churn-prediction

April 9, 2025

1 Importing Necessary Libraries

[3]:
```
import pandas as pd #Data Manipulation
import numpy as np #Scientific Computing
import seaborn as sns #Data Visualization
import matplotlib.pyplot as plt #Data Visualization
import plotly.express as px #Data Visualization
import plotly.graph_objects as go #Data Visualization
```

Added the intially required Libraries, but in case more rquired then it'll be added later.

2 Load the Dataset

[4]:
```
df = pd.read_csv("Churn_ Data.csv")
```

3 Understanding the Dataset

[5]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 25000 entries, 0 to 24999
Columns: 111 entries, s6.new.rev.p2.m2 to s3.rev.p1
dtypes: float64(80), int64(31)
memory usage: 21.2 MB
```

[6]:
```
df.head()
```

[6]:
```
   s6.new.rev.p2.m2  s1.new.rev.m1  s3.og.rev.4db.p5  s3.new.rev.4db.p5  \
0             -0.76        88.0482          3.106604           3.754955
1             -0.98        67.5039          3.094574           5.550865
2             -0.98        33.9248          2.324016           2.438114
3             -0.92        82.6780          2.630749           2.858961
4             -0.97        96.8379          2.674316           2.912397

   s4.usg.ins.p2  s4.og.unq.any.p2  s2.rch.val.p6  s1.og.rev.all.m1  \
0              4                14          39.29            57.320
```

```
1              1                 2          21.67            38.700
2              2                 3          30.00            15.320
3              2                 3          50.00            51.956
4              3                 2          22.50            66.886

   s8.new.rev.p6  s4.loc.ic.ins.p1  …  prop.og.mou.tot.mou.all.p6  \
0          -0.17                 1  …                    0.454642
1          -0.32                 3  …                    0.343190
2          -0.05                 3  …                    0.101838
3          -0.18                 4  …                    0.066602
4           0.01                 4  …                    0.219821

   prop.i2i.og.mou.p6  s4.loc.ic.ins.p2  s4.std.ic.ins.l14  \
0            0.497397                 4                  0
1            0.767617                 6                  0
2            0.619034                 6                  1
3            0.437088                 7                  2
4            0.585977                 6                  1

   s4.low.blnc.ins.p4  s3.og.rev.all.m2  s3.new.rev.m2  prop.og.mou.any.p6  \
0                   9              6.02           8.20           46.465636
1                  20              3.66           8.10           34.525456
2                  19              4.33           4.36           10.298451
3                  11              3.40           3.53            6.670783
4                  14              3.85           3.87           21.998905

   prop.loc.i2i.mou.og.mou.p3  s3.rev.p1
0                    0.609456       0.22
1                    1.000000       0.38
2                    0.699592       0.11
3                    0.086617       5.18
4                    0.683105       0.10

[5 rows x 111 columns]
```

[7]:
```
df.tail()
```

[7]:
```
       s6.new.rev.p2.m2  s1.new.rev.m1  s3.og.rev.4db.p5  s3.new.rev.4db.p5  \
24995              0.21       132.0365          2.652236           2.857739
24996              0.80        77.0154          3.763389           5.012503
24997              0.01       148.8337          3.823940           4.334250
24998              0.17      1012.4398         14.667580          14.579567
24999             -1.00       275.3530          5.134579           5.954062

       s4.usg.ins.p2  s4.og.unq.any.p2  s2.rch.val.p6  s1.og.rev.all.m1  \
24995              5                 8          26.67           123.396
24996              2                 8          27.88            62.140
```

```
24997                6            10       10.00          98.900
24998                7            67       42.92         734.005
24999                1             1       53.50         250.340

       s8.new.rev.p6  s4.loc.ic.ins.p1  …  prop.og.mou.tot.mou.all.p6  \
24995          -0.16                 4  …                    0.145831
24996           0.19                 4  …                    0.529829
24997          -0.03                 2  …                    0.327245
24998           0.70                 4  …                    0.824671
24999          -0.48                 4  …                    0.377281

       prop.i2i.og.mou.p6  s4.loc.ic.ins.p2  s4.std.ic.ins.l14  \
24995            0.200151                 7                  0
24996            0.169835                 7                  0
24997            0.407944                 3                  0
24998            0.889239                 7                  1
24999            0.609046                 7                  0

       s4.low.blnc.ins.p4  s3.og.rev.all.m2  s3.new.rev.m2  \
24995                  18              3.57           3.83
24996                  18              6.89           7.70
24997                  12              6.63           7.48
24998                   1             19.36          22.26
24999                  18              5.42           8.02

       prop.og.mou.any.p6  prop.loc.i2i.mou.og.mou.p3  s3.rev.p1
24995           14.896154                    0.328027       0.76
24996           55.156230                    0.288006      12.74
24997           33.222018                    0.235918       8.07
24998           82.549378                    0.952962      21.21
24999           38.590040                    1.000000       0.00

[5 rows x 111 columns]
```

[8]:
```python
print(df.shape)
```

```
(25000, 111)
```

[9]:
```python
df.describe().T
```

[9]:
```
                             count        mean         std       min  \
s6.new.rev.p2.m2           25000.0   -0.003730    2.727916 -1.000000
s1.new.rev.m1              25000.0  281.073083  276.075983  0.000000
s3.og.rev.4db.p5           25000.0    4.890003    4.212452  0.000000
s3.new.rev.4db.p5          25000.0    7.070194    6.318992  0.000833
s4.usg.ins.p2              25000.0    5.460080    2.184444  0.000000
…                              …           …           …         …
```

```
s3.og.rev.all.m2           25000.0    8.008660    6.152429  0.000000
s3.new.rev.m2              25000.0   12.540182   11.540611  0.000000
prop.og.mou.any.p6         25000.0   53.594165   21.408486  0.000000
prop.loc.i2i.mou.og.mou.p3 25000.0    0.483975    0.292349  0.000000
s3.rev.p1                  25000.0    9.951366   17.648128  0.000000

                                  25%         50%         75%          max
s6.new.rev.p2.m2            -0.580000   -0.170000    0.280000   316.860000
s1.new.rev.m1              101.563800  204.859600  370.711650  5702.924300
s3.og.rev.4db.p5             2.367288    3.729944    5.993342   153.221695
s3.new.rev.4db.p5            3.318825    5.231268    8.395736   170.200441
s4.usg.ins.p2                5.000000    7.000000    7.000000     7.000000
…                                 …           …          …           …
s3.og.rev.all.m2             4.207500    6.345000    9.830000   171.780000
s3.new.rev.m2                6.167500    9.350000   14.620000   386.480000
prop.og.mou.any.p6          39.378142   53.976203   68.312416   100.000000
prop.loc.i2i.mou.og.mou.p3   0.251304    0.477621    0.716538     1.000000
s3.rev.p1                    1.970000    5.380000   11.400000   585.500000

[111 rows x 8 columns]
```

4 Processing of the Data

4.0.1 Check for Misclassified Data Types

[10]:
```python
misclassified_columns = []
for col in df.columns:
    if df[col].dtype == 'object':
        try:
            df[col] = pd.to_numeric(df[col])
        except ValueError:
            misclassified_columns.append(col)
print("Misclassified columns:", misclassified_columns)
```

```
Misclassified columns: []
```

4.0.2 Removing NULL, Duplicates & Round the values upto 2 decimals

[11]:
```python
def clean_data(df):
    # Drop duplicate rows across all columns
    df = df.drop_duplicates()
    # Drop rows with missing data across all columns
    df = df.dropna()
    # Round columns 's6.new.rev.p2.m2', 's1.new.rev.m1' and 109 other columns␣
 ↪(Number of decimals: 2)
```

```
    df = df.round({'s6.new.rev.p2.m2': 2, 's1.new.rev.m1': 2, 's3.og.rev.4db.
↪p5': 2, 's3.new.rev.4db.p5': 2, 's4.usg.ins.p2': 2, 's4.og.unq.any.p2': 2,␣
↪'s2.rch.val.p6': 2,
                    's1.og.rev.all.m1': 2, 's8.new.rev.p6': 2, 's4.loc.ic.ins.
↪p1': 2, 's8.mbl.p2': 2, 's2.rch.val.l67': 2, 's7.s4.day.no.mou.p2.p4': 2,␣
↪'s3.new.rev.p3': 2,
                    's7.s5.s4.day.nomou.p4': 2, 's8.og.rev.p3': 2, 's8.ic.mou.
↪all.p3': 2, 'target': 2, 's7.new.rev.p2.p6': 2, 's6.rtd.mou.p2.m2': 2, 's7.
↪rtd.mou.p2.p6': 2,
                    's1.new.rev.p2': 2, 's1.new.rev.p1': 2, 's1.og.hom.mou.p1':␣
↪2, 's7.rev.p2.p6': 2, 's1.og.hom.rev.p2': 2, 's1.rtd.mou.p1': 2, 's1.og.rev.
↪all.p1': 2,
                    's1.og.mou.all.p1': 2, 's3.og.rev.all.p1': 2, 's7.new.rev.p3.
↪p6': 2, 'ds.usg.p6': 2, 'snd.dec.p2': 2, 's3.og.mou.all.p1': 2, 'ds.og.usg.
↪p4': 2,
                    's1.og.mou.all.p2': 2, 's8.og.rev.p6': 2, 's1.og.hom.mou.p2':
↪ 2, 's5.og.rev.all.p1': 2, 's1.og.rev.all.p2': 2, 's1.rtd.mou.p2': 2, 's5.
↪rtd.mou.p1': 2,
                    's1.og.mou.any.p2': 2, 's4.day.no.mou.p2': 2, 's1.hom.rmg.
↪rev.p2': 2, 's7.rtd.mou.p3.p6': 2, 's5.og.mou.all.p1': 2, 's5.og.hom.mou.p1':
↪ 2, 's3.new.rev.p1': 2,
                    's4.usg.ins.p1': 2, 's2.s4.day.no.mou.p2': 2, 's7.new.rev.
↪l21.p6': 2, 's5.rev.p1': 2, 's5.s4.day.no.mou.p2': 2, 'tot.s4.day.no.mou.p2':
↪ 2, 's8.new.rev.p3': 2,
                    's3.og.mou.all.p2': 2, 's1.rev.p1': 2, 's4.loc.og.ins.p1':␣
↪2, 's1.loc.og.mou.p1': 2, 's4.og.any.p2': 2, 'prop.og.mou.any.p2': 2, 's4.
↪low.blnc.ins.p3': 2,
                    's1.loc.og.mou.p2': 2, 's5.new.rev.p2': 2, 's5.new.rev.p1':␣
↪2, 's4.low.blnc.ins.l14': 2, 's3.og.hom.mou.p1': 2, 's7.rtd.mou.l21.p6': 2,␣
↪'s4.loc.og.ins.l14': 2,
                    's8.rtd.mou.p3': 2, 's4.dec.ins.l14': 2, 's2.s4.day.no.mou.
↪p3': 2, 's3.new.rev.p2': 2, 'tot.s4.day.no.mou.p3': 2, 's5.og.mou.all.p2':␣
↪2, 's4.loc.ic.ins.l14': 2,
                    's4.usg.ins.l14': 2, 's4.loc.og.ins.p2': 2, 's3.rtd.mou.p1':␣
↪2, 's7.s5.s4.day.nomou.p2': 2, 's8.og.mou.all.p6': 2, 's5.og.hom.mou.p2': 2,␣
↪'s7.rtd.mou.m1.m2': 2,
                    'prop.og.mou.tot.mou.all.p2': 2, 's8.rev.p6': 2, 's7.s5.s4.
↪day.nomou.p3': 2, 's5.rev.p2': 2, 's1.new.rev.m2': 2, 's3.og.rev.3db.p5': 2,␣
↪'s4.rch.val.gt.30.p2': 2,
                    's8.rtd.mou.p6': 2, 's4.std.ins.l14': 2, 's4.low.blnc.ins.
↪p2': 2, 's4.low.blnc.ins.p6': 2, 's4.loc.ins.l14': 2, 's4.low.blnc.ins.m2':␣
↪2, 's4.data.ins.l14': 2,
                    'prop.loc.i2i.mou.og.mou.p6': 2, 's4.dec.ins.p2': 2, 's1.rev.
↪p2': 2, 'prop.og.mou.tot.mou.all.p6': 2, 'prop.i2i.og.mou.p6': 2, 's4.loc.ic.
↪ins.p2': 2,
```

```
                    's4.std.ic.ins.l14': 2, 's4.low.blnc.ins.p4': 2, 's3.og.rev.
↪all.m2': 2, 's3.new.rev.m2': 2, 'prop.og.mou.any.p6': 2, 'prop.loc.i2i.mou.
↪og.mou.p3': 2, 's3.rev.p1': 2})
    return df

df_clean = clean_data(df.copy())
df_clean.shape
```

[11]: (25000, 111)

[12]:
```
#reassigning df_clean data to df
df = df_clean
```

4.0.3 Check For Unique Values

[13]:
```
unique_value_columns = []
for col in df.columns:
    if df[col].nunique() == df.shape[0]:
        unique_value_columns.append(col)
data_clean = df.drop(columns=unique_value_columns)
print(unique_value_columns)
print(data_clean.shape)
```

```
[]
(25000, 111)
```

4.0.4 Check For Zero Variance Values

[14]:
```
zero_variance_columns = []
for col in df.columns:
    if df[col].std() == 0:
        zero_variance_columns.append(col)
data_clean = df.drop(columns=zero_variance_columns)
print(zero_variance_columns)
print(data_clean.shape)
```

```
[]
(25000, 111)
```

[15]:
```
#again reassigning df_clean data to df
df = df_clean
```

4.0.5 Recursive Feature Selection (RFE)

[16]:
```
import warnings
warnings.filterwarnings("ignore")
from sklearn.feature_selection import RFECV
from sklearn.linear_model import LogisticRegression

def remove_features_rfe(df):
    # Create a logistic regression estimator
    estimator = LogisticRegression()

    # Create an RFECV object with the estimator
    rfe = RFECV(estimator, step=1)

    # Fit the RFECV object to the data
    rfe.fit(df.drop('target', axis=1), df['target'])

    # Get the support mask (True for selected features, False for eliminated␣
↪features)
    support = rfe.support_

    # Create a new dataframe with the selected features
    df_filtered = df.drop('target', axis=1).loc[:, support]

    return df_filtered

# Remove features using RFE
df_filtered = remove_features_rfe(df.copy())

print("Shape of filtered dataframe:", df_filtered.shape)
```

```
Shape of filtered dataframe: (25000, 103)
```

[17]:
```
df0 = df_filtered
```

4.0.6 Outliers Treatment (IQR Method)

[18]:
```
# Function to treat outliers using IQR method
def treat_outliers(df0):
    for column in df0.select_dtypes(include=[np.number]).columns:
        Q1 = df0[column].quantile(0.25)
        Q3 = df0[column].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR
        df0[column] = np.where(df0[column] < lower_bound, lower_bound,␣
↪df0[column])
```

```
        df0[column] = np.where(df0[column] > upper_bound, upper_bound,␣
↪df0[column])
    return df0

# Treat outliers
df_treated = treat_outliers(df0)
df_treated.shape
```

[18]: (25000, 103)

[19]:
```
df0 = df_treated
```

4.0.7 Removing Highly Correlated Values

[20]:
```
def remove_correlated_features(df0, threshold=0.8):
  # Create correlation matrix
  corr_matrix = df0.corr().abs()

  # Exclude the diagonal (self-correlation)
  upper_tri = corr_matrix.where(~np.tril(np.ones(corr_matrix.shape)).
↪astype(bool))

  # Find columns with correlations above the threshold
  to_drop = [col for col in upper_tri.columns if any(upper_tri[col] >␣
↪threshold)]

  # Drop the correlated features
  return df0.drop(to_drop, axis=1)

# Remove features with correlation above 0.8
df_filtered = remove_correlated_features(df0.copy(), threshold=0.8)

df_filtered.shape
```

[20]: (25000, 33)

[21]:
```
df1 = df_filtered
```

4.0.8 MultiCollinerity (VIF > 5)

[22]:
```
from statsmodels.stats.outliers_influence import variance_inflation_factor
import warnings
warnings.filterwarnings("ignore")  # Suppress warnings (optional)

def calculate_vif(df1):
    vif_data = pd.DataFrame()
```

```
    vif_data["feature"] = df1.columns
    vif_data["VIF"] = [variance_inflation_factor(df1.values, i)
                       for i in range(len(df1.columns))]
    return vif_data

def remove_high_vif_features(df1, threshold=5):
    initial_features = set(df1.columns)
    while True:
        vif_data = calculate_vif(df1)
        print(vif_data.sort_values('VIF', ascending=False).head())
        if vif_data['VIF'].max() <= threshold:
            break
        feature_to_remove = vif_data.loc[vif_data['VIF'].idxmax(), 'feature']
        df1 = df1.drop(columns=[feature_to_remove])
        print(f"Removed feature with high VIF: {feature_to_remove}")
    removed_features = initial_features - set(df1.columns)
    print(f"Total features removed: {len(removed_features)}")
    print(f"Removed features: {removed_features}")
    return df1

df1.shape
```

[22]: (25000, 33)

[23]:
```
df1['target'] = df['target']
```

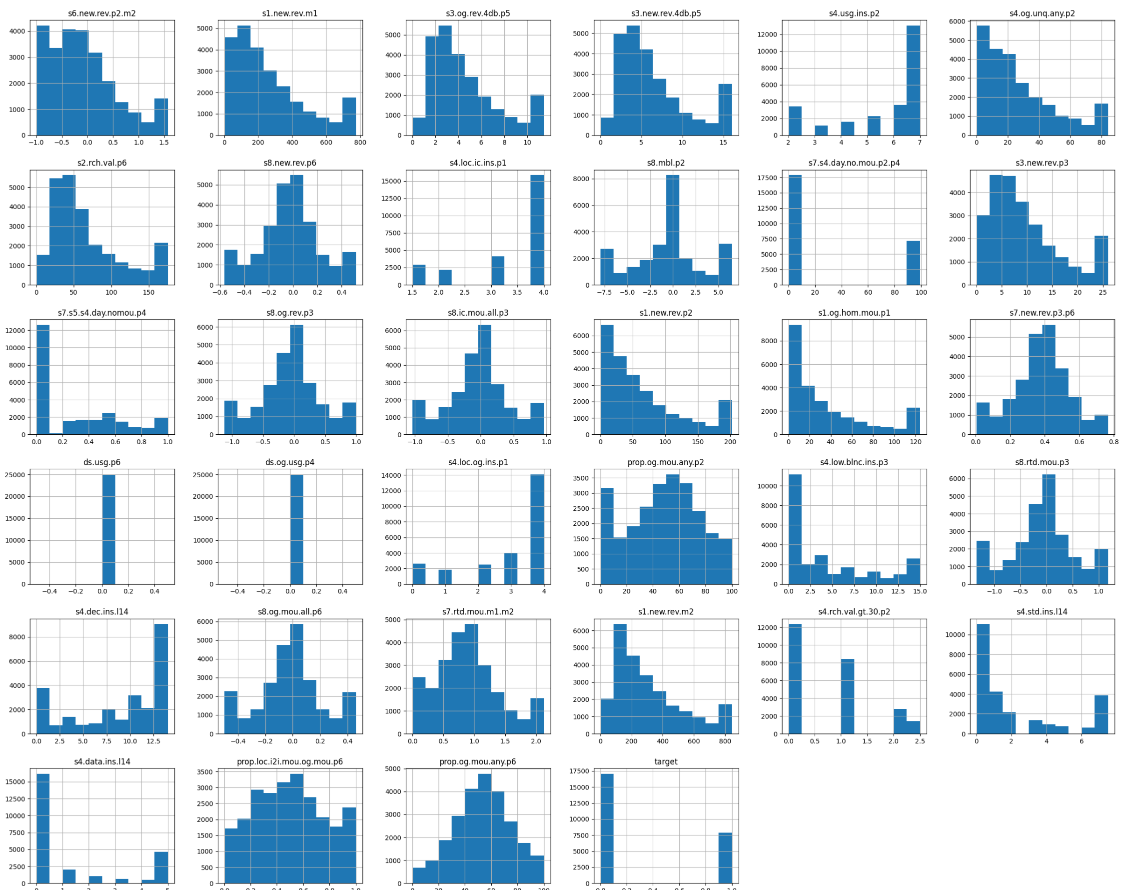
[24]:
```
df1.shape
```

[24]: (25000, 34)

5 Visualize the Distribution

Histogram

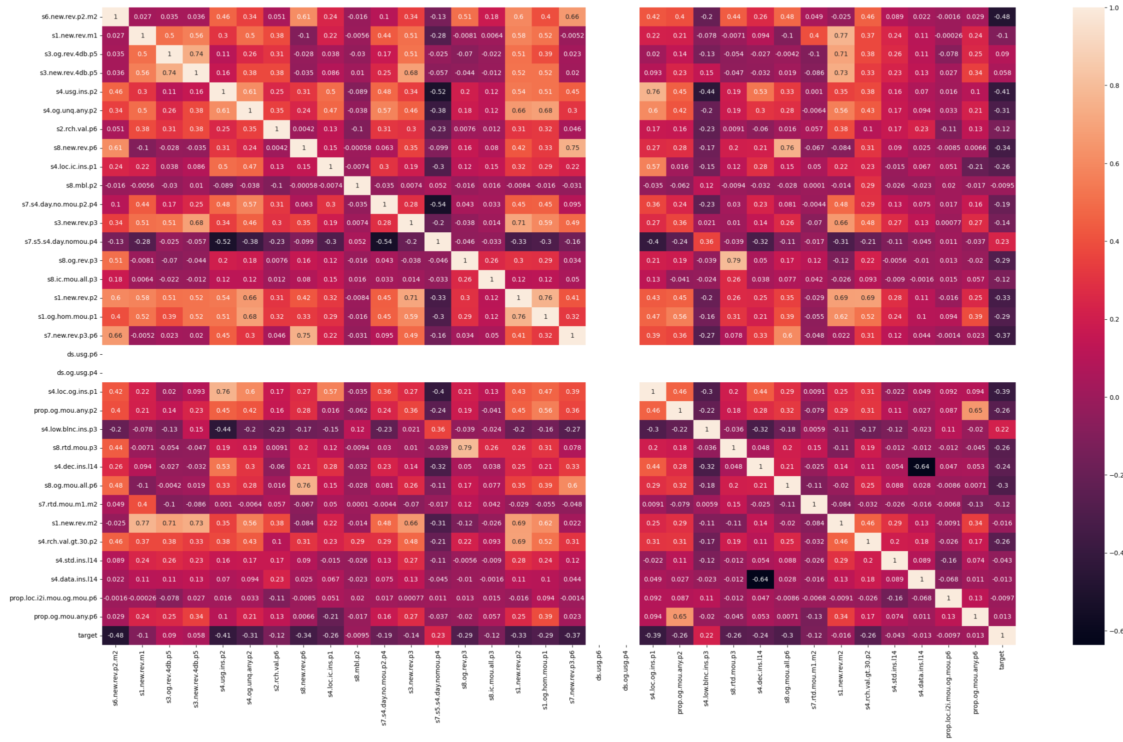
[25]:
```
df1.hist(figsize=(30,24))
plt.show()
```



Heat Map

[26]:
```
s=df1.select_dtypes(include=["integer","float"]).corr()
plt.figure(figsize=(32,18))
sns.heatmap(s,annot=True)
```

[26]: <Axes: >



5.1 Droping the Irrelevant Columns

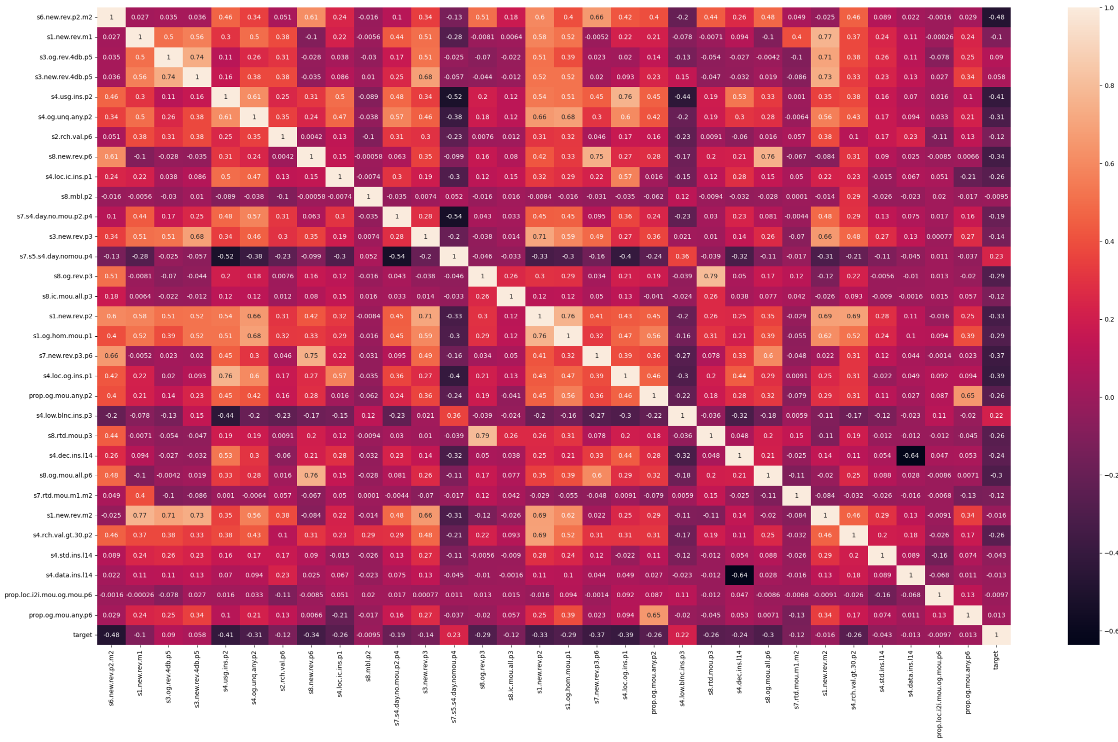
[27]:
```python
df1 = df1.drop(['ds.og.usg.p4','ds.usg.p6'], axis=1)
```

[28]:
```python
df1.shape
```

[28]: (25000, 32)

[29]:
```python
s=df1.select_dtypes(include=["integer","float"]).corr()
plt.figure(figsize=(32,18))
sns.heatmap(s,annot=True)
```

[29]: <Axes: >



6 Overview of Target Variable

[30]:
```
df1['target'].value_counts()
```

[30]:
```
target
0    17083
1     7917
Name: count, dtype: int64
```

[31]:
```
import plotly.graph_objects as go
fig=go.Figure(data=[go.Pie(labels =['Retained (0)', 'Exited␣
↪(1)'],values=df['target'].value_counts())])
fig.update_layout(width=500, height=400)
fig.show()
```

7 Feature Selection and Prepare the Data

[32]:
```
X = df1.drop('target', axis='columns')
y = df1['target']
```

[33]:
```
X.shape
```

[33]: (25000, 31)

[34]:
```
y.shape
```

[34]: (25000,)

7.0.1 Standardize the data

[35]:
```
from sklearn.preprocessing import StandardScaler
# Create a StandardScaler object
scaler = StandardScaler()

# Fit the scaler to the features
scaler.fit(X)

# Transform the features (standardize the data)
standardized_features = scaler.transform(X)
```

7.1 Splitting The Dataset

[36]:
```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
↪random_state=42)

print("X_train shape:", X_train.shape)
print("X_test shape:", X_test.shape)
print("y_train shape:", y_train.shape)
print("y_test shape:", y_test.shape)
```

```
X_train shape: (20000, 31)
X_test shape: (5000, 31)
y_train shape: (20000,)
y_test shape: (5000,)
```

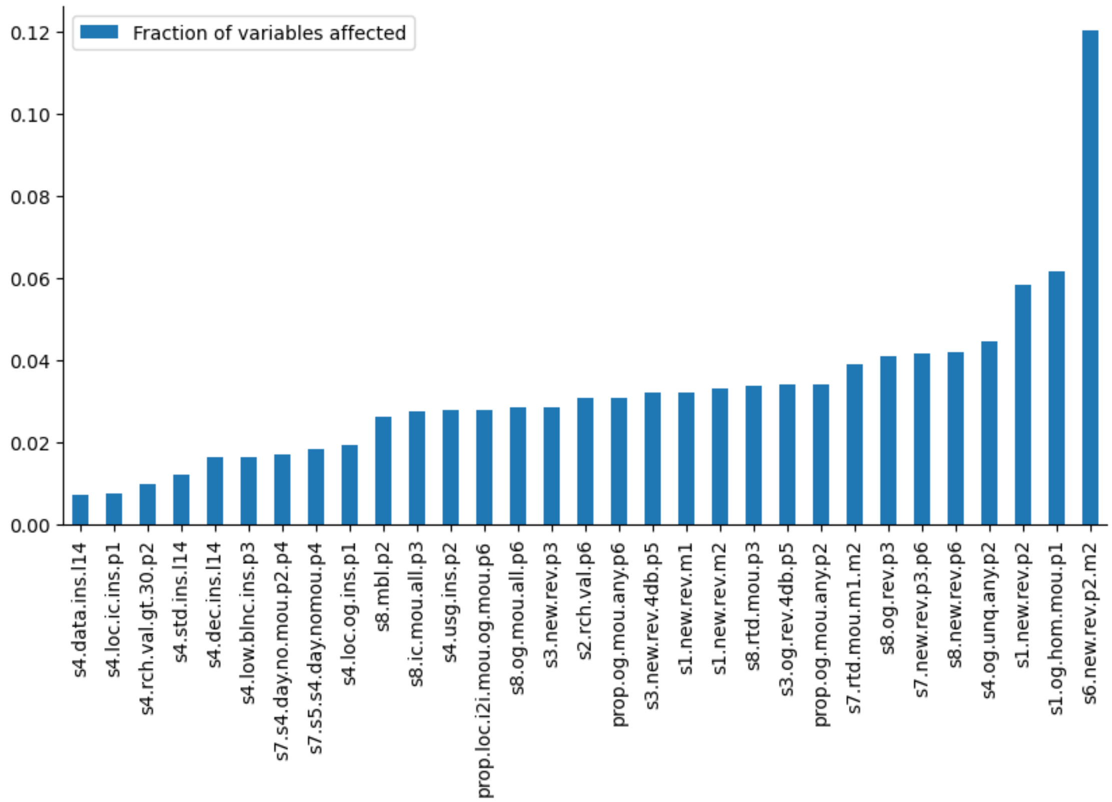
7.2 Check Importance of Features

[37]:
```
from sklearn.ensemble import RandomForestClassifier
rf = RandomForestClassifier()
rf.fit(X_train,y_train.values.ravel())
```

[37]: RandomForestClassifier()

[38]:
```
feat_scores = pd.DataFrame({"Fraction of variables affected" : rf.
↪feature_importances_},index=X.columns)
feat_scores = feat_scores.sort_values(by = "Fraction of variables affected")
feat_scores.plot(kind ="bar", figsize = (10,5))
```

```
sns.despine()
```



[39]:
```
importances = rf.feature_importances_
feature_importance_df = pd.DataFrame({'feature': X_train.columns, 'importance':␣
↪importances})
feature_importance_df = feature_importance_df.sort_values('importance',␣
↪ascending=False)
print(feature_importance_df)
```

```
                   feature  importance
0           s6.new.rev.p2.m2    0.120055
16          s1.og.hom.mou.p1    0.061628
15             s1.new.rev.p2    0.058491
5           s4.og.unq.any.p2    0.044474
7              s8.new.rev.p6    0.041998
17          s7.new.rev.p3.p6    0.041785
13              s8.og.rev.p3    0.040948
24          s7.rtd.mou.m1.m2    0.038856
19        prop.og.mou.any.p2    0.034123
2           s3.og.rev.4db.p5    0.033943
21             s8.rtd.mou.p3    0.033794
25             s1.new.rev.m2    0.033078
```

```
1                 s1.new.rev.m1     0.032209
3             s3.new.rev.4db.p5     0.031998
30           prop.og.mou.any.p6     0.030764
6                 s2.rch.val.p6     0.030744
11                s3.new.rev.p3     0.028537
23              s8.og.mou.all.p6     0.028532
29   prop.loc.i2i.mou.og.mou.p6     0.027858
4                 s4.usg.ins.p2     0.027721
14              s8.ic.mou.all.p3     0.027651
9                     s8.mbl.p2     0.026228
18              s4.loc.og.ins.p1     0.019515
12          s7.s5.s4.day.nomou.p4     0.018248
10         s7.s4.day.no.mou.p2.p4     0.016930
20           s4.low.blnc.ins.p3     0.016536
22               s4.dec.ins.l14     0.016465
27               s4.std.ins.l14     0.012075
26          s4.rch.val.gt.30.p2     0.009920
8               s4.loc.ic.ins.p1     0.007618
28              s4.data.ins.l14     0.007278
```

8 Model Building & Training

8.0.1 Random Forest

[40]:
```python
from sklearn.ensemble import RandomForestClassifier
model_rf = RandomForestClassifier(n_estimators=200, random_state=42)
# Train the model
model_rf.fit(X_train, y_train)
```

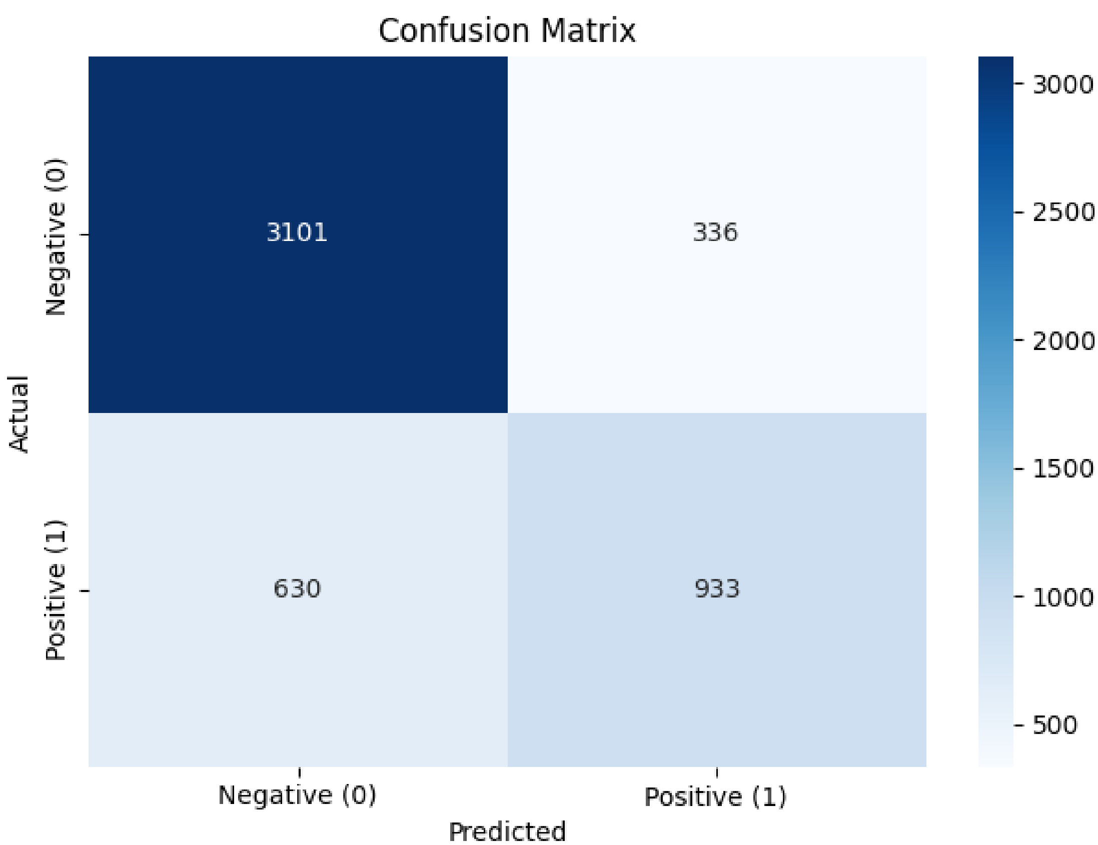
[40]:
```
RandomForestClassifier(n_estimators=200, random_state=42)
```

[41]:
```python
y_predict = model_rf.predict(X_test)
```

[42]:
```python
from sklearn.metrics import classification_report
print(classification_report(y_test,y_predict))
```

```
              precision    recall  f1-score   support

           0       0.83      0.90      0.87      3437
           1       0.74      0.60      0.66      1563

    accuracy                           0.81      5000
   macro avg       0.78      0.75      0.76      5000
weighted avg       0.80      0.81      0.80      5000
```

[43]:
```python
from sklearn.metrics import confusion_matrix
cm = confusion_matrix(y_test,y_predict)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=True)

# Add labels and title
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')

# Add class labels
class_names = ['Negative (0)', 'Positive (1)']
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks + 0.5, class_names)
plt.yticks(tick_marks + 0.5, class_names)

# Show the plot
plt.tight_layout()
plt.show()
```



Accuracy:(3092 + 952) / 5000 = 0.8088 or 80.88%   81%

Precision for Positive class: 952 / (952 + 345) = 0.7425 or 74.25%   74%

Recall for Positive class: 952 / (952 + 611) = 0.6090 or 60.90%   61%

F1 Score for Positive class: 2 * (0.7425 * 0.6090) / (0.7425 + 0.6090) = 0.6691 or 66.91%   67%

Specificity (True Negative Rate): 3092 / (3092 + 345) = 0.8996 or 89.96%   90%

So as per the above, The model shows strong performance in correctly identifying negative cases (90% specificity) but struggles more with correctly identifying positive cases (61% recall)

8.0.2 HyperParameter Tuning Using RandomizedCV

[44]:
```
import numpy as np
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV, train_test_split
from sklearn.metrics import accuracy_score, classification_report



# Create a subset of the training data for faster tuning (e.g., 25% of the␣
↪training data)
X_train_subset, _, y_train_subset, _ = train_test_split(X_train, y_train,␣
↪test_size=0.75, random_state=42)

# Define hyperparameter distributions for Random Forest
param_grid = {
    'n_estimators': [200, 300, 800],
    'max_features': ['auto', 'sqrt', 'log2'],
    'max_depth': [10, 20, 30]
}

# Create the initial Random Forest model
model_rf = RandomForestClassifier(random_state=42)

# Create RandomizedSearchCV object for Random Forest
r_search_rf = RandomizedSearchCV(
    estimator=RandomForestClassifier(random_state=42),
    param_distributions=param_grid,
    n_iter=50,  # Reduced number of iterations
    cv=3,
    verbose=1,
    random_state=42,
    n_jobs=-1,
    scoring='accuracy'
)

# Perform hyperparameter tuning on the subset of training data
r_search_rf.fit(X_train_subset, y_train_subset)
```

```python
# Print best parameters and score for Random Forest
print("Random Forest - Best Parameters:", r_search_rf.best_params_)
print("Best Cross-Validation Score:", r_search_rf.best_score_)

# Use the tuned model for prediction on the full testing set
y_pred_rf = r_search_rf.predict(X_test)

# Calculate accuracy on the full test set
test_accuracy = accuracy_score(y_test, y_pred_rf)
print("Test Set Accuracy:", test_accuracy)

# Print detailed classification report
print("\nClassification Report:")
print(classification_report(y_test, y_pred_rf))

# Optional: Calculate feature importances of the best model
best_rf = r_search_rf.best_estimator_
feature_importance = best_rf.feature_importances_
for i, importance in enumerate(feature_importance):
    print(f"Feature {i}: {importance}")

# If you want to retrain on the full training set with the best parameters:
best_params = r_search_rf.best_params_
final_model = RandomForestClassifier(**best_params, random_state=42)
final_model.fit(X_train, y_train)

# Evaluate the final model on the test set
final_predictions = final_model.predict(X_test)
final_accuracy = accuracy_score(y_test, final_predictions)
print("\nFinal Model Test Set Accuracy:", final_accuracy)
print("\nFinal Model Classification Report:")
print(classification_report(y_test, final_predictions))
```

```
Fitting 3 folds for each of 27 candidates, totalling 81 fits
Random Forest - Best Parameters: {'n_estimators': 800, 'max_features': 'sqrt',
'max_depth': 10}
Best Cross-Validation Score: 0.8042009845329853
Test Set Accuracy: 0.8046

Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.91      0.86      3437
           1       0.74      0.58      0.65      1563

    accuracy                           0.80      5000
   macro avg       0.78      0.74      0.76      5000
```

```
weighted avg       0.80      0.80      0.80      5000

Feature 0: 0.15807730617338187
Feature 1: 0.02590825225490584
Feature 2: 0.029210824604385687
Feature 3: 0.025817017293860548
Feature 4: 0.03455751422179129
Feature 5: 0.04439388566065968
Feature 6: 0.02562600040051376
Feature 7: 0.045539537615241335
Feature 8: 0.007191420794751608
Feature 9: 0.02055612621533659
Feature 10: 0.013368475226461488
Feature 11: 0.025902311266827276
Feature 12: 0.01480599691849528
Feature 13: 0.03869219468520218
Feature 14: 0.02116528655005848
Feature 15: 0.06891855618733321
Feature 16: 0.0694149843757862
Feature 17: 0.05200004516108975
Feature 18: 0.021755981504536406
Feature 19: 0.02853424872604049
Feature 20: 0.0136045038350219
Feature 21: 0.034189871192019515
Feature 22: 0.017566842093819535
Feature 23: 0.03413752491044964
Feature 24: 0.03281359923893729
Feature 25: 0.026316230936654184
Feature 26: 0.008415260102313861
Feature 27: 0.009264642048504567
Feature 28: 0.006402657851843164
Feature 29: 0.01999029206992341
Feature 30: 0.025862609883854044

Final Model Test Set Accuracy: 0.8068

Final Model Classification Report:
              precision    recall  f1-score   support

           0       0.83      0.90      0.87      3437
           1       0.74      0.59      0.66      1563

    accuracy                           0.81      5000
   macro avg       0.78      0.75      0.76      5000
weighted avg       0.80      0.81      0.80      5000
```

There was not any changes in the model's accuracy. So I'm considering the 81% accuracy for now.

8.0.3 Logistic Regression

[45]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import classification_report, confusion_matrix

model_lr = LogisticRegression()
model_lr.fit(X_train,y_train)
```

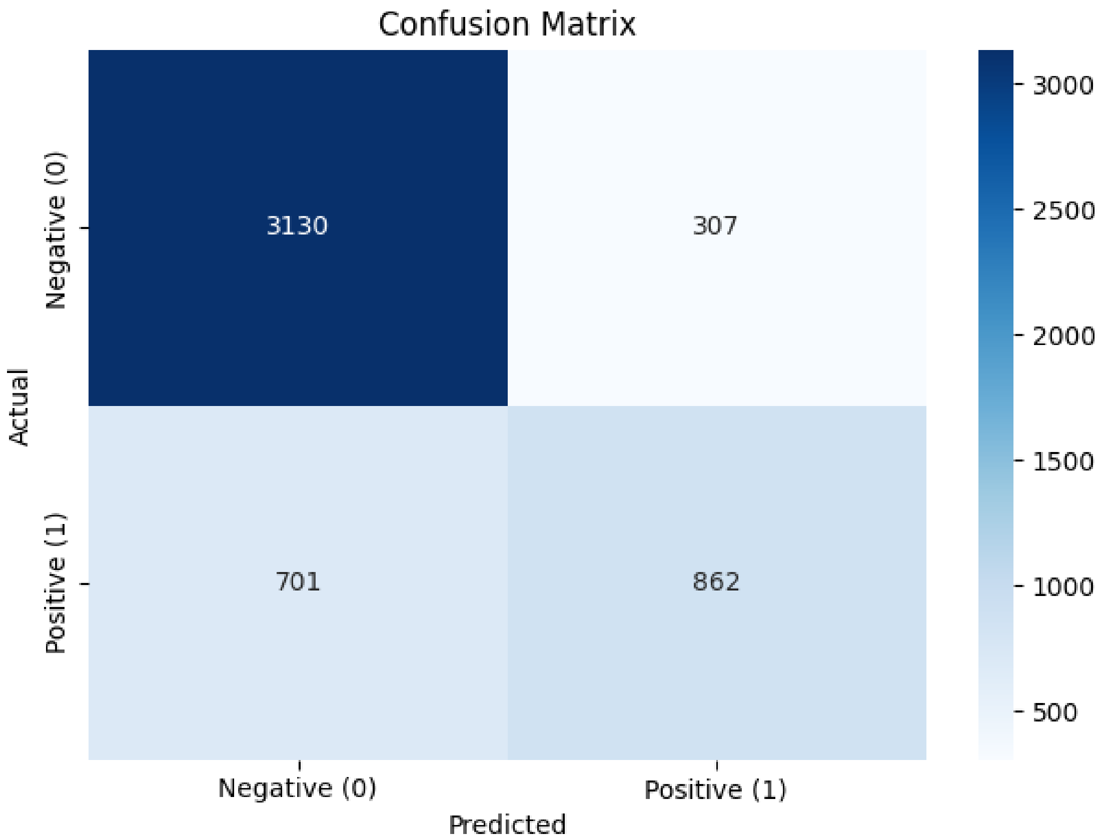
[45]:
```
LogisticRegression()
```

[46]:
```
y_predict = model_lr.predict(X_test)
```

[47]:
```
print(classification_report(y_test,y_predict))
```

```
              precision    recall  f1-score   support

           0       0.82      0.91      0.86      3437
           1       0.74      0.55      0.63      1563

    accuracy                           0.80      5000
   macro avg       0.78      0.73      0.75      5000
weighted avg       0.79      0.80      0.79      5000
```

[48]:
```
cm = confusion_matrix(y_test,y_predict)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=True)

# Add labels and title
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')

# Add class labels
class_names = ['Negative (0)', 'Positive (1)']
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks + 0.5, class_names)
plt.yticks(tick_marks + 0.5, class_names)

# Show the plot
plt.tight_layout()
plt.show()
```



Accuracy:(3108 + 878) / 5000 = 0.7972 or   80%

Precision for Positive class: 878 / (878 + 329) = 0.7277 or   73%

Recall for Positive class: 878 / (878 + 685) = 0.5616 or   56%

F1 Score for Positive class: 2 * (0.7277 * 0.5616) / (0.7277 + 0.5616) = 0.6346 or 63.46%   63%

Specificity (True Negative Rate): 3108 / (3108 + 329) = 0.9042 or 90.42%   90%

8.0.4 K - Nearest Neighbour

[49]:
```
from sklearn.neighbors import KNeighborsClassifier
model_knn = KNeighborsClassifier()
model_knn.fit(X_train,y_train)
```

[49]:
```
KNeighborsClassifier()
```

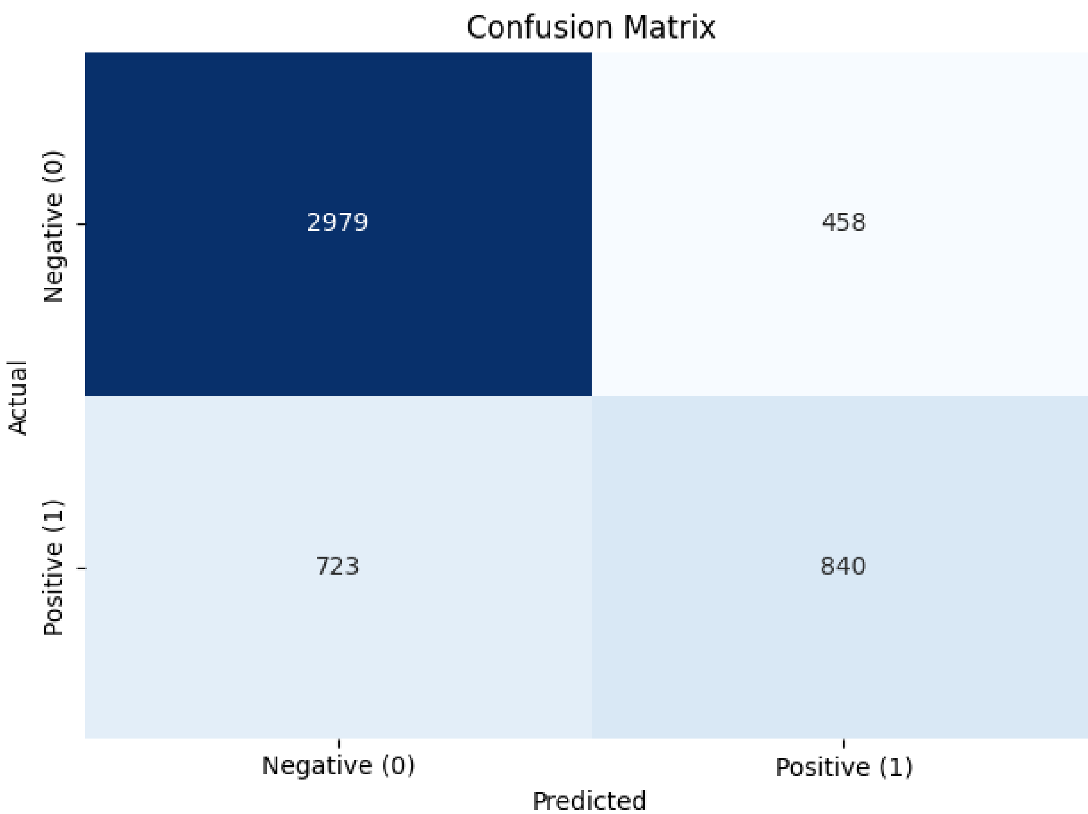
[50]:
```
y_predict = model_knn.predict(X_test)
print(classification_report(y_test, y_predict))
```

```
              precision    recall  f1-score   support
```

```
             0       0.80      0.87      0.83      3437
             1       0.65      0.54      0.59      1563

      accuracy                           0.76      5000
     macro avg       0.73      0.70      0.71      5000
  weighted avg       0.76      0.76      0.76      5000
```

[51]:

```python
cm = confusion_matrix(y_test,y_predict)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=False)

# Add labels and title
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')

# Add class labels
class_names = ['Negative (0)', 'Positive (1)']
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks + 0.5, class_names)
plt.yticks(tick_marks + 0.5, class_names)

# Show the plot
plt.tight_layout()
plt.show()
```



8.0.5 Linear Support Vector Machine

[52]:
```
from sklearn.calibration import CalibratedClassifierCV
from sklearn.svm import LinearSVC

model_svc = LinearSVC()
model_svc.fit(X_train,y_train)
```

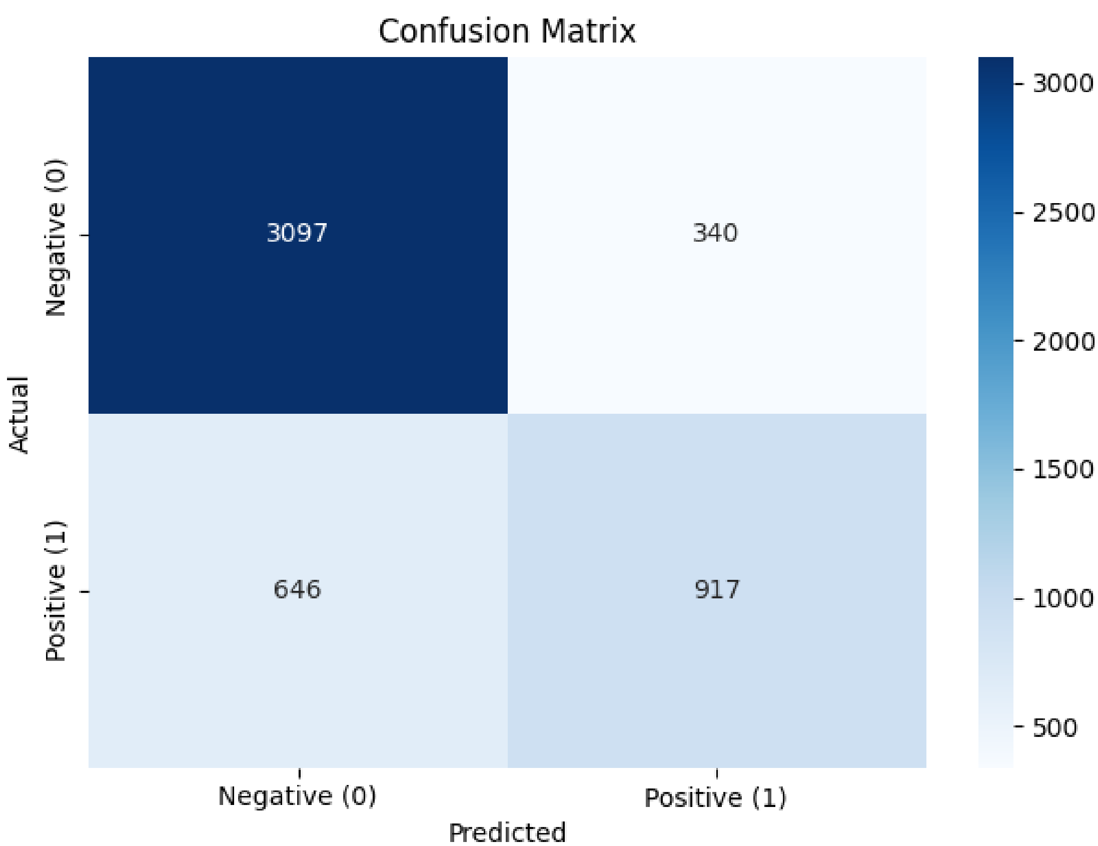
[52]:
```
LinearSVC()
```

[53]:
```
y_predict = model_svc.predict(X_test)
print(classification_report(y_test, y_predict))
```

```
              precision    recall  f1-score   support

           0       0.83      0.90      0.86      3437
           1       0.73      0.59      0.65      1563

    accuracy                           0.80      5000
   macro avg       0.78      0.74      0.76      5000
weighted avg       0.80      0.80      0.80      5000
```

[54]:

```
cm = confusion_matrix(y_test,y_predict)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=True)

# Add labels and title
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')

# Add class labels
class_names = ['Negative (0)', 'Positive (1)']
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks + 0.5, class_names)
plt.yticks(tick_marks + 0.5, class_names)

# Show the plot
plt.tight_layout()
plt.show()
```



8.0.6 Naive Bayes

[55]:
```
from sklearn.naive_bayes import GaussianNB
model_nb = GaussianNB()
model_nb.fit(X_train,y_train)
```

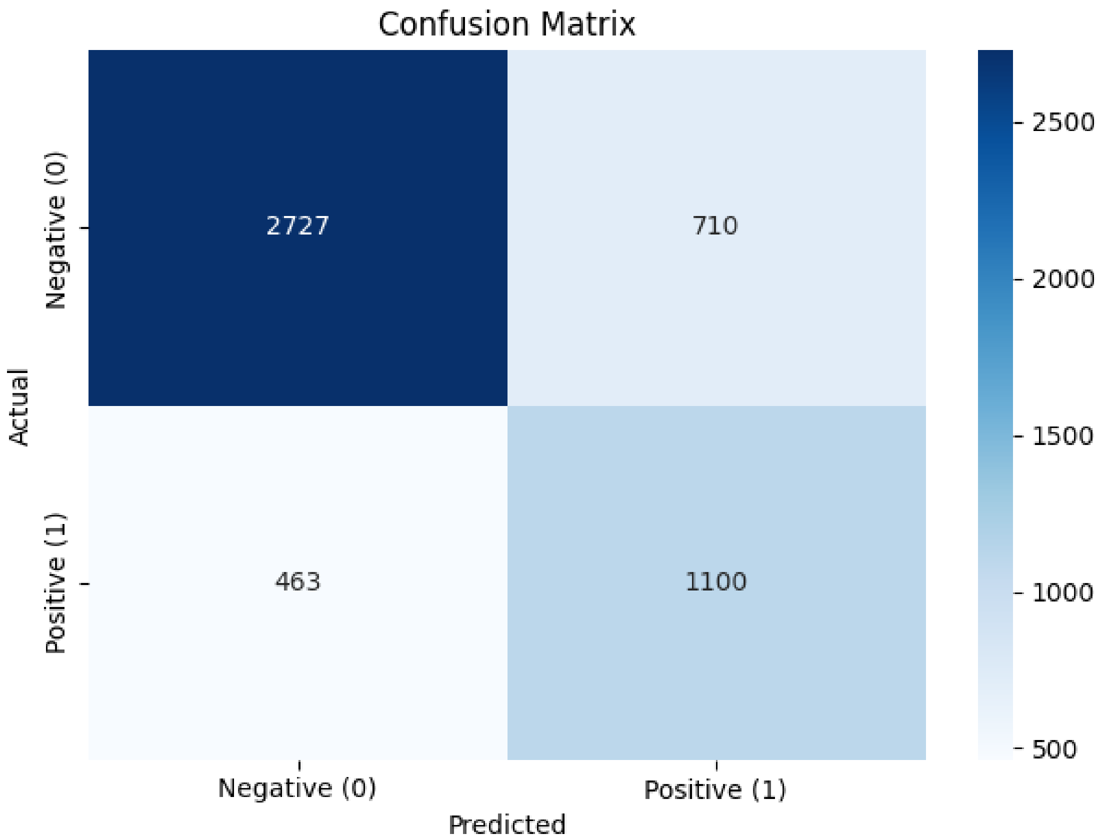
[55]:
```
GaussianNB()
```

[56]:
```
y_predict = model_nb.predict(X_test)
```

[57]:
```
print(classification_report(y_test, y_predict))
```

```
              precision    recall  f1-score   support

           0       0.85      0.79      0.82      3437
           1       0.61      0.70      0.65      1563

    accuracy                           0.77      5000
   macro avg       0.73      0.75      0.74      5000
weighted avg       0.78      0.77      0.77      5000
```

[58]:
```
cm = confusion_matrix(y_test,y_predict)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=True)

# Add labels and title
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')

# Add class labels
class_names = ['Negative (0)', 'Positive (1)']
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks + 0.5, class_names)
plt.yticks(tick_marks + 0.5, class_names)

# Show the plot
plt.tight_layout()
plt.show()
```



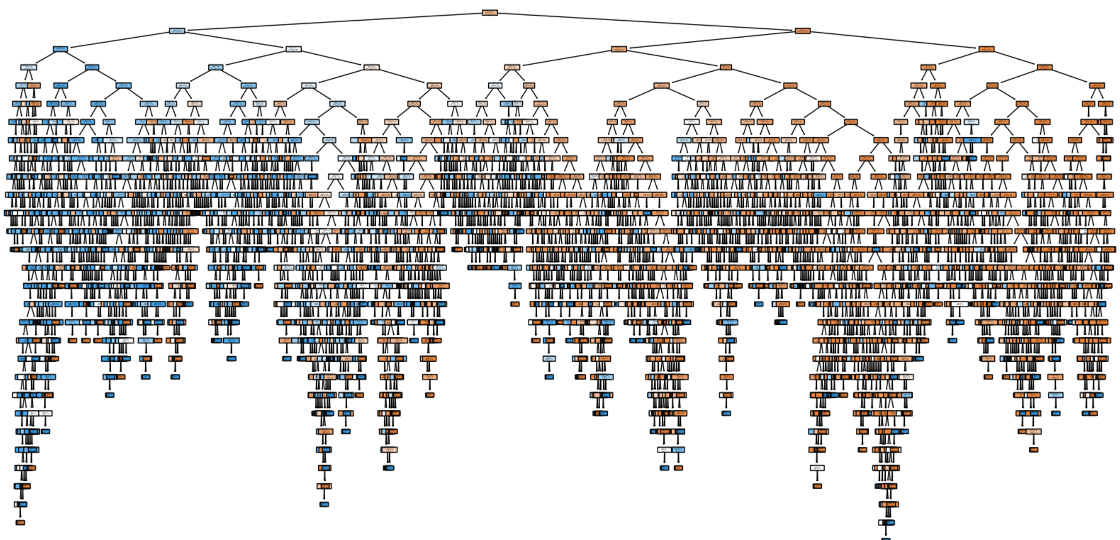
8.0.7 Decision Tree

[72]:
```python
from sklearn.tree import DecisionTreeClassifier
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score, classification_report,␣
↪confusion_matrix
import matplotlib.pyplot as plt
from sklearn.tree import plot_tree

# Create and train the model
model_dt = DecisionTreeClassifier(random_state=42)
model_dt.fit(X_train, y_train)

# Make predictions
y_pred = model_dt.predict(X_test)

# Evaluate the model
accuracy = accuracy_score(y_test, y_pred)
print(f"Accuracy: {accuracy:.2f}")
```

```
print("\nClassification Report:")
print(classification_report(y_test, y_pred))

print("\nConfusion Matrix:")
print(confusion_matrix(y_test, y_pred))

# Visualize the decision tree
plt.figure(figsize=(20,10))
plot_tree(model_dt, feature_names=X.columns, class_names=model_dt.classes_.
↪astype(str), filled=True, rounded=True)
plt.show()

# Feature importance
feature_importance = pd.DataFrame({'feature': X.columns, 'importance': model_dt.
↪feature_importances_})
feature_importance = feature_importance.sort_values('importance',␣
↪ascending=False).reset_index(drop=True)
print("\nFeature Importance:")
print(feature_importance)

# Hyperparameter tuning (optional)
from sklearn.model_selection import GridSearchCV

param_grid = {
    'max_depth': [3, 5, 7, 10],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

grid_search = GridSearchCV(DecisionTreeClassifier(random_state=42), param_grid,␣
↪cv=5)
grid_search.fit(X_train, y_train)

print("\nBest parameters:", grid_search.best_params_)
print("Best cross-validation score:", grid_search.best_score_)

# Train the model with best parameters
best_dt = grid_search.best_estimator_
best_dt.fit(X_train, y_train)

# Evaluate the tuned model
y_pred_tuned = best_dt.predict(X_test)
accuracy_tuned = accuracy_score(y_test, y_pred_tuned)
print(f"\nTuned Model Accuracy: {accuracy_tuned:.2f}")
```

```
Accuracy: 0.71
```

```
Classification Report:
              precision    recall  f1-score   support

           0       0.80      0.78      0.79      3437
           1       0.54      0.58      0.56      1563

    accuracy                           0.71      5000
   macro avg       0.67      0.68      0.67      5000
weighted avg       0.72      0.71      0.72      5000


Confusion Matrix:
[[2666  771]
 [ 661  902]]
```

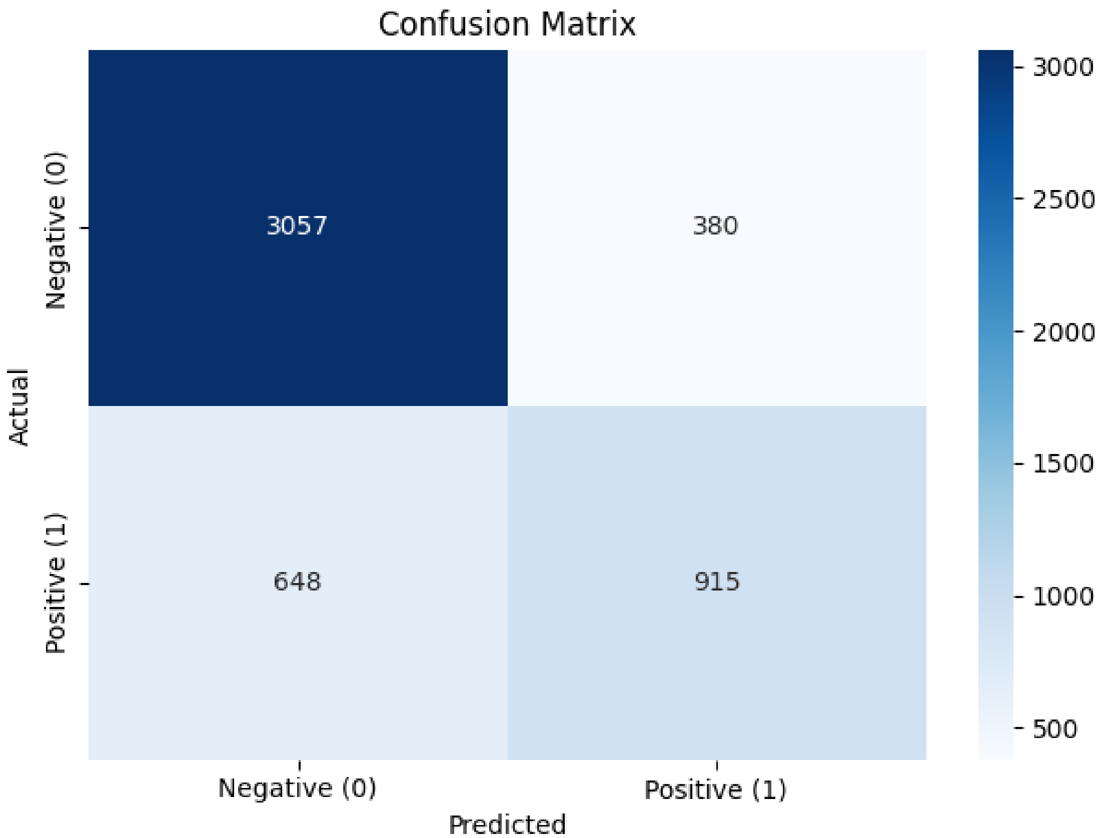


```
Feature Importance:
                       feature  importance
0               s6.new.rev.p2.m2    0.310550
1               s7.rtd.mou.m1.m2    0.052553
2                  s2.rch.val.p6    0.038212
3               s3.og.rev.4db.p5    0.037177
4              s3.new.rev.4db.p5    0.032750
5                  s8.new.rev.p6    0.031276
6    prop.loc.i2i.mou.og.mou.p6    0.031016
7             prop.og.mou.any.p6    0.030550
8               s8.ic.mou.all.p3    0.029318
9                   s8.og.rev.p3    0.029139
10                 s3.new.rev.p3    0.027934
```

```
11                s1.new.rev.m1    0.027503
12             s1.og.hom.mou.p1    0.027378
13                    s8.mbl.p2    0.026297
14           prop.og.mou.any.p2    0.025734
15                s1.new.rev.m2    0.024601
16             s4.og.unq.any.p2    0.024239
17             s7.new.rev.p3.p6    0.022633
18             s8.og.mou.all.p6    0.022312
19                s8.rtd.mou.p3    0.021391
20                s1.new.rev.p2    0.019740
21       s7.s5.s4.day.nomou.p4    0.016595
22           s4.low.blnc.ins.p3    0.016200
23      s7.s4.day.no.mou.p2.p4    0.015163
24               s4.dec.ins.l14    0.014387
25               s4.std.ins.l14    0.011982
26                s4.usg.ins.p2    0.010809
27             s4.loc.og.ins.p1    0.008627
28              s4.data.ins.l14    0.005200
29          s4.rch.val.gt.30.p2    0.005072
30             s4.loc.ic.ins.p1    0.003659

Best parameters: {'max_depth': 5, 'min_samples_leaf': 2, 'min_samples_split': 2}
Best cross-validation score: 0.7878499999999999

Tuned Model Accuracy: 0.79
```

[73]:

```
cm = confusion_matrix(y_test,y_pred_tuned)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=True)

# Add labels and title
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')

# Add class labels
class_names = ['Negative (0)', 'Positive (1)']
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks + 0.5, class_names)
plt.yticks(tick_marks + 0.5, class_names)

# Show the plot
plt.tight_layout()
plt.show()
```



9 ROC Curve

[74]:
```
from sklearn.metrics import roc_curve
fpr1,tpr1,thresh1 = roc_curve(y_test, best_rf.predict_proba(X_test)[:
 ↪,1],pos_label=1)
fpr2,tpr2,thresh2 = roc_curve(y_test, model_lr.predict_proba(X_test)[:
 ↪,1],pos_label=1)
fpr3,tpr3,thresh3 = roc_curve(y_test, model_knn.predict_proba(X_test)[:
 ↪,1],pos_label=1)
fpr4,tpr4,thresh4 = roc_curve(y_test, model_svc.
 ↪decision_function(X_test),pos_label=1)
fpr5,tpr5,thresh5 = roc_curve(y_test, model_nb.predict_proba(X_test)[:
 ↪,1],pos_label=1)
fpr6,tpr6,thresh6 = roc_curve(y_test, model_dt.predict_proba(X_test)[:
 ↪,1],pos_label=1)
```

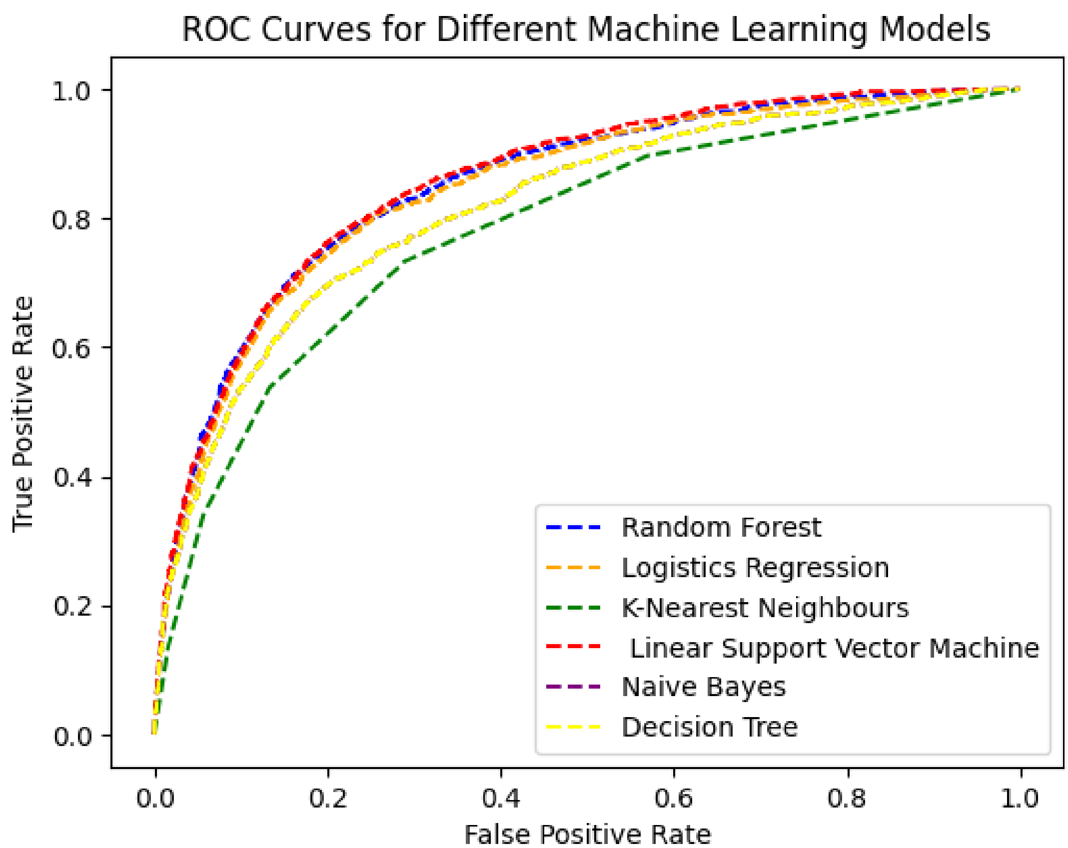
[75]:
```
from sklearn.metrics import roc_auc_score
roc_auc_score1 = roc_auc_score(y_test, best_rf.predict_proba(X_test)[:,1])
roc_auc_score2 = roc_auc_score(y_test, model_lr.predict_proba(X_test)[:,1])
```

```
roc_auc_score3 = roc_auc_score(y_test, model_knn.predict_proba(X_test)[:,1])
roc_auc_score4 = roc_auc_score(y_test, model_svc.decision_function(X_test))
roc_auc_score5 = roc_auc_score(y_test, model_nb.predict_proba(X_test)[:,1])
roc_auc_score6 = roc_auc_score(y_test, model_dt.predict_proba(X_test)[:,1])

print("Random Forest:", roc_auc_score1)
print("Logistic Regression", roc_auc_score2)
print("Support Vector Machine", roc_auc_score3)
print("K-Nearest Neighbours", roc_auc_score4)
print("Naive Bayes", roc_auc_score4)
print("Decision Tree", roc_auc_score5)
```

```
Random Forest: 0.8517385696396764
Logistic Regression 0.8450394273599686
Support Vector Machine 0.7800799176326421
K-Nearest Neighbours 0.8568986292149097
Naive Bayes 0.8568986292149097
Decision Tree 0.8169498649579646
```

[76]:

```
plt.plot(fpr1,tpr1,linestyle='--', color='blue', label='Random Forest')
plt.plot(fpr2,tpr2,linestyle='--', color='orange', label='Logistics Regression')
plt.plot(fpr3,tpr3,linestyle='--', color='green', label='K-Nearest Neighbours')
plt.plot(fpr4,tpr4,linestyle='--', color='red', label=' Linear Support Vector␣
↪Machine')
plt.plot(fpr5,tpr5,linestyle='--', color='purple', label='Naive Bayes')
plt.plot(fpr5,tpr5,linestyle='--', color='yellow', label='Decision Tree')

plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves for Different Machine Learning Models')

plt.legend(loc='best')  # add legend

plt.show()
```



10 SMOTE

[77]:
```
from imblearn.over_sampling import SMOTE

#initialize SMOTE with a random state for reproducibility
smote = SMOTE(random_state=42)

#Fit SMOTE to the training data and transform it
X_smote, y_smote = smote.fit_resample(X_train, y_train)

print("Original Training Shape:", X_train.shape, y_train.shape)
print("Resampled SMOTE Training Shape:", X_smote.shape, y_smote.shape)
```

```
Original Training Shape: (20000, 31) (20000,)
Resampled SMOTE Training Shape: (27292, 31) (27292,)
```

[78]:
```
print("y_train distribution:")
print(y_train.value_counts())
```

```
print("y_test distribution:")
print(y_test.value_counts())
```

```
y_train distribution:
target
0    13646
1     6354
Name: count, dtype: int64
y_test distribution:
target
0    3437
1    1563
Name: count, dtype: int64
```

[79]:
```
print("y_smote distribution:")
print(y_smote.value_counts())
```

```
y_smote distribution:
target
0    13646
1    13646
Name: count, dtype: int64
```

10.1 SMOTE RF

[80]:
```
from sklearn.ensemble import RandomForestClassifier
model_rf = RandomForestClassifier(n_estimators=200, random_state=42)
# Train the model
model_rf.fit(X_smote, y_smote)
```

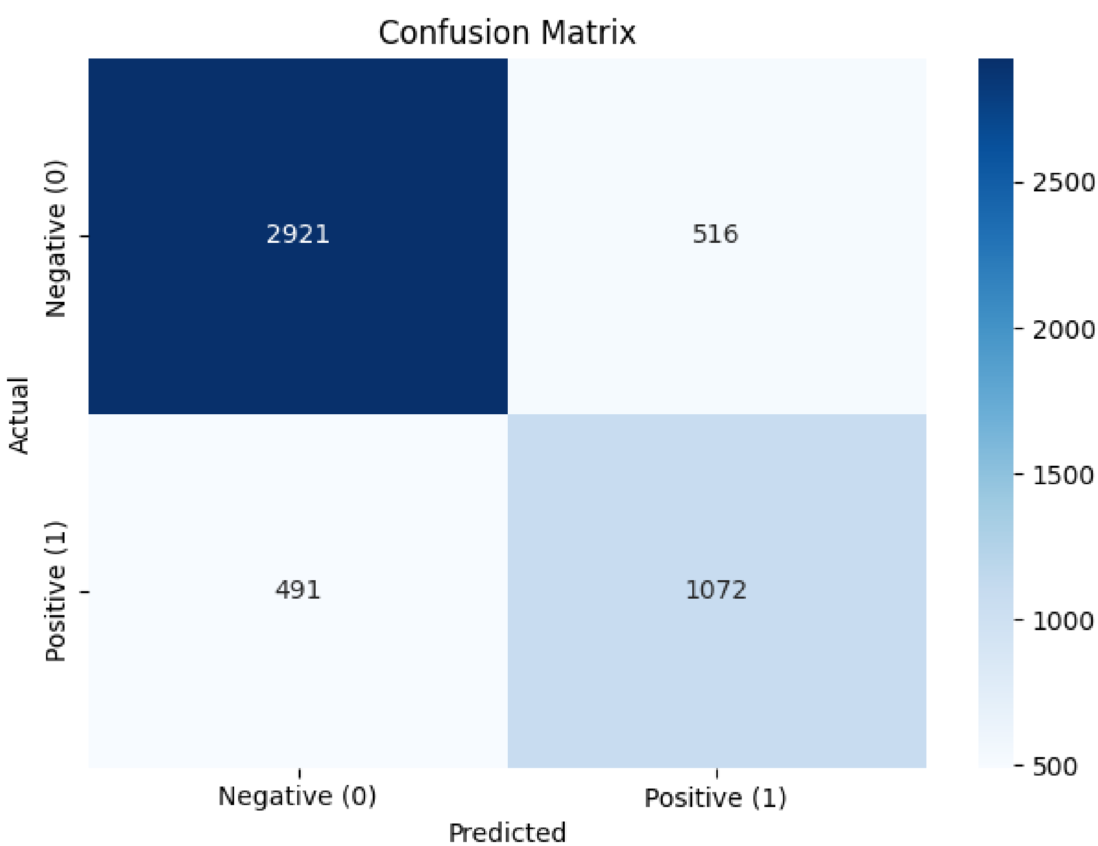
[80]:
```
RandomForestClassifier(n_estimators=200, random_state=42)
```

[81]:
```
y_predict = model_rf.predict(X_test)
```

[82]:
```
from sklearn.metrics import classification_report
print(classification_report(y_test,y_predict))
```

```
              precision    recall  f1-score   support

           0       0.86      0.85      0.85      3437
           1       0.68      0.69      0.68      1563

    accuracy                           0.80      5000
   macro avg       0.77      0.77      0.77      5000
weighted avg       0.80      0.80      0.80      5000
```

[83]:
```
cm = confusion_matrix(y_test,y_predict)
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues', cbar=True)

# Add labels and title
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix')

# Add class labels
class_names = ['Negative (0)', 'Positive (1)']
tick_marks = np.arange(len(class_names))
plt.xticks(tick_marks + 0.5, class_names)
plt.yticks(tick_marks + 0.5, class_names)

# Show the plot
plt.tight_layout()
plt.show()
```



CONCLUSION

Before SMOTE (After Hyperparameter tuning):

Class 0: Precision 0.83, Recall 0.90, F1-score 0.87
Class 1: Precision 0.74, Recall 0.59, F1-score 0.66
Overall accuracy: 0.81

After SMOTE:

Class 0: Precision 0.86, Recall 0.86, F1-score 0.86
Class 1: Precision 0.69, Recall 0.68, F1-score 0.68
Overall accuracy: 0.80

While overall accuracy decreased, there's significant changes in other metrics as well.
Churn prediction (Class 1) improved:

1. Recall increased significantly from 0.59 to 0.68. This means the model is now better at identifying customers who will churn.
2. F1-score for churn improved from 0.66 to 0.68, indicating a better balance between precision and recall.

Non-churn prediction (Class 0) balanced out:

1. Precision improved slightly (0.83 to 0.86)
2. Recall decreased (0.90 to 0.86)
3. F1-score remained nearly the same (0.87 to 0.86)

Conclusion:

1. Improved churn detection: The increase in recall for the churn class (0.59 to 0.68) means the model is now catching more potential churners. This is often more valuable than overall accuracy in churn prediction scenarios.
2. Balanced performance: The more balanced recall between classes (0.86 for non-churn, 0.68 for churn) suggests the model is less biased towards the majority class.
3. False positives vs. false negatives: The slight decrease in precision for churn predictions (0.74 to 0.69) means there might be more false positives.
   > However, in churn prediction, it's often preferable to have more false positives (predicting churn when it doesn't happen) than false negatives (missing actual churners).