April 10, 2024

[16]:
```python
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings('ignore')
df = pd.read_csv('/Users/liziqi/Downloads/Churn_Modelling.csv')
df.head()
```

[16]:
```
   RowNumber  CustomerId   Surname  CreditScore Geography  Gender  Age  \
0          1    15634602  Hargrave          619    France  Female   42
1          2    15647311      Hill          608     Spain  Female   41
2          3    15619304      Onio          502    France  Female   42
3          4    15701354      Boni          699    France  Female   39
4          5    15737888  Mitchell          850     Spain  Female   43

   Tenure    Balance  NumOfProducts  HasCrCard  IsActiveMember  \
0       2       0.00              1          1               1
1       1   83807.86              1          0               1
2       8  159660.80              3          1               0
3       1       0.00              2          0               0
4       2  125510.82              1          1               1

   EstimatedSalary  Exited
0        101348.88       1
1        112542.58       0
2        113931.57       1
3         93826.63       0
4         79084.10       0
```

[17]:
```python
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   RowNumber        10000 non-null  int64
 1   CustomerId       10000 non-null  int64
 2   Surname          10000 non-null  object
```

```
 3   CreditScore      10000 non-null  int64
 4   Geography        10000 non-null  object
 5   Gender           10000 non-null  object
 6   Age              10000 non-null  int64
 7   Tenure           10000 non-null  int64
 8   Balance          10000 non-null  float64
 9   NumOfProducts    10000 non-null  int64
 10  HasCrCard        10000 non-null  int64
 11  IsActiveMember   10000 non-null  int64
 12  EstimatedSalary  10000 non-null  float64
 13  Exited           10000 non-null  int64
dtypes: float64(2), int64(9), object(3)
memory usage: 1.1+ MB
```

[18]:
```
# Remove rows with null values
df.dropna(inplace=True)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 14 columns):
 #   Column           Non-Null Count  Dtype
---  ------           --------------  -----
 0   RowNumber        10000 non-null  int64
 1   CustomerId       10000 non-null  int64
 2   Surname          10000 non-null  object
 3   CreditScore      10000 non-null  int64
 4   Geography        10000 non-null  object
 5   Gender           10000 non-null  object
 6   Age              10000 non-null  int64
 7   Tenure           10000 non-null  int64
 8   Balance          10000 non-null  float64
 9   NumOfProducts    10000 non-null  int64
 10  HasCrCard        10000 non-null  int64
 11  IsActiveMember   10000 non-null  int64
 12  EstimatedSalary  10000 non-null  float64
 13  Exited           10000 non-null  int64
dtypes: float64(2), int64(9), object(3)
memory usage: 1.1+ MB
```

[19]:
```
pd.set_option('display.float_format', '{:.2f}'.format)
df.describe()
```

[19]:
```
       RowNumber  CustomerId  CreditScore      Age   Tenure  Balance  \
count   10000.00    10000.00     10000.00 10000.00 10000.00 10000.00
mean     5000.50 15690940.57       650.53    38.92     5.01 76485.89
std      2886.90    71936.19        96.65    10.49     2.89 62397.41
min         1.00 15565701.00       350.00    18.00     0.00     0.00
```

```
25%      2500.75 15628528.25       584.00     32.00      3.00       0.00
50%      5000.50 15690738.00       652.00     37.00      5.00   97198.54
75%      7500.25 15753233.75       718.00     44.00      7.00  127644.24
max     10000.00 15815690.00       850.00     92.00     10.00  250898.09

       NumOfProducts  HasCrCard  IsActiveMember  EstimatedSalary   Exited
count       10000.00   10000.00        10000.00         10000.00 10000.00
mean            1.53       0.71            0.52        100090.24     0.20
std             0.58       0.46            0.50         57510.49     0.40
min             1.00       0.00            0.00            11.58     0.00
25%             1.00       0.00            0.00         51002.11     0.00
50%             1.00       1.00            1.00        100193.91     0.00
75%             2.00       1.00            1.00        149388.25     0.00
max             4.00       1.00            1.00        199992.48     1.00
```

[20]:
```python
print(df['Geography'].value_counts())
print(df['Gender'].value_counts())
```

```
Geography
France     5014
Germany    2509
Spain      2477
Name: count, dtype: int64
Gender
Male      5457
Female    4543
Name: count, dtype: int64
```

[21]:
```python
#churn and gender were cross-analyzed
df[['Gender','Exited']].groupby(['Gender'],as_index=False).mean().
↪sort_values(by='Exited',ascending=False)
```

[21]:
```
   Gender  Exited
0  Female    0.25
1    Male    0.16
```

[22]:
```python
df[['Geography','Exited']].groupby(['Geography'],as_index=False).mean().
↪sort_values(by='Exited',ascending=False)
```

[22]:
```
  Geography  Exited
1   Germany    0.32
2     Spain    0.17
0    France    0.16
```

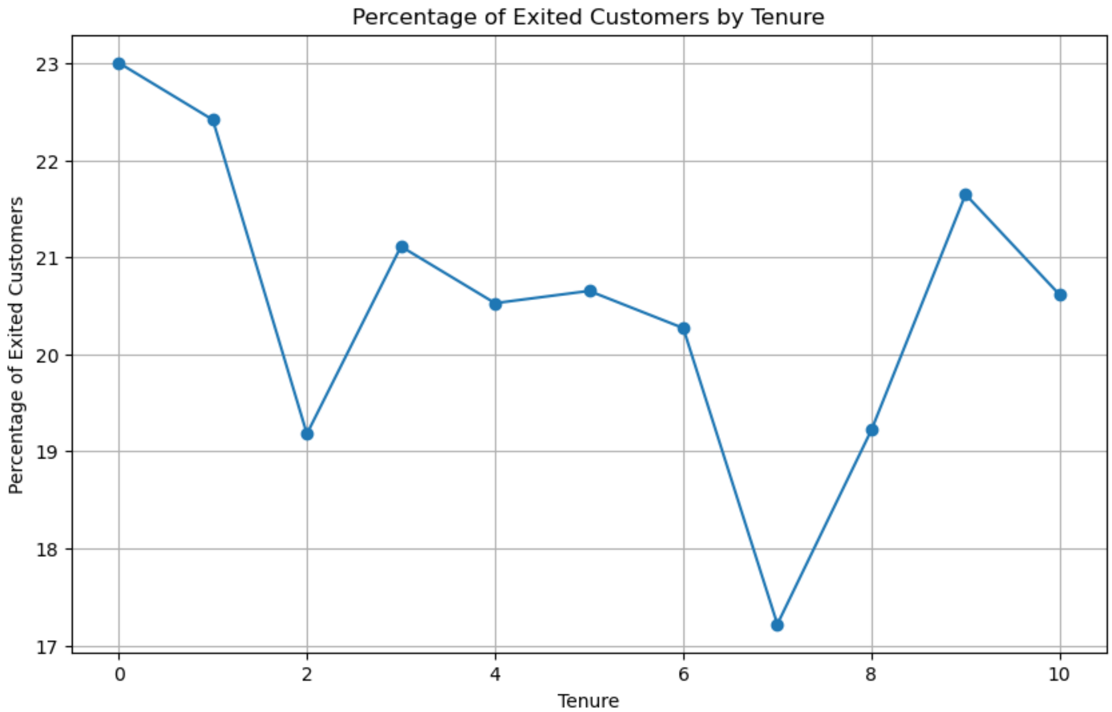
[23]:
```python
df[['IsActiveMember','Exited']].groupby(['IsActiveMember'],as_index=False).
↪mean().sort_values(by='Exited',ascending=False)
```

[23]:
```
   IsActiveMember  Exited
0               0    0.27
1               1    0.14
```

[24]:
```python
df[['Age','Exited']].groupby(['Age'],as_index=False).mean().
 ↪sort_values(by='Exited',ascending=False)
```

[24]:
```
    Age  Exited
38   56    0.71
34   52    0.63
36   54    0.61
37   55    0.59
33   51    0.55
..  …      …
65   83    0.00
57   75    0.00
67   85    0.00
68   88    0.00
69   92    0.00

[70 rows x 2 columns]
```

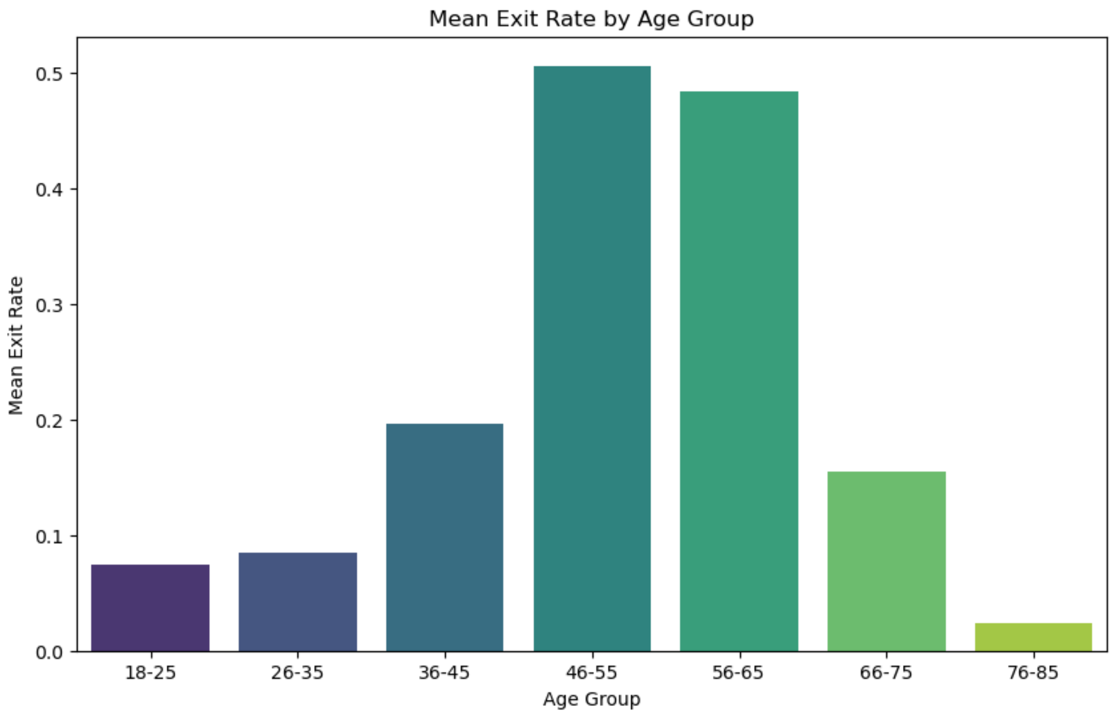
[25]:
```python
import matplotlib.pyplot as plt

# Grouping by Tenure value and calculating the proportion of Exited=1 under␣
 ↪each Tenure value
tenure_exit_percentage = df.groupby('Tenure')['Exited'].mean() * 100


plt.figure(figsize=(10, 6))
plt.plot(tenure_exit_percentage.index, tenure_exit_percentage.values,␣
 ↪marker='o', linestyle='-')
plt.title('Percentage of Exited Customers by Tenure')
plt.xlabel('Tenure')
plt.ylabel('Percentage of Exited Customers')
plt.grid(True)
plt.show()
```



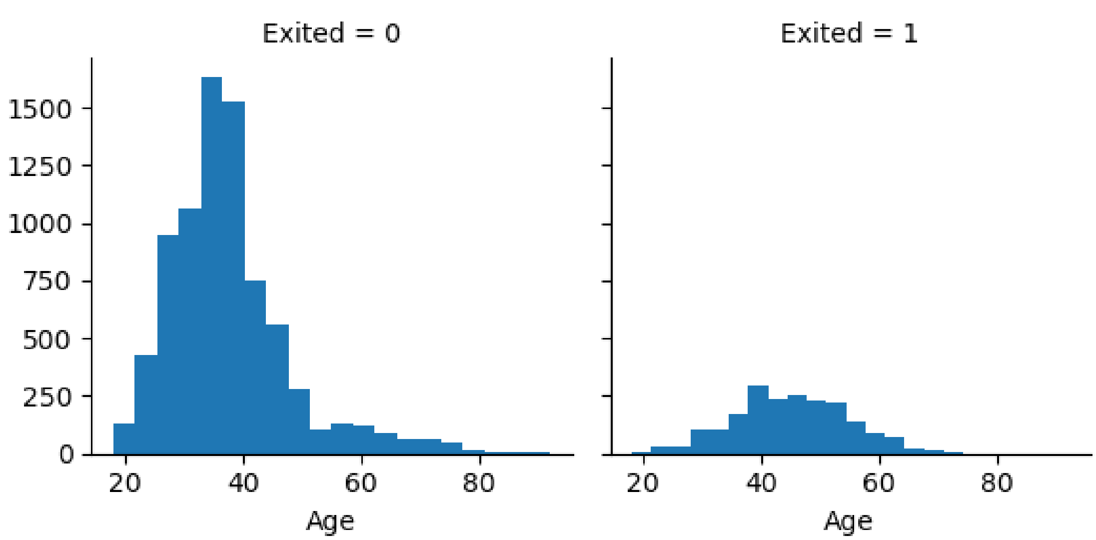
[26]:
```
import seaborn as sns


bins = [18, 25, 35, 45, 55, 65, 75, 85]
labels = ['18-25', '26-35', '36-45', '46-55', '56-65', '66-75', '76-85']
df['Age_Group'] = pd.cut(df['Age'], bins=bins, labels=labels)

# Exit ratios are averaged and ranked according to age groups
result = df[['Age_Group', 'Exited']].groupby(['Age_Group'], as_index=False).
    ↪mean().sort_values(by='Exited', ascending=False)

print(result)

plt.figure(figsize=(10, 6))
sns.barplot(x='Age_Group', y='Exited', data=result, palette='viridis')
plt.title('Mean Exit Rate by Age Group')
plt.xlabel('Age Group')
plt.ylabel('Mean Exit Rate')
plt.show()
```

```
  Age_Group  Exited
3     46-55    0.51
4     56-65    0.48
2     36-45    0.20
```

```
5      66-75     0.16
1      26-35     0.08
0      18-25     0.07
6      76-85     0.02
```

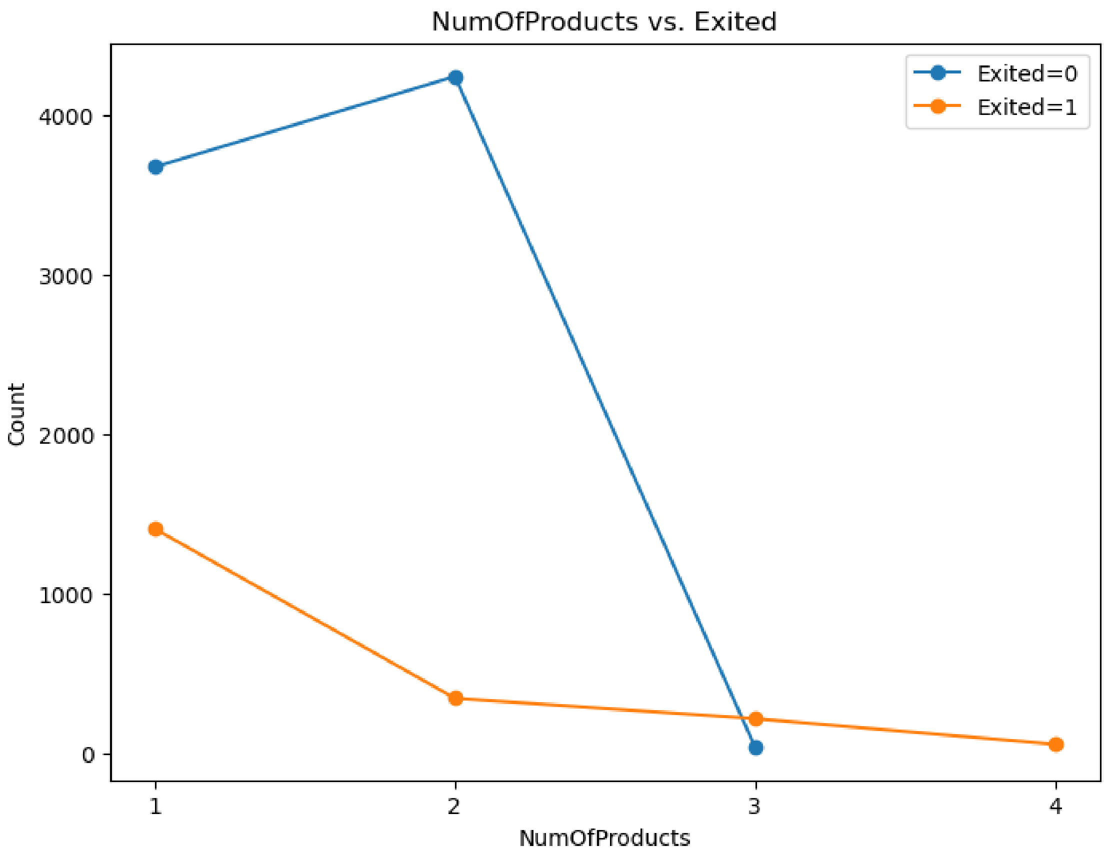


[27]:
```
age_map = sns.FacetGrid(df,col="Exited")
age_map.map(plt.hist,'Age',bins=20)
plt.show()
```

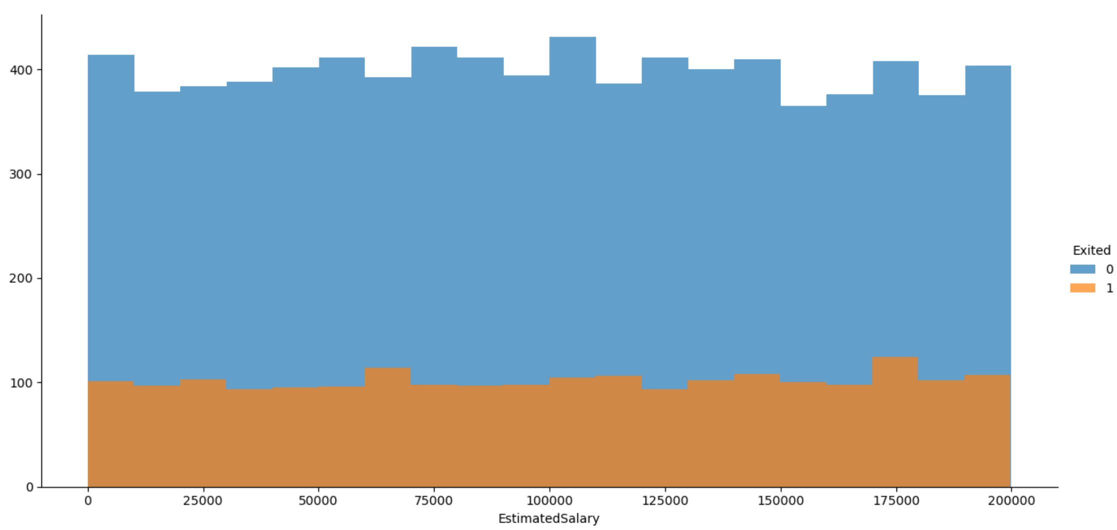


[28]:

```
# Filter the data separately according to Exited=0 and Exited=1
exited_0 = df[df['Exited'] == 0]
exited_1 = df[df['Exited'] == 1]

# Calculate the number of Exited=0 and Exited=1 for each number of cards held
counts_0 = exited_0['NumOfProducts'].value_counts().sort_index()
counts_1 = exited_1['NumOfProducts'].value_counts().sort_index()
plt.figure(figsize=(8, 6))

# Draw a line graph with Exited=0
plt.plot(counts_0.index, counts_0.values, marker='o', label='Exited=0')

# Draw a line graph with Exited=1
plt.plot(counts_1.index, counts_1.values, marker='o', label='Exited=1')

plt.title('NumOfProducts vs. Exited')
plt.xlabel('NumOfProducts')
plt.ylabel('Count')
plt.xticks([1, 2, 3, 4])
plt.legend()
plt.show()
```



[31]:
```
age_map = sns.FacetGrid(df, hue='Exited', height=6, aspect=2)
age_map.map(plt.hist, 'EstimatedSalary', bins=20, alpha=0.7)
age_map.add_legend()
plt.show()
```



[32]:
```python
from sklearn.preprocessing import LabelEncoder

def bin_and_encode(df, column_name, bins):
    df[column_name+'_bin'] = pd.cut(df[column_name], bins=bins)
    label = LabelEncoder()
    df[column_name+'_code'] = label.fit_transform(df[column_name+'_bin'])

mapping_dict = {
    'Geography': {'France': 'F', 'Germany': 'G', 'Spain': 'S'},
    'Gender': {'Female': 0, 'Male': 1}
}
df.replace(mapping_dict, inplace=True)

# Bins are divided according to the data distribution at the beginning
creditscore_bins = [-1, 400, 600, 800, 1000]
age_bins = [17, 30, 40, 50, 100]
tenure_bins = [-1, 2, 5, 8, 10]
balance_bins = [-1, 50000, 100000, 150000, 200000, 300000]
estimatedsalary_bins = [10, 51000, 110000, 150000, 200000]

# The columns that need to be boxed and coded are processed
columns_to_bin_and_encode = ['CreditScore', 'Age', 'Tenure', 'Balance',␣
↪'EstimatedSalary']

for column in columns_to_bin_and_encode:
    bin_and_encode(df, column, eval(column.lower() + '_bins'))

# Code country column
label = LabelEncoder()
df['country_code'] = label.fit_transform(df['Geography'])

df.head()
```

[32]:
```
   RowNumber  CustomerId   Surname  CreditScore Geography  Gender  Age  \
0          1    15634602  Hargrave          619         F       0   42
1          2    15647311      Hill          608         S       0   41
2          3    15619304      Onio          502         F       0   42
3          4    15701354      Boni          699         F       0   39
4          5    15737888  Mitchell          850         S       0   43

   Tenure    Balance  NumOfProducts  …  CreditScore_code   Age_bin  Age_code  \
0       2       0.00              1  …                 2  (40, 50]         2
1       1   83807.86              1  …                 2  (40, 50]         2
2       8  159660.80              3  …                 1  (40, 50]         2
```

```
3       1      0.00              2  …                2  (30, 40]          1
4       2 125510.82              1  …                3  (40, 50]          2

   Tenure_bin Tenure_code       Balance_bin  Balance_code EstimatedSalary_bin  \
0     (-1, 2]           0       (-1, 50000]             0     (51000, 110000]
1     (-1, 2]           0   (50000, 100000]             1    (110000, 150000]
2      (5, 8]           2  (150000, 200000]             3    (110000, 150000]
3     (-1, 2]           0       (-1, 50000]             0     (51000, 110000]
4     (-1, 2]           0  (100000, 150000]             2     (51000, 110000]

   EstimatedSalary_code country_code
0                     1            0
1                     2            2
2                     2            0
3                     1            0
4                     1            2

[5 rows x 26 columns]
```

[33]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Data columns (total 26 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   RowNumber             10000 non-null  int64
 1   CustomerId            10000 non-null  int64
 2   Surname               10000 non-null  object
 3   CreditScore           10000 non-null  int64
 4   Geography             10000 non-null  object
 5   Gender                10000 non-null  int64
 6   Age                   10000 non-null  int64
 7   Tenure                10000 non-null  int64
 8   Balance               10000 non-null  float64
 9   NumOfProducts         10000 non-null  int64
 10  HasCrCard             10000 non-null  int64
 11  IsActiveMember        10000 non-null  int64
 12  EstimatedSalary       10000 non-null  float64
 13  Exited                10000 non-null  int64
 14  Age_Group             9975 non-null   category
 15  CreditScore_bin       10000 non-null  category
 16  CreditScore_code      10000 non-null  int64
 17  Age_bin               10000 non-null  category
 18  Age_code              10000 non-null  int64
 19  Tenure_bin            10000 non-null  category
 20  Tenure_code           10000 non-null  int64
 21  Balance_bin           10000 non-null  category
```

```
 22  Balance_code           10000 non-null  int64
 23  EstimatedSalary_bin    10000 non-null  category
 24  EstimatedSalary_code   10000 non-null  int64
 25  country_code           10000 non-null  int64
dtypes: category(6), float64(2), int64(16), object(2)
memory usage: 1.6+ MB
```

[ ]:

[34]:
```
#Decision tree
```

[35]:
```
from sklearn.model_selection import cross_val_score, GridSearchCV,␣
 ↪train_test_split
from sklearn.tree import DecisionTreeClassifier
from sklearn.metrics import classification_report

#Prepare the characteristics and target variables
X = df[['CreditScore_code', 'country_code', 'Gender', 'Age_code', 'Tenure_code',
         'Balance_code', 'NumOfProducts', 'HasCrCard', 'IsActiveMember',␣
 ↪'EstimatedSalary_code']]
y = df['Exited']
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,␣
 ↪random_state=42)
clf = DecisionTreeClassifier()

# Define parameter grid
param_grid = {
    'max_depth': [3, 5, 7, 8, 9],
    'min_samples_split': [2, 5, 6, 7],
    'min_samples_leaf': [4, 5, 6, 7]
}

# Using grid search for parameter tuning
grid_search = GridSearchCV(estimator=clf, param_grid=param_grid, cv=5)
grid_search.fit(X_train, y_train)
print("Best parameters found: ", grid_search.best_params_)

# Forecasts are made using the model with the best parameters
best_clf = grid_search.best_estimator_
y_pred = best_clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
Best parameters found:  {'max_depth': 8, 'min_samples_leaf': 7,
'min_samples_split': 5}
              precision    recall  f1-score   support

           0       0.88      0.95      0.92      2416
           1       0.72      0.48      0.58       584
```

```
    accuracy                           0.86      3000
   macro avg       0.80      0.72      0.75      3000
weighted avg       0.85      0.86      0.85      3000
```

[36]:

```python
'''
The model's performance on the test set is as follows:

Precision: 88% accuracy for non-churn customers (label 0) and 72% accuracy for churn customers (label 1).
Recall: The recall rate is 95% for non-churn customers and 48% for churn customers.
F1-score: The F1 score for non-churning customers is 0.92 and the F1 score for churning customers is 0.58.
Accuracy: The overall accuracy of the model is 86%.
Summary:
The model performs well for predicting non-churn customers (label 0) with high precision and recall.
The poor prediction performance for churn customers (label 1), especially the low recall rate, indicates that the model misses more actual churn customers.
In general, the macro average F1 score of the model is 0.75, and the weighted average F1 score is 0.85, indicating that there is room for improvement in the model when dealing with unbalanced data sets.

Decision tree

Accuracy: 86 percent
Accuracy: 88% (non-churn), 72% (churn)
Recall: 95% (non-churn), 48% (churn)
F1 score: 92% (non-churn), 58% (churn)
Weighted average F1 score: 85%'''
```

[36]:

```
"\nThe model's performance on the test set is as follows:\n\nPrecision: 88%
accuracy for non-churn customers (label 0) and 72% accuracy for churn customers
(label 1).\nRecall: The recall rate is 95% for non-churn customers and 48% for
churn customers.\nF1-score: The F1 score for non-churning customers is 0.92 and
the F1 score for churning customers is 0.58.\nAccuracy: The overall accuracy of
the model is 86%.\nSummary:\nThe model performs well for predicting non-churn
customers (label 0) with high precision and recall.\nThe poor prediction
performance for churn customers (label 1), especially the low recall rate,
indicates that the model misses more actual churn customers.\nIn general, the
macro average F1 score of the model is 0.75, and the weighted average F1 score
is 0.85, indicating that there is room for improvement in the model when dealing
with unbalanced data sets.\n\nDecision tree\n\nAccuracy: 86 percent\nAccuracy:
88% (non-churn), 72% (churn)\nRecall: 95% (non-churn), 48% (churn)\nF1 score:
92% (non-churn), 58% (churn)\nWeighted average F1 score: 85%"
```

[ ]:

[37]:
```
#logistic regression
```

[ ]:

[38]:
```
from sklearn.model_selection import GridSearchCV
from sklearn.linear_model import LogisticRegression


logistic_model = LogisticRegression(solver='lbfgs')

# Define parameter grid
param_grid = {'C': [0.001, 0.01, 0.1, 1, 10, 100]}

grid_search = GridSearchCV(estimator=logistic_model, param_grid=param_grid,␣
↪cv=5, scoring='accuracy')

# Cross-validation was used to search for the best parameters
grid_search.fit(X_train, y_train)
print("Best parameter:", grid_search.best_params_)
print("Best accurancy:", grid_search.best_score_)
```

```
Best parameter: {'C': 10}
Best accurancy: 0.8197142857142856
```

[39]:
```
# best parameters
best_params = grid_search.best_params_

# The logistic regression model was created and the best parameters were used
best_logistic_model = LogisticRegression(solver='lbfgs', **best_params)

# The best parameters are used to train the model
best_logistic_model.fit(X_train, y_train)

# Predictions are made on the test set
y_pred = best_logistic_model.predict(X_test)

# count accurancy
accuracy = (y_pred == y_test).mean()
print("Logistic regression model accuracy (using the best parameters):",␣
↪accuracy)

# Calculation classification report
from sklearn.metrics import classification_report
report = classification_report(y_test, y_pred)
print("classification_report:\n", report)
```

```
Logistic regression model accuracy (using the best parameters):
0.8266666666666667
classification_report:
              precision    recall  f1-score   support

           0       0.84      0.97      0.90      2416
           1       0.64      0.25      0.36       584

    accuracy                           0.83      3000
   macro avg       0.74      0.61      0.63      3000
weighted avg       0.80      0.83      0.79      3000
```

[40]:
```python
#Output the parameters (coefficients) of each variable
coefficients = best_logistic_model.coef_[0]
intercept = best_logistic_model.intercept_[0]

print("Logistic regression model parameters:")
for feature, coefficient in zip(X.columns, coefficients):
    print(f"{feature}: {coefficient:.6f}")

print("intercept:", intercept)
```

```
Logistic regression model parameters:
CreditScore_code: -0.122279
country_code: 0.102463
Gender: -0.551107
Age_code: 0.957877
Tenure_code: -0.041858
Balance_code: 0.277328
NumOfProducts: -0.031327
HasCrCard: -0.044504
IsActiveMember: -1.019306
EstimatedSalary_code: 0.020071
intercept: -2.148801012084326
```

[42]:
```python
from sklearn.model_selection import cross_val_predict, cross_val_score
from sklearn.metrics import precision_recall_fscore_support

# Predictions are made using cross validation
y_pred_cv = cross_val_predict(best_logistic_model, X, y, cv=5)

# Precision, recall, F1 score, and support were calculated
precision, recall, f1_score, support = precision_recall_fscore_support(y,␣
 ↪y_pred_cv)
print("precision:", precision)
print("recall:", recall)
print("f1_score:", f1_score)
```

```
print("support:", support)

# Cross validation
scores = cross_val_score(best_logistic_model, X, y, cv=5)
print("Cross validation accuracy:", scores)
print("Average accuracy rate:", scores.mean())
```

```
precision: [0.83452304 0.67241379]
recall: [0.96898154 0.24889543]
f1_score: [0.89674008 0.36331064]
support: [7963 2037]
Cross validation accuracy: [0.8145 0.8275 0.8215 0.836  0.812 ]
Average accuracy rate: 0.8222999999999999
```

[43]:

```
'''
When the logistic regression model is applied to predict bank customer churn,
the model shows some bias in distinguishing customer churn.
Although the model has a high prediction accuracy of 84% and a recall rate of␣
 ↪97% for non-churn customers (label 0),
it has a prediction accuracy of only 64% and a low recall rate of only 25% for␣
 ↪churn customers (label 1).
This indicates that the performance of the model in identifying potential churn␣
 ↪customers needs to be improved.
Despite the overall accuracy of 83%, the model's low recall of churn customers
may result in banks failing to accurately identify customers at risk of churn
and thus missing intervention opportunities.
Therefore, further adjustments to the model are recommended, especially to␣
 ↪increase the ability to identify
churn customers, in order to predict the two scenarios more balanced.
In addition, the cross-validation results with an average accuracy of 82.23%␣
 ↪also show that
the model has a relatively stable generalization ability.
The ultimate goal is to improve the overall forecasting performance of the␣
 ↪model,
especially the recall of churn customers.
'''
```

[43]:

```
"\nWhen the logistic regression model is applied to predict bank customer churn,
\nthe model shows some bias in distinguishing customer churn. \nAlthough the
model has a high prediction accuracy of 84% and a recall rate of 97% for non-
churn customers (label 0),\nit has a prediction accuracy of only 64% and a low
recall rate of only 25% for churn customers (label 1). \nThis indicates that the
performance of the model in identifying potential churn customers needs to be
improved. \nDespite the overall accuracy of 83%, the model's low recall of churn
customers \nmay result in banks failing to accurately identify customers at risk
of churn \nand thus missing intervention opportunities. \nTherefore, further
adjustments to the model are recommended, especially to increase the ability to
```

identify \nchurn customers, in order to predict the two scenarios more balanced. \nIn addition, the cross-validation results with an average accuracy of 82.23% also show that \nthe model has a relatively stable generalization ability. \nThe ultimate goal is to improve the overall forecasting performance of the model, \nespecially the recall of churn customers.\n"

[ ]:
```
'''
Logistic regression

Accuracy: 83 percent
Accuracy: 84% (non-churn), 64% (churn)
Recall: 97% (non-churn), 25% (churn)
F1 score: 90% (non-churn), 36% (churn)
Weighted average F1 score: 79%
'''
```

[ ]:

[ ]:

[ ]:

[ ]:

[ ]:
```
#random forest
```

[36]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import GridSearchCV, train_test_split
# Prepare the characteristics and target variables
X = df[['CreditScore_code', 'country_code', 'Gender', 'Age_code', 'Tenure_code',
        'Balance_code', 'NumOfProducts', 'HasCrCard', 'IsActiveMember',␣
 ↪'EstimatedSalary_code']]
y = df['Exited']

# Divide the training set and test set
X_train, X_test, y_train, y_test = train_test_split(X,  y, test_size=0.3,␣
 ↪random_state=11)

# Create a random forest classifier object
rf_classifier = RandomForestClassifier(random_state=11)

# Define parameters
param_grid = {
    'n_estimators': [200,300,400],
    'max_depth': [15,20,25],
    'min_samples_split': [1,2,3],
    'min_samples_leaf': [4,6,8]
```

```
}

grid_search = GridSearchCV(rf_classifier, param_grid, cv=5, n_jobs=-1)

# The model was fitted and cross-validated using the training dataset
grid_search.fit(X_train, y_train)

# Best parameters
print("Best parameters:", grid_search.best_params_)
```

```
Best parameters: {'max_depth': 15, 'min_samples_leaf': 6, 'min_samples_split':
2, 'n_estimators': 200}
```

[37]:
```
from sklearn.metrics import accuracy_score
# Predictions are made using the test data set
y_pred = grid_search.predict(X_test)

accuracy = accuracy_score(y_test, y_pred)
print("Accuracy:", accuracy)
print(classification_report(y_true=y_test, y_pred=y_pred))
```

```
Accuracy: 0.8566666666666667
              precision    recall  f1-score   support

           0       0.87      0.96      0.91      2375
           1       0.76      0.45      0.57       625

    accuracy                           0.86      3000
   macro avg       0.82      0.71      0.74      3000
weighted avg       0.85      0.86      0.84      3000
```

[ ]:
```
'''
Random forest

Accuracy: 86.5%
Accuracy: 88% (non-churn), 77% (churn)
Recall rate: 97% (non-churn), 43% (churn)
F1 score: 92% (non-churn), 55% (churn)
Weighted average F1 score: 85%
'''
```

[43]:
```

```

[ ]:
```
'''Model comparison and analysis
Accuracy: Random forest and decision tree have the same and slightly higher␣
↪accuracy than logistic regression.
```

```
This indicates that for the classification of the overall data set, these two␣
↪models are slightly better.

Precision and recall: The decision tree has the highest recall (95%) in␣
↪identifying non-churn customers,
indicating that it is best at identifying true non-churn customers.
However, when it comes to identifying churn customers, random Forest is more␣
↪accurate (77%),
indicating that it is more reliable in predicting churn.

F1 score: Considering the balance of precision and recall, both random forest
and decision trees have high F1 scores for non-churn customers,
but random forest has a higher F1 score for churn customers, although its␣
↪recall is slightly lower.

Weighted average F1 score: Logistic regression has the lowest weighted average␣
↪F1 score among the three models,
indicating its poor performance when dealing with imbalanced datasets.

Conclusion
Random forest shows an advantage in overall performance, especially in handling␣
↪the prediction of churn customers
with more precision, although its recall rate needs to be improved.
Decision trees perform well in predicting non-churn customers, which may be a␣
↪good choice
if the goal is to avoid any non-churn customers as much as possible.
Logistic regression performs poorly in all models, especially in the recall of␣
↪churn customers,
which may result in a lot of actual churn customers not being correctly␣
↪predicted.

It is recommended to select the model according to the actual business needs:
if reducing the underreporting of churn customer forecasts is the primary goal,
random forest may be the best choice; If ensuring that hardly any non-churn␣
↪customers are missed is key,
then a decision tree may be a better fit.
At the same time, further adjusting the parameters of the random forest model␣
↪or trying other ensemble
learning methods can be considered to improve the recall rate.
''''''
```

[ ]:

```
```

[ ]:

```
```

[ ]:

```
```

[ ]:

```

```

[ ]:

```

```

[ ]:

```

```

[46]:

```
pip install -U imbalanced-learn
```

```
Requirement already satisfied: imbalanced-learn in
/opt/anaconda3/lib/python3.8/site-packages (0.12.2)
Requirement already satisfied: numpy>=1.17.3 in
/opt/anaconda3/lib/python3.8/site-packages (from imbalanced-learn) (1.23.5)
Requirement already satisfied: scipy>=1.5.0 in
/opt/anaconda3/lib/python3.8/site-packages (from imbalanced-learn) (1.10.1)
Requirement already satisfied: scikit-learn>=1.0.2 in
/opt/anaconda3/lib/python3.8/site-packages (from imbalanced-learn) (1.3.2)
Requirement already satisfied: joblib>=1.1.1 in
/opt/anaconda3/lib/python3.8/site-packages (from imbalanced-learn) (1.2.0)
Requirement already satisfied: threadpoolctl>=2.0.0 in
/opt/anaconda3/lib/python3.8/site-packages (from imbalanced-learn) (3.2.0)
WARNING: There was an error checking the latest version of pip.

Note: you may need to restart the kernel to use updated packages.
```

[42]:

```python
# USe SMOTE for oversampling
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=11)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# The oversampled data were used to train the random forest model
rf_classifier.fit(X_train_smote, y_train_smote)

# prediction
y_pred_smote = rf_classifier.predict(X_test)

# report
print("Classification report after SMOTE:\n", classification_report(y_test,
 ↪y_pred_smote))
```

```
Classification report after SMOTE:
               precision    recall  f1-score   support

           0       0.89      0.85      0.87      2375
           1       0.51      0.61      0.56       625

    accuracy                           0.80      3000
   macro avg       0.70      0.73      0.71      3000
weighted avg       0.81      0.80      0.80      3000
```

[43]:
```python
#Adjusting model parameters
from imblearn.over_sampling import SMOTE

# Increase the sampling intensity of SMOTE and increase the proportion of␣
↪minority class synthesis
smote = SMOTE(sampling_strategy=0.8, random_state=11)
X_train_smote, y_train_smote = smote.fit_resample(X_train, y_train)

# Use the adjusted model parameters

param_grid_adjusted = {
    'n_estimators': [200, 300, 400],
    'max_depth': [5, 10, 15],
    'min_samples_split': [5, 10, 15],
    'min_samples_leaf': [5, 10, 15]
}

grid_search = GridSearchCV(RandomForestClassifier(random_state=42),␣
↪param_grid_adjusted, cv=5, n_jobs=-1)
grid_search.fit(X_train_smote, y_train_smote)

print("Adjusted Best Parameters:", grid_search.best_params_)
y_pred_adjusted = grid_search.predict(X_test)
print("Adjusted Classification Report:\n", classification_report(y_test,␣
↪y_pred_adjusted))
```

```
Adjusted Best Parameters: {'max_depth': 15, 'min_samples_leaf': 5,
'min_samples_split': 5, 'n_estimators': 400}
Adjusted Classification Report:
               precision    recall  f1-score   support

           0       0.90      0.86      0.88      2375
           1       0.55      0.65      0.60       625

    accuracy                           0.82      3000
   macro avg       0.73      0.76      0.74      3000
weighted avg       0.83      0.82      0.82      3000
```

[44]:
```python
#Customize the threshold
# Get the model prediction probability
y_probs = grid_search.predict_proba(X_test)[:, 1]

# Customize the threshold
threshold = 0.35

# Classification is made according to the threshold value
```

```
y_pred_custom = [1 if prob > threshold else 0 for prob in y_probs]
print("Classification Report with Custom Threshold:")
print(classification_report(y_test, y_pred_custom))
```

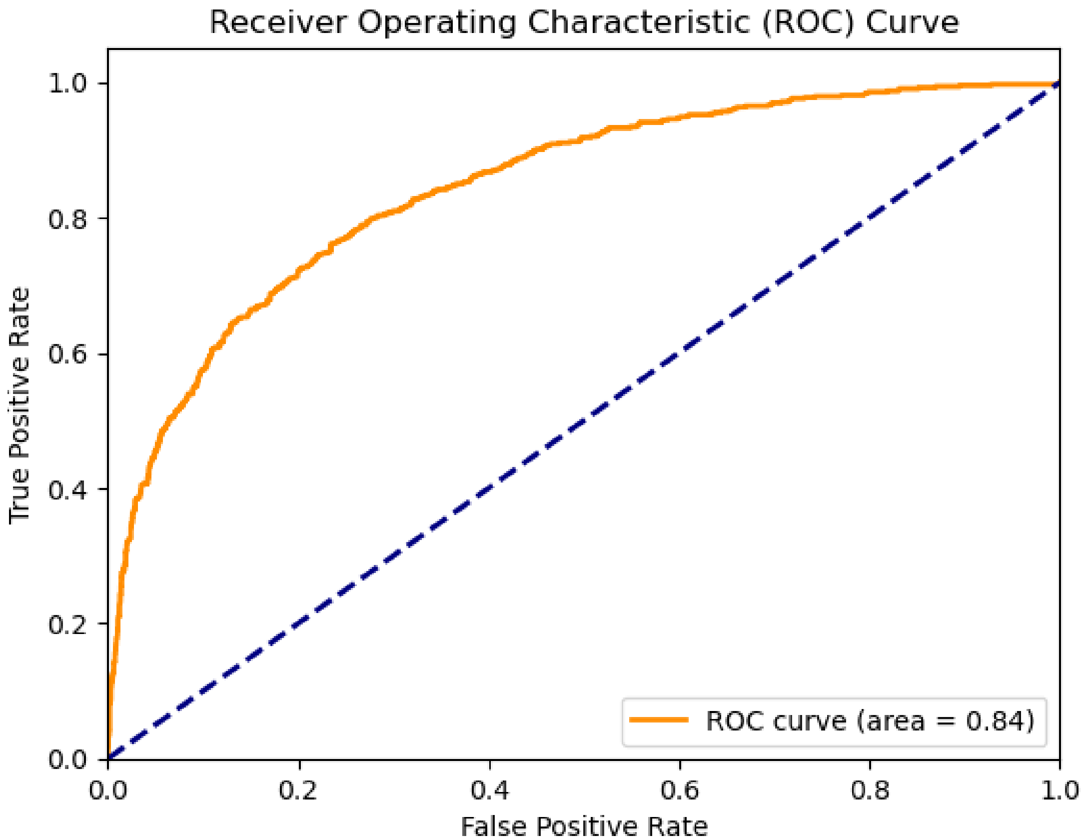
```
Classification Report with Custom Threshold:
              precision    recall  f1-score   support

           0       0.93      0.73      0.82      2375
           1       0.43      0.79      0.56       625

    accuracy                           0.74      3000
   macro avg       0.68      0.76      0.69      3000
weighted avg       0.83      0.74      0.76      3000
```

[ ]:

```
#roc
```

[46]:

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# fitting model
best_model = grid_search.best_estimator_
best_model.fit(X_train_smote, y_train_smote)

# Possibility Prediction
y_probs = best_model.predict_proba(X_test)[:, 1]

# Calculate ROC curve data
fpr, tpr, thresholds = roc_curve(y_test, y_probs)
roc_auc = auc(fpr, tpr)
plt.figure()
plt.plot(fpr, tpr, color='darkorange', lw=2, label='ROC curve (area = %0.2f)' %
 ↪roc_auc)
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver Operating Characteristic (ROC) Curve')
plt.legend(loc="lower right")
plt.show()
```



[ ]:
```
'''
The area under the ROC curve (AUC) was 84%.
This value shows that our model performs well in predicting the loss of credit␣
 ↪card users.
In this case, our ability to distinguish churn users (positive example) from␣
 ↪non-churn users (negative example)
is quite robust, and the model does a good job of distinguishing churn users at␣
 ↪various possible thresholds.

Specifically, an AUC of 84% means that the model correctly identifies churn␣
 ↪users 84% of the time
when one churn user and one non-churn user are randomly selected.
This suggests that our model can effectively distinguish between those users␣
 ↪who are likely to churn
and those who are unlikely to churn, providing banks with important predictive␣
 ↪power.
In practice, this means that we can use the predictions of the model to␣
 ↪identify potential
```

```
churn customers and take appropriate measures, such as providing personalized␣
↪services or offers,
to reduce the likelihood of churn.'''
```

```
[ ]:
```

```
[ ]:
```

```
[ ]:
```