Enhancing Credit Scoring Models Through Time Series Clustering

Note:
1. Some Code in the Notebook Has Been Modified to Comply with Experian's Data and Privacy Policies.
2. Visualizations Are Attached Separately for Added Clarity.

Experiment 1

1. Import Libraries

In [2]:
```python
# Importing easy_peas3 for data acquisition and loading from S3 buckets
import easy_peas3
from easy_peas3 import S3

# Importing necessary libraries for data manipulation and analysis
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

# Importing garbage collector to manage memory usage during runtime
import gc

# Importing tqdm for displaying progress bars in Jupyter Notebooks
from tqdm.autonotebook import tqdm

# Importing statistical tools for data analysis
from scipy.stats import skew
from math import ceil

# Importing preprocessing tools for scaling and transforming data
from sklearn.preprocessing import MinMaxScaler, QuantileTransformer, StandardScaler

# Importing KMeans clustering algorithm for time series data
from sklearn.cluster import KMeans

# Importing t-SNE for dimensionality reduction and visualization
from sklearn.manifold import TSNE

# Importing tools for model selection and evaluation
from sklearn.model_selection import train_test_split, StratifiedKFold
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import confusion_matrix, classification_report
from sklearn.metrics import roc_curve, roc_auc_score
from tabulate import tabulate

# Configuring warnings to ignore unnecessary warnings
import warnings
warnings.filterwarnings('ignore')

# Configuring pandas display options for improved dataframe visualization
pd.set_option('display.max_columns', None)
pd.set_option('future.no_silent_downcasting', True)
```

2. Data Loading

In [2]:
```python
# Create an S3 object and connect to the specified project bucket
s3 = S3(project_bucket='**')
bucket = s3.project_bucket()
```

In [3]:
```python
# Reading Flag Data from the specified path
# 'security_classification' indicates the sensitivity level of the data
# 'subpath' specifies the exact location of the CSV file within the S3 bucket
# 'low_memory' is set to False to ensure the file is processed without memory optimization concerns

data_flag = bucket.read_data_assets_csv(
    security_classification="**",
    subpath='**',
    low_memory=False
)
```

In [4]:
```python
# Filter the Flag Data to retain only rows where the 'GB_FLAG' column contains 'G' or 'B'
data_flag = data_flag.loc[data_flag['GB_FLAG'].isin(['G', 'B'])]

# Create a set of unique IDs from the 'UNIQUEID' column in the filtered Flag Data
selected_ids = set(data_flag.UNIQUEID.values)
```

In [5]:
```python
# Column names for balance data
balances = [f'BALANCE_{i+1}' for i in range(72)]

# Column names for account status data
statuses = [f'STATUS_{i+1}' for i in range(72)]

# Column names for monthly payment changes (MMYY)
monthly_payment = [f'MONTHLY_PAYMENT_CHANGE_{i+1}' for i in range(12)]
monthly_payment_date = [f'MONTHLY_PAYMENT_CHANGE_DT_{i+1}' for i in range(12)]

# Column names for credit limit changes (MMYY)
credit_limit = [f'CREDIT_LIMIT_CHANGE_{i+1}' for i in range(12)]
credit_limit_date = [f'CREDIT_LIMIT_CHANGE_DATE_{i+1}' for i in range(12)]

# Column names for various significant dates related to CAIS accounts (DDMMYY)
other_dates = ['DATE_INFORMATION_LAST_UPDATED', 'START_DATE', 'SETTLEMENT_DATE']

# Column names for various static attributes associated with CAIS accounts
account_values = ['BUREAU_REF_GROUP', 'UNIQUEID', 'NUMBER_OF_MONTHS_HISTORY', 'ACCOUNT_TYPE', 'CURRENT_CREDIT_LIMIT', 'CURRENT_MONTHLY_PAYMENT']

# Column name for retroactive date
retro_date = ['RETRO_DATE']
```

In [6]:
```python
# Define the columns to be included for the project to optimize memory usage
cols_wanted = (balances + statuses +
               monthly_payment + monthly_payment_date +
               credit_limit + credit_limit_date + other_dates +
               account_values + retro_date)
```

In [7]:

```python
# Function to downcast data types for memory efficiency
def downcast_dtypes(df):
    # Identify columns with float64 and int64 data types
    float_cols = [c for c in df if df[c].dtype == "float64"]
    int_cols = [c for c in df if df[c].dtype == "int64"]
    # Downcast float64 to float32 and int64 to int32
    df[float_cols] = df[float_cols].astype("float32")
    df[int_cols] = df[int_cols].astype("int32")
    return df

# Initialize an empty DataFrame to store CAIS data
cais_data = pd.DataFrame()

# Read the CAIS data in chunks from the specified S3 bucket
df_iter = bucket.read_data_assets_csv(
    security_classification="**",
    subpath='**',
    chunksize=300_000,
    iterator=True,
    usecols=cols_wanted,
    low_memory=False
)

# Print success message
print("Data successfully read from S3 bucket.")

# Process each chunk of data
for i, chunk in tqdm(enumerate(df_iter)):
    # Filter rows and downcast data types
    chunk = chunk.loc[(chunk.BUREAU_REF_GROUP == 1) & (chunk.UNIQUEID.isin(selected_ids))]
    chunk = downcast_dtypes(chunk)
    # Append the processed chunk to the main DataFrame
    cais_data = pd.concat([cais_data, chunk])
    # Clear the chunk from memory
    del chunk
    gc.collect()

# Output the total number of unique IDs and CAIS accounts processed
total_unique_ids = cais_data.UNIQUEID.nunique()
total_cais_accounts = len(cais_data)
print(f'\nTotal UNIQUEIDs: {total_unique_ids}, Total CAIS Accounts: {total_cais_accounts}')
```

```
Data successfully read from S3 bucket.
34/? [12:05&lt;00:00, 22.54s/it]
['\nTotal UNIQUEIDs:', 313513, 'Total CAIS Accounts:', 4222024]
```

In [8]:

```python
# Get the shape of the CAIS data
cais_data.shape
```

Out[8]:

```
(4222024, 230)
```

3. Data Pre-Processing

A. Data Formating

In [9]:
```
# Standardizing MMYY Date Columns

for f in monthly_payment_date + credit_limit_date:
    cais_data[f] = cais_data[f].apply(lambda x: f'{int(x):04d}' if not pd.isnull(x) else x)

# Print confirmation message
print("Date columns cleaned successfully.")
```

```
Date columns cleaned successfully.
```

In [10]:
```
# Converting Dates to Datetime Columns

# Define date formats for conversion
date_format_other = '%d%m%y'  # Format: day, month, year (two-digit year)
date_format_retro = '%Y%m%d'  # Format: year, month, day (four-digit year)

# Convert 'RETRO_DATE' column to datetime, coercing invalid dates to NaT
cais_data[retro_date[0]] = pd.to_datetime(cais_data[retro_date[0]], format=date_format_retro, errors='coerce')

# Convert other date columns to datetime, using the other date format
for f in other_dates:
    cais_data[f] = pd.to_datetime(cais_data[f], format=date_format_other, errors='coerce')

# Print confirmation message
print("Date columns cleaned successfully.")
```

```
Date columns cleaned successfully.
```

In [11]:
```
# Update missing values in 'DATE_INFORMATION_LAST_UPDATED'

# Set to RETRO_DATE if SETTLEMENT_DATE is also missing; otherwise, use SETTLEMENT_DATE
cais_data['DATE_INFORMATION_LAST_UPDATED'] = np.where(
    cais_data.DATE_INFORMATION_LAST_UPDATED.isna(),
    np.where(cais_data.SETTLEMENT_DATE.isna(), cais_data.RETRO_DATE, cais_data.SETTLEMENT_DATE),
    cais_data.DATE_INFORMATION_LAST_UPDATED
)

# Print confirmation message
print("DATE_INFORMATION_LAST_UPDATED column updated successfully.")
```

```
DATE_INFORMATION_LAST_UPDATED column updated successfully.
```

In [12]:
```
# Adjust 'START_DATE' values if they are later than 'RETRO_DATE' by subtracting 100 years

cais_data.loc[cais_data.START_DATE > cais_data.RETRO_DATE, 'START_DATE'] = \
    cais_data.loc[cais_data.START_DATE > cais_data.RETRO_DATE, 'START_DATE'].apply(lambda d: d.replace(year=d.year - 100))

# Print confirmation message
print("START_DATE values modified successfully.")
```

```
START_DATE values modified successfully.
```

In [13]:
```python
# Clean 'STATUS' columns by replacing specific values and converting to integer type
for s in statuses:
    cais_data.loc[cais_data[s] == 'U', s] = '-1'  # Replace 'U' with '-1'
    cais_data.loc[cais_data[s] == 'D', s] = '-2'  # Replace 'D' with '-2'
    cais_data.loc[cais_data[s] == '?', s] = '-3'  # Replace '?' with '-3'
    cais_data.loc[cais_data[s].isna(), s] = '0'   # Replace NaN with '0'
    cais_data[s] = cais_data[s].astype(np.int16)   # Convert to integer type

# Print confirmation message
print("Status columns cleaned successfully.")
```

```
Status columns cleaned successfully.
```

B. Account Type Segmentation

In [14]:
```python
# Class for defining account types and their characteristics

class AccTypes:
    # Set of account types related to debt repayment
    debt_to_pay = set([
        2, 3, 1, 46, 61, 17, 28, 27, 29, 45, 30, 19, 23, 16, 31, 33, 35,
        22, 24, 26, 20, 62, 36, 48, 32, 34, 47, 49, 50, 51, 60, 64, 69, 25
    ])

    # Set of account types related to credit spending
    credit_to_spend = set([
        5, 15, 8, 4, 6, 37, 38, 71, 70
    ])

    # Set of account types related to bill payments
    bills = set([
        18, 39, 41, 40, 59, 21, 43, 53, 7, 58, 57, 54, 55, 56, 42, 44
    ])
```

C. Implementation of Ageing and Monthly Aggregated Time Series Creation

In [ ]:
```python
def customer_TS_data(cus_df):
    """
    Processes consumer data to create a time series DataFrame with aggregated monthly statistics.

    Parameters:
    cus_df (pd.DataFrame): DataFrame containing consumer account data with various attributes.

    Returns:
    dict: A dictionary with aggregated data for each unique consumer ID.
    """

    # Define a one-month offset for date adjustments
    one_month_offset = pd.DateOffset(months=1)

    # Function to classify account types into categories
    # 'D' for debt_to_pay, 'C' for credit_to_spend, 'B' for bills, 'O' for others
    account_class = lambda acctype: ('D' if acctype in AccTypes.debt_to_pay else
                                     'C' if acctype in AccTypes.credit_to_spend else
                                     'B' if acctype in AccTypes.bills else
                                     'O')
```

```python
def fix_status_balance(a):
    """
    Forward-fills missing status and balance values based on the next month's values.

    Parameters:
    a (pd.Series): Series containing account data including statuses and balances.

    Returns:
    pd.Series: Updated account data with fixed status and balance values.
    """
    m = a.NUMBER_OF_MONTHS_HISTORY
    idx = np.nonzero(a[statuses].values == -3)[0]
    if idx.shape[0] > 0:
        if idx[-1]+1 == m:
            idx = idx[:-1]
        for i in idx[::-1]:
            a[statuses[i]] = a[statuses[i+1]]
            a[balances[i]] = a[balances[i+1]]
    return a

def unpack_events(current_value, acc_df, new_col, values_list, change_dates_list, last_month_date):
    """
    Updates a DataFrame with values based on change events and corresponding dates.

    Parameters:
    current_value (float): The current value to start with.
    acc_df (pd.DataFrame): DataFrame to be updated with change events.
    new_col (str): The column name to update with new values.
    values_list (list): List of values to apply based on change dates.
    change_dates_list (list): List of change dates corresponding to values.
    last_month_date (str): The last month in the historical window.

    Returns:
    pd.DataFrame: Updated DataFrame with unpacked events.
    """
    if pd.isnull(change_dates_list[0]):
        acc_df[new_col] = current_value
        change_dates = []
    else:
        acc_df[new_col] = np.nan
        acc_df.iloc[0, acc_df.columns.get_loc(new_col)] = current_value
        change_dates = change_dates_list[~pd.isnull(change_dates_list)]

        for i, d1 in enumerate(change_dates):
            try:
                d2 = pd.to_datetime(d1, format='%m%y')
            except ValueError:
                try:
                    d2 = pd.to_datetime(d1, format='%d%m%y')
                except ValueError:
                    continue
            d3 = (d2 - one_month_offset).strftime('%Y%m')
            if d3 < last_month_date:
                break
            acc_df.loc[d3, new_col] = values_list[i]
            current_value = values_list[i]
```

```
    acc_df[new_col] = acc_df[new_col].ffill()
    return acc_df

def monthly_aggregates(month_df):
    """
    Computes aggregated metrics for a given month's DataFrame.

    Parameters:
    month_df (pd.DataFrame): DataFrame containing account data for a single consumer groupby month.

    Returns:
    dict: Aggregated metrics including balance, credit limit, etc.
    """
    r = {'M': month_df.index.values[0],  # Set the month value
         'WSRatio': np.mean(month_df.Stat.values > 0),  # Ratio of positive statuses
         'nb_Acc': len(month_df)}  # Count the number of accounts
    r['CLmt'] = 0  # Initialize credit limit

    for cls in ['C', 'D', 'B']:
        r[f'{cls}Bal'] = 0  # Initialize balance for each account class
        tmp = month_df.loc[month_df['class'].values == cls]  # Filter accounts by class
        n = len(tmp)
        if n > 0:
            r[f'{cls}Bal'] = np.sum(tmp.Bal.values)  # Total balance for the class
            if cls == 'C':
                cl = tmp.CLmt.values > 0
                r['CLmt'] = np.sum(tmp.CLmt.values, where=cl, initial=0)  # Total credit limit
    return r

TS_len = 24  # Number of months for the time series

cus_df = cus_df.copy()

# Assign unique account IDs
cus_df['ACC_ID'] = np.arange(len(cus_df), dtype=np.int16)

# Find the maximum RETRO_DATE for historical window
max_retro_date = cus_df.RETRO_DATE.max()

# Generate a sequence of historical months
history_months_seq = pd.date_range(start=max_retro_date, periods=TS_len, freq='-1ME').strftime('%Y%m')

# Define output DataFrame
out_df = pd.DataFrame()

# Process each account in the DataFrame
for _, acc in cus_df.iterrows():
    update_date = acc.DATE_INFORMATION_LAST_UPDATED
    N = int(acc.NUMBER_OF_MONTHS_HISTORY)
    months_seq = pd.date_range(start=update_date, periods=N, freq='-1ME').strftime('%Y%m')

    if months_seq.intersection(history_months_seq).shape[0] == 0:
        continue

    acc = fix_status_balance(acc)
```

```python
        acc_df = pd.DataFrame(data={'ACC_ID': acc.ACC_ID,
                                    'Stat': acc[statuses[:N]].values,
                                    'Bal': acc[balances[:N]].values, 'M': months_seq}).set_index('M')

        acc_df['Bal'] = acc_df['Bal'].astype(np.float32)
        acc_df['Stat'] = acc_df['Stat'].astype(np.float32)

        # Process monthly payment
        if acc.ACCOUNT_TYPE in AccTypes.has_monthly_payment:
            acc_df = unpack_events(acc['CURRENT_MONTHLY_PAYMENT'], acc_df, 'Pym', acc[monthly_payment].values,
                                   acc[monthly_payment_date].values, months_seq[-1])
        else:
            acc_df['Pym'] = 0

        # Process credit limit
        if acc.ACCOUNT_TYPE in AccTypes.has_credit_limit:
            acc_df = unpack_events(acc['CURRENT_CREDIT_LIMIT'], acc_df, 'CLmt', acc[credit_limit].values,
                                   acc[credit_limit_date].values, months_seq[-1])
        else:
            acc_df['CLmt'] = 0
            acc_df['CLU'] = 0

        # Sort the DataFrame by date
        acc_df.sort_index(inplace=True)

        # Keep only the records within the historical window
        acc_df = acc_df.loc[acc_df.index.isin(history_months_seq)]

        if len(acc_df) == 0:
            continue

        # Add account class and type information
        acc_df['class'] = account_class(acc.ACCOUNT_TYPE)
        acc_df['type'] = acc.ACCOUNT_TYPE

        # Append the account DataFrame to the output DataFrame
        out_df = pd.concat([out_df, acc_df[['ACC_ID', 'Stat', 'Bal', 'CLmt', 'Pym', 'class', 'type']]])

    if len(out_df) == 0:
        # Return an empty DataFrame if no data is processed
        return pd.DataFrame(columns=['nb_Acc', 'WSRatio', 'CLmt', 'CBal', 'DBal', 'BBal'])

    # Group the output DataFrame by month and calculate aggregates
    monthly_agg_df = pd.DataFrame([monthly_aggregates(m) for i, m in out_df.groupby(out_df.index)]).fillna(0)

    # Sort the aggregated data by month
    monthly_agg_df.sort_values(by='M', inplace=True, ascending=False)

    # Initialize a dictionary to store the results
    r = {'UNIQUEID': cus_df['UNIQUEID'].values[0]}

    # Populate the dictionary with aggregated data for each month creating a time series
    for m, (_, month_row) in enumerate(monthly_agg_df.iterrows()):
        for f in ['nb_Acc', 'WSRatio', 'CLmt', 'CBal', 'DBal', 'BBal']:
            r[f'{f}_{m}'] = month_row[f]

    return r
```

```
# Process account data for each unique ID
combined_data = pd.DataFrame([customer_TS_data(m) for i, m in cais_data.groupby('UNIQUEID')])
```

In [16]:
```
# Get the shape of the final combined_data Dataframe
combined_data.shape
```

Out[16]: (313513, 145)

D. Combined Data Statistics

In [17]:
```
# List of month attributes for time series data
month_attrbs = ['nb_Acc', 'WSRatio', 'CLmt', 'CBal', 'DBal', 'BBal']

# Number of months in the time series
TS_len = 24

# List to store column names for each attribute and month
TS_cols = []

# Generate column names for each attribute and month
# E.g., 'nb_Acc_0', 'WSRatio_0', ..., 'BBal_23'
for m in range(TS_len):
    for f in month_attrbs:
        TS_cols.append(f'{f}_{m}')

# List to store scaled column names
# E.g., 'nb_Acc_0_scaled', 'WSRatio_0_scaled', ..., 'BBal_23_scaled'
TS_cols_scaled = [f'{f}_scaled' for f in TS_cols]
```

In [18]:
```
# Fill missing values in 'combined_data' with 0
combined_data.fillna(0, inplace=True)

# Merge 'combined_data' with 'data_flag' on 'UNIQUEID', using a left join
combined_data = pd.merge(combined_data, data_flag, on='UNIQUEID', how='left')

# Filter to keep only rows where 'GB_FLAG' is not NaN
combined_data = combined_data.loc[~combined_data.GB_FLAG.isna()]

# Create 'target' column: 1 where 'GB_FLAG' is 'B', otherwise 0
combined_data['target'] = np.int16(combined_data.GB_FLAG == 'B')

# Remove 'cais_data' and 'data_flag' from memory
del cais_data, data_flag

# Run garbage collection to free up memory
gc.collect()

# Print the shape of the cleaned 'combined_data' DataFrame
print(combined_data.shape)
```

Out[18]: (311506, 147)

In [20]:
```
# GB_FLAG Value Count
combined_data.GB_FLAG.value_counts()
```

Out[20]:
```
GB_FLAG
G    257545
B     53961
Name: count, dtype: int64
```

In [21]:
```
# GB_FLAG Proportion Values
combined_data.GB_FLAG.value_counts(normalize=1)
```

Out[21]:
```
GB_FLAG
G    0.826774
B    0.173226
Name: proportion, dtype: float64
```

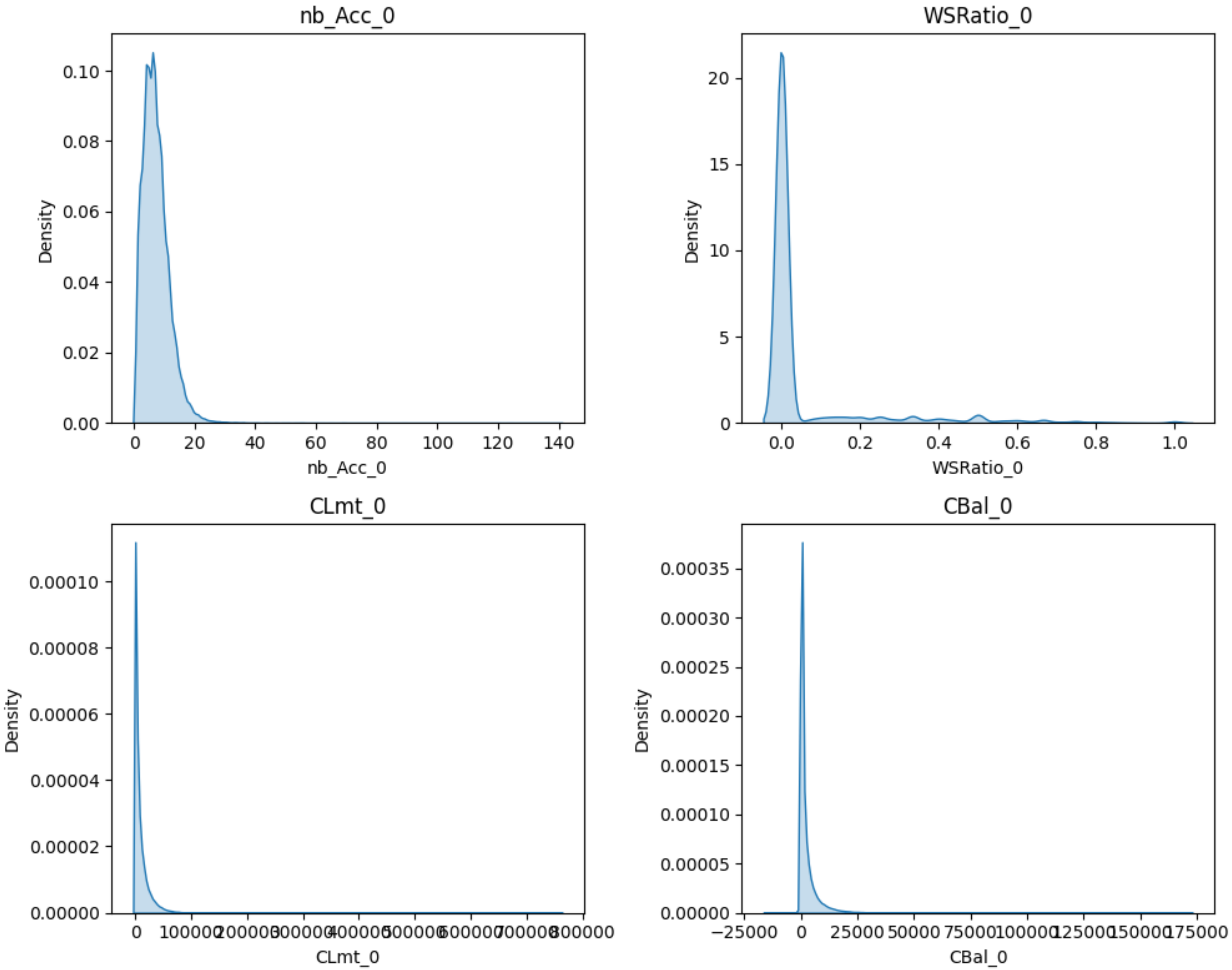
E. Data Partitioning

In [22]:
```
# Split indices of 'combined_data' into training and testing sets
indices_train, indices_test = train_test_split(
    np.arange(len(combined_data)),  # Array of row indices
    test_size=0.6,  # 60% for testing
    stratify=combined_data['GB_FLAG'],  # Stratify by 'GB_FLAG'
    random_state=217  # Seed for reproducibility
)

# Create training set using selected indices
cluster_train = combined_data.iloc[indices_train]

# Create testing set using selected indices
cluster_test = combined_data.iloc[indices_test]

# Delete temporary variables to free memory
del indices_train, indices_test

# Run garbage collection to clean up memory
gc.collect()

# Print shapes of the training and testing datasets
cluster_train.shape, cluster_test.shape
```

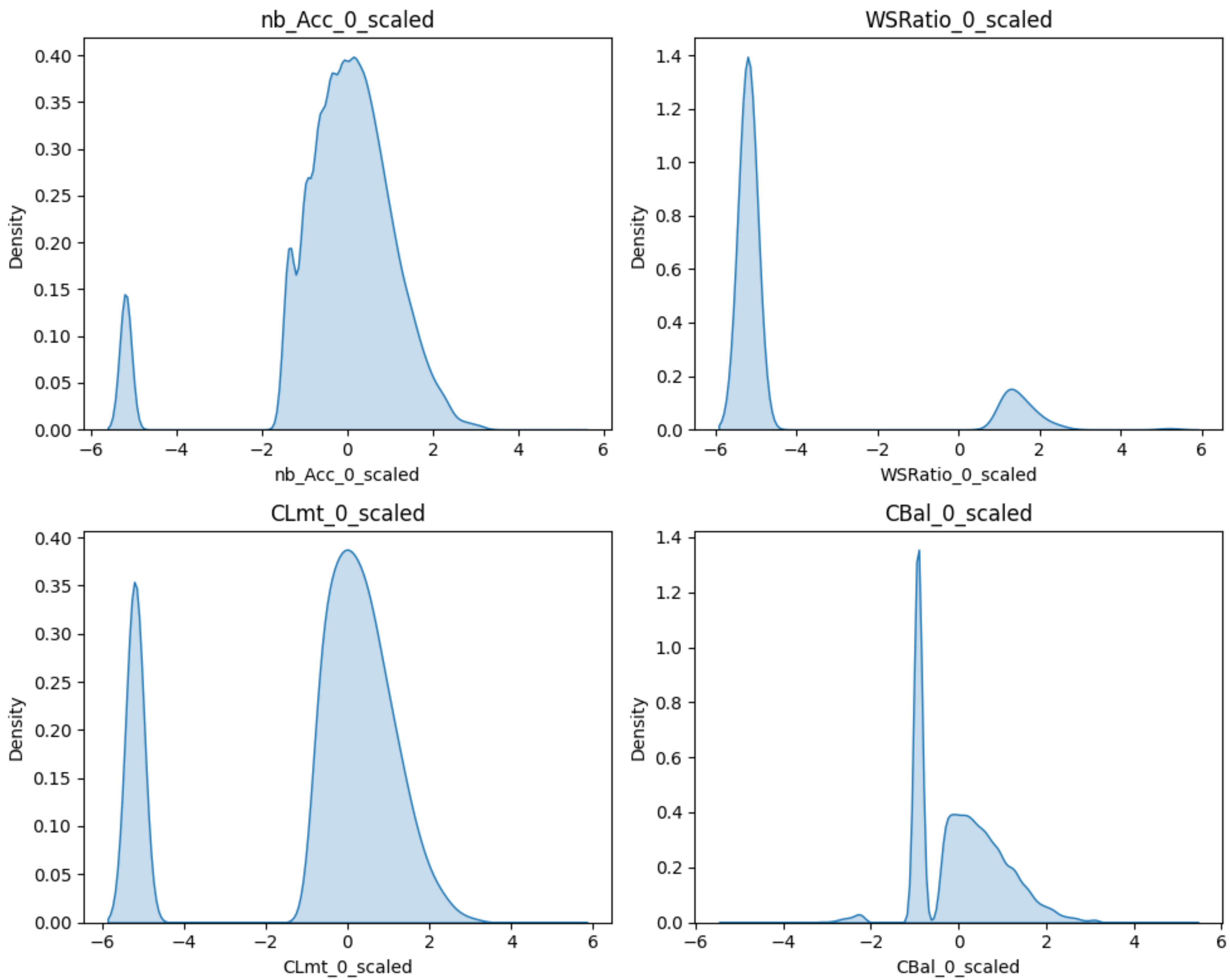
Out[22]:
```
((124602, 147), (186904, 147))
```

F. Time-Series Feature Scaling

In [24]:
```
# List of attributes to visualize
attributes = ['nb_Acc_0', 'WSRatio_0', 'CLmt_0', 'CBal_0']

# Create a 2x2 grid of subplots
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 8))

# Plot KDE for each attribute in its subplot
for i, col in enumerate(attributes):
    sns.kdeplot(cluster_train[col], ax=axes[i // 2, i % 2], fill=True)
    axes[i // 2, i % 2].set_title(col)

# Adjust layout and display plots
plt.tight_layout()
plt.show()
```


nb_Acc_0
WSRatio_0
CLmt_0
CBal_0
Density

In [25]:
```python
# Initialize QuantileTransformer for normalizing data distribution
quntTrans = QuantileTransformer(output_distribution='normal', random_state=217)

# Transform training data and store in scaled columns
cluster_train.loc[:, TS_cols_scaled] = quntTrans.fit_transform(cluster_train[TS_cols])

# Transform test data using the same parameters
cluster_test.loc[:, TS_cols_scaled] = quntTrans.transform(cluster_test[TS_cols])

# Attributes for visualizing scaled data
attributes = ['nb_Acc_0_scaled', 'WSRatio_0_scaled', 'CLmt_0_scaled', 'CBal_0_scaled']

# Create a 2x2 grid of subplots for visualizing distributions
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 8))

# Plot KDE for each scaled attribute
for i, col in enumerate(attributes):
    sns.kdeplot(cluster_train[col], ax=axes[i // 2, i % 2], fill=True)
    axes[i // 2, i % 2].set_title(col)

# Adjust layout and display plots
plt.tight_layout()
plt.show()
```


nb_Acc_0_scaled
WSRatio_0_scaled
CLmt_0_scaled
CBal_0_scaled
Density

In [26]:
```python
# Initialize MinMaxScaler for scaling data to the [0, 1] range
scaler = MinMaxScaler()

# Scale training data using MinMaxScaler
cluster_train.loc[:, TS_cols_scaled] = scaler.fit_transform(cluster_train[TS_cols_scaled])

# Apply the same scaling to test data
cluster_test.loc[:, TS_cols_scaled] = scaler.transform(cluster_test[TS_cols_scaled])
```

4. Time-Series Clustering

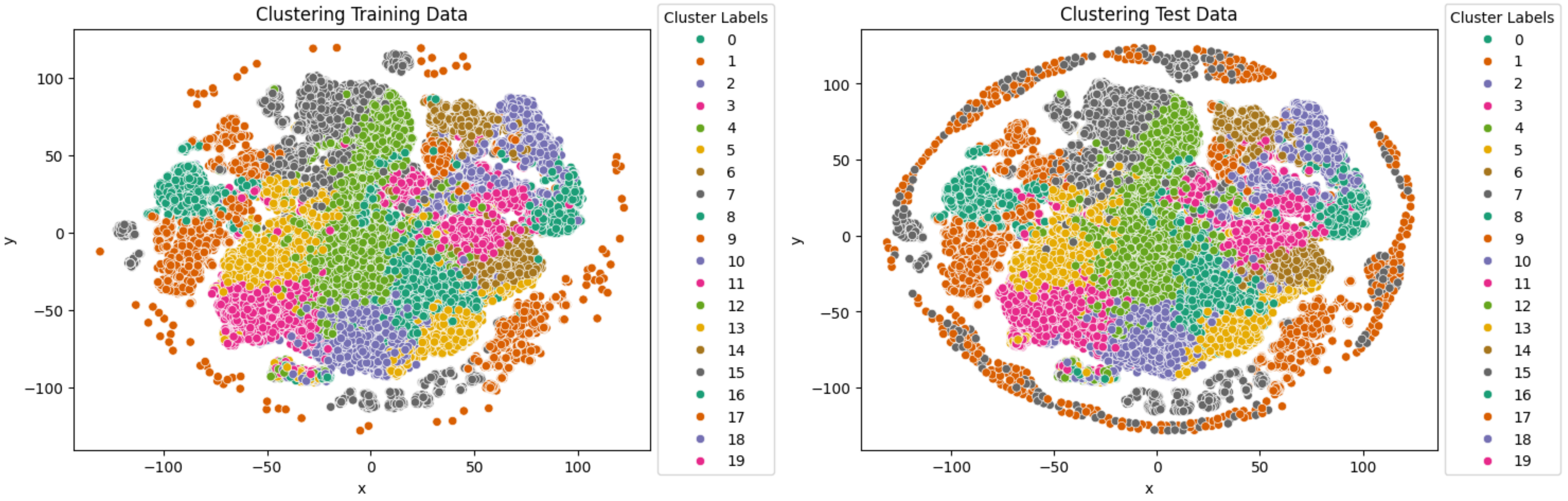
A. Time-Series Clustering Implementation

In [32]:
```python
# Define sample rate for visualization and number of clusters
sample_rate = 0.50
K = 20

# Initialize KMeans clustering model with specified number of clusters
km_model = KMeans(n_clusters=K, max_iter=10000, random_state=616)

# Fit the KMeans model to the scaled training data and assign cluster labels
cluster_train['cluster_label'] = np.int16(km_model.fit_predict(cluster_train[TS_cols_scaled]))

# Retrieve cluster centroids from the trained KMeans model
centroids_train = km_model.cluster_centers_

# Predict cluster labels for the scaled test data
cluster_test['cluster_label'] = np.int16(km_model.predict(cluster_test[TS_cols_scaled]))

# Define column names for distances to each cluster centroid
cluster_dist_cols = [f'cluster{i}_dist' for i in range(K)]

# Calculate distances from each test sample to each cluster centroid
cluster_test.loc[:, cluster_dist_cols] = km_model.transform(cluster_test[TS_cols_scaled].values)
```

In [ ]:
```python
# Sample 50% of the training data
tmp1 = cluster_train.sample(int(len(cluster_train) * sample_rate))
tmp1['holdout'] = 0  # Indicate this is training data

# Sample 50% of the test data
tmp2 = cluster_test.sample(int(len(cluster_test) * sample_rate))
tmp2['holdout'] = 1  # Indicate this is test data

# Combine the sampled training and test data
tmp1 = pd.concat([tmp1, tmp2])
```

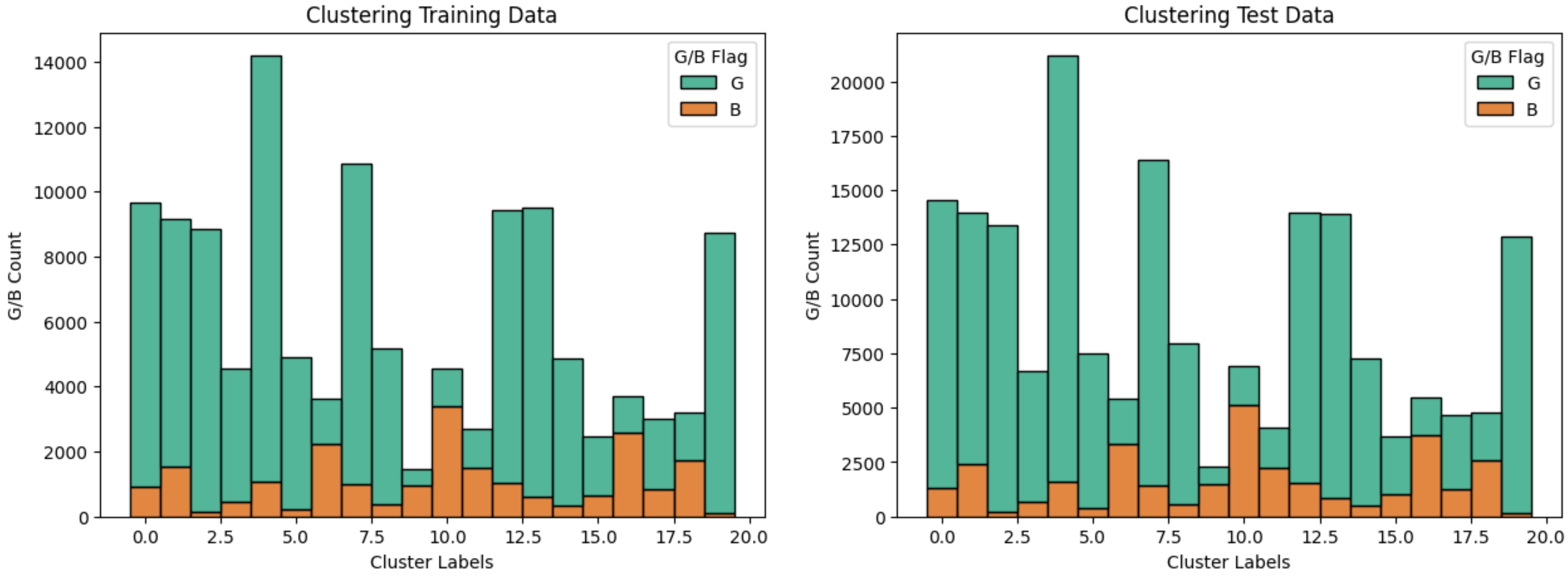
A.1 Clustering Visualization - Plot of Train and Test Subset with Legends as Cluster Labels

In [34]:
```
# Apply t-SNE to reduce the dimensionality of the scaled data to 2D
# Add the resulting coordinates to 'tmp1' DataFrame as 'x' and 'y'
tmp1.loc[:, ['x', 'y']] = TSNE(n_components=2, init='random', random_state=217, perplexity=50).fit_transform(tmp1[TS_cols_scaled])

# Create a figure with 2 subplots for training and test data visualization
fig, ax = plt.subplots(1, 2, figsize=(15, 5))

# Scatter plot for the training data subset
sns.scatterplot(data=tmp1[tmp1.holdout == 0], x='x', y='y', hue='cluster_label', palette='Dark2', hue_order=list(range(K)), ax=ax[0])
ax[0].set_title('Clustering Training Data')
ax[0].get_legend().set_title('Cluster Labels')

# Scatter plot for the test data subset
sns.scatterplot(data=tmp1[tmp1.holdout == 1], x='x', y='y', hue='cluster_label', palette='Dark2', hue_order=list(range(K)), ax=ax[1])
ax[1].set_title('Clustering Test Data')
ax[1].get_legend().set_title('Cluster Labels')

# Position legends outside the plots for clarity
for i in range(2):
    ax[i].legend(loc='center left', bbox_to_anchor=(1, 0.5), title='Cluster Labels')

# Adjust layout and display the plots
plt.tight_layout()
plt.show()
```



A.2 Clustering Visualization - Plot of Train and Test Subset by G/B Flag Count

In [36]:
```python
# Create a figure with 2 subplots for histograms of training and test data
fig, ax = plt.subplots(1, 2, figsize=(15, 5))

# Histogram for the training data by G/B Flag Count
sns.histplot(data=cluster_train, x='cluster_label', hue='GB_FLAG', palette='Dark2', discrete=True,
             element='bars', multiple="stack", hue_order=['G', 'B'], ax=ax[0])
ax[0].set_title('Clustering Training Data')
ax[0].get_legend().set_title('G/B Flag')
ax[0].set(xlabel='Cluster Labels', ylabel='G/B Count')

# Histogram for the test data by G/B Flag Count
sns.histplot(data=cluster_test, x='cluster_label', hue='GB_FLAG', palette='Dark2', discrete=True,
             element='bars', multiple="stack", hue_order=['G', 'B'], ax=ax[1])
ax[1].set_title('Clustering Test Data')
ax[1].get_legend().set_title('G/B Flag')
ax[1].set(xlabel='Cluster Labels', ylabel='G/B Count')

# Display the histograms
plt.show()
```



In [37]:
```
# Group 'cluster_test' by 'cluster_label' and calculate mean and count of 'target'
result = cluster_test.groupby('cluster_label', as_index=False).agg({'target': ['mean', 'count']})

# Flatten MultiIndex columns and rename them
result.columns = ['Cluster Label', 'Bad Proportion', 'Cluster Proportion']

# Normalize cluster proportions
result['Cluster Proportion'] /= result['Cluster Proportion'].sum()

# Display the resulting DataFrame
result
```

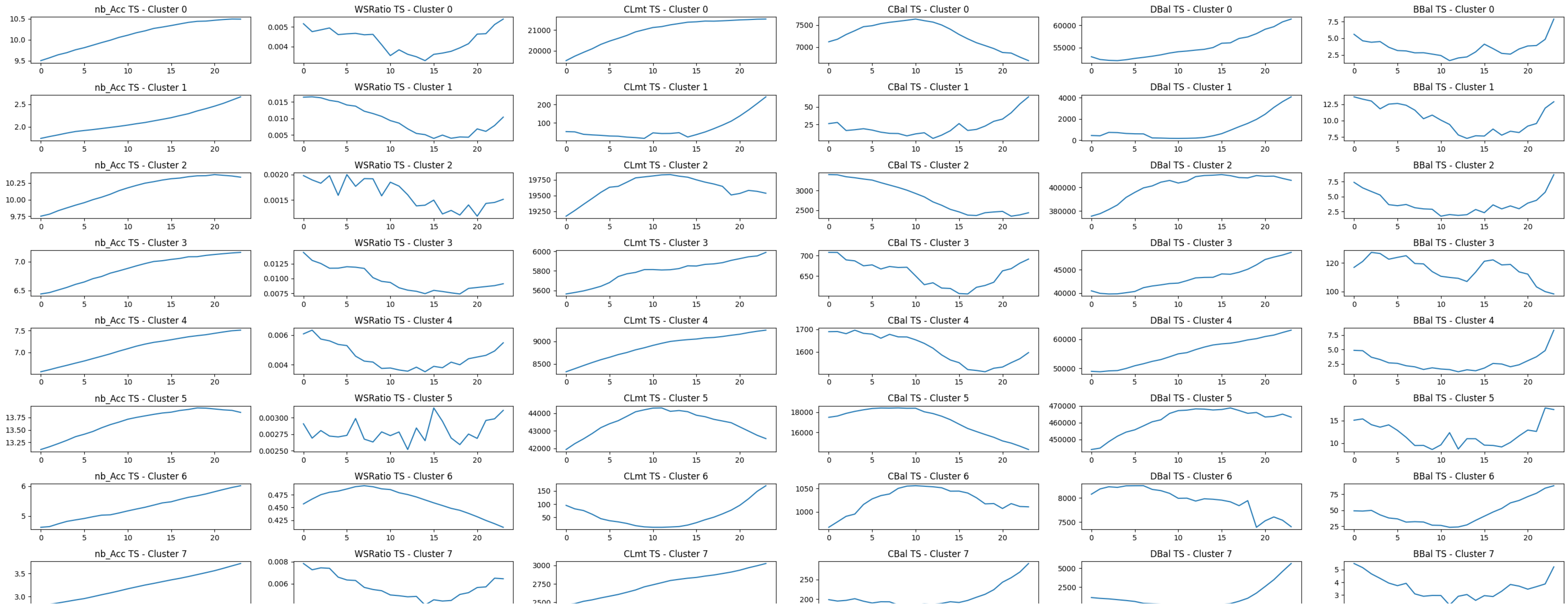
Out[37]:

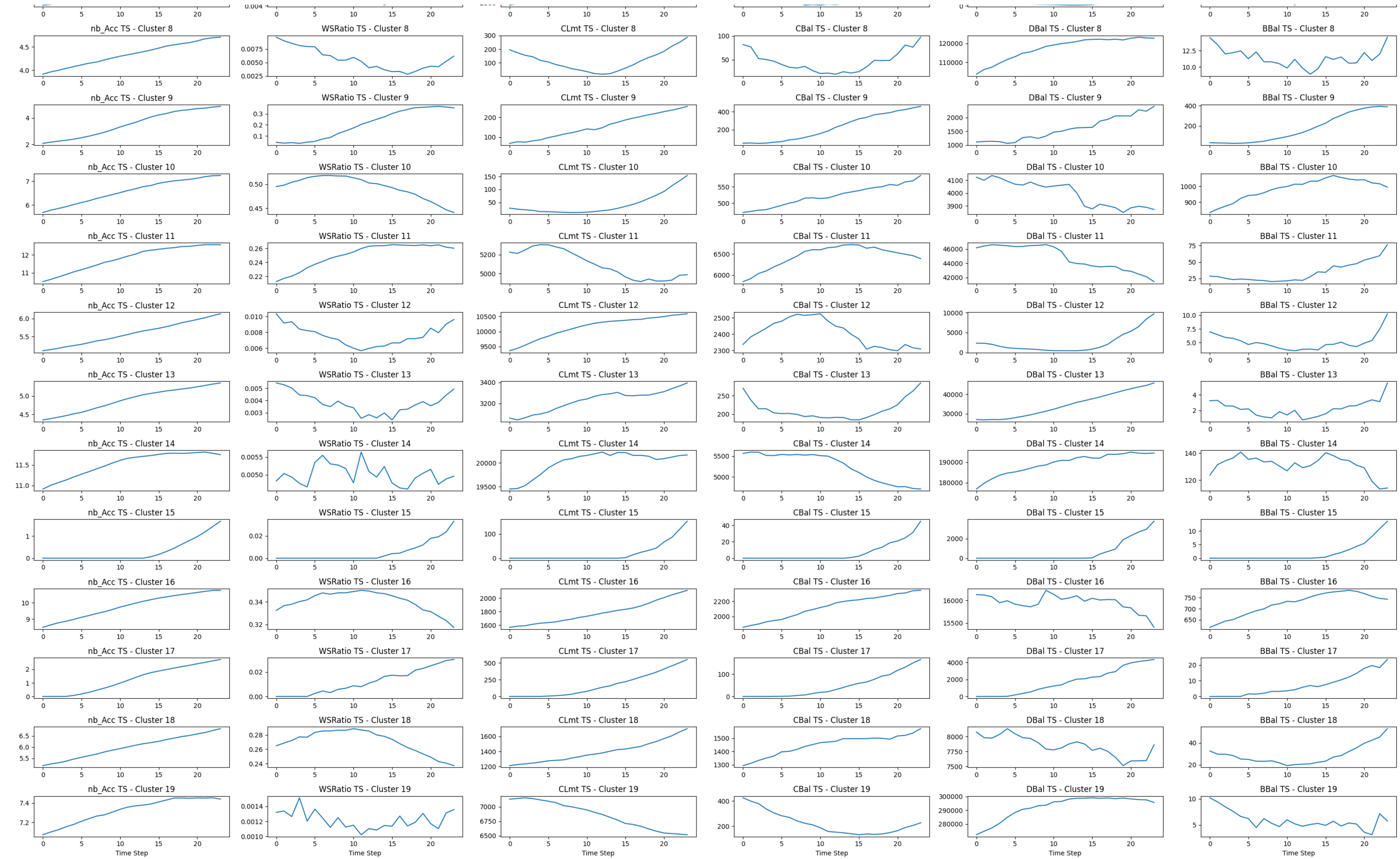
	Cluster Label	Bad Proportion	Cluster Proportion
0	0	0.091535	0.077740
1	1	0.170252	0.074857
2	2	0.016054	0.071652
3	3	0.101517	0.035628
4	4	0.075195	0.113347
5	5	0.049152	0.040058
6	6	0.616221	0.029026
7	7	0.086858	0.087778
8	8	0.068410	0.042546
9	9	0.655172	0.012258
10	10	0.737608	0.037131
11	11	0.550049	0.021915
12	12	0.109615	0.074680
13	13	0.062504	0.074471
14	14	0.072458	0.038988
15	15	0.272678	0.019641
16	16	0.680601	0.029181
17	17	0.272551	0.024852
18	18	0.545244	0.025425
19	19	0.010106	0.068827

A.3 Clustering Visualization - Visualisation of the Mean of Each Time Series Attribute for Each Cluster

In [38]:

```python
# Compute the mean of each time series attribute for each cluster
tmp1 = cluster_train[TS_cols + ['cluster_label']].groupby('cluster_label', as_index=False, sort=True).mean()

# Create a subplot grid with K rows (clusters) and len(month_attrbs) columns (attributes)
fig, axs = plt.subplots(K, len(month_attrbs), figsize=(30, 30))

# Iterate over each centroid (cluster) in the mean DataFrame
for _, centroid_i in tmp1.iterrows():
    # Iterate over each attribute
    for j, f in enumerate(month_attrbs):
        # Generate column names for the current attribute
        cols = [f'{f}_{i}' for i in range(TS_len)]

        # Plot the time series data for the current attribute and cluster
        axs[int(centroid_i['cluster_label']), j].plot(range(TS_len), centroid_i[cols].values[::-1])
        axs[int(centroid_i['cluster_label']), j].set_title(f"{f} TS - Cluster {int(centroid_i['cluster_label'])}")

# Set x-axis label for all subplots in the last row
for ax in axs[-1, :]:
    ax.set_xlabel("Time Step")

# Adjust layout to prevent overlap
plt.tight_layout()

# Display the plots
plt.show()
```





5. Classification - Random Forest

In [39]:
```
# Function to compute the average of lists using a specified function (e.g., mean)
def avg_list(Ls, f=np.mean):
    return [f([l[i] for l in Ls]) for i in range(len(Ls[0]))]

# List of features (distances to cluster centroids)
```

```python
features = cluster_dist_cols

# Lists to store ROC AUC scores
score_test = []
score_train = []

# Parameters for RandomForestClassifier
param = {
    'class_weight': {1: 0.8, 0: 0.2},
    'n_jobs': -1,
    'n_estimators': 100,
    'max_depth': 4,
    'min_samples_leaf': 4,
    'min_samples_split': 4,
    'bootstrap': True,
    'random_state': 217
}

# DataFrames to store predictions and true labels for training and test sets
results_train = pd.DataFrame()
results_test = pd.DataFrame()

# Initialize StratifiedKFold for cross-validation
kfold = StratifiedKFold(n_splits=10, random_state=217, shuffle=True)

# Perform cross-validation
for i, (itrain, itest) in enumerate(kfold.split(X=np.zeros(len(cluster_test)), y=cluster_test.target)):

    # Split data into training and test sets
    train = cluster_test.iloc[itrain]
    test = cluster_test.iloc[itest]

    # Standardize the features
    scaler = StandardScaler()
    train[features] = scaler.fit_transform(train[features])
    test[features] = scaler.transform(test[features])

    # Initialize and fit RandomForestClassifier
    model = RandomForestClassifier(**param)
    model.fit(train[features], train.target)

    # Predict probabilities for both training and test sets
    train_probs = model.predict_proba(train[features])[:, 1]
    test_probs = model.predict_proba(test[features])[:, 1]

    # Store predictions and true labels for each fold
    results_train = pd.concat([results_train, pd.DataFrame({'pred_target': train_probs, 'target': train.target, 'fold': i})])
    results_test = pd.concat([results_test, pd.DataFrame({'pred_target': test_probs, 'target': test.target, 'fold': i})])

    # Calculate and store ROC AUC scores
    score_train.append(roc_auc_score(y_true=train.target, y_score=train_probs))
    score_test.append(roc_auc_score(y_true=test.target, y_score=test_probs))

# Print mean and standard deviation of ROC AUC scores
print('Mean train score:', np.mean(score_train), 'StD train score:', np.std(score_train))
print('Mean test score:', np.mean(score_test), 'StD test score:', np.std(score_test))
```

```
# Display the first few rows of the results DataFrames
print(results_train.head())
print(results_test.head())

# Clean up memory
del test, train
gc.collect()
```

```
Mean train score: 0.8743088640293826 StD train score: 0.00040717180319O5919
Mean test score: 0.8727692183747775 StD test score: 0.003005355482644954З
        pred_target  target  fold
0          0.324843       0     0
1          0.707296       1     0
2          0.167528       1     0
3          0.305258       1     0
4          0.887850       1     0
...             ...     ...   ...
168209     0.175536       1     9
168210     0.391821       0     9
168211     0.721756       1     9
168212     0.568249       0     9
168213     0.207845       0     9

[1682136 rows x 3 columns]
       pred_target  target  fold
0         0.179912       0     0
1         0.402689       1     0
2         0.183486       0     0
3         0.180601       0     0
4         0.888499       0     0
...            ...     ...   ...
18685     0.798271       1     9
18686     0.238993       0     9
18687     0.130176       0     9
18688     0.135401       0     9
18689     0.144414       0     9

[186904 rows x 3 columns]
```

Out[39]: 280

In [41]:

```
def results_report(y_true_train, y_pred_prob_train, y_true_test, y_pred_prob_test, plab='Pos', nlab='Neg',
                   report_title='Model Performance Report', prob_threshold=0.5):
    """
    Generate a performance report for a classification model including confusion matrix, ROC curve, and metrics.

    Parameters:
    - y_true: True labels
    - y_pred_prob: Predicted probabilities
    - plab: Label for positive class (default 'Pos')
    - nlab: Label for negative class (default 'Neg')
    - report_title: Title of the report (default 'Model Performance Report')
    - w: Optional sample weights (default None)
    - model_name: Optional model name for logging metrics (default None)
    - prob_threshold: Threshold for converting probabilities to binary predictions (default 0.5)
    """

    def report(y_true, y_pred_prob, dataset_type):
```

```python
    """
    Creates and displays a performance report for a given dataset.

    Parameters:
    - y_true: True labels for the dataset
    - y_pred_prob: Predicted probabilities for the dataset
    - dataset_type: Type of dataset ('Training' or 'Test')
    """
    # Print the report title and dataset type
    print(f'----------------------------------------- {report_title} ({dataset_type}) -----------------------------------------')

    # Convert predicted probabilities to binary predictions based on the threshold
    y_pred = np.int8(y_pred_prob >= prob_threshold)

    # Create a figure for confusion matrix and ROC curve
    fig = plt.figure(figsize=(20, 6))

    # Plot the normalized confusion matrix
    plt.subplot(1, 2, 1)
    mc = confusion_matrix(y_true, y_pred)
    mc = mc.astype(float)
    mc[0, :] = mc[0, :] / (len(y_true) - sum(y_true))
    mc[1, :] = mc[1, :] / sum(y_true)
    heatmap = sns.heatmap(mc, annot=True, annot_kws={'size':12}, fmt='0.4f')
    heatmap.yaxis.set_ticklabels([nlab, plab], rotation=90, ha='right', fontsize=12)
    heatmap.xaxis.set_ticklabels([nlab, plab], rotation=0, ha='right', fontsize=12)
    plt.ylabel('True label')
    plt.xlabel('Predicted label')
    plt.title(f'{dataset_type} Confusion Matrix')

    # Plot the ROC curve
    plt.subplot(1, 2, 2)
    fpr, tpr, _ = roc_curve(y_true, y_pred_prob)
    roc_auc = auc(fpr, tpr)
    plt.plot(fpr, tpr, 'b', label=f'AUC = {roc_auc:.4f}')
    plt.plot([0, 1], [0, 1], 'r--')
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate (Sensitivity)')
    plt.xlabel('False Positive Rate (1-Specificity)')
    plt.title(f'{dataset_type} ROC')
    plt.legend(loc='lower right')

    # Calculate and print various performance metrics
    accuracy = accuracy_score(y_true, y_pred)
    precision = precision_score(y_true, y_pred)
    recall = recall_score(y_true, y_pred)
    f1 = f1_score(y_true, y_pred)
    gini_coefficient = 2 * roc_auc - 1
    table_data = [
        ['Accuracy', accuracy],
        ['Precision', precision],
        ['Recall', recall],
        ['F1 Score', f1],
        ['AUC', roc_auc],
        ['Gini Coefficient', gini_coefficient]
    ]
```

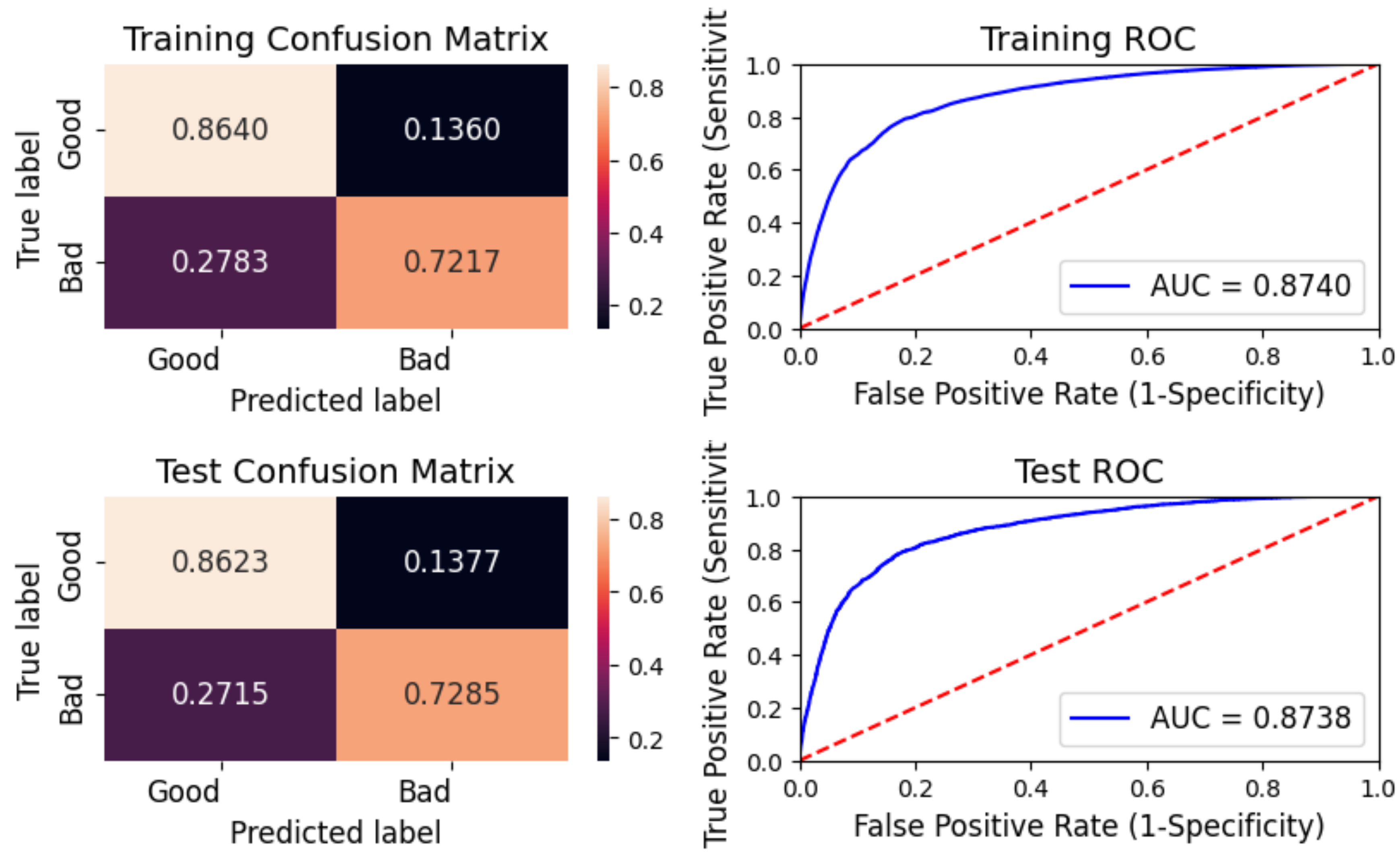
```python
        print(tabulate(table_data, headers=['Metric', 'Score'], tablefmt='fancy_grid'))

    # Generate reports for both training and test datasets
    report(y_true_train, y_pred_prob_train, 'Training')
    report(y_true_test, y_pred_prob_test, 'Test')

# The`results_train` and `results_test` are DataFrames with prediction results
results_train = pd.DataFrame({'pred_target': train_probs, 'target': cluster_test.iloc[itrain].target})
results_test = pd.DataFrame({'pred_target': test_probs, 'target': cluster_test.iloc[itest].target})

# Call the function to generate the performance report
results_report(results_train.target, results_train.pred_target, results_test.target, results_test.pred_target,
               plab='Bad', nlab='Good', report_title='Model Performance Report', prob_threshold=0.5)
```

```
----------------------------------------- Model Performance Report (Training) -----------------------------------------
```

Metric	Score
Accuracy	0.839377
Precision	0.526541
Recall	0.721679
F1 Score	0.608857
AUC	0.874035
Gini Coefficient	0.74807

```
----------------------------------------- Model Performance Report (Test) -----------------------------------------
```

Metric	Score
Accuracy	0.839165
Precision	0.525858
Recall	0.728536
F1 Score	0.610823
AUC	0.873783
Gini Coefficient	0.747566


Training Confusion Matrix
True label
Good
Bad
0.8640
0.1360
0.2783
0.7217
Predicted label
Training ROC
True Positive Rate (Sensitivit
False Positive Rate (1-Specificity)
AUC = 0.8740
Test Confusion Matrix
0.8623
0.1377
0.2715
0.7285
Test ROC
AUC = 0.8738