

实践报告

🌸 项目介绍

赛题地址: kaggle.com

项目地址: [泰坦尼克生存预测](#)

Titanic 数据集是非常适合数据科学和机器学习新手入门练习的数据集。数据集为 1912 年泰坦尼克号沉船事件中一些船员的个人信息以及存活状况。这些历史数据已经非分为训练集和测试集, 需要根据训练集训练出合适的模型并预测测试集中的存活状况。

🌸 人员分工

李宁: 基础代码框架, 包含数据预处理、可视化, 随机森林模型构建、调参, 模型评估。

江岩: 优化数据预处理(**Age** 缺失值使用随机森林模型填充), 挖取新特征

李乐禎: 优化数据预处理(**Age** 和 **Fare** 离散化处理, **Ticket** 项的利用)

报告撰写: 每人负责自己的部分, 其余部分为合作完成

🌸 任务流程

数据预处理

先预览一下数据, 看数据集提供了多少特征, 分别是代表什么, 然后看各项有没有 **NaN** 的值。代码如下:

```
1 # 数据加载
2 train = pd.read_csv("../data/train.csv")
3 test = pd.read_csv("../data/test.csv")
4 # 数据规模
5 print("train shape:", train.shape)
6 print("test shape:", test.shape)
7 # 数据预览
8 display(train.head(10))
9 display(train.isna().sum())
10 display(test.isna().sum())
```

数据清洗

然后考虑什么数据可以删除、数据的缺失值用什么填充、能否通过提取某些数据让特征更鲜明。对于 **Embarked** 项，表示乘客登船的港口，一共只有三种可能取值，且只缺失了两个，因此直接用众数填充。对于 **Cabin** 项，缺失条目太多，而且特征取值格式不统一，考虑直接剔除，或者只保留代表是否缺失的二元取值（测试后发现保留二元取值效果略好），对于 **Ticket** 项，是独一无二的，直接剔除即可（后面发现存在多人用一张票的情况，可以优化代码）

对于 **Age** 项，缺失的条目较多，因此先计算其与其他项的相关程度，得出相关程度最大的是 **Pclass**，即船舱等级（很合理，富人一般不会太年轻）。那么将数据按 **Pclass** 分类，再取出在同一分类中的年龄中位数填充缺失值。对于 **Name** 项，为了简单起见，只保留称谓的特征，用正则匹配提取出称谓。（姓氏也可以作为特征，可以作为优化方向）

定义一个函数实现数据清洗，代码如下：

```

1 def data_process(data):
2     # 缺失的登船地点直接用众数填充
3     data['Embarked'].fillna(data['Embarked'].mode()[0], inplace=True)
4     # 统计年龄与其他数字项的相关程度
5     # tmp = data[['PassengerId', 'Survived', 'Pclass', 'Age', 'SibSp',
6     #             'Parch', 'Fare']]
7     # display(tmp.corr()['Age'].sort_values())
8     # 计算各个 Pclass 的年龄中位数
9     display(data.groupby('Pclass')['Age'].median())
10    # 用各个 Pclass 的年龄中位数填充
11    data['Age'].fillna(data.groupby('Pclass')['Age'].transform('median'),
12                      inplace=True)
13    # 船舱号缺失值过多，只考虑有无船舱号
14    data['Cabin'].fillna(0, inplace=True)
15    data.loc[data['Cabin'] != 0, 'Cabin'] = 1
16    # 直接删除
17    # train.drop(['Cabin'], axis=1, inplace=True)
18    # 姓名，只保留称谓
19    data['Name'] = data['Name'].str.extract(' ([A-Za-z]+)\.',
20    expand=False)
21    # Ticket 是独一无二的，直接删除
22    data.drop(['Ticket'], axis=1, inplace=True)
23    return data

```

数据可视化

为了方便提取相关性高的特征，可以将每个特征值作为 **x** 轴，对应的总人数作为 **y** 轴（绿色代表存活的人数，蓝色代表未存活）画出堆叠直方图。对于非数值型的特征，比如 **Sex**，需要先映射为数值型，再参与画图。

定义画图函数如下：

```

1  # 画出特征与对应存活人数的堆叠柱状图
2  def plot_bar(feature, type=0):
3      # 离散型特征
4      if type == 0:
5          # 横坐标为特征取值，可能是非数值型
6          value = train[feature].unique()
7          # 横坐标排序
8          value.sort()
9          y1 = []
10         y2 = []
11         for x_i in value:
12             y1.append(train.loc[(train[feature] == x_i) &
(train['Survived'] == 0)].shape[0])
13             y2.append(train.loc[(train[feature] == x_i) &
(train['Survived'] == 1)].shape[0])
14         # 画图
15         x = range(len(value))
16         plt.bar(x, y1, color='b', label='dead')
17         plt.bar(x, y2, bottom=y1, color='g', label='alive')
18         # 设置横坐标刻度
19         plt.xticks(x, value)
20         plt.legend()
21         plt.title(feature)

```

对于连续型的特征，比如 `Age` 和 `Fare`，可以设置横坐标为区间，再画图。相关处理过程如下：

```

1  # 年龄划分为四个区间：0~10为儿童；10~30为年轻人；30~60为中年人；60以上为老年人
2  # 新建一列 Age_cut，字符串类型
3  train['Age_cut'] = '(0, 10]'
4  # 将年龄划分为四个区间
5  train.loc[train['Age'] > 10, 'Age_cut'] = '(10, 30]'
6  train.loc[train['Age'] > 30, 'Age_cut'] = '(30, 60]'
7  train.loc[train['Age'] > 60, 'Age_cut'] = '(60, 100]'
8  # 画出年龄分布
9  plt.figure(figsize=(15, 10))
10 plt.subplots_adjust(hspace=0.5)
11 plt.subplot(2, 2, 1)
12 plot_bar('Age_cut')
13 # 票价每 50 为一个区间
14 train['Fare_cut'] = '<=50'

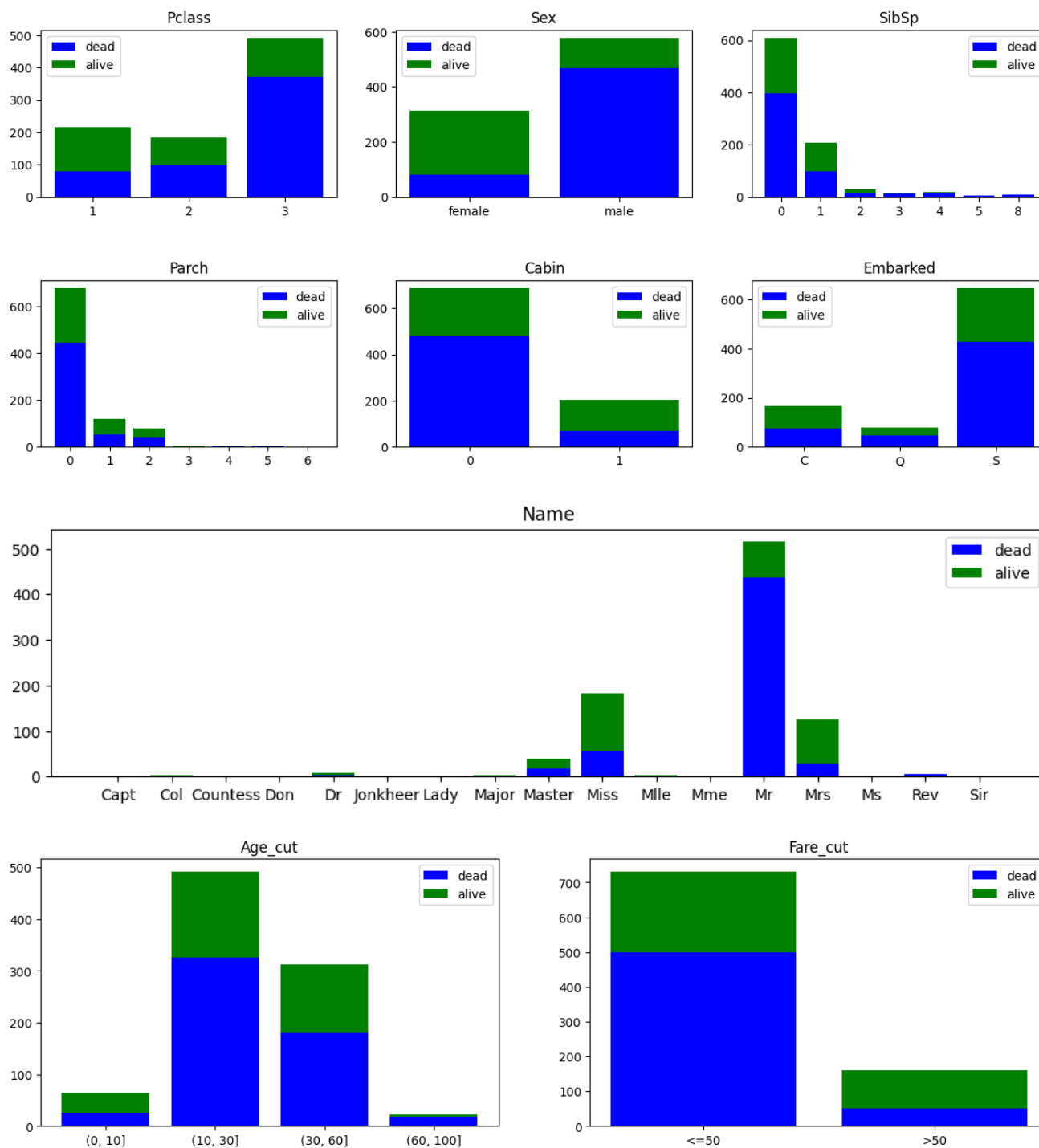
```

```

15 train.loc[train['Fare'] > 50, 'Fare_cut'] = '>50'
16 plt.subplot(2, 2, 2)
17 plot_bar('Fare_cut')
18 plt.show()

```

对每个特征值调用一次函数，可以画出图像结果如下：



模型构建

特征工程

首先需要根据模型的输入要求修改特征，比如对于离散非数值型使用 **one-hot** 编码，对于数值型可能需要标准化处理。定义特征因子化和归一化函数如下：

```

1  # 特征因子化, one-hot 编码
2  def set_numeralization(data):
3      # 针对定类性属性进行因子化, 分别有Embarked, Sex, Pclass
4      dummies_Embarked = pd.get_dummies(data['Embarked'], prefix='Embarked')
5      dummies_Sex = pd.get_dummies(data['Sex'], prefix='Sex')
6      dummies_Pclass = pd.get_dummies(data['Pclass'], prefix='Pclass')
7      # 将 Name 转为数值型, 保留人数大于 10 的称谓
8      name = data['Name'].value_counts()
9      name = name[name > 10].index
10     data.loc[~data['Name'].isin(name), 'Name'] = 0
11     dummies_Name = pd.get_dummies(data['Name'], prefix='Name')
12     # 将新的属性拼合
13     df = pd.concat([data, dummies_Name, dummies_Embarked, dummies_Sex,
14                     dummies_Pclass], axis=1)
15     # 将旧的属性剔除
16     df.drop(['Name', 'Pclass', 'Sex', 'Embarked'], axis=1, inplace=True)
17     return df
18
19 # 特征归一化
20 def set_normalization(df):
21     scaler = preprocessing.StandardScaler()
22     age_scale_param = scaler.fit(df['Age'].values.reshape(-1,1))
23     df['Age_scaled'] =
24     scaler.fit_transform(df['Age'].values.reshape(-1,1), age_scale_param)
25     fare_scale_param = scaler.fit(df['Fare'].values.reshape(-1,1))
26     df['Fare_scaled'] =
27     scaler.fit_transform(df['Fare'].values.reshape(-1,1), fare_scale_param)
28     df.drop(['Age', 'Fare'], axis=1, inplace=True)
29     return df

```

修改后的特征如下表所示：

PassSp	Parch	Cabin	Name_0	Name_Master	Name_Miss	Name_Mr	Name_Mrs	Embarked_C	Embarked_Q	Embarked_S	Sex_female	Sex_male	Pclass_1	Pclass_2	Pclass_3	Age_scaled	Fare_scaled
1	0	0	False	False	False	True	False	False	False	True	False	True	False	False	True	-0.533834	-0.502445
1	0	1	False	False	False	False	True	True	False	False	True	False	True	False	False	0.674891	0.786845
0	0	0	False	False	True	False	False	False	False	True	True	False	False	False	True	-0.231653	-0.488854
1	0	1	False	False	False	False	True	False	False	True	True	False	True	False	False	0.448255	0.420730
0	0	0	False	False	False	True	False	False	False	True	False	True	False	False	True	0.448255	-0.486337
0	0	0	False	False	True	True	False	False	True	False	False	True	False	False	True	-0.382743	-0.478116
0	0	1	False	False	False	True	False	False	False	True	False	True	True	False	False	1.883615	0.395814
3	1	0	False	True	False	False	False	False	False	True	False	True	False	False	True	-2.044739	-0.224083
0	2	0	False	False	False	False	True	False	False	True	True	False	False	False	True	-0.156107	-0.424256
1	0	0	False	False	False	False	True	True	False	False	True	False	False	True	False	-1.138196	-0.042956

模型训练

先将训练集分出一部分作为验证集，使用 `train_test_split` 函数。这里以随机森林模型为例，直接从 `sklearn` 库中调用即可。代码如下：

```

1  # 随机森林模型
2  from sklearn.ensemble import RandomForestClassifier
3  rf = RandomForestClassifier()
4  # 训练模型
5  rf.fit(x_train, y_train)
6  # 模型得分
7  print("Accuracy:", rf.score(x_test, y_test))
8  # 预测
9  y_test = rf.predict(x_exam)
10 result = pd.DataFrame({'PassengerId': test['PassengerId'], 'Survived':
    y_test})
11 # 结果写入文件
12 result.to_csv("../result/rf_predict.csv", index=False)

```

得分: 0.7836

模型调优

使用网格搜索找出最适合的模型参数，由于逐个调时间太长，所以使用 `RandomizedSearchCV`，加快调参速度。相关代码如下：

```

1  # 调参 n_estimators, max_depth, min_samples_split, min_samples_leaf
2  # 使用 RandomizedSearchCV, 速度更快
3  from sklearn.model_selection import RandomizedSearchCV
4  # 参数范围
5  param_grid = {'n_estimators': np.arange(10, 100), 'max_depth':
    np.arange(1, 10), 'min_samples_split': np.arange(2, 10),
    'min_samples_leaf': np.arange(1, 10)}
6  grid = RandomizedSearchCV(RandomForestClassifier(), param_grid, cv=5)
7  grid.fit(x_train, y_train)
8  print(grid.best_params_)
9  rf_best = grid.best_estimator_
10 # 模型得分
11 print("Accuracy:", rf_best.score(x_test, y_test))
12 # 预测

```


```

13 y_exam = rf_best.predict(x_exam)
14 # 保存结果
15 result = pd.DataFrame({'PassengerId': test['PassengerId'], 'Survived':
    y_exam})
16 result.to_csv("../result/rf_best_predict.csv", index=False)

```

得分 0.8134

可以看到，调参后模型表现提升。将预测结果上传 [Kaggle](#) 网站上，得分 0.78229，排名 3184，还有比较大的优化空间。

3184	ningli03		0.78229	10	14h
 Your Best Entry! Your most recent submission scored 0.78229, which is the same as your previous score. Keep trying!					

模型评估

交叉验证

使用 `cross_val_score` 函数即可，代码如下：

```

1 # 随机森林
2 rf_cv = cross_val_score(rf_best, x_train, y_train, cv=5)
3 print("rf_cv:", rf_cv.mean())

```

得分：0.8234

混淆矩阵

使用 `confusion_matrix` 函数，代码如下：

```

1 # 随机森林
2 rf_cm = confusion_matrix(y_train, rf_best.predict(x_train))
3 rf_precision = rf_cm[1, 1] / (rf_cm[0, 1] + rf_cm[1, 1])
4 rf_recall = rf_cm[1, 1] / (rf_cm[1, 0] + rf_cm[1, 1])
5 rf_f1 = 2 * rf_precision * rf_recall / (rf_precision + rf_recall)

```

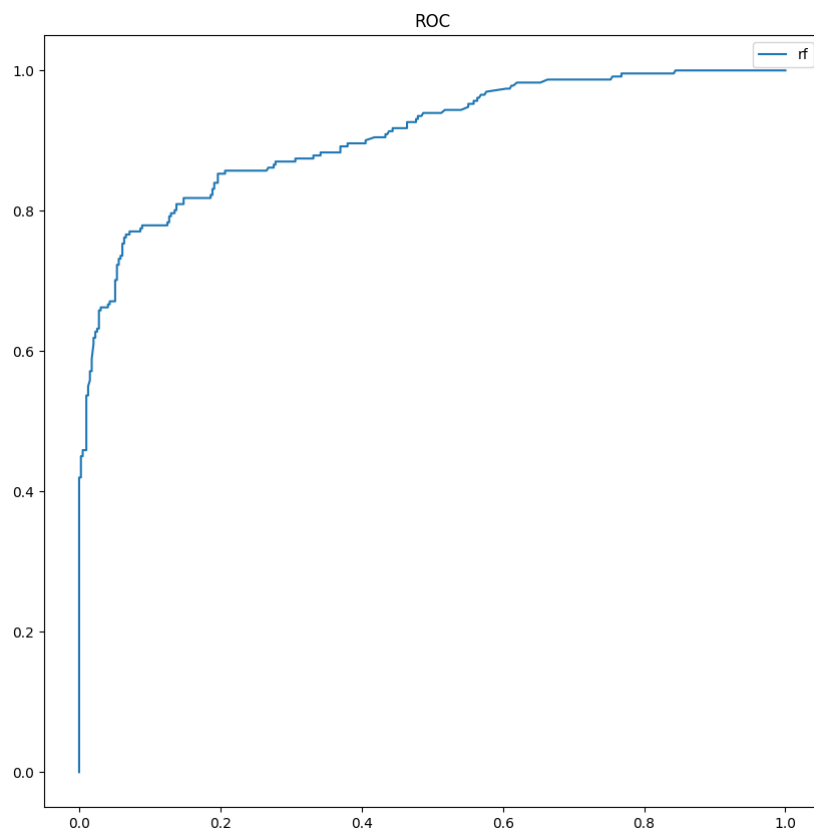
得分分别为：0.8808， 0.7359， 0.8019

ROC 曲线

使用 `roc_curve` 函数，代码如下：

```
1 # 随机森林
2 rf_fpr, rf_tpr, rf_thresholds = roc_curve(y_train,
    rf_best.predict_proba(x_train)[: , 1])
3 # 画出 ROC 曲线
4 plt.figure(figsize=(10, 10))
5 plt.title('ROC')
6 plt.plot(rf_fpr, rf_tpr, label='rf')
7 plt.legend()
8 plt.show()
```

结果如下图：



数据再调整

离散化数据与ticket项的利用

在初步完成基础代码后，再对数据做进一步的调整，例如离散化数据减少噪声，对ticket项进行利用等

修改部分代码如下：


```

1 # Ticket 是独一无二的, 直接删除
2 #data.drop(['Ticket'], axis=1, inplace=True)
3
4 #用 -1 填充 Ticket
5 data['Ticket'].fillna(-1, inplace=True)
6 return data

```

```

1 def set_planarization(data):
2     #离散化 Age Fare
3     #使用pd中的cut离散化
4     #data['Age'] = pd.cut(data['Age'], bins=4, labels=[1, 2, 3, 4])
5     #data['Fare'] = pd.cut(data['Fare'], bins=2, labels=[1, 2])
6     #手动离散化
7     for i in range(0,data['Age'].size):
8         if (0 < data.loc[i,'Age'] and data.loc[i,'Age'] <= 10):
9             data.loc[i,'Age'] = 1
10        if (10 < data.loc[i,'Age'] and data.loc[i,'Age'] <= 30):
11            data.loc[i,'Age'] = 2
12        if (30 < data.loc[i,'Age'] and data.loc[i,'Age'] <= 60):
13            data.loc[i,'Age'] = 3
14        if (60 < data.loc[i,'Age'] and data.loc[i,'Age'] <= 100):
15            data.loc[i,'Age'] = 4
16    for i in range(0,data['Fare'].size):
17        if (data.loc[i,'Fare'] <= 50):
18            data.loc[i,'Fare'] = 1
19        if (data.loc[i,'Fare'] > 50):
20            data.loc[i,'Fare'] = 2
21    #处理ticket 同一张ticket被多个人使用, 则他们在船上有同伴, 赋值为1, 否则赋值为0
22    #使用pd里的duplicated查重重复值
23    #data['Companion'] = data.duplicated(subset='Ticket',
24    keep=False).astype(int)
25    #手动
26    for i in range(0,data['Ticket'].size):
27        flag = 0
28        #-1表示缺失
29        if data.loc[i,'Ticket'] == -1:
30            data.loc[i,'Ticket'] = 0
31            continue
32        if data.loc[i,'Ticket'] != 1:
33            for j in range(i + 1,data['Ticket'].size):
34                if (data.loc[i,'Ticket'] == data.loc[j,'Ticket']):

```

```

34         flag = 1
35         data.loc[j, 'Ticket'] = flag
36         data.loc[i, 'Ticket'] = flag
37     return data

```

随机森林预测模型填充Age缺失值

根据算的年龄和其他特征的相关性，发现Age和Pclass, SibSp, Parch相关系数较高，因此利用这三个特征和随机森林模型，对年龄的缺失值进行预测。

```

1  def fill_age(data):
2      # 将数据集划分为有缺失值和无缺失值的两部分
3      data_with_age = data.dropna(subset=['Age'])
4      data_without_age = data[data['Age'].isnull()]
5      # 将数据集划分为有缺失值和无缺失值的两部分
6      data_with_age = data.dropna(subset=['Age'])
7      data_without_age = data[data['Age'].isnull()]
8      # 定义特征和目标变量。根据前面获取的其他特征与Age的相关性，只选择Pclass、SibSp、
      Parch这三个特征作为训练模型的特征。
9      features = ['Pclass', 'SibSp', 'Parch']
10     target = 'Age'
11     # 创建训练集和测试集
12     X_train = data_with_age[features]
13     y_train = data_with_age[target]
14     X_test = data_without_age[features]
15     # 初始化随机森林回归模型
16     rf_model = RandomForestRegressor()
17     # 训练模型
18     rf_model.fit(X_train, y_train)
19     # 预测缺失值
20     predicted_age = rf_model.predict(X_test)
21     # 填充缺失值
22     data.loc[data['Age'].isnull(), 'Age'] = predicted_age
23     # return data

```

运行结果对比如下：

	模型融合准确率	交叉验证 lr	svc	knn	dt	rf	voting
中位数	0.813433	0.839445	0.829845	0.812167	0.829896	0.842670	0.833032

	模型融合准确率	交叉验证 lr	svc	knn	dt	rf	voting
随机森林	0.809701	0.837845	0.828245	0.812180	0.831483	0.833070	0.831445

中位数填充混淆矩阵结果如下：

	precision	recall	f1
lr	0.820755	0.753247	0.785553
svc	0.867725	0.709957	0.780952
knn	0.875648	0.731602	0.797170
dt	0.870466	0.727273	0.792453
rf	0.918919	0.735931	0.817308
voting	0.865672	0.753247	0.805556

随机森林预测填充混淆矩阵结果如下：

	precision	recall	f1
lr	0.820755	0.753247	0.785553
svc	0.864583	0.718615	0.784870
knn	0.875648	0.731602	0.797170
dt	0.870466	0.727273	0.792453
rf	0.871134	0.731602	0.795294
voting	0.864322	0.744589	0.800000

可以看到，中位数填充和随机森林预测填充的效果差异并不显著。可能是其他特征和Age属性的相关性不够强导致的。

挖掘新特征Title

旅客姓名数据中包含头衔信息，不同头衔也可以反映旅客的身份，而不同身份的旅客其生存率有可能会出现较大差异。因此我们通过Name特征提取旅客头衔Title信息，并分析Title与Survived之间的关系。

```

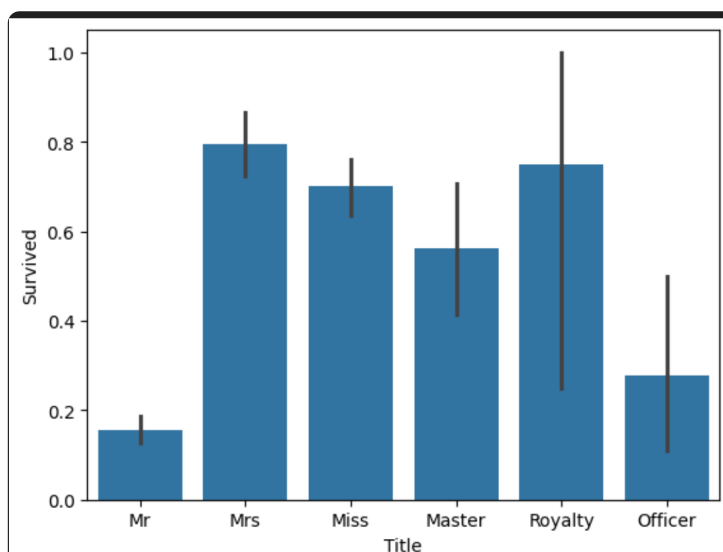
1 def add_title(full):
2     #构造新特征Title

```

```

3     full['Title']=full['Name'].map(lambda x:x.split(',')[1].split('.')[0].strip())
4     #查看title数据分布
5     # display(full['Title'].value_counts())
6     # display(full)
7     #将title信息进行整合
8     TitleDict = {}
9     TitleDict['Mr'] = 'Mr'
10    TitleDict['Mlle'] = 'Miss'
11    TitleDict['Miss'] = 'Miss'
12    TitleDict['Master'] = 'Master'
13    TitleDict['Jonkheer'] = 'Master'
14    TitleDict['Mme'] = 'Mrs'
15    TitleDict['Ms'] = 'Mrs'
16    TitleDict['Mrs'] = 'Mrs'
17    TitleDict['Don'] = 'Royalty'
18    TitleDict['Sir'] = 'Royalty'
19    TitleDict['the Countess'] = 'Royalty'
20    TitleDict['Dona'] = 'Royalty'
21    TitleDict['Lady'] = 'Royalty'
22    TitleDict['Capt'] = 'Officer'
23    TitleDict['Col'] = 'Officer'
24    TitleDict['Major'] = 'Officer'
25    TitleDict['Dr'] = 'Officer'
26    TitleDict['Rev'] = 'Officer'
27
28    full['Title'] = full['Title'].map(TitleDict)
29    # display(full['Title'].value_counts())
30    sns.barplot(data=full, x='Title', y='Survived')

```



可以看到，Mr, Officer存活率较低，可能因为他们阶级较低，且女士优先

下面利用**Title**属性训练模型，得到下面的对比结果：

	模型融合准确率	交叉验证lr	svc	knn	dt	rf	voting
Name	0.813432	0.839445	0.829845	0.812167	0.829896	0.842670	0.833032
Title	0.813432	0.839445	0.829832	0.810554	0.836309	0.820219	0.833019

除dt模型外，其余模型的准确率都有一定程度下降

	precision	recall	f1
lr	0.819048	0.744589	0.780045
svc	0.867725	0.709957	0.780952
knn	0.875648	0.731602	0.797170
dt	0.886364	0.675325	0.766585
rf	0.885246	0.701299	0.782609
voting	0.859296	0.740260	0.795349

总结与收获

由于我们组三个人都没有数据分析基础，所以选了 **Titanic** 这个入门级的简单题目。整个过程实践下来学到了很多，除了数据采集和存储部分题目给好了（这一部分在 **Web 信息处理与应用** 这门课得到了实践），基本上完整的体验了数据分析的全流程，也熟悉了 **python** 机器学习相关库的使用，收获颇丰。