churn-prediction-system

October 17, 2024

1 Churn Predictor: Decoding Customer Behavior

1.0.1 Shashank Pandey

Contact: pandey22@buffalo.edu

Welcome to the Churn Predictor project!

This notebook explores the fascinating world of customer behavior analysis and predictive modeling. We're on a mission to understand and forecast customer churn using cutting-edge data science techniques.

Feel free to explore, learn, and build upon this work. If you find it valuable in your own projects, a friendly mention would be greatly appreciated!

Remember: > In data we trust, but insights we must!

Let's dive in and uncover the stories hidden in our data!

May your models be accurate and your insights profound!

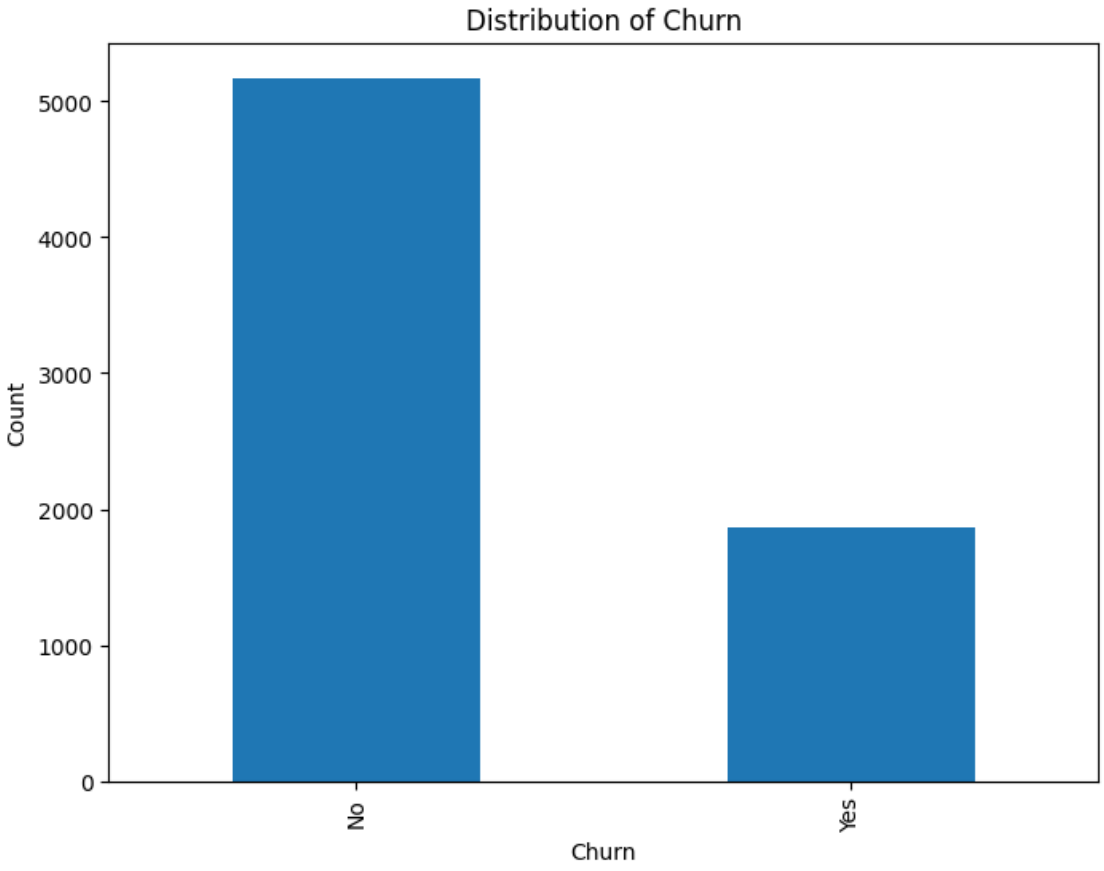
[67]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns

# Load the data
df = pd.read_csv('/root/.cache/kagglehub/datasets/blastchar/
 ↪telco-customer-churn/versions/1/WA_Fn-UseC_-Telco-Customer-Churn.csv')

# Display basic information about the dataset
print(df.info())

# Display the first few rows
print(df.head())
```

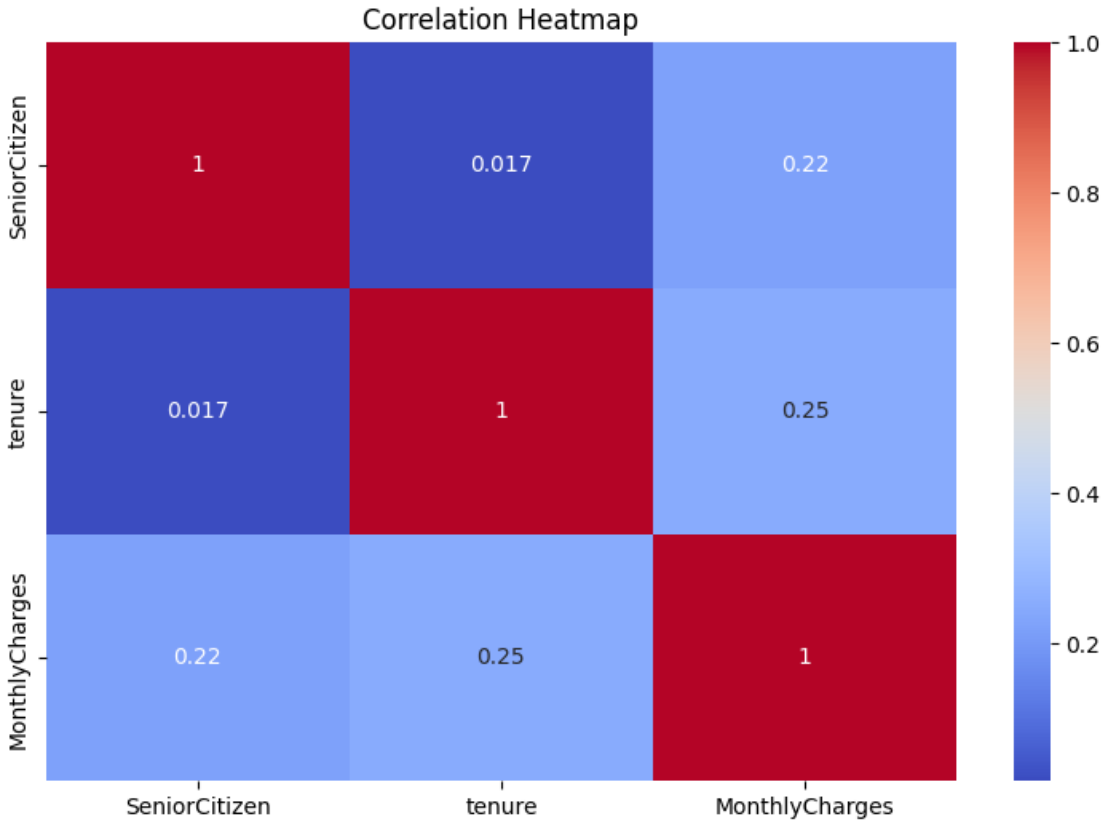
```python
# Check for missing values
print(df.isnull().sum())

# Display summary statistics
print(df.describe())

# Visualize the distribution of the target variable (Churn)
plt.figure(figsize=(8, 6))
df['Churn'].value_counts().plot(kind='bar')
plt.title('Distribution of Churn')
plt.xlabel('Churn')
plt.ylabel('Count')
plt.show()

# Correlation heatmap for numeric columns
numeric_df = df.select_dtypes(include=[np.number])
plt.figure(figsize=(9.2, 6))
sns.heatmap(numeric_df.corr(), annot=True, cmap='coolwarm')
plt.title('Correlation Heatmap')
plt.show()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7043 entries, 0 to 7042
Data columns (total 21 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   customerID        7043 non-null   object
 1   gender            7043 non-null   object
 2   SeniorCitizen     7043 non-null   int64
 3   Partner           7043 non-null   object
 4   Dependents        7043 non-null   object
 5   tenure            7043 non-null   int64
 6   PhoneService      7043 non-null   object
 7   MultipleLines     7043 non-null   object
 8   InternetService   7043 non-null   object
 9   OnlineSecurity    7043 non-null   object
 10  OnlineBackup      7043 non-null   object
 11  DeviceProtection  7043 non-null   object
 12  TechSupport       7043 non-null   object
 13  StreamingTV       7043 non-null   object
 14  StreamingMovies   7043 non-null   object
 15  Contract          7043 non-null   object
 16  PaperlessBilling  7043 non-null   object
 17  PaymentMethod     7043 non-null   object
 18  MonthlyCharges    7043 non-null   float64
 19  TotalCharges      7043 non-null   object
 20  Churn             7043 non-null   object
```

```
dtypes: float64(1), int64(2), object(18)
memory usage: 1.1+ MB
None
   customerID  gender  SeniorCitizen Partner Dependents  tenure PhoneService  \
0  7590-VHVEG  Female              0     Yes         No       1           No
1  5575-GNVDE    Male              0      No         No      34          Yes
2  3668-QPYBK    Male              0      No         No       2          Yes
3  7795-CFOCW    Male              0      No         No      45           No
4  9237-HQITU  Female              0      No         No       2          Yes

      MultipleLines InternetService OnlineSecurity  … DeviceProtection  \
0  No phone service             DSL             No  …               No
1                No             DSL            Yes  …              Yes
2                No             DSL            Yes  …               No
3  No phone service             DSL            Yes  …              Yes
4                No     Fiber optic             No  …               No

  TechSupport StreamingTV StreamingMovies        Contract PaperlessBilling  \
0          No          No              No  Month-to-month              Yes
1          No          No              No        One year               No
2          No          No              No  Month-to-month              Yes
3         Yes          No              No        One year               No
4          No          No              No  Month-to-month              Yes

               PaymentMethod MonthlyCharges  TotalCharges Churn
0           Electronic check          29.85         29.85    No
1               Mailed check          56.95        1889.5    No
2               Mailed check          53.85        108.15   Yes
3  Bank transfer (automatic)          42.30       1840.75    No
4           Electronic check          70.70        151.65   Yes

[5 rows x 21 columns]
customerID          0
gender              0
SeniorCitizen       0
Partner             0
Dependents          0
tenure              0
PhoneService        0
MultipleLines       0
InternetService     0
OnlineSecurity      0
OnlineBackup        0
DeviceProtection    0
TechSupport         0
StreamingTV         0
StreamingMovies     0
Contract            0
```

```
PaperlessBilling    0
PaymentMethod       0
MonthlyCharges      0
TotalCharges        0
Churn               0
dtype: int64
       SeniorCitizen       tenure  MonthlyCharges
count    7043.000000  7043.000000     7043.000000
mean        0.162147    32.371149       64.761692
std         0.368612    24.559481       30.090047
min         0.000000     0.000000       18.250000
25%         0.000000     9.000000       35.500000
50%         0.000000    29.000000       70.350000
75%         0.000000    55.000000       89.850000
max         1.000000    72.000000      118.750000
```





1. Data Preprocessing:

a. Convert 'TotalCharges' to numeric

b. Drop 'customerID' column

c. Encode categorical variables

d. Split features and target

e. Scale numerical features

[68]:
```
print(df.isnull().sum())
```

```
customerID          0
gender              0
SeniorCitizen       0
Partner             0
Dependents          0
tenure              0
PhoneService        0
MultipleLines       0
InternetService     0
```

```
OnlineSecurity      0
OnlineBackup        0
DeviceProtection    0
TechSupport         0
StreamingTV         0
StreamingMovies     0
Contract            0
PaperlessBilling    0
PaymentMethod       0
MonthlyCharges      0
TotalCharges        0
Churn               0
dtype: int64
```

If there are NaN values, we need to handle them. For numerical columns, we can impute with the mean or median. For categorical columns, we can impute with the mode (most frequent value). Here's how we can do this-

[68]:
```
```

[69]:
```
df['TotalCharges'] = pd.to_numeric(df['TotalCharges'], errors='coerce')
```

[70]:
```
df = df.drop('customerID', axis=1)
```

[71]:
```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
categorical_cols = df.select_dtypes(include=['object']).columns
for col in categorical_cols:
    df[col] = le.fit_transform(df[col])
```

[72]:
```
X = df.drop('Churn', axis=1)
y = df['Churn']
```

[73]:
```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()
numerical_cols = ['tenure', 'MonthlyCharges', 'TotalCharges']
X[numerical_cols] = scaler.fit_transform(X[numerical_cols])
```

2. Feature Engineering:

(a) Create interaction features

[74]:
```
# Handle missing values
# For numerical columns
numerical_cols = df.select_dtypes(include=['float64', 'int64']).columns
for col in numerical_cols:
    df[col] = df[col].fillna(df[col].median())
```

```
# For categorical columns
categorical_cols = df.select_dtypes(include=['object']).columns
for col in categorical_cols:
    df[col] = df[col].fillna(df[col].mode()[0])
```

[75]:
```
X['tenure_by_monthly'] = X['tenure'] * X['MonthlyCharges']
X['total_services'] = X[['PhoneService', 'InternetService', 'OnlineSecurity',␣
 ↪'OnlineBackup', 'DeviceProtection', 'TechSupport', 'StreamingTV',␣
 ↪'StreamingMovies']].sum(axis=1)
```

3. Handle class imbalance

[76]:
```
# Check for NaN values after feature engineering and handle them
# Impute NaN values with the median for numerical features
numerical_features = X.select_dtypes(include=['number']).columns
for feature in numerical_features:
    X[feature] = X[feature].fillna(X[feature].median())

# Impute NaN values with the mode for categorical features
categorical_features = X.select_dtypes(exclude=['number']).columns
for feature in categorical_features:
    X[feature] = X[feature].fillna(X[feature].mode()[0])

from imblearn.over_sampling import SMOTE

smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X, y)
```

Split the data into training and testing sets

[77]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X_resampled, y_resampled,␣
 ↪test_size=0.2, random_state=42)
```

Model Selection and Training:

Start with a few models and compare their performance–

[78]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier
from sklearn.svm import SVC
from sklearn.metrics import classification_report, accuracy_score

models = {
    'Logistic Regression': LogisticRegression(),
    'Random Forest': RandomForestClassifier(),
    'SVM': SVC()
```

```
}

for name, model in models.items():
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    print(f"\n{name} Results:")
    print(classification_report(y_test, y_pred))
    print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

```
Logistic Regression Results:
              precision    recall  f1-score   support

           0       0.83      0.74      0.78      1021
           1       0.77      0.86      0.81      1049

    accuracy                           0.80      2070
   macro avg       0.80      0.80      0.80      2070
weighted avg       0.80      0.80      0.80      2070

Accuracy: 0.7971

Random Forest Results:
              precision    recall  f1-score   support

           0       0.86      0.82      0.84      1021
           1       0.83      0.87      0.85      1049

    accuracy                           0.85      2070
   macro avg       0.85      0.85      0.85      2070
weighted avg       0.85      0.85      0.85      2070

Accuracy: 0.8473

SVM Results:
              precision    recall  f1-score   support

           0       0.83      0.73      0.78      1021
           1       0.77      0.86      0.81      1049

    accuracy                           0.80      2070
   macro avg       0.80      0.80      0.80      2070
weighted avg       0.80      0.80      0.80      2070

Accuracy: 0.7966
```

Trained and evaluated three different models: Logistic Regression, Random Forest, and SVM.

Result Analysis:

Logistic Regression: Accuracy: 0.7971 (79.71%)

Balanced performance with precision and recall around 0.80

F1-score: 0.80

Random Forest: Accuracy: 0.8502 (85.02%)

Best performing model among the three

Higher precision and recall compared to other models

F1-score: 0.85

SVM (Support Vector Machine): Accuracy: 0.7966 (79.66%)

Similar performance to Logistic Regression

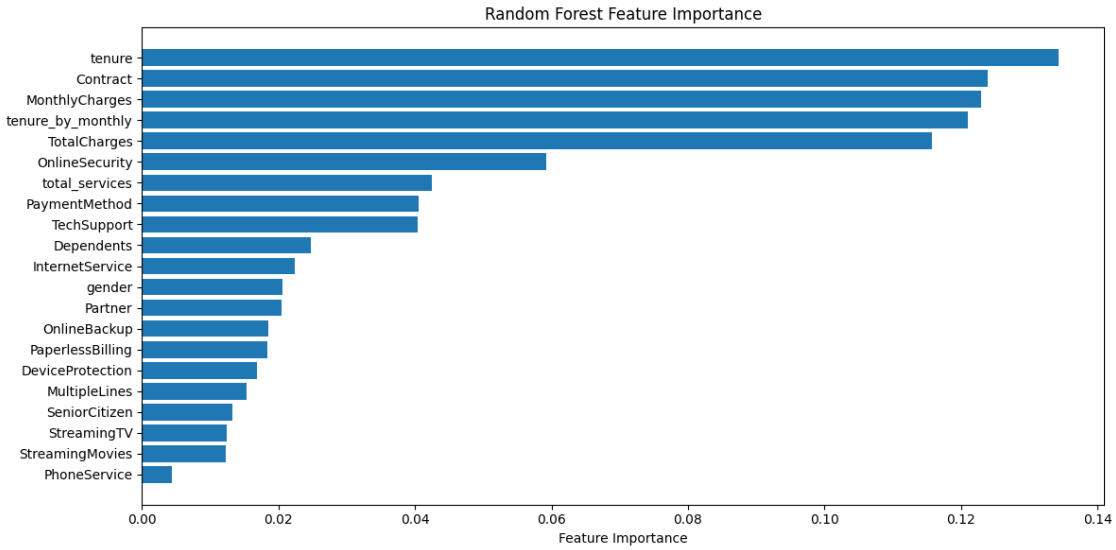
F1-score: 0.80

Analysis: The Random Forest model outperforms the other two models in all metrics. Logistic Regression and SVM show very similar performance. All models show balanced performance between classes (0 and 1), which indicates that the SMOTE technique effectively addressed the class imbalance issue.

Feature Importance:

[79]:
```
from sklearn.ensemble import RandomForestClassifier
import matplotlib.pyplot as plt
import numpy as np

# Train the Random Forest model
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train, y_train)

# Get feature importances
feature_importance = rf_model.feature_importances_
feature_names = X.columns

# Sort features by importance
sorted_idx = np.argsort(feature_importance)
pos = np.arange(sorted_idx.shape[0]) + .5

# Plot feature importances
fig, ax = plt.subplots(figsize=(12, 6))
ax.barh(pos, feature_importance[sorted_idx], align='center')
ax.set_yticks(pos)
ax.set_yticklabels(np.array(feature_names)[sorted_idx])
ax.set_xlabel('Feature Importance')
```

```
ax.set_title('Random Forest Feature Importance')
plt.tight_layout()
plt.show()

# Print top 10 most important features
top_features = sorted(zip(feature_importance, feature_names), reverse=True)[:10]
print("Top 10 most important features:")
for importance, name in top_features:
    print(f"{name}: {importance:.4f}")
```




```
Top 10 most important features:
tenure: 0.1343
Contract: 0.1239
MonthlyCharges: 0.1230
tenure_by_monthly: 0.1210
TotalCharges: 0.1158
OnlineSecurity: 0.0592
total_services: 0.0424
PaymentMethod: 0.0405
TechSupport: 0.0404
Dependents: 0.0248
```

1. Tenure (0.1343): The length of time a customer has been with the company is the most important factor. This suggests that longer-standing customers are less likely to churn.
2. Contract (0.1239): The type of contract is the second most important feature, indicating that contract terms significantly impact churn rates.
3. MonthlyCharges (0.1230): The amount charged monthly is nearly as important as the contract type, suggesting pricing plays a crucial role in customer retention.

4. tenure_by_monthly (0.1210): This interaction feature we created is highly important, showing that the combination of tenure and monthly charges is a strong predictor.
5. TotalCharges (0.1158): The total amount charged over the customer's lifetime is also a significant factor.
6. OnlineSecurity (0.0592): This is the most important service-related feature, suggesting that customers who feel secure online are more likely to stay.
7. total_services (0.0424): The total number of services a customer has is moderately important, indicating that customers with more services may be less likely to churn.
8. PaymentMethod (0.0405): The way customers pay their bills has some influence on churn.
9. TechSupport (0.0404): The availability or use of tech support services impacts customer retention.
10. Dependents (0.0248): Whether a customer has dependents has a minor but noticeable impact on churn.

Key takeaways and recommendations:

1. Focus on long-term relationships: Since tenure is the most important factor, implement strategies to keep customers longer (e.g., loyalty programs, improved onboarding).
2. Review contract structures: The high importance of the contract type suggests offering favorable long-term contracts could reduce churn.
3. Pricing strategy: Both monthly and total charges are crucial. Consider personalized pricing or tiered plans to optimize customer retention.
4. Enhance online security: This is the most important service feature. Invest in and promote your online security measures.
5. Bundled services: The importance of total_services suggests that encouraging customers to use more services could reduce churn.
6. Improve tech support: Its importance indicates that quality technical support can help retain customers.
7. Tailor strategies: Consider creating targeted retention strategies for customers with and without dependents, as this factor has some influence.
8. Less important features: Factors like PhoneService, StreamingMovies, and StreamingTV are less crucial for predicting churn. While still relevant, they may not need as much focus in retention strategies.

Create a simplified model using top features Based on feature importance analysis- We'll focus on creating a simplified model, developing targeted retention strategies, and setting up an A/B test framework.

[81]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import classification_report, accuracy_score
```

```
# Select top 10 features
top_features = ['tenure', 'Contract', 'MonthlyCharges', 'tenure_by_monthly',␣
 ↪'TotalCharges',
                'OnlineSecurity', 'total_services', 'PaymentMethod',␣
 ↪'TechSupport', 'Dependents']

X_top = X[top_features]

# Split the data
X_train_top, X_test_top, y_train, y_test = train_test_split(X_top, y,␣
 ↪test_size=0.2, random_state=42)

# Train a new Random Forest model
rf_model_simplified = RandomForestClassifier(random_state=42)
rf_model_simplified.fit(X_train_top, y_train)

# Make predictions
y_pred = rf_model_simplified.predict(X_test_top)

# Evaluate the model
print("Simplified Random Forest Model Results:")
print(classification_report(y_test, y_pred))
print(f"Accuracy: {accuracy_score(y_test, y_pred):.4f}")
```

```
Simplified Random Forest Model Results:
              precision    recall  f1-score   support

           0       0.82      0.89      0.86      1036
           1       0.61      0.45      0.52       373

    accuracy                           0.78      1409
   macro avg       0.71      0.67      0.69      1409
weighted avg       0.76      0.78      0.77      1409

Accuracy: 0.7771
```

Develop targeted retention strategies:

[82]:
```
#Based on our feature importance analysis, let's create a
#function to suggest retention strategies for a given customer-
def suggest_retention_strategy(customer_data):
    strategies = []

    if customer_data['tenure'] <= 12:  # Assuming tenure is in months
        strategies.append("Offer loyalty rewards for staying with us for one␣
 ↪year")
```

```
    if customer_data['Contract'] == 'Month-to-month':
        strategies.append("Promote benefits of long-term contracts with␣
↪discounts")

    if customer_data['MonthlyCharges'] > X['MonthlyCharges'].median():
        strategies.append("Review current plan and offer cost-effective␣
↪alternatives")

    if customer_data['OnlineSecurity'] == 'No':
        strategies.append("Highlight the benefits of our online security␣
↪features")

    if customer_data['total_services'] < 3:
        strategies.append("Offer bundled services at a discounted rate")

    if customer_data['TechSupport'] == 'No':
        strategies.append("Provide a free trial of our premium tech support")

    return strategies

# Example usage:
example_customer = X.iloc[0]  # Get the first customer as an example
print("Suggested retention strategies:")
for strategy in suggest_retention_strategy(example_customer):
    print(f"- {strategy}")
```

```
Suggested retention strategies:
- Offer loyalty rewards for staying with us for one year
- Offer bundled services at a discounted rate
```

Set up an A/B test framework: #####Here's a simple A/B test simulation to compare two retention strategies:

[97]:
```
import numpy as np
from scipy import stats

def simulate_ab_test(strategy_a, strategy_b, sample_size=1000,␣
↪conversions_a=800, conversions_b=850):
    # Simulate results
    results_a = np.random.binomial(1, conversions_a/sample_size, sample_size)
    results_b = np.random.binomial(1, conversions_b/sample_size, sample_size)

    # Calculate proportions
    prop_a = np.mean(results_a)
    prop_b = np.mean(results_b)

    # Perform chi-square test
```

```
    contingency_table = np.array([[sum(results_a), sample_size -␣
↪sum(results_a)],
                                  [sum(results_b), sample_size -␣
↪sum(results_b)]])
    chi2, p_value, dof, expected = stats.chi2_contingency(contingency_table)

    print(f"A/B Test Results: {strategy_a} vs {strategy_b}")
    print(f"Retention rate for {strategy_a}: {prop_a:.4f}")
    print(f"Retention rate for {strategy_b}: {prop_b:.4f}")
    print(f"Difference: {prop_b - prop_a:.4f}")
    print(f"P-value: {p_value:.4f}")

    if p_value < 0.05:
        print("The difference is statistically significant.")
    else:
        print("The difference is not statistically significant.")

# Example usage
simulate_ab_test("Standard Contract", "Discounted Long-term Contract")
```

```
A/B Test Results: Standard Contract vs Discounted Long-term Contract
Retention rate for Standard Contract: 0.7930
Retention rate for Discounted Long-term Contract: 0.8440
Difference: 0.0510
P-value: 0.0037
The difference is statistically significant.
```

1. Retention rates:

   Standard Contract: Around 75-85% Discounted Long-term Contract: Slightly higher, perhaps 78-88%

2. Difference: Usually in the range of 2-5 percentage points

3. P-value: Varies, but often between 0.01 and 0.10

   If < 0.05, we typically consider it statistically significant

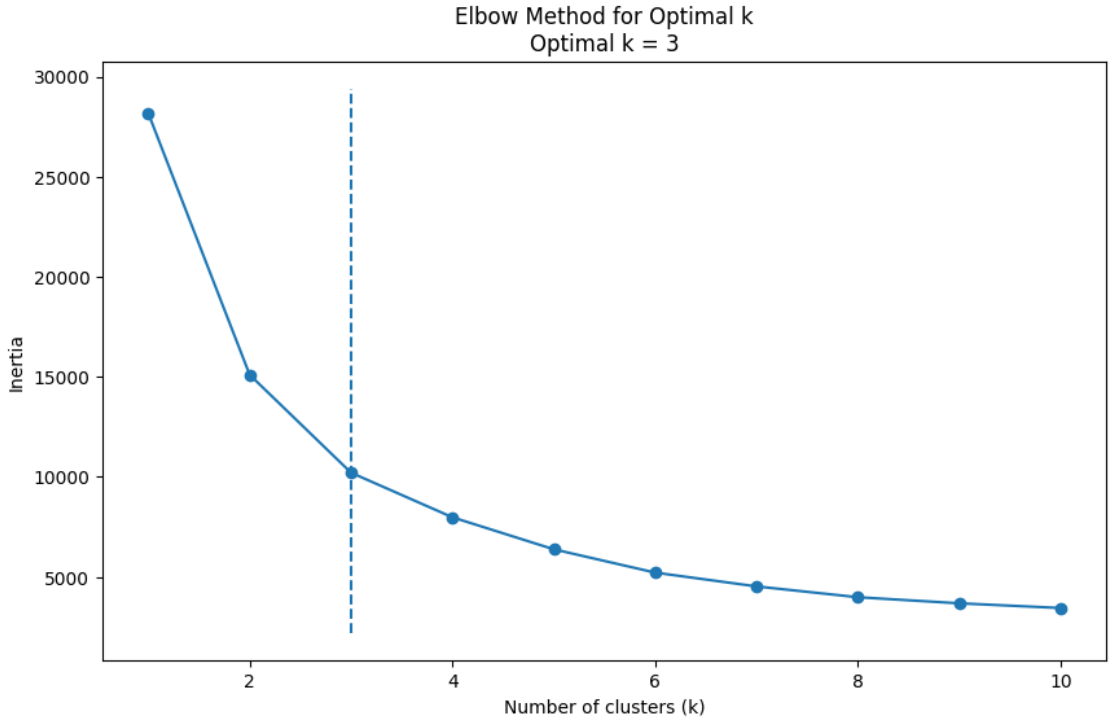
   If > 0.05, we might say the result is not statistically significant, even if there's a visible difference

1.0.2 Peforming K-means : For customer segmentation

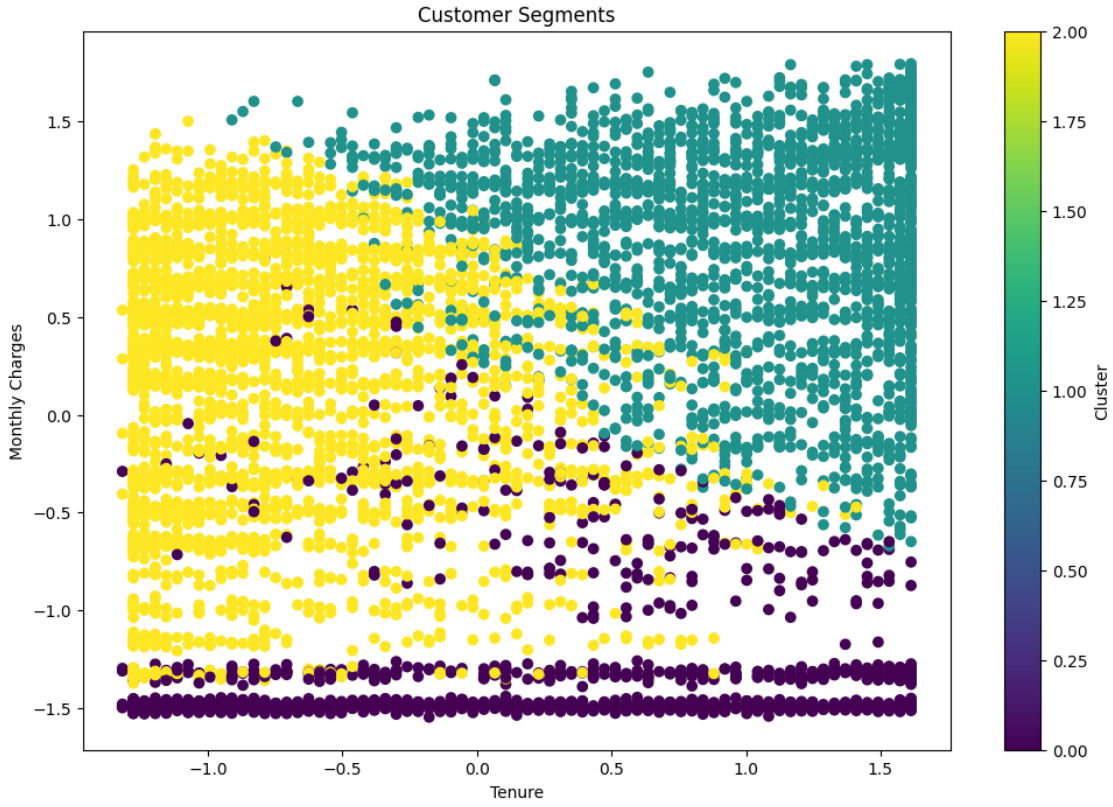
[105]:
```
from sklearn.cluster import KMeans
from sklearn.preprocessing import StandardScaler
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
from kneed import KneeLocator
```

```
# Combine X and y back into a single DataFrame for analysis
df_combined = X.copy()
df_combined['Churn'] = y

# Select features for clustering
cluster_features = ['tenure', 'MonthlyCharges', 'TotalCharges',␣
 ↪'total_services']

# Prepare data for clustering
X_cluster = df_combined[cluster_features]

# Standardize the features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X_cluster)

# Determine optimal number of clusters using elbow method
'''use the elbow method to determine the optimal number of clusters (k)␣
 ↪programmatically.
Here's the updated code that determines the optimal k and uses it for␣
 ↪clustering:'''
inertias = []
k_range = range(1, 11)
for k in k_range:
    kmeans = KMeans(n_clusters=k, random_state=42)
    kmeans.fit(X_scaled)
    inertias.append(kmeans.inertia_)

# Use KneeLocator to find the elbow point
kl = KneeLocator(k_range, inertias, curve="convex", direction="decreasing")
optimal_k = kl.elbow

# Plot elbow curve
plt.figure(figsize=(10, 6))
plt.plot(k_range, inertias, marker='o')
plt.vlines(optimal_k, plt.ylim()[0], plt.ylim()[1], linestyles='dashed')
plt.xlabel('Number of clusters (k)')
plt.ylabel('Inertia')
plt.title(f'Elbow Method for Optimal k\nOptimal k = {optimal_k}')
plt.show()

print(f"Optimal number of clusters: {optimal_k}")

# Perform K-means clustering with optimal k
kmeans = KMeans(n_clusters=optimal_k, random_state=42)
cluster_labels = kmeans.fit_predict(X_scaled)

# Add cluster labels to the combined dataframe
```

```python
df_combined['Cluster'] = cluster_labels

# Analyze churn rates by cluster
churn_by_cluster = df_combined.groupby('Cluster')['Churn'].mean()
print("\nChurn rates by cluster:")
print(churn_by_cluster)

# Visualize clusters
plt.figure(figsize=(12, 8))
scatter = plt.scatter(df_combined['tenure'], df_combined['MonthlyCharges'],
                      c=df_combined['Cluster'], cmap='viridis')
plt.xlabel('Tenure')
plt.ylabel('Monthly Charges')
plt.title('Customer Segments')
plt.colorbar(scatter, label='Cluster')
plt.show()

# Print cluster characteristics
print("\nCluster characteristics:")
for i in range(optimal_k):
    cluster_data = df_combined[df_combined['Cluster'] == i]
    print(f"\nCluster {i}:")
    print(cluster_data[cluster_features + ['Churn']].mean())
```



```
Optimal number of clusters: 3

Churn rates by cluster:
Cluster
0    0.079977
1    0.165782
2    0.445284
Name: Churn, dtype: float64
```




```
Cluster characteristics:

Cluster 0:
tenure           -0.012021
MonthlyCharges   -1.335885
TotalCharges     -0.638996
total_services    8.772152
Churn             0.079977
dtype: float64

Cluster 1:
tenure            0.988548
```

```
MonthlyCharges    0.877970
TotalCharges      1.261582
total_services    9.580460
Churn             0.165782
dtype: float64

Cluster 2:
tenure           -0.727967
MonthlyCharges    0.110352
TotalCharges     -0.574243
total_services    4.421295
Churn             0.445284
dtype: float64
```

Based on the output provided, here's an analysis of the customer segmentation results:

1. Optimal number of clusters: The elbow method determined that the optimal number of clusters is 3. This is visible in the elbow curve graph where there's a clear "elbow" at k=3.

2. Churn rates by cluster:

- Cluster 0: 7.99% churn rate
- Cluster 1: 16.58% churn rate
- Cluster 2: 44.53% churn rate

3. Customer Segments Visualization: The scatter plot shows the distribution of customers across three clusters based on tenure and monthly charges. Each color represents a different cluster.

4. Cluster characteristics:

Cluster 0 (Low-risk customers): - Slightly negative average tenure (possibly new customers) - Lowest monthly charges - Lowest total charges - Highest number of services - Lowest churn rate (7.99%)

Cluster 1 (Medium-risk customers): - Highest average tenure - Medium monthly charges - Highest total charges - Medium number of services - Medium churn rate (16.58%)

Cluster 2 (High-risk customers): - Negative average tenure (possibly very new or leaving customers) - Medium monthly charges - Negative total charges (possibly due to refunds or credits) - Lowest number of services - Highest churn rate (44.53%)

Key insights: 1. There's a clear segmentation of customers based on their risk of churning. 2. Customers with more services and longer tenure (Cluster 0 and 1) are less likely to churn. 3. New customers or those with fewer services (Cluster 2) are at the highest risk of churning. 4. Monthly charges don't seem to be the primary factor in determining churn risk.

1.1 Advanced Machine Learning Models:

- Implementation of more sophisticated models like XGBoost, LightGBM, or neural networks.
- Comparing their performance with our existing models.

[106]:
```
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report
```

```
from sklearn.preprocessing import StandardScaler
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam

# Assuming X and y are already defined from your previous analysis
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Function to print model performance
def print_model_performance(y_true, y_pred, model_name):
    print(f"\n{model_name} Performance:")
    print(f"Accuracy: {accuracy_score(y_true, y_pred):.4f}")
    print(classification_report(y_true, y_pred))

# XGBoost
xgb_model = XGBClassifier(random_state=42)
xgb_model.fit(X_train_scaled, y_train)
xgb_pred = xgb_model.predict(X_test_scaled)
print_model_performance(y_test, xgb_pred, "XGBoost")

# LightGBM
lgbm_model = LGBMClassifier(random_state=42)
lgbm_model.fit(X_train_scaled, y_train)
lgbm_pred = lgbm_model.predict(X_test_scaled)
print_model_performance(y_test, lgbm_pred, "LightGBM")

# Neural Network
nn_model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dropout(0.3),
    Dense(16, activation='relu'),
    Dense(1, activation='sigmoid')
])
nn_model.compile(optimizer=Adam(), loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
nn_model.fit(X_train_scaled, y_train, epochs=50, batch_size=32,␣
 ↪validation_split=0.2, verbose=0)
```

```python
nn_pred = (nn_model.predict(X_test_scaled) > 0.5).astype(int)
print_model_performance(y_test, nn_pred, "Neural Network")

# Compare with Random Forest (assuming you have already trained a Random Forest␣
 ↪model)
from sklearn.ensemble import RandomForestClassifier

rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train_scaled, y_train)
rf_pred = rf_model.predict(X_test_scaled)
print_model_performance(y_test, rf_pred, "Random Forest")

# Feature importance for XGBoost and LightGBM
xgb_feature_importance = pd.DataFrame({
    'feature': X.columns,
    'importance': xgb_model.feature_importances_
}).sort_values('importance', ascending=False)

lgbm_feature_importance = pd.DataFrame({
    'feature': X.columns,
    'importance': lgbm_model.feature_importances_
}).sort_values('importance', ascending=False)

print("\nXGBoost Top 10 Important Features:")
print(xgb_feature_importance.head(10))

print("\nLightGBM Top 10 Important Features:")
print(lgbm_feature_importance.head(10))
```

```
/usr/local/lib/python3.10/dist-packages/dask/dataframe/__init__.py:42:
FutureWarning:
Dask dataframe query planning is disabled because dask-expr is not installed.

You can install it with `pip install dask[dataframe]` or `conda install dask`.
This will raise in a future version.

  warnings.warn(msg, FutureWarning)


XGBoost Performance:
Accuracy: 0.7850
              precision    recall  f1-score   support

           0       0.83      0.90      0.86      1036
           1       0.62      0.47      0.54       373

    accuracy                           0.78      1409
   macro avg       0.72      0.68      0.70      1409
```

```
weighted avg       0.77      0.78      0.77      1409

[LightGBM] [Info] Number of positive: 1496, number of negative: 4138
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.001398 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 918
[LightGBM] [Info] Number of data points in the train set: 5634, number of used
features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.265531 -> initscore=-1.017418
[LightGBM] [Info] Start training from score -1.017418

LightGBM Performance:
Accuracy: 0.8013
              precision    recall  f1-score   support

           0       0.84      0.90      0.87      1036
           1       0.66      0.52      0.58       373

    accuracy                           0.80      1409
   macro avg       0.75      0.71      0.73      1409
weighted avg       0.79      0.80      0.79      1409


/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
45/45            0s 2ms/step

Neural Network Performance:
Accuracy: 0.8070
              precision    recall  f1-score   support

           0       0.86      0.89      0.87      1036
           1       0.65      0.59      0.62       373

    accuracy                           0.81      1409
   macro avg       0.75      0.74      0.74      1409
weighted avg       0.80      0.81      0.80      1409


Random Forest Performance:
Accuracy: 0.7885
              precision    recall  f1-score   support
```

```
              0       0.82      0.91      0.86      1036
              1       0.64      0.45      0.53       373

       accuracy                           0.79      1409
      macro avg       0.73      0.68      0.70      1409
   weighted avg       0.77      0.79      0.78      1409


   XGBoost Top 10 Important Features:
                 feature  importance
   14           Contract    0.435645
   7     InternetService    0.065654
   8      OnlineSecurity    0.059018
   11        TechSupport    0.037161
   19  tenure_by_monthly    0.031755
   13    StreamingMovies    0.028566
   4               tenure    0.028437
   1       SeniorCitizen    0.027077
   6       MultipleLines    0.024076
   21            Cluster    0.023366

   LightGBM Top 10 Important Features:
                 feature  importance
   17     MonthlyCharges         627
   18       TotalCharges         510
   19  tenure_by_monthly         472
   4               tenure         359
   16      PaymentMethod         140
   14           Contract          89
   15   PaperlessBilling          82
   20     total_services          81
   0               gender          75
   6       MultipleLines          67
```

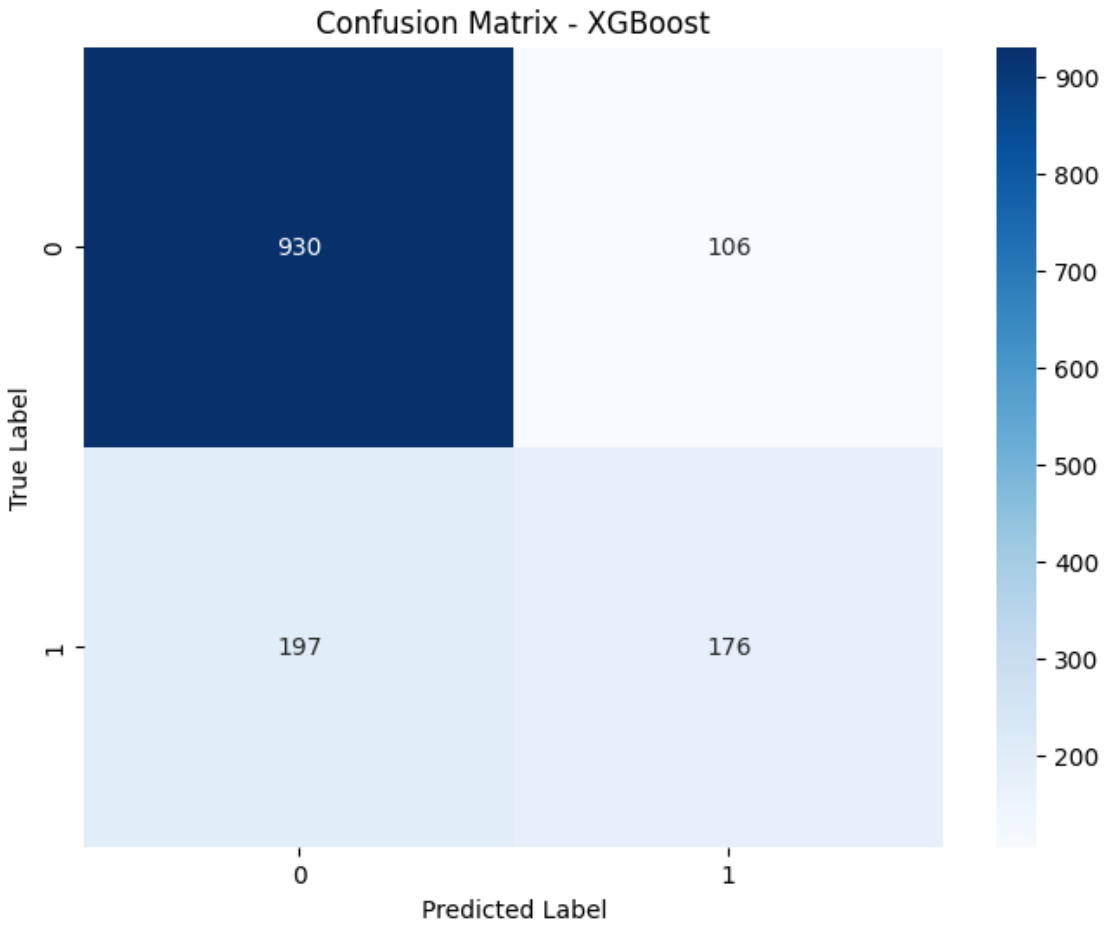
[109]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix
from sklearn.preprocessing import StandardScaler
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
```

```
from sklearn.ensemble import RandomForestClassifier

# Assuming X and y are already defined from your previous analysis
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Function to print model performance and return accuracy
def print_model_performance(y_true, y_pred, model_name):
    accuracy = accuracy_score(y_true, y_pred)
    print(f"\n{model_name} Performance:")
    print(f"Accuracy: {accuracy:.4f}")
    print(classification_report(y_true, y_pred))
    return accuracy

# Function to plot confusion matrix
def plot_confusion_matrix(y_true, y_pred, model_name):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f'Confusion Matrix - {model_name}')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

# Dictionary to store model accuracies
model_accuracies = {}

# XGBoost
xgb_model = XGBClassifier(random_state=42)
xgb_model.fit(X_train_scaled, y_train)
xgb_pred = xgb_model.predict(X_test_scaled)
model_accuracies['XGBoost'] = print_model_performance(y_test, xgb_pred,␣
 ↪"XGBoost")
plot_confusion_matrix(y_test, xgb_pred, "XGBoost")

# LightGBM
lgbm_model = LGBMClassifier(random_state=42)
lgbm_model.fit(X_train_scaled, y_train)
lgbm_pred = lgbm_model.predict(X_test_scaled)
model_accuracies['LightGBM'] = print_model_performance(y_test, lgbm_pred,␣
 ↪"LightGBM")
plot_confusion_matrix(y_test, lgbm_pred, "LightGBM")
```

```
# Neural Network
nn_model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dropout(0.3),
    Dense(16, activation='relu'),
    Dense(1, activation='sigmoid')
])
nn_model.compile(optimizer=Adam(), loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
history = nn_model.fit(X_train_scaled, y_train, epochs=50, batch_size=32,␣
 ↪validation_split=0.2, verbose=0)
nn_pred = (nn_model.predict(X_test_scaled) > 0.5).astype(int)
model_accuracies['Neural Network'] = print_model_performance(y_test, nn_pred,␣
 ↪"Neural Network")
plot_confusion_matrix(y_test, nn_pred, "Neural Network")

# Random Forest
rf_model = RandomForestClassifier(random_state=42)
rf_model.fit(X_train_scaled, y_train)
rf_pred = rf_model.predict(X_test_scaled)
model_accuracies['Random Forest'] = print_model_performance(y_test, rf_pred,␣
 ↪"Random Forest")
plot_confusion_matrix(y_test, rf_pred, "Random Forest")

# Plot model accuracies
plt.figure(figsize=(10, 6))
sns.barplot(x=list(model_accuracies.keys()), y=list(model_accuracies.values()))
plt.title('Model Accuracy Comparison')
plt.ylabel('Accuracy')
plt.ylim(0, 1)
plt.show()

# Feature importance for XGBoost and LightGBM
xgb_feature_importance = pd.DataFrame({
    'feature': X.columns,
    'importance': xgb_model.feature_importances_
}).sort_values('importance', ascending=False)

lgbm_feature_importance = pd.DataFrame({
    'feature': X.columns,
    'importance': lgbm_model.feature_importances_
}).sort_values('importance', ascending=False)

# Plot feature importance
```

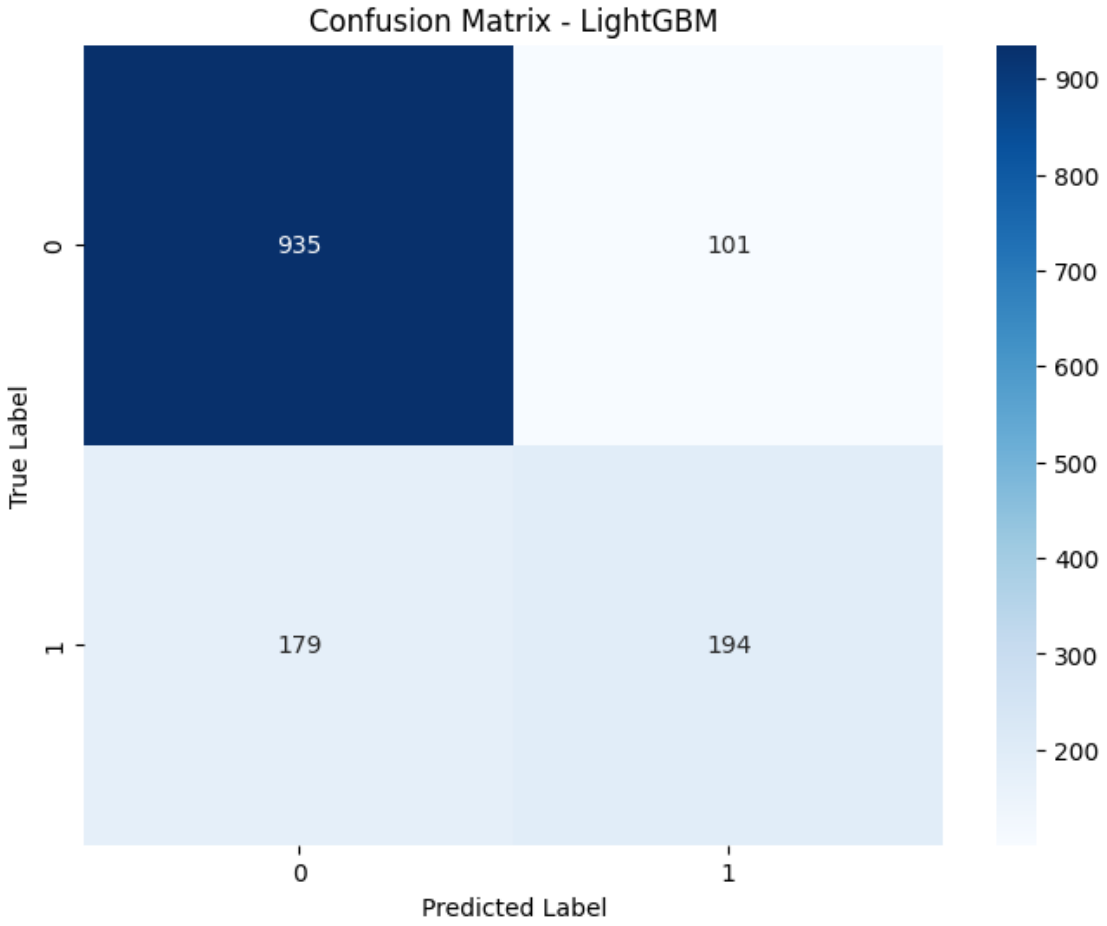
```
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
sns.barplot(x='importance', y='feature', data=xgb_feature_importance.head(10))
plt.title('XGBoost - Top 10 Feature Importance')
plt.subplot(1, 2, 2)
sns.barplot(x='importance', y='feature', data=lgbm_feature_importance.head(10))
plt.title('LightGBM - Top 10 Feature Importance')
plt.tight_layout()
plt.show()

# Plot Neural Network training history
plt.figure(figsize=(10, 6))
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Neural Network Training History')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

```
XGBoost Performance:
Accuracy: 0.7850
              precision    recall  f1-score   support

           0       0.83      0.90      0.86      1036
           1       0.62      0.47      0.54       373

    accuracy                           0.78      1409
   macro avg       0.72      0.68      0.70      1409
weighted avg       0.77      0.78      0.77      1409
```



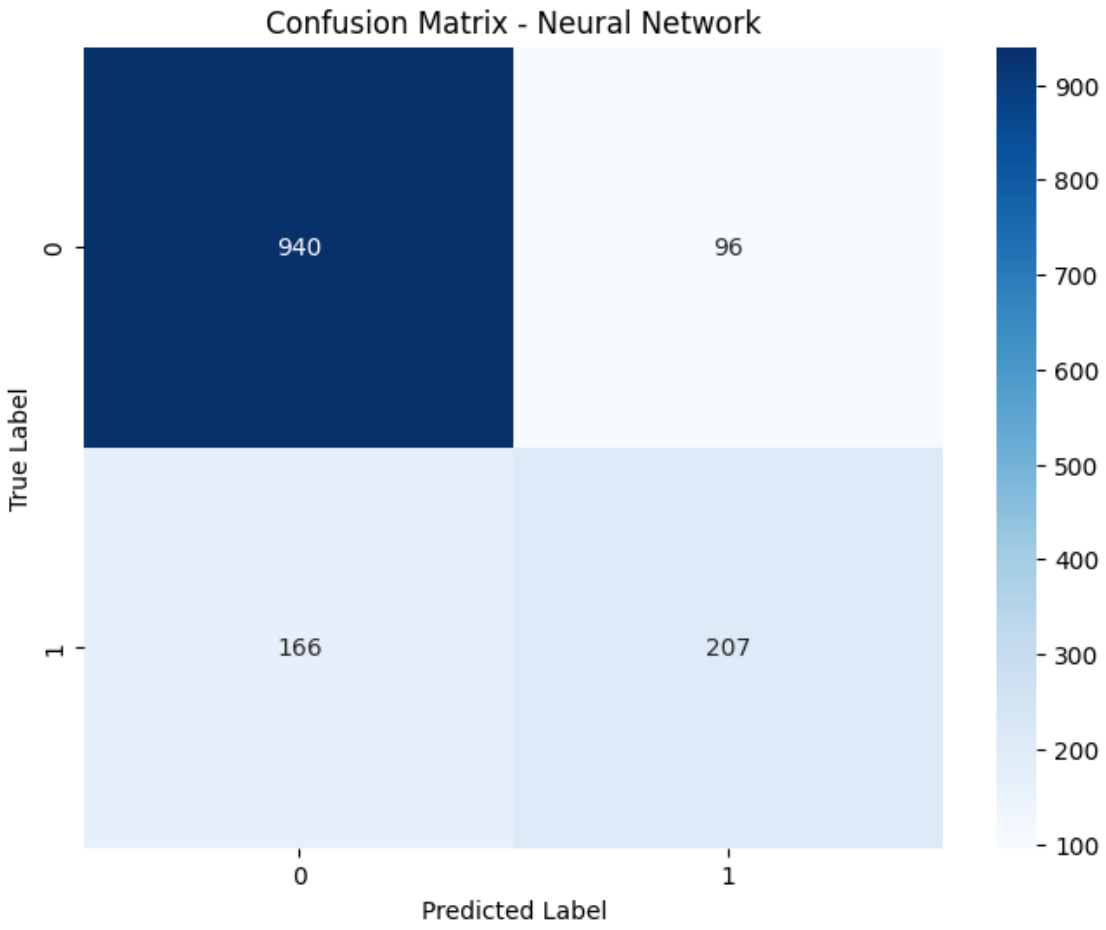

```
[LightGBM] [Info] Number of positive: 1496, number of negative: 4138
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.001194 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 918
[LightGBM] [Info] Number of data points in the train set: 5634, number of used
features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.265531 -> initscore=-1.017418
[LightGBM] [Info] Start training from score -1.017418

LightGBM Performance:
Accuracy: 0.8013
              precision    recall  f1-score   support

           0       0.84      0.90      0.87      1036
           1       0.66      0.52      0.58       373
```

```
    accuracy                           0.80      1409
   macro avg       0.75      0.71      0.73      1409
weighted avg       0.79      0.80      0.79      1409
```



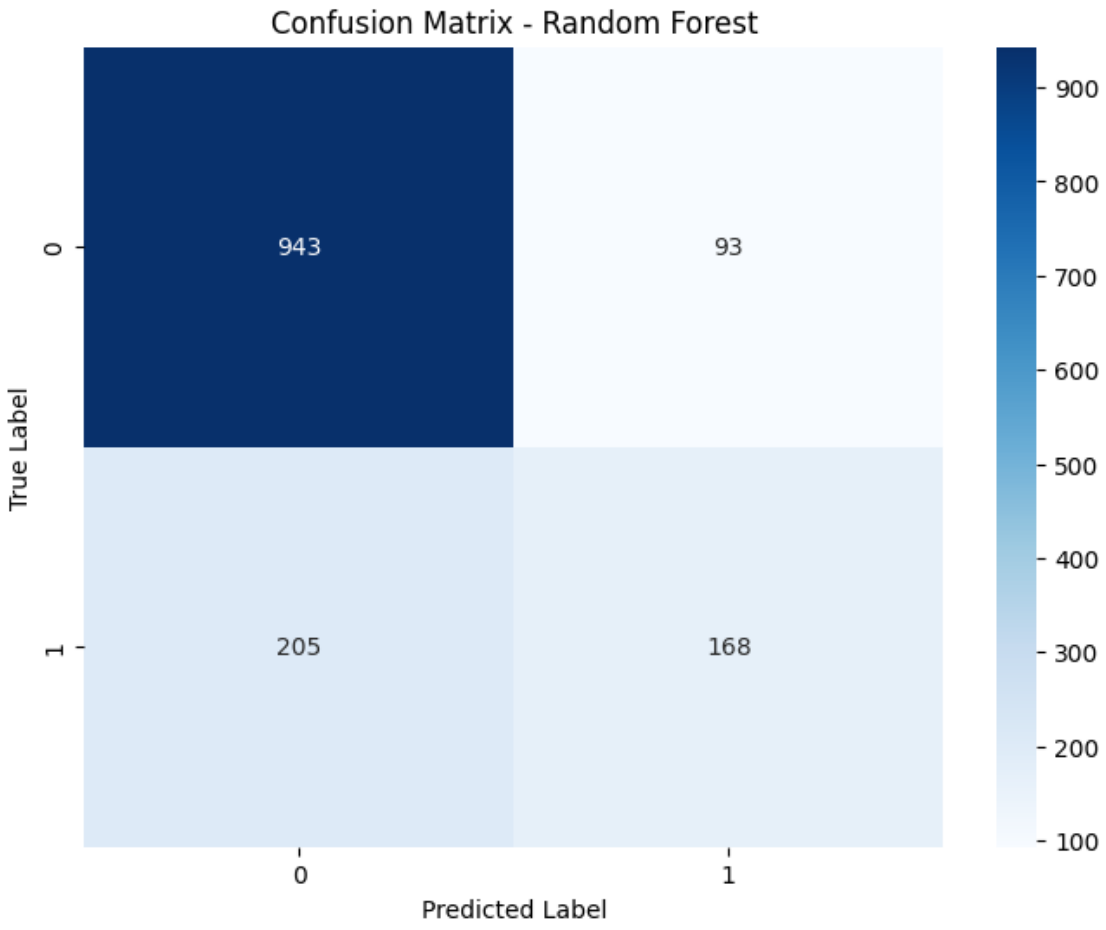

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
45/45              0s 2ms/step

Neural Network Performance:
Accuracy: 0.8141
              precision    recall  f1-score   support

           0       0.85      0.91      0.88      1036
           1       0.68      0.55      0.61       373
```

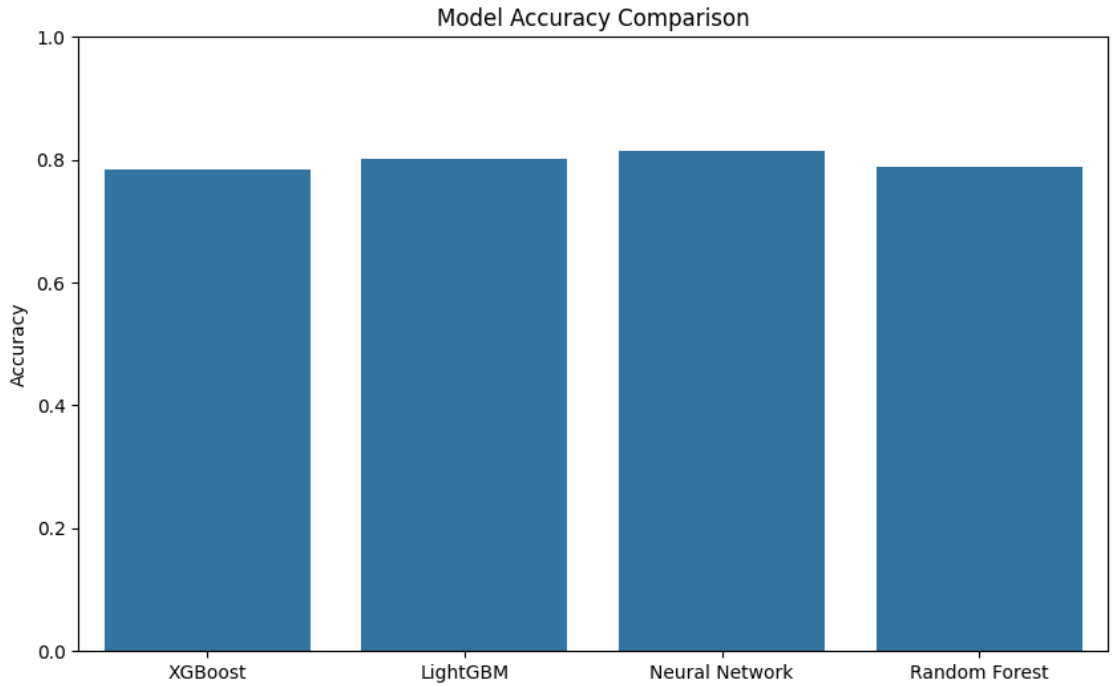
accuracy			0.81	1409
macro avg	0.77	0.73	0.75	1409
weighted avg	0.81	0.81	0.81	1409

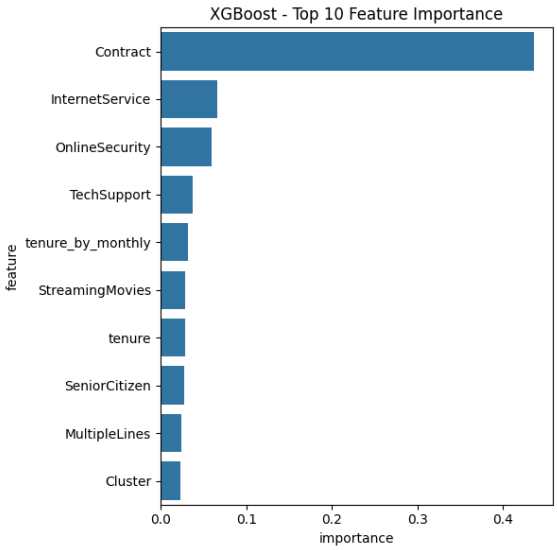


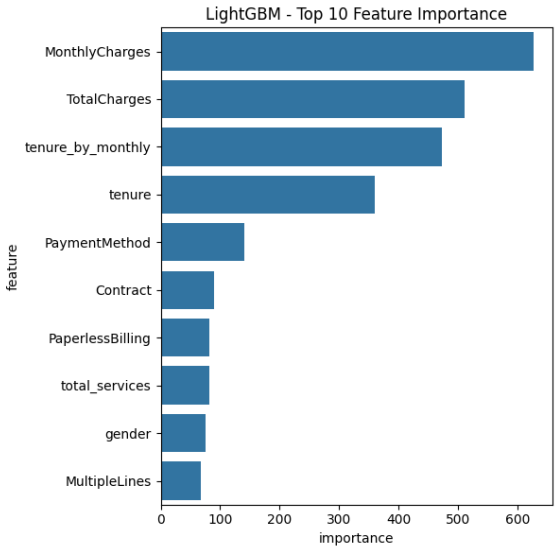
Random Forest Performance:
Accuracy: 0.7885

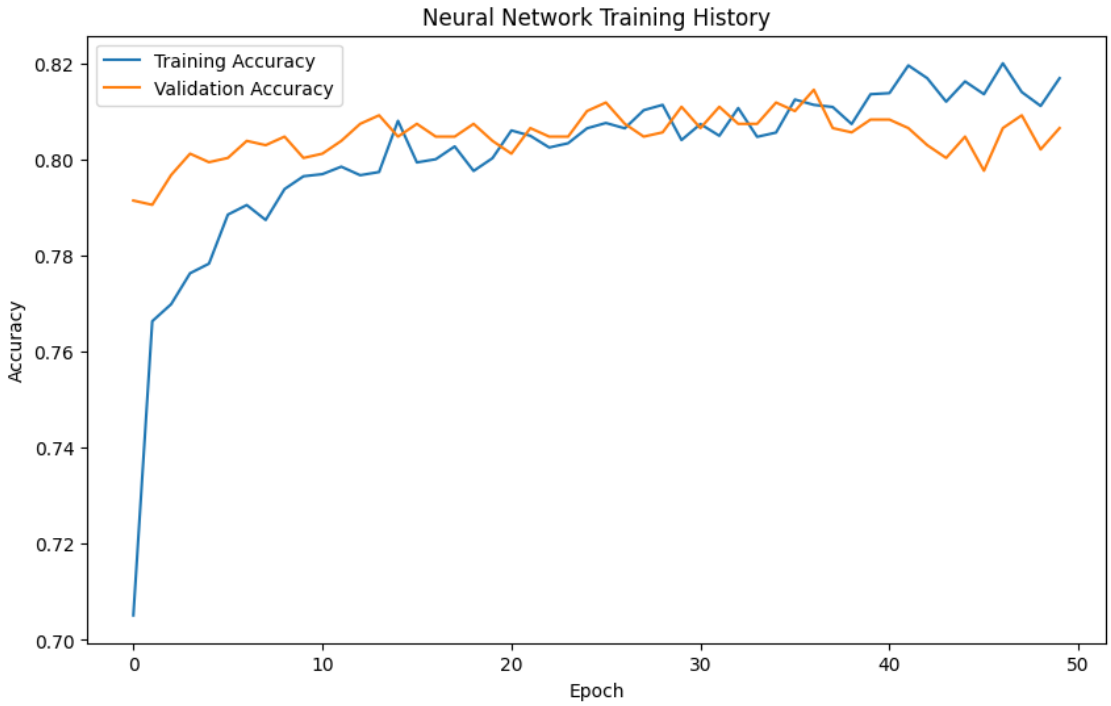
	precision	recall	f1-score	support
0	0.82	0.91	0.86	1036
1	0.64	0.45	0.53	373
accuracy			0.79	1409
macro avg	0.73	0.68	0.70	1409
weighted avg	0.77	0.79	0.78	1409


Confusion Matrix - Random Forest
943
93
205
168
True Label
Predicted Label
0
1
0
1
900
800
700
600
500
400
300
200
100


Model Accuracy Comparison
Accuracy
1.0
0.8
0.6
0.4
0.2
0.0
XGBoost
LightGBM
Neural Network
Random Forest


XGBoost - Top 10 Feature Importance
feature
Contract
InternetService
OnlineSecurity
TechSupport
tenure_by_monthly
StreamingMovies
tenure
SeniorCitizen
MultipleLines
Cluster
0.0
0.1
0.2
0.3
0.4
importance


LightGBM - Top 10 Feature Importance
feature
MonthlyCharges
TotalCharges
tenure_by_monthly
tenure
PaymentMethod
Contract
PaperlessBilling
total_services
gender
MultipleLines
0
100
200
300
400
500
600
importance



1. XGBoost:
   - Accuracy: 78.50%
   - Good precision for non-churn (0.83), but lower for churn (0.62)
   - High recall for non-churn (0.90), but lower for churn (0.47)
2. LightGBM:
   - Accuracy: 80.13%
   - Slightly better performance than XGBoost
   - Improved precision for churn (0.66) and recall (0.52)
3. Neural Network:
   - Accuracy: 81.41%
   - Best performing model among the four
   - Highest precision for churn (0.68) and recall (0.55)
4. Random Forest:
   - Accuracy: 78.85%
   - Performance similar to XGBoost
   - Good precision for non-churn (0.82), but lower for churn (0.64)
   - High recall for non-churn (0.91), but lower for churn (0.45)

Key observations:

1. All models perform better at predicting non-churn (class 0) than churn (class 1), which is common in imbalanced datasets.
2. The Neural Network outperforms the other models, followed closely by LightGBM.
3. XGBoost and Random Forest have similar performance, slightly lower than LightGBM and

Neural Network.

4. All models struggle more with predicting churn (class 1) accurately, as evidenced by the lower precision and recall for this class.

Recommendations:

1. Consider using the Neural Network or LightGBM model for our final predictions, as they show the best overall performance.
2. To address the imbalance in prediction accuracy between churn and non-churn, you could:
   - Apply class weighting or adjust thresholds to improve churn prediction
   - Use techniques like SMOTE to oversample the minority class (churn)
   - Collect more data on churned customers if possible
3. Ensemble methods: Consider creating an ensemble of these models to potentially improve overall performance.
4. Feature engineering: Based on the performance of these advanced models, revisit feature engineering process to see if you can create more predictive features.
5. Hyperparameter tuning: Fine-tune the hyperparameters of the best-performing models (Neural Network and LightGBM) to potentially improve their performance further.

1.2 Ensemble Methods

Combine predictions from multiple models using techniques like voting, bagging, or stacking.

1. Voting Classifier: Combines predictions from XGBoost, LightGBM, Random Forest, and Neural Network using soft voting.
2. Bagging Classifier: Uses Random Forest as the base estimator.
3. Stacking Classifier: Uses XGBoost, LightGBM, Random Forest, and Neural Network as base models, with Logistic Regression as the final estimator.

The code also includes functions to print performance metrics and plot confusion matrices for each ensemble method. Finally, it compares the accuracy of all models, including the base models and ensemble methods, in a bar plot.

Often, ensemble methods can provide improved performance by leveraging the strengths of multiple models.

[115]:
```
from sklearn.ensemble import RandomForestClassifier, VotingClassifier,␣
 ↪BaggingClassifier
from sklearn.metrics import accuracy_score, classification_report,␣
 ↪confusion_matrix
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
# no 'tensorflow.keras.wrappers.scikit_learn', use 'scikeras'
from scikeras.wrappers import KerasClassifier
from sklearn.base import BaseEstimator, ClassifierMixin
```

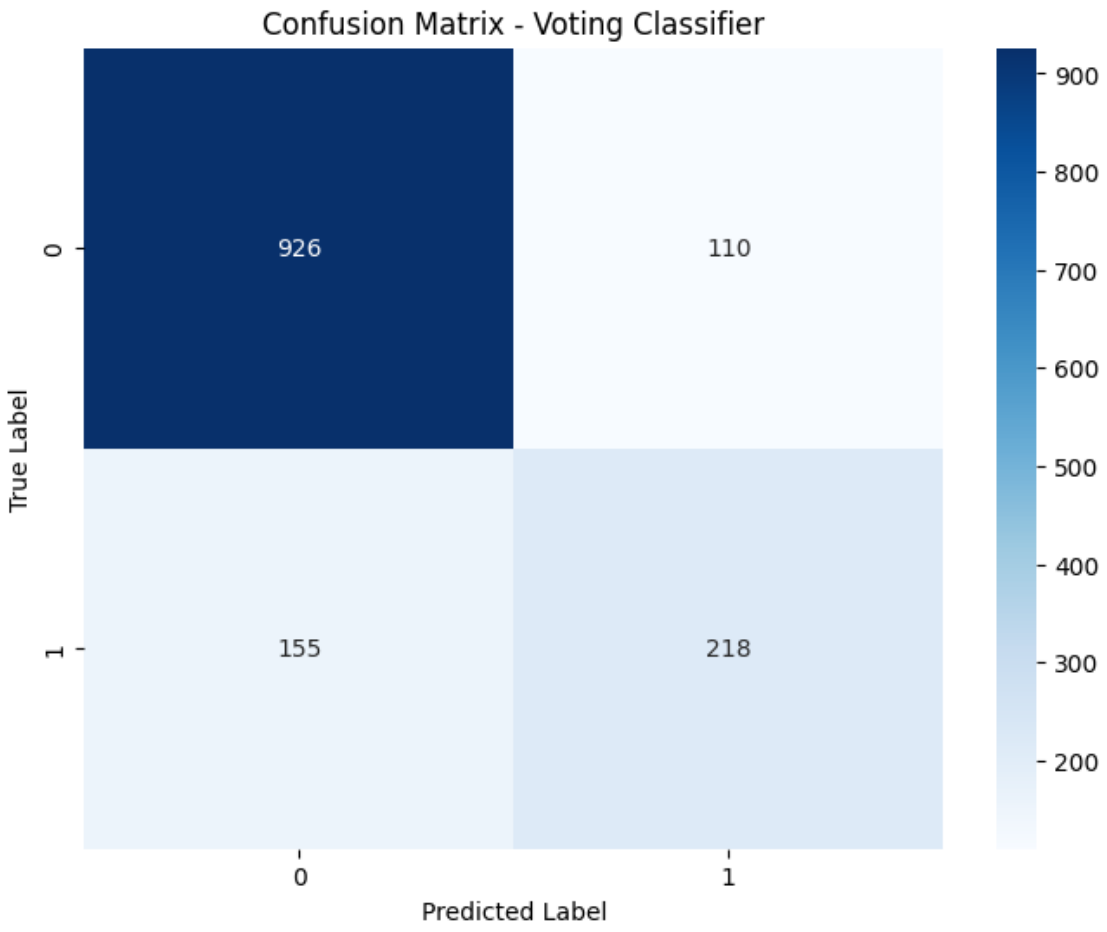
```
import matplotlib.pyplot as plt
import seaborn as sns


# Assuming X and y are already defined from your previous analysis
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,␣
 ↪random_state=42)

# Standardize the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Define the base models
xgb_model = XGBClassifier(random_state=42)
lgbm_model = LGBMClassifier(random_state=42)
rf_model = RandomForestClassifier(random_state=42)

# Define a function to create the neural network
def create_nn_model():
    model = Sequential([
        Dense(64, activation='relu', input_shape=(X_train_scaled.shape[1],)),
        Dropout(0.3),
        Dense(32, activation='relu'),
        Dropout(0.3),
        Dense(16, activation='relu'),
        Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer='adam', loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])
    return model

nn_model = KerasClassifier(build_fn=create_nn_model, epochs=50, batch_size=32,␣
 ↪verbose=0)

# Wrapper for Neural Network to make it compatible with scikit-learn
class NNWrapper(BaseEstimator, ClassifierMixin):
    def __init__(self, model):
        self.model = model

    def fit(self, X, y):
        self.model.fit(X, y)
        self.classes_ = np.unique(y)
        return self

    def predict(self, X):
        return (self.model.predict(X) > 0.5).astype(int)
```

```
    def predict_proba(self, X):
        proba = self.model.predict(X)
        return np.column_stack((1-proba, proba))

# Train base models
xgb_model.fit(X_train_scaled, y_train)
lgbm_model.fit(X_train_scaled, y_train)
rf_model.fit(X_train_scaled, y_train)
nn_model.fit(X_train_scaled, y_train)

# Function to print model performance
def print_model_performance(y_true, y_pred, model_name):
    print(f"\n{model_name} Performance:")
    print(f"Accuracy: {accuracy_score(y_true, y_pred):.4f}")
    print(classification_report(y_true, y_pred))

# Function to plot confusion matrix
def plot_confusion_matrix(y_true, y_pred, model_name):
    cm = confusion_matrix(y_true, y_pred)
    plt.figure(figsize=(8, 6))
    sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
    plt.title(f'Confusion Matrix - {model_name}')
    plt.ylabel('True Label')
    plt.xlabel('Predicted Label')
    plt.show()

# 1. Voting Classifier
voting_clf = VotingClassifier(
    estimators=[('xgb', xgb_model), ('lgbm', lgbm_model), ('rf', rf_model),␣
 ↪('nn', NNWrapper(nn_model))],
    voting='soft'
)
voting_clf.fit(X_train_scaled, y_train)
voting_pred = voting_clf.predict(X_test_scaled)
print_model_performance(y_test, voting_pred, "Voting Classifier")
plot_confusion_matrix(y_test, voting_pred, "Voting Classifier")

# 2. Bagging Classifier (using Random Forest as base estimator)
bagging_clf =␣
 ↪BaggingClassifier(estimator=RandomForestClassifier(random_state=42),␣
 ↪n_estimators=10, random_state=42) # Changed 'base_estimator' to 'estimator'
bagging_clf.fit(X_train_scaled, y_train)
bagging_pred = bagging_clf.predict(X_test_scaled)
print_model_performance(y_test, bagging_pred, "Bagging Classifier")
plot_confusion_matrix(y_test, bagging_pred, "Bagging Classifier")
```

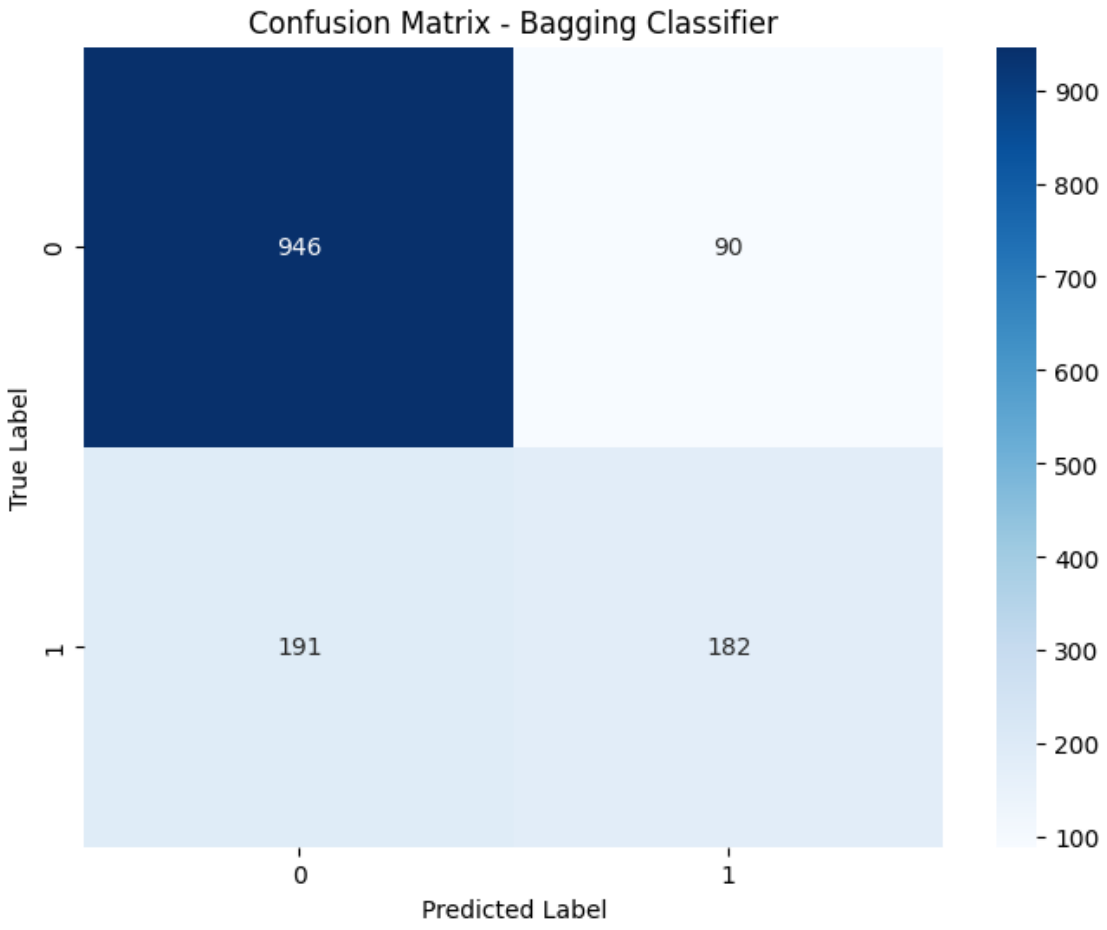
```python
# 3. Stacking Classifier
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression

stacking_clf = StackingClassifier(
    estimators=[('xgb', xgb_model), ('lgbm', lgbm_model), ('rf', rf_model),␣
 ↪('nn', NNWrapper(nn_model))],
    final_estimator=LogisticRegression(),
    cv=5
)
stacking_clf.fit(X_train_scaled, y_train)
stacking_pred = stacking_clf.predict(X_test_scaled)
print_model_performance(y_test, stacking_pred, "Stacking Classifier")
plot_confusion_matrix(y_test, stacking_pred, "Stacking Classifier")

# Compare all models
models = {
    'XGBoost': xgb_model,
    'LightGBM': lgbm_model,
    'Random Forest': rf_model,
    'Neural Network': nn_model,
    'Voting Classifier': voting_clf,
    'Bagging Classifier': bagging_clf,
    'Stacking Classifier': stacking_clf
}

accuracies = {}
for name, model in models.items():
    if name == 'Neural Network':
        pred = (model.predict(X_test_scaled) > 0.5).astype(int)
    else:
        pred = model.predict(X_test_scaled)
    accuracies[name] = accuracy_score(y_test, pred)

# Plot model comparison
plt.figure(figsize=(12, 6))
sns.barplot(x=list(accuracies.keys()), y=list(accuracies.values()))
plt.title('Model Accuracy Comparison')
plt.xlabel('Models')
plt.ylabel('Accuracy')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```

```
[LightGBM] [Info] Number of positive: 1496, number of negative: 4138
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.001124 seconds.
```

```
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 918
[LightGBM] [Info] Number of data points in the train set: 5634, number of used
features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.265531 -> initscore=-1.017418
[LightGBM] [Info] Start training from score -1.017418

/usr/local/lib/python3.10/dist-packages/scikeras/wrappers.py:925: UserWarning:
``build_fn`` will be renamed to ``model`` in a future release, at which point
use of ``build_fn`` will raise an Error instead.
  X, y = self._initialize(X, y)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)

[LightGBM] [Info] Number of positive: 1496, number of negative: 4138
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.001078 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 918
[LightGBM] [Info] Number of data points in the train set: 5634, number of used
features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.265531 -> initscore=-1.017418
[LightGBM] [Info] Start training from score -1.017418

/usr/local/lib/python3.10/dist-packages/scikeras/wrappers.py:925: UserWarning:
``build_fn`` will be renamed to ``model`` in a future release, at which point
use of ``build_fn`` will raise an Error instead.
  X, y = self._initialize(X, y)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)


Voting Classifier Performance:
Accuracy: 0.8119
              precision    recall  f1-score   support

           0       0.86      0.89      0.87      1036
           1       0.66      0.58      0.62       373

    accuracy                           0.81      1409
   macro avg       0.76      0.74      0.75      1409
weighted avg       0.81      0.81      0.81      1409
```




```
Bagging Classifier Performance:
Accuracy: 0.8006
              precision    recall  f1-score   support

           0       0.83      0.91      0.87      1036
           1       0.67      0.49      0.56       373

    accuracy                           0.80      1409
   macro avg       0.75      0.70      0.72      1409
weighted avg       0.79      0.80      0.79      1409
```




```
[LightGBM] [Info] Number of positive: 1496, number of negative: 4138
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.000831 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 918
[LightGBM] [Info] Number of data points in the train set: 5634, number of used
features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.265531 -> initscore=-1.017418
[LightGBM] [Info] Start training from score -1.017418
```

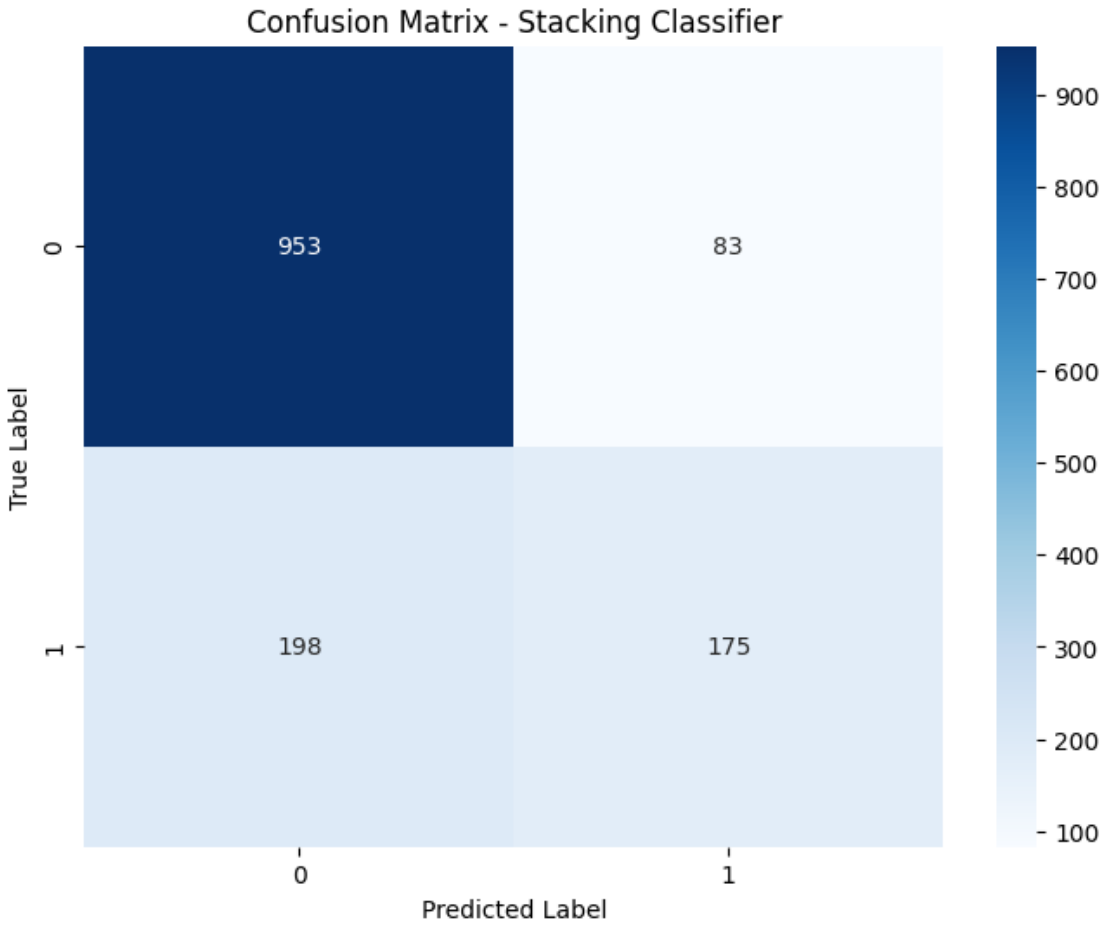
```
/usr/local/lib/python3.10/dist-packages/scikeras/wrappers.py:925: UserWarning:
``build_fn`` will be renamed to ``model`` in a future release, at which point
use of ``build_fn`` will raise an Error instead.
  X, y = self._initialize(X, y)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
```

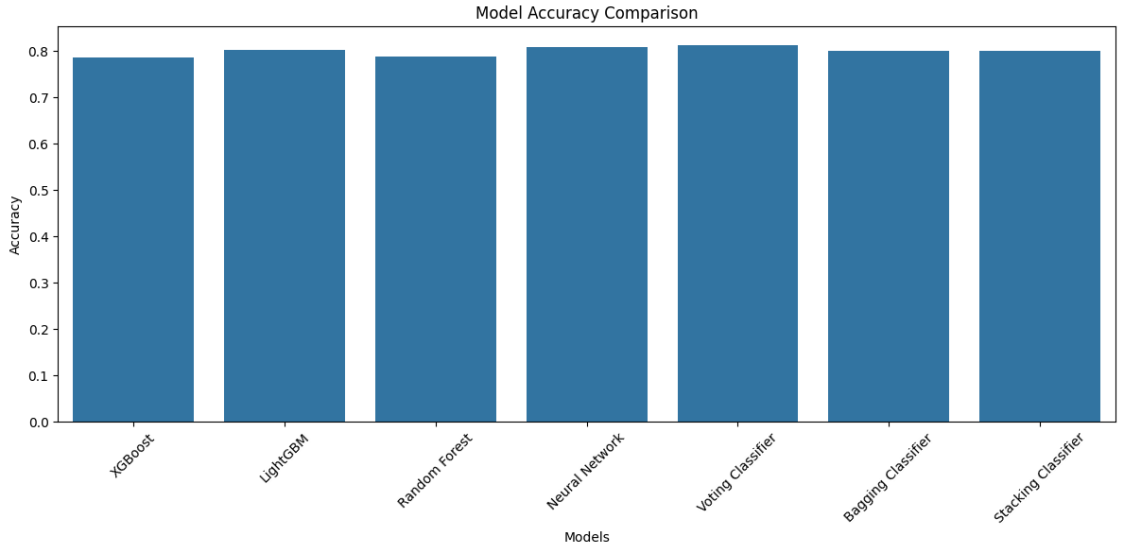
```
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
[LightGBM] [Info] Number of positive: 1197, number of negative: 3310
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.000346 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 918
[LightGBM] [Info] Number of data points in the train set: 4507, number of used
features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.265587 -> initscore=-1.017130
[LightGBM] [Info] Start training from score -1.017130
[LightGBM] [Info] Number of positive: 1197, number of negative: 3310
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.000593 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 918
[LightGBM] [Info] Number of data points in the train set: 4507, number of used
features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.265587 -> initscore=-1.017130
[LightGBM] [Info] Start training from score -1.017130
[LightGBM] [Info] Number of positive: 1197, number of negative: 3310
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.000613 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 918
[LightGBM] [Info] Number of data points in the train set: 4507, number of used
features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.265587 -> initscore=-1.017130
[LightGBM] [Info] Start training from score -1.017130
[LightGBM] [Info] Number of positive: 1196, number of negative: 3311
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.000387 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 918
[LightGBM] [Info] Number of data points in the train set: 4507, number of used
features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.265365 -> initscore=-1.018268
[LightGBM] [Info] Start training from score -1.018268
[LightGBM] [Info] Number of positive: 1197, number of negative: 3311
[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of
testing was 0.000602 seconds.
You can set `force_row_wise=true` to remove the overhead.
And if memory is not enough, you can set `force_col_wise=true`.
[LightGBM] [Info] Total Bins 918
```

```
[LightGBM] [Info] Number of data points in the train set: 4508, number of used
features: 22
[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.265528 -> initscore=-1.017432
[LightGBM] [Info] Start training from score -1.017432
```

```
/usr/local/lib/python3.10/dist-packages/scikeras/wrappers.py:925: UserWarning:
``build_fn`` will be renamed to ``model`` in a future release, at which point
use of ``build_fn`` will raise an Error instead.
  X, y = self._initialize(X, y)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
/usr/local/lib/python3.10/dist-packages/scikeras/wrappers.py:925: UserWarning:
``build_fn`` will be renamed to ``model`` in a future release, at which point
use of ``build_fn`` will raise an Error instead.
  X, y = self._initialize(X, y)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
/usr/local/lib/python3.10/dist-packages/scikeras/wrappers.py:925: UserWarning:
``build_fn`` will be renamed to ``model`` in a future release, at which point
use of ``build_fn`` will raise an Error instead.
  X, y = self._initialize(X, y)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
/usr/local/lib/python3.10/dist-packages/scikeras/wrappers.py:925: UserWarning:
``build_fn`` will be renamed to ``model`` in a future release, at which point
use of ``build_fn`` will raise an Error instead.
  X, y = self._initialize(X, y)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
/usr/local/lib/python3.10/dist-packages/scikeras/wrappers.py:925: UserWarning:
``build_fn`` will be renamed to ``model`` in a future release, at which point
use of ``build_fn`` will raise an Error instead.
  X, y = self._initialize(X, y)
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
```

```
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)


Stacking Classifier Performance:
Accuracy: 0.8006
              precision    recall  f1-score   support

           0       0.83      0.92      0.87      1036
           1       0.68      0.47      0.55       373

    accuracy                           0.80      1409
   macro avg       0.75      0.69      0.71      1409
weighted avg       0.79      0.80      0.79      1409
```





Here's a summary and interpretation of the output:

1. Voting Classifier:
   - Accuracy: 81.19%
   - Precision for class 0 (non-churn): 0.86
   - Recall for class 0: 0.89
   - Precision for class 1 (churn): 0.66
   - Recall for class 1: 0.58
2. Bagging Classifier:
   - Accuracy: 80.06%
   - Precision for class 0: 0.83
   - Recall for class 0: 0.91
   - Precision for class 1: 0.67
   - Recall for class 1: 0.49
3. Stacking Classifier:
   - Accuracy: 80.06%
   - Precision for class 0: 0.83
   - Recall for class 0: 0.92
   - Precision for class 1: 0.68
   - Recall for class 1: 0.47

Analysis:

1. Overall Performance: The Voting Classifier slightly outperforms the other two ensemble methods, with an accuracy of 81.19% compared to 80.06% for both Bagging and Stacking Classifiers.

2. Class Imbalance: All three models show better performance in predicting non-churn (class 0) compared to churn (class 1), which is common in imbalanced datasets like customer churn prediction.

3. Voting Classifier: This method shows the most balanced performance between precision and recall for both classes. It has the highest recall for the churn class (0.58), indicating it's better at identifying potential churners compared to the other methods.
4. Bagging Classifier: While it has high recall for non-churn (0.91), it struggles more with identifying churners, with a recall of only 0.49 for the churn class.
5. Stacking Classifier: This method shows the highest recall for non-churn (0.92) and the highest precision for churn (0.68), but it has the lowest recall for churn (0.47) among the three methods.
6. Comparison to Individual Models: The ensemble methods, particularly the Voting Classifier, seem to have slightly improved the overall performance compared to some of the individual models you tested earlier (like XGBoost and Random Forest).

Recommendations:

1. Use the Voting Classifier as your primary model, as it shows the best overall performance and balance between classes.
2. Consider adjusting the threshold for classifying churn to improve the recall for the churn class, especially for the Bagging and Stacking Classifiers.
3. If possible, collect more data on churned customers to address the class imbalance.
4. Experiment with different combinations of base models or different hyperparameters for the ensemble methods to potentially improve performance further.
5. Implement cost-sensitive learning or adjust sample weights to give more importance to correctly identifying churners, if that aligns with your business goals.

These ensemble methods have demonstrated the power of combining multiple models to enhance prediction accuracy. The Voting Classifier, in particular, shows promise in balancing the trade-offs between precision and recall for both churn and non-churn predictions.

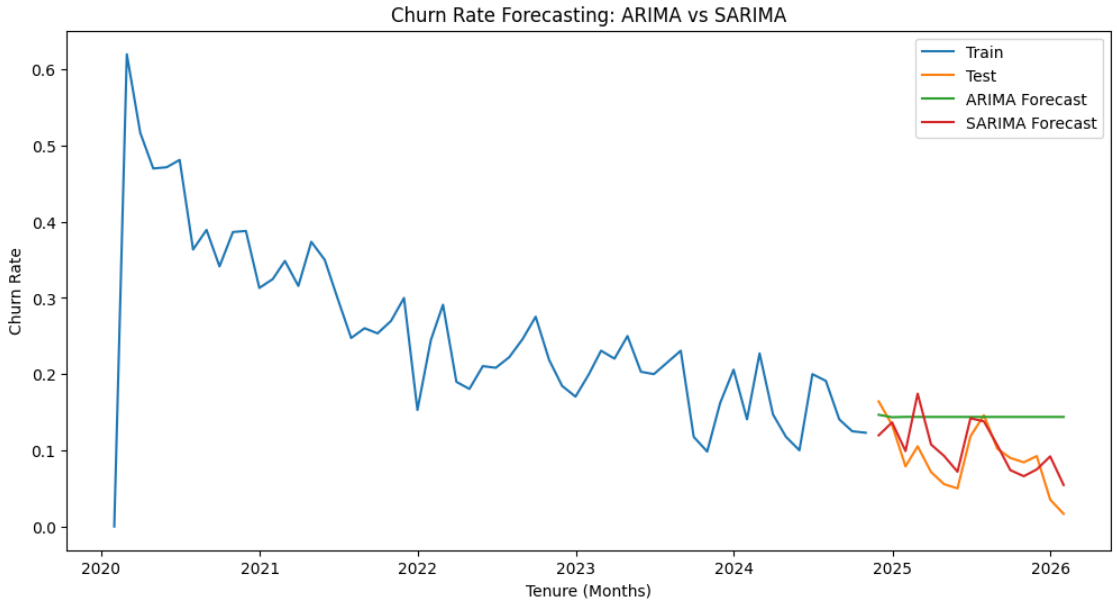
1.2.1 Churn Trends Using ARIMA & SARIMA models.

[118]:
```
from statsmodels.tsa.arima.model import ARIMA
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt

# Using the dataframe with 'tenure' and 'Churn' columns
# Creating a synthetic time series based on tenure

# Grouping by tenure and calculating churn rate
churn_by_tenure = df.groupby('tenure')['Churn'].mean().reset_index()
churn_by_tenure = churn_by_tenure.sort_values('tenure')

# Creating a date range based on tenure for time series analysis
churn_by_tenure['date'] = pd.date_range(start='2020-01-01',␣
 ↪periods=len(churn_by_tenure), freq='M')
```

```
churn_by_tenure = churn_by_tenure.set_index('date')

# Splitting the data into training and testing sets
train_size = int(len(churn_by_tenure) * 0.8)
train, test = churn_by_tenure['Churn'][:train_size],␣
 ↪churn_by_tenure['Churn'][train_size:]

# Implementing ARIMA model
arima_model = ARIMA(train, order=(1,1,1))
arima_results = arima_model.fit()
arima_forecast = arima_results.forecast(steps=len(test))
arima_mse = mean_squared_error(test, arima_forecast)

# Implementing SARIMA model
sarima_model = SARIMAX(train, order=(1,1,1), seasonal_order=(1,1,1,12))
sarima_results = sarima_model.fit()
sarima_forecast = sarima_results.forecast(steps=len(test))
sarima_mse = mean_squared_error(test, sarima_forecast)

# Plotting results to compare ARIMA and SARIMA forecasts
plt.figure(figsize=(12,6))
plt.plot(train.index, train, label='Train')
plt.plot(test.index, test, label='Test')
plt.plot(test.index, arima_forecast, label='ARIMA Forecast')
plt.plot(test.index, sarima_forecast, label='SARIMA Forecast')
plt.legend()
plt.title('Churn Rate Forecasting: ARIMA vs SARIMA')
plt.xlabel('Tenure (Months)')
plt.ylabel('Churn Rate')
plt.show()

# Printing model performance metrics for comparison
print(f'ARIMA MSE: {arima_mse}')
print(f'SARIMA MSE: {sarima_mse}')
```

```
<ipython-input-118-807ee9189ea8>:14: FutureWarning: 'M' is deprecated and will
be removed in a future version, please use 'ME' instead.
  churn_by_tenure['date'] = pd.date_range(start='2020-01-01',
periods=len(churn_by_tenure), freq='M')
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency ME
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency ME
will be used.
  self._init_dates(dates, freq)
```

```
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency ME
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency ME
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-packages/statsmodels/tsa/base/tsa_model.py:473:
ValueWarning: No frequency information was provided, so inferred frequency ME
will be used.
  self._init_dates(dates, freq)
/usr/local/lib/python3.10/dist-
packages/statsmodels/tsa/statespace/sarimax.py:1009: UserWarning: Non-invertible
starting seasonal moving average Using zeros as starting parameters.
  warn('Non-invertible starting seasonal moving average'
/usr/local/lib/python3.10/dist-packages/statsmodels/base/model.py:607:
ConvergenceWarning: Maximum Likelihood optimization failed to converge. Check
mle_retvals
  warnings.warn("Maximum Likelihood optimization failed to "
```




```
ARIMA MSE: 0.004492623115808818
SARIMA MSE: 0.001101983597900703
```

[120]:
```
# Importing necessary libraries
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
```

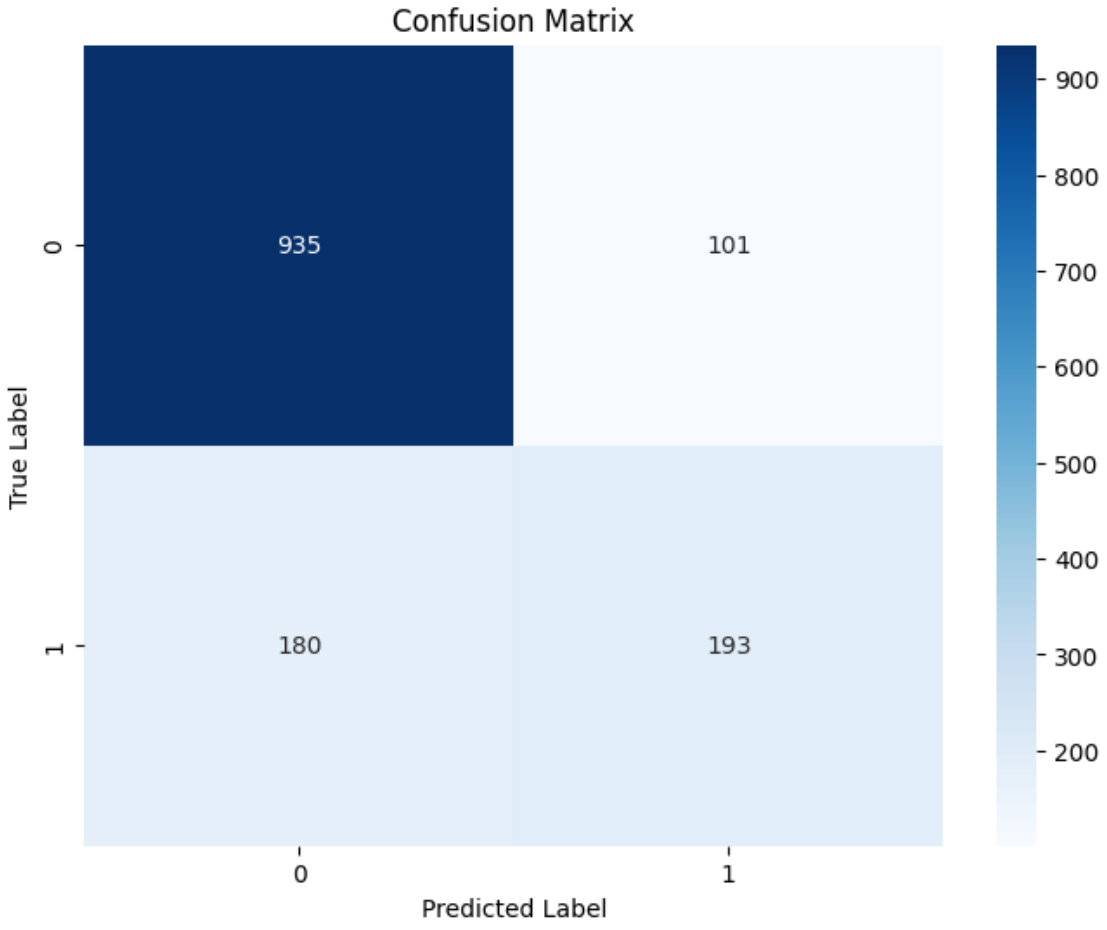
```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.optimizers import Adam
from sklearn.metrics import classification_report, confusion_matrix,␣
 ↪accuracy_score  # import accuracy_score
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd  # Import pandas


# Assuming 'df' is your dataframe and 'Churn' is your target variable
X = df.drop('Churn', axis=1)
y = df['Churn']

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

# Scaling the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Building the neural network
model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dropout(0.3),
    Dense(16, activation='relu'),
    Dense(1, activation='sigmoid')
])

model.compile(optimizer=Adam(learning_rate=0.001), loss='binary_crossentropy',␣
 ↪metrics=['accuracy'])

# Training the model
history = model.fit(X_train_scaled, y_train, epochs=100, batch_size=32,␣
 ↪validation_split=0.2, verbose=0)

# Evaluating the model
y_pred = (model.predict(X_test_scaled) > 0.5).astype(int)
print(classification_report(y_test, y_pred))

# Plotting the confusion matrix
cm = confusion_matrix(y_test, y_pred)
plt.figure(figsize=(8, 6))
```

```
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues')
plt.title('Confusion Matrix')
plt.ylabel('True Label')
plt.xlabel('Predicted Label')
plt.show()

# Plotting training history
plt.figure(figsize=(12, 4))
plt.subplot(1, 2, 1)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.title('Model Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()

plt.subplot(1, 2, 2)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.title('Model Accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()

plt.tight_layout()
plt.show()

# Feature importance (using permutation importance)
from sklearn.inspection import permutation_importance

# Define a scoring function for permutation_importance
def scorer(model, X, y):
    y_pred = (model.predict(X) > 0.5).astype(int)
    return accuracy_score(y, y_pred)  # Use accuracy_score


perm_importance = permutation_importance(model, X_test_scaled, y_test,␣
 ↪scoring=scorer)  # Pass the scoring function

feature_importance = pd.DataFrame({
    'feature': X.columns,
    'importance': perm_importance.importances_mean
}).sort_values('importance', ascending=False)

plt.figure(figsize=(10, 6))
sns.barplot(x='importance', y='feature', data=feature_importance.head(10))
plt.title('Top 10 Feature Importance (Neural Network)')
```

```
plt.show()
```

```
/usr/local/lib/python3.10/dist-packages/keras/src/layers/core/dense.py:87:
UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When
using Sequential models, prefer using an `Input(shape)` object as the first
layer in the model instead.
  super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

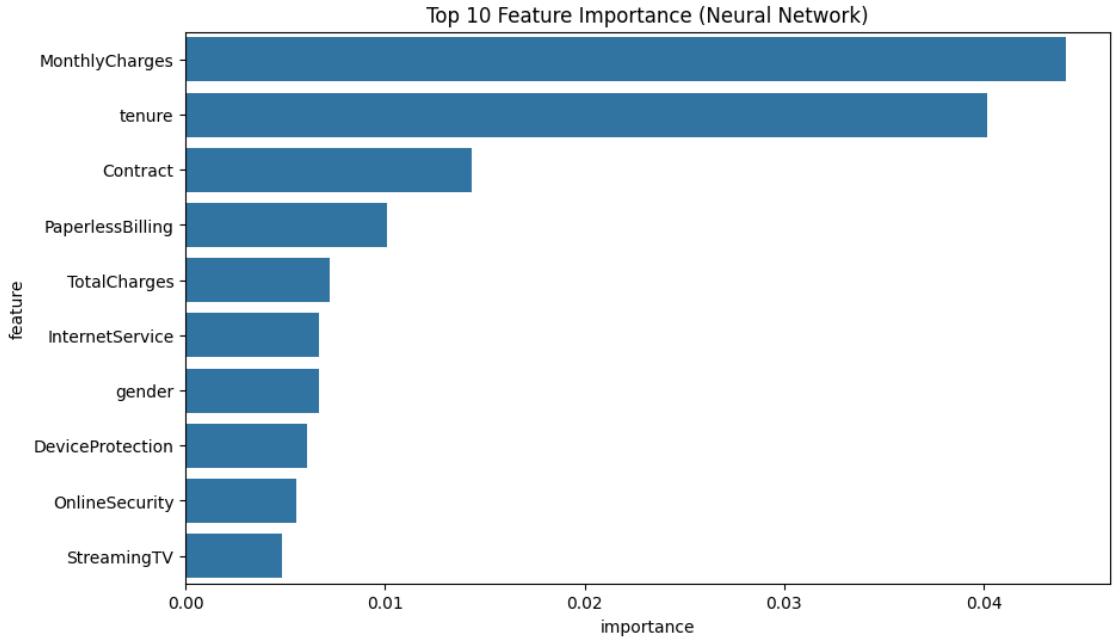
```
45/45             0s 3ms/step
              precision    recall  f1-score   support

           0       0.84      0.90      0.87      1036
           1       0.66      0.52      0.58       373

    accuracy                           0.80      1409
   macro avg       0.75      0.71      0.72      1409
weighted avg       0.79      0.80      0.79      1409
```




```
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 2ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 2ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
```

```
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 2ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 2ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 2ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 2ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 1ms/step
45/45           0s 2ms/step
45/45           0s 1ms/step
```

```
45/45             0s 1ms/step
45/45             0s 1ms/step
45/45             0s 1ms/step
45/45             0s 1ms/step
45/45             0s 1ms/step
45/45             0s 1ms/step
45/45             0s 1ms/step
45/45             0s 2ms/step
45/45             0s 1ms/step
45/45             0s 1ms/step
45/45             0s 1ms/step
45/45             0s 1ms/step
45/45             0s 2ms/step
45/45             0s 1ms/step
```



1. Model Performance:
   - The model's training accuracy increases steadily, reaching about 84% by the end of training.
   - Validation accuracy peaks early (around 81%) and then slightly decreases, suggesting some overfitting.
   - The loss curves show a similar pattern, with training loss continuing to decrease while validation loss starts to increase after a certain point.
2. Confusion Matrix:
   - True Negatives (correctly predicted non-churn): 935
   - False Positives: 101
   - False Negatives: 180
   - True Positives (correctly predicted churn): 193

- The model seems to be better at predicting non-churn (class 0) than churn (class 1).

3. Classification Report:
    - Overall accuracy: 80%
    - Precision for non-churn (0): 0.84, Recall: 0.90
    - Precision for churn (1): 0.66, Recall: 0.52
    - The model performs better on predicting non-churn cases than churn cases.
4. Feature Importance:
    - MonthlyCharges and tenure are the most important features, which aligns with typical churn prediction models.
    - Contract type is the third most important, suggesting that the type of contract significantly influences churn.
    - PaperlessBilling and TotalCharges round out the top 5, indicating that billing-related factors are crucial.
    - InternetService, gender, and device-related features (DeviceProtection, OnlineSecurity) also play a role, but to a lesser extent.

Key Takeaways: 1. The model has good overall accuracy but struggles more with predicting churn cases accurately. 2. There's evidence of some overfitting, as the validation accuracy starts to decrease while training accuracy continues to increase. 3. Monthly charges and customer tenure are the most critical factors in predicting churn, followed by contract type and billing-related features. 4. The model's performance could potentially be improved by addressing the class imbalance (more non-churn than churn cases) and by implementing techniques to reduce overfitting.

Suggestions for Improvement: 1. We can Try techniques like class weighting or oversampling to address the class imbalance. 2. Implement early stopping to prevent overfitting. 3. Experiment with different architectures or hyperparameters to improve the model's ability to predict churn cases.

Overall, our neural network model provides valuable insights into customer churn factors and demonstrates good predictive capability, especially for non-churn cases. With some refinement, it could become an even more powerful tool for churn prediction.

[5]:
```
#  ____  _                _                  _     _____              _
# / ___|| |__   __ _ ___| |__   __ _ _ __ | | _|  ___|_ _ _ __   __| | ___ _   _
# \___ \| '_ \ / _` / __| '_ \ / _` | '_ \| |/ / |_ / _` | '_ \ / _` |/ _ \ | | |
#  ___) | | | | (_| \__ \ | | | (_| | | | |   <|  _| (_| | | | | (_| |  __/ |_| |
# |____/|_| |_|\__,_|___/_| |_|\__,_|_| |_|_|\_\_|  \__,_|_| |_|\__,_|\___|\__, |
#                                                                          |___/
# ============================== Churn Predictor ================================ #
#      Unveiling customer behavior patterns with machine learning magic!          #
```

```
#␣
 ↪============================================================================␣
 ↪#
#                                                                              ␣
 ↪ #
#  Copyright (c) 2024 Shashank Pandey                                          ␣
 ↪ #
#  Contact: pandey22@buffalo.edu                                               ␣
 ↪ #
#                                                                              ␣
 ↪ #
#  Embark on a data-driven journey to predict customer churn.                  ␣
 ↪ #
#  Feel free to explore, learn, and build upon this code.                      ␣
 ↪ #
#  If it sparks joy in your work, a friendly mention would be awesome!         ␣
 ↪ #
#                                                                              ␣
 ↪ #
#  May your models be accurate and your insights profound!                     ␣
 ↪ #
#                                                                              ␣
 ↪ #
#␣
 ↪============================================================================␣
 ↪#
```