import pandas as pd

import numpy as np

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from imblearn.over\_sampling import SMOTE

from tensorflow.keras.models import Sequential

from tensorflow.keras.layers import Dense

from sklearn.metrics import classification\_report, confusion\_matrix, accuracy\_score

# Step 1: Generate Sample Data

np.random.seed(42)

data = {

    "CustomerID": range(1, 101),

    "ServiceQuality": np.random.randint(1, 11, size=100),

    "PricingAndPlans": np.random.choice(['Basic', 'Standard', 'Premium'], size=100),

    "ContractualObligations": np.random.choice(['None', '1 Year', '2 Years'], size=100),

    "CompetitorOffers": np.random.randint(0, 2, size=100),

    "CustomerEngagement": np.random.randint(1, 6, size=100),

    "BillingIssues": np.random.randint(0, 2, size=100),

    "ServiceChanges": np.random.randint(0, 2, size=100),

    "CustomerLifecycleStage": np.random.choice(['New', 'Active', 'At-Risk', 'Churned'], size=100),

    "UsagePatterns": np.random.randint(1, 101, size=100),

    "EconomicFactors": np.random.randint(1, 11, size=100),

    "TechnologyTrends": np.random.randint(1, 11, size=100),

    "CustomerSentiment": np.random.randint(1, 6, size=100),

    "Churn": np.random.randint(0, 2, size=100)

}

df = pd.DataFrame(data)

# Step 2: Preprocess the Data

df = pd.get\_dummies(df, columns=['PricingAndPlans', 'ContractualObligations', 'CustomerLifecycleStage'], drop\_first=True)

X = df.drop(columns=['CustomerID', 'Churn'])

y = df['Churn']

# Handle class imbalance

smote = SMOTE(random\_state=42)

X\_resampled, y\_resampled = smote.fit\_resample(X, y)

# Scaling features

scaler = StandardScaler()

X\_resampled = scaler.fit\_transform(X\_resampled)

# Splitting the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X\_resampled, y\_resampled, test\_size=0.3, random\_state=42)

# Step 3: Define a function to create the model

def create\_model(optimizer='adam', neurons=32):

    model = Sequential()

    model.add(Dense(neurons, input\_dim=X\_train.shape[1], activation='relu'))

    model.add(Dense(neurons // 2, activation='relu'))

    model.add(Dense(1, activation='sigmoid'))

    model.compile(loss='binary\_crossentropy', optimizer=optimizer, metrics=['accuracy'])

    return model

# Step 4: Hyperparameter tuning (manually)

best\_accuracy = 0

best\_params = {}

# Define parameter grid

neurons\_options = [16, 32, 64]

optimizers = ['adam', 'rmsprop']

batch\_sizes = [5, 10, 20]

epochs\_options = [50, 100]

for neurons in neurons\_options:

    for optimizer in optimizers:

        for batch\_size in batch\_sizes:

            for epochs in epochs\_options:

                # Create model

                model = create\_model(optimizer=optimizer, neurons=neurons)

                # Fit model

                model.fit(X\_train, y\_train, batch\_size=batch\_size, epochs=epochs, verbose=0)

                # Evaluate model

                y\_pred = (model.predict(X\_test) > 0.5).astype("int32")

                accuracy = accuracy\_score(y\_test, y\_pred)

                print(f'Neurons: {neurons}, Optimizer: {optimizer}, Batch Size: {batch\_size}, Epochs: {epochs}, Accuracy: {accuracy:.4f}')

                # Check if this is the best model so far

                if accuracy > best\_accuracy:

                    best\_accuracy = accuracy

                    best\_params = {

                        'neurons': neurons,

                        'optimizer': optimizer,

                        'batch\_size': batch\_size,

                        'epochs': epochs

                    }

# Print best parameters and accuracy

print("Best Hyperparameters:", best\_params)

print("Best Accuracy:", best\_accuracy)

# Step 5: Final evaluation using the best model

best\_model = create\_model(optimizer=best\_params['optimizer'], neurons=best\_params['neurons'])

best\_model.fit(X\_train, y\_train, batch\_size=best\_params['batch\_size'], epochs=best\_params['epochs'], verbose=0)

y\_pred = (best\_model.predict(X\_test) > 0.5).astype("int32")

print("Confusion Matrix:\n", confusion\_matrix(y\_test, y\_pred))

print("Classification Report:\n", classification\_report(y\_test, y\_pred))

# Save the best model

best\_model.save('best\_customer\_churn\_model\_nn.h5')

import matplotlib.pyplot as plt

import seaborn as sns

# Plotting the confusion matrix

confusion = confusion\_matrix(y\_test, y\_pred)

plt.figure(figsize=(8, 6))

sns.heatmap(confusion, annot=True, fmt='d', cmap='Blues', xticklabels=['Not Churned', 'Churned'], yticklabels=['Not Churned', 'Churned'])

plt.ylabel('Actual')

plt.xlabel('Predicted')

plt.title('Confusion Matrix')

plt.show()

from sklearn.metrics import classification\_report

# Get the classification report

report = classification\_report(y\_test, y\_pred, output\_dict=True)

report\_df = pd.DataFrame(report).transpose()

# Plotting the classification metrics

plt.figure(figsize=(10, 6))

report\_df[['precision', 'recall', 'f1-score']].plot(kind='bar', legend=True)

plt.title('Classification Report Metrics')

plt.xlabel('Class')

plt.ylabel('Score')

plt.xticks(rotation=0)

plt.show()

from sklearn.ensemble import RandomForestClassifier

# Train a Random Forest model to get feature importance

rf\_model = RandomForestClassifier(random\_state=42)

rf\_model.fit(X\_train, y\_train)

# Get feature importances

importances = rf\_model.feature\_importances\_

feature\_names = df.drop(columns=['CustomerID', 'Churn']).columns

importance\_df = pd.DataFrame({'Feature': feature\_names, 'Importance': importances})

# Sort by importance

importance\_df = importance\_df.sort\_values(by='Importance', ascending=False)

# Plot feature importance

plt.figure(figsize=(10, 6))

sns.barplot(data=importance\_df, x='Importance', y='Feature', palette='viridis')

plt.title('Feature Importance from Random Forest Model')

plt.xlabel('Importance Score')

plt.ylabel('Features')

plt.show()

# Calculate correlations

correlation\_matrix = df.corr()

# Plotting the heatmap

plt.figure(figsize=(12, 8))

sns.heatmap(correlation\_matrix, annot=True, fmt=".2f", cmap='coolwarm', square=True, cbar\_kws={"shrink": .8})

plt.title('Correlation Heatmap')

plt.show()

# Visualizing distribution of Service Quality

plt.figure(figsize=(10, 6))

sns.histplot(df, x='ServiceQuality', hue='Churn', multiple="stack", kde=True)

plt.title('Distribution of Service Quality by Churn Status')

plt.xlabel('Service Quality')

plt.ylabel('Count')

plt.show()

# Visualizing distribution of Customer Engagement

plt.figure(figsize=(10, 6))

sns.histplot(df, x='CustomerEngagement', hue='Churn', multiple="stack", kde=True)

plt.title('Distribution of Customer Engagement by Churn Status')

plt.xlabel('Customer Engagement')

plt.ylabel('Count')

plt.show()

# Calculate churn rate by Customer Lifecycle Stage

churn\_rate\_stage = df.groupby('CustomerLifecycleStage\_New')['Churn'].mean()

# Plot churn rate by customer lifecycle stage

plt.figure(figsize=(8, 6))

churn\_rate\_stage.plot(kind='bar', color='salmon')

plt.title('Churn Rate by Customer Lifecycle Stage')

plt.xlabel('Customer Lifecycle Stage')

plt.ylabel('Churn Rate')

plt.xticks(rotation=0)

plt.show()

# Box plot for Service Quality

plt.figure(figsize=(10, 6))

sns.boxplot(x='Churn', y='ServiceQuality', data=df, palette='Set2')

plt.title('Service Quality by Churn Status')

plt.xlabel('Churn')

plt.ylabel('Service Quality')

plt.xticks(ticks=[0, 1], labels=['Not Churned', 'Churned'])

plt.show()

# Box plot for Customer Engagement

plt.figure(figsize=(10, 6))

sns.boxplot(x='Churn', y='CustomerEngagement', data=df, palette='Set2')

plt.title('Customer Engagement by Churn Status')

plt.xlabel('Churn')

plt.ylabel('Customer Engagement')

plt.xticks(ticks=[0, 1], labels=['Not Churned', 'Churned'])

plt.show()

# Check if 'PricingAndPlans' or similar column exists

# Replace 'PricingAndPlans' with the actual column name if different

if 'PricingAndPlans\_Standard' in df.columns:

    churn\_rate\_pricing = df.groupby('PricingAndPlans\_Standard')['Churn'].mean()

    # Plot churn rate by pricing plans

    plt.figure(figsize=(8, 6))

    churn\_rate\_pricing.plot(kind='bar', color='salmon')

    plt.title('Churn Rate by Pricing Plans')

    plt.xlabel('Pricing Plans')

    plt.ylabel('Churn Rate')

    plt.xticks(rotation=45)

    plt.show()

else:

    print("Column 'PricingAndPlans' not found in the DataFrame.")

# Replace 'UsagePatterns' with the actual column name if it exists

if 'UsagePatterns' in df.columns:

    churn\_rate\_usage = df.groupby('UsagePatterns')['Churn'].mean()

    # Plot churn rate by usage patterns

    plt.figure(figsize=(8, 6))

    churn\_rate\_usage.plot(kind='bar', color='salmon')

    plt.title('Churn Rate by Usage Patterns')

    plt.xlabel('Usage Patterns')

    plt.ylabel('Churn Rate')

    plt.xticks(rotation=45)

    plt.show()

else:

    print("Column 'UsagePatterns' not found in the DataFrame.")

# Calculate overall churn rate

overall\_churn\_rate = df['Churn'].mean()

# Visualize overall churn rate

plt.figure(figsize=(6, 4))

sns.countplot(x='Churn', data=df, palette='pastel')

plt.title('Overall Churn Rate')

plt.xlabel('Churn')

plt.ylabel('Count')

plt.xticks(ticks=[0, 1], labels=['Not Churned', 'Churned'])

plt.show()

print(f"Overall Churn Rate: {overall\_churn\_rate:.2%}")