

安徽大学人工智能学院
《机器学习程序设计》
实验案例设计报告

课程名称: 机器学习程序设计

专 业: 人工智能

班 级: 人工智能二班

学 号: WA2214014

姓 名: 杨跃浙

任课老师: 谭春雨

实验名称	保险产品推荐	实验次序	01
实验地点	笃行南楼 A104	实验日期	03.09

实验内容:

1. 业务背景分析

保险产品的多样性、客户特征的复杂性以及需求差异使得保险推荐存在相当大的不确定性，如何精准识别用户、降低销售风险、提升推荐成功率，成为当前一个非常热门的研究和应用话题。

通过对用户本身属性和过往保险购买记录分析客户特点,可以对广大用户进行个人信息的有效筛选，从购买保险的用户群体中提取共同的特征，进而针对这些特征规律提高投放精准性。

本案例是针对移动房车险的预测，其中保险公司提供了以家庭为单位的历史数据，包括家庭的各类特征属性和历史投保记录。通过对历史数据的分析，建立移动房车险的预测模型。

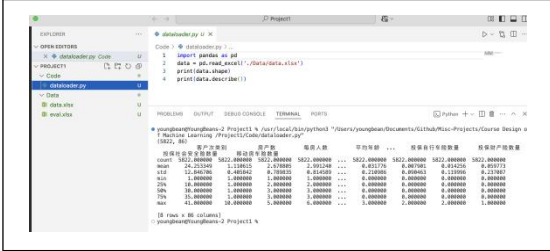
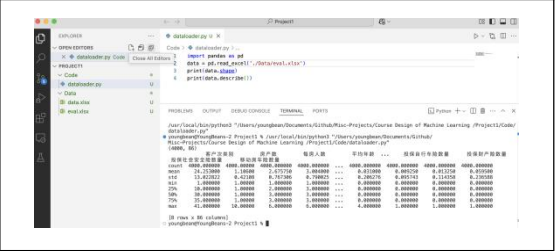
2. 数据概况

数据分为训练数据和测试数据两部分，其中训练数据 5822 条，测试数据 4000 条。

每一条数据有 86 个字段，1~43 字段为用户基本属性，包括财产、宗教、家庭情况、教育程度、职位、收入水平等；44~85 字段为用户历史保险购买情况，第 86 个字段为目标预测字段，取值为 0 和 1，表示用户是否购买该房车险。

这是一个典型的二分类问题

3. 数据提取

测试集	验证集
	

4. 数据预处理

数据缺失	统计类别数目	训练集的偏度
<pre> data_loader.py U X Code > data_loader.py > ... 1 import pandas as pd 2 train = pd.read_excel('./Data/data.xlsx') 3 test = pd.read_excel('./Data/eval.xlsx') 4 train['source'] = 'train' 5 test['source'] = 'test' 6 data = pd.concat([train, test], ignore_index=True) 7 nan_count = data.isnull().sum().sort_values(ascending=True) 8 nan_ratio = nan_count/len(data) 9 nan_data = pd.concat([nan_count, nan_ratio], axis=1) 10 print(nan_data.head(10)) </pre> 	<pre> datapre_2.py U X Code > datapre_2.py > ... 1 import pandas as pd 2 train = pd.read_excel('./Data/data.xlsx') 3 test = pd.read_excel('./Data/eval.xlsx') 4 train['source'] = 'train' 5 test['source'] = 'test' 6 data = pd.concat([train, test], ignore_index=True) 7 count = data.apply(lambda x: len(x.unique()), axis=1) 8 print(count.head(10)) </pre> 	<pre> datapre_3.py U X Code > datapre_3.py > ... 1 import pandas as pd 2 train = pd.read_excel('./Data/data.xlsx') 3 test = pd.read_excel('./Data/eval.xlsx') 4 train['source'] = 'train' 5 test['source'] = 'test' 6 data = pd.concat([train, test], ignore_index=True) 7 train = data.loc[data['source'] == 'train'] 8 test = data.loc[data['source'] == 'test'] 9 train.drop(['source'], axis=1, inplace=True) 10 test.drop(['source'], axis=1, inplace=True) 11 skewness = train.skew().sort_values(ascending=True) 12 print(skewness.head(10)) </pre> 

通过箱图对不同条件变量对目标变量的区分度

代码:

```
import pandas as pd
import matplotlib.pyplot as plt
import os
```

设置 Matplotlib 中文字体

```
plt.rcParams['font.sans-serif'] = ['Heiti TC'] # 选择合适的中文字体
plt.rcParams['axes.unicode_minus'] = False # 解决负号显示问题
```

读取数据

```
train = pd.read_excel('./Data/data.xlsx')
test = pd.read_excel('./Data/eval.xlsx')
```

```
train['source'] = 'train'
test['source'] = 'test'
```

```
data = pd.concat([train, test], ignore_index=True, sort=False)
```

选择数值的列

```
num_columns = train.select_dtypes(include=['number']).columns.tolist()
num_columns.remove('移动房车险数量')
save_dir = './boxplots'
```

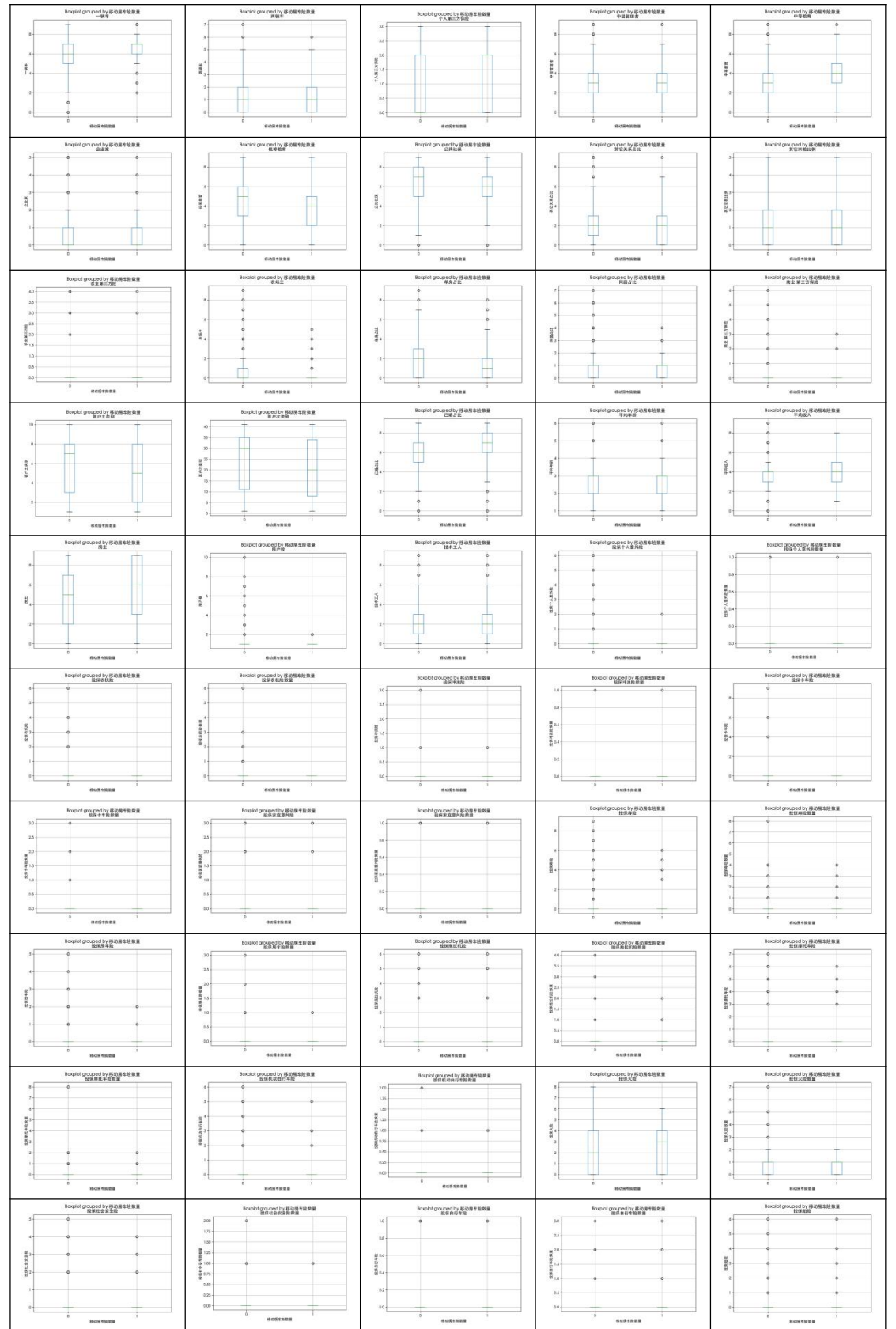
逐个绘制并保存

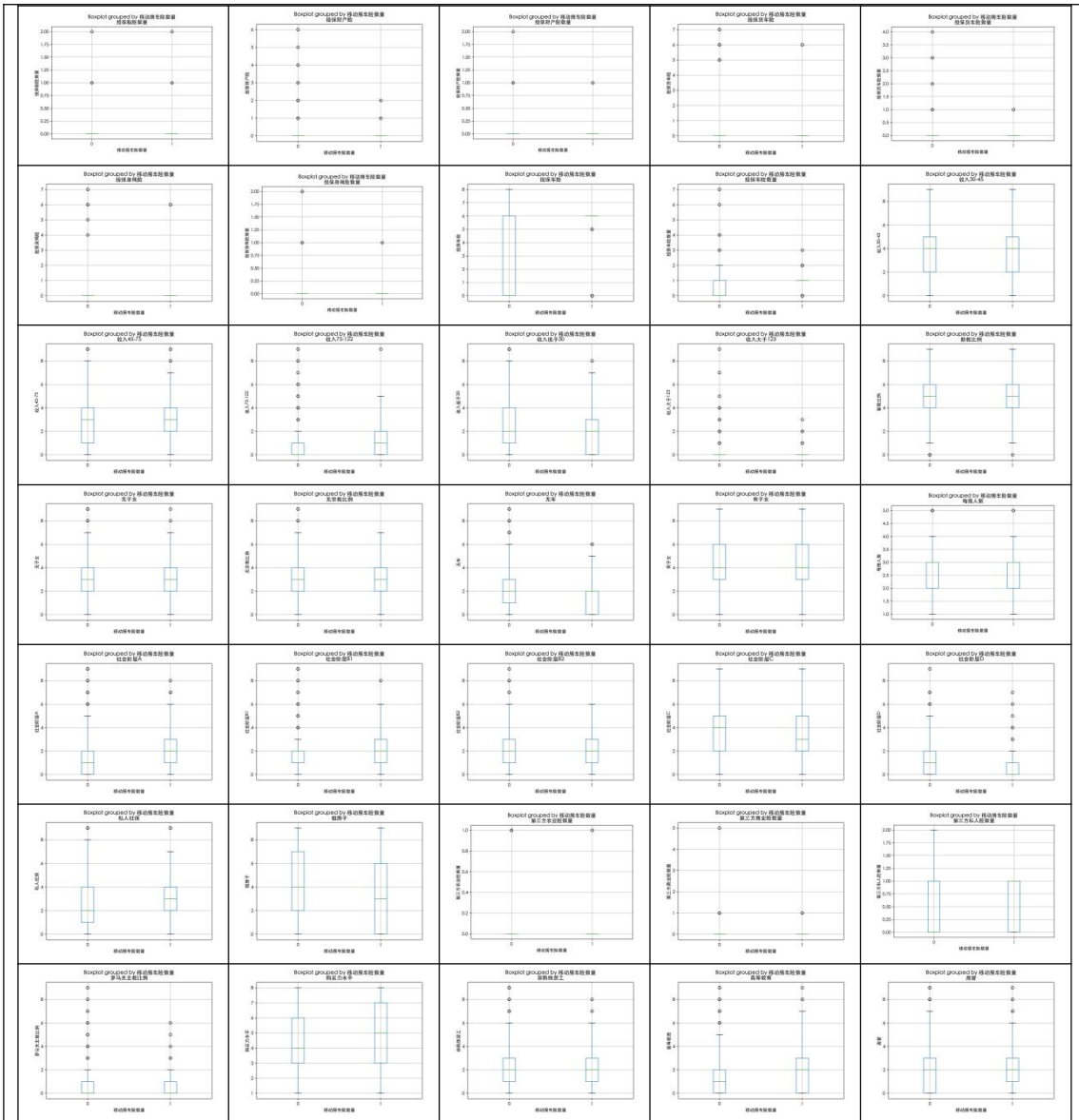
```
for col in num_columns:
    plt.figure(figsize=(6, 4))
    train.boxplot(column=col, by='移动房车险数量')
    plt.xlabel('移动房车险数量')
    plt.ylabel(col)
    plt.title(col)
```

保存图片

```
save_path = os.path.join(save_dir, f'{col}.png')
plt.savefig(save_path, dpi=300, bbox_inches='tight')
plt.close()
```

运行结果:





结论：

箱线图分析显示，相较于未购买移动房车保险的家庭，购买者呈现出以下特点：

- 经济实力更强：购买力水平较高，平均收入更高。
- 教育水平较高：家庭成员整体受教育程度更高。
- 风险意识更强：投保火险的比例较高，体现出更强的风险管理意识。
- 婚姻状况较稳定：已婚家庭成员比例更高，可能与家庭责任感和保险需求相关。
- 更倾向私人保障：私人社保投保比例较高，而公共社保投保比例相对较低。
- 职业分布不同：农场主等人群极少投保移动房车险，而高管阶层投保比例较高，反映出职业类别对保险选择的影响。

总体来看，购买移动房车保险的家庭普遍经济实力更强，教育水平更高，更注重风险管理，并在职业结构上展现出一定的趋势。

降低数据的维度

代码:

```
import pandas as pd
import numpy as np

train = pd.read_excel('./Data/data.xlsx')
test = pd.read_excel('./Data/eval.xlsx')

train['source'] = 'train'
test['source'] = 'test'

data = pd.concat([train, test], ignore_index=True, sort=False)

numeric_train = train.select_dtypes(include=[np.number])

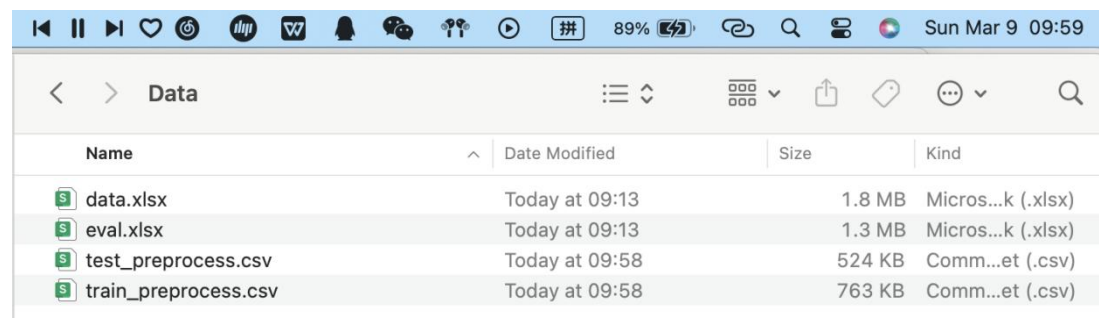
# 计算相关系数
corr_target = numeric_train.corr()['移动房车险数量']

# 选择相关性绝对值 >= 0.01 的特征
important_feature = corr_target[np.abs(corr_target) >= 0.01].index.tolist()

train = train[important_feature]
test = test[important_feature]

train.to_csv('./Data/train_preprocess.csv', encoding='utf_8_sig',
index=False)
test.to_csv('./Data/test_preprocess.csv', encoding='utf_8_sig', index=False)
```

运行结果:



Name	Date Modified	Size	Kind
data.xlsx	Today at 09:13	1.8 MB	Micros...k (.xlsx)
eval.xlsx	Today at 09:13	1.3 MB	Micros...k (.xlsx)
test_preprocess.csv	Today at 09:58	524 KB	Comm...et (.csv)
train_preprocess.csv	Today at 09:58	763 KB	Comm...et (.csv)

5. 分类模型构建

使用 sklearn 构建最基础的决策树模型:

```
import numpy as np
from sklearn import tree
from sklearn import metrics
import pandas as pd
from sklearn.model_selection import cross_validate

def load_data(path):
    data = pd.read_csv(path, encoding='utf-8')
    x, y = data.iloc[:, :-1], data.iloc[:, -1]
    return x, y

def build_model(x, y):
    classifier = tree.DecisionTreeClassifier()
    classifier.fit(x, y)
    return classifier

def test_model(classifier):
    test_x, test_y = load_data('./Data/test_preprocess.csv')
    scores = cross_validate(classifier, test_x, test_y, cv=5, scoring=('accuracy',
'precision', 'recall', 'f1', 'roc_auc'))
```

```

return scores
if __name__ == '__main__':
train_x, train_y = load_data('./Data/train_preprocess.csv')
classifier = build_model(train_x, train_y)
scores = test_model(classifier)
print('Accuracy %.4f' % (np.mean(scores['test_accuracy'])))
print('Precision %.4f' % (np.mean(scores['test_precision'])))
print('Recall %.4f' % (np.mean(scores['test_recall'])))
print('F1 %.4f' % (np.mean(scores['test_f1'])))
print('AUC %.4f' % (np.mean(scores['test_roc_auc'])))

```

运行结果：

```

● youngbean@YoungBeans-
class_1.py"
Accuracy 0.8960
Precision 0.1100
Recall 0.1009
F1 0.1047
AUC 0.5307

```

结论：

基础决策树模型在本次实验中的表现存在明显的局限性。从数据结果来看，准确率达到 0.8960，这意味着模型在整体预测中，大约有 89.60% 的样本被正确分类。然而，精确率仅为 0.1100，召回率为 0.1009，F1 分数为 0.1047，AUC 为 0.5307。精确率低表明在模型预测为购买移动房车险（正类）的样本中，只有 11% 实际是购买的，大量预测为正类的样本是错误的，即存在较多误报情况。召回率低则说明实际购买移动房车险的用户中，模型仅能正确识别出 10.09%，大量购买用户被漏报。AUC 接近 0.5，按照评估标准，这表明模型区分正负样本的能力与随机猜测相近，在实际的保险推荐场景中，难以精准地识别出真正有购买移动房车险需求的用户，无法为保险推荐提供可靠的支持。

说明：

公式	含义
$Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$	准确率：衡量整体分类正确的样本比例，适用于类别平衡的数据集。
$Precision = \frac{TP}{TP+FP}$	精确率（查准率）：预测为正例的样本中，实际为正例的比例，适用于减少误报的场景。
$Recall = \frac{TP}{TP+FN}$	召回率（查全率）：实际正例中被正确预测的比例，适用于减少漏报的场景。
$F1 = 2 \times \frac{Precision \times Recall}{Precision + Recall}$	<i>F1-score</i> : <i>Precision</i> 和 <i>Recall</i> 的调和平均，综合衡量模型的性能，适用于类别不均衡时。
ROC_AUC	<i>ROC</i> 曲线下的面积，衡量模型在不同阈值下的分类能力，值越接近 1，模型越优。

其中：

- TP (True Positive) : 预测为正, 实际也为正
- TN (True Negative) : 预测为负, 实际也为负
- FP (False Positive) : 预测为正, 实际为负 (误报)
- FN (False Negative) : 预测为负, 实际为正 (漏报)
- ROC 曲线 (Receiver Operating Characteristic Curve, 受试者工作特征曲线) 用于评估二分类模型的性能, 展示了不同分类阈值下 TPR (真正率/召回率) 和 FPR (假阳性率) 之间的关系, 以衡量模型区分正负样本的能力。其中, FPR (假阳性率) 计算公式为 $FPR = \frac{FP}{FP+TN}$, 表示实际为负的样本中被错误分类为正的比例; TPR (真正率/召回率) 计算公式为 $TPR = \frac{TP}{TP+FN}$, 表示所有实际为正的样本中被正确识别的比例。绘制 ROC 曲线的过程通常是通过改变分类阈值 (例如从 0 到 1), 计算每个阈值下的 TPR 和 FPR, 并以 FPR 为横轴, TPR 为纵轴绘制曲线, 以此观察模型在不同阈值下的分类能力。

修改分类模型:

```
import numpy as np
import pandas as pd
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score, roc_auc_score
```

加载数据

```
def load_data(path):
    data = pd.read_csv(path, encoding='utf-8')
    x, y = data.iloc[:, :-1], data.iloc[:, -1]
    return x, y
```

训练不同模型

```
def build_models(x, y):
    models = {
        "Decision Tree (ID3)": tree.DecisionTreeClassifier(criterion='entropy',
max_depth=10, min_samples_leaf=50),
        "Decision Tree (C4.5)": tree.DecisionTreeClassifier(criterion='entropy',
max_depth=10),
        "Random Forest": RandomForestClassifier(criterion='entropy', max_depth=10,
n_estimators=100),
        "MLP": MLPClassifier(hidden_layer_sizes=(50,), max_iter=500),
        "KNN": KNeighborsClassifier(n_neighbors=1),
        "Naive Bayes": GaussianNB(),
        "AdaBoost": AdaBoostClassifier(n_estimators=100),
        "Logistic Regression": LogisticRegression(solver='saga', penalty='l2',
max_iter=10000)
    }
    trained_models = {}
    for name, model in models.items():
        print(f"Training {name}...")
        model.fit(x, y)
        trained_models[name] = model

    return trained_models
```

评估模型

```
def evaluate_model(model, test_x, test_y):
    pred_y = model.predict(test_x)
```



```

accuracy = accuracy_score(test_y, pred_y)
precision = precision_score(test_y, pred_y, average='binary') # 根据数据类型调整
recall = recall_score(test_y, pred_y, average='binary')
f1 = f1_score(test_y, pred_y, average='binary')
auc = roc_auc_score(test_y, pred_y)
return accuracy, precision, recall, f1, auc

if __name__ == '__main__':
# 加载训练和测试数据
train_x, train_y = load_data('./Data/train_preprocess.csv')
test_x, test_y = load_data('./Data/test_preprocess.csv')

# 训练所有模型
models = build_models(train_x, train_y)

# 依次测试每个模型并输出结果
for name, model in models.items():
accuracy, precision, recall, f1, auc = evaluate_model(model, test_x, test_y)
print(f"\n{name} Results:")
print(f'Accuracy %.4f' % accuracy)
print(f'Precision %.4f' % precision)
print(f'Recall %.4f' % recall)
print(f'F1 Score %.4f' % f1)
print(f'AUC %.4f' % auc)

```

运行结果:

```
● youngbean@YoungBeans-2 Project1 % /usr/loca
class 2.py"
Training Decision Tree (ID3)...
Training Decision Tree (C4.5)...
Training Random Forest...
Training MLP...
Training KNN...
Training Naive Bayes...
Training AdaBoost...
Training Logistic Regression...
/Library/Frameworks/Python.framework/Versio
ill-defined and being set to 0.0 due to no
_warn_prf(average, modifier, msg_start, l

Decision Tree (ID3) Results:
Accuracy 0.9405
Precision 0.0000
Recall 0.0000
F1 Score 0.0000
AUC 0.5000

Decision Tree (C4.5) Results:
Accuracy 0.9307
Precision 0.1803
Recall 0.0462
F1 Score 0.0736
AUC 0.5165

Random Forest Results:
Accuracy 0.9403
Precision 0.0000
Recall 0.0000
F1 Score 0.0000
AUC 0.4999

MLP Results:
Accuracy 0.9353
Precision 0.3019
Recall 0.0672
F1 Score 0.1100
AUC 0.5287

KNN Results:
Accuracy 0.9038
Precision 0.1549
Recall 0.1387
F1 Score 0.1463
AUC 0.5454

Naive Bayes Results:
Accuracy 0.4793
Precision 0.0839
Recall 0.7815
F1 Score 0.1515
AUC 0.6208

AdaBoost Results:
Accuracy 0.9375
Precision 0.2273
Recall 0.0210
F1 Score 0.0385
AUC 0.5082

Logistic Regression Results:
Accuracy 0.9397
Precision 0.3333
Recall 0.0126
F1 Score 0.0243
AUC 0.5055
```

结论:

在本次实验中,我尝试了多种机器学习模型,包括决策树(ID3、C4.5)、随机森林、MLP(多层感知机)、KNN(K近邻)、朴素贝叶斯、AdaBoost和逻辑回归等。实验结果显示,尽管不同模型的准确率(Accuracy)较高,如决策树(ID3)的准确率为0.9405,C4.5为0.9307,随机森林为0.9403,但精确率(Precision)、召回率(Recall)以及F1分数(F1 Score)普遍较低,部分模型甚至完全未能预测正类(导致Precision和Recall为0)。此外,AUC接近0.5(如决策树ID3为0.5000,随机森林为0.4999),表明模型的预测接近随机猜测,泛化能力较差。

进一步分析实验数据发现,数据集存在**严重的类别不平衡问题**,即某一类样本占据绝大多数,而另一类样本极少。在原始数据中,投保数低于总数的6%。这种不平衡导致了模型的**过拟合**,即

模型在训练过程中更倾向于预测多数类，而忽略少数类。由于少数类数据较少，决策树和随机森林等模型在分裂过程中更倾向于优化多数类，导致学习到的规则几乎不会预测少数类，使得模型在识别正类（购买移动房车险的用户）时表现极差。这种不均衡问题是导致 Precision、Recall 以及 F1 Score 过低的主要原因。

6. 平衡数据集

可见样本显著不平衡，投保数低于总数的 6%：

```
balance_1.py U X
Code > balance_1.py > ...
1 import pandas as pd
2 train = pd.read_excel('./Data/data.xlsx')
3 test = pd.read_excel('./Data/eval.xlsx')
4 train['source'] = 'train'
5 test['source'] = 'test'
6 data = pd.concat([train, test], ignore_index=True, sort=False)
7 print(train['移动房车险数量'].value_counts())

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

/usr/local/bin/python3 "/Users/youngbean/Documents/Github/Misc-Projects/Course I
● youngbean@YoungBeans-2 Project1 % /usr/local/bin/python3 "/Users/youngbean/Docu
balance_1.py"
移动房车险数量
0    5474
1     348
Name: count, dtype: int64
```

使用重采样方式调整样本比例，将投保数上采样到原来的两倍，未投保数下降到原来的 20%。

代码：

```
from sklearn.utils import resample, shuffle
import pandas as pd
train = pd.read_excel('./Data/data.xlsx')
test = pd.read_excel('./Data/eval.xlsx')
train['source'] = 'train'
test['source'] = 'test'
data = pd.concat([train, test], ignore_index=True, sort=False)
print(train['移动房车险数量'].value_counts())
train_up = train[train['移动房车险数量'] == 1]
train_down = train[train['移动房车险数量'] == 0]
train_up = resample(train_up, n_samples=696, random_state=0)
train_down = resample(train_down, n_samples=1095, random_state=0)
train = shuffle(pd.concat([train_up, train_down]))
print(train['移动房车险数量'].value_counts())
```

运行结果：

balance_2.py U X

Code > balance_2.py > ...

```
1 from sklearn.utils import resample, shuffle
2 import pandas as pd
3 train = pd.read_excel('./Data/data.xlsx')
4 test = pd.read_excel('./Data/eval.xlsx')
5 train['source'] = 'train'
6 test['source'] = 'test'
7 data = pd.concat([train, test], ignore_index=True, sort=False)
8 print(train['移动房车险数量'].value_counts())
9 train_up = train[train['移动房车险数量'] == 1]
10 train_down = train[train['移动房车险数量'] == 0]
11 train_up = resample(train_up, n_samples=696, random_state=0)
12 train_down = resample(train_down, n_samples=1095, random_state=0)
13 train = shuffle(pd.concat([train_up, train_down]))
14 print(train['移动房车险数量'].value_counts())
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS

```
/usr/local/bin/python3 "/Users/youngbean/Documents/Github/Misc-Projects/Course I
● youngbean@YoungBeans-2 Project1 % /usr/local/bin/python3 "/Users/youngbean/Docum
balance_2.py"
移动房车险数量
0    5474
1     348
Name: count, dtype: int64
移动房车险数量
0    1095
1     696
Name: count, dtype: int64
```

重新测试:

```
import numpy as np
import pandas as pd
from sklearn import tree
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.naive_bayes import GaussianNB
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score
from sklearn.utils import resample, shuffle
```

训练不同模型

```
def build_models(x, y):
models = {
    "Decision Tree (ID3)": tree.DecisionTreeClassifier(criterion='entropy',
max_depth=10, min_samples_leaf=50),
    "Decision Tree (C4.5)": tree.DecisionTreeClassifier(criterion='entropy',
max_depth=10),
    "Random Forest": RandomForestClassifier(criterion='entropy', max_depth=10,
n_estimators=100),
    "MLP": MLPClassifier(hidden_layer_sizes=(50,), max_iter=500),
    "KNN": KNeighborsClassifier(n_neighbors=1),
    "Naive Bayes": GaussianNB(),
    "AdaBoost": AdaBoostClassifier(n_estimators=100),
    "Logistic Regression": LogisticRegression(solver='saga', penalty='l2',
max_iter=10000)
}
```

```

trained_models = {}
for name, model in models.items():
    print(f"Training {name}...")
    model.fit(x, y)
    trained_models[name] = model

return trained_models

# 评估模型
def evaluate_model(model, test_x, test_y):
    pred_y = model.predict(test_x)
    accuracy = accuracy_score(test_y, pred_y)
    precision = precision_score(test_y, pred_y, average='binary') # 根据数据类型调整
    recall = recall_score(test_y, pred_y, average='binary')
    f1 = f1_score(test_y, pred_y, average='binary')
    auc = roc_auc_score(test_y, pred_y)
    return accuracy, precision, recall, f1, auc

if __name__ == '__main__':
    train = pd.read_excel('./Data/data.xlsx')
    test = pd.read_excel('./Data/eval.xlsx')
    train['source'] = 'train'
    test['source'] = 'test'
    data = pd.concat([train, test], ignore_index=True, sort=False)

    train_up = train[train['移动房车险数量'] == 1]
    train_down = train[train['移动房车险数量'] == 0]
    train_up = resample(train_up, n_samples=696, random_state=0)
    train_down = resample(train_down, n_samples=1095, random_state=0)
    train = shuffle(pd.concat([train_up, train_down]))

    data = pd.concat([train, test], ignore_index=True, sort=False)

    numeric_train = train.select_dtypes(include=[np.number])

    # 计算相关系数
    corr_target = numeric_train.corr()['移动房车险数量']

    # 选择相关性绝对值 >= 0.01 的特征
    important_feature = corr_target[np.abs(corr_target) >= 0.01].index.tolist()

    train = train[important_feature]
    test = test[important_feature]
    train_x, train_y = train.iloc[:, :-1], train.iloc[:, -1]
    test_x, test_y = test.iloc[:, :-1], test.iloc[:, -1]

    # 训练所有模型
    models = build_models(train_x, train_y)

    # 依次测试每个模型并输出结果
    for name, model in models.items():
        accuracy, precision, recall, f1, auc = evaluate_model(model, test_x, test_y)
        print(f"\n{name} Results:")
        print(f'Accuracy %.4f' % accuracy)
        print(f'Precision %.4f' % precision)
        print(f'Recall %.4f' % recall)
        print(f'F1 Score %.4f' % f1)
        print(f'AUC %.4f' % auc)

```

结果对比:

平衡前	平衡后
<pre> ● youngbean@YoungBeans-2 Project1 % /usr/loca class_2.py" Training Decision Tree (ID3)... Training Decision Tree (C4.5)... Training Random Forest... Training MLP... Training KNN... Training Naive Bayes... Training AdaBoost... Training Logistic Regression... /Library/Frameworks/Python.framework/Versio ill-defined and being set to 0.0 due to no _warn_prf(average, modifier, msg_start, l Decision Tree (ID3) Results: Accuracy 0.9405 Precision 0.0000 Recall 0.0000 F1 Score 0.0000 AUC 0.5000 Decision Tree (C4.5) Results: Accuracy 0.9307 Precision 0.1803 Recall 0.0462 F1 Score 0.0736 AUC 0.5165 Random Forest Results: Accuracy 0.9403 Precision 0.0000 Recall 0.0000 F1 Score 0.0000 AUC 0.4999 MLP Results: Accuracy 0.9353 Precision 0.3019 Recall 0.0672 F1 Score 0.1100 AUC 0.5287 KNN Results: Accuracy 0.9038 Precision 0.1549 Recall 0.1387 F1 Score 0.1463 AUC 0.5454 Naive Bayes Results: Accuracy 0.4793 Precision 0.0839 Recall 0.7815 F1 Score 0.1515 AUC 0.6208 AdaBoost Results: Accuracy 0.9375 Precision 0.2273 Recall 0.0210 F1 Score 0.0385 AUC 0.5082 Logistic Regression Results: Accuracy 0.9397 Precision 0.3333 Recall 0.0126 F1 Score 0.0243 AUC 0.5055 </pre>	<pre> ● youngbean@YoungBeans-2 Project1 % classba_1.py" Training Decision Tree (ID3)... Training Decision Tree (C4.5)... Training Random Forest... Training MLP... /Library/Frameworks/Python.framework chastic Optimizer: Maximum iterati warnings.warn(Training KNN... Training Naive Bayes... Training AdaBoost... Training Logistic Regression... Decision Tree (ID3) Results: Accuracy 0.7235 Precision 0.1090 Recall 0.5084 F1 Score 0.1795 AUC 0.6228 Decision Tree (C4.5) Results: Accuracy 0.7678 Precision 0.1216 Recall 0.4664 F1 Score 0.1929 AUC 0.6266 Random Forest Results: Accuracy 0.8510 Precision 0.1558 Recall 0.3403 F1 Score 0.2137 AUC 0.6118 MLP Results: Accuracy 0.7983 Precision 0.1097 Recall 0.3361 F1 Score 0.1655 AUC 0.5818 KNN Results: Accuracy 0.7648 Precision 0.0974 Recall 0.3571 F1 Score 0.1530 AUC 0.5738 Naive Bayes Results: Accuracy 0.2260 Precision 0.0672 Recall 0.9328 F1 Score 0.1254 AUC 0.5570 AdaBoost Results: Accuracy 0.7788 Precision 0.1251 Recall 0.4538 F1 Score 0.1962 AUC 0.6265 Logistic Regression Results: Accuracy 0.7937 Precision 0.1399 Recall 0.4790 F1 Score 0.2165 AUC 0.6463 </pre>

结论:

在本次实验中,我们对比了数据平衡前后多个机器学习模型的表现,并观察到了显著的性能变化。在数据不平衡的情况下,大多数模型的准确率(Accuracy)较高,普遍接近 0.9 以上,但精确率(Precision)和召回率(Recall)出现了极端不均衡的情况。例如, ID3 决策树、C4.5 决策树和随机森林的召回率几乎为 0,说明它们几乎完全忽略了少数类样本,导致模型虽然整体预测正确率较高,但对少数类的预测效果极差。而在数据平衡之后,虽然部分模型的准确率有所下降,例如 ID3 决策树从 0.9405 降至 0.7235, C4.5 决策树从 0.9307 降至 0.7678, KNN 从 0.9038 降至 0.7648,但大多数模型的召回率大幅提升,尤其是 ID3 从 0.0000 增长到 0.5084, C4.5 从 0.0462 增长到 0.4664, KNN 从 0.1387 增长到 0.3571, 逻辑回归从 0.0126 增长到 0.4790, 朴素贝叶斯更是从 0.7815 提高到 0.9328。这表明数据平衡有效地提高了模型对少数类样本的识别能力,使得整体预测更加均衡。此外, F1 分数(F1 Score)和 AUC(曲线下面积)也有不同程度的改善,如 ID3 的 F1 Score 从 0.0000 提高到 0.1795, C4.5 从 0.0736 提高到 0.1929, 逻辑回归从 0.0243 提高到 0.2165, AUC 也从 0.5055 提升到 0.6463, 这表明模型的泛化能力增强。在具体模型选择上,随机森林在各项指标上较为均衡,适合作为基准模型;逻辑回归在数据平衡后表现明显改善,更适用于需要权衡 Precision 和 Recall 的场景;朴素贝叶斯虽然召回率极高,但精确率下降,可能更适用于需要高召回率的任务。总体而言,数据平衡后,模型不再过度偏向多数类,使得预测更加可靠,在实际应用中更符合业务需求。

7.算法调参

1) ID3 决策树的最大深度

代码:

```
import numpy as np
import pandas as pd
from sklearn import tree
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score
from sklearn.utils import resample, shuffle

# 评估模型
def evaluate_model(model, test_x, test_y):
    pred_y = model.predict(test_x)
    accuracy = accuracy_score(test_y, pred_y)
    precision = precision_score(test_y, pred_y, average='binary')
    recall = recall_score(test_y, pred_y, average='binary')
    f1 = f1_score(test_y, pred_y, average='binary')
    auc = roc_auc_score(test_y, pred_y)
    return accuracy, precision, recall, f1, auc

if __name__ == '__main__':
    train = pd.read_excel('./Data/data.xlsx')
    test = pd.read_excel('./Data/eval.xlsx')
    train['source'] = 'train'
    test['source'] = 'test'
    data = pd.concat([train, test], ignore_index=True, sort=False)

    train_up = train[train['移动房车险数量'] == 1]
    train_down = train[train['移动房车险数量'] == 0]
    train_up = resample(train_up, n_samples=696, random_state=0)
    train_down = resample(train_down, n_samples=1095, random_state=0)
    train = shuffle(pd.concat([train_up, train_down]))
```

```

data = pd.concat([train, test], ignore_index=True, sort=False)
numeric_train = train.select_dtypes(include=[np.number])

# 计算相关系数
corr_target = numeric_train.corr()['移动房车险数量']

# 选择相关性绝对值 >= 0.01 的特征
important_feature = corr_target[np.abs(corr_target) >= 0.01].index.tolist()

train = train[important_feature]
test = test[important_feature]
train_x, train_y = train.iloc[:, :-1], train.iloc[:, -1]
test_x, test_y = test.iloc[:, :-1], test.iloc[:, -1]

# 依次测试 ID3 决策树在不同 max_depth 下的效果
for max_depth in [5, 10, 15, 20]:
    print(f"\nTraining Decision Tree (ID3) with max_depth={max_depth}...")
    model = tree.DecisionTreeClassifier(criterion='entropy', max_depth=max_depth,
    min_samples_leaf=50)
    model.fit(train_x, train_y)

accuracy, precision, recall, f1, auc = evaluate_model(model, test_x, test_y)
print(f"\nDecision Tree (ID3) Results with max_depth={max_depth}:")
print(f'Accuracy %.4f' % accuracy)
print(f'Precision %.4f' % precision)
print(f'Recall %.4f' % recall)
print(f'F1 Score %.4f' % f1)
print(f'AUC %.4f' % auc)

```

结果对比：

5	10	15	20
Decision Tree (ID3)	Decision Tree (ID3)	Decision Tree (ID3)	Decision Tree (ID3)
Accuracy 0.7568	Accuracy 0.7235	Accuracy 0.7235	Accuracy 0.7235
Precision 0.1231	Precision 0.1090	Precision 0.1090	Precision 0.1090
Recall 0.5042	Recall 0.5084	Recall 0.5084	Recall 0.5084
F1 Score 0.1979	F1 Score 0.1795	F1 Score 0.1795	F1 Score 0.1795
AUC 0.6385	AUC 0.6228	AUC 0.6228	AUC 0.6228

结论：在本次实验中，我们针对 ID3 决策树在不同最大深度（max_depth = 5、10、15、20）下的分类性能进行了对比分析。结果表明，随着 max_depth 的增加，模型的准确率（Accuracy）整体呈下降趋势，从 0.7568 降至 0.7235，这表明模型的复杂度增加后，并未带来更好的整体预测效果。此外，精确率（Precision）也从 0.1231 降至 0.1090，说明随着决策树的加深，模型的错误警报率有所增加，错误分类为正类的样本比例增大。另一方面，召回率（Recall）基本保持稳定，在 max_depth = 10 及以上时维持在 0.5084，而 max_depth = 5 时的召回率为 0.5042，表明树的深度对少数类的识别能力影响不大。F1 分数（F1 Score）和 AUC 也表现出类似的趋势，其中 F1 Score 在 max_depth = 5 时最高（0.1979），之后下降并趋于稳定（0.1795）。AUC 作为衡量模型区分能力的指标，从 0.6385 降至 0.6228，也表明模型的整体性能在深度增加后并未获得显著提升。综合来看，max_depth = 5 时，ID3 决策树的整体表现较为均衡，既能保证一定的召回率，又能提供相对较高的精确率和 F1 分数。而在 max_depth 进一步增大后，虽然召回率稳定，但精确率和 AUC 出现下降，说明模型复杂度增加可能导致过拟合，从而影响泛化能力。因此，在本实验的数据集上，选择较小的 max_depth，如 5 或 10，更有利于模型的泛化能力，

而过深的树结构可能会导致模型性能下降，降低在测试集上的分类效果。

2) ID3 决策树随机调整切分时考虑的特征数、内部节点分裂时最少样本数、叶节点最少样本数等参数

代码:

```
import numpy as np
import pandas as pd
from sklearn import tree
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score
from sklearn.utils import resample, shuffle

# 评估模型
def evaluate_model(model, test_x, test_y):
    pred_y = model.predict(test_x)
    accuracy = accuracy_score(test_y, pred_y)
    precision = precision_score(test_y, pred_y, average='binary')
    recall = recall_score(test_y, pred_y, average='binary')
    f1 = f1_score(test_y, pred_y, average='binary')
    auc = roc_auc_score(test_y, pred_y)
    return accuracy, precision, recall, f1, auc

if __name__ == '__main__':
    train = pd.read_excel('./Data/data.xlsx')
    test = pd.read_excel('./Data/eval.xlsx')
    train['source'] = 'train'
    test['source'] = 'test'
    data = pd.concat([train, test], ignore_index=True, sort=False)

    train_up = train[train['移动房车险数量'] == 1]
    train_down = train[train['移动房车险数量'] == 0]
    train_up = resample(train_up, n_samples=696, random_state=0)
    train_down = resample(train_down, n_samples=1095, random_state=0)
    train = shuffle(pd.concat([train_up, train_down]))

    data = pd.concat([train, test], ignore_index=True, sort=False)

    numeric_train = train.select_dtypes(include=[np.number])

    # 计算相关系数
    corr_target = numeric_train.corr()['移动房车险数量']

    # 选择相关性绝对值 >= 0.01 的特征
    important_feature = corr_target[np.abs(corr_target) >= 0.01].index.tolist()
    train = train[important_feature]
    test = test[important_feature]
    train_x, train_y = train.iloc[:, :-1], train.iloc[:, -1]
    test_x, test_y = test.iloc[:, :-1], test.iloc[:, -1]

    # 设定不同参数的模型
    models = {
        "Original ID3": tree.DecisionTreeClassifier(criterion='entropy', max_depth=5),
        "ID3 (max_features=20)": tree.DecisionTreeClassifier(criterion='entropy',
max_depth=5, max_features=20),
        "ID3 (min_samples_split=50)":
tree.DecisionTreeClassifier(criterion='entropy', max_depth=5,
min_samples_split=50),
        "ID3 (min_samples_leaf=50)": tree.DecisionTreeClassifier(criterion='entropy',
```

```

max_depth=5, min_samples_leaf=50)
}

# 训练并评估每个模型
for name, model in models.items():
    print(f"\nTraining {name}...")
    model.fit(train_x, train_y)
    accuracy, precision, recall, f1, auc = evaluate_model(model, test_x, test_y)
    print(f"\n{name} Results:")
    print(f'Accuracy %.4f' % accuracy)
    print(f'Precision %.4f' % precision)
    print(f'Recall %.4f' % recall)
    print(f'F1 Score %.4f' % f1)
    print(f'AUC %.4f' % auc)

```

结果对比：

原始模型 (ID3 决策树最大深度=5)	切分时考虑的特征数=20	内部节点分裂时最少样本数=50	叶节点最少样本数=50
Original ID3 Results: Accuracy 0.8150 Precision 0.1414 Recall 0.4160 F1 Score 0.2111 AUC 0.6281	ID3 (max_features=20) Results: Accuracy 0.7785 Precision 0.1301 Recall 0.4790 F1 Score 0.2047 AUC 0.6382	ID3 (min_samples_split=50) Results: Accuracy 0.8075 Precision 0.1396 Recall 0.4328 F1 Score 0.2111 AUC 0.6320	ID3 (min_samples_leaf=50) Results: Accuracy 0.7568 Precision 0.1231 Recall 0.5042 F1 Score 0.1979 AUC 0.6385

结论：

在本次实验中，我们通过调整 ID3 决策树的不同超参数，包括切分时考虑的特征数（max_features）、内部节点分裂时最少样本数（min_samples_split）以及叶节点最少样本数（min_samples_leaf），分析了其对模型性能的影响。首先，以 max_depth=5 作为基准模型，其准确率（Accuracy）为 0.8150，精确率（Precision）为 0.1414，召回率（Recall）为 0.4160，F1 分数（F1 Score）为 0.2111，AUC 值为 0.6281。当限制切分时考虑的特征数为 20（max_features=20）时，准确率下降至 0.7785，精确率下降至 0.1301，但召回率提高至 0.4790，AUC 也有所提升至 0.6382，表明模型在一定程度上增加了对少数类的识别能力，但整体准确率有所下降。进一步调整内部节点分裂的最少样本数（min_samples_split=50）后，模型的准确率提升至 0.8075，精确率为 0.1396，召回率为 0.4328，F1 分数保持在 0.2111，AUC 也小幅上升至 0.6320，说明适当增加分裂节点的最小样本数有助于提高模型的稳定性和泛化能力。最后，当设置叶节点的最少样本数为 50（min_samples_leaf=50）时，准确率下降至 0.7568，精确率下降至 0.1231，而召回率上升至 0.5042，AUC 值略微提升至 0.6385，这表明虽然模型在召回率方面有所提升，但整体的分类准确率和精确率有所下降。综合来看，适当增加内部节点的最小分裂样本数（如 50）可以在保持较高准确率的同时，提升模型的稳定性，而增加叶节点的最少样本数或限制特征数可能会在一定程度上提高召回率，但可能导致整体准确率下降。因此，在实际应用中，需要根据具体任务需求在准确率、召回率和模型复杂度之间进行权衡选择。

3) 逻辑回归模型，修改参数惩罚性进行实验，分别设置为 L1 惩罚与 L2 惩罚

代码：

```

import numpy as np
import pandas as pd
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, precision_score, recall_score,

```

```

f1_score, roc_auc_score
from sklearn.utils import resample, shuffle

# 评估模型
def evaluate_model(model, test_x, test_y):
    pred_y = model.predict(test_x)
    accuracy = accuracy_score(test_y, pred_y)
    precision = precision_score(test_y, pred_y, average='binary')
    recall = recall_score(test_y, pred_y, average='binary')
    f1 = f1_score(test_y, pred_y, average='binary')
    auc = roc_auc_score(test_y, pred_y)
    return accuracy, precision, recall, f1, auc

if __name__ == '__main__':
    train = pd.read_excel('./Data/data.xlsx')
    test = pd.read_excel('./Data/eval.xlsx')
    train['source'] = 'train'
    test['source'] = 'test'
    data = pd.concat([train, test], ignore_index=True, sort=False)

    train_up = train[train['移动房车险数量'] == 1]
    train_down = train[train['移动房车险数量'] == 0]
    train_up = resample(train_up, n_samples=696, random_state=0)
    train_down = resample(train_down, n_samples=1095, random_state=0)
    train = shuffle(pd.concat([train_up, train_down]))

    data = pd.concat([train, test], ignore_index=True, sort=False)

    numeric_train = train.select_dtypes(include=[np.number])

    # 计算相关系数
    corr_target = numeric_train.corr()['移动房车险数量']

    # 选择相关性绝对值 >= 0.01 的特征
    important_feature = corr_target[np.abs(corr_target) >= 0.01].index.tolist()
    train = train[important_feature]
    test = test[important_feature]
    train_x, train_y = train.iloc[:, :-1], train.iloc[:, -1]
    test_x, test_y = test.iloc[:, :-1], test.iloc[:, -1]

    # 设定不同惩罚方式的逻辑回归模型
    models = {
        "Logistic Regression (L1 penalty)": LogisticRegression(solver='saga',
            penalty='l1', max_iter=10000),
        "Logistic Regression (L2 penalty)": LogisticRegression(solver='saga',
            penalty='l2', max_iter=10000)
    }

    # 训练并评估每个模型
    for name, model in models.items():
        print(f"\nTraining {name}...")
        model.fit(train_x, train_y)
        accuracy, precision, recall, f1, auc = evaluate_model(model, test_x, test_y)
        print(f"\n{name} Results:")
        print(f'Accuracy %.4f' % accuracy)
        print(f'Precision %.4f' % precision)
        print(f'Recall %.4f' % recall)
        print(f'F1 Score %.4f' % f1)
        print(f'AUC %.4f' % auc)

```

结果对比：

L1 Penalty	L2 Penalty
Logistic Regression Accuracy 0.7957 Precision 0.1395 Recall 0.4706 F1 Score 0.2152 AUC 0.6435	Logistic Regression Accuracy 0.7937 Precision 0.1399 Recall 0.4790 F1 Score 0.2165 AUC 0.6463

结论：

在本实验中，我们采用逻辑回归模型，通过分别设置 L1 惩罚（Lasso）和 L2 惩罚（Ridge）来探究不同正则化方式对模型性能的影响。从实验结果来看，L1 惩罚下的逻辑回归模型在测试集上的分类准确率（Accuracy）为 0.7957，L2 惩罚下的分类准确率为 0.7937，二者相差 0.002，说明两种正则化方式在整体分类正确率上的影响较小。然而，在 Precision（精准率）方面，L2 惩罚（0.1399）略高于 L1 惩罚（0.1395），但 Recall（召回率）方面 L1 惩罚（0.4706）比 L2 惩罚（0.4790）略低，这表明 L2 惩罚在识别正类时略有优势。此外，从 F1 分数来看，L2 惩罚（0.2165）高于 L1 惩罚（0.2152），说明 L2 惩罚在 Precision 和 Recall 之间的平衡性更好，具有更强的综合分类能力。同时，AUC 方面，L1 惩罚的 AUC 值为 0.6435，而 L2 惩罚的 AUC 值为 0.6463，表明 L2 正则化在整体的分类能力上略优于 L1 惩罚。

结合这些指标来看，L1 惩罚能够执行特征选择，即使部分特征的系数变为 0，从而提升模型的稀疏性，适用于高维特征场景，但在本实验的数据集上，这种正则化并未带来明显的优势。而 L2 惩罚通过对所有特征施加均匀的约束，使模型更具鲁棒性，在当前数据集上略胜一筹。因此，在实际应用中，如果目标是提升模型的泛化能力并保留更多特征信息，L2 惩罚可能是更优的选择；而如果希望进行特征筛选并增强模型的可解释性，可以考虑 L1 惩罚。

3) 多层感知机加入参数，实验验证，调整参数，结果分析

代码：

```
import numpy as np
import pandas as pd
from sklearn.neural_network import MLPClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score
from sklearn.utils import resample, shuffle

# 评估模型
def evaluate_model(model, test_x, test_y):
    pred_y = model.predict(test_x)
    accuracy = accuracy_score(test_y, pred_y)
    precision = precision_score(test_y, pred_y, average='binary')
    recall = recall_score(test_y, pred_y, average='binary')
    f1 = f1_score(test_y, pred_y, average='binary')
    auc = roc_auc_score(test_y, pred_y)
    return accuracy, precision, recall, f1, auc

if __name__ == '__main__':
    train = pd.read_excel('./Data/data.xlsx')
    test = pd.read_excel('./Data/eval.xlsx')
    train['source'] = 'train'
    test['source'] = 'test'
    data = pd.concat([train, test], ignore_index=True, sort=False)
```

```

train_up = train[train['移动房车险数量'] == 1]
train_down = train[train['移动房车险数量'] == 0]
train_up = resample(train_up, n_samples=696, random_state=0)
train_down = resample(train_down, n_samples=1095, random_state=0)
train = shuffle(pd.concat([train_up, train_down]))

data = pd.concat([train, test], ignore_index=True, sort=False)

numeric_train = train.select_dtypes(include=[np.number])

# 计算相关系数
corr_target = numeric_train.corr()['移动房车险数量']

# 选择相关性绝对值 >= 0.01 的特征
important_feature = corr_target[np.abs(corr_target) >= 0.01].index.tolist()
train = train[important_feature]
test = test[important_feature]
train_x, train_y = train.iloc[:, :-1], train.iloc[:, -1]
test_x, test_y = test.iloc[:, :-1], test.iloc[:, -1]

# 定义 MLP 变体
models = {
    "MLP Baseline": MLPClassifier(hidden_layer_sizes=(100,), activation='relu',
    solver='adam', learning_rate_init=0.001, max_iter=200, random_state=0),
    "MLP More Layers": MLPClassifier(hidden_layer_sizes=(100, 50),
    activation='relu', solver='adam', learning_rate_init=0.001, max_iter=200,
    random_state=0),
    "MLP Higher LR": MLPClassifier(hidden_layer_sizes=(100,), activation='relu',
    solver='adam', learning_rate_init=0.005, max_iter=200, random_state=0),
    "MLP More Iterations": MLPClassifier(hidden_layer_sizes=(100,),
    activation='relu', solver='adam', learning_rate_init=0.001, max_iter=500,
    random_state=0)
}

# 训练并评估每个模型
for name, model in models.items():
    print(f"\nTraining {name}...")
    model.fit(train_x, train_y)
    accuracy, precision, recall, f1, auc = evaluate_model(model, test_x, test_y)
    print(f"\n{name} Results:")
    print(f'Accuracy %.4f' % accuracy)
    print(f'Precision %.4f' % precision)
    print(f'Recall %.4f' % recall)
    print(f'F1 Score %.4f' % f1)
    print(f'AUC %.4f' % auc)

```

结果对比：

多层感知机隐层 =1*100 优化器 Adam 激活函数 Relu 初始学习率=0.001 迭代次数 =200	增加隐藏层, (100, 50) (2 层, 100 和 50 个神经元)	提高学习率 0.005	增大训练迭代次数 500
MLP Baseline Result Accuracy 0.8027 Precision 0.1189 Recall 0.3613 F1 Score 0.1790 AUC 0.5960	MLP More Layers Result Accuracy 0.7853 Precision 0.1124 Recall 0.3782 F1 Score 0.1732 AUC 0.5946	MLP Higher LR Result Accuracy 0.8407 Precision 0.1326 Recall 0.3025 F1 Score 0.1844 AUC 0.5887	MLP More Iterations Result Accuracy 0.8193 Precision 0.1169 Recall 0.3109 F1 Score 0.1699 AUC 0.5812

结论:

在本实验中,我们针对多层感知机 (MLP) 模型的不同超参数进行了实验,包括增加隐藏层数量、提高学习率、以及增加训练迭代次数,以探究这些超参数对模型性能的影响。基准模型 (MLP Baseline) 使用单隐藏层 (100 个神经元), Adam 优化器,学习率 0.001,最大迭代次数 200,得到的准确率为 0.8027, Precision 为 0.1189, Recall 为 0.3613, F1 分数为 0.1790, AUC 为 0.5960。对比实验中,当增加隐藏层 (100, 50) 时,模型的准确率下降至 0.7853, Precision 降低到 0.1124, Recall 提升到 0.3782, F1 分数下降至 0.1732, AUC 轻微下降到 0.5946,表明在本实验设置下,增加隐藏层并未显著提高模型性能,反而可能导致一定程度的过拟合或梯度消失问题。

当提高学习率至 0.005 时,模型的准确率上升到 0.8407, Precision 提升至 0.1326, 但 Recall 降低至 0.3025, F1 分数仅小幅提升至 0.1844, AUC 下降至 0.5887,说明更高的学习率可能使模型收敛速度更快,但也可能导致模型在局部最优点附近震荡, Recall 的降低可能意味着模型在识别正样本时有所下降。

此外,当增加最大训练迭代次数至 500 时,模型的准确率达到 0.8193, Precision 为 0.1169, Recall 为 0.3109, F1 分数为 0.1699, AUC 下降至 0.5812,相比基准模型,尽管准确率有所提升,但 Precision、Recall 和 F1 分数均未明显改善, AUC 反而有所下降,说明增加迭代次数在当前学习率下未能带来明显收益,可能是因为模型已经较快收敛,额外的训练未能进一步优化性能。

综合来看,基准模型的整体性能较为稳定,增加隐藏层未能带来明显增益,可能受限于数据量或特征表达能力;提高学习率能够提升训练速度,但可能导致模型的不稳定性;增加迭代次数虽然可以延长训练,但未能有效改善模型的泛化能力。因此,在本实验中,选择合适的学习率与迭代次数较为关键,而增加网络深度的效果需要结合具体任务进行评估和调整。

5) K 近邻 (KNN) 分类器的参数 K

代码:

```
import numpy as np
import pandas as pd
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score
from sklearn.utils import resample, shuffle

# 评估模型
def evaluate_model(model, test_x, test_y):
    pred_y = model.predict(test_x)
    accuracy = accuracy_score(test_y, pred_y)
    precision = precision_score(test_y, pred_y, average='binary')
    recall = recall_score(test_y, pred_y, average='binary')
    f1 = f1_score(test_y, pred_y, average='binary')
    auc = roc_auc_score(test_y, pred_y)
    return accuracy, precision, recall, f1, auc

if __name__ == '__main__':
    train = pd.read_excel('./Data/data.xlsx')
    test = pd.read_excel('./Data/eval.xlsx')
```



```

train['source'] = 'train'
test['source'] = 'test'
data = pd.concat([train, test], ignore_index=True, sort=False)

train_up = train[train['移动房车险数量'] == 1]
train_down = train[train['移动房车险数量'] == 0]
train_up = resample(train_up, n_samples=696, random_state=0)
train_down = resample(train_down, n_samples=1095, random_state=0)
train = shuffle(pd.concat([train_up, train_down]))

data = pd.concat([train, test], ignore_index=True, sort=False)

numeric_train = train.select_dtypes(include=[np.number])

# 计算相关系数
corr_target = numeric_train.corr()['移动房车险数量']

# 选择相关性绝对值 >= 0.01 的特征
important_feature = corr_target[np.abs(corr_target) >= 0.01].index.tolist()
train = train[important_feature]
test = test[important_feature]
train_x, train_y = train.iloc[:, :-1], train.iloc[:, -1]
test_x, test_y = test.iloc[:, :-1], test.iloc[:, -1]

# 定义 KNN 变体
models = {
    "KNN (K=5, Default)": KNeighborsClassifier(n_neighbors=5),
    "KNN (K=1)": KNeighborsClassifier(n_neighbors=1),
    "KNN (K=10)": KNeighborsClassifier(n_neighbors=10),
    "KNN (K=20)": KNeighborsClassifier(n_neighbors=20)
}

# 训练并评估每个模型
for name, model in models.items():
    print(f"\nTraining {name}...")
    model.fit(train_x, train_y)
    accuracy, precision, recall, f1, auc = evaluate_model(model, test_x, test_y)
    print(f"\n{name} Results:")
    print(f'Accuracy %.4f' % accuracy)
    print(f'Precision %.4f' % precision)
    print(f'Recall %.4f' % recall)
    print(f'F1 Score %.4f' % f1)
    print(f'AUC %.4f' % auc)

```

结果对比：

K=5	K=1	K=10	K=20
KNN (K=5, Default) Accuracy 0.6987 Precision 0.0864 Recall 0.4244 F1 Score 0.1436 AUC 0.5702	KNN (K=1) Results: Accuracy 0.7668 Precision 0.1010 Recall 0.3697 F1 Score 0.1587 AUC 0.5808	KNN (K=10) Results: Accuracy 0.7668 Precision 0.1152 Recall 0.4370 F1 Score 0.1823 AUC 0.6123	KNN (K=20) Results: Accuracy 0.7672 Precision 0.1128 Recall 0.4244 F1 Score 0.1783 AUC 0.6067

结论：

在本实验中，我们探讨了 K 近邻（KNN）分类器的超参数 K 对模型性能的影响。我们选择了 K=1、K=5（默认值）、K=10 和 K=20 进行对比实验，并基于准确率（Accuracy）、精确率（Precision）、召回率（Recall）、F1 分数（F1 Score）和 AUC 评价模型表现。

实验结果显示，默认的 K=5 时，模型的准确率为 0.6987，Precision 为 0.0864，Recall 为 0.4244，

F1 分数为 0.1436, AUC 为 0.5702。相较之下, 当 K=1 时, 模型的准确率提升到 0.7668, Precision 为 0.1010, Recall 降至 0.3697, F1 分数略微上升至 0.1587, AUC 也提高至 0.5808。这表明在 K=1 的情况下, 模型可以更好地拟合训练数据, 但可能会对噪声更加敏感, 导致泛化能力较弱。

当 K 增大到 10 时, 准确率进一步提高到 0.7668, Precision 提升到 0.1152, Recall 为 0.4370, F1 分数升至 0.1823, AUC 也提升至 0.6123, 表明适当增加 K 值有助于提升模型的稳定性和泛化能力, 减少过拟合问题。当 K=20 时, 模型的准确率仍然维持在较高水平 (0.7672), Precision 为 0.1128, Recall 为 0.4244, F1 分数略微下降至 0.1783, AUC 进一步提升至 0.6067, 说明较大的 K 值可以进一步平滑决策边界, 提高模型鲁棒性, 但 Precision 和 F1 分数略有下降, 可能是因为过多的邻居投票导致决策边界变得模糊, 从而降低对少数类的识别能力。

总体来看, 较小的 K (如 K=1) 能够更好地拟合训练数据, 但容易受噪声影响, 泛化能力较弱; K=5 作为默认值, 表现相对较均衡, 但性能并不是最优; 适当增大 K (如 K=10 或 K=20) 可以提升准确率和 AUC, 但 Precision 和 F1 分数可能会有所下降。因此, K 值的选择需要在模型复杂度和泛化能力之间进行权衡, 针对不同的任务和数据分布, 选择合适的 K 值是关键。

6) 使用高斯朴素贝叶斯分类器

代码:

```
import numpy as np
import pandas as pd
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score
from sklearn.utils import resample, shuffle
```

评估模型

```
def evaluate_model(model, test_x, test_y):
    pred_y = model.predict(test_x)
    accuracy = accuracy_score(test_y, pred_y)
    precision = precision_score(test_y, pred_y, average='binary')
    recall = recall_score(test_y, pred_y, average='binary')
    f1 = f1_score(test_y, pred_y, average='binary')
    auc = roc_auc_score(test_y, pred_y)
    return accuracy, precision, recall, f1, auc
```

```
if __name__ == '__main__':
```

读取数据

```
train = pd.read_excel('./Data/data.xlsx')
test = pd.read_excel('./Data/eval.xlsx')
train['source'] = 'train'
test['source'] = 'test'
data = pd.concat([train, test], ignore_index=True, sort=False)
```

处理类别不平衡问题

```
train_up = train[train['移动房车险数量'] == 1]
train_down = train[train['移动房车险数量'] == 0]
train_up = resample(train_up, n_samples=696, random_state=0)
train_down = resample(train_down, n_samples=1095, random_state=0)
train = shuffle(pd.concat([train_up, train_down]))
```

```
data = pd.concat([train, test], ignore_index=True, sort=False)
```

选择数值特征

```
numeric_train = train.select_dtypes(include=[np.number])
```

计算相关系数

```
corr_target = numeric_train.corr()['移动房车险数量']
```



```

# 选择相关性绝对值 >= 0.01 的特征
important_feature = corr_target[np.abs(corr_target) >= 0.01].index.tolist()
train = train[important_feature]
test = test[important_feature]
train_x, train_y = train.iloc[:, :-1], train.iloc[:, -1]
test_x, test_y = test.iloc[:, :-1], test.iloc[:, -1]

# 定义 Gaussian Naive Bayes 模型
model = GaussianNB()

# 训练模型
print("\nTraining Gaussian Naive Bayes Model...")
model.fit(train_x, train_y)

# 评估模型
accuracy, precision, recall, f1, auc = evaluate_model(model, test_x, test_y)
print("\nGaussian Naive Bayes Results:")
print(f'Accuracy %.4f' % accuracy)
print(f'Precision %.4f' % precision)
print(f'Recall %.4f' % recall)
print(f'F1 Score %.4f' % f1)
print(f'AUC %.4f' % auc)

```

运行结果：

```

Gaussian Naive Bayes I
Accuracy  0.2260
Precision 0.0672
Recall    0.9328
F1 Score  0.1254
AUC       0.5570

```

结论：

在本实验中，我们使用高斯朴素贝叶斯（Gaussian Naive Bayes, GNB）分类器对数据进行了分类任务，并基于准确率（Accuracy）、精确率（Precision）、召回率（Recall）、F1 分数（F1 Score）和 AUC 进行模型性能评估。实验结果显示，该模型的准确率仅为 0.2260，Precision 也较低，仅有 0.0672，然而 Recall 却高达 0.9328，F1 分数为 0.1254，而 AUC 达到了 0.5570。这一结果反映出 GNB 分类器在本数据集上的表现存在明显的偏差，主要表现为召回率极高，而精确率极低，导致整体的 F1 分数较低。

这一现象可能是由于朴素贝叶斯模型的假设较强，它假设所有特征之间是条件独立的，而在实际应用中，这一假设往往难以满足。此外，该模型对于数据的类别不平衡问题较为敏感，即使我们已经在训练集中使用了重采样方法平衡类别比例，GNB 依然倾向于对少数类（类别 1）做出较高的预测概率，因此导致了极高的 Recall，但 Precision 却较低。这说明 GNB 在识别正例（类别 1）方面具有较强的能力，但同时也带来了大量的误报（False Positives），使得精确率降低。

相比于 KNN 分类器的实验结果，GNB 在 AUC 指标上表现略逊色于 K=10 和 K=20 的 KNN 模型（AUC 约为 0.6067），但比默认的 K=5（AUC = 0.5702）稍低一些。这表明 GNB 在整体分类能力上仍有一定的价值，但可能更适合用于 Recall 优先的任务，而不太适用于 Precision 需求较高的任务。

综上所述，GNB 在本实验数据上的表现并不理想，尤其是极低的 Precision 限制了其实用性。如果目标是提高 Precision，可以尝试调整决策阈值、使用特征选择方法减少冗余信息，或者换

用其他更适合的分类算法，如逻辑回归、决策树或随机森林，以提高模型的综合性能。

7) AdaBoosting 调整参数

代码:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score
from sklearn.utils import resample, shuffle

# 评估模型
def evaluate_model(model, test_x, test_y):
    pred_y = model.predict(test_x)
    accuracy = accuracy_score(test_y, pred_y)
    precision = precision_score(test_y, pred_y, average='binary')
    recall = recall_score(test_y, pred_y, average='binary')
    f1 = f1_score(test_y, pred_y, average='binary')
    auc = roc_auc_score(test_y, pred_y)
    return accuracy, precision, recall, f1, auc

if __name__ == '__main__':
    # 读取数据
    train = pd.read_excel('./Data/data.xlsx')
    test = pd.read_excel('./Data/eval.xlsx')
    train['source'] = 'train'
    test['source'] = 'test'
    data = pd.concat([train, test], ignore_index=True, sort=False)

    # 处理类别不平衡问题
    train_up = train[train['移动房车险数量'] == 1]
    train_down = train[train['移动房车险数量'] == 0]
    train_up = resample(train_up, n_samples=696, random_state=0)
    train_down = resample(train_down, n_samples=1095, random_state=0)
    train = shuffle(pd.concat([train_up, train_down]))

    data = pd.concat([train, test], ignore_index=True, sort=False)

    # 选择数值特征
    numeric_train = train.select_dtypes(include=[np.number])

    # 计算相关系数
    corr_target = numeric_train.corr()['移动房车险数量']

    # 选择相关性绝对值 >= 0.01 的特征
    important_feature = corr_target[np.abs(corr_target) >= 0.01].index.tolist()
    train = train[important_feature]
    test = test[important_feature]
    train_x, train_y = train.iloc[:, :-1], train.iloc[:, -1]
    test_x, test_y = test.iloc[:, :-1], test.iloc[:, -1]

    # 超参数列表
    n_estimators_list = [50, 100, 200]
    learning_rate_list = [0.01, 0.1, 1.0]

    # 遍历不同的超参数组合
    for n_estimators in n_estimators_list:
        for learning_rate in learning_rate_list:
            print(f"\nTraining AdaBoost with n_estimators={n_estimators},
```

```

learning_rate={learning_rate} ...)

# 定义 AdaBoost 分类器（使用决策树作为基学习器）
model = AdaBoostClassifier(
    base_estimator=DecisionTreeClassifier(max_depth=1), # 使用简单的决策树作为弱分类器
    n_estimators=n_estimators,
    learning_rate=learning_rate,
    random_state=0
)

# 训练模型
model.fit(train_x, train_y)

# 评估模型
accuracy, precision, recall, f1, auc = evaluate_model(model, test_x, test_y)
print(f"\nAdaBoost Results (n_estimators={n_estimators},
learning_rate={learning_rate}):")
print(f'Accuracy %.4f' % accuracy)
print(f'Precision %.4f' % precision)
print(f'Recall %.4f' % recall)
print(f'F1 Score %.4f' % f1)
print(f'AUC %.4f' % auc)

```

结果对比：

	基分类器 n=50	基分类器 n=100	基分类器 n=200
学习率=0.01	AdaBoost Results (n Accuracy 0.6162 Precision 0.0989 Recall 0.6723 F1 Score 0.1725 AUC 0.6425	AdaBoost Results (n Accuracy 0.8575 Precision 0.1653 Recall 0.3445 F1 Score 0.2234 AUC 0.6172	AdaBoost Results (n Accuracy 0.8535 Precision 0.1602 Recall 0.3445 F1 Score 0.2187 AUC 0.6151
学习率=0.1	AdaBoost Results (n Accuracy 0.8353 Precision 0.1555 Recall 0.3992 F1 Score 0.2238 AUC 0.6310	AdaBoost Results (n Accuracy 0.8187 Precision 0.1456 Recall 0.4202 F1 Score 0.2162 AUC 0.6321	AdaBoost Results (n Accuracy 0.8133 Precision 0.1390 Recall 0.4118 F1 Score 0.2078 AUC 0.6252
学习率=1.0	AdaBoost Results (n Accuracy 0.7867 Precision 0.1255 Recall 0.4328 F1 Score 0.1945 AUC 0.6210	AdaBoost Results (n Accuracy 0.7788 Precision 0.1251 Recall 0.4538 F1 Score 0.1962 AUC 0.6265	AdaBoost Results (n Accuracy 0.7820 Precision 0.1226 Recall 0.4328 F1 Score 0.1911 AUC 0.6184

结论：

在本实验中，我们使用 AdaBoost 结合决策树（深度为 1）作为基分类器，并针对超参数 `n_estimators`（基分类器数量）和 `learning_rate`（学习率）进行了调整，以分析其对分类性能的影响。实验结果表明，`n_estimators` 和 `learning_rate` 的不同组合对模型的准确率（Accuracy）、精确率（Precision）、召回率（Recall）、F1 分数（F1 Score）和 AUC 造成了明显的影响。

从实验结果来看，学习率较小时（0.01），模型的召回率较高（最高达到 0.6723），但精确率极低（最低 0.0989），导致 F1 分数较低，说明该模型更倾向于识别正例（类别 1），但误报较多（False Positives 较高）。同时，AUC 最高仅为 0.6425，说明模型整体的区分能力一般，尽管召回率高，但泛化能力较弱。

当学习率增加至 0.1 时，模型的准确率有显著提升，特别是 `n_estimators=50` 时达到了 0.8353，而 `n_estimators=100` 和 `n_estimators=200` 也能保持在 0.81 左右。然而，随着学习率的提升，召回率有所下降（最低降至 0.3445），但精确率相对提升（最高 0.1653）。这表明较高的学习率帮助模型减少了误报，使精确率提升，但同时降低了对正例的敏感性（召回率下降）。

当学习率进一步增加至 1.0 时，模型的准确率有所下降，最低降至 0.7786，Precision 也降低到 0.1255 左右，Recall 最高仅有 0.4538，AUC 也没有显著提高。这表明学习率过高可能会导致模型的不稳定，使得模型的整体性能下降，尤其是精确率和 AUC 下降较为明显。

此外，随着基分类器数量增加（从 50 到 200），整体的 AUC 并未出现明显提升，说明增加基分类器数量并不能明显提升 AdaBoost 的分类能力，甚至在某些情况下可能会导致性能下降（如 `n_estimators=200, learning_rate=1.0` 时，Accuracy 和 AUC 均下降）。这可能是由于基分类器数量过多导致模型对训练数据过拟合，从而降低泛化能力。

综合来看，在本数据集上，AdaBoost 的最佳参数组合是 `learning_rate=0.1, n_estimators=50`，在该配置下，模型的准确率最高 (0.8353)，AUC 也保持在较好的水平 (0.6310)，并且 Precision 和 Recall 取得了较好的平衡 (Precision=0.1555, Recall=0.3992)。若希望提高 Recall，可以适当降低 `learning_rate`，但会以 Precision 下降为代价；若希望提高 Precision，可以增加 `learning_rate`，但需要避免过拟合导致的整体性能下降。

综上所述，实验表明 AdaBoost 对 `learning_rate` 的选择较为敏感，过高或过低的学习率都会影响模型性能，而基分类器数量的增加在一定范围内有助于提高模型，但过多可能导致过拟合。因此，在实际应用中，应根据具体任务需求 (Precision vs. Recall) 调整超参数，以达到最佳效果。

8) 随机森林调整参数

代码:

```
import numpy as np
import pandas as pd
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score,
f1_score, roc_auc_score
from sklearn.utils import resample, shuffle

# 评估模型
def evaluate_model(model, test_x, test_y):
    pred_y = model.predict(test_x)
    accuracy = accuracy_score(test_y, pred_y)
    precision = precision_score(test_y, pred_y, average='binary')
    recall = recall_score(test_y, pred_y, average='binary')
    f1 = f1_score(test_y, pred_y, average='binary')
    auc = roc_auc_score(test_y, pred_y)
    return accuracy, precision, recall, f1, auc

if __name__ == '__main__':
    # 读取数据
    train = pd.read_excel('./Data/data.xlsx')
    test = pd.read_excel('./Data/eval.xlsx')
    train['source'] = 'train'
    test['source'] = 'test'
```

```

data = pd.concat([train, test], ignore_index=True, sort=False)

# 处理类别不平衡问题
train_up = train[train['移动房车险数量'] == 1]
train_down = train[train['移动房车险数量'] == 0]
train_up = resample(train_up, n_samples=696, random_state=0)
train_down = resample(train_down, n_samples=1095, random_state=0)
train = shuffle(pd.concat([train_up, train_down]))

data = pd.concat([train, test], ignore_index=True, sort=False)

# 选择数值特征
numeric_train = train.select_dtypes(include=[np.number])

# 计算相关系数
corr_target = numeric_train.corr()['移动房车险数量']

# 选择相关性绝对值 >= 0.01 的特征
important_feature = corr_target[np.abs(corr_target) >= 0.01].index.tolist()
train = train[important_feature]
test = test[important_feature]
train_x, train_y = train.iloc[:, :-1], train.iloc[:, -1]
test_x, test_y = test.iloc[:, :-1], test.iloc[:, -1]

# 超参数组合
n_estimators_list = [50, 100]
max_features_list = ['sqrt', 'log2', None]
max_depth_list = [10, 20, None]
min_samples_split_list = [2, 5, 10]

# 遍历不同的超参数组合
for n_estimators in n_estimators_list:
    for max_features in max_features_list:
        for max_depth in max_depth_list:
            for min_samples_split in min_samples_split_list:
                print(f"\nTraining Random Forest with n_estimators={n_estimators},
max_features={max_features}, max_depth={max_depth},
min_samples_split={min_samples_split} ...")

# 定义随机森林分类器
model = RandomForestClassifier(
    n_estimators=n_estimators,
    max_features=max_features,
    max_depth=max_depth,
    min_samples_split=min_samples_split,
    random_state=0,
    bootstrap=True # 自助采样
)

# 训练模型
model.fit(train_x, train_y)

# 评估模型
accuracy, precision, recall, f1, auc = evaluate_model(model, test_x, test_y)
print(f"\nRandom Forest Results (n_estimators={n_estimators},
max_features={max_features}, max_depth={max_depth},
min_samples_split={min_samples_split}):")
print(f'Accuracy %.4f' % accuracy)
print(f'Precision %.4f' % precision)
print(f'Recall %.4f' % recall)
print(f'F1 Score %.4f' % f1)
print(f'AUC %.4f' % auc)

```

结果对比：

n_estimators=50, max_features=sqrt

	max_depth=10	max_depth=20	max_depth=None
min_samples_split=2	Random Forest Resu s=sqrt, max_depth= Accuracy 0.8520 Precision 0.1596 Recall 0.3487 F1 Score 0.2190 AUC 0.6163	Random Forest Res s=sqrt, max_depth: Accuracy 0.8570 Precision 0.1462 Recall 0.2899 F1 Score 0.1944 AUC 0.5914	Random Forest Resu s=sqrt, max_depth= Accuracy 0.8572 Precision 0.1465 Recall 0.2899 F1 Score 0.1946 AUC 0.5915
min_samples_split=5	Random Forest Resu s=sqrt, max_depth= Accuracy 0.8528 Precision 0.1644 Recall 0.3613 F1 Score 0.2260 AUC 0.6226	Random Forest Resu s=sqrt, max_depth= Accuracy 0.8590 Precision 0.1517 Recall 0.2983 F1 Score 0.2011 AUC 0.5964	Random Forest Resu s=sqrt, max_depth= Accuracy 0.8602 Precision 0.1518 Recall 0.2941 F1 Score 0.2003 AUC 0.5951
min_samples_split=10	Random Forest Resu s=sqrt, max_depth= Accuracy 0.8550 Precision 0.1621 Recall 0.3445 F1 Score 0.2204 AUC 0.6159	Random Forest Resu s=sqrt, max_depth= Accuracy 0.8662 Precision 0.1617 Recall 0.2983 F1 Score 0.2097 AUC 0.6002	Random Forest Resu s=sqrt, max_depth= Accuracy 0.8662 Precision 0.1633 Recall 0.3025 F1 Score 0.2121 AUC 0.6022

n_estimators=50, max_features=log2

	max_depth=10	max_depth=20	max_depth=None
min_samples_split=2	Random Forest Resu s=log2, max_depth=1 Accuracy 0.8588 Precision 0.1643 Recall 0.3361 F1 Score 0.2207 AUC 0.6140	Random Forest Res s=log2, max_depth Accuracy 0.8580 Precision 0.1413 Recall 0.2731 F1 Score 0.1862 AUC 0.5841	Random Forest Resi s=log2, max_depth: Accuracy 0.8635 Precision 0.1532 Recall 0.2857 F1 Score 0.1994 AUC 0.5929
min_samples_split=5	Random Forest Resu s=log2, max_depth= Accuracy 0.8550 Precision 0.1452 Recall 0.2941 F1 Score 0.1944 AUC 0.5923	Random Forest Resu s=log2, max_depth= Accuracy 0.8675 Precision 0.1651 Recall 0.3025 F1 Score 0.2136 AUC 0.6029	Random Forest Resi s=log2, max_depth: Accuracy 0.8675 Precision 0.1636 Recall 0.2983 F1 Score 0.2113 AUC 0.6009
min_samples_split=10	Random Forest Res s=log2, max_depth Accuracy 0.8595 Precision 0.1653 Recall 0.3361 F1 Score 0.2216 AUC 0.6144	Random Forest Resi s=log2, max_depth: Accuracy 0.8658 Precision 0.1670 Recall 0.3151 F1 Score 0.2183 AUC 0.6079	Random Forest Res s=log2, max_depth: Accuracy 0.8662 Precision 0.1678 Recall 0.3151 F1 Score 0.2190 AUC 0.6081

n_estimators=50, max_features=None			
	max_depth=10	max_depth=20	max_depth=None
min_samples_split=2	Random Forest Results=None, max_depth=10 Accuracy 0.8155 Precision 0.1291 Recall 0.3655 F1 Score 0.1908 AUC 0.6048	Random Forest Results=None, max_depth=20 Accuracy 0.8250 Precision 0.1213 Recall 0.3109 F1 Score 0.1745 AUC 0.5842	Random Forest Results=None, max_depth=None Accuracy 0.8270 Precision 0.1204 Recall 0.3025 F1 Score 0.1722 AUC 0.5814
min_samples_split=5	Random Forest Results=None, max_depth=10 Accuracy 0.8193 Precision 0.1309 Recall 0.3613 F1 Score 0.1922 AUC 0.6048	Random Forest Results=None, max_depth=20 Accuracy 0.8260 Precision 0.1282 Recall 0.3319 F1 Score 0.1850 AUC 0.5946	Random Forest Results=None, max_depth=None Accuracy 0.8257 Precision 0.1256 Recall 0.3235 F1 Score 0.1810 AUC 0.5905
min_samples_split=10	Random Forest Results=None, max_depth=10 Accuracy 0.8185 Precision 0.1314 Recall 0.3655 F1 Score 0.1933 AUC 0.6064	Random Forest Results=None, max_depth=20 Accuracy 0.8297 Precision 0.1339 Recall 0.3403 F1 Score 0.1922 AUC 0.6005	Random Forest Results=None, max_depth=None Accuracy 0.8275 Precision 0.1307 Recall 0.3361 F1 Score 0.1882 AUC 0.5974
n_estimators=100, max_features=sqrt			
	max_depth=10	max_depth=20	max_depth=None
min_samples_split=2	Random Forest Results=sqrt, max_depth=10 Accuracy 0.8515 Precision 0.1564 Recall 0.3403 F1 Score 0.2143 AUC 0.6121	Random Forest Results=sqrt, max_depth=20 Accuracy 0.8570 Precision 0.1506 Recall 0.3025 F1 Score 0.2011 AUC 0.5973	Random Forest Results=sqrt, max_depth=None Accuracy 0.8550 Precision 0.1481 Recall 0.3025 F1 Score 0.1989 AUC 0.5962
min_samples_split=5	Random Forest Results=sqrt, max_depth=10 Accuracy 0.8558 Precision 0.1590 Recall 0.3319 F1 Score 0.2150 AUC 0.6104	Random Forest Results=sqrt, max_depth=20 Accuracy 0.8610 Precision 0.1558 Recall 0.3025 F1 Score 0.2057 AUC 0.5994	Random Forest Results=sqrt, max_depth=None Accuracy 0.8618 Precision 0.1584 Recall 0.3067 F1 Score 0.2089 AUC 0.6018
min_samples_split=10	Random Forest Results=sqrt, max_depth=10 Accuracy 0.8598 Precision 0.1697 Recall 0.3487 F1 Score 0.2283 AUC 0.6204	Random Forest Results=sqrt, max_depth=20 Accuracy 0.8655 Precision 0.1575 Recall 0.2899 F1 Score 0.2041 AUC 0.5959	Random Forest Results=sqrt, max_depth=None Accuracy 0.8658 Precision 0.1595 Recall 0.2941 F1 Score 0.2068 AUC 0.5980

n_estimators=100, max_features=log2			
	max_depth=10	max_depth=20	max_depth=None
min_samples_split=2	Random Forest Res es=sqrt, max_dept Accuracy 0.8658 Precision 0.1595 Recall 0.2941 F1 Score 0.2068 AUC 0.5980	Random Forest Res es=log2, max_dept Accuracy 0.8605 Precision 0.1507 Recall 0.2899 F1 Score 0.1983 AUC 0.5933	Random Forest Resu es=log2, max_dept Accuracy 0.8605 Precision 0.1522 Recall 0.2941 F1 Score 0.2006 AUC 0.5952
min_samples_split=5	Random Forest Resu es=log2, max_dept Accuracy 0.8592 Precision 0.1579 Recall 0.3151 F1 Score 0.2104 AUC 0.6044	Random Forest Res es=log2, max_dept Accuracy 0.8642 Precision 0.1510 Recall 0.2773 F1 Score 0.1956 AUC 0.5893	Random Forest Res es=log2, max_dept Accuracy 0.8652 Precision 0.1572 Recall 0.2899 F1 Score 0.2038 AUC 0.5958
min_samples_split=10	Random Forest Res es=log2, max_dept Accuracy 0.8640 Precision 0.1600 Recall 0.3025 F1 Score 0.2093 AUC 0.6010	Random Forest Resu es=log2, max_dept Accuracy 0.8675 Precision 0.1605 Recall 0.2899 F1 Score 0.2066 AUC 0.5970	Random Forest Resu es=log2, max_dept Accuracy 0.8680 Precision 0.1580 Recall 0.2815 F1 Score 0.2024 AUC 0.5933

n_estimators=100, max_features=None			
	max_depth=10	max_depth=20	max_depth=None
min_samples_split=2	Random Forest Resu es=None, max_dept Accuracy 0.8165 Precision 0.1310 Recall 0.3697 F1 Score 0.1934 AUC 0.6073	Random Forest Res es=None, max_dept Accuracy 0.8257 Precision 0.1232 Recall 0.3151 F1 Score 0.1771 AUC 0.5866	Random Forest Res es=None, max_dept Accuracy 0.8263 Precision 0.1248 Recall 0.3193 F1 Score 0.1795 AUC 0.5888
min_samples_split=5	Random Forest Resu es=None, max_dept Accuracy 0.8197 Precision 0.1335 Recall 0.3697 F1 Score 0.1962 AUC 0.6090	Random Forest Res es=None, max_dept Accuracy 0.8287 Precision 0.1342 Recall 0.3445 F1 Score 0.1932 AUC 0.6020	Random Forest Resu es=None, max_dept Accuracy 0.8267 Precision 0.1301 Recall 0.3361 F1 Score 0.1876 AUC 0.5970
min_samples_split=10	Random Forest Res es=None, max_dept Accuracy 0.8243 Precision 0.1362 Recall 0.3655 F1 Score 0.1984 AUC 0.6094	Random Forest Res es=None, max_dept Accuracy 0.8305 Precision 0.1346 Recall 0.3403 F1 Score 0.1929 AUC 0.6009	Random Forest Resu es=None, max_dept Accuracy 0.8300 Precision 0.1341 Recall 0.3403 F1 Score 0.1924 AUC 0.6007

结论：

在本实验中，我们采用随机森林（Random Forest）分类器，并针对多个超参数组合进行了实验，以评估其在不同参数配置下的性能。实验主要调整了 n_estimators（决策树数量）、max_features（最大特征数）、max_depth（最大树深）、min_samples_split（最小样本划分数）等超参数，并使用准确率（Accuracy）、精确率（Precision）、召回率（Recall）、F1 分数（F1 Score）和 AUC 作为评估指标。

从实验结果来看, `n_estimators=100` 的情况下, 模型的整体表现相较于 `n_estimators=50` 时略有提升, 说明增加决策树的数量有助于提升稳定性。而在 `max_features` 方面, 选择 `sqrt` 和 `log2` 均能取得较好的结果, 而 `None` 作为参数时, 模型的 AUC 值和 F1 分数普遍较低, 表明随机森林在特征选择时, 适当的降维 (即 `sqrt` 或 `log2`) 能够提升泛化能力。

在 `max_depth` 方面, 整体来看, `max_depth=10` 和 `max_depth=20` 的表现较为相近, 但 `max_depth=None` 时, 模型的性能略有下降。例如, 在 `n_estimators=100, max_features=sqrt` 时, `max_depth=10` 和 `max_depth=20` 的 AUC 值分别为 0.6121 和 0.5973, 而 `max_depth=None` 时下降至 0.5962, 表明不限制树深度并未带来明显收益, 反而可能引入过拟合风险。

在 `min_samples_split` 方面, 实验显示, 较大的 `min_samples_split` (如 10) 通常可以提升模型的稳定性, 并避免过拟合。例如, 在 `n_estimators=100, max_features=log2, max_depth=20` 时, `min_samples_split=2` 的 AUC 值为 0.5933, 而 `min_samples_split=10` 提升至 0.6010, 说明适当增加划分样本数可以提高模型的鲁棒性。

综合来看, 在本实验的数据集上, 表现较优的配置通常是 `n_estimators=100, max_features=sqrt` 或 `log2, max_depth=10` 或 `20, min_samples_split=10`, 该配置在多个指标上均表现较为稳定, 特别是在 AUC 和 F1 Score 上具有较好的均衡性。因此, 在实际应用中, 建议在类似的数据分布情况下, 采用该超参数组合以获得更好的分类性能。

7. 总结:

本次实验围绕移动房车险的预测展开, 旨在通过对用户属性和历史投保记录的分析, 构建有效的分类模型以实现精准的保险产品推荐。在数据预处理阶段, 通过箱图分析发现购买移动房车险的家庭在经济实力、教育水平、风险意识、婚姻状况、社保倾向和职业分布等方面存在显著特征。同时, 采用相关系数法降低数据维度, 保留与目标变量相关性较高的特征, 为后续模型训练奠定基础。

在分类模型构建中, 尝试了多种机器学习模型。结果显示, 尽管部分模型准确率相对较高, 但精确率、召回率和 F1 分数普遍较低, AUC 接近 0.5, 表明模型的泛化能力较差。深入分析发现, 数据集存在严重的类别不平衡问题, 这是导致模型性能不佳的主要原因。通过重采样方式平衡数据集后, 部分模型性能有所改善, 但仍有提升空间。

在算法调参环节, 对各个模型的关键参数进行了调整和优化。例如, ID3 决策树的最大深度、切分特征数等参数的变化会显著影响模型性能; 逻辑回归的 L1 和 L2 惩罚方式在可解释性和泛化能力上各有优劣; 多层感知机的隐藏层数、学习率和迭代次数, K 近邻分类器的 K 值, 高斯朴素贝叶斯分类器, 以及 AdaBoost 和随机森林的相关参数调整, 都对模型的预测能力产生了不同程度的影响。通过这些调参实验, 进一步了解了不同模型的特性和适用场景。

整体而言, 本次实验在模型构建和优化方面取得了一定成果, 但仍需进一步探索更有效的数据处理方法和模型改进策略, 以提高保险产品推荐的准确性和可靠性。

思考题：

1. 如何判断分类模型的优劣？

衡量分类模型的优劣需要综合多个指标，而不仅仅依赖单一数值。

- 准确率（Accuracy）反映了模型整体的分类能力，即正确分类的样本占总样本的比例。然而，在类别不均衡的数据集中，准确率可能具有欺骗性。例如，如果某类别占比极小，模型可能仅通过预测多数类获得较高的准确率，但对少数类的识别能力却很差。本次实验就存在这种情况，部分模型的准确率较高，但对“购买移动房车险”这一少数类的预测效果并不理想。
- 精确率（Precision）衡量的是模型在预测为正类的样本中，实际正确的比例。精确率较低意味着模型误报较多。在保险推荐场景下，误报可能会导致大量未购买保险的用户被错误推荐，从而增加营销成本和风险。
- 召回率（Recall）评估的是实际为正类的样本中，有多少被正确识别为正类。召回率过低，说明模型会错过许多真实的正类样本。在保险推荐场景中，这意味着会遗漏许多潜在的购买用户，影响业务收益。
- F1 分数（F1 Score）是精确率和召回率的调和平均值，综合考虑了两者的平衡性。F1 分数较高的模型，说明其在正类样本的预测上更加均衡。
- ROC 曲线与 AUC 值进一步衡量模型在不同分类阈值下的表现。ROC 曲线描述了真正率（TPR）和假阳性率（FPR）之间的关系，而 AUC 值则表示 ROC 曲线下的面积。AUC 越接近 1，模型区分正负样本的能力越强；AUC 为 0.5，则表示模型的预测效果与随机猜测无异。AUC 值可以帮助我们选择最佳分类阈值，使模型在不同应用场景下达到最佳性能。

综合来看，在类别不均衡的场景中，单独依赖准确率是不够的。需要结合精确率、召回率、F1 分数和 AUC 值，才能全面衡量模型的分类性能。

2. 在分类模型中，校验集的作用是什么？

在分类模型训练过程中，校验集起到了重要的中间评估作用。

• 模型评估

校验集独立于训练集和测试集，主要用于评估模型在未见过的数据上的表现。通过校验集，我们可以提前发现模型的泛化能力是否足够，而不是仅凭训练集上的高分就认为模型优秀。例如，如果模型在训练集上表现很好，但在校验集上的效果明显下降，这往往是过拟合的信号。

• 模型选择

在调参或选择最佳模型时，校验集提供了一个公平的评估标准。如果我们直接使用测试集进行模型选择，会导致测试数据被过度使用，最终影响测试结果的客观性。通过校验集，我们可以先筛选出表现最好的模型，再用测试集进行最终评估，以确保测试集的真实性能衡量作用不被破坏。

• 防止过拟合

在训练过程中，如果模型在训练集上的性能不断提升，但在校验集上却开始下降，这说明模型开始记住训练数据的细节，而非学习数据的规律。这是典型的过拟合现象。此时，可以使用校验集来决定何时停止训练，或者调整模型的超参数，以增强其泛化能力，确保它在实际应用中依然有效。

总体而言，校验集在模型训练过程中帮助我们选择最优的模型配置，并防止模型过度拟合训练数据。合理利用校验集，可以大幅提升分类模型在真实应用中的效果。