# 机器学习基础 实验一 实验报告

[机器学习基础 实验一 实验报告](#机器学习基础-实验一-实验报告)  
 [实验目的](#实验目的)  
 [实验要求](#实验要求)  
 [实验过程](#实验过程)  
 [1.数据简易分析](#X535ca1b0fdc5c265feca0225661958a2a6ce8b1)  
 [2.数据填充](#X36aa5433931e77c37f55c9d9570ab3245b20c9b)  
 [3.数据转换](#Xe19d2c6c15b4c11a9098db2eace6c3304d9d2e9)  
 [3.1 文本类型转换为数值类型](#X324663da90bc77f7b299d9346e89e84f9e53791)  
 [3.2 简单日期类型转换为数值类型](#Xfb938e7330118746f0ef83df513fae447708166)  
 [3.3 繁琐的日期类型转换为数值类型](#X0d8977bf7c68ef5eb500e7d8da2e054d872afb7)  
 [4.数据删减](#X7345d0a9ba8f44185c59d87757b5dc74104237d)  
 [5. 数据分箱](#Xeacf1f8df04837ae4cec9825282de3f85c18ade)  
 [6.数据处理器代码实现](#X269eec7c1075df5f5777c6837afba52f34bf714)  
 [7.贝叶斯决策器实现](#X8f7b7a218d4f0eafbe06d682e812f455aa0a076)  
 [7.1 理论简述与简化](#X54d7af1dac7db313a04e32365cb6255dbcfdb77)  
 [7.2 拉普拉斯平滑](#X22844fec1a3440b79f0658ba62f834409793a51)  
 [7.3 完整贝叶斯决策器代码实现](#X58813d0dd02e4327a6c4882cded93c8a3ea6a66)  
 [8.实验执行](#Xb23c2098c775d56cff50caea14a9d97fac25a51)  
 [9.优化调整](#Xf10a8c4557cfa4b8324a22296bac30b4f494d67)  
 [9.1 随机搜索](#X2a202fae9c3da5daeeb87e88b638c5710aa6b92)  
 [9.2 模拟退火搜索](#Xcce22a0c78707857c231dcf9c3293ef226df984)  
 [9.3 模拟退火+K-Fold(k=5)](#X7259f1b8cb6b6a6efef53e4e057eb0ff4250222)  
 [10.代码目录结构](#X06b88f23f8a93f499475e23b0ec4af1af31618f)  
 [心得体会](#心得体会)

## 实验目的

本实验以贷款违约为背景，要求使用贝叶斯决策论的相关知识在训练集上构建模型，在测试集上进行贷款违约预测并计算分类准确度。

## 实验要求

1. 实验不限制使用何种高级语言，推荐使用python中pandas库处理csv文件。
   * 安装：pip install pandas/conda install pandas【在使用conda命令，需安装anaconda环境】
   * 导入：import pandas as pd【建议】
2. 在进行贝叶斯分类之前重点是对数据进行预处理操作，如，缺失值的填充、将文字表述转为数值型、日期处理格式（处理成“年-月-日”三列属性或者以最早时间为基准计算差值）、无关属性的删除、多列数据融合等方面。
3. 数据中存在大量连续值的属性，不能直接计算似然，需要将连续属性离散化。
4. 另外，特别注意零概率问题，贝叶斯算法中如果乘以0的话就会失去意义，需要使用平滑技术。【可以百度了解一下拉普拉斯平滑】
5. 实验目的是使用贝叶斯处理实际问题，不得使用现成工具包直接进行分类。【该点切记！！！这个一定要自己写，才能感受贝叶斯的魅力】
6. 实验代码中需要有必要的注释，具有良好的可读性。

## 实验过程

### 1.数据简易分析

在进行贝叶斯决策之前，我应当先对训练数据进行一个整体的分析。对此我编写了如下代码来粗略检测训练数据（train.csv）每个特征的基本情况，即缺失性，数值类型，特征值个数等信息。得到如下的表格。

'''  
DataAnaslyze.py (Part1)  
'''  
  
import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
  
dataAnasly = {}  
  
def checkColumn(columns):  
 for column in columns:  
 dataAnasly[column]={}  
 data = df[column]  
 values\_counts = data.value\_counts()  
 dataAnasly[column]['count'] = len(values\_counts)  
  
 if(len(values\_counts) < 30):  
 unique\_values = df[column].drop\_duplicates().tolist()  
 dataAnasly[column]['unique'] = unique\_values  
 else:  
 dataAnasly[column]['unique'] = []  
  
 dtype = df[column].dtype  
 dataAnasly[column]['dtype'] = dtype  
  
 has\_null = df[column].isnull().any()  
 dataAnasly[column]['has\_null'] = has\_null  
  
 analysis\_df = pd.DataFrame(dataAnasly).T  
 analysis\_df.to\_excel('../data/数据分析器.xlsx')  
  
train\_data\_path = '../data/train.csv'  
df = pd.read\_csv(train\_data\_path)  
checkColumn(df.columns.tolist())

执行这部分代码，可以得到 数据分析器.xlsx 文件，点击查看后可以获得如下表格来对特征进行分析。在unique一栏中，仅展示了取值种类小于等于30个的种类的取值。随后的type 一栏中，与数值有关的类型均为 object，其余已经被成功分成int64和float64两类。

| columnName | count | unique | type | has\_null |
| --- | --- | --- | --- | --- |
| loan\_id | 9000 | [] | int64 | FALSE |
| user\_id | 9000 | [] | int64 | FALSE |
| total\_loan | 1526 | [] | float64 | FALSE |
| year\_of\_loan | 2 | [3, 5] | int64 | FALSE |
| interest | 1001 | [] | float64 | FALSE |
| monthly\_payment | 5872 | [] | float64 | FALSE |
| class | 7 | ['B', 'C', 'A', 'G', 'D', 'E', 'F'] | **object** | FALSE |
| employer\_type | 6 | ['幼教与中小学校', '政府机构', '上市企业', '普通企业', '世界五百强', '高等教育机构'] | **object** | FALSE |
| industry | 14 | ['金融业', '国际组织', '文化和体育业', '建筑业', '电力、热力生产供应业', '批发和零售业', '采矿业', '房地产业', '交通运输、仓储和邮政业', '公共服务、社会组织', '住宿和餐饮业', '信息传输、软件和信息技术服务业', '农、林、牧、渔业', '制造业'] | **object** | FALSE |
| work\_year | 11 | [nan, '8 years', '10+ years', '3 years', '7 years', '4 years', '1 year', '< 1 year', '6 years', '2 years', '9 years', '5 years'] | **object** | **TRUE** |
| house\_exist | 5 | [2, 1, 0, 3, 4] | int64 | FALSE |
| censor\_status | 3 | [2, 0, 1] | int64 | FALSE |
| issue\_date | 126 | [] | **object** | FALSE |
| use | 14 | [0, 4, 2, 9, 5, 3, 6, 1, 7, 8, 10, 12, 11, 13] | int64 | FALSE |
| post\_code | 786 | [] | int64 | FALSE |
| region | 50 | [] | int64 | FALSE |
| debt\_loan\_ratio | 5822 | [] | float64 | FALSE |
| del\_in\_18month | 12 | [0, 3, 1, 4, 2, 9, 5, 6, 7, 8, 10, 15] | int64 | FALSE |
| scoring\_low | 132 | [] | float64 | FALSE |
| scoring\_high | 360 | [] | float64 | FALSE |
| known\_outstanding\_loan | 47 | [] | int64 | FALSE |
| known\_dero | 13 | [1, 0, 2, 3, 8, 10, 4, 9, 5, 6, 11, 12, 7] | int64 | FALSE |
| pub\_dero\_bankrup | 5 | [1.0, 0.0, 3.0, 2.0, nan, 5.0] | float64 | **TRUE** |
| recircle\_b | 8558 | [] | float64 | FALSE |
| recircle\_u | 2991 | [] | float64 | FALSE |
| initial\_list\_status | 2 | [1, 0] | int64 | FALSE |
| app\_type | 2 | [0, 1] | int64 | FALSE |
| earlies\_credit\_mon | 520 | [] | **object** | FALSE |
| title | 843 | [] | int64 | FALSE |
| policy\_code | 1 | [1] | int64 | FALSE |
| f0 | 29 | [3.0, 6.0, 4.0, 16.0, 5.0, 12.0, 0.0, 8.0, 14.0, 2.0, nan, 11.0, 10.0, 18.0, 26.0, 1.0, 7.0, 9.0, 13.0, 15.0, 20.0, 17.0, 19.0, 23.0, 21.0, 22.0, 33.0, 24.0, 25.0, 28.0] | float64 | **TRUE** |
| f1 | 2 | [0.0, nan, 1.0] | float64 | **TRUE** |
| f2 | 58 | [] | float64 | **TRUE** |
| f3 | 62 | [] | float64 | **TRUE** |
| f4 | 39 | [] | float64 | **TRUE** |
| early\_return | 6 | [0, 3, 1, 2, 5, 4] | int64 | FALSE |
| early\_return\_amount | 3967 | [] | int64 | FALSE |
| early\_return\_amount\_3mon | 3484 | [] | float64 | FALSE |
| isDefault | 2 | [0, 1] | int64 | FALSE |

基于如上的数据表格，与实验指导书所描述，isDefault 为y。而剩余的其他特征为X。

基于如上的数据表格，部分属性存在缺失，在进行贝叶斯预测前需要对空白数据进行填充，由于不确定测试数据是否和训练数据保持一样的空值分布，因此需要进行填充，在第二部分会阐述这一部分。

基于如上的数据表格，部分数据并非为数值型，例如文本描述和日期格式等纯文本内容，需要通过某种映射方式来对文本型数据进行转换。此处假设测试数据中不存在训练数据中不存在的特征值，在第三部分会阐述这一部分。

基于如上的数据表格，部分数据之间的关联性较高，并且部分数据具有唯一性或者独立性，这种数据都可以删除。对各列之间进行线性相关性判定，此处假设两个列如果是强相关，则删去其中一列即可，在第四部分会阐述这一部分。

基于如上的数据表格，部分数据为数值较多的连续值。对于数值较少的连续值在计算似然时可以简单的计算，而对于数值较多的连续值计算似然比较复杂，因此需要对数据进行分箱处理，本次实验将会先筛选需要分箱的数据，并且先进行简单的分箱，随后再对各分箱的箱数进行调整来提高判别的准确率（虽然可能意义不大），在第五部分会阐述相关内容。

对于整个数据处理的过程，我对其进行了包装，将其包装成为一个数据处理器的类（DataProcess）来实现测试数据和训练数据的数据处理一致性。该类中各个处理部分采用流水线工作，处理部分会执行如上的四个部分，即填充、转换、删除、分箱。

### 2.数据填充

在通常情况下数据填充存在多种填充方式，比如众数填充、中位数填充、平均值填充、指定值填充。为了更好的编辑和调试代码，我设计该类在创建时会传入一个fillNanRules 列表来规范约束数据填充的过程。在该类中实现fillNanMethod 方法既可以根据初始化的数据填充规则进行填充。

以下为数据填充的基本方法。0代表填充众数、1代表填充中位数、2代表平均数、3为填充具体值。默认采用众数填充。

def \_\_init\_\_(self, dataFrame,segmentMap, fillNanRules={}):  
 '''  
 :param segmentMap: 分段分箱列表  
 :param fillNanRules: 填充规则 name:[method,defaultValue]这样的字典  
 '''  
 self.data = dataFrame  
 self.fillNanRules = fillNanRules  
 # ... 其他内容  
   
 def \_\_fillNan(self, column, typ=0, default\_value=0):  
 '''  
 :param typ: 0 为众数、1为中位数、2为平均数、3为填充为default\_value值  
 '''  
 if typ not in [0, 1, 2, 3]:  
 raise ValueError("type must be 0, 1, 2, or 3.")  
 if typ == 0:  
 column.fillna(column.mode()[0], inplace=True)  
 if typ == 1:  
 column.fillna(column.median(), inplace=True)  
 if typ == 2:  
 column.fillna(column.mean(), inplace=True)  
 if typ == 3:  
 column.fillna(default\_value, inplace=True)  
 return column  
   
 def fillNanMethod(self, data):  
 # 默认均采用众数来填充所有缺失值  
 for columnName in data.columns:  
 if columnName in self.fillNanRules:  
 if self.fillNanRules[columnName][0] == 3:  
 data[columnName] = self.\_\_fillNan(data[columnName], self.fillNanRules[columnName][0],  
 self.fillNanRules[columnName][1])  
 else:  
 data[columnName] = self.\_\_fillNan(data[columnName], self.fillNanRules[columnName][0])  
 else:  
 data[columnName] = self.\_\_fillNan(data[columnName])  
 return data

### 3.数据转换

通过对上述数据表格的查阅，可以得知如下表的需要进行数据转换的特征值列表，将对其逐步进行处理。

| columnName | count | unique | type | has\_null |
| --- | --- | --- | --- | --- |
| class | 7 | ['B', 'C', 'A', 'G', 'D', 'E', 'F'] | object | FALSE |
| employer\_type | 6 | ['幼教与中小学校', '政府机构', '上市企业', '普通企业', '世界五百强', '高等教育机构'] | object | FALSE |
| industry | 14 | ['金融业', '国际组织', '文化和体育业', '建筑业', '电力、热力生产供应业', '批发和零售业', '采矿业', '房地产业', '交通运输、仓储和邮政业', '公共服务、社会组织', '住宿和餐饮业', '信息传输、软件和信息技术服务业', '农、林、牧、渔业', '制造业'] | object | FALSE |
| work\_year | 11 | [nan, '8 years', '10+ years', '3 years', '7 years', '4 years', '1 year', '< 1 year', '6 years', '2 years', '9 years', '5 years'] | object | TRUE |
| issue\_date | 126 | []（日期格式 yyyy/mm/dd） | object | FALSE |
| earlies\_credit\_mon | 520 | [] （日期格式，其中存在一个英文月份缩写和一些数字以及其他字符） | object | FALSE |

该过程的整体调用函数如下：随后将分类型来阐述该过程。

def dealStringMethod(self, data):  
 # 第一步处理 class 属性  
 data['class'] = self.\_\_dealString(data['class'], self.Mapping['class'])  
 # 第二步处理 employer\_type 属性  
 data['employer\_type'] = self.\_\_dealString(data['employer\_type'], self.Mapping['employer\_type'])  
 # 第三步处理 industry 属性  
 data['industry'] = self.\_\_dealString(data['industry'], self.Mapping['industry'])  
 # 第四步处理 work\_year 属性  
 data['work\_year'] = self.\_\_dealString(data['work\_year'], self.Mapping['work\_year'])  
   
 # 第五步处理 issue\_date 属性，该属性的含义 贷款发放的月份,因此只用截至到年份  
 data['issue\_date'] = self.\_\_dealDate2Month(data['issue\_date'])  
 # 第六步处理 earlies\_credit\_mon 属性，该属性约束到月份  
 data['earlies\_credit\_mon'] = self.\_\_dealString2Month(data['earlies\_credit\_mon'])  
 return data

#### 3.1 文本类型转换为数值类型

对于表格中的前四行，其特点均为一个纯字符串类型的文本，可以通过写一个映射关系来对文本转换成数值。Map映射如下代码所示，在定义好Map映射后可以通过定义好的函数来执行\_\_dealString，来对数据进行填充。整体实现过程如下：

Mapping = {'class': {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6},  
 'employer\_type': {'幼教与中小学校': 0, '政府机构': 1, '上市企业': 2, '普通企业': 3, '世界五百强': 4,  
 '高等教育机构': 5},  
 'industry': {'金融业': 0, '国际组织': 1, '文化和体育业': 2, '建筑业': 3, '电力、热力生产供应业': 4,  
 '批发和零售业': 5, '采矿业': 6,  
 '房地产业': 7, '交通运输、仓储和邮政业': 8, '公共服务、社会组织': 9, '住宿和餐饮业': 10,  
 '信息传输、软件和信息技术服务业': 11, '农、林、牧、渔业': 12, '制造业': 13},  
 'work\_year': {'8 years': 8, '10+ years': 10, '3 years': 3, '7 years': 7, '4 years': 4, '1 year': 1,  
 '< 1 year': 0, '6 years': 6, '2 years': 2, '9 years': 9, '5 years': 5}}  
  
 @staticmethod  
 def \_\_dealString(column, mapping):  
 values = column.values.tolist()  
 for value in values:  
 if value not in mapping:  
 raise ValueError(f"Value '{value}' is not in {list(mapping.keys())}")  
 column = column.map(mapping)  
 return column

#### 3.2 简单日期类型转换为数值类型

对于表格中的第五行 issue\_date ，其中的数值类型为 yyyy/mm/dd ，通过实验指导书中 issue\_date 为贷款发放的月份，因此只需要统计到月份的差值即可。利用python内部自带的处理方法即可。计算1970年1月到现在的月份数，即可将该列转换成数值类型。

def \_\_dealDate2Month(self, column):  
 column = pd.to\_datetime(column, format='%Y/%m/%d')  
 column = column.apply(lambda x: self.\_\_calcMon2Target(x, 1970, 1))  
 return column

#### 3.3 繁琐的日期类型转换为数值类型

对于表格中的第六行数据 earlies\_credit\_mon ，其内部的含义为借款人最早报告的信用额度开立的月份。可以发现该类型的数据构造为一个英文月份缩写和一些数字以及其他字符，可以通过正则表达式的方法将该月份提取处理，并进行英文缩写和月份的转换，即可获得对应的月份的数值。

def \_\_dealString2Month(self, column):  
 column = column.apply(lambda x: self.matchMonth(x))  
 return column  
  
 @staticmethod  
 def matchMonth(text):  
 pattern = re.compile(r'[A-Za-z]+')  
 matches = pattern.findall(text)  
 try:  
 month = datetime.strptime(matches[0], "%b").month  
 except ValueError:  
 raise ValueError(f"Value '{matches[0]}' is a valid month")  
 return month

至此，六个不为数值型的特征都已经被处理好了，可以对数据进行进一步的分析与处理。

### 4.数据删减

根据上述表格中的数据先进行唯一性（可能取值仅有一个）或者独立性（没个数据的值均一样）判断，可以得到下表：

| columnName | count | unique | dtype | has\_null |
| --- | --- | --- | --- | --- |
| loan\_id | 9000 | [] | int64 | FALSE |
| user\_id | 9000 | [] | int64 | FALSE |
| policy\_code | 1 | [1] | int64 | FALSE |

在本实验中，假定各个特征是基本独立的，公式如下：

那么在这种情况下，每一个不同的数据取同一个值或者均取不同的值，对最后的整体答案没有任何区别，因此为了计算的便捷，删去这三行。

贝叶斯模型通常涉及到概率分布的估计和使用先验知识。如果两个特征强相关，它们可能会在概率估计中引入冗余，这可能导致先验或似然估计不准确。因此，在一些情况下，移除一个特征可能有助于更准确地估计模型参数。

高度相关的特征增加了数据的维度但没有相应增加信息量，这可能使模型更难以从数据中学习，并增加过拟合的风险。减少特征的数量可以帮助缓解维度的诅咒，特别是在数据点较少的情况下。

贝叶斯模型的计算通常比较复杂和昂贵。减少特征数量可以降低计算负担，提高模型训练和预测的效率。

基于以上三点我对整个训练集 X 中的数据进行任意两列的相关性分析，设计如下的代码生成其各列之间的相关性矩阵（为了便于处理已经对其全部进行绝对值处理）。

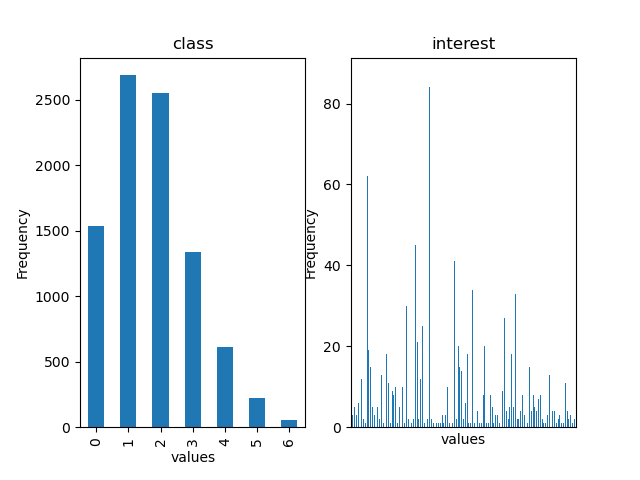
correlation\_matrix = df.corr()  
correlation\_matrix = correlation\_matrix.abs()  
correlation\_matrix.to\_excel('../data/相关系数矩阵.xlsx', sheet\_name='Correlation Matrix')

得到的结果在 Exp1/data/相关系数矩阵.xlsx 之中，在该相关系数矩阵的基础上，分析任意两列之间的相关性，编写如下代码检查其相关系数的绝对值大于0.75的列组合。利用如下代码，得到如下表。

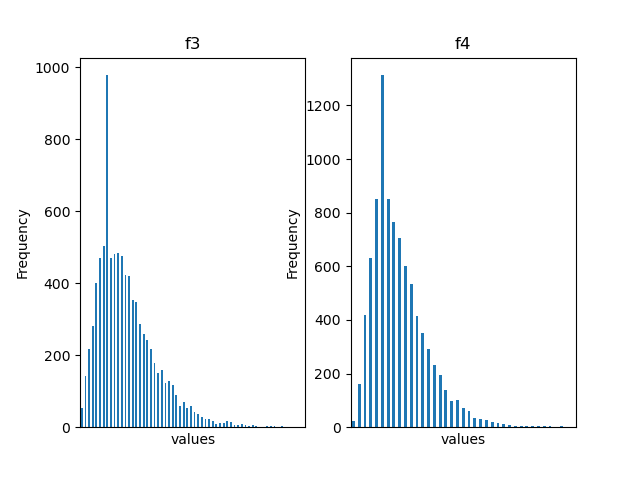
def check\_correlation\_matrix():  
 correlation\_matrix = df.corr().abs()  
  
 correlation\_matrix\_absolute = correlation\_matrix.abs()  
  
 # 获取上三角矩阵，忽略对角线  
 upper\_tri = np.triu(np.ones(correlation\_matrix\_absolute.shape))  
  
 # 从上三角矩阵中筛选出所有大于等于0.7并且小于1的元素的位置  
 to\_select = (correlation\_matrix\_absolute >= 0.7) & (correlation\_matrix\_absolute < 1) & upper\_tri  
  
 # 获取满足条件的行名与列名的组合  
 selected\_correlations = correlation\_matrix\_absolute.where(to\_select).stack()  
  
 print("Selected Column Pairs with Correlation >= 0.7 and < 1 and Their Correlations:")  
 for (pair, correlation) in selected\_correlations.items():  
 print(f"{pair}: {correlation}")  
  
check\_correlation\_matrix()

| 列1 | 列2 | 相关系数绝对值(4位有效数字) | 思考 |
| --- | --- | --- | --- |
| interest | class | 0.9254 | interest 表示当前贷款利率为连续的，而class 表示贷款级别为离散的，从现实生活中考虑这两者之间理应存在某些联系，并且在interest 进行分箱后的效果不一定比直接采用class 这种离散值方便，可以绘制两者的直方图进行比较，将interest 进行细分后大致分布形式应与class 相似，本次实验中将interest 删除。直方图如下图。 |
| f3 | f4 | 0.8438 | 这两个属性均为为一些贷款人行为计数特征的处理，两个值不为离散的，我在本实验中选取数据较多的f3，删除f4,两者的直方图如下。 |
| early\_return\_amount | early\_return\_amount\_3mon | 0.7530 | early\_return\_amount贷款人提前还款累积金额，early\_return\_amount\_3mon近3个月内提前还款金额，两个值不为离散的，我在本实验中选取数据较多的early\_return\_amount\_3mon，删除early\_return\_amount，数据极度偏态，不再展示直方图。 |
| scoring\_low | scoring\_high | 0.8890 | scoring\_low 表示借款人在贷款评分中所属的下限范围，scoring\_high 表示借款人在贷款评分中所属的上限范围，两个值不为离散的，我在本实验中选取数据较多的scoring\_low，删除scoring\_high,两者的直方图如下。 |
| total\_loan | monthly\_payment | 0.9256 | total\_loan 表示贷款数额，monthly\_payment 表示分期付款金额，两个值不为离散的，我在本实验中选取数据较多的scoring\_low，删除scoring\_high,两者的直方图如下。并且考虑实际情况下total\_loan会决定monthly\_payment，并且在日常生活中似乎后者更会影响借款人是否还贷。 |

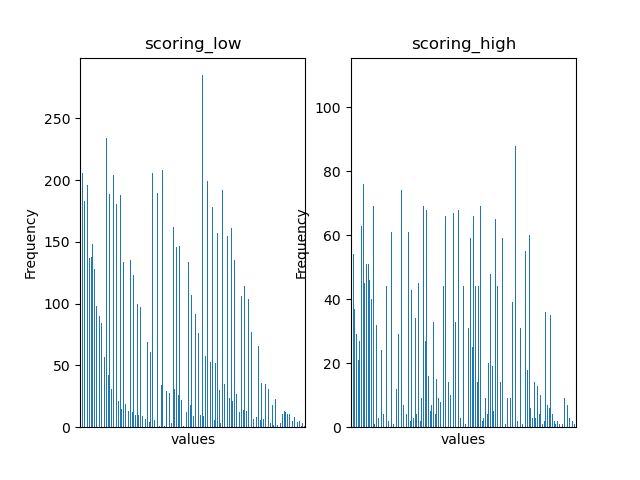
interest 和 class 的分布直方图：



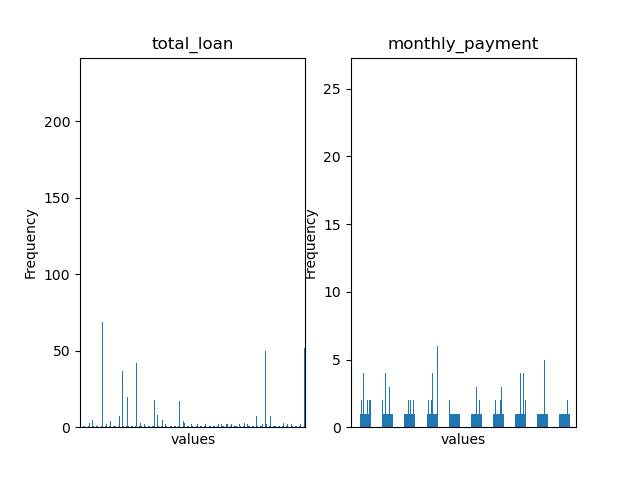
f3 和 f4 的分布直方图：



scoring\_low 和 scoring\_high分布直方图：



total\_loan 和 monthly\_payment 分布直方图：



### 5. 数据分箱

数据分箱可以简化数据结构，使模型更容易理解和实现。通过将连续的数据值分组到有限数量的“箱”或“区间”中，可以减少数据的复杂性，从而降低模型的过拟合风险。

分箱可以提高模型对于异常值和噪声数据的鲁棒性。原始数据中的小波动可能不会导致分箱后的数据发生变化，因此模型对于输入数据的小变动不太敏感。

在贝叶斯决策和概率估计中，处理较少的分类可以简化概率计算。分箱后，每个箱中的观测值可以被用来估计该区间的条件概率，从而在贝叶斯决策中提供有用的先验和似然信息。

对于有很多数值特征的数据集，分箱可以有效减少模型需要处理的数据维度。这不仅可以提高计算效率，还可以在一定程度上避免维度灾难。

通常利用经验对数据进行分箱的方法如下：

* **平方根法**：将分箱数设为数据点总数的平方根。这是一种简单且广泛使用的方法，特别是在缺乏其他信息时。
* **Sturges' 规则**：这个公式是 ，其中 是样本数量。这个规则基于数据分布是近似正态的假设，且目的是尽可能减少在估计概率分布时的总方差。
* **Rice 规则**：提议使用 作为分箱个数，这个方法试图在不增加过多箱子的前提下，提供足够的细致划分。

在这里为了便于分箱，我先初步采用 Sturges 规则，分箱数 。对于取值大于50种的特征进行分箱，而对于特征取值数小于50的特征保留原有形态，来保证部分数据的原始性，防止分箱每个箱子内数据过多。

通过定义不同名称的分箱数，并同一调用函数进行分箱，在训练过程种将分箱边界保存起来，以便后续处理测试数据，整体代码如下：

segmentMap = {  
 'post\_code': 14,  
 'title': 14,  
 'known\_outstanding\_loan': 14,  
 'monthly\_payment': 14,  
 'issue\_date': 14,  
 'debt\_loan\_ratio': 14,  
 'scoring\_high': 14,  
 'recircle\_b': 14,  
 'recircle\_u': 14,  
 'f0': 14,  
 'f2': 14,  
 'f3': 14,  
 'early\_return\_amount': 14,  
 }  
   
  
 def dealSegment(self, data, typ='train'):  
 for columnName in data.columns:  
 data[columnName] = self.\_\_dealDataSegment(data[columnName], columnName, typ=typ)  
 return data  
   
 def \_\_dealDataSegment(self, column, column\_name, typ='train'):  
 if column\_name in self.Segment:  
 column = pd.cut(column, bins=self.Segment[column\_name], labels=range(len(self.Segment[column\_name]) - 1),  
 include\_lowest=True)  
 return column  
 if typ == 'test':  
 return column  
 if column\_name not in self.segmentMap:  
 return column  
  
 num = self.segmentMap[column\_name]  
 bins = np.linspace(column.min(), column.max(), num + 1)  
 column = pd.cut(column, bins=bins, labels=range(num),  
 include\_lowest=True)  
 self.Segment[column\_name] = bins  
 return column

在此先进行简单的分箱处理，整个预测模型构建好之后再对分箱数进行优化细分。

### 6.数据处理器代码实现

'''  
Exp1/code/DataProcess.py  
'''  
  
  
import re  
import pandas as pd  
import matplotlib.pyplot as plt  
import numpy as np  
from datetime import datetime  
from dateutil.relativedelta import relativedelta  
  
  
class DataProcessor:  
 def \_\_init\_\_(self, dataFrame,segmentMap, fillNanRules={}):  
 '''  
 :param segmentMap: 分段分箱列表  
 :param fillNanRules: 填充规则 name:[method,defaultValue]这样的字典  
 '''  
 self.data = dataFrame  
 self.fillNanRules = fillNanRules  
 self.isProcessed = False  
 self.segmentMap = segmentMap  
  
 def \_\_fillNan(self, column, typ=0, default\_value=0):  
 '''  
 :param typ: 0 为众数、1为中位数、2为平均数、3为填充为default\_value值  
 '''  
 if typ not in [0, 1, 2, 3]:  
 raise ValueError("type must be 0, 1, 2, or 3.")  
 if typ == 0:  
 column.fillna(column.mode()[0], inplace=True)  
 if typ == 1:  
 column.fillna(column.median(), inplace=True)  
 if typ == 2:  
 column.fillna(column.mean(), inplace=True)  
 if typ == 3:  
 column.fillna(default\_value, inplace=True)  
 return column  
  
 @staticmethod  
 def dropColumn(data):  
 data = data.drop(['isDefault'], axis=1)  
  
 # 删除若干列,从训练集和测试集可以初步看出  
 # loan\_id和user\_id是基本不重复的，因此可以删去。当前数据下policy\_code下均为1，也可删去。  
 data = data.drop(['loan\_id'], axis=1)  
 data = data.drop(['user\_id'], axis=1)  
 data = data.drop(['policy\_code'], axis=1)  
  
 # interest 与 class的强相关  
 # ('interest', 'class'): 0.9254597054479434  
 data = data.drop(['interest'], axis=1)  
  
 # ('f3', 'f4'): 0.8438089877232243  
 data = data.drop(['f4'], axis=1)  
  
 # ('early\_return\_amount', 'early\_return\_amount\_3mon'): 0.7530913899047247  
 data = data.drop(['early\_return\_amount'], axis=1)  
  
 # ('scoring\_low', 'scoring\_high'): 0.8890661841570701  
 data = data.drop(['scoring\_low'], axis=1)  
  
 # ('total\_loan', 'monthly\_payment'): 0.9256103360334527  
 data = data.drop(['total\_loan'], axis=1)  
 return data  
  
 def fillNanMethod(self, data):  
 # 默认均采用众数来填充所有缺失值  
 for columnName in data.columns:  
 if columnName in self.fillNanRules:  
 if self.fillNanRules[columnName][0] == 3:  
 data[columnName] = self.\_\_fillNan(data[columnName], self.fillNanRules[columnName][0],  
 self.fillNanRules[columnName][1])  
 else:  
 data[columnName] = self.\_\_fillNan(data[columnName], self.fillNanRules[columnName][0])  
 else:  
 data[columnName] = self.\_\_fillNan(data[columnName])  
 return data  
  
 Mapping = {'class': {'A': 0, 'B': 1, 'C': 2, 'D': 3, 'E': 4, 'F': 5, 'G': 6},  
 'employer\_type': {'幼教与中小学校': 0, '政府机构': 1, '上市企业': 2, '普通企业': 3, '世界五百强': 4,  
 '高等教育机构': 5},  
 'industry': {'金融业': 0, '国际组织': 1, '文化和体育业': 2, '建筑业': 3, '电力、热力生产供应业': 4,  
 '批发和零售业': 5, '采矿业': 6,  
 '房地产业': 7, '交通运输、仓储和邮政业': 8, '公共服务、社会组织': 9, '住宿和餐饮业': 10,  
 '信息传输、软件和信息技术服务业': 11, '农、林、牧、渔业': 12, '制造业': 13},  
 'work\_year': {'8 years': 8, '10+ years': 10, '3 years': 3, '7 years': 7, '4 years': 4, '1 year': 1,  
 '< 1 year': 0, '6 years': 6, '2 years': 2, '9 years': 9, '5 years': 5}}  
  
 @staticmethod  
 def \_\_dealString(column, mapping):  
 values = column.values.tolist()  
 for value in values:  
 if value not in mapping:  
 raise ValueError(f"Value '{value}' is not in {list(mapping.keys())}")  
 column = column.map(mapping)  
 return column  
  
 @staticmethod  
 def \_\_calcMon2Target(date, target\_year, target\_month):  
 target\_date = datetime(target\_year, target\_month, 1).date()  
 diff = relativedelta(date, target\_date)  
 total\_months = total\_months = diff.years \* 12 + diff.months  
 return total\_months  
  
 def \_\_dealDate2Month(self, column):  
 column = pd.to\_datetime(column, format='%Y/%m/%d')  
 column = column.apply(lambda x: self.\_\_calcMon2Target(x, 1970, 1))  
 return column  
  
 def \_\_dealString2Month(self, column):  
 column = column.apply(lambda x: self.matchMonth(x))  
 return column  
  
 @staticmethod  
 def matchMonth(text):  
 pattern = re.compile(r'[A-Za-z]+')  
 matches = pattern.findall(text)  
 try:  
 month = datetime.strptime(matches[0], "%b").month  
 except ValueError:  
 raise ValueError(f"Value '{matches[0]}' is a valid month")  
 return month  
  
 def dealStringMethod(self, data):  
 data['class'] = self.\_\_dealString(data['class'], self.Mapping['class'])  
 # 第二步处理 employer\_type 属性  
 data['employer\_type'] = self.\_\_dealString(data['employer\_type'], self.Mapping['employer\_type'])  
 # 第三步处理 industry 属性  
 data['industry'] = self.\_\_dealString(data['industry'], self.Mapping['industry'])  
 # 第四步处理 work\_year 属性  
 data['work\_year'] = self.\_\_dealString(data['work\_year'], self.Mapping['work\_year'])  
 # 第五步处理 issue\_date 属性，该属性的含义 贷款发放的月份,因此只用截至到年份  
 data['issue\_date'] = self.\_\_dealDate2Month(data['issue\_date'])  
 # 第六步处理 earlies\_credit\_mon 属性，该属性约束到月份  
 data['earlies\_credit\_mon'] = self.\_\_dealString2Month(data['earlies\_credit\_mon'])  
 return data  
  
 @staticmethod  
 def calculate\_within\_bin\_variance(data, bins):  
 hist, bin\_edges = np.histogram(data, bins=bins)  
 within\_variances = []  
 for i in range(len(bin\_edges) - 1):  
 bin\_data = data[(data >= bin\_edges[i]) & (data < bin\_edges[i + 1])]  
 if len(bin\_data) > 1:  
 within\_variances.append(np.var(bin\_data))  
 else:  
 within\_variances.append(0)  
 return np.mean(within\_variances) if within\_variances else float('inf')  
  
 @staticmethod  
 def calculate\_between\_bin\_variance(data, bins):  
 hist, bin\_edges = np.histogram(data, bins=bins)  
 bin\_means = []  
 for i in range(len(bin\_edges) - 1):  
 bin\_data = data[(data >= bin\_edges[i]) & (data < bin\_edges[i + 1])]  
 if len(bin\_data) > 0:  
 bin\_means.append(np.mean(bin\_data))  
 overall\_mean = np.mean(data)  
 between\_variance = sum(  
 [(mean - overall\_mean) \*\* 2 \* len(data[(data >= bin\_edges[i]) & (data < bin\_edges[i + 1])]) for i, mean in  
 enumerate(bin\_means)])  
 return between\_variance / len(data)  
  
 def find\_optimal\_bins(self, data, max\_bins=90):  
 best\_bins = 2  
 best\_score = float('-inf')  
 for bins in range(2, max\_bins + 1):  
 within\_variance = self.calculate\_within\_bin\_variance(data, bins)  
 between\_variance = self.calculate\_between\_bin\_variance(data, bins)  
 score = between\_variance - within\_variance # Aim for low within and high between  
 if score > best\_score:  
 best\_score = score  
 best\_bins = bins  
 # print(f"Bins: {bins}, Score: {score}, Within Variance: {within\_variance}, Between Variance: {between\_variance}")  
 return best\_bins  
  
 Segment = {}  
  
 def \_\_dealDataSegment(self, column, column\_name, typ='train'):  
 if column\_name in self.Segment:  
 column = pd.cut(column, bins=self.Segment[column\_name], labels=range(len(self.Segment[column\_name]) - 1),  
 include\_lowest=True)  
 return column  
 if typ == 'test':  
 return column  
 if column\_name not in self.segmentMap:  
 return column  
  
 num = self.segmentMap[column\_name]  
 bins = np.linspace(column.min(), column.max(), num + 1)  
 column = pd.cut(column, bins=bins, labels=range(num),  
 include\_lowest=True)  
 self.Segment[column\_name] = bins  
 return column  
  
 # length = len(column.value\_counts().index)  
 # if 50 <= length <= 2500:  
 # num = 1 + int(np.log2(length))  
 # bins = np.linspace(column.min(), column.max(), num + 1)  
 # column = pd.cut(column, bins=bins, labels=range(num),  
 # include\_lowest=True)  
 # self.Segment[column\_name] = bins  
 # return column  
 # elif length > 2500:  
 # # num = int(length / 100)  
 # num = 2 \* int(length \*\* (1 / 3))  
 # bins = np.linspace(column.min(), column.max(), num + 1)  
 # column = pd.cut(column, bins=bins, labels=range(num),  
 # include\_lowest=True)  
 # self.Segment[column\_name] = bins  
 # return column  
 # return column  
  
 def dealSegment(self, data, typ='train'):  
 for columnName in data.columns:  
 data[columnName] = self.\_\_dealDataSegment(data[columnName], columnName, typ=typ)  
 return data  
  
 def Process(self):  
 self.isProcessed = True  
 y\_process = self.data['isDefault']  
 self.data = self.dropColumn(self.data)  
 self.data = self.fillNanMethod(self.data)  
 self.data = self.dealStringMethod(self.data)  
 self.data = self.dealSegment(self.data)  
 X\_process = self.data  
 print("Process")  
 return X\_process, y\_process  
  
 def Deal(self, df):  
 print("Deal For Test")  
 if self.isProcessed is False:  
 raise ValueError(f'This DataProcessor is not processed or trained')  
 y\_process = df['isDefault']  
 # print("Test Drop")  
 df = self.dropColumn(df)  
 # print("Test Fill")  
 df = self.fillNanMethod(df)  
 # print("Test Trans")  
 df = self.dealStringMethod(df)  
 # print("Test Segment")  
 df = self.dealSegment(df, 'test')  
 X\_process = df  
 # print("Test Data Finished")  
 return X\_process, y\_process

### 7.贝叶斯决策器实现

#### 7.1 理论简述与简化

贝叶斯理论的公式如下：

其中，为后验概率，为类条件概率密度，为先验概率。显然当边缘概率确定时，我们只需要计算并比较分子的大小，即可取得最大的后验概率。

在此基础上，假定剩余的所有特征均是独立的，那么

因此在决策器训练过程中，可以将当前出现的所有产生的类条件概率密度进行保存即可。

对于先验概率，也可以通过简单的计数过程来实现。，并进行保存即可。

#### 7.2 拉普拉斯平滑

拉普拉斯平滑（Laplace Smoothing），也称为加一平滑，是一种在概率估计中常用的技术，尤其在处理分类数据时，例如在文本分类和自然语言处理的贝叶斯模型中。拉普拉斯平滑的主要目的是解决零概率问题，确保在模型中不会出现任何概率的估计值为零。

当在测试数据或其他数据中某个类别的某个特征从未出现过时，按照常规的最大似然估计，这个类别下这个特征的概率将会是零。这会导致整个数据样本的概率计算结果也为零，影响模型的预测效果。通过拉普拉斯平滑，即使数据在训练集中没有出现，也可以赋予它一个小的非零概率，从而避免零概率的问题。

拉普拉斯平滑通过在计数中加上一个正数（通常是1），对概率估计进行平滑。这种方法特别适用于数据稀疏的情况，可以减少估计值对于未见数据的敏感性。在此实验中设置一个拉普拉斯系数 来表示这个正数。在未见数据（即训练数据中未出现的特征组合）上，拉普拉斯平滑可以帮助模型做出更合理的预测，而不是直接判定这些情况的概率为零。这有助于增强模型对新数据的泛化能力。

拉普拉斯平滑可以看作是在贝叶斯框架下的先验知识的引入。在没有足够信息确定某个事件的概率时，默认所有可能结果等可能，这反映了一种非信息先验的思想。

对于训练过程中： 表示分类标签的所有集合，表示所有取值的集合

对于先验概率：

对于类条件概率密度：

那么对于在测试过程中出现的未在中出现的值，

基于如上的简化和拉普拉斯平滑，即可以写出贝叶斯决策的代码。

#### 7.3 完整贝叶斯决策器代码实现

整体实现代码如下：

'''  
NaiveBayesClassifier.py  
'''  
  
class NaiveBayesClassifier:  
 def \_\_init\_\_(self, alpha):  
 self.alpha = alpha  
 self.class\_prior = {}  
 self.cond\_prob = {}  
  
 def fit(self, X, y):  
 self.classes = np.unique(y)  
 self.features = [np.unique(X.iloc[:, col]) for col in range(X.shape[1])]  
 self.X = X  
 self.y = y  
 total\_count = len(y)  
 for cls in self.classes:  
 cls\_count = np.sum(y == cls)  
 self.class\_prior[cls] = (cls\_count + self.alpha) / (total\_count + len(self.classes) \* self.alpha)  
 self.cond\_prob[cls] = {}  
 for i, feature in enumerate(self.features):  
 self.cond\_prob[cls][i] = {}  
 for value in feature:  
 feature\_count = np.sum((X.iloc[:, i] == value) & (y == cls))  
 self.cond\_prob[cls][i][value] = (feature\_count + self.alpha) / (  
 cls\_count + len(feature) \* self.alpha)  
  
 def predict(self,X\_test):  
 predictions = []  
 for x in X\_test.values:  
 probs ={}  
 for cls in self.classes:  
 probs[cls] = self.class\_prior[cls]  
 for i,value in enumerate(x):  
 if value in self.cond\_prob[cls][i]:  
 probs[cls] \*= self.cond\_prob[cls][i][value]  
 else:  
 probs[cls] \*= self.alpha / (np.sum(self.y == cls) + len(self.features[i]) \* self.alpha)  
 # print(max(probs, key=probs.get))  
 predictions.append(max(probs, key=probs.get))  
 return predictions

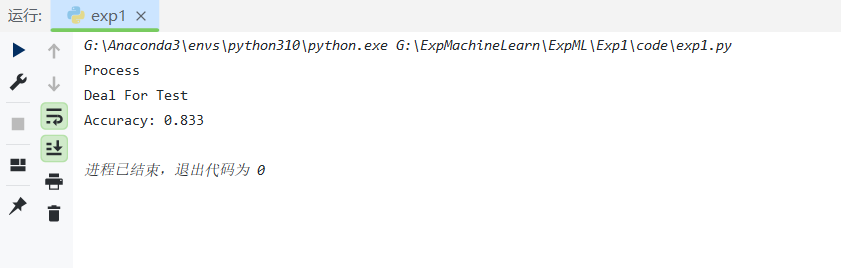
### 8.实验执行

编写实验主程序exp1.py

利用上述描述的最基本的分段，即均分为14段，执行代码并观测准确率

'''  
exp1.py  
'''  
  
import pandas as pd  
from DataProcess import DataProcessor  
from NaiveBayesClassifier import NaiveBayesClassifier  
  
segmentMap = {  
 'post\_code': 14,  
 'title': 14,  
 'known\_outstanding\_loan': 14,  
 'monthly\_payment': 14,  
 'issue\_date': 14,  
 'debt\_loan\_ratio': 14,  
 'scoring\_high': 14,  
 'recircle\_b': 14,  
 'recircle\_u': 14,  
 'f0': 14,  
 'f2': 14,  
 'f3': 14,  
 'early\_return\_amount\_3mon': 14,  
 }  
  
train\_data = pd.read\_csv('../data/train.csv')  
test\_data = pd.read\_csv('../data/test.csv')  
  
dp = DataProcessor(train\_data,segmentMap)  
  
x\_train, y\_train = dp.Process()  
x\_test, y\_test = dp.Deal(test\_data)  
# x\_test.to\_csv('../process/testData.csv', index=False)  
  
nbc = NaiveBayesClassifier(alpha=1)  
nbc.fit(x\_train, y\_train)  
pred = nbc.predict(x\_test)  
  
correct\_predictions = sum(p == t for p, t in zip(pred, y\_test))  
accuracy = correct\_predictions / len(y\_test)  
print(f'Accuracy: {accuracy}')

运行后结果如下：



### 9.优化调整

基于以上的数据处理过程，能进行优化的即细化各类别的分箱个数。在实际情况下需要考虑分箱的可解释性。由于缺乏实际的经济管理知识，对某些特定值分箱的解释性并不能做到很好。

如果把分箱数看作超参数，通过一些超参数搜索优化方法，利用网格调参搜索、随机参数搜索以及退火等方法，可以找到一个或者一系列准确率较高的分箱个数，这时，可以对比部分变化较大的箱数来探究不同的分箱策略对某些特定值分箱的可视化效果。

先初步考虑分箱的优化方法，首先可以采用搜索的方法去在一定范围内穷举所有的分箱数，但是显然如果采用这种方式，整个代码的运行时间是阶乘级的，是一个NP-Hard问题，基本是不可取的，尤其是在如上方法种共有14个列需要分箱。

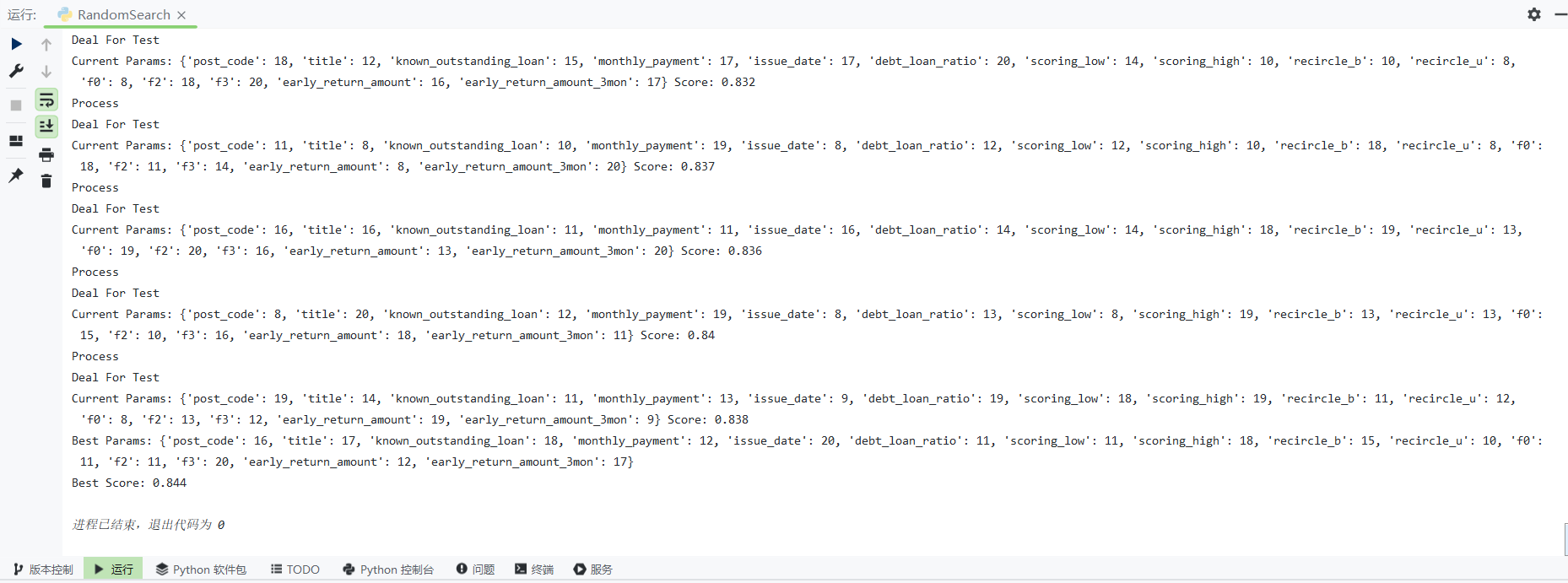
#### 9.1 随机搜索

首先考虑采用随机搜索的方法，来尝试搜索更好的分箱组数，采用如下代码：

'''  
RandomSearch.py  
随机搜索查询更好的分箱数，以此追求更高的准确率  
'''  
  
import numpy as np  
import pandas as pd  
  
from Exp1.code.DataProcess import DataProcessor  
from Exp1.code.NaiveBayesClassifier import NaiveBayesClassifier  
  
  
train\_data = pd.read\_csv('../data/train.csv')  
test\_data = pd.read\_csv('../data/test.csv')  
  
segmentMap = {  
 'post\_code': 13,  
 'title': 8,  
 'known\_outstanding\_loan': 13,  
 'monthly\_payment': 20,  
 'issue\_date': 12,  
 'debt\_loan\_ratio': 13,  
 'scoring\_low': 13,  
 'scoring\_high': 13,  
 'recircle\_b': 13,  
 'recircle\_u': 11,  
 'f0': 11,  
 'f2': 13,  
 'f3': 13,  
 'early\_return\_amount\_3mon': 13,  
}  
  
def evaluate\_model(parameters, dataT):  
 # 这个函数应该基于提供的参数处理数据并返回模型准确率  
 data = dataT.copy()  
 dp = DataProcessor(data, parameters)  
 x\_train, y\_train = dp.Process() # 处理训练数据  
 x\_test, y\_test = dp.Deal(test\_data) # 处理测试数据  
  
 nbc = NaiveBayesClassifier(alpha=1)  
 nbc.fit(x\_train, y\_train)  
 predictions = nbc.predict(x\_test)  
 accuracy = np.mean(predictions == y\_test)  
  
 dp.Segment={}  
 dp.isProcessed = False  
 dp.data = None  
  
 return accuracy  
  
  
def random\_search(data, iterations=100):  
 best\_params = None  
 best\_score = 0  
  
 for \_ in range(iterations):  
 # 随机生成参数  
 params = {key: np.random.randint(8, 21) for key in segmentMap.keys()}  
  
 # 评估当前参数集  
 score = evaluate\_model(params, data)  
  
 # 检查是否是最佳参数集  
 if score > best\_score:  
 best\_score = score  
 best\_params = params  
  
 print(f"Current Params: {params} Score: {score}")  
  
 return best\_params, best\_score  
  
  
# 调用随机搜索  
best\_params, best\_score = random\_search(train\_data)  
print("Best Params:", best\_params)  
print("Best Score:", best\_score)

可以得到如下的搜索结果：

由于每次的搜索存在差异性，因此每次搜索最优结果都不一定一致。



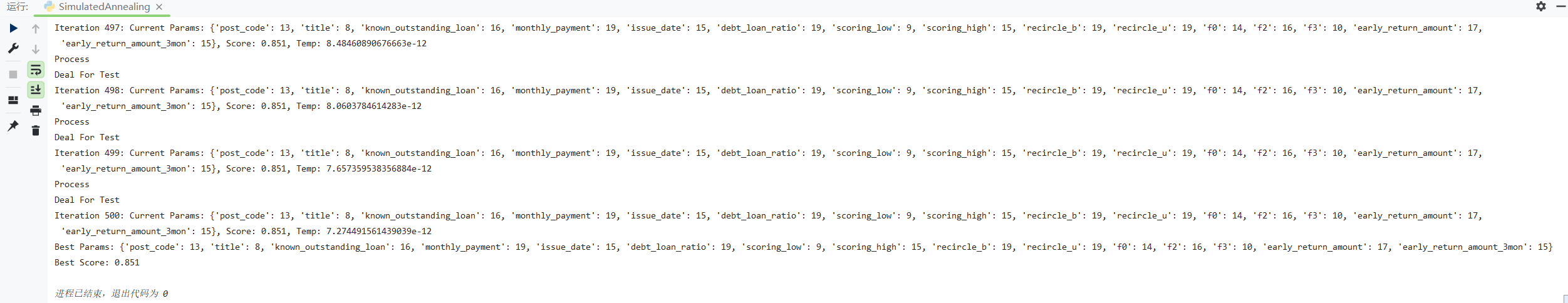
#### 9.2 模拟退火搜索

但是显然这种搜索更像是漫无目的的，因此考虑整体趋势更加良好的模拟退火来对这些超参数进行搜索估计，基于如下代码实现模拟退火搜索，进行超参搜索。

'''  
SimulatedAnnealing.py  
模拟退火搜索查询更好的分箱数，以此追求更高的准确率  
'''  
  
import numpy as np  
import pandas as pd  
  
from Exp1.code.DataProcess import DataProcessor  
from Exp1.code.NaiveBayesClassifier import NaiveBayesClassifier  
  
train\_data = pd.read\_csv('../data/train.csv')  
test\_data = pd.read\_csv('../data/test.csv')  
  
segmentMap = {  
 'post\_code': 13, 'title': 8, 'known\_outstanding\_loan': 13,  
 'monthly\_payment': 20, 'issue\_date': 12, 'debt\_loan\_ratio': 13,  
 'scoring\_low': 13, 'scoring\_high': 13, 'recircle\_b': 13,  
 'recircle\_u': 11, 'f0': 11, 'f2': 13, 'f3': 13,  
 'early\_return\_amount': 13, 'early\_return\_amount\_3mon': 13,  
}  
  
def evaluate\_model(parameters, dataT):  
 data = dataT.copy()  
 dp = DataProcessor(data, parameters)  
 x\_train, y\_train = dp.Process()  
 x\_test, y\_test = dp.Deal(test\_data)  
  
 nbc = NaiveBayesClassifier(alpha=1)  
 nbc.fit(x\_train, y\_train)  
 predictions = nbc.predict(x\_test)  
 accuracy = np.mean(predictions == y\_test)  
  
 return accuracy  
  
def simulated\_annealing(data, iterations=300, temp=1.0, temp\_decay=0.95):  
 current\_params = {key: np.random.randint(8, 21) for key in segmentMap.keys()}  
 current\_score = evaluate\_model(current\_params, data)  
 best\_params = current\_params.copy()  
 best\_score = current\_score  
  
 for i in range(iterations):  
 new\_params = current\_params.copy()  
 for key in new\_params.keys():  
 if np.random.rand() < 0.5:  
 new\_params[key] = np.random.randint(8, 21)  
  
 new\_score = evaluate\_model(new\_params, data)  
  
 if new\_score > current\_score:  
 accept = True  
 else:  
 delta = new\_score - current\_score  
 accept\_prob = np.exp(delta / temp)  
 accept = np.random.rand() < accept\_prob  
  
 if accept:  
 current\_params, current\_score = new\_params, new\_score  
 if new\_score > best\_score:  
 best\_params, best\_score = new\_params.copy(), new\_score  
  
 temp \*= temp\_decay  
  
 print(f"Iteration {i+1}: Current Params: {current\_params}, Score: {current\_score}, Temp: {temp}")  
  
 return best\_params, best\_score  
  
best\_params, best\_score = simulated\_annealing(train\_data)  
print("Best Params:", best\_params)  
print("Best Score:", best\_score)

搜索结果如下：

由于每次的搜索存在差异性，因此每次搜索最优结果都不一定一致。



#### 9.3 模拟退火+K-Fold(k=5)

考虑到上述的模型都是利用测试集来进行，这缺乏一些泛化性，对训练集采用k-Fold，采用5个叠

import numpy as np  
import pandas as pd  
  
from Exp1.code.DataProcess import DataProcessor  
from Exp1.code.NaiveBayesClassifier import NaiveBayesClassifier  
  
train\_data = pd.read\_csv('../data/train.csv')  
test\_data = pd.read\_csv('../data/test.csv')  
  
segmentMap = {  
 'post\_code': 13, 'title': 8, 'known\_outstanding\_loan': 13,  
 'monthly\_payment': 20, 'issue\_date': 12, 'debt\_loan\_ratio': 13,  
 'scoring\_low': 13, 'scoring\_high': 13, 'recircle\_b': 13,  
 'recircle\_u': 11, 'f0': 11, 'f2': 13, 'f3': 13,  
 'early\_return\_amount': 13, 'early\_return\_amount\_3mon': 13,  
}  
  
def evaluate\_model(parameters, data, n\_folds=5):  
 accuracies = []  
  
 # 计算每个折的大小  
 fold\_size = len(data) // n\_folds  
  
 for i in range(n\_folds):  
 # 确定验证集的索引范围  
 start\_idx = i \* fold\_size  
 end\_idx = (i + 1) \* fold\_size if i < n\_folds - 1 else len(data)  
  
 # 分割数据为训练集和验证集  
 validation\_data = data.iloc[start\_idx:end\_idx]  
 train\_data = pd.concat([data.iloc[:start\_idx], data.iloc[end\_idx:]])  
  
 dp = DataProcessor(train\_data, parameters)  
 x\_train, y\_train = dp.Process()  
 x\_test, y\_test = dp.Deal(validation\_data)  
  
 # 训练模型  
 nbc = NaiveBayesClassifier(alpha=1)  
 nbc.fit(x\_train, y\_train)  
  
 # 进行预测并计算准确率  
 predictions = nbc.predict(x\_test)  
 accuracy = np.mean(predictions == y\_test)  
 accuracies.append(accuracy)  
  
 mean\_accuracy = np.mean(accuracies)  
 return mean\_accuracy  
  
def simulated\_annealing(data, iterations=300, temp=1.0, temp\_decay=0.95):  
 current\_params = {key: np.random.randint(8, 21) for key in segmentMap.keys()}  
 current\_score = evaluate\_model(current\_params, data)  
 best\_params = current\_params.copy()  
 best\_score = current\_score  
  
 for i in range(iterations):  
 new\_params = current\_params.copy()  
 for key in new\_params.keys():  
 if np.random.rand() < 0.5:  
 new\_params[key] = np.random.randint(8, 21)  
  
 new\_score = evaluate\_model(new\_params, data)  
  
 if new\_score > current\_score:  
 accept = True  
 else:  
 delta = new\_score - current\_score  
 accept\_prob = np.exp(delta / temp)  
 accept = np.random.rand() < accept\_prob  
  
 if accept:  
 current\_params, current\_score = new\_params, new\_score  
 if new\_score > best\_score:  
 best\_params, best\_score = new\_params.copy(), new\_score  
  
 temp \*= temp\_decay  
  
 print(f"Iteration {i+1}: Current Params: {current\_params}, Score: {current\_score}, Temp: {temp}")  
  
 return best\_params, best\_score  
  
best\_params, best\_score = simulated\_annealing(train\_data)  
print("Best Params:", best\_params)  
print("Best Score:", best\_score)

搜索结果如下：

显然考虑更多效果时的准确率不如之前两种，但这种的泛化性更好，过拟合的情况更少。但迭代300次的时候的运行速度非常的缓慢。

### 10.代码目录结构

代码的目录结构如下：

/Exp1/  
|------- code/  
| |------- DataAnaslyze.py 数据分析的函数与整体分析  
| |------- DataProcess.py 数据处理器  
| |------- NaiveBayesClassifier.py 贝叶斯决策器  
| |------- exp1.py 实验一主代码  
| |------- RandomSearch.py 随机搜索分箱数  
| |------- SimulatedAnnealing.py 模拟退火搜索  
| |------- kFoldandSL.py k折叠模拟退火  
|  
|-------- data/  
| |------- train.csv 训练数据  
| |------- test.csv 测试数据  
| |------- 数据分析器.xlsx 数据分析表格  
| |------- 相关系数矩阵.xlsx   
|  
|-------- pic/  
| |------- res/ 实验报告有关图片  
| |------- 其他信息  
|  
|-------- process/  
| |------- Processed\_TrainData.csv 临时处理数据  
| |------- test.csv 临时处理的数据  
|  
|-------- Exp1.md 实验报告Markdown  
|  
|-------- Exp1.pdf 实验报告pdf

## 心得体会