10^{-1}
10^{0}
Learning Rate

三、 实验结果及分析

1. 实验结果展示示例（可图可表可文字，尽量可视化）

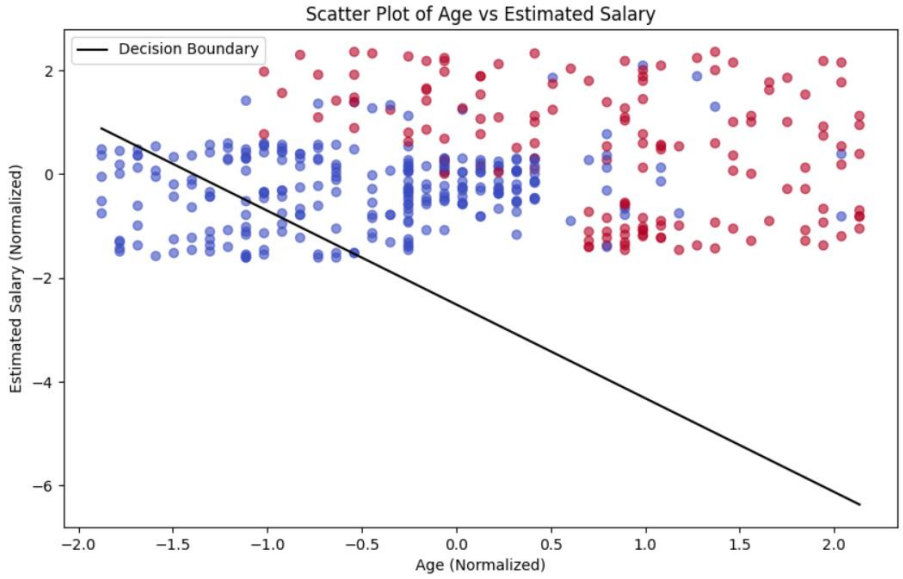
逻辑回归模型：

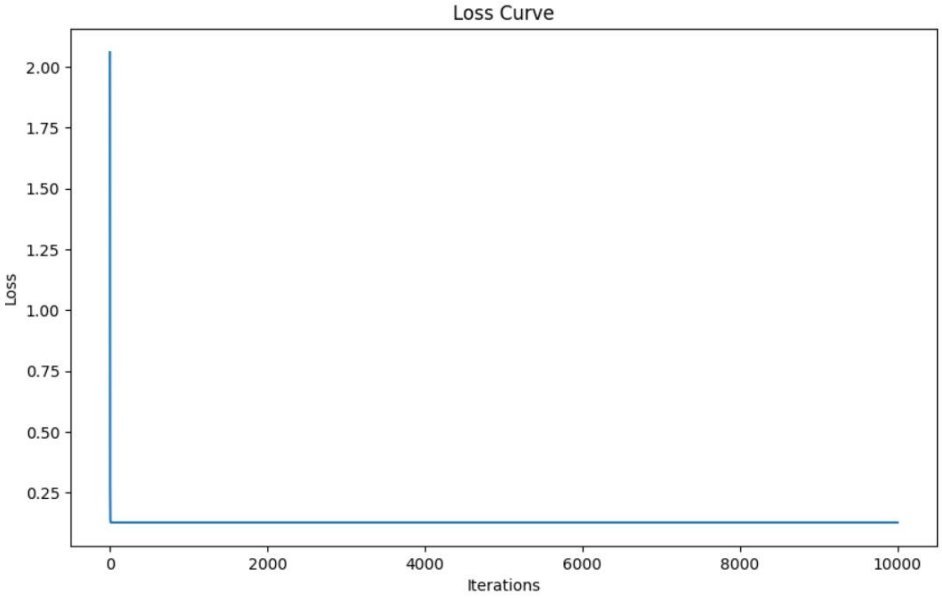




```
Learning Rate: 0.001, Accuracy: 0.8375
Learning Rate: 0.01, Accuracy: 0.845
Learning Rate: 0.1, Accuracy: 0.845
Learning Rate: 0.5, Accuracy: 0.845
Learning Rate: 1, Accuracy: 0.845
```

感知机模型：





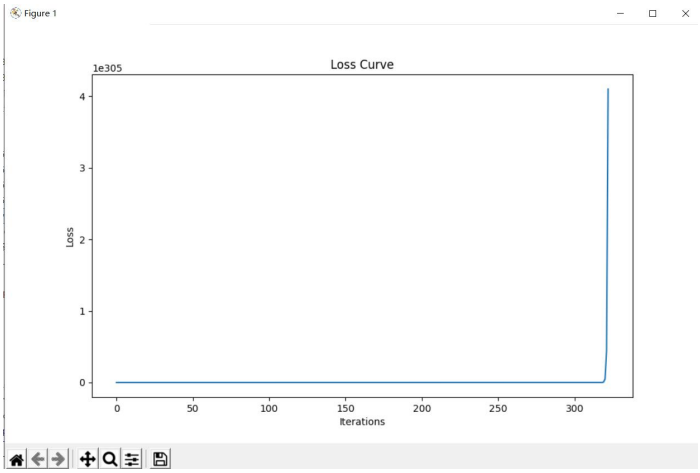

```
训练集准确率: 0.5125
测试集准确率: 0.475
```

2. 评测指标展示及分析（机器学习实验必须有此项，其它可分析运行时间等）

α 在梯度下降法中称为学习率或者步长，意味着我们可以通过 α 来控制下山每一步走的距离，以保证不要步子跨的太大，错过最低点，同时也保证不要走得太

慢，迟迟走不到最低点。

在感知机模型中，学习率太大时，loss 函数就出现了不收敛的情况



学习率太小时，收敛得就太慢：