证券公司客户流失预测

任务一：数据分析与预处理

此次任务中使用的数据集为 train.csv,数据集格式为csv，先导入了数据并进行了观察。

df=pd.read_csv('大作业/train.csv')

导入数据与观察数据

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

```
df=pd.read_csv('大作业/train.csv')
```

```
df.head()
```

	ID	Attrition_Flag	Customer_Age	Gender	Dependent_count	Education_Level	Marital_Status	Income_Category	Card_Category	Months_on_book	...
0	716210283	1	34.0	NaN	B	High School	M	Unknown	Red	42	...
1	711615933	1	44.5	M	D	Graduate	M	$80K-120K	Red	36	...
2	785311158	1	61.0	NaN	C	Graduate	S	Less than $40K	Red	66	...
3	717267108	1	35.5	M	B	College	S	$60K-80K	Red	22	...
4	711636558	0	29.5	M	D	Graduate	M	$80K-120K	Red	42	...

5 rows × 24 columns

```
In [4]: df.isnull().sum()
```

```
Out[4]: ID                          0
Attrition_Flag              0
Customer_Age                0
Gender                    621
Dependent_count             0
Education_Level          1068
Marital_Status              0
Income_Category             0
Card_Category             114
Months_on_book              0
Total_Relationship_Count    0
Months_Inactive_12_mon      0
Contacts_Count_12_mon     865
Contacts_Count_13_mon       0
Credit_Limit                0
Total_Revolving_Bal         0
Avg_Open_To_Buy             0
Open_To_Buy                 0
Total_Amt_Chng_Q4_Q1        0
Total_Trans_Amt             0
Total_Trans_Ct              0
Total_Ct_Chng_Q4_Q1         0
Avg_Utilization_Ratio       0
Total                       0
dtype: int64
```

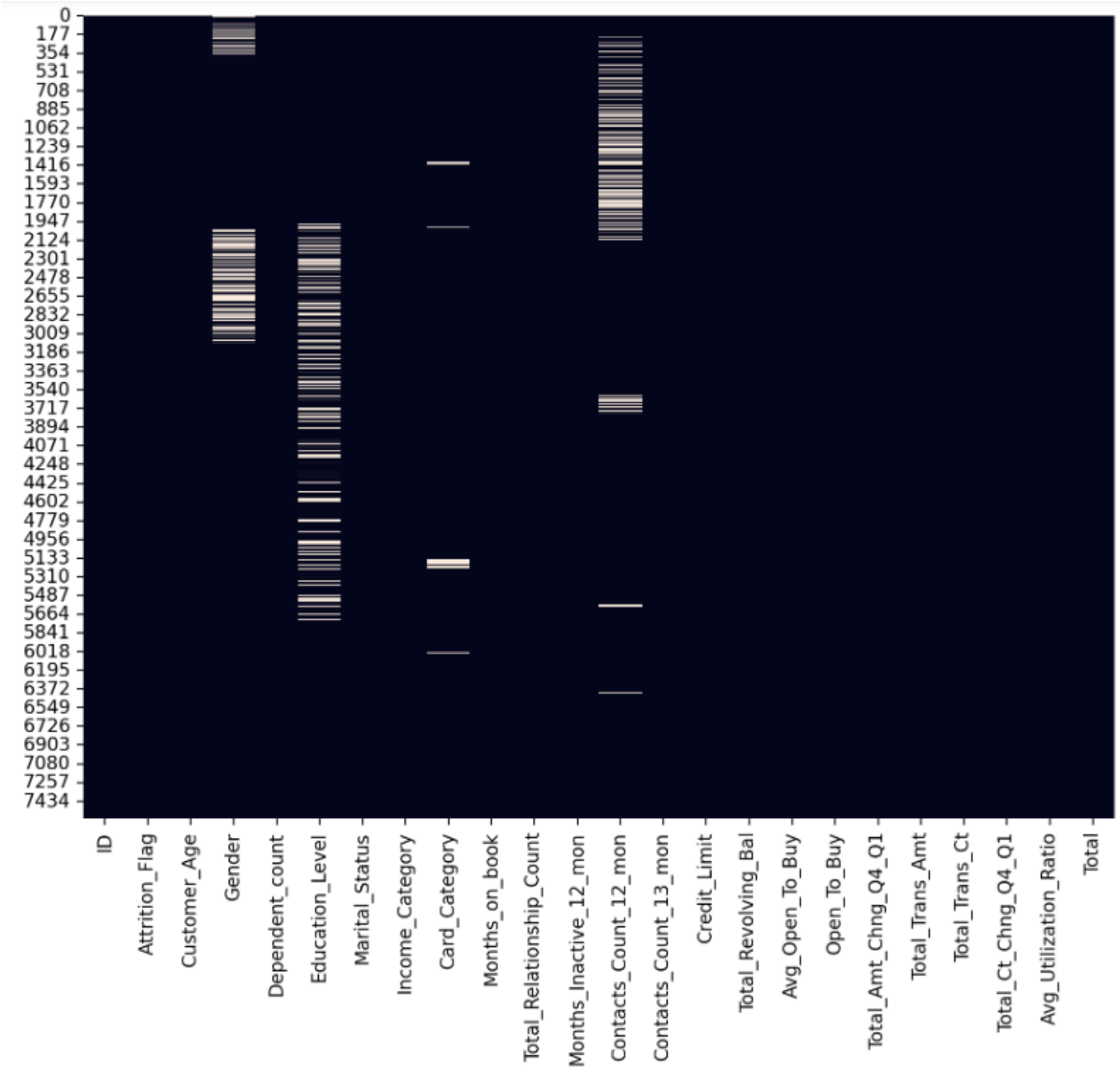
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7600 entries, 0 to 7599
Data columns (total 24 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   ID                        7600 non-null   int64
 1   Attrition_Flag            7600 non-null   int64
 2   Customer_Age              7600 non-null   float64
 3   Gender                    6979 non-null   object
 4   Dependent_count           7600 non-null   object
 5   Education_Level           6532 non-null   object
 6   Marital_Status            7600 non-null   object
 7   Income_Category           7600 non-null   object
 8   Card_Category             7486 non-null   object
 9   Months_on_book            7600 non-null   int64
 10  Total_Relationship_Count  7600 non-null   object
 11  Months_Inactive_12_mon    7600 non-null   int64
 12  Contacts_Count_12_mon     6735 non-null   float64
 13  Contacts_Count_13_mon     7600 non-null   int64
 14  Credit_Limit              7600 non-null   float64
 15  Total_Revolving_Bal       7600 non-null   int64
 16  Avg_Open_To_Buy           7600 non-null   float64
 17  Open_To_Buy               7600 non-null   float64
 18  Total_Amt_Chng_Q4_Q1      7600 non-null   float64
 19  Total_Trans_Amt           7600 non-null   int64
 20  Total_Trans_Ct            7600 non-null   int64
 21  Total_Ct_Chng_Q4_Q1       7600 non-null   float64
 22  Avg_Utilization_Ratio     7600 non-null   float64
 23  Total                     7600 non-null   int64
dtypes: float64(8), int64(9), object(7)
memory usage: 1.4+ MB
```

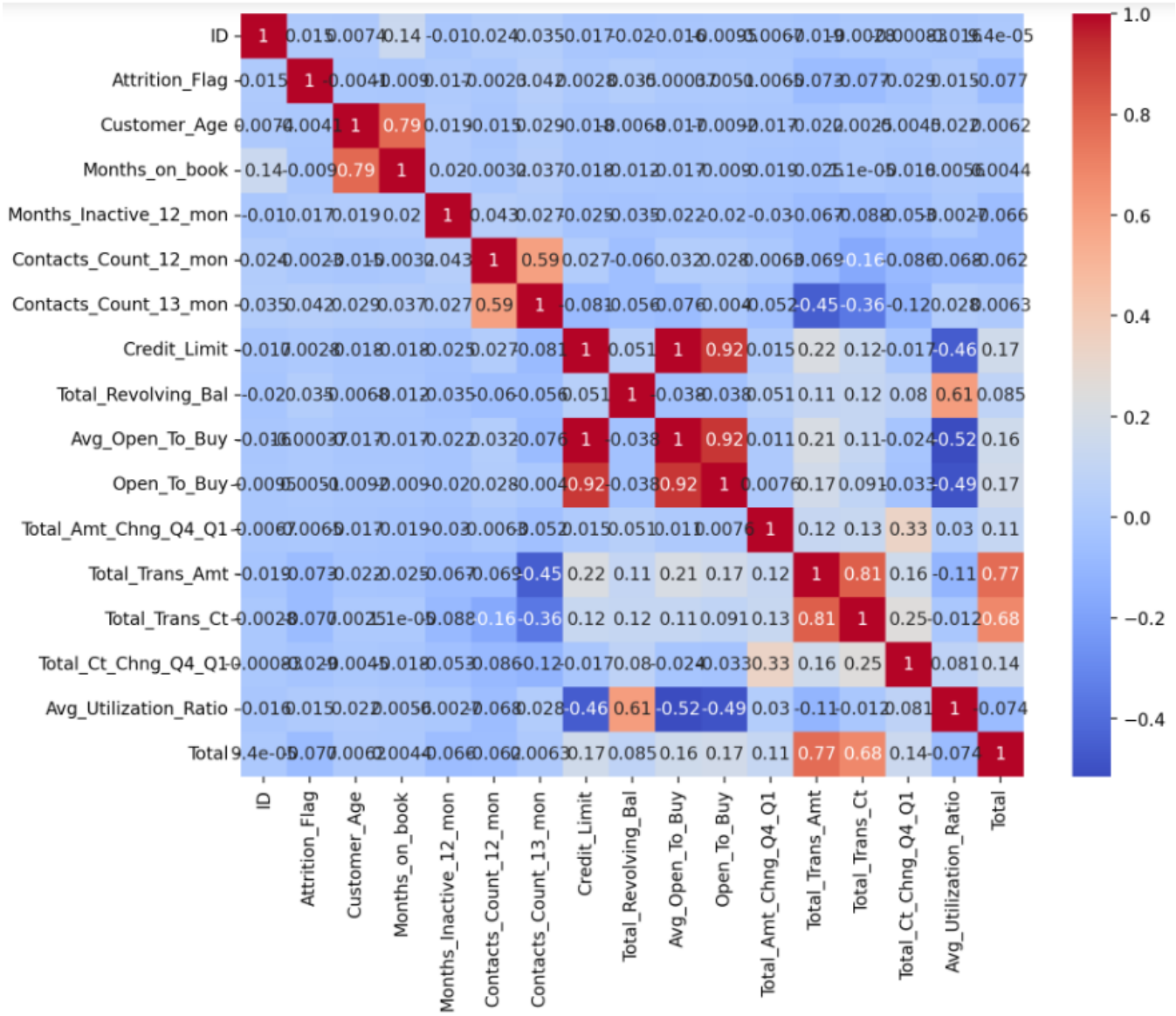
-数据解读

通过观察数据我们可以发现，7600条数据在四个标签里包含100到1000左右不等的缺失值。通过以下代码可以对缺失值的分布进行可视化。

```
import seaborn as sns
plt.figure(figsize=(10, 8), dpi=200)
sns.heatmap(df.isnull(),cbar=False)
```



```
corr=df.corr(method='pearson')
plt.figure(figsize=(10, 8), dpi=200)
sns.heatmap(data=corr, annot=True, cmap='coolwarm')
```



-缺失值填充处理

缺失值填充处理

In [10]:
```
# 使用最频繁的值填充'Gender'列的缺失值
df['Gender'].fillna(df['Gender'].mode()[0], inplace=True)
```

In [11]:
```
# 使用最频繁的值填充'Education_Level'列的缺失值
df['Education_Level'].fillna(df['Education_Level'].mode()[0], inplace=True)
```

In [12]:
```
# 使用最频繁的值填充'Card_Category'列的缺失值
df['Card_Category'].fillna(df['Card_Category'].mode()[0], inplace=True)
```

In [13]:
```
# 使用中位数填充'Contacts_Count_12_mon'列的缺失值
df['Contacts_Count_12_mon'].fillna(df['Contacts_Count_12_mon'].median(), inplace=True)
```

用最频繁出现的值，中位数填充了数据集中的缺失值。

中位数填充是指使用数据集中的中位数来填补缺失值。它可以在数据集中缺失值不是很多的情况下使用，因为使用中位数的方法可以将数据的分布对称化，同时也不会对数据的均值造成太大的影响。

最频繁值填充是指使用数据集中最常见的值来填补缺失值。这种方法通常适用于数据集中缺失值很多的情况，因为使用最频繁值来填补缺失值可以使数据更加平滑，同时也不会对数据的均值造成太大的影响。

在缺失值在数据中的比例，以及标签类型，数据构成等方面来看，这两种处理方法是可行的。

填充过后观察数据集的结果如下：

```
In [14]: df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7600 entries, 0 to 7599
Data columns (total 24 columns):
 #   Column                    Non-Null Count  Dtype
---  ------                    --------------  -----
 0   ID                        7600 non-null   int64
 1   Attrition_Flag            7600 non-null   int64
 2   Customer_Age              7600 non-null   float64
 3   Gender                    7600 non-null   object
 4   Dependent_count           7600 non-null   object
 5   Education_Level           7600 non-null   object
 6   Marital_Status            7600 non-null   object
 7   Income_Category           7600 non-null   object
 8   Card_Category             7600 non-null   object
 9   Months_on_book            7600 non-null   int64
 10  Total_Relationship_Count  7600 non-null   object
 11  Months_Inactive_12_mon    7600 non-null   int64
 12  Contacts_Count_12_mon     7600 non-null   float64
 13  Contacts_Count_13_mon     7600 non-null   int64
 14  Credit_Limit              7600 non-null   float64
 15  Total_Revolving_Bal       7600 non-null   int64
 16  Avg_Open_To_Buy           7600 non-null   float64
 17  Open_To_Buy               7600 non-null   float64
 18  Total_Amt_Chng_Q4_Q1      7600 non-null   float64
 19  Total_Trans_Amt           7600 non-null   int64
 20  Total_Trans_Ct            7600 non-null   int64
 21  Total_Ct_Chng_Q4_Q1       7600 non-null   float64
 22  Avg_Utilization_Ratio     7600 non-null   float64
 23  Total                     7600 non-null   int64
dtypes: float64(8), int64(9), object(7)
memory usage: 1.4+ MB
```

```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 86222 entries, 0 to 86221
Data columns (total 9 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   stuid        86222 non-null  int64
 1   id           86222 non-null  object
 2   name         86222 non-null  object
 3   gender       85722 non-null  object
 4   exam         85747 non-null  float64
 5   assignments  85693 non-null  float64
 6   labs         85579 non-null  float64
 7   final        86222 non-null  int64
 8   passed       86222 non-null  object
dtypes: float64(3), int64(2), object(4)
memory usage: 5.9+ MB
```

```
data.isnull().sum()
```

```
stuid            0
id               0
name             0
gender         500
exam           475
assignments    529
labs           643
final            0
passed           0
dtype: int64
```

```
data = data.dropna()
```

-特征选择以及其他处理

首先对数据集进行了标签编码处理：

```
object_columns = df.select_dtypes(include=['object']).columns

non_numerical_columns = object_columns.tolist()
```

```
#对非数值数据集进行标签编码
from sklearn.preprocessing import LabelEncoder

encoder = LabelEncoder()
for col in non_numerical_columns:
    encoder.fit(df[col])
    df[col] = encoder.transform(df[col])
```

接下来的部分我用sklearn中的SelectKBest方法进行了最有特征选择，选取了对目标变量影响最大的15个元素，取出了其余的8个多余特征，代码如下：

```
import pandas as pd
from sklearn.feature_selection import SelectKBest, f_classif

X = df.drop("Attrition_Flag", axis=1) # 特征矩阵
y = df["Attrition_Flag"] # 目标变量

# 创建SelectKBest类
skb = SelectKBest(score_func=f_classif, k=15)

# 训练模型
skb.fit(X, y)

# 打印被选中的特征
selected_features = X.columns[skb.get_support()]
print(selected_features)
```

```
Index(['ID', 'Gender', 'Dependent_count', 'Education_Level', 'Marital_Status',
       'Income_Category', 'Total_Relationship_Count', 'Months_Inactive_12_mon',
       'Contacts_Count_13_mon', 'Total_Revolving_Bal', 'Total_Trans_Amt',
       'Total_Trans_Ct', 'Total_Ct_Chng_Q4_Q1', 'Avg_Utilization_Ratio',
       'Total'],
      dtype='object')
```

```
X_columns = set(X.columns)

selected_features = set(selected_features)

# 这样就可以得到未被选中的特征
dropped_features = X_columns - selected_features
dropped_features = list(dropped_features)

print(dropped_features)
```

```
['Credit_Limit', 'Total_Amt_Chng_Q4_Q1', 'Contacts_Count_12_mon', 'Card_Category', 'Months_on_book', 'Open_To_Buy', 'Customer_Age', 'Avg
_Open_To_Buy']
```

多余的特征分别是：'Credit_Limit', 'Total_Amt_Chng_Q4_Q1', 'Contacts_Count_12_mon', 'Card_Category', 'Months_on_book', 'Open_To_Buy', 'Customer_Age', 'Avg_Open_To_Buy'

我们在特征选择的时候可以使用过滤器法（使用单变量特征选择方法选择最优的特征）或包裹器法（使用特定的模型来选择最优的特征）进行特征选择。

这次作业中我使用 sklearn 库的 SelectKBest 类选择了最优的 15 个特征，其中使用 f_classif 函数计算特征的相关性。

特征选择的合理性在于，选择的特征对模型的表现有着较大的影响力，并且不包含无关或冗余的特征。这有助于减少模型的复杂度，提高模型的泛化能力，并加快训练时间。

其他处理

```
#特征选择部分我们已经做了数据标签编码的处理

#特征提取（删掉未被选中的特征列）
X_selected = X.drop(dropped_features, axis=1)

# 查看新的特征矩阵的形状
print(X_selected.shape)
```

```
(7600, 15)
```

任务二：错误标签检测

这里我使用逻辑回归构建了错误标签检测的模型，并按照要求返回了Excel文件，代码如下：

```
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression

# 读取已知的错误标签样本 ID
noisy_ids = pd.read_excel("大作业/noisy ID1.xlsx")

# 获取所有的样本 ID
ids = df["ID"]

# 将数据分成训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(df.drop("ID", axis=1),
df["Attrition_Flag"], test_size=0.2, random_state=42)

# 训练分类器
clf = LogisticRegression()
clf.fit(X_train, y_train)

# 在测试集上评估准确率
accuracy = clf.score(X_test, y_test)
print("Accuracy:", accuracy)

# 找出另外一半的错误标签对应的样本 ID
predictions = clf.predict(X_test)
wrong_ids = ids[X_test.index[predictions != y_test]]

# 将所有的错误标签样本 ID 写入 Excel 文件
result = pd.concat([noisy_ids, wrong_ids], ignore_index=True)
result = result.drop_duplicates(keep='last')  # 删除重复的 ID


# 确保结果包含 760 个样本 ID
while len(result) < 760:
    missing = 760 - len(result)
    # 找出尚未检测到的错误标签样本 ID
```

```
    remaining_ids = ids[~ids.isin(result["ID"])]
    # 手动检查剩余的样本
    for i, row in remaining_ids.iteritems():
        # 选择一个样本
        sample = df[df["ID"] == row]
        # 预测标签
        prediction = clf.predict(sample.drop("ID", axis=1))
        # 如果预测结果与实际标签不一致，则认为是错误标签
        if prediction != sample["Attrition_Flag"].values[0]:
            result = pd.concat([result, sample[["ID"]]])# 将样本 ID 加入结果列表
            if len(result) == 760:
                break

result.to_excel("result.xlsx", index=False)
```

准确率的输出如下：

```
Accuracy: 0.8171052631578948
```

代码使用的是逻辑回归（Logistic Regression）作为错误标签检测算法。

逻辑回归是一种常用的分类算法，训练分类器的时间复杂度为 O(n*p^2)，其中 n 是训练数据的大小，p 是特征的数量。

然后，在测试集上评估准确率的时间复杂度为 O(m*p)，其中 m 是测试数据的大小。

除了计算复杂度，还可以考虑逻辑回归的其他优点，如：

1，算法实现简单，适用于小规模数据。

2，特征的组合可以是连续的或离散的，对于特征的缺失也能很好地适应。

3，在训练时，逻辑回归可以较快地收敛。

对于此题，逻辑回归是一个合理的选择，因为它在训练和预测时间都很短，而且在二元分类问题中表现很好。 同时，逻辑回归也很容易解释，因为它的决策边界是一条直线或者一条平面，因此可以方便地掌握模型的决策过程。

任务三：分类模型构建

首先，导入了测试数据集test1.xlsx,然后跟训练数据合并在一起进行标签编码，之后再重新拆分。

然后在分类模型的构建方面，我用了决策树，逻辑回归，knn分类算法进行了二元分类回归模型的构建。

每个回归算法模型方面，使用网格搜索方法进行了调参，输出了最佳参数组合和在训练集上，测试集上的准确率Accuracy，具体代码如下：

```
#准备测试集，导入test1.xlsx并进行预处理
tdf=pd.read_excel("大作业/test1.xlsx")
```

```
# 合并两个数据集
combined_data = pd.concat([filtered_df, tdf], ignore_index=True)

# 标签编码
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
for col in non_numerical_columns:
    le.fit(combined_data[col])
    combined_data[col] = le.transform(combined_data[col])

# 将合并后的数据集拆分成训练集和测试集
train = combined_data[:len(filtered_df)]
test = combined_data[len(filtered_df):]
```

```
#训练集和测试集的拆分，特征矩阵和目标变量的拆分
X_train=train.drop(["ID","Attrition_Flag"], axis=1)
y_train=train["Attrition_Flag"]
X_test=test.drop(["ID","Attrition_Flag"], axis=1)
y_test=test["Attrition_Flag"]

X_train = X_train.reset_index(drop=True)
y_train = y_train.reset_index(drop=True)
X_test = X_test.reset_index(drop=True)
y_test = y_test.reset_index(drop=True)
```

```
#决策树

from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

param_grid = {"max_depth": [5, 10, 20, 30],
              "min_samples_split": [2, 5, 10],
              "min_samples_leaf": [1, 2, 4],
              "max_features": [0.5, 0.8, 1.0]}

model1 = DecisionTreeClassifier()
grid_search = GridSearchCV(model1, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)

print("最优参数：", grid_search.best_params_)

best_model1 = grid_search.best_estimator_

train_score = best_model1.score(X_train, y_train)
print("训练集上的分数：", train_score)
accuracy = best_model.score(X_test, y_test)
print("在test1中的准确率:", accuracy)
```

```
最优参数： {'max_depth': 5, 'max_features': 0.5, 'min_samples_leaf': 4, 'min_samples_split': 5}
训练集上的分数： 0.8978827596139997
在test1中的准确率: 0.5
```

```
#逻辑回归
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import GridSearchCV

param_grid = {"C": [0.01, 0.1, 1, 10, 100],
              "penalty": ["l1", "l2"]}

model = LogisticRegression()
grid_search = GridSearchCV(model, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)

print("最优参数：", grid_search.best_params_)
best_model = grid_search.best_estimator_

train_score = best_model.score(X_train, y_train)
print("训练集上的分数：", train_score)
accuracy = best_model.score(X_test, y_test)
print('在测试集上的准确率：', accuracy)
```

```
最优参数： {'C': 0.01, 'penalty': 'l2'}
训练集上的分数： 0.893849920783523
在测试集上的准确率： 0.5
```

```
# knn分类回归
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import GridSearchCV

param_grid = {"n_neighbors": [3, 5, 7, 9],
              "weights": ["uniform", "distance"]}

model = KNeighborsClassifier()
grid_search = GridSearchCV(model, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)

print("最优参数：", grid_search.best_params_)

best_model = grid_search.best_estimator_

train_score = best_model.score(X_train, y_train)
print("训练集上的分数：", train_score)
accuracy = best_model.score(X_test, y_test)
print('在测试集上的准确率：', accuracy)
```

```
最优参数： {'n_neighbors': 9, 'weights': 'uniform'}
训练集上的分数： 0.8967305199481492
在测试集上的准确率： 0.5
```

-分类算法的选取、和调参的策略介绍

1，决策树是一种常用的分类算法，它的思路是通过对数据的不断划分来逼近数据的真实分布。

在选择决策树模型时，需要考虑训练数据的特征类型，以及是否需要对模型进行调参。对于连续值的特征，决策树可以直接使用；对于离散值的特征，决策树也可以使用，但需要对离散值进行编码。这次作业中，先是将数据里的非线性数据进行了标签编码的处理，将数据处理成适合构建模型的状态，然后再进行了分类。

超参数是指在模型训练过程中需要手动调节的参数。对于决策树，常见的超参数有最大深度、最小叶子节点样本数量、最小分裂样本数量、特征最大选择数量等。超参数调参可以使用网格搜索或者随机搜索的方式进行，在此代码中使用的是网格搜索。

网格搜索是指对每一个超参数列出可能的取值，然后遍历所有可能的组合来确定最优参数。上述代码中使用了交叉验证（cv=5）来确定最优参数。在代码中，最优参数被找到后，使用最优参数来构建模型，并在训练集和测试集上分别计算准确率。

2，逻辑回归是一种常用的分类算法，它使用线性模型来对数据进行建模，并使用sigmoid函数将预测值转化为概率。对于逻辑回归，常见的超参数有正则化系数C和正则化类型（L1或L2）。超参数调参使用了网格搜索的方式进行，最优参数的确定思路跟上一个模型相同。

3，KNN（K-Nearest Neighbors）是一种常用的分类算法，它的思路是通过计算一个数据点与其他数据点的距离来决定其类别。

在选择KNN模型时，需要考虑训练数据的特征类型，以及是否需要对模型进行调参。KNN的优点在于可以处理连续值的特征，因此在选择KNN时，如果训练数据的特征都是连续值，则KNN是一个合理的选择。

对于KNN，常见的超参数有邻居数量K和权重类型（uniform或distance）。超参数调参使用了网格搜索的方式进行，最优参数的确定思路跟上一个模型相同。

-返回excel文件，代码如下：

```
#在test2数据集里预测结果并返回excel文件

tdata=pd.read_excel("大作业/test2.xlsx")
predata=tdata.copy()
```

```
# 标签编码
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
for col in non_numerical_columns:
    le.fit(predata[col])
    predata[col] = le.transform(predata[col])
```

```
X=predata.drop(["ID","Attrition_Flag"], axis=1)
```

```
#使用决策树分类回归预测y值
y=best_model1.predict(X)
```

```
tdata["Attrition_Flag"]=y
```

```
tdata
```

```
tdata.to_excel("test2_result.xlsx", index=False)
```