DTSA 5510 Final Project

GitHub repo URL

https://github.com/k-khuu/DTSA-5510-Final-Project

Introduction and Problem Description

Customer attrition is a major concern in the financial industry, where retaining credit card users is essential for long-term profitability. Identifying customers who are likely to disengage allows businesses to take proactive steps to improve retention and build stronger relationships.

This project explores a dataset available on Kaggle containing information about credit card holders, including demographic details, account activity, and whether they have stopped using their cards. The objective is to uncover patterns that may indicate early signs of churn.

To tackle this, I applied two unsupervised learning techniques: **K-Means Clustering** and **Agglomerative Clustering**. These methods help group customers based on shared behaviors and characteristics, revealing potential segments with varying levels of attrition risk. I also used XGBoost as a supervised model to benchmark predictive performance and assess feature relevance.

Throughout the process, I ran multiple iterations of each model, experimenting with different feature sets and tuning parameters to see what combinations produced the most meaningful insights. It was a hands-on exploration of how clustering and classification can complement each other in understanding customer behavior

Import required modules

```
import warnings
warnings.filterwarnings("ignore", category=RuntimeWarning)  # Suppress
minor runtime warnings

!pip install gower  # Install Gower package

# Data manipulation
import pandas as pd
import numpy as np
from itertools import permutations

# Visualization
import altair as alt
import seaborn as sns
import matplotlib.pyplot as plt

# Preprocessing
from sklearn.preprocessing import LabelEncoder, StandardScaler
```

```python
from sklearn.model_selection import train_test_split

# Clustering
from sklearn.cluster import KMeans, AgglomerativeClustering
from scipy.cluster.hierarchy import dendrogram

# Tree-based models
from sklearn.tree import DecisionTreeClassifier
from sklearn import tree

# Evaluation metrics
from sklearn.metrics import (
    accuracy_score,
    balanced_accuracy_score,
    confusion_matrix,
    precision_recall_fscore_support
)

# Gradient boosting
import xgboost as xgb

# Altair configuration
alt.data_transformers.disable_max_rows()

# Distance metric for mixed data types
import gower
```

```
Requirement already satisfied: gower in
/usr/local/lib/python3.11/dist-packages (0.1.2)
Requirement already satisfied: numpy in
/usr/local/lib/python3.11/dist-packages (from gower) (1.26.4)
Requirement already satisfied: scipy in
/usr/local/lib/python3.11/dist-packages (from gower) (1.15.3)
Requirement already satisfied: mkl_fft in
/usr/local/lib/python3.11/dist-packages (from numpy->gower) (1.3.8)
Requirement already satisfied: mkl_random in
/usr/local/lib/python3.11/dist-packages (from numpy->gower) (1.2.4)
Requirement already satisfied: mkl_umath in
/usr/local/lib/python3.11/dist-packages (from numpy->gower) (0.1.1)
Requirement already satisfied: mkl in /usr/local/lib/python3.11/dist-
packages (from numpy->gower) (2025.2.0)
Requirement already satisfied: tbb4py in
/usr/local/lib/python3.11/dist-packages (from numpy->gower) (2022.2.0)
Requirement already satisfied: mkl-service in
/usr/local/lib/python3.11/dist-packages (from numpy->gower) (2.4.1)
Requirement already satisfied: intel-openmp<2026,>=2024 in
/usr/local/lib/python3.11/dist-packages (from mkl->numpy->gower)
(2024.2.0)
Requirement already satisfied: tbb==2022.* in
/usr/local/lib/python3.11/dist-packages (from mkl->numpy->gower)
```

```
(2022.2.0)
Requirement already satisfied: tcmlib==1.* in
/usr/local/lib/python3.11/dist-packages (from tbb==2022.*->mkl->numpy-
>gower) (1.4.0)
Requirement already satisfied: intel-cmplr-lib-rt in
/usr/local/lib/python3.11/dist-packages (from mkl_umath->numpy->gower)
(2024.2.0)
Requirement already satisfied: intel-cmplr-lib-ur==2024.2.0 in
/usr/local/lib/python3.11/dist-packages (from intel-
openmp<2026,>=2024->mkl->numpy->gower) (2024.2.0)
```

Exploratory Data Analysis

Kaggle link: https://www.kaggle.com/datasets/sakshigoyal7/credit-card-customers

Data Description

This is a dataset of credit card customers. It consists of over 10,000 customers mentioning their age, salary, marital_status, credit card limit, credit card category, etc. There are 23 customer features/columns.

The dataset captures a variety of customer attributes relevant to credit card usage and retention analysis. These features are organized into three key groups:

Demographic information includes:

- Age
- Gender
- Number of Dependents
- Education Level

Account-related data covers:

- Card Category
- Tenure with the Bank (Months on Book)
- Credit Limit
- Total Revolving Balance

Target label:

- A binary indicator specifying whether the customer is active or has attrited (i.e., stopped using their credit card) The dataset contains both categorical variables (such as Gender, Education Level, and Card Category) and numerical variables (such as Age, Tenure, and Credit Limit), making it suitable for mixed-type analysis in clustering, classification, and churn prediction

Load Data and Analyzing the Dataset

In this step, I load the dataset from a CSV file and perform initial preprocessing. This includes assigning appropriate categorical data types and explicitly ordering any ordinal features to ensure correct handling during analysis.

Additionally, I remove three columns: CLIENTNUM, along with two others that show little relevance to the modeling objectives and do not contribute meaningful information.

```
# Load and prepare dataset
def load_data(path):
    return pd.read_csv(path)

def assign_categories(df, categorical_map):
    for col, order in categorical_map.items():
        df[col] = pd.Categorical(df[col], categories=order, 
ordered=bool(order))
    return df

def drop_columns(df, columns):
    return df.drop(columns=columns)

# File path
csv_path = "/kaggle/input/credit-card-customers/BankChurners.csv"

# Categorical configuration
categorical_config = {
    "Education_Level": [
        "Unknown", "Uneducated", "High School", "College",
        "Graduate", "Post-Graduate", "Doctorate"
    ],
    "Income_Category": [
        "Unknown", "Less than $40K", "$40K - $60K",
        "$60K - $80K", "$80K - $120K", "$120K +"
    ],
    "Gender": None,
    "Marital_Status": None,
    "Card_Category": None,
    "Attrition_Flag": None
}

# Columns to remove
irrelevant_columns = [
    "Attrition_Flag",
    "CLIENTNUM",

"Naive_Bayes_Classifier_Attrition_Flag_Card_Category_Contacts_Count_12
_mon_Dependent_count_Education_Level_Months_Inactive_12_mon_1",

"Naive_Bayes_Classifier_Attrition_Flag_Card_Category_Contacts_Count_12
_mon_Dependent_count_Education_Level_Months_Inactive_12_mon_2"
]

# Pipeline
data_raw = load_data(csv_path)
data_prepped = assign_categories(data_raw.copy(), categorical_config)

labels = data_prepped[["Attrition_Flag"]]
label_classes = labels["Attrition_Flag"].unique()

data_final = drop_columns(data_prepped, irrelevant_columns)

# Display the first few rows of the cleaned dataset

data_final.head()
```

```
   Customer_Age Gender  Dependent_count Education_Level Marital_Status
\
0            45      M                3     High School        Married

1            49      F                5        Graduate         Single

2            51      M                3        Graduate        Married

3            40      F                4     High School        Unknown

4            40      M                3      Uneducated        Married


  Income_Category Card_Category  Months_on_book
Total_Relationship_Count  \
0     $60K - $80K          Blue              39
5
1  Less than $40K          Blue              44
6
2    $80K - $120K          Blue              36
4
3  Less than $40K          Blue              34
3
4     $60K - $80K          Blue              21
5

   Months_Inactive_12_mon  Contacts_Count_12_mon  Credit_Limit  \
0                       1                      3       12691.0
1                       1                      2        8256.0
2                       1                      0        3418.0
3                       4                      1        3313.0
4                       1                      0        4716.0

   Total_Revolving_Bal  Avg_Open_To_Buy  Total_Amt_Chng_Q4_Q1  \
0                  777          11914.0                 1.335
```

```
1                   864          7392.0                 1.541
2                     0          3418.0                 2.594
3                  2517           796.0                 1.405
4                     0          4716.0                 2.175

   Total_Trans_Amt  Total_Trans_Ct  Total_Ct_Chng_Q4_Q1
Avg_Utilization_Ratio
0             1144              42                1.625
0.061
1             1291              33                3.714
0.105
2             1887              20                2.333
0.000
3             1171              20                2.333
0.760
4              816              28                2.500
0.000
```

Initial Observations

Upon inspecting the dataset, we observe a mix of ordinal, nominal, and numerical features. The dataset contains 10,127 records, and all columns are complete—no missing values are present.

A review of the numerical variables reveals no apparent outliers or inconsistencies, suggesting that the data is clean and reliable at this stage. Therefore, no further cleaning or imputation is required before proceeding.

It's worth noting that during model development, additional preprocessing steps are applied to convert categorical features into numerical representations, enabling their use in machine learning algorithms.

```
# Generate a summary of column types, non-null counts, and unique
values
summary = pd.DataFrame({
    "Data Type": data_final.dtypes,
    "Non-Null Count": data_final.count(),
    "Unique Values": data_final.nunique()
})

# Display the summary sorted by column name
summary.sort_index()

                         Data Type  Non-Null Count  Unique Values
Avg_Open_To_Buy            float64           10127           6813
Avg_Utilization_Ratio      float64           10127            964
Card_Category             category           10127              4
Contacts_Count_12_mon        int64           10127              7
Credit_Limit               float64           10127           6205
Customer_Age                 int64           10127             45
Dependent_count              int64           10127              6
```

```
Education_Level            category        10127              7
Gender                     category        10127              2
Income_Category            category        10127              6
Marital_Status             category        10127              4
Months_Inactive_12_mon        int64        10127              7
Months_on_book                int64        10127             44
Total_Amt_Chng_Q4_Q1        float64        10127           1158
Total_Ct_Chng_Q4_Q1         float64        10127            830
Total_Relationship_Count      int64        10127              6
Total_Revolving_Bal           int64        10127           1974
Total_Trans_Amt               int64        10127           5033
Total_Trans_Ct                int64        10127            126
```

```
# Generate descriptive statistics for numerical columns only
numeric_summary = data_final.select_dtypes(include="number").agg(
    ["count", "mean", "std", "min", "median", "max"]
).transpose()

# Rename columns for clarity
numeric_summary.columns = ["Count", "Mean", "Std Dev", "Min",
"Median", "Max"]

# Display the summary
numeric_summary
```

Churn versus Existing Customers

The target variable for this classification task is the Attrition Flag, which indicates whether a customer has attrited. A quick inspection reveals that the class distribution is notably imbalanced. To address this, we apply undersampling techniques during model development to help ensure more balanced training and improve predictive performance.

```
# Bar chart showing distribution of churn labels
alt.Chart(labels).mark_bar(color="#5276A7").encode(
    x=alt.X("Attrition_Flag:N", title="Customer Status"),
    y=alt.Y("count()", title="Number of Customers")
).properties(
    width=350,
    height=300,
    title="Distribution of Churn vs Existing Customers"
)
```

```
alt.Chart(...)
```

Customer Breakdown by Gender

The gender distribution appears reasonably balanced, which is consistent with expectations for a broad customer base. There's no indication of skew or irregularity

```
# Bar chart showing gender distribution
alt.Chart(data_final).mark_bar(color="#D65F5F").encode(
    x=alt.X("Gender:N", title="Gender"),
    y=alt.Y("count()", title="Customer Count")
).properties(
    width=350,
    height=300,
    title="Gender Breakdown of Credit Card Customers"
)

alt.Chart(...)
```

Breakdown of Customer Education Levels

The distribution of education levels among customers shows a clear concentration in a few categories. Most customers have either a Graduate or High School education, followed by a smaller group with unknown education status. College, Post-Graduate, and Doctorate levels are less represented. Overall, the spread is moderately skewed, but not unexpected for a general consumer credit dataset.

```
# Display sorted counts of education levels
education_counts =
data_final["Education_Level"].value_counts(sort=True)
education_counts

Education_Level
Graduate         3128
High School      2013
Unknown          1519
Uneducated       1487
College          1013
Post-Graduate     516
Doctorate         451
Name: count, dtype: int64

# Bar chart showing distribution of education levels
alt.Chart(data_final).mark_bar(color="#F2C94C").encode(
    x=alt.X("Education_Level:N", title="Education Level",
sort=education_counts.index.tolist()),
    y=alt.Y("count()", title="Number of Customers")
).properties(
    width=350,
    height=300,
    title="Customer Distribution by Education Level"
)

alt.Chart(...)
```

Breakdown of Customer Income Categories

The customer base is concentrated in the lower income brackets, with the largest group earning less than 40K. Moderate representation follows in the 40K to 60K and 60K to 80K ranges. Higher income categories, including 80K to 120K and 120K and above, are less common. A notable portion of customers have their income listed as "Unknown." Overall, the distribution leans toward lower to middle income levels.

```
# Display sorted counts of income categories
income_counts =
data_final["Income_Category"].value_counts(ascending=False)
income_counts
```

```
Income_Category
Less than $40K    3561
$40K - $60K       1790
$80K - $120K      1535
$60K - $80K       1402
Unknown           1112
$120K +            727
Name: count, dtype: int64
```

```
# Bar chart showing distribution of income categories
alt.Chart(data_final).mark_bar(color="#76B7B2").encode(
    x=alt.X("Income_Category:N", title="Income Category", sort="-y"),
    y=alt.Y("count()", title="Number of Customers")
).properties(
    width=350,
    height=300,
    title="Customer Distribution by Income Category"
)
```

```
alt.Chart(...)
```

Customer Segmentation by Card Category

The distribution of Card Category is heavily skewed, which suggests it may offer limited value for modeling or analysis without further transformation.

```
# Bar chart showing distribution of card categories
alt.Chart(data_final).mark_bar(color="#9B51E0").encode(
    x=alt.X("Card_Category:N", title="Card Category", sort="-y"),
    y=alt.Y("count()", title="Number of Customers")
).properties(
    width=350,
    height=300,
    title="Customer Distribution by Card Category"
)
```

```
alt.Chart(...)
```

Age Distribution of Customers

The age distribution of customers follows a roughly bell-shaped curve, with the highest concentration in the mid-40s to early 50s range. This segment represents the peak of the customer base, while both younger and older age groups appear less represented. The overall spread suggests a mature customer population, with most individuals falling between their late 30s and mid-60s.

```
# Histogram of customer age distribution
alt.Chart(data_final).mark_bar(color="#6FCF97").encode(
    x=alt.X("Customer_Age:Q", bin=alt.Bin(maxbins=20), title="Age
Range"),
    y=alt.Y("count()", title="Number of Customers")
).properties(
    width=350,
    height=300,
    title="Distribution of Customer Age"
)
```

```
alt.Chart(...)
```

Number of Dependents per Account Holder

The distribution of dependents per account holder peaks at three dependents, indicating this is the most common household size among customers. Zero and two dependents follow closely behind, while one, four, and five dependents show progressively lower counts. Overall, the data suggests a mix of small to mid-sized households, with three dependents being the most typical.

```
# Bar chart showing distribution of dependent counts
alt.Chart(data_final).mark_bar(color="#F2994A").encode(
    x=alt.X("Dependent_count:O", title="Number of Dependents"),
    y=alt.Y("count()", title="Customer Count")
).properties(
    width=350,
    height=300,
    title="Distribution of Dependents per Customer"
)
```

```
alt.Chart(...)
```

Feature Selection Based on Attrition Correlation

To get a clearer picture of what influences customer churn, I looked at how each feature correlates with Attrition_Flag. A handful of variables stood out with strong signals:

- Total_Relationship_Count
- Months_Inactive_12_mon

- Contacts_Count_12_mon
- Total_Revolving_Bal
- Total_Amt_Chng_Q4_Q1
- Total_Trans_Ct

These are the ones most closely tied to attrition behavior and are likely to add real value to predictive models. The rest did not show much correlation and may not contribute meaningfully without further engineering.

To test this, I will run two sets of models:

- One using just the correlated features
- Another using all available features

This way, we can see whether a leaner input set improves performance or if the extra data adds nuance worth keeping.

Analysis (Model Building and Training)

Models - Unsupervised

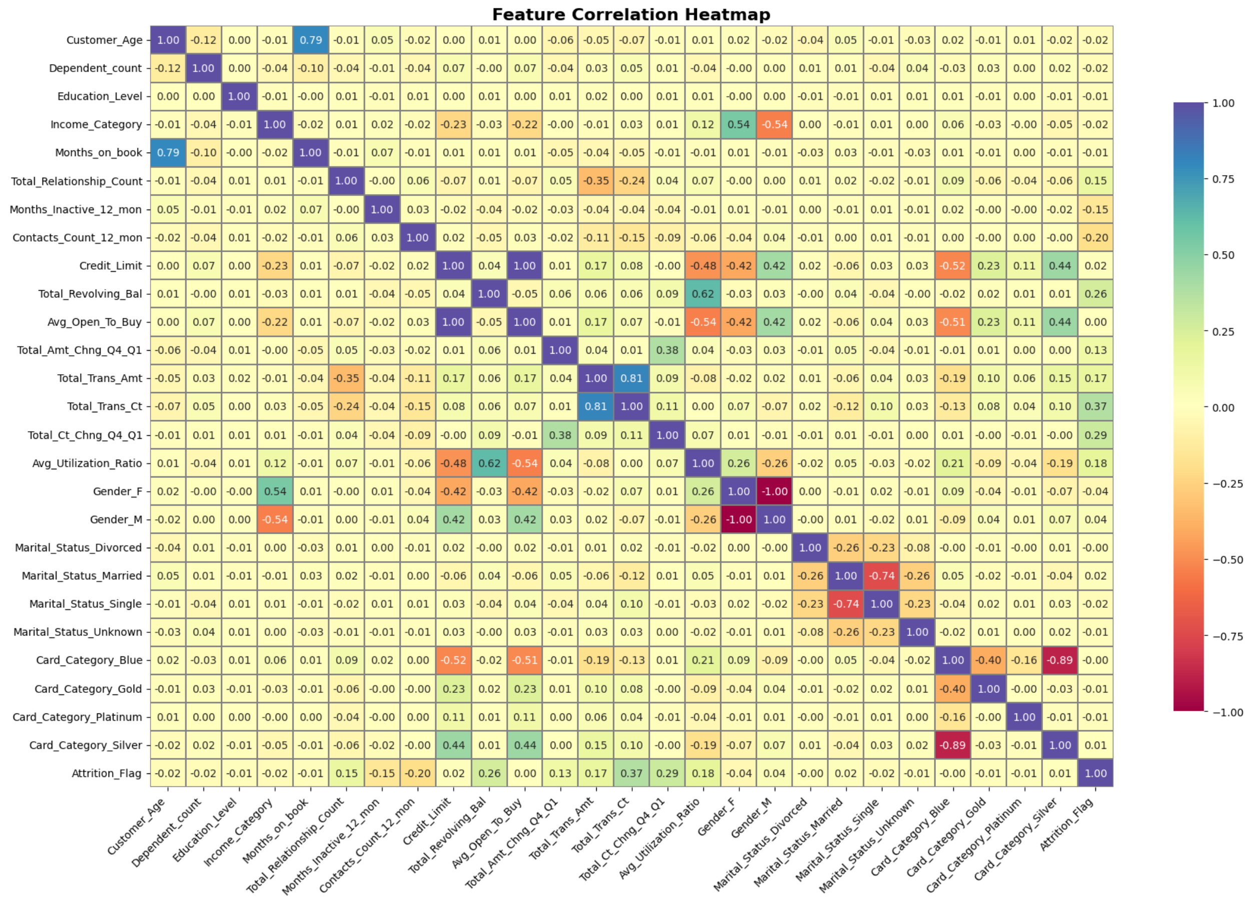
To explore customer segmentation without relying on labeled data, I tested two clustering approaches KMeans and Agglomerative Clustering across seven configurations. Each variation tweaks the input features or preprocessing strategy to uncover different structural patterns in the data:

1. KMeans using all available columns
2. KMeans using only the features most correlated with attrition
3. Agglomerative Clustering with all columns, converting categorical features to numeric codes
4. Agglomerative Clustering with all columns and ward linkage for tighter cluster formation
5. Agglomerative Clustering on an undersampled dataset to balance class representation
6. Agglomerative Clustering using only the correlated features, with categorical data numerically encoded
7. Agglomerative Clustering using Gower distance to better handle mixed data types

For each setup, I evaluated clustering quality using accuracy, precision, recall, and F1 score; giving a clearer sense of which configurations best separate customer behaviors.

```
# Try different label orders to find the best match with actual labels

def find_best_label_mapping(true_labels, predicted_labels,
class_names, metric="accuracy"):
    def evaluate_mapping(mapping):
        remapped = true_labels.map(mapping)
        scores = precision_recall_fscore_support(remapped,
predicted_labels, average="weighted")
        return {
            "accuracy": accuracy_score(remapped, predicted_labels),
            "precision": scores[0],
            "recall": scores[1],
            "f1": scores[2]
        }

    top_score = float("-inf")
    top_metrics = {}
    top_mapping = {}

    for combo in permutations(range(len(class_names))):
        current_map = {name: idx for name, idx in zip(class_names,
combo)}
        current_metrics = evaluate_mapping(current_map)

        if current_metrics[metric] > top_score:
            top_score = current_metrics[metric]
            top_metrics = current_metrics
            top_mapping = {idx: name for idx, name in zip(combo,
class_names)}

    return combo, top_score, top_metrics, top_mapping

# Extract column groups from config
ordinal_cols = [col for col, order in categorical_config.items() if
order is not None]
nominal_cols = [col for col, order in categorical_config.items() if
order is None and col != "Attrition_Flag"]

# Encode ordinal features
for col in ordinal_cols:
    data_final[col] = LabelEncoder().fit_transform(data_final[col])

# One-hot encode nominal features
data_final = pd.get_dummies(data_final, columns=nominal_cols)

# Compute correlation matrix
correlations = corr_data.corr().round(2)

# Set up the plot
plt.figure(figsize=(18, 12))
```

```
sns.heatmap(
    correlations,
    annot=True,
    fmt=".2f",
    cmap="Spectral",           # Vibrant diverging color palette
    center=0,                  # Centered at zero for better contrast
    linewidths=0.3,
    linecolor="gray",
    cbar_kws={"shrink": 0.8}   # Smaller color bar
)

plt.title("Feature Correlation Heatmap", fontsize=16,
fontweight="bold")
plt.xticks(rotation=45, ha="right")
plt.yticks(rotation=0)
plt.tight_layout()
plt.show()
```



```
key_features_for_attrition = [
    "Total_Relationship_Count",
    "Months_Inactive_12_mon",
    "Contacts_Count_12_mon",
    "Total_Revolving_Bal",
    "Total_Amt_Chng_Q4_Q1",
    "Total_Trans_Ct"
]

# Combine features and labels for sampling
df = pd.concat([data_final, labels], axis=1)

majority = df[df["Attrition_Flag"] == "Existing Customer"]
minority = df[df["Attrition_Flag"] == "Attrited Customer"]

# Downsample majority class to match minority count
majority_sampled = majority.sample(n=len(minority), random_state=42)
df_balanced = pd.concat([majority_sampled, minority]).sample(frac=1,
random_state=42)

X_resampled = df_balanced.drop("Attrition_Flag", axis=1)
y_resampled = df_balanced["Attrition_Flag"]

# Scale all features
scaler_all = StandardScaler()
scaled_all = scaler_all.fit_transform(data_final)
scaled_all_resampled = scaler_all.fit_transform(X_resampled)

# Scale selected correlative features
scaler_subset = StandardScaler()
scaled_subset =
scaler_subset.fit_transform(data_final[key_features_for_attrition])
scaled_subset_resampled =
scaler_subset.fit_transform(X_resampled[key_features_for_attrition])
```

KMeans with All Available Columns

KMeans clustering across the full feature set yielded modest results, with an F1 score of 0.755.

```
# Fit KMeans model on full feature set
kmeans_all = KMeans(n_clusters=2, n_init="auto", random_state=42)
cluster_labels = kmeans_all.fit_predict(scaled_all)

# Evaluate clustering alignment with true labels
_, _, kmeans_metrics, label_map = label_permute_compare(
    labels["Attrition_Flag"], cluster_labels, label_classes,
"accuracy"
)

kmeans_metrics
```

```
{'accuracy': 0.792436062012442,
 'precision': 0.7293343426305999,
 'recall': 0.792436062012442,
 'f1': 0.7559451980077926}
```

KMeans with Correlation-Selected Features

KMeans delivered comparable results when limited to the correlation-selected features, reaching an F1 score of 0.571.

```python
# Fit KMeans model on selected correlative features
kmeans_subset = KMeans(n_clusters=2, n_init="auto", random_state=42)
subset_cluster_labels = kmeans_subset.fit_predict(scaled_subset)

# Evaluate clustering performance against true labels
_, _, subset_metrics, label_map_subset = label_permute_compare(
    labels["Attrition_Flag"], subset_cluster_labels, label_classes,
"accuracy"
)

subset_metrics
```

```
{'accuracy': 0.5083440308087291,
 'precision': 0.666806152582151,
 'recall': 0.5083440308087291,
 'f1': 0.5713161095441398}
```

Agglomerative Clustering with All Available Columns

Agglomerative Clustering using all columns, with categorical features numerically encoded, showed stronger performance. This approach achieved an F1 score of 0.766.

```python
# Fit Agglomerative Clustering on full feature set
agg_model_all = AgglomerativeClustering(
    n_clusters=2,
    metric="euclidean",
    linkage="complete",
    compute_distances=True
)

agg_cluster_labels = agg_model_all.fit_predict(scaled_all)

# Evaluate clustering performance against true labels
_, _, agg_metrics_all, label_map_agg = label_permute_compare(
    labels["Attrition_Flag"], agg_cluster_labels, label_classes,
"accuracy"
)

agg_metrics_all
```

```
{'accuracy': 0.8383529179421348,
 'precision': 0.7448055607507813,
 'recall': 0.8383529179421348,
 'f1': 0.7664726575589648}
```

Agglomerative Clustering with All Available Columns + Ward Linkage

Agglomerative Clustering with Ward linkage, applied to the full feature set including numerically encoded categorical variables, delivered comparable performance, yielding an F1 score of 0.756.

```
# Fit Agglomerative Clustering using Ward linkage on full feature set
agg_model_ward = AgglomerativeClustering(
    n_clusters=2,
    linkage="ward",
    compute_distances=True
)

ward_cluster_labels = agg_model_ward.fit_predict(scaled_all)

# Evaluate clustering performance against true labels
_, _, ward_metrics, label_map_ward = label_permute_compare(
    labels["Attrition_Flag"], ward_cluster_labels, label_classes,
"accuracy"
)

ward_metrics
```

```
{'accuracy': 0.7925348079391725,
 'precision': 0.7293850521403183,
 'recall': 0.7925348079391724,
 'f1': 0.7560039370982169}
```

Agglomerative Clustering with All Available Columns + Undersampled Dataset

Agglomerative Clustering on the undersampled, balanced dataset resulted in notably weaker performance, with an F1 score of just 0.384.

```
# Fit Agglomerative Clustering with Ward linkage on resampled data
agg_model_resampled = AgglomerativeClustering(
    n_clusters=2,
    linkage="ward",
    compute_distances=True
)

resampled_cluster_labels =
agg_model_resampled.fit_predict(scaled_all_resampled)

# Evaluate clustering performance against resampled true labels
```

```
_, _, agg_metrics_resampled, label_map_resampled =
label_permute_compare(
    y_resampled, resampled_cluster_labels, label_classes, "accuracy"
)

agg_metrics_resampled
```

```
{'accuracy': 0.5003073140749846,
 'precision': 0.501245054600274,
 'recall': 0.5003073140749846,
 'f1': 0.38439274479881486}
```

Agglomerative Clustering with Correlation-Selected Features

Agglomerative Clustering using only the correlation-selected features showed slightly lower performance compared to the full-feature model, with an F1 score of 0.748.

```
# Fit Agglomerative Clustering with complete linkage on correlative
features
agg_model_subset = AgglomerativeClustering(
    n_clusters=2,
    linkage="complete",
    compute_distances=True
)

subset_cluster_labels = agg_model_subset.fit_predict(scaled_subset)

# Evaluate clustering performance against true labels
_, _, agg_metrics_subset, label_map_subset = label_permute_compare(
    labels["Attrition_Flag"], subset_cluster_labels, label_classes,
"accuracy"
)

agg_metrics_subset
```

```
{'accuracy': 0.8032981139527995,
 'precision': 0.702351576184927,
 'recall': 0.8032981139527995,
 'f1': 0.748687833815765}
```

Agglomerative Clustering with Gower Distance

This approach performed similarly to the Euclidean-based method with numerically encoded categorical data, yielding a comparable F1 score of 0.762.

```
# Convert categorical columns to string for Gower compatibility
for col in ordinal_cols + nominal_cols:
    data_original[col] = data_original[col].astype(str)
```

```
# Compute Gower distance matrix
gower_dist_matrix = gower.gower_matrix(data_original)

# Fit Agglomerative Clustering using precomputed Gower distances
agg_model_gower = AgglomerativeClustering(
    n_clusters=2,
    metric="precomputed",
    linkage="average"
)

gower_cluster_labels = agg_model_gower.fit_predict(gower_dist_matrix)

# Evaluate clustering performance
_, _, gower_metrics, label_map_gower = label_permute_compare(
    labels["Attrition_Flag"], gower_cluster_labels, label_classes,
"accuracy"
)

gower_metrics
```

```
{'accuracy': 0.8124814851387381,
 'precision': 0.7308330405617858,
 'recall': 0.8124814851387381,
 'f1': 0.7623152727247017}
```

Models - Supervised

To evaluate how unsupervised clustering compares to predictive modeling, I used XGBoost, a tree-based supervised learning algorithm known for its performance and interpretability. The dataset was split with 20 percent reserved for testing. Each model was trained on the remaining data and assessed using consistent metrics including accuracy, precision, recall, and F1 score. This provided a clear benchmark for comparing supervised learning outcomes with the patterns identified through clustering.

XGBoost with All Available Columns

XGBoost outperformed all unsupervised approaches, achieving a notably high F1 score of 0.968.

```
# Encode target labels for classification
encoded_labels = label_encoder.fit_transform(labels["Attrition_Flag"])

# Split data into training and testing sets with stratified sampling
X_train, X_test, y_train, y_test = train_test_split(
    scaled_all,
    encoded_labels,
    test_size=0.2,
    stratify=encoded_labels,
    random_state=42
)
```

```
# Convert datasets into XGBoost's DMatrix format
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Define model parameters
params = {
    "objective": "multi:softmax",   # Multiclass classification
    "num_class": 5,                 # Number of target classes
    "max_depth": 10,                # Maximum tree depth
    "eta": 1,                       # Learning rate
    "eval_metric": "merror"         # Evaluation metric:
classification error
}
rounds = 20

# Train the XGBoost model
model = xgb.train(params, dtrain, rounds)

# Generate predictions on the test set
predictions = model.predict(dtest)

# Compute evaluation metrics
precision, recall, f1, _ = precision_recall_fscore_support(y_test,
predictions, average="weighted")

metrics = {
    "accuracy": accuracy_score(y_test, predictions),
    "precision": precision,
    "recall": recall,
    "f1": f1
}

metrics

{'accuracy': 0.9684106614017769,
 'precision': 0.9684106614017769,
 'recall': 0.9684106614017769,
 'f1': 0.9684106614017769}
```

XGBoost with Correlation-Selected Features

Restricting XGBoost to the correlation-selected features resulted in a modest drop in performance, yielding an F1 score of 0.913.

```
# Encode target labels
encoded_labels = label_encoder.fit_transform(labels["Attrition_Flag"])

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    scaled_subset,
```

```python
    encoded_labels,
    test_size=0.2,
    stratify=encoded_labels,
    random_state=42
)

# Prepare data for XGBoost
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# Model parameters
params = {
    "objective": "multi:softmax",
    "num_class": 5,
    "max_depth": 10,
    "eta": 1,
    "eval_metric": "merror"
}
rounds = 20

# Train and predict
model = xgb.train(params, dtrain, rounds)
predictions = model.predict(dtest)

# Evaluate
precision, recall, f1, _ = precision_recall_fscore_support(y_test,
predictions, average="weighted")

metrics = {
    "accuracy": accuracy_score(y_test, predictions),
    "precision": precision,
    "recall": recall,
    "f1": f1
}

metrics
```

```
{'accuracy': 0.9155972359328727,
 'precision': 0.9128463891113534,
 'recall': 0.9155972359328727,
 'f1': 0.9138648365985037}
```

Discussion / Conclusion

This project explored different approaches to predicting customer attrition, starting with unsupervised clustering methods like KMeans and Agglomerative Clustering. I tested several configurations across both algorithms and compared their results to a supervised model using XGBoost.

XGBoost clearly outperformed the clustering models. While clustering helped uncover some structure in the data, it was not nearly as effective for classification. The supervised model delivered much stronger results across all metrics.

One surprising outcome was that using all available features consistently led to better performance than limiting the models to only the most correlated columns. I initially expected that focusing on the strongest predictors would improve accuracy, but the full feature set proved more valuable. This suggests that even weaker individual features may contribute meaningful signal when combined.

Agglomerative Clustering performed especially poorly on the undersampled dataset. This likely reflects how sensitive clustering is to data volume and distribution. With more records, even a balanced subset might have produced better results.

Going forward, I would like to explore clustering in contexts where it is better suited, such as customer segmentation or behavioral grouping. For classification tasks like churn prediction, supervised models are clearly the more reliable choice.