

《机器学习》课程学习报告（一）

学号: 16281012 姓名: 聂小禹 日期: 20181031

1 学习目的

简要描述实验需要达到的学习目标

理解线性模型，决策树，神经网络是怎样学习数据并进行数据预测。

线性模型

给定 d 个属性描述的实例 $x=(x_1;x_2;\dots;x_d)$, 其中 x_i 是 x 在第 i 个属性上的取值, 线性模型试图学得一个通过属性的线性组合来进行预测的函数。我们来主要学习如何确定函数是什么并且求得函数的相应参数。（斜率 w 和偏置 b ）

决策树

我们希望从给定训练数据集学得一个模型用以对新的示例进行分类, 这个把样本分类的任务, 可以看作是对“当前样本属于正类吗?”这个问题的决策或判定过程。顾名思义, 决策树是机遇树结构来进行决策的, 这恰巧是人类在面临决策问题时一种很自然的处理机制。一般来讲, 一棵决策树包含一个根节点、若干个内部节点和若干个叶节点, 每个叶节点对应的是决策结果, 其他每个节点则对应一个属性测试; 每个节点包含的样本集合根据属性测试的结果被划分到子节点中; 根节点包含样本全集, 从根节点到叶子节点的路径对应了一个判定测试序列, 决策树学习的目的是为了产生一棵泛化能力强的决策树, 我们需要学习如何建立一个决策树的基本算法, 如何递归决策树, 如何生成不同节点直至决策树彻底长成。同时更深入的了解剪枝是如何工作的。

神经网络

神经网络中最基本的成分时神经元模型, 即上述定义中的“简单单元”。在生物神经网络中, 每个神经元与其他神经元相连, 我们需要学习如何在机器层面学习模拟这种结构和工作方式, 如何像神经元一样, 受到兴奋刺激后, 就向相连的神经元发送化学物质, 从而改变这些神经元内的电位, 如果某神经元电位超过了阈值, 那么就被激活, 向其它神经元发生。我们要学习如何用权重代替阈值模拟神经元之间的传输过程, 将信号通过带权重的连接进行传递, 然后通过激活函数处理以产生对应神经元的输出。学习如何对权重进行调整进行正确的分类, 我们使用 bp 算法, 即将结果逆传播, 不断修改, 最终达到高效的结果。

2 学习内容

详细描述上机实践学习的内容

一、 线性模型

1. 线性回归

实验内容:

- 1) 利用最小二乘法解一元二次回归模型
- 2) 对房价 SalePrice 进行预测
- 3) 实现 MAE 和 RMSE 的编写并计算该线性模型的指标

2. 对数线性回归

实验内容:

- 1) 使用 sklearn 做一元对数线性回归的十折交叉验证
- 2) 用不同的属性组合作为特征进行预测
- 3) 进行指标计算

3. 对数几率回归

实验内容:

- 1) 使用对数几率回归完成垃圾邮件分类问题和 Dota2 结果预测问题。
- 2) 计算十折交叉验证下的精度(accuracy)，查准率(precision)，查全率(recall)，F1 值。

评测指标[I](#):

- 1) 精度
- 2) 查准率
- 3) 查全率
- 4) F1

4. 线性判别分析

实验内容

1. 使用线性判别分析完成垃圾邮件分类问题和 Dota2 结果预测问题。

- 计算十折交叉验证下的精度(accuracy), 查准率(precision), 查全率(recall), F1 值。

评测指标

- 精度
- 查准率
- 查全率
- F1

二、 决策树

1. 决策树处理分类任务

实验内容

- 使用 `sklearn.tree.DecisionTreeClassifier` 完成 `dota2` 比赛结果预测问题
- 计算最大深度为 10 时, 十折交叉验证的精度(accuracy), 查准率(precision), 查全率(recall), F1 值
- 绘制最大深度从 1 到 10 的决策树十折交叉验证精度的变化图

2. 决策树处理回归任务

实验内容

- 使用 `sklearn.tree.DecisionTreeRegressor` 完成 `kaggle` 房价预测问题
- 计算最大深度为 10 的决策树, 训练集上十折交叉验证的 MAE 和 RMSE
- 绘制最大深度从 1 到 30, 决策树在训练集和测试集上 MAE 的变化曲线
- 选择一个合理的树的最大深度, 并给出理由

3. 实现决策树

实验内容:

使用 LendingClub Safe Loans 数据集:

- 实现信息增益、信息增益率、基尼指数三种划分标准
- 使用给定的训练集完成三种决策树的训练过程
- 计算三种决策树在最大深度为 10 时在训练集和测试集上的精度, 查准率, 查全率, F1 值

在这部分, 我们会实现一个很简单的二叉决策树

4. 实现预剪枝

实验内容

1. 实现使用信息增益率划分的预剪枝
2. 计算出带有预剪枝和不带预剪枝的决策树的精度，查准率，查全率和 F1 值(最大深度为 6， 使用信息增益率)，保留 4 位小数，四舍五入

我们会以“信息增益率(information gain ratio)”作为划分准则，构造带有预剪枝的二叉决策树

使用的数据和第三题一样，剪枝需要使用验证集，所以数据划分策略会和第三题不同

三、神经网络

1. 使用 sklearn 感知机

实验内容：

- 1) 使用 `sklearn.neural_network.MLPClassifier` 完成手写数字分类任务
- 2) 绘制学习率为 3, 1, 0.1, 0.01 训练集损失函数的变化曲线

2. 神经网络：线性回归

实验内容：

- 1) 学会梯度下降的基本思想
- 2) 学会使用梯度下降求解线性回归
- 3) 了解归一化处理的作用

3. 神经网络：对数几率回归

实验内容：

- 1) 完成对数几率回归
- 2) 使用梯度下降求解模型参数
- 3) 绘制模型损失值的变化曲线

- 4) 调整学习率和迭代轮数，观察损失值曲线的变化
- 5) 按照给定的学习率和迭代轮数，初始化新的参数，绘制新模型在训练集和测试集上损失值的变化曲线，完成表格内精度的填写

4. 神经网络：三层感知机

实现内容：

- 1) 实现一个三层感知机
- 2) 对手写数字数据集进行分类
- 3) 绘制损失值变化曲线
- 4) 完成 kaggle MNIST 手写数字分类任务，根据给定的超参数训练模型，完成表格的填写。

3 学习过程

3.1 方案设计

简述上机实验的设计方案（可参考实验内容，简介总结）

(1) 线性模型

使用最小二乘法解线性回归模型，让导数为零得到 w 和 b 的闭式解，求出直线方程，自己编写 mylinearregression 类，调用 LinearRegression, LogisticRegression 类和交叉验证类训练模型。

(2) 决策树

利用 sklearn 中的 DecisionTreeClassifier 类进行分类任务的处理和模型的训练。
利用 sklearn 中的 DecisionTreeRegressor 类进行回归任务的处理和模型的训练。
使用 LendingClub Safe Loans 数据集，通过对于数学模型的抽象，实现三种划分：
信息增益、增益率和基尼指数，编写方法。编写创建决策树的结点方法，递归调用方法。决策树编写完成之后开始进行对于数据的预测，并完成指标的计算。
利用树类中已经设置的参数控制预剪枝

(3) 神经网络

模拟生物的神经元的波及方式，利用 sklearn 的多层感知机完成预测。
训练神经网络的基本思路：

1. 首先对参数进行初始化，对参数进行随机初始化（也就是取随机值）

2. 将样本输入神经网络，计算神经网络预测值 Z
3. 计算损失值 MSE
4. 通过 Z, y 以及 X 计算梯度
5. 使用梯度下降更新参数
6. 循环 1-5 步，在反复迭代的过程中可以看到损失值不断减小的现象，如果没有下降说明出了问题

3.2 实现过程

描述上机实验的实现过程（描述代码分几部分，各自使用什么 API，或根据什么公式编写实现什么功能）

(1) 线性模型

第一题 一元线性回归

1. 首先我们导入包和数据

1. 读取数据

```
import numpy as np
import math

import pandas as pd

# 读取数据
data = pd.read_csv('data/kaggle_house_price_prediction/kaggle_hourse_price_train.csv')

# 丢弃有缺失值的特征 (列)
data.dropna(axis = 1, inplace = True)

# 只保留整数的特征
data = data[[col for col in data.dtypes.index if data.dtypes[col] == 'int64']]
```

本题的要求是，针对LotArea, BsmtUnfSF, GarageArea三个特征，使用训练集训练三个一元线性回归模型，对比其在测试集上的MAE和RMSE，并且要绘制曲线。

所以我们只保留这三列与标记这一列，总共四列的数据即可

2. 为了选取训练集和测试集，我们要打乱数据的顺序
3. 选取训练集和测试集

2. 打乱数据顺序

```
from sklearn.utils import shuffle  
  
data_shuffled = shuffle(data, random_state = 32) # 这个32不要改变  
  
data_shuffled.head()  
  
LotArea BsmntUnfSF GarageArea SalePrice  
363 1680 321 264 118000  
529 32668 816 484 200624  
1394 4045 286 648 246578  
803 13891 1734 1020 58933  
832 9548 458 613 237000
```

3. 取前70%的数据为训练集，后30%为测试集

```
num_of_samples = data_shuffled.shape[0]  
split_line = int(num_of_samples * 0.7)  
train_data = data.iloc[:split_line]  
test_data = data.iloc[split_line:]  
  
train_data.shape  
(1021, 4)  
  
test_data.shape
```

我们可以调用.shape 查看列表的大小，查看数据的尺寸和规模

4. 开始编写线性模型

使均方误差最小化，求导为零寻找拐点。解得 w 和 b 的表达式分别为(6)(7)

本实验要求使用最小二乘法求解一元线性回归模型

求解w和b使均方误差 $E_{(w,b)} = \sum_{i=1}^m (y_i - wx_i - b)^2$ 最小化的过程，称为线性回归模型的最小二乘“参数估计”(parameter estimation)。我们可将 $E_{(w,b)}$ 分别对w和b求导，得到

$$\frac{\partial E_{(w,b)}}{\partial w} = 2(w \sum_{i=1}^m x_i^2 - \sum_{i=1}^m (y_i - b)x_i), \quad (4)$$

$$\frac{\partial E_{(w,b)}}{\partial b} = 2(mb - \sum_{i=1}^m (y_i - wx_i)) \quad (5)$$

然后令式(4)和式(5)为0，可得到w和b的闭式解(closed-form solution)

$$w = \frac{\sum_{i=1}^m y_i(x_i - \bar{x})}{\sum_{i=1}^m x_i^2 - \frac{1}{m}(\sum_{i=1}^m x_i)^2} \quad (6)$$

$$b = \frac{1}{m} \sum_{i=1}^m (y_i - wx_i) \quad (7)$$

其中， $\bar{x} = \frac{1}{m} \sum_{i=1}^m x_i$ 为x的均值

实现求解 w 的方法

```

def get_w(x, y):
    """
    这个函数是计算模型w的值的函数，  

    传入的参数分别是x和y，表示数据与标记

    Parameter
    -----
        x: np.ndarray, pd.Series, 传入的特征数据
        y: np.ndarray, pd.Series, 对应的标记

    Returns
    -----
        w: float, 模型w的值
    ...

    # m表示样本的数量
    m = len(x)

    # 求x的均值
    x_mean = sum(x)/m # YOUR CODE HERE

    # 求w的分子部分
    numerator = sum(y*(x-x_mean)) # YOUR CODE HERE

    # 求w的分母部分
    denominator = sum(x**2)-(sum(x)**2/m) # YOUR CODE HERE

    # 求w
    w = numerator/denominator # YOUR CODE HERE

    # 返回w
    return w

```

再实现求解 b 的方法

```

def get_b(x, y, w):
    """
    这个函数是计算模型b的值的函数，  

    传入的参数分别是x, y, w，表示数据，标记以及模型的w值

    Parameter
    -----
        x: np.ndarray, pd.Series, 传入的特征数据
        y: np.ndarray, pd.Series, 对应的标记
        w: np.ndarray, pd.Series, 模型w的值

    Returns
    -----
        b: float, 模型b的值
    ...

    # 样本个数
    m = len(x)

    # 求b
    b = (sum(y-w*x))/m # YOUR CODE HERE

    # 返回b
    return b

```

完成类: mylinearregression 自定义的线性回归

```

class myLinearRegression:
    def __init__(self):
        """
        类的初始化方法，不需要初始化的参数
        这里设置了两个成员变量，用来存储模型w和b的值
        ...
        self.w = None
        self.b = None

    def fit(self, x, y):
        """
        这里需要编写训练的函数，也就是调用模型的fit方法，传入特征x的数据和标记y的数据
        这个方法就可以求解出w和b
        ...
        self.w = get_w(x, y)
        self.b = get_b(x, y, self.w)

    def predict(self, x):
        """
        这是预测的函数，传入特征的数据，返回模型预测的结果
        ...
        if self.w == None or self.b == None:
            print("模型还未训练，请先调用fit方法训练")
            return

        return self.w * x + self.b

```

在准备工作做好后我们开始用数据集进行模型的训练和预测

创建变量类为 model1，实现我们刚刚自定义的类，用 LotArea 和 SalePrice 作为训练的属性。prediction1 中存的是利用了自定义的线性回归类中的方法实现的基于 lotarea 对于 saleprice 的预测。

```
# 创建一个模型的实例
model1 = myLinearRegression()

# 使用训练集对模型进行训练，传入训练集的LotArea和标记SalePrice
model1.fit(train_data['LotArea'], train_data['SalePrice'])

# 对测试集进行预测，并将结果存储在变量prediction中
prediction1 = model1.predict(test_data['LotArea'])
```

指标计算这里使用的是 MAE 和 RMSE

计算公式和方法实现如下

6. 性能度量

模型训练完成后，还需要在测试集上验证其预测能力，这就需要计算模型的一些性能指标，如MAE和RMSE等。

$$MAE(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m |\hat{y}_i - y_i| \quad (8)$$

$$RMSE(\hat{y}, y) = \sqrt{\frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2} \quad (9)$$

其中， \hat{y} 是模型的预测值， y 是真值， m 是样本数

```
def MAE(y_hat, y):
    # 请你完成MAE的计算过程
    # YOUR CODE HERE
    MAE=(sum(abs(y_hat-y)))/len(y)
    return MAE

def RMSE(y_hat, y):
    # 请你完成RMSE的计算过程
    # YOUR CODE HERE
    RMSE=((sum((y_hat-y)**2))/len(y))**0.5
    return RMSE
```

利用 matplotlib 对预测结果进行可视化

7. 模型预测效果可视化

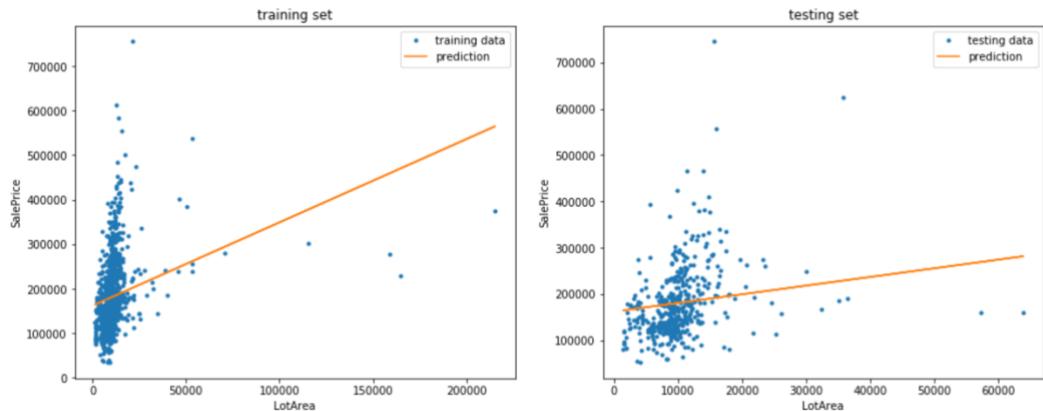
```
import matplotlib.pyplot as plt
%matplotlib inline

plt.figure(figsize = (16, 6))

plt.subplot(121)
plt.plot(train_data['LotArea'].values, train_data['SalePrice'].values, '.', label = 'training data')
plt.plot(train_data['LotArea'].values, model1.predict(train_data['LotArea']), '-', label = 'prediction')
plt.xlabel('LotArea')
plt.ylabel('SalePrice')
plt.title("training set")
plt.legend()

plt.subplot(122)
plt.plot(test_data['LotArea'].values, test_data['SalePrice'].values, '.', label = 'testing data')
plt.plot(test_data['LotArea'].values, prediction1, '-', label = 'prediction')
plt.xlabel('LotArea')
plt.ylabel('SalePrice')
plt.title("testing set")
plt.legend()
```

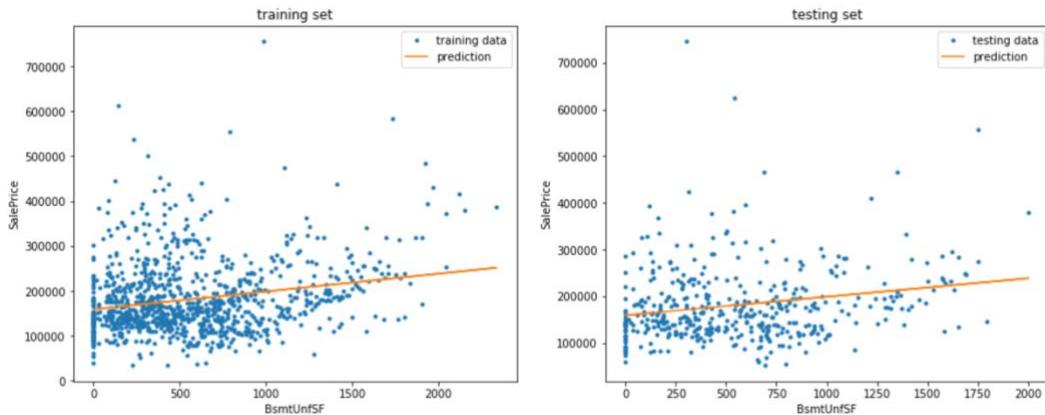
观察两个图训练集和测试集的结果。



其中蓝色的点是样本集，而橙色的线是预测的线性模型。

换用其它属性作为特征完成训练和预测

下面是用 BsmtUnfSF, GarageArea 作为特征的可视化结果。



第二题 对数线性回归

本道题使用十折交叉验证完成对数线性回归

- 导入包读取数据并对数据进行清洁和处理

1. 读取数据

```
import numpy as np
import pandas as pd

# 读取数据
data = pd.read_csv('data/kaggle_house_price_prediction/kaggle_hourse_price_train.csv')

# 丢弃有缺失值的特征 (列)
data.dropna(axis = 1, inplace = True)

# 只保留整数的特征
data = data[[col for col in data.dtypes.index if data.dtypes[col] == 'int64']]
```

- 引入本次要使用的模型

2. 引入模型

```
from sklearn.linear_model import LinearRegression
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error
from sklearn.model_selection import cross_val_predict
```

分别为线性回归模型， MAE， RMSE 和交叉验证的模型

- 注意此处要去的是 e 的指数（预测值），如果直接输出数据值应该是 $\ln y$

3. 使用sklearn完成一元对数线性回归的十折交叉验证验证

创建模型

```
model = LinearRegression()
```

选取数据

因为我们要做对数线性回归，所以需要先将标记y取对数

```
features = ['LotArea']
x = data[features]
y = np.log(data['SalePrice'])
```

做十折交叉验证的预测

注意，我们的模型是对数线性回归

$$\ln y = WX + b \quad (1)$$

这个模型要求， y 的值一定是大于0的，所以这个模型是有它的适用范围的。我们这个任务是房价预测，房价通常来说是大于等于0的，所以在使用这个模型的时候，一定要确定 $y > 0$ 。

我们在使用sklearn进行实现的时候，会将 y 先取对数，然后使用普通的一元线性回归对 $\ln y$ 进行拟合。

需要注意的是，当我们的模型训练完成后，模型的预测值应该做 e 的指数运算，才能输出模型预测出的房价，否则输出值就是" \ln 房价"

```
prediction = np.exp(cross_val_predict(model, x, y, cv = 10))
```

- 计算指标

4. 计算评价指标

MAE

```
mean_absolute_error(prediction, data['SalePrice'])
```

57297.333304518805

RMSE

```
mean_squared_error(prediction, data['SalePrice']) ** 0.5
```

132257.6716524966

- 分别采用不同的属性组合作为不同的特征，我们在此使用的是

```
features = ['LotArea', 'BsmtUnfSF']
```

```
features = ['GarageArea', 'BsmtUnfSF']
```

```
features = ['LotArea', 'GarageArea']
```

三种组合

下面是训练、预测和指标计算的代码

```

features = [ 'LotArea', 'BsmtUnfSF' ]
x = data[features]
y = np.log(data[ 'SalePrice' ])
prediction = np.exp(cross_val_predict(model, x, y, cv = 10))
prediction.shape
print("NO.1")
print(mean_absolute_error(prediction, data[ 'SalePrice' ]))
print(mean_squared_error(prediction, data[ 'SalePrice' ]) ** 0.5)
features = [ 'GarageArea', 'BsmtUnfSF' ]
x = data[features]
y = np.log(data[ 'SalePrice' ])
prediction = np.exp(cross_val_predict(model, x, y, cv = 10))
prediction.shape
print("NO.2")
print(mean_absolute_error(prediction, data[ 'SalePrice' ]))
print(mean_squared_error(prediction, data[ 'SalePrice' ]) ** 0.5)
features = [ 'LotArea', 'GarageArea' ]
x = data[features]
y = np.log(data[ 'SalePrice' ])
prediction = np.exp(cross_val_predict(model, x, y, cv = 10))
prediction.shape
print("NO.3")
print(mean_absolute_error(prediction, data[ 'SalePrice' ]))
print(mean_squared_error(prediction, data[ 'SalePrice' ]) ** 0.5)

```

结果如下图

1. 模型1使用的特征:LotArea, BsmtUnfSF
2. 模型2使用的特征:GarageArea, BsmtUnfSF
3. 模型3使用的特征:LotArea, GarageArea

	模型	MAE	RMSE
模型1	54141.76	115973.59	
模型2	41128.77	63203.99	
模型3	41405.38	65860.18	

第三题 对数几率回归

实验内容

1. 使用对数几率回归完成垃圾邮件分类问题和 Dota2 结果预测问题。
2. 计算十折交叉验证下的精度(accuracy), 查准率(precision), 查全率(recall), F1 值。

评测指标

1. 精度

2. 查准率
3. 查全率
4. F1

这次将数据集换成 dota2 的比赛结果，用 logisticRegression

1. 导入包和读取数据和之前没有差别
引入下列的模型

2. 导入模型

```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.model_selection import cross_val_predict
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

3. 提取数据

这里的spamx和dota2x包含了数据集内所有的特征

```
spamx = spambase[:, :57]
spamy = spambase[:, 57]

dota2x = dota2results[:, 1:]
dota2y = dota2results[:, 0]
```

2. 训练模型

4. 训练并预测

请你完成两个模型使用全部特征的训练与预测，并将预测结果存储起来

```
# YOUR CODE HERE
#-----#
# TRAINING SPAM DATA
spamx = StandardScaler().fit_transform(spamx)
spamx_train, spamx_test, spamy_train, spamy_test = train_test_split(spamx, spamy, test_size=.4, random_state=42)

slr = LogisticRegression()
slr.fit(spamx_train, spamy_train)

prediction1 = slr.predict(spamx_test)
#-----#
# TRAINING DOTA DATA
dota2x = StandardScaler().fit_transform(dota2x)
dota2x_train, dota2x_test, dota2y_train, dota2y_test = train_test_split(dota2x, dota2y, test_size=.4, random_state=42)
dlr = LogisticRegression()
dlr.fit(dota2x_train, dota2y_train)
prediction2 = dlr.predict(dota2x_test)
```

3. 计算相应指标

此处要求的指标有精度，查准率，查全率和 f1 值

5. 评价指标的计算

请你计算两个模型的四项指标

```
# YOUR CODE HERE

acc = accuracy_score(prediction1, spamy_test)
precision = precision_score(prediction1, spamy_test)
recall = recall_score(prediction1, spamy_test)
f1 = f1_score(prediction1, spamy_test)

print('对数几率回归在该Spamx,Spamy测试集上的四项指标：')
print('精度:', acc)
print('查准率:', precision)
print('查全率:', recall)
print('f1值:', f1)

print('\n')
acc_2= accuracy_score(prediction2, dota2y_test)
precision_2 = precision_score(prediction2, dota2y_test)
recall_2 = recall_score(prediction2, dota2y_test)
f1_2 = f1_score(prediction2, dota2y_test)

print('对数几率回归在该dota2x,dota2y测试集上的四项指标：')
print('精度:', acc_2)
print('查准率:', precision_2)
print('查全率:', recall_2)
print('f1值:', f1_2)
```

对数几率回归在该Spamx,Spamy测试集上的四项指标：

精度：0.9217816404128191
查准率：0.8789893617021277
查全率：0.9257703081232493
f1值：0.9017735334242838

对数几率回归在该dota2x,dota2y测试集上的四项指标：

精度：0.5943335132218025
查准率：0.6680155721749821
查全率：0.6039177549319256
f1值：0.6343515906216559

双击此处填写

数据集	精度	查准率	查全率	F1
spambase	0.92	0.88	0.93	0.90
dota2Results	0.59	0.67	0.60	0.63

第四题 线性判别分析

实验内容

1. 使用线性判别分析完成垃圾邮件分类问题和 Dota2 结果预测问题。
2. 计算十折交叉验证下的精度(accuracy), 查准率(precision), 查全率(recall), F1 值。

评测指标

1. 精度
2. 查准率
3. 查全率
4. F1

导入模型如下

2. 导入模型

```
from sklearn.discriminant_analysis import LinearDiscriminantAnalysis
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.model_selection import cross_val_predict
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
```

其中第一行 linear discriminant analysis 就是线性判别分析的相应包
下面开始对数据处理并训练

3. 提取数据

这里的spamx和dota2x包含了数据集内所有的特征

```
spamx = spambase[:, :57]
spamy = spambase[:, 57]

dota2x = dota2results[:, 1:]
dota2y = dota2results[:, 0]
```

4. 训练

请你完成两个模型使用全部特征的训练与预测，并将预测结果存储起来

注意：dota2数据集上，线性判别分析模型在训练的过程中会有警告出现，不会影响程序运行

```
# YOUR CODE HERE

#-----#
# TRAINING SPAM DATA
spamx = StandardScaler().fit_transform(spamx)
spamx_train, spamx_test, spamy_train, spamy_test = train_test_split(spamx, spamy, test_size=.4, random_state=42)

sLDA = LinearDiscriminantAnalysis()
sLDA.fit(spamx_train, spamy_train)

prediction1 = sLDA.predict(spamx_test)
#-----#
# TRAINING DOTA DATA
dota2x = StandardScaler().fit_transform(dota2x)
dota2x_train, dota2x_test, dota2y_train, dota2y_test = train_test_split(dota2x, dota2y, test_size=.4, random_state=42)
dLDA = LinearDiscriminantAnalysis()
dLDA.fit(dota2x_train, dota2y_train)
prediction2 = dLDA.predict(dota2x_test)
```

接下来计算指标

5. 评价指标的计算

请你计算两个模型的四项指标

```
# YOUR CODE HERE

acc = accuracy_score(prediction1, spamy_test)
precision = precision_score(prediction1, spamy_test)
recall = recall_score(prediction1, spamy_test)
f1 = f1_score(prediction1, spamy_test)

print('线性判别分析在该Spamx,Spamy测试集上的四项指标：')
print('精度:', acc)
print('查准率:', precision)
print('查全率:', recall)
print('f1值:', f1)

print('\n')
acc_2= accuracy_score(prediction2, dota2y_test)
precision_2 = precision_score(prediction2, dota2y_test)
recall_2 = recall_score(prediction2, dota2y_test)
f1_2 = f1_score(prediction2, dota2y_test)

print('线性判别分析在该dota2x,dota2y测试集上的四项指标：')
print('精度:', acc_2)
print('查准率:', precision_2)
print('查全率:', recall_2)
print('f1值:', f1_2)
```

结果如下

线性判别分析在该Spamx,Spamy测试集上的四项指标：

精度： 0.8913633894622488
查准率： 0.8071808510638298
查全率： 0.9169184290030211
f1值： 0.8585572842998586

线性判别分析在该dota2x,dota2y测试集上的四项指标：

精度： 0.594657312466271
查准率： 0.6682204692142198
查全率： 0.6042149143121816
f1值： 0.6346079003697217

双击此处填写

数据集	精度	查准率	查全率	F1
spambase	0.89	0.81	0.92	0.86
dota2Results	0.59	0.67	0.60	0.63

(2) 决策树

第一题：决策树处理分类任务

1. 使用 sklearn.tree.DecisionTreeClassifier 完成 dota2 比赛结果预测问题
2. 计算最大深度为 10 时，十折交叉验证的精度(accuracy)，查准率(precision)，查全率(recall)，F1 值
3. 绘制最大深度从 1 到 10 的决策树十折交叉验证精度的变化图

1. 读取数据

```
import numpy as np
dota2results = np.loadtxt('data/dota2Dataset/dota2Train.csv', delimiter=',')
dota2x = dota2results[:, 1:]
dota2y = dota2results[:, 0]
```

2. 导入模型

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.tree import DecisionTreeClassifier
```

3. 训练与预测

计算最大深度为10的决策树，在使用数据dota2x，标记dota2y下，十折交叉验证的精度，查准率，查全率和F1值

```
model = DecisionTreeClassifier(max_depth = 10) # 参数max_depth决定了决策树的最大深度
# YOUR CODE HERE
prediction=cross_val_predict(model, dota2x,dota2y, cv = 10)
prediction.shape
print(accuracy_score(dota2y,prediction))
print(precision_score(dota2y,prediction))
print(recall_score(dota2y,prediction))
print(f1_score(dota2y,prediction))

0.550566486778198
0.5557167821256943
0.730125866098151
0.6310930772365646
```

双击此处填写下面的表格

最大深度为10：

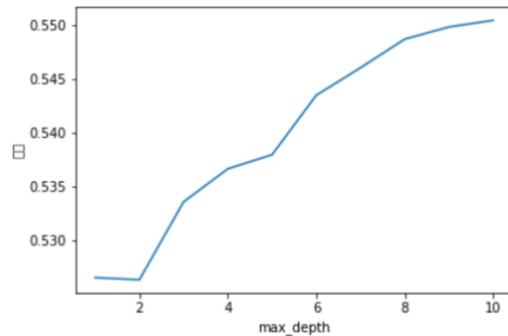
精度	查准率	查全率	F1
0.550566486778198	0.5555694266720764	0.7298183756303555	0.6308831868443436

4. 改变最大深度，绘制决策树的精度变换图

绘制最大深度从1到10，决策树十折交叉验证精度的变化图

```
import matplotlib.pyplot as plt
%matplotlib inline

# YOUR CODE HERE
y=[]
for i in range(1,11):
    model = DecisionTreeClassifier(max_depth = i)
    dotaprediction=cross_val_predict(model, dota2x,dota2y, cv = 10)
    y.append(accuracy_score(dota2y,dotaprediction))
x=range(1,11)
plt.plot(x,y)
plt.xlabel('max_depth')
plt.ylabel('精度')
plt.show()
```



第二题：决策树处理回归任务

实验内容

1. 使用 `sklearn.tree.DecisionTreeRegressor` 完成 kaggle 房价预测问题
2. 计算最大深度为 10 的决策树，训练集上十折交叉验证的 MAE 和 RMSE
3. 绘制最大深度从 1 到 30，决策树在训练集和测试集上 MAE 的变化曲线
4. 选择一个合理的树的最大深度，并给出理由

1. 读取数据

```
import pandas as pd
data = pd.read_csv('data/kaggle_house_price_prediction/kaggle_hourse_price_train.csv')

# 去弃有缺失值的特征 (列)
data.dropna(axis = 1, inplace = True)

# 只保留整数的特征
data = data[[col for col in data.dtypes.index if data.dtypes[col] == 'int64']]
```

```
data.head()
```

2. 数据集划分

70%做训练集，30%做测试集

```
from sklearn.utils import shuffle

data_shuffled = shuffle(data, random_state = 32)
split_line = int(len(data_shuffled) * 0.7)
training_data = data_shuffled[:split_line]
testing_data = data_shuffled[split_line:]
```

3. 导入模型

```
from sklearn.model_selection import cross_val_predict
from sklearn.metrics import mean_absolute_error
from sklearn.metrics import mean_squared_error

from sklearn.tree import DecisionTreeRegressor
```

4. 选取特征和标记

```
features = data.columns.tolist()
target = 'SalePrice'
features.remove(target)
```

5. 训练与预测

请你在下面计算树的最大深度为10时，使用训练集全量特征训练的决策树的十折交叉验证的MAE和RMSE

```
# YOUR CODE HERE
model=DecisionTreeRegressor(max_depth=10)
prediction = cross_val_predict(model, training_data[features], training_data['SalePrice'], cv = 10)

print(mean_absolute_error(prediction, training_data['SalePrice']))
print(mean_squared_error(prediction, training_data['SalePrice']) ** 0.5)

27237.086365195002
45544.34235312367
```

cv 此处指的是折数

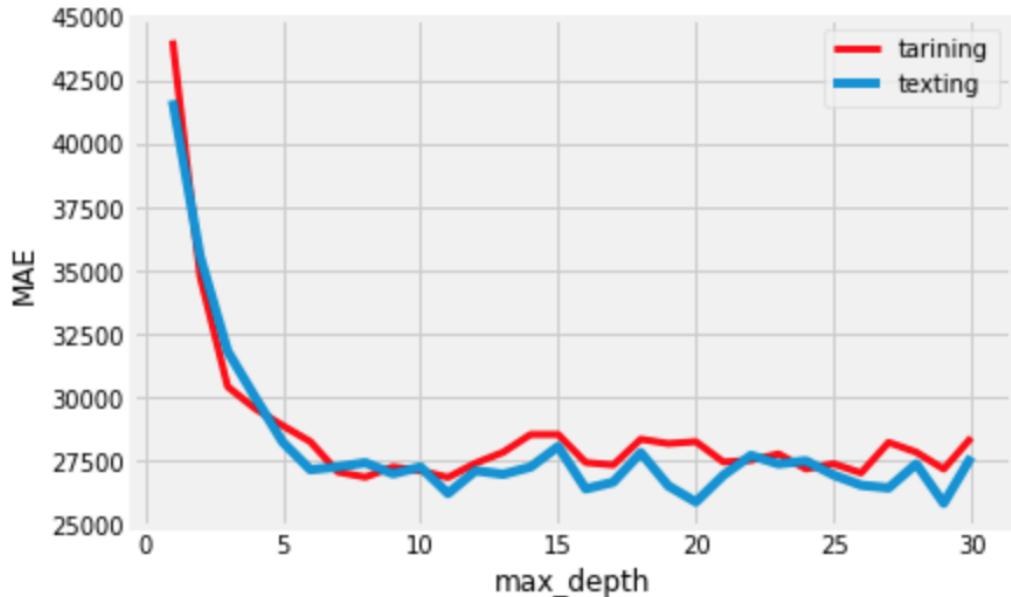
6. 改变最大深度，绘制决策树的精度变换图

绘制最大深度从1到30，决策树训练集和测试集MAE的变化图

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.style.use("fivethirtyeight")

# YOUR CODE HERE
trainy=[]
testy=[]
x1=range(1,31)
x2=range(1,31)
for i in range(1,31):
    model=DecisionTreeRegressor(max_depth=i)
    prediction = cross_val_predict(model, training_data[features], training_data['SalePrice'], cv = 10)
    trainy.append(mean_absolute_error(prediction, training_data['SalePrice']))
    prediction = cross_val_predict(model, testing_data[features], testing_data['SalePrice'], cv = 10)
    testy.append(mean_absolute_error(prediction, testing_data['SalePrice']))
plt.plot(x1,trainy,label='tarining',linewidth=3,color='r')
plt.plot(x2,testy,label='texting')
plt.xlabel('max_depth')
plt.ylabel('MAE')
plt.legend()
plt.show()
```

MAE	RMSE
27237.086365195002	45544.34235312367



第三题：实现决策树

实验内容：

使用 LendingClub Safe Loans 数据集：

- 实现信息增益、信息增益率、基尼指数三种划分标准
- 使用给定的训练集完成三种决策树的训练过程
- 计算三种决策树在最大深度为 10 时在训练集和测试集上的精度，查准率，查全率，F1 值

在这部分，我们会实现一个很简单的二叉决策树

1. 读取数据

```
# 导入类库
import pandas as pd
import numpy as np
import json

# 导入数据
loans = pd.read_csv('data/lendingclub/lending-club-data.csv', low_memory=False)
```

数据中有两列是我们想预测的指标，一项是safe_loans，一项是bad_loans，分别表示正例和负例，我们对其进行处理，将正例的safe_loans设为1，负例设为-1，删除bad_loans这列

```
# 对数据进行预处理，将safe_loans作为标记
loans['safe_loans'] = loans['bad_loans'].apply(lambda x : +1 if x==0 else -1)
del loans['bad_loans']
```

我们只使用grade, term, home_ownership, emp_length这四列作为特征，safe_loans作为标记，只保留loans中的这五列

```
features = ['grade',           # grade of the loan
            'term',             # the term of the loan
            'home_ownership',   # home_ownership status: own, mortgage or rent
            'emp_length',       # number of years of employment
            ]
target = 'safe_loans'
loans = loans[features + [target]]
```

导入数据，保留 grade, term, homeownership, emolength 为特征
safe loans 当作是 label，对 safe_loans 做正负 1 处理

查看前五行数据

	grade	term	home_ownership	emp_length	safe_loans
0	B	36 months	RENT	10+ years	1
1	C	60 months	RENT	< 1 year	-1
2	C	36 months	RENT	10+ years	1
3	C	36 months	RENT	10+ years	1
4	A	36 months	RENT	3 years	1

2. 划分训练集和测试集

```
from sklearn.utils import shuffle
loans = shuffle(loans, random_state = 34)

split_line = int(len(loans) * 0.6)
train_data = loans.iloc[: split_line]
test_data = loans.iloc[split_line:]
```

抽取 60% 作为训练集，剩下的作为测试集

3. 特征预处理

可以看到所有的特征都是离散类型的特征，需要对数据进行预处理，使用one-hot编码对其进行处理。

one-hot编码的思想就是将离散特征变成向量，假设特征A有三种取值 $\{a, b, c\}$ ，这三种取值等价，如果我们使用1,2,3三个数字表示这三种取值，那么在计算时就会产生偏差，有一些涉及距离度量的算法会认为，2和1离得近，3和1离得远，但这三个值应该是等价的，这种表示方法会造成模型在判断上出现偏差。解决方案就是使用一个三维向量表示他们，用 $[1, 0, 0]$ 表示a， $[0, 1, 0]$ 表示b， $[0, 0, 1]$ 表示c，这样三个向量之间的距离就都是相等的了，任意两个向量在欧式空间的距离都是 $\sqrt{2}$ 。这就是one-hot编码的思想。

pandas中使用get_dummies生成one-hot向量

```
def one_hot_encoding(data, features_categorical):
    ...
    Parameter
    -----
    data: pd.DataFrame
    features_categorical: list(str)
    ...

    # 对所有的离散特征遍历
    for cat in features_categorical:

        # 对这列进行one-hot编码，前缀为这个变量名
        one_encoding = pd.get_dummies(data[cat], prefix = cat)

        # 将生成的one-hot编码与之前的dataframe拼接起来
        data = pd.concat([data, one_encoding], axis=1)

        # 删除掉原始的这列离散特征
        del data[cat]

    return data
```

首先对训练集生成one-hot向量，然后对测试集生成one-hot向量，这里需要注意的是，如果训练集中，特征A的取值为 $\{a, b, c\}$ ，这样我们生成的特征就有三列，分别为 A_a, A_b, A_c ，然后我们使用这个训练集训练模型，模型就就会考虑这三个特征，在测试集中如果有一个样本的特征A的值为d，那它的 A_a, A_b, A_c 就都为0，我们不去考虑 A_d ，因为这个特征在训练模型的时候是不存在的。

接下来是对测试集进行one_hot编码，但只要保留出现在one_hot_features中的特征即可。

```
test_data_tmp = one_hot_encoding(test_data, features)

# 创建一个空的DataFrame
test_data = pd.DataFrame(columns = train_data.columns)
for feature in train_data.columns:
    # 如果训练集中当前特征在test_data_tmp中出现了，将其复制到test_data中
    if feature in test_data_tmp.columns:
        test_data[feature] = test_data_tmp[feature].copy()
    else:
        # 否则就用全为0的列去替代
        test_data[feature] = np.zeros(test_data_tmp.shape[0], dtype = 'uint8')
```

```
test_data.head()
```

	safe_loans	grade_A	grade_B	grade_C	grade_D	grade_E	grade_F	grade_G	term_36 months	term_60 months	...	emp_length_10+ years	emp_length_2 years	emp_length_3 years	emp
37225	1	0	0	0	0	1	0	0	1	0	...	0	1	0	0
101585	-1	0	1	0	0	0	0	0	1	0	...	0	0	0	0
31865	1	1	0	0	0	0	0	0	1	0	...	0	0	0	0
97692	1	0	0	1	0	0	0	0	1	0	...	1	0	0	0
88181	1	1	0	0	0	0	0	0	1	0	...	0	0	0	0

5 rows × 25 columns

```
train_data.shape
```

```
(73564, 25)
```

4. 实现3种特征划分准则

决策树中有很多常用的特征划分方法，比如信息增益、信息增益率、基尼指数

我们需要实现一个函数，它的作用是，给定决策树的某个结点内的所有样本的标记，让它计算出对应划分指标的值是多少

接下来我们会实现上述三种划分指标

这里我们约定，将所有特征取值为0的样本，划分到左子树，特征取值为1的样本，划分到右子树

4.1 信息增益

信息熵：

$$\text{Ent}(D) = - \sum_{k=1}^{|Y|} p_k \log_2 p_k$$

信息增益：

$$\text{Gain}(D, a) = \text{Ent}(D) - \sum_{v=1}^V \frac{|D^v|}{|D|} \text{Ent}(D^v)$$

计算信息熵时约定：若 $p = 0$ ，则 $p \log_2 p = 0$

求当前结点的信息熵

```
Parameter
-----
labels_in_node: np.ndarray, 如[-1, 1, -1, 1, 1]

Returns
-----
float: information entropy

def information_entropy(labels_in_node):
    """
    求当前结点的信息熵

    Parameter
    -----
    labels_in_node: np.ndarray, 如[-1, 1, -1, 1, 1]

    Returns
    -----
    float: information entropy
    """

    # 统计样本总个数
    num_of_samples = labels_in_node.shape[0]

    if num_of_samples == 0:
        return 0

    # 统计出标记为1的个数
    num_of_positive = len(labels_in_node[labels_in_node == 1])

    # 统计出标记为-1的个数
    num_of_negative = len(labels_in_node[labels_in_node == -1])

    # 统计正例的概率
    prob_positive = num_of_positive / num_of_samples

    # 统计负例的概率
    prob_negative = num_of_negative / num_of_samples

    if prob_positive == 0:
        positive_part = 0
    else:
        positive_part = prob_positive * np.log2(prob_positive)

    if prob_negative == 0:
        negative_part = 0
    else:
        negative_part = prob_negative * np.log2(prob_negative)

    return -(positive_part + negative_part)
```

计算所有特征的信息增益

Parameter

data: pd.DataFrame, 传入的样本, 带有特征和标记的 dataframe

features: list(str), 特征名组成的 list

target: str, 标记(label)的名字

annotate, boolean, 是否打印所有特征的信息增益值, 默认为 False

Returns

information_gains: dict, key: str, 特征名
value: float, 信息增益

```
def compute_information_gains(data, features, target, annotate = False):  
    ...  
    计算所有特征的信息增益  
  
    Parameter  
    -----  
        data: pd.DataFrame, 传入的样本, 带有特征和标记的 dataframe  
        features: list(str), 特征名组成的 list  
        target: str, 标记(label)的名字  
        annotate, boolean, 是否打印所有特征的信息增益值, 默认为 False  
  
    Returns  
    -----  
        information_gains: dict, key: str, 特征名  
        value: float, 信息增益  
    ...  
  
    # 我们将每个特征划分的信息增益值存储在一个dict中  
    # 键是特征名, 值是信息增益值  
    information_gains = dict()  
  
    # 对所有的特征进行遍历, 使用信息增益对每个特征进行计算  
    for feature in features:  
  
        # 左子树保证所有的样本的这个特征取值为0  
        left_split_target = data[data[feature] == 0][target]  
  
        # 右子树保证所有的样本的这个特征取值为1  
        right_split_target = data[data[feature] == 1][target]  
  
        # 计算左子树的信息熵  
        left_entropy = information_entropy(left_split_target)  
  
        # 计算左子树的权重  
        left_weight = len(left_split_target) / (len(left_split_target) + len(right_split_target))  
  
        # 计算右子树的信息熵  
        right_entropy = information_entropy(right_split_target) # YOUR CODE HERE
```

```

# 计算右子树的权重
right_weight = len(right_split_target) / (len(left_split_target) + len(right_split_target))

# 计算当前结点的信息熵
current_entropy = information_entropy(data[target])

# 计算使用当前特征划分的信息增益
gain = current_entropy - left_entropy * left_weight - right_entropy * right_weight # YOUR CODE HERE

# 将特征名与增益值以键值对的形式存储在information_gains中
information_gains[feature] = gain

if annotate:
    print(" ", feature, gain)

return information_gains

```

4.2 信息增益率

信息增益率：

$$\text{Gain_ratio}(D, a) = \frac{\text{Gain}(D, a)}{\text{IV}(a)}$$

其中

$$\text{IV}(a) = - \sum_{v=1}^V \frac{|D^v|}{|D|} \log_2 \frac{|D^v|}{|D|}$$

完成计算所有特征信息增益率的函数

这里要完成五个部分

```

def compute_information_gain_ratios(data, features, target, annotate = False):
    """
    计算所有特征的信息增益率并保存起来

    Parameter
    -----
    data: pd.DataFrame, 带有特征和标记的数据
    features: list(str), 特征名组成的list
    target: str, 特征的名字
    annotate: boolean, default False, 是否打印注释

    Returns
    -----
    gain_ratios: dict, key: str, 特特征名
                  value: float, 信息增益率
    ...

    gain_ratios = dict()

    # 对所有的特征进行遍历, 使用当前的划分方法对每个特征进行计算
    for feature in features:

        # 左子树保证所有的样本的这个特征取值为0
        left_split_target = data[data[feature] == 0][target]

        # 右子树保证所有的样本的这个特征取值为1
        right_split_target = data[data[feature] == 1][target]

        # 计算左子树的信息熵
        left_entropy = information_entropy(left_split_target)

        # 计算左子树的权重
        left_weight = len(left_split_target) / (len(left_split_target) + len(right_split_target))

        # 计算右子树的信息熵
        right_entropy = information_entropy(right_split_target) # YOUR CODE HERE

```

```

# 计算右子树的权重
right_weight = len(right_split_target) / (len(left_split_target) + len(right_split_target))

# 计算当前结点的信息熵
current_entropy = information_entropy(data[target])

# 计算当前结点的信息增益

gain = current_entropy - left_entropy*left_weight - right_entropy*right_weight      # YOUR CC

# 计算IV公式中，当前特征为0的值
if left_weight == 0:
    left_IV = 0
else:
    left_IV = left_weight*np.log2(left_weight)                                     # YOUR CC

# 计算IV公式中，当前特征为1的值
if right_weight == 0:
    right_IV = 0
else:
    right_IV = right_weight*np.log2(right_weight)                                # YOUR CC

# IV 等于所有子树iv之和的相反数
IV = -(left_IV + right_IV)

# 计算使用当前特征划分的信息增益率
# 这里为了防止iv是0，导致除法得到np.inf（无穷），在分母加了一个很小的小数
gain_ratio = gain / (IV + np.finfo(np.longdouble).eps)

# 信息增益率的存储
gain_ratios[feature] = gain_ratio

if annotate:
    print(" ", feature, gain_ratio)

return gain_ratios

```

4.3 基尼指数

数据集 D 的基尼值：

$$\begin{aligned} \text{Gini}(D) &= \sum_{k=1}^{|Y|} \sum_{k' \neq k} p_k p_{k'} \\ &= 1 - \sum_{k=1}^{|Y|} p_k^2. \end{aligned}$$

属性 a 的基尼指数：

$$\text{Gini_index}(D, a) = \sum_{v=1}^V \frac{|D^v|}{|D|} \text{Gini}(D^v)$$

完成数据集基尼值的计算
这里需要填写三部分

```

def gini(labels_in_node):
    ...
    计算一个结点内样本的基尼指数

    Parameters
    -----
    label_in_data: np.ndarray, 样本的标记, 如[-1, -1, 1, 1, 1]

    Returns
    -----
    gini: float, 基尼指数
    ...

    # 统计样本总个数
    num_of_samples = labels_in_node.shape[0]

    if num_of_samples == 0:
        return 0

    # 统计出1的个数
    num_of_positive = len(labels_in_node[labels_in_node == 1])

    # 统计出-1的个数
    num_of_negative = len(labels_in_node[labels_in_node == -1]) # YOUR CODE HERE

    # 统计正例的概率
    prob_positive = num_of_positive / num_of_samples

    # 统计负例的概率
    prob_negative = num_of_negative / num_of_samples # YOUR CODE HERE

    # 计算基尼值
    gini = 1 - prob_positive*prob_positive - prob_negative*prob_negative

    return gini

```

然后计算所有特征的基尼指数

这里需要填写三部分

```

def compute_gini_indices(data, features, target, annotate = False):
    ...
    计算使用各个特征进行划分时，各特征的基尼指数

    Parameter
    -----
    data: pd.DataFrame, 带有特征和标记的数据

    features: list(str), 特征名组成的list

    target: str, 特征的名字

    annotate: boolean, default False, 是否打印注释

    Returns
    -----
    gini_indices: dict, key: str, 特征名
                  value: float, 基尼指数
    ...

    gini_indices = dict()
    # 对所有的特征进行遍历，使用当前的划分方法对每个特征进行计算
    for feature in features:
        # 左子树保证所有的样本的这个特征取值为0
        left_split_target = data[data[feature] == 0][target]

        # 右子树保证所有的样本的这个特征取值为1
        right_split_target = data[data[feature] == 1][target]

        # 计算左子树的基尼值
        left_gini = gini(left_split_target)

        # 计算左子树的权重
        left_weight = len(left_split_target) / (len(left_split_target) + len(right_split_target))

        # 计算右子树的基尼值
        right_gini = gini(right_split_target) # YOUR CODE HERE

```

```
# 计算右子树的权重
right_weight = len(right_split_target) / (len(left_split_target) + len(right_split_target))

# 计算当前结点的基尼指数
gini_index = left_gini*left_weight + right_gini*right_weight # YOUR CODE HERE

# 存储
gini_indices[feature] = gini_index

if annotate:
    print(" ", feature, gini_index)

return gini_indices
```

给定划分方法和数据，找到最优的划分特征

Parameters

data: pd.DataFrame, 带有特征和标记的数据

features: list(str), 特征名组成的 list

target: str, 特征的名字

criterion: str, 使用哪种指标，三种选项: 'information_gain', 'gain_ratio', 'gini'

annotate: boolean, default False, 是否打印注释

Returns

best_feature: str, 最佳的划分特征的名字

5. 完成最优特征的选择

到此，我们完成了三种划分策略的实现，接下来就是完成获取最优特征的函数
这里需要填写三个部分

```
def best_splitting_feature(data, features, target, criterion = 'gini', annotate = False):
    """
    给定划分方法和数据，找到最优的划分特征

    Parameters
    -----
    data: pd.DataFrame, 带有特征和标记的数据

    features: list(str), 特征名组成的list

    target: str, | 特征的名字

    criterion: str, 使用哪种指标，三种选项: 'information_gain', 'gain_ratio', 'gini'

    annotate: boolean, default False, 是否打印注释

    Returns
    -----
    best_feature: str, 最佳的划分特征的名字

    ...
    if criterion == 'information_gain':
        if annotate:
            print('using information gain')

        # 得到当前所有特征的信息增益
        information_gains = compute_information_gains(data, features, target, annotate)

        # information_gains是一个dict类型的对象，我们要找值最大的那个元素的键是谁
        # 根据这些特征和他们的信息增益，找到最佳的划分特征
        best_feature = max(information_gains, key = information_gains.get) # YOUR CODE HERE

        return best_feature

    elif criterion == 'gain_ratio':
        if annotate:
            print('using information gain ratio')

        # 得到当前所有特征的信息增益率
        gain_ratios = compute_information_gain_ratios(data, features, target, annotate)

        # 根据这些特征和他们的信息增益率，找到最佳的划分特征
        best_feature = max(gain_ratios, key = gain_ratios.get) # YOUR CODE HERE

        return best_feature

    elif criterion == 'gini':
        if annotate:
            print('using gini')

        # 得到当前所有特征的基尼指数
        gini_indices = compute_gini_indices(data, features, target, annotate)

        # 根据这些特征和他们的基尼指数，找到最佳的划分特征
        best_feature = max(gini_indices, key = gini_indices.get) # YOUR CODE HERE

        return best_feature
    else:
        raise Exception("传入的criterion不合规!", criterion)
```

6. 判断结点内样本的类别是否为同一类

这里需要填写两个部分

```
def intermediate_node_num_mistakes(labels_in_node):
    ...
    求树的结点中，样本数少的那个类的样本有多少，比如输入是[1, 1, -1, -1, 1]，返回2

    Parameter:
    -----
    labels_in_node: np.ndarray, pd.Series

    Returns:
    -----
    int: 个数

    ...
    # 如果传入的array为空，返回0
    if len(labels_in_node) == 0:
        return 0

    # 统计1的个数
    num_of_one = len(labels_in_node[labels_in_node == 1]) # YOUR CODE HERE

    # 统计-1的个数
    num_of_minus_one = len(labels_in_node[labels_in_node == -1]) # YOUR CODE HERE

    return num_of_one if num_of_minus_one > num_of_one else num_of_minus_one
```

7. 创建叶子结点

```
def create_leaf(target_values):
    ...
    计算出当前叶子结点的标记是什么，并且将叶子结点信息保存在一个dict中

    Parameter:
    -----
    target_values: pd.Series, 当前叶子结点内样本的标记

    Returns:
    -----
    leaf: dict, 表示一个叶结点，
        leaf['splitting_features'], None, 叶结点不需要划分特征
        leaf['left'], None, 叶结点没有左子树
        leaf['right'], None, 叶结点没有右子树
        leaf['is_leaf'], True, 是否是叶子结点
        leaf['prediction'], int, 表示该叶子结点的预测值
    ...

    # 创建叶子结点
    leaf = {'splitting_feature' : None,
            'left' : None,
            'right' : None,
            'is_leaf': True}

    # 数结点内-1和+1的个数
    num_ones = len(target_values[target_values == +1])
    num_minus_ones = len(target_values[target_values == -1])

    # 叶子结点的标记使用少数服从多数的原则，为样本数多的那类的标记，保存在 leaf['prediction']
    if num_ones > num_minus_ones:
        leaf['prediction'] = 1
    else:
        leaf['prediction'] = -1

    # 返回叶子结点
    return leaf
```

8. 递归地创建决策树

递归的创建决策树

递归算法终止的三个条件：

1. 如果结点内所有的样本的标记都相同，该结点就不再继续划分，直接做叶子结点即可
2. 如果结点所有的特征都已经在之前使用过了，在当前结点无剩余特征可供划分样本，该结点直接做叶子结点
3. 如果当前结点的深度已经达到了我们限制的树的最大深度，直接做叶子结点

```
def decision_tree_create(data, features, target, criterion = 'gini', current_depth = 0, max_depth = 10, annce
...
Parameter:
-----
data: pd.DataFrame, 数据

features: iterable, 特征组成的可迭代对象，比如一个list

target: str, 标记的名字

criterion: 'str', 特征划分方法，只支持三种：'information_gain', 'gain_ratio', 'gini'

current_depth: int, 当前深度，递归的时候需要记录

max_depth: int, 树的最大深度，我们设定的树的最大深度，达到最大深度需要终止递归

Returns:
-----
dict, dict['is_leaf']           : False, 当前顶点不是叶子结点
dict['prediction']             : None, 不是叶子结点就没有预测值
dict['splitting_feature']: splitting_feature, 当前结点是使用哪个特征进行划分的
dict['left']                   : dict
dict['right']                  : dict
...

if criterion not in ['information_gain', 'gain_ratio', 'gini']:
    raise Exception("传入的criterion不合规!", criterion)

# 复制一份特征，存储起来，每使用一个特征进行划分，我们就删除一个
remaining_features = features[:]

# 取出标记值
target_values = data[target]
print("-" * 50)
print("Subtree, depth = %s (%s data points)." % (current_depth, len(target_values)))

# 终止条件1
# 如果当前结点内所有样本同属一类，即这个结点中，各类别样本数最小的那个等于0
# 使用前面写的intermediate_node_num_mistakes来完成这个判断
if intermediate_node_num_mistakes(target_values) == 0:                                # YOUR CODE HERE
    print("Stopping condition 1 reached.")
```

```

    return create_leaf(target_values)  # 创建叶子结点

# 终止条件2
# 如果已经没有剩余的特征可供分割，即remaining_features为空

if remaining_features == None:
    print("Stopping condition 2 reached.")
    return create_leaf(target_values)  # 创建叶子结点

# YOUR CODE HERE

# 终止条件3
# 如果已经到达了我们要求的最大深度，即当前深度达到了最大深度

if current_depth == max_depth:
    print("Reached maximum depth. Stopping for now.")
    return create_leaf(target_values)  # 创建叶子结点

# YOUR CODE HERE

# 找到最优划分特征
# 使用best_splitting_feature这个函数

splitting_feature = best_splitting_feature(data, features, target, criterion)      # YOUR CODE HERE

# 使用我们找到的最优特征将数据划分成两份
# 左子树的数据
left_split = data[data[splitting_feature] == 0]

# 右子树的数据
right_split = data[data[splitting_feature] == 1]                                     # YOUR CODE HERE

# 现在已经完成划分，我们要从剩余特征中删除掉当前这个特征
remaining_features.remove(splitting_feature)

# 打印当前划分使用的特征，打印左子树样本个数，右子树样本个数
print("Split on feature %s. (%s, %s)" % (\n
                                             splitting_feature, len(left_split), len(right_split)))

# 如果使用当前的特征，将所有的样本都划分到一棵子树中，那么就直接将这棵子树变成叶子结点
# 判断左子树是不是“完美”的
if len(left_split) == len(data):
    print("Creating leaf node.")
    return create_leaf(left_split[target])

```

```

# 判断右子树是不是“完美”的
if len(right_split) == len(data):
    print("Creating right node.")                                              # YOUR CODE HERE
    return create_right(right_split[target])

# 递归地创建左子树
left_tree = decision_tree_create(left_split, remaining_features, target,
                                  criterion, current_depth + 1, max_depth, annotate)

# 递归地创建右子树

right_tree = decision_tree_create(right_split, remaining_features, target, criterion,
                                   current_depth + 1, max_depth, annotate) # YOUR CODE HERE

# 返回树的非叶子结点
return {'is_leaf'          : False,
        'prediction'       : None,
        'splitting_feature': splitting_feature,
        'left'              : left_tree,
        'right'             : right_tree}

```

训练一个模型

```
my_decision_tree = decision_tree_create(train_data, one_hot_features, target, 'gini', max_depth = 6, annotate = False)

Subtree, depth = 0 (73564 data points).
Split on feature emp_length_5 years. (67854, 5710)
-----
Subtree, depth = 1 (67854 data points).
Split on feature emp_length_< 1 year. (61228, 6626)
-----
Subtree, depth = 2 (61228 data points).
Split on feature home_ownership_own. (56181, 5047)
-----
Subtree, depth = 3 (56181 data points).
Split on feature emp_length_4 years. (51399, 4782)
-----
Subtree, depth = 4 (51399 data points).
Split on feature emp_length_9 years. (48916, 2483)
-----
Subtree, depth = 5 (48916 data points).
Split on feature emp_length_8 years. (45939, 2977)
-----
Subtree, depth = 6 (45939 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (2977 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (2483 data points).
Split on feature emp_length_1 year. (2483, 0)
Creating leaf node.
-----
Subtree, depth = 4 (4782 data points).
Split on feature emp_length_1 year. (4782, 0)
Creating leaf node.
-----
Subtree, depth = 3 (5047 data points).
Split on feature home_ownership_MORTGAGE. (5047, 0)
Creating leaf node.
-----
Subtree, depth = 2 (6626 data points).
Split on feature emp_length_1 year. (6626, 0)
Creating leaf node.
-----
Subtree, depth = 1 (5710 data points).
Split on feature emp_length_1 year. (5710, 0)

Creating leaf node.
```

现在，模型就训练好了

9. 预测

接下来我们需要完成预测函数

```
def classify(tree, x, annotate = False):
    """
    递归的进行预测，一次只能预测一个样本

    Parameters
    -----
    tree: dict

    x: pd.Series，待预测的样本

    annotate: boolean, 是否显示注释

    Returns
    -----
    返回预测的标记
    """

    if tree[ 'is_leaf' ]:
        if annotate:
            print ("At leaf, predicting %s" % tree[ 'prediction' ])
        return tree[ 'prediction' ]
    else:
        split_feature_value = x[tree[ 'splitting_feature' ]]
        if annotate:
            print ("Split on %s = %s" % (tree[ 'splitting_feature' ], split_feature_value))
        if split_feature_value == 0:
            return classify(tree[ 'left' ], x, annotate)
        else:
            return classify(tree[ 'right' ], x, annotate)
```

我们取测试集第一个样本来测试

```
test_sample = test_data.iloc[0]
print(test_sample)
```

```
safe_loans          1
grade_A             0
grade_B             0
grade_C             0
grade_D             0
grade_E             1
grade_F             0
grade_G             0
term_ 36 months    1
term_ 60 months    0
home_ownership_MORTGAGE  1
home_ownership_OTHER   0
home_ownership_OWN     0
home_ownership_RENT    0
emp_length_1 year    0
emp_length_10+ years  0
emp_length_2 years    1
emp_length_3 years    0
emp_length_4 years    0
emp_length_5 years    0
emp_length_6 years    0
emp_length_7 years    0
emp_length_8 years    0
emp_length_9 years    0
emp_length_< 1 year    0
Name: 37225, dtype: int64
```

```

print('True class: %s' % (test_sample['safe_loans']))
print('Predicted class: %s' % classify(my_decision_tree, test_sample))

True class: 1
Predicted class: 1

```

打印出使用决策树判断的过程

```

classify(my_decision_tree, test_sample, annotate=True)

Split on emp_length_5 years = 0
Split on emp_length< 1 year = 0
Split on home_ownership_OWN = 0
Split on emp_length_4 years = 0
Split on emp_length_9 years = 0
Split on emp_length_8 years = 0
At leaf, predicting 1
1

```

10. 在测试集上对我们的模型进行评估

```

from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score

```

先来编写一个批量预测的函数，传入的是整个测试集那样的pd.DataFrame，这个函数返回一个np.ndarray，存储模型的预测结果
这里需要填写一个部分

```

def predict(tree, data):
    """
    按行遍历data，对每个样本进行预测，将值存在prediction中，最后返回np.ndarray

    Parameter
    -----
    tree: dict, 模型

    data: pd.DataFrame, 数据

    Returns
    -----
    predictions: np.ndarray, 模型对这些样本的预测结果
    """
    predictions = np.zeros(len(data)) # 长度和data一样

    # YOUR CODE HERE
    for i in range(len(data)):
        predictions[i] = classify(tree, data.iloc[i])

    return predictions

```

11. 请你计算使用不同评价指标得到模型的四项指标的值，填写在下方表格内

树的最大深度为6

```

# YOUR CODE HERE
#信息增益指标
my_decision_tree_test= decision_tree_create(train_data, one_hot_features, target, 'information_gain', max_depth = 6, an
prediction_Test_Information= predict(my_decision_tree_test, test_data)
accuracy = accuracy_score(test_data[target], prediction_Test_Information)
precision = precision_score(test_data[target], prediction_Test_Information)
recall = recall_score(test_data[target], prediction_Test_Information)
f1 = f1_score(test_data[target], prediction_Test_Information)
print("精度：" ,accuracy)
print("查准率：" ,precision)
print("查全率：" ,recall)
print("f1：" ,f1)

```

结果：

```

-----
Subtree, depth = 0 (73564 data points).
Split on feature grade_A. (60204, 13360)
-----
Subtree, depth = 1 (60204 data points).
Split on feature grade_B. (37768, 22436)
-----
Subtree, depth = 2 (37768 data points).
-----
```

```

Split on feature grade_C. (19878, 17890)
-----
Subtree, depth = 3 (19878 data points).
Split on feature term_ 36 months. (8812, 11066)
-----
Subtree, depth = 4 (8812 data points).
Split on feature home_ownership_RENT. (5438, 3374)
-----
```

```
Subtree, depth = 5 (5438 data points).
Split on feature grade_D. (3299, 2139)
-----
Subtree, depth = 6 (3299 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (2139 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (3374 data points).
Split on feature grade_D. (2198, 1176)
-----
Subtree, depth = 6 (2198 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (1176 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (11066 data points).
Split on feature home_ownership_MORTGAGE. (6987, 4079)
-----
Subtree, depth = 5 (6987 data points).
Split on feature grade_F. (6650, 337)
-----
Subtree, depth = 6 (6650 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (337 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (4079 data points).
Split on feature grade_G. (4003, 76)
-----
Subtree, depth = 6 (4003 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (76 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 3 (17890 data points).
Split on feature term_60 months. (14064, 3826)
-----
Subtree, depth = 4 (14064 data points).
Split on feature home_ownership_MORTGAGE. (8209, 5855)
-----
Subtree, depth = 5 (8209 data points).
Split on feature emp_length_2 years. (7209, 1000)
-----
Subtree, depth = 6 (7209 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (1000 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (5855 data points).
Split on feature emp_length_10+ years. (3802, 2053)
-----
Subtree, depth = 6 (3802 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (2053 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (3826 data points).
Split on feature home_ownership_RENT. (2750, 1076)
-----
Subtree, depth = 5 (2750 data points).
Split on feature emp_length_1 year. (2616, 134)
-----
Subtree, depth = 6 (2616 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (134 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (1076 data points).
Split on feature emp_length_6 years. (1018, 58)
-----
Subtree, depth = 6 (1018 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (58 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 2 (22436 data points).
Split on feature term_36 months. (1934, 20502)
-----
Subtree, depth = 3 (1934 data points).
Split on feature home_ownership_MORTGAGE. (689, 1245)
-----
Subtree, depth = 4 (689 data points).
Split on feature emp_length_7 years. (647, 42)
-----
Subtree, depth = 5 (647 data points).
Split on feature emp_length_10+ years. (483, 164)
-----
Subtree, depth = 6 (483 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (164 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (42 data points).
Split on feature home_ownership_RENT. (8, 34)
-----
Subtree, depth = 6 (8 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (34 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (1245 data points).
Split on feature emp_length_3 years. (1152, 93)
-----
Subtree, depth = 5 (1152 data points).
Split on feature emp_length_7 years. (1081, 71)
-----
Subtree, depth = 6 (1081 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (71 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (93 data points).
Split on feature grade_C. (93, 0)
Creating leaf node.
-----
Subtree, depth = 3 (20502 data points).
Split on feature home_ownership_MORTGAGE. (10775, 9727)
-----
Subtree, depth = 4 (10775 data points).
Split on feature emp_length_1 year. (9785, 990)
-----
Subtree, depth = 5 (9785 data points).
Split on feature home_ownership_OTHER. (9754, 31)
-----
Subtree, depth = 6 (9754 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (31 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (990 data points).
Split on feature home_ownership_OTHER. (987, 3)
-----
Subtree, depth = 6 (987 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (3 data points).
Stopping condition 1 reached.
```

```
-----  
Subtree, depth = 4 (9727 data points).  
Split on feature emp_length_< 1 year. (9186, 541)  
-----  
Subtree, depth = 5 (9186 data points).  
Split on feature emp_length_9 years. (8807, 379)  
-----  
Subtree, depth = 6 (8807 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (379 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (541 data points).  
Split on feature grade_C. (541, 0)  
Creating leaf node.  
-----  
Subtree, depth = 1 (13360 data points).  
Split on feature home_ownership_MORTGAGE. (5900, 74  
60)  
-----  
Subtree, depth = 2 (5900 data points).  
Split on feature term_ 36 months. (70, 5830)  
-----  
Subtree, depth = 3 (70 data points).  
Split on feature home_ownership_OWN. (50, 20)  
-----  
Subtree, depth = 4 (50 data points).  
Split on feature emp_length_4 years. (48, 2)  
-----  
Subtree, depth = 5 (48 data points).  
Split on feature emp_length_7 years. (47, 1)  
-----  
Subtree, depth = 6 (47 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (1 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 5 (2 data points).  
Split on feature grade_B. (2, 0)  
Creating leaf node.  
-----  
Subtree, depth = 4 (20 data points).  
Split on feature emp_length_5 years. (17, 3)  
-----  
Subtree, depth = 5 (17 data points).  
Split on feature emp_length_1 year. (15, 2)  
-----  
Subtree, depth = 6 (15 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (2 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 5 (3 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 3 (5830 data points).  
Split on feature emp_length_3 years. (5283, 547)  
-----  
Subtree, depth = 4 (5283 data points).  
Split on feature emp_length_1 year. (4705, 578)  
-----  
Subtree, depth = 5 (4705 data points).  
Split on feature home_ownership_OTHER. (4689, 16)  
-----  
Subtree, depth = 6 (4689 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (16 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 5 (578 data points).  
Split on feature home_ownership_OTHER. (576, 2)  
-----  
Subtree, depth = 6 (576 data points).  
-----  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (2 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 4 (547 data points).  
Split on feature home_ownership_RENT. (77, 468)  
-----  
Subtree, depth = 6 (77 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (468 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (2 data points).  
Split on feature grade_B. (2, 0)  
Creating leaf node.  
-----  
Subtree, depth = 2 (7460 data points).  
Split on feature emp_length_2 years. (6879, 581)  
-----  
Subtree, depth = 3 (6879 data points).  
Split on feature emp_length_4 years. (6397, 482)  
-----  
Subtree, depth = 4 (6397 data points).  
Split on feature emp_length_1 year. (6036, 361)  
-----  
Subtree, depth = 5 (6036 data points).  
Split on feature emp_length_3 years. (5475, 561)  
-----  
Subtree, depth = 6 (5475 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (561 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (361 data points).  
Split on feature term_ 36 months. (7, 354)  
-----  
Subtree, depth = 6 (7 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 6 (354 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 4 (482 data points).  
Split on feature term_ 36 months. (14, 468)  
-----  
Subtree, depth = 5 (14 data points).  
Split on feature grade_B. (14, 0)  
Creating leaf node.  
-----  
Subtree, depth = 5 (468 data points).  
Split on feature grade_B. (468, 0)  
Creating leaf node.  
-----  
Subtree, depth = 3 (581 data points).  
Split on feature term_ 60 months. (569, 12)  
-----  
Subtree, depth = 4 (569 data points).  
Split on feature grade_B. (569, 0)  
Creating leaf node.  
-----  
Subtree, depth = 4 (12 data points).  
Split on feature grade_B. (12, 0)  
Creating leaf node.  
精度: 0.8122056154802928  
查准率: 0.8122387390142941  
查全率: 0.9999497928956947  
f1: 0.8963724740087312
```

```

#信息增益率指标
my_decision_tree_test= decision_tree_create(train_
prediction_Test_Information= predict(my_decision_t
accuracy = accuracy_score(test_data[target], predicti
precision = precision_score(test_data[target], predi
recall = recall_score(test_data[target], predicti
f1 = f1_score(test_data[target], prediction_Test_]
print("精度：" ,accuracy)
print("查准率：" ,precision)
print("查全率：" ,recall)
print("f1：" ,f1)

-----
Subtree, depth = 0 (73564 data points).
Split on feature grade_F. (71229, 2335)
-----
Subtree, depth = 1 (71229 data points).
Split on feature grade_A. (57869, 13360)
-----
Subtree, depth = 2 (57869 data points).
Split on feature grade_G. (57232, 637)
-----
Subtree, depth = 3 (57232 data points).
Split on feature grade_E. (51828, 5404)
-----
Subtree, depth = 4 (51828 data points).
Split on feature grade_D. (40326, 11502)
-----
Subtree, depth = 5 (40326 data points).
Split on feature term_36 months. (5760, 34566)
-----
Subtree, depth = 6 (5760 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (34566 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (11502 data points).
Split on feature term_60 months. (8187, 3315)
-----
Subtree, depth = 6 (8187 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (3315 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (5404 data points).
Split on feature term_36 months. (3185, 2219)
-----
Subtree, depth = 5 (3185 data points).
Split on feature home_ownership_OTHER. (3184, 1)
-----
Subtree, depth = 6 (3184 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (1 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 5 (2219 data points).
Split on feature emp_length_1 year. (2011, 208)
-----
Subtree, depth = 6 (2011 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (208 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 3 (637 data points).
Split on feature emp_length_3 years. (590, 47)
-----
Subtree, depth = 4 (590 data points).
Split on feature emp_length_2 years. (541, 49)
-----
Subtree, depth = 5 (541 data points).
Split on feature home_ownership_OWN. (495, 46)
-----
Subtree, depth = 6 (495 data points).

Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (46 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (49 data points).
Split on feature term_36 months. (32, 17)
-----
Subtree, depth = 6 (32 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (17 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (47 data points).
Split on feature home_ownership_OTHER. (46, 1)
-----
Subtree, depth = 5 (46 data points).
Split on feature home_ownership_OWN. (44, 2)
-----
Subtree, depth = 6 (44 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (2 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (1 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 2 (13360 data points).
Split on feature term_36 months. (259, 13101)
-----
Subtree, depth = 3 (259 data points).
Split on feature emp_length_9 years. (252, 7)
-----
Subtree, depth = 4 (252 data points).
Split on feature home_ownership_RENT. (202, 50)
-----
Subtree, depth = 5 (202 data points).
Split on feature emp_length_8 years. (192, 10)
-----
Subtree, depth = 6 (192 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (10 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 5 (50 data points).
Split on feature emp_length_4 years. (48, 2)
-----
Subtree, depth = 6 (48 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (2 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (7 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 3 (13101 data points).
Split on feature home_ownership_MORTGAGE. (5830, 72
71)
-----
Subtree, depth = 4 (5830 data points).
Split on feature emp_length_7 years. (5592, 238)
-----
Subtree, depth = 5 (5592 data points).
Split on feature emp_length_3 years. (5045, 547)
-----
Subtree, depth = 6 (5045 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (547 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (238 data points).
Split on feature home_ownership_RENT. (54, 184)

```

```
-----  
Subtree, depth = 6 (54 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (184 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 4 (7271 data points).  
Split on feature emp_length_2 years. (6702, 569)  
-----  
Subtree, depth = 5 (6702 data points).  
Split on feature emp_length_4 years. (6234, 468)  
-----  
Subtree, depth = 6 (6234 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (468 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (569 data points).  
Split on feature grade_B. (569, 0)  
Creating leaf node.  
-----  
Subtree, depth = 1 (2335 data points).  
Split on feature emp_length_7 years. (2197, 138)  
-----  
Subtree, depth = 2 (2197 data points).  
Split on feature term_60 months. (478, 1719)  
-----  
Subtree, depth = 3 (478 data points).  
Split on feature emp_length_8 years. (460, 18)  
-----  
Subtree, depth = 4 (460 data points).  
Split on feature emp_length_4 years. (433, 27)  
-----  
Subtree, depth = 5 (433 data points).  
Split on feature home_ownership_MORTGAGE. (287, 14  
6)  
-----  
Subtree, depth = 6 (287 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (146 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (27 data points).  
Split on feature home_ownership_own. (25, 2)  
-----  
Subtree, depth = 6 (25 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (2 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 4 (18 data points).  
Split on feature home_ownership_own. (17, 1)  
-----  
Subtree, depth = 5 (17 data points).  
Split on feature home_ownership_MORTGAGE. (11, 6)  
-----  
Subtree, depth = 6 (11 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (6 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (1 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 3 (1719 data points).  
Split on feature home_ownership_OTHER. (1717, 2)  
-----  
Subtree, depth = 4 (1717 data points).  
Split on feature emp_length_3 years. (1577, 140)  
-----  
Subtree, depth = 5 (1577 data points).  
Split on feature home_ownership_RENT. (904, 673)  
-----  
Subtree, depth = 6 (904 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (673 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (140 data points).  
Split on feature home_ownership_RENT. (73, 67)  
-----  
Subtree, depth = 6 (73 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (67 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 4 (2 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 2 (138 data points).  
Split on feature term_60 months. (29, 109)  
-----  
Subtree, depth = 3 (29 data points).  
Split on feature home_ownership_own. (25, 4)  
-----  
Subtree, depth = 4 (25 data points).  
Split on feature home_ownership_MORTGAGE. (13, 12)  
-----  
Subtree, depth = 5 (13 data points).  
Split on feature grade_A. (13, 0)  
Creating leaf node.  
-----  
Subtree, depth = 5 (12 data points).  
Split on feature grade_A. (12, 0)  
Creating leaf node.  
-----  
Subtree, depth = 4 (4 data points).  
Split on feature grade_A. (4, 0)  
Creating leaf node.  
-----  
Subtree, depth = 3 (109 data points).  
Split on feature home_ownership_RENT. (51, 58)  
-----  
Subtree, depth = 4 (51 data points).  
Split on feature home_ownership_MORTGAGE. (8, 43)  
-----  
Subtree, depth = 5 (8 data points).  
Split on feature grade_A. (8, 0)  
Creating leaf node.  
-----  
Subtree, depth = 5 (43 data points).  
Split on feature grade_A. (43, 0)  
Creating leaf node.  
-----  
Subtree, depth = 4 (58 data points).  
Split on feature grade_A. (58, 0)  
Creating leaf node.  
精度: 0.8117774198152642  
查准率: 0.8124897892501225  
查全率: 0.9987699259445212  
f1: 0.8960508090942874
```

```

#基尼指数指标
my_decision_tree_test= decision_tree_create(train_data, one_hot_features, target, 'gini', max_depth = 6, annotate = False)
prediction_Test_Information= predict(my_decision_tree_test, test_data)
accuracy = accuracy_score(test_data[target], prediction_Test_Information)
precision = precision_score(test_data[target], prediction_Test_Information)
recall = recall_score(test_data[target], prediction_Test_Information)
f1 = f1_score(test_data[target], prediction_Test_Information)
print("精度：" ,accuracy)
print("查准率：" ,precision)
print("查全率：" ,recall)
print("f1：" ,f1)

```

```

-----
Subtree, depth = 0 (73564 data points).
Split on feature emp_length_5 years. (67854, 5710)
-----
Subtree, depth = 1 (67854 data points).
Split on feature emp_length_< 1 year. (61228, 6626)
-----
Subtree, depth = 2 (61228 data points).
Split on feature home_ownership_OWN. (56181, 5047)
-----
Subtree, depth = 3 (56181 data points).
Split on feature emp_length_4 years. (51399, 4782)
-----
Subtree, depth = 4 (51399 data points).
Split on feature emp_length_9 years. (48916, 2483)
-----
Subtree, depth = 5 (48916 data points).
Split on feature emp_length_8 years. (45939, 2977)
-----
Subtree, depth = 6 (45939 data points).
Reached maximum depth. Stopping for now.
-----  

-----  

Subtree, depth = 6 (2977 data points).
Reached maximum depth. Stopping for now.
-----  

-----  

Subtree, depth = 5 (2483 data points).
Split on feature emp_length_1 year. (2483, 0)
Creating leaf node.
-----  

-----  

Subtree, depth = 4 (4782 data points).
Split on feature emp_length_1 year. (4782, 0)
Creating leaf node.
-----  

-----  

Subtree, depth = 3 (5047 data points).
Split on feature home_ownership_MORTGAGE. (5047, 0)
Creating leaf node.
-----  

-----  

Subtree, depth = 2 (6626 data points).
Split on feature emp_length_1 year. (6626, 0)
Creating leaf node.
-----  

-----  

Subtree, depth = 1 (5710 data points).
Split on feature emp_length_1 year. (5710, 0)
Creating leaf node.
精度: 0.8122463960198193
查准率: 0.8122463960198193
查全率: 1.0
f1: 0.8963973086703121

```

树的最大深度为6

双击此处编写

划分标准	精度	查准率	查全率	F1
信息增益	0.8122056154802928	0.8122387390142941	0.9999497928956947	0.8963724740087312
信息增益率	0.8117774198152642	0.8124897892501225	0.9987699259445212	0.8960508090942874
基尼指数	0.8122463960198193	0.8122463960198193	1.0	0.8963973086703121

12. 扩展：使用Echarts绘制决策树

我们可以使用echarts绘制出我们训练的决策树，这时候可以利用pyecharts这个库 [pyecharts](#)
pyecharts可以与jupyter notebook无缝衔接，直接在notebook中绘制图表。提醒：[pyecharts](#)还未支持jupyter lab

```
# 导入树形图
from pyecharts import Tree
```

其实和我们训练得到的树结构类似，只不过每个结点有个“name”属性，表示这个结点的名字，“value”表示它的值，“children”是一个list，里面还有这样的dict，我们可以写一个递归的函数完成这种数据的生成

```
def generate_echarts_data(tree):
    # 当前顶点的dict
    value = dict()

    # 如果传入的tree已经是叶子结点了
    if tree['is_leaf'] == True:
        # 它的value就设置为预测的标记
        value['value'] = tree['prediction']

        # 它的名字就叫"label: 标记"
        value['name'] = 'label: %s' % (tree['prediction'])

        # 直接返回这个dict即可
        return value

    # 如果传入的tree不是叶子结点，名字就叫当前这个顶点的划分特征，子树是一个list
    # 分别增加左子树和右子树到children中
    value['name'] = tree['splitting_feature']
    value['children'] = [generate_echarts_data(tree['left']), generate_echarts_data(tree['right'])]
    return value

data = generate_echarts_data(my_decision_tree)
```

使用下面的代码进行绘制，绘制完成后，树的结点是可以点击的，点击后会展开它的子树

```
tree = Tree(width=800, height=400)
tree.add("", 
         [data,
          tree_collapse_interval=5,
          tree_top="15%",
          tree_right="20%",
          tree_symbol = 'rect',
          tree_symbol_size = 20,
         ])
tree.render()
tree
```

第四题：实现预剪枝

实验内容

1. 实现使用信息增益率划分的预剪枝
2. 计算出带有预剪枝和不带预剪枝的决策树的精度，查准率，查全率和 F1 值(最大深度为 6， 使用信息增益率)，保留 4 位小数，四舍五入

我们会以“信息增益率(information gain ratio)”作为划分准则，构造带有预剪枝的二叉决策树
使用的数据和第三题一样，剪枝需要使用验证集，所以数据划分策略会和第三题不同

1. 读取数据

```
# 导入类库
import pandas as pd
import numpy as np

# 导入数据
loans = pd.read_csv('data/lendingclub/lending-club-data.csv', low_memory=False)

# 对数据进行预处理，将safe_loans作为标记
loans['safe_loans'] = loans['bad_loans'].apply(lambda x : +1 if x==0 else -1)
del loans['bad_loans']

features = ['grade',                      # grade of the loan
            'term',                        # the term of the loan
            'home_ownership',              # home_ownership status: own, mortgage or rent
            'emp_length',                  # number of years of employment
           ]
target = 'safe_loans'
loans = loans[features + [target]]
```

2. 划分训练集和测试集

```
from sklearn.utils import shuffle

loans = shuffle(loans, random_state = 34)
```

我们使用数据的60%做训练集，20%做验证集，20%做测试集

```
split_line1 = int(len(loans) * 0.6)
split_line2 = int(len(loans) * 0.8)
train_data = loans.iloc[: split_line1]
validation_data = loans.iloc[split_line1: split_line2]
test_data = loans.iloc[split_line2:]
```

1. 读取数据

```
# 导入类库
import pandas as pd
import numpy as np

# 导入数据
loans = pd.read_csv('data/lendingclub/lending-club-data.csv', low_memory=False)

# 对数据进行预处理，将safe_loans作为标记
loans['safe_loans'] = loans['bad_loans'].apply(lambda x : +1 if x==0 else -1)
del loans['bad_loans']

features = ['grade',                      # grade of the loan
            'term',                        # the term of the loan
            'home_ownership',              # home_ownership status: own, mortgage or rent
            'emp_length',                  # number of years of employment
           ]
target = 'safe_loans'
loans = loans[features + [target]]
```

2. 划分训练集和测试集

```
from sklearn.utils import shuffle

loans = shuffle(loans, random_state = 34)
```

我们使用数据的60%做训练集，20%做验证集，20%做测试集

```
split_line1 = int(len(loans) * 0.6)
split_line2 = int(len(loans) * 0.8)
train_data = loans.iloc[: split_line1]
validation_data = loans.iloc[split_line1: split_line2]
test_data = loans.iloc[split_line2:]

test_data_tmp = one_hot_encoding(test_data, features)
test_data = pd.DataFrame(columns = train_data.columns)
for feature in train_data.columns:
    if feature in test_data_tmp.columns:
        test_data[feature] = test_data_tmp[feature].copy()
    else:
        test_data[feature] = np.zeros(test_data_tmp.shape[0], dtype = 'uint8')
```

打印一下3个数据集的shape

```
print(train_data.shape, validation_data.shape, test_data.shape)

(73564, 25) (24521, 25) (24522, 25)
```

4. 实现信息增益率的计算

信息熵：

$$\text{Ent}(D) = - \sum_{k=1}^{|Y|} p_k \log_2 p_k$$

信息增益：

$$\text{Gain}(D, a) = \text{Ent}(D) - \sum_{v=1}^V \frac{|D^v|}{|D|} \text{Ent}(D^v)$$

信息增益率：

$$\text{Gain_ratio}(D, a) = \frac{\text{Gain}(D, a)}{\text{IV}(a)}$$

其中

$$\text{IV}(a) = - \sum_{v=1}^V \frac{|D^v|}{|D|} \log_2 \frac{|D^v|}{|D|}$$

计算信息熵时约定：若 $p = 0$ ，则 $p \log_2 p = 0$

先实现信息熵，再实现信息增益率

```
def compute_information_gain_ratios(data, features, target, annotate = False):
    ...
    计算所有特征的信息增益率并保存起来

    Parameter
    -----
    data: pd.DataFrame, 带有特征和标记的数据
    features: list(str), 特征名组成的list
    target: str, 特征的名字
    annotate: boolean, default False, 是否打印注释

    Returns
    -----
    gain_ratios: dict, key: str, 特征名
                  value: float, 信息增益率
    ...

    gain_ratios = dict()

    # 对所有的特征进行遍历, 使用当前的划分方法对每个特征进行计算
    for feature in features:

        # 左子树保证所有的样本的这个特征取值为0
        left_split_target = data[data[feature] == 0][target]

        # 右子树保证所有的样本的这个特征取值为1
        right_split_target = data[data[feature] == 1][target]

        # 计算左子树的信息熵
        left_entropy = information_entropy(left_split_target)

        # 计算左子树的权重
        left_weight = len(left_split_target) / (len(left_split_target) + len(right_split_target))

        # 计算右子树的信息熵
        right_entropy = information_entropy(right_split_target)
```

```

# 计算右子树的权重
right_weight = len(right_split_target) / (len(left_split_target) + len(right_split_target))

# 计算当前结点的信息熵
current_entropy = information_entropy(data[target])

# 计算当前结点的信息增益
gain = current_entropy - left_entropy * left_weight - right_entropy * right_weight

# 计算IV公式中，当前特征为0的值
if left_weight == 0:
    left_IV = 0
else:
    left_IV = left_weight * np.log2(left_weight)

# 计算IV公式中，当前特征为1的值
if right_weight == 0:
    right_IV = 0
else:
    right_IV = right_weight * np.log2(right_weight)

# IV 等于所有子树IV之和的相反数
IV = -(left_IV + right_IV)

# 计算使用当前特征划分的信息增益率
# 这里为了防止IV是0，导致除法得到np.inf，在分母加了一个很小的小数
gain_ratio = gain / (IV + np.finfo(np.longdouble).eps)

# 信息增益率的存储
gain_ratios[feature] = gain_ratio

if annotate:
    print(" ", feature, gain_ratio)

return gain_ratios

```

5. 完成最优特征的选择

这里我们没有实现信息增益和基尼指数的最优特征求解，感兴趣的同学可以按上一题实现

```

def best_splitting_feature(data, features, target, criterion = 'gain_ratios', annotate = False):
    ...
    给定划分方法和数据，找到最优的划分特征

    Parameters
    -----
    data: pd.DataFrame, 带有特征和标记的数据
    features: list(str)，特征名组成的list
    target: str， 特征的名字
    criterion: str, 使用哪种指标，三种选项: 'information_gain', 'gain_ratio', 'gini'
    annotate: boolean, default False，是否打印注释

    Returns
    -----
    best_feature: str, 最佳的划分特征的名字
    ...

    if criterion == 'information_gain':
        if annotate:
            print('using information gain')
        return None

    elif criterion == 'gain_ratio':
        if annotate:
            print('using information gain ratio')

    # 得到当前所有特征的信息增益率
    gain_ratios = compute_information_gain_ratios(data, features, target, annotate)

    # 根据这些特征和他们的信息增益率，找到最佳的划分特征
    best_feature = max(gain_ratios.items(), key = lambda x: x[1])[0]

```

```

        return best_feature

    elif criterion == 'gini':
        if annotate:
            print('using gini')
        return None
    else:
        raise Exception("传入的criterion不合规!", criterion)

```

6. 判断结点内样本的类别是否为同一类

```

def intermediate_node_num_mistakes(labels_in_node):
    """
    求树的结点中，样本数少的那个类的样本有多少，比如输入是[1, 1, -1, -1, 1]，返回2

    Parameter
    -----
    labels_in_node: np.ndarray, pd.Series

    Returns
    -----
    int: 个数
    ...

    # 如果传入的array为空，返回0
    if len(labels_in_node) == 0:
        return 0

    # 统计1的个数
    num_of_one = len(labels_in_node[labels_in_node == 1]) # YOUR CODE HERE

    # 统计-1的个数
    num_of_minus_one = len(labels_in_node[labels_in_node == -1]) # YOUR CODE HERE

    return num_of_one if num_of_minus_one > num_of_one else num_of_minus_one

```

7. 创建叶子结点

先编写一个辅助函数majority_class，求树的结点中，样本数多的那个类是什么

```

def majority_class(labels_in_node):
    """
    求树的结点中，样本数多的那个类是什么
    ...

    # 如果传入的array为空，返回0
    if len(labels_in_node) == 0:
        return 0

    # 统计1的个数
    num_of_one = len(labels_in_node[labels_in_node == 1])

    # 统计-1的个数
    num_of_minus_one = len(labels_in_node[labels_in_node == -1])

    return 1 if num_of_minus_one < num_of_one else -1

```

```

def create_leaf(target_values):
    ...
    计算出当前叶子结点的标记是什么，并且将叶子结点信息保存在一个dict中

    Parameter:
    -----
    target_values: pd.Series, 当前叶子结点内样本的标记

    Returns:
    -----
    leaf: dict, 表示一个叶结点,
        leaf['splitting_features'], None, 叶结点不需要划分特征
        leaf['left'], None, 叶结点没有左子树
        leaf['right'], None, 叶结点没有右子树
        leaf['is_leaf'], True, 是否是叶子结点
        leaf['prediction'], int, 表示该叶子结点的预测值
    ...

    # 创建叶子结点
    leaf = {'splitting_feature' : None,
            'left' : None,
            'right' : None,
            'is_leaf': True}

    # 数结点内-1和+1的个数
    num_ones = len(target_values[target_values == +1])
    num_minus_ones = len(target_values[target_values == -1])

    # 叶子结点的标记使用少数服从多数的原则，为样本数多的那类的标记，保存在 leaf['prediction']
    leaf['prediction'] = majority_class(target_values)

    # 返回叶子结点
    return leaf

```

8. 递归地创建决策树

递归的创建决策树

决策树终止的三个条件：

1. 如果结点内所有的样本的标记都相同，该结点就不需要再继续划分，直接做叶子结点即可
2. 如果结点所有的特征都已经在之前使用过了，在当前结点无剩余特征可供划分样本，该结点直接做叶子结点
3. 如果当前结点的深度已经达到了我们限制的树的最大深度，直接做叶子结点

对于预剪枝来说，实质上是增加了第四个终止条件：

4. 如果当前结点划分后，模型的泛化能力没有提升，则不进行划分

如何判断泛化能力有没有提升？我们需要使用验证集

就像使用训练集递归地划分数据一样，我们在递归地构造决策树时，也需要递归地将验证集进行划分，计算决策树在验证集上的精度

因为我们是递归地对决策树进行划分，所以每次计算验证集上精度是否提升时，也只是针对当前结点内的样本，因为是否对当前结点内的样本进行划分，不会影响它的兄弟结点及兄弟结点的子结点的精度

需要完成11个部分

```

def decision_tree_create(training_data, validation_data, features, target, criterion = 'gain_ratios',
                        pre_pruning = False, current_depth = 0, max_depth = 10, annotate = False):
    ...
    Parameter:
    -----
    trianing_data: pd.DataFrame, 数据
    features: iterable, 特征组成的可迭代对象
    target: str, 标记的名字
    criterion: 'str', 特征划分方法
    current_depth: int, 当前深度
    max_depth: int, 树的最大深度
    Returns:
    -----
    dict, dict['is_leaf']      : False, 当前顶点不是叶子结点
    dict['prediction']        : None, 不是叶子结点就没有预测值
    dict['splitting_feature']: splitting_feature, 当前结点是使用哪个特征进行划分的
    dict['left']               : dict
    dict['right']              : dict
    ...
    if criterion not in ['information_gain', 'gain_ratio', 'gini']:
        raise Exception("传入的criterion不合规!", criterion)

    # 复制一份特征，存储起来，每使用一个特征进行划分，我们就删除一个
    remaining_features = features[:]

    # 取出标记值
    target_values = training_data[target]
    validation_values = validation_data[target]
    print("." * 50)
    print("Subtree, depth = %s (%s data points)." % (current_depth, len(target_values)))

    # 终止条件1
    # 如果当前结点内所有样本同属一类，即这个结点中，各类别样本数最小的那个等于0
    # 使用前面写的intermediate_node_num_mistakes来完成这个判断
    if intermediate_node_num_mistakes(target_values) == 0:
        print("Stopping condition 1 reached.")
        return create_leaf(target_values)  # 创建叶子结点

    # 终止条件2
    # 如果已经没有剩余的特征可供分割
    if remaining_features == {}:
        print("Stopping condition 2 reached.") #
        return create_leaf(target_values)  # 创建叶子结点

    # 终止条件3
    # 如果已经到达了我们要求的最大深度
    if current_depth == max_depth:
        print("Reached maximum depth. Stopping for now.")
        return create_leaf(target_values)  # 创建叶子结点

    # 找到最优划分特征
    # 使用best_splitting_feature这个函数
    splitting_feature = best_splitting_feature(training_data, features, target, criterion)

    # 使用我们找到的最优特征将数据划分成两份
    # 左子树的数据
    left_split = training_data[training_data[splitting_feature] == 0]

    # 右子树的数据
    right_split = training_data[training_data[splitting_feature] == 1]

    # 使用这个最优特征对验证集进行划分
    validation_left_split = validation_data[validation_data[splitting_feature] == 0]
    validation_right_split = validation_data[validation_data[splitting_feature] == 1]

    # 如果使用预剪枝，需要判断在验证集上的精度是否提升了
    if pre_pruning:
        # 首先计算不划分的时候的在验证集上的精度，也就是当前结点为叶子结点
        # 统计当前结点样本中，样本数多的那个类(majority class)
        true_class = majority_class(validation_data[target])

        # 判断验证集在不划分时的精度，分子加eps是因为，有可能在划分的时候，验证集的样本数为0
        acc_without_splitting = len(validation_values[validation_values == true_class]) / (len(validation_values) +
                                            np.finfo(np.longdouble).eps)

        # 对当前结点进行划分，统计划分后，左子树的majority class
        left_true_class = majority_class(left_split[target])

```

```

# 对当前结点进行划分，统计右子树的majority class
right_true_class = majority_class(right_split[target]) # YOUR CODE HERE

# 统计验证集左子树中有多少样本是左子树的majority class
vali_left_num_of_majority = len(validation_left_split[validation_left_split[target] == left_true_class])

# 统计验证集右子树中有多少样本是右子树的majority class
vali_right_num_of_majority = len(validation_right_split[validation_right_split[target] == right_true_class])

# 计算划分后的精度
acc_with_splitting = (vali_left_num_of_majority + vali_right_num_of_majority) / (len(validation_data) + np.finfo(np.longdouble).eps)

if annotate == True:
    print('acc before splitting: %.3f' % (acc_without_splitting))
    print('acc after splitting: %.3f' % (acc_with_splitting))

# 如果划分后的精度小于等于划分前的精度，那就不划分，当前结点直接变成叶子结点
# 否则继续划分，创建左右子树
if acc_with_splitting < acc_without_splitting:
    print('Pre-Pruning')
    # 创建叶子结点
    return create_leaf(target_values)

# 从剩余特征中删除掉当前这个特征
remaining_features.remove(splitting_feature)

print("Split on feature %s. (%s, %s)" % (\n
    splitting_feature, len(left_split), len(right_split)))

# 如果使用当前的特征，将所有的样本都划分到一棵子树中，那么就直接将这棵子树变成叶子结点
# 判断左子树是不是“完美”的
if len(left_split) == len(training_data):
    print("Creating leaf node.")
    return create_leaf(left_split[target])

# 判断右子树是不是“完美”的
if len(right_split) == len(training_data):
    print("Creating right node.")
    return create_leaf(right_split[target])

# 递归地创建左子树，需要传入验证集的左子树
left_tree = decision_tree_create(left_split, validation_left_split, remaining_features, target, criterion,
                                 pre_pruning, current_depth + 1, max_depth, annotate)

# 递归地创建右子树，需要传入验证集的右子树
right_tree = right_tree = decision_tree_create(right_split, validation_right_split, remaining_features, target,
                                               criterion, pre_pruning, current_depth + 1, max_depth, annotate)

return {'is_leaf': False,
        'prediction': None,
        'splitting_feature': splitting_feature,
        'left': left_tree,
        'right': right_tree}

```

训练一棵没有预剪枝的决策树，最大深度为 6

tree_without_pre_pruning = decision_tree_create(train_data, validation_data,
one_hot_features, target, criterion = 'gain_ratio', pre_pruning = False, current_depth =
0, max_depth = 6, annotate = False)

训练一棵有预剪枝的决策树，最大深度为 6

tree_with_pre_pruning = decision_tree_create(train_data, validation_data,
one_hot_features, target, criterion = "gain_ratio", pre_pruning = True, current_depth
= 0, max_depth = 6, annotate = False)

9. 预测

```

def classify(tree, x, annotate = False):
    """
    递归的进行预测，一次只能预测一个样本

    Parameters
    -----
    tree: dict

    x: pd.Series, 样本

    x: pd.DataFrame, 待预测的样本

    annotate, boolean, 是否显示注释

    Returns
    -----
    返回预测的标记
    """

    if tree['is_leaf']:
        if annotate:
            print ("At leaf, predicting %s" % tree['prediction'])
        return tree['prediction']
    else:
        split_feature_value = x[tree['splitting_feature']]
        if annotate:
            print ("Split on %s = %s" % (tree['splitting_feature'], split_feature_value))
        if split_feature_value == 0:
            return classify(tree['left'], x, annotate)
        else:
            return classify(tree['right'], x, annotate)

```

```

def predict(tree, data):
    """
    按行遍历data，对每个样本进行预测，将值存储起来，最后返回np.ndarray

    Parameter
    -----
    tree, dict, 模型

    data, pd.DataFrame, 数据

    Returns
    -----
    predictions, np.ndarray, 模型对这些样本的预测结果
    """

    predictions = np.zeros(len(data))

    # YOUR CODE HERE
    for i in range(len(data)):
        predictions[i] = classify(tree, data.iloc[i])

    return predictions

```

在下方计算出带有预剪枝和不带预剪枝的决策树的精度，查准率，查全率和F1值(最大深度为6，使用信息增益率)，保留4位小数，四舍五入

```

# YOUR CODE HERE
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
#有预剪枝
tree_with_pre_pruning = decision_tree_create(train_data, validation_data, one_hot_features, target,
                                              criterion = "gain_ratio", pre_pruning = True, current_depth = 0,
                                              max_depth = 6, annotate = False)
prediction_Test_gain_ratio= predict(tree_with_pre_pruning,test_data)
accuracy = accuracy_score(test_data[target],prediction_Test_gain_ratio)
precision = precision_score(test_data[target], prediction_Test_gain_ratio)
recall = recall_score(test_data[target], prediction_Test_gain_ratio)
f1 = f1_score(test_data[target], prediction_Test_gain_ratio)
print('精度:', accuracy)
print('查准率:', precision)
print('查全率:', recall)
print('f1值:', f1)

```

```
-----  
Subtree, depth = 0 (73564 data points).  
Split on feature grade_F. (71229, 2335)  
-----  
Subtree, depth = 1 (71229 data points).  
Split on feature grade_A. (57869, 13360)  
-----  
Subtree, depth = 2 (57869 data points).  
Split on feature grade_G. (57232, 637)  
-----  
Subtree, depth = 3 (57232 data points).  
Split on feature grade_E. (51828, 5404)  
-----  
Subtree, depth = 4 (51828 data points).  
Split on feature grade_D. (40326, 11502)  
-----  
Subtree, depth = 5 (40326 data points).  
Split on feature term_ 36 months. (5760, 34566)  
-----  
Subtree, depth = 6 (5760 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (34566 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (11502 data points).  
Split on feature term_ 60 months. (8187, 3315)  
-----  
Subtree, depth = 6 (8187 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (3315 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 4 (5404 data points).  
Split on feature term_ 36 months. (3185, 2219)  
-----  
Subtree, depth = 5 (3185 data points).  
Split on feature home_ownership_OTHER. (3184, 1)  
-----  
Subtree, depth = 6 (3184 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (1 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 5 (2219 data points).  
Split on feature emp_length_1 year. (2011, 208)  
-----  
Subtree, depth = 6 (2011 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (208 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 3 (637 data points).  
Split on feature emp_length_3 years. (590, 47)  
-----  
Subtree, depth = 4 (590 data points).  
Split on feature emp_length_2 years. (541, 49)  
-----  
Subtree, depth = 5 (541 data points).  
Pre-Pruning  
-----  
Subtree, depth = 5 (49 data points).  
Split on feature term_ 36 months. (32, 17)  
-----  
Subtree, depth = 6 (32 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (17 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 4 (47 data points).  
Pre-Pruning  
-----  
Subtree, depth = 2 (13360 data points).  
Split on feature term_ 36 months. (259, 13101)  
-----  
Subtree, depth = 3 (259 data points).  
Split on feature emp_length_9 years. (252, 7)  
-----  
Subtree, depth = 4 (252 data points).  
Split on feature home_ownership_RENT. (202, 50)  
-----  
Subtree, depth = 5 (202 data points).  
-----  
Split on feature emp_length_8 years. (192, 10)  
-----  
Subtree, depth = 6 (192 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (10 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 5 (50 data points).  
Split on feature emp_length_4 years. (48, 2)  
-----  
Subtree, depth = 6 (48 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (2 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 4 (7 data points).  
Stopping condition 1 reached.  
-----  
Subtree, depth = 3 (13101 data points).  
Split on feature home_ownership_MORTGAGE. (5830, 7271)  
-----  
Subtree, depth = 4 (5830 data points).  
Split on feature emp_length_7 years. (5592, 238)  
-----  
Subtree, depth = 5 (5592 data points).  
Split on feature emp_length_3 years. (5045, 547)  
-----  
Subtree, depth = 6 (5045 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (547 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (238 data points).  
Split on feature home_ownership_RENT. (54, 184)  
-----  
Subtree, depth = 6 (54 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (184 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 4 (7271 data points).  
Split on feature emp_length_2 years. (6702, 569)  
-----  
Subtree, depth = 5 (6702 data points).  
Split on feature emp_length_4 years. (6234, 468)  
-----  
Subtree, depth = 6 (6234 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (468 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (569 data points).  
Split on feature grade_B. (569, 0)  
Creating leaf node.  
-----  
Subtree, depth = 1 (2335 data points).  
Split on feature emp_length_7 years. (2197, 138)  
-----  
Subtree, depth = 2 (2197 data points).  
Split on feature term_ 60 months. (478, 1719)  
-----  
Subtree, depth = 3 (478 data points).  
Split on feature emp_length_8 years. (460, 18)  
-----  
Subtree, depth = 4 (460 data points).  
Pre-Pruning  
-----  
Subtree, depth = 4 (18 data points).  
Split on feature home_ownership_own. (17, 1)  
-----  
Subtree, depth = 5 (17 data points).  
Split on feature home_ownership_MORTGAGE. (11, 6)  
-----  
Subtree, depth = 6 (11 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 6 (6 data points).  
Reached maximum depth. Stopping for now.  
-----  
Subtree, depth = 5 (1 data points).  
Stopping condition 1 reached.
```

```

Subtree, depth = 3 (1719 data points).
Split on feature home_ownership_OTHER. (1717, 2)
-----
Subtree, depth = 4 (1717 data points).
Split on feature emp_length_3 years. (1577, 140)
-----
Subtree, depth = 5 (1577 data points).
Split on feature home_ownership_RENT. (904, 673)
-----
Subtree, depth = 6 (904 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (673 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (140 data points).
Split on feature home_ownership_RENT. (73, 67)

```

```

-----
Subtree, depth = 6 (73 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (67 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (2 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 2 (138 data points).
Pre-Pruning
精度: 0.809436424435201
查准率: 0.8095024469820554
查全率: 0.9998992494080903
f1值: 0.8946834644249622

```

```

#无预剪枝
tree_without_pre_pruning = decision_tree_create(train_data, validation_data, one_hot_features, target,
                                                criterion = 'gain_ratio', pre_pruning = False, current_depth = 0,
                                                max_depth = 6, annotate = False)
prediction_Test_gain_ratio = predict(tree_without_pre_pruning, test_data)
accuracy = accuracy_score(test_data[target], prediction_Test_gain_ratio)
precision = precision_score(test_data[target], prediction_Test_gain_ratio)
recall = recall_score(test_data[target], prediction_Test_gain_ratio)
f1 = f1_score(test_data[target], prediction_Test_gain_ratio)
print('精度:', accuracy)
print('查准率:', precision)
print('查全率:', recall)
print('f1值:', f1)

```

```

-----
Subtree, depth = 0 (73564 data points).
Split on feature grade_F. (71229, 2335)
-----
Subtree, depth = 1 (71229 data points).
Split on feature grade_A. (57869, 13360)
-----
Subtree, depth = 2 (57869 data points).
Split on feature grade_G. (57232, 637)
-----
Subtree, depth = 3 (57232 data points).
Split on feature grade_E. (51828, 5404)
-----
Subtree, depth = 4 (51828 data points).
Split on feature grade_D. (40326, 11502)
-----
Subtree, depth = 5 (40326 data points).
Split on feature term_36 months. (5760, 34566)
-----
Subtree, depth = 6 (5760 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (34566 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (11502 data points).
Split on feature term_60 months. (8187, 3315)
-----
Subtree, depth = 6 (8187 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (3315 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (5404 data points).
Split on feature term_36 months. (3185, 2219)
-----
Subtree, depth = 5 (3185 data points).
Split on feature home_ownership_OTHER. (3184, 1)
-----
Subtree, depth = 6 (3184 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (1 data points).

```

```

-----
Stopping condition 1 reached.
-----
Subtree, depth = 5 (2219 data points).
Split on feature emp_length_1 year. (2011, 208)
-----
Subtree, depth = 6 (2011 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (208 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 3 (637 data points).
Split on feature emp_length_3 years. (590, 47)
-----
Subtree, depth = 4 (590 data points).
Split on feature emp_length_2 years. (541, 49)
-----
Subtree, depth = 5 (541 data points).
Split on feature home_ownership_OWN. (495, 46)
-----
Subtree, depth = 6 (495 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (46 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (49 data points).
Split on feature term_36 months. (32, 17)
-----
Subtree, depth = 6 (32 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (17 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (47 data points).
Split on feature home_ownership_OTHER. (46, 1)
-----
Subtree, depth = 5 (46 data points).
Split on feature home_ownership_OWN. (44, 2)
-----
Subtree, depth = 6 (44 data points).
Reached maximum depth. Stopping for now.
-----
```

```

Subtree, depth = 6 (2 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (1 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 2 (13360 data points).
Split on feature term_ 36 months. (259, 13101)
-----
Subtree, depth = 3 (259 data points).
Split on feature emp_length_9 years. (252, 7)
-----
Subtree, depth = 4 (252 data points).
Split on feature home_ownership_RENT. (202, 50)
-----
Subtree, depth = 5 (202 data points).
Split on feature emp_length_8 years. (192, 10)
-----
Subtree, depth = 6 (192 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (10 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 5 (50 data points).
Split on feature emp_length_4 years. (48, 2)
-----
Subtree, depth = 6 (48 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (2 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (7 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 3 (13101 data points).
Split on feature home_ownership_MORTGAGE. (5830, 7271)
-----
Subtree, depth = 4 (5830 data points).
Split on feature emp_length_7 years. (5592, 238)
-----
Subtree, depth = 5 (5592 data points).
Split on feature emp_length_3 years. (5045, 547)
-----
Subtree, depth = 6 (5045 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (547 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (238 data points).
Split on feature home_ownership_RENT. (54, 184)
-----
Subtree, depth = 6 (54 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (184 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (7271 data points).
Split on feature emp_length_2 years. (6702, 569)
-----
Subtree, depth = 5 (6702 data points).
Split on feature emp_length_4 years. (6234, 468)
-----
Subtree, depth = 6 (6234 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (468 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (569 data points).
Split on feature grade_B. (569, 0)
Creating leaf node.
-----
Subtree, depth = 1 (2335 data points).
Split on feature emp_length_7 years. (2197, 138)
-----
Subtree, depth = 2 (2197 data points).
Split on feature term_ 60 months. (478, 1719)
-----
Subtree, depth = 3 (478 data points).
Split on feature emp_length_8 years. (460, 18)
-----
Subtree, depth = 4 (460 data points).
Split on feature emp_length_4 years. (433, 27)

-----
```

```

Subtree, depth = 5 (433 data points).
Split on feature home_ownership_MORTGAGE. (287, 146)
-----
Subtree, depth = 6 (287 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (146 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (27 data points).
Split on feature home_ownership_OWN. (25, 2)
-----
Subtree, depth = 6 (25 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (2 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 4 (18 data points).
Split on feature home_ownership_OWN. (17, 1)
-----
Subtree, depth = 5 (17 data points).
Split on feature home_ownership_MORTGAGE. (11, 6)
-----
Subtree, depth = 6 (11 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (6 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (1 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 3 (1719 data points).
Split on feature home_ownership_OTHER. (1717, 2)
-----
Subtree, depth = 4 (1717 data points).
Split on feature emp_length_3 years. (1577, 140)
-----
Subtree, depth = 5 (1577 data points).
Split on feature home_ownership_RENT. (904, 673)
-----
Subtree, depth = 6 (904 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (673 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 5 (140 data points).
Split on feature home_ownership_RENT. (73, 67)
-----
Subtree, depth = 6 (73 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 6 (67 data points).
Reached maximum depth. Stopping for now.
-----
Subtree, depth = 4 (2 data points).
Stopping condition 1 reached.
-----
Subtree, depth = 2 (138 data points).
Split on feature term_ 60 months. (29, 109)
-----
Subtree, depth = 3 (29 data points).
Split on feature home_ownership_OWN. (25, 4)
-----
Subtree, depth = 4 (25 data points).
Split on feature home_ownership_MORTGAGE. (13, 12)
-----
Subtree, depth = 5 (13 data points).
Split on feature grade_A. (13, 0)
Creating leaf node.
-----
Subtree, depth = 5 (12 data points).
Split on feature grade_A. (12, 0)
Creating leaf node.
-----
Subtree, depth = 4 (4 data points).
Split on feature grade_A. (4, 0)
Creating leaf node.
-----
Subtree, depth = 3 (109 data points).
Split on feature home_ownership_RENT. (51, 58)
-----
Subtree, depth = 4 (51 data points).
Split on feature home_ownership_MORTGAGE. (8, 43)
```

```
-----  
Subtree, depth = 5 (8 data points).  
Split on feature grade_A. (8, 0)  
Creating leaf node.  
-----  
Subtree, depth = 5 (43 data points).  
Split on feature grade_A. (43, 0)  
Creating leaf node.
```

```
-----  
Subtree, depth = 4 (58 data points).  
Split on feature grade_A. (58, 0)  
Creating leaf node.  
精度: 0.8088247288149417  
查准率: 0.8097397556890141  
查全率: 0.9984383658253992  
f1 值: 0.8942429164410755
```

双击此处编辑

模型	精度	查准率	查全率	F1
有预剪枝	0.8088247288149417	0.8095024469820554	0.9998992494080903	0.8946834644249622
无预剪枝	0.8088247288149417	0.8097397556890141	0.9984383658253992	0.8942429164410755

(3) 神经网络

第一题：使用 sklearn 的多层感知机

实验内容：

1. 使用 `sklearn.neural_network.MLPClassifier` 完成手写数字分类任务
2. 绘制学习率为 3, 1, 0.1, 0.01 训练集损失函数的变化曲线

1. 读取数据集

我们使用的是 `sklearn` 里面自带的手写数字数据集

```
from sklearn.datasets import load_digits
```

数据集有这几个键

```
load_digits().keys()  
dict_keys(['data', 'target', 'target_names', 'images', 'DESCR'])
```

打印数据集的描述

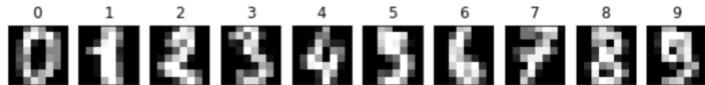
```
print(load_digits()['DESCR'])
```

```
load_digits()['images'][0]
array([[ 0.,  0.,  5., 13.,  9.,  1.,  0.,  0.],
       [ 0.,  0., 13., 15., 10., 15.,  5.,  0.],
       [ 0.,  3., 15.,  2.,  0., 11.,  8.,  0.],
       [ 0.,  4., 12.,  0.,  0.,  8.,  8.,  0.],
       [ 0.,  5.,  8.,  0.,  0.,  9.,  8.,  0.],
       [ 0.,  4., 11.,  0.,  1., 12.,  7.,  0.],
       [ 0.,  2., 14.,  5., 10., 12.,  0.,  0.],
       [ 0.,  0.,  6., 13., 10.,  0.,  0.,  0.]])
```

对数据集的前十张图片可视化

```
import matplotlib.pyplot as plt
%matplotlib inline

_, figs = plt.subplots(1, 10, figsize=(10, 4))
for f, img, lbl in zip(figs, load_digits()['images'][:10], load_digits()['target'][:10]):
    f.imshow(img, cmap = 'gray')
    f.set_title(lbl)
    f.axes.get_xaxis().set_visible(False)
    f.axes.get_yaxis().set_visible(False)
```



2. 划分数据集

```
from sklearn.model_selection import train_test_split
```

取40%为测试集，60%为训练集

```
trainX, testX, trainY, testY = train_test_split(load_digits()['data'], load_digits()['target'], test_size = 0.4, random_state=42)

trainX.shape, trainY.shape, testX.shape, testY.shape
```

((1078, 64), (1078,), (719, 64), (719,))

3. 数据预处理

```
from sklearn.preprocessing import StandardScaler
```

神经网络的训练方法一般是基于梯度的优化算法，如梯度下降，为了让这类算法能更好的优化神经网络，我们往往需要对数据集进行归一化，这里我们选择对数据进行标准化

$$X' = \frac{X - \bar{X}}{\text{std}(X)}$$

其中， \bar{X} 是均值， std 是标准差。减去均值可以让数据以0为中心，除以标准差可以让数据缩放到一个较小的范围内。这样可以使得梯度的下降方向更多样，同时缩小梯度的数量级，让学习变得稳定。

首先需要对训练集进行标准化，针对每个特征求出其均值和标准差，然后用训练集的每个样本减去均值除以标准差，就得到了新的训练集。然后用测试集的每个样本，减去训练集的均值，除以训练集的标准差，完成对测试集的标准化。

```
# 初始化一个标准化器的实例
standard = StandardScaler()

# 对训练集进行标准化，它会计算训练集的均值和标准差保存起来
trainX = standard.fit_transform(trainX)

# 使用标准化器在训练集上的均值和标准差，对测试集进行归一化
testX = standard.transform(testX)
```

可以打印看一下数据集归一化后的效果，均值很接近0

```
trainX.mean(), testX.mean()
(1.4418480839287748e-18, -0.005673919242693978)
```

4. 引入模型

```
from sklearn.neural_network import MLPClassifier
```

我们使用sklearn中自带的MLPClassifier，MLP是多层感知机(multi-layer perceptron)的简称。
在训练的时候需要指定参数，这里我们需要设置的几个参数有：

1. solver: 'sgd'，这个参数的含义是，使用随机梯度下降作为优化算法
2. learning_rate: 'constant'，学习率固定，不衰减
3. momentum: 0，动量设置为0，这是随机梯度下降需要的一个参数，我们设置为0即可
4. max_iter: 设定最大迭代轮数，如果超过这个轮数还没有收敛，就停止训练，并抛出一个warning
5. learning_rate_init，这个参数需要我们进行调整，这是学习率

这个模型会判断，如果连续两轮损失值都没有减少了，就停止训练。

```
model = MLPClassifier(solver = 'sgd', learning_rate = 'constant', momentum = 0, learning_rate_init = 0.1, max_iter = 50)
model.fit(trainX, trainY)
prediction = model.predict(testX)
```

5. 预测与评估

```
from sklearn.metrics import accuracy_score
```

```
accuracy_score(prediction, testY)
```

```
0.9763560500695411
```

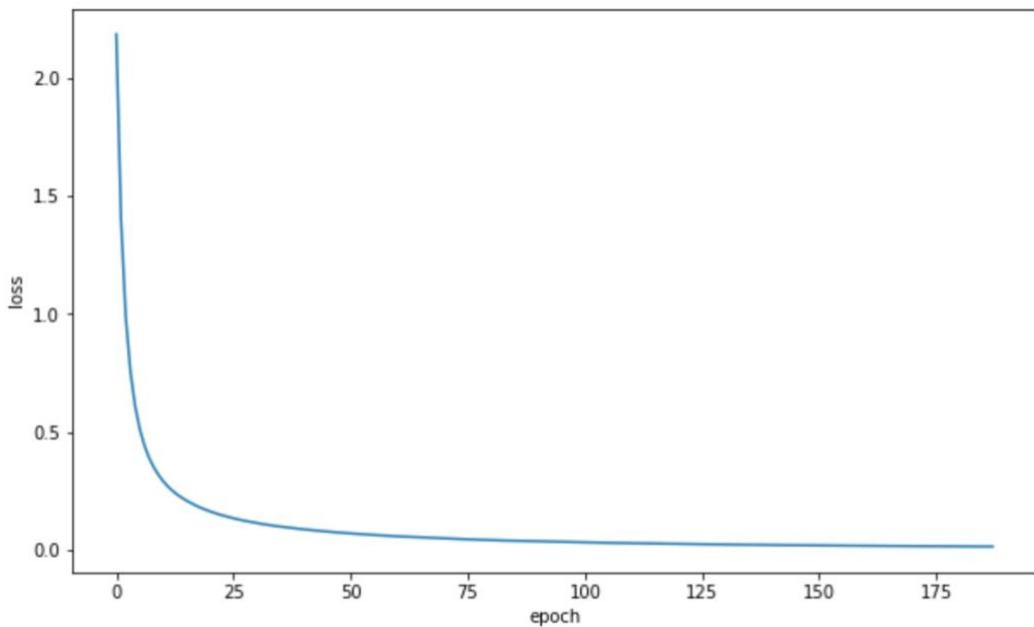
精度达到了98%

6. 绘制训练集损失函数值的变化曲线

我们可以通过model.loss_curve_获取模型在训练过程中，损失函数损失值的变化曲线

```
plt.figure(figsize = (10, 6))
plt.plot(model.loss_curve_)
plt.xlabel('epoch')
plt.ylabel('loss')

Text(0, 0.5, 'loss')
```



可以看到随着迭代轮数的增加，loss降低地越来越缓慢

可以改变学习率，查看模型的学习效果。

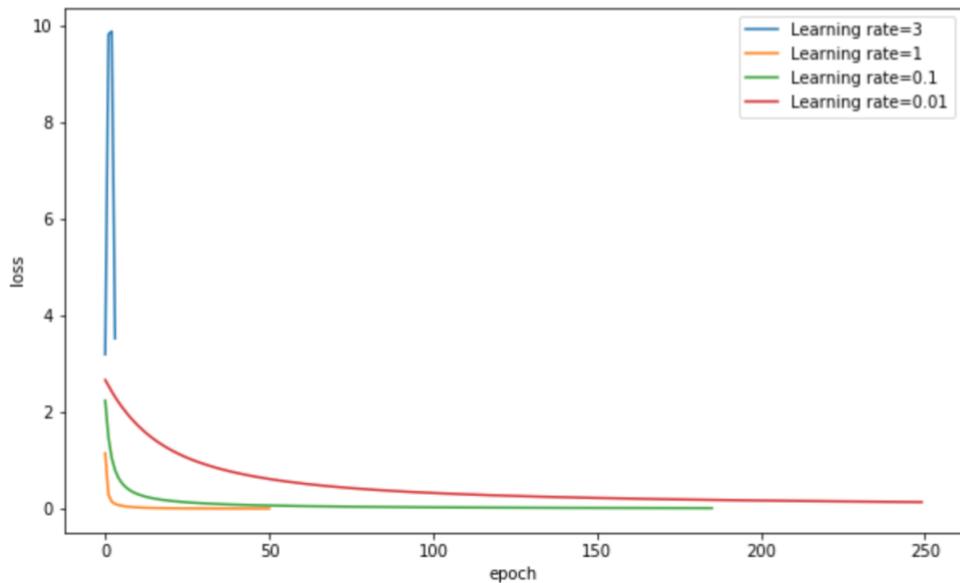
```
#Learning Rate=3  
model_3 = MLPClassifier(solver = 'sgd', learning_rate = 'constant', momentum = 0,  
learning_rate_init = 3, max_iter = 250)  
model_3.fit(trainX, trainY)  
prediction_3 = model_3.predict(testX)
```

```
#Learning Rate=1  
model_1 = MLPClassifier(solver = 'sgd', learning_rate = 'constant', momentum = 0,  
learning_rate_init = 1, max_iter = 250)  
model_1.fit(trainX, trainY)  
prediction_1 = model_1.predict(testX)
```

```
#Learning Rate=0.1  
model_01 = MLPClassifier(solver = 'sgd', learning_rate = 'constant', momentum = 0,  
learning_rate_init = 0.1, max_iter = 250)  
model_01.fit(trainX, trainY)  
prediction_01 = model_01.predict(testX)
```

```
#Learning Rate=0.01
model_001 = MLPClassifier(solver = 'sgd', learning_rate = 'constant', momentum = 0,
learning_rate_init = 0.01, max_iter = 250)
model_001.fit(trainX, trainY)
prediction_001= model_001.predict(testX)
```

```
from pylab import mpl
plt.figure(figsize = (10, 6))
plt.plot(model_3.loss_curve_,label='Learning rate=3')
plt.plot(model_1.loss_curve_,label='Learning rate=1')
plt.plot(model_01.loss_curve_,label='Learning rate=0.1')
plt.plot(model_001.loss_curve_,label='Learning rate=0.01')
plt.xlabel('epoch')
plt.ylabel('loss')
plt.legend()
plt.show()
```



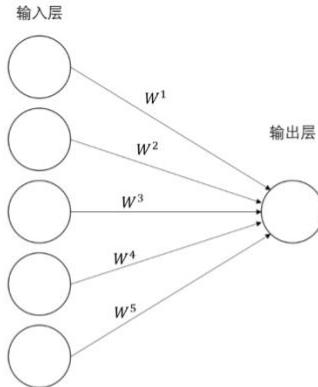
学习率太高的时候会发生过拟合

第二题：神经网络：线性回归

实验内容：

1. 学会梯度下降的基本思想
2. 学会使用梯度下降求解线性回归
3. 了解归一化处理的作用

线性回归



我们来完成最简单的线性回归，上图是一个最简单的神经网络，一个输入层，一个输出层，没有激活函数。我们记输入为 $X \in \mathbb{R}^{n \times m}$ ，输出为 $Z \in \mathbb{R}^n$ 。输入包含了 n 个样本， m 个特征，输出是对这 n 个样本的预测值。输入层到输出层的权重和偏置，我们记为 $W \in \mathbb{R}^{m \times n}$ 和 $b \in \mathbb{R}$ 。输出层没有激活函数，所以上面的神经网络的前向传播过程写为：

$$Z = XW + b$$

我们使用均方误差作为模型的损失函数

$$\text{loss}(y, \hat{y}) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

我们通过调整参数 W 和 b 来降低均方误差，或者说是以降低均方误差为目标，学习参数 W 和参数 b 。当均方误差下降的时候，我们认为当前的模型的预测值 Z 与真值 y 越来越接近，也就是说模型正在学习如何让自己的预测值变得更准确。

梯度下降是一种常用的优化算法，通俗来说就是计算出参数的梯度（损失函数对参数的偏导数的导数值），然后将参数减去参数的梯度乘以一个很小的数（下面的公式），来改变参数，然后重新计算损失函数，再次计算梯度，再次进行调整，通过一定次数的迭代，参数就会收敛到最优点附近。

在我们的这个线性回归问题中，我们的参数是 W 和 b ，使用以下的策略更新参数：

$$W := W - \alpha \frac{\partial \text{loss}}{\partial W}$$

$$b := b - \alpha \frac{\partial \text{loss}}{\partial b}$$

其中， α 是学习率，一般设置为 0.1, 0.01 等。

接下来我们会求解损失函数对参数的偏导数。

损失函数MSE记为：

$$\text{loss}(y, Z) = \frac{1}{n} \sum_{i=1}^n (y_i - Z_i)^2$$

其中， $Z \in \mathbb{R}^n$ 是我们的预测值，也就是神经网络输出层的输出值。这里我们有 n 个样本，实际上是将 n 个样本的预测值与他们的真值相减，取平方后加和。

我们计算损失函数对参数 W 的偏导数，根据链式法则，可以将偏导数拆成两项，分别求解后相乘：

这里我们以矩阵的形式写出推导过程，感兴趣的同学可以尝试使用单个样本进行推到，然后推广到矩阵形式

$$\begin{aligned} \frac{\partial \text{loss}}{\partial W} &= \frac{\partial \text{loss}}{\partial Z} \frac{\partial Z}{\partial W} \\ &= -\frac{2}{n} X^T (y - Z) \\ &= \frac{2}{n} X^T (Z - y) \end{aligned}$$

同理，求解损失函数对参数 b 的偏导数：

$$\begin{aligned} \frac{\partial \text{loss}}{\partial b} &= \frac{\partial \text{loss}}{\partial Z} \frac{\partial Z}{\partial b} \\ &= -\frac{2}{n} \sum_{i=1}^n (y_i - Z_i) \\ &= \frac{2}{n} \sum_{i=1}^n (Z_i - y_i) \end{aligned}$$

1. 导入数据

使用kaggle房价数据，选3列作为特征

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline

# 读取数据
data = pd.read_csv('data/kaggle_house_price_prediction/kaggle_hourse_price_train.csv')

# 使用这3列作为特征
features = ['LotArea', 'BsmtUnfSF', 'GarageArea']
target = 'SalePrice'
data = data[features + [target]]
```

2. 数据预处理

40%做测试集，60%做训练集

```
from sklearn.model_selection import train_test_split
trainX, testX, trainY, testY = train_test_split(data[features], data[target], test_size = 0.4, random_state = 32)
```

训练集876个样本，3个特征，测试集584个样本，3个特征

```
trainX.shape, trainY.shape, testX.shape, testY.shape
((876, 3), (876,), (584, 3), (584,))
```

3. 参数初始化

这里，我们要初始化参数 W 和 b ，其中 $W \in \mathbb{R}^m$, $b \in \mathbb{R}$ ，初始化的策略是将 W 初始化成一个随机数矩阵，参数 b 为0。

```
def initialize(m):
    ...
    参数初始化，将w初始化成一个随机向量，b是一个长度为1的向量

    Parameters
    -----
    m: int, 特征数

    Returns
    -----
    W: np.ndarray, shape = (m, ), 参数W
    b: np.ndarray, shape = (1, ), 参数b
    ...

    # 指定随机种子，这样生成的随机数就是固定的了，这样就可以与下面的测试样例进行比对
    np.random.seed(32)

    W = np.random.normal(size = (m, )) * 0.01
    b = np.zeros((1,))

    return W, b
```

4. 前向传播

这里，我们要完成输入矩阵 X 在神经网络中的计算，也就是完成 $Z = XW + b$ 的计算。

```
def forward(X, W, b):
    """
    前向传播，计算Z = XW + b

    Parameters
    -----
    X: np.ndarray, shape = (n, m)，输入的数据
    W: np.ndarray, shape = (m, )，权重
    b: np.ndarray, shape = (1, )，偏置

    Returns
    -----
    Z: np.ndarray, shape = (n, )，线性组合后的值
    ...

    # 完成Z = XW + b的计算
    Z = np.dot(X,W) + b           # YOUR CODE HERE

    return Z
```

```
# 测试样例
Wt, bt = initialize(trainX.shape[1])
tmp = forward(trainX, Wt, bt)
print(tmp.mean()) # -28.37377
```

-28.37377228144393

5. 损失函数

接下来编写损失函数，我们以均方误差(MSE)作为损失函数，需要大家实现MSE的计算：

$$\text{loss}(y, Z) = \frac{1}{n} \sum_{i=1}^n (y_i - Z_i)^2$$

```
def mse(y_true, y_pred):
    """
    MSE，均方误差

    Parameters
    -----
    y_true: np.ndarray, shape = (n, )，真值
    y_pred: np.ndarray, shape = (n, )，预测值

    Returns
    -----
    loss: float，损失值
    ...

    # 计算MSE
    loss = ( sum((y_true-y_pred)*(y_true-y_pred)) )/len(y_true)

    return loss
```

```
# 测试样例
Wt, bt = initialize(trainX.shape[1])
tmp = mse(trainY, forward(trainX, Wt, bt))
print(tmp) # 39381033680.5
```

39381033680.46006

6. 反向传播

这里我们要完成梯度的计算，也就是计算出损失函数对参数的偏导数的导数值：

$$\frac{\partial \text{loss}}{\partial W} = \frac{2}{n} X^T (Z - y)$$

$$\frac{\partial \text{loss}}{\partial b} = \frac{2}{n} \sum_{i=1}^n (Z_i - y_i)$$

```
def compute_gradient(X, Z, y_true):
    ...
    计算梯度

    Parameters
    -----
    X: np.ndarray, shape = (n, m), 输入的数据
    Z: np.ndarray, shape = (n, ), 线性组合后的值
    y_true: np.ndarray, shape = (n, ), 真值

    Returns
    -----
    dW, np.ndarray, shape = (m, ), 参数w的梯度
    db, np.ndarray, shape = (1, ), 参数b的梯度
    ...
    n = len(y_true)

    # 计算w的梯度
    dW = 2/n * np.dot(X.T, (Z-y_true))           # YOUR CODE HERE

    # 计算b的梯度
    db = 2*np.sum(Z-y_true)/n                      # YOUR CODE HERE

    return dW, db
```

```
# 测试样例
Wt, bt = initialize(trainX.shape[1])
Zt = forward(trainX, Wt, bt)
dWt, dbt = compute_gradient(trainX, Zt, trainY)
print(dWt.shape) # (3,)
print(dWt.mean()) # -1532030241.25
print(dbt.mean()) # -364308.555764
```

```
(3,)
-1532030241.2528899
-364308.5557637409
```

7. 梯度下降

这部分需要实现梯度下降的函数

$$W := W - \alpha \frac{\partial \text{loss}}{\partial W}$$

$$b := b - \alpha \frac{\partial \text{loss}}{\partial b}$$

```
def update(dw, db, w, b, learning_rate):
    ...
    梯度下降，参数更新，不需要返回值，w和b实际上是以引用的形式传入到函数内部，  
函数内改变w和b会直接影响到它们本身，所以不需要返回值

    Parameters
    -----
    dw, np.ndarray, shape = (m, ), 参数w的梯度
    db, np.ndarray, shape = (1, ), 参数b的梯度
    w: np.ndarray, shape = (m, ), 权重
    b: np.ndarray, shape = (1, ), 偏置
    learning_rate, float, 学习率
    ...

    # 更新w
    w -= learning_rate * dw

    # 更新b
    b -= learning_rate * db

# 测试样例
Wt, bt = initialize(trainX.shape[1])
print(Wt.mean()) # 0.00405243937693
print(bt.mean()) # 0.0

Zt = forward(trainX, Wt, bt)
dWt, dbt = compute_gradient(trainX, Zt, trainY)
update(dWt, dbt, Wt, bt, 0.01)
print(Wt.shape) # (3,)
print(Wt.mean()) # 15320302.4166
print(bt.mean()) # 3643.08555764

0.004052439376931716
0.0
(3,)
15320302.416581342
3643.0855576374092
```

完成整个参数更新的过程，先计算梯度，再更新参数，将compute_gradient和update组装在一起。

```
def backward(X, Z, y_true, W, b, learning_rate):
    ...
    使用compute_gradient和update函数，先计算梯度，再更新参数

    Parameters
    -----
    X: np.ndarray, shape = (n, m), 输入的数据
    Z: np.ndarray, shape = (n, ), 线性组合后的值
    y_true: np.ndarray, shape = (n, ), 真值
    W: np.ndarray, shape = (m, ), 权重
    b: np.ndarray, shape = (1, ), 偏置
    learning_rate, float, 学习率
    ...
    # 计算参数的梯度
    dW, db = compute_gradient(X, Z, y_true)                      # YOUR CODE HERE

    # 更新参数
    # YOUR CODE HERE
    update(dW, db, W, b, learning_rate)
```

```
# 测试样例
Wt, bt = initialize(trainX.shape[1])
print(Wt.mean()) # 0.00405243937693
print(bt.mean()) # 0.0

Zt = forward(trainX, Wt, bt)
backward(trainX, Zt, trainY, Wt, bt, 0.01)

print(Wt.shape) # (3,)
print(Wt.mean()) # 15320302.4166
print(bt.mean()) # 3643.08555764
```

8. 训练

训练，我们要迭代 epochs 次，每次迭代的过程中，做一次前向传播和一次反向传播，更新参数

同时记录训练集和测试集上的损失值，后面画图用。然后循环往复，直达到到最大迭代次数 epochs

Parameters

trainX: np.ndarray, shape = (n, m), 训练集

trainY: np.ndarray, shape = (n,), 训练集标记

testX: np.ndarray, shape = (n_test, m), 测试集

testY: np.ndarray, shape = (n_test,), 测试集的标记

W: np.ndarray, shape = (m,), 参数 W

b: np.ndarray, shape = (1,), 参数 b

epochs: int, 要迭代的轮数

learning_rate: float, default 0.01, 学习率

verbose: boolean, default False, 是否打印损失值

Returns

training_loss_list: list(float), 每迭代一次之后，训练集上的损失值

testing_loss_list: list(float), 每迭代一次之后，测试集上的损失值

```
def train(trainX, trainY, testX, testY, W, b, epochs, learning_rate = 0.01, verbose = False):
    training_loss_list = []
    testing_loss_list = []

    for epoch in range(epochs):
        # 这里我们要将神经网络的输出值保存起来，因为后面反向传播的时候需要这个值
        Z = forward(trainX, W, b)

        # 计算训练集的损失值
        training_loss = mse(trainY, Z)

        # 计算测试集的损失值
        testing_loss = mse(testY, forward(testX, W, b))

        # 将损失值存起来
        training_loss_list.append(training_loss)
        testing_loss_list.append(testing_loss)

        # 打印损失值，debug用
        if verbose:
            print('epoch %s training loss: %s' % (epoch+1, training_loss))
            print('epoch %s testing loss: %s' % (epoch+1, testing_loss))
            print()

        # 反向传播，参数更新
        backward(trainX, Z, trainY, W, b, learning_rate)

    return training_loss_list, testing_loss_list

# 测试样例
Wt, bt = initialize(trainX.shape[1])
print(Wt.mean())          # 0.00405243937693
print(bt.mean())          # 0.0

training_loss_list, testing_loss_list = train(trainX, trainY, testX, testY, Wt, bt, 2, learning_rate = 0.01, verbose = True)

print(training_loss_list) # [39381033680.460075, 3.3902307664083424e+23]
print(testing_loss_list) # [38555252685.093872, 4.1516070070405267e+23]
print(Wt.mean())          # -5.70557904608e+13
print(bt.mean())          # -8824267814.59
```

```
# 测试样例
Wt, bt = initialize(trainX.shape[1])
print(Wt.mean())           # 0.00405243937693
print(bt.mean())           # 0.0

training_loss_list, testing_loss_list = train(trainX, trainY, testX, testY, Wt, bt, 2, learning_rate = 0.01, verbose = 1)

print(training_loss_list)  # [39381033680.460075, 3.3902307664083424e+23]
print(testing_loss_list)   # [38555252685.093872, 4.1516070070405267e+23]
print(Wt.mean())           # -5.70557904608e+13
print(bt.mean())           # -8824267814.59106

0.004052439376931716
0.0
[39381033680.46006, 3.390230782482135e+23]
[38555252685.09385, 4.151607023181587e+23]
-57055790600891.0
-8824267814.59106
```

9. 检查

编写一个绘制损失值变化曲线的函数

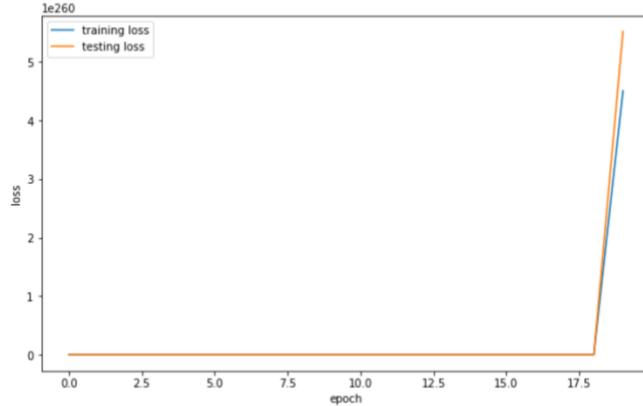
一般我们通过绘制损失函数的变化曲线来判断模型的拟合状态。

一般来说，随着迭代轮数的增加，训练集的loss在下降，而测试集的loss在上升，这说明我们正在不断地让模型在训练集上表现得越来越好，在测试集上表现得越来越糟糕，这就是过拟合的体现。

如果训练集loss和测试集loss共同下降，这就是我们想要的结果，说明模型正在很好的学习。

绘制损失值的变化曲线

```
plot_loss_curve(training_loss_list, testing_loss_list)
```



通过打印损失的信息我们可以看到损失值持续上升，这就说明哪里出了问题。但是如果所有的测试样例都通过了，就说明我们的实现是没有问题的。运行下面的测试样例，观察哪里出了问题。

```
# 测试样例
Wt, bt = initialize(trainX.shape[1])
print('epoch 0, W:', Wt) # [-0.00348894  0.00983703  0.00580923]
print('epoch 0, b:', bt) # [ 0.]
print()

zt = forward(trainX, Wt, bt)
dwt, dbt = compute_gradient(trainX, zt, trainY)
print('dwt:', dwt) # [-4.18172940e+09 -2.19880296e+08 -1.94481031e+08]
print('db:', dbt) # -364308.555764
print()

update(dwt, dbt, Wt, bt, 0.01)
print('epoch 1, W:', Wt) # [ 41817293.96016914  2198802.97412493  1944810.31544993]
print('epoch 1, b:', bt) # [ 3643.08555764]

epoch 0, W: [-0.00348894  0.00983703  0.00580923]
epoch 0, b: [0.]

dwt: [-4.18172940e+09 -2.19880296e+08 -1.94481031e+08]
db: -364308.5557637409

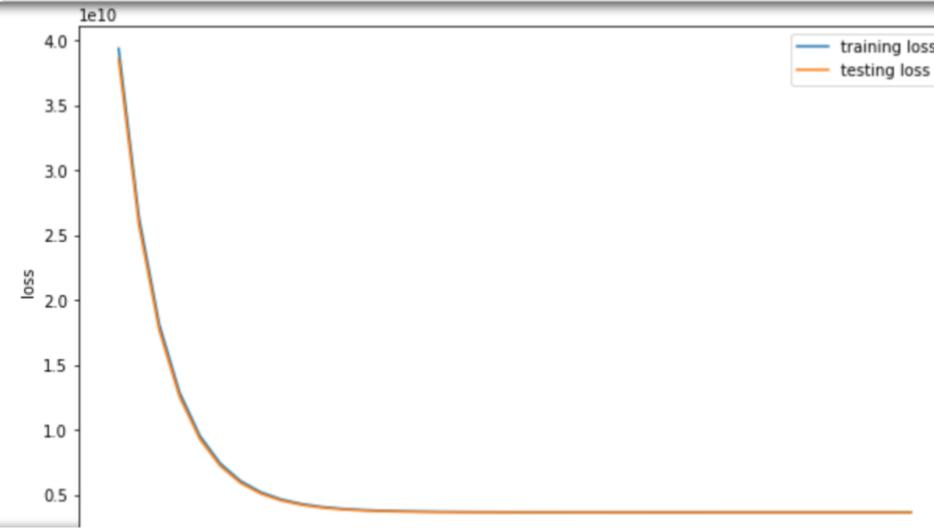
epoch 1, W: [41817293.96016916  2198802.97412493  1944810.31544993]
epoch 1, b: [3643.08555764]
```

可以看到，我们最开始的参数都是在 10^{-3} 这个数量级上，而第一轮迭代时计算出的梯度的数量级在 10^8 左右，这就导致使用梯度下降更新的时候，让参数变成了 10^0 这个数量级左右（学习率为 0.01）。产生这样的问题的主要原因是：我们的原始数据 X 没有经过适当的处理，直接扔到了神经网络中进行训练，导致在计算梯度时，由于 X 的数量级过大，导致梯度的数量级变大，在参数更新时使得参数的数量级不断上升，导致参数无法收敛。

解决的方法也很简单，对参数进行归一化处理，将其标准化，使均值为 0，缩放到 $[-1, 1]$ 附近。

打印损失值变化曲线

```
plot_loss_curve(training_loss_list, testing_loss_list)
```



计算测试集上的MSE

```
prediction = forward(testX_normalized, W, b)  
mse(testY, prediction) ** 0.5
```

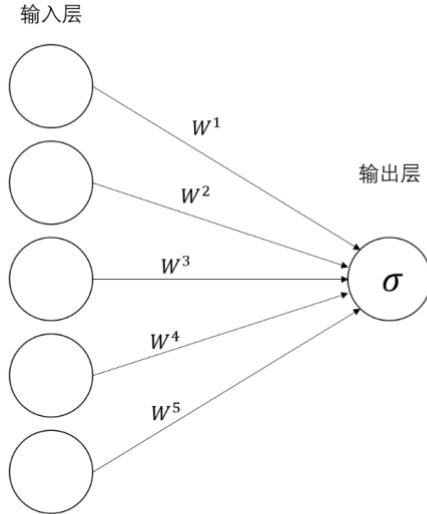
60305.85267910155

第三题：神经网络：对数几率回归

实验内容：

1. 完成对数几率回归
2. 使用梯度下降求解模型参数
3. 绘制模型损失值的变化曲线
4. 调整学习率和迭代轮数，观察损失值曲线的变化
5. 按照给定的学习率和迭代轮数，初始化新的参数，绘制新模型在训练集和测试集上损失值的变化曲线，完成表格内精度的填写

对数几率回归，二分类问题的分类算法，属于线性模型中的一种，我们可以将其抽象为最简单的神经网络



只有一个输入层和一个输出层，还有一个激活函数，sigmoid，简记为 σ 。
我们设输入为 $X \in \mathbb{R}^{n \times m}$ ，输入层到输出层的权重为 $W \in \mathbb{R}^m$ ，偏置 $b \in \mathbb{R}$ 。

激活函数

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

这个激活函数，会将输出层的神经元的输出值转换为一个(0, 1)区间内的数。

因为是二分类问题，我们设类别为0和1，我们将输出值大于0.5的样本分为1类，输出值小于0.5的类分为0类。

前向传播

$$Z = XW + b$$

$$\hat{y} = \sigma(Z)$$

其中， $O \in \mathbb{R}^n$ 为输出层的结果， σ 为sigmoid激活函数。

注意：这里我们其实是做了广播，将 b 复制了 $n - 1$ 份后拼接成了维数为 n 的向量。

所以对数几率回归就可以写为：

$$\hat{y} = \frac{1}{1 + e^{-XW+b}}$$

损失函数

使用对数损失函数，因为对数损失函数较其他损失函数有更好的性质，感兴趣的同学可以去查相关的资料。

针对二分类问题的对数损失函数：

$$\text{loss}(y, \hat{y}) = -y \log \hat{y} - (1 - y) \log (1 - \hat{y})$$

在这个对数几率回归中，我们的损失函数对所有样本取个平均值：

$$\text{loss}(y, \hat{y}) = -\frac{1}{n} \sum_{i=1}^n [y_i \log \hat{y}_i + (1 - y_i) \log (1 - \hat{y}_i)]$$

注意，这里我们的提到的 \log 均为 \ln ，在numpy中为 `np.log`。

因为我们的类别只有0和1，所以在这个对数损失函数中，要么前一项为0，要么后一项为0。

如果当前样本的类别为0，那么前一项就为0，损失函数变为 $-\log(1 - \hat{y})$ ，因为我们的预测值 $0 < \hat{y} < 1$ ，所以 $0 < 1 - \hat{y} < 1$ ， $-\log(1 - \hat{y}) > 0$ ，为了降低损失值，模型需要让预测值 \hat{y} 不断地趋于0。

同理，如果当前样本的类别为1，那么降低损失值就可以使模型的预测值趋于1。

参数更新

求得损失函数对参数的偏导数后，我们就可以使用梯度下降进行参数更新：

$$W := W - \alpha \frac{\partial \text{loss}}{\partial W}$$
$$b := b - \alpha \frac{\partial \text{loss}}{\partial b}$$

其中， α 是学习率，一般设置为 0.1, 0.01 等。

经过一定次数的迭代后，参数会收敛至最优点。这种基于梯度的优化算法很常用，训练神经网络主要使用这类优化算法。

反向传播

我们使用梯度下降更新参数 W 和 b 。为此需要求得损失函数对参数 W 和 b 的偏导数，根据链式法则有：

$$\frac{\partial \text{loss}}{\partial W} = \frac{\partial \text{loss}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z} \frac{\partial Z}{\partial W}$$

这里我们一项一项求，先求第一项：

$$\frac{\partial \text{loss}}{\partial \hat{y}} = -\frac{1}{n} \sum_{i=1}^n \left[\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i} \right]$$

第二项：

$$\begin{aligned} \frac{\partial \hat{y}}{\partial Z} &= \frac{\partial (\frac{1}{1+e^{-Z}})}{\partial Z} \\ &= \frac{e^{-Z}}{(1+e^{-Z})^2} \\ &= \frac{e^{-Z}}{(1+e^{-Z})(1+e^{-Z})} \\ &= \frac{e^{-Z}}{(1+e^{-Z})} \left(1 - \frac{e^{-Z}}{(1+e^{-Z})}\right) \\ &= \sigma(Z)(1 - \sigma(Z)) \end{aligned}$$

第三项：

$$\frac{\partial Z}{\partial W} = X^T$$

综上：

$$\begin{aligned} \frac{\partial \text{loss}}{\partial W} &= \frac{\partial \text{loss}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial Z} \frac{\partial Z}{\partial W} \\ &= -\frac{1}{n} \sum_{i=1}^n \left[\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i} \right] [\sigma(Z_i)(1 - \sigma(Z_i))] X_i^T \\ &= -\frac{1}{n} \sum_{i=1}^n \left[\frac{y_i}{\hat{y}_i} - \frac{1-y_i}{1-\hat{y}_i} \right] [\hat{y}_i(1 - \hat{y}_i)] X_i^T \\ &= -\frac{1}{n} \sum_{i=1}^n [y_i(1 - \hat{y}_i) - \hat{y}_i(1 - y_i)] X_i^T \\ &= -\frac{1}{n} \sum_{i=1}^n (y_i - y_i \hat{y}_i - \hat{y}_i + y_i \hat{y}_i) X_i^T \\ &= -\frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i) X_i^T \\ &= \frac{1}{n} [X^T (\hat{y} - y)] \end{aligned}$$

同理，求 loss 对 b 的偏导数：

注意，由于 b 是被广播成 $n \times K$ 的矩阵，因此实际上 b 对每个样本的损失都有贡献，因此对其求偏导时，要把 n 个样本对它的偏导数加和。

1. 导入数据集

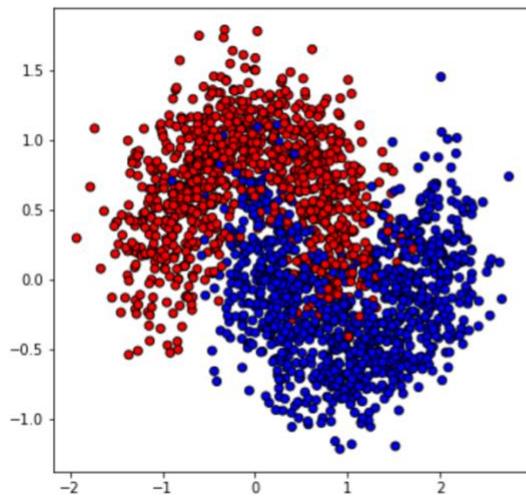
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from matplotlib.colors import ListedColormap
```

我们生成半月形数据

```
from sklearn.datasets import make_moons
X, y = make_moons(n_samples = 2000, noise = 0.3, random_state=0)

plt.figure(figsize = (6, 6))
cm_bright = ListedColormap(['#FF0000', '#0000FF'])
plt.scatter(X[:, 0], X[:, 1], c = y, cmap = cm_bright, edgecolors = 'k')

<matplotlib.collections.PathCollection at 0x1a20c6f828>
```



选择40%的数据作为测试集，60%作为训练集

```
from sklearn.model_selection import train_test_split
trainX, testX, trainY, testY = train_test_split(X, y, test_size = 0.4, random_state = 32)
trainY = trainY
testY = testY

trainX.shape, trainY.shape, testX.shape, testY.shape
```

((1200, 2), (1200,), (800, 2), (800,))

2. 数据预处理

使用和第一题一样的预处理方式

```
from sklearn.preprocessing import StandardScaler
s = StandardScaler()
trainX = s.fit_transform(trainX)
testX = s.transform(testX)
```

3. 定义神经网络

3.1 参数初始化

我们需要对神经网络的参数进行初始化，这个网络中只有两个参数，一个 $W \in \mathbb{R}^m$ ，一个 $b \in \mathbb{R}$ 。初始化的时候，我们将参数 W 随机初始化，参数 b 初始化为0。
为什么要对神经网络的参数进行随机初始化，感兴趣的同学可以去查相关的资料。

```

from scipy.special import expit

def sigmoid(X):
    return expit(X)

# 测试样例
sigmoid(np.array([-1e56]))
array([0.])

```

接下来完成整个前向传播的函数，也就是 $Z = XW + b$ 和 $\hat{y} = \text{sigmoid}(Z)$

```

def forward(X, W, b):
    """
    完成输入矩阵X到最后激活后的预测值y_pred的计算过程

    Parameters
    -----
    X: np.ndarray, shape = (n, m), 数据，一行一个样本，一列一个特征
    W: np.ndarray, shape = (m, ), 权重
    b: np.ndarray, shape = (1, ), 偏置

    Returns
    -----
    y_pred: np.ndarray, shape = (n, ) '模型对每个样本的预测值

    ...
    # 求Z
    Z = np.dot(X,W)+b                         # YOUR CODE HERE
    # 求激活后的预测值
    y_pred = sigmoid(Z)                        # YOUR CODE HERE
    return y_pred

```

```

# 测试样例
Wt, bt = initialize(trainX.shape[1])
linear_combination(trainX, Wt, bt).shape #(1200,)

(1200,)

```

接下来实现激活函数sigmoid

```

def my_sigmoid(x):
    """
    sigmoid 1 / (1 + exp(-x))

    Parameters
    -----
    X: np.ndarray, 待激活的值

    ...
    # YOUR CODE HERE
    activations = 1/(1+np.exp(-x))

    return activations

# 测试样例
Wt, bt = initialize(trainX.shape[1])
Zt = linear_combination(trainX, Wt, bt)
my_sigmoid(Zt).mean() # 0.49999

```

0.49999999939462925

在实现sigmoid的时候，可能会遇到上溢(overflow)的问题，可以看到sigmoid中有一个指数运算

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

当 x 很大的时候，我们使用 `numpy.exp(x)` 会直接溢出

```

np.exp(1e56)
inf

```

3.3 实验结果

描述实验性能指标

模型训练完成后，还需要在测试集上验证其预测能力，这就需要计算模型的一些性能指标，如MAE和RMSE等。

$$MAE(\hat{y}, y) = \frac{1}{m} \sum_{i=1}^m |\hat{y}_i - y_i| \quad (8)$$

$$RMSE(\hat{y}, y) = \sqrt{\frac{1}{m} \sum_{i=1}^m (\hat{y}_i - y_i)^2} \quad (9)$$

其中， \hat{y} 是模型的预测值， y 是真值， m 是样本数

```
def MAE(y_hat, y):
    # 请你完成MAE的计算过程
    # YOUR CODE HERE
    MAE=(sum(abs(y_hat-y)))/len(y)
    return MAE
```

```
def RMSE(y_hat, y):
    # 请你完成RMSE的计算过程
    # YOUR CODE HERE
    RMSE=((sum((y_hat-y)**2))/len(y))**0.5
    return RMSE
```

(1) 线性模型

精度(Accuracy): 是分类正确的样本数占样本总数的比例。对样例集D，精度计算公式如2所示。

注意: 这里的分类正确的样本数指的不仅是正例分类正确的个数还有反例分类正确的个数。

$$\begin{aligned} acc(f; D) &= \frac{1}{m} \sum_{i=1}^m \mathbb{I}(f(\mathbf{x}_i) = y_i) \\ &= 1 - E(f; D). \end{aligned} \quad (2)$$

对公式(2)的解释：先统计分类正确的样本数，然后除以总的样例集D的个数。

查准率、查全率 (Precision, Recall)

(1) 查准率、查全率出现的原因:

情景一：

错误率和精度虽然常用，但是并不能满足所有任务需求。以西瓜问题为例，假定瓜农拉来一车西瓜，我们用训练好的模型对这些西瓜进行判别，显然，错误率衡量了有多少比例的瓜被判别错误。但是若我们关心的是“挑出的西瓜中有多少比例是好瓜”，或者“所有好瓜中有多少比例被挑了出来”，那么错误率显然就不够用了，这时需要使用其他的性能度量。

情景二：

类似的需求在信息检索、Web 搜索等应用中经常出现，例如在信息检索中，我们经常会关心“检索出的信息中有多少比例是用户感兴趣的”，“用户感兴趣的信息中有多少被检索出来了”。

“查准率”与“查全率”是更为合适于此类需求的性能度量。

(2) 什么是查准率和查全率

对于二分类问题，可将样例根据其真实类别与学习器预测类别的组合划分为真正例(true positive)、假正例(false positive)、真反例(true negative)、假反例(false negative)四种情形，令 TP、FP、TN、FN 分别表示其对应的样例数，则显然有 $TP+FP+TN+FN=\text{样例总数}$ 。分类结果的“混淆矩阵”(confusion matrix)如表 1 所示。

表1：分类结果混淆矩阵

真实情况	预测结果	
	正例	反例
正例	TP(真正例)	FN(假反例)
反例	FP(假正例)	TN(真反例)

查准率(Precision)，又叫准确率，缩写表示用P。查准率是针对我们预测结

果而言的，它表示的是预测为正的样例中有多少是真正的正样例。定义公式如3所示。

$$P = \frac{TP}{TP+FP} \quad (3)$$

查全率(Recall)，又叫召回率，缩写表示用 R。查全率是针对我们原来的样本而言的，它表示的是样本中的正例有多少被预测正确。定义公式如 5 所示。

$$R = \frac{TP}{TP+FN} \quad (5)$$

F1 度量

综合评价指标(F-Measure)是Precision和Recall加权调和平均：

$$F = \frac{(a^2 + 1)P * R}{a^2(P + R)}$$

当参数a=1时，就是最常见的F1了：

$$F1 = \frac{2PR}{P + R}$$

P和R指标有的时候是矛盾的，综合考虑精确率(precision)和召回率(recall)这两个度量值。很容易理解，F1 综合了 P 和 R 的结果，当 F1 较高时则比较说明实验方法比较理想。

(2) 决策树

常见的划分方法有：信息增益，增益率，基尼指数(Gini 指数)

1 信息增益

$$Info(D) = - \sum_{i=1}^m p_i \log_2(p_i)$$

信息熵公式定义：

其中m为分类个数， p_i 为第i个类别的所有样本数量占所有样本的数量比例。这个公式衡量的是带分类样本即整个数据集D的熵。

首先，计算整个数据集的信息熵(数据集的杂乱程度)

信息增益划分的缺点：

偏向具有能分割更分散的数据的属性，例如如果划分的属性为 product_id，那么最终计算出来的信息熵为 0，信息增益最大。但是这种划分是没有意义的。

2 增益率

为了克服信息增益的缺点，采用增益率来作为衡量指标。

思考：既然信息增益有偏向大量值的倾向，那么找到一种方法归一化这种大量值属性的信息增益，使所有的信息增益都处在一个公平的度量环境下就好。

信息增益率的定义：

$$GainRate(A) = \frac{Gain(A)}{SplitInfo_A(D)}$$

其中：

$$SplitInfo_A(D) = Info_A(D)$$

信息增益率的缺点：

信息增益率虽然解决了信息增益的缺点，但是它倾向于产生不平衡的划分，其中一个分区比其他分区小得多。

3 基尼指数

基尼指数要求树是二叉树，衡量的是数据集D的不纯度，基尼指数的定义为：

$$Gini(D) = 1 - \sum_{i=1}^m p_i^2$$

P_i 是 D 中元组属于 C_i 类的概率。为何这里用元组了，以 `income` 为例，`income` 下面的类型有：{low, medium, high} 三个类别。这三个类别产生的元组为：{low, medium, high}, {low, medium}, {low, high}, {medium, high}, {low}, {medium}, {high} 和 {}。这些都是 `income` 属性下面的类别产生的划分元组，不考虑全集{low, medium, high} 和 空集，那么如果一个属性下面有 v 个类别，则有 2^{v-1} 种划分方式。

(3) 神经网络

损失值变化曲线

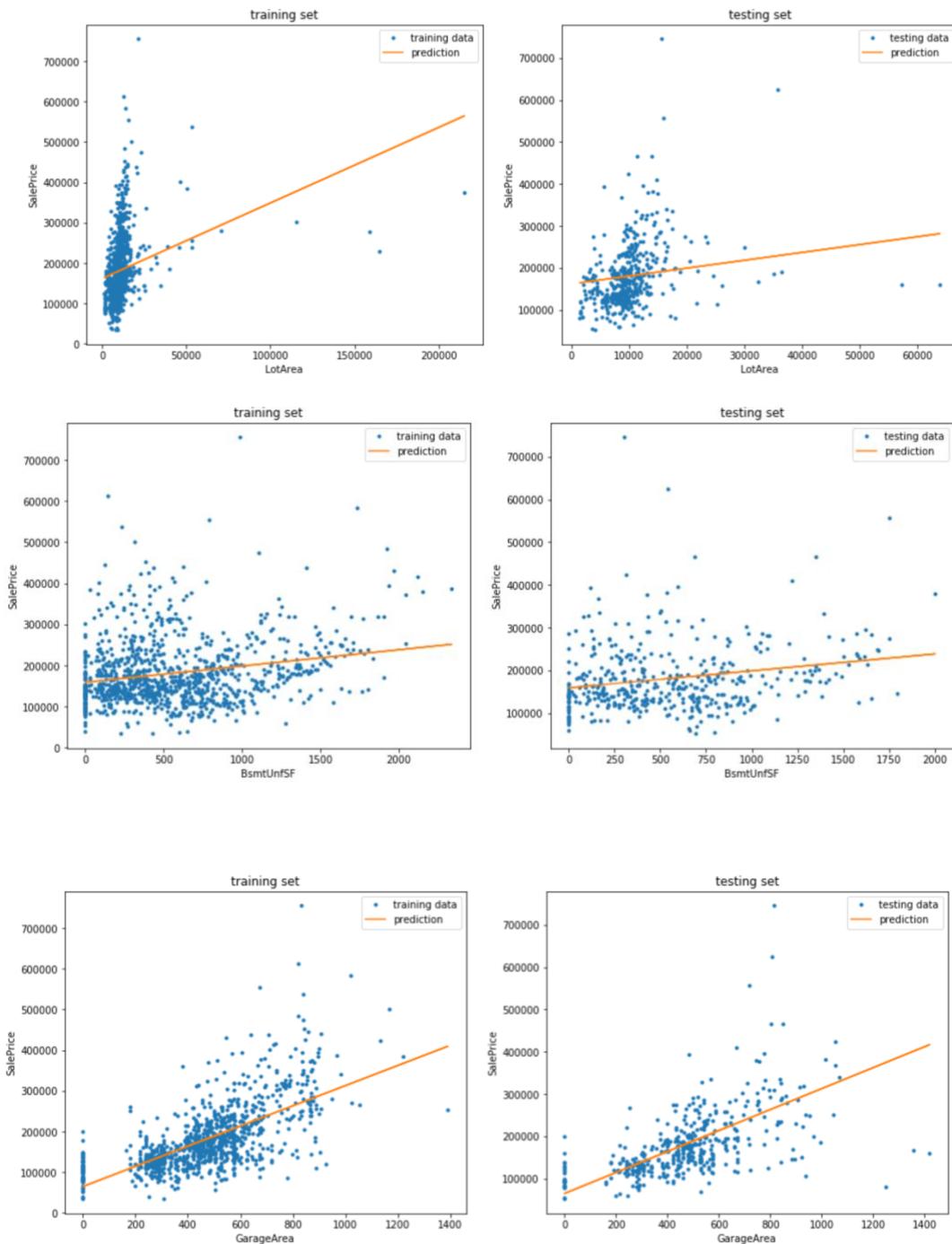
$$\text{loss}(y, \hat{y}) = -\frac{1}{n} [y \log \hat{y} + (1 - y) \log (1 - \hat{y})]$$

4 结果分析

分析实验的结果（实验结果的意义，为什么会出现这样的结果，可以做哪些调整）

(1) 线性模型

第一题



用 lotarea,bsmtufsf,garagearea 为特征，训练模型，可以看见橙色的直线为模型的可视化，即预测结果。

三个模型的指标计算为

模型 1，特征：LotArea

MAE: 53912.821831126574

RMSE: 75795.63312280484

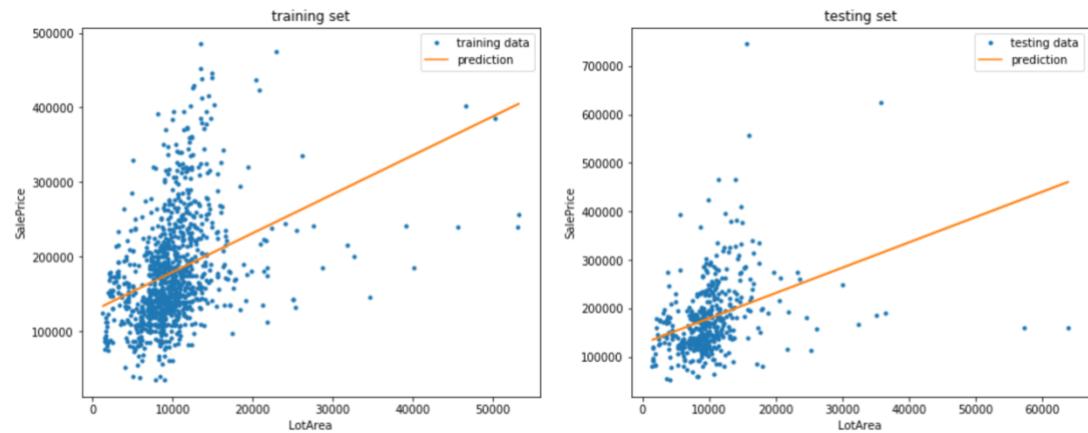
模型 2，特征：BsmtUnfSF

MAE: 53944.93997603965

RMSE: 76895.8671879706

模型 3, 特征: BsmtUnfSF
MAE: 43064.63752598681
RMSE: 65406.08811697361

当对数据进行处理后, 剔除原数据中的过分高或者过分低的部分, 再次进行训练



模型 4, 特征: LotArea 修订
MAE: 52423.413193762746
RMSE: 74965.38186222738

可以看到同样的特征值(组合), 指标上对比模型1有所提升, 这也说明了线性模型中的w和b等参数是根据数据得来的, 不同的数据会有不同的参数反映。

第二题

NO.1
54141.75966870633
115973.59042478741
NO.2
41128.887679848645
63203.98987814105
NO.3
41405.38255279856
65860.1831790994

1. 模型1使用的特征:LotArea, BsmtUnfSF
2. 模型2使用的特征:GarageArea, BsmtUnfSF
3. 模型3使用的特征:LotArea, GarageArea

模型	MAE	RMSE
模型1	54141.76	115973.59
模型2	41128.77	63203.99
模型3	41405.38	65860.18

第三题

数据集	精度	查准率	查全率	F1
spambase	0.92	0.88	0.93	0.90
dota2Results	0.59	0.67	0.60	0.63

对数几率回归在该Spamx, Spamy测试集上的四项指标：

精度： 0.9217816404128191

查准率： 0.8789893617021277

查全率： 0.9257703081232493

f1值： 0.9017735334242838

对数几率回归在该dota2x, dota2y测试集上的四项指标：

精度： 0.5943335132218025

查准率： 0.6680155721749821

查全率： 0.6039177549319256

f1值： 0.6343515906216559

第四题

线性判别分析在该Spamx, Spamy测试集上的四项指标：

精度： 0.8913633894622488

查准率： 0.8071808510638298

查全率： 0.9169184290030211

f1值： 0.8585572842998586

线性判别分析在该dota2x, dota2y测试集上的四项指标：

精度： 0.594657312466271

查准率： 0.6682204692142198

查全率： 0.6042149143121816

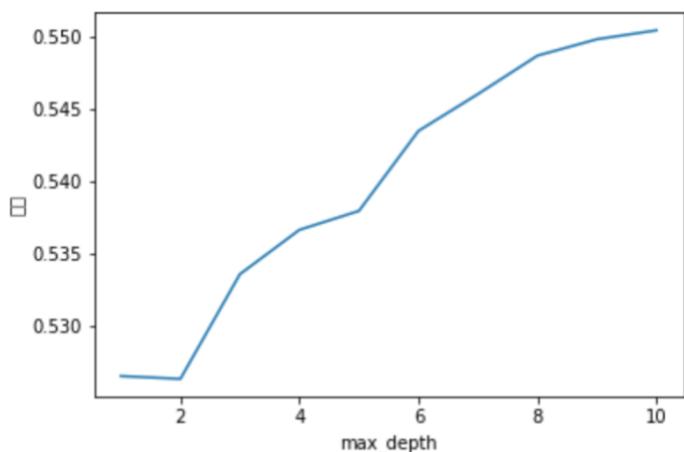
f1值： 0.6346079003697217

双击此处填写

数据集	精度	查准率	查全率	F1
spambase	0.89	0.81	0.92	0.86
dota2Results	0.59	0.67	0.60	0.63

(2) 决策树

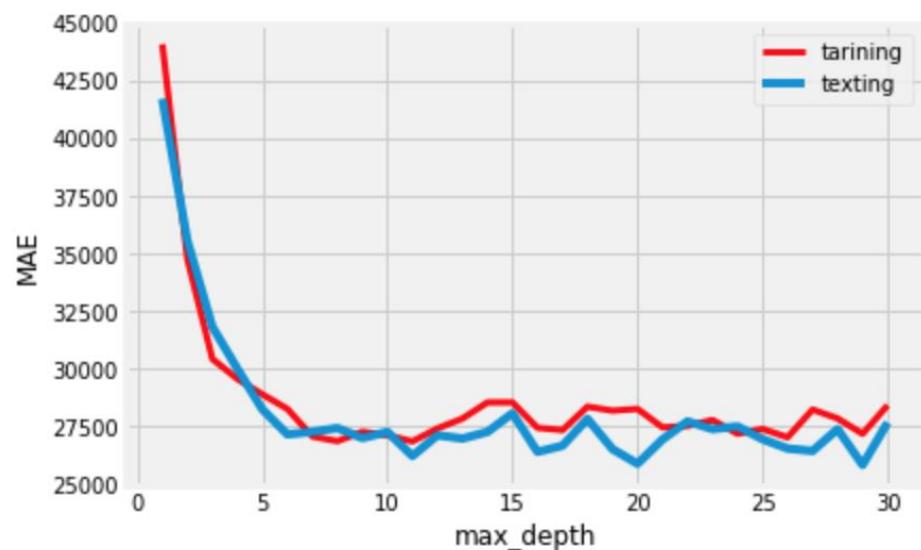
第一题



当层数量增大时，精度逐渐提高。

第二题

MAE	RMSE
27237.086365195002	45544.34235312367



第三题

```
Split on emp_length_5 years = 0
Split on emp_length_< 1 year = 0
Split on home_ownership_OWN = 0
Split on emp_length_4 years = 0
Split on emp_length_9 years = 0
Split on emp_length_8 years = 0
At leaf, predicting 1
```

1

划分标准	精度	查准率	查全率	F1
信息增益 0.8122056154802928	0.8122387390142941	0.9999497928956947	0.8963724740087312	
信息增益率 0.8117774198152642	0.8124897892501225	0.9987699259445212	0.8960508090942874	
基尼指数 0.8122463960198193	0.8122463960198193		1.0 0.8963973086703121	

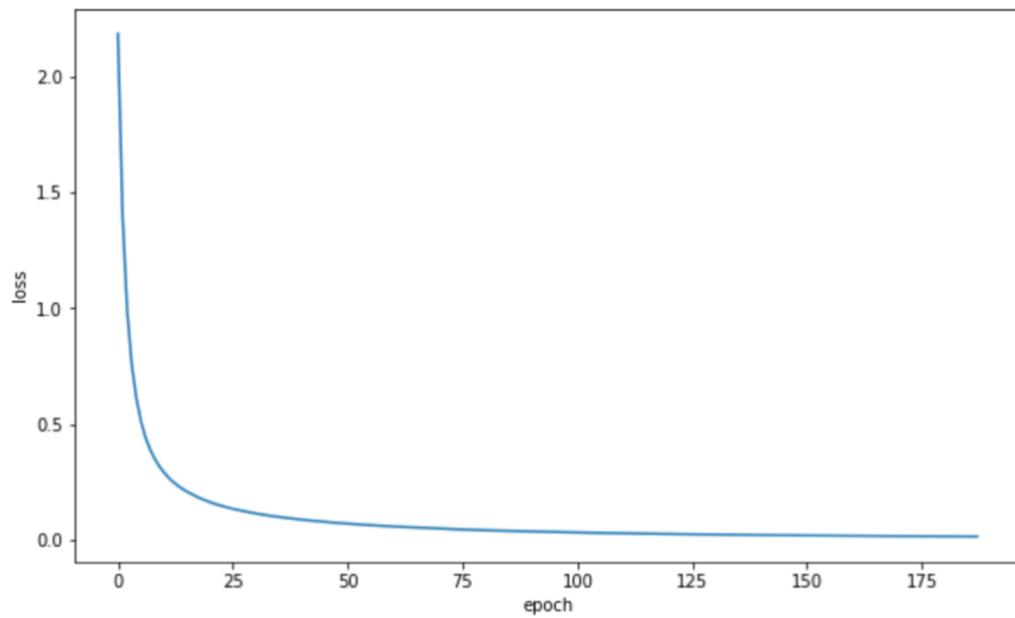
第四题

模型	精度	查准率	查全率	F1
有预剪枝 0.8088247288149417	0.8095024469820554	0.9998992494080903	0.8946834644249622	
无预剪枝 0.8088247288149417	0.8097397556890141	0.9984383658253992	0.8942429164410755	

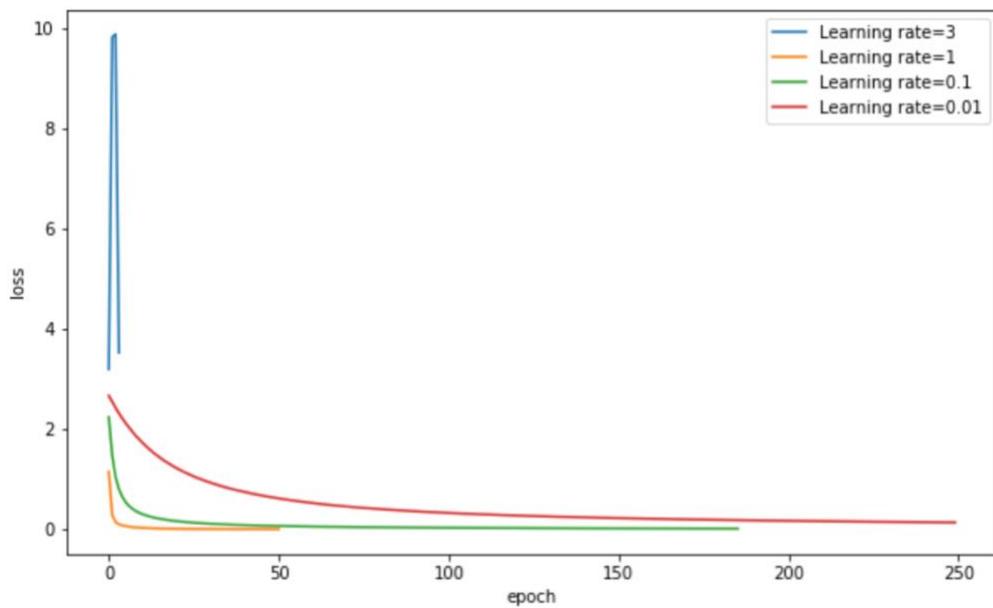
预剪枝对本次训练查准率查全率 f1 值有所提升

(3) 神经网络

第一题

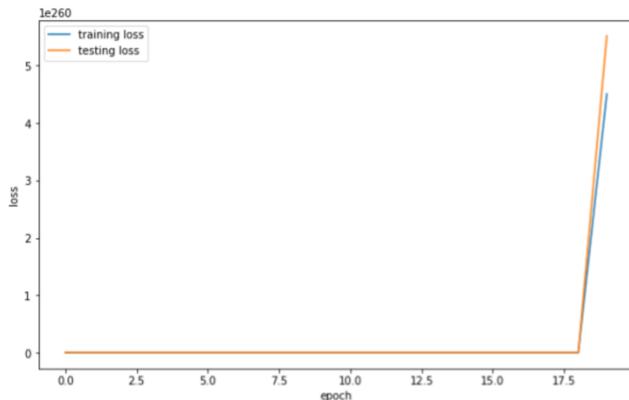


损失函数如上图所示



通过修改不同的学习率，建立不同模型，并可视化，可以发现在一定合理范围内，学习率越高损失函数越快降落并平滑，而超出一定范围，如上图中的蓝色曲线，当学习率过高会产生过拟合，所以损失函数不会像正常的函数快速收敛。

第二题

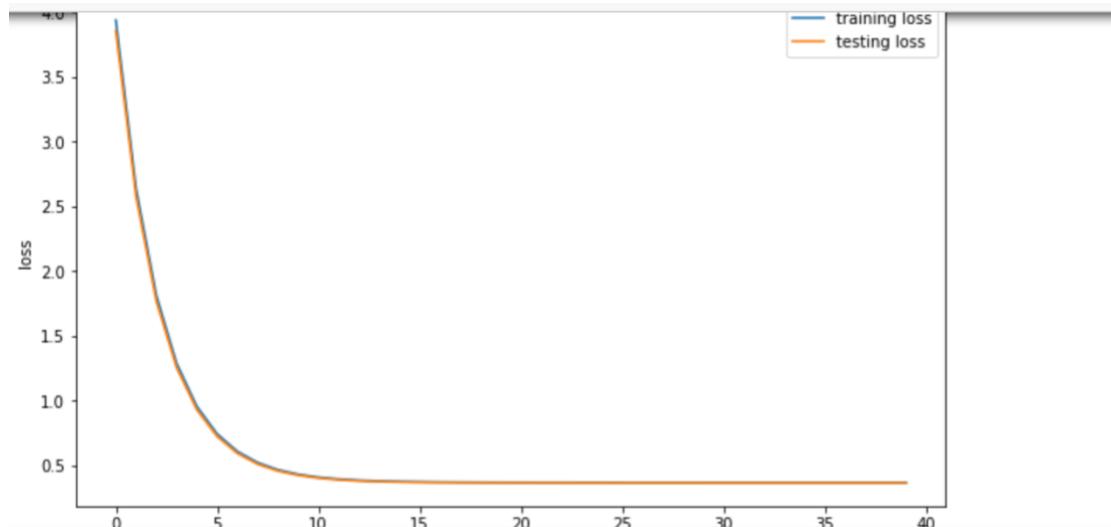


通过打印损失的信息我们可以看到损失值持续上升，这就说明哪里出了问题。但是如果所有的测试样例都通过了，就说明我们的实现是没有问题的。运行下面的测试样例，观察哪里出了问题。

可以看到，我们最开始的参数都是在 10^{-3} 这个数量级上，而第一轮迭代时计算出的梯度的数量级在 10^8 左右，这就导致使用梯度下降更新的时候，让参数变成了 10^6 这个数量级左右（学习率为 0.01）。产生这样的问题的主要原因是：我们的原始数据 X 没有经过适当的处理，直接扔到了神经网络中进行训练，导致在计算梯度时，由于 X 的数量级过大，导致梯度的数量级变大，在参数更新时使得参数的数量级不断上升，导致参数无法收敛。

解决的方法也很简单，对参数进行归一化处理，将其标准化，使均值为 0，缩放到 $[-1, 1]$ 附近。

数据标准化处理以解决问题

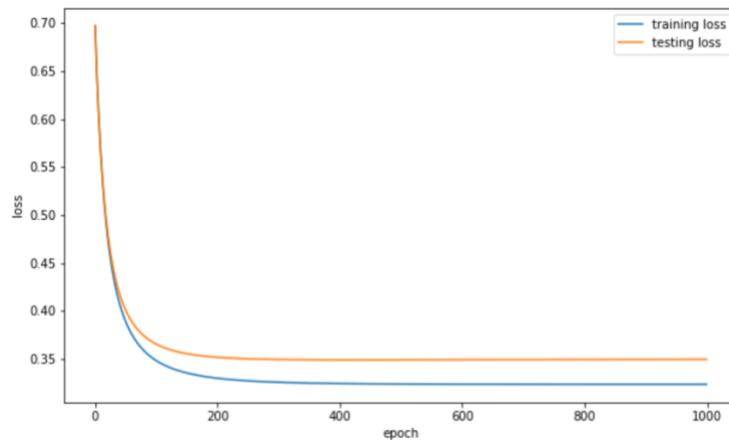
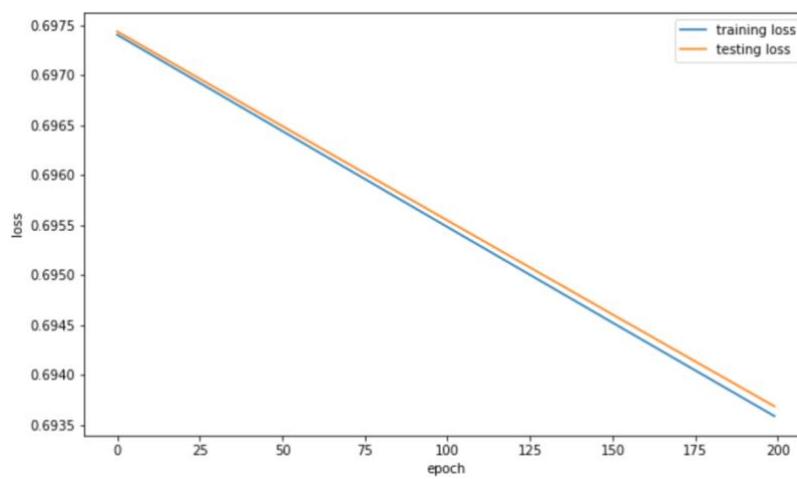
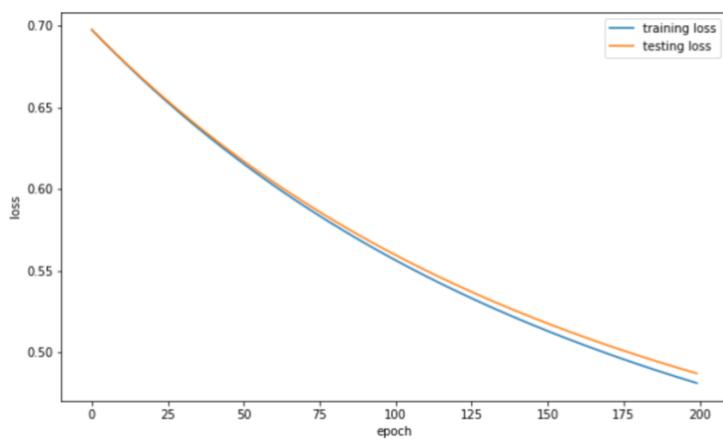


计算测试集上的MSE

```
prediction = forward(testX, parameters)
mse(testY, prediction)
```

60305.85267910155

第三题



三张图的学习率分别为 **0.01, 0.0001, 0.1**

可从损失图像看出前两个模型的学习率不够，第二张图的曲线几乎是线性的，说明学习率过低，当学习率提升到 **0.1** 的时候达到满意的效果。

5 学习总结

详细总结学到的知识点和掌握的技能总结、对机器学习的观点与看法等

在这三章中，我学习了线性回归模型、决策树、神经网络这三种机器学习的模型，它们都是通过数据集的不断喂食训练成长成一定结构的模型，通过学习数据集间属性与最终标记的微妙关系来进行未来数据的预测。

在线性回归模型中，我们学习了如何将很多个特征值组成线性的函数，其中权重或斜率是 w ，它是一个向量，最终还有一个偏置 b 。我们的任务就是将这个线性的模型计算出来， w 和 b 的求解我们利用线性回归模型的最小二乘“参数估计”，对均方误差求导，得出值。另外我们初次接触了 MAE 和 RMSE 这两个计量指标。我们在大多数情况下可以使用 sklearn 中的写好的包来直接使用，在这章我们运用了 LinearRegression 线性回归，LogisticRegression 逻辑回归，LinearDiscriminantAnalysis 线性判别分析。这些类都有对应的 fit 和 predict 方法，调用之后就可以得到预测结果，我们可以将这些结果存入变量中以便进行后续的操作。

在决策树这一章，我们使用 sklearn 中的 DecisionTreeClassifier 类完成有关分类问题的处理，该类中有 max_depth 参数，可以设置决策树的深度，通过修改树的深度可以得到不同的决策树。通过不断调整，可以找到泛化能力最强的一个模型。通过 DecisionTreeRegressor 可以来处理回归问题。我们在决策树的实现过程中，实际上把离散特征转换为向量。决策树中的划分方法有很多，其中我们实现的三种是信息增益、信息增益率、基尼指数。决策树的建立分为两个方法，第一是创建叶子结点，计算出当前叶子结点的标记是什么，并且将叶子结点信息保存在一个 dict 中；第二就是递归的创建决策树了，递归算法终止有三个条件：如果结点内所有的样本的标记都相同，该结点就不需要再继续划分，直接做叶子结点即可。如果结点所有的特征都已经在之前使用过了，在当前结点无剩余特征可供划分样本，该结点直接做叶子结点。如果当前结点的深度已经达到了我们限制的树的最大深度，直接做叶子结点。创建后要完成预测函数，返回的是标记类别。在第四题中我们使用了预剪枝，对于预剪枝来说，实质上是增加了第四个终止条件：如果当前结点划分后，模型的泛化能力没有提升，则不进行划分。如何判断泛化能力有没有提升？我们需要使用验证集。就像使用训练集递归地划分数一样，我们在递归地构造决策树时，也需要递归地将验证集进行划分，计算决策树在验证集上的精度。

神经网络中，我们使用了 MLPClassifier 完成手写数字分类任务，学习梯度下降算法，bp 算法，明白反向传播的原理，它是如何调整权重的。我们还编写了一个三层感知机，它分为输入层，隐藏层，输出层。通过练习，我们对神经网络更加熟悉。

