

通信客户流失预警模型数据挖掘报告

介应奇

November 16, 2023

Contents

1 业务背景	1
2 数据理解	1
3 数据准备	2
3.1 数据导入	2
3.2 数据清理	2
3.3 数据选择与变换	3
3.3.1 数据选择	3
3.3.2 数据转换	4
3.3.3 数据归一化	5
3.4 数据平衡性改善	6
3.5 数据压缩	7
3.5.1 PCA 主成分分析法对数据实现不同程度的无监督压缩	7
3.5.2 使用特征检定的方式进行特征筛选	8

4 探索性分析	8
5 模型构建	14
5.1 建立模型评估的框架	14
5.2 模型定义和声明	15
6 模型评估	15
6.1 对比不同数据压缩和变化在同一个模型上的效果	15
6.2 使用上采样的数据将各种模型都跑一遍 10 折交叉验证对比效果	19
6.3 手动实现集成的过程	21
6.4 集成不同类别的分类器	23
6.5 决策树调参	24
6.6 尝试自己实现一个 MLP 的模型	24
6.7 模型评估总结	33
7 模型的部署与分析	33
8 实验小结	33
8.1 工程方面	33
8.2 算法方面	34

1 业务背景

使用分类模型构建客户流失预测模型，通过客户为流失客户的概率预测该客户是否为流失客户，并根据客户为流失客户的概率生成流失概率排序名单。

2 数据理解

在本案例中使用的数据来自于 IBM Sample Data Sets，是某电信公司一段时间内的客户消费数据。共包含 7043 笔客户资料，每笔客户资料包含 21 个字段，其中 1 个客户 ID 字段，19 个属性特征字段及 1 个标签字段（Yes 代表流失，No 代表未流失）。属性特征字段主要包含以下三个维度指标：客户画像指标（如性别、是否老年等）、消费产品指标（如是否开通互联网服务、是否开通电话服务等）、消费信息指标（如付款方式、月费用等）。字段的具体说明如表 1 所示：

3 数据准备

这里的数据准备包含了一共 5 个部分，数据导入，数据清理，数据选择与变换，数据平衡性改善，数据压缩。这其中数据导入、数据清理、数据转换是公用模块，在后续的实验中的数据都是经过了这些共同的步骤，至于数据的平衡性改善和数据压缩模块产生的数据并不会替换原有的数据，而是用新的变量记录下来了不同情况处理的值，方便后续的对比分析，下面呈现代码处理的过程以及说明。

3.1 数据导入

```
1 import numpy as np
2 import pandas as pd
3
4 df = pd.read_csv('./Telco-Customer-Churn.csv')
5 df.head()
```

3.2 数据清理

首先进行初步的数据清洗工作，包含错误值和异常值处理，并识别字段类型。其中清洗工作主要包含：

- 对 MultipleLines（是否开通多线服务）、OnlineSecurity（是否开通网络安全服务）、OnlineBackup（是否开通在线备份业务）、DeviceProtection（是否开通了设备保护业务）、TechSupport（是否开通了技术支持服务）、StreamingTV（是否开通网络电视）、StreamingMovies（是否开通网络电影）等属性特征进行错误值处理。如不满足前置条件，则这些特征直接取值为 No（若未开通互联网服务，自然不会开通网络电视等服务）。
- 对 TotalCharges（总费用）特征进行异常值处理。将空白取值替换空值 nan，由于含空值的行较少，将其一并删除。删除后该特征取值只有数值，因此将此特征类型设为浮点型。
- 判断字段的类别

数据清洗代码如下所示：

```
1 repl_columns = [ 'OnlineSecurity', 'OnlineBackup', 'DeviceProtection', 'TechSupport',
  'StreamingTV', 'StreamingMovies']
2 for i in repl_columns:
3     df[i] = df[i].replace({ 'No internet service': 'No'})
4 df['MultipleLines'] = df['MultipleLines'].replace({ 'No phone service': 'No'})
5
6 # 替换值SeniorCitizen
7 df["SeniorCitizen"] = df["SeniorCitizen"].replace({1: "Yes", 0: "No"})
8
9 # 替换值TotalCharges
10 df[ 'TotalCharges'] = df[ 'TotalCharges'].replace( ' ', np.nan)
11 # TotalCharges空值：数据量小，直接删除
12 df = df.dropna(subset=[ 'TotalCharges'])
13 df.reset_index(drop= True, inplace= True) # 重置索引
```

3.3 数据选择与变换

3.3.1 数据选择

区分离散和连续变量

```
1 df[ 'TotalCharges' ] = df[ 'TotalCharges' ].astype( 'float' )
2 Id_col = [ 'customerID' ]
3 target_col = [ 'Churn' ]
4 cat_cols = df.nunique()[df.nunique() < 10].index.tolist()
5 num_cols = [ i for i in df.columns if i not in cat_cols + Id_col ]
```

3.3.2 数据转换

```
● ● ● ● ●  
1 from sklearn.preprocessing import LabelEncoder  
2 df_model = df  
3 Id_col = ['customerID']  
4 target_col = ['Churn']  
5 # 分类型  
6 cat_cols = df_model.nunique()[df_model.nunique() < 10].index.tolist()  
7 # 二分类属性  
8 binary_cols = df_model.nunique()[df_model.nunique() == 2].index.tolist()  
9 # 多分类属性  
10 multi_cols = [i for i in cat_cols if i not in binary_cols]  
11 # 数值型  
12 num_cols = [i for i in df_model.columns if i not in cat_cols + Id_col]  
13 # 二分类-标签编码  
14 le = LabelEncoder()  
15 for i in binary_cols:  
16     df_model[i] = le.fit_transform(df_model[i])  
17 # 多分类-哑变量转换  
18 df_model = pd.get_dummies(data=df_model, columns=multi_cols)  
19 df_model.head()
```

将转换后的数据定义为自变量和标签值

```
● ● ● ● ●  
1 X = df_model.copy().drop(['customerID', 'Churn'], axis=1)  
2 print(X.shape)  
3 y = df_model[target_col]
```

这个数据是没有经过特征筛选的，因为不是所有的模型都需要经过特征筛选才能使用，有一些特征选择是在模型中自己隐形选择的，因此我们在这里要保留一份最完整特征版的数据

3.3.3 数据归一化

保证模型能够快速收敛，并且消除数据的属性之间阈值不一致的差异性

满足 python 建模需要，需要对数据做以下处理。

- 对于分类变量，编码为 0 和 1;
- 对于多分类变量，进行哑变量转换；对于数值型变量，部分模型如 KNN、神经网络、Logistic 需要进行标准化处理。

```
● ● ●  
1 from sklearn.preprocessing import StandardScaler  
2  
3 st= StandardScaler()  
4  
5 X_data_norm = pd.DataFrame(st.fit_transform(X[num_cols]), columns=num_cols)  
6 X_data_norm = pd.concat([X.drop(num_cols, axis= 1), X_data_norm], axis= 1)  
7 X = X_data_norm
```

3.4 数据平衡性改善

因为我们发现流失客户和未流失客户的样本数量是不平衡的，因此使用上采样的方式增加流失客户的数据量从而使数据平衡性得到改善

```
● ○ ●

1 from imblearn.over_sampling import RandomOverSampler
2
3 ros = RandomOverSampler(random_state=0)
4 X_resampled, y_resampled = ros.fit_resample(X, y)
5 print(X.shape)
6 print(X_resampled.shape)
7 print(y['Churn'].value_counts())
8 print(y_resampled['Churn'].value_counts())
```

观察上述记过我们可以看到经过上采样之后，我们的 y 对应的两种标签值的数据数量一致了，具体观察对应的 y 值分布变化可以看出来是将流失客户数据进行了补齐，补齐到了和未流失客户数据两相同的程度

3.5 数据压缩

3.5.1 PCA 主成分分析法对数据实现不同程度的无监督压缩

```
● ● ●  
1 from sklearn.decomposition import PCA  
2  
3 pca5 = PCA(n_components=5,copy=True,whiten=False,svd_solver="auto",  
4 tol=0.0,iterated_power="auto",random_state=None)  
5  
6 pca10 = PCA(n_components=10,copy=True,whiten=False,svd_solver="auto",  
7 tol=0.0,iterated_power="auto",random_state=None)  
8  
9 pca20 = PCA(n_components=20,copy=True,whiten=False,svd_solver="auto",  
10 tol=0.0,iterated_power="auto",random_state=None)  
11  
12 pca_Compress_to_5X = pca5.fit_transform(X)  
13 pca_Compress_to_10X = pca10.fit_transform(X)  
14 pca_Compress_to_20X = pca20.fit_transform(X)
```

通过 pca 方法我获取到了保留了维度值分别为 5/10/20 的自变量

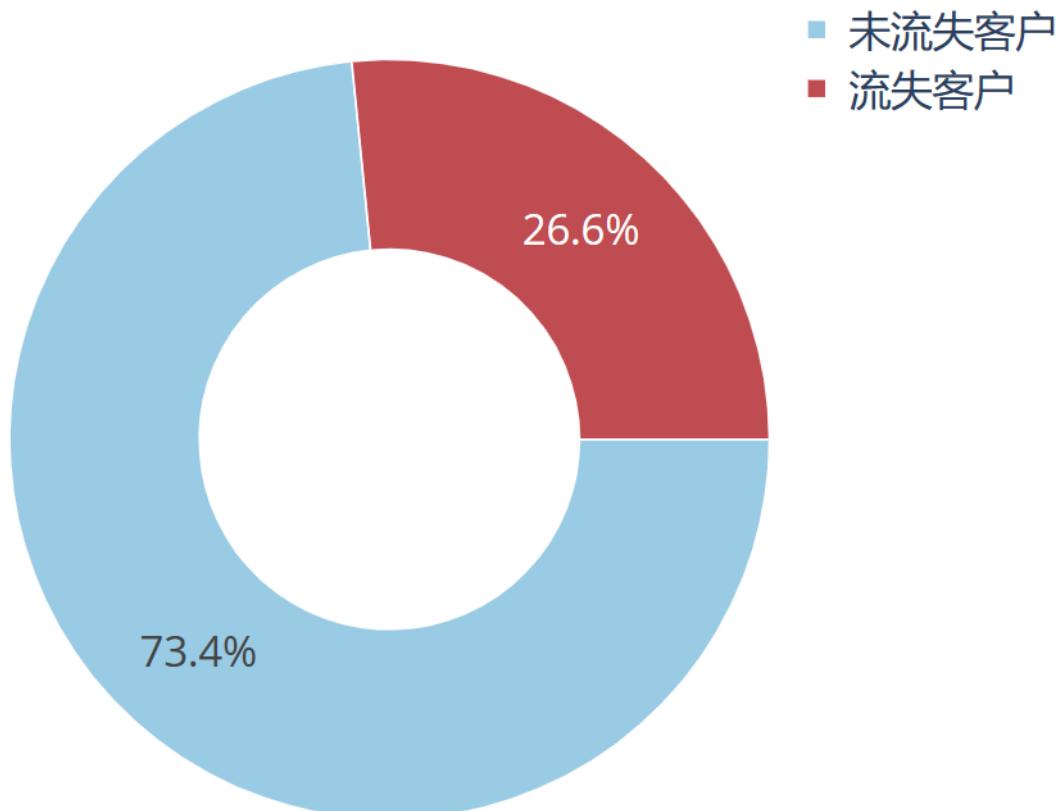
3.5.2 使用特征检定的方式进行特征筛选

```
1 from sklearn.feature_selection import SelectKBest, f_classif
2
3 fs5 = SelectKBest(score_func=f_classif, k=5)
4 fs10 = SelectKBest(score_func=f_classif, k=10)
5 fs20 = SelectKBest(score_func=f_classif, k=20)
6
7 X_train_fs5 = fs5.fit_transform(X, y)
8 X_train_fs10 = fs10.fit_transform(X, y)
9 X_train_fs20 = fs20.fit_transform(X, y)
10
11 def SelectName(feature_data, model):
12     scores = model.scores_
13     indices = np.argsort(scores)[::-1]
14     return list(feature_data.columns.values[indices[0:model.k]])
15
16 # 输出选择变量名称
17 fea_name = [i for i in X.columns if i in SelectName(X, fs5)]
18 st_Compress_to_5X = pd.DataFrame(X_train_fs5, columns = fea_name)
19 fea_name = [i for i in X.columns if i in SelectName(X, fs10)]
20 st_Compress_to_10X = pd.DataFrame(X_train_fs10, columns = fea_name)
21 fea_name = [i for i in X.columns if i in SelectName(X, fs20)]
22 st_Compress_to_20X = pd.DataFrame(X_train_fs20, columns = fea_name)
```

通过特征检定的方法我获取到了保留了维度值分别为 5/10/20 的自变量

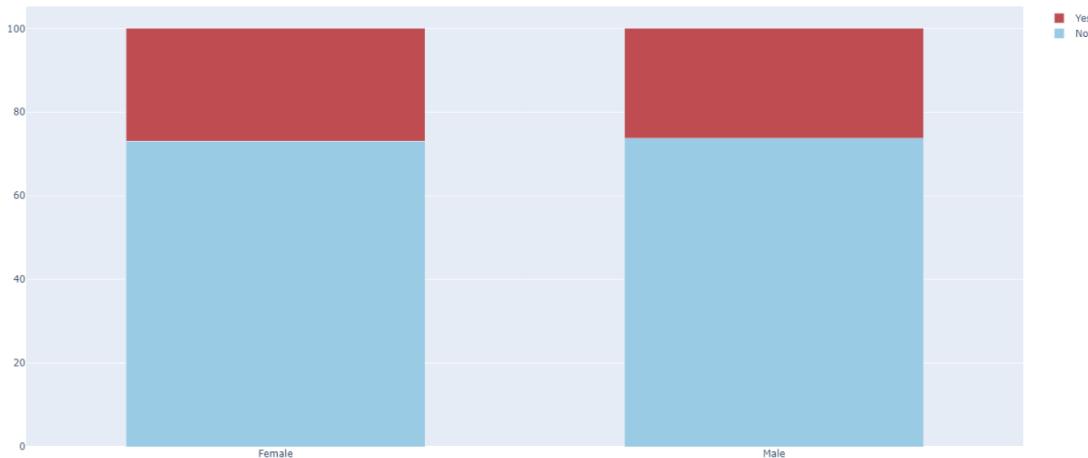
4 探索性分析

对指标进行归纳梳理，分用户画像指标，消费产品指标，消费信息指标。探索影响用户流失的关键因素。首先查看流失用户与非流失用户的整体分布情况。



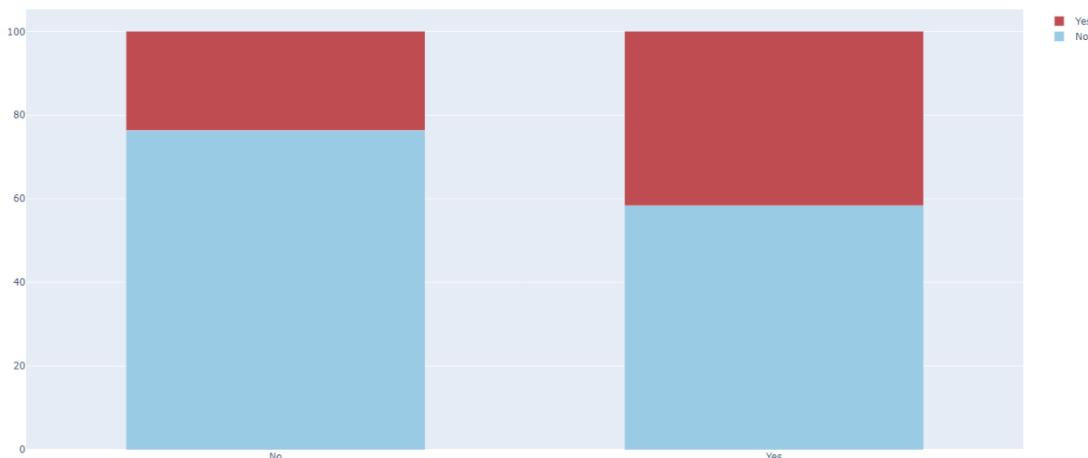
可以看出，经过初步清洗之后的数据集大小为 7032 条记录，其中流失客户为 1869 条，占比 26.6 再分别探索对各特征的影响。将每一个变量不同取值条件下用户流失与否用柱形图表示出来，可以直观地看出变量取值是否影响用户流失。

性别与是否流失的关系



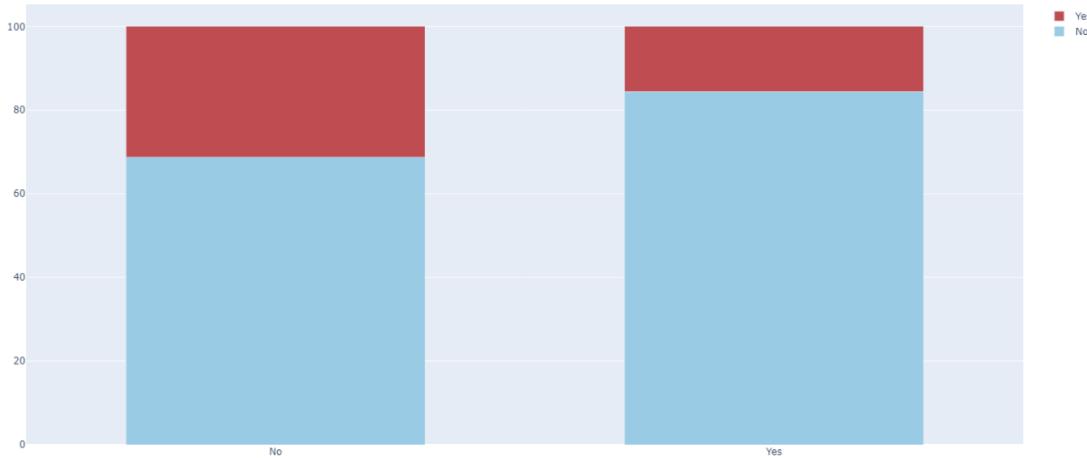
可以看出，男性和女性在客户流失比例上没有显著差异。

老年用户与是否流失的关系



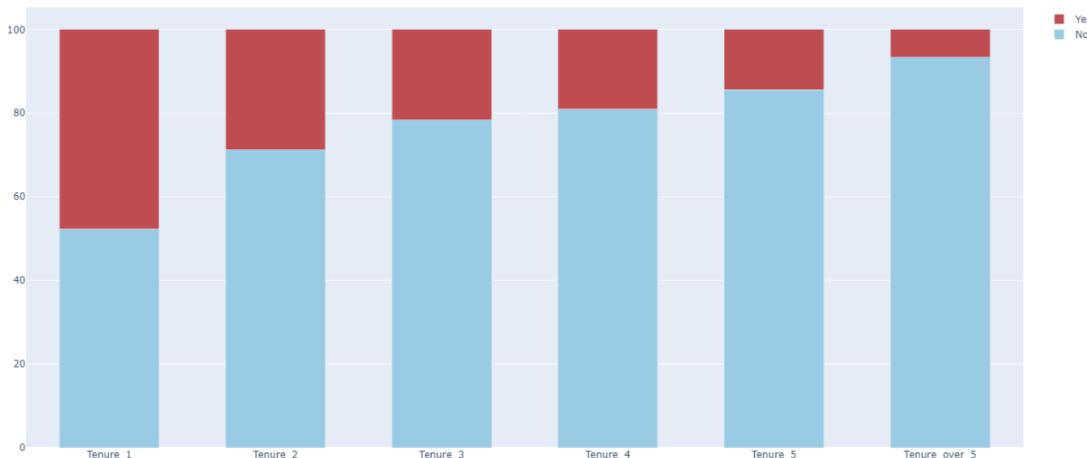
可以看出，老年用户流失比例更高，为 41.68%，比非老年用户高近两倍，此部分原因有待进一步探讨。

是否经济独立与是否流失的关系



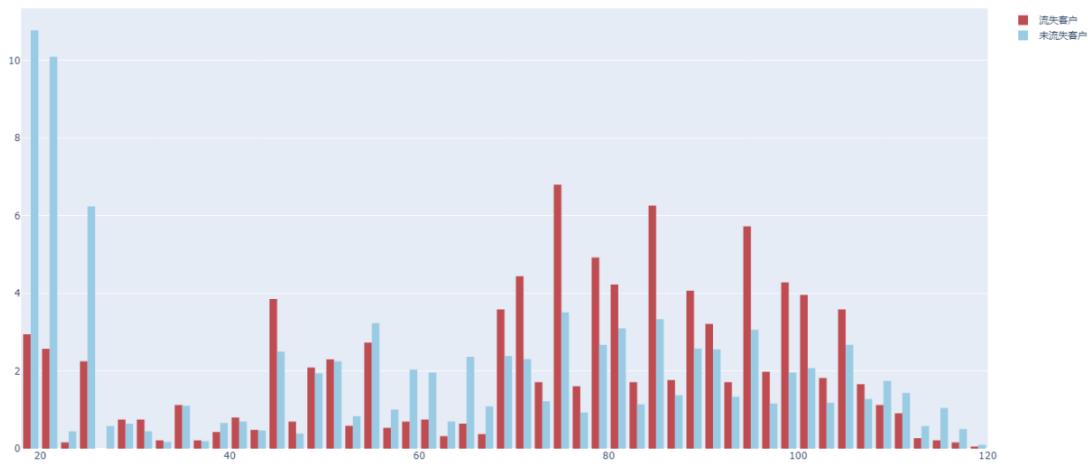
从经济独立情况来看，经济未独立的用户流失率要远远高于经济独立的用户。

在网时长与是否流失的关系



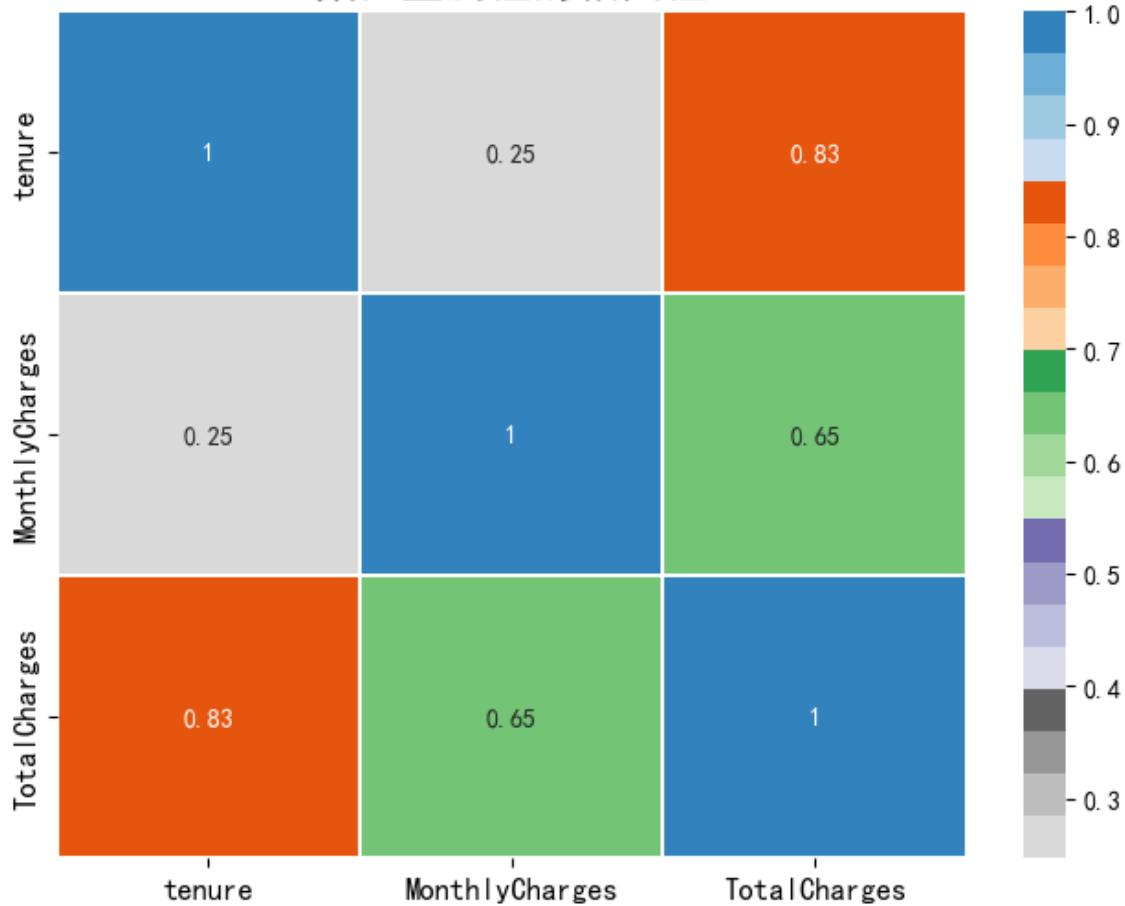
可以看出，用户的在网时长越长，表示用户的忠诚度越高，其流失的概率越低；新用户在 1 年内的流失率显著高于整体流失率，为 47.68%。

月费用与是否流失关系



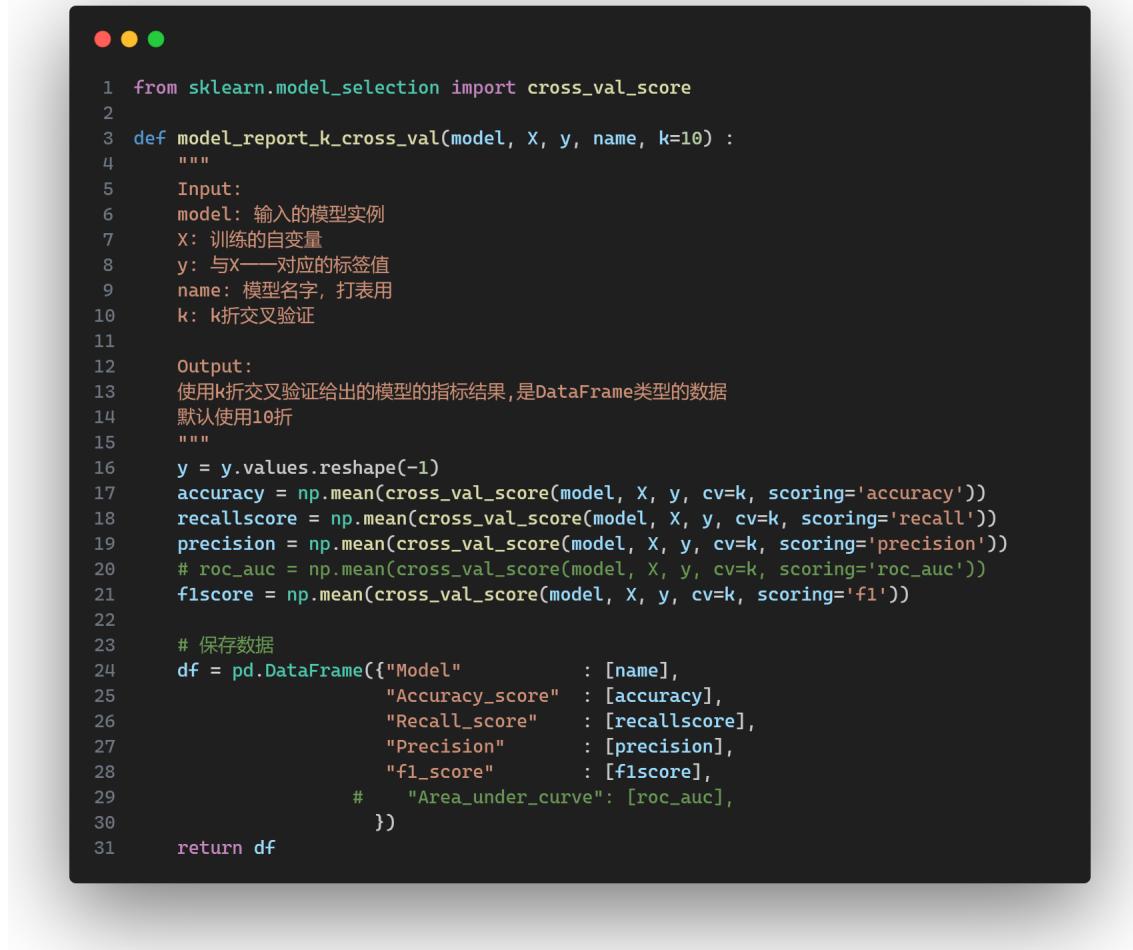
整体来看，随着月费用的增加，流失用户的比例呈现高高低低的变化，月消费 80-100 元的用户相对较高。

数值型属性的相关性



5 模型构建

5.1 建立模型评估的框架



```
1 from sklearn.model_selection import cross_val_score
2
3 def model_report_k_cross_val(model, X, y, name, k=10) :
4     """
5         Input:
6             model: 输入的模型实例
7             X: 训练的自变量
8             y: 与X一一对应的标签值
9             name: 模型名字, 打表用
10            k: k折交叉验证
11
12         Output:
13             使用k折交叉验证给出的模型的指标结果, 是DataFrame类型的数据
14             默认使用10折
15             """
16
17     y = y.values.reshape(-1)
18     accuracy = np.mean(cross_val_score(model, X, y, cv=k, scoring='accuracy'))
19     recallscore = np.mean(cross_val_score(model, X, y, cv=k, scoring='recall'))
20     precision = np.mean(cross_val_score(model, X, y, cv=k, scoring='precision'))
21     # roc_auc = np.mean(cross_val_score(model, X, y, cv=k, scoring='roc_auc'))
22     f1score = np.mean(cross_val_score(model, X, y, cv=k, scoring='f1'))
23
24     # 保存数据
25     df = pd.DataFrame({
26         "Model" : [name],
27         "Accuracy_score" : [accuracy],
28         "Recall_score" : [recallscore],
29         "Precision" : [precision],
30         "f1_score" : [f1score],
31         # "Area_under_curve": [roc_auc],
32     })
33
34     return df
```

k 折交叉验证评估框架 这个框架接受一个输入的模型、数据、标签、模型名称、k 值，给出 k 折交叉验证的评估指标结果，之后的所有模型评估都是用这个框架给出结果从而保证衡量的一致性和公平性。由于我们直接调用了 sklearn 库中的 cross_val_score 函数，这个函数接受数据和标签的评估自然包含了测试集和训练集的划分方案，因此我们不需要进行手动的数据划分模块。

5.2 模型定义和声明

这里定义了一些基础的模型，待后续模型评估实验使用。

```
● ● ●
1 from sklearn.linear_model import LogisticRegression
2 from sklearn.neighbors import KNeighborsClassifier
3 from sklearn.tree import DecisionTreeClassifier
4 from sklearn import tree
5 from sklearn.ensemble import RandomForestClassifier
6 from sklearn.naive_bayes import GaussianNB
7 from sklearn.neural_network import MLPClassifier
8 from sklearn.svm import SVC
9 from lightgbm import LGBMClassifier
10 from xgboost import XGBClassifier
11
12 # 实例化模型
13
14 # 训练时间较短的1-5s左右
15 logit = LogisticRegression()
16 decision_tree = DecisionTreeClassifier(criterion='entropy', max_depth=5, random_state=0)
17 knn = KNeighborsClassifier(n_neighbors=5) # knn的系数
18 gnb = GaussianNB()
19
20 # 训练时间中等短的5-10s
21
22 # lgbm 是一种决策树的集成算法，跟xgboost处于同一地位
23 lgbm_c = LGBMClassifier(boosting_type='gbdt', n_estimators=100, random_state=0)
24
25 # 时间训练时间偏长的1min左右
26 xgc = XGBClassifier(n_estimators=100, eta=0.02, max_depth=15, random_state=0, learning_rate=0.001)
# 1min
27 rfc = RandomForestClassifier(n_estimators=100, random_state=0)
28 mlp_model = MLPClassifier(hidden_layer_sizes=(8,), alpha=0.05, max_iter=50000,
activation='logistic', random_state=0)
29
30
31 # 训练时间较长的3mins往上
32 svc_lin = SVC(kernel='linear', random_state=0, probability=True)
33 svc_rbf = SVC(kernel='rbf', random_state=0, probability=True)
```

6 模型评估

6.1 对比不同数据压缩和变化在同一个模型上的效果

为了查看各种数据降维的效果以及数据上采样后的效果，我们进行了如下实验的设计，写了一个函数，他接受一个模型，会将所有类型的预处理数据和原始数据在这个模型上跑一遍 10 折交叉

验证

```
1 def data_preprocess_test(model):
2     """
3         这个函数输入一个模型，可以用这个模型在压缩过的数据和上采样过的数据上进行训练，
4         给出各种经过预处理数据训练结果，以及最基础的未经预处理的数据训练的结果
5     """
6     # 对比各种不同压缩方法的效果
7
8     ## pca不同程度的压缩
9     pca5_report = model_report_k_cross_val(model, pca_Compress_to_5X, y, 'pac5')
10    pca10_report = model_report_k_cross_val(model, pca_Compress_to_10X, y, 'pac10')
11    pca20_report = model_report_k_cross_val(model, pca_Compress_to_20X, y, 'pac20')
12
13    # 特征提取不同程度的压缩
14    fs5_report = model_report_k_cross_val(model, st_Compress_to_5X, y, 'fs5')
15    fs10_report = model_report_k_cross_val(model, st_Compress_to_10X, y, 'fs10')
16    fs20_report = model_report_k_cross_val(model, st_Compress_to_20X, y, 'fs20')
17
18    ## 对比上采样后的数据和未上采样的数据在同一个模型上的效果
19    resample_report = model_report_k_cross_val(model, X_resampled, y_resampled, 'resampled')
20    base_report = model_report_k_cross_val(model, X, y, 'base')
21
22    draw_result([base_report, pca5_report, pca10_report, pca20_report,
23                 fs5_report, fs10_report, fs20_report,
24                 resample_report])
```

```
1 def data_preprocess_test(model):
2     """
3         这个函数输入一个模型，可以用这个模型在压缩过的数据和上采样过的数据上进行训练，
4         给出各种经过预处理数据训练结果，以及最基础的未经预处理的数据训练的结果
5     """
6     # 对比各种不同压缩方法的效果
7
8     ## pca不同程度的压缩
9     pca5_report = model_report_k_cross_val(model, pca_Compress_to_5X, y, 'pac5')
10    pca10_report = model_report_k_cross_val(model, pca_Compress_to_10X, y, 'pac10')
11    pca20_report = model_report_k_cross_val(model, pca_Compress_to_20X, y, 'pac20')
12
13    # 特征提取不同程度的压缩
14    fs5_report = model_report_k_cross_val(model, st_Compress_to_5X, y, 'fs5')
15    fs10_report = model_report_k_cross_val(model, st_Compress_to_10X, y, 'fs10')
16    fs20_report = model_report_k_cross_val(model, st_Compress_to_20X, y, 'fs20')
17
18    ## 对比上采样后的数据和未上采样的数据在同一个模型上的效果
19    resample_report = model_report_k_cross_val(model, X_resampled, y_resampled, 'resampled')
20    base_report = model_report_k_cross_val(model, X, y, 'base')
21
22    draw_result([base_report, pca5_report, pca10_report, pca20_report,
23                  fs5_report, fs10_report, fs20_report,
24                  resample_report])
```

使用这个函数我跑了三个速度较快的模型进行对比，朴素贝叶斯分类器，逻辑回归，决策树

实验结果如下

Model	Accuracy_score	Recall_score	Precision	f1_score	Area_under_curve
base	0.8039	0.55109	0.65591	0.59863	0.84523
pac5	0.79408	0.50881	0.64206	0.56738	0.83711
pac10	0.79721	0.51684	0.64883	0.57513	0.83682
pac20	0.80091	0.53504	0.65323	0.58797	0.84024
fs5	0.79351	0.4933	0.64613	0.55894	0.83233
fs10	0.7975	0.52595	0.646	0.57963	0.83921
fs20	0.8029	0.54683	0.65497	0.59579	0.84308
resampled	0.76564	0.8067	0.74562	0.77487	0.84667

Model	Accuracy_score	Recall_score	Precision	f1_score	Area_under_curve
base	0.79067	0.58375	0.61245	0.59628	0.83296
pac5	0.79223	0.48256	0.65128	0.54991	0.82987
pac10	0.79038	0.50664	0.63491	0.56102	0.82289
pac20	0.79038	0.50877	0.63284	0.56214	0.82065
fs5	0.79423	0.57197	0.62324	0.59633	0.8342
fs10	0.79152	0.59873	0.61022	0.60364	0.83066
fs20	0.79067	0.58322	0.61242	0.59606	0.82903
resampled	0.76758	0.83344	0.73688	0.78193	0.84351

Model	Accuracy_score	Recall_score	Precision	f1_score	Area_under_curve
base	0.74516	0.78598	0.51362	0.62116	0.83085
pac5	0.78441	0.57303	0.59859	0.58531	0.83021
pac10	0.78598	0.60245	0.59659	0.59933	0.82505
pac20	0.78513	0.61316	0.59298	0.60262	0.8239
fs5	0.67975	0.86303	0.44708	0.58892	0.827
fs10	0.7089	0.82398	0.47299	0.60086	0.82783
fs20	0.74403	0.78866	0.51213	0.6209	0.83096
resampled	0.7546	0.81329	0.72794	0.76818	0.83201

通过上述实验结果我们可以看出来，通过上采样对于模型的效果提升最好，在所有的指标上有提升；同时我们也发现，两种数据降维的压缩方式对于模型反而有了负面效果，至少在贝叶斯分类器上是这样的效果，或许是由于朴素贝叶斯分类器对于冗余数据的抗干扰性较强，或是这些属性本身的相关程度就不大，因此自变量被减少之后，导致了信息量的真实损失，而不是解除了维度诅咒，因此导致效果变差。因此得出结论，指导我在之后的操作过程中直接使用上采样的平衡数据，不要采用降维方法应该会得到更好的效果，因此在后续的实验中我们直接采用重采样后的平衡数据做模型训练。

6.2 使用上采样的数据将各种模型都跑一遍 10 折交叉验证对比效果

代码如下：

```

● ● ●

1 logit_report = model_report_k_cross_val(logit, X_resampled, y_resampled, 'logit')
2 svm_linear_report = model_report_k_cross_val(svc_lin, X_resampled, y_resampled, 'svc_linear')
3 svm_rbf_report = model_report_k_cross_val(svc_rbf, X_resampled, y_resampled, 'svc_rbf')
4 mlp_report = model_report_k_cross_val(mlp_model, X_resampled, y_resampled, 'mlp_model')
5 gnb_report = model_report_k_cross_val(gnb, X_resampled, y_resampled, 'Naive Bayes')
6 decision_report = model_report_k_cross_val(decision_tree, X_resampled, y_resampled, 'decision_tree')
7 rfc_report = model_report_k_cross_val(rfc, X_resampled, y_resampled, 'Random Forest Classifier')
8 xgc_report = model_report_k_cross_val(xgc, X_resampled, y_resampled, 'XGBoost Classifier')
9 knn_report = model_report_k_cross_val(knn, X_resampled, y_resampled, 'KNN Classifier')
10 lgbm_report = model_report_k_cross_val(lgbm_c, X_resampled, y_resampled, 'LGBM Classifier')

```

结果如下

Model	Accuracy_score	Recall_score	Precision	f1_score	Area_under_curve
logit	0.76564	0.8067	0.74562	0.77487	0.84667
svc_linear	0.74327	0.8342	0.70592	0.76467	0.825
svc_rbf	0.77891	0.8158	0.75998	0.78676	0.85265
mlp_model	0.76651	0.80031	0.74979	0.77413	0.84775
Naive Bayes	0.7546	0.81329	0.72794	0.76818	0.83201
decision_tree	0.76758	0.83344	0.73688	0.78193	0.84351
Random Forest Classifier	0.8655	0.97094	0.8605	0.91196	0.97688
LGBM Classifier	0.82869	0.88999	0.79292	0.83849	0.89952
XGBoost Classifier	0.8192	0.86907	0.79059	0.82754	0.88686

从上述的结果我们可以看出来使用了集成思想的后面三个模型的所有指标都显著高于前面的那些单一模型，尤其是随机森林的效果最好，召回率和 auc 面积都达到了 97% 以上的精确率，可以说是非常高的水平了，改良的效果很好，由于我们使用的是 10 折交叉验证，因此这个结果具有可信度，不是随机导致的某次结果很好产生的效果。

而下面是老师提供的 baseline 方法的效果：

Model	Accuracy_score	Recall_score	Precision	f1_score	Area_under_curve	Kappa_metric
Logistic Regression	0.8024	0.5214	0.6633	0.5838	0.8481	0.4567
KNN Classifier	0.7811	0.5134	0.6038	0.5549	0.778	0.411
SVM Classifier Linear	0.8038	0.4626	0.6976	0.5563	0.8308	0.4369
SVM Classifier RBF	0.8053	0.4893	0.688	0.5719	0.7957	0.4504
MLP Classifier	0.8038	0.5294	0.6644	0.5893	0.8476	0.4626
Naive Bayes	0.7292	0.7834	0.4941	0.606	0.833	0.4154
Decision Tree	0.7875	0.5294	0.6168	0.5698	0.8286	0.4298
Random Forest Classifier	0.7854	0.4786	0.6259	0.5424	0.8241	0.4055
LGBM Classifier	0.8003	0.5107	0.6609	0.5762	0.8413	0.4483
XGBoost Classifier	0.7342	0.0	0.0	0.0	0.81	0.0

和未经数据上采样的各项指标对比，我们会发现各个模型要么有很大的提升，要么会有很小的提升，但是都没有出现负面的效果，而其中提升最显著的就是决策树相关的四种算法，说明决策树对于数据不平衡较为敏感，且在数据平衡后，能产生非常好的效果。

6.3 手动实现集成的过程

我们知道决策树是一种不稳定的算法，虽然库中已经提供了他的很多集成算法模型，但是为了手动体验这个过程，我打算自己实现一遍集成的算法看看能否提高性能，具体来说，我采用的 bagging 方法，实现了一个满足 sklearn 库模型接口的类 my_bagging_decision_tree，这个自己实现的模型类中使用了 bagging 的方式集成了多个决策树模型，使用最大数投票原则给出了最终的分类结果，同时，实现了 10 折交叉验证的分数计算接口，实现了和打框架的特匹配和统一，可以直接调用 cross_val_score 函数进行交叉验证，给出模型效果指标。

具体的函数如下：

```

● ● ●

1  from sklearn.model_selection import train_test_split
2  from sklearn.base import BaseEstimator, ClassifierMixin
3  from sklearn.metrics import accuracy_score, precision_score, roc_auc_score, f1_score, recall_score
4
5  class my_bagging_decision_tree(ClassifierMixin, BaseEstimator):
6      def __init__(self, bagging_num=10, sample_rate=0.67):
7          self.decision_tree = DecisionTreeClassifier(criterion='entropy', max_depth=5, random_state=0)
8          self.bagging_num = bagging_num
9          self.sample_rate = sample_rate
10         self.bagging_models = []
11         self.isPredicted = False
12         super().__init__()
13
14     def fit(self, X, y):
15         self.X = X
16         self.y = y
17         self.classes_ = sorted(set(y))
18         for i in range(self.bagging_num):
19             X_sample, _, y_sample, _ = train_test_split(X, y, test_size=(1 - self.sample_rate))
20             self.bagging_models.append(self.decision_tree.fit(X_sample, y_sample))
21         return self
22
23     # 多数投票原则，每一个bagging的模型都跑一遍得到结果
24     def predict(self, X):
25         self.result = np.zeros(len(X))
26         for model in self.bagging_models:
27             self.result = np.vstack((self.result, model.predict(X)))
28         self.result = np.sum(self.result, axis=0) / self.bagging_num
29         self.result = np.where(self.result > 0.5, 1, 0)
30         self.isPredicted = True
31         return self.result
32
33     def precision(self, X, y):
34         # 精确度计算代码
35         if self.isPredicted == False:
36             self.predictions = self.predict(X)
37         return precision_score(y, self.predictions)
38
39     def accuracy(self, X, y):
40         # 精确度计算代码
41         if self.isPredicted == False:
42             self.predictions = self.predict(X)
43         return accuracy_score(y, self.predictions)
44
45     def f1(self, X, y):
46         # 精确度计算代码
47         if self.isPredicted == False:
48             self.predictions = self.predict(X)
49         return f1_score(y, self.predictions)
50
51     def recall(self, X, y):
52         if self.isPredicted == False:
53             self.predictions = self.predict(X)
54         return recall_score(y, self.predictions)
55
56     def roc_auc(self, X, y):
57         if self.isPredicted == False:
58             self.predictions = self.predict(X)
59         return roc_auc_score(y, self.predictions)
60
61     def predict_proba(self, X):
62         # 使用基础模型获取类别的概率
63         self.prob = np.zeros(len(X))
64         for model in self.bagging_models:
65             self.prob = np.vstack((self.prob, model.predict_proba(X)))
66         self.prob = np.sum(self.prob, axis=0) / self.bagging_num
67
68         return self.prob

```

6.4 集成不同类别的分类器

```
1 from sklearn.ensemble import VotingClassifier
2
3 # 创建一个投票分类器
4 ensemble_classifier = VotingClassifier(estimators=[
5     ('logistic_regression', logit),
6     ('decision_tree', decision_tree),
7     ('random_forest', rfc)
8 ], voting='soft')
9
10 mult_classifier_report = model_report_k_cross_val(
    ensemble_classifier, X_resampled, y_resampled, "ensemble")
```

结果如下

	Model	Accuracy_score	Recall_score	Precision	f1_score	\
0	ensemble	0.84215	0.913416	0.799747	0.852614	
	Area_under_curve					
0		0.92651				

通过上面的实验，我们可以发现，如果集成的分类器有稳定的也有不稳定的，不稳定的分类器反而会将本来效果很好的稳定的分类器的性能给拉低，因此没有将弱分类器和强分类器结合的必要。

6.5 决策树调参

```
1 from sklearn.tree import DecisionTreeClassifier
2 from sklearn.model_selection import GridSearchCV
3 parameters = { 'splitter': ( 'best', 'random'),
4                 'criterion': ( "gini", "entropy"),
5                 "max_depth": [* range( 3, 20)],
6 }
7
8 clf = DecisionTreeClassifier(random_state= 25)
9
10 GS = GridSearchCV(clf, parameters, scoring= 'f1', cv= 10)
11
12 refine_decision_tree_report = model_report_k_cross_val(GS,
   X_resampled, y_resampled, 'refined Decision Tree')
```

决策树调参后的结果如下：

Model	Accuracy_score	Recall_score	Precision	f1_score	\
refined Decision Tree	0.877016	0.959704	0.823505	0.886037	
Area_under_curve	0.884341				

我们可以发现相较于不调参的决策树，这个调参后的效果要更好

6.6 尝试自己实现一个 MLP 的模型

这是数据加载器的定义部分

```
1 # 定义数据集
2 import torch
3 import torch.nn as nn
4 import torch.optim as optim
5 from torch.utils.data import DataLoader
6 from torch.utils.data import Dataset
7
8 # 步骤1：准备数据集
9 class CustomDataset(Dataset):
10     def __init__(self, data, labels):
11         self.data = data
12         self.labels = labels
13
14     def __len__(self):
15         return len(self.data)
16
17     def __getitem__(self, idx):
18         x = self.data[idx]
19         y = self.labels[idx]
20         return x, y
21
22
23 X_data = df_model.drop(columns=["customerID", "Churn"])
24 X_label = df_model["Churn"]
25
26 X_data_norm = pd.DataFrame(st.fit_transform(X_data[num_cols]), columns=num_cols)
27 X_data_norm = pd.concat([X_data.drop(num_cols, axis=1), X_data_norm], axis=1)
28
29 import random
30
31 # 假设 data 是你的数据集，labels 是对应的标签
32 data = X_data_norm.values.astype(float) # numpy强制类型转换
33 labels = X_label.values.astype(int)
34 # 将数据类型统一
35 print(type(data))
36 data = torch.from_numpy(data).float()
37 labels = torch.from_numpy(labels).long()
38
39 # 首先，确保 data 和 labels ——对应
40
41 # 随机打乱数据集
42 random.seed(42) # 设置随机种子以确保可重复性
43 print(len(data))
44 shuffled_indices = list(range(len(data)))
45 random.shuffle(shuffled_indices)
46
47 shuffled_data = []
48 shuffled_labels = []
49
50 for i in range(len(data)):
51     shuffled_data.append(data[shuffled_indices[i]])
52     shuffled_labels.append(labels[shuffled_indices[i]])
53
54 # 划分数据集
55 total_samples = len(shuffled_data)
56 train_ratio = 0.7
57 val_ratio = 0.15
58 test_ratio = 0.15
59
60 train_split = int(total_samples * train_ratio)
61 val_split = train_split + int(total_samples * val_ratio)
62
63 train_data = shuffled_data[:train_split]
64 train_labels = shuffled_labels[:train_split]
```

使用了如下定义的模型，输入层有 32 个神经元作为特征提取层，接下来的两层中间层神经元数量分别为 100、2，计算损失函数时还有一个 softmax 层作为输出层，构造代码如下：

```
● ● ●
```

```
1 # 步骤3: 定义模型
2 class SimpleModel(nn.Module):
3     def __init__(self, input_size, hidden_size, num_classes):
4         super(SimpleModel, self).__init__()
5         self.fc1 = nn.Linear(input_size, hidden_size)
6         self.fc2 = nn.Linear(hidden_size, int(hidden_size / 2))
7         self.fc3 = nn.Linear(int(hidden_size / 2), int(hidden_size / 4))
8         self.fc4 = nn.Linear(int(hidden_size / 4), num_classes)
9         self.relu = nn.ReLU()
10
11    def forward(self, x):
12        x = self.fc1(x)
13        x = self.relu(x)
14        x = self.fc2(x)
15        x = self.relu(x)
16        x = self.fc3(x)
17        # x = self.relu(x)
18        # x = self.fc4(x)
19        return x
20
21 model = SimpleModel(input_size=32, hidden_size=100, num_classes=2)
22
23 # 步骤4: 定义损失函数
24 criterion = nn.CrossEntropyLoss()
25
26 # 步骤5: 选择优化器
27 optimizer = optim.Adam(model.parameters(), lr=0.001)
28
29 # 步骤6: 训练模型
30 num_epochs = 1000
31 last_f1 = 0.0
32
33 for epoch in range(num_epochs):
34     for inputs, labels in train_loader:
35         optimizer.zero_grad()
36         outputs = model.forward(inputs)
37         loss = criterion(outputs, labels)
38         loss.backward()
39         optimizer.step()
40
41     correct = 0
42     total = 0
43
44     real_one = 0
45     pre_one_true = 0
46     pre_one_false = 0
47     pre_zero_true = 0
48     pre_zero_false = 0
49
50     # 使用f1值作为过拟合点判断
51     with torch.no_grad():
52         for inputs, labels in test_loader:
53             outputs = model.forward(inputs)
54             _, predicted = torch.max(outputs, 1)
55             for pre in predicted:
56                 if pre == 1:
57                     pre_one_false += (labels != predicted).sum().item()
58                     pre_one_true += (labels == predicted).sum().item()
59                 else:
59                     29
60                     pre_zero_false += (labels != predicted).sum().item()
61                     pre_zero_true += (labels == predicted).sum().item()
62             correct += (predicted == labels).sum().item()
63             total += labels.size(0)
64
65     accuracy = correct / total
66     recall = pre_one_true / (pre_one_true + pre_zero_false)
67     precision = pre_one_true / (pre_one_true + pre_one_false)
68     f1_score = 2 * precision * recall / (precision + recall)
69     if last_f1 > f1_score:
70         torch.save(model.state_dict(), 'model.pkl')
71         print(f'last epoch : {epoch}, Accuracy on the valid set: {100 * accuracy:.2f}%')
```

```
epoch : 0, Accuracy on the valid set: 79.32%
last epoch : 1, Accuracy on the valid set: 78.37%
Accuracy on the test set: 79.81%
Recall on the test set: 60.28%
Precision on the test set: 81.06%
F1-score on the test set: 69.14%
```

虽然这个网络结构较为简单，层数也不多，但是给出的效果是和上述 baseline 中的最好的结果不相上下的。虽然数据集的划分是不变的，但是每次我们训练出来的模型迭代到的最佳性能值都有所差别，这也反应出来了神经网络对函数的拟合是局部最优而不是全局最优，因此导致每次初始化的值不同，迭代的方式的随机性导致的最终得到的模型性能差异有所区别，但是在这个应用场景下，数据量也不是很大，我们可以通过多次尝试找到最大 F1 值作为最终最优的模型输出。细致观察召回率、精确率和准确率这三个指标，我们还能发现如下规律：每次训练出来的模型，在测试集上给出的结果都是十分接近的 80% 左右，没有很大的差异，而主要的变化在于精确率和召回率，如果我们再做横向对比，会发现其他几个 baseline 中的基本模型给出的准确率也都是在 80% 左右，这是一个值得探索的问题，而深度学习对于 f1 值的提升主要来自于精确率和召回率的提升召回率相比于除贝叶斯模型的其他所有模型有 10% 的提升，精确率则超过了所有的 baseline 中的模型 10%-30%，因此综合表现有了提高。可以看到深度学习在数据挖掘领域也有很强的应用价值，我们不需要使用各种复杂的算法进行特征提取和筛选，直接将所有的数据都输入网络，并对网络进行简单的优化，就可以得到远超传统方法的效果，也由此看出来了神经网络拟合函数的能力之强。

尝试对特征进行简单的探索

```
1 # 神经网络给出的重要性特征的探索
2 X_data_headers = X_data.columns.values.tolist()
3 print(X_data_headers)
4 # 根据权重参数进行排序，我们认为绝对值越大的对于最终的结果影响越大，基于这个认知，我们
    进行下述操作
5 # 将每个特征对应的权重值进行求绝对值运算，然后将这个权重值进行累加，用这个累加后的
    结果参与排序，作为重要性指标
6 # 获取特征层的权重参数
7 feature_weight = model.state_dict()['fc1.weight']
8 abs_feature_weight = torch.abs(feature_weight)
9 sum_abs_feature_weight = list(abs_feature_weight.sum(dim=1))
10 print(sum_abs_feature_weight)
11
12
13 for i in range(len(X_data_headers)):
14     for j in range(0, len(X_data_headers) - i - 1):
15         if sum_abs_feature_weight[j] < sum_abs_feature_weight[j + 1]:
16             X_data_headers[j], X_data_headers[j + 1] = X_data_headers[j
17             + 1], X_data_headers[j]
18             sum_abs_feature_weight[j], sum_abs_feature_weight[j + 1] =
19             sum_abs_feature_weight[j + 1], sum_abs_feature_weight[j]
20 print(sum_abs_feature_weight)
21 for i in range(len(X_data_headers)):
22     print(X_data_headers[i], sum_abs_feature_weight[i])
```

根据权重参数进行排序，我们认为绝对值越大的对于最终的结果影响越大，基于这个认知，我们进行下述操作：

- 将每个特征对应的权重值进行求绝对值运算，然后将这个权重值进行累加，用这个累加后的结果参与排序，作为重要性指标
- 获取特征层的权重参数
- 根据这个值进行排序，给出如下结果

feature	importances
Dependents	tensor(3.7915)
tenure_group_Tenure_3	tensor(3.6621)
Contract_One year	tensor(3.6162)
InternetService_Fiber optic	tensor(3.4668)
PaperlessBilling	tensor(3.3612)
PhoneService	tensor(3.3275)
tenure_group_Tenure_2	tensor(3.2807)
PaymentMethod_Credit card (automatic)	tensor(3.2530)
tenure_group_Tenure_1	tensor(3.2289)
TotalCharges	tensor(3.2204)
SeniorCitizen	tensor(3.1937)
StreamingMovies	tensor(3.1157)
OnlineSecurity	tensor(3.0980)
tenure_group_Tenure_5	tensor(3.0936)
InternetService_DSL	tensor(3.0461)
Partner	tensor(3.0274)
PaymentMethod_Electronic check	tensor(2.9997)
MonthlyCharges	tensor(2.9838)
PaymentMethod_Mailed check	tensor(2.9792)
tenure_group_Tenure_4	tensor(2.9679)
MultipleLines	tensor(2.9555)
OnlineBackup	tensor(2.9032)
gender	tensor(2.8875)
tenure_group_Tenure_over_5	tensor(2.8553)
PaymentMethod_Bank transfer (automatic)	tensor(2.8189)
StreamingTV	tensor(2.8047)
TechSupport	tensor(2.8023)
Contract_Month-to-month	tensor(2.7576)
DeviceProtection	tensor(2.6759)
tenure	tensor(2.4041)
Contract_Two year	tensor(2.3888)
InternetService_No	tensor(2.3274)

不足：上面的排序仅考虑了第一层提取特征层的权重，但是神经网络的结构不是这么简单的仅靠着第一层的权重就可以了解到特征的重要性的；而且这个累加的结果貌似快要出现了高位数据失效的情况，累加后趋近于同一个值的情况就要出现了，因此还有待改进，可以将第二层网络的维度减小进行再次的学习观察。因此，为了验证我的这个思路是否正确，我重新跑了即便模型，分别记录他

们给出的重要性排序，结果如下：

feature	importances
StreamingMovies	tensor(3.7906)
SeniorCitizen	tensor(3.6460)
tenure	tensor(3.5574)
TotalCharges	tensor(3.4934)
tenure_group_Tenure_5	tensor(3.4709)
Contract_One year	tensor(3.3876)
InternetService_Fiber optic	tensor(3.2949)
PaymentMethod_Bank transfer (automatic)	tensor(3.2660)
PaymentMethod_Electronic check	tensor(3.2453)
StreamingTV	tensor(3.2239)
Contract_Month-to-month	tensor(3.2172)
Contract_Two year	tensor(3.2161)
TechSupport	tensor(3.1876)
PaperlessBilling	tensor(3.1865)
MonthlyCharges	tensor(3.1462)
tenure_group_Tenure_4	tensor(3.1061)
tenure_group_Tenure_over_5	tensor(3.0845)
OnlineSecurity	tensor(3.0813)
tenure_group_Tenure_2	tensor(3.0652)
PhoneService	tensor(3.0204)
InternetService_No	tensor(2.9747)
gender	tensor(2.9350)
tenure_group_Tenure_1	tensor(2.9194)
PaymentMethod_Mailed check	tensor(2.9128)
InternetService_DSL	tensor(2.9036)
DeviceProtection	tensor(2.8614)
Partner	tensor(2.7935)
PaymentMethod_Credit card (automatic)	tensor(2.7839)
OnlineBackup	tensor(2.7621)
tenure_group_Tenure_3	tensor(2.6276)
Dependents	tensor(2.6248)
MultipleLines	tensor(2.5751)

feature	importances
MonthlyCharges	tensor(3.6594)
tenure	tensor(3.5502)
Dependents	tensor(3.5035)
InternetService_Fiber optic	tensor(3.4966)
PaperlessBilling	tensor(3.4537)
PaymentMethod_Electronic check	tensor(3.4412)
PaymentMethod_Bank transfer (automatic)	tensor(3.3899)
PaymentMethod_Credit card (automatic)	tensor(3.3192)
InternetService_No	tensor(3.2636)
StreamingTV	tensor(3.2280)
InternetService_DSL	tensor(3.1716)
tenure_group_Tenure_over_5	tensor(3.1536)
tenure_group_Tenure_2	tensor(3.1106)
TechSupport	tensor(3.0954)
DeviceProtection	tensor(3.0562)
tenure_group_Tenure_4	tensor(3.0244)
gender	tensor(2.9170)
tenure_group_Tenure_1	tensor(2.8996)
Contract_One year	tensor(2.8675)
PaymentMethod_Mailed check	tensor(2.8583)
TotalCharges	tensor(2.8168)
OnlineSecurity	tensor(2.7831)
Contract_Two year	tensor(2.7820)
tenure_group_Tenure_5	tensor(2.7608)
Contract_Month-to-month	tensor(2.7596)
MultipleLines	tensor(2.7367)
OnlineBackup	tensor(2.6853)
SeniorCitizen	tensor(2.6775)
tenure_group_Tenure_3	tensor(2.6710)
StreamingMovies	tensor(2.5965)
Partner	tensor(2.5248)
PhoneService	tensor(2.3979)

通过上述三个结果的对比，我发现使用这种方法给出的重要性排序具有很强的不确定性，初步断定是我们的测算方式导致的，因为这个方式没有考虑到整个网络里的参数传递，而且出现了高维数据的加和，因此不可靠，需要寻找其他办法给出特征重要性排序。

但是总的来说，这个方法去做预测是一个好的性能很好的模型，是一个成功的探索。

6.7 模型评估总结

最终我对比了所有实验效果，发现随机森林加上采样数据会实现最好的效果，因此最终部署的模型也将使用这个模型。

7 模型的部署与分析

这部分是为了完善整个业务流程，但实际上并没有真正实现，部署即是把挖掘结果以要求的方式呈现给用户，本阶段要解决一些实际问题，比如长期运行的模型是否有足够的机器来支撑，数据量以及并发程度会不会造成部署的服务器出现问题。部署是一个挖掘项目的结束，也是一个数据挖掘项目的开始。

8 实验小结

本实验是在老师提供的 baseline 的基础上做进一步的性能提升，尝试更多的方法，体会数据挖掘过程的整个流程，这部分主要由 baseline 给的思路和上课所学了解到，本人在实验主要贡献有以下几个方面：

8.1 工程方面

- 在与在 baseline 的基础上将整个流程用 markdown 的标题重构组织了一遍，使得整个流程的数据流更加清晰规范，之后想要扩展这个流程的话只需要在相应的模块进行更改即可，提升了代码的可读性和友好性
- 将各种经过不同操作处理的数据进行了分别保存和规范命名，方便后续实验和后人使用，也方便加入新的数据处理方式

- 对使用不同的模型和数据的情况进行了多种实验，筛选出了 baseline 中最好数据挖掘效果的模型

8.2 算法方面

- 数据压缩方面，我加入了 PCA 主成分分析方法的数据压缩，评测了数据降维的效果，发现各种降维的方式在这个业务数据背景中效果并不好，有负面的影响，因此将 baseline 中的特征检定筛选模块剔除了
- 将 10 折交叉验证封装进了原本代码中的 report_model 函数中，提高了模型评估的公平性，消除了偶然性
- 使用上采样流失客户数据的方式解决了数据不平衡的问题，使得模型效果有了很大的提升
- 自己实现了决策树模型的集成过程
- 决策树调参（baseline 中已有）
- 自己实现了 MLP 算法并在数据上进行了测评取得了较好的效果

以上就是实验的所有内容。