

```
In [1]: #install necessary libraries
```

```
In [2]: pip install tensorflow
```

```
Requirement already satisfied: tensorflow in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (2.17.0)
Requirement already satisfied: tensorflow-intel==2.17.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow) (2.17.0)
Requirement already satisfied: absl-py>=1.0.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (2.1.0)
Requirement already satisfied: astunparse>=1.6.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (1.6.3)
Requirement already satisfied: flatbuffers>=24.3.25 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (24.3.25)
Requirement already satisfied: gast!=0.5.0,!0.5.1,!0.5.2,>=0.2.1 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (0.6.0)
Requirement already satisfied: google-pasta>=0.1.1 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (0.2.0)
Requirement already satisfied: h5py>=3.10.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (3.11.0)
Requirement already satisfied: libclang>=13.0.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (18.1.1)
Requirement already satisfied: ml-dtypes<0.5.0,>=0.3.1 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (0.4.1)
Requirement already satisfied: opt-einsum>=2.3.2 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (3.4.0)
Requirement already satisfied: packaging in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (23.2)
Requirement already satisfied: protobuf!=4.21.0,!=4.21.1,!=4.21.2,!=4.21.3,!=4.21.4,!=4.21.5,<5.0.0dev,>=3.20.3 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (3.20.3)
Requirement already satisfied: requests<3,>=2.21.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (2.32.2)
Requirement already satisfied: setuptools in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (75.3.0)
Requirement already satisfied: six>=1.12.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (1.16.0)
Requirement already satisfied: termcolor>=1.1.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (2.5.0)
Requirement already satisfied: typing-extensions>=3.6.6 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (4.11.0)
Requirement already satisfied: wrapt>=1.11.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (1.14.1)
Requirement already satisfied: grpcio<2.0,>=1.24.3 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (1.66.2)
Requirement already satisfied: tensorflowboard<2.18,>=2.17 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (2.17.1)
Requirement already satisfied: keras>=3.2.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (3.6.0)
Requirement already satisfied: numpy<2.0.0,>=1.26.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflow-intel==2.17.0->tensorflow) (1.26.4)
Requirement already satisfied: wheel<1.0,>=0.23.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from astunparse>=1.6.0->tensorflow-intel==2.17.0->tensorflow) (0.44.0)
Requirement already satisfied: rich in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from keras>=3.2.0->tensorflow-intel==2.17.0->tensorflow) (13.3.5)
Requirement already satisfied: namex in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from keras>=3.2.0->tensorflow-intel==2.17.0->tensorflow) (0.0.8)
Requirement already satisfied: optree in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from keras>=3.2.0->tensorflow-intel==2.17.0->tensorflow) (0.13.0)
Requirement already satisfied: charset-normalizer<4,>=2 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from requests<3,>=2.21.0->tensorflow-intel==2.17.0->tensorflow) (2.0.4)
Requirement already satisfied: idna<4,>=2.5 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from requests<3,>=2.21.0->tensorflow-intel==2.17.0->tensorflow) (3.7)
Requirement already satisfied: urllib3<3,>=1.21.1 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from requests<3,>=2.21.0->tensorflow-intel==2.17.0->tensorflow) (2.2.2)
Requirement already satisfied: certifi>=2017.4.17 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from requests<3,>=2.21.0->tensorflow-intel==2.17.0->tensorflow) (2024.8.30)
Requirement already satisfied: markdown>=2.6.8 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflowboard<2.18,>=2.17->tensorflow-intel==2.17.0->tensorflow) (3.4.1)
Requirement already satisfied: tensorflowboard-data-server<0.8.0,>=0.7.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflowboard<2.18,>=2.17->tensorflow-intel==2.17.0->tensorflow) (0.7.2)
Requirement already satisfied: werkzeug>=1.0.1 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from tensorflowboard<2.18,>=2.17->tensorflow-intel==2.17.0->tensorflow) (3.0.3)
Requirement already satisfied: MarkupSafe>=2.1.1 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from werkzeug>=1.0.1->tensorflowboard<2.18,>=2.17->tensorflow-intel==2.17.0->tensorflow) (2.1.3)
Requirement already satisfied: markdown-it-py<3.0.0,>=2.2.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from rich->keras>=3.2.0->tensorflow-intel==2.17.0->tensorflow) (2.2.0)
Requirement already satisfied: pygments<3.0.0,>=2.13.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from rich->keras>=3.2.0->tensorflow-intel==2.17.0->tensorflow) (2.15.1)
Requirement already satisfied: mdurl~0.1 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from markdown-it-py<3.0.0,>=2.2.0->rich->keras>=3.2.0->tensorflow-intel==2.17.0->tensorflow) (0.1.0)
Note: you may need to restart the kernel to use updated packages.
```

```
In [3]: pip install tabulate
```

```
Requirement already satisfied: tabulate in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (0.9.0)
Note: you may need to restart the kernel to use updated packages.
```

```
In [4]: pip install scikit-learn
```

```
Requirement already satisfied: scikit-learn in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (1.5.2)
Requirement already satisfied: numpy>=1.19.5 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from scikit-learn) (1.26.4)
Requirement already satisfied: scipy>=1.6.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from scikit-learn) (1.13.1)
Requirement already satisfied: joblib>=1.2.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from scikit-learn) (1.4.2)
Requirement already satisfied: threadpoolctl>=3.1.0 in c:\users\brady\onedrive\apps\anaconda\lib\site-packages (from scikit-learn) (3.5.0)
Note: you may need to restart the kernel to use updated packages.
```

```
In [5]: #import necessary packages
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow
from sklearn.model_selection import train_test_split
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, Embedding, Flatten, Dense, Concatenate
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import StandardScaler
from sklearn.cluster import KMeans
from scipy.spatial.distance import cdist
from tabulate import tabulate
from sklearn.metrics import silhouette_score, calinski_harabasz_score, davies_bouldin_score
from sklearn.neighbors import LocalOutlierFactor
from sklearn.cluster import DBSCAN
from sklearn.decomposition import PCA
from imblearn.over_sampling import SMOTE
import pickle
import platform
import sys
import subprocess
import pkg_resources
```

```
In [6]: #PULL THE DATA
```

```
#create pandas DataFrame for financial anomaly data
financial_df = pd.read_csv("~/Analytics-Practicum/data/financial_anomaly_data2.csv")
```

```
In [7]: #CONDUCT DATA PREPROCESSING AND EXPLORATORY DATA ANALYSIS
```

```
#print first 5 columns of DataFrame
financial_df.head(5)
```

```
Out[7]:
```

	Timestamp	TransactionID	AccountID	Amount	Merchant	TransactionType	Location
0	1/1/2023 8:00	TXN1127	ACC4	95071.92	MerchantH	Purchase	Tokyo
1	1/1/2023 8:01	TXN1639	ACC10	15607.89	MerchantH	Purchase	London
2	1/1/2023 8:02	TXN872	ACC8	65092.34	MerchantE	Withdrawal	London
3	1/1/2023 8:03	TXN1438	ACC6	87.87	MerchantE	Purchase	London
4	1/1/2023 8:04	TXN1338	ACC6	716.56	MerchantI	Purchase	Los Angeles

```
In [8]: #print class, RangeIndex, columns, non-null count, data type, and memory usage information
financial_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 216960 entries, 0 to 216959
Data columns (total 7 columns):
 #   Column           Non-Null Count  Dtype  
 --- 
 0   Timestamp        216960 non-null   object  
 1   TransactionID   216960 non-null   object  
 2   AccountID       216960 non-null   object  
 3   Amount           216960 non-null   float64 
 4   Merchant         216960 non-null   object  
 5   TransactionType 216960 non-null   object  
 6   Location          216960 non-null   object  
dtypes: float64(1), object(6)
memory usage: 11.6+ MB
```

```
In [9]: #print shape of DataFrame
financial_df.shape
```

```
Out[9]: (216960, 7)
```

```
In [10]: #print sum of null occurrences of each variable in DataFrame  
print(financial_df.isnull().sum())
```

```
Timestamp      0  
TransactionID 0  
AccountID     0  
Amount         0  
Merchant       0  
TransactionType 0  
Location       0  
dtype: int64
```

```
In [11]: #create a new DataFrame excluding null occurrences  
new_financial_df = financial_df.dropna()
```

```
In [12]: #print shape of new DataFrame  
new_financial_df.shape
```

```
Out[12]: (216960, 7)
```

```
In [13]: #verify that null occurrences were handled properly  
print(new_financial_df.isnull().sum())
```

```
Timestamp      0  
TransactionID 0  
AccountID     0  
Amount         0  
Merchant       0  
TransactionType 0  
Location       0  
dtype: int64
```

```
In [14]: #print number of unique occurrences of each variable in DataFrame  
print(f"Number of unique Timestamp: {new_financial_df['Timestamp'].nunique()}")  
print(f"Number of unique TransactionID: {new_financial_df['TransactionID'].nunique()}")  
print(f"Number of unique AccountID: {new_financial_df['AccountID'].nunique()}")  
print(f"Number of unique Amount: {new_financial_df['Amount'].nunique()}")  
print(f"Number of unique Merchant: {new_financial_df['Merchant'].nunique()}")  
print(f"Number of unique TransactionType: {new_financial_df['TransactionType'].nunique()}")  
print(f"Number of unique Location: {new_financial_df['Location'].nunique()}")
```

```
Number of unique Timestamp: 216960  
Number of unique TransactionID: 1999  
Number of unique AccountID: 15  
Number of unique Amount: 214687  
Number of unique Merchant: 10  
Number of unique TransactionType: 3  
Number of unique Location: 5
```

```
In [15]: #introduce new variables to DataFrame for analysis of certain variables' interactions
```

```
new_financial_df['AccountID/Merchant'] = new_financial_df['AccountID'].astype(str) + '_' + new_financial_df['Merchant'].astype(str)  
new_financial_df['AccountID/TransactionID'] = new_financial_df['AccountID'].astype(str) + '_' + new_financial_df['TransactionID'].astype(str)  
new_financial_df['AccountID/Merchant/TransactionID'] = new_financial_df['AccountID'].astype(str) + '_' + new_financial_df['Merchant'].astype(str) + '_' + new_financial_df['TransactionID'].astype(str)  
new_financial_df['TransactionType/Merchant'] = new_financial_df['TransactionType'].astype(str) + '_' + new_financial_df['Merchant'].astype(str)  
new_financial_df['Location/TransactionType'] = new_financial_df['Location'].astype(str) + '_' + new_financial_df['TransactionType'].astype(str)  
new_financial_df['Merchant/Location'] = new_financial_df['Merchant'].astype(str) + '_' + new_financial_df['Location'].astype(str)
```

```
In [16]: #verify that new variables have been created successfully  
new_financial_df.head(5)
```

```
Out[16]:
```

	Timestamp	TransactionID	AccountID	Amount	Merchant	TransactionType	Location	AccountID/Merchant	AccountID/Transact
0	1/1/2023 8:00	TXN1127	ACC4	95071.92	MerchantH	Purchase	Tokyo	ACC4_MerchantH	ACC4_TXN
1	1/1/2023 8:01	TXN1639	ACC10	15607.89	MerchantH	Purchase	London	ACC10_MerchantH	ACC10_TXN
2	1/1/2023 8:02	TXN872	ACC8	65092.34	MerchantE	Withdrawal	London	ACC8_MerchantE	ACC8_TXN
3	1/1/2023 8:03	TXN1438	ACC6	87.87	MerchantE	Purchase	London	ACC6_MerchantE	ACC6_TXN
4	1/1/2023 8:04	TXN1338	ACC6	716.56	MerchantI	Purchase	Los Angeles	ACC6_MerchantI	ACC6_TXN

```
In [17]: #convert Timestamp variable to a DateTime object
```

```
new_financial_df['Timestamp'] = pd.to_datetime(new_financial_df['Timestamp'], format='%d/%m/%Y %H:%M')
```

```
In [18]: #create distinct features for minute/hour of the day, day of the week, and month
```

```

new_financial_df['Minute'] = new_financial_df['Timestamp'].dt.minute
new_financial_df['Hour'] = new_financial_df['Timestamp'].dt.hour
new_financial_df['Day'] = new_financial_df['Timestamp'].dt.dayofweek
new_financial_df['Month'] = new_financial_df['Timestamp'].dt.month

```

In [19]: #verify again that new variables have been created successfully
new_financial_df.head(5)

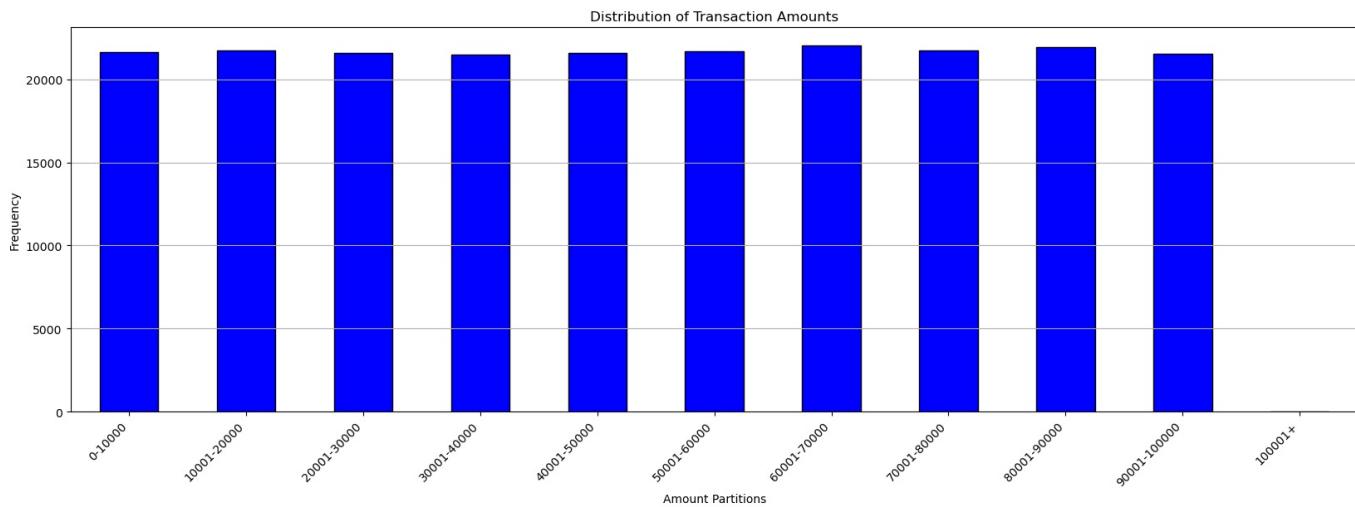
Out[19]:

	Timestamp	TransactionID	AccountID	Amount	Merchant	TransactionType	Location	AccountID/Merchant	AccountID/Transact
0	2023-01-01 08:00:00	TXN1127	ACC4	95071.92	MerchantH	Purchase	Tokyo	ACC4_MerchantH	ACC4_TXN
1	2023-01-01 08:01:00	TXN1639	ACC10	15607.89	MerchantH	Purchase	London	ACC10_MerchantH	ACC10_TXN
2	2023-01-01 08:02:00	TXN872	ACC8	65092.34	MerchantE	Withdrawal	London	ACC8_MerchantE	ACC8_TXN
3	2023-01-01 08:03:00	TXN1438	ACC6	87.87	MerchantE	Purchase	London	ACC6_MerchantE	ACC6_TXN
4	2023-01-01 08:04:00	TXN1338	ACC6	716.56	MerchantI	Purchase	Los Angeles	ACC6_MerchantI	ACC6_TXN

In [20]: #divide amount variable into appropriately-sized partitions
bins = [0, 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000, 100000, float('inf')]
labels = ['0-10000', '10001-20000', '20001-30000', '30001-40000', '40001-50000', '50001-60000', '60001-70000',
new_financial_df['Amount_Partitions'] = pd.cut(new_financial_df['Amount'], bins=bins, labels=labels)

In [21]: #construct bar graph for distribution of transaction in each amount partition
partition_counts = new_financial_df['Amount_Partitions'].value_counts().reindex(labels)

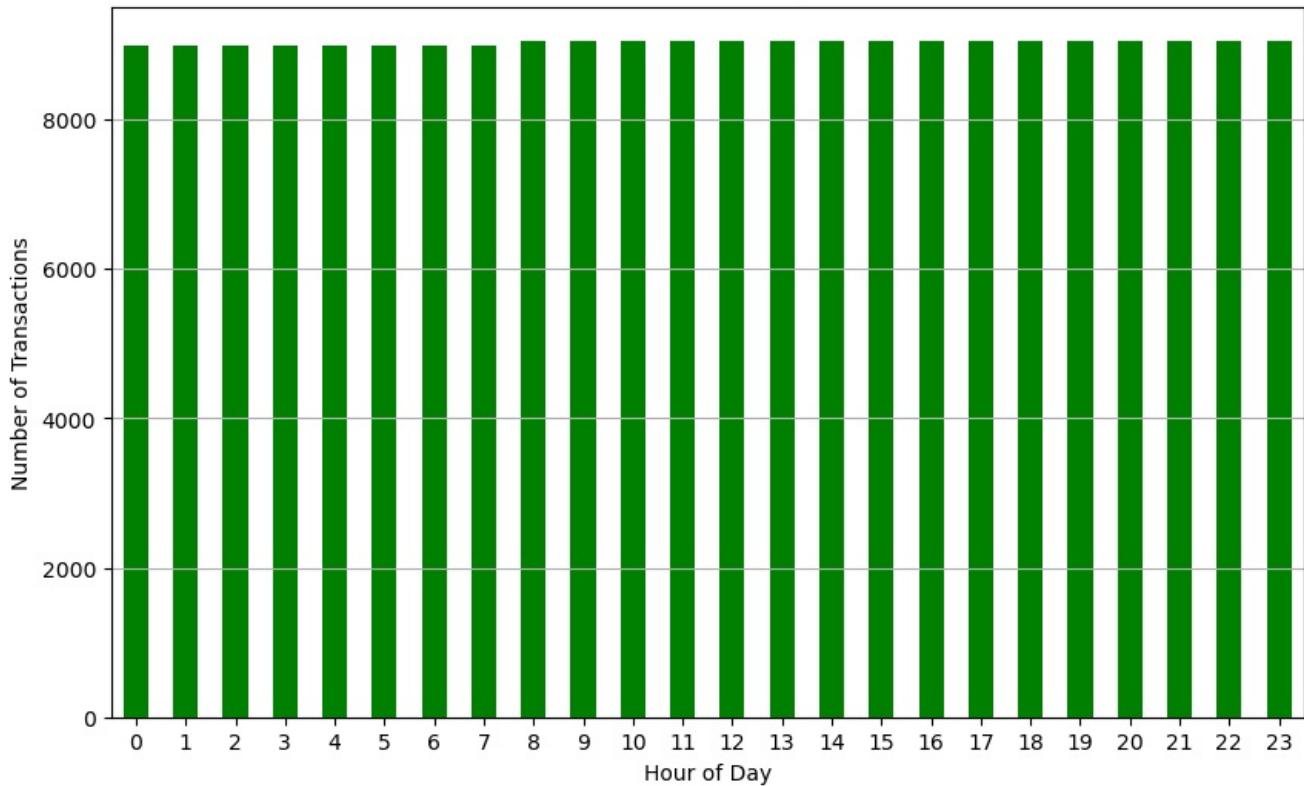
plt.figure(figsize=(20, 6))
partition_counts.plot(kind='bar', color='blue', edgecolor='black')
plt.title('Distribution of Transaction Amounts')
plt.xlabel('Amount Partitions')
plt.ylabel('Frequency')
plt.xticks(rotation=45, ha='right')
plt.grid(axis='y')
plt.show()



In [22]: #construct bar graph for total number of transactions per hour
hour_counts = new_financial_df['Hour'].value_counts().sort_index()

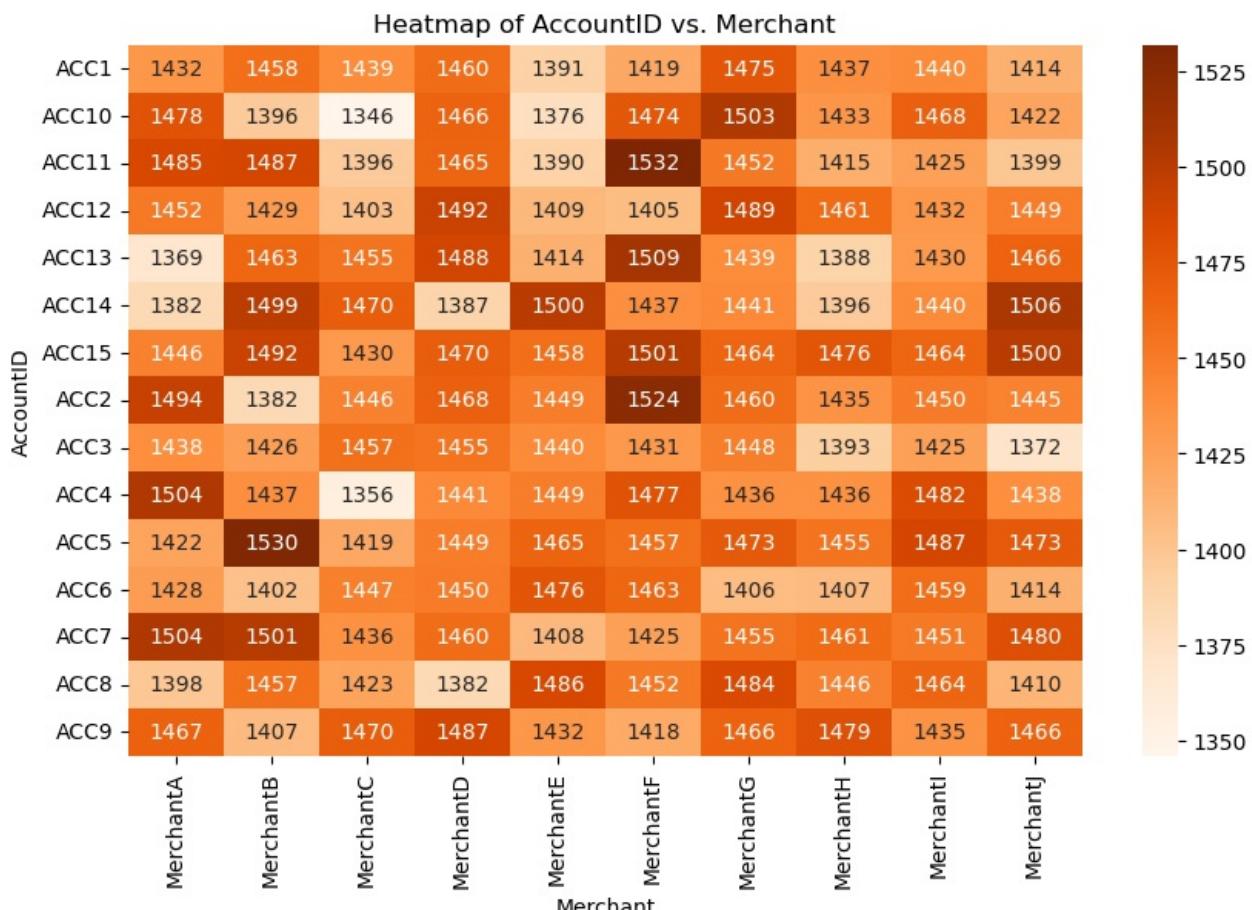
plt.figure(figsize=(10, 6))
hour_counts.plot(kind='bar', color='green')
plt.title('Transaction Counts by Hour')
plt.xlabel('Hour of Day')
plt.ylabel('Number of Transactions')
plt.xticks(rotation=0)
plt.grid(axis='y')
plt.show()

Transaction Counts by Hour



```
In [23]: #construct heat map to visualize total amounts of each combination of AccountID and Merchant (150 combinations)
pivot_table = pd.crosstab(new_financial_df['AccountID'], new_financial_df['Merchant'])

plt.figure(figsize=(10, 6))
sns.heatmap(pivot_table, annot=True, cmap='Oranges', fmt='d')
plt.title('Heatmap of AccountID vs. Merchant')
plt.xlabel('Merchant')
plt.ylabel('AccountID')
plt.show()
```



```
In [24]: #print a sample of the first 10 values of the cleaned dataset with new variables added
new_financial_df.head(10)
```

Out [24]:		Timestamp	TransactionID	AccountID	Amount	Merchant	TransactionType	Location	AccountID/Merchant	AccountID/Transact
0	2023-01-01 08:00:00	TXN1127	ACC4	95071.92	MerchantH		Purchase	Tokyo	ACC4_MerchantH	ACC4_TXN
1	2023-01-01 08:01:00	TXN1639	ACC10	15607.89	MerchantH		Purchase	London	ACC10_MerchantH	ACC10_TXN
2	2023-01-01 08:02:00	TXN872	ACC8	65092.34	MerchantE		Withdrawal	London	ACC8_MerchantE	ACC8_TXN
3	2023-01-01 08:03:00	TXN1438	ACC6	87.87	MerchantE		Purchase	London	ACC6_MerchantE	ACC6_TXN
4	2023-01-01 08:04:00	TXN1338	ACC6	716.56	MerchantI		Purchase	Los Angeles	ACC6_MerchantI	ACC6_TXN
5	2023-01-01 08:05:00	TXN1083	ACC15	13957.99	MerchantC		Transfer	London	ACC15_MerchantC	ACC15_TXN
6	2023-01-01 08:06:00	TXN832	ACC9	4654.58	MerchantC		Transfer	Tokyo	ACC9_MerchantC	ACC9_TXN
7	2023-01-01 08:07:00	TXN841	ACC7	1336.36	MerchantI		Withdrawal	San Francisco	ACC7_MerchantI	ACC7_TXN
8	2023-01-01 08:08:00	TXN777	ACC10	9776.23	MerchantD		Transfer	London	ACC10_MerchantD	ACC10_TXN
9	2023-01-01 08:09:00	TXN1479	ACC12	49522.74	MerchantC		Withdrawal	New York	ACC12_MerchantC	ACC12_TXN

```
In [25]: #print class, RangeIndex, columns, non-null count, data type, and memory usage information for the updated Dataframe
new_financial_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 216960 entries, 0 to 216959
Data columns (total 18 columns):
 #   Column           Non-Null Count  Dtype  
 ---  -- 
 0   Timestamp        216960 non-null   datetime64[ns]
 1   TransactionID   216960 non-null   object 
 2   AccountID        216960 non-null   object 
 3   Amount            216960 non-null   float64
 4   Merchant          216960 non-null   object 
 5   TransactionType  216960 non-null   object 
 6   Location          216960 non-null   object 
 7   AccountID/Merchant  216960 non-null   object 
 8   AccountID/TransactionID  216960 non-null   object 
 9   AccountID/Merchant/TransactionID  216960 non-null   object 
 10  TransactionType/Merchant  216960 non-null   object 
 11  Location/TransactionType  216960 non-null   object 
 12  Merchant/Location    216960 non-null   object 
 13  Minute             216960 non-null   int32  
 14  Hour               216960 non-null   int32  
 15  Day                216960 non-null   int32  
 16  Month              216960 non-null   int32  
 17  Amount_Partitions  216960 non-null   category
dtypes: category(1), datetime64[ns](1), float64(1), int32(4), object(11)
memory usage: 25.0+ MB
```

```
In [26]: #print number of unique occurrences of newly created variables
print(f"Number of unique AccountID/Merchant: {new_financial_df['AccountID/Merchant'].nunique()}")
print(f"Number of unique AccountID/TransactionID: {new_financial_df['AccountID/TransactionID'].nunique()}")
print(f"Number of unique AccountID/Merchant/TransactionID: {new_financial_df['AccountID/Merchant/TransactionID'].nunique()}")
print(f"Number of unique TransactionType/Merchant: {new_financial_df['TransactionType/Merchant'].nunique()}")
print(f"Number of unique Location/TransactionType: {new_financial_df['Location/TransactionType'].nunique()}")
print(f"Number of unique Merchant/Location: {new_financial_df['Merchant/Location'].nunique()}")
print(f"Number of unique Minute: {new_financial_df['Minute'].nunique()}")
print(f"Number of unique Hour: {new_financial_df['Hour'].nunique()}")
print(f"Number of unique Day: {new_financial_df['Day'].nunique()}")
print(f"Number of unique Month: {new_financial_df['Month'].nunique()}")
print(f"Number of unique Amount_Partitions: {new_financial_df['Amount_Partitions'].nunique()}")
```

```
Number of unique AccountID/Merchant: 150
Number of unique AccountID/TransactionID: 29967
Number of unique AccountID/Merchant/TransactionID: 154226
Number of unique TransactionType/Merchant: 30
Number of unique Location/TransactionType: 15
Number of unique Merchant/Location: 50
Number of unique Minute: 60
Number of unique Hour: 24
Number of unique Day: 7
Number of unique Month: 5
Number of unique Amount_Partitions: 11
```

```
In [27]: #describe numerical data to better understand these columns
```

```
new_financial_df.describe()
```

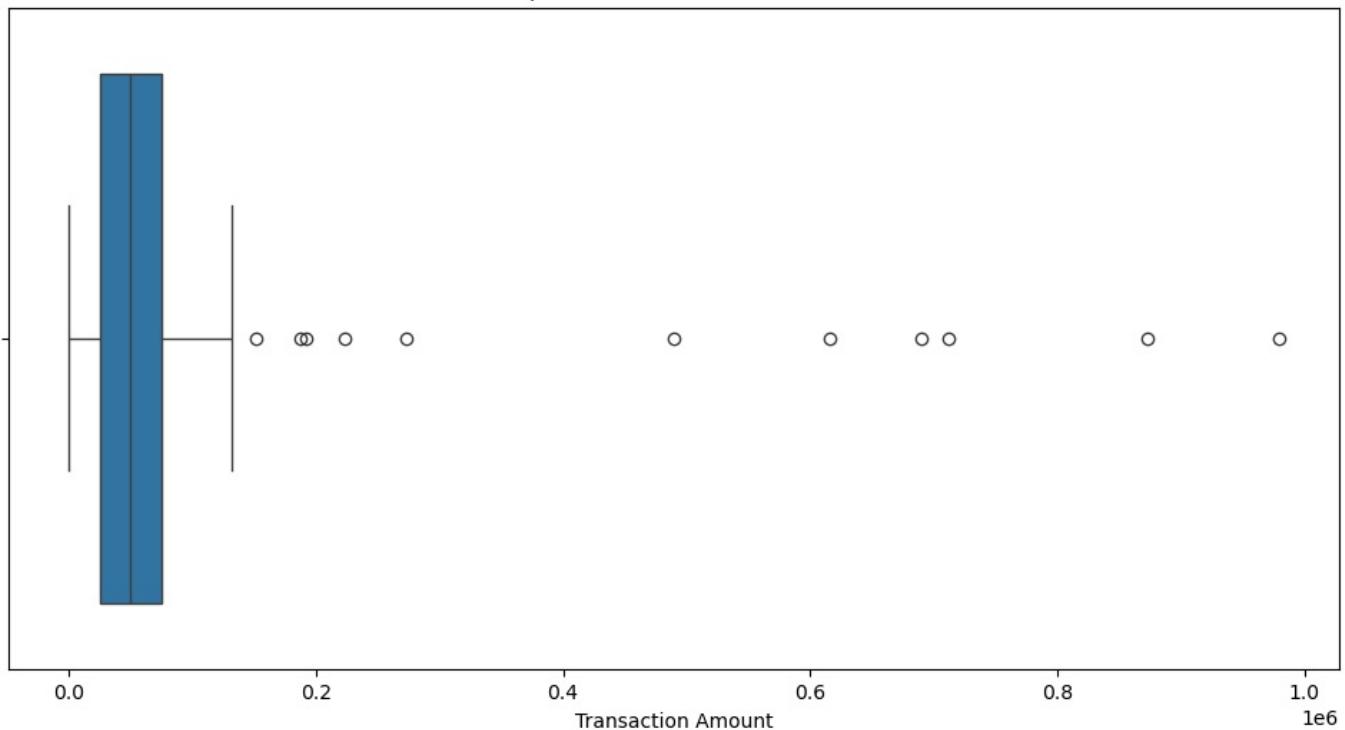
Out[27]:

	Timestamp	Amount	Minute	Hour	Day	Month
count	216960	216960.000000	216960.000000	216960.000000	216960.000000	216960.000000
mean	2023-03-17 15:59:30	50090.025108	29.500000	11.517699	2.973451	3.017699
min	2023-01-01 08:00:00	10.510000	0.000000	0.000000	0.000000	1.000000
25%	2023-02-07 23:59:45	25061.242500	14.750000	6.000000	1.000000	2.000000
50%	2023-03-17 15:59:30	50183.980000	29.500000	12.000000	3.000000	3.000000
75%	2023-04-24 07:59:15	75080.460000	44.250000	18.000000	5.000000	4.000000
max	2023-05-31 23:59:00	978942.260000	59.000000	23.000000	6.000000	5.000000
std		NaN	29097.905016	17.318142	6.918770	2.008659

In [28]:

```
#construct a plot of the spread of transaction amounts
plt.figure(figsize=(12, 6))
sns.boxplot(x='Amount', data=new_financial_df)
plt.title('Boxplot of Transaction Amounts')
plt.xlabel('Transaction Amount')
plt.show()
```

Boxplot of Transaction Amounts



In [29]:

```
#print counts of each unique value in each column of the DataFrame
for column in new_financial_df.columns:
    column_count = new_financial_df[column].value_counts()
    print(column_count)
```

```
Timestamp
2023-01-01 08:00:00    1
2023-04-11 18:57:00    1
2023-04-11 18:33:00    1
2023-04-11 18:34:00    1
2023-04-11 18:35:00    1
...
2023-02-20 13:23:00    1
2023-02-20 13:24:00    1
2023-02-20 13:25:00    1
2023-02-20 13:26:00    1
2023-05-31 23:59:00    1
Name: count, Length: 216960, dtype: int64
```

```
TransactionID
TXN838      139
TXN1768     139
TXN1658     139
TXN1389     138
TXN340      137
...
TXN60       79
TXN891      78
TXN605      78
```

```
TXN201      73
TXN799      70
Name: count, Length: 1999, dtype: int64
AccountID
ACC15      14701
ACC5       14630
ACC7       14581
ACC2       14553
ACC9       14527
ACC14      14458
ACC4       14456
ACC11      14446
ACC12      14421
ACC13      14421
ACC8       14402
ACC1       14365
ACC10      14362
ACC6       14352
ACC3       14285
Name: count, dtype: int64
Amount
18010.00    3
34588.69    3
74109.74    3
86099.64    3
7309.50     3
...
56652.57    1
36336.36    1
49174.76    1
71557.91    1
65004.99    1
Name: count, Length: 214687, dtype: int64
Merchant
MerchantF   21924
MerchantG   21891
MerchantD   21820
MerchantB   21766
MerchantI   21752
MerchantA   21699
MerchantJ   21654
MerchantE   21543
MerchantH   21518
MerchantC   21393
Name: count, dtype: int64
TransactionType
Transfer     72793
Purchase     72235
Withdrawal   71932
Name: count, dtype: int64
Location
San Francisco 43613
New York      43378
London        43343
Los Angeles   43335
Tokyo         43291
Name: count, dtype: int64
AccountID/Merchant
ACC11_MerchantF 1532
ACC5_MerchantB  1530
ACC2_MerchantF  1524
ACC13_MerchantF 1509
ACC14_MerchantJ 1506
...
ACC10_MerchantE 1376
ACC3_MerchantJ  1372
ACC13_MerchantA 1369
ACC4_MerchantC  1356
ACC10_MerchantC 1346
Name: count, Length: 150, dtype: int64
AccountID/TransactionID
ACC8_TXN239   22
ACC6_TXN154   20
ACC11_TXN1614 19
ACC11_TXN410   19
ACC1_TXN220   19
...
ACC14_TXN20   1
ACC5_TXN938   1
ACC12_TXN1314 1
ACC3_TXN127   1
ACC2_TXN737   1
Name: count, Length: 29967, dtype: int64
```

```
AccountID/Merchant/TransactionID
ACC3_MerchantF_TXN1801      7
ACC11_MerchantJ_TXN1488     6
ACC11_MerchantE_TXN153      6
ACC14_MerchantJ_TXN1389     6
ACC15_MerchantG_TXN220      6
                            ..
ACC10_MerchantH_TXN286      1
ACC7_MerchantF_TXN1587      1
ACC5_MerchantA_TXN1930      1
ACC6_MerchantF_TXN1695      1
ACC3_MerchantG_TXN1807      1
Name: count, Length: 154226, dtype: int64
TransactionType/Merchant
Purchase_MerchantF          7399
Transfer_MerchantG          7354
Transfer_MerchantH          7342
Transfer_MerchantA          7332
Withdrawal_MerchantD        7323
Withdrawal_MerchantI        7308
Transfer_MerchantF          7302
Purchase_MerchantG          7298
Transfer_MerchantB          7291
Transfer_MerchantJ          7286
Purchase_MerchantB          7274
Purchase_MerchantA          7269
Purchase_MerchantD          7250
Transfer_MerchantD          7247
Withdrawal_MerchantG        7239
Transfer_MerchantI          7238
Withdrawal_MerchantF        7223
Purchase_MerchantE          7216
Purchase_MerchantJ          7216
Transfer_MerchantE          7209
Purchase_MerchantI          7206
Withdrawal_MerchantB        7201
Transfer_MerchantC          7192
Withdrawal_MerchantC        7164
Withdrawal_MerchantJ        7152
Withdrawal_MerchantE        7118
Withdrawal_MerchantH        7106
Withdrawal_MerchantA        7098
Purchase_MerchantH          7070
Purchase_MerchantC          7037
Name: count, dtype: int64
Location/TransactionType
London_Transfer              14653
San_Francisco_Transfer       14610
Los_Angeles_Transfer         14580
San_Francisco_Withdrawal     14515
New_York_Transfer            14510
Tokyo_Purchase               14506
San_Francisco_Purchase       14488
New_York_Purchase             14445
Tokyo_Transfer                14440
New_York_Withdrawal           14423
Los_Angeles_Purchase          14411
London_Purchase               14385
Tokyo_Withdrawal              14345
Los_Angeles_Withdrawal        14344
London_Withdrawal              14305
Name: count, dtype: int64
Merchant/Location
MerchantF_Los_Angeles        4476
MerchantD_London              4453
MerchantG_London              4446
MerchantI_Tokyo                4445
MerchantG_New_York              4432
MerchantE_San_Francisco        4424
MerchantB_Los_Angeles          4399
MerchantE_New_York              4395
MerchantA_Los_Angeles          4394
MerchantH_New_York              4393
MerchantA_Tokyo                4393
MerchantB_London                4391
MerchantI_San_Francisco        4390
MerchantB_San_Francisco        4385
MerchantF_Tokyo                4376
MerchantG_Tokyo                4373
MerchantA_San_Francisco        4368
MerchantF_San_Francisco        4367
MerchantD_San_Francisco        4360
MerchantD_Los_Angeles          4360
```

```
MerchantF_New York      4356
MerchantJ_Tokyo         4353
MerchantJ_San Francisco 4350
MerchantF_London        4349
MerchantG_San Francisco 4348
MerchantD_Tokyo         4347
MerchantJ_New York       4346
MerchantH_San Francisco 4334
MerchantE_London         4332
MerchantJ_Los Angeles    4332
MerchantB_New York        4332
MerchantA_London          4332
MerchantI_Los Angeles    4330
MerchantH_Tokyo          4311
MerchantC_New York        4310
MerchantI_New York         4302
MerchantD_New York        4300
MerchantC_Tokyo          4296
MerchantG_Los Angeles    4292
MerchantC_San Francisco   4287
MerchantI_London          4285
MerchantC_Los Angeles    4285
MerchantJ_London          4273
MerchantH_London          4267
MerchantB_Tokyo          4259
MerchantE_Los Angeles    4254
MerchantC_London          4215
MerchantH_Los Angeles     4213
MerchantA_New York         4212
MerchantE_Tokyo           4138
Name: count, dtype: int64
```

Minute

```
0      3616
1      3616
32     3616
33     3616
34     3616
35     3616
36     3616
37     3616
38     3616
39     3616
40     3616
41     3616
42     3616
43     3616
44     3616
45     3616
46     3616
47     3616
48     3616
49     3616
50     3616
51     3616
52     3616
53     3616
54     3616
55     3616
56     3616
57     3616
58     3616
31     3616
30     3616
29     3616
14     3616
2      3616
3      3616
4      3616
5      3616
6      3616
7      3616
8      3616
9      3616
10     3616
11     3616
12     3616
13     3616
15     3616
28     3616
16     3616
17     3616
18     3616
19     3616
```

```
20    3616
21    3616
22    3616
23    3616
24    3616
25    3616
26    3616
27    3616
59    3616
Name: count, dtype: int64
Hour
8     9060
17    9060
23    9060
22    9060
21    9060
9     9060
19    9060
18    9060
20    9060
16    9060
15    9060
14    9060
13    9060
12    9060
11    9060
10    9060
0     9000
1     9000
2     9000
3     9000
4     9000
5     9000
6     9000
7     9000
Name: count, dtype: int64
Day
0     31680
1     31680
2     31680
6     31200
3     30240
4     30240
5     30240
Name: count, dtype: int64
Month
3     44640
5     44640
1     44160
4     43200
2     40320
Name: count, dtype: int64
Amount_Partitions
60001-70000    22015
80001-90000    21938
10001-20000    21743
70001-80000    21736
50001-60000    21661
0-10000       21651
40001-50000    21605
20001-30000    21601
90001-100000   21530
30001-40000    21466
100001+         14
Name: count, dtype: int64
```

```
In [30]: #list variables to be one-hot encoded
```

```
one_hot_encoding = [
    'AccountID/Merchant',
    'TransactionType',
    'Location',
    'Amount_Partitions'
]

#apply one-hot encoding
new_financial_df_encoded = pd.get_dummies(new_financial_df, columns=one_hot_encoding)

#display the first few rows of the encoded DataFrame
print(new_financial_df_encoded.head())
```

```

Timestamp TransactionID AccountID      Amount    Merchant \
0 2023-01-01 08:00:00      TXN1127      ACC4  95071.92 MerchantH
1 2023-01-01 08:01:00      TXN1639      ACC10 15607.89 MerchantH
2 2023-01-01 08:02:00      TXN872       ACC8  65092.34 MerchantE
3 2023-01-01 08:03:00      TXN1438      ACC6   87.87 MerchantE
4 2023-01-01 08:04:00      TXN1338      ACC6  716.56 MerchantI

   AccountID/TransactionID AccountID/Merchant/TransactionID \
0           ACC4_TXN1127          ACC4_MerchantH_TXN1127
1           ACC10_TXN1639         ACC10_MerchantH_TXN1639
2           ACC8_TXN872          ACC8_MerchantE_TXN872
3           ACC6_TXN1438          ACC6_MerchantE_TXN1438
4           ACC6_TXN1338          ACC6_MerchantI_TXN1338

   TransactionType/Merchant Location/TransactionType      Merchant/Location \
0     Purchase_MerchantH        Tokyo_Purchase    MerchantH_Tokyo
1     Purchase_MerchantH        London_Purchase  MerchantH_London
2   Withdrawal_MerchantE      London_Withdrawal MerchantE_London
3     Purchase_MerchantE        London_Purchase  MerchantE_London
4     Purchase_MerchantI      Los Angeles_Purchase MerchantI_Los Angeles

...  Amount_Partitions_10001-20000  Amount_Partitions_20001-30000 \
0 ...                           False             False
1 ...                           True              False
2 ...                           False             False
3 ...                           False             False
4 ...                           False             False

  Amount_Partitions_30001-40000  Amount_Partitions_40001-50000 \
0 ...                           False             False
1 ...                           False             False
2 ...                           False             False
3 ...                           False             False
4 ...                           False             False

  Amount_Partitions_50001-60000  Amount_Partitions_60001-70000 \
0 ...                           False             False
1 ...                           False             False
2 ...                           False             True
3 ...                           False             False
4 ...                           False             False

  Amount_Partitions_70001-80000  Amount_Partitions_80001-90000 \
0 ...                           False             False
1 ...                           False             False
2 ...                           False             False
3 ...                           False             False
4 ...                           False             False

  Amount_Partitions_90001-100000 Amount_Partitions_100001+
0 ...                           True              False
1 ...                           False             False
2 ...                           False             False
3 ...                           False             False
4 ...                           False             False

```

[5 rows x 183 columns]

In [31]: `#print DataFrame info to maintain understanding of DataFrame properties
new_financial_df_encoded.info()`

```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 216960 entries, 0 to 216959
Columns: 183 entries, Timestamp to Amount_Partitions_100001+
dtypes: bool(169), datetime64[ns](1), float64(1), int32(4), object(8)
memory usage: 54.8+ MB

```

In [32]: `#CORRELATION ANALYSIS
#retrieve one-hot encoded columns
account_merchant_columns = [col for col in new_financial_df_encoded.columns if 'AccountID/Merchant' in col]
transaction_type_columns = [col for col in new_financial_df_encoded.columns if 'TransactionType' in col]
location_columns = [col for col in new_financial_df_encoded.columns if 'Location' in col]
amount_partitions_columns = [col for col in new_financial_df_encoded.columns if 'Amount_Partitions' in col]

#create a dictionary to store correlations
correlation_results = {}

#iterate through each pair of one-hot encoded columns to compute correlations
for account_merchant in account_merchant_columns:
 for transaction_type in transaction_type_columns:
 correlation1 = new_financial_df_encoded[account_merchant].corr(new_financial_df_encoded[transaction_type])
 correlation_results[(account_merchant, transaction_type)] = correlation1

for account_merchant in account_merchant_columns:`

```

for location in location_columns:
    correlation2 = new_financial_df_encoded[account_merchant].corr(new_financial_df_encoded[location])
    correlation_results[(account_merchant, location)] = correlation2

for account_merchant in account_merchant_columns:
    for amount_partitions in amount_partitions_columns:
        correlation3 = new_financial_df_encoded[account_merchant].corr(new_financial_df_encoded[amount_partitions])
        correlation_results[(account_merchant, amount_partitions)] = correlation3

for transaction_type in transaction_type_columns:
    for location in location_columns:
        correlation4 = new_financial_df_encoded[transaction_type].corr(new_financial_df_encoded[location])
        correlation_results[(transaction_type, location)] = correlation4

for transaction_type in transaction_type_columns:
    for amount_partitions in amount_partitions_columns:
        correlation5 = new_financial_df_encoded[transaction_type].corr(new_financial_df_encoded[amount_partitions])
        correlation_results[(transaction_type, amount_partitions)] = correlation5

for location in location_columns:
    for amount_partitions in amount_partitions_columns:
        correlation6 = new_financial_df_encoded[location].corr(new_financial_df_encoded[amount_partitions])
        correlation_results[(location, amount_partitions)] = correlation6

```

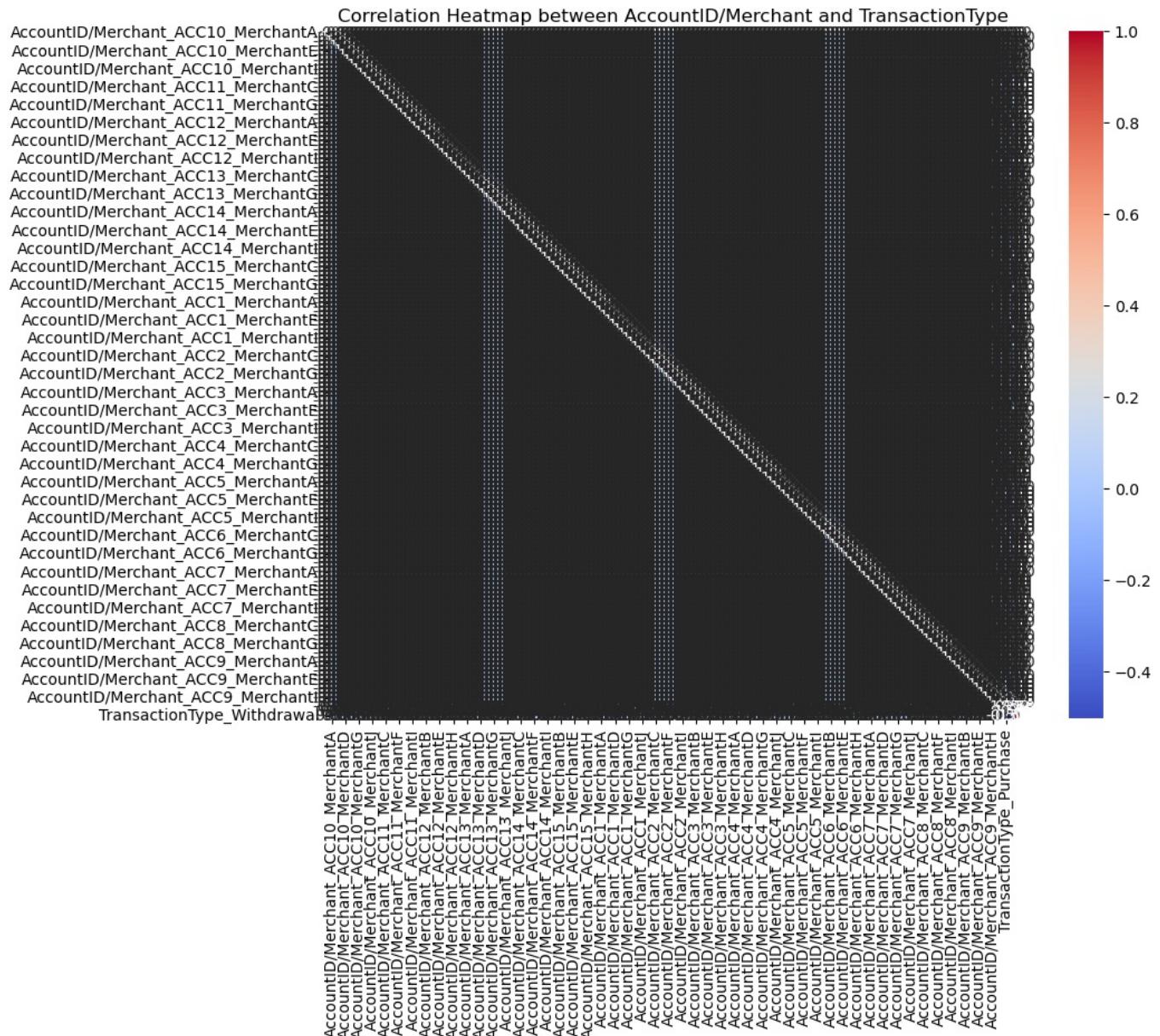
In [33]: #example of a correlation within the dataset (account_merchant and transaction_type)

```

correlation_matrix = new_financial_df_encoded[account_merchant_columns + transaction_type_columns].corr()

# Create a heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm', square=True)
plt.title('Correlation Heatmap between AccountID/Merchant and TransactionType')
plt.show()

```



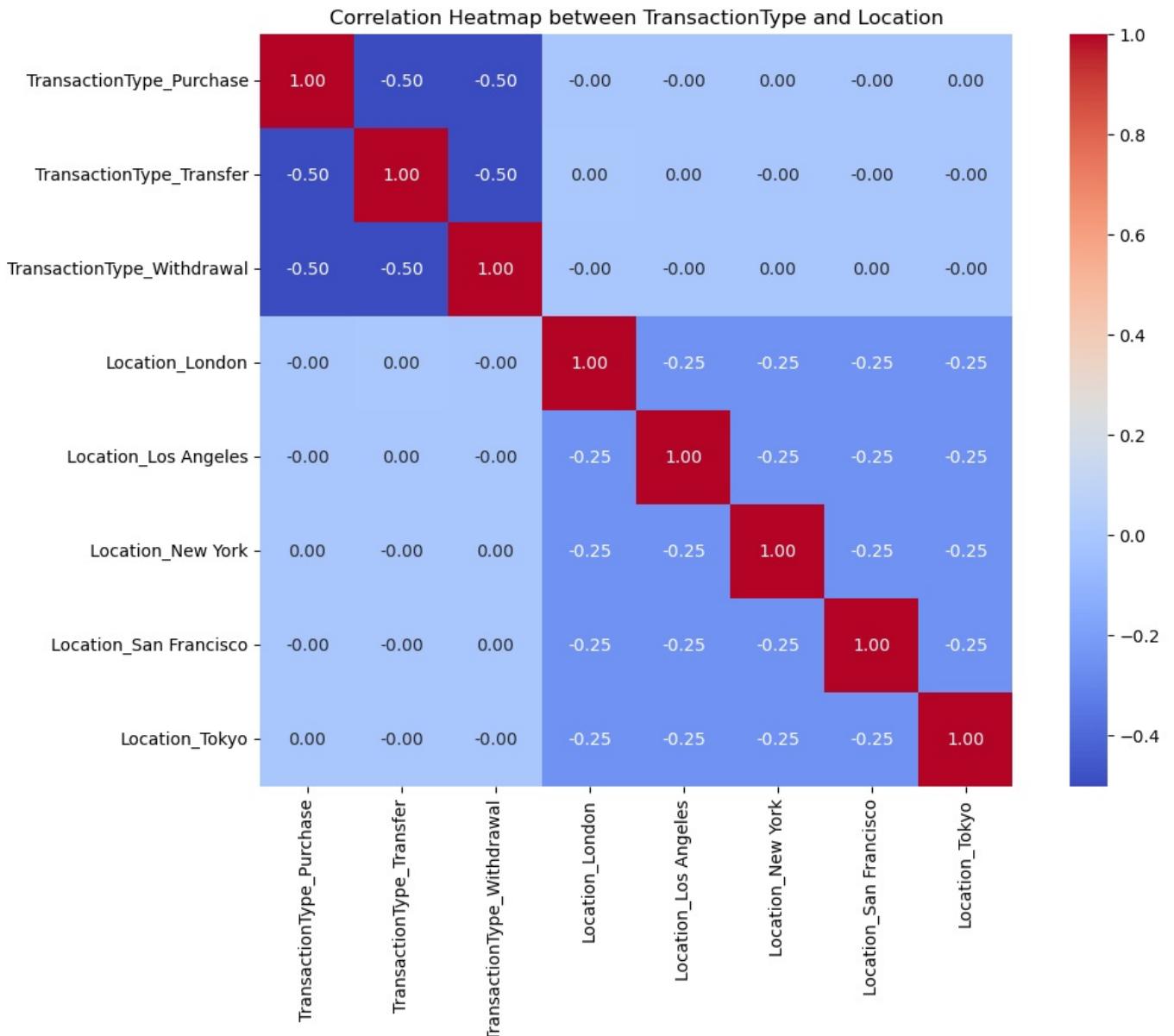
In [34]: #example of a correlation within the dataset (transaction_type and location)

```

correlation_matrix = new_financial_df_encoded[transaction_type_columns + location_columns].corr()

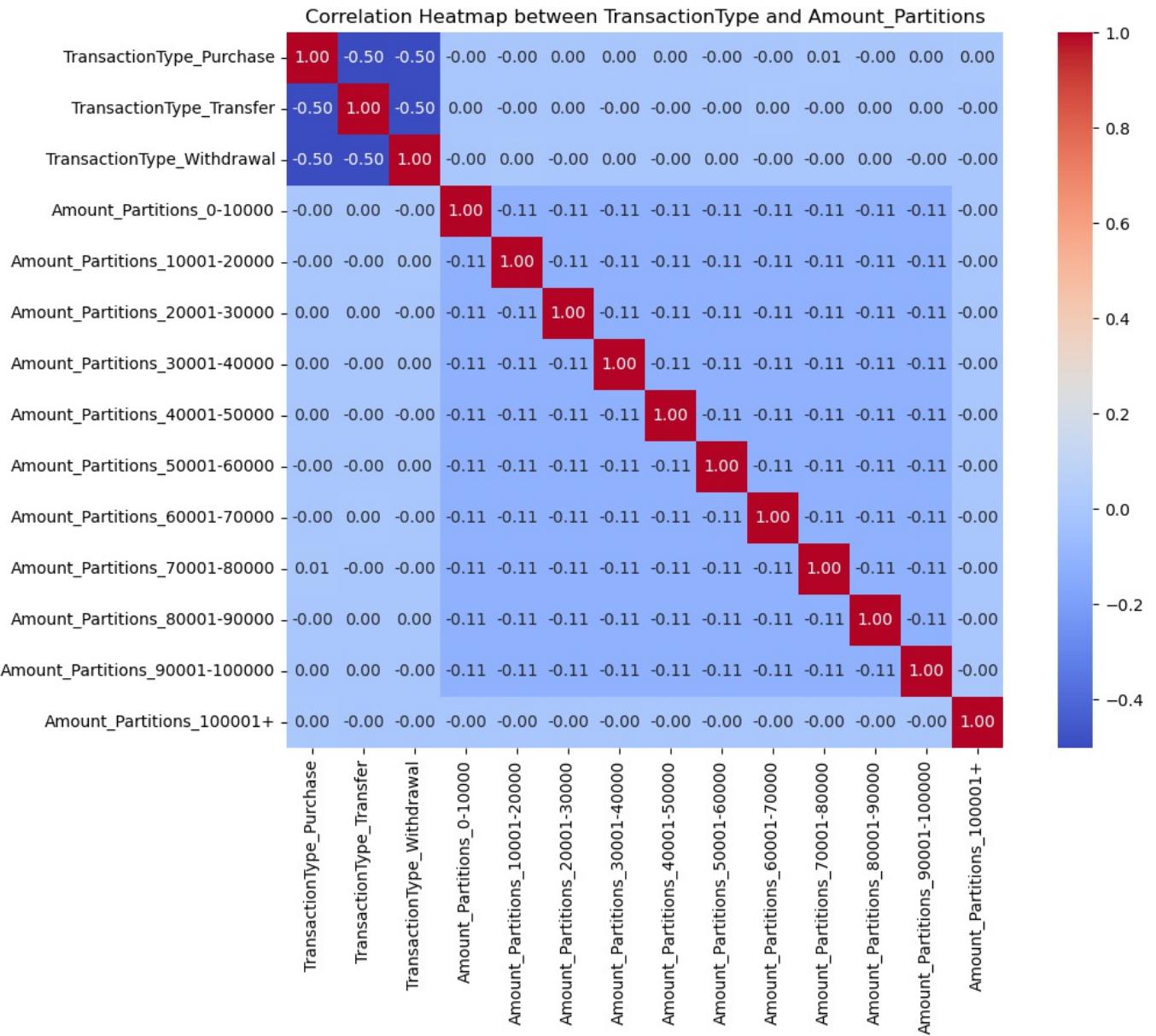
```

```
# Create a heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm', square=True)
plt.title('Correlation Heatmap between TransactionType and Location')
plt.show()
```



```
In [35]: #example of a correlation within the dataset (transaction_type and amount_partitions)
correlation_matrix = new_financial_df_encoded[transaction_type_columns + amount_partitions_columns].corr()

# Create a heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(correlation_matrix, annot=True, fmt=".2f", cmap='coolwarm', square=True)
plt.title('Correlation Heatmap between TransactionType and Amount_Partitions')
plt.show()
```



```
In [36]: #USE TRAIN, VALIDATION, AND TEST SET APPROACH
#create train set (70%) and temporary other set (30%)
train_df, temp_df = train_test_split(new_financial_df, test_size=0.30, random_state=1)

#split the leftover temp set into validation and test sets (50% of 30% each- 15% each)
validation_df, test_df = train_test_split(temp_df, test_size=0.50, random_state=42)

#verify shape of train, validation, and test DataFrames
print(f'Training set shape: {train_df.shape}')
print(f'Validation set shape: {validation_df.shape}')
print(f'Test set shape: {test_df.shape}')

Training set shape: (151872, 18)
Validation set shape: (32544, 18)
Test set shape: (32544, 18)
```

```
In [37]: #apply log transformation to Amount variable
train_df['Amount'] = np.log1p(train_df['Amount'])
```

```
In [38]: #identify trends in volume of transactions per day per account
train_df['Date'] = train_df['Timestamp'].dt.date
account_activity = train_df.groupby(['Date', 'AccountID']).agg(
    total_transactions=('Amount', 'count'),
    total_amount=('Amount', 'sum'),
    average_amount=('Amount', 'mean'),
    max_transaction=('Amount', 'max'),
    min_transaction=('Amount', 'min'),
).reset_index()

print(account_activity)
```

```

          Date AccountID total_transactions total_amount average_amount \
0  2023-01-01    ACC1           45  465.643415   10.347631
1  2023-01-01    ACC10          39  401.756535   10.301450
2  2023-01-01    ACC11          45  466.881311   10.375140
3  2023-01-01    ACC12          48  494.428680   10.300598
4  2023-01-01    ACC13          51  537.094450   10.531264
...
2260 2023-05-31    ACC5           79  829.043252   10.494218
2261 2023-05-31    ACC6           55  572.603696   10.410976
2262 2023-05-31    ACC7           61  652.687543   10.699796
2263 2023-05-31    ACC8           61  636.764154   10.438757
2264 2023-05-31    ACC9           76  804.576583   10.586534

      max_transaction min_transaction
0           11.455237     6.655865
1           11.486849     4.735672
2           11.449300     6.820377
3           11.504645     7.298147
4           11.502063     5.600198
...
2260         11.510775     7.355871
2261         11.506599     6.599966
2262         11.506273     8.001646
2263         11.481302     4.934834
2264         11.489540     7.732558

```

[2265 rows x 7 columns]

```
In [39]: #identify trends in volume of transactions per day per merchant
merchant_activity = train_df.groupby(['Date', 'Merchant']).agg(
    total_transactions=('Amount', 'count'),
    total_amount=('Amount', 'sum'),
    average_amount=('Amount', 'mean'),
    max_transaction=('Amount', 'max'),
    min_transaction=('Amount', 'min'),
).reset_index()

print(merchant_activity)
```

```

          Date Merchant total_transactions total_amount average_amount \
0  2023-01-01 MerchantA           65  693.193900   10.664522
1  2023-01-01 MerchantB           71  738.537511   10.401937
2  2023-01-01 MerchantC           83  862.766604   10.394778
3  2023-01-01 MerchantD           77  807.919846   10.492466
4  2023-01-01 MerchantE           51  529.932174   10.390827
...
1505 2023-05-31 MerchantF           98  1044.751467   10.660729
1506 2023-05-31 MerchantG          104  1091.297405   10.493244
1507 2023-05-31 MerchantH          100  1052.697256   10.526973
1508 2023-05-31 MerchantI          98  1024.712858   10.456254
1509 2023-05-31 MerchantJ          94  1000.775385   10.646547

      max_transaction min_transaction
0           11.484058     7.952207
1           11.503438     3.548180
2           11.479025     5.491950
3           11.488507     5.600198
4           11.491695     6.264293
...
1505         11.511874     8.208598
1506         11.511314     4.934834
1507         11.506599     3.700314
1508         11.501197     6.432731
1509         11.505241     8.141434

```

[1510 rows x 7 columns]

```
In [40]: #identify trends in volume of transactions per day by location
location_activity = train_df.groupby(['Date', 'Location']).agg(
    total_transactions=('Amount', 'count'),
    total_amount=('Amount', 'sum'),
    average_amount=('Amount', 'mean'),
    max_transaction=('Amount', 'max'),
    min_transaction=('Amount', 'min'),
).reset_index()

print(location_activity)
```

```

      Date      Location  total_transactions  total_amount \
0   2023-01-01        London                 146  1530.237727
1   2023-01-01    Los Angeles                 139  1454.136479
2   2023-01-01     New York                 135  1402.617642
3   2023-01-01  San Francisco                138  1445.854505
4   2023-01-01       Tokyo                  133  1410.359817
..          ...
750  2023-05-31        London                 205  2186.373655
751  2023-05-31    Los Angeles                 191  2004.248091
752  2023-05-31     New York                 213  2215.559017
753  2023-05-31  San Francisco                187  1982.163977
754  2023-05-31       Tokyo                  197  2080.626500

  average_amount  max_transaction  min_transaction
0           10.481080         11.502063        5.491950
1           10.461414         11.504645        3.548180
2           10.389760         11.507789        5.600198
3           10.477207         11.504023        4.735672
4           10.604209         11.499541        7.197413
..          ...
750          10.665237         11.511553        4.934834
751          10.493446         11.511874        6.505141
752          10.401686         11.511455        3.700314
753          10.599807         11.512224        4.856862
754          10.561556         11.506885        6.397313

```

[755 rows x 7 columns]

In [41]: `#verify current state of dataset
train_df.head(5)`

Out[41]:

	Timestamp	TransactionID	AccountID	Amount	Merchant	TransactionType	Location	AccountID/Merchant	AccountID/Ti
9230	2023-01-07 17:50:00	TXN1858	ACC12	9.064231	MerchantB	Withdrawal	London	ACC12_MerchantB	ACC
41764	2023-01-30 08:04:00	TXN76	ACC9	10.757187	MerchantJ	Transfer	London	ACC9_MerchantJ	A
136513	2023-04-06 03:13:00	TXN847	ACC11	10.996651	MerchantD	Transfer	New York	ACC11_MerchantD	AC
158548	2023-04-21 10:28:00	TXN852	ACC12	11.204528	MerchantI	Withdrawal	Los Angeles	ACC12_MerchantI	AC
9929	2023-01-08 05:29:00	TXN1822	ACC1	9.295688	MerchantF	Withdrawal	London	ACC1_MerchantF	AC

In [42]: `#drop columns with too many unique values to analyze efficiently
train_df.drop(columns=['TransactionID', 'AccountID/TransactionID', 'AccountID/Merchant/TransactionID', 'AccountID/Merch`

In [43]: `#up-to-date dataset
train_df.head(5)`

Out[43]:

	Timestamp	AccountID	Amount	Merchant	TransactionType	Location	Minute	Hour	Day	Month	Amount_Partitions
9230	2023-01-07 17:50:00	ACC12	9.064231	MerchantB	Withdrawal	London	50	17	5	1	0-10000
41764	2023-01-30 08:04:00	ACC9	10.757187	MerchantJ	Transfer	London	4	8	0	1	40001-50000
136513	2023-04-06 03:13:00	ACC11	10.996651	MerchantD	Transfer	New York	13	3	3	4	50001-60000
158548	2023-04-21 10:28:00	ACC12	11.204528	MerchantI	Withdrawal	Los Angeles	28	10	4	4	70001-80000
9929	2023-01-08 05:29:00	ACC1	9.295688	MerchantF	Withdrawal	London	29	5	6	1	10001-20000

In [44]: `#one hot encode categorical variables
train_encoded_df = pd.get_dummies(train_df, columns=['AccountID', 'Merchant', 'TransactionType', 'Location', 'A`

In [45]: `#dataset with one hot encoded categorical variables
train_encoded_df.head()`

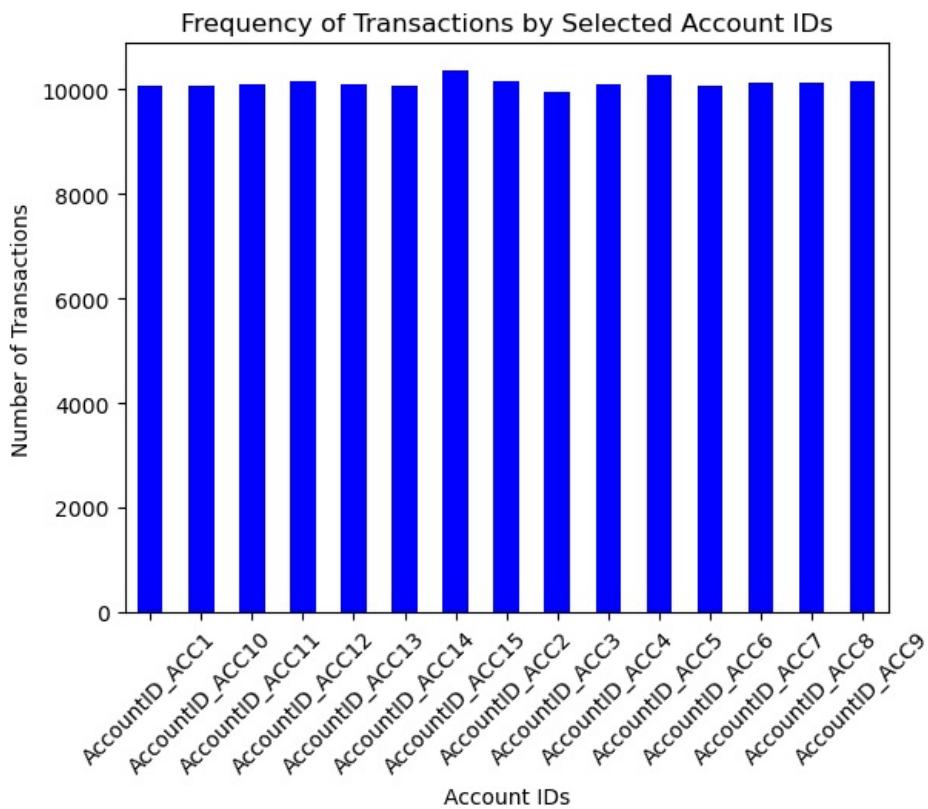
Out[45]:

		Timestamp	Amount	Minute	Hour	Month	Date	AccountID_ACC1	AccountID_ACC10	AccountID_ACC11	AccountID_AC
9230		2023-01-07 17:50:00	9.064231	50	17	1	2023-01-07		False	False	False
41764		2023-01-30 08:04:00	10.757187	4	8	1	2023-01-30		False	False	False
136513		2023-04-06 03:13:00	10.996651	13	3	4	2023-04-06		False	False	True
158548		2023-04-21 10:28:00	11.204528	28	10	4	2023-04-21		False	False	False
9929		2023-01-08 05:29:00	9.295688	29	5	1	2023-01-08		True	False	False

5 rows × 57 columns

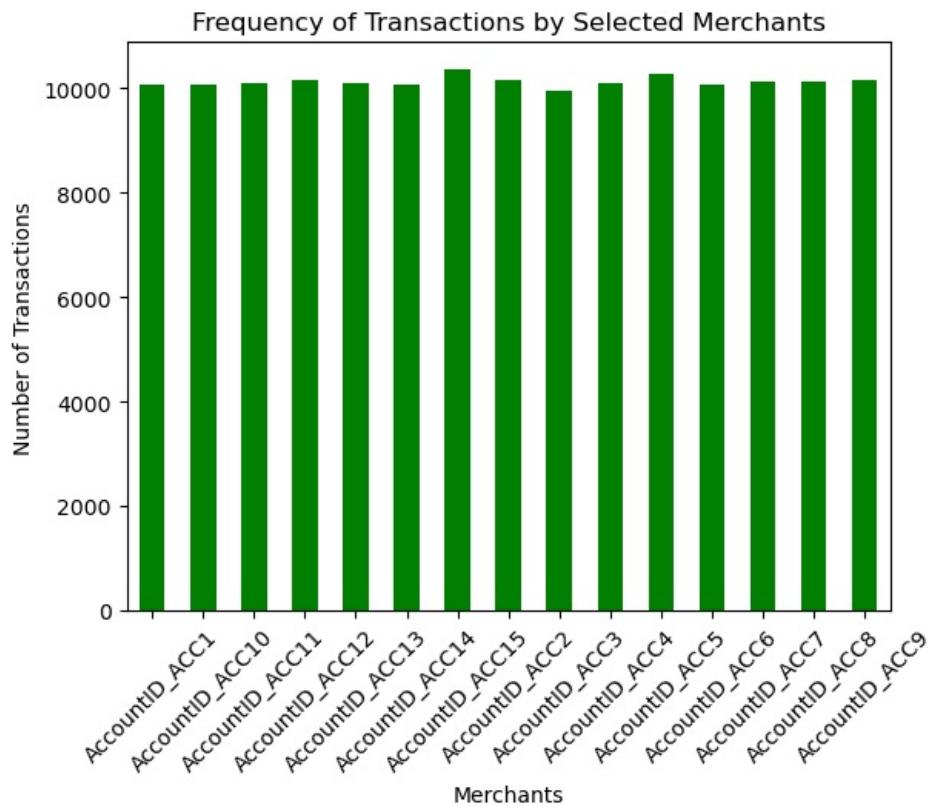
In [46]:

```
#visualize encoded AccountID Data
account_columns = [col for col in train_encoded_df.columns if col.startswith('AccountID_')]
account_counts = train_encoded_df[account_columns].sum()
account_counts.plot(kind='bar', color='blue')
plt.title('Frequency of Transactions by Selected Account IDs')
plt.xlabel('Account IDs')
plt.ylabel('Number of Transactions')
plt.xticks(rotation=45)
plt.show()
```

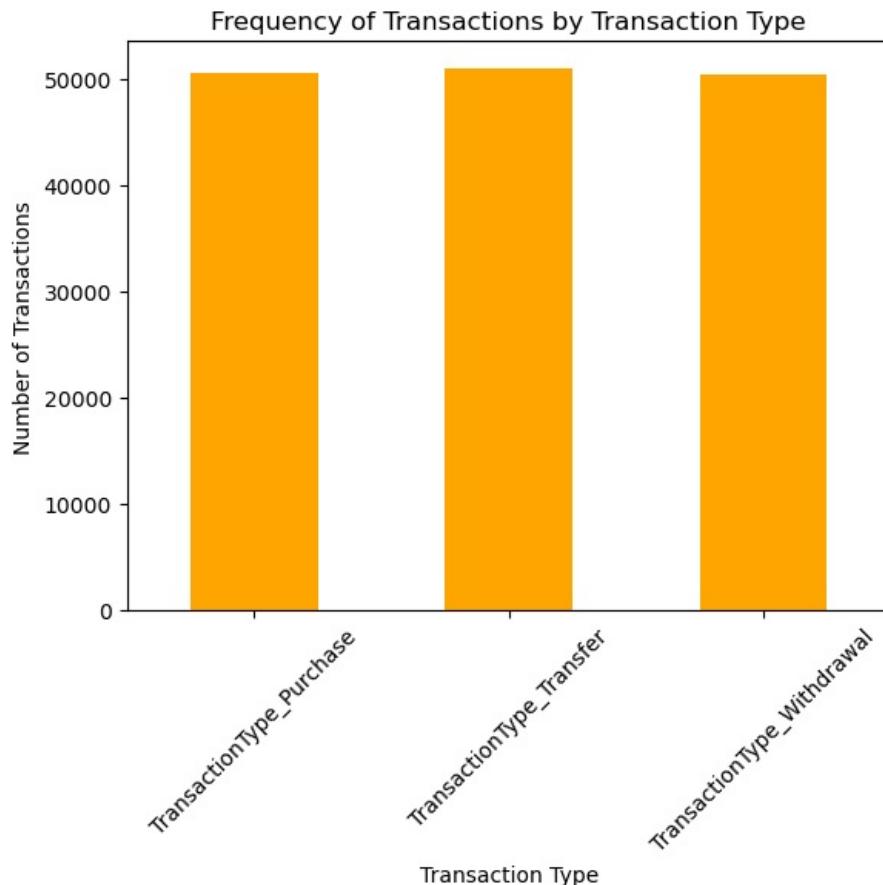


In [47]:

```
#visualize encoded AccountID Data
merchant_columns = [col for col in train_encoded_df.columns if col.startswith('Merchant_')]
merchant_counts = train_encoded_df[merchant_columns].sum()
merchant_counts.plot(kind='bar', color='green')
plt.title('Frequency of Transactions by Selected Merchants')
plt.xlabel('Merchants')
plt.ylabel('Number of Transactions')
plt.xticks(rotation=45)
plt.show()
```

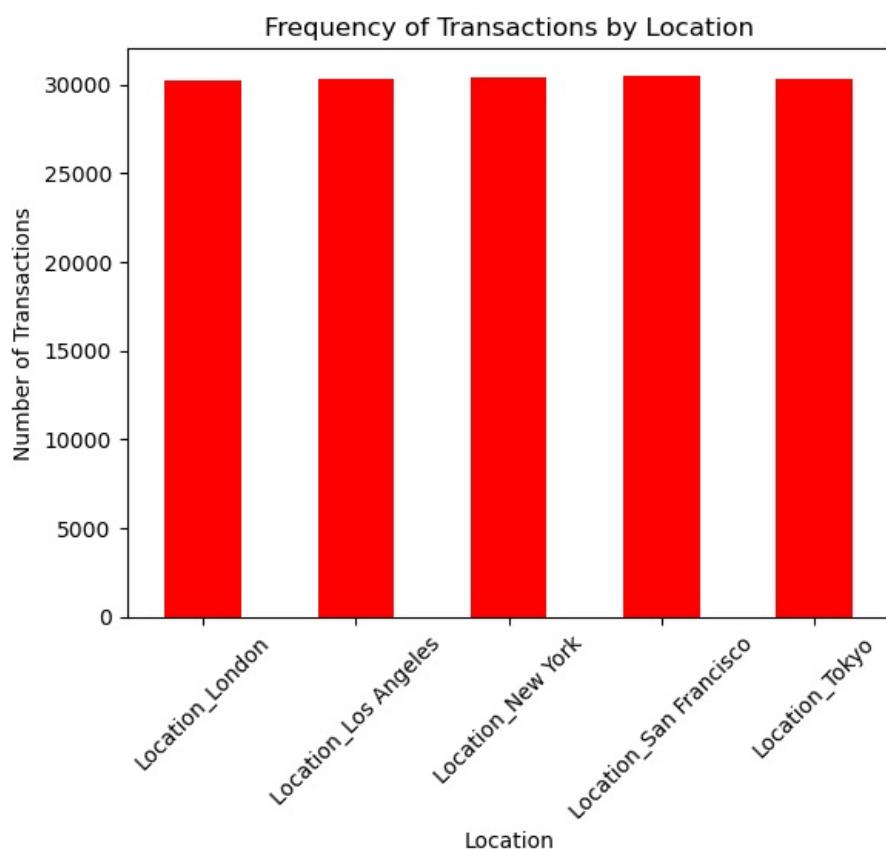


```
In [48]: #visualize encoded AccountID Data
TransactionType_columns = [col for col in train_encoded_df.columns if col.startswith('TransactionType_')]
TransactionType_counts = train_encoded_df[TransactionType_columns].sum()
TransactionType_counts.plot(kind='bar', color='orange')
plt.title('Frequency of Transactions by Transaction Type')
plt.xlabel('Transaction Type')
plt.ylabel('Number of Transactions')
plt.xticks(rotation=45)
plt.show()
```

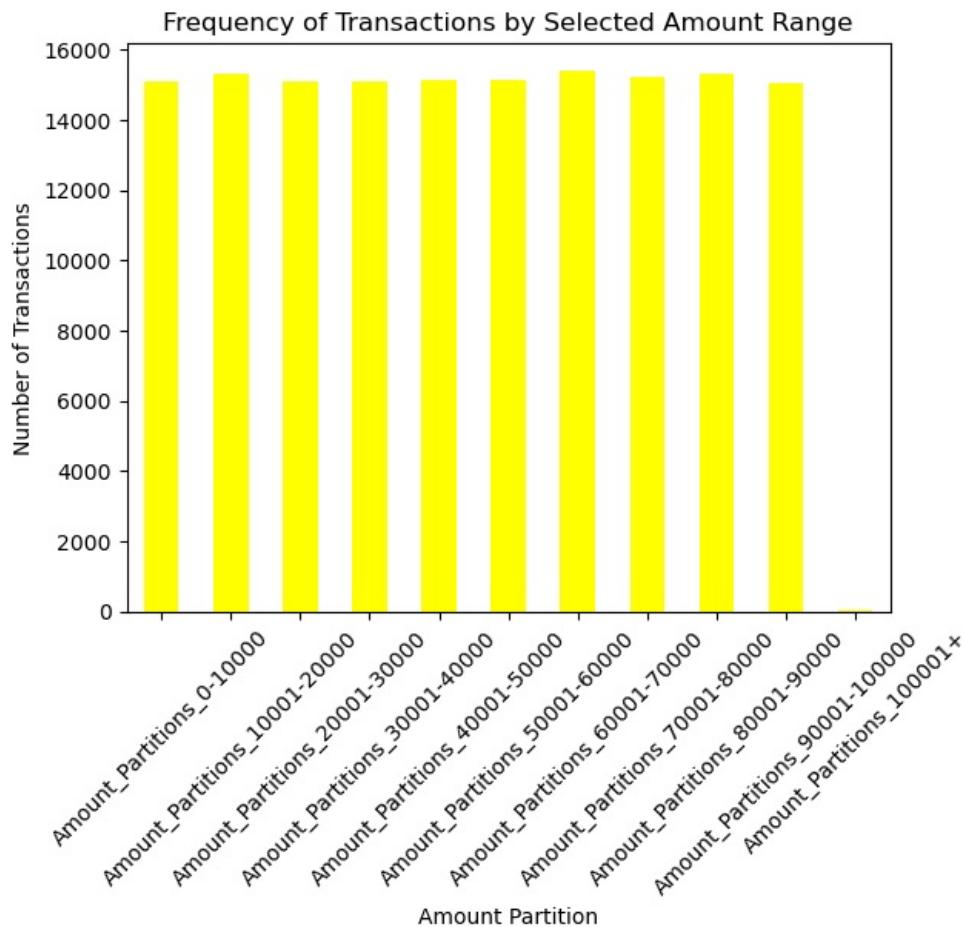


```
In [49]: #visualize encoded AccountID Data
location_columns = [col for col in train_encoded_df.columns if col.startswith('Location_')]
location_counts = train_encoded_df[location_columns].sum()
location_counts.plot(kind='bar', color='red')
plt.title('Frequency of Transactions by Location')
```

```
plt.xlabel('Location')
plt.ylabel('Number of Transactions')
plt.xticks(rotation=45)
plt.show()
```



```
In [50]: #visualize encoded AccountID Data
amount_partitions_columns = [col for col in train_encoded_df.columns if col.startswith('Amount_Partitions_')]
amount_partitions_counts = train_encoded_df[amount_partitions_columns].sum()
amount_partitions_counts.plot(kind='bar', color='yellow')
plt.title('Frequency of Transactions by Selected Amount Range')
plt.xlabel('Amount Partition')
plt.ylabel('Number of Transactions')
plt.xticks(rotation=45)
plt.show()
```



```
In [51]: #BIAS REDUCTION AND DATA LEAKAGE PREVENTION
#Perform same preprocessing steps on both Validation and Test Sets
#Apply log transformation to Amount variable
validation_df['Amount'] = np.log1p(validation_df['Amount'])
test_df['Amount'] = np.log1p(test_df['Amount'])

In [52]: #identify trends in volume of transactions per day per account (not printing results to reduce potential for bias)
validation_df['Date'] = validation_df['Timestamp'].dt.date
val_account_activity = validation_df.groupby(['Date', 'AccountID']).agg(
    total_transactions=('Amount', 'count'),
    total_amount=('Amount', 'sum'),
    average_amount=('Amount', 'mean'),
    max_transaction=('Amount', 'max'),
    min_transaction=('Amount', 'min'),
).reset_index()

In [53]: #identify trends in volume of transactions per day per account
test_df['Date'] = test_df['Timestamp'].dt.date
test_account_activity = test_df.groupby(['Date', 'AccountID']).agg(
    total_transactions=('Amount', 'count'),
    total_amount=('Amount', 'sum'),
    average_amount=('Amount', 'mean'),
    max_transaction=('Amount', 'max'),
    min_transaction=('Amount', 'min'),
).reset_index()

In [54]: #identify trends in volume of transactions per day per merchant
val_merchant_activity = validation_df.groupby(['Date', 'Merchant']).agg(
    total_transactions=('Amount', 'count'),
    total_amount=('Amount', 'sum'),
    average_amount=('Amount', 'mean'),
    max_transaction=('Amount', 'max'),
    min_transaction=('Amount', 'min'),
).reset_index()

In [55]: #identify trends in volume of transactions per day per merchant
test_merchant_activity = test_df.groupby(['Date', 'Merchant']).agg(
    total_transactions=('Amount', 'count'),
    total_amount=('Amount', 'sum'),
    average_amount=('Amount', 'mean'),
    max_transaction=('Amount', 'max'),
    min_transaction=('Amount', 'min'),
).reset_index()

In [56]: #identify trends in volume of transactions per day by location
```

```
val_location_activity = validation_df.groupby(['Date', 'Location']).agg(
    total_transactions=('Amount', 'count'),
    total_amount=('Amount', 'sum'),
    average_amount=('Amount', 'mean'),
    max_transaction=('Amount', 'max'),
    min_transaction=('Amount', 'min'),
).reset_index()
```

```
In [57]: #identify trends in volume of transactions per day by location
test_location_activity = test_df.groupby(['Date', 'Location']).agg(
    total_transactions=('Amount', 'count'),
    total_amount=('Amount', 'sum'),
    average_amount=('Amount', 'mean'),
    max_transaction=('Amount', 'max'),
    min_transaction=('Amount', 'min'),
).reset_index()
```

```
In [58]: #drop columns with too many unique values to analyze efficiently
validation_df.drop(columns=['TransactionID', 'AccountID/TransactionID', 'AccountID/Merchant/TransactionID', 'Ac
#drop columns with too many unique values to analyze efficiently
test_df.drop(columns=['TransactionID', 'AccountID/TransactionID', 'AccountID/Merchant/TransactionID', 'AccountID/Mercha
```

```
In [59]: #one hot encode categorical variables
validation_encoded_df = pd.get_dummies(validation_df, columns=['AccountID', 'Merchant', 'TransactionType', 'Locat
test_encoded_df = pd.get_dummies(test_df, columns=['AccountID', 'Merchant', 'TransactionType', 'Location', 'Amou
```

```
In [60]: #FEATURE ENGINEERING
# Define bins and labels for groups of hours of the day
bins = [0, 5, 11, 17, 23]
labels = ['0-5', '6-11', '12-17', '18-23']

# Create a new column 'Hour_Group' that bucketizes data into four segments of the day
train_encoded_df['Hour_Group'] = pd.cut(train_encoded_df['Hour'], bins=bins, labels=labels, right=True)
```

```
In [61]: #assign values of Hour_Group to each AccountID
for account in range(1, 15):
    AccountID_column = f'AccountID_ACC{account}'
    if AccountID_column in train_encoded_df.columns:
        train_encoded_df[f'Hour_Group_{account}'] = train_encoded_df.apply(
            lambda row: row['Hour_Group'] if row[AccountID_column] == 1 else None,
            axis=1
        )
```

```
In [62]: #repeat this action to assign values of Hour_Group to each Merchant, TransactionType, and Location
def create_hour_group_columns(train_encoded_df, variable_info, hour_group_column='Hour_Group'):
    for prefix, count in variable_info.items():
        for i in range(1, count + 1):
            column_name = f'{prefix}{i}'
            if column_name in train_encoded_df.columns:
                train_encoded_df[f'{column_name}_{hour_group_column}'] = train_encoded_df.apply(
                    lambda row: row[column_name] if row[column_name] == 1 else None,
                    axis=1
                )

#define the variable prefixes and their respective counts
variable_info = {
    'Merchant_Merchant': 10, #Merchants A-J
    'TransactionType_': 3,     # Purchase, Transfer, Withdrawal
    'Location_': 5           # London, Los Angeles, New York, San Francisco, Tokyo
}

#call the function to create the new columns
create_hour_group_columns(train_encoded_df, variable_info)
```

```
In [63]: #define a function to calculate mean and standard deviation for each one-hot encoded account by iterating across
def calculate_stats(train_encoded_df, account_prefix='AccountID_ACC', num_accounts=15):
    stats = {}

    for i in range(1, num_accounts + 1):
        account_columns = f'{account_prefix}{i}'
        #only include amounts where the specific account value is true (1 as its binary representation)
        account_data = train_encoded_df[train_encoded_df[account_columns] == 1]['Amount']

        #perform the mean and standard deviatoion calculations
        mean = account_data.mean()
        std = account_data.std()

        #store the mean and standard deviation for each account
        stats[f'AccountID_ACC{i}'] = {'mean': mean, 'std': std}
```

```

    return stats

#call the function to calculate mean and standard deviation for AccountID_ACC1 to AccountID_ACC15
account_stats = calculate_stats(train_encoded_df)

#add mean and standard deviation columns to train_encoded_df
for account, values in account_stats.items():
    train_encoded_df[f'{account}_mean'] = values['mean']
    train_encoded_df[f'{account}_std'] = values['std']

```

In [64]:

```

#create a new column for deviation from mean for each transaction
train_encoded_df['Deviation_From_Mean'] = 0.0

#loop through each account and calculate the deviation
for i in range(1, 15):
    account_columns = f'AccountID_ACC{i}'
    mean_columns = f'AccountID_ACC{i}_mean'
    std_columns = f'AccountID_ACC{i}_std'

    #calculate the deviation only for transactions in the current account
    condition = train_encoded_df[account_columns] == 1

    #calculate number of standard deviations of a transaction's Amount value from its Account's Amount mean
    train_encoded_df.loc[condition, 'Deviation_From_Mean'] = (
        (train_encoded_df.loc[condition, 'Amount'] - train_encoded_df.loc[condition, mean_columns])
    ) / train_encoded_df.loc[condition, std_columns])

```

In [65]:

```
#verify current state of dataset
train_encoded_df.head()
```

Out[65]:

	Timestamp	Amount	Minute	Hour	Month	Date	AccountID_ACC1	AccountID_ACC10	AccountID_ACC11	AccountID_AC
9230	2023-01-07 17:50:00	9.064231	50	17	1	2023-01-07	False	False	False	
41764	2023-01-30 08:04:00	10.757187	4	8	1	2023-01-30	False	False	False	
136513	2023-04-06 03:13:00	10.996651	13	3	4	2023-04-06	False	False	True	
158548	2023-04-21 10:28:00	11.204528	28	10	4	2023-04-21	False	False	False	
9929	2023-01-08 05:29:00	9.295688	29	5	1	2023-01-08	True	False	False	

5 rows × 103 columns

In [66]:

```

#create a function to plot the frequency of occurrence of transactions from a specific account at certain time
def plot_interaction_frequencies(df, account_columns, hour_group_column):

    #create a new DataFrame to hold the frequencies of specific accounts' transactions occurring in certain hours
    interaction_frequencies = df.groupby(account_columns + [hour_group_column], observed=False).size().reset_index()

    #create a pivot table for better visualization
    pivot_table = interaction_frequencies.pivot(index=hour_group_column, columns=account_columns, values='Frequency')

    #plotting
    plt.figure(figsize=(10, 6))
    sns.heatmap(pivot_table, cmap='YlGnBu', annot=True, fmt=".0f")
    plt.title('Frequencies of Transactions by AccountID (ACC1 to ACC3) and Hour Group')
    plt.xlabel('AccountID')
    plt.ylabel('Hour Group')
    plt.xticks(rotation=45)
    plt.tight_layout()
    plt.show()

hour_group_column = 'Hour_Group'
account_columns = [f'AccountID_ACC{i}' for i in range(1, 4)] #only ACC1 to ACC3 for better visualization purposes

#call the function
plot_interaction_frequencies(train_encoded_df, account_columns, hour_group_column)

```



```
In [67]: #GET THE DATA MODEL-READY
#create a list of prefixes for our one-hot columns
one_hot_prefixes = ['AccountID_', 'Merchant_', 'TransactionType_', 'Location_', 'Amount_Partitions_', 'Day']

#create a variable that stores the columns starting with the respective prefixes from our list
binary = train_encoded_df.columns.str.startswith(tuple(one_hot_prefixes))

#convert TRUE/FALSE entries to 1/0 entries with exception handling condition
if binary.any():
    try:
        #check data types of the selected columns
        for col in train_encoded_df.columns[binary]:
            #get the data type of the column
            column_dtype = train_encoded_df[col].dtype

            if column_dtype != 'int32':
                #convert directly to int if boolean
                if column_dtype == 'bool':
                    train_encoded_df[col] = train_encoded_df[col].astype(int)
                else:
                    #ensure int type
                    train_encoded_df[col] = train_encoded_df[col].astype('int32')
    except Exception as e:
        print(f"Potential incompatible dtype error during conversion: {e}")
    else:
        print("No one-hot encoded columns found with the specified prefixes.")
else:
    print("No one-hot encoded columns found with the specified prefixes.")
```

```
In [68]: #conduct embedding for dimensionality reduction purposes
#create lists for AccountID, Merchant, TransactionType, Location, and Hour Groups
account_ids = [f'AccountID_ACC{i}' for i in range(1, 15)] # AccountID's 1 to 15
merchants = [f'Merchant_Merchant{chr(i)}' for i in range(ord('A'), ord('J') + 1)] # Merchants A to J
transaction_types = ['TransactionType_Purchase', 'TransactionType_Transfer', 'TransactionType_Withdrawal']
locations = ['Location_London', 'Location_Los Angeles', 'Location_New York', 'Location_San Francisco', 'Location_Tokyo']
hour_groups = [f'Hour_Group_{i}' for i in range(1, 15)]

def encode_columns(df, columns):
    encoder = LabelEncoder()
    encoded_columns = {}

    for column in columns:
        if column in df.columns:
            encoded_columns[column] = encoder.fit_transform(df[column])
        else:
            print(f"Warning: {column} not found in DataFrame.")
    return encoded_columns

#encode each variable
encoded_account_ids = encode_columns(train_encoded_df, account_ids)
encoded_merchants = encode_columns(train_encoded_df, merchants)
```

```

encoded_transaction_types = encode_columns(train_encoded_df, transaction_types)
encoded_locations = encode_columns(train_encoded_df, locations)
encoded_hour_groups = encode_columns(train_encoded_df, hour_groups)

#assign the encoded values back to the DataFrame
for account_id, encoded_values in encoded_account_ids.items():
    train_encoded_df[account_id] = encoded_values

for merchant, encoded_values in encoded_merchants.items():
    train_encoded_df[merchant] = encoded_values

for transaction_type, encoded_values in encoded_transaction_types.items():
    train_encoded_df[transaction_type] = encoded_values

for location, encoded_values in encoded_locations.items():
    train_encoded_df[location] = encoded_values

for hour_group, encoded_values in encoded_hour_groups.items():
    train_encoded_df[hour_group] = encoded_values

#prepare Inputs for Embedding
numerical_input = Input(shape=(1,), name='numerical_input')
account_input = Input(shape=(1,), name='account_input')
merchant_input = Input(shape=(1,), name='merchant_input')
transaction_input = Input(shape=(1,), name='transaction_input')
location_input = Input(shape=(1,), name='location_input')
hour_group_input = Input(shape=(1,), name='hour_group_input')

#create Embedding Layers
embedding_dim = 8
num_accounts = len(account_ids)
num_merchants = len(merchants)
num_transaction_types = len(transaction_types)
num_locations = len(locations)
num_hour_groups = len(hour_groups)

#embeddings for each one-hot encoded category
account_embedding = Embedding(input_dim=num_accounts, output_dim=embedding_dim)(account_input)
merchant_embedding = Embedding(input_dim=num_merchants, output_dim=embedding_dim)(merchant_input)
transaction_embedding = Embedding(input_dim=num_transaction_types, output_dim=embedding_dim)(transaction_input)
location_embedding = Embedding(input_dim=num_locations, output_dim=embedding_dim)(location_input)
hour_group_embedding = Embedding(input_dim=num_hour_groups, output_dim=embedding_dim)(hour_group_input)

#flatten the embeddings to make a one-dimensional array representation of the variables
flattened_account = Flatten()(account_embedding)
flattened_merchant = Flatten()(merchant_embedding)
flattened_transaction = Flatten()(transaction_embedding)
flattened_location = Flatten()(location_embedding)
flattened_hour_group = Flatten()(hour_group_embedding)

#concatenate inputs to a single output
concat = Concatenate()([numerical_input, flattened_account, flattened_merchant, flattened_transaction, flattened_location, flattened_hour_group])

#add Dense Layers to interconnect previous layers
output = Dense(1, activation='sigmoid')(concat)

#build the Model
model = Model(inputs=[numerical_input, account_input, merchant_input, transaction_input, location_input, hour_group_input], outputs=[output])
model.compile(optimizer='adam', loss='mean_squared_error')

#prepare input data for model training
X_numerical = train_encoded_df[['Amount']].values
X_accounts = train_encoded_df[account_ids].values.argmax(axis=1).reshape(-1, 1) # Get index for account input
X_merchants = train_encoded_df[merchants].values.argmax(axis=1).reshape(-1, 1)
X_transaction_types = train_encoded_df[transaction_types].values.argmax(axis=1).reshape(-1, 1)
X_locations = train_encoded_df[locations].values.argmax(axis=1).reshape(-1, 1)
X_hour_groups = train_encoded_df[hour_groups].values.argmax(axis=1).reshape(-1, 1)

#make sure to pass the inputs as a list
model.fit([X_numerical, X_accounts, X_merchants, X_transaction_types, X_locations, X_hour_groups],
          np.zeros(X_numerical.shape[0]),
          epochs=1,
          batch_size=16)

#create a model to get the embedding outputs
embedding_model = Model(inputs=[account_input, merchant_input, transaction_input, location_input, hour_group_input], outputs=[account_embedding, merchant_embedding, transaction_embedding, location_embedding])

#get the embedding outputs
embedding_output = embedding_model.predict([X_accounts, X_merchants, X_transaction_types, X_locations, X_hour_groups])

#store the embeddings in a DataFrame
embedding_df = pd.DataFrame({

```

```

'Account_EMBEDDINGS': list(embedding_output[0]),
'Merchant_EMBEDDINGS': list(embedding_output[1]),
'Transaction_EMBEDDINGS': list(embedding_output[2]),
'Location_EMBEDDINGS': list(embedding_output[3]),
'Hour_Group_EMBEDDINGS': list(embedding_output[4])
})

#combine with the original DataFrame
train_embeddings_df = pd.concat([train_encoded_df.reset_index(drop=True), embedding_df.reset_index(drop=True)], axis=1)

```

C:\Users\bbrady\OneDrive\Apps\Anaconda\Lib\site-packages\keras\src\models\functional.py:225: UserWarning: The structure of `inputs` doesn't match the expected structure: ['numerical_input', 'account_input', 'merchant_input', 'transaction_input', 'location_input', 'hour_group_input']. Received: the structure of inputs=('*', '*', '*', '*', '*', '*')
warnings.warn(
9492/9492 ━━━━━━━━ 14s 1ms/step - loss: 0.0125
C:\Users\bbrady\OneDrive\Apps\Anaconda\Lib\site-packages\keras\src\models\functional.py:225: UserWarning: The structure of `inputs` doesn't match the expected structure: ['account_input', 'merchant_input', 'transaction_input', 'location_input', 'hour_group_input']. Received: the structure of inputs=('*', '*', '*', '*', '*', '*')
warnings.warn(
4746/4746 ━━━━━━━━ 7s 1ms/step

In [69]: #new dataset with embeddings
train_embeddings_df.head()

Out[69]:

	Timestamp	Amount	Minute	Hour	Month	Date	AccountID_ACC1	AccountID_ACC10	AccountID_ACC11	AccountID_ACC12
0	2023-01-07 17:50:00	9.064231	50	17	1	2023-01-07	0	0	0	1
1	2023-01-30 08:04:00	10.757187	4	8	1	2023-01-30	0	0	0	0
2	2023-04-06 03:13:00	10.996651	13	3	4	2023-04-06	0	0	1	0
3	2023-04-21 10:28:00	11.204528	28	10	4	2023-04-21	0	0	0	1
4	2023-01-08 05:29:00	9.295688	29	5	1	2023-01-08	1	0	0	0

5 rows × 108 columns

In [70]: #create a list of column names to view all columns and determine which ones are not necessary
print(train_embeddings_df.columns.tolist())

```

['Timestamp', 'Amount', 'Minute', 'Hour', 'Month', 'Date', 'AccountID_ACC1', 'AccountID_ACC10', 'AccountID_ACC11',
 'AccountID_ACC12', 'AccountID_ACC13', 'AccountID_ACC14', 'AccountID_ACC15', 'AccountID_ACC2', 'AccountID_ACC3',
 'AccountID_ACC4', 'AccountID_ACC5', 'AccountID_ACC6', 'AccountID_ACC7', 'AccountID_ACC8', 'AccountID_ACC9',
 'Merchant_MerchantA', 'Merchant_MerchantB', 'Merchant_MerchantC', 'Merchant_MerchantD', 'Merchant_MerchantE',
 'Merchant_MerchantF', 'Merchant_MerchantG', 'Merchant_MerchantH', 'Merchant_MerchantI', 'Merchant_MerchantJ',
 'TransactionType_Purchase', 'TransactionType_Transfer', 'TransactionType_Withdrawal', 'Location_London',
 'Location_Los_Angeles', 'Location_New_York', 'Location_San_Francisco', 'Location_Tokyo', 'Amount_Partitions_0-10000',
 'Amount_Partitions_10001-20000', 'Amount_Partitions_20001-30000', 'Amount_Partitions_30001-40000', 'Amount_Partitions_40001-50000',
 'Amount_Partitions_50001-60000', 'Amount_Partitions_60001-70000', 'Amount_Partitions_70001-80000',
 'Amount_Partitions_80001-90000', 'Amount_Partitions_90001-100000', 'Amount_Partitions_100001+', 'Day_0', 'Day_1',
 'Day_2', 'Day_3', 'Day_4', 'Day_5', 'Day_6', 'Hour_Group', 'Hour_Group_1', 'Hour_Group_2', 'Hour_Group_3',
 'Hour_Group_4', 'Hour_Group_5', 'Hour_Group_6', 'Hour_Group_7', 'Hour_Group_8', 'Hour_Group_9', 'Hour_Group_10',
 'Hour_Group_11', 'Hour_Group_12', 'Hour_Group_13', 'Hour_Group_14', 'AccountID_ACC1_mean', 'AccountID_ACC1_std',
 'AccountID_ACC2_mean', 'AccountID_ACC2_std', 'AccountID_ACC3_mean', 'AccountID_ACC3_std', 'AccountID_ACC4_mean',
 'AccountID_ACC4_std', 'AccountID_ACC5_mean', 'AccountID_ACC5_std', 'AccountID_ACC6_mean', 'AccountID_ACC6_std',
 'AccountID_ACC7_mean', 'AccountID_ACC7_std', 'AccountID_ACC8_mean', 'AccountID_ACC8_std', 'AccountID_ACC9_mean',
 'AccountID_ACC9_std', 'AccountID_ACC10_mean', 'AccountID_ACC10_std', 'AccountID_ACC11_mean', 'AccountID_ACC11_std',
 'AccountID_ACC12_mean', 'AccountID_ACC12_std', 'AccountID_ACC13_mean', 'AccountID_ACC13_std', 'AccountID_ACC14_mean',
 'AccountID_ACC14_std', 'AccountID_ACC15_mean', 'AccountID_ACC15_std', 'Deviation_From_Mean', 'Account_EMBEDDINGS',
 'Merchant_EMBEDDINGS', 'Transaction_EMBEDDINGS', 'Location_EMBEDDINGS', 'Hour_Group_EMBEDDINGS']

```

In [71]: #list of columns to drop (already converted to one-hot/embeddings or not relevant to problem statement resolution)
columns_to_drop = [
 'Minute',
 'Month',
 'Hour',
 'Hour_Group'
] + [f'AccountID_ACC{i}' for i in range(1, 16)] + \
[f'Merchant_Merchant{chr(i)}' for i in range(ord('A'), ord('J') + 1)] + \
['TransactionType_Purchase', 'TransactionType_Transfer', 'TransactionType_Withdrawal'] + \
['Location_London', 'Location_Los_Angeles', 'Location_New_York',
 'Location_San_Francisco', 'Location_Tokyo'] + \

```

[f'Amount_Partitions_{i}' for i in ['0-10000', '10001-20000', '20001-30000',
                                    '30001-40000', '40001-50000', '50001-60000',
                                    '60001-70000', '70001-80000', '80001-90000',
                                    '90001-100000', '100001+']] + \
[f'AccountID_ACC{i}_mean' for i in range(1, 16)] + \
[f'AccountID_ACC{i}_std' for i in range(1, 16)] + \
[f'Hour_Group_{i}' for i in range(1, 15)]

#drop the specified columns
train_embeddings_df = train_embeddings_df.drop(columns=columns_to_drop)

#check the remaining columns
print(train_embeddings_df.columns.tolist())

```

['Timestamp', 'Amount', 'Date', 'Day_0', 'Day_1', 'Day_2', 'Day_3', 'Day_4', 'Day_5', 'Day_6', 'Deviation_From_Mean', 'Account_EMBEDDINGS', 'Merchant_EMBEDDINGS', 'Transaction_EMBEDDINGS', 'Location_EMBEDDINGS', 'Hour_Group_EMBEDDINGS']

In [72]: #repeat exact same steps for both validation and test sets

```

#create a new column 'Hour_Group' that bucketizes data into four segments of the day
validation_encoded_df['Hour_Group'] = pd.cut(validation_encoded_df['Hour'], bins=bins, labels=labels, right=True)

#assign values of Hour_Group to each AccountID
for account in range(1, 15):
    AccountID_column = f'AccountID_ACC{account}'
    if AccountID_column in validation_encoded_df.columns:
        validation_encoded_df[f'Hour_Group_{account}'] = validation_encoded_df.apply(
            lambda row: row['Hour_Group'] if row[AccountID_column] == 1 else None,
            axis=1
        )

#repeat this action to assign values of Hour_Group to each Merchant, TransactionType, and Location
def create_hour_group_columns(validation_encoded_df, variable_info, hour_group_column='Hour_Group'):
    for prefix, count in variable_info.items():
        for i in range(1, count + 1):
            column_name = f'{prefix}{i}'
            if column_name in validation_encoded_df.columns:
                validation_encoded_df[f'{column_name}_{hour_group_column}'] = validation_encoded_df.apply(
                    lambda row: row[hour_group_col] if row[column_name] == 1 else None,
                    axis=1
                )

#define the variable prefixes and their respective counts
variable_info = {
    'Merchant_Merchant': 10, # Merchants A-J
    'TransactionType_': 3,     # Purchase, Transfer, Withdrawal
    'Location_': 5           # London, Los Angeles, New York, San Francisco, Tokyo
}

#define the function to create the new columns
create_hour_group_columns(validation_encoded_df, variable_info)

#define a function to calculate mean and standard deviation for each one-hot encoded account by iterating across accounts
def calculate_stats(validation_encoded_df, account_prefix='AccountID_ACC', num_accounts=15):
    stats = {}

    for i in range(1, num_accounts + 1):
        account_columns = f'{account_prefix}{i}'
        #only include amounts where the specific account value is true (1 as its binary representation)
        account_data = validation_encoded_df[validation_encoded_df[account_columns] == 1]['Amount']

        #perform the mean and standard deviatoion calculations
        mean = account_data.mean()
        std = account_data.std()

        #store the mean and standard deviation for each account
        stats[f'AccountID_ACC{i}'] = {'mean': mean, 'std': std}

    return stats

#call the function to calculate mean and standard deviation for AccountID_ACC1 to AccountID_ACC15
account_stats = calculate_stats(validation_encoded_df)

#add mean and standard deviation columns to validation_encoded_df
for account, values in account_stats.items():
    validation_encoded_df[f'{account}_mean'] = values['mean']
    validation_encoded_df[f'{account}_std'] = values['std']

#create a new column for deviation from mean for each transaction
validation_encoded_df['Deviation_From_Mean'] = 0.0

#loop through each account and calculate the deviation
for i in range(1, 15):

```

```

account_columns = f'AccountID_ACC{i}'
mean_columns = f'AccountID_ACC{i}_mean'
std_columns = f'AccountID_ACC{i}_std'

#calculate the deviation only for transactions in the current account
condition = validation_encoded_df[account_columns] == 1

#calculate number of standard deviations of a transaction's Amount value from its Account's Amount mean
validation_encoded_df.loc[condition, 'Deviation_From_Mean'] = (
    (validation_encoded_df.loc[condition, 'Amount'] - validation_encoded_df.loc[condition, mean_columns]) /
    validation_encoded_df.loc[condition, std_columns])

#create a list of prefixes for our one-hot columns
one_hot_prefixes = ['AccountID_', 'Merchant_', 'TransactionType_', 'Location_', 'Amount_Partitions_', 'Day']

#create a variable that stores the columns starting with the respective prefixes from our list
binary = validation_encoded_df.columns.str.startswith(tuple(one_hot_prefixes))

#convert TRUE/FALSE entries to 1/0 entries with exception handling condition
if binary.any():
    try:
        #check data types of the selected columns
        for col in validation_encoded_df.columns[binary]:
            #get the data type of the column
            column_dtype = validation_encoded_df[col].dtype

            if column_dtype != 'int32':
                #convert directly to int if boolean
                if column_dtype == 'bool':
                    validation_encoded_df[col] = validation_encoded_df[col].astype(int)
                else:
                    #ensure int type
                    validation_encoded_df[col] = validation_encoded_df[col].astype('int32')
    except Exception as e:
        print(f"Potential incompatible dtype error during conversion: {e}")
    else:
        print("No one-hot encoded columns found with the specified prefixes.")

#create lists for AccountID, Merchant, TransactionType, Location, and Hour Groups
account_ids = [f'AccountID_ACC{i}' for i in range(1, 15)] # AccountID's 1 to 15
merchants = [f'Merchant_Merchant{chr(i)}' for i in range(ord('A'), ord('J') + 1)] # Merchants A to J
transaction_types = ['TransactionType_Purchase', 'TransactionType_Transfer', 'TransactionType_Withdrawal']
locations = ['Location_London', 'Location_Los Angeles', 'Location_New York', 'Location_San Francisco', 'Location_Sydney']
hour_groups = [f'Hour_Group_{i}' for i in range(1, 15)]

def encode_columns(df, columns):
    encoder = LabelEncoder()
    encoded_columns = {}

    for column in columns:
        if column in df.columns:
            encoded_columns[column] = encoder.fit_transform(df[column])
        else:
            print(f"Warning: {column} not found in DataFrame.")
    return encoded_columns

#encode each variable
encoded_account_ids = encode_columns(validation_encoded_df, account_ids)
encoded_merchants = encode_columns(validation_encoded_df, merchants)
encoded_transaction_types = encode_columns(validation_encoded_df, transaction_types)
encoded_locations = encode_columns(validation_encoded_df, locations)
encoded_hour_groups = encode_columns(validation_encoded_df, hour_groups)

#assign the encoded values back to the DataFrame
for account_id, encoded_values in encoded_account_ids.items():
    validation_encoded_df[account_id] = encoded_values

for merchant, encoded_values in encoded_merchants.items():
    validation_encoded_df[merchant] = encoded_values

for transaction_type, encoded_values in encoded_transaction_types.items():
    validation_encoded_df[transaction_type] = encoded_values

for location, encoded_values in encoded_locations.items():
    validation_encoded_df[location] = encoded_values

for hour_group, encoded_values in encoded_hour_groups.items():
    validation_encoded_df[hour_group] = encoded_values

#prepare Inputs for Embedding
numerical_input = Input(shape=(1,), name='numerical_input')
account_input = Input(shape=(1,), name='account_input')
merchant_input = Input(shape=(1,), name='merchant_input')

```

```

transaction_input = Input(shape=(1,), name='transaction_input')
location_input = Input(shape=(1,), name='location_input')
hour_group_input = Input(shape=(1,), name='hour_group_input')

#create Embedding Layers
embedding_dim = 8
num_accounts = len(account_ids)
num_merchants = len(merchants)
num_transaction_types = len(transaction_types)
num_locations = len(locations)
num_hour_groups = len(hour_groups)

#embeddings for each one-hot encoded category
account_embedding = Embedding(input_dim=num_accounts, output_dim=embedding_dim)(account_input)
merchant_embedding = Embedding(input_dim=num_merchants, output_dim=embedding_dim)(merchant_input)
transaction_embedding = Embedding(input_dim=num_transaction_types, output_dim=embedding_dim)(transaction_input)
location_embedding = Embedding(input_dim=num_locations, output_dim=embedding_dim)(location_input)
hour_group_embedding = Embedding(input_dim=num_hour_groups, output_dim=embedding_dim)(hour_group_input)

#flatten the embeddings to make a one-dimensional array representation of the variables
flattened_account = Flatten()(account_embedding)
flattened_merchant = Flatten()(merchant_embedding)
flattened_transaction = Flatten()(transaction_embedding)
flattened_location = Flatten()(location_embedding)
flattened_hour_group = Flatten()(hour_group_embedding)

#concatenate inputs to a single output
concat = Concatenate()([numerical_input, flattened_account, flattened_merchant, flattened_transaction, flattened_location, flattened_hour_group])

#add Dense Layers to interconnect previous layers
output = Dense(1, activation='sigmoid')(concat)

#build the Model
model = Model(inputs=[numerical_input, account_input, merchant_input, transaction_input, location_input, hour_group_input], outputs=output)
model.compile(optimizer='adam', loss='mean_squared_error')

#prepare input data for model training
X_numerical = validation_encoded_df[['Amount']].values
X_accounts = validation_encoded_df[account_ids].values.argmax(axis=1).reshape(-1, 1) # Get index for account in validation_encoded_df
X_merchants = validation_encoded_df[merchants].values.argmax(axis=1).reshape(-1, 1)
X_transaction_types = validation_encoded_df[transaction_types].values.argmax(axis=1).reshape(-1, 1)
X_locations = validation_encoded_df[locations].values.argmax(axis=1).reshape(-1, 1)
X_hour_groups = validation_encoded_df[hour_groups].values.argmax(axis=1).reshape(-1, 1)

#make sure to pass the inputs as a list
model.fit([X_numerical, X_accounts, X_merchants, X_transaction_types, X_locations, X_hour_groups],
          np.zeros(X_numerical.shape[0]),
          epochs=1,
          batch_size=16)

#create a model to get the embedding outputs
embedding_model = Model(inputs=[account_input, merchant_input, transaction_input, location_input, hour_group_input], outputs=[account_embedding, merchant_embedding, transaction_embedding, location_embedding])

#get the embedding outputs
embedding_output = embedding_model.predict([X_accounts, X_merchants, X_transaction_types, X_locations, X_hour_groups])

#store the embeddings in a DataFrame
embedding_df = pd.DataFrame({
    'Account_EMBEDDINGS': list(embedding_output[0]),
    'Merchant_EMBEDDINGS': list(embedding_output[1]),
    'Transaction_EMBEDDINGS': list(embedding_output[2]),
    'Location_EMBEDDINGS': list(embedding_output[3]),
    'Hour_Group_EMBEDDINGS': list(embedding_output[4])
})

#combine with the original DataFrame
validation_embeddings_df = pd.concat([validation_encoded_df.reset_index(drop=True), embedding_df.reset_index(drop=True)], axis=1)

#list of columns to drop (already converted to one-hot/embeddings or not relevant to problem statement resolution)
columns_to_drop = [
    'Minute',
    'Month',
    'Hour',
    'Hour_Group'
] + [f'AccountID_ACC{i}' for i in range(1, 16)] + \
[f'Merchant_Merchant{chr(i)}' for i in range(ord('A'), ord('J') + 1)] + \
['TransactionType_Purchase', 'TransactionType_Transfer', 'TransactionType_Withdrawal'] + \
['Location_London', 'Location_Los Angeles', 'Location_New York', 'Location_San Francisco', 'Location_Tokyo'] + \
[f'Amount_Partitions_{i}' for i in ['0-10000', '10001-20000', '20001-30000', '30001-40000', '40001-50000', '50001-60000', '60001-70000', '70001-80000', '80001-90000']]

```

```

'90001-100000', '100001+']] + \
[f'AccountID_ACC{i}_mean' for i in range(1, 16)] + \
[f'AccountID_ACC{i}_std' for i in range(1, 16)] + \
[f'Hour_Group_{i}' for i in range(1, 15)]

#drop the specified columns
validation_embeddings_df = validation_embeddings_df.drop(columns=columns_to_drop)

```

```

C:\Users\b Brady\OneDrive\apps\Anaconda\Lib\site-packages\keras\src\models\functional.py:225: UserWarning: The structure of `inputs` doesn't match the expected structure: ['numerical_input', 'account_input', 'merchant_input', 'transaction_input', 'location_input', 'hour_group_input']. Received: the structure of inputs='(*, *, *, *, *, *)'
  warnings.warn(
2034/2034 ━━━━━━━━ 5s 2ms/step - loss: 0.0591
  35/1017 ━━━━━━ 1s 1ms/step
C:\Users\b Brady\OneDrive\apps\Anaconda\Lib\site-packages\keras\src\models\functional.py:225: UserWarning: The structure of `inputs` doesn't match the expected structure: ['account_input', 'merchant_input', 'transaction_input', 'location_input', 'hour_group_input']. Received: the structure of inputs='(*, *, *, *, *, *)'
  warnings.warn(
  1017/1017 ━━━━━━ 2s 1ms/step

```

```

In [73]: #create a new column 'Hour_Group' that bucketizes data into four segments of the day
test_encoded_df['Hour_Group'] = pd.cut(test_encoded_df['Hour'], bins=bins, labels=labels, right=True)

#assign values of Hour_Group to each AccountID
for account in range(1, 15):
    AccountID_column = f'AccountID_ACC{account}'
    if AccountID_column in test_encoded_df.columns:
        test_encoded_df[f'Hour_Group_{account}'] = test_encoded_df.apply(
            lambda row: row['Hour_Group'] if row[AccountID_column] == 1 else None,
            axis=1
        )

#repeat this action to assign values of Hour_Group to each Merchant, TransactionType, and Location
def create_hour_group_columns(test_encoded_df, variable_info, hour_group_column='Hour_Group'):
    for prefix, count in variable_info.items():
        for i in range(1, count + 1):
            column_name = f'{prefix}{i}'
            if column_name in test_encoded_df.columns:
                test_encoded_df[f'{column_name}_{hour_group_column}'] = test_encoded_df.apply(
                    lambda row: row[hour_group_col] if row[column_name] == 1 else None,
                    axis=1
                )

#define the variable prefixes and their respective counts
variable_info = {
    'Merchant_Merchant': 10, # Merchants A-J
    'TransactionType_': 3, # Purchase, Transfer, Withdrawal
    'Location_': 5 # London, Los Angeles, New York, San Francisco, Tokyo
}

#call the function to create the new columns
create_hour_group_columns(test_encoded_df, variable_info)

#define a function to calculate mean and standard deviation for each one-hot encoded account by iterating across
def calculate_stats(test_encoded_df, account_prefix='AccountID_ACC', num_accounts=15):
    stats = {}

    for i in range(1, num_accounts + 1):
        account_columns = f'{account_prefix}{i}'
        #only include amounts where the specific account value is true (1 as its binary representation)
        account_data = test_encoded_df[test_encoded_df[account_columns] == 1]['Amount']

        #perform the mean and standard deviatoin calculations
        mean = account_data.mean()
        std = account_data.std()

        #store the mean and standard deviation for each account
        stats[f'AccountID_ACC{i}'] = {'mean': mean, 'std': std}

    return stats

#call the function to calculate mean and standard deviation for AccountID_ACC1 to AccountID_ACC15
account_stats = calculate_stats(test_encoded_df)

#add mean and standard deviation columns to test_encoded_df
for account, values in account_stats.items():
    test_encoded_df[f'{account}_mean'] = values['mean']
    test_encoded_df[f'{account}_std'] = values['std']

#create a new column for deviation from mean for each transaction
test_encoded_df['Deviation_From_Mean'] = 0.0

```

```

#loop through each account and calculate the deviation
for i in range(1, 15):
    account_columns = f'AccountID_ACC{i}'
    mean_columns = f'AccountID_ACC{i}_mean'
    std_columns = f'AccountID_ACC{i}_std'

#calculate the deviation only for transactions in the current account
condition = test_encoded_df[account_columns] == 1

#calculate number of standard deviations of a transaction's Amount value from its Account's Amount mean
test_encoded_df.loc[condition, 'Deviation_From_Mean'] = (
    (test_encoded_df.loc[condition, 'Amount'] - test_encoded_df.loc[condition, mean_columns])
) / test_encoded_df.loc[condition, std_columns]

#create a list of prefixes for our one-hot columns
one_hot_prefixes = ['AccountID_', 'Merchant_', 'TransactionType_', 'Location_', 'Amount_Partitions_', 'Day']

#create a variable that stores the columns starting with the respective prefixes from our list
binary = test_encoded_df.columns.str.startswith(tuple(one_hot_prefixes))

#convert TRUE/FALSE entries to 1/0 entries with exception handling condition
if binary.any():
    try:
        #check data types of the selected columns
        for col in test_encoded_df.columns[binary]:
            #get the data type of the column
            column_dtype = test_encoded_df[col].dtype

            if column_dtype != 'int32':
                #convert directly to int if boolean
                if column_dtype == 'bool':
                    test_encoded_df[col] = test_encoded_df[col].astype(int)
                else:
                    #ensure int type
                    test_encoded_df[col] = test_encoded_df[col].astype('int32')
    except Exception as e:
        print(f"Potential incompatible dtype error during conversion: {e}")
    else:
        print("No one-hot encoded columns found with the specified prefixes.")

#create lists for AccountID, Merchant, TransactionType, Location, and Hour Groups
account_ids = [f'AccountID_ACC{i}' for i in range(1, 15)] # AccountID's 1 to 15
merchants = [f'Merchant_Merchant{chr(i)}' for i in range(ord('A'), ord('J') + 1)] # Merchants A to J
transaction_types = ['TransactionType_Purchase', 'TransactionType_Transfer', 'TransactionType_Withdrawal']
locations = ['Location_London', 'Location_Los Angeles', 'Location_New York', 'Location_San Francisco', 'Location_Tokyo']
hour_groups = [f'Hour_Group_{i}' for i in range(1, 15)]

def encode_columns(df, columns):
    encoder = LabelEncoder()
    encoded_columns = {}

    for column in columns:
        if column in df.columns:
            encoded_columns[column] = encoder.fit_transform(df[column])
        else:
            print(f"Warning: {column} not found in DataFrame.")
    return encoded_columns

#encode each variable
encoded_account_ids = encode_columns(test_encoded_df, account_ids)
encoded_merchants = encode_columns(test_encoded_df, merchants)
encoded_transaction_types = encode_columns(test_encoded_df, transaction_types)
encoded_locations = encode_columns(test_encoded_df, locations)
encoded_hour_groups = encode_columns(test_encoded_df, hour_groups)

#assign the encoded values back to the DataFrame
for account_id, encoded_values in encoded_account_ids.items():
    test_encoded_df[account_id] = encoded_values

for merchant, encoded_values in encoded_merchants.items():
    test_encoded_df[merchant] = encoded_values

for transaction_type, encoded_values in encoded_transaction_types.items():
    test_encoded_df[transaction_type] = encoded_values

for location, encoded_values in encoded_locations.items():
    test_encoded_df[location] = encoded_values

for hour_group, encoded_values in encoded_hour_groups.items():
    test_encoded_df[hour_group] = encoded_values

#prepare Inputs for Embedding
numerical_input = Input(shape=(1,), name='numerical_input')

```

```

account_input = Input(shape=(1,), name='account_input')
merchant_input = Input(shape=(1,), name='merchant_input')
transaction_input = Input(shape=(1,), name='transaction_input')
location_input = Input(shape=(1,), name='location_input')
hour_group_input = Input(shape=(1,), name='hour_group_input')

#create Embedding Layers
embedding_dim = 8
num_accounts = len(account_ids)
num_merchants = len(merchants)
num_transaction_types = len(transaction_types)
num_locations = len(locations)
num_hour_groups = len(hour_groups)

#embeddings for each one-hot encoded category
account_embedding = Embedding(input_dim=num_accounts, output_dim=embedding_dim)(account_input)
merchant_embedding = Embedding(input_dim=num_merchants, output_dim=embedding_dim)(merchant_input)
transaction_embedding = Embedding(input_dim=num_transaction_types, output_dim=embedding_dim)(transaction_input)
location_embedding = Embedding(input_dim=num_locations, output_dim=embedding_dim)(location_input)
hour_group_embedding = Embedding(input_dim=num_hour_groups, output_dim=embedding_dim)(hour_group_input)

#flatten the embeddings to make a one-dimensional array representation of the variables
flattened_account = Flatten()(account_embedding)
flattened_merchant = Flatten()(merchant_embedding)
flattened_transaction = Flatten()(transaction_embedding)
flattened_location = Flatten()(location_embedding)
flattened_hour_group = Flatten()(hour_group_embedding)

#concatenate inputs to a single output
concat = Concatenate()([numerical_input, flattened_account, flattened_merchant, flattened_transaction, flattened_location, flattened_hour_group])

#add Dense Layers to interconnect previous layers
output = Dense(1, activation='sigmoid')(concat)

#build the Model
model = Model(inputs=[numerical_input, account_input, merchant_input, transaction_input, location_input, hour_group_input], outputs=output)
model.compile(optimizer='adam', loss='mean_squared_error')

#prepare input data for model training
X_numerical = test_encoded_df[['Amount']].values
X_accounts = test_encoded_df[account_ids].values.argmax(axis=1).reshape(-1, 1) # Get index for account input
X_merchants = test_encoded_df[merchants].values.argmax(axis=1).reshape(-1, 1)
X_transaction_types = test_encoded_df[transaction_types].values.argmax(axis=1).reshape(-1, 1)
X_locations = test_encoded_df[locations].values.argmax(axis=1).reshape(-1, 1)
X_hour_groups = test_encoded_df[hour_groups].values.argmax(axis=1).reshape(-1, 1)

#make sure to pass the inputs as a list
model.fit([X_numerical, X_accounts, X_merchants, X_transaction_types, X_locations, X_hour_groups],
          np.zeros(X_numerical.shape[0]),
          epochs=1,
          batch_size=16)

#create a model to get the embedding outputs
embedding_model = Model(inputs=[account_input, merchant_input, transaction_input, location_input, hour_group_input], outputs=[account_embedding, merchant_embedding, transaction_embedding, location_embedding])

#get the embedding outputs
embedding_output = embedding_model.predict([X_accounts, X_merchants, X_transaction_types, X_locations, X_hour_groups])

#store the embeddings in a DataFrame
embedding_df = pd.DataFrame({
    'Account_EMBEDDINGS': list(embedding_output[0]),
    'Merchant_EMBEDDINGS': list(embedding_output[1]),
    'Transaction_EMBEDDINGS': list(embedding_output[2]),
    'Location_EMBEDDINGS': list(embedding_output[3]),
    'Hour_Group_EMBEDDINGS': list(embedding_output[4])
})

#combine with the original DataFrame
test_embeddings_df = pd.concat([test_encoded_df.reset_index(drop=True), embedding_df.reset_index(drop=True)], axis=1)

# List of columns to drop (already converted to one-hot/embeddings or not relevant to problem statement resolver)
columns_to_drop = [
    'Minute',
    'Month',
    'Hour',
    'Hour_Group'
] + [f'AccountID_ACC{i}' for i in range(1, 16)] + \
[f'Merchant_Merchant{chr(i)}' for i in range(ord('A'), ord('J') + 1)] + \
['TransactionType_Purchase', 'TransactionType_Transfer', 'TransactionType_Withdrawal'] + \
['Location_London', 'Location_Los Angeles', 'Location_New York',
 'Location_San Francisco', 'Location_Tokyo'] + \
[f'Amount_Partitions_{i}' for i in ['0-10000', '10001-20000', '20001-30000', '30001-40000', '40001-50000', '50001-60000', '60001-70000', '70001-80000', '80001-90000', '90001-100000']]

```

```

        '30001-40000', '40001-50000', '50001-60000',
        '60001-70000', '70001-80000', '80001-90000',
        '90001-100000', '100001+')]] + \
[f'AccountID_ACC{i}_mean' for i in range(1, 16)] + \
[f'AccountID_ACC{i}_std' for i in range(1, 16)] + \
[f'Hour_Group_{i}' for i in range(1, 15)]

# Drop the specified columns
test_embeddings_df = test_embeddings_df.drop(columns=columns_to_drop)

```

```

C:\Users\brady\OneDrive\Apps\Anaconda\Lib\site-packages\keras\src\models\functional.py:225: UserWarning: The structure of `inputs` doesn't match the expected structure: ['numerical_input', 'account_input', 'merchant_input', 'transaction_input', 'location_input', 'hour_group_input']. Received: the structure of inputs='*', '*', '*', '*', '*', '*')
    warnings.warn(
2034/2034 ━━━━━━━━ 4s 1ms/step - loss: 0.0432
 38/1017 ━━━━━━ 1s 1ms/step
C:\Users\brady\OneDrive\Apps\Anaconda\Lib\site-packages\keras\src\models\functional.py:225: UserWarning: The structure of `inputs` doesn't match the expected structure: ['account_input', 'merchant_input', 'transaction_input', 'location_input', 'hour_group_input']. Received: the structure of inputs='*', '*', '*', '*', '*', '*')
    warnings.warn(
1017/1017 ━━━━━━ 2s 2ms/step

```

In [74]: # Print a sample to see the current state of the DataFrame
train_embeddings_df.head()

	Timestamp	Amount	Date	Day_0	Day_1	Day_2	Day_3	Day_4	Day_5	Day_6	Deviation_From_Mean	Account_EMBEDDINGS
0	2023-01-07 17:50:00	9.064231	2023-01-07	0	0	0	0	0	1	0	-1.457638	[-0.24838004, -0.18754466, -0.27753416, 0.315...]
1	2023-01-30 08:04:00	10.757187	2023-01-30	1	0	0	0	0	0	0	0.248660	[-0.25014406, -0.19447853, -0.3003745, 0.2428...]
2	2023-04-06 03:13:00	10.996651	2023-04-06	0	0	0	1	0	0	0	0.498987	[-0.28238913, -0.16333285, -0.33246365, 0.274...]
3	2023-04-21 10:28:00	11.204528	2023-04-21	0	0	0	0	1	0	0	0.691171	[-0.24838004, -0.18754466, -0.27753416, 0.315...]
4	2023-01-08 05:29:00	9.295688	2023-01-08	0	0	0	0	0	0	1	-1.204878	[-0.41863093, -0.23580849, -0.4800192, 0.4766...]

In [75]: #create a features list for modeling purposes
features = [
 'Day_0', 'Day_1', 'Day_2', 'Day_3', 'Day_4', 'Day_5', 'Day_6',
 'Deviation_From_Mean', 'Account_EMBEDDINGS', 'Merchant_EMBEDDINGS',
 'Transaction_EMBEDDINGS', 'Location_EMBEDDINGS', 'Hour_Group_EMBEDDINGS'
]

#save feature data in variable X
X_train = train_embeddings_df[features]

#flatten embeddings to reduce dimensionality
for col in ['Account_EMBEDDINGS', 'Merchant_EMBEDDINGS', 'Transaction_EMBEDDINGS', 'Location_EMBEDDINGS', 'Hour_Group_EMBEDDINGS']:
 if isinstance(X_train[col].iloc[0], np.ndarray): #check if the first entry is an ndarray
 embedding_array = pd.DataFrame(X_train[col].apply(lambda x: x.flatten()).tolist())
 else:
 embedding_array = pd.DataFrame(X_train[col].tolist())
#concatenate newly flattened columns with DataFrame and drop the older, higher dimensional columns
embedding_array.columns = [f'{col}_{i}' for i in range(embedding_array.shape[1])]
X_train = pd.concat([X_train, embedding_array], axis=1)
X_train.drop(columns=[col], inplace=True)

#ensure all data is numeric
X_train = X_train.apply(pd.to_numeric, errors='coerce')

In [81]: #MODEL BUILDING (1)
#define performance metrics for models to be evaluated on
#function to calculate Dunn Index
def dunn_index(X, labels):
 unique_clusters = np.unique(labels)
 intra_distances = []
 inter_distances = []

 #calculate intra-cluster distances (furthest distance between two points within a cluster)
 for cluster in unique_clusters:
 points = X_train[labels == cluster]

```

    if len(points) > 1:
        intra_distances.append(np.max(cdist(points, points)))

#calculate inter-cluster distances (distance between respective clusters)
for i in range(len(unique_clusters)):
    for j in range(i + 1, len(unique_clusters)):
        points1 = X_train[labels == unique_clusters[i]]
        points2 = X_train[labels == unique_clusters[j]]
        inter_distances.append(np.min(cdist(points1, points2)))

return min(inter_distances) / max(intra_distances) if max(intra_distances) > 0 else 0

#HYPERPARAMETER TUNING
#best k value from previous evaluation (ideally this would have been done, but 5 was simply chosen due to the code)
best_k_value = 5

#vary init and n_init
init_values = ['random']
n_init_values = [1]

train_results_variation1 = []

for init in init_values:
    for n_init in n_init_values:
        #create and fit the KMeans model
        kmeans = KMeans(n_clusters=best_k_value, init=init, n_init=n_init, random_state=42)
        kmeans.fit(X_train)

        #get labels and inertia
        labels = kmeans.labels_
        inertia = kmeans.inertia_

        #calculate Silhouette Score
        silhouette_avg = silhouette_score(X_train, labels)

        #calculate Dunn Index
        dunn_idx = dunn_index(X_train, labels)

        #store results
        train_results_variation1.append({
            'Init': init,
            'n_init': n_init,
            'Silhouette Score': silhouette_avg,
            'Dunn Index': dunn_idx,
            'Inertia': inertia
        })

#convert results to DataFrame for better readability
train_results_variation1_df = pd.DataFrame(train_results_variation1)

print(train_results_variation1_df)

```

	Init	n_init	Silhouette Score	Dunn Index	Inertia
0	random	1	0.182405	0.000693	148629.322234

In [82]:

```

#model variation
best_init = 'k-means++'
best_n_init = 1
best_k_value = 5

#vary max_iter and tol
max_iter_values = [100]
tol_values = [1e-2]

train_results_variation2 = []

for max_iter in max_iter_values:
    for tol in tol_values:
        #create and fit the KMeans model
        kmeans = KMeans(n_clusters=best_k_value, init=best_init, n_init=best_n_init,
                        max_iter=max_iter, tol=tol, random_state=42)
        kmeans.fit(X_train)

        #get labels and inertia
        labels = kmeans.labels_
        inertia = kmeans.inertia_

        #calculate Silhouette Score
        silhouette_avg = silhouette_score(X_train, labels)

        #calculate Dunn Index
        distances = cdist(X_train, kmeans.cluster_centers_)
        intra_cluster_distances = np.min(distances, axis=1)
        inter_cluster_distances = np.max(distances)

```

```

dunn_index = np.min(intra_cluster_distances) / inter_cluster_distances

#store results
train_results_variation2.append({
    'max_iter': max_iter,
    'tol': tol,
    'Silhouette Score': silhouette_avg,
    'Dunn Index': dunn_index,
    'Inertia': inertia
})

#convert results to DataFrame for better readability
train_results_variation2_df = pd.DataFrame(train_results_variation2)

print(train_results_variation2_df)

#print train variation 2 results
train_results_variation2 = train_results_variation2_df.loc[train_results_variation2_df['Silhouette Score'].idxmax()]
print(train_results_variation2)

max_iter      tol   Silhouette Score  Dunn Index      Inertia
0            100  0.01           0.244653    0.01619  144921.487808
max_iter
tol
Silhouette Score
Dunn Index
Inertia
Name: 0, dtype: float64

```

```

In [83]: #model variation
best_init = 'k-means++'
best_n_init = 1
best_k_value = 10

#vary max_iter and tol
max_iter_values = [200]
tol_values = [1e-3]

train_results_variation3 = []

for max_iter in max_iter_values:
    for tol in tol_values:
        #create and fit the KMeans model
        kmeans = KMeans(n_clusters=best_k_value, init=best_init, n_init=best_n_init,
                         max_iter=max_iter, tol=tol, random_state=42)
        kmeans.fit(X_train)

        #get labels and inertia
        labels = kmeans.labels_
        inertia = kmeans.inertia_

        #calculate Silhouette Score
        silhouette_avg = silhouette_score(X_train, labels)

        #calculate Dunn Index
        distances = cdist(X_train, kmeans.cluster_centers_)
        intra_cluster_distances = np.min(distances, axis=1)
        inter_cluster_distances = np.max(distances)
        dunn_index = np.min(intra_cluster_distances) / inter_cluster_distances

        #store results
        train_results_variation3.append({
            'max_iter': max_iter,
            'tol': tol,
            'Silhouette Score': silhouette_avg,
            'Dunn Index': dunn_index,
            'Inertia': inertia
        })

#convert results to DataFrame for better readability
train_results_variation3_df = pd.DataFrame(train_results_variation3)

print(train_results_variation3_df)

#identify the best performing final variation
train_results_variation3 = train_results_variation3_df.loc[train_results_variation3_df['Silhouette Score'].idxmax()]
print(train_results_variation3)

```

```

max_iter      tol   Silhouette Score Dunn Index      Inertia
0           200  0.001          0.467028  0.015271  71258.191129
max_iter
tol
Silhouette Score          0.467028
Dunn Index                0.015271
Inertia                   71258.191129
Name: 0, dtype: float64

```

```

In [85]: #perform model on validation set
features = [
    'Day_0', 'Day_1', 'Day_2', 'Day_3', 'Day_4', 'Day_5', 'Day_6',
    'Deviation_From_Mean', 'Account_EMBEDDINGS', 'Merchant_EMBEDDINGS',
    'Transaction_EMBEDDINGS', 'Location_EMBEDDINGS', 'Hour_Group_EMBEDDINGS'
]

#prepare the feature data
X_val = validation_embeddings_df[features]

#flatten embeddings
for col in ['Account_EMBEDDINGS', 'Merchant_EMBEDDINGS', 'Transaction_EMBEDDINGS', 'Location_EMBEDDINGS', 'Hour_Group_EMBEDDINGS']:
    if isinstance(X_val[col].iloc[0], np.ndarray): # Check if the first entry is an ndarray
        embedding_array = pd.DataFrame(X_val[col].apply(lambda x: x.flatten()).tolist())
    else:
        embedding_array = pd.DataFrame(X_val[col].tolist())

    embedding_array.columns = [f"{col}_{i}" for i in range(embedding_array.shape[1])]
    X_val = pd.concat([X_val, embedding_array], axis=1)
    X_val.drop(columns=[col], inplace=True)

#ensure all data is numeric
X_val = X_val.apply(pd.to_numeric, errors='coerce')

```

```

In [89]: #function to calculate Dunn Index
def dunn_index(X, labels):
    unique_clusters = np.unique(labels)
    intra_distances = []
    inter_distances = []

    #calculate intra-cluster distances
    for cluster in unique_clusters:
        points = X[labels == cluster]
        if len(points) > 1:
            intra_distances.append(np.max(cdist(points, points)))

    #calculate inter-cluster distances
    for i in range(len(unique_clusters)):
        for j in range(i + 1, len(unique_clusters)):
            points1 = X[labels == unique_clusters[i]]
            points2 = X[labels == unique_clusters[j]]
            inter_distances.append(np.min(cdist(points1, points2)))

    return min(inter_distances) / max(intra_distances) if max(intra_distances) > 0 else 0

#best k value from previous evaluation
best_k_value = 5

```

```

#vary init and n_init
init_values = ['random']
n_init_values = [1]

val_results_variation1 = []

for init in init_values:
    for n_init in n_init_values:
        #create and fit the KMeans model
        kmeans = KMeans(n_clusters=best_k_value, init=init, n_init=n_init, random_state=42)
        kmeans.fit(X_train)

        #get labels and inertia
        labels = kmeans.predict(X_val)
        inertia = kmeans.inertia_

        #calculate Silhouette Score
        silhouette_avg = silhouette_score(X_val, labels)

        #calculate Dunn Index
        dunn_idx = dunn_index(X_val, labels)

        #store results
        val_results_variation1.append({
            'Init': init,
            'n_init': n_init,
            'Inertia': inertia,
            'Silhouette Score': silhouette_avg,
            'Dunn Index': dunn_idx
        })

```

```

        'Silhouette Score': silhouette_avg,
        'Dunn Index': dunn_idx,
        'Inertia': inertia
    })

#convert results to DataFrame for better readability
val_results_variation1_df = pd.DataFrame(val_results_variation1)

print(val_results_variation1_df)

   Init n_init Silhouette Score Dunn Index      Inertia
0  random      1       0.176601    0.001016  148629.322234

```

```
In [90]: #model variation
best_init = 'k-means++'
best_n_init = 1
best_k_value = 5

#vary max_iter and tol
max_iter_values = [100]
tol_values = [1e-2]

val_results_variation2 = []

for max_iter in max_iter_values:
    for tol in tol_values:
        #create and fit the KMeans model
        kmeans = KMeans(n_clusters=best_k_value, init=best_init, n_init=best_n_init,
                         max_iter=max_iter, tol=tol, random_state=42)
        kmeans.fit(X_train)

        #get labels and inertia
        labels = kmeans.predict(X_val)
        inertia = kmeans.inertia_

        #calculate Silhouette Score
        silhouette_avg = silhouette_score(X_val, labels)

        #calculate Dunn Index
        distances = cdist(X_val, kmeans.cluster_centers_)
        intra_cluster_distances = np.min(distances, axis=1)
        inter_cluster_distances = np.max(distances)
        dunn_index = np.min(intra_cluster_distances) / inter_cluster_distances

        #store results
        val_results_variation2.append({
            'max_iter': max_iter,
            'tol': tol,
            'Silhouette Score': silhouette_avg,
            'Dunn Index': dunn_index,
            'Inertia': inertia
        })

#convert results to DataFrame for better readability
val_results_variation2_df = pd.DataFrame(val_results_variation2)

print(val_results_variation2_df)

#identify the best performing final variation
val_results_variation2 = val_results_variation2_df.loc[val_results_variation2_df['Silhouette Score'].idxmax()]
print(val_results_variation2)

   max_iter  tol  Silhouette Score  Dunn Index      Inertia
0         100  0.01       0.245544    0.303867  144921.487808
max_iter
tol
Silhouette Score
Dunn Index
Inertia
Name: 0, dtype: float64
```

```
In [91]: #model variation
best_init = 'k-means++'
best_n_init = 1
best_k_value = 10

#vary max_iter and tol
max_iter_values = [200]
tol_values = [1e-3]

val_results_variation3 = []

for max_iter in max_iter_values:
    for tol in tol_values:
```

```

#Create and fit the KMeans model
kmeans = KMeans(n_clusters=best_k_value, init=best_init, n_init=best_n_init,
                 max_iter=max_iter, tol=tol, random_state=42)
kmeans.fit(X_train)

#get labels and inertia
labels = kmeans.predict(X_val)
inertia = kmeans.inertia_

#calculate Silhouette Score
silhouette_avg = silhouette_score(X_val, labels)

#calculate Dunn Index
distances = cdist(X_val, kmeans.cluster_centers_)
intra_cluster_distances = np.min(distances, axis=1)
inter_cluster_distances = np.max(distances)
dunn_index = np.min(intra_cluster_distances) / inter_cluster_distances

#store results
val_results_variation3.append({
    'max_iter': max_iter,
    'tol': tol,
    'Silhouette Score': silhouette_avg,
    'Dunn Index': dunn_index,
    'Inertia': inertia
})

#convert results to DataFrame for better readability
val_results_variation3_df = pd.DataFrame(val_results_variation3)

print(val_results_variation3_df)

#identify the best performing final variation
val_results_variation3 = val_results_variation3_df.loc[val_results_variation3_df['Silhouette Score'].idxmax()]
print(val_results_variation3)

```

	max_iter	tol	Silhouette Score	Dunn Index	Inertia
0	200	0.001	0.468116	0.298864	71258.191129
max_iter			200.000000		
tol			0.001000		
Silhouette Score			0.468116		
Dunn Index			0.298864		
Inertia			71258.191129		

Name: 0, dtype: float64

```

In [92]: #normalize inertia scores with respect to dataset sizes
train_inertia_1 = 147040.18201/151872
train_inertia_2 = 143682.817524/151872
train_inertia_3 = 69707.25485/151872
print(train_inertia_1, train_inertia_2, train_inertia_3)

```

0.9681849321138853 0.9460783918299621 0.4589868761193637

```

In [93]: val_inertia_1 = 31632.758442/32544
val_inertia_2 = 31714.576743/32544
val_inertia_3 = 16051.701494/32544
print(val_inertia_1, val_inertia_2, val_inertia_3)

```

0.9719997063053097 0.9745137888089971 0.4932307489552606

```

In [94]: #PERFORMANCE EVALUATION
data1 = {
    'Variation 1 Silhouette': [0.1814,  0.2434],
    'Variation 1 Dunn Index': [0.0007,  0.0023],
    'Variation 1 Inertia': [0.9682,  0.9720]
}

data2 = {
    'Variation 2 Silhouette': [0.2420,  0.2438],
    'Variation 2 Dunn Index': [0.01740, 0.0190],
    'Variation 2 Inertia': [0.9461,  0.9745]
}

data3 = {
    'Variation 3 Silhouette': [0.4815,  0.4629],
    'Variation 3 Dunn Index': [0.0176,  0.0177],
    'Variation 3 Inertia': [0.4590,  0.4932]
}
#create a DataFrame
results1_df = pd.DataFrame(data1, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results1_df, headers='keys', tablefmt='pretty'))

#create a DataFrame

```

```

results2_df = pd.DataFrame(data2, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results2_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results3_df = pd.DataFrame(data3, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results3_df, headers='keys', tablefmt='pretty'))

```

	Variation 1 Silhouette	Variation 1 Dunn Index	Variation 1 Inertia	
Training	0.1814	0.0007	0.9682	
Validation	0.2434	0.0023	0.972	
	Variation 2 Silhouette	Variation 2 Dunn Index	Variation 2 Inertia	
Training	0.242	0.0174	0.9461	
Validation	0.2438	0.019	0.9745	
	Variation 3 Silhouette	Variation 3 Dunn Index	Variation 3 Inertia	
Training	0.4815	0.0176	0.459	
Validation	0.4629	0.0177	0.4932	

```

In [95]: #initialize the LOF model
n_neighbors = 2
lof = LocalOutlierFactor(n_neighbors=n_neighbors)

#fit the model and get the labels for training data
train_labels = lof.fit_predict(X_train)
#LOF returns -1 for outliers and 1 for inliers
train_labels[train_labels == 1] = 0 # Convert inliers to 0
train_labels[train_labels == -1] = 1 # Convert outliers to 1

#fit the model and get the labels for validation data
val_labels = lof.fit_predict(X_val)
val_labels[val_labels == 1] = 0
val_labels[val_labels == -1] = 1

#calculate metrics for training data
silhouette_train = silhouette_score(X_train, train_labels)
calinski_train = calinski_harabasz_score(X_train, train_labels)
davies_train = davies_bouldin_score(X_train, train_labels)

#calculate metrics for validation data
silhouette_val = silhouette_score(X_val, val_labels)
calinski_val = calinski_harabasz_score(X_val, val_labels)
davies_val = davies_bouldin_score(X_val, val_labels)

#print results
print("Training Data Metrics:")
print(f"Silhouette Score: {silhouette_train:.4f}")
print(f"Calinski-Harabasz Index: {calinski_train:.4f}")
print(f"Davies-Bouldin Index: {davies_train:.4f}")

print("\nValidation Data Metrics:")
print(f"Silhouette Score: {silhouette_val:.4f}")
print(f"Calinski-Harabasz Index: {calinski_val:.4f}")
print(f"Davies-Bouldin Index: {davies_val:.4f}")

```

Training Data Metrics:
Silhouette Score: -0.0500
Calinski-Harabasz Index: 636.6217
Davies-Bouldin Index: 5.2383

Validation Data Metrics:
Silhouette Score: -0.0194
Calinski-Harabasz Index: 9.0257
Davies-Bouldin Index: 15.6266

```

In [96]: #MODEL BUILDING (2)
#HYPERPARAMETER TUNING
#n_neighbors for varying method of evaluating local density (5 nearest points vs 10 nearest points vs 15 nearest points)
n_neighbors_list = [5, 10, 15] # Different values for n_neighbors
#leaf_size varies search speed (30 is default, 50 is faster)
leaf_size_list = [30, 50] # Different leaf sizes
#metric considers different scoring methods that yield results tailored towards certain statistics
metric_list = ['euclidean', 'manhattan']

```

```

#store results
results = []

for n_neighbors in n_neighbors_list:
    for leaf_size in leaf_size_list:
        for metric in metric_list:
            #initialize the LOF model
            lof = LocalOutlierFactor(n_neighbors=n_neighbors,
                                     leaf_size=leaf_size,
                                     metric=metric)

            #fit the model and get the labels for training data
            train_labels = lof.fit_predict(X_train)
            train_labels[train_labels == 1] = 0 # Convert inliers to 0
            train_labels[train_labels == -1] = 1 # Convert outliers to 1

            #fit the model and get the labels for validation data
            val_labels = lof.fit_predict(X_val)
            val_labels[val_labels == 1] = 0
            val_labels[val_labels == -1] = 1

            #calculate metrics for training data
            silhouette_train = silhouette_score(X_train, train_labels)
            calinski_train = calinski_harabasz_score(X_train, train_labels)
            davies_train = davies_bouldin_score(X_train, train_labels)

            #calculate metrics for validation data
            silhouette_val = silhouette_score(X_val, val_labels)
            calinski_val = calinski_harabasz_score(X_val, val_labels)
            davies_val = davies_bouldin_score(X_val, val_labels)

            #store results
            results.append({
                'n_neighbors': n_neighbors,
                'leaf_size': leaf_size,
                'metric': ['euclidean', 'manhattan'],
                'Silhouette Score (Train)': silhouette_train,
                'Calinski-Harabasz Index (Train)': calinski_train,
                'Davies-Bouldin Index (Train)': davies_train,
                'Silhouette Score (Val)': silhouette_val,
                'Calinski-Harabasz Index (Val)': calinski_val,
                'Davies-Bouldin Index (Val)': davies_val
            })
}

#convert results to DataFrame for better readability
results_df = pd.DataFrame(results)

#print results
print(results_df)

```

n_neighbors	leaf_size	metric	Silhouette Score (Train)	\
0	5	[euclidean, manhattan]	0.116275	
1	5	[euclidean, manhattan]	-0.012691	
2	5	[euclidean, manhattan]	0.116275	
3	5	[euclidean, manhattan]	-0.012691	
4	10	[euclidean, manhattan]	-0.031291	
5	10	[euclidean, manhattan]	-0.065154	
6	10	[euclidean, manhattan]	-0.031291	
7	10	[euclidean, manhattan]	-0.065154	
8	15	[euclidean, manhattan]	0.119812	
9	15	[euclidean, manhattan]	0.050357	
10	15	[euclidean, manhattan]	0.119812	
11	15	[euclidean, manhattan]	0.050357	
Calinski-Harabasz Index (Train) Davies-Bouldin Index (Train) \				
0	102.761748	4.452429		
1	29.013510	9.622174		
2	102.761748	4.452429		
3	29.013510	9.622174		
4	207.861969	5.775110		
5	192.985984	5.968536		
6	207.861969	5.775110		
7	192.985984	5.968536		
8	177.411221	4.019922		
9	100.570734	5.318967		
10	177.411221	4.019922		
11	100.570734	5.318967		
Silhouette Score (Val) Calinski-Harabasz Index (Val) \				
0	0.225661	69.635594		
1	0.129040	17.486821		
2	0.225661	69.635594		
3	0.129040	17.486821		
4	0.291592	69.548741		
5	0.244692	28.738124		
6	0.291592	69.548741		
7	0.244692	28.738124		
8	0.293352	75.984432		
9	0.248833	34.154034		
10	0.293352	75.984432		
11	0.248833	34.154034		
Davies-Bouldin Index (Val)				
0	3.934102			
1	6.476632			
2	3.934102			
3	6.476632			
4	3.838041			
5	5.108599			
6	3.838041			
7	5.108599			
8	3.703373			
9	4.970628			
10	3.703373			
11	4.970628			

```
In [97]: #PERFORMANCE EVALUATION
#n_neighbors=5; euclidean
data1 = {
    'Variation 1.1 Silhouette': [0.0996, 0.2972],
    'Variation 1.1 Calinski-Harabasz Index': [88.1241, 64.4077],
    'Variation 1.1 Davies-Bouldin Index': [4.7576, 3.6680]
}
#n_neighbors=10; euclidean
data2 = {
    'Variation 2.1 Silhouette': [-0.0267, 0.4804],
    'Variation 2.1 Calinski-Harabasz Index': [198.7778, 143.9189],
    'Variation 2.1 Davies-Bouldin Index': [5.5308, 2.0772]
}
#n_neighbors=15; euclidean
data3 = {
    'Variation 3.1 Silhouette': [0.0713, 0.5046],
    'Variation 3.1 Calinski-Harabasz Index': [162.3979, 194.8525],
    'Variation 3.1 Davies-Bouldin Index': [4.5217, 1.6319]
}
#n_neighbors=5; manhattan
data4 = {
    'Variation 1.2 Silhouette': [-0.0140, 0.0949],
    'Variation 1.2 Calinski-Harabasz Index': [30.7389, 5.5474],
    'Variation 1.2 Davies-Bouldin Index': [9.3945, 10.6232]
}
#n_neighbors=10; manhattan
data5 = {
```

```

'Variation 2.2 Silhouette': [-0.0595,  0.3006],
'Variation 2.2 Calinski-Harabasz Index': [182.6127,  21.2976],
'Variation 2.2 Davies-Bouldin Index': [5.6467,  5.0632]
}

#n_neighbors=15; manhattan
data6 = {
    'Variation 3.2 Silhouette': [0.0234,  0.3937],
    'Variation 3.2 Calinski-Harabasz Index': [113.4173,  51.1523],
    'Variation 3.2 Davies-Bouldin Index': [5.4388,  3.4516]
}
#create a DataFrame
results1_df = pd.DataFrame(data1, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results1_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results2_df = pd.DataFrame(data2, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results2_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results3_df = pd.DataFrame(data3, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results3_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results4_df = pd.DataFrame(data4, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results4_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results5_df = pd.DataFrame(data5, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results5_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results6_df = pd.DataFrame(data6, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results6_df, headers='keys', tablefmt='pretty'))

```

Variation 1 Silhouette Variation 1 Calinski-Harabasz Index Variation 1 Davies-Bouldin Index				
	Training	Validation		
	0.0996	0.2972	88.1241	4.7576
			64.4077	3.668
Variation 2 Silhouette Variation 2 Calinski-Harabasz Index Variation 2 Davies-Bouldin Index				
	Training	Validation		
	-0.0267	0.4804	198.7778	5.5308
			143.9189	2.0772
Variation 3 Silhouette Variation 3 Calinski-Harabasz Index Variation 3 Davies-Bouldin Index				
	Training	Validation		
	0.0713	0.5046	162.3979	4.5217
			194.8525	1.6319
Variation 1.2 Silhouette Variation 1.2 Calinski-Harabasz Index Variation 1.2 Davies-Bouldin Index				
	Training	Validation		
	-0.014	0.0949	30.7389	9.3945
			5.5474	10.6232
Variation 2.2 Silhouette Variation 2.2 Calinski-Harabasz Index Variation 2.2 Davies-Bouldin Index				
	Training	Validation		
	-0.0595	0.3006	182.6127	5.6467
			21.2976	5.0632
Variation 3.2 Silhouette Variation 3.2 Calinski-Harabasz Index Variation 3.2 Davies-Bouldin Index				
	Training	Validation		
	0.0234	0.3937	113.4173	5.4388
			51.1523	3.4516

```
In [98]: #MODEL BUILDING (3)
#HYPERPARAMETER TUNING
eps_list = [0.3, 0.5, 0.7] #radius of neighborhood - determines maximum distance allowed between points for the
min_samples_list = [3, 5, 10] #minimum number of samples in a neighborhood to consider a point as a core point
```

```
In [99]: #store results
results = []
```

```
In [100]: # Create for loops to iterate through hyperparameter tuning combinations
for eps in eps_list:
```

```

for min_samples in min_samples_list:
    #initialize the DBSCAN model
    dbscan = DBSCAN(eps=eps, min_samples=min_samples)

    #fit the model and get the labels for training data - determines anomaly/normal classification based on
    train_labels = dbscan.fit_predict(X_train)
    #convert core points and noise points
    train_labels[train_labels == -1] = 1 #convert noise to 1 (anomalies)
    train_labels[train_labels != 1] = 0 #convert core points to 0 (normal)

    #fit the model and get the labels for validation data - determines anomaly/normal classification based on
    val_labels = dbscan.fit_predict(X_val)
    val_labels[val_labels == -1] = 1 #convert noise to 1 (anomalies)
    val_labels[val_labels != 1] = 0 #convert core points to 0 (normal)

    #calculate metrics for training data (higher silhouette score = better clustering = better anomaly detection;
    #higher calinski-harabasz score = better clustering = better anomaly detection;
    #lower davies-bouldin score = better clustering = better anomaly detection)

    silhouette_train = silhouette_score(X_train, train_labels)
    calinski_train = calinski_harabasz_score(X_train, train_labels)
    davies_train = davies_bouldin_score(X_train, train_labels)

    #calculate metrics for validation data
    silhouette_val = silhouette_score(X_val, val_labels)
    calinski_val = calinski_harabasz_score(X_val, val_labels)
    davies_val = davies_bouldin_score(X_val, val_labels)

    #store results
    results.append({
        'eps': eps,
        'min_samples': min_samples,
        'Silhouette Score (Train)': silhouette_train,
        'Calinski-Harabasz Index (Train)': calinski_train,
        'Davies-Bouldin Index (Train)': davies_train,
        'Silhouette Score (Val)': silhouette_val,
        'Calinski-Harabasz Index (Val)': calinski_val,
        'Davies-Bouldin Index (Val)': davies_val
    })
}

```

In [101]:

```

#convert results to DataFrame for better readability
results_df = pd.DataFrame(results)

#print results
print(results_df)

```

	eps	min_samples	Silhouette Score (Train)	\
0	0.3	3	0.109069	
1	0.3	5	0.113414	
2	0.3	10	0.123694	
3	0.5	3	0.110345	
4	0.5	5	0.113122	
5	0.5	10	0.115567	
6	0.7	3	0.108457	
7	0.7	5	0.110170	
8	0.7	10	0.112053	

	Calinski-Harabasz Index (Train)	Davies-Bouldin Index (Train)	\
0	9974.634194	1.963102	
1	9952.648733	1.985104	
2	9849.781235	2.050437	
3	11062.648456	1.897563	
4	11055.103869	1.907527	
5	11041.002465	1.917789	
6	11074.062950	1.890064	
7	11064.065930	1.896591	
8	11061.745184	1.902983	

	Silhouette Score (Val)	Calinski-Harabasz Index (Val)	\
0	0.125915	2268.964336	
1	0.144931	2245.200251	
2	0.156916	2235.163134	
3	0.118051	2295.350872	
4	0.128301	2286.300183	
5	0.144029	2271.065544	
6	0.113587	2299.182510	
7	0.121044	2292.438916	
8	0.128391	2289.868988	

	Davies-Bouldin Index (Val)
0	1.991290
1	2.115797
2	2.237998
3	1.931304
4	1.976896
5	2.069323
6	1.913498
7	1.942961
8	1.973624

```
In [102]: #PERFORMANCE EVALUATION
#eps=0.3; min_samples=3
data1 = {
    'Variation 1.1 Silhouette': [0.1194, 0.1283],
    'Variation 1.1 Calinski-Harabasz Index': [10971.9597, 2266.2809],
    'Variation 1.1 Davies-Bouldin Index': [1.9702, 2.0129]
}
#eps=0.3; min_samples=5
data2 = {
    'Variation 1.2 Silhouette': [0.1319, 0.1464],
    'Variation 1.2 Calinski-Harabasz Index': [10957.8269, 2253.4459],
    'Variation 1.2 Davies-Bouldin Index': [2.0250, 2.1316]
}
#eps=0.3; min_samples=10
data3 = {
    'Variation 1.3 Silhouette': [0.1484, 0.1604],
    'Variation 1.3 Calinski-Harabasz Index': [10993.0411, 2254.2103],
    'Variation 1.3 Davies-Bouldin Index': [2.1291, 2.2791]
}
#eps=0.5; min_samples=3
data4 = {
    'Variation 2.1 Silhouette': [0.1111, 0.1190],
    'Variation 2.1 Calinski-Harabasz Index': [11105.0407, 2304.5240],
    'Variation 2.1 Davies-Bouldin Index': [1.9193, 1.9360]
}
#eps=0.5; min_samples=5
data5 = {
    'Variation 2.2 Silhouette': [0.1137, 0.1284],
    'Variation 2.2 Calinski-Harabasz Index': [11098.1885, 2298.5742],
    'Variation 2.2 Davies-Bouldin Index': [1.9289, 1.9756]
}
#eps=0.5; min_samples=10
data6 = {
    'Variation 2.3 Silhouette': [0.1170, 0.1438],
    'Variation 2.3 Calinski-Harabasz Index': [11080.3622, 2285.6402],
    'Variation 2.3 Davies-Bouldin Index': [1.9422, 2.0631]
}
#eps=0.7; min_samples=3
data7 = {
```

```

'Variation 3.1 Silhouette': [0.1250,  0.1138],
'Variation 3.1 Calinski-Harabasz Index': [12604.1641,  2310.2167],
'Variation 3.1 Davies-Bouldin Index': [1.9145,  1.9141]
}

#eps=0.7; min_samples=5
data8 = {
    'Variation 3.2 Silhouette': [0.1257,  0.1208],
    'Variation 3.2 Calinski-Harabasz Index': [12600.0272,  2303.5582],
    'Variation 3.2 Davies-Bouldin Index': [1.9170,  1.9419]
}
#eps=0.7; min_samples=10
data9 = {
    'Variation 3.3 Silhouette': [0.1270,  0.1295],
    'Variation 3.3 Calinski-Harabasz Index': [12592.7226,  2299.7924],
    'Variation 3.3 Davies-Bouldin Index': [1.9216,  1.9782]
}

#create a DataFrame
results1_df = pd.DataFrame(data1, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results1_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results2_df = pd.DataFrame(data2, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results2_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results3_df = pd.DataFrame(data3, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results3_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results4_df = pd.DataFrame(data4, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results4_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results5_df = pd.DataFrame(data5, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results5_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results6_df = pd.DataFrame(data6, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results6_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results7_df = pd.DataFrame(data7, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results7_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results8_df = pd.DataFrame(data8, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results8_df, headers='keys', tablefmt='pretty'))

#create a DataFrame
results9_df = pd.DataFrame(data9, index=['Training', 'Validation'])

#print the DataFrame using tabulate for better formatting
print(tabulate(results9_df, headers='keys', tablefmt='pretty'))

```

	Variation 1.1 Silhouette	Variation 1.1 Calinski-Harabasz Index	Variation 1.1 Davies-Bouldin Index
	Training	Validation	
Training	0.1194	10971.9597	1.9702
Validation	0.1283	2266.2809	2.0129

Variation 1.2 Silhouette Variation 1.2 Calinski-Harabasz Index Variation 1.2 Davies-Bouldin Index				
Training	0.1319		10957.8269	2.025
Validation	0.1464		2253.4459	2.1316
Variation 1.3 Silhouette Variation 1.3 Calinski-Harabasz Index Variation 1.3 Davies-Bouldin Index				
Training	0.1484		10993.0411	2.1291
Validation	0.1604		2254.2103	2.2791
Variation 2.1 Silhouette Variation 2.1 Calinski-Harabasz Index Variation 2.1 Davies-Bouldin Index				
Training	0.1111		11105.0407	1.9193
Validation	0.119		2304.524	1.936
Variation 2.2 Silhouette Variation 2.2 Calinski-Harabasz Index Variation 2.2 Davies-Bouldin Index				
Training	0.1137		11098.1885	1.9289
Validation	0.1284		2298.5742	1.9756
Variation 2.3 Silhouette Variation 2.3 Calinski-Harabasz Index Variation 2.3 Davies-Bouldin Index				
Training	0.117		11080.3622	1.9422
Validation	0.1438		2285.6402	2.0631
Variation 3.1 Silhouette Variation 3.1 Calinski-Harabasz Index Variation 3.1 Davies-Bouldin Index				
Training	0.125		12604.1641	1.9145
Validation	0.1138		2310.2167	1.9141
Variation 3.2 Silhouette Variation 3.2 Calinski-Harabasz Index Variation 3.2 Davies-Bouldin Index				
Training	0.1257		12600.0272	1.917
Validation	0.1208		2303.5582	1.9419

Variation 3.3 Silhouette Variation 3.3 Calinski-Harabasz Index Variation 3.3 Davies-Bouldin Index					
Training	0.127		12592.7226		1.9216
Validation	0.1295		2299.7924		1.9782

```
In [104]: #deploy all 9 models (the top 3 performing variations from each of model type) on the validation set
#K-Means Models on Validation Set

#initialize results storage
km_val_results = []

#variation km_v1: k=5, init='random', n_init=1
k_value_v1 = 5
init_v1 = 'random'
n_init_v1 = 1

#fit the model for Variation km_v1 using validation data
kmeans_v1 = KMeans(n_clusters=k_value_v1, init=init_v1, n_init=n_init_v1, random_state=42)
kmeans_v1.fit(X_train)
labels_v1 = kmeans_v1.predict(X_val)
silhouette_avg_v1 = silhouette_score(X_val, labels_v1)
km_val_results.append({'Variation': 'km_v1', 'Init': init_v1, 'n_init': n_init_v1, 'k': k_value_v1, 'Silhouette': silhouette_avg_v1})

#variation km_v2: k=5, init='k-means++', n_init=1, max_iter=100, tol=1e-2
k_value_v2 = 5
init_v2 = 'k-means++'
n_init_v2 = 1
max_iter_v2 = 100
tol_v2 = 1e-2

#fit the model for Variation km_v2 using validation data
kmeans_v2 = KMeans(n_clusters=k_value_v2, init=init_v2, n_init=n_init_v2, max_iter=max_iter_v2, tol=tol_v2, random_state=42)
kmeans_v2.fit(X_train)
labels_v2 = kmeans_v2.predict(X_val)
silhouette_avg_v2 = silhouette_score(X_val, labels_v2)
km_val_results.append({'Variation': 'km_v2', 'Init': init_v2, 'n_init': n_init_v2, 'k': k_value_v2, 'max_iter': max_iter_v2, 'tol': tol_v2, 'Silhouette': silhouette_avg_v2})

#variation km_v3: k=10, init='k-means++', n_init=1, max_iter=200, tol=1e-3 (Ultimately the winning model with highest Silhouette Score)
k_value_v3 = 10
init_v3 = 'k-means++'
n_init_v3 = 1
max_iter_v3 = 200
tol_v3 = 1e-3

#fit the model for Variation km_v3 using validation data
kmeans_v3 = KMeans(n_clusters=k_value_v3, init=init_v3, n_init=n_init_v3, max_iter=max_iter_v3, tol=tol_v3, random_state=42)
kmeans_v3.fit(X_train)
labels_v3 = kmeans_v3.predict(X_val)
silhouette_avg_v3 = silhouette_score(X_val, labels_v3)
km_val_results.append({'Variation': 'km_v3', 'Init': init_v3, 'n_init': n_init_v3, 'k': k_value_v3, 'max_iter': max_iter_v3, 'tol': tol_v3, 'Silhouette': silhouette_avg_v3})

#convert results to DataFrame for better readability
km_val_results_df = pd.DataFrame(km_val_results)

#print results
print(km_val_results_df)
```

Variation	Init	n_init	k	Silhouette Score	max_iter	tol
0 km_v1	random	1	5	0.176601	NaN	NaN
1 km_v2	k-means++	1	5	0.245544	100.0	0.010
2 km_v3	k-means++	1	10	0.468116	200.0	0.001

```
In [105]: #Local Outlier Factor Models on Validation Set

#initialize results storage
lof_val_results = []

#Variation lof_v1: n_neighbors=15, metric='euclidean'
n_neighbors_v1 = 15
metric_v1 = 'euclidean'

#Fit the model for Variation lof_v1 using validation data
lof_v1 = LocalOutlierFactor(n_neighbors=n_neighbors_v1, metric=metric_v1)
val_labels_v1 = lof_v1.fit_predict(X_val)

silhouette_val_v1 = silhouette_score(X_val, val_labels_v1)
lof_val_results.append({'Variation': 'lof_v1', 'n_neighbors': n_neighbors_v1, 'metric': metric_v1, 'Silhouette': silhouette_val_v1})
```

```

#variation lof_v2: n_neighbors=10, metric='euclidean'
n_neighbors_v2 = 10
metric_v2 = 'euclidean'

#fit the model for Variation lof_v2 using validation data
lof_v2 = LocalOutlierFactor(n_neighbors=n_neighbors_v2, metric=metric_v2)
val_labels_v2 = lof_v2.fit_predict(X_val)

silhouette_val_v2 = silhouette_score(X_val, val_labels_v2)
lof_val_results.append({'Variation': 'lof_v2', 'n_neighbors': n_neighbors_v2, 'metric': metric_v2, 'Silhouette Score': silhouette_val_v2})

#variation lof_v3: n_neighbors=15, metric='manhattan'
n_neighbors_v3 = 15
metric_v3 = 'manhattan'

#fit the model for Variation lof_v3 using validation data
lof_v3 = LocalOutlierFactor(n_neighbors=n_neighbors_v3, metric=metric_v3)
val_labels_v3 = lof_v3.fit_predict(X_val)

silhouette_val_v3 = silhouette_score(X_val, val_labels_v3)
lof_val_results.append({'Variation': 'lof_v3', 'n_neighbors': n_neighbors_v3, 'metric': metric_v3, 'Silhouette Score': silhouette_val_v3})

#convert results to DataFrame for better readability
lof_val_results_df = pd.DataFrame(lof_val_results)

#print results
print(lof_val_results_df)

```

Variation	n_neighbors	metric	Silhouette Score (Val)	
0	lof_v1	15	euclidean	0.293352
1	lof_v2	10	euclidean	0.291592
2	lof_v3	15	manhattan	0.248833

```

In [106]: #Density-Based Spatial Clustering of Applications with Noise Models on Validation Set
# Initialize results storage
dbSCAN_val_results = []

#variation db_v1: eps=0.7, min_samples=3
eps_v1 = 0.7
min_samples_v1 = 3

#fit the model for Variation db_v1 using validation data
dbSCAN_v1 = DBSCAN(eps=eps_v1, min_samples=min_samples_v1)
val_labels_v1 = dbSCAN_v1.fit_predict(X_val)
val_labels_v1[val_labels_v1 == -1] = 1 # Convert noise to 1 (anomalies)
val_labels_v1[val_labels_v1 != 1] = 0 # Convert core points to 0 (normal)

#calculate silhouette score for validation data
silhouette_val_v1 = silhouette_score(X_val, val_labels_v1)
dbSCAN_val_results.append({'Variation': 'db_v1', 'eps': eps_v1, 'min_samples': min_samples_v1, 'Silhouette Score': silhouette_val_v1})

#variation db_v2: eps=0.3, min_samples=10
eps_v2 = 0.3
min_samples_v2 = 10

#fit the model for Variation db_v2 using validation data
dbSCAN_v2 = DBSCAN(eps=eps_v2, min_samples=min_samples_v2)
val_labels_v2 = dbSCAN_v2.fit_predict(X_val)

#calculate silhouette score for validation data
silhouette_val_v2 = silhouette_score(X_val, val_labels_v2)
dbSCAN_val_results.append({'Variation': 'db_v2', 'eps': eps_v2, 'min_samples': min_samples_v2, 'Silhouette Score': silhouette_val_v2})

#variation db_v3: eps=0.3, min_samples=5
eps_v3 = 0.3
min_samples_v3 = 5

#fit the model for Variation db_v3 using validation data
dbSCAN_v3 = DBSCAN(eps=eps_v3, min_samples=min_samples_v3)
val_labels_v3 = dbSCAN_v3.fit_predict(X_val)
val_labels_v3[val_labels_v3 == -1] = 1 # Convert noise to 1 (anomalies)
val_labels_v3[val_labels_v3 != 1] = 0 # Convert core points to 0 (normal)

#calculate silhouette score for validation data
silhouette_val_v3 = silhouette_score(X_val, val_labels_v3)
dbSCAN_val_results.append({'Variation': 'db_v3', 'eps': eps_v3, 'min_samples': min_samples_v3, 'Silhouette Score': silhouette_val_v3})

#convert results to DataFrame for better readability
dbSCAN_val_results_df = pd.DataFrame(dbSCAN_val_results)

#print results
print(dbSCAN_val_results_df)

```

Variation	eps	min_samples	Silhouette Score (Val)
0	db_v1	0.7	0.113587
1	db_v2	0.3	0.241693
2	db_v3	0.3	0.144931

```
In [107]: #perform winning model on test set
features = [
    'Day_0', 'Day_1', 'Day_2', 'Day_3', 'Day_4', 'Day_5', 'Day_6',
    'Deviation_From_Mean', 'Account_EMBEDDINGS', 'Merchant_EMBEDDINGS',
    'Transaction_EMBEDDINGS', 'Location_EMBEDDINGS', 'Hour_Group_EMBEDDINGS'
]

#prepare the feature data
X_test = test_embeddings_df[features]

#flatten embeddings
for col in ['Account_EMBEDDINGS', 'Merchant_EMBEDDINGS', 'Transaction_EMBEDDINGS', 'Location_EMBEDDINGS', 'Hour_Group_EMBEDDINGS']:
    if isinstance(X_test[col].iloc[0], np.ndarray): # Check if the first entry is an ndarray
        embedding_array = pd.DataFrame(X_test[col].apply(lambda x: x.flatten()).tolist())
    else:
        embedding_array = pd.DataFrame(X_test[col].tolist())

    embedding_array.columns = [f'{col}_{i}' for i in range(embedding_array.shape[1])]
X_test = pd.concat([X_test, embedding_array], axis=1)
X_test.drop(columns=[col], inplace=True)

#ensure all data is numeric
X_test = X_test.apply(pd.to_numeric, errors='coerce')
```

```
In [109]: #variation km_v3: k=10, init='k-means++', n_init=1, max_iter=200, tol=1e-3
k_value_v3 = 10
init_v3 = 'k-means++'
n_init_v3 = 1
max_iter_v3 = 200
tol_v3 = 1e-3

#fit the model for Variation km_v3
kmeans_v3 = KMeans(n_clusters=k_value_v3, init=init_v3, n_init=n_init_v3, max_iter=max_iter_v3, tol=tol_v3, random_state=42)
kmeans_v3.fit(X_train)
labels_v3 = kmeans_v3.predict(X_test)
silhouette_avg_v3 = silhouette_score(X_test, labels_v3)

#store results
km_v3_test_results = {
    'Variation': 'km_v3',
    'Init': init_v3,
    'n_init': n_init_v3,
    'k': k_value_v3,
    'max_iter': max_iter_v3,
    'tol': tol_v3,
    'Silhouette Score': silhouette_avg_v3
}

#print results
print(km_v3_test_results)

{'Variation': 'km_v3', 'Init': 'k-means++', 'n_init': 1, 'k': 10, 'max_iter': 200, 'tol': 0.001, 'Silhouette Score': 0.47121813997971046}
```

```
In [110]: #Winning Model Train Performance
#variation km_v3: k=10, init='k-means++', n_init=1, max_iter=200, tol=1e-3
k_value_v3 = 10
init_v3 = 'k-means++'
n_init_v3 = 1
max_iter_v3 = 200
tol_v3 = 1e-3

#fit the model for Variation km_v3 using
kmeans_v3 = KMeans(n_clusters=k_value_v3, init=init_v3, n_init=n_init_v3, max_iter=max_iter_v3, tol=tol_v3, random_state=42)
kmeans_v3.fit(X_train)
labels_v3 = kmeans_v3.predict(X_train)
silhouette_avg_v3 = silhouette_score(X_train, labels_v3)

#store results
km_v3_train_results = {
    'Variation': 'km_v3',
    'Init': init_v3,
    'n_init': n_init_v3,
    'k': k_value_v3,
    'max_iter': max_iter_v3,
    'tol': tol_v3,
    'Silhouette Score': silhouette_avg_v3
}
```

```
#print results
print(km_v3_train_results)

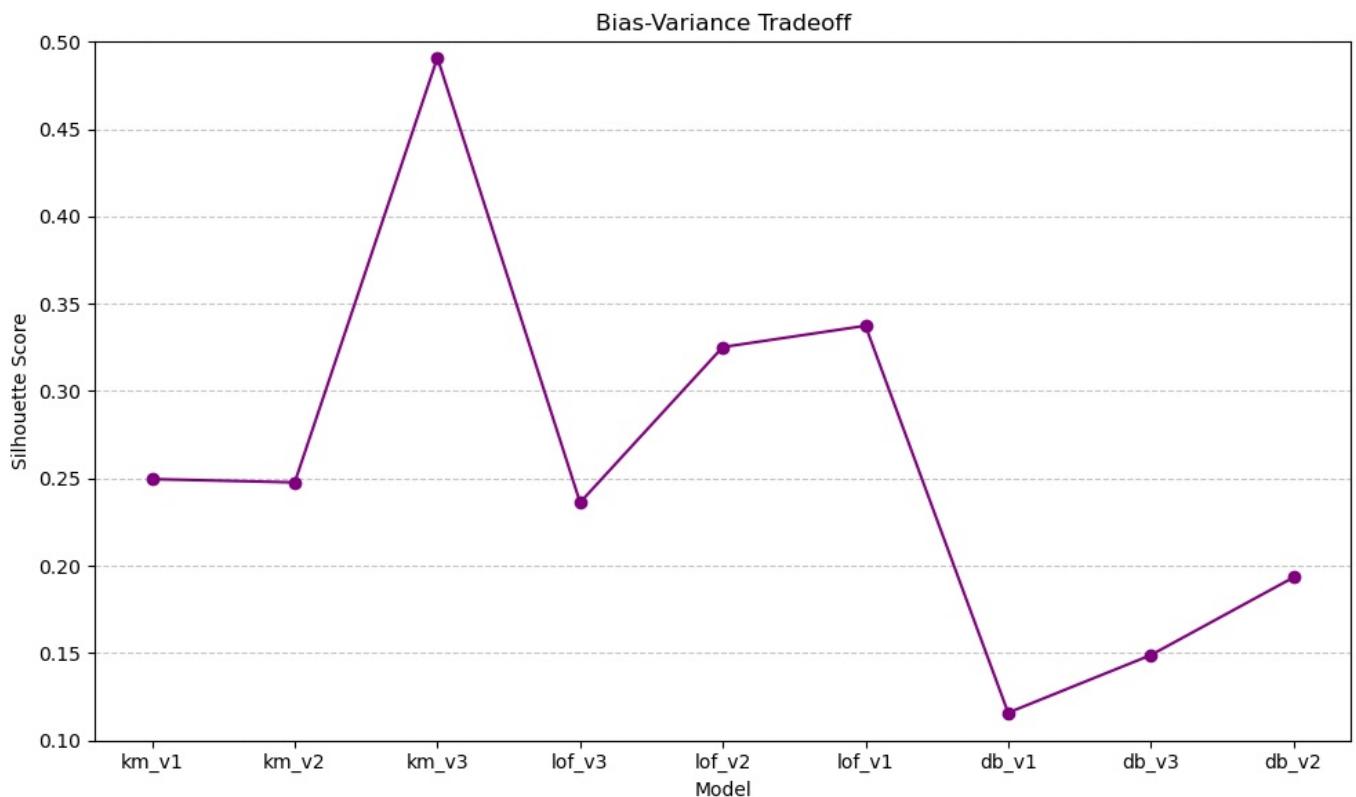
{'Variation': 'km_v3', 'Init': 'k-means++', 'n_init': 1, 'k': 10, 'max_iter': 200, 'tol': 0.001, 'Silhouette Score': 0.4670284163947236}
```

In [111]: #Bias Variance Tradeoff Plot

```
#9 model variations to plot (High Variance, Low Bias: K-Means; Middle Variance, Middle Bias: LOF; Low Variance, models = ['km_v1', 'km_v2', 'km_v3', 'lof_v3', 'lof_v2', 'lof_v1', 'db_v1', 'db_v3', 'db_v2']
#corresponding silhouette scores to analyze bias-variance tradeoff for unsupervised learning task
silhouette_scores = [0.2496, 0.2477, 0.4909, 0.2362, 0.3252, 0.3374, 0.1158, 0.1488, 0.1933]

#create the plot
plt.figure(figsize=(10, 6))
plt.plot(models, silhouette_scores, marker='o', color='purple', linestyle='--')
plt.xlabel('Model')
plt.ylabel('Silhouette Score')
plt.title('Bias-Variance Tradeoff')
plt.ylim(0.1, 0.5)
plt.grid(axis='y', linestyle='--', alpha=0.7)

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



In [112]: #create a dictionary with the performance metrics

```
kmeans_v3_scores = {
    "Model Stage": ["Train", "Validation", "Test"],
    "Silhouette Score": [0.4638, 0.4909, 0.4865]
}

#create a DataFrame
performance_metrics = pd.DataFrame(kmeans_v3_scores)

#print the title and the table using tabulate
table_title = "Winning Model Performance Metrics by Stage"
print(table_title)
print(tabulate(performance_metrics, headers='keys', tablefmt='pretty'))
```

Winning Model Performance Metrics by Stage

	Model Stage	Silhouette Score
0	Train	0.4638
1	Validation	0.4909
2	Test	0.4865

In [113]: #identify small clusters as anomalies based on k-means clustering of winning model
#this will be used to create labeled data, which enables SMOTE to be used
def get_anomaly_clusters(X, n_clusters=10, percentile_threshold=10):

```

kmeans = KMeans(n_clusters=n_clusters, init='k-means++', n_init=1, max_iter=200, tol=1e-3, random_state=42)
kmeans_labels = kmeans.fit_predict(X)

#calculate the size of each cluster
cluster_sizes = np.bincount(kmeans_labels)

#define threshold size based on the given percentile
threshold_size = np.percentile(cluster_sizes, percentile_threshold)

#define identify small clusters based on the threshold
small_clusters = np.where(cluster_sizes <= threshold_size)[0]

return small_clusters, kmeans_labels

```

In [114]: #Data-Centric AI 1: Apply SMOTE to the train data only (on small clusters (smallest 10th percentile of clusters))

```

def apply_smote(X_train, kmeans_labels, small_clusters):
    smote = SMOTE(random_state=42)
    X_train_resampled = X_train.copy()

    #create a new DataFrame for the small clusters
    X_small_clusters = pd.DataFrame()
    y_small_clusters = []

    #labels array: `0` for small clusters and `1` for normal clusters
    y_train_labels = np.ones(X_train.shape[0]) # Default to normal (1) for all points
    for cluster in small_clusters:
        #get the indices of the points in the small cluster
        cluster_indices = np.where(kmeans_labels == cluster)[0]

        #assign label `0` to points in small clusters
        y_train_labels[cluster_indices] = 0

        #extract the points in that small cluster
        X_cluster = X_train.iloc[cluster_indices]

        #only apply SMOTE if the cluster has more than 1 sample
        if X_cluster.shape[0] > 1:
            #create pseudo-labels for SMOTE: assign a unique label for each small cluster
            y_cluster = np.full(X_cluster.shape[0], cluster) # Each small cluster gets a unique label

            #append to the small clusters dataset
            X_small_clusters = pd.concat([X_small_clusters, X_cluster], ignore_index=True)
            y_small_clusters.extend(y_cluster)

    #apply SMOTE on the combined small clusters with labels
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train_labels)

    return X_resampled

```

In [115]: #Data-Centric AI 2: Apply PCA to reduce dimensionality while retaining 95% of the dataset's variance (train, validation, test)

```

def apply_pca(X_train, X_val, X_test, n_components=0.95):
    pca = PCA(n_components=n_components)
    X_train_pca = pca.fit_transform(X_train)
    X_val_pca = pca.transform(X_val)
    X_test_pca = pca.transform(X_test)

    return X_train_pca, X_val_pca, X_test_pca

```

In [116]: # Data-Centric AI 3: Inject Gaussian noise into the data (train, validation, test)

```

def inject_noise(X_train, X_val, X_test, noise_factor=0.1):
    noise_train = np.random.normal(loc=0, scale=noise_factor, size=X_train.shape)
    noise_val = np.random.normal(loc=0, scale=noise_factor, size=X_val.shape)
    noise_test = np.random.normal(loc=0, scale=noise_factor, size=X_test.shape)

    X_train_noisy = X_train + noise_train
    X_val_noisy = X_val + noise_val
    X_test_noisy = X_test + noise_test

    return X_train_noisy, X_val_noisy, X_test_noisy

```

In [117]: # Function to apply all Data-Centric AI Techniques

```

def apply_data_centric_ai(X_train, X_val, X_test, n_clusters=10, percentile_threshold=10, noise_factor=0.1, pca_n_components=0.95):
    #identify small clusters using k-means
    small_clusters, kmeans_labels = get_anomaly_clusters(X_train, n_clusters=n_clusters, percentile_threshold=percentile_threshold)

    #apply SMOTE to the train set based on small clusters
    X_train_smote = apply_smote(X_train, kmeans_labels, small_clusters)

    #apply PCA to reduce dimensionality for train, validation, and test sets
    X_train_pca, X_val_pca, X_test_pca = apply_pca(X_train_smote, X_val, X_test, n_components=pca_n_components)

    #inject Gaussian noise to the data

```

```
X_train_noisy, X_val_noisy, X_test_noisy = inject_noise(X_train_pca, X_val_pca, X_test_pca, noise_factor=noise_factor)

return X_train_noisy, X_val_noisy, X_test_noisy, small_clusters, kmeans_labels
```

```
In [118]: #apply the data-centric AI techniques (SMOTE, PCA, Noise Injection)
X_train_final, X_val_final, X_test_final, small_clusters, kmeans_labels = apply_data_centric_ai(
X_train, X_val, X_test, n_clusters=10, percentile_threshold=10, noise_factor=0.1, pca_n_components=0.95)
```

```
In [119]: #fit the k-means model again on the final data
kmeans = KMeans(n_clusters=10, init='k-means++', n_init=1, max_iter=200, tol=1e-3, random_state=42)
kmeans.fit(X_train_final) # Fit on the final transformed train data
labels_train_final = kmeans.labels_
silhouette_avg_train = silhouette_score(X_train_final, labels_train_final)

labels_val_final = kmeans.predict(X_val_final) # Predict the labels for the validation set
silhouette_avg_val = silhouette_score(X_val_final, labels_val_final) # Compute silhouette score for validation

labels_test_final = kmeans.predict(X_test_final) # Predict the labels for the test set
silhouette_avg_test = silhouette_score(X_test_final, labels_test_final) # Compute silhouette score for test
```

```
In [120]: #print results of model on data-centric AI-transformed datasets
print(f"Final Silhouette Score on Transformed Train Data: {silhouette_avg_train}")
print(f"Final Silhouette Score on Transformed Validation Data: {silhouette_avg_val}")
print(f"Final Silhouette Score on Transformed Test Data: {silhouette_avg_test}")
```

```
Final Silhouette Score on Transformed Train Data: 0.3168917846266198
Final Silhouette Score on Transformed Validation Data: 0.3307452133405773
Final Silhouette Score on Transformed Test Data: 0.33167209815920734
```

```
In [121]: #create a dictionary with the performance metrics
data_centric_ai_kmeans_scores = {
    "Model Stage": ["Train", "Validation", "Test"],
    "Silhouette Score": [0.2975, 0.3100, 0.3083]
}

#create a DataFrame
performance_metrics = pd.DataFrame(data_centric_ai_kmeans_scores)

#print the title and the table using tabulate
table_title = "Winning Model Performance Metrics by Stage on Data-Centric AI-Transformed Data"
print(table_title)
print(tabulate(performance_metrics, headers='keys', tablefmt='pretty'))
```

```
Winning Model Performance Metrics by Stage on Data-Centric AI-Transformed Data
+-----+-----+
| Model Stage | Silhouette Score |
+-----+-----+
| 0 | Train | 0.2975 |
| 1 | Validation | 0.31 |
| 2 | Test | 0.3083 |
+-----+-----+
```

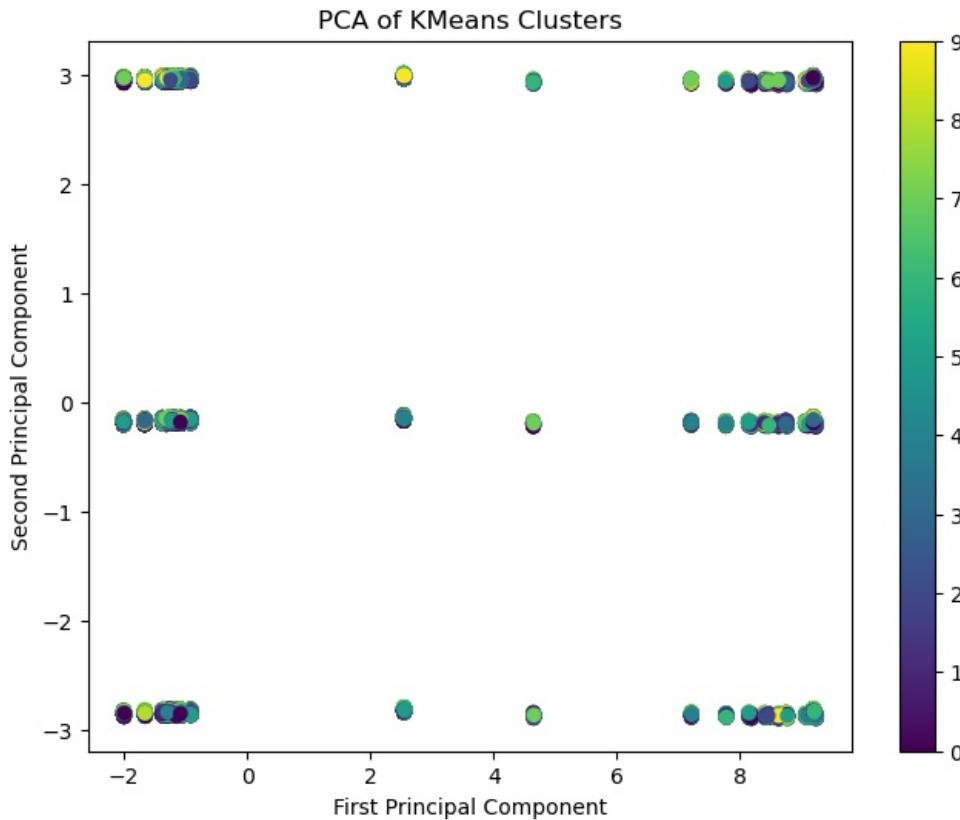
```
In [129]: #perform PCA to aid in identifying most relevant predictors in the dataset
#scale the data
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)

#perform PCA
pca = PCA(n_components=4)
X_pca = pca.fit_transform(X_train_scaled)

#plot the first two principal components
plt.figure(figsize=(8,6))
plt.scatter(X_pca[:, 0], X_pca[:, 1], c=kmeans_v3.labels_, cmap='viridis')
plt.colorbar()
plt.title('PCA of KMeans Clusters')
plt.xlabel('First Principal Component')
plt.ylabel('Second Principal Component')
plt.show()

#explained variance ratio tells us how much of the variance is captured by each component
print("Explained variance ratio by component:", pca.explained_variance_ratio_)

#examine which features contribute most to the first principal component
features_pca = pd.DataFrame(pca.components_, columns=X_train.columns, index=[f'PC{i+1}' for i in range(len(pca.components_))])
print(features_pca)
```



Explained variance ratio by component: [0.20309139 0.11743129 0.08247242 0.07047652]

	Day_0	Day_1	Day_2	Day_3	Day_4	Day_5	Day_6	\
PC1	0.000538	0.000168	-0.000191	0.000497	-0.000446	-0.000570	-0.000007	
PC2	0.000289	-0.002065	0.000242	-0.000643	-0.002477	0.003898	0.000779	
PC3	-0.000907	0.002440	0.000049	-0.001050	0.001592	-0.001414	-0.000734	
PC4	-0.000095	-0.001071	0.002658	-0.004678	0.002629	-0.000394	0.000914	

	Deviation_From_Mean	Account_EMBEDDINGS_0	Account_EMBEDDINGS_1	...	\
PC1	0.001025	-0.206456	-0.160800	...	
PC2	-0.001878	-0.000682	-0.003325	...	
PC3	0.002006	0.301938	0.251505	...	
PC4	0.000547	0.004697	0.004279	...	

	Location_EMBEDDINGS_6	Location_EMBEDDINGS_7	Hour_Group_EMBEDDINGS_0	\
PC1	-0.001340	0.001438	-0.297689	
PC2	0.002474	-0.001925	0.000762	
PC3	-0.002506	-0.000497	-0.182555	
PC4	0.228868	-0.081236	-0.003123	

	Hour_Group_EMBEDDINGS_1	Hour_Group_EMBEDDINGS_2	\
PC1	0.290015	-0.294804	
PC2	-0.000824	0.000717	
PC3	0.209992	-0.194136	
PC4	0.003490	-0.003409	

	Hour_Group_EMBEDDINGS_3	Hour_Group_EMBEDDINGS_4	\
PC1	0.298893	-0.291579	
PC2	-0.000684	0.000727	
PC3	0.174222	-0.205742	
PC4	0.003235	-0.003528	

	Hour_Group_EMBEDDINGS_5	Hour_Group_EMBEDDINGS_6	Hour_Group_EMBEDDINGS_7
PC1	-0.293736	0.290946	0.294980
PC2	0.000785	-0.000725	-0.000811
PC3	-0.199825	0.208054	0.194066
PC4	-0.003452	0.003584	0.003294

[4 rows x 48 columns]

```
In [130]: #store the original feature names before scaling
original_feature_names = X_train.columns # This stores the original feature names

#fit model with scaled data
kmeans = KMeans(n_clusters=10, init='k-means++', n_init=1, max_iter=200, tol=1e-3, random_state=42)
kmeans.fit(X_train_scaled)

#get the cluster centroids
centroids = kmeans.cluster_centers_

#assign clusters to each data point
assigned_centroids = kmeans.predict(X_train_scaled)
```

```

#identify random points farthest from their centroids (anomalies)
# Calculate distances from each point to its assigned centroid
distances = np.linalg.norm(X_train_scaled - centroids[assigned_centroids], axis=1)

#sort the indices of the points based on the distance to centroid (farthest first)
farthest_indices = np.argsort(distances)[::-1]

#select 5 random points from those farthest from the centroid
random_furthest_points = farthest_indices[:5]

#calculate the feature-wise contribution to the distance from the centroid for each point
feature_contributions = []

for idx in random_furthest_points:
    point = X_train_scaled[idx] # Use array indexing to select the point
    assigned_cluster = assigned_centroids[idx]
    centroid = centroids[assigned_cluster]

    #calculate the squared difference for each feature (contribution to distance)
    feature_contrib = (point - centroid) ** 2
    feature_contributions.append(feature_contrib)

#manually generate column names based on the original dataset columns
#since X_train_scaled is a NumPy array, we need to manually use the feature names from the original dataset
columns = original_feature_names

#convert the contributions to a DataFrame for better readability
#manually create a DataFrame and match it with the original feature names
feature_contributions_df = pd.DataFrame(feature_contributions, columns=columns)
print("Feature-wise contributions to distance from centroid for selected points:")
print(feature_contributions_df)

#simulate adjusting the features of a randomly selected point to move 10% closer to its centroid
adjusted_points = []

for idx in random_furthest_points:
    point = X_train_scaled[idx] # Use array indexing to select the point
    assigned_cluster = assigned_centroids[idx]
    centroid = centroids[assigned_cluster]

    #calculate the difference for each feature
    feature_differences = point - centroid

    #calculate how much to adjust each feature (reduce by 10% of the difference)
    adjustment = 0.1 * feature_differences # 10% adjustment

    #apply the adjustment to the features
    new_point = point - adjustment # Move the point closer to the centroid
    adjusted_points.append(new_point)

#convert the adjusted points into a DataFrame for better readability
#again, manually assigning feature names
adjusted_points_df = pd.DataFrame(adjusted_points, columns=columns)
print("\nAdjusted feature values for points moved closer to centroids (10% adjustment):")
print(adjusted_points_df)

#calculate the new distance after the adjustment to see how much the point moved
new_distances = np.linalg.norm(np.array(adjusted_points_df) - centroids[assigned_centroids[random_furthest_points]])
print("\nNew distances to centroid after adjustment:")
print(new_distances)

```

Feature-wise contributions to distance from centroid for selected points:

	Day_0	Day_1	Day_2	Day_3	Day_4	Day_5	Day_6	\
0	0.172944	0.178478	0.168865	6.193674	0.170708	0.149959	0.168822	
1	5.853997	0.178478	0.168865	0.157520	0.170708	0.149959	0.168822	
2	0.172944	0.178478	5.863488	0.157520	0.170708	0.149959	0.168822	
3	0.172944	0.178478	5.863488	0.157520	0.170708	0.149959	0.168822	
4	5.853997	0.178478	0.168865	0.157520	0.170708	0.149959	0.168822	

	Deviation_From_Mean	Account_EMBEDDINGS_0	Account_EMBEDDINGS_1	...	\
0	54.933322	0.663448	4.583788	...	
1	55.334588	3.789776	0.091831	...	
2	50.722760	3.694462	0.532415	...	
3	68.583963	0.851135	0.497069	...	
4	36.652422	0.663448	4.583788	...	

	Location_EMBEDDINGS_6	Location_EMBEDDINGS_7	Hour_Group_EMBEDDINGS_0	\
0	1.234927	1.318217	0.299601	
1	0.103593	0.000229	0.326588	
2	1.234927	1.318217	0.351242	
3	1.376403	1.234520	0.099590	
4	0.186114	1.307812	0.299601	

```

Hour_Group_EMBEDDINGS_1 Hour_Group_EMBEDDINGS_2 Hour_Group_EMBEDDINGS_3 \
0 0.546429 1.102460 0.995470
1 2.166443 1.692519 0.120167
2 1.599637 1.160543 0.358457
3 0.316073 0.173764 0.057935
4 0.546429 1.102460 0.995470

Hour_Group_EMBEDDINGS_4 Hour_Group_EMBEDDINGS_5 Hour_Group_EMBEDDINGS_6 \
0 1.018894 0.946160 2.409737
1 1.028531 1.674078 2.082085
2 1.261799 0.766135 1.021011
3 0.269722 0.216959 0.291788
4 1.018894 0.946160 2.409737

Hour_Group_EMBEDDINGS_7
0 0.710120
1 1.211242
2 0.140541
3 0.172984
4 0.710120

[5 rows x 48 columns]

Adjusted feature values for points moved closer to centroids (10% adjustment):
Day_0 Day_1 Day_2 Day_3 Day_4 Day_5 Day_6 \
0 -0.371195 -0.371911 -0.372300 2.233960 -0.360731 -0.36308 -0.369594
1 2.180637 -0.371911 -0.372300 -0.363077 -0.360731 -0.36308 -0.369594
2 -0.371195 -0.371911 2.176856 -0.363077 -0.360731 -0.36308 -0.369594
3 -0.371195 -0.371911 2.176856 -0.363077 -0.360731 -0.36308 -0.369594
4 2.180637 -0.371911 -0.372300 -0.363077 -0.360731 -0.36308 -0.369594

Deviation_From_Mean Account_EMBEDDINGS_0 Account_EMBEDDINGS_1 ... \
0 -6.655709 -0.569170 0.941719 ...
1 -6.680027 0.449820 -0.712429 ...
2 -6.394970 0.427647 -0.328461 ...
3 -7.438567 -2.132554 -1.619689 ...
4 -5.433889 -0.569170 0.941719 ...

Location_EMBEDDINGS_6 Location_EMBEDDINGS_7 Hour_Group_EMBEDDINGS_0 \
0 0.984752 -1.021594 -3.458931
1 -0.305065 -0.001903 -3.480639
2 0.984752 -1.021594 -3.499699
3 1.040489 -0.988252 -2.682287
4 -0.403662 1.040965 -3.458931

Hour_Group_EMBEDDINGS_1 Hour_Group_EMBEDDINGS_2 Hour_Group_EMBEDDINGS_3 \
0 3.562429 -3.881170 3.871824
1 4.221837 -4.107058 3.285850
2 4.035433 -3.905743 3.512706
3 2.391158 -2.561021 2.757238
4 3.562429 -3.881170 3.871824

Hour_Group_EMBEDDINGS_4 Hour_Group_EMBEDDINGS_5 Hour_Group_EMBEDDINGS_6 \
0 -3.816084 -3.807281 4.300482
1 -3.820370 -4.096320 4.202031
2 -3.918590 -3.719606 3.812788
3 -2.440209 -2.512634 2.417226
4 -3.816084 -3.807281 4.300482

Hour_Group_EMBEDDINGS_7
0 3.702535
1 3.934625
2 3.281516
3 2.569795
4 3.702535

[5 rows x 48 columns]

```

New distances to centroid after adjustment:
[9.7736433 9.40544983 9.30040267 9.10915659 8.81125534]

In [126]: #to quantify bias, we will extract the most important contributing features to classification and manually determine the likelihood that these features contain inherent bias due to their correlations to protected classes that are not explicitly present in our data or model.

```

#find the features with the largest loadings (absolute values) for each principal component
top_features_per_pc = {}

for pc in features_pca.index:
    sorted_features = features_pca.loc[pc].abs().sort_values(ascending=False)
    top_features_per_pc[pc] = sorted_features.head(3) # Top 3 features for each PC

#display the most important features for each principal component

```

```

print("\nTop features per PC based on their contribution to variance:")
for pc, features in top_features_per_pc.items():
    print(f"{pc}: {features.index.tolist()}")

#add clustering results to the dataset from K-Means model for visualization
X_train_scaled_with_clusters = pd.DataFrame(X_train_scaled, columns=X_train.columns)
X_train_scaled_with_clusters['cluster'] = kmeans.labels_

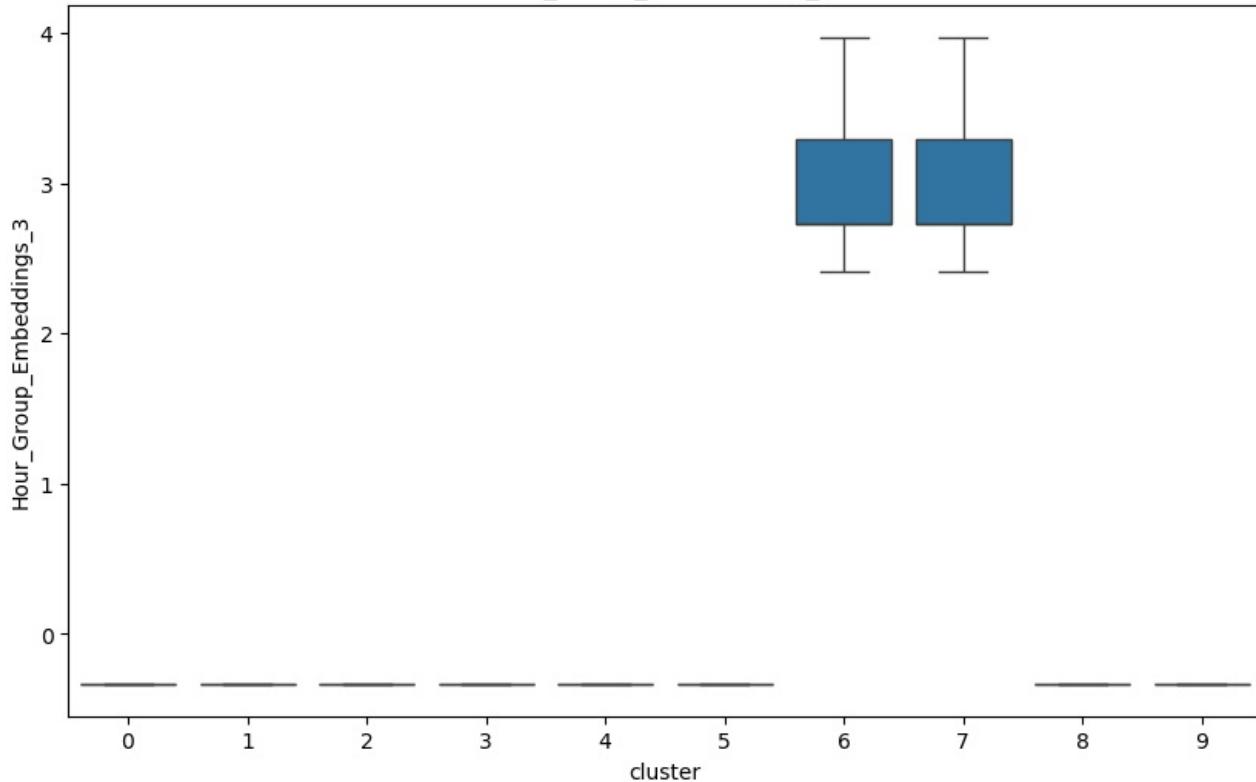
#visualize how the identified features vary across clusters (for bias detection)
for pc, features in top_features_per_pc.items():
    for feature in features.index:
        plt.figure(figsize=(10, 6))
        sns.boxplot(x='cluster', y=feature, data=X_train_scaled_with_clusters)
        plt.title(f'Distribution of {feature} across Clusters')
        plt.show()

```

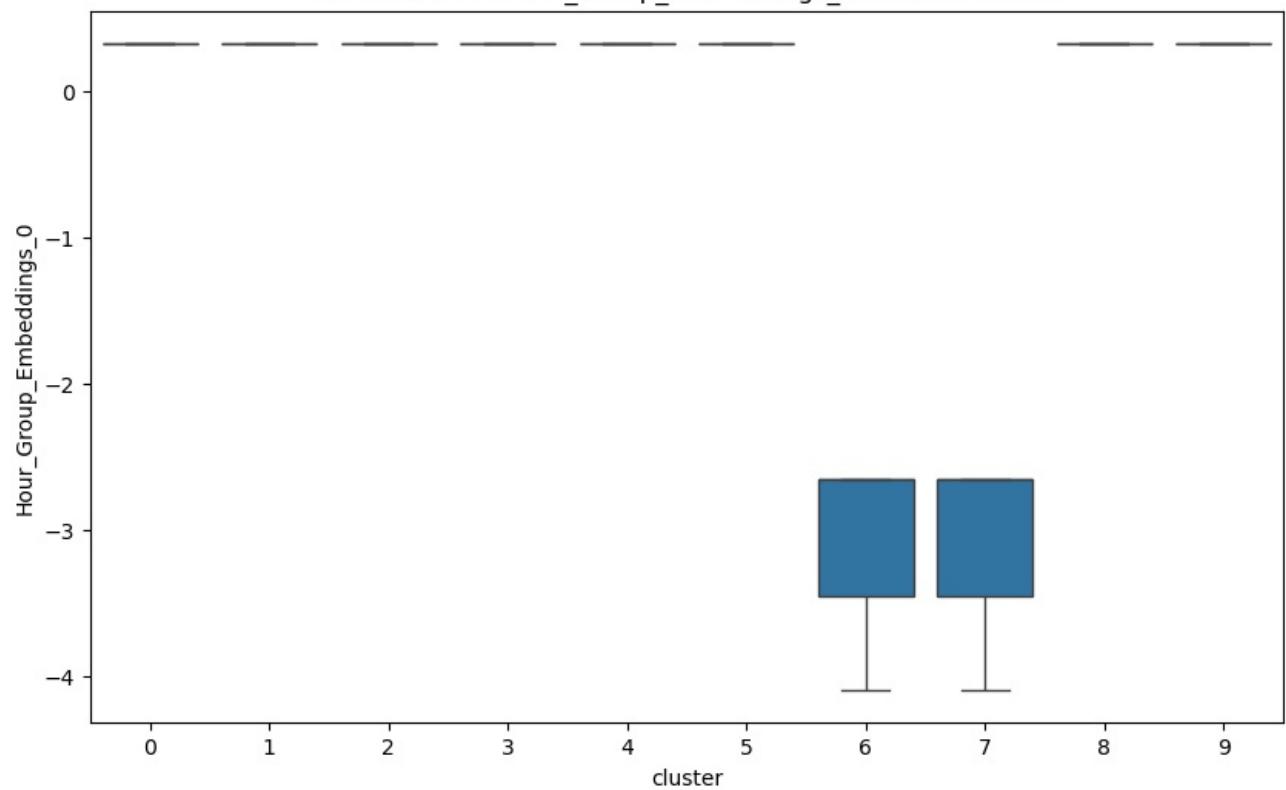
Top features per PC based on their contribution to variance:

PC1: ['Hour_Group_EMBEDDINGS_3', 'Hour_Group_EMBEDDINGS_0', 'Hour_Group_EMBEDDINGS_7']
 PC2: ['Transaction_EMBEDDINGS_0', 'Transaction_EMBEDDINGS_5', 'Transaction_EMBEDDINGS_3']
 PC3: ['Account_EMBEDDINGS_5', 'Account_EMBEDDINGS_3', 'Account_EMBEDDINGS_2']
 PC4: ['Location_EMBEDDINGS_3', 'Location_EMBEDDINGS_5', 'Location_EMBEDDINGS_2']

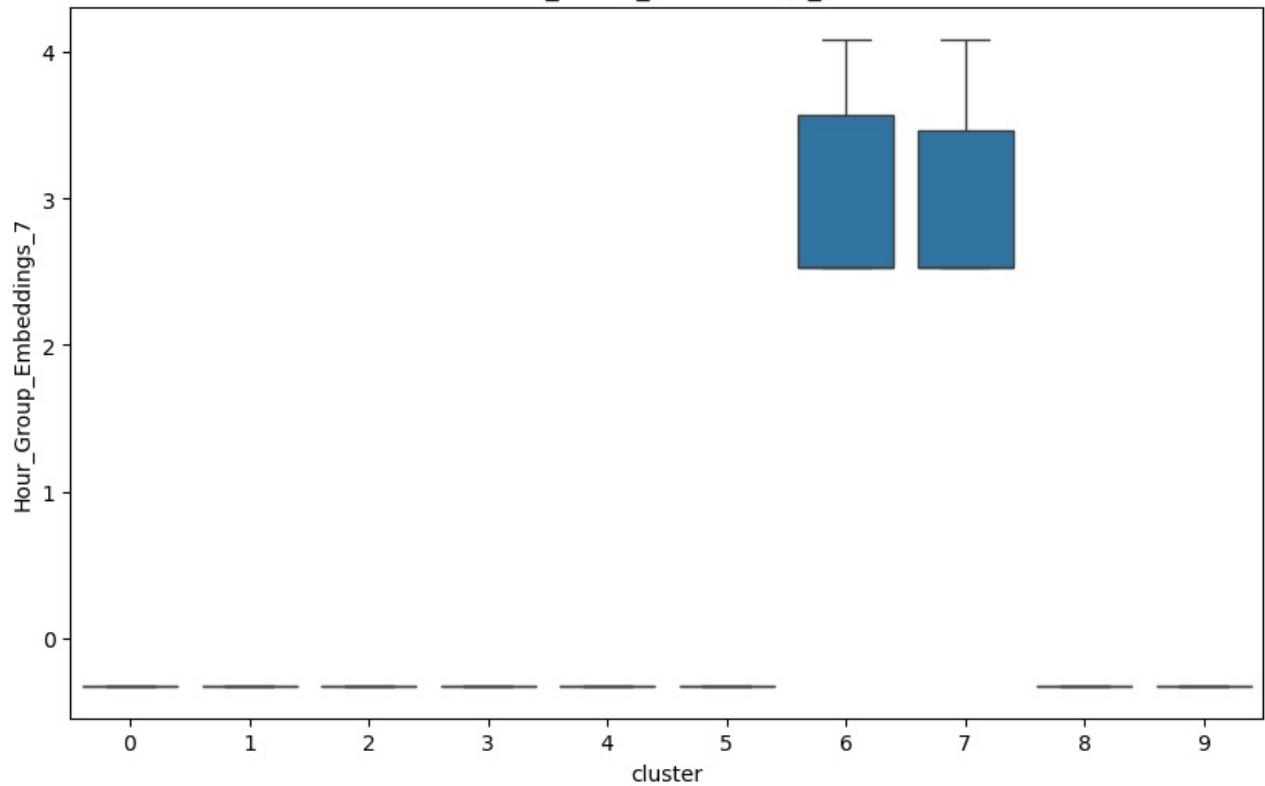
Distribution of Hour_Group_EMBEDDINGS_3 across Clusters



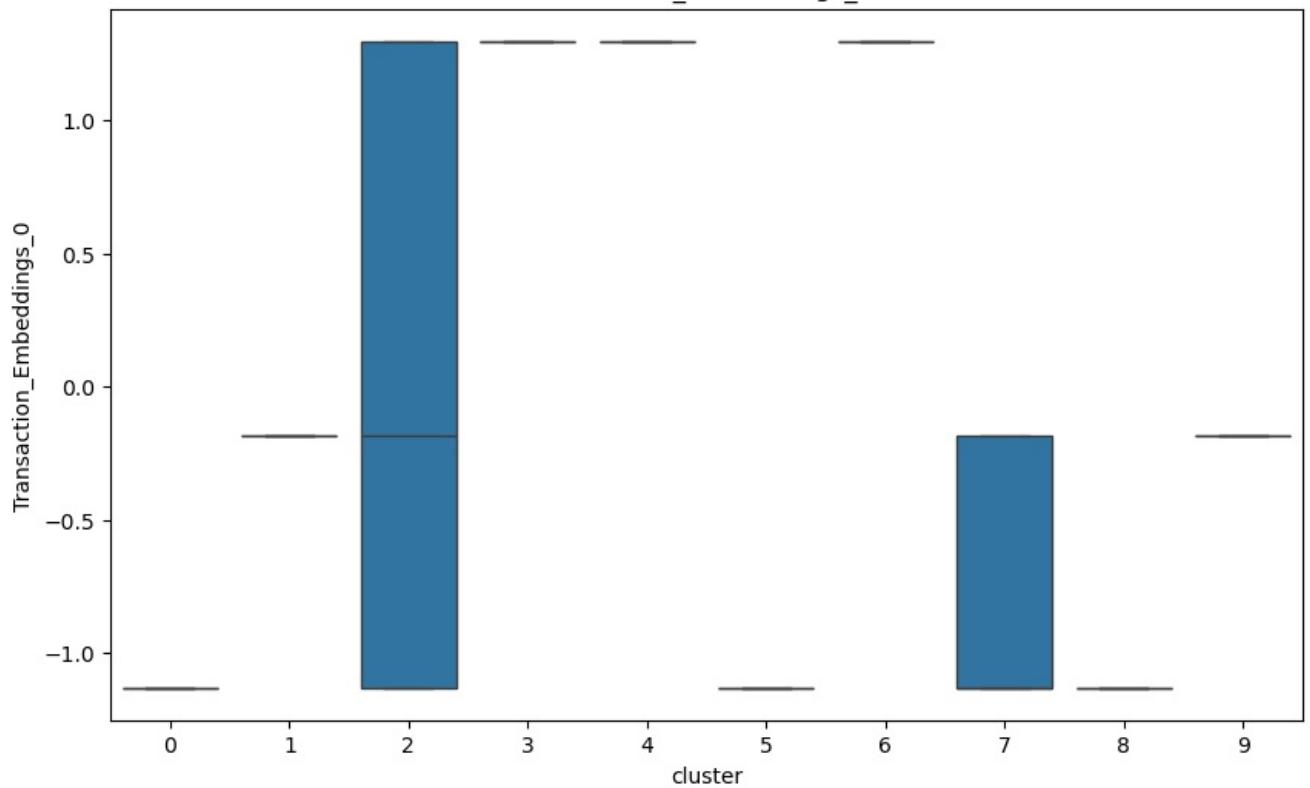
Distribution of Hour_Group_EMBEDDINGS_0 across Clusters



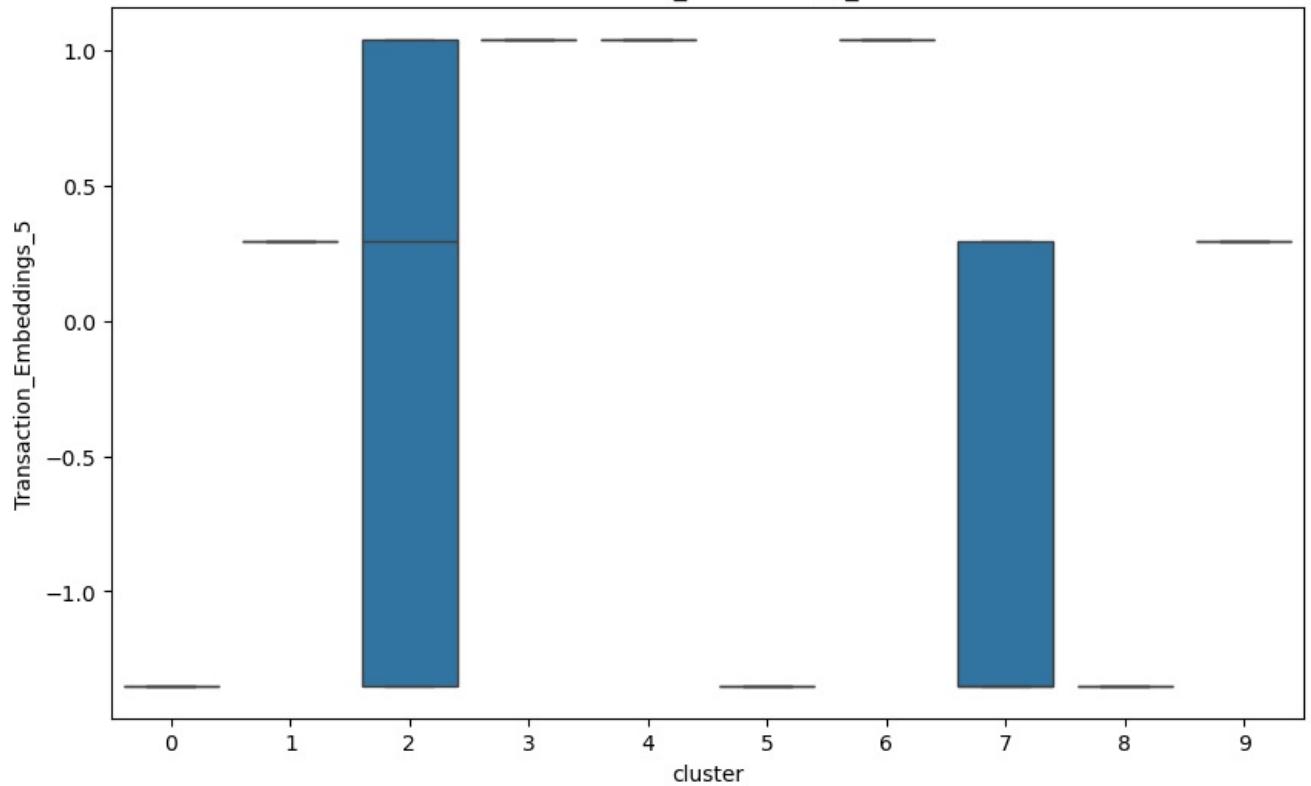
Distribution of Hour_Group_EMBEDDINGS_7 across Clusters



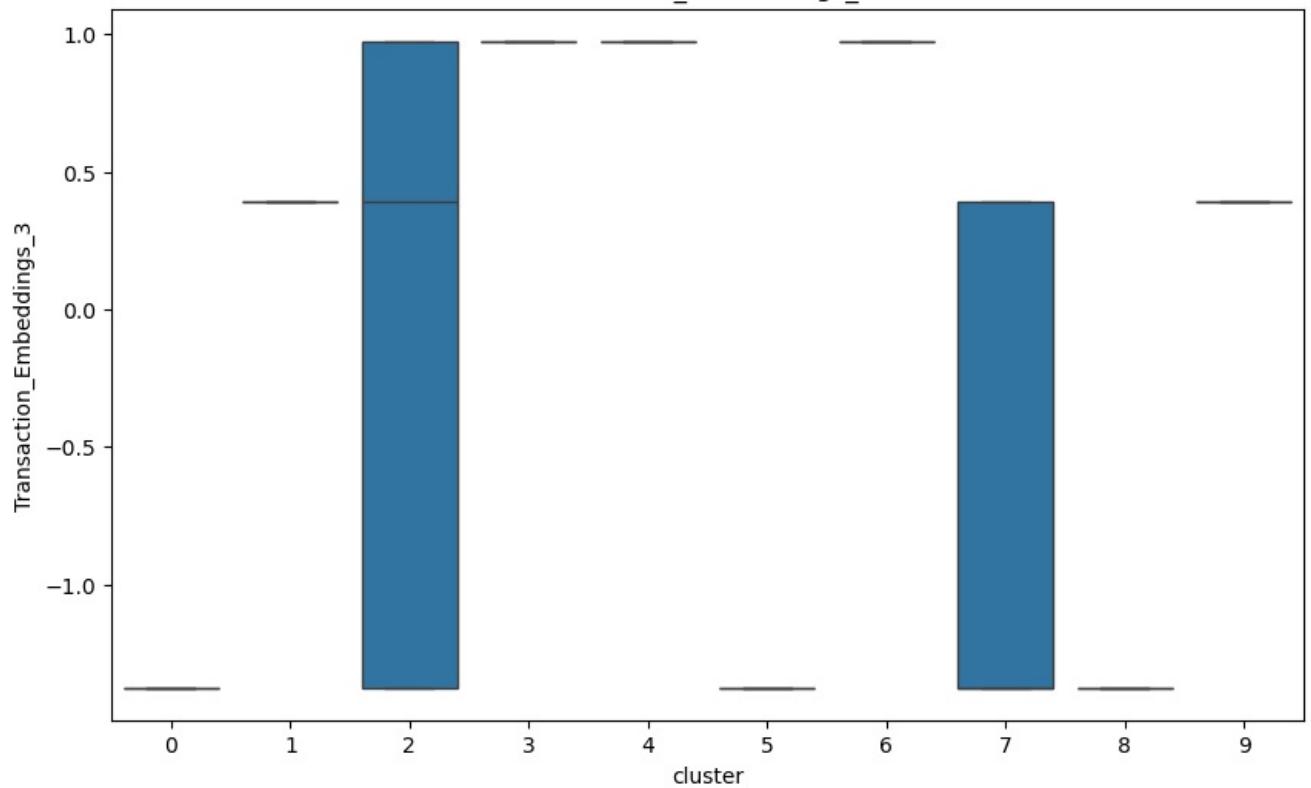
Distribution of Transaction_EMBEDDINGS_0 across Clusters



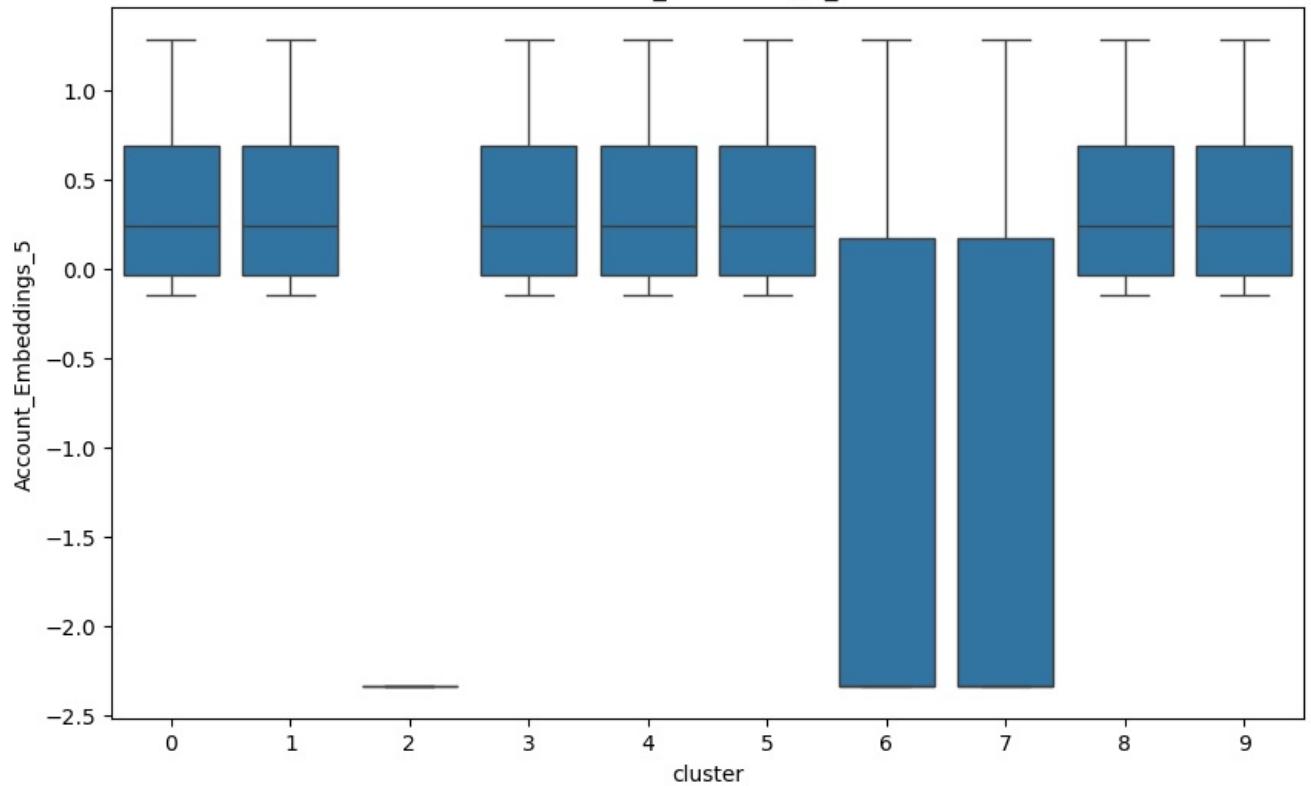
Distribution of Transaction_EMBEDDINGS_5 across Clusters

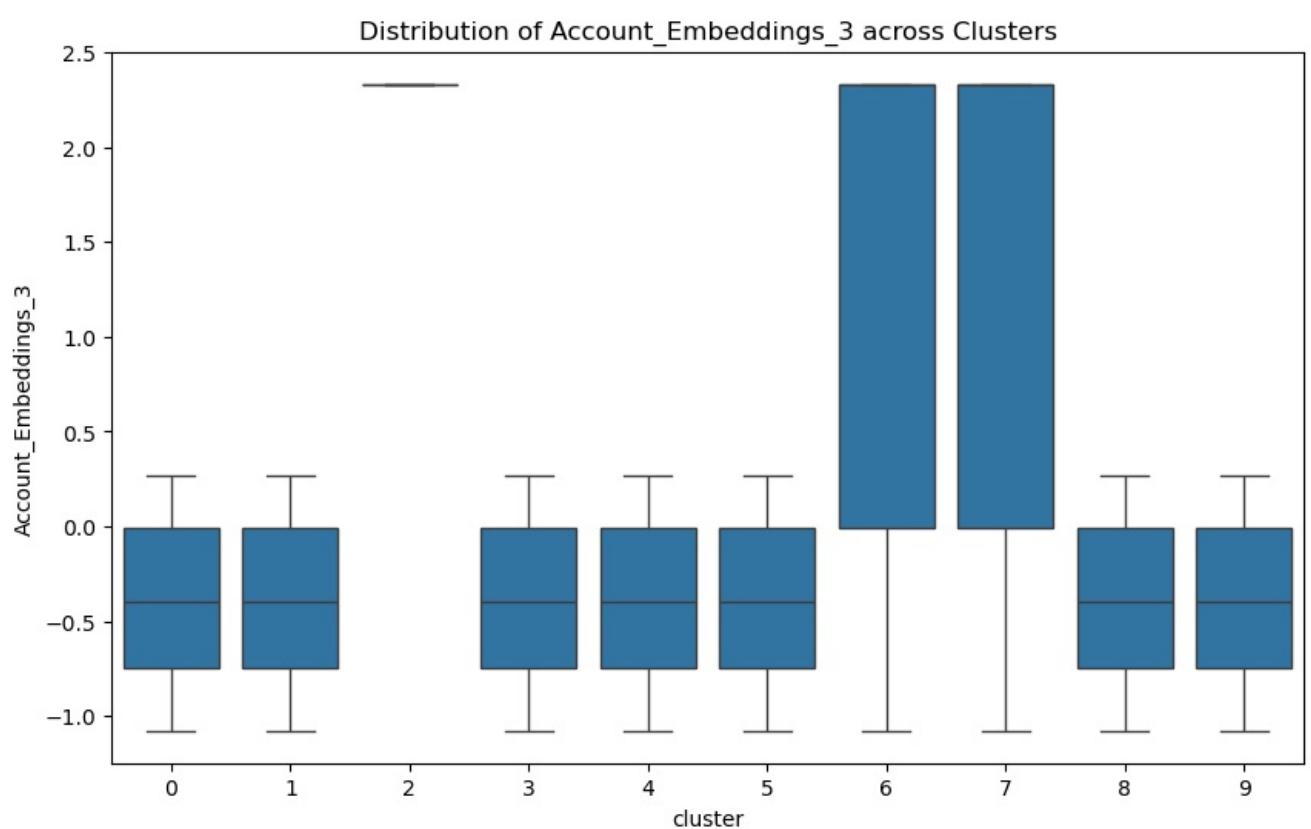


Distribution of Transaction_EMBEDDINGS_3 across Clusters

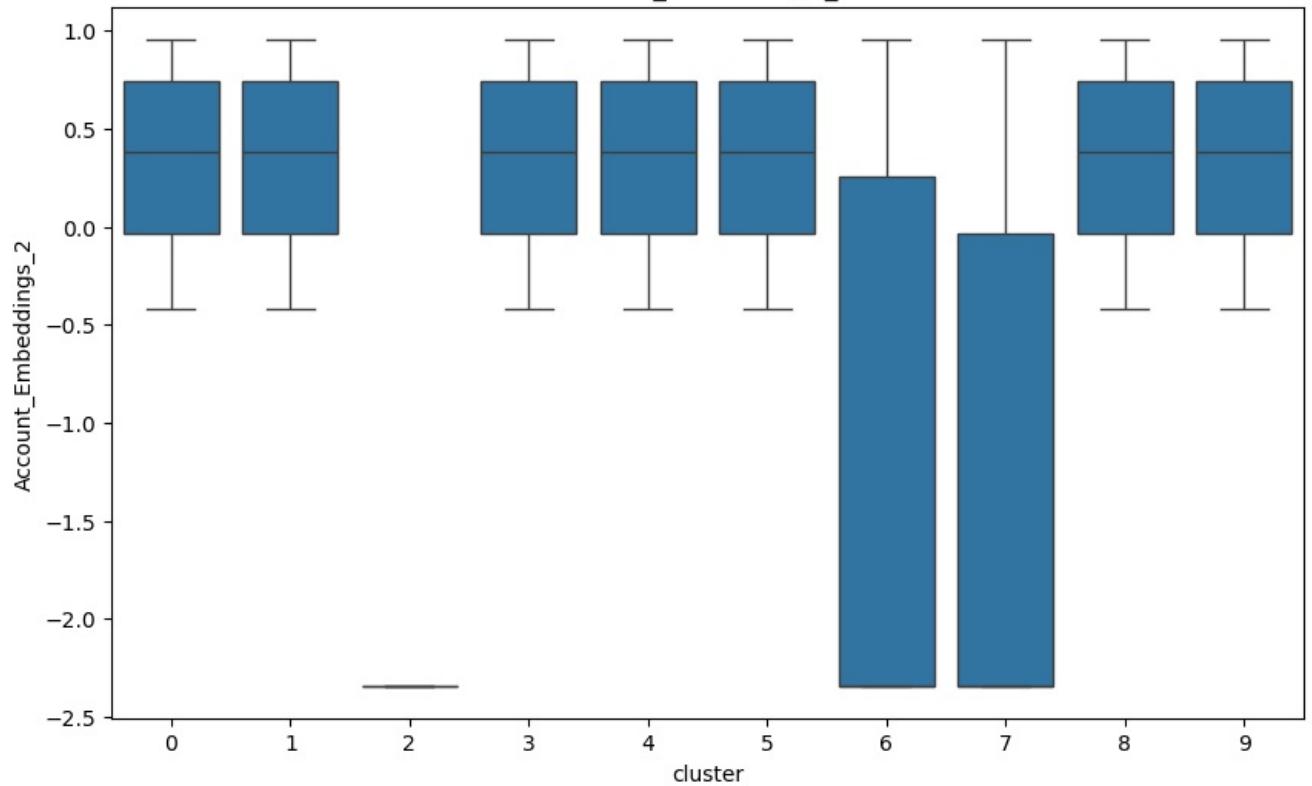


Distribution of Account_EMBEDDINGS_5 across Clusters

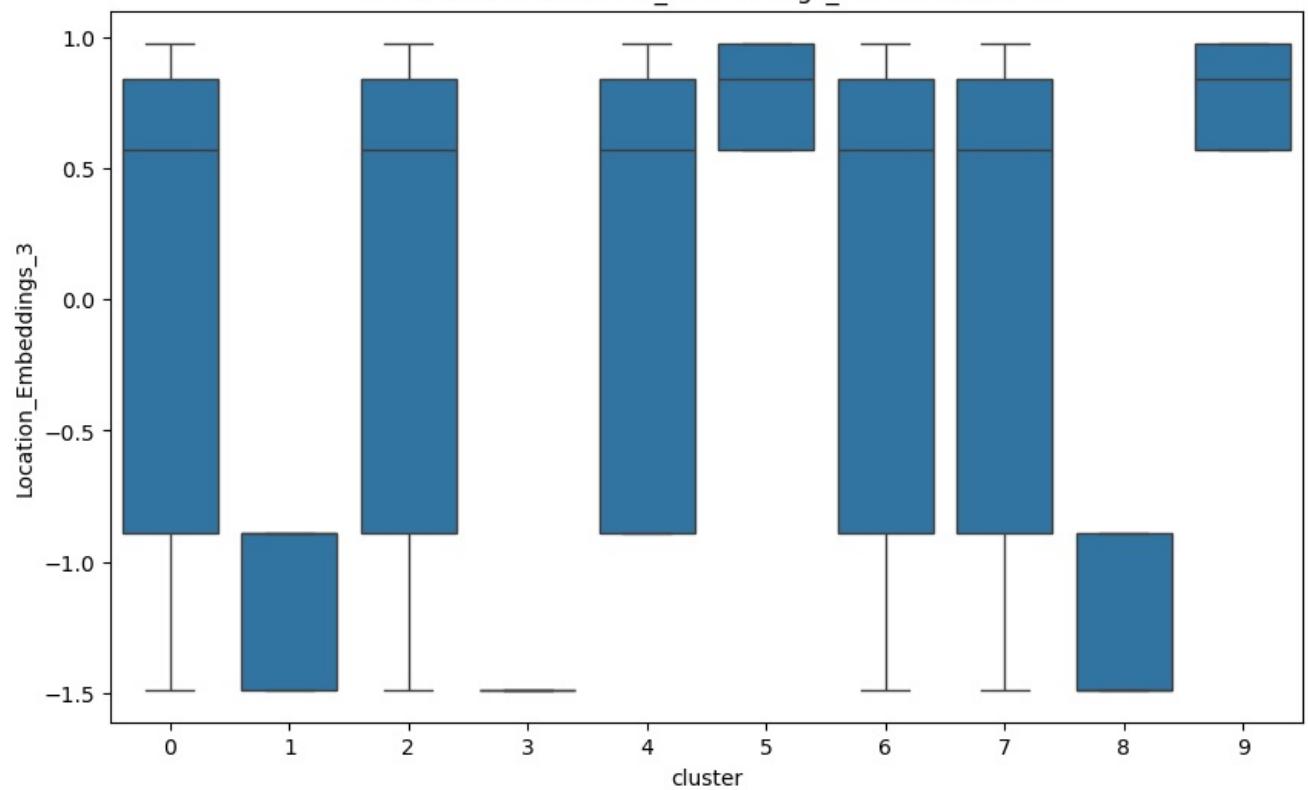




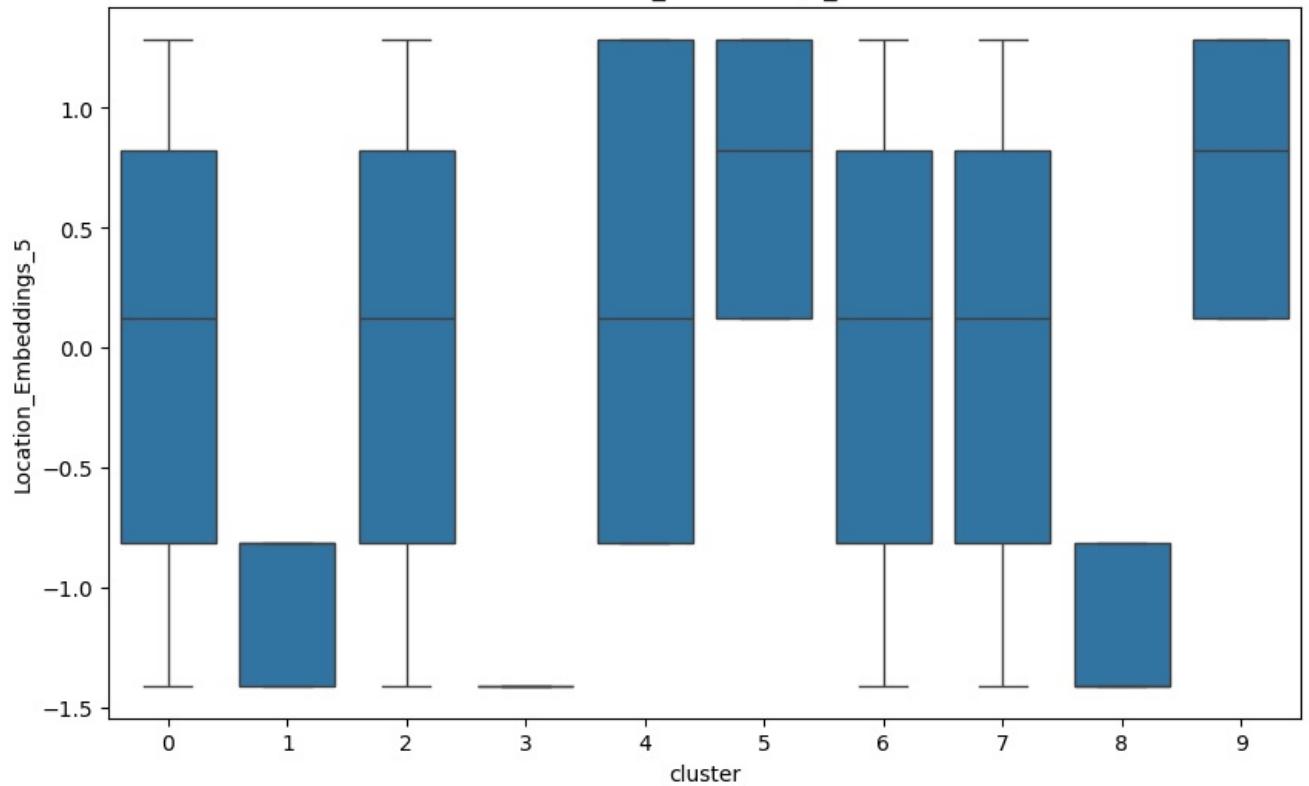
Distribution of Account_EMBEDDINGS_2 across Clusters

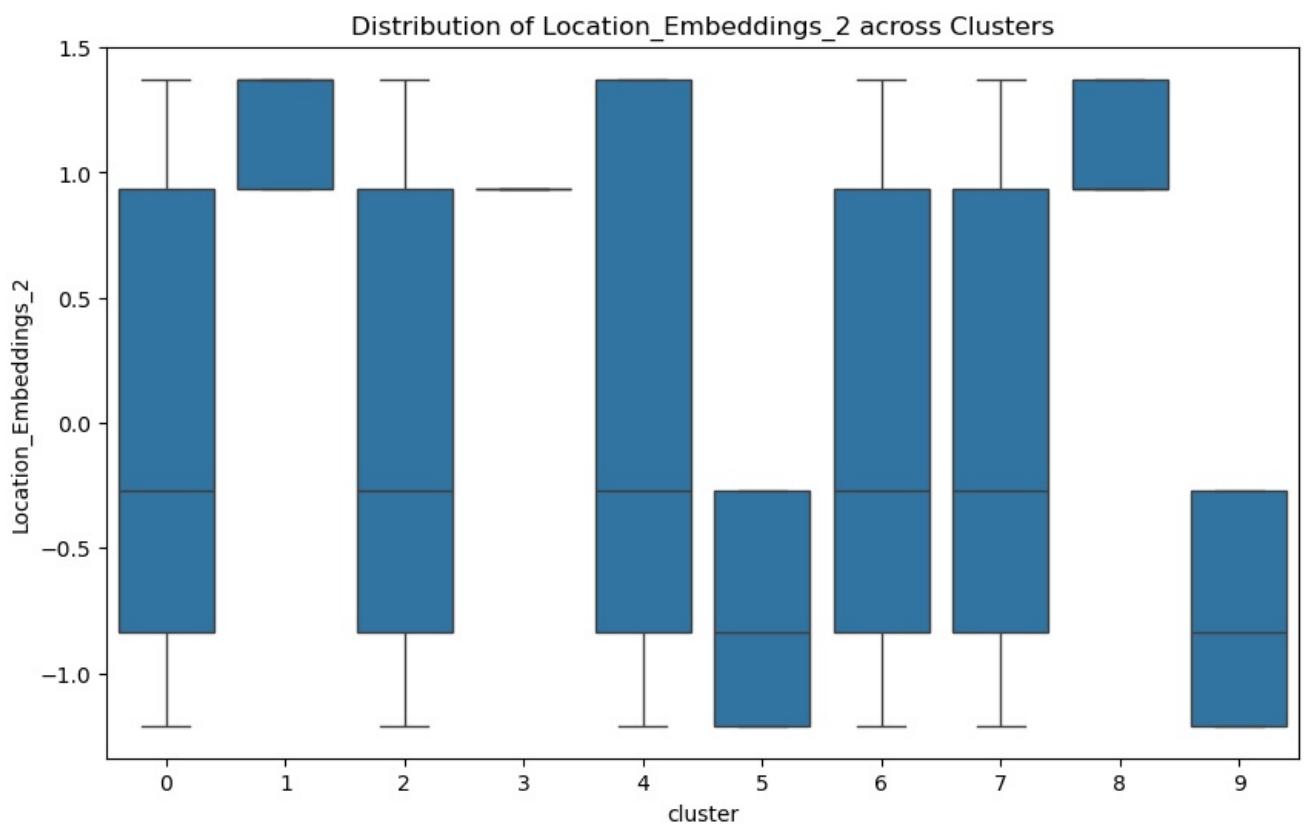


Distribution of Location_EMBEDDINGS_3 across Clusters



Distribution of Location_EMBEDDINGS_5 across Clusters





```
In [127]: #perform same process on validation and test datasets to analyze
#bias across these datasets as well

#apply the same scaler used for the training set
X_val_scaled = scaler.transform(X_val) # Use the same scaler for validation
X_test_scaled = scaler.transform(X_test) # Use the same scaler for test

#apply PCA transformation to the validation and test sets
X_val_pca = pca.transform(X_val_scaled) # Apply PCA to the validation set
X_test_pca = pca.transform(X_test_scaled) # Apply PCA to the test set

#predict cluster labels for validation and test sets
val_clusters = kmeans.predict(X_val_scaled) # Predict clusters for validation
test_clusters = kmeans.predict(X_test_scaled) # Predict clusters for test

#visualize bias in features based on PCA loadings
# Add clusters to the validation and test sets for visualization
X_val_scaled_with_clusters = pd.DataFrame(X_val_scaled, columns=X_val.columns)
X_val_scaled_with_clusters['cluster'] = val_clusters
```

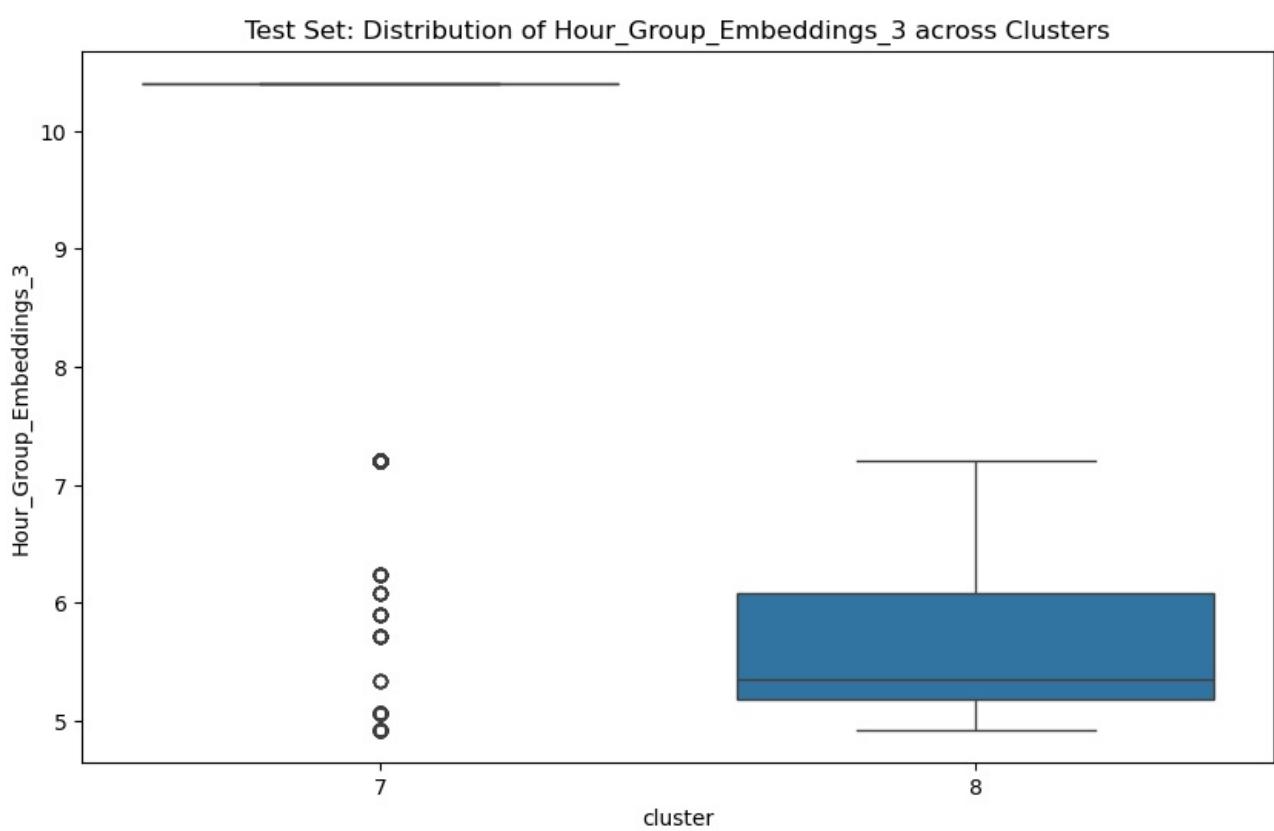
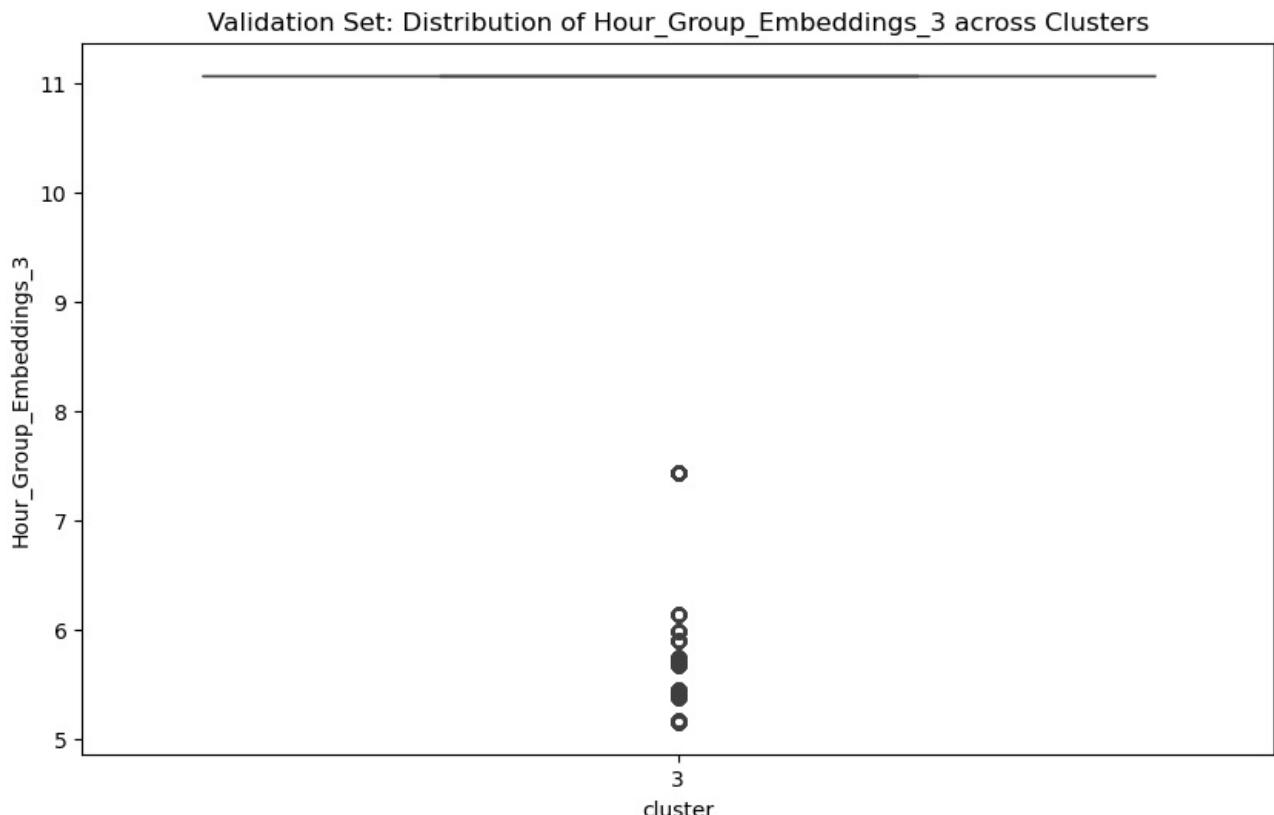
```

X_test_scaled_with_clusters = pd.DataFrame(X_test_scaled, columns=X_test.columns)
X_test_scaled_with_clusters['cluster'] = test_clusters

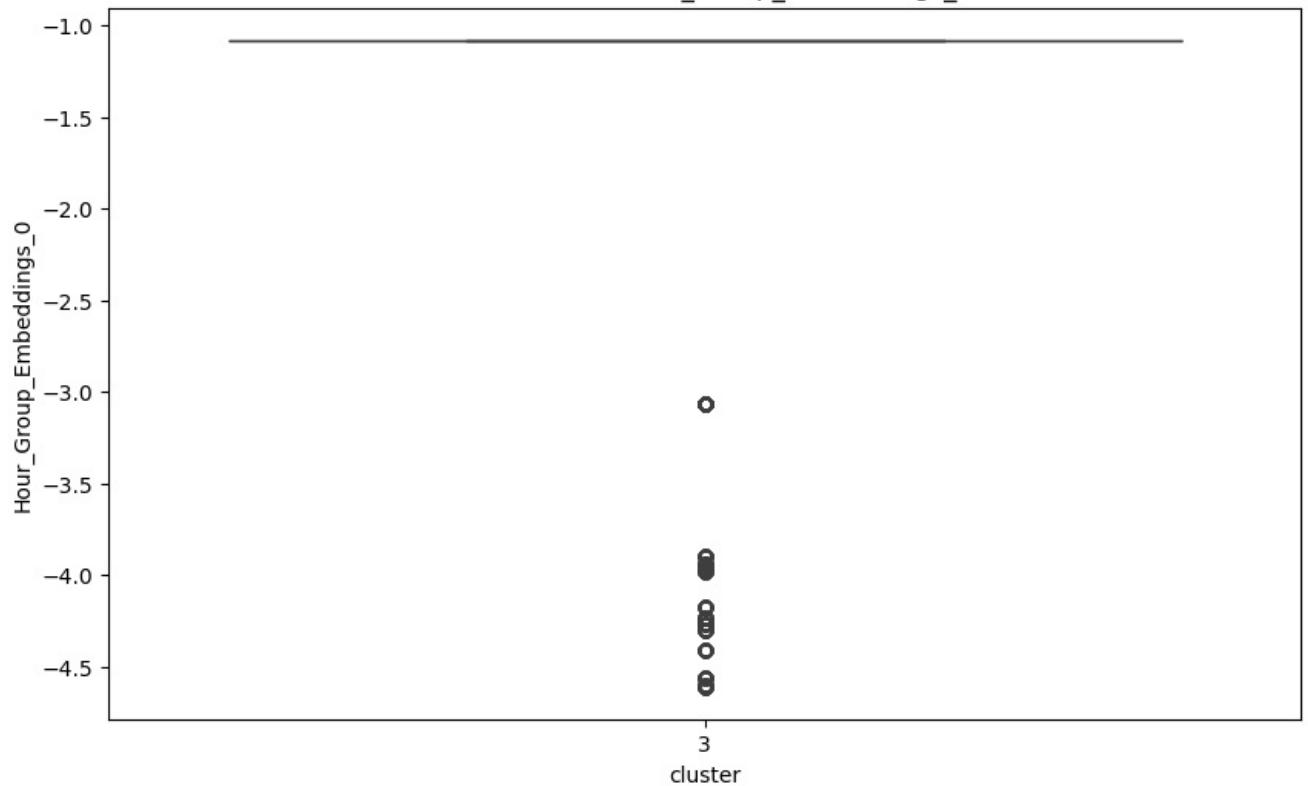
#Bias Analysis: Visualize key features across clusters
for pc, features in top_features_per_pc.items():
    for feature in features.index:
        #Validation Set visualization
        plt.figure(figsize=(10, 6))
        sns.boxplot(x='cluster', y=feature, data=X_val_scaled_with_clusters)
        plt.title(f'Validation Set: Distribution of {feature} across Clusters')
        plt.show()

#Test Set visualization
plt.figure(figsize=(10, 6))
sns.boxplot(x='cluster', y=feature, data=X_test_scaled_with_clusters)
plt.title(f'Test Set: Distribution of {feature} across Clusters')
plt.show()

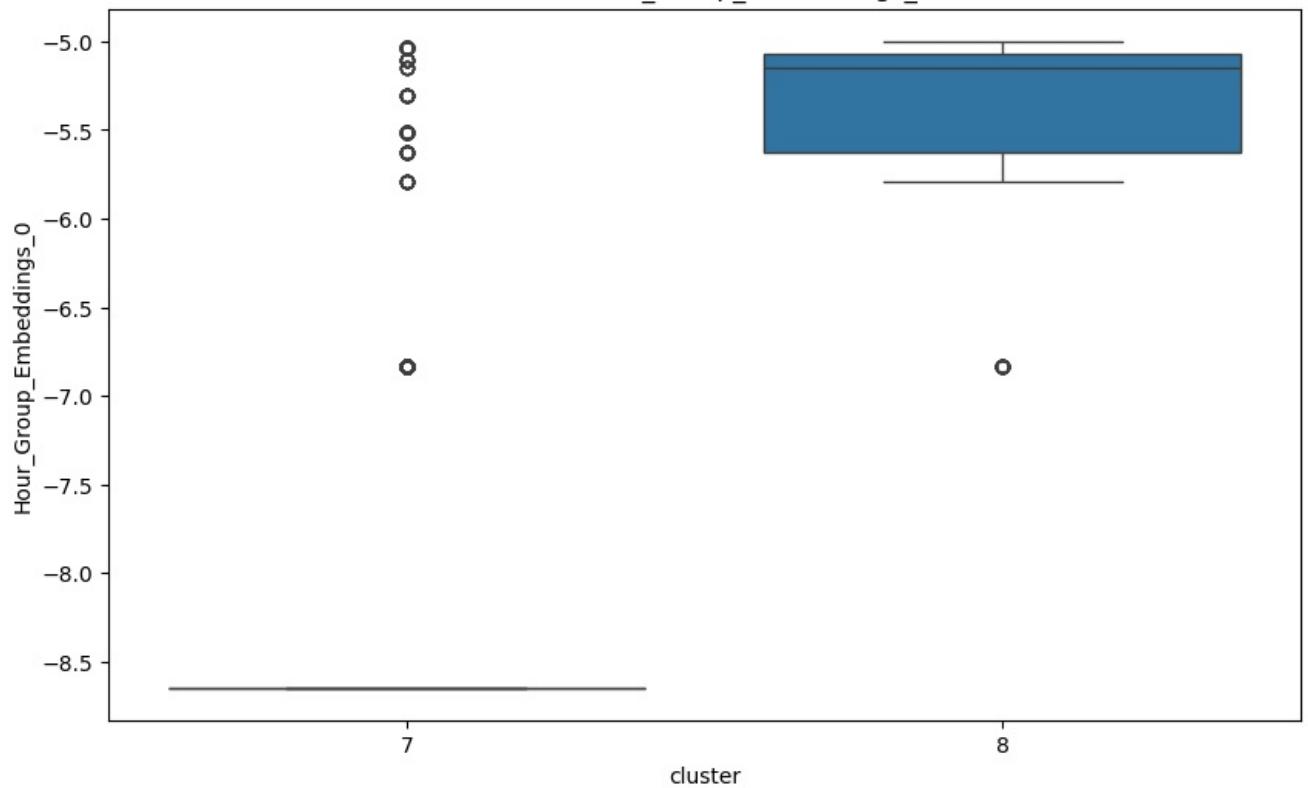
```



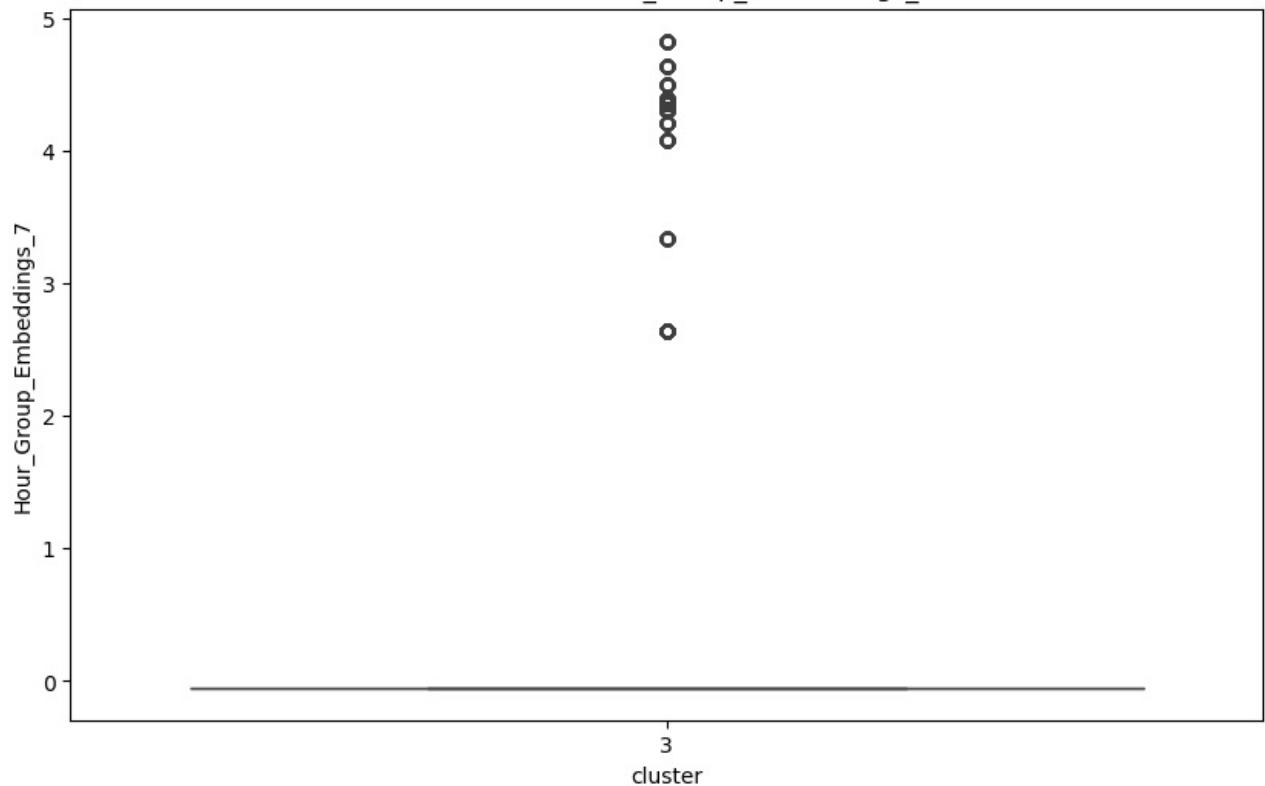
Validation Set: Distribution of Hour_Group_EMBEDDINGS_0 across Clusters



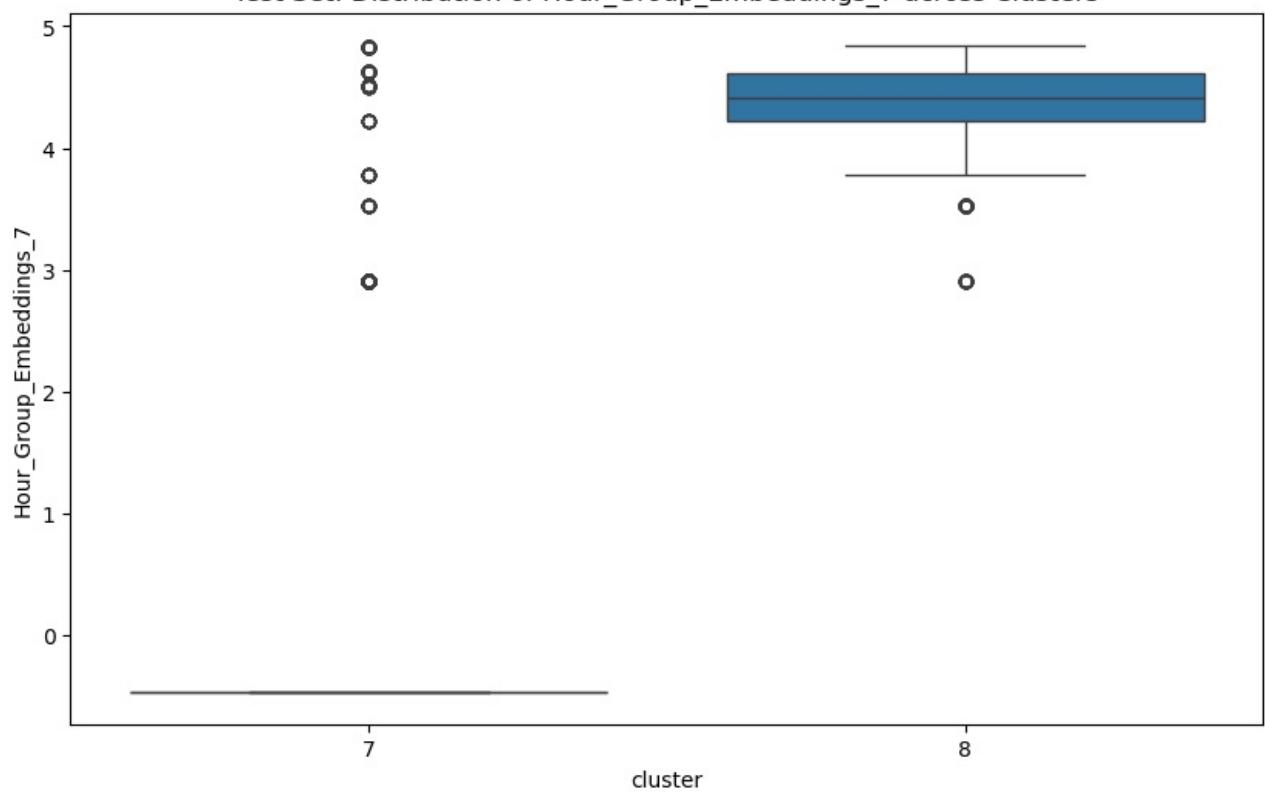
Test Set: Distribution of Hour_Group_EMBEDDINGS_0 across Clusters



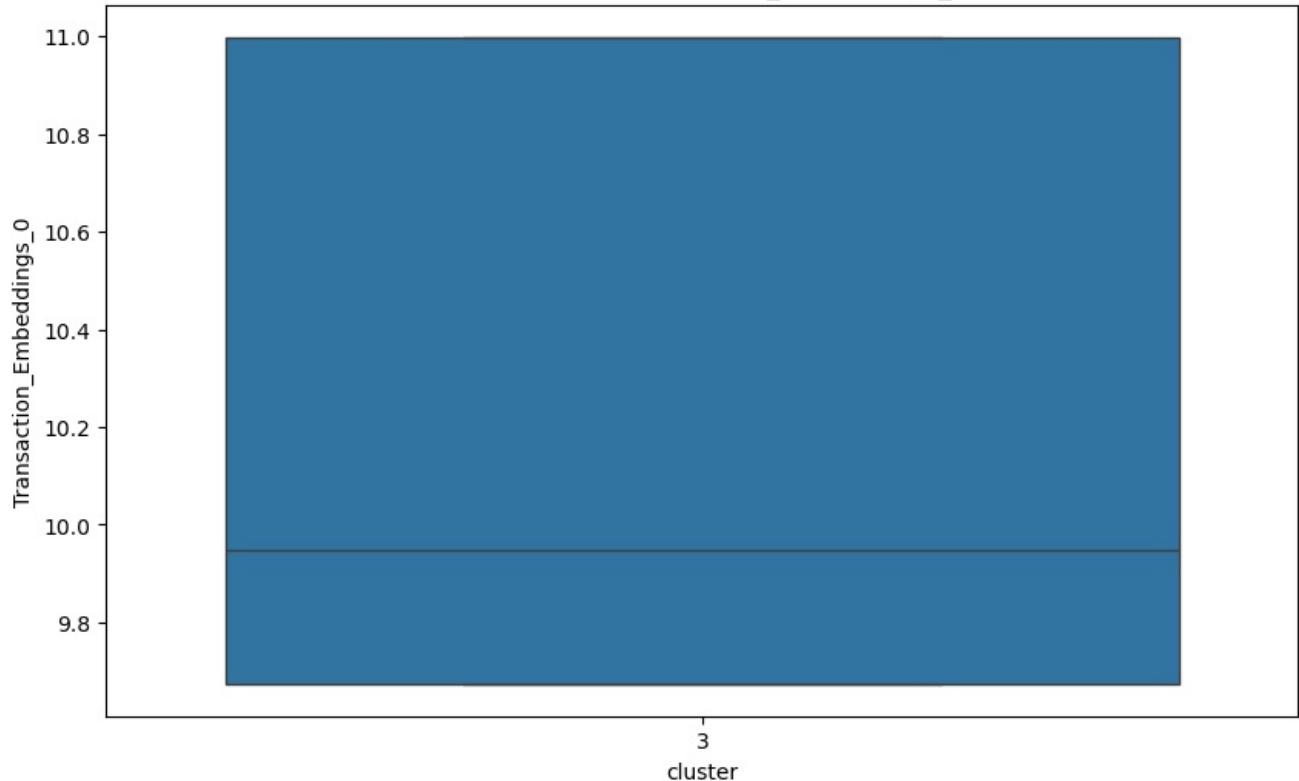
Validation Set: Distribution of Hour_Group_EMBEDDINGS_7 across Clusters



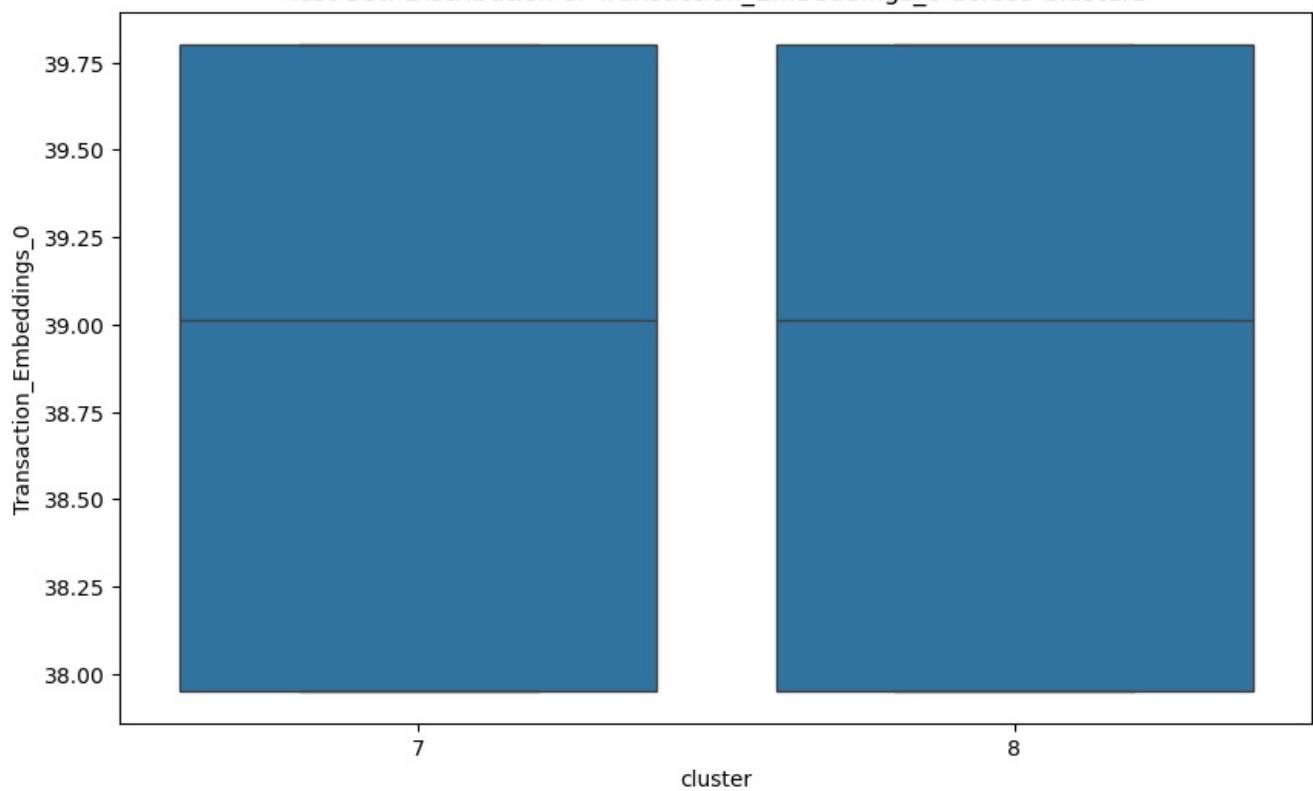
Test Set: Distribution of Hour_Group_EMBEDDINGS_7 across Clusters



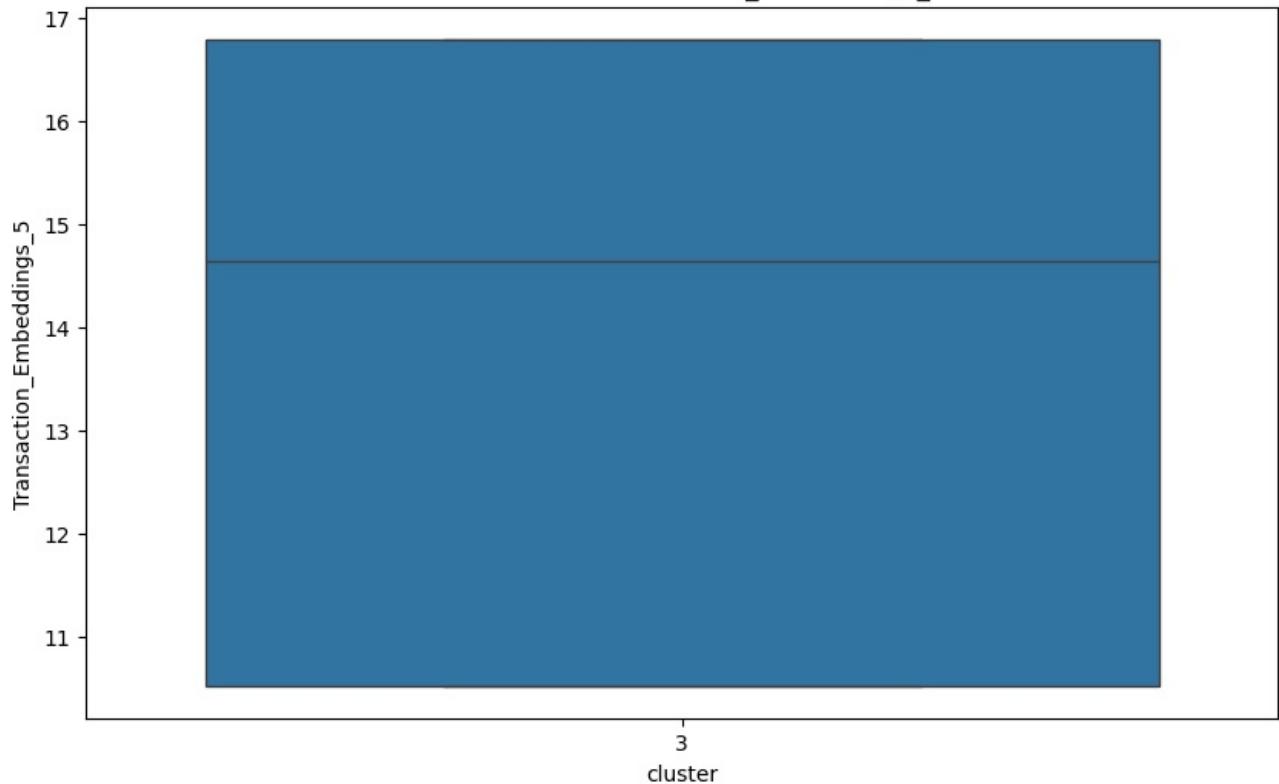
Validation Set: Distribution of Transaction_EMBEDDINGS_0 across Clusters



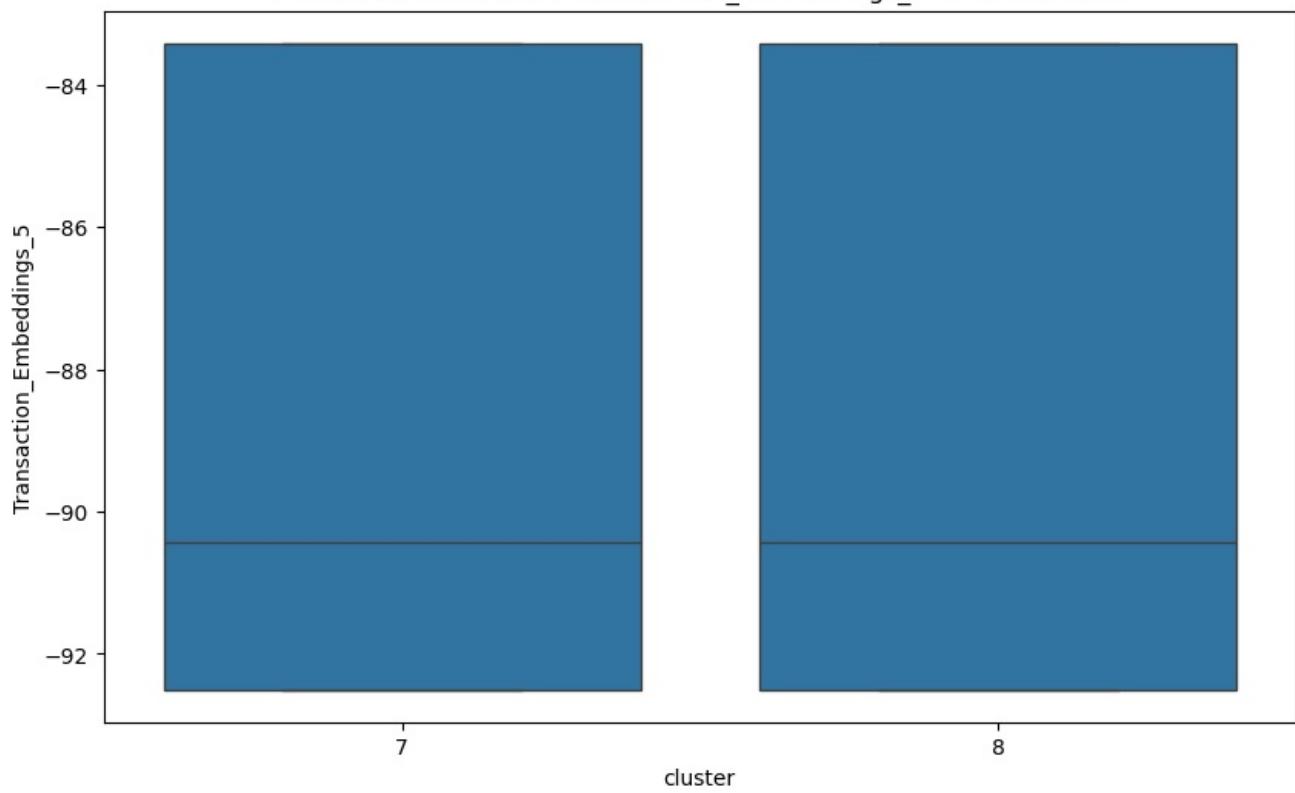
Test Set: Distribution of Transaction_EMBEDDINGS_0 across Clusters



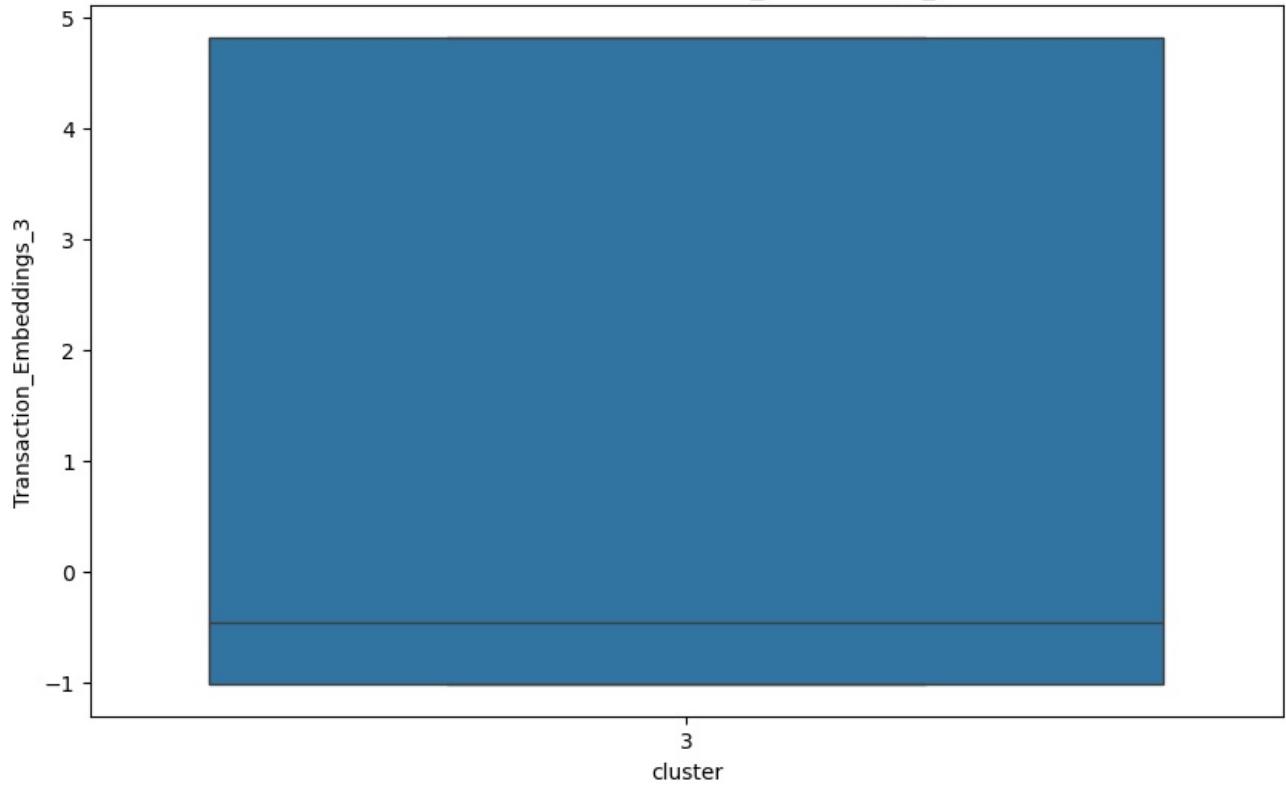
Validation Set: Distribution of Transaction_EMBEDDINGS_5 across Clusters



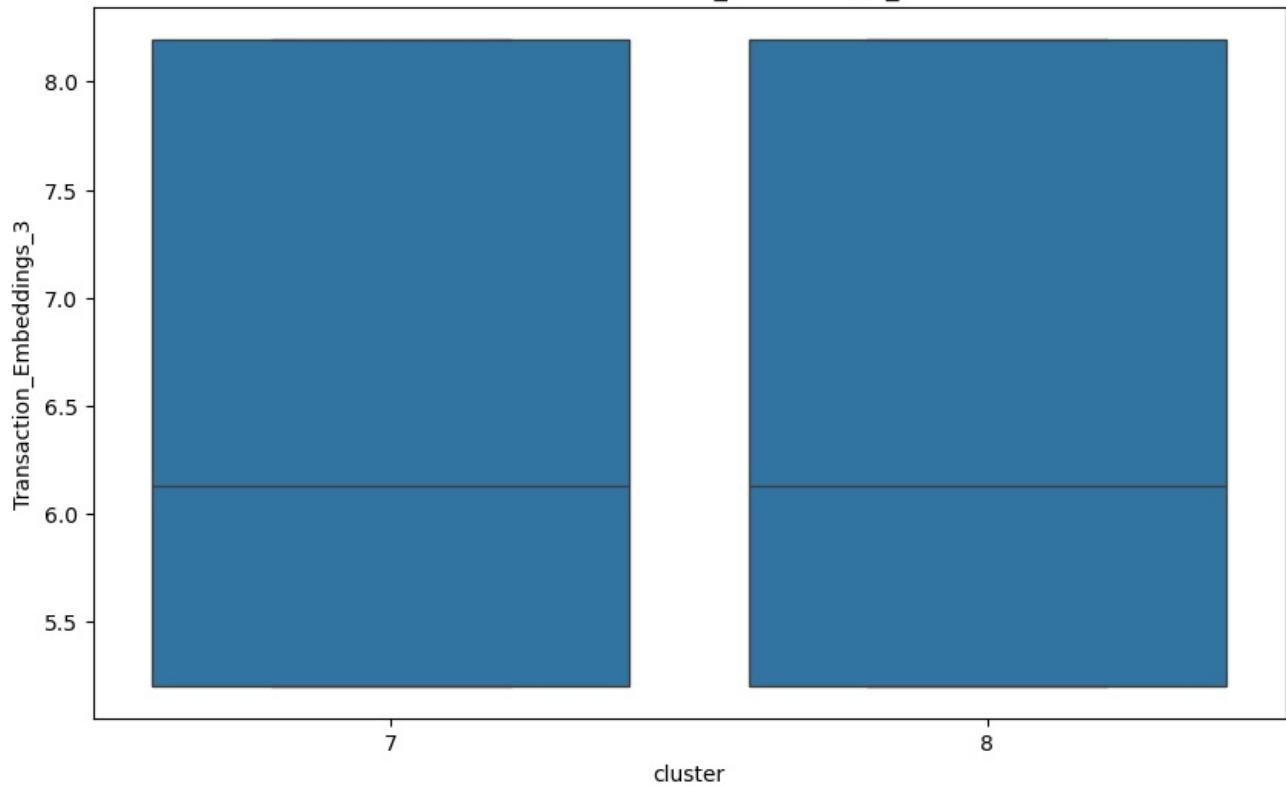
Test Set: Distribution of Transaction_EMBEDDINGS_5 across Clusters



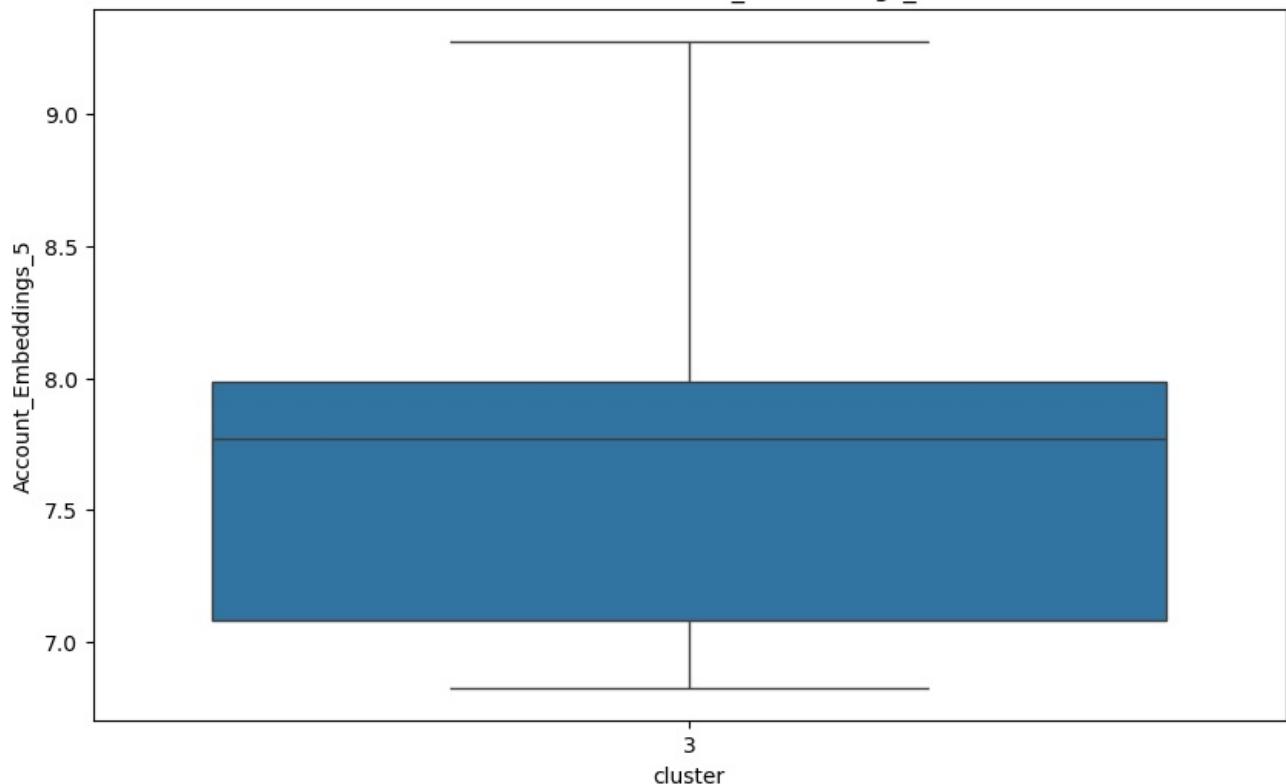
Validation Set: Distribution of Transaction_EMBEDDINGS_3 across Clusters



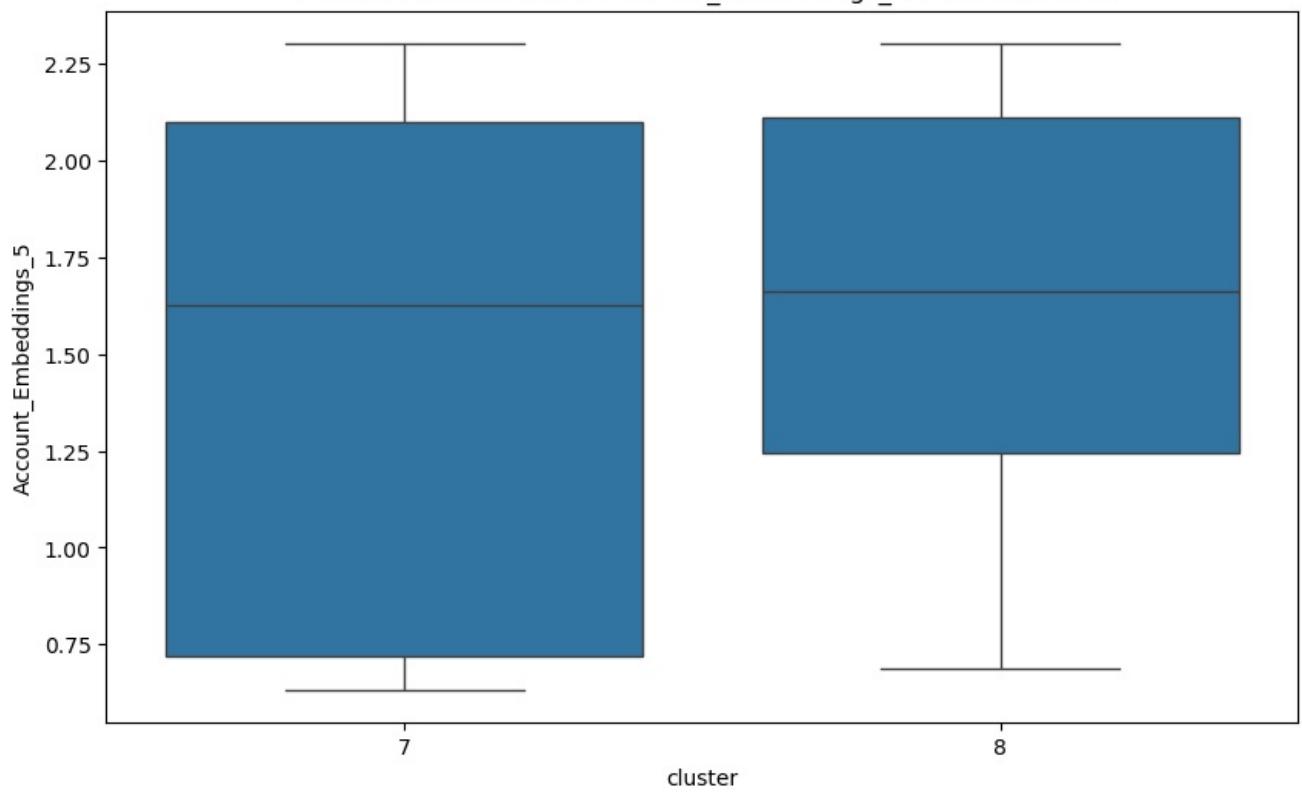
Test Set: Distribution of Transaction_EMBEDDINGS_3 across Clusters



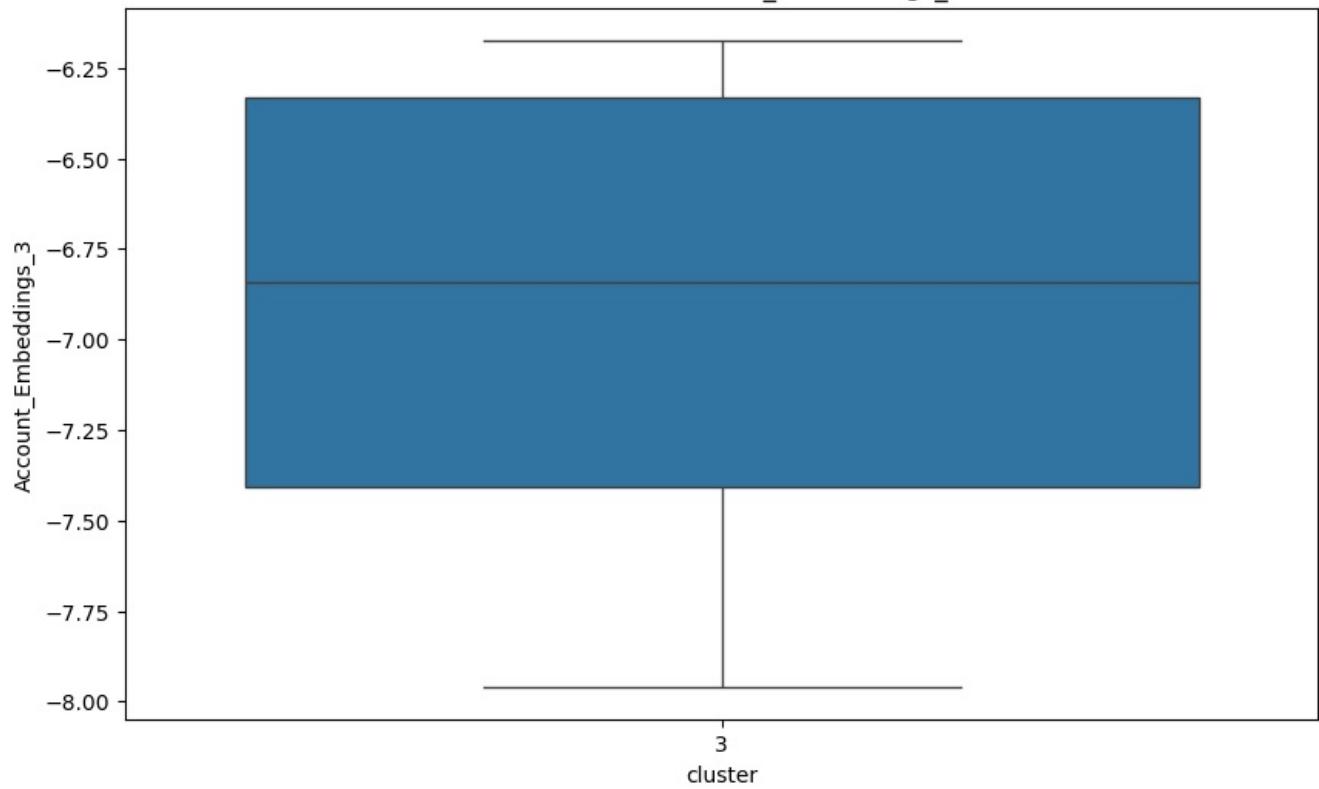
Validation Set: Distribution of Account_EMBEDDINGS_5 across Clusters



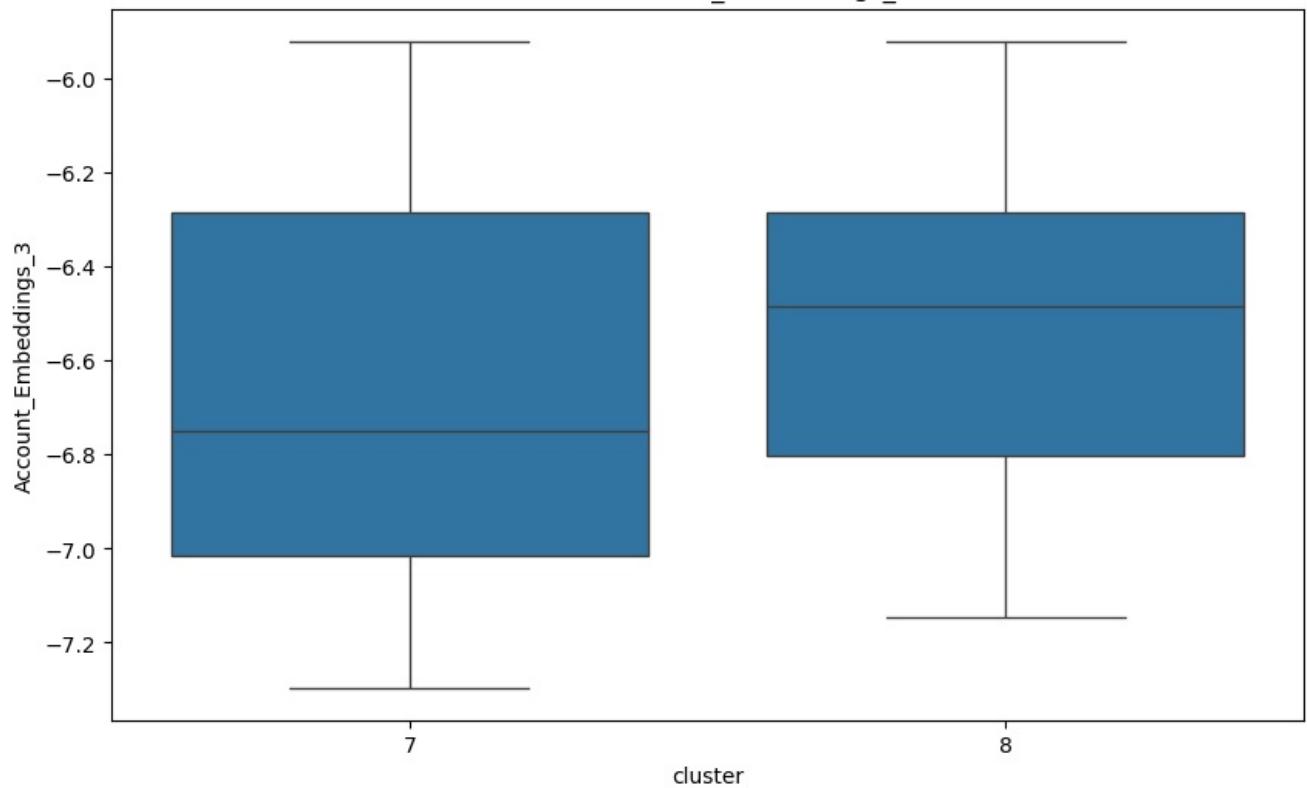
Test Set: Distribution of Account_EMBEDDINGS_5 across Clusters



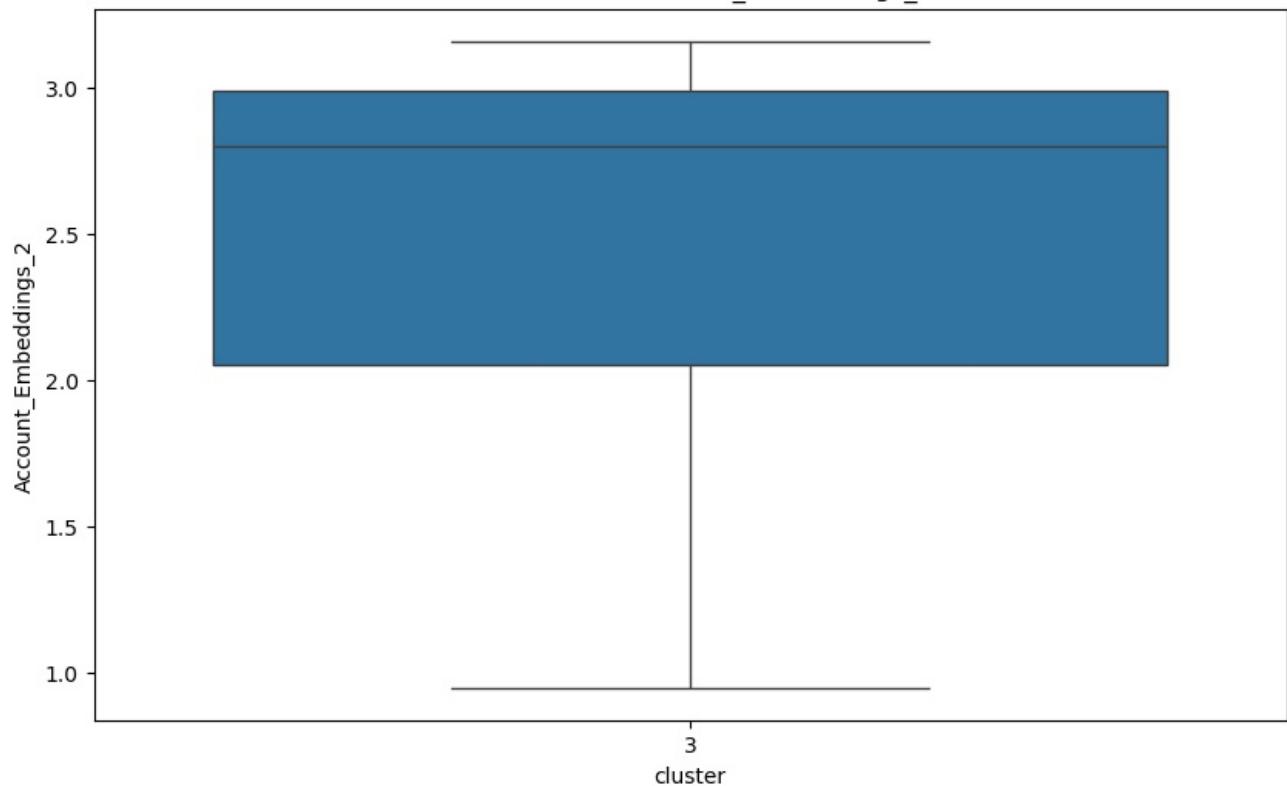
Validation Set: Distribution of Account_EMBEDDINGS_3 across Clusters



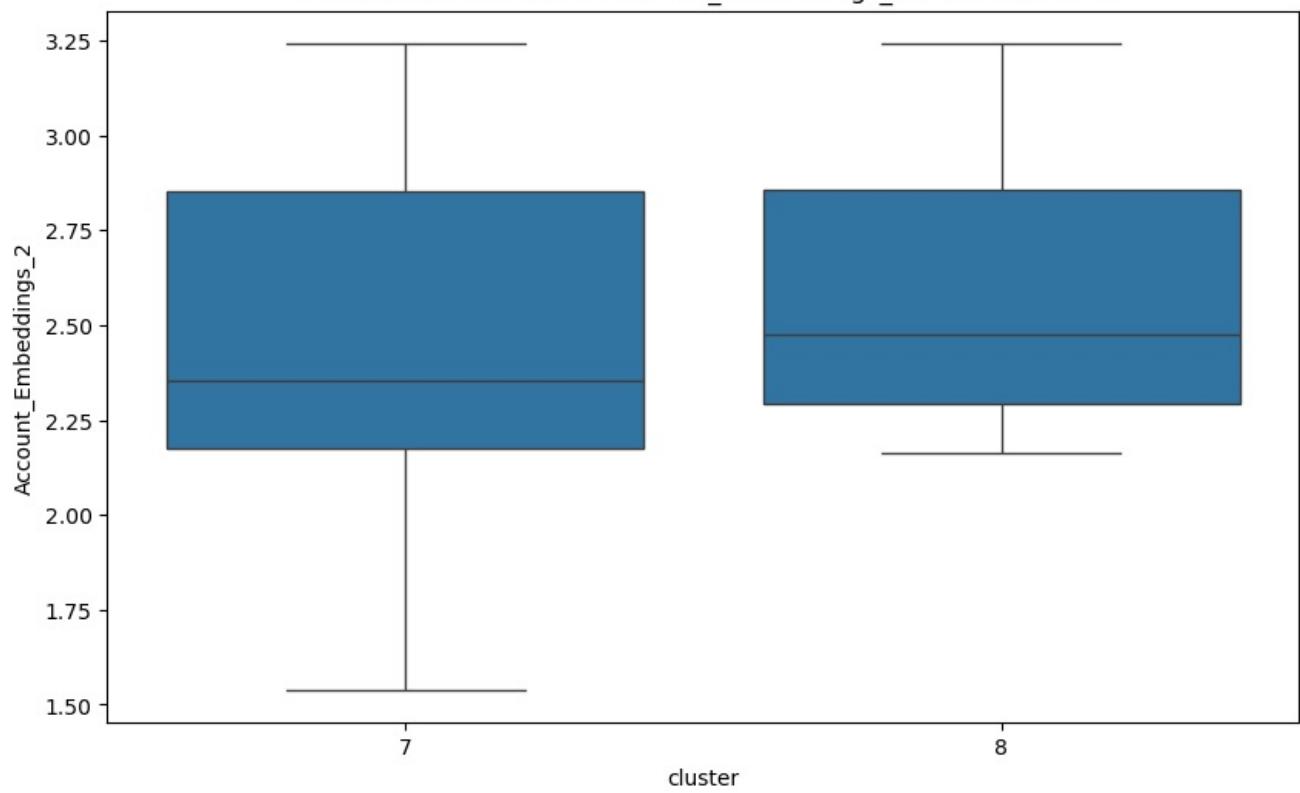
Test Set: Distribution of Account_EMBEDDINGS_3 across Clusters



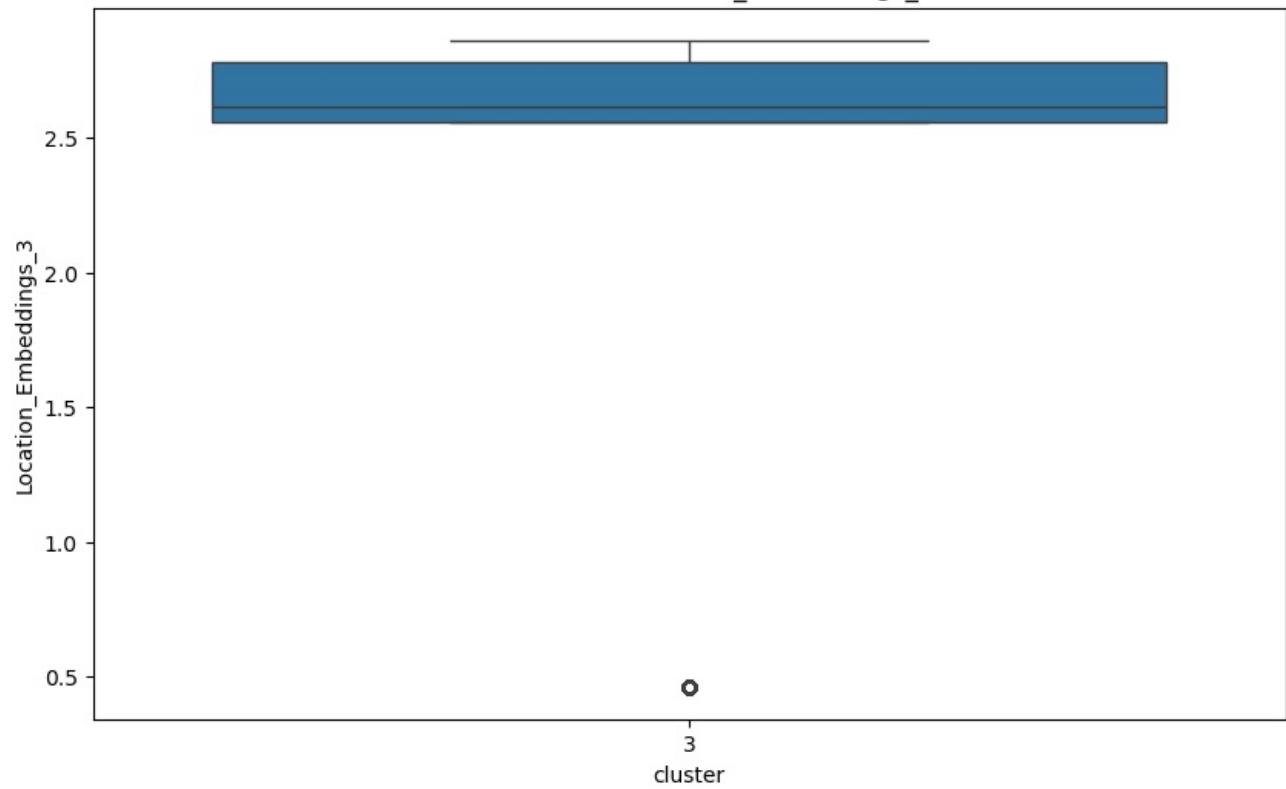
Validation Set: Distribution of Account_EMBEDDINGS_2 across Clusters



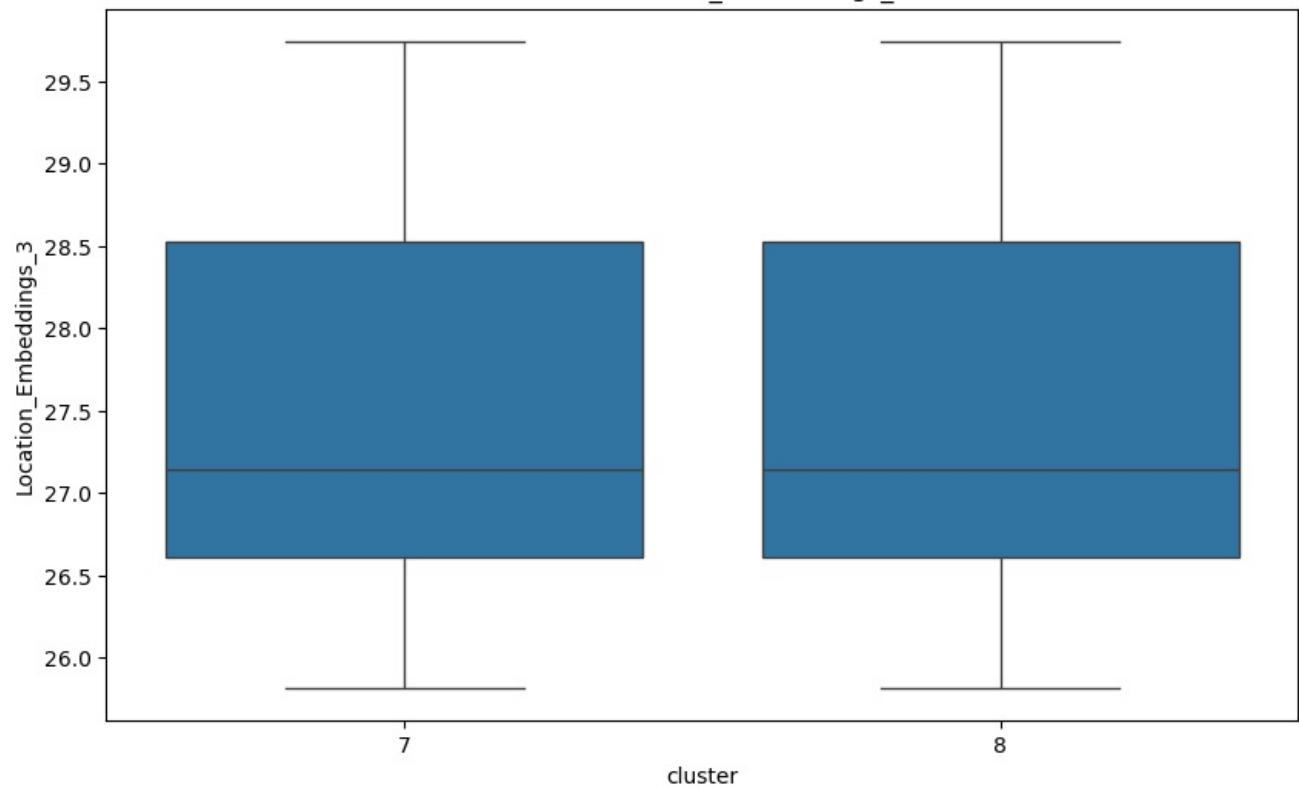
Test Set: Distribution of Account_EMBEDDINGS_2 across Clusters

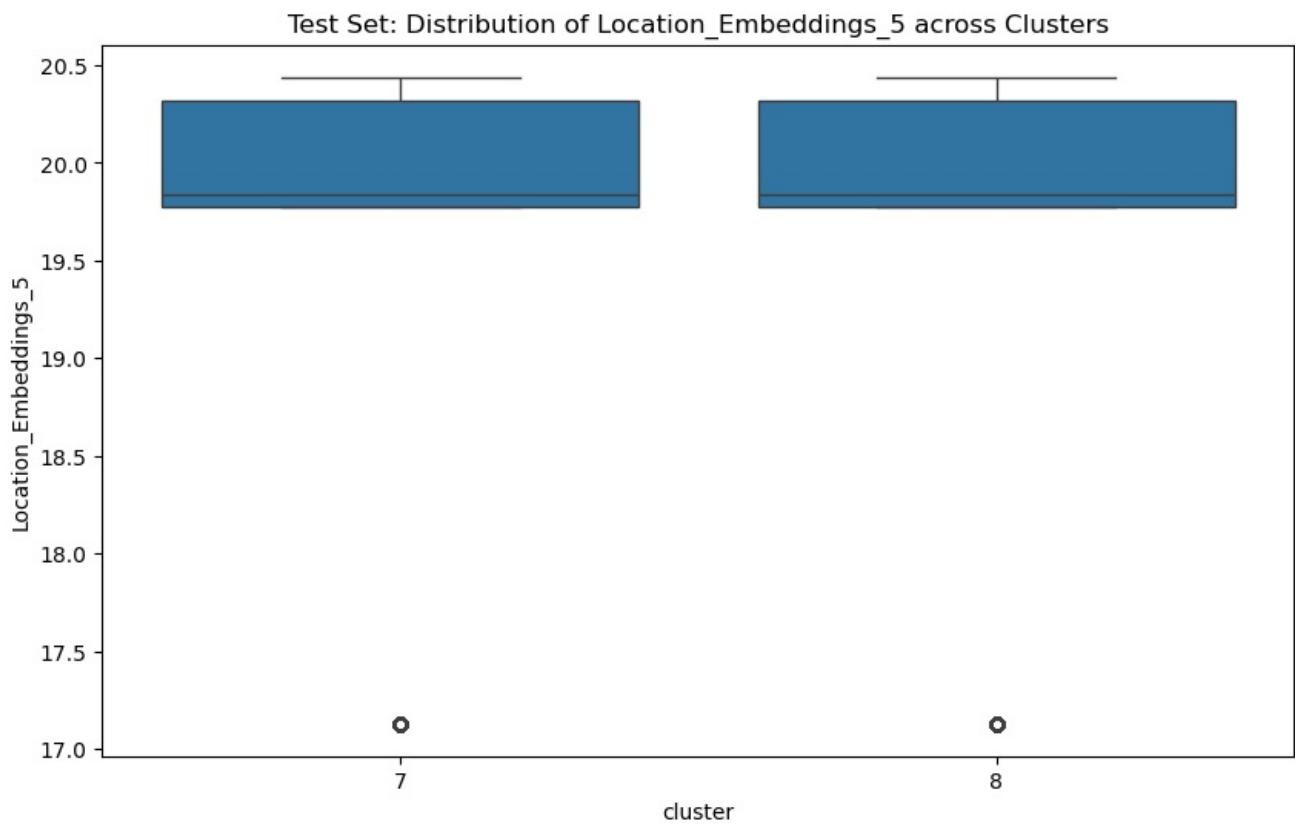
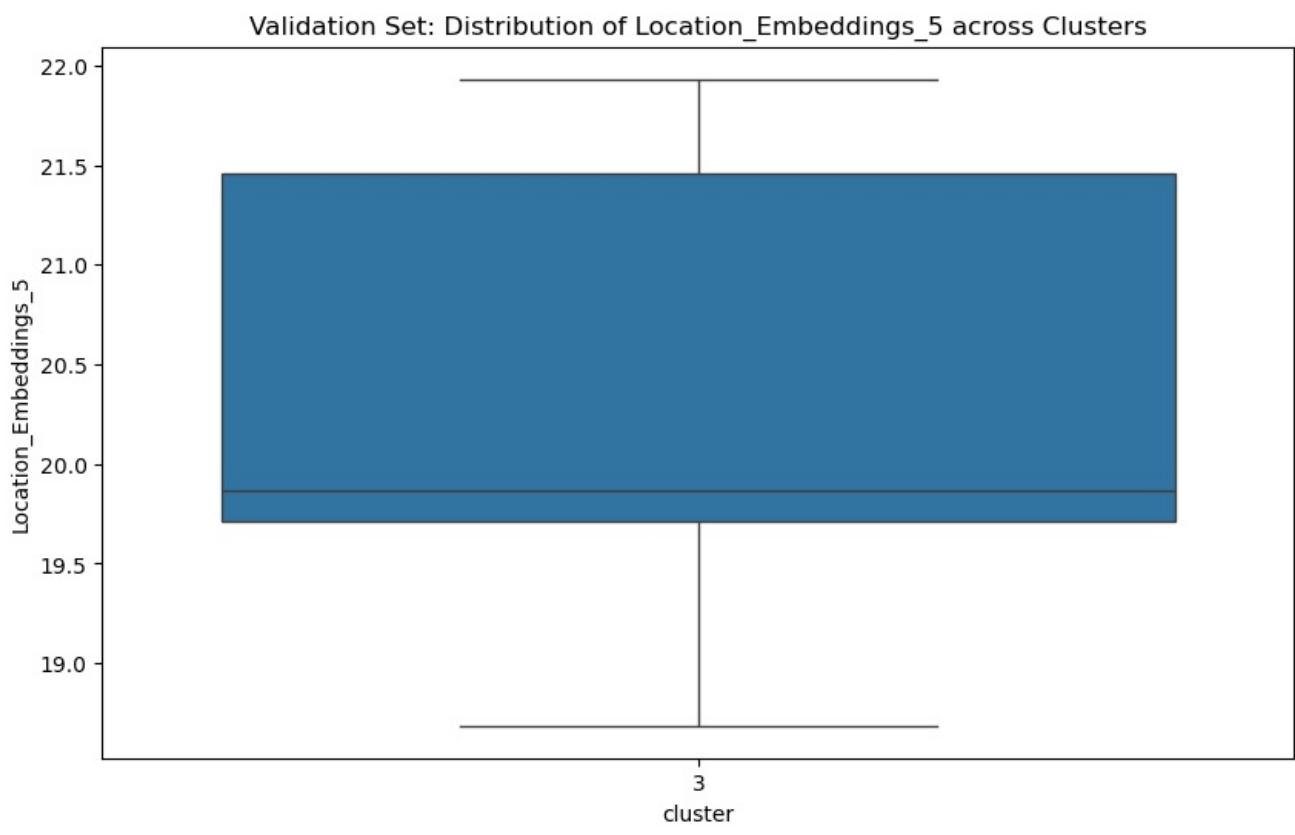


Validation Set: Distribution of Location_EMBEDDINGS_3 across Clusters

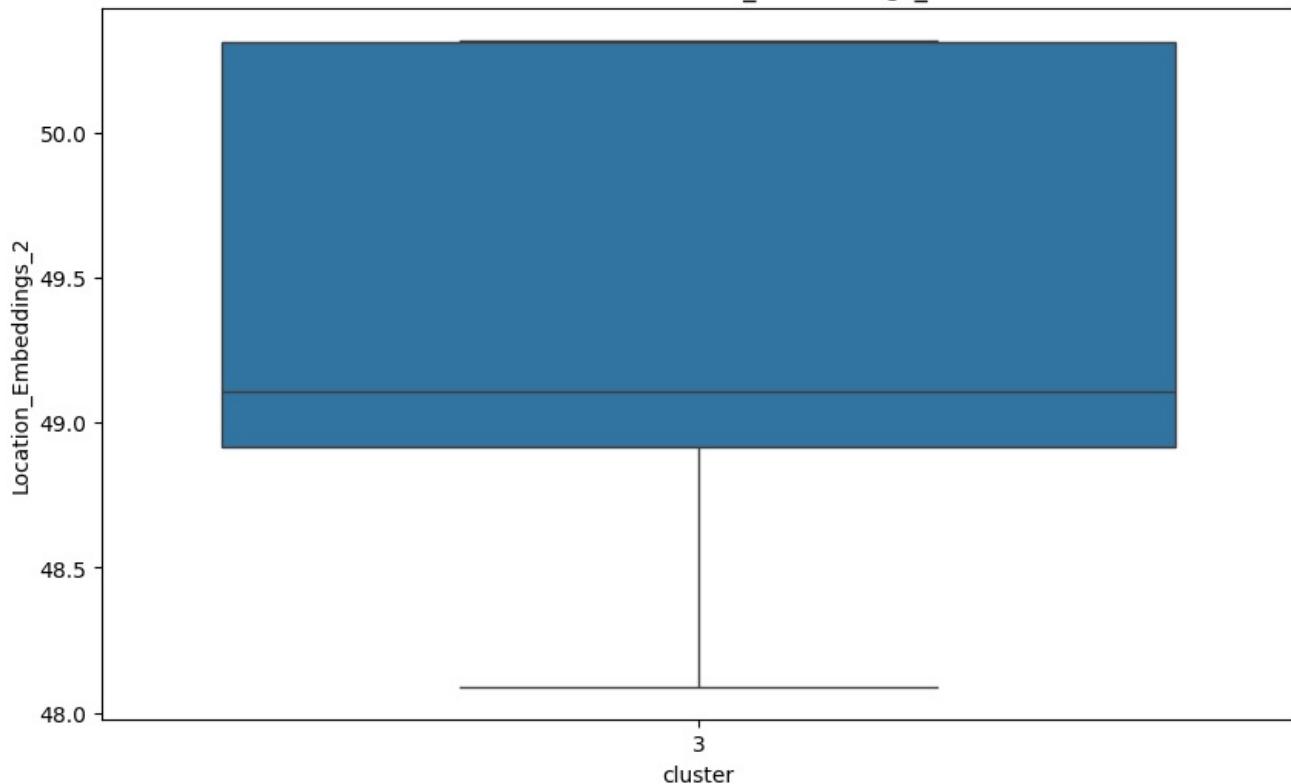


Test Set: Distribution of Location_EMBEDDINGS_3 across Clusters

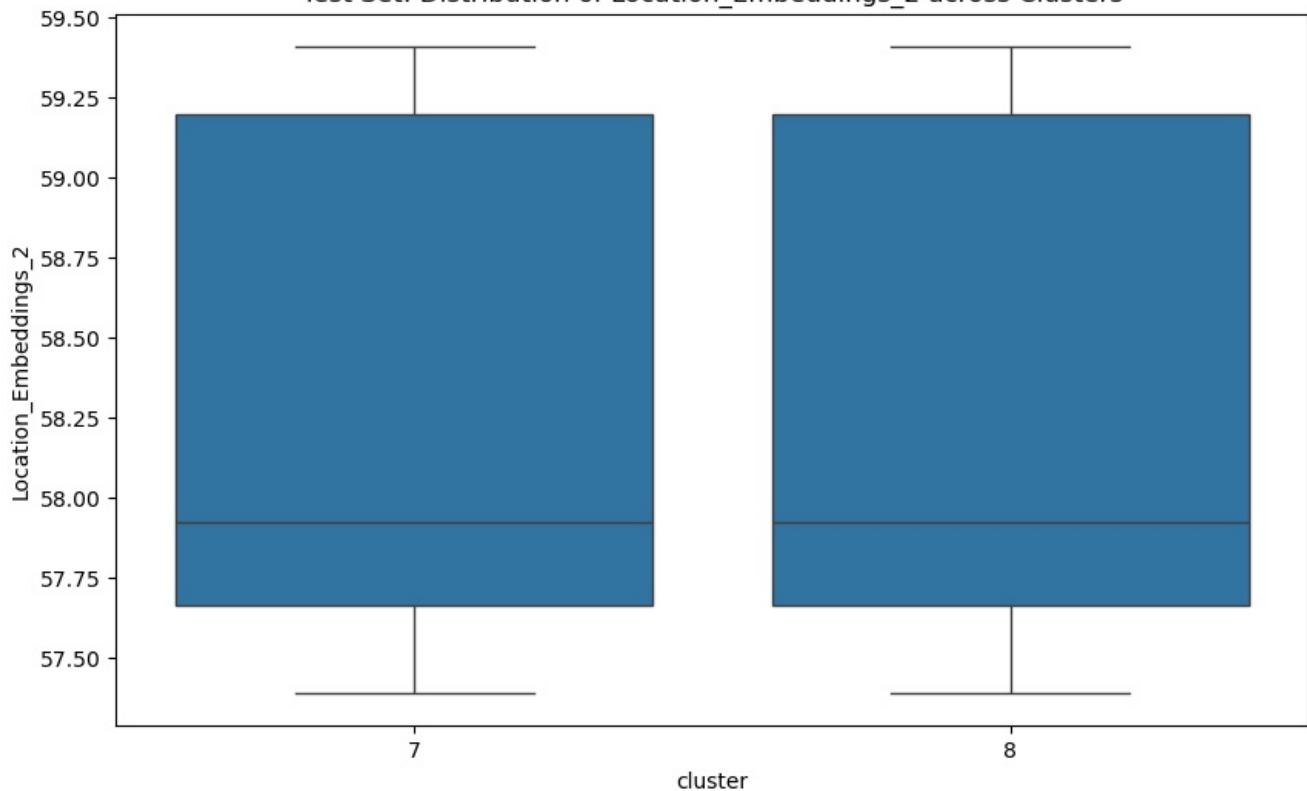




Validation Set: Distribution of Location_EMBEDDINGS_2 across Clusters



Test Set: Distribution of Location_EMBEDDINGS_2 across Clusters



```
In [128]: #MODEL SAVING FOR DEPLOYMENT
import pickle

# Fit the model to training data
kmeans.fit(X_train)

# Use the fitted model to predict cluster labels for test data
predicted_labels = kmeans.predict(X_test)

# Save model using pickle
with open('kmeans.pkl', 'wb') as f: # 'wb' = writing in binary mode
    pickle.dump(kmeans, f)

# Step 4: Load the model back from the file
with open('kmeans.pkl', 'rb') as f: # 'rb' = reading in binary mode
    loaded_model = pickle.load(f)
```

```
# Use the loaded model to predict cluster labels for the test data
predicted_labels = loaded_model.predict(X_test)

# Test the loaded model
silhouette_score = silhouette_score(X_test, predicted_labels)
print(f"Silhouette Score of the loaded model: {silhouette_score:.4f}")
```

Silhouette Score of the loaded model: 0.4712

```
In [131]: #get OS details
os_name = platform.system() # OS name (Windows, Linux, etc.)
os_version = platform.release() # OS version
os_full_version = platform.version() # Full OS version

#get Python version
python_version = sys.version

#print out all collected information
print(f"Operating System: {os_name}")
print(f"OS Version: {os_version}")
print(f"Full OS Version: {os_full_version}")
print(f"Python Version: {python_version}")

#get the list of installed packages
installed_packages = {pkg.key: pkg.version for pkg in pkg_resources.working_set}

#get the standard library modules
stdlib_modules = sys.builtin_module_names

#get only third-party packages (non-default)
third_party_packages = {pkg: version for pkg, version in installed_packages.items() if pkg not in stdlib_modules}

third_party_packages
```

Operating System: Windows
 OS Version: 11
 Full OS Version: 10.0.22631
 Python Version: 3.12.4 | packaged by Anaconda, Inc. | (main, Jun 18 2024, 15:03:56) [MSC v.1929 64 bit (AMD64)]

```
Out[131]: {'automat': '20.2.0',
 'babel': '2.11.0',
 'bottleneck': '1.3.7',
 'brotli': '1.0.9',
 'gitpython': '3.1.37',
 'heapdict': '1.0.1',
 'markdown': '3.4.1',
 'markupsafe': '2.1.3',
 'protego': '0.1.16',
 'pydispatcher': '2.0.5',
 'pyjwt': '2.8.0',
 'pynacl': '1.5.0',
 'pyqt5': '5.15.10',
 'pyqt5-sip': '12.13.0',
 'pyqtwebengine': '5.15.6',
 'pysocks': '1.7.1',
 'pyyaml': '6.0.1',
 'pygments': '2.15.1',
 'qdarkstyle': '3.2.3',
 'qtawesome': '1.2.2',
 'qtpy': '2.4.1',
 'rtree': '1.0.1',
 'sqlalchemy': '2.0.30',
 'scrapy': '2.11.1',
 'send2trash': '1.8.2',
 'unidecode': '1.2.0',
 'absl-py': '2.1.0',
 'aiobotocore': '2.12.3',
 'aiohttp': '3.9.5',
 'aioitertools': '0.7.1',
 'aiosignal': '1.2.0',
 'alabaster': '0.7.16',
 'altair': '5.0.1',
 'anaconda-anon-usage': '0.4.4',
 'anaconda-catalogs': '0.2.0',
 'anaconda-client': '1.12.3',
 'anaconda-cloud-auth': '0.5.1',
 'anaconda-navigator': '2.6.0',
 'anaconda-project': '0.11.1',
 'annotated-types': '0.6.0',
 'anyascii': '0.3.2',
 'anyio': '4.2.0',
 'appdirs': '1.4.4',
 'archspec': '0.2.3',
 'argon2-cffi': '21.3.0',
 'argon2-cffi-bindings': '21.2.0',
```

'arrow': '1.2.3',
'astroid': '2.14.2',
'astropy': '6.1.0',
'astropy-iers-data': '0.2024.6.3.0.31.14',
'asttokens': '2.0.5',
'astunparse': '1.6.3',
'async-lru': '2.0.4',
'atomicwrites': '1.4.0',
'attrs': '23.1.0',
'autopep8': '2.0.4',
'backcall': '0.2.0',
'bcrypt': '3.2.0',
'beautifulsoup4': '4.12.3',
'binaryornot': '0.4.4',
'black': '24.4.2',
'bleach': '4.1.0',
'blinker': '1.6.2',
'blis': '1.0.1',
'bokeh': '3.4.1',
'boltons': '23.0.0',
'boto3': '1.35.46',
'botocore': '1.35.46',
'cachetools': '5.3.3',
'catalogue': '2.0.10',
'certifi': '2024.8.30',
'cffi': '1.16.0',
'cfgv': '3.4.0',
'chardet': '4.0.0',
'charset-normalizer': '2.0.4',
'click': '8.1.7',
'cloudpathlib': '0.19.0',
'cloudpickle': '2.2.1',
'colorama': '0.4.6',
'colorcet': '3.1.0',
'comm': '0.2.1',
'conda': '24.7.1',
'conda-build': '24.5.1',
'conda-content-trust': '0.2.0',
'conda-index': '0.5.0',
'conda-libmamba-solver': '24.1.0',
'conda-pack': '0.7.1',
'conda-package-handling': '2.3.0',
'conda-package-streaming': '0.10.0',
'conda-repo-cli': '1.0.88',
'conda-token': '0.5.0+1.g2209e04',
'confection': '0.1.5',
'constantly': '23.10.4',
'contourpy': '1.2.0',
'contractions': '0.1.73',
'cookiecutter': '2.6.0',
'cryptography': '42.0.5',
'cssselect': '1.2.0',
'cycler': '0.11.0',
'cymem': '2.0.8',
'cytoolz': '0.12.2',
'dask': '2024.5.0',
'dask-expr': '1.1.0',
'datashader': '0.16.2',
'debugpy': '1.6.7',
'decorator': '5.1.1',
'deepdiff': '8.0.1',
'defusedxml': '0.7.1',
'diff-match-patch': '20200713',
'dill': '0.3.8',
'distlib': '0.3.8',
'distributed': '2024.5.0',
'distro': '1.9.0',
'docopt': '0.6.2',
'docstring-to-markdown': '0.11',
'docutils': '0.18.1',
'einops': '0.8.0',
'en-core-web-sm': '3.8.0',
'entrypoints': '0.4',
'et-xmlfile': '1.1.0',
'executing': '0.8.3',
'fastcore': '1.7.10',
'fastjsonschema': '2.16.2',
'filelock': '3.13.1',
'filetype': '1.2.0',
'fire': '0.7.0',
'flake8': '7.0.0',
'flask': '3.0.3',
'flatbuffers': '24.3.25',

```
'fonttools': '4.51.0',
'frozendict': '2.4.2',
'frozenlist': '1.4.0',
'fsspec': '2024.3.1',
'fuzzywuzzy': '0.18.0',
'gast': '0.6.0',
'gensim': '4.3.3',
'gitdb': '4.0.7',
'google-api-core': '2.21.0',
'google-auth': '2.35.0',
'google-cloud-core': '2.4.1',
'google-cloud-storage': '2.18.2',
'google-crc32c': '1.6.0',
'google-pasta': '0.2.0',
'google-resumable-media': '2.7.2',
'googleapis-common-protos': '1.65.0',
'greenlet': '3.0.1',
'grpcio': '1.66.2',
'h5py': '3.11.0',
'hdbscan': '0.8.39',
'holoviews': '1.19.0',
'huggingface-hub': '0.26.0',
'hvplot': '0.10.0',
'hyperlink': '21.0.0',
'icecream': '2.1.3',
'identify': '2.6.1',
'idna': '3.7',
'imagecodecs': '2023.1.23',
'imageio': '2.33.1',
'imagesize': '1.4.1',
'imbalanced-learn': '0.12.3',
'imgaug': '0.4.0',
'importlib-metadata': '7.0.1',
'incremental': '22.10.0',
'inflection': '0.5.1',
'iniconfig': '1.1.1',
'intake': '0.7.0',
'intervaltree': '3.1.0',
'ipykernel': '6.28.0',
'ipython': '8.17.1',
'ipython-genutils': '0.2.0',
'ipywidgets': '7.8.1',
'isort': '5.13.2',
'itemadapter': '0.3.0',
'itemloaders': '1.1.0',
'itsdangerous': '2.2.0',
'jaraco.classes': '3.2.1',
'jedi': '0.18.1',
'jellyfish': '1.0.1',
'jinja2': '3.1.4',
'jmespath': '1.0.1',
'joblib': '1.4.2',
'json5': '0.9.6',
'jsonlines': '4.0.0',
'jsonpatch': '1.33',
'jsonpointer': '2.1',
'jsonschema': '4.19.2',
'jsonschema-specifications': '2023.7.1',
'jupyter': '1.0.0',
'jupyter-client': '8.6.0',
'jupyter-console': '6.6.3',
'jupyter-core': '5.7.2',
'jupyter-events': '0.10.0',
'jupyter-lsp': '2.2.0',
'jupyter-server': '2.14.1',
'jupyter-server-terminals': '0.4.4',
'jupyterlab': '4.0.11',
'jupyterlab-pygments': '0.1.2',
'jupyterlab-server': '2.25.1',
'jupyterlab-widgets': '1.0.0',
'keras': '3.6.0',
'keyring': '24.3.1',
'kiwisolver': '1.4.4',
'langcodes': '3.4.0',
'language-data': '1.2.0',
'lazy-loader': '0.4',
'lazy-object-proxy': '1.10.0',
'lckr-jupyterlab-variableinspector': '3.1.0',
'levenshtein': '0.26.0',
'libarchive-c': '2.9',
'libclang': '18.1.1',
'libmambapy': '1.5.8',
'linkify-it-py': '2.0.0',
```

'llvmlite': '0.42.0',
'lmdb': '1.4.1',
'locket': '1.0.0',
'loguru': '0.7.2',
'lxml': '5.2.1',
'lz4': '4.3.2',
'marisa-trie': '1.2.0',
'markdown-it-py': '2.2.0',
'matplotlib': '3.8.4',
'matplotlib-inline': '0.1.6',
'mccabe': '0.7.0',
'mdit-py-plugins': '0.3.0',
'mdurl': '0.1.0',
'menuinst': '2.1.1',
'mergedeep': '1.3.4',
'mistune': '2.0.4',
'mkl-fft': '1.3.8',
'mkl-random': '1.2.4',
'mkl-service': '2.4.0',
'ml-dtypes': '0.4.1',
'more-itertools': '10.1.0',
'mpmath': '1.3.0',
'mrjob': '0.7.4',
'msgpack': '1.0.3',
'multidict': '6.0.4',
'multipledispatch': '0.6.0',
'murmurhash': '1.0.10',
'mypypy': '1.10.0',
'mypypy-extensions': '1.0.0',
'namex': '0.0.8',
'navigator-updater': '0.5.1',
'nbclient': '0.8.0',
'nbconvert': '7.16.4',
'nbformat': '5.9.2',
'nest-asyncio': '1.6.0',
'networkx': '3.2.1',
'nltk': '3.9.1',
'nodeenv': '1.9.1',
'notebook': '7.0.8',
'notebook-shim': '0.2.3',
'numba': '0.59.1',
'numexpr': '2.8.7',
'numpy': '1.26.4',
'numpydoc': '1.7.0',
'opencv-python': '4.9.0.80',
'opencv-python-headless': '4.10.0.84',
'openpyxl': '3.1.2',
'opt-einsum': '3.4.0',
'optree': '0.13.0',
'orderly-set': '5.2.2',
'overrides': '7.4.0',
'packaging': '23.2',
'pandas': '2.2.2',
'pandas-datareader': '0.10.0',
'pandocfilters': '1.5.0',
'panel': '1.4.4',
'param': '2.1.0',
'paramiko': '2.8.1',
'parsel': '1.8.1',
'parso': '0.8.3',
'partd': '1.4.1',
'pathspec': '0.10.3',
'patsy': '0.5.6',
'pexpect': '4.8.0',
'pickleshare': '0.7.5',
'pillow': '10.3.0',
'pip': '24.3.1',
'pipreqs': '0.5.0',
'pkce': '1.0.3',
'pkginfo': '1.10.0',
'platformdirs': '3.10.0',
'plotly': '5.22.0',
'pluggy': '1.0.0',
'ply': '3.11',
'pre-commit': '3.8.0',
'preshed': '3.0.9',
'prometheus-client': '0.14.1',
'prompt-toolkit': '3.0.43',
'proto-plus': '1.24.0',
'protobuf': '3.20.3',
'psutil': '5.9.0',
'psycopg2': '2.9.9',
'ptyprocess': '0.7.0',

'pure-eval': '0.2.2',
'pyopenssl': '24.0.0',
'py-cpuinfo': '9.0.0',
'pyahocorasick': '2.1.0',
'pyarrow': '14.0.2',
'pyasn1': '0.4.8',
'pyasn1-modules': '0.2.8',
'pybbox': '0.1.6',
'pycodestyle': '2.11.1',
'pycosat': '0.6.6',
'pycparser': '2.21',
'pyct': '0.5.0',
'pycurl': '7.45.2',
'pydantic': '2.9.2',
'pydantic-core': '2.23.4',
'pydeck': '0.8.0',
'pydocstyle': '6.3.0',
'pyerfa': '2.0.1.4',
'pyflakes': '3.2.0',
'pylint': '2.16.2',
'pylint-venv': '3.0.3',
'pyls-spyder': '0.4.0',
'pyodbc': '5.0.1',
'pyparsing': '3.0.9',
'pytest': '7.4.4',
'python-levenshtein': '0.26.0',
'python-dateutil': '2.9.0.post0',
'python-dotenv': '0.21.0',
'python-json-logger': '2.0.7',
'python-lsp-black': '2.0.0',
'python-lsp-jsonrpc': '1.1.2',
'python-lsp-server': '1.10.0',
'python-slugify': '5.0.2',
'python-snappy': '0.6.1',
'pytoolconfig': '1.2.6',
'pytz': '2024.1',
'pyviz-comms': '3.0.2',
'pywavelets': '1.5.0',
'pywin32': '305.1',
'pywin32-ctypes': '0.2.2',
'pywinpty': '2.0.10',
'pyzmq': '25.1.2',
'qstylizer': '0.2.2',
'qtconsole': '5.5.1',
'queuelib': '1.6.2',
'rake-nltk': '1.0.6',
'rapidfuzz': '3.10.0',
'referencing': '0.30.2',
'regex': '2023.10.3',
'requests': '2.32.2',
'requests-file': '1.5.1',
'requests-toolbelt': '1.0.0',
'rfc3339-validator': '0.1.4',
'rfc3986-validator': '0.1.1',
'rich': '13.3.5',
'roboflow': '1.1.48',
'rope': '1.12.0',
'rpds-py': '0.10.6',
'rsa': '4.9',
'ruamel.yaml': '0.17.21',
'ruamel.yaml-conda': '0.17.21',
's3fs': '2024.3.1',
's3transfer': '0.10.3',
'safetensors': '0.4.5',
'sahi': '0.11.18',
'scikit-image': '0.23.2',
'scikit-learn': '1.5.2',
'scikit-surprise': '1.1.4',
'scipy': '1.13.1',
'seaborn': '0.13.2',
'semver': '3.0.2',
'service-identity': '18.1.0',
'setuptools': '75.3.0',
'shapely': '2.0.6',
'shellingham': '1.5.4',
'sip': '6.7.12',
'six': '1.16.0',
'smart-open': '5.2.1',
'smmap': '4.0.0',
'sniffio': '1.3.0',
'snowballstemmer': '2.2.0',
'sortedcontainers': '2.4.0',
'soupsieve': '2.5',

'spacy': '3.8.2',
'spacy-legacy': '3.0.12',
'spacy-loggers': '1.0.5',
'sphinx': '7.3.7',
'sphinxcontrib-applehelp': '1.0.2',
'sphinxcontrib-devhelp': '1.0.2',
'sphinxcontrib-htmlhelp': '2.0.0',
'sphinxcontrib-jsmath': '1.0.1',
'sphinxcontrib-qthelp': '1.0.3',
'sphinxcontrib-serializinghtml': '1.1.10',
'spyder': '5.5.1',
'spyder-kernels': '2.5.0',
'srsly': '2.4.8',
'stack-data': '0.2.0',
'statsmodels': '0.14.2',
'streamlit': '1.32.0',
'sympy': '1.12',
'tables': '3.9.2',
'tabulate': '0.9.0',
'tblib': '1.7.0',
'tenacity': '8.2.2',
'tensorboard': '2.17.1',
'tensorboard-data-server': '0.7.2',
'tensorflow': '2.17.0',
'tensorflow-intel': '2.17.0',
'termcolor': '2.5.0',
'terminado': '0.17.1',
'terminaltables': '3.1.10',
'text-unidecode': '1.3',
'textdistance': '4.2.1',
'textsearch': '0.0.24',
'thinc': '8.3.2',
'thop': '0.1.1.post2209072238',
'threadpoolctl': '3.5.0',
'three-merge': '0.1.1',
'tifffile': '2023.4.12',
'tinycc2': '1.2.1',
'tldextract': '3.2.0',
'tokenizers': '0.20.1',
'toml': '0.10.2',
'tomli': '2.0.1',
'tomlkit': '0.11.1',
'toolz': '0.12.0',
'torch': '2.4.1',
'torch-snippets': '0.545',
'torchvision': '0.19.1',
'tornado': '6.4.1',
'tqdm': '4.66.4',
'traitlets': '5.14.3',
'transformers': '4.45.2',
'truststore': '0.8.0',
'twisted': '23.10.0',
'twisted-iocpsupport': '1.0.2',
'typer': '0.12.5',
'typing': '3.7.4.3',
'typing-extensions': '4.11.0',
'tzdata': '2023.3',
'uc-micro-py': '1.0.1',
'ujson': '5.10.0',
'ultralytics': '8.3.21',
'ultralytics-thop': '2.0.9',
'unicodedata2': '15.1.0',
'urllib3': '2.2.2',
'virtualenv': '20.26.6',
'w3lib': '2.1.2',
'wasabi': '1.1.3',
'watchdog': '4.0.1',
'wcwidth': '0.2.5',
'weasel': '0.4.1',
'webencodings': '0.5.1',
'websocket-client': '1.8.0',
'werkzeug': '3.0.3',
'whatthepatch': '1.0.2',
'wheel': '0.44.0',
'widgetsnbextension': '3.6.6',
'win32-setctime': '1.1.0',
'win-inet-pton': '1.1.0',
'wordcloud': '1.9.3',
'wrapt': '1.14.1',
'xarray': '2023.6.0',
'xlwings': '0.31.4',
'xmltodict': '0.13.0',
'xyzservices': '2022.9.0',

```
'yapf': '0.40.2',
'yarg': '0.1.9',
'yarl': '1.9.3',
'yolov5': '7.0.13',
'zict': '3.0.0',
'zipp': '3.17.0',
'zope.interface': '5.4.0',
'zstandard': '0.22.0',
'autocommand': '2.2.2',
'backports.tarfile': '1.2.0',
'importlib-resources': '6.4.0',
'inflect': '7.3.1',
'jaraco.collections': '5.1.0',
'jaraco.context': '5.3.0',
'jaraco.functools': '4.0.1',
'jaraco.text': '3.12.1',
'typeguard': '4.3.0'}
```

In [132]:

```
#save final versions of datasets being used for modeling
#save the train set
train_embeddings_df.to_csv('train_data.csv', index=False)

#save the validation set
validation_embeddings_df.to_csv('validation_data.csv', index=False)

#save the test set
test_embeddings_df.to_csv('test_data.csv', index=False)

print("DataFrames have been saved as CSV files.")
```

DataFrames have been saved as CSV files.

In []: