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	202320346	专业（方向）	计算机科学与技术
学号	21312450	姓名	林隽哲

一、 实验题目

购房预测分类任务

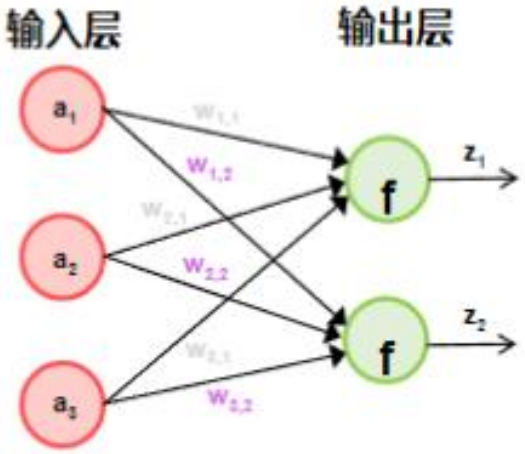
> 购房预测分类任务
>
> data.csv数据集包含三列共400条数据，其中第一列Age表示用户年龄，第二列EstimatedSalary表示用户估计的薪水，第三列Purchased表示用户是否购房。请根据用户的年龄以及估计的薪水，利用逻辑回归算法和感知机算法预测用户是否购房，并画出数据可视化图、loss曲线图，计算模型收敛后的分类准确率。

二、 实验内容

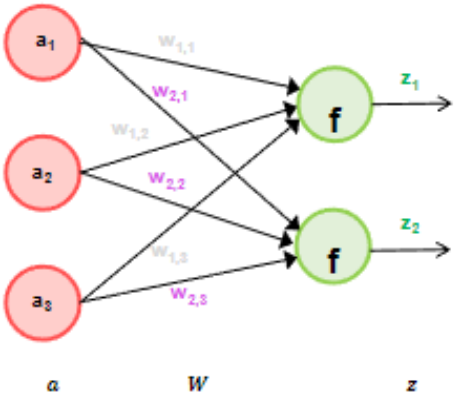
1. 算法原理

单层感知机

- **感知机**（Perceptron）：由两层神经元构成
  - 输入层：在 MP 模型的输入位置添加神经元节点，只传输数据，不做计算
  - 输出层：在前面一层的输入进行计算

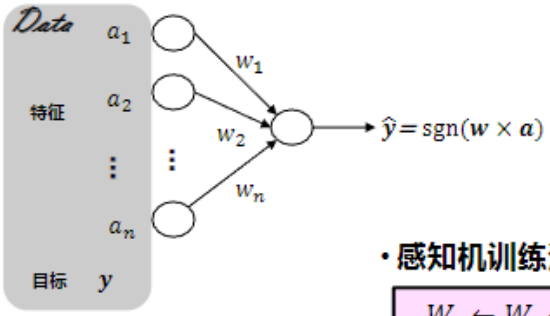


- **感知机的数学化公式：**
  - 输入向量 $\boldsymbol{\alpha} = [a_1, a_2, a_3]^T$，输出向量 $\boldsymbol{z} = [z_1, z_2]^T$，系数矩阵 $\boldsymbol{W} \in R^{2\times3}$
  - 则用矩阵乘法表达感知机的计算公式为：$\boldsymbol{z} = g(\boldsymbol{W} \times \boldsymbol{\alpha})$
  - 其中，g 为激活函数，对于单层感知机我们使用 **sgn** 函数作为激活函数



$$z_1 = g(a_1 w_{1,1} + a_2 w_{1,2} + a_3 w_{1,3})$$
$$z_2 = g(a_1 w_{2,1} + a_2 w_{2,2} + a_1 w_{2,3})$$

- **感知机的训练法则：**感知机中的权重是通过训练得到的



$$\min_{\boldsymbol{W}} |\boldsymbol{y} - \hat{\boldsymbol{y}}|$$

·感知机训练法则：

$$W_t \leftarrow W_t + \Delta W_t$$
$$\Delta W_t = \eta(y - \hat{y})\boldsymbol{a}$$

y：是真实的目标
$\hat{y}$：是感知机的输出
η：学习速率（如 0.1）
$\boldsymbol{a}$：训练数据

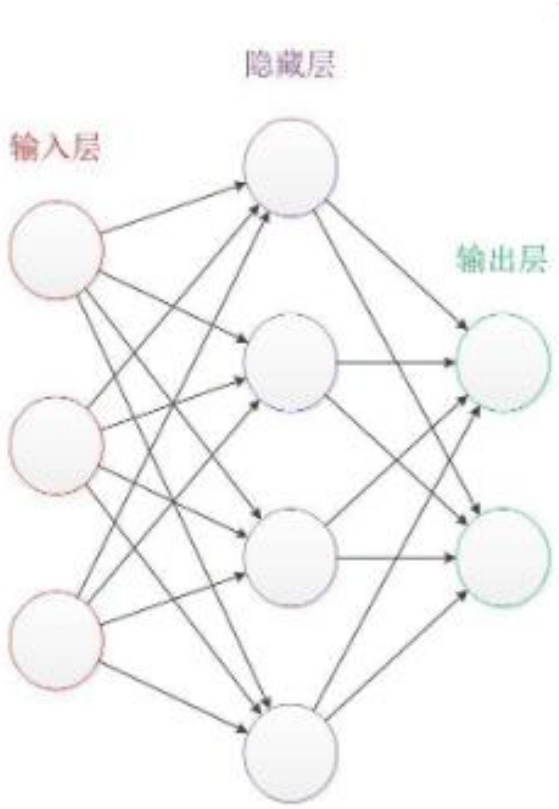
- **单层感知机的损失函数：**我在本次实验中使用**均方误差（MSE）**作为单层感知机的损失函数

$$\text{均方误差(MSE):}\quad L_{MSE} = \frac{1}{2n}\sum_{i=1}^{n}\left(y^{(i)} - \hat{y}^{(i)}\right)^2$$

- **单层感知机的梯度：**不难计算出感知机的梯度为 $y - \hat{y}$

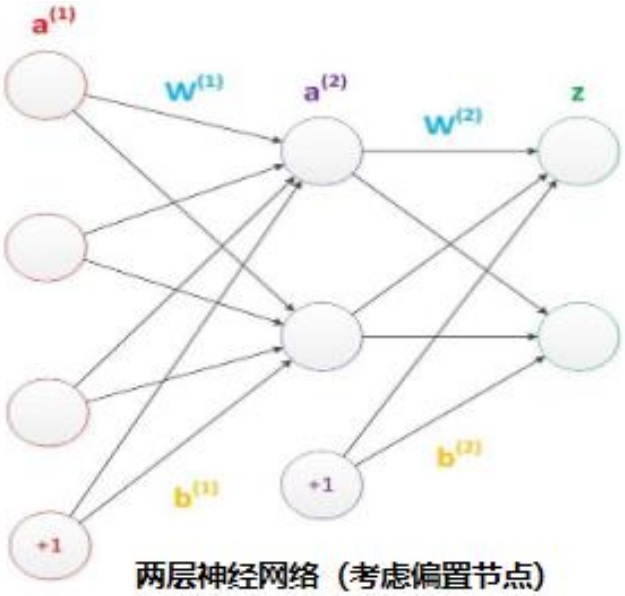
多层感知机

- **两层神经网络（多层感知机（MLP））：**一个经典的神经网络包含三个次：**输入层**、**输出层**与**中间层**（也称**隐藏层**）



神经网络结构图

- **偏置单元：**多层感知机一般还会引入偏置单元，它与后一层的所有节点都有连接，如下图：



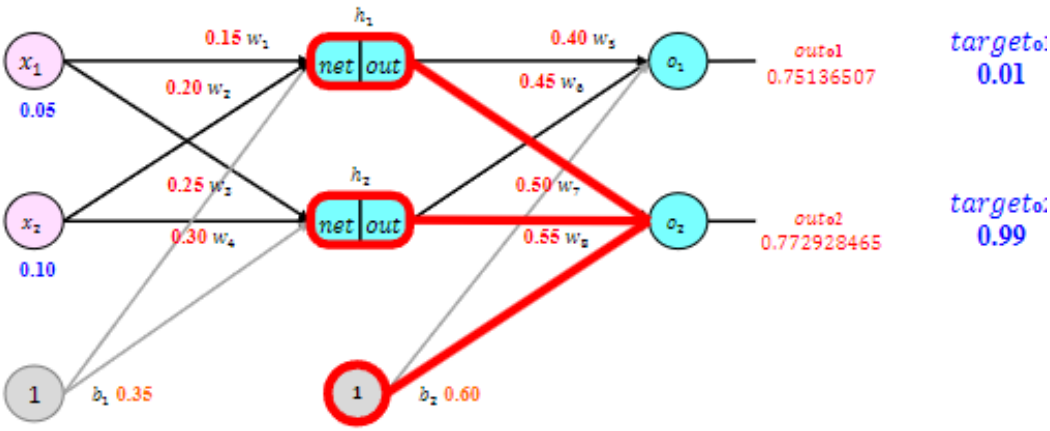
两层神经网络（考虑偏置节点）

- **两层神经网络的数学化运算（考虑偏执单元）：**

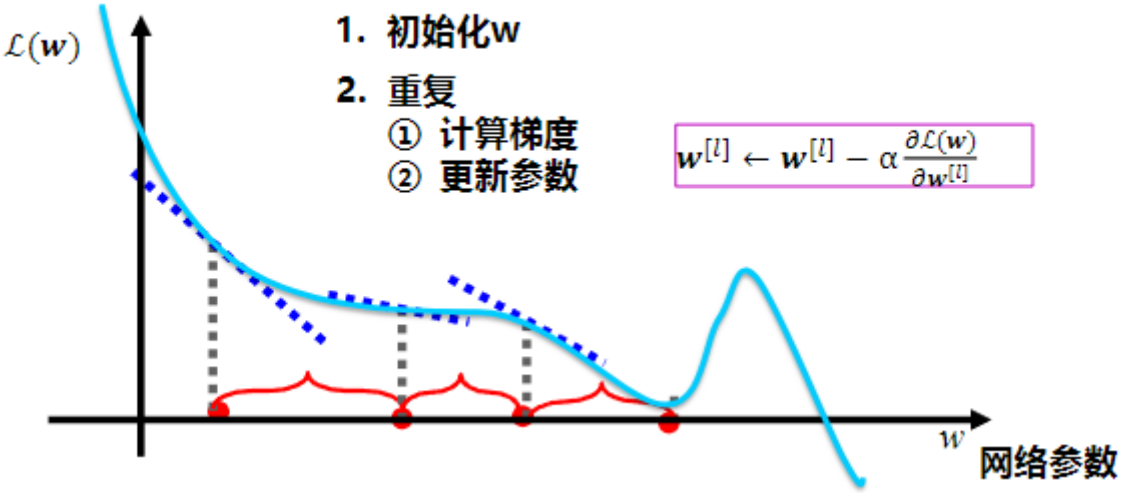
$$g\left(\boldsymbol{W}^{(1)} \times \boldsymbol{a}^{(1)} + \boldsymbol{b}^{(1)}\right) = \boldsymbol{a}^{(2)};$$
$$g\left(\boldsymbol{W}^{(2)} \times \boldsymbol{a}^{(2)} + \boldsymbol{b}^{(2)}\right) = \boldsymbol{z}$$

  其中 g 为激活函数，对于多层感知机，我们使用 **sigmoid** 函数作为激活函数

- **前向传播：**按顺序（从输入层到输出层）计算和存储神经网络中每层的结果，在于**计算**当前参数下的**误差**。

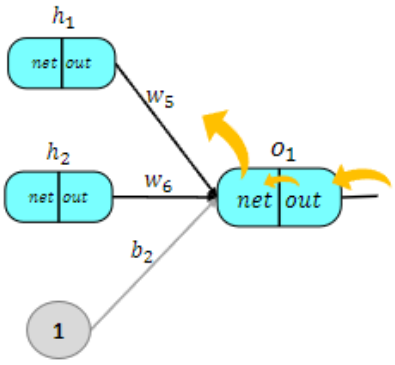


- **梯度下降法：**沿着梯度下降的方向更新参数，使得$L(\boldsymbol{w})$取值最小。



- **反向传播：**计算神经网络参数梯度的方法。根据微积分中的链式法则，按相反的顺序从输出层到输入层遍历网络，在于**根据误差调整参数**。反向传播算法不一次性计算所有参数的梯度，而是从后往前。首先计算输出层的梯度，然后是第二个参数矩阵的梯度，接着是中间层的梯度。计算结束后，所要的两个参数矩阵的梯度就都有了。

- 反向传播----考虑w_5，探究w_5的变化对总误差的影响有多大?



$$\frac{\partial E_{\text{total}}}{\partial w_5} = \frac{\partial E_{\text{total}}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$= -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * h_1$$

$$= 0.74136507*0.186815602*0.593269992$$

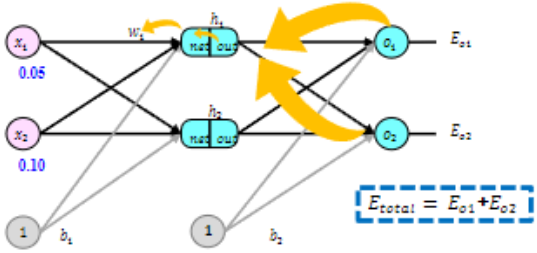
$$= 0.082167041$$

为了减少误差，设置学习率η = 0.5

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5}$$

$$= 0.4-0.5* 0.082167041=0.35891648$$

- 反向传播----继续反向传播w_1：



$$\frac{\partial E_{\text{total}}}{\partial w_1} = \frac{\partial E_{\text{total}}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{\text{total}}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

数据预处理

- **归一化：**数据归一化的目的是使得各特征对目标变量的影响一致，会将特征数据进行伸缩变化，所以数据归一化是会改变特征数据分布的。

$$x^* = \frac{x - x_{\min}}{x_{\max} - x_{\min}}$$

- **Z-Score：**数据标准化为了不同特征之间具备可比性，经过标准化变化之后的特征数据分布没有发生变化，本次实验中我才用的预处理方式为 Z-Score 算法。

$$x^* = \frac{x - \mu}{\sigma}$$

$$\sigma^2 = \frac{1}{m}\sum_{i=1}^{m}(x^{(i)} - \mu)^2$$

$$\mu = \frac{1}{m}\sum_{i=1}^{m}x^{(i)}$$

2. 伪代码

```
# --- Perceptron --- #
for i = 1 to epochs do:
    y_pred = w*x + b
    loss = 0.5*(y_pred - y)^2
    w = w - lr*(y_pred - y)*x
    b = b - lr*(y_pred - y)


# --- Multi-Layer Perceptron --- #
for i = 1 to epochs do:
    # forward pass
    hidden_layer_input = w1*x + b1
    hidden_layer_output = activation(hidden_layer_input)
    output_layer_input = w2*hidden_layer_output + b2
    output_layer_output = activation(output_layer_input)

    loss = 0.5*(output_layer_output - y)^2

    # backward pass
    [w2, b2] -= lr*gradient_output_layer
    [w1, b1] -= lr*gradient_hidden_layer
```

3. 关键代码展示

单层感知机

```
class Perceptron:

    def __init__(self, input_size, output_size):
        self.weights = np.random.randn(input_size, output_size)
        self.bias = np.random.randn(1, output_size)
        self.losses = []

    def fit(self, X, y, learning_rate=0.01, epochs=1000):
        for _ in range(epochs):
            y_pred = self.predict(X)
            self.losses.append(np.sum(0.5 * np.power(y_pred - y, 2)))
            gradient = np.dot(X.T, y_pred - y) / len(y)
            self.weights -= learning_rate * gradient
            self.bias -= learning_rate * np.sum(y_pred - y) / len(y)

    def predict(self, X):
        return np.where(np.dot(X, self.weights) + self.bias > 0, 1, 0)
```

多层感知机（两层神经网络）

```
class Multilayer_Perceptron:

    def __init__(self, input_size, hidden_size, output_size):
        self.weights1 = np.random.randn(input_size, hidden_size)
        self.weights2 = np.random.randn(hidden_size, output_size)
        self.bias1 = np.random.randn(1, hidden_size)
        self.bias2 = np.random.randn(1, output_size)
        self.losses = []

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

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def loss(self, y_true, y_pred):
        return 0.5 * np.power(y_pred - y_true, 2)

    def loss_derivative(self, y_true, y_pred):
        return y_pred - y_true
```

```
    def fit(self, X, y, learning_rate=0.01, epochs=1000):
        w1 = np.vstack((self.weights1, self.bias1))
        w2 = np.vstack((self.weights2, self.bias2))
        bias = np.ones((X.shape[0], 1))

        for _ in range(epochs):
            # forward pass
            hidden_layer_input = np.dot(np.column_stack((X, bias)), w1)
            hidden_layer_output = self.sigmoid(hidden_layer_input)

            output_layer_input =
np.dot(np.column_stack((hidden_layer_output, bias)), w2)
            output_layer_output = self.sigmoid(output_layer_input)

            self.losses.append(np.sum(self.loss(y,
output_layer_output)))

            # backward pass (output layer)
            # delta(E_total) /  delta(output_o)
            gradient_cal1 = self.loss_derivative(y, output_layer_output)
# data_size x output_layer_size
            # delta(output_o) / delta(net_o)
            gradient_cal2 = self.sigmoid_derivative(output_layer_output)
# data_size x output_layer_size
            # delta(net_o) / delta(w2)
            gradient_cal3 = np.column_stack((hidden_layer_output, bias))
# data_size x (hidden_layer_size + 1)
            # delta(E_total) / delta(w2)
            gradient_cal4 = np.dot(gradient_cal3.T, gradient_cal1 *
gradient_cal2) # (hidden_layer_size + 1) x output_layer_size
            # update w2
            weights2_update = gradient_cal4 * learning_rate
            w2 -= weights2_update

            # backward pass (hidden layer)
            # delta(E_total) / delta(output_h)
            gradient_cal5 = np.dot(gradient_cal1 * gradient_cal2, w2[:-
1].T) # data_size x hidden_layer_size
            # delta(output_h) / delta(net_h)
            gradient_cal6 = self.sigmoid_derivative(hidden_layer_output)
# data_size x hidden_layer_size
            # delta(net_h) / delta(w1)
```

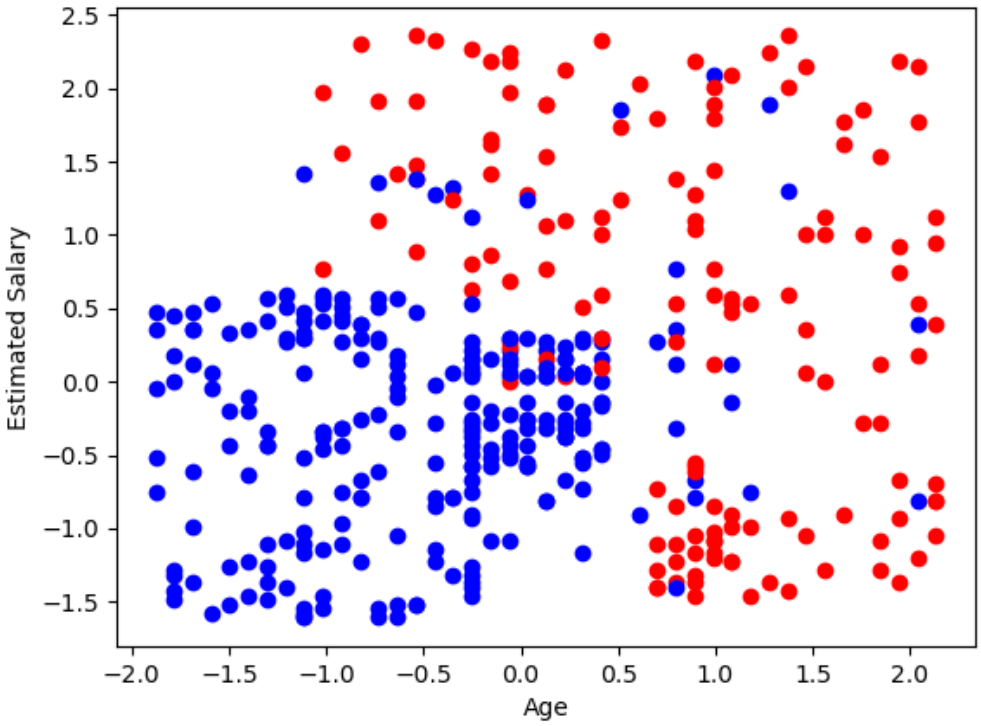
```
            gradient_cal7 = np.column_stack((X, bias)) # data_size x
(input_size + 1)
            # delta(E_total) / delta(w1)
            gradient_cal8 = np.dot(gradient_cal7.T, gradient_cal5 *
gradient_cal6) # (input_size + 1) x hidden_layer_size
            # update w1
            weights1_update = gradient_cal8 * learning_rate
            w1 -= weights1_update

        self.weights1 = w1[:-1]
        self.weights2 = w2[:-1]
        self.bias1 = w1[-1]
        self.bias2 = w2[-1]
```

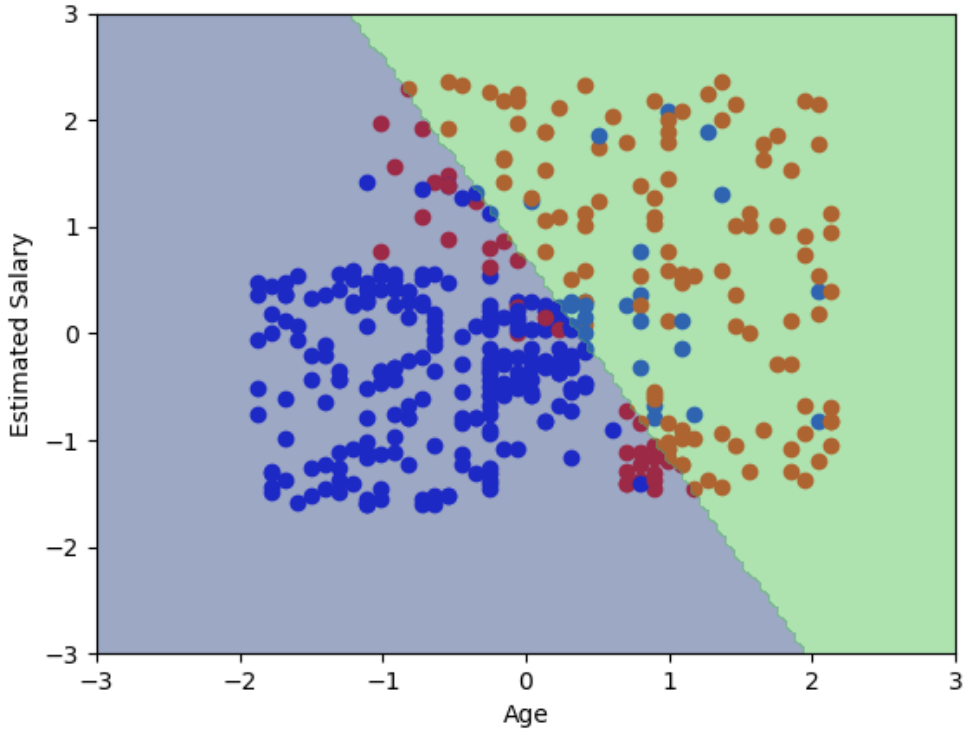
三、 实验结果及分析

1. 实验结果展示示例

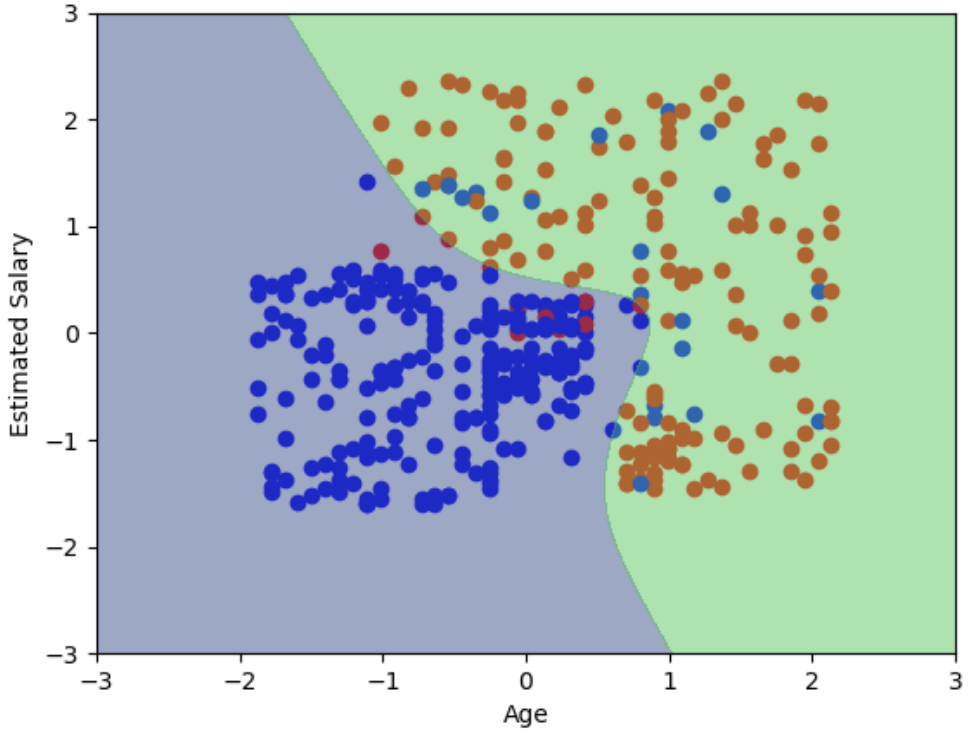
原数据分布



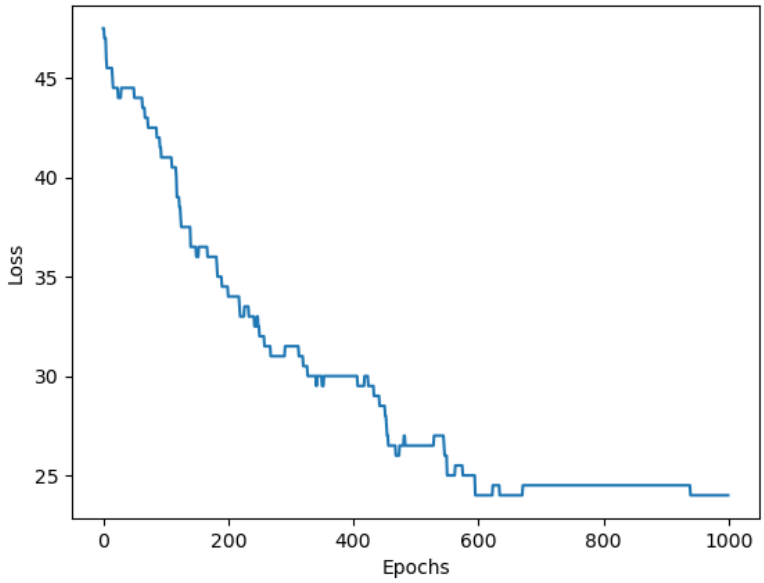
单层向量机分类结果



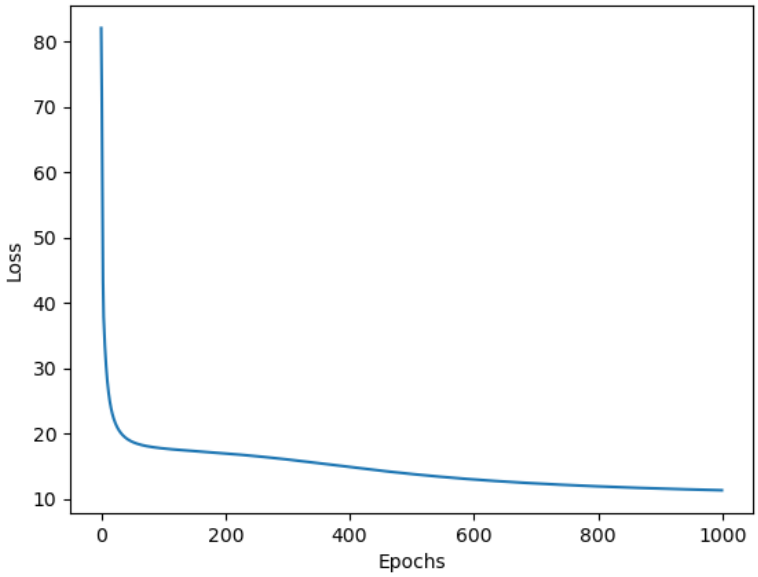
多层向量机分类结果



2. 评测指标展示及分析




```
Prediction: 0.00, Actual: 0.0
Prediction: 0.00, Actual: 1.0
Accuracy: 85.50%
```




```
Predicted: 0.00, Actual: [0.]
Predicted: 0.00, Actual: [0.]
Accuracy: 92.50%
```

如上所示，第一二张图分别为单层向量机的在迭代过程总每代对应的损失值与最终训练完后进行测试所得到的分类准确率；第三四张图则对应了多层向量机。

比较明显的是，两者的损失值在迭代的过程中总体都呈现了下降的趋势，由于单层向量机中使用了只有 0 和 1 两个函数值的 sgn 函数作为激活函数，因此其损失值的图像会显得更为曲折；而使用平滑的 sigmoid 函数作为激活函数的多层向量机，其损失值图像的走向也相应会显得相对平滑。

然后是对于本次的分类问题中，只能够处理线性分类问题的单层向量机最终的分类效果会低于能够处理非线性分类问题的多层向量机，这在上面的两者的分类图像中也可以看出。

四、 参考资料

- **课堂 ppt**