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 2

1. Import Libraries

In [1]:
```python
# Importing easy_peas3 for data acquisition and loading
import easy_peas3
from easy_peas3 import S3
from easy_peas3 import DerivedDataAssetTags

# Importing necessary libraries for data manipulation, visualization, 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 during runtime
import gc

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

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

# Importing preprocessing tools from scikit-learn
from sklearn.preprocessing import Normalizer
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import FunctionTransformer

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

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

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

# Setting up warnings to ignore any unnecessary warnings
import warnings
warnings.filterwarnings('ignore')

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

2. Data Loading

In [3]:
```python
# Create an S3 object and connect to the specified project bucket
s3 = S3(project_bucket='**')
bucket = s3.project_bucket()
```

In [ ]:
```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 [ ]:
```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.
0it [00:00, ?it/s]
['\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]:
```python
# 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]:
```python
# 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]:
```python
# 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]:
```python
# 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 [9]:
```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]}
    Def = month_df.Stat.values == 8 # Identify default accounts
    WS = (month_df.Stat.values > 0) & ~Def # Identify non-default accounts but arrears
    r['nb_Acc'] = len(month_df) # Total number of accounts
    r['ArrBal'] = np.sum(month_df.Bal.values, where=WS, initial=0) # Sum balances for arrear accounts
    r['DefBal'] = np.sum(month_df.Bal.values, where=Def, initial=0) # Sum balances for default accounts
    r['WSRatio'] = np.mean(month_df.Stat.values > 0) # Ratio of positive statuses
    r['TtlBal'] = np.sum(month_df.Bal.values, initial=0) # Total balance
    r['CLmt'] = r['CLU'] = 0 # Initialize credit limit and utilisation to 0

    # Iterate over the account classes
    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)  # Calculate total balance for the class
            if cls == 'C':
                cl = tmp.CLmt.values > 0
                r['CLmt'] = np.sum(tmp.CLmt.values, where=cl, initial=0)  # Calculate total credit limit
                # Calculate credit utilization if credit limit and balance are greater than 0, else set it to 0
                r['CLU'] = r['CBal'] / r['CLmt'] if r['CLmt'] > 0 and r['CBal'] > 0 else 0
        return r  # Return the aggregated results


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)

    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 acc.ACCOUNT_TYPE == 15 and np.any(acc_df.CLmt.values>20_000):
        continue

    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  ['CLmt', 'CLU', 'DBal', 'CBal', 'BBal', 'TtlBal', 'ArrBal', 'DefBal',
            'WSRatio', 'nb_Acc']:
        r[f'{f}_{m}'] = month_row[f]


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

D. Combined Data Statistics

In [11]:
```
# 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[11]:
```
(311418, 244)
```

E. Creation of Segments/Subsequences from Time Series for Each Consumer

In [12]:
```python
# List of attributes to calculate for each month
monthly_attributes = ['TtlBal', 'CLmt', 'CLU', 'DBal', 'CBal', 'BBal', 'ArrBal', 'DefBal', 'WSRatio', 'nb_Acc']

# Parameters for time series segmentation
time_series_length = 24  # Total number of months in the time series
segment_length = 6  # Length of each time series segment
step_size = 3  # Step size for sliding window

# Generate column names for each time series segment
time_series_columns = []
for month in range(segment_length):
    for feature in monthly_attributes:
        time_series_columns.append(f'{feature}_{month}')

# Generate column names for scaled features
scaled_time_series_columns = [f'{feature}_scaled' for feature in time_series_columns]

# Create a mapping from segment index to columns in that segment
map_time_series_columns = {}
for segment, index in enumerate(range(0, time_series_length - segment_length + 1, step_size)):
    columns = [f'{feature}_{j}' for j in range(index, index + segment_length) for feature in monthly_attributes]
    map_time_series_columns[segment] = columns

# Extract unique IDs, target values, and GB_FLAG from the original DataFrame
temp_id = combined_data.UNIQUEID.values
temp_target = combined_data.target.values
temp_GB_Flag = combined_data.GB_FLAG.values

# Initialize an empty DataFrame to store segmented time series data
segmented_time_series_data = pd.DataFrame()

# Populate the DataFrame with segmented time series data
for segment, columns in map_time_series_columns.items():
    temp_time_series = combined_data[columns].copy()
    temp_time_series.columns = time_series_columns
    temp_time_series['UNIQUEID'] = temp_id
    temp_time_series['target'] = temp_target
    temp_time_series['GB_FLAG'] = temp_GB_Flag
    temp_time_series['Segment'] = segment
    segmented_time_series_data = pd.concat([segmented_time_series_data, temp_time_series], ignore_index=True)

# Clean up temporary variables to free up memory
del temp_GB_Flag, temp_target
gc.collect()
```
Out[12]:
```
0
```
In [13]:
```python
# Segmented Data shape
segmented_time_series_data.shape
```
Out[13]:
```
(2179926, 64)
```
F. Data Partitioning

In [16]:
```
# Split unique IDs into training and testing sets, with 75% of the data as the test set
train_ids, test_ids = train_test_split(
    temp_id,                    # Array of unique IDs
    test_size=0.75,             # Proportion of data to be used for the test set
    stratify=combined_data['target'],  # Ensure that the distribution of target values is similar in both sets
    random_state=217            # Seed for reproducibility
)

# Create the training dataset by selecting rows with training IDs
cluster_train = segmented_time_series_data[segmented_time_series_data['UNIQUEID'].isin(train_ids)]

# Create the testing dataset by selecting rows with testing IDs
cluster_test = segmented_time_series_data[segmented_time_series_data['UNIQUEID'].isin(test_ids)]

# Clean up memory by running garbage collection
gc.collect()

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

Out[16]:
```
((544978, 64), (1634948, 64))
```

G. Time-Series Feature Scaling

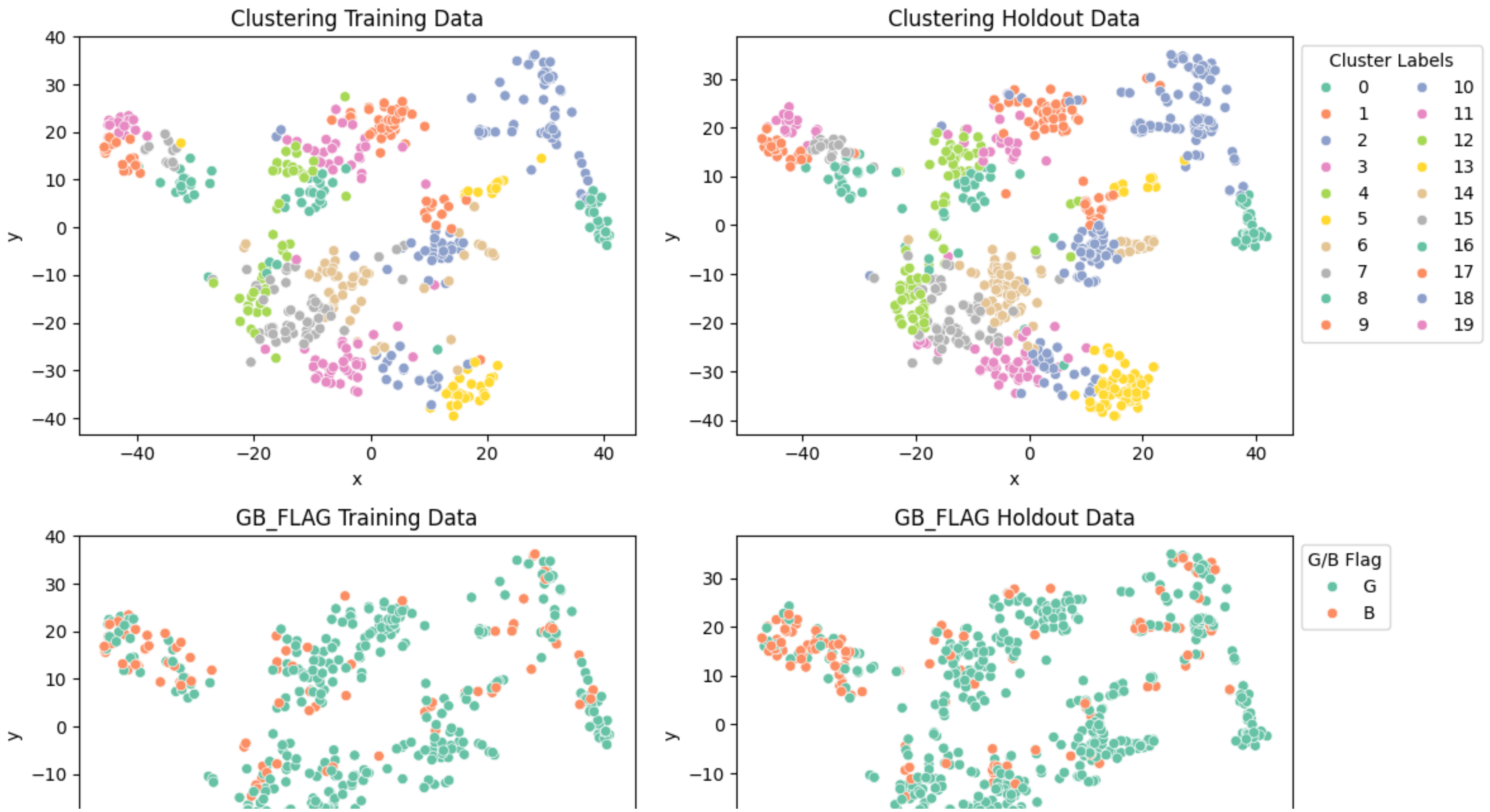
In [ ]:
```
# Define a lambda function to filter column names based on specified keywords
get_cols = lambda l1, l2: [f for f in l1 if np.any([x in f for x in l2])]

# Set up a ColumnTransformer to apply logarithmic transformations to selected columns
ct = ColumnTransformer(transformers=[
    # Apply log transformation to specific columns
    ('LogTransform1', FunctionTransformer(lambda x: np.log(1 + x)),
     get_cols(time_series_columns, ['TtlBal', 'CLmt', 'DBal', 'CBal', 'BBal', 'ArrBal', 'DefBal'])),

    # Apply scaled log transformation to 'CLU' column
    ('LogTransform2', FunctionTransformer(lambda x: np.log(1 + 100 * x)),
     get_cols(time_series_columns, ['CLU']))
], remainder='passthrough')

# Create a pipeline to first apply the ColumnTransformer, then standardize and normalize the data
scaler = Pipeline([
    ('CT', ct),          # Apply the ColumnTransformer for logarithmic transformations
    ('StanScale', StandardScaler()),  # Standardize features (mean=0, variance=1)
    ('Norm', Normalizer())  # Normalize each sample to unit norm
])

# Transform and scale the training dataset
cluster_train.loc[:, scaled_time_series_columns] = scaler.fit_transform(cluster_train[time_series_columns])

# Apply the same transformations to the test dataset
cluster_test.loc[:, scaled_time_series_columns] = scaler.transform(cluster_test[time_series_columns])
```

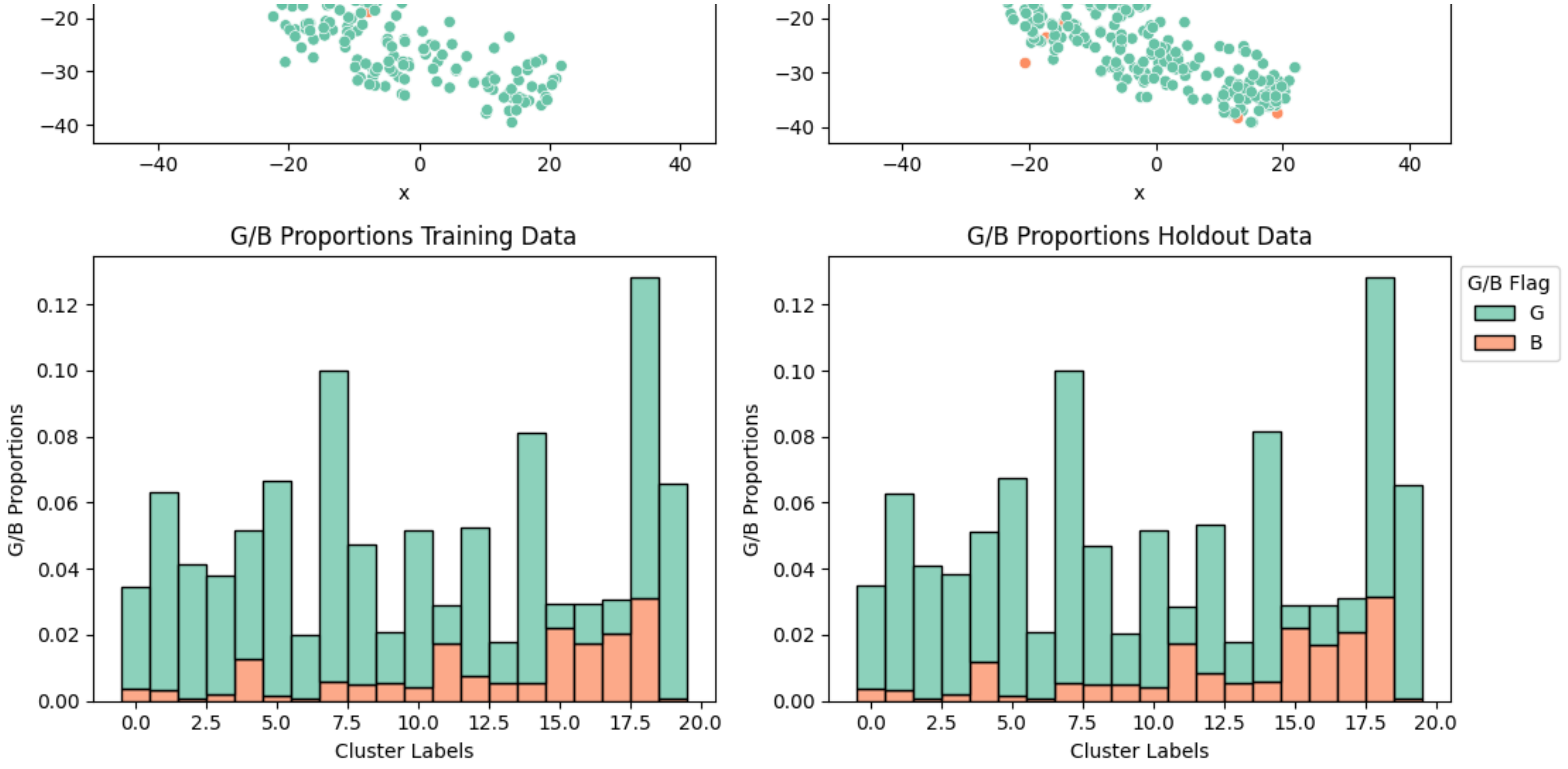
4. Time-Series Clustering

A. Time-Series Clustering and Visualizations

```
In [19]: # Define sample rate and number of clusters
sample_rate = 0.0005
K = 20

# Initialize KMeans clustering model
km_model = KMeans(n_clusters=K, max_iter=300, n_init=10, tol=0.0001, random_state=616)

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

# Retrieve cluster centroids for training data
centroids_train = km_model.cluster_centers_

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

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

# Compute and assign distances from each test sample to cluster centroids
cluster_test[cluster_dist_cols] = km_model.transform(cluster_test[TS_cols_scaled].values)



# Sample subsets of training and test data
tmp1 = cluster_train.sample(int(len(cluster_train) * sample_rate))  # Sample training data
tmp1['holdout'] = 0  # Indicate training data

tmp2 = cluster_test.sample(int(len(cluster_test) * sample_rate))  # Sample test data
tmp2['holdout'] = 1  # Indicate test data

# Combine sampled training and test data into a single DataFrame
tmp1 = pd.concat([tmp1, tmp2])



# Apply t-SNE for dimensionality reduction to 2D
tmp1[['x', 'y']] = TSNE(n_components=2, init='random', random_state=217).fit_transform(tmp1[scaled_time_series_columns])

# Create a figure with 3 rows and 2 columns for subplots
fig, ax = plt.subplots(3, 2, figsize=(12, 12))

# Scatter plot for training data clusters
sns.scatterplot(data=tmp1[tmp1.holdout == 0], x='x', y='y', hue='cluster_label', palette='Set2', hue_order=list(range(K)), legend=False, ax=ax[0, 0])
ax[0, 0].set_title('Clustering Training Data')

# Scatter plot for test data clusters
sns.scatterplot(data=tmp1[tmp1.holdout == 1], x='x', y='y', hue='cluster_label', palette='Set2', hue_order=list(range(K)), ax=ax[0, 1])
ax[0, 1].set_title('Clustering Holdout Data')
ax[0, 1].get_legend().set_title('Cluster Labels')
sns.move_legend(ax[0, 1], 'upper left', ncol=2, bbox_to_anchor=(1, 1))

# Scatter plot for GB_FLAG in training data
sns.scatterplot(data=tmp1[tmp1.holdout == 0], x='x', y='y', hue='GB_FLAG', palette='Set2', hue_order=['G', 'B'], legend=False, ax=ax[1, 0])
ax[1, 0].set_title('GB_FLAG Training Data')

# Scatter plot for GB_FLAG in test data
```

```python
sns.scatterplot(data=tmp1[tmp1.holdout == 1], x='x', y='y', hue='GB_FLAG', palette='Set2', hue_order=['G', 'B'], ax=ax[1, 1])
ax[1, 1].set_title('GB_FLAG Holdout Data')
ax[1, 1].get_legend().set_title('G/B Flag')
sns.move_legend(ax[1, 1], 'upper left', bbox_to_anchor=(1, 1))

# Histogram of G/B proportions for training data
sns.histplot(data=cluster_train, x='cluster_label', hue='GB_FLAG', palette='Set2', discrete=True, element='bars', multiple="stack", stat='proportion', hue_order=['G', '
ax[2, 0].set_title('G/B Proportions Training Data')
ax[2, 0].set(xlabel='Cluster Labels', ylabel='G/B Proportions')

# Histogram of G/B proportions for test data
sns.histplot(data=cluster_test, x='cluster_label', hue='GB_FLAG', palette='Set2', discrete=True, element='bars', multiple="stack", stat='proportion', hue_order=['G', 'B
ax[2, 1].set_title('G/B Proportions Holdout Data')
ax[2, 1].get_legend().set_title('G/B Flag')
sns.move_legend(ax[2, 1], 'upper left', bbox_to_anchor=(1, 1))

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





```
In [21]: # Group by 'cluster_label' in cluster_test and calculate the mean and count of 'target'
         result = cluster_test.groupby('cluster_label', as_index=False).agg({'target': ['mean', 'count']})

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

         # Normalize 'Cluster Proportion' to represent the proportion of each cluster in the dataset
         result['Cluster Proportion'] /= result['Cluster Proportion'].sum()

         # Display the resulting DataFrame
         result
```

Out[21]:

	Cluster Label	Bad Proportion	Cluster Proportion
0	0	0.102520	0.035164
1	1	0.053994	0.062620
2	2	0.017763	0.041147
3	3	0.058165	0.038308
4	4	0.235599	0.051221
5	5	0.021673	0.067619
6	6	0.036003	0.020896
7	7	0.056467	0.099848
8	8	0.103612	0.046972
9	9	0.245502	0.020534
10	10	0.080729	0.051808
11	11	0.606938	0.028526
12	12	0.154613	0.053196
13	13	0.304339	0.017875
14	14	0.070579	0.081686
15	15	0.759871	0.029030
16	16	0.585900	0.029055
17	17	0.676923	0.030910
18	18	0.246124	0.128109
19	19	0.014264	0.065477

B. Creation of Distance Features Between All Segments and Cluster Labels for Each Consumer

```
In [ ]: # Function to rename columns for each data segment
        rename_segment_columns = lambda segment, columns: {column: f'S{segment}_{column}' for column in columns}

        # Columns for distance data and cluster labels
        columns = cluster_dist_cols + ['cluster_label']

        # Select and rename relevant columns for the first data segment
        distance_data = cluster_test.loc[cluster_test.Segment == 0, ['UNIQUEID', 'target', 'GB_FLAG'] + columns]
        distance_data.rename(columns=rename_segment_columns(0, columns), inplace=True)

        # Merge distance data across all segments
        for segment in range(1, len(map_time_series_columns)):
            distance_data = distance_data.merge(
                cluster_test.loc[cluster_test.Segment == segment, ['UNIQUEID'] + columns].rename(columns=rename_segment_columns(segment, columns)),
                on='UNIQUEID', how='left')

        # List of all distance feature names
        distance_features = []
        for segment in range(len(map_time_series_columns)):
            for column in cluster_dist_cols:
                distance_features.append(f'S{segment}_{column}')
```

```
In [23]: # Shape of distance data
         distance_data.shape
```

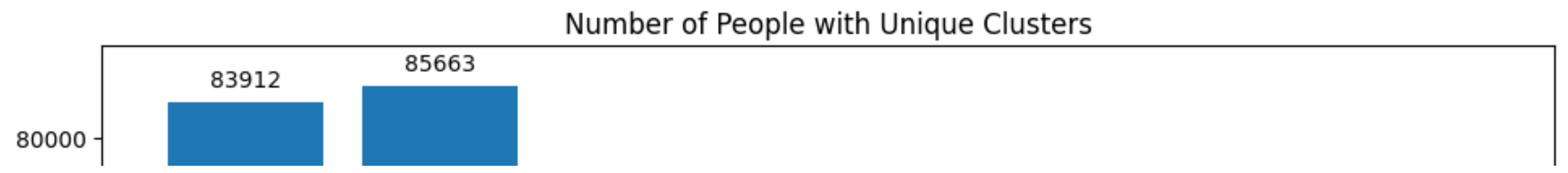
```
Out[23]: (233564, 150)
```

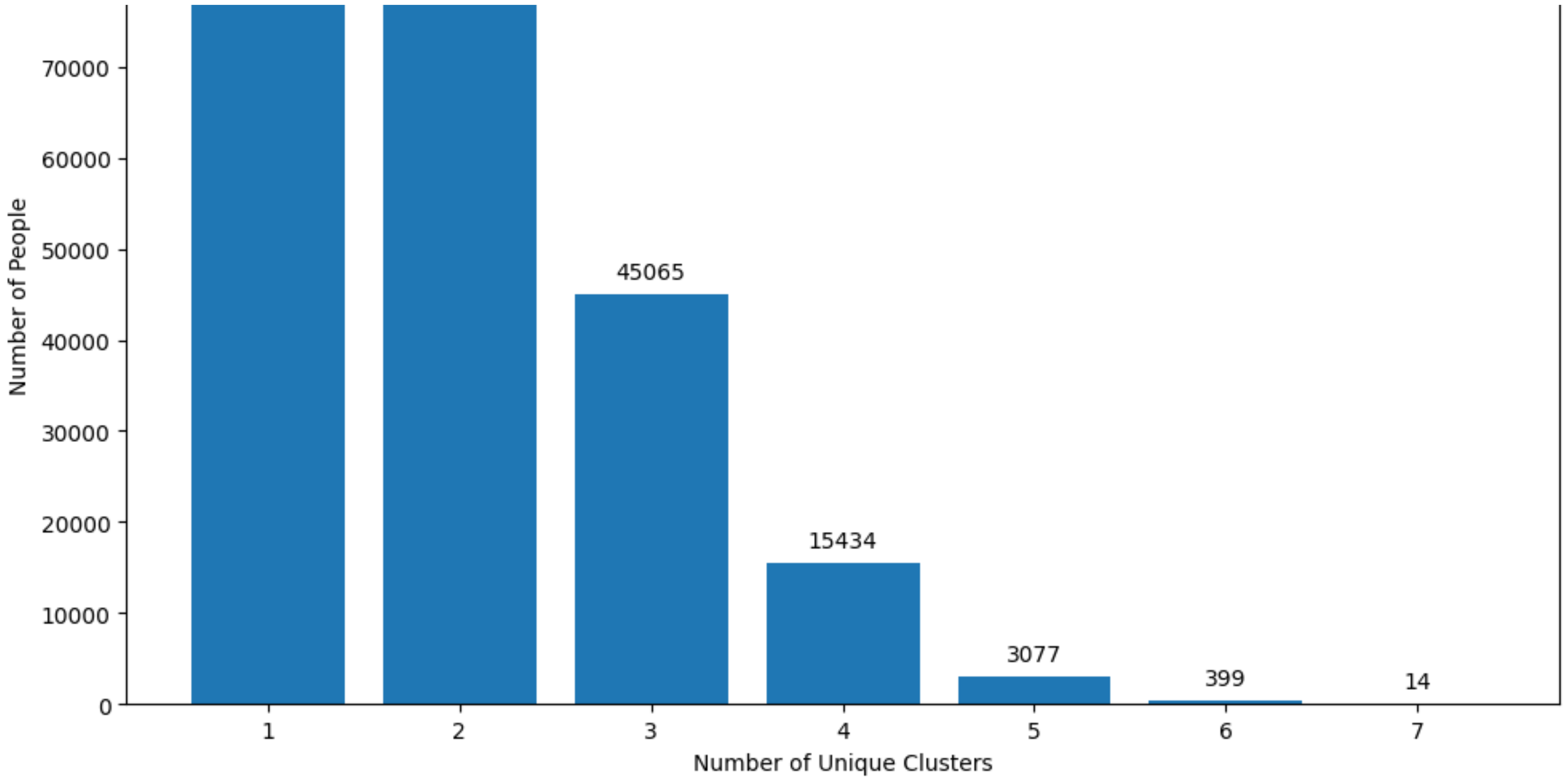
5. Consumer Journey and Transition Visualization

A. Analysis of Unique Cluster Movement Among Consumers

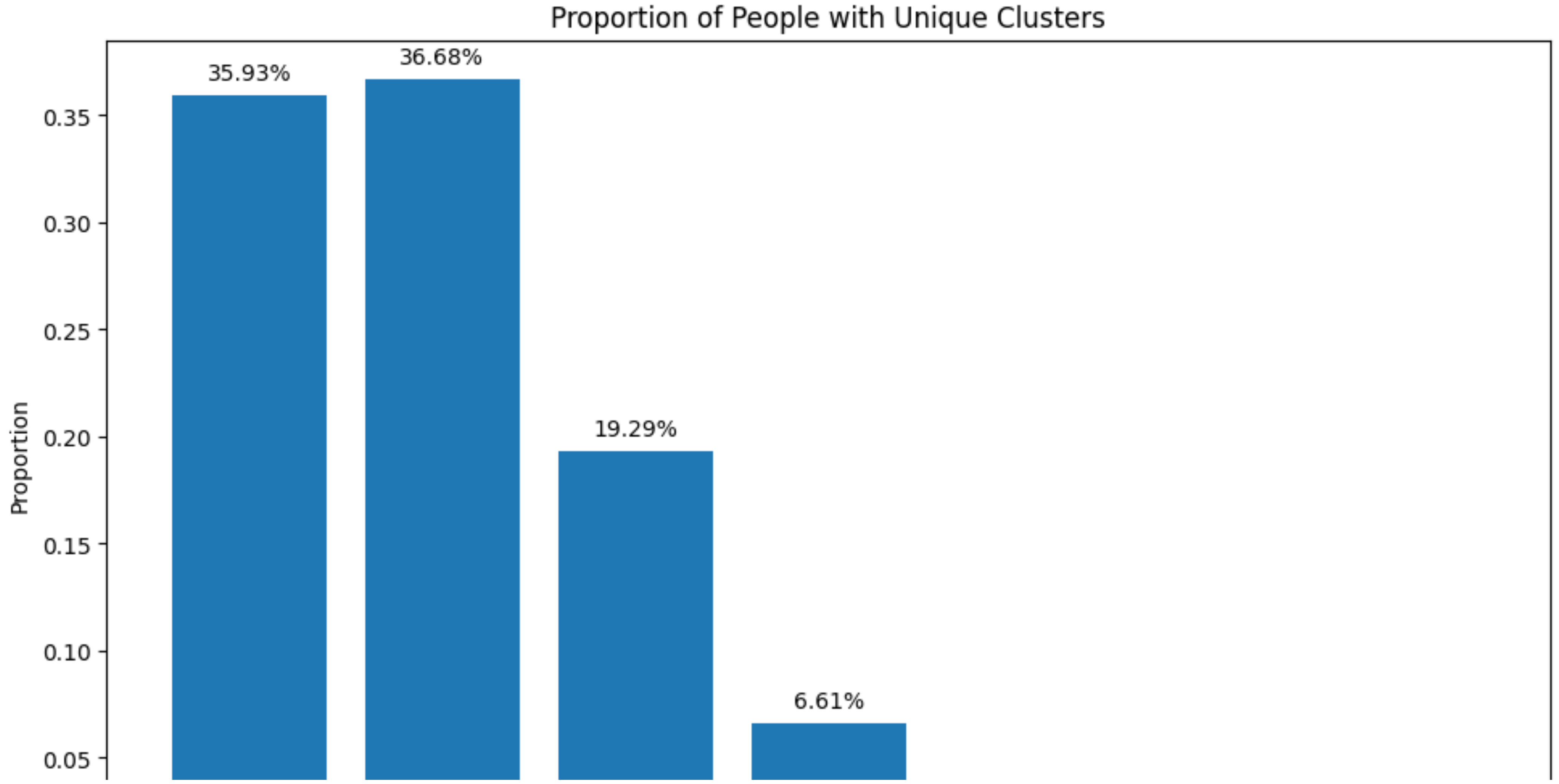
In [8]:
```python
# Extract cluster labels for each segment from the distance_data DataFrame
cluster_labels = distance_data.filter(regex='S[0-9]+_cluster_label')

# Count the number of unique clusters each person belongs to
num_unique_clusters = cluster_labels.nunique(axis=1)

# Calculate the average number of unique clusters across all individuals
average_unique_clusters = num_unique_clusters.mean()
print(f'Average number of unique clusters: {average_unique_clusters}')

# Count the number of people with each unique number of clusters
cluster_counts = num_unique_clusters.value_counts().sort_index()

# Calculate the proportion of people with each unique number of clusters
proportions = cluster_counts / cluster_counts.sum()

# Create a figure with two subplots for visualizing the clustering results
fig, axs = plt.subplots(2, figsize=(10, 12))

# Plot the number of people with each unique cluster count
axs[0].bar(cluster_counts.index, cluster_counts, color='skyblue')
axs[0].set_title('Number of People with Unique Clusters')
axs[0].set_xlabel('Number of Unique Clusters')
axs[0].set_ylabel('Number of People')

# Annotate bars with their values
for p in axs[0].patches:
    axs[0].annotate(str(p.get_height()),
                    (p.get_x() + p.get_width() / 2., p.get_height()),
                    ha='center', va='center', xytext=(0, 10),
                    textcoords='offset points')

# Plot the proportion of people with each unique cluster count
axs[1].bar(proportions.index, proportions, color='lightgreen')
axs[1].set_title('Proportion of People with Unique Clusters')
axs[1].set_xlabel('Number of Unique Clusters')
axs[1].set_ylabel('Proportion')

# Annotate bars with their percentage values
for p in axs[1].patches:
    percentage = '{:.2f}%'.format(100 * p.get_height())
    axs[1].annotate(percentage,
                    (p.get_x() + p.get_width() / 2, p.get_height()),
                    ha='center', va='center', xytext=(0, 10),
                    textcoords='offset points')

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

```
Average number of unique clusters: 2.0124933637033102
```





Number of People
70000
60000
50000
40000
30000
20000
10000
0
45065
15434
3077
399
14
1
2
3
4
5
6
7
Number of Unique Clusters

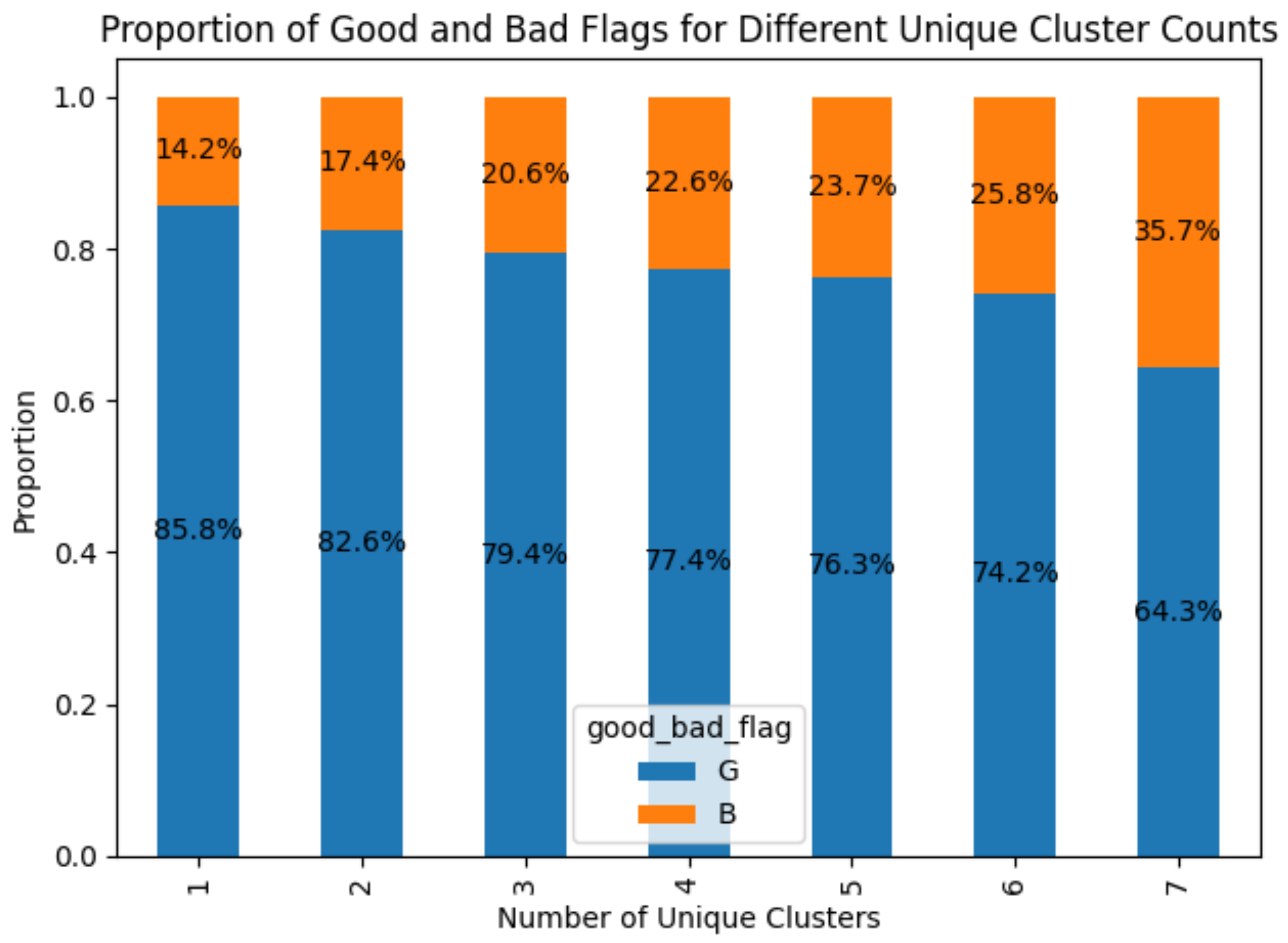


Proportion of People with Unique Clusters
Proportion
0.35
0.30
0.25
0.20
0.15
0.10
0.05
35.93%
36.68%
19.29%
6.61%

1.32%
0.17%
0.01%
0.00
1
2
3
4
5
6
7
Cluster

B. Analysis of G/B Flag Across Unique Cluster Movement Among Consumers

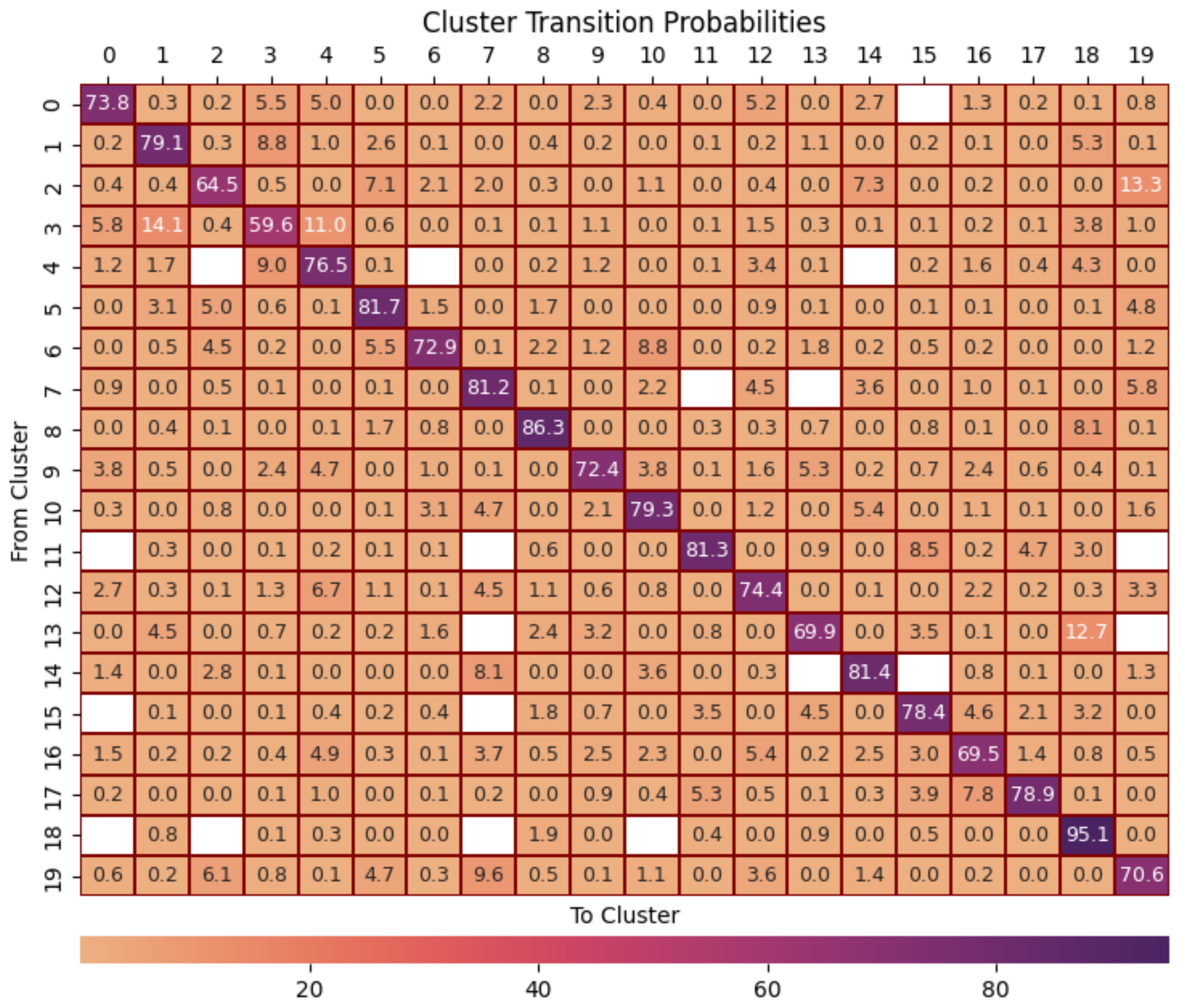
In [9]:
```python
# Extract 'GB_FLAG' column for good/bad flags
good_bad_flags = distance_data['GB_FLAG']

# Create a DataFrame containing the number of unique clusters and corresponding good/bad flags
df = pd.DataFrame({
    'num_unique_clusters': num_unique_clusters,
    'good_bad_flag': good_bad_flags
})

# Count occurrences of good and bad flags for each unique cluster count
flag_counts = df.groupby(['num_unique_clusters', 'good_bad_flag']).size().unstack(fill_value=0)

# Calculate the proportion of each flag type within each unique cluster count
flag_proportions = flag_counts.divide(flag_counts.sum(axis=1), axis=0)

# Plot the proportion of good and bad flags for each unique cluster count
# Reorder columns to switch the colors in the stacked bar plot
ax = flag_proportions[flag_proportions.columns[::-1]].plot(kind='bar', stacked=True)

# Set plot title and axis labels
plt.title('Proportion of Good and Bad Flags for Different Unique Cluster Counts')
plt.xlabel('Number of Unique Clusters')
plt.ylabel('Proportion')

# Annotate each bar with its proportion value
for p in ax.patches:
    width, height = p.get_width(), p.get_height()
    x, y = p.get_xy()
    ax.text(x + width / 2,
            y + height / 2,
            '{:.1f}%'.format(height * 100),
            horizontalalignment='center',
            verticalalignment='center')

# Adjust layout for better readability and display the plot
plt.tight_layout()
plt.show()
```

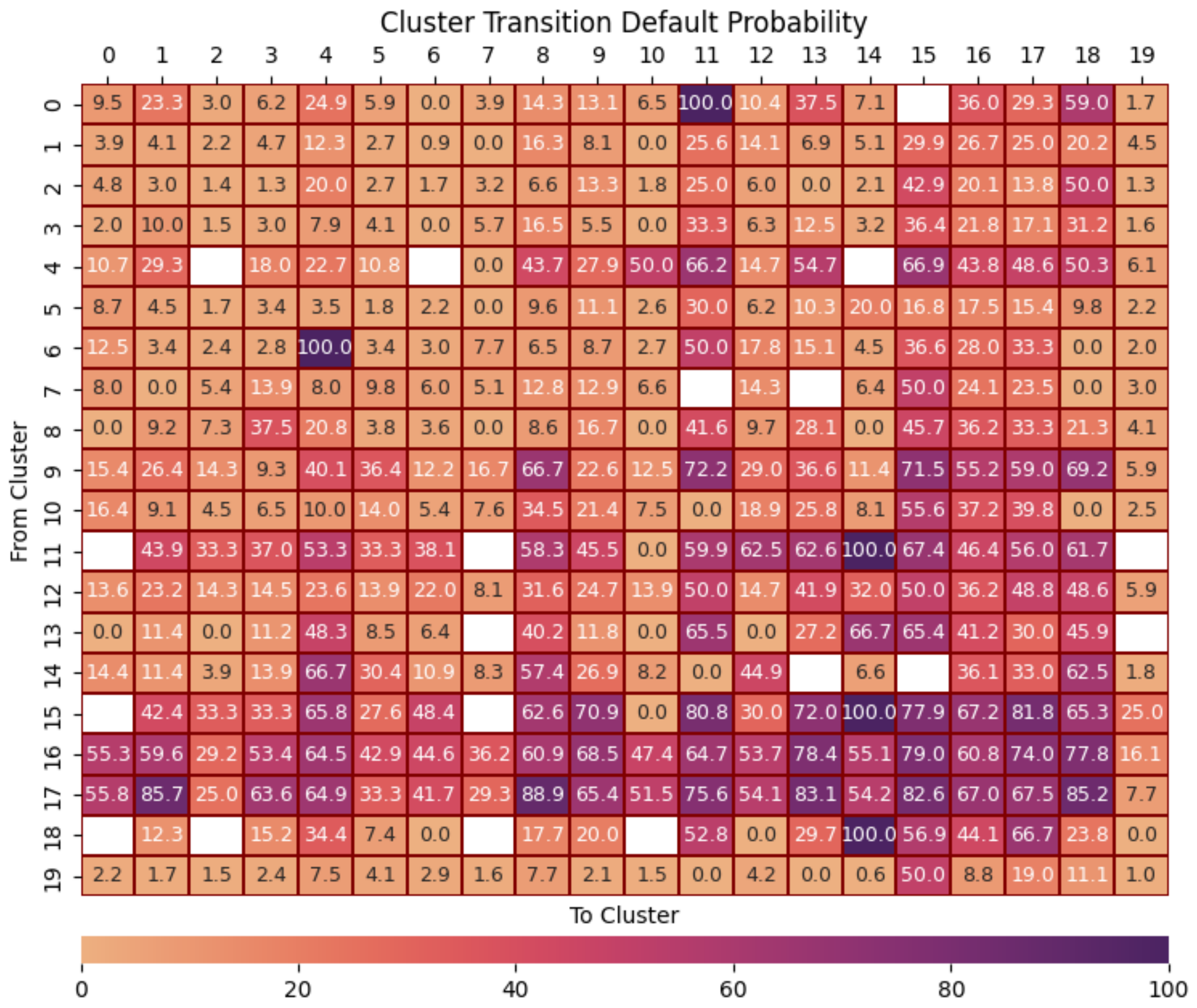


C. Heatmap of Consumer Transition and Default Transition

```
In [10]: # Extract cluster label columns and create pairs of consecutive labels
cluster_labels = sorted([f for f in distance_data.columns if 'label' in f])
labels_pairs = list(zip(cluster_labels[:-1], cluster_labels[1:]))

# Initialize an empty DataFrame for storing cluster transitions
trans_df = pd.DataFrame()

# Concatenate data for each pair of consecutive cluster labels
for c1, c2 in labels_pairs:
    # Extract relevant columns, rename them for clarity, and append to trans_df
    trans_df = pd.concat([trans_df, distance_data[[c1, c2, 'target']].rename(columns={c1: 'C0', c2: 'C1'})])

# Compute the frequency of transitions between clusters
trans_counts = pd.crosstab(index=trans_df.C0, columns=trans_df.C1)

# Calculate transition probabilities by normalizing the transition counts
trans_probs = trans_counts.div(trans_counts.sum(axis=1), axis=0)

# Create a figure with two subplots
fig, axes = plt.subplots(1, 2, figsize=(15, 7))

# Plot transition probabilities as a heatmap
sns.heatmap(
    trans_probs * 100,
    annot=True,
    fmt='.1f',
```

```python
    annot_kws={'fontsize': 9},
    cbar_kws={'orientation': 'horizontal', 'pad': 0.04, 'aspect': 40},
    linecolor='maroon',
    cbar=True,
    cmap='flare',
    linewidth=.01,
    ax=axes[0]
)
axes[0].set_title('Cluster Transition Probabilities')
axes[0].set_xlabel('To Cluster')
axes[0].set_ylabel('From Cluster')
axes[0].xaxis.tick_top()

# Compute mean default probability for transitions between clusters
trans_probs = pd.crosstab(index=trans_df.C0, columns=trans_df.C1, values=trans_df.target, aggfunc='mean')

# Plot transition default probabilities as a heatmap
sns.heatmap(
    trans_probs * 100,
    annot=True,
    fmt='.1f',
    cbar=True,
    cmap='flare',
    linewidth=.01,
    cbar_kws={'location': 'bottom', 'pad': 0.04, 'aspect': 40},
    linecolor='maroon',
    annot_kws={'fontsize': 9},
    ax=axes[1]
)
axes[1].set_title('Cluster Transition Default Probability')
axes[1].set_xlabel('To Cluster')
axes[1].set_ylabel('From Cluster')
axes[1].xaxis.tick_top()

# Adjust layout to fit subplots nicely
plt.tight_layout()
plt.show()
```



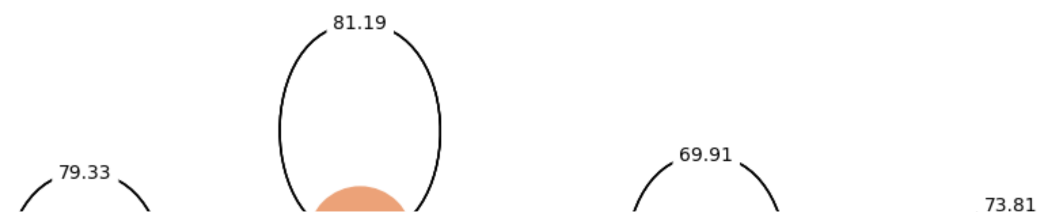


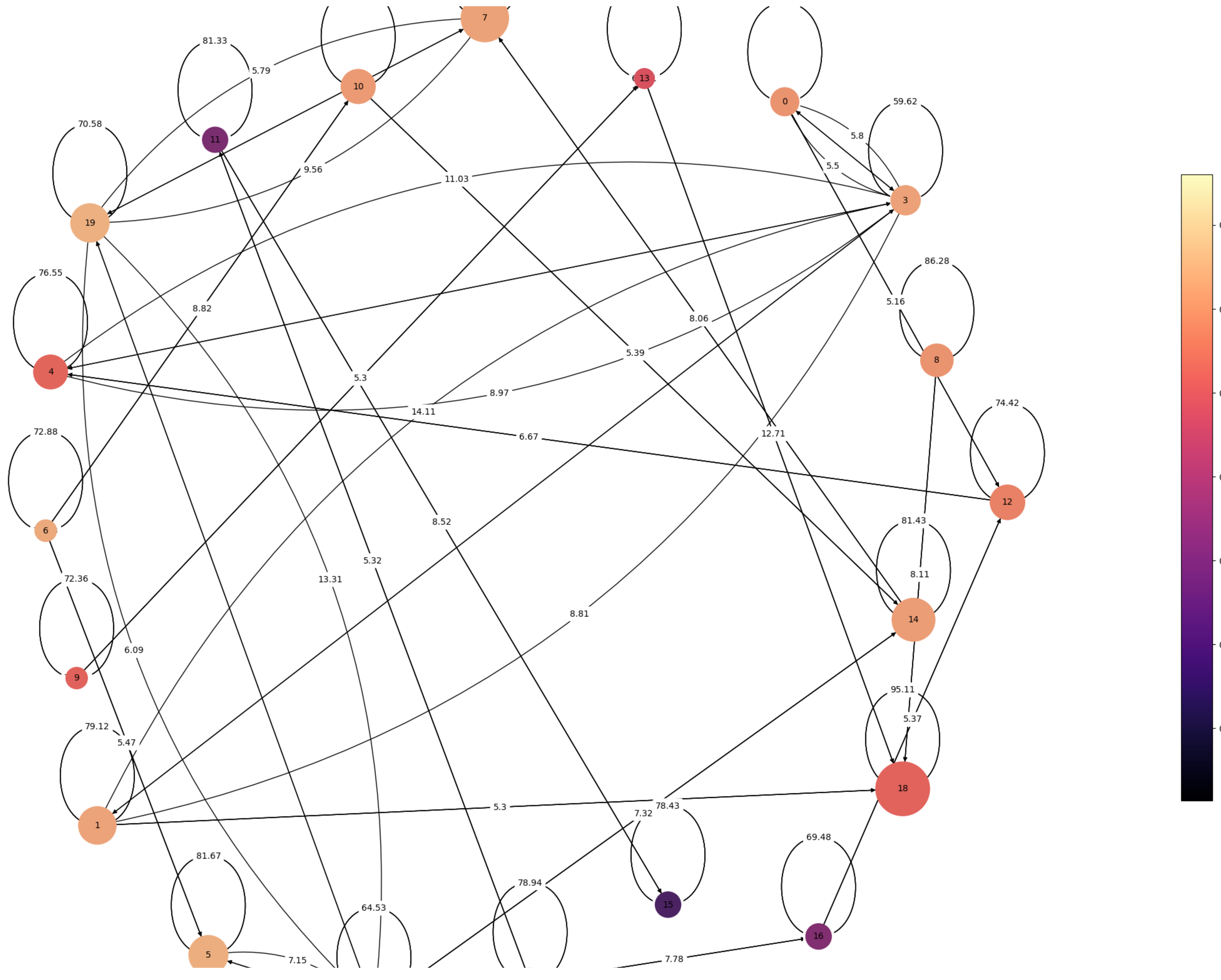
D. Network Graph

```
In [40]: # Initialize a directed graph
         G = nx.DiGraph()

         # Add nodes for each unique cluster
         for cluster in transition_probabilities.index:
             default_rate = default_rates_df.loc[cluster].mean()
             proportion = proportions_df.loc[cluster].mean() * 300
             # Add the node to the graph with size and color attributes
             G.add_node(cluster, size=proportion, color=default_rate, label=f"Cluster {cluster}")

         # Add edges between clusters with non-zero transition probabilities
         for source_cluster in transition_probabilities.index:
             for target_cluster in transition_probabilities.columns:
                 prob = transition_probabilities.loc[source_cluster, target_cluster]
                 if prob > 5.0:  # Filter out edges with very small probabilities
                     # Add an edge with the transition probability as the weight
                     G.add_edge(source_cluster, target_cluster, weight=prob.round(2))

         # Create a plot
         fig, ax = plt.subplots(figsize=(30, 26))
```

```python
# Position nodes using spring layout with increased 'k' parameter for better spacing
pos = nx.spring_layout(G, seed=616, k=7)

# Extract node colors and sizes from node attributes
node_colors = [G.nodes[node]['color'] for node in G]
node_sizes = [G.nodes[node]['size'] for node in G]

# Draw the graph
nodes = nx.draw(G, pos, with_labels=True, node_color=node_colors, node_size=node_sizes,
                cmap=plt.colormaps.get_cmap('flare'), font_size=10, ax=ax)

# Add a colorbar for default rates
sm = plt.cm.ScalarMappable(cmap=plt.cm.magma, norm=mcolors.Normalize(vmin=min(node_colors), vmax=max(node_colors)))
sm.set_array([])
fig.colorbar(sm, orientation='vertical', shrink=0.5, label='Default Rate', ax=ax)

# Separate edges into categories: curved, straight, and same-node edges
curved_edges = [edge for edge in G.edges() if reversed(edge) in G.edges()]
straight_edges = [edge for edge in G.edges() if edge not in curved_edges and edge[0] != edge[1]]
same_node_edges = [(u, v) for u, v in G.edges() if u == v]

# Draw edges
nx.draw_networkx_edges(G, pos, ax=ax, edgelist=straight_edges)
nx.draw_networkx_edges(G, pos, ax=ax, edgelist=same_node_edges)
nx.draw_networkx_edges(G, pos, ax=ax, edgelist=straight_edges)

# Draw curved edges with a specified arc radius
arc_rad = 0.25
nx.draw_networkx_edges(G, pos, ax=ax, edgelist=curved_edges, connectionstyle=f'arc3, rad = {arc_rad}')

# Retrieve edge weights for labels
edge_weights = nx.get_edge_attributes(G, 'weight')
curved_edge_labels = {edge: edge_weights[edge] for edge in curved_edges}
straight_edge_labels = {edge: edge_weights[edge] for edge in straight_edges}
same_node_labels = {edge: edge_weights[edge] for edge in same_node_edges}

# Draw edge labels
my_nx.my_draw_networkx_edge_labels(G, pos, ax=ax, edge_labels=curved_edge_labels, rotate=False, rad=arc_rad)
nx.draw_networkx_edge_labels(G, pos, ax=ax, edge_labels=straight_edge_labels, rotate=False)
nx.draw_networkx_edge_labels(G, pos, ax=ax, edge_labels=same_node_labels, rotate=False)

# Set plot title
plt.title("Transition Probability Graph - With Default Rate and Node Size based on Proportion (All Clusters)")

# Show plot
plt.show()
```

Transition Probability Graph - With Defualt Rate and Node Size based on Proportion (All Clusters)





Default Rate
0.1
0.2
0.3
0.4
0.5
0.6
0.7

6. Classification - Logistic Regression

In [23]:
```python
# Function to calculate the average of lists element-wise
def avg_list(Ls, f=np.mean):
    n = range(len(Ls[0]))
    return [f([l[i] for l in Ls]) for i in n]

# Initialize lists to store feature importance and test AUC scores
feature_importance = []
score_test = []

# Logistic Regression parameters
param = {
    'C': 0.5,
    'class_weight': 'balanced',
    'solver': 'sag',
    'penalty': 'l2',
    'max_iter': 100,
    'n_jobs': -1
}

# DataFrame to store predictions and actual values
results = pd.DataFrame()

# Cross-validation setup with 10 folds
kfold = StratifiedKFold(n_splits=10, random_state=251, shuffle=True)

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

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

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

    # Initialize and train the Logistic Regression model
    model = LogisticRegression(**param)
    model.fit(train[distance_features], train.target)

    # Predict probabilities and store results
    t = model.predict_proba(test[distance_features])[:, 1]
    results = pd.concat([results, pd.DataFrame({'pred_target': t, 'target': test.target.values, 'fold': i})])
    # Compute and store performance metrics
    score_test.append(roc_auc_score(y_true=test.target, y_score=t))
    feature_importance.append([model.intercept_[0]] + list(model.coef_[0]))

    # Print AUC scores for the current fold
    print(['Test AUC:', score_test[-1], '- Train AUC:', roc_auc_score(y_true=train.target, y_score=model.predict_proba(train[distance_features])[:, 1])])

# Output mean and standard deviation of AUC scores, and Gini coefficient
mean_score = np.mean(score_test)
print(['Mean score:', mean_score, 'StD:', np.std(score_test), 'Gini', 2 * mean_score - 1])

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

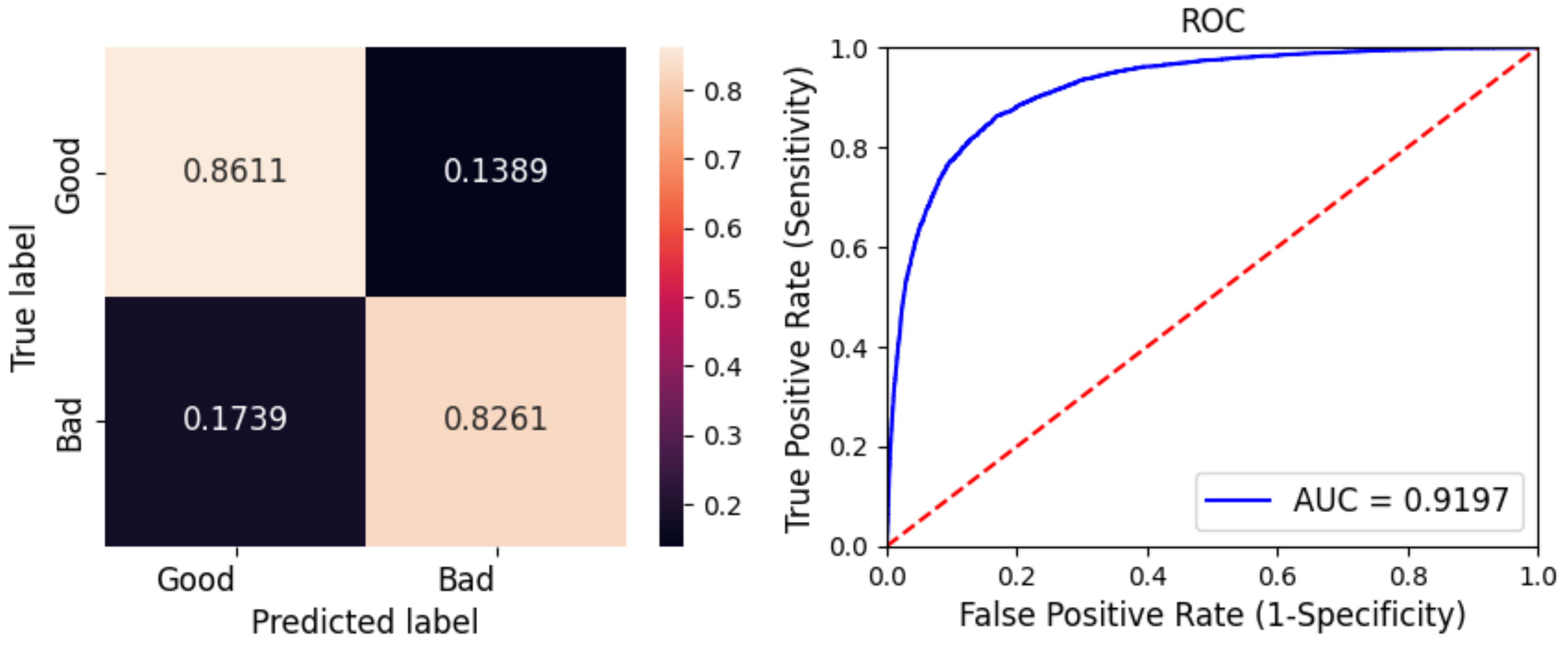
```
Test AUC: 0.90449 - Train AUC: 0.91957
Test AUC: 0.91409 - Train AUC: 0.91844
Test AUC: 0.90999 - Train AUC: 0.91883
Test AUC: 0.91331 - Train AUC: 0.91821
Test AUC: 0.93036 - Train AUC: 0.91640
Test AUC: 0.91248 - Train AUC: 0.91858
Test AUC: 0.92633 - Train AUC: 0.91700
Test AUC: 0.91012 - Train AUC: 0.91875
Test AUC: 0.91487 - Train AUC: 0.91814
Test AUC: 0.91111 - Train AUC: 0.91877
Mean score: 0.91472 StD: 0.00740 Gini 0.82943
```

```
Out[23]: 56
```

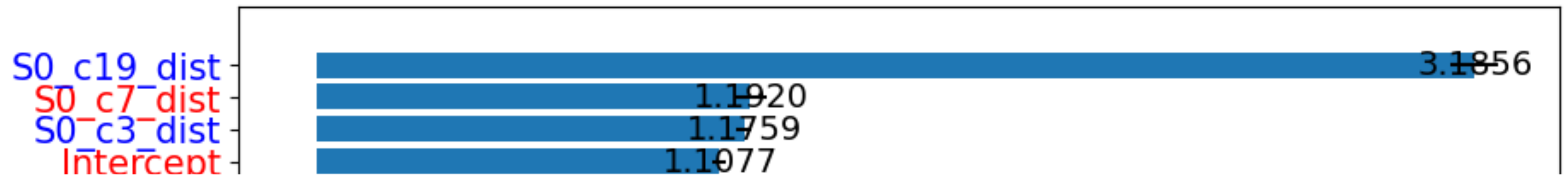
```
In [29]: def results_report(y_true, y_pred_prob, plab='Pos', nlab='Neg', report_title='Model Performance Report',
                        w=None, model_name=None, prob_threshold=0.5):
    """
    Generate a performance report for a classification model including confusion matrix, ROC curve, and metrics.

    Parameters:
    - y_true: Array-like of true labels
    - y_pred_prob: Array-like of 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)
    """

    # Print the report title
    print(f'----------------------------------------- {report_title} -----------------------------------------')

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

    # Create a figure for the plots
    fig = plt.figure(figsize=(10, 3.5))

    # Plot the normalized confusion matrix
    plt.subplot(1, 2, 1)
    mc = confusion_matrix(y_true, y_pred)
```

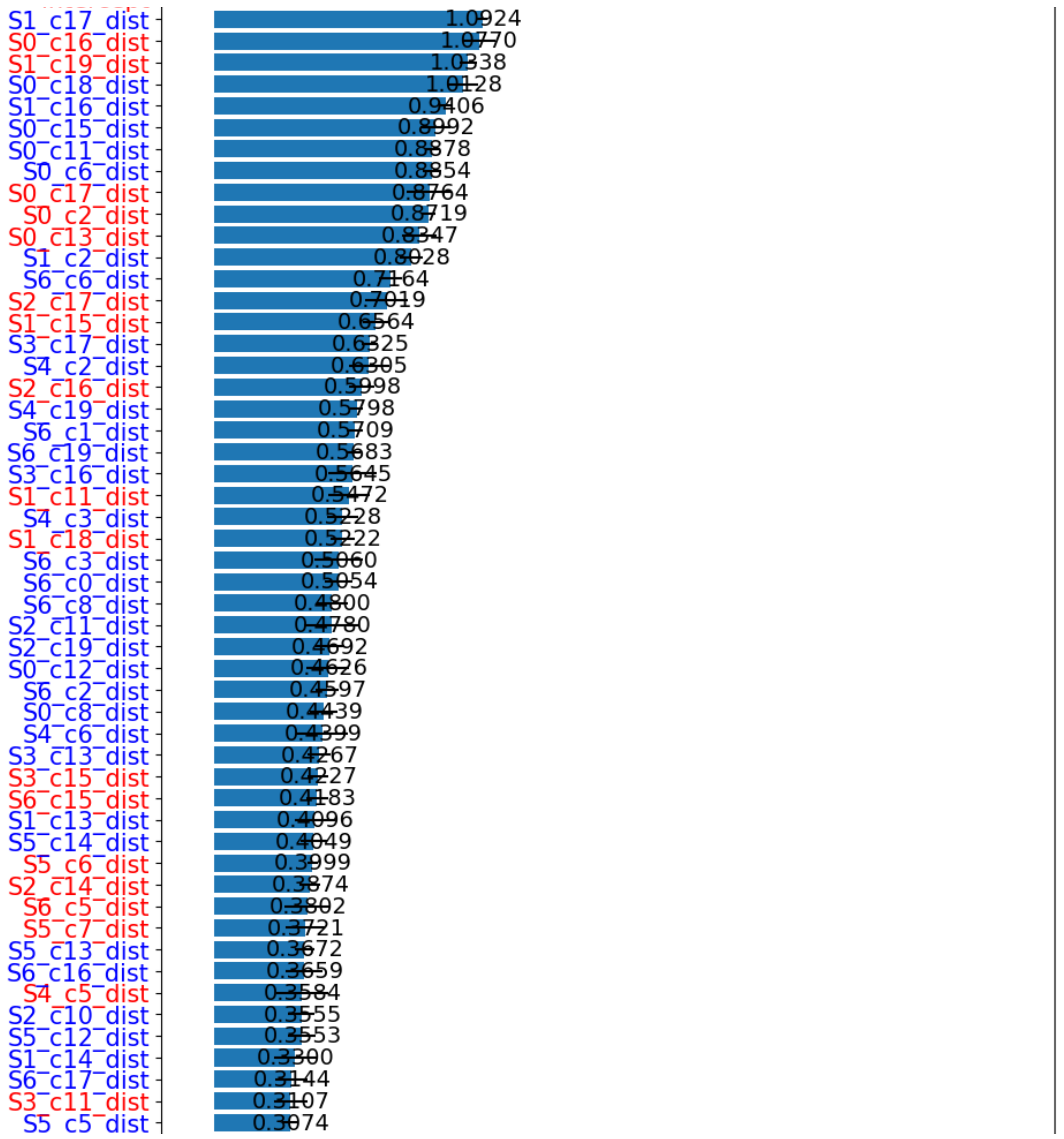
```python
    mc = mc.astype(float)
    mc[0, :] /= (len(y_true) - sum(y_true))  # Normalize negative class
    mc[1, :] /= sum(y_true)  # Normalize positive class

    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', fontsize=12)
    plt.xlabel('Predicted label', fontsize=12)

    # Plot the ROC curve
    plt.subplot(1, 2, 2)
    fpr, tpr, _ = roc_curve(y_true, y_pred_prob, sample_weight=w)
    roc_auc = auc(fpr, tpr)

    print(f'Accuracy: {accuracy_score(y_true, y_pred, sample_weight=w):.5f}', end='\t')
    print(f'Precision: {precision_score(y_true, y_pred, sample_weight=w):.5f}', end='\t')
    print(f'Recall: {recall_score(y_true, y_pred, sample_weight=w):.5f}', end='\t')
    print(f'F1 Score: {f1_score(y_true, y_pred, sample_weight=w):.5f}', end='\t')
    print(f'AUC: {roc_auc:.5f}', end='\t')
    print(f'Gini Coefficient: {(2 * roc_auc - 1):.5f}')

    # Log metrics if a model name is provided
    if model_name:
        logvalue(f'{model_name}_auc', roc_auc)
        logvalue(f'{model_name}_gini', 2 * roc_auc - 1)

    # Plot the ROC curve
    plt.title('ROC Curve', fontsize=12)
    plt.plot(fpr, tpr, 'b', label=f'AUC = {roc_auc:.4f}')
    plt.plot([0, 1], [0, 1], 'r--')  # Diagonal line
    plt.xlim([0, 1])
    plt.ylim([0, 1])
    plt.ylabel('True Positive Rate (Sensitivity)', fontsize=12)
    plt.xlabel('False Positive Rate (1-Specificity)', fontsize=12)
    plt.legend(loc='lower right', fontsize=12)

    plt.tight_layout()
    plt.show()

# Example call to the function
results_report(results.target, results.pred_target, plab='Bad', nlab='Good',
               report_title='Model Performance Report', prob_threshold=0.5)
```

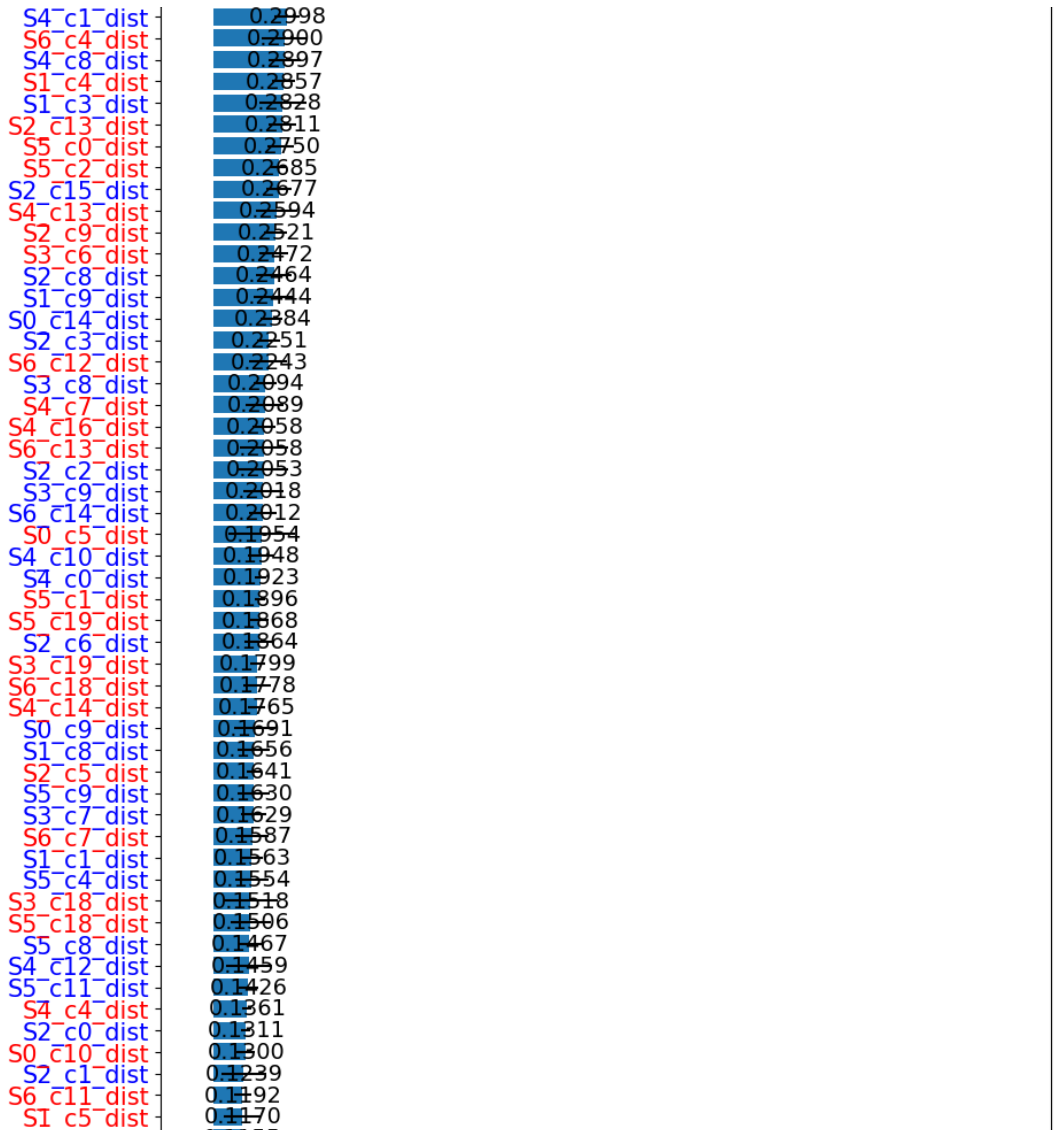
```
----------------------------------------- Model Performance Report -----------------------------------------
Accuracy Scores: 0.85453        Precision Scores: 0.58029       Recall Scores: 0.82611  F1 Scores: 0.68172      AUC: 0.91967    Gini Coefficient: 0.83933
```

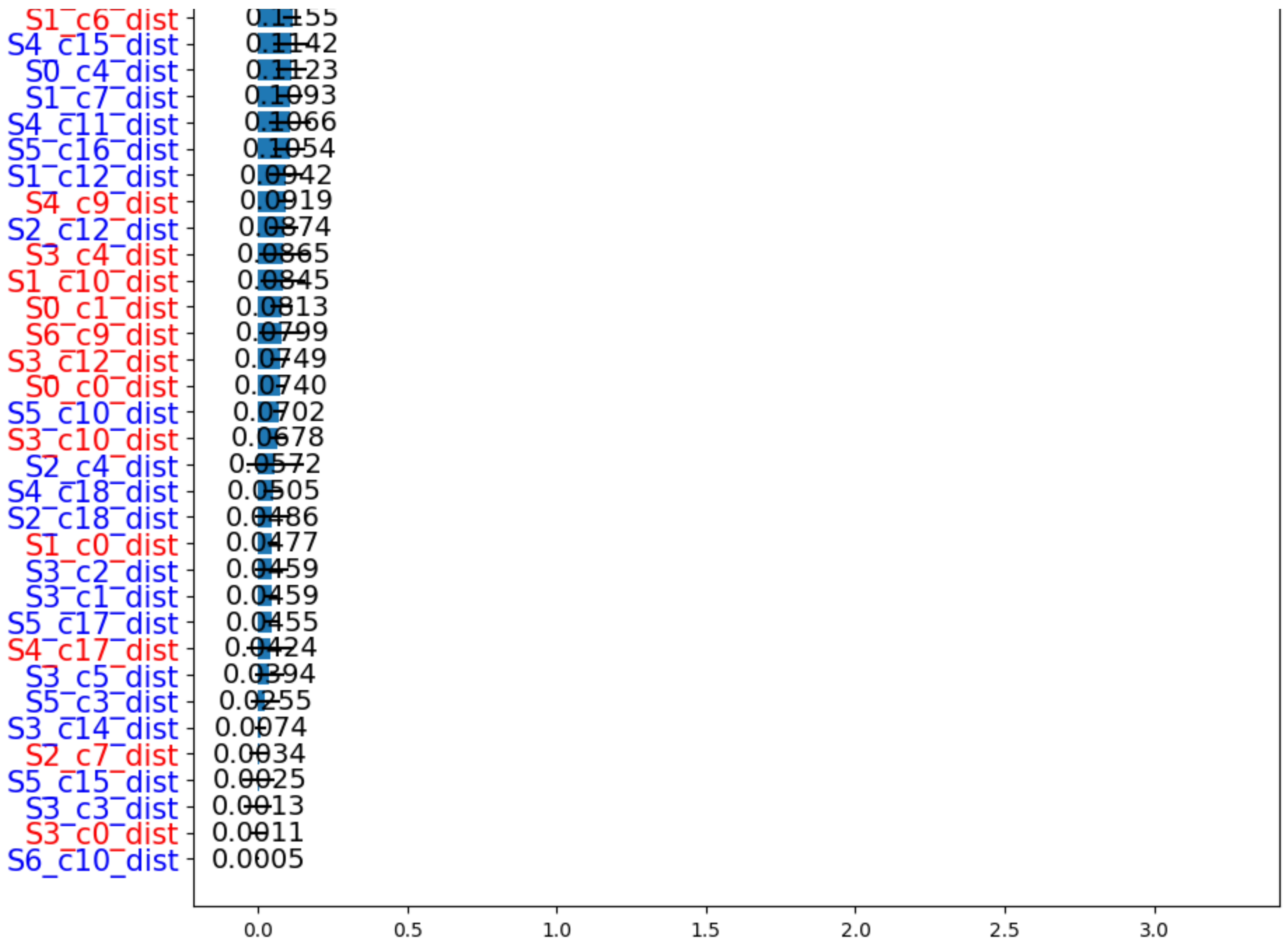


In [14]:
```
# Calculate average feature importance and sort along with feature names, including intercept
fi1 = sorted(zip(avg_list(feature_importance), ['Intercept'] + distance_features), key=lambda x: abs(x[0]))
fi = [(abs(x), f) for (x, f) in fi1]  # Use absolute values for visualization

# Create a horizontal bar plot
fig, ax = plt.subplots(figsize=(10, len(distance_features)//4))

# Plot feature importances with error bars representing standard deviation
ax.barh(range(len(fi)), [x[0] for x in fi], xerr=avg_list(feature_importance, np.std), color=['b' if x[1] >= 0 else 'r' for x in fi])
ax.set_yticks(range(len(fi)))
ax.set_yticklabels([x[1] for x in fi], fontsize=15)
plt.margins(y=0.01)

# Add labels to the end of each bar
rects = ax.patches
labels = ['%0.4f' % x for x, _ in fi]
for rect, label in zip(rects, labels):
    width = rect.get_width()
    height = rect.get_height()
    ax.text(width + 0.003, rect.get_y() + height / 2, label, ha='center', va='center', fontsize=14)

# Color-code the feature names based on the sign of their importance values
for i, tick in enumerate(ax.get_yticklabels()):
    tick.set_color('r' if fi1[i][0] < 0 else 'b')

# Display the plot
plt.title('Feature Importances with Error Bars', fontsize=16)
plt.show()
```




S1_c17_dist 1.0924
S0_c16_dist 1.0770
S1_c19_dist 1.0338
S0_c18_dist 1.0128
S1_c16_dist 0.9406
S0_c15_dist 0.8992
S0_c11_dist 0.8878
S0_c6_dist 0.8854
S0_c17_dist 0.8764
S0_c2_dist 0.8719
S0_c13_dist 0.8347
S1_c2_dist 0.8028
S6_c6_dist 0.7164
S2_c17_dist 0.7019
S1_c15_dist 0.6564
S3_c17_dist 0.6325
S4_c2_dist 0.6305
S2_c16_dist 0.5998
S4_c19_dist 0.5798
S6_c1_dist 0.5709
S6_c19_dist 0.5683
S3_c16_dist 0.5645
S1_c11_dist 0.5472
S4_c3_dist 0.5228
S1_c18_dist 0.5222
S6_c3_dist 0.5060
S6_c0_dist 0.5054
S6_c8_dist 0.4800
S2_c11_dist 0.4780
S2_c19_dist 0.4692
S0_c12_dist 0.4626
S6_c2_dist 0.4597
S0_c8_dist 0.4439
S4_c6_dist 0.4399
S3_c13_dist 0.4267
S3_c15_dist 0.4227
S6_c15_dist 0.4183
S1_c13_dist 0.4096
S5_c14_dist 0.4049
S5_c6_dist 0.3999
S2_c14_dist 0.3874
S6_c5_dist 0.3802
S5_c7_dist 0.3721
S5_c13_dist 0.3672
S6_c16_dist 0.3659
S4_c5_dist 0.3584
S2_c10_dist 0.3555
S5_c12_dist 0.3553
S1_c14_dist 0.3300
S6_c17_dist 0.3144
S3_c11_dist 0.3107
S5_c5_dist 0.3074


S4_c1_dist 0.2998
S6_c4_dist 0.2900
S4_c8_dist 0.2897
S1_c4_dist 0.2857
S1_c3_dist 0.2828
S2_c13_dist 0.2811
S5_c0_dist 0.2750
S5_c2_dist 0.2685
S2_c15_dist 0.2677
S4_c13_dist 0.2594
S2_c9_dist 0.2521
S3_c6_dist 0.2472
S2_c8_dist 0.2464
S1_c9_dist 0.2444
S0_c14_dist 0.2384
S2_c3_dist 0.2251
S6_c12_dist 0.2243
S3_c8_dist 0.2094
S4_c7_dist 0.2089
S4_c16_dist 0.2058
S6_c13_dist 0.2058
S2_c2_dist 0.2053
S3_c9_dist 0.2018
S6_c14_dist 0.2012
S0_c5_dist 0.1954
S4_c10_dist 0.1948
S4_c0_dist 0.1923
S5_c1_dist 0.1896
S5_c19_dist 0.1868
S2_c6_dist 0.1864
S3_c19_dist 0.1799
S6_c18_dist 0.1778
S4_c14_dist 0.1765
S0_c9_dist 0.1691
S1_c8_dist 0.1656
S2_c5_dist 0.1641
S5_c9_dist 0.1630
S3_c7_dist 0.1629
S6_c7_dist 0.1587
S1_c1_dist 0.1563
S5_c4_dist 0.1554
S3_c18_dist 0.1518
S5_c18_dist 0.1506
S5_c8_dist 0.1467
S4_c12_dist 0.1459
S5_c11_dist 0.1426
S4_c4_dist 0.1361
S2_c0_dist 0.1311
S0_c10_dist 0.1300
S2_c1_dist 0.1239
S6_c11_dist 0.1192
S1_c5_dist 0.1170


S1_c6_dist 0.1155
S4_c15_dist 0.1142
S0_c4_dist 0.1123
S1_c7_dist 0.1093
S4_c11_dist 0.1066
S5_c16_dist 0.1054
S1_c12_dist 0.0942
S4_c9_dist 0.0919
S2_c12_dist 0.0874
S3_c4_dist 0.0865
S1_c10_dist 0.0845
S0_c1_dist 0.0813
S6_c9_dist 0.0799
S3_c12_dist 0.0749
S0_c0_dist 0.0740
S5_c10_dist 0.0702
S3_c10_dist 0.0678
S2_c4_dist 0.0572
S4_c18_dist 0.0505
S2_c18_dist 0.0486
S1_c0_dist 0.0477
S3_c2_dist 0.0459
S3_c1_dist 0.0459
S5_c17_dist 0.0455
S4_c17_dist 0.0424
S3_c5_dist 0.0394
S5_c3_dist 0.0255
S3_c14_dist 0.0074
S2_c7_dist 0.0034
S5_c15_dist 0.0025
S3_c3_dist 0.0013
S3_c0_dist 0.0011
S6_c10_dist 0.0005
0.0
0.5
1.0
1.5
2.0
2.5
3.0

In [ ]: