ML lab 1

How to make environment

在文件夹下运行

```
pip install -r requirements.txt
```

LR

- 结果展示
  - 0.875

	precision	recall	f1-score	support
0	0.86	0.91	0.88	105
1	0.90	0.83	0.86	95
accuracy			0.88	200
macro avg	0.88	0.87	0.87	200
weighted avg	0.88	0.88	0.87	200

  - 测试精度达到87.5%
- 实现思路
  1. 读取数据
  2. 初始化`weight`与`bias`为0相量
  3. 使用梯度下降来训练模型
     - ```
       for _ in range(self.n_iterations):
                   model = np.dot(X, self.weights) + self.bias
                   predictions = self.sigmoid(model)

                   # Compute gradients
                   dw = (1 / n_samples) * np.dot(X.T, (predictions - y))
                   db = (1 / n_samples) * np.sum(predictions - y)

                   # Update weights and bias
                   self.weights -= self.learning_rate * dw
                   self.bias -= self.learning_rate * db
       ```
  4. 用sigmoid来规划输出，并且判定精度
     - ```
       def predict(self, X):
               model = np.dot(X, self.weights) + self.bias
               predictions = self.sigmoid(model)
               return [1 if i > 0.5 else 0 for i in predictions]
       ```
- 超参数选择
  - 我选取了`learning rate`为0.001，`iteration`为5000

SVM

- 结果展示
  - 0.91

	precision	recall	f1-score	support
0	0.89	0.94	0.92	105
1	0.93	0.87	0.90	95
accuracy			0.91	200
macro avg	0.91	0.91	0.91	200
weighted avg	0.91	0.91	0.91	200

  - 测试精度达到91%
- 实现思路
  1. 读取数据
  2. 初始化`weight`与`bias`为0相量
  3. 使用梯度下降来训练模型
     - ```
       for _ in range(self.n_iterations):
                   for idx, xi in enumerate(X):
                       condition = y_[idx] * (np.dot(xi, self.weights) -
       self.bias) >= 1
                       if condition:
                           self.weights -= self.learning_rate * (2 *
       self.lambda_param * self.weights)
                       else:
                           self.weights -= self.learning_rate * (2 *
       self.lambda_param * self.weights - np.dot(xi, y_[idx]))
                           self.bias -= self.learning_rate * y_[idx]
       ```
  4. 输出结果并且判定精度
     - ```
       def predict(self, X):
               linear_output = np.dot(X, self.weights) - self.bias
               return np.sign(linear_output)
       ```
- 超参数选择
  - 我选取了`learning rate`为0.0001，`iteration`为10000，`lambda_param`（用于防止过拟合）为0.0005

MLP

- 结果展示
  -

```
0.91              precision    recall  f1-score   support

           0       0.91      0.92      0.92       105
           1       0.91      0.89      0.90        95

    accuracy                           0.91       200
   macro avg       0.91      0.91      0.91       200
weighted avg       0.91      0.91      0.91       200
```

  - 测试精度达到91%
  - 由于使用torch，其中存在随机相量，每次训练结果可能不同
- 实现思路
  - 使用pytorch形成mlp模型
- 超参数选择
  - 我选取了两层的`hidden layer`，第一层神经元为16，第二层神经元为8，由于是判断二元问题，输出神经元为1
  - 我选取了`learning rate`（使用adam优化器）为0.001，`iteration`为1100

Comparation among three models

From common perspective

1. **基本原理**：
   - **LR**：是一个线性模型，常用于二分类问题。它通过Sigmoid函数将线性函数的输出转换为0和1之间的概率。
   - **SVM**：是一个最大化分类边界的模型。对于线性可分的数据，SVM试图找到一个超平面来分隔两个类别，同时最大化该超平面与数据点之间的距离。
   - **MLP**：是一个前馈神经网络，由输入层、隐藏层和输出层组成。使用非线性激活函数，如ReLU，使其能够拟合非线性关系。
2. **优点**：
   - **LR**：计算效率高，输出是概率，易于解释。
   - **SVM**：在高维数据上表现良好，可以通过核函数处理非线性数据。
   - **MLP**：能够拟合非线性和复杂的模式，灵活性高。
3. **缺点**：
   - **LR**：不能很好地处理非线性数据，需要特征工程来增强模型的表现。
   - **SVM**：对于大数据集，计算效率低。选择合适的核函数和参数可能需要很多试验。
   - **MLP**：需要更多的数据来避免过拟合，对超参数的选择敏感（如层数、神经元数、学习率等）。
4. **计算复杂性**：
   - **LR**：相对较低。
   - **SVM**：对于大数据集，可能较高。
   - **MLP**：通常比LR和SVM高，尤其是当网络结构复杂时。

5. **可解释性**：
    - **LR**：高。权重与特征的重要性成正比。
    - **SVM**：中等。核SVM模型较难解释。
    - **MLP**：低。深度神经网络的解释性通常较差。

Analyzing with results

数据集特点：

- 特征数量：29
- 训练样本数量：400
- 二分类问题

模型测试结果：

1. **逻辑回归 (LR)**：
    - 准确率：87.5%
    - 对于非欺诈类（标签0），召回率：91%
    - 对于欺诈类（标签1），召回率：83%
2. **支持向量机 (SVM)**：
    - 准确率：91%
    - 对于非欺诈类（标签0），召回率：94%
    - 对于欺诈类（标签1），召回率：87%
3. **多层感知器 (MLP)**：
    - 准确率：91%
    - 对于非欺诈类（标签0），召回率：92%
    - 对于欺诈类（标签1），召回率：89%

结论：由于LR的精度已经比较高，所以当前数据近似为线性可分的（使用sklearn库的LR甚至有92%的准确率）。而SVM可以进一步提取信息，因此训练精度更高。MLP我只选取了两层，所以精度没有很多的提升。但如果选取过深的网络，可能会出现过拟合的情况，因此根据训练数据的线性特点，三者在数据集上的表现都很不错。