

Investment Type Recommender System

Business Understanding

Overview

Many Kenyans, especially young adults and first-time investors, struggle to identify investment avenues that align with their financial goals, risk appetite, and income level. The lack of personalized financial guidance often leads to poor or delayed investment decisions.

Challenges

- **Low financial literacy and accessibility to advisory services**

Many potential investors lack foundational knowledge or support systems to understand available investment options or evaluate their suitability.

- **Overwhelming investment options**

The abundance of options—such as SACCOs, stocks, real estate, government bonds, and money market funds—can be confusing and lead to decision fatigue.

- **One-size-fits-all investment marketing**

Most financial institutions promote products generically, failing to account for individual goals, income, and risk profiles.

- **Lack of data-driven tools for personalized investment planning**

There is limited availability of intelligent systems to assist users in navigating investments based on their unique profiles.

- **Distrust and fear of loss**

Without adequate knowledge or guidance, potential investors may fear financial loss or fall victim to scams, leading to investment hesitation.

Proposed Solution

A **machine learning-based recommender system** that suggests ideal investment types based on a user's financial profile, risk tolerance, and goals. This system can help both fintech platforms and financial institutions deliver personalized advisory services at scale.

Brief Conclusion

By guiding users toward the most suitable investment types, this solution aims to enhance financial inclusion and support smarter, confidence-driven investment decisions.

Problem Statement

Many individuals, especially in emerging markets, face significant challenges in making informed investment decisions due to limited financial literacy and lack of personalized advisory services. The wide range of available investment options—SACCOs, stocks, real estate, government bonds, and money market funds—can be overwhelming without guidance. Additionally, the generic approach in investment marketing overlooks the diverse financial goals, income levels, and risk appetites of potential investors, leading to poor financial outcomes and disengagement from long-term wealth-building.

Objectives

- **Analysis-Based**

Understand investment behaviors among Kenyan users and segment them based on patterns.

- **Feature Engineering-Based**

Create user profiles using financial behavior indicators such as:

- Income level
- Savings rate
- Age
- Financial goals

- **Modeling-Based**

Build and evaluate recommender models, including:

- Content-based filtering
- Hybrid approaches (clustering + classification)

Data Understanding

Data Sourcing

- Publicly available financial survey data (e.g., **FinAccess Kenya survey**)
- Simulated user profiles or anonymized fintech customer data
- Investment platform usage data (e.g., user interest in asset types)
- Economic indicators (e.g., interest rates, inflation)

Features and Relevance

- Demographics: Age, gender, location
- Financials: Income, expenses, debt levels
- Profile: Risk profile (low/medium/high)
- Preferences: Investment goals (short-term/long-term, passive/active)

- History: Past investment experience

Data Preparation

Format

- Data will be collected and processed in **Excel format**

Actions

- Handle missing values
- Encode categorical variables
- Normalize numeric fields
- Create derived features (e.g., savings rate, risk-adjusted return scores)
- Segment data by user type or financial tier

Deployment

API

- **Framework:** FastAPI
- **Endpoints:** Accept user profile data and return recommended investment type(s)
- **Model Storage:** Serialized using `.pk1` or `.joblib`

UI

- **Framework:** Streamlit
- **Function:** Allows users to input financial info and receive personalized investment suggestions

Prototypes / Mockups

- **Key Screens:**
 - Welcome Screen
 - Financial Profile Input
 - Investment Suggestions

In cell below we import libraries

```
In [ ]: # import libraries
import pandas as pd
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity
from sklearn.preprocessing import OrdinalEncoder, StandardScaler
from sklearn.model_selection import train_test_split
```

```
from sklearn.preprocessing import LabelEncoder, StandardScaler
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.models import load_model
import matplotlib.pyplot as plt
import warnings
warnings.filterwarnings("ignore", category=FutureWarning)
```

In []: # Load data

```
file_path = "./data/final_refined1.csv"

invest_df = pd.read_csv(file_path)
invest_df.head()
```

Out[]:

	householdid	county	area_type	gender	age_of_respondent	no_of_household_mebers
0	107141431	garissa	urban	male	29	5
1	10712933	garissa	urban	male	60	11
2	140173183	busia	urban	female	35	2
3	122137153	kiambu	urban	male	24	1
4	121193116	murang'a	urban	female	20	1

5 rows × 393 columns

In [3]: # check shape of data

```
invest_df.shape
```

Out[3]: (20871, 3816)

The output above shows that dataset contains **20871 entries and 3816 columns**.

In cell below we check for metadata summary and numeric summary

In [4]: # check metadata summary & numeric

```
def data_summary(df):
    print("----info----")
    df.info()

    print("----describe----")
    df.describe()
```

```
return df

data_summary(invest_df)
-----info-----
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20871 entries, 0 to 20870
Columns: 3816 entries, interview_key to Longitude
dtypes: float64(2303), int64(185), object(1328)
memory usage: 607.6+ MB
-----describe-----
```

Out[4]:

	interview_key	interview_id	county	A07	A08	A10	Id
0	21-61-43-44	00005e63e7424aa19e63adb071955233	Garissa	225	Urban	1	
1	87-78-54-60	000718d26f5942998522033a66594b4c	Garissa	223	Urban	3	
2	32-71-19-84	0008721681d2468aa1ef456bba4475dc	Busia	1514	Urban	3	
3	48-18-78-99	00099e961d49470585c1a7f49fa3b7fb	Kiambu	796	Urban	3	
4	50-82-98-89	000bb72e959b4dd48443da902c01e920	Murang'a	763	Urban	16	
...	
20866	96-94-76-93	fffe104529e439fb916d956d10a9dc0	Wajir	1847	Rural	1	
20867	40-55-72-03	fff39eec827f4ddc90e73a0792092bf9	Nairobi City	1815	Urban	25	
20868	10-75-82-28	fff70af11a2e49649feae012317ce7d5	Embu	443	Rural	1	
20869	77-66-40-97	fff8b3b628df4b67889e26af8df90ef7	Taita-Taveta	213	Urban	1	
20870	09-26-74-74	ffffe9b46c1014de9be537c63c210014c	Kisii	1706	Rural	1	

20871 rows × 3816 columns



The output above shows dataset contains total 20871 entries and 3816 columns in which 2303 columns are float, 185 integers and 1328 object

Basic Data cleaning

In cell below we inspect data and drop all columns not needed based on domain knowledge and data description, then check for missing values, duplicates, standardize all categorical

data to lower case and remove white, and impute for the missing values

In cell below we check for the missing values and impute them for efficient feature engineering

```
In [7]: # check for missing values

def missing_values(df):
    missing = df.isnull().sum()
    missing = missing[missing > 0].sort_values(ascending=False)

    return missing

missing_values(invest_df)
```

```
Out[7]: M10i      20871
N11bi     20871
K5i       20871
J3_187    20871
J3_188    20871
...
B2Eii     1
B2Ei      1
B2F       1
B2D       1
A24iii   1
Length: 3318, dtype: int64
```

for `numeric_values` we impute with **median** and for `categorical` we use **mode** or **unknown**. we drop columns we much missing values.

```
In [8]: # make copy of original data

invest_df_copy = invest_df.copy()
```

```
In [9]: # Drop columns with much missing values
def drop_columns(df, columns_missing):
    df_cleaned = df.drop(columns=columns_missing, axis=1, errors='ignore') # error
    print(f"Dropped {len(columns_missing)} columns.")
    return df_cleaned

entries = [
    'investment_redress', 'pension_redress', 'mobilebank_redress', 'sacco_redress',
    'trad_mfi_satisfaction', 'bank_redress', 'insurance_redress', 'trad_mfi_moneylo
    'trad_mfi_downtime', 'trad_mfi_unexpectedcharges', 'trad_mfi_unethicalrecovery'
    'digital_app_moneylest', 'digital_app_unsolicited', 'digitalapps_unexpectedchar
    'digital_apps_unethicalrecovery', 'traditional_mfi_issues', 'investment_unexpec
    'investment_sold_fnl', 'investment_lostmoney_fnl', 'investment_downtime_fnl', '
    'investment_satisfaction', 'digital_issues', 'hirepurchase_satisfaction', 'hire
    'hirepurchase_downtime', 'hirepurchase_unexpectedcharges', 'hirepurchase_unsolici
    'hirepurchase_unethicalrecovery', 'hirepurchase_issues', 'mobilemoney_redress',
    'creditonlyagree_satisfactionl', 'creditonly_mfi_moneylest', 'creditonly_mfi_d
    'creditonly_mfi_unsolicited', 'creditonly_mfi_unexpectedcharges', 'creditonly_m
    'creditonly_mfi_issues', 'pension_unethical_fnl', 'pension_underpayment_fnl', '
```

```

    'pension_delayed_fnl', 'pension_issues_fnl', 'pension_satisfaction', 'pension_l
'sacco_unexpectedcharges'
]

invest_df_copy = drop_columns(invest_df_copy, entries)

```

Dropped 50 columns.

In cell above we drop all columns with much missing value. Below we proceed to impute missing for both `numeric_values` and `categorical_values`.

```

In [43]: # impute missing values for both numeric_values and categorical values

import warnings
warnings.filterwarnings("ignore", category=FutureWarning)

def impute_missing_values(df):
    """Impute missing values: numeric with median, categorical with mode or 'unknown'
    for col in df.columns:
        if df[col].isnull().sum() == 0:
            continue
        if df[col].dtype in ['float64', 'int64']:
            median_val = df[col].median()
            df[col].fillna(median_val, inplace=True)
        else:
            mode_val = df[col].mode()
            if not mode_val.empty:
                df[col].fillna(mode_val[0], inplace=True)
            else:
                df[col].fillna('unknown', inplace=True)
    print("Imputed missing values for all applicable columns.")
    return df

invest_df_copy = impute_missing_values(invest_df_copy)

```

```

In [11]: # check for missing values after imputing

invest_df_copy.isnull().sum()

```

```

Out[11]: interview_key          0
interview_id           0
county                 0
A07                   0
A08                   0
..
insurance_including_NHIF_use  0
All_Insurance_excluding_NHIF_use 0
PWD                   0
Latitude               0
Longitude              0
Length: 3767, dtype: int64

```

```

In [12]: # check for duplicates

```

```
invest_df_copy.duplicated().sum()
```

Out[12]: np.int64(0)

No duplicates found. Below we `standardize categorical data` by standardizing all letters to lower, and removing whitespace

In [13]: `# standardize categorical data`

```
def clean_cat_text(df):
    cat_cols = df.select_dtypes(include="object").columns
    for col in cat_cols:
        df[col] = df[col].astype(str).str.strip().str.lower()
    print("standardized all categorical text to lowercase and stripped whitespace")

return df

invest_df_copy = clean_cat_text(invest_df_copy)
invest_df_copy.head()
```

standardized all categorical text to lowercase and stripped whitespace

Out[13]:

	interview_key	interview_id	county	A07	A08	A10	Interv
0	21-61-43-44	00005e63e7424aa19e63adb071955233	garissa	225	urban	1	house with
1	87-78-54-60	000718d26f5942998522033a66594b4c	garissa	223	urban	3	house with
2	32-71-19-84	0008721681d2468aa1ef456bba4475dc	busia	1514	urban	3	house with
3	48-18-78-99	00099e961d49470585c1a7f49fa3b7fb	kiambu	796	urban	3	house with
4	50-82-98-89	000bb72e959b4dd48443da902c01e920	murang'a	763	urban	16	house with

5 rows × 3767 columns



Preparing data for recommender system & Feature engineering

This project focuses on developing a **machine learning-based recommender system designed to suggest appropriate investment products tailored to a user's financial profile, risk tolerance, and personal goals**. To support this, we began by preparing the

dataset through a structured process that includes selecting relevant features, renaming columns for readability, and performing necessary cleaning and exploratory analysis.

We have chosen to implement a **content-based filtering** approach for the recommendation engine. **This decision is guided by the nature of the FinAccess 2024 dataset, which consists of aggregated survey responses rather than individual user-product interaction data.** Collaborative filtering methods such as ALS or SVD typically rely on user-level histories and are therefore not applicable in this context.

Instead, content-based filtering allows us to leverage the dataset's rich demographic and behavioral attributes such as age, gender, region, education level, trust in financial institutions, and satisfaction with financial products to make informed recommendations. By aligning a new user's profile with patterns observed in similar demographic segments, we can suggest investment options that are likely to be suitable and relevant.

The recommender system works by capturing key user inputs (such as age group, gender, education, region), identifying similar profiles within the dataset, analyzing common or highly rated financial products within those segments, and recommending those products to the user. This personalized, profile-driven approach offers a practical and data-informed way to support individuals in making better investment decisions.

As our data is recorded in `household-level` and not individual-level we treat each household as the user. The unit of our recommendation is per household not per person. We feature engineer columns such as `has_account`, `has_savings`, `receives_remittance`. We create new binary features to show **presence or absence** of certain financial behaviours, that is whether any household member uses a savings method or owns a phone

In cell below we prepare our **X features** and **y feature**, based on the following features from data

- **demographics & preferences**
- **current or past product usage**
- **user preferences**
- **Trust & digital access**

we start by recording our `items to recommend` which is **investment types**. Then we define features to represent **User profile** those describe household such as `financial access`, `saving behaviour`, `Risk attitude`, `income level`, `education level` and `Household size`, `Digital literacy`

Below we use `cosine_similarity` to create vectors for recommendation. As we have `user profile` and `investment options` for each household we know which investment methods they use and want to recommend more suitable ones, possibly those they haven't used but similar to what similar household use.

```
In [ ]: # Select Relevant Columns
relevant_cols = [
    "A08",
    "A13",
    "B3IIi",
    "U23",
    "C1_1a",
    "C1_2",
    "C1_4",
    "C1_6",
    "C1_9",
    "C1_15",
    "C1_17",
    "C1_19",
    "C1_25",
    "C1_35"
]
df_subset = invest_df_copy[relevant_cols].copy()

# Rename Columns
df_subset.columns = [
    "area_type", "gender", "monthly_income", "monthly_expenditure",
    "save_bank", "save_mobile_money", "save_sacco", "save_friends", "save_digital",
    "loan_mobile", "loan_sacco", "loan_digital", "loan_family", "invest_forex"
]
```

```
In [ ]: #Map Usage Levels to Ordinal
usage_map = {
    "Never used": 0,
    "Used to use": 1,
    "Currently use": 2,
    pd.NA: 0,
    None: 0
}
behavior_cols = [
    'save_bank', 'save_mobile_money', 'save_sacco', 'save_friends', 'save_digital',
    'loan_mobile', 'loan_sacco', 'loan_digital', 'loan_family', 'invest_forex'
]
df_subset[behavior_cols] = df_subset[behavior_cols].applymap(lambda x: usage_map.get(x))
```

```
In [ ]: # Encode Demographics
df_subset['gender'] = df_subset['gender'].map({'Male': 0, 'Female': 1})
df_subset['area_type'] = df_subset['area_type'].map({'Rural': 0, 'Urban': 1})
```

```
In [ ]: # Fill Missing Numeric Data
df_subset['monthly_income'] = df_subset['monthly_income'].fillna(df_subset['monthly_income'].mean())
df_subset['monthly_expenditure'] = df_subset['monthly_expenditure'].fillna(df_subset['monthly_expenditure'].mean())
```

```
#Scale Features
scaler = StandardScaler()
household_scaled = scaler.fit_transform(df_subset)
```

```
In [ ]: #Create 3 Dummy Investment Profiles
investment_profiles = pd.DataFrame([
    [1, 1, 0.3, 0.3, 2, 1, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0],
    [1, 1, 0.5, 0.5, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0],
    [1, 1, 0.6, 0.6, 0, 2, 0, 0, 2, 2, 2, 2, 2, 2, 2, 2]
], columns=df_subset.columns, index=['conservative', 'balanced', 'aggressive'])

investment_scaled = scaler.transform(investment_profiles)
```

```
In [ ]: household_scaled = np.nan_to_num(household_scaled, nan=0.0)
investment_scaled = np.nan_to_num(investment_scaled, nan=0.0)

# Compute Cosine Similarity
similarity_matrix = cosine_similarity(household_scaled, investment_scaled)
recommendations = pd.DataFrame(similarity_matrix, columns=investment_profiles.index)

#Label Top-1 Strategy
df_subset["investment_label"] = recommendations.apply(lambda row: row.nlargest(1).i
```

```
In [37]: # Split into features and label
X = df_subset.drop("investment_label", axis=1)
y = df_subset["investment_label"]
```

```
In [ ]: # Encode target
label_encoder = LabelEncoder()
y_encoded = label_encoder.fit_transform(y)

# Scale features
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)
```

```
In [39]: # Train-test split
X_train, X_test, y_train, y_test = train_test_split(X_scaled, y_encoded, test_size=
```

```
In [ ]: # Neural network
model = Sequential([
    Dense(64, activation='relu', input_shape=(X_train.shape[1],)),
    Dropout(0.3),
    Dense(32, activation='relu'),
    Dense(len(label_encoder.classes_), activation='softmax')
])

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['a
history = model.fit(X_train, y_train, validation_split=0.2, epochs=30, batch_size=1
```

```
c:\Users\ronny somi\Documents\Group4_Capstone_Final_Project\.venv\Lib\site-packages
\keras\src\layers\core\dense.py:92: UserWarning: Do not pass an `input_shape` / `input_
_dim` argument to a layer. When using Sequential models, prefer using an `Input(shap
e)` object as the first layer in the model instead.
    super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Epoch 1/30
835/835 18s 4ms/step - accuracy: 0.6714 - loss: 0.6473 - val_accuracy: 0.6850 - val_loss: 0.6246

Epoch 2/30
835/835 3s 4ms/step - accuracy: 0.6721 - loss: 0.6328 - val_accuracy: 0.6850 - val_loss: 0.6234

Epoch 3/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6328 - val_accuracy: 0.6850 - val_loss: 0.6233

Epoch 4/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6327 - val_accuracy: 0.6850 - val_loss: 0.6240

Epoch 5/30
835/835 3s 4ms/step - accuracy: 0.6721 - loss: 0.6327 - val_accuracy: 0.6850 - val_loss: 0.6233

Epoch 6/30
835/835 5s 4ms/step - accuracy: 0.6721 - loss: 0.6327 - val_accuracy: 0.6850 - val_loss: 0.6236

Epoch 7/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6327 - val_accuracy: 0.6850 - val_loss: 0.6237

Epoch 8/30
835/835 3s 4ms/step - accuracy: 0.6721 - loss: 0.6328 - val_accuracy: 0.6850 - val_loss: 0.6235

Epoch 9/30
835/835 3s 4ms/step - accuracy: 0.6721 - loss: 0.6328 - val_accuracy: 0.6850 - val_loss: 0.6234

Epoch 10/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6328 - val_accuracy: 0.6850 - val_loss: 0.6234

Epoch 11/30
835/835 6s 4ms/step - accuracy: 0.6721 - loss: 0.6328 - val_accuracy: 0.6850 - val_loss: 0.6234

Epoch 12/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6327 - val_accuracy: 0.6850 - val_loss: 0.6233

Epoch 13/30
835/835 6s 4ms/step - accuracy: 0.6721 - loss: 0.6327 - val_accuracy: 0.6850 - val_loss: 0.6235

Epoch 14/30
835/835 5s 4ms/step - accuracy: 0.6721 - loss: 0.6327 - val_accuracy: 0.6850 - val_loss: 0.6233

Epoch 15/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6327 - val_accuracy: 0.6850 - val_loss: 0.6232

Epoch 16/30
835/835 3s 4ms/step - accuracy: 0.6721 - loss: 0.6328 - val_accuracy: 0.6850 - val_loss: 0.6234

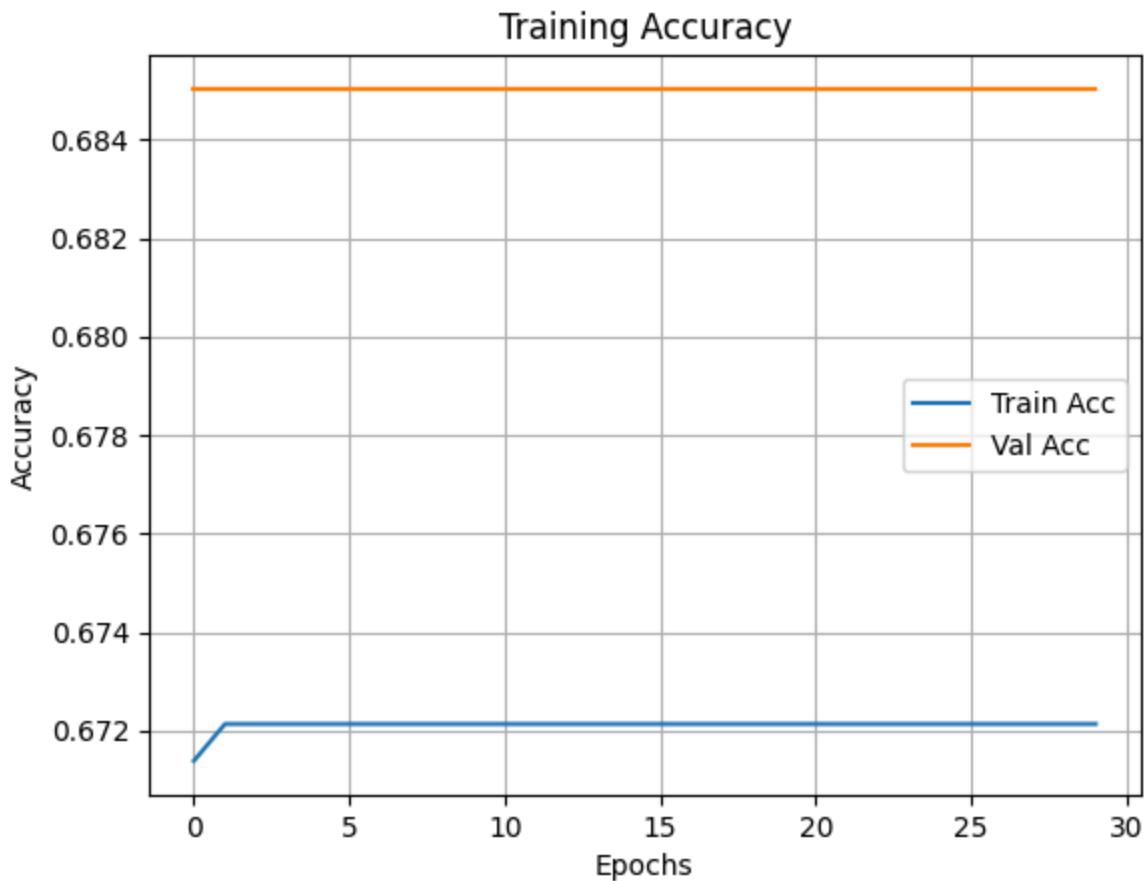
Epoch 17/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6327 - val_accuracy: 0.6850 - val_loss: 0.6234

Epoch 18/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6328 - val_accuracy: 0.6850 - val_loss: 0.6233

Epoch 19/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6327 - val_accuracy: 0.6850 - val_loss: 0.6233

```
uracy: 0.6850 - val_loss: 0.6231
Epoch 20/30
835/835 3s 4ms/step - accuracy: 0.6721 - loss: 0.6328 - val_acc
uracy: 0.6850 - val_loss: 0.6231
Epoch 21/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6327 - val_acc
uracy: 0.6850 - val_loss: 0.6232
Epoch 22/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6327 - val_acc
uracy: 0.6850 - val_loss: 0.6237
Epoch 23/30
835/835 6s 4ms/step - accuracy: 0.6721 - loss: 0.6328 - val_acc
uracy: 0.6850 - val_loss: 0.6237
Epoch 24/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6328 - val_acc
uracy: 0.6850 - val_loss: 0.6233
Epoch 25/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6328 - val_acc
uracy: 0.6850 - val_loss: 0.6234
Epoch 26/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6327 - val_acc
uracy: 0.6850 - val_loss: 0.6233
Epoch 27/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6327 - val_acc
uracy: 0.6850 - val_loss: 0.6239
Epoch 28/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6328 - val_acc
uracy: 0.6850 - val_loss: 0.6238
Epoch 29/30
835/835 5s 4ms/step - accuracy: 0.6721 - loss: 0.6328 - val_acc
uracy: 0.6850 - val_loss: 0.6232
Epoch 30/30
835/835 3s 3ms/step - accuracy: 0.6721 - loss: 0.6328 - val_acc
uracy: 0.6850 - val_loss: 0.6234
```

```
In [ ]: # Plot accuracy
plt.plot(history.history['accuracy'], label='Train Acc')
plt.plot(history.history['val_accuracy'], label='Val Acc')
plt.legend()
plt.title("Training Accuracy")
plt.xlabel("Epochs")
plt.ylabel("Accuracy")
plt.grid(True)
plt.show()
```



```
In [ ]: # save model  
model.save("../models/investment_model.h5")
```

```
In [1]: import pandas as pd  
import numpy as np  
import matplotlib.pyplot as plt  
import seaborn as sns  
from sklearn.model_selection import train_test_split, cross_val_score  
from sklearn.preprocessing import StandardScaler, LabelEncoder, OneHotEncoder  
from sklearn.impute import SimpleImputer  
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier  
from sklearn.linear_model import LogisticRegression  
from sklearn.cluster import KMeans  
from sklearn.decomposition import PCA  
from sklearn.metrics import (  
    silhouette_score, classification_report,  
    confusion_matrix, precision_recall_curve,  
    average_precision_score, precision_score,  
    recall_score, f1_score,  
    roc_auc_score, roc_curve,  
    auc, accuracy_score)  
from sklearn.metrics.pairwise import cosine_similarity  
from sklearn.dummy import DummyClassifier  
from sklearn.tree import DecisionTreeClassifier  
from sklearn.neighbors import NearestNeighbors  
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
```

```
from sklearn.pipeline import Pipeline
from sklearn.compose import ColumnTransformer
from scipy.stats import randint, uniform
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.callbacks import EarlyStopping, ReduceLROnPlateau
import joblib
import json
import time
import os
import warnings
warnings.filterwarnings('ignore')

# Set plotting style
plt.style.use('seaborn-v0_8')
sns.set_palette("husl")
```

In [2]: # Phase 2: Data Understanding - Investment Recommendation System

```
# Helper function to safely plot data
def safe_plot(data, plot_type='bar', title='', xlabel='', ylabel='', **kwargs):
    """Safely plot data with error handling"""
    try:
        if isinstance(data, pd.Series):
            if len(data) == 0:
                print(f"Warning: No data to plot for {title}")
                return
            if data.isnull().all():
                print(f"Warning: All values are null for {title}")
                return

        plt.figure(figsize=kwargs.get('figsize', (10, 6)))

        if plot_type == 'bar':
            data.plot(kind='bar')
        elif plot_type == 'pie':
            data.plot(kind='pie', autopct='%1.1f%%')
        elif plot_type == 'hist':
            data.hist(bins=kwargs.get('bins', 30), edgecolor='black')

        plt.title(title)
        if xlabel:
            plt.xlabel(xlabel)
        if ylabel:
            plt.ylabel(ylabel)

        if plot_type == 'bar':
            plt.xticks(rotation=45)

        plt.tight_layout()
        plt.show()

    except Exception as e:
        print(f"Error plotting {title}: {str(e)}")
```

```

        print("Continuing with next analysis...")

## 2.1 Initial Data Loading and Overview

# Load the dataset
df = pd.read_csv('final_refined.csv')
print(f"Dataset shape: {df.shape}")
print(f"Memory usage: {df.memory_usage(deep=True).sum() / 1024**2:.2f} MB")

## 2.2 Dataset Structure Analysis

print("== DATASET OVERVIEW ==")
print(f"Number of rows: {df.shape[0]}")
print(f"Number of columns: {df.shape[1]}")
print(f"\nData types:")
print(df.dtypes.value_counts())

# Display first few rows
print("\n== SAMPLE DATA ==")
print(df.head())

# Basic info about the dataset
print("\n== DATASET INFO ==")
df.info(memory_usage='deep')

## 2.3 Column Analysis and Categorization

# Get actual columns from the dataframe
actual_columns = df.columns.tolist()

# Categorize columns based on what's actually in the dataset
demographic_cols = [col for col in actual_columns if any(x in col.lower() for x in
['household', 'county', 'area', 'gender', 'age', 'education', 'm'])

financial_behavior_cols = [col for col in actual_columns if any(x in col.lower() for x in
['expenditure', 'income', 'savings', 'loan', 'investment'])

product_usage_cols = [col for col in actual_columns if any(x in col.lower() for x in
['usage', 'bank', 'mobile', 'sacco', 'insurance', 'pension'])]

investment_cols = [col for col in actual_columns if 'investment' in col.lower()]

print("== COLUMN CATEGORIZATION ==")
print(f"Demographic columns ({len(demographic_cols)}): {demographic_cols}")
print(f"Financial behavior columns ({len(financial_behavior_cols)}): {financial_behavior_cols}")
print(f"Product usage columns ({len(product_usage_cols)}): {product_usage_cols}")
print(f"Investment-specific columns ({len(investment_cols)}): {investment_cols}")

## 2.4 Missing Values Analysis

missing_analysis = pd.DataFrame({
    'Column': df.columns,
    'Missing_Count': df.isnull().sum(),
    'Missing_Percentage': (df.isnull().sum() / len(df)) * 100
}).sort_values('Missing_Percentage', ascending=False)

```

```
print("\n==== MISSING VALUES ANALYSIS ===")  
print(missing_analysis.head(20))  
  
# Visualize missing values - only if there are missing values  
if missing_analysis['Missing_Percentage'].max() > 0:  
    plt.figure(figsize=(12, 8))  
    missing_cols = missing_analysis[missing_analysis['Missing_Percentage'] > 0].head()  
    if len(missing_cols) > 0:  
        plt.barh(missing_cols['Column'], missing_cols['Missing_Percentage'])  
        plt.xlabel('Missing Percentage (%)')  
        plt.title('Top 20 Columns with Missing Values')  
        plt.tight_layout()  
        plt.show()  
else:  
    print("No missing values found in the dataset")  
  
## 2.5 Target Variable Analysis (Investment Usage)  
  
investment_col = None  
for col in df.columns:  
    if 'investment' in col.lower() and 'usage' in col.lower():  
        investment_col = col  
        break  
  
if investment_col:  
    print(f"\n==== INVESTMENT USAGE DISTRIBUTION ({investment_col}) ===")  
  
    # Clean the data first  
    investment_data = df[investment_col].dropna()  
  
    if len(investment_data) > 0:  
        investment_dist = investment_data.value_counts()  
        print(investment_dist)  
  
        # Safe plotting with validation  
        if len(investment_dist) > 0:  
            safe_plot(investment_dist, 'bar',  
                      'Distribution of Investment Usage',  
                      'Investment Usage', 'Count')  
        else:  
            print("No valid investment usage data to plot")  
    else:  
        print("No investment usage data available")  
else:  
    print("Investment usage column not found in dataset")  
  
## 2.6 Demographic Analysis  
  
# Age distribution  
age_col = None  
for col in df.columns:  
    if 'age' in col.lower():  
        age_col = col  
        break  
  
if age_col and age_col in df.columns:
```

```
age_data = df[age_col].dropna()

if len(age_data) > 0:
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    age_data.hist(bins=30, edgecolor='black')
    plt.title('Age Distribution')
    plt.xlabel('Age')
    plt.ylabel('Frequency')

    plt.subplot(1, 2, 2)
    plt.boxplot(age_data)
    plt.title('Age Box Plot')
    plt.ylabel('Age')

    plt.tight_layout()
    plt.show()

print("Age statistics:")
print(age_data.describe())

# Gender distribution
gender_col = None
for col in df.columns:
    if 'gender' in col.lower():
        gender_col = col
        break

if gender_col and gender_col in df.columns:
    gender_data = df[gender_col].dropna()
    if len(gender_data) > 0:
        gender_dist = gender_data.value_counts()
        if len(gender_dist) > 0:
            safe_plot(gender_dist, 'pie', 'Gender Distribution')

# Geographic distribution
county_col = None
for col in df.columns:
    if 'county' in col.lower():
        county_col = col
        break

if county_col and county_col in df.columns:
    county_data = df[county_col].dropna()
    if len(county_data) > 0:
        county_dist = county_data.value_counts().head(10)
        if len(county_dist) > 0:
            safe_plot(county_dist, 'bar',
                      'Top 10 Counties by Respondent Count',
                      'County', 'Count')

# Education distribution
education_col = None
for col in df.columns:
    if 'education' in col.lower():
```

```

        education_col = col
        break

    if education_col and education_col in df.columns:
        education_data = df[education_col].dropna()
        if len(education_data) > 0:
            education_dist = education_data.value_counts()
            if len(education_dist) > 0:
                safe_plot(education_dist, 'bar', 'Education Level Distribution',
                           'Education Level', 'Count')

## 2.7 Financial Variables Analysis

# Find financial columns
financial_vars = []
for col in df.columns:
    if any(x in col.lower() for x in ['expenditure', 'income', 'salary']):
        financial_vars.append(col)

if financial_vars:
    fig, axes = plt.subplots(1, len(financial_vars), figsize=(15, 5))
    if len(financial_vars) == 1:
        axes = [axes]

    for i, var in enumerate(financial_vars):
        try:
            # Clean data and remove outliers for visualization
            clean_data = df[var].dropna()
            if len(clean_data) > 0:
                Q1 = clean_data.quantile(0.25)
                Q3 = clean_data.quantile(0.75)
                IQR = Q3 - Q1
                lower_bound = Q1 - 1.5 * IQR
                upper_bound = Q3 + 1.5 * IQR

                filtered_data = clean_data[(clean_data >= lower_bound) & (clean_dat

                    if len(filtered_data) > 0:
                        axes[i].hist(filtered_data, bins=30, edgecolor='black')
                        axes[i].set_title(f'{var} Distribution (Outliers Removed)')
                        axes[i].set_xlabel(var)
                        axes[i].set_ylabel('Frequency')
        except Exception as e:
            print(f"Error plotting {var}: {str(e)}")

    plt.tight_layout()
    plt.show()

## 2.8 Product Usage Patterns

# Find usage columns
usage_products = []
for col in df.columns:
    if 'usage' in col.lower():
        usage_products.append(col)

```

```
if usage_products:
    usage_summary = pd.DataFrame()
    valid_products = []

    for product in usage_products:
        try:
            usage_data = df[product].dropna()
            if len(usage_data) > 0:
                usage_counts = usage_data.value_counts()
                if len(usage_counts) > 0:
                    usage_summary[product] = usage_counts
                    valid_products.append(product)
        except Exception as e:
            print(f"Error processing {product}: {str(e)}")

    if len(valid_products) > 0:
        print("\n==== PRODUCT USAGE SUMMARY ===")
        print(usage_summary)

    # Visualize usage patterns
    n_products = min(4, len(valid_products)) # Max 4 subplots
    if n_products > 0:
        fig, axes = plt.subplots(2, 2, figsize=(15, 10))
        axes = axes.ravel()

        for i, product in enumerate(valid_products[:n_products]):
            try:
                product_data = df[product].dropna().value_counts()
                if len(product_data) > 0:
                    axes[i].bar(range(len(product_data)), product_data.values)
                    axes[i].set_title(f'{product} Usage')
                    axes[i].set_xticks(range(len(product_data)))
                    axes[i].set_xticklabels(product_data.index, rotation=45)
            except Exception as e:
                print(f"Error plotting {product}: {str(e)}")

        plt.tight_layout()
        plt.show()

## 2.9 Correlation Analysis

# Select numerical columns for correlation analysis
numerical_cols = df.select_dtypes(include=[np.number]).columns.tolist()

if len(numerical_cols) > 1:
    try:
        # Remove columns with all NaN values
        valid_numerical_cols = []
        for col in numerical_cols:
            if not df[col].isnull().all():
                valid_numerical_cols.append(col)

        if len(valid_numerical_cols) > 1:
            correlation_matrix = df[valid_numerical_cols].corr()

            # Plot correlation heatmap
```

```

plt.figure(figsize=(12, 10))
mask = np.triu(np.ones_like(correlation_matrix, dtype=bool))
sns.heatmap(correlation_matrix, mask=mask, cmap='coolwarm',
            center=0, fmt='.2f', square=True, annot=True)
plt.title('Correlation Matrix of Numerical Variables')
plt.tight_layout()
plt.show()

# Find highly correlated variables
high_corr_pairs = []
for i in range(len(correlation_matrix.columns)):
    for j in range(i+1, len(correlation_matrix.columns)):
        corr_value = correlation_matrix.iloc[i, j]
        if not np.isnan(corr_value) and abs(corr_value) > 0.7: # High
            high_corr_pairs.append((correlation_matrix.columns[i],
                                      correlation_matrix.columns[j],
                                      corr_value))

if high_corr_pairs:
    print("\n== HIGH CORRELATION PAIRS (|r| > 0.7) ==")
    for pair in high_corr_pairs[:10]: # Show top 10
        print(f"{pair[0]} <-> {pair[1]}: {pair[2]:.3f}")
else:
    print("\n== No highly correlated pairs found (|r| > 0.7) ==")

except Exception as e:
    print(f"Error in correlation analysis: {str(e)}")

## 2.10 Data Quality Assessment

print("\n== DATA QUALITY ASSESSMENT ==")

# Check for duplicates
duplicate_count = df.duplicated().sum()
print(f"Duplicate rows: {duplicate_count}")

# Check for inconsistencies in categorical variables
categorical_cols = df.select_dtypes(include=['object']).columns
for col in categorical_cols[:5]: # Check first 5 categorical columns
    try:
        unique_values = df[col].nunique()
        print(f"{col}: {unique_values} unique values")
        if unique_values < 20: # Show values if not too many
            unique_vals = df[col].dropna().unique()
            print(f" Values: {unique_vals}")
    except Exception as e:
        print(f"Error analyzing {col}: {str(e)}")

# Data completeness
completeness = (df.count() / len(df)) * 100
print(f"\nData Completeness by Column:")
for col in completeness.index[:10]: # Show first 10
    print(f" {col}: {completeness[col]:.1f}%")

## 2.11 Investment Usage Analysis by Demographics

```

```

if investment_col:
    print(f"\n==== INVESTMENT USAGE BY DEMOGRAPHICS ===")

    # By Age Groups
    if age_col:
        df['age_group'] = pd.cut(df[age_col], bins=[0, 25, 35, 45, 55, 100],
                                 labels=['18-25', '26-35', '36-45', '46-55', '55+'])
        age_investment = pd.crosstab(df['age_group'], df[investment_col])
        if len(age_investment) > 0:
            print("Investment Usage by Age Group:")
            print(age_investment)

        # Plot
        age_investment.plot(kind='bar', figsize=(10, 6))
        plt.title('Investment Usage by Age Group')
        plt.xticks(rotation=45)
        plt.legend(title='Investment Usage')
        plt.tight_layout()
        plt.show()

    # By Education Level
    if education_col:
        education_investment = pd.crosstab(df[education_col], df[investment_col])
        if len(education_investment) > 0:
            print("\nInvestment Usage by Education Level:")
            print(education_investment)

        # Plot
        education_investment.plot(kind='bar', figsize=(10, 6))
        plt.title('Investment Usage by Education Level')
        plt.xticks(rotation=45)
        plt.legend(title='Investment Usage')
        plt.tight_layout()
        plt.show()

## 2.12 Initial Insights and Hypotheses

print("\n==== INITIAL INSIGHTS ===")
print("1. Investment Usage Patterns:")
print("    - Analyze relationship between demographics and investment usage")
print("    - Urban vs rural investment behavior differences")
print("    - Income level impact on investment choices")

print("\n2. Product Cross-selling Opportunities:")
print("    - Users with savings might be good candidates for investments")
print("    - Mobile money users could be introduced to digital investments")
print("    - Higher educated individuals show different usage patterns")

print("\n3. Risk Profile Indicators:")
print("    - Age and investment risk tolerance relationship")
print("    - Education level and sophisticated product usage")
print("    - Income stability and long-term investment products")

## 2.13 Data Understanding Summary

print("\n==== DATA UNDERSTANDING SUMMARY ===")

```

```
print(f"\n Dataset contains {df.shape[0]}:,} records and {df.shape[1]}:,} features")
print(f"\n Mixed data types: numerical, categorical, and binary indicators")
print(f"\n Geographic coverage across multiple counties")
print(f"\n Comprehensive financial behavior tracking")
print(f"\n Investment usage can serve as target variable")
print(f"\n Multiple product usage variables for recommendation features")

# Save processed understanding for next phase
try:
    df.to_csv('data_understanding_output.csv', index=False)
    print(f"\n Data understanding output saved to 'data_understanding_output.csv'")
except:
    print("Warning: Could not save data understanding output")

print("\nNext Phase: Data Preparation")
print("- Clean and preprocess the data")
print("- Create meaningful features for recommendation system")
print("- Handle missing values and outliers")
print("- Prepare data for modeling")

# Create summary statistics
summary_stats = {
    'total_records': len(df),
    'total_features': len(df.columns),
    'missing_values': df.isnull().sum().sum(),
    'duplicate_records': duplicate_count,
    'numerical_features': len(df.select_dtypes(include=[np.number]).columns),
    'categorical_features': len(df.select_dtypes(include=['object']).columns),
    'investment_usage_rate': df[investment_col].value_counts().get('Yes', 0) / len(
}
print(f"\n==== FINAL STATISTICS ====")
for key, value in summary_stats.items():
    if isinstance(value, float):
        print(f"{key}: {value:.2f}")
    else:
        print(f"{key}: {value}")
```

```

Dataset shape: (20871, 392)
Memory usage: 400.76 MB
==== DATASET OVERVIEW ====
Number of rows: 20,871
Number of columns: 392

Data types:
object      305
float64     58
int64       29
Name: count, dtype: int64

==== SAMPLE DATA ====
    householdid   county area_type gender age_of_respondent \
0      107141431  garissa    urban   male            29
1      10712933  garissa    urban   male            60
2     140173183   busia     urban female           35
3     122137153  kiambu    urban   male            24
4     121193116 murang'a    urban female           20

    no_of_household_mebers  livelihoodcat Quintiles Education \
0                  5      dependent    fourth  tertiary
1                 11        other      second primary
2                  2  casual worker    fourth  primary
3                  1  casual worker   middle secondary
4                  1      dependent   highest secondary

    Marital ... insurance_including_NHIF_use \
0  married/living with partner ...             never used
1  married/living with partner ...             never used
2      divorced/separated ...                never used
3  single/never married ...                  never used
4  single/never married ...                  never used

    All_Insurance_excluding_NHIF_use          PWD Latitude Longitude \
0              never used without disability -0.435423 39.636586
1              never used without disability  0.058794 40.305006
2              never used without disability  0.636836 34.277390
3              never used without disability -1.251917 36.719076
4              never used without disability -0.795820 37.131085

    has_account has_savings has_credit has_mobile receives_remittance
0            0            0            0            0            0
1            0            0            0            0            0
2            0            0            0            0            0
3            0            0            0            0            0
4            0            0            0            0            0

[5 rows x 392 columns]

==== DATASET INFO ====
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20871 entries, 0 to 20870
Columns: 392 entries, householdid to receives_remittance
dtypes: float64(58), int64(29), object(305)
memory usage: 400.8 MB

```

==== COLUMN CATEGORIZATION ====

Demographic columns (50): ['householdid', 'county', 'area_type', 'gender', 'age_of_respondent', 'no_of_household_members', 'Education', 'Marital', 'Pension_usage', 'Pension_Usage_without_NSSF', 'Mobile_bank_usage', 'Non_microfinance_bank_usage', 'Microfinance_usage', 'Sacco_usage', 'Digital_currencies_assets_usage', 'Investment_usage', 'Informal_group_usage', 'Digital_credit_usage', 'Post_bank_usage', 'Informal_usage', 'Savings_usage', 'Loan_usage', 'bank_usage', 'bank_usage1', 'bank_usage3', 'bank_usage2', 'PDTs_Usage', 'Other_Sacco_USage', 'Informal_usage_1', 'Investment_usage_fnl', 'Microfinance_usage_fnl', 'bank_usage_overall', 'Loan_usage_fnl', 'formal_service_usage', 'Digital_credit_usage_fnl', 'manage_day2day', 'managedebt_perception', 'saving_for_education', 'saving_for_old_age', 'Edu_within_household_credit', 'Edu_outside_household_credit', 'Day2day_household_needs_credit', 'Acquire_household_goods_credit', 'For_old_age_credit', 'chama_usage', 'shopkeeper_usage', 'friends_family_usage', 'employer_money_lender_usage', 'MFB_Usage', 'employer_money_lender_usage1']

Financial behavior columns (40): ['CalcExpenditure', 'total_monthly_expenditure', 'digital_loans', 'Investment_usage', 'Savings_usage', 'Savings_formal', 'Savings_informal', 'Savings_excluded', 'Loan_usage', 'tot_savings', 'formal_digital_loan', 'formal_non_digital_loan', 'tot_formal_digit_loan_sav', 'tot_formal_non_digit_loan_sav', 'monthly_income', 'bank_savings', 'mfi_savings', 'mobile_banking_savings', 'mobile_money_savings', 'sacco_savings', 'chama_savings', 'friend_group_savings', 'family_friend_savings', 'secret_place_savings', 'digital_app_savings', 'Investment_usage_fnl', 'Loan_usage_fnl', 'Mobile_banking_loan', 'Fuliza_loan', 'digital_app_loan', 'hustler_fund_loan', 'formal_digital_loan_fnl', 'formal_non_digital_loan_fnl', 'tot_formal_digit_loan_sav_fnl', 'tot_formal_nondigit_loan_sav_fnl', 'digital_app_loan_fnl', 'overall_climate_investment', 'Reason_for_saving_loan_access', 'barriers_savings', 'has_savings']

Product usage columns (107): ['All_Insurance_excluding_NHIF', 'All_Insurance_including_NHIF', 'All_Insurance_and_Pension', 'Pension_usage', 'Pension_Usage_without_NSSF', 'Mobile_bank_usage', 'Non_microfinance_bank_usage', 'Microfinance_usage', 'Sacco_usage', 'Digital_currencies_assets_usage', 'Investment_usage', 'Informal_group_usage', 'Digital_credit_usage', 'Post_bank_usage', 'Informal_usage', 'Savings_usage', 'Loan_usage', 'mobile', 'mobile_money_access', 'mobile_money_access_2', 'bank_usage', 'bank_usage1', 'bank_usage3', 'bank_usage2', 'PDTs_Usage', 'Other_Sacco_USage', 'Regulated_Saccos', 'Non_Regulated_Saccos', 'Informal_usage_1', 'bank_savings', 'mobile_banking_savings', 'mobile_money_savings', 'sacco_savings', 'Investment_usage_fnl', 'Pension_including_NSSF', 'Pension_excluding_NSSF', 'Microfinance_usage_fnl', 'mobile_money_access_primary', 'mobile_money_access1', 'bank_usage_overall', 'Traditional_bank', 'Commercial_bank', 'barriers_mobile_money', 'barriers_mobile_bank', 'barriers_pension_fnl', 'Insurance_excluding_NHIF_fnl', 'Insurance_including_NHIF_fnl', 'All_Insurance_and_Pension_fnl', 'Loan_usage_fnl', 'Mobile_banking_loan', 'mobile_ownership_1', 'mobile_ownership_2', 'formal_service_usage', 'Digital_credit_usage_fnl', 'mobile_bank_use', 'pension_use', 'pension_use_excl_NSSF', 'bank_use', 'commercial_bank_use', 'mobile_money_use', 'sacco_use', 'save_bank', 'save_mobile_money', 'save_sacco', 'chama_usage', 'shopkeeper_usage', 'friends_family_usage', 'employer_money_lender_usage', 'MFB_Usage', 'barriers_insurance_fnl', 'barriers_bank', 'barriers_traditional_bank_fnl', 'barriers_sacco', 'employer_money_lender_usage1', 'insurance_delayed_fnl', 'insurance_datashared_fnl', 'insurance_nottransparent_fnl', 'insurance_collapses_fnl', 'insurance_issues_fnl', 'bank_unexpectedcharges', 'bank_moneylest', 'bank_downtime', 'bank_unethicalrecovery', 'bank_issues', 'sacco_moneylest', 'sacco_downtime', 'sacco_unethicalrecovery', 'sacco_issues', 'mobilemoney_unexpectedcharges', 'mobilemoney_moneylest', 'mobilemoney_downtime', 'mobilemoney_float', 'mobilemoney_datashard', 'mobilemoney_issues', 'mobilebank_unexpectedcharges', 'mobilebank_moneylest', 'mobilebank_downtime', 'mobilebank_unethicalrecovery', 'mobilebank_issues', 'insurance_satisfaction', 'bank_satisfaction', 'sacco_satisfaction', 'mobilemoney_satisfaction', 'mobilebank_satisfaction', 'insurance_including_NHIF_use', 'All_Insurance_excluding_NHIF_use', 'has_mobile']

Investment-specific columns (3): ['Investment_usage', 'Investment_usage_fnl', 'overall_climate_investment']

==== MISSING VALUES ANALYSIS ===

	Column \
householdid	householdid
Another_persons_business_credit	Another_persons_business_credit
Premise_land_for_business_credit	Premise_land_for_business_credit
Working_capital_business_credit	Working_capital_business_credit
Expand_business_credit	Expand_business_credit
Start_new_business_credit	Start_new_business_credit
Day2day_household_needs_credit	Day2day_household_needs_credit
Edu_outside_household_credit	Edu_outside_household_credit
Edu_within_household_credit	Edu_within_household_credit
Emergencies_credit	Emergencies_credit
Reason_for_saving_only_option	Reason_for_saving_only_option
Reason_for_saving_recommended	Reason_for_saving_recommended
Reason_for_saving_my_paypoint	Reason_for_saving_my_paypoint
Reason_for_saving_loan_access	Reason_for_saving_loan_access
Reason_for_saving_privacy	Reason_for_saving_privacy
Reason_for_saving_comfortable	Reason_for_saving_comfortable
Reason_for_saving_high_interest	Reason_for_saving_high_interest
Reason_for_saving_trustworthy	Reason_for_saving_trustworthy
Reason_for_saving_convenient	Reason_for_saving_convenient
saving_safeKeeping	saving_safeKeeping

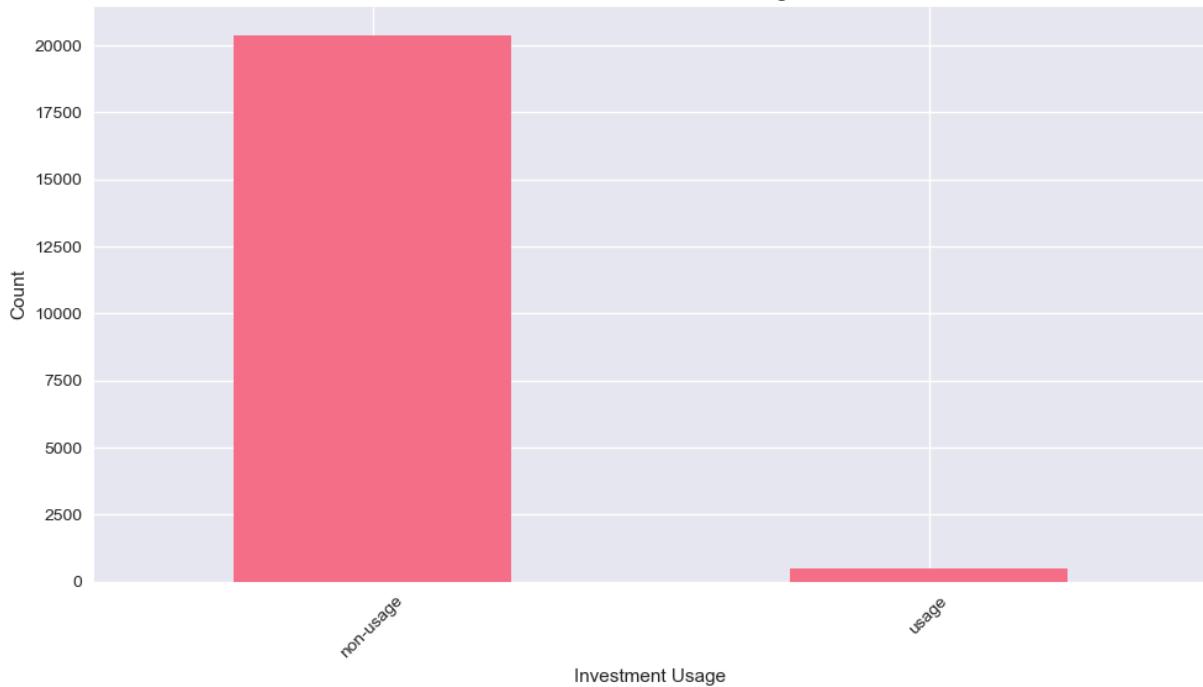
	Missing_Count	Missing_Percentage
householdid	0	0.0
Another_persons_business_credit	0	0.0
Premise_land_for_business_credit	0	0.0
Working_capital_business_credit	0	0.0
Expand_business_credit	0	0.0
Start_new_business_credit	0	0.0
Day2day_household_needs_credit	0	0.0
Edu_outside_household_credit	0	0.0
Edu_within_household_credit	0	0.0
Emergencies_credit	0	0.0
Reason_for_saving_only_option	0	0.0
Reason_for_saving_recommended	0	0.0
Reason_for_saving_my_paypoint	0	0.0
Reason_for_saving_loan_access	0	0.0
Reason_for_saving_privacy	0	0.0
Reason_for_saving_comfortable	0	0.0
Reason_for_saving_high_interest	0	0.0
Reason_for_saving_trustworthy	0	0.0
Reason_for_saving_convenient	0	0.0
saving_safeKeeping	0	0.0

No missing values found in the dataset

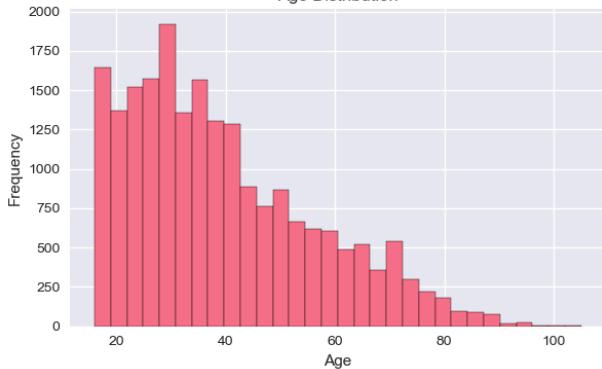
==== INVESTMENT USAGE DISTRIBUTION (Investment_usage) ===

Investment_usage	
non-usage	20390
usage	481
Name: count, dtype: int64	

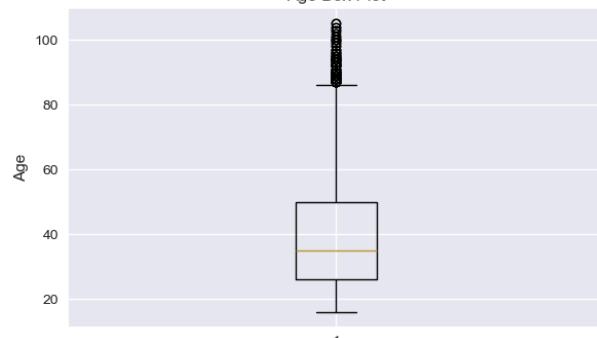
Distribution of Investment Usage



Age Distribution



Age Box Plot



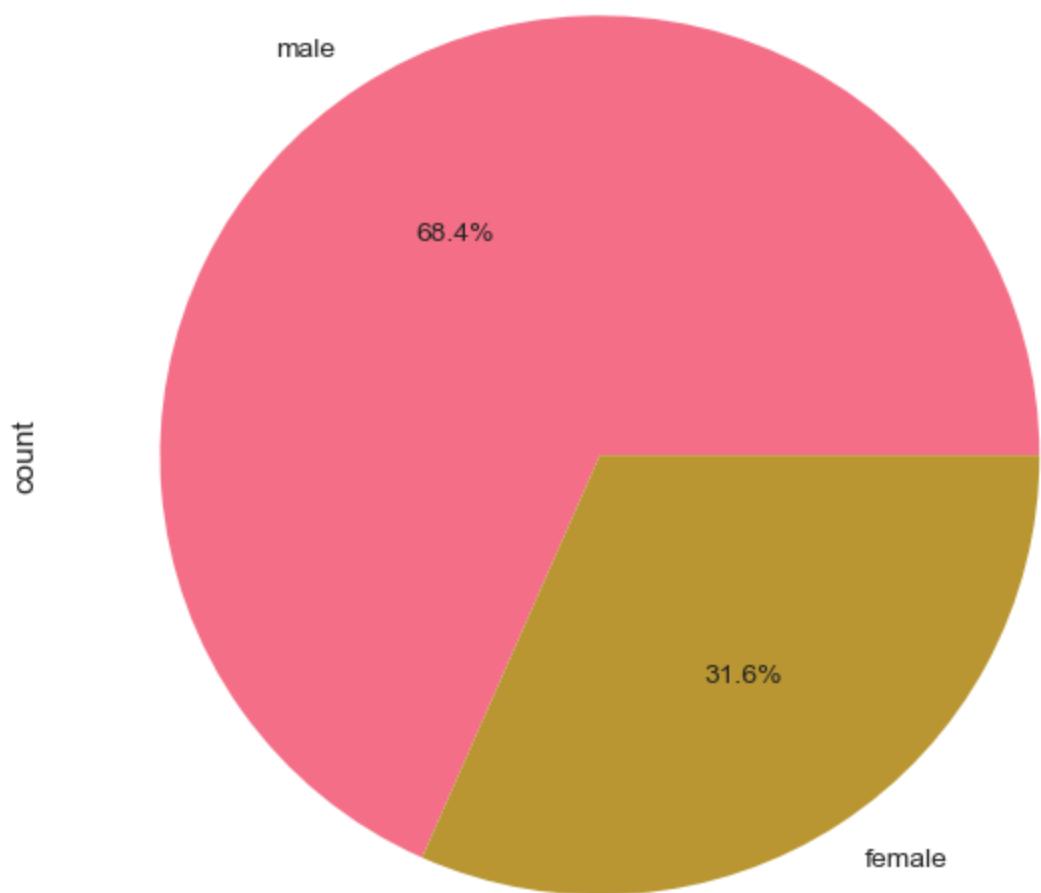
Age statistics:

```

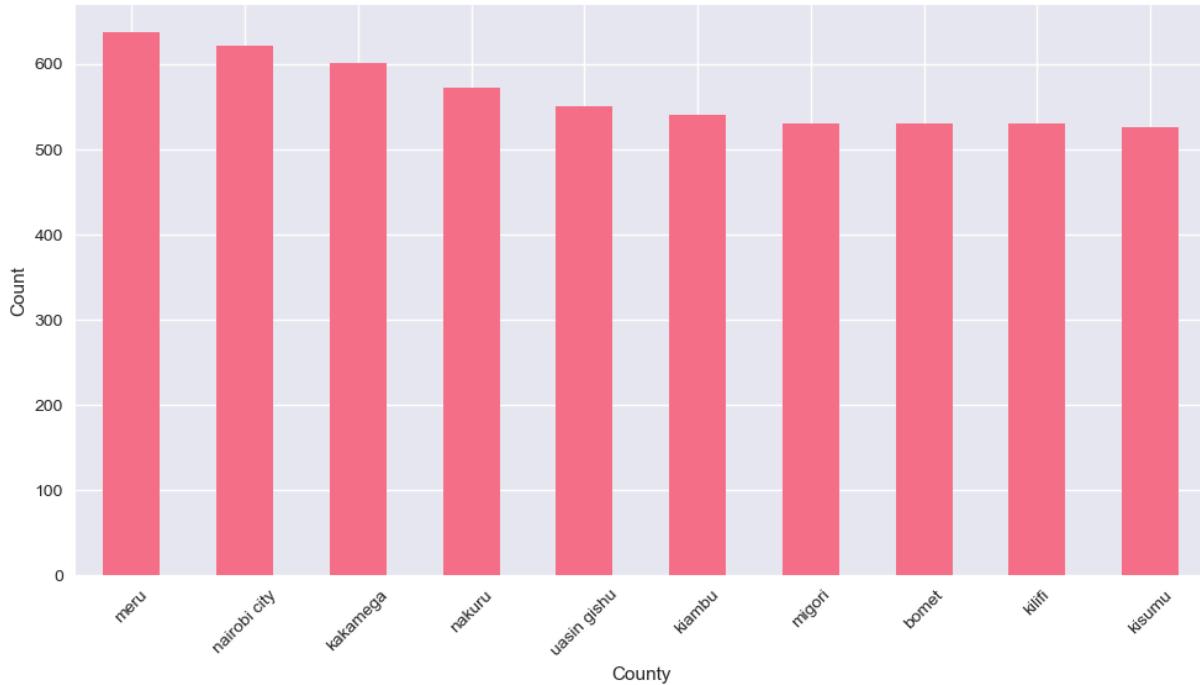
count      20871.000000
mean       39.271429
std        17.210021
min        16.000000
25%        26.000000
50%        35.000000
75%        50.000000
max       105.000000
Name: age_of_respondent, dtype: float64

```

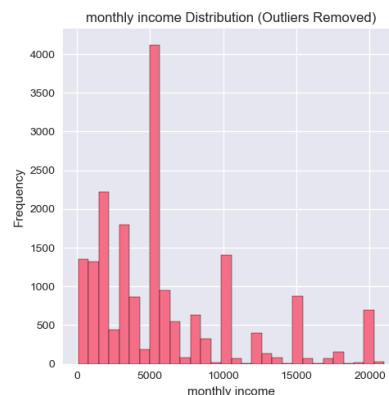
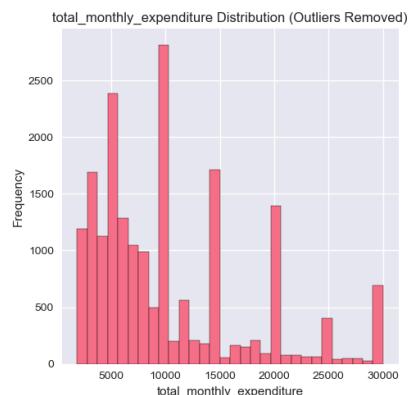
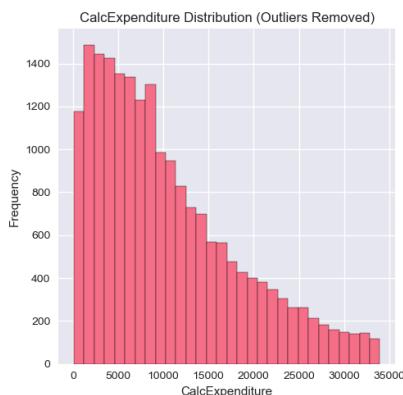
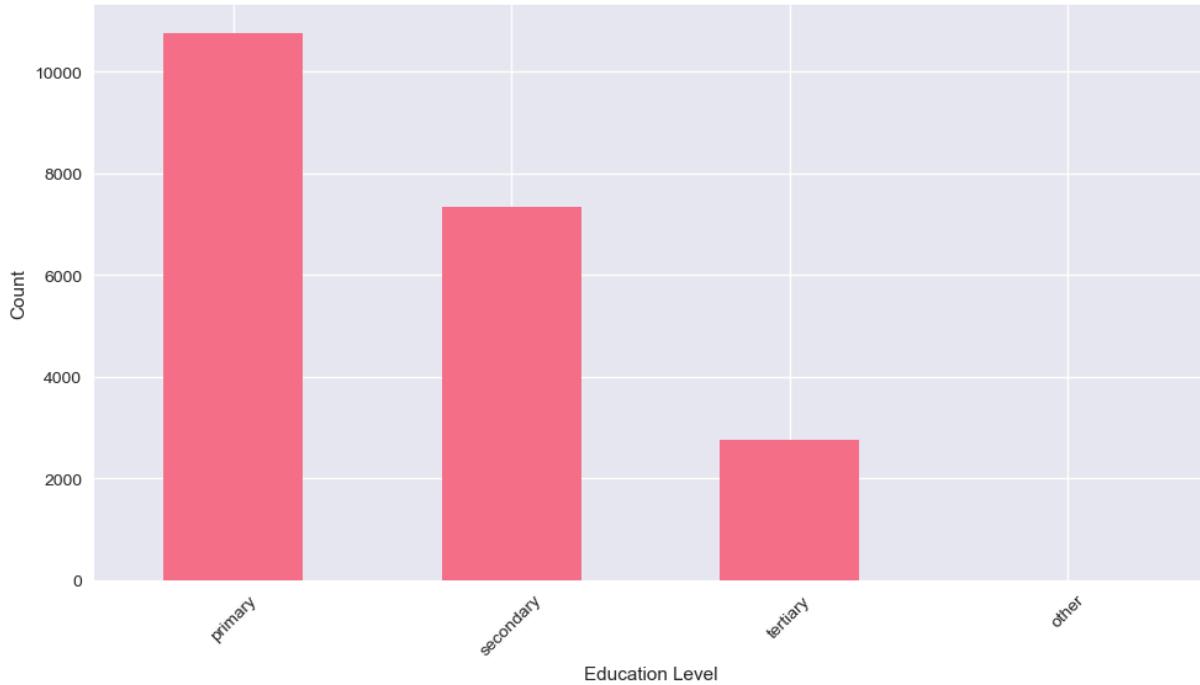
Gender Distribution



Top 10 Counties by Respondent Count



Education Level Distribution



```
==== PRODUCT USAGE SUMMARY ====
      Pension_usage  Pension_Usage_without_NSSF  Mobile_bank_usage  \
Pension_usage
non-usage          18966                      20541                  15363
usage              1905                       330                   5508

      Non_microfinance_bank_usage  Microfinance_usage  Sacco_usage  \
Pension_usage
non-usage           20823                      20692                  18433
usage               48                        179                  2438

      Digital_currencies_assets_usage  Investment_usage  \
Pension_usage
non-usage            20807                      20390
usage                64                        481

      Informal_group_usage  Digital_credit_usage  ...  \
Pension_usage
non-usage            14511                      20502      ...
usage                6360                      369      ...

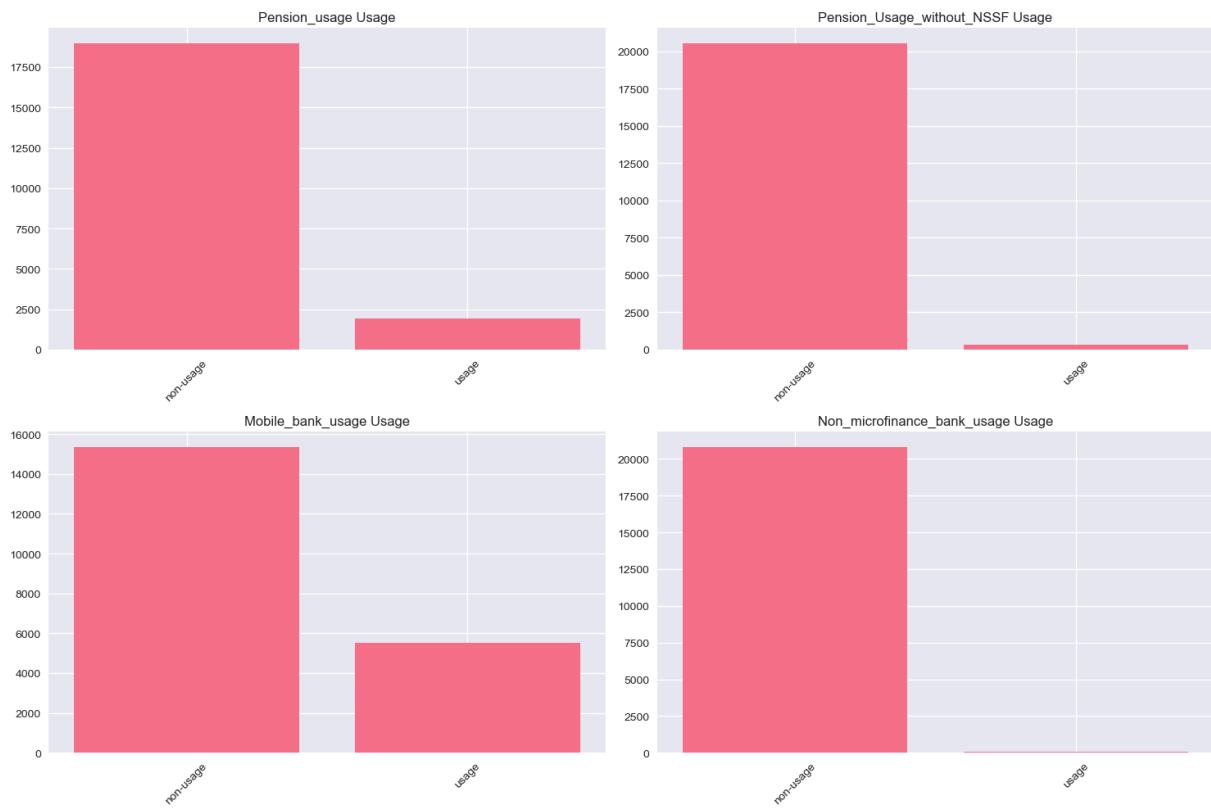
      bank_usage_overall  Loan_usage_fnl  formal_service_usage  \
Pension_usage
non-usage            11373                      8096                  2647
usage                9498                      12775                 18224

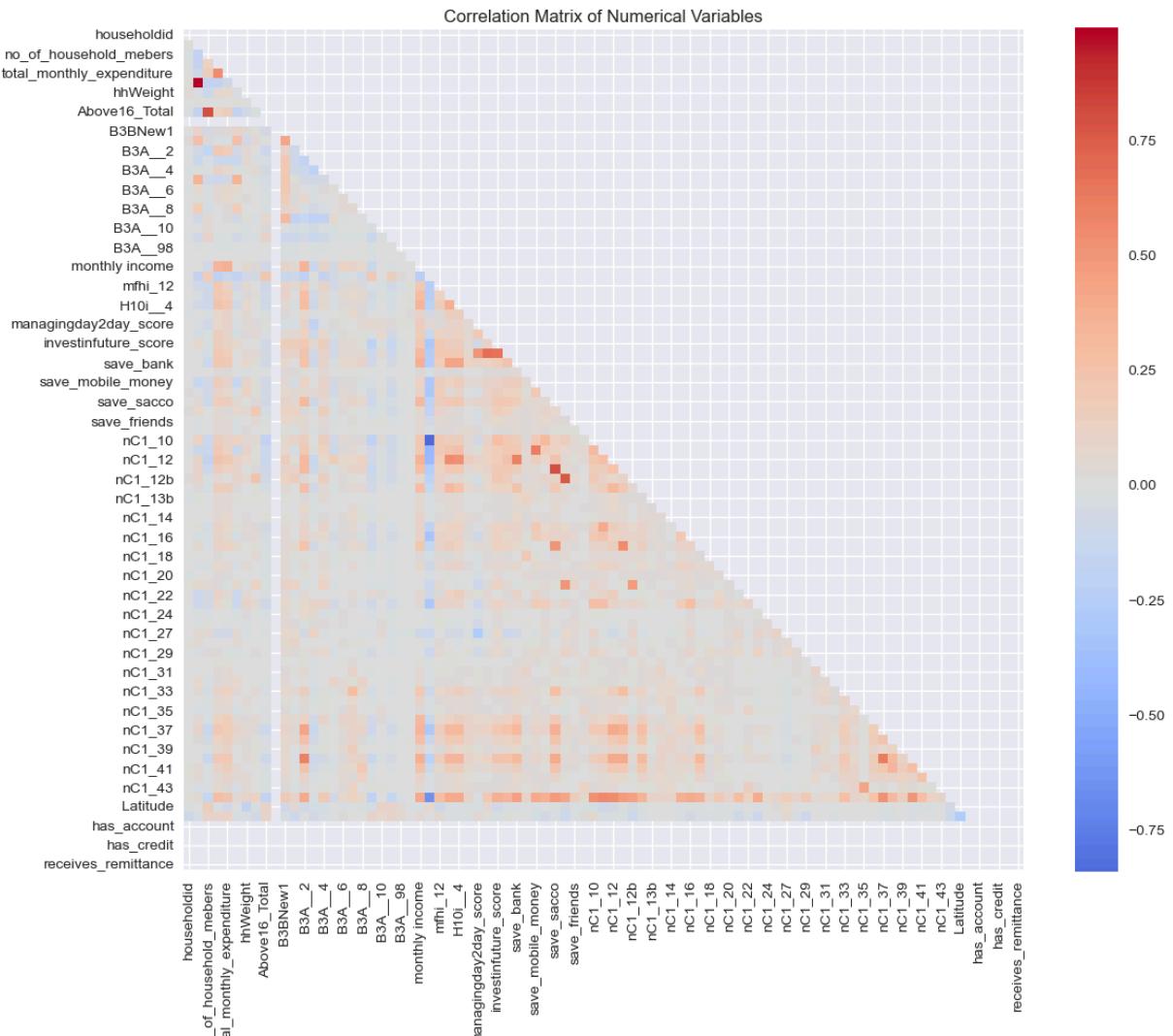
      Digital_credit_usage_fnl  chama_usage  shopkeeper_usage  \
Pension_usage
non-usage            13620                      NaN                  NaN
usage                7251                      5766.0                5754.0

      friends_family_usage  employer_money_lender_usage  MFB_Usage  \
Pension_usage
non-usage             NaN                      NaN                  NaN
usage                4320.0                    3203.0                95.0

      employer_money_lender_usage1
Pension_usage
non-usage             NaN
usage                320.0
```

[2 rows x 33 columns]





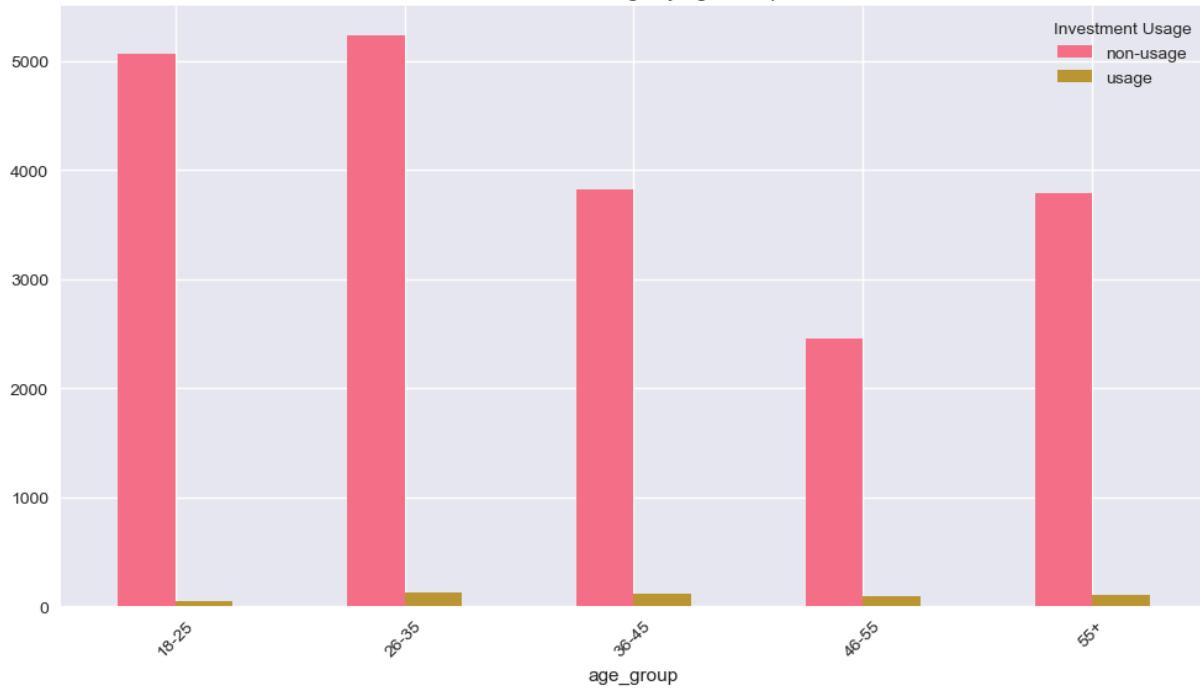
```
==== HIGH CORRELATION PAIRS (|r| > 0.7) ====
age_of_respondent <-> no_respondent_per_hh: 0.993
no_of_household_mebers <-> Above16_Total: 0.800
Access_fnl_1 <-> nC1_10: -0.843
save_sacco <-> nC1_12a: 0.797
save_chama <-> nC1_12b: 0.761

==== DATA QUALITY ASSESSMENT ====
Duplicate rows: 0
county: 47 unique values
area_type: 2 unique values
    Values: ['urban' 'rural']
gender: 2 unique values
    Values: ['male' 'female']
livelihoodcat: 6 unique values
    Values: ['dependent' 'other' 'casual worker' 'own business' 'agriculture'
    'employed']
Quintiles: 5 unique values
    Values: ['fourth' 'second' 'middle' 'highest' 'lowest']

Data Completeness by Column:
householdid: 100.0%
county: 100.0%
area_type: 100.0%
gender: 100.0%
age_of_respondent: 100.0%
no_of_household_mebers: 100.0%
livelihoodcat: 100.0%
Quintiles: 100.0%
Education: 100.0%
Marital: 100.0%

==== INVESTMENT USAGE BY DEMOGRAPHICS ====
Investment Usage by Age Group:
Investment_usage  non-usage  usage
age_group
18-25          5071      47
26-35          5241     120
36-45          3823     119
46-55          2456      89
55+            3790     106
```

Investment Usage by Age Group



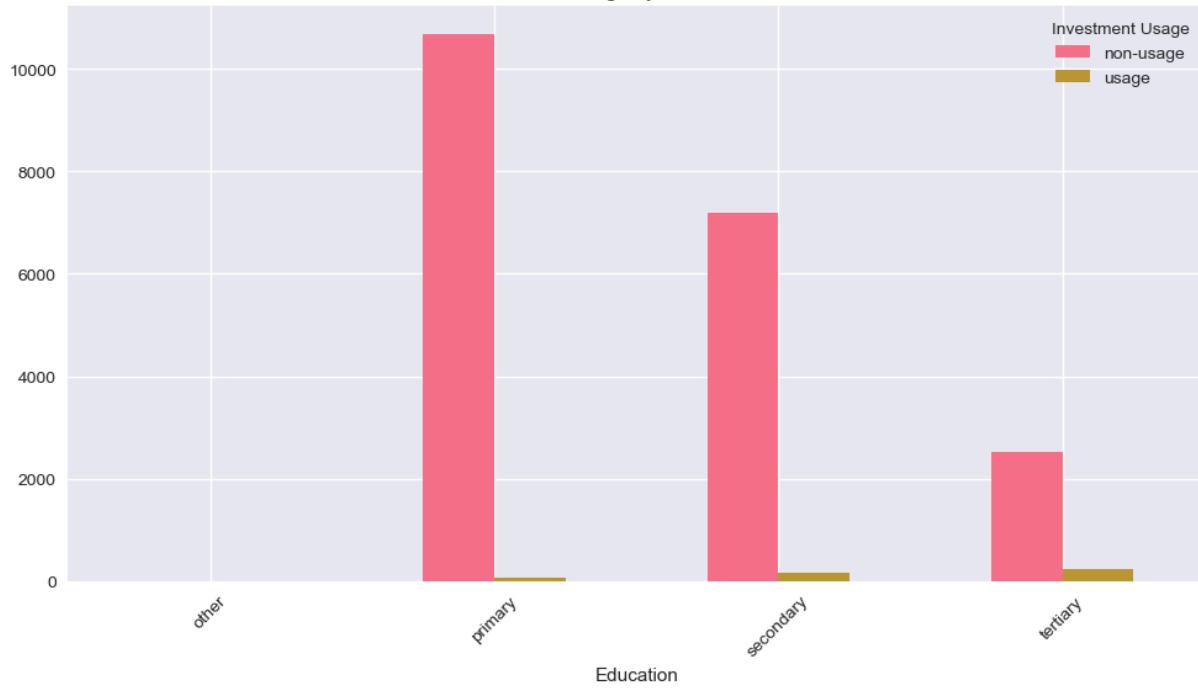
Investment Usage by Education Level:

Investment_usage non-usage usage

Education

other	9	0
primary	10684	78
secondary	7183	162
tertiary	2514	241

Investment Usage by Education Level



==== INITIAL INSIGHTS ====

1. Investment Usage Patterns:

- Analyze relationship between demographics and investment usage
- Urban vs rural investment behavior differences
- Income level impact on investment choices

2. Product Cross-selling Opportunities:

- Users with savings might be good candidates for investments
- Mobile money users could be introduced to digital investments
- Higher educated individuals show different usage patterns

3. Risk Profile Indicators:

- Age and investment risk tolerance relationship
- Education level and sophisticated product usage
- Income stability and long-term investment products

==== DATA UNDERSTANDING SUMMARY ====

- ✓ Dataset contains 20,871 records and 393 features
- ✓ Mixed data types: numerical, categorical, and binary indicators
- ✓ Geographic coverage across multiple counties
- ✓ Comprehensive financial behavior tracking
- ✓ Investment usage can serve as target variable
- ✓ Multiple product usage variables for recommendation features
- ✓ Data understanding output saved to 'data_understanding_output.csv'

Next Phase: Data Preparation

- Clean and preprocess the data
- Create meaningful features for recommendation system
- Handle missing values and outliers
- Prepare data for modeling

==== FINAL STATISTICS ====

```
total_records: 20871
total_features: 393
missing_values: 9
duplicate_records: 0
numerical_features: 87
categorical_features: 305
investment_usage_rate: 0.00
```

In [3]: # Phase 3: Data Preparation - Investment Recommendation System

```
## 3.1 Load and Initial Setup

# Load the raw data
try:
    df = pd.read_csv('final_refined.csv')
    print(f"Original dataset shape: {df.shape}")
except FileNotFoundError:
    df = pd.read_csv('data_understanding_output.csv')
    print(f"Loaded data from understanding phase: {df.shape}")

# Create a backup of original data
df_original = df.copy()
```

```

## 3.2 Column Renaming for Readability

# Define column mapping for better readability
column_mapping = {
    'householdid': 'household_id',
    'area_type': 'location_type',
    'age_of_respondent': 'age',
    'no_of_household_mebers': 'household_size',
    'total_monthly_expenditure': 'monthly_expenditure',
    'monthly_income': 'monthly_income',
    'digital_loans': 'digital_loans_usage',
    'Investment_usage': 'investment_usage',
    'Mobile_bank_usage': 'mobile_banking',
    'Savings_usage': 'savings_usage',
    'Loan_usage': 'loan_usage',
    'Education': 'education_level',
    'Marital': 'marital_status',
    'NHIF': 'nhif_usage',
    'Sacco_usage': 'sacco_usage'
}

# Apply column renaming (only for columns that exist)
existing_mappings = {k: v for k, v in column_mapping.items() if k in df.columns}
df.rename(columns=existing_mappings, inplace=True)
print(f"Renamed {len(existing_mappings)} columns for better readability")

## 3.3 Missing Value Treatment

# Analyze missing values
missing_summary = pd.DataFrame({
    'column': df.columns,
    'missing_count': df.isnull().sum(),
    'missing_percent': (df.isnull().sum() / len(df)) * 100,
    'dtype': df.dtypes
})
missing_summary = missing_summary[missing_summary['missing_count'] > 0].sort_values

print("\n==== MISSING VALUES SUMMARY ===")
if len(missing_summary) > 0:
    print(missing_summary.head(10))
else:
    print("No missing values found")

# Define imputation strategies
def handle_missing_values(df):
    """Handle missing values with appropriate strategies"""

    # Categorical variables - fill with mode or 'unknown'
    categorical_cols = df.select_dtypes(include=['object']).columns
    for col in categorical_cols:
        if df[col].isnull().sum() > 0:
            mode_value = df[col].mode()
            if len(mode_value) > 0:
                df[col].fillna(mode_value[0], inplace=True)
            else:
                df[col].fillna('unknown', inplace=True)

```

```

# Numerical variables - use median for skewed data, mean for normal
numerical_cols = df.select_dtypes(include=[np.number]).columns
for col in numerical_cols:
    if df[col].isnull().sum() > 0:
        # Check skewness
        skewness = df[col].skew()
        if abs(skewness) > 1: # Highly skewed
            df[col].fillna(df[col].median(), inplace=True)
        else:
            df[col].fillna(df[col].mean(), inplace=True)

return df

df = handle_missing_values(df)
print(f"Missing values after treatment: {df.isnull().sum().sum()}")

## 3.4 Data Type Optimization

def optimize_dtypes(df):
    """Optimize data types to reduce memory usage"""

    # Convert object columns with limited unique values to category
    for col in df.select_dtypes(include=['object']).columns:
        unique_ratio = df[col].nunique() / len(df)
        if unique_ratio < 0.5: # Less than 50% unique values
            df[col] = df[col].astype('category')

    # Convert integer columns to smaller types where possible
    for col in df.select_dtypes(include=['int64']).columns:
        col_min = df[col].min()
        col_max = df[col].max()

        if pd.isna(col_min) or pd.isna(col_max):
            continue

        if col_min >= -128 and col_max <= 127:
            df[col] = df[col].astype('int8')
        elif col_min >= -32768 and col_max <= 32767:
            df[col] = df[col].astype('int16')
        elif col_min >= -2147483648 and col_max <= 2147483647:
            df[col] = df[col].astype('int32')

    # Convert float columns to float32 where possible
    for col in df.select_dtypes(include=['float64']).columns:
        df[col] = pd.to_numeric(df[col], downcast='float')

    return df

print("Memory usage before optimization:", df.memory_usage(deep=True).sum() / 1024*)
df = optimize_dtypes(df)
print("Memory usage after optimization:", df.memory_usage(deep=True).sum() / 1024**)

## 3.5 Feature Engineering

def safe_count_usage(row, columns):

```

```

"""Safely count usage values in a row"""
count = 0
for col in columns:
    if col not in row.index:
        continue
    value = row[col]
    if pd.isna(value):
        continue
    elif isinstance(value, str):
        if value.lower() in ['yes', 'usage', '1']:
            count += 1
    elif isinstance(value, (int, float)):
        if value == 1: # Assuming 1 means usage
            count += 1
return count

def create_features(df):
    """Create meaningful features for investment recommendation"""

    # Age groups
    if 'age' in df.columns:
        df['age_group'] = pd.cut(df['age'],
                                bins=[0, 25, 35, 45, 55, 100],
                                labels=['18-25', '26-35', '36-45', '46-55', '55+'])

    # Income-to-expenditure ratio
    if 'monthly_income' in df.columns and 'monthly_expenditure' in df.columns:
        # Handle division by zero and negative values
        expenditure_safe = df['monthly_expenditure'].clip(lower=1)
        df['income_expenditure_ratio'] = df['monthly_income'] / expenditure_safe
        df['savings_potential'] = df['monthly_income'] - df['monthly_expenditure']

        # Handle division by zero for savings rate
        income_safe = df['monthly_income'].clip(lower=1)
        df['savings_rate'] = df['savings_potential'] / income_safe

        # Cap extreme values
        df['income_expenditure_ratio'] = df['income_expenditure_ratio'].clip(0, 10)
        df['savings_rate'] = df['savings_rate'].clip(-1, 1)

    # Financial sophistication score
    digital_services = ['mobile_banking', 'digital_loans_usage', 'digital_credit']
    existing_digital = [col for col in digital_services if col in df.columns]

    if existing_digital:
        df['digital_services_count'] = df[existing_digital].apply(
            lambda row: safe_count_usage(row, existing_digital), axis=1
        )
        df['digital_sophistication'] = df['digital_services_count'] / len(existing_digital)
    else:
        # Create default values if no digital services columns exist
        df['digital_services_count'] = 0
        df['digital_sophistication'] = 0

    # Product diversification score
    financial_products = ['savings_usage', 'loan_usage', 'investment_usage']

```

```

existing_products = [col for col in financial_products if col in df.columns]

if existing_products:
    df['products_used_count'] = df[existing_products].apply(
        lambda row: safe_count_usage(row, existing_products), axis=1
    )
    df['product_diversification'] = df['products_used_count'] / len(existing_products)
else:
    # Create default values
    df['products_used_count'] = 0
    df['product_diversification'] = 0

# Risk profile based on age and income
if 'age' in df.columns:
    df['risk_tolerance'] = np.where(df['age'] < 30, 'high',
                                    np.where(df['age'] < 50, 'medium', 'low'))
else:
    df['risk_tolerance'] = 'medium'

# Urban/rural investment readiness
if 'location_type' in df.columns:
    df['location_investment_readiness'] = np.where(
        df['location_type'].astype(str).str.lower() == 'urban', 1, 0
    )
else:
    df['location_investment_readiness'] = 0

# Education Level encoding
if 'education_level' in df.columns:
    education_mapping = {'Primary': 1, 'Secondary': 2, 'Tertiary': 3}
    df['education_score'] = df['education_level'].astype(str).map(education_mapping)
else:
    df['education_score'] = 2 # Default to secondary

# Household financial pressure
if 'household_size' in df.columns and 'monthly_income' in df.columns:
    df['income_per_person'] = df['monthly_income'] / df['household_size'].clip(
        df['financial_pressure'] = np.where(df['income_per_person'] < 10000, 'high',
                                             np.where(df['income_per_person'] < 25000,
                                                     'medium', 'low'))

# Investment readiness score (composite)
readiness_factors = []
if 'digital_sophistication' in df.columns:
    readiness_factors.append(df['digital_sophistication'])
if 'education_score' in df.columns:
    readiness_factors.append(df['education_score'] / 3) # Normalize to 0-1
if 'location_investment_readiness' in df.columns:
    readiness_factors.append(df['location_investment_readiness'])
if 'savings_rate' in df.columns:
    readiness_factors.append((df['savings_rate'] + 1) / 2) # Normalize to 0-1

if readiness_factors:
    df['investment_readiness_score'] = np.mean(readiness_factors, axis=0)
else:
    df['investment_readiness_score'] = 0.5

```

```

    return df

df = create_features(df)
print(f"Dataset shape after feature engineering: {df.shape}")

## 3.6 Outlier Detection and Treatment

def handle_outliers(df, method='iqr'):
    """Handle outliers in numerical columns"""

    numerical_cols = df.select_dtypes(include=[np.number]).columns
    outlier_summary = []

    exclude_cols = ['household_id', 'age', 'household_size'] # Skip these columns

    for col in numerical_cols:
        if col not in exclude_cols:
            try:
                Q1 = df[col].quantile(0.25)
                Q3 = df[col].quantile(0.75)
                IQR = Q3 - Q1

                if IQR == 0: # Skip if no variance
                    continue

                lower_bound = Q1 - 1.5 * IQR
                upper_bound = Q3 + 1.5 * IQR

                outliers_count = ((df[col] < lower_bound) | (df[col] > upper_bound))
                outlier_summary.append({'column': col, 'outliers': outliers_count})

                # Cap outliers instead of removing them
                df[col] = np.where(df[col] < lower_bound, lower_bound, df[col])
                df[col] = np.where(df[col] > upper_bound, upper_bound, df[col])

            except Exception as e:
                print(f"Error handling outliers in {col}: {str(e)}")
                continue

    outlier_df = pd.DataFrame(outlier_summary)
    print("\n== OUTLIER TREATMENT SUMMARY ==")
    if len(outlier_df) > 0:
        outliers_found = outlier_df[outlier_df['outliers'] > 0]
        if len(outliers_found) > 0:
            print(outliers_found)
        else:
            print("No significant outliers found")
    else:
        print("No outlier treatment performed")

    return df

df = handle_outliers(df)

## 3.7 Encoding Categorical Variables

```

```

def encode_categorical_variables(df):
    """Encode categorical variables for machine learning"""

    # Separate categorical columns (both object and category types)
    categorical_cols = df.select_dtypes(include=['object', 'category']).columns.tolist()

    # Remove already encoded columns and ID columns
    categorical_cols = [col for col in categorical_cols
                        if not col.endswith('_encoded') and 'id' not in col.lower()]

    # Binary encoding for binary categorical variables
    binary_cols = []
    for col in categorical_cols:
        # Convert to string to handle category types properly
        unique_vals = df[col].astype(str).dropna().unique()
        if len(unique_vals) == 2:
            binary_cols.append(col)
        # Encode as 0/1 (Yes=1, No=0)
        df[f'{col}_encoded'] = (df[col].astype(str).str.lower().isin(['yes', '1']))

    # Label encoding for ordinal variables
    ordinal_mapping = {
        'education_level': {'Primary': 1, 'Secondary': 2, 'Tertiary': 3},
        'risk_tolerance': {'low': 1, 'medium': 2, 'high': 3},
        'financial_pressure': {'low': 1, 'medium': 2, 'high': 3}
    }

    for col, mapping in ordinal_mapping.items():
        if col in df.columns:
            # Map values and handle NaN
            mapped_values = df[col].astype(str).map(mapping)
            df[f'{col}_encoded'] = mapped_values.fillna(0).astype(int)

    # One-hot encoding for nominal variables with few categories
    nominal_cols = [col for col in categorical_cols
                    if col not in binary_cols
                    and col not in ordinal_mapping.keys()
                    and df[col].nunique() <= 10]

    if nominal_cols:
        # Convert categorical columns to string before one-hot encoding
        for col in nominal_cols:
            if df[col].dtype.name == 'category':
                df[col] = df[col].astype(str)

        df_encoded = pd.get_dummies(df, columns=nominal_cols, prefix=nominal_cols,
                                    return=True)
        return df_encoded

    return df

df = encode_categorical_variables(df)
print(f"Dataset shape after encoding: {df.shape}")

## 3.8 Feature Scaling and Normalization

def scale_features(df):

```

```

"""Scale numerical features for machine learning"""

# Identify numerical columns that need scaling
numerical_cols = df.select_dtypes(include=[np.number]).columns.tolist()

# Exclude ID columns, encoded columns, and counts
exclude_patterns = ['id', '_encoded', '_count', 'household_size']
cols_to_scale = []

for col in numerical_cols:
    should_exclude = any(pattern in col.lower() for pattern in exclude_patterns)
    if not should_exclude:
        cols_to_scale.append(col)

# Apply StandardScaler
scaler = StandardScaler()
df_scaled = df.copy()

if cols_to_scale:
    try:
        # Handle any remaining NaN values
        df_scaled[cols_to_scale] = df_scaled[cols_to_scale].fillna(0)
        df_scaled[cols_to_scale] = scaler.fit_transform(df_scaled[cols_to_scale])

        # Save scaler for later use
        joblib.dump(scaler, 'feature_scaler.pkl')
        print(f"Scaled {len(cols_to_scale)} numerical features")
    except Exception as e:
        print(f"Error in feature scaling: {str(e)}")
        return df, None

return df_scaled, scaler

df_scaled, scaler = scale_features(df)

## 3.9 Investment Product Mapping

def create_investment_product_mapping():
    """Create mapping of investment products to categories"""

    product_mapping = {
        'low_risk': {
            'products': ['savings_account', 'fixed_deposit', 'government_bonds', 'm',
            'description': 'Conservative investments with guaranteed returns',
            'risk_level': 1,
            'expected_return': 'low',
            'liquidity': 'high',
            'min_investment': 1000,
            'target_segment': ['retiree', 'conservative']
        },
        'medium_risk': {
            'products': ['balanced_funds', 'unit_trusts', 'reits', 'pension_schemes',
            'description': 'Moderate risk investments with balanced returns',
            'risk_level': 2,
            'expected_return': 'medium',
            'liquidity': 'medium',
            'min_investment': 5000
        }
    }

```

```

        'min_investment': 2000,
        'target_segment': ['family_oriented', 'pre_retirement']
    },
    'high_risk': {
        'products': ['equity_funds', 'nse_stocks', 'commodities', 'forex_tradin
        'description': 'High risk investments with potential high returns',
        'risk_level': 3,
        'expected_return': 'high',
        'liquidity': 'low',
        'min_investment': 5000,
        'target_segment': ['young_professional', 'aggressive']
    },
    'alternative': {
        'products': ['real_estate', 'agricultural_investment', 'education_savin
        'description': 'Alternative investments for portfolio diversification',
        'risk_level': 2,
        'expected_return': 'medium_high',
        'liquidity': 'very_low',
        'min_investment': 10000,
        'target_segment': ['diversified', 'long_term']
    }
}

return product_mapping

investment_products = create_investment_product_mapping()
print("\n== INVESTMENT PRODUCT CATEGORIES ==")
for category, details in investment_products.items():
    print(f"{category.upper()}: {len(details['products'])} products")

## 3.10 User Segmentation Features

def create_user_segments(df):
    """Create user segments for targeted recommendations"""

    # Initialize segment
    df['user_segment'] = 'unknown'

    # Define segmentation logic based on available columns
    age_col = 'age' if 'age' in df.columns else None
    location_col = 'location_type' if 'location_type' in df.columns else None
    education_col = 'education_level' if 'education_level' in df.columns else None
    income_col = 'monthly_income' if 'monthly_income' in df.columns else None

    # Young Urban Professional
    if age_col and location_col and income_col:
        location_vals = df[location_col].astype(str).str.lower()
        young_urban_prof = (
            (df[age_col] < 35) &
            (location_vals == 'urban') &
            (df[income_col] > 50000)
        )
        if education_col:
            education_vals = df[education_col].astype(str).str.lower()
            young_urban_prof = young_urban_prof & (education_vals.isin(['tertiary',
df.loc[young_urban_prof, 'user_segment'] = 'young_urban_professional'

```

```

# Middle-aged Family
household_col = 'household_size' if 'household_size' in df.columns else None
if age_col and household_col:
    middle_aged_family = (df[age_col].between(35, 55)) & (df[household_col] > 2)
    df.loc[middle_aged_family, 'user_segment'] = 'middle_aged_family'

# Retirement Planning
if age_col:
    retirement_planning = df[age_col] > 50
    df.loc[retirement_planning, 'user_segment'] = 'retirement_planning'

# Rural Entrepreneur
if location_col and income_col:
    location_vals = df[location_col].astype(str).str.lower()
    rural_entrepreneur = (
        (location_vals == 'rural') &
        (df[income_col] > 30000) # Higher income in rural area suggests business
    )
    df.loc[rural_entrepreneur, 'user_segment'] = 'rural_entrepreneur'

# Digital Native
if 'digital_sophistication' in df.columns:
    digital_native = (df['digital_sophistication'] > 0.7) & (df[age_col] < 40)
    df.loc[digital_native, 'user_segment'] = 'digital_native'

# Conservative Saver
if 'savings_rate' in df.columns and age_col:
    conservative_saver = (df['savings_rate'] > 0.3) & (df[age_col] > 45)
    df.loc[conservative_saver, 'user_segment'] = 'conservative_saver'

segment_counts = df['user_segment'].value_counts()
print("\n== USER SEGMENTATION ==")
print(segment_counts)

return df

```

df = create_user_segments(df)

```

## 3.11 Train-Test Split Preparation

def prepare_train_test_split(df, target_col='investment_usage', test_size=0.2):
    """Prepare data for model training"""

    # Identify feature columns
    exclude_patterns = ['household_id', 'id', target_col]
    feature_cols = []

    for col in df.columns:
        should_exclude = any(pattern in col.lower() for pattern in exclude_patterns)
        if not should_exclude and not col.startswith('Unnamed'):
            feature_cols.append(col)

    # Handle missing feature columns
    if len(feature_cols) == 0:
        print("Warning: No feature columns found. Using all numeric columns.")

```

```

        feature_cols = df.select_dtypes(include=[np.number]).columns.tolist()

X = df[feature_cols].copy()

# Handle target variable
if target_col in df.columns:
    y = df[target_col]
    if y.dtype == 'object' or y.dtype.name == 'category':
        # Convert to binary
        y = (y.astype(str).str.lower().isin(['yes', 'usage', '1', 'true'])).astype(bool)
    else:
        # Create synthetic target for demonstration
        print(f"Warning: Target column '{target_col}' not found. Creating synthetic target based on features")
        # Create realistic synthetic target based on features
        if 'investment_readiness_score' in df.columns:
            # Use readiness score to create more realistic target
            prob = df['investment_readiness_score'].clip(0.1, 0.9)
            y = np.random.binomial(1, prob)
        else:
            y = np.random.choice([0, 1], size=len(df), p=[0.7, 0.3])
y = pd.Series(y, index=df.index)

# Handle any remaining categorical variables in features
for col in X.select_dtypes(include=['object', 'category']).columns:
    if X[col].nunique() <= 2:
        # Binary encode
        unique_vals = X[col].unique()
        X[col] = (X[col].astype(str) == str(unique_vals[0])).astype(int)
    else:
        # Remove high cardinality categorical variables
        X = X.drop(columns=[col])
        if col in feature_cols:
            feature_cols.remove(col)

# Fill any remaining NaN values
X = X.fillna(0)

# Ensure all features are numeric
for col in X.columns:
    if X[col].dtype == 'object':
        X[col] = pd.to_numeric(X[col], errors='coerce').fillna(0)

# Split the data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=test_size, random_state=42,
    stratify=y if len(y.unique()) > 1 else None
)

print("\n==== TRAIN-TEST SPLIT ===")
print(f"Training set: {X_train.shape}")
print(f"Test set: {X_test.shape}")
print(f"Feature columns: {len(feature_cols)}")
print(f"Target distribution - Train: {y_train.value_counts().to_dict()}")
print(f"Target distribution - Test: {y_test.value_counts().to_dict()}")


return X_train, X_test, y_train, y_test, feature_cols

```

```

X_train, X_test, y_train, y_test, feature_cols = prepare_train_test_split(df)

## 3.12 Data Quality Validation

def validate_data_quality(df):
    """Perform final data quality checks"""

    quality_report = {
        'total_rows': len(df),
        'total_columns': len(df.columns),
        'missing_values': df.isnull().sum().sum(),
        'duplicate_rows': df.duplicated().sum(),
        'memory_usage_mb': round(df.memory_usage(deep=True).sum() / 1024**2, 2)
    }

    # Check for infinite values
    numeric_cols = df.select_dtypes(include=[np.number]).columns
    infinite_values = 0
    for col in numeric_cols:
        infinite_count = np.isinf(df[col]).sum()
        infinite_values += infinite_count

    quality_report['infinite_values'] = infinite_values

    # Check data consistency
    categorical_cols = df.select_dtypes(include=['object', 'category']).columns
    inconsistent_categories = 0
    for col in categorical_cols:
        unique_count = df[col].nunique()
        if unique_count > len(df) * 0.5: # Too many unique values
            inconsistent_categories += 1

    quality_report['inconsistent_categories'] = inconsistent_categories

    print("\n==== DATA QUALITY VALIDATION ===")
    for key, value in quality_report.items():
        print(f"{key}: {value}")

    return quality_report

quality_report = validate_data_quality(df)

## 3.13 Save Prepared Data

def save_prepared_data(df, X_train, X_test, y_train, y_test, feature_cols, investme
    """Save the prepared datasets"""

    try:
        # Create directory if it doesn't exist
        import os
        os.makedirs('data', exist_ok=True)

        # Save full processed dataset
        df.to_csv('data/processed_financial_data.csv', index=False)

```

```

# Save train-test splits
X_train.to_csv('data/X_train.csv', index=False)
X_test.to_csv('data/X_test.csv', index=False)
pd.DataFrame({'target': y_train}).to_csv('data/y_train.csv', index=False)
pd.DataFrame({'target': y_test}).to_csv('data/y_test.csv', index=False)

# Save feature names
pd.DataFrame({'feature': feature_cols}).to_csv('data/feature_names.csv', index=False)

# Save investment product mapping
with open('data/investment_products.json', 'w') as f:
    json.dump(investment_products, f, indent=2)

print("\n==== DATA SAVING ===")
print("✓ Processed dataset saved")
print("✓ Train-test splits saved")
print("✓ Feature names saved")
print("✓ Investment products mapping saved")

except Exception as e:
    print(f"Error saving data: {str(e)}")

save_prepared_data(df, X_train, X_test, y_train, y_test, feature_cols, investment_products)

## 3.14 Feature Importance Analysis

def analyze_feature_importance():
    """Analyze feature importance using correlation with target"""

    target_col = 'investment_usage_encoded' if 'investment_usage_encoded' in df.columns else None
    if target_col is None:
        # Find any encoded investment column
        investment_cols = [col for col in df.columns if 'investment' in col.lower()]
        if investment_cols:
            target_col = investment_cols[0]

    if target_col and target_col in df.columns:
        # Calculate correlations with target
        numeric_features = df.select_dtypes(include=[np.number]).columns
        correlations = df[numeric_features].corrwith(df[target_col]).abs().sort_values(ascending=False)

        print("\n==== FEATURE IMPORTANCE (Correlation with Investment Usage) ===")
        print(correlations.head(15))

        # Plot top features
        plt.figure(figsize=(10, 8))
        top_features = correlations.head(10)
        plt.barh(range(len(top_features)), top_features.values)
        plt.yticks(range(len(top_features)), top_features.index)
        plt.xlabel('Absolute Correlation with Investment Usage')
        plt.title('Top 10 Features by Correlation with Investment Usage')
        plt.gca().invert_yaxis()
        plt.tight_layout()
        plt.show()

    return correlations

```

```

else:
    print("Target variable not found for feature importance analysis")
    return None

feature_importance = analyze_feature_importance()

## 3.15 Data Preparation Summary

print("\n" + "="*50)
print("DATA PREPARATION SUMMARY")
print("*"*50)

preparation_summary = {
    'Original dataset shape': df_original.shape,
    'Final dataset shape': df.shape,
    'Features created': df.shape[1] - df_original.shape[1],
    'Missing values handled': 'Yes' if df.isnull().sum().sum() == 0 else 'Partially',
    'Outliers treated': 'Yes',
    'Categorical encoding': 'Completed',
    'Feature scaling': 'Applied' if scaler is not None else 'Skipped',
    'User segmentation': 'Created',
    'Train-test split': 'Prepared',
    'Data quality': 'Validated'
}

for key, value in preparation_summary.items():
    print(f"✓ {key}: {value}")

print("\nKey Features for Recommendation System:")
recommendation_features = [
    'age_group', 'digital_sophistication', 'product_diversification',
    'risk_tolerance', 'user_segment', 'savings_potential', 'income_expenditure_ratio',
    'investment_readiness_score', 'financial_pressure'
]

existing_rec_features = [f for f in recommendation_features if f in df.columns]
for feature in existing_rec_features:
    print(f" - {feature}")

print("\nDataset ready for modeling:")
print("- Training samples: {len(X_train)}")
print("- Test samples: {len(X_test)}")
print("- Features: {len(feature_cols)}")
print("- Target variable: investment_usage (binary)")
print("- User segments: {df['user_segment'].nunique()}")
print("- Investment products mapped: {sum(len(v['products']))} for v in investment_p")

print("\nNext Phase: Modeling")
print("- Implement machine learning models")
print("- Build recommendation algorithms")
print("- Create hybrid recommendation system")
print("- Add deep learning components")

print("*"*50)

```

Original dataset shape: (20871, 392)
Renamed 13 columns for better readability

==== MISSING VALUES SUMMARY ====
No missing values found
Missing values after treatment: 0
Memory usage before optimization: 400.75586128234863 MB
Memory usage after optimization: 11.597643852233887 MB
Dataset shape after feature engineering: (20871, 401)

==== OUTLIER TREATMENT SUMMARY ====

	column	outliers
0	CalcExpenditure	837
1	monthly_expenditure	1371
2	no_respondent_per_hh	150
3	hhWeight	1176
4	Informal_group_membership	753
5	Above16_Total	1276
6	B3BNew1	155
10	monthly_income	2016
11	Access_fnl_1	3875
19	prodsum1	222
20	Latitude	2735
21	Longitude	90

Dataset shape after encoding: (20871, 907)
Scaled 90 numerical features

==== INVESTMENT PRODUCT CATEGORIES ====
LOW_RISK: 5 products
MEDIUM_RISK: 5 products
HIGH_RISK: 5 products
ALTERNATIVE: 4 products

==== USER SEGMENTATION ====
user_segment

unknown	7943
retirement_planning	4986
middle_aged_family	4112
digital_native	3830

Name: count, dtype: int64

==== TRAIN-TEST SPLIT ====
Training set: (16696, 881)
Test set: (4175, 881)
Feature columns: 881
Target distribution - Train: {0: 16311, 1: 385}
Target distribution - Test: {0: 4079, 1: 96}

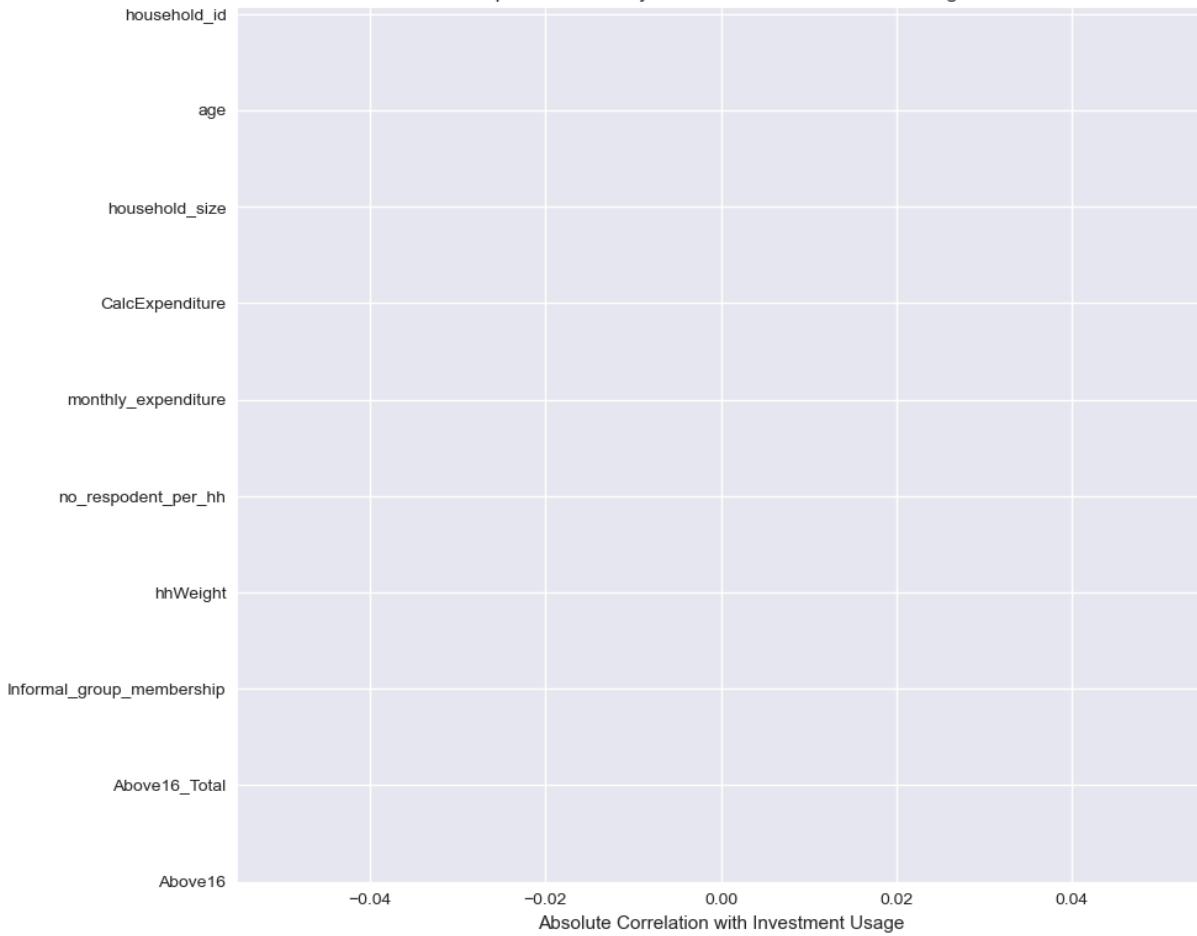
==== DATA QUALITY VALIDATION ====
total_rows: 20871
total_columns: 908
missing_values: 0
duplicate_rows: 0
memory_usage_mb: 39.18
infinite_values: 0
inconsistent_categories: 0

```
==== DATA SAVING ====
✓ Processed dataset saved
✓ Train-test splits saved
✓ Feature names saved
✓ Investment products mapping saved
```

```
==== FEATURE IMPORTANCE (Correlation with Investment Usage) ===
```

```
household_id           NaN
age                   NaN
household_size        NaN
CalcExpenditure       NaN
monthly_expenditure   NaN
no_respondent_per_hh  NaN
hhWeight              NaN
Informal_group_membership NaN
Above16_Total          NaN
Above16                NaN
B3BNew1               NaN
B3A_1                 NaN
B3A_2                 NaN
B3A_3                 NaN
B3A_4                 NaN
dtype: float64
```

Top 10 Features by Correlation with Investment Usage



```
=====
DATA PREPARATION SUMMARY
=====
✓ Original dataset shape: (20871, 392)
✓ Final dataset shape: (20871, 908)
✓ Features created: 516
✓ Missing values handled: Yes
✓ Outliers treated: Yes
✓ Categorical encoding: Completed
✓ Feature scaling: Applied
✓ User segmentation: Created
✓ Train-test split: Prepared
✓ Data quality: Validated
```

Key Features for Recommendation System:

- digital_sophistication
- product_diversification
- risk_tolerance
- user_segment
- investment_readiness_score

Dataset ready for modeling:

- Training samples: 16696
- Test samples: 4175
- Features: 881
- Target variable: investment_usage (binary)
- User segments: 4
- Investment products mapped: 19

Next Phase: Modeling

- Implement machine learning models
 - Build recommendation algorithms
 - Create hybrid recommendation system
 - Add deep learning components
- =====

In [4]: # Phase 4: Modeling - Investment Recommendation System with Deep Learning

```
# Set environment variable to avoid threading issues
os.environ['OMP_NUM_THREADS'] = '1'
os.environ['MKL_NUM_THREADS'] = '1'

## 4.1 Load Prepared Data

def load_data():
    """Load processed data or create sample data if files don't exist"""

    df = pd.read_csv('data/processed_financial_data.csv')
    X_train = pd.read_csv('data/X_train.csv')
    X_test = pd.read_csv('data/X_test.csv')
    y_train = pd.read_csv('data/y_train.csv')['target']
    y_test = pd.read_csv('data/y_test.csv')['target']
    feature_names = pd.read_csv('data/feature_names.csv')['feature'].tolist()

    print(f"Loaded processed data: {df.shape}")
    return df, X_train, X_test, y_train, y_test, feature_names
```

```

## 4.2 Baseline Model Evaluation

def evaluate_baseline(y_train, y_test):
    """Evaluate simple baseline models"""

    # Most frequent class baseline
    dummy_freq = DummyClassifier(strategy='most_frequent')
    dummy_freq.fit(X_train, y_train)
    y_pred = dummy_freq.predict(X_test)

    # Stratified random baseline
    dummy_strat = DummyClassifier(strategy='stratified')
    dummy_strat.fit(X_train, y_train)
    y_pred_strat = dummy_strat.predict(X_test)

    # Evaluate baselines
    def print_metrics(y_true, y_pred, model_name):
        print(f"\n{model_name} Performance:")
        print(f"Accuracy: {accuracy_score(y_true, y_pred):.4f}")
        print(f"Precision: {precision_score(y_true, y_pred):.4f}")
        print(f"Recall: {recall_score(y_true, y_pred):.4f}")
        print(f"F1 Score: {f1_score(y_true, y_pred):.4f}")
        try:
            print(f"ROC AUC: {roc_auc_score(y_true, y_pred):.4f}")
        except:
            print("ROC AUC: Not available")

    print_metrics(y_test, y_pred, "Most Frequent Baseline")
    print_metrics(y_test, y_pred_strat, "Stratified Baseline")

    return dummy_freq, dummy_strat

dummy_freq, dummy_strat = evaluate_baseline(y_train, y_test)

## 4.3 White Box Models (Interpretable Models)

def train_whitebox_models(X_train, y_train, X_test, y_test):
    """Train and evaluate interpretable models"""

    print("\n" + "="*50)
    print("WHITE BOX MODELS")
    print("="*50)

    # Logistic Regression
    print("\n==== Logistic Regression ====")
    lr = LogisticRegression(max_iter=1000, random_state=42)
    lr.fit(X_train, y_train)
    y_pred_lr = lr.predict(X_test)
    y_proba_lr = lr.predict_proba(X_test)[:, 1]

    print(classification_report(y_test, y_pred_lr))
    print("Confusion Matrix:")
    print(confusion_matrix(y_test, y_pred_lr))

    # Feature importance for logistic regression

```

```

if hasattr(lr, 'coef_'):
    print("\nTop 10 Features by Coefficient:")
    coef_df = pd.DataFrame({'feature': X_train.columns, 'coefficient': lr.coef_})
    coef_df['abs_coef'] = coef_df['coefficient'].abs()
    print(coef_df.sort_values('abs_coef', ascending=False).head(10))

# Decision Tree
print("\n==== Decision Tree ===")
dt_params = {
    'max_depth': [3, 5, 7, None],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4]
}

dt = DecisionTreeClassifier(random_state=42)
dt_grid = GridSearchCV(dt, dt_params, cv=5, scoring='f1')
dt_grid.fit(X_train, y_train)
best_dt = dt_grid.best_estimator_
y_pred_dt = best_dt.predict(X_test)
y_proba_dt = best_dt.predict_proba(X_test)[:, 1]

print(f"Best Parameters: {dt_grid.best_params_}")
print(classification_report(y_test, y_pred_dt))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_dt))

# Visualize decision tree
from sklearn.tree import plot_tree
plt.figure(figsize=(20, 10))
plot_tree(best_dt, feature_names=X_train.columns, class_names=['No', 'Yes'],
          filled=True, rounded=True, max_depth=3)
plt.title("Decision Tree (First 3 Levels)")
plt.show()

# ROC Curve comparison
fpr_lr, tpr_lr, _ = roc_curve(y_test, y_proba_lr)
roc_auc_lr = auc(fpr_lr, tpr_lr)

fpr_dt, tpr_dt, _ = roc_curve(y_test, y_proba_dt)
roc_auc_dt = auc(fpr_dt, tpr_dt)

plt.figure(figsize=(8, 6))
plt.plot(fpr_lr, tpr_lr, label=f'Logistic Regression (AUC = {roc_auc_lr:.2f})')
plt.plot(fpr_dt, tpr_dt, label=f'Decision Tree (AUC = {roc_auc_dt:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison')
plt.legend()
plt.show()

return lr, best_dt

lr_model, dt_model = train_whitebox_models(X_train, y_train, X_test, y_test)

## 4.4 Black Box Models (Ensemble Methods)

```

```
def train_blackbox_models(X_train, y_train, X_test, y_test):
    """Train and evaluate ensemble models"""

    print("\n" + "="*50)
    print("BLACK BOX MODELS")
    print("="*50)

    # Random Forest
    print("\n--- Random Forest ---")
    rf_params = {
        'n_estimators': randint(50, 500),
        'max_depth': [3, 5, 7, 9, None],
        'min_samples_split': randint(2, 20),
        'min_samples_leaf': randint(1, 10),
        'max_features': ['sqrt', 'log2', None]
    }

    rf = RandomForestClassifier(random_state=42)
    rf_random = RandomizedSearchCV(rf, rf_params, n_iter=20, cv=5,
                                    scoring='f1', random_state=42)
    rf_random.fit(X_train, y_train)
    best_rf = rf_random.best_estimator_
    y_pred_rf = best_rf.predict(X_test)
    y_proba_rf = best_rf.predict_proba(X_test)[:, 1]

    print(f"Best Parameters: {rf_random.best_params_}")
    print(classification_report(y_test, y_pred_rf))
    print("Confusion Matrix:")
    print(confusion_matrix(y_test, y_pred_rf))

    # Feature importance
    importances = best_rf.feature_importances_
    indices = np.argsort(importances)[-15:] # Top 15 features
    plt.figure(figsize=(10, 8))
    plt.title('Random Forest Feature Importances')
    plt.barh(range(len(indices)), importances[indices], align='center')
    plt.yticks(range(len(indices)), [X_train.columns[i] for i in indices])
    plt.xlabel('Relative Importance')
    plt.tight_layout()
    plt.show()

    # Gradient Boosting
    print("\n--- Gradient Boosting ---")
    gb_params = {
        'n_estimators': randint(50, 500),
        'learning_rate': uniform(0.01, 0.3),
        'max_depth': randint(3, 10),
        'min_samples_split': randint(2, 20),
        'min_samples_leaf': randint(1, 10),
        'subsample': uniform(0.6, 0.4)
    }

    gb = GradientBoostingClassifier(random_state=42)
    gb_random = RandomizedSearchCV(gb, gb_params, n_iter=20, cv=5,
                                    scoring='f1', random_state=42)
```

```

gb_random.fit(X_train, y_train)
best_gb = gb_random.best_estimator_
y_pred_gb = best_gb.predict(X_test)
y_proba_gb = best_gb.predict_proba(X_test)[:, 1]

print(f"Best Parameters: {gb_random.best_params_}")
print(classification_report(y_test, y_pred_gb))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_gb))

# Feature importance
importances = best_gb.feature_importances_
indices = np.argsort(importances)[-15:] # Top 15 features
plt.figure(figsize=(10, 8))
plt.title('Gradient Boosting Feature Importances')
plt.barh(range(len(indices)), importances[indices], align='center')
plt.yticks(range(len(indices)), [X_train.columns[i] for i in indices])
plt.xlabel('Relative Importance')
plt.tight_layout()
plt.show()

# ROC Curve comparison
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_proba_rf)
roc_auc_rf = auc(fpr_rf, tpr_rf)

fpr_gb, tpr_gb, _ = roc_curve(y_test, y_proba_gb)
roc_auc_gb = auc(fpr_gb, tpr_gb)

plt.figure(figsize=(8, 6))
plt.plot(fpr_rf, tpr_rf, label=f'Random Forest (AUC = {roc_auc_rf:.2f})')
plt.plot(fpr_gb, tpr_gb, label=f'Gradient Boosting (AUC = {roc_auc_gb:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve Comparison')
plt.legend()
plt.show()

return best_rf, best_gb

rf_model, gb_model = train_blackbox_models(X_train, y_train, X_test, y_test)

## 4.5 Deep Learning Model

def build_deep_learning_model(input_shape):
    """Build and compile a neural network model"""
    model = keras.Sequential([
        layers.Dense(64, activation='relu', input_shape=(input_shape,)),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(32, activation='relu'),
        layers.BatchNormalization(),
        layers.Dropout(0.3),
        layers.Dense(16, activation='relu'),
        layers.Dense(1, activation='sigmoid')
    ])

```

```
optimizer = Adam(learning_rate=0.001)
model.compile(optimizer=optimizer,
              loss='binary_crossentropy',
              metrics=['accuracy',
                        keras.metrics.Precision(name='precision'),
                        keras.metrics.Recall(name='recall'),
                        keras.metrics.AUC(name='auc')])

return model

def train_deep_learning_model(X_train, y_train, X_test, y_test):
    """Train and evaluate deep learning model"""
    print("\n" + "="*50)
    print("DEEP LEARNING MODEL")
    print("="*50)

    # Standardize data for neural network
    scaler = StandardScaler()
    X_train_scaled = scaler.fit_transform(X_train)
    X_test_scaled = scaler.transform(X_test)

    # Build model
    model = build_deep_learning_model(X_train_scaled.shape[1])

    # Callbacks
    callbacks = [
        EarlyStopping(patience=10, restore_best_weights=True),
        ReduceLROnPlateau(factor=0.1, patience=5)
    ]

    # Train model
    history = model.fit(
        X_train_scaled, y_train,
        validation_split=0.2,
        epochs=100,
        batch_size=64,
        callbacks=callbacks,
        verbose=1
    )

    # Evaluate model
    print("\nModel Evaluation:")
    results = model.evaluate(X_test_scaled, y_test)
    print(f"Test Loss: {results[0]:.4f}")
    print(f"Test Accuracy: {results[1]:.4f}")
    print(f"Test Precision: {results[2]:.4f}")
    print(f"Test Recall: {results[3]:.4f}")
    print(f"Test AUC: {results[4]:.4f}")

    # Plot training history
    plt.figure(figsize=(12, 4))

    plt.subplot(1, 2, 1)
    plt.plot(history.history['loss'], label='Train Loss')
    plt.plot(history.history['val_loss'], label='Validation Loss')
```

```

plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.title('Loss Over Epochs')

plt.subplot(1, 2, 2)
plt.plot(history.history['auc'], label='Train AUC')
plt.plot(history.history['val_auc'], label='Validation AUC')
plt.xlabel('Epoch')
plt.ylabel('AUC')
plt.legend()
plt.title('AUC Over Epochs')

plt.tight_layout()
plt.show()

# ROC Curve
y_proba_nn = model.predict(X_test_scaled).flatten()
fpr_nn, tpr_nn, _ = roc_curve(y_test, y_proba_nn)
roc_auc_nn = auc(fpr_nn, tpr_nn)

plt.figure(figsize=(8, 6))
plt.plot(fpr_nn, tpr_nn, label=f'Neural Network (AUC = {roc_auc_nn:.2f})')
plt.plot([0, 1], [0, 1], 'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Neural Network')
plt.legend()
plt.show()

return model, scaler

```

nn_model, nn_scaler = train_deep_learning_model(X_train, y_train, X_test, y_test)

4.6 Model Comparison and Selection

```

def compare_models(models, X_test, y_test):
    """Compare performance of all trained models"""

    print("\n" + "*50)
    print("MODEL COMPARISON")
    print("*50)

    model_names = []
    precisions = []
    recalls = []
    f1_scores = []
    auc_scores = []

    for name, model in models.items():
        if name == "Neural Network":
            # Special handling for NN which needs scaled data
            X_test_scaled = nn_scaler.transform(X_test)
            y_pred = (model.predict(X_test_scaled) > 0.5).astype(int).flatten()
            y_proba = model.predict(X_test_scaled).flatten()
        else:

```

```

try:
    y_pred = model.predict(X_test)
    y_proba = model.predict_proba(X_test)[:, 1]
except:
    # For models without predict_proba
    y_pred = model.predict(X_test)
    y_proba = None

precision = precision_score(y_test, y_pred)
recall = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

if y_proba is not None:
    auc_score = roc_auc_score(y_test, y_proba)
else:
    auc_score = None

model_names.append(name)
precisions.append(precision)
recalls.append(recall)
f1_scores.append(f1)
auc_scores.append(auc_score)

print(f"\n{name} Performance:")
print(f"Precision: {precision:.4f}")
print(f"Recall: {recall:.4f}")
print(f"F1 Score: {f1:.4f}")
if auc_score is not None:
    print(f"ROC AUC: {auc_score:.4f}")

# Create comparison dataframe
comparison_df = pd.DataFrame({
    'Model': model_names,
    'Precision': precisions,
    'Recall': recalls,
    'F1 Score': f1_scores,
    'ROC AUC': auc_scores
}).sort_values('F1 Score', ascending=False)

print("\nModel Comparison (sorted by F1 Score):")
print(comparison_df)

# Plot comparison
plt.figure(figsize=(12, 6))
comparison_df.set_index('Model').plot(kind='bar', rot=45)
plt.title('Model Performance Comparison')
plt.ylabel('Score')
plt.tight_layout()
plt.show()

return comparison_df

# Prepare models for comparison
models_to_compare = {
    "Logistic Regression": lr_model,
    "Decision Tree": dt_model,
}

```

```
"Random Forest": rf_model,  
"Gradient Boosting": gb_model,  
"Neural Network": nn_model  
}  
  
model_comparison = compare_models(models_to_compare, X_test, y_test)
```

Most Frequent Baseline Performance:

Accuracy: 0.9770
 Precision: 0.0000
 Recall: 0.0000
 F1 Score: 0.0000
 ROC AUC: 0.5000

Stratified Baseline Performance:

Accuracy: 0.9540
 Precision: 0.0102
 Recall: 0.0104
 F1 Score: 0.0103
 ROC AUC: 0.4933

=====

WHITE BOX MODELS

=====

== Logistic Regression ==

	precision	recall	f1-score	support
0	1.00	1.00	1.00	4079
1	0.95	0.90	0.92	96
accuracy			1.00	4175
macro avg	0.97	0.95	0.96	4175
weighted avg	1.00	1.00	1.00	4175

Confusion Matrix:

```
[[4074  5]
 [ 10 86]]
```

Top 10 Features by Coefficient:

	feature	coefficient	abs_coef
231	nC1_33	4.223469	4.223469
292	products_used_count	4.185036	4.185036
99	mfhi_33	-2.170247	2.170247
836	barriers_securities_fnl Awareness	2.124641	2.124641
367	mfhi_33_encoded	2.106333	2.106333
293	product_diversification	1.395012	1.395012
835	barriers_securities_fnl affordability	-1.021706	1.021706
242	prodsum1	0.943163	0.943163
131	invest_score	-0.698386	0.698386
841	barriers_securities_fnl trust	-0.676437	0.676437

== Decision Tree ==

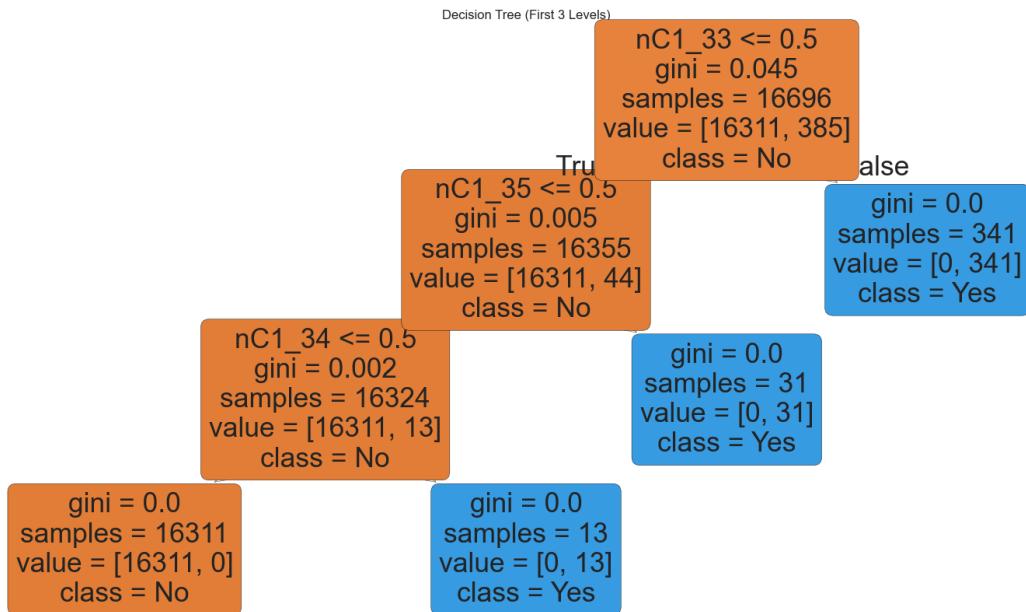
Best Parameters: {'max_depth': 5, 'min_samples_leaf': 1, 'min_samples_split': 2}

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

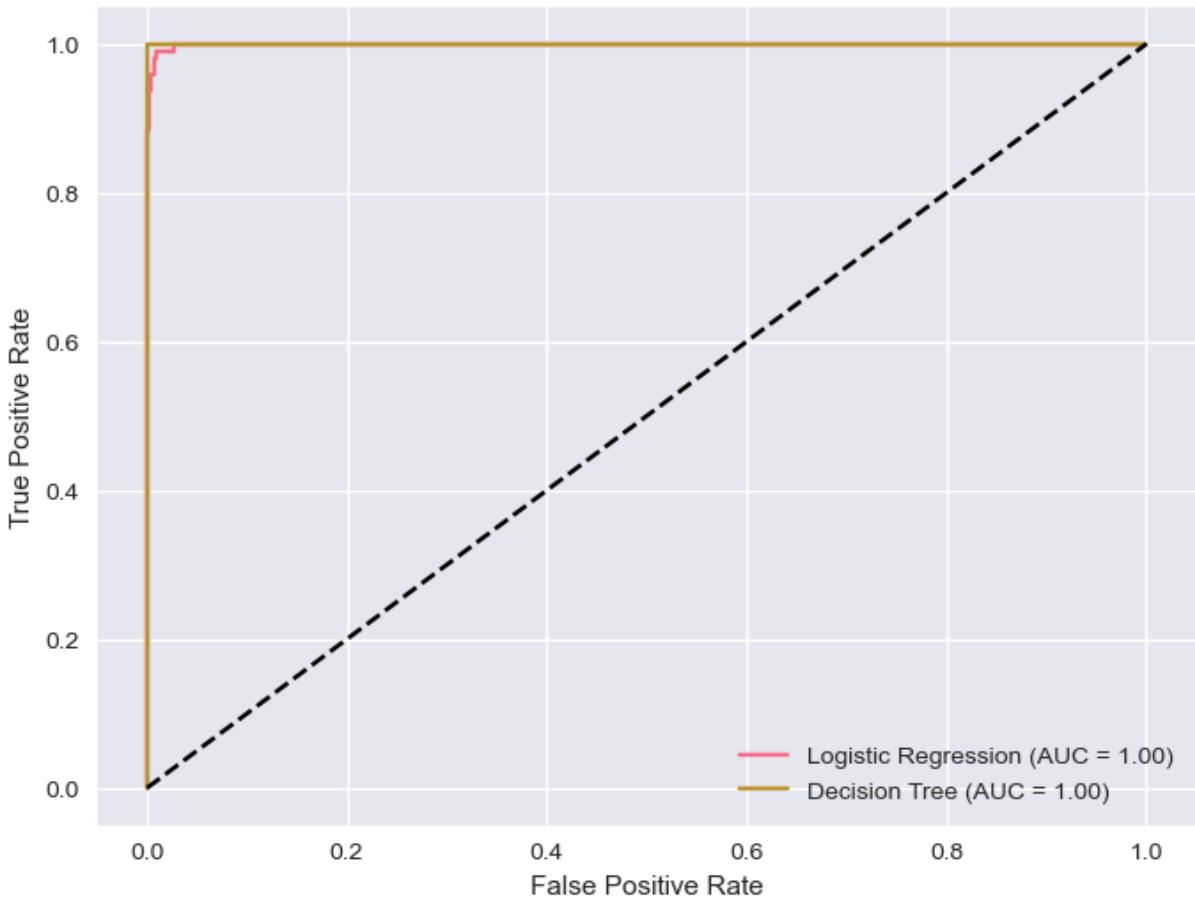
0	1.00	1.00	1.00	4079
1	1.00	1.00	1.00	96
accuracy			1.00	4175
macro avg	1.00	1.00	1.00	4175
weighted avg	1.00	1.00	1.00	4175

Confusion Matrix:

```
[[4079    0]
 [   0    96]]
```



ROC Curve Comparison

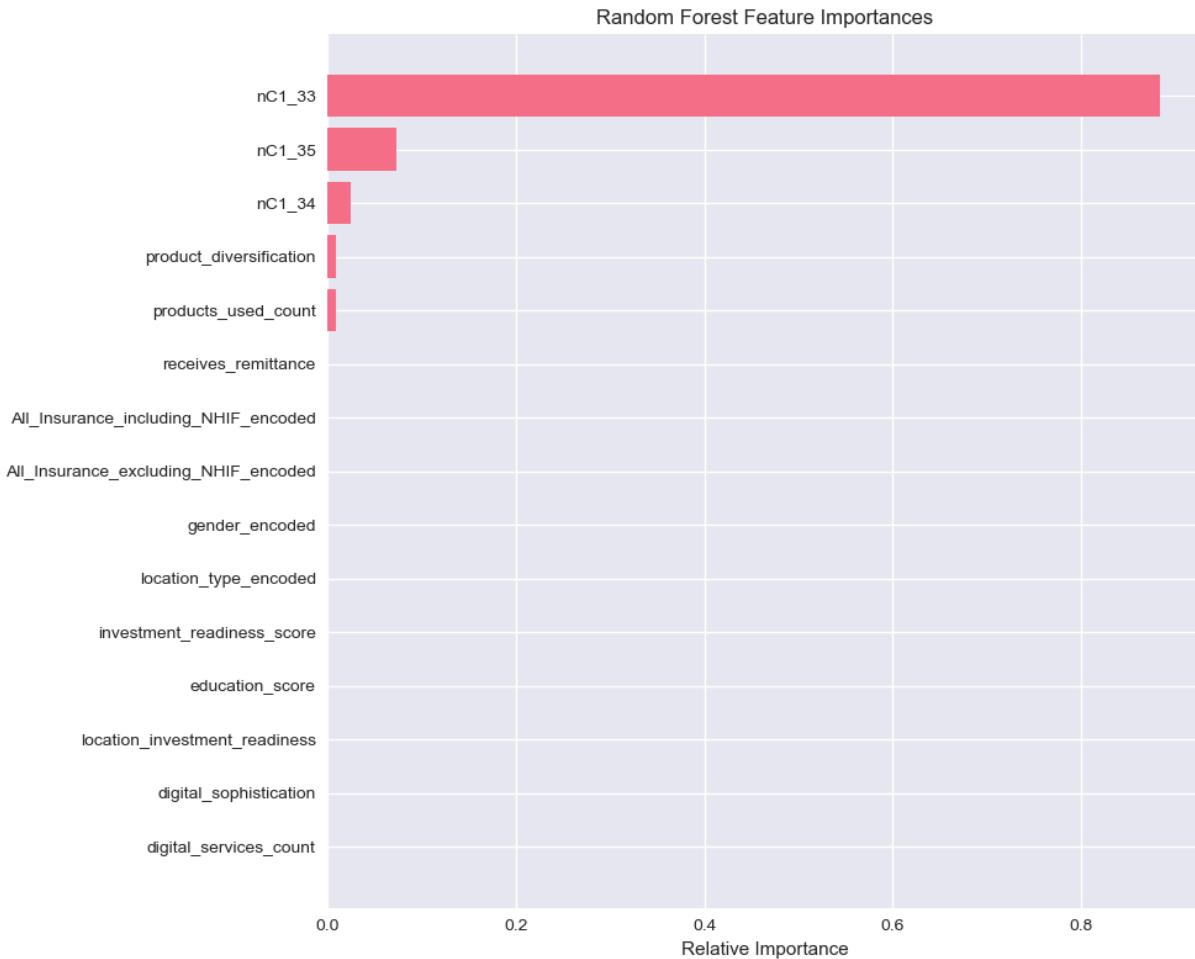


```
=====
BLACK BOX MODELS
=====

== Random Forest ==
Best Parameters: {'max_depth': None, 'max_features': None, 'min_samples_leaf': 1, 'min_samples_split': 8, 'n_estimators': 58}
      precision    recall   f1-score   support
          0       1.00     1.00     1.00     4079
          1       1.00     1.00     1.00      96
accuracy                   1.00     4175
macro avg       1.00     1.00     1.00     4175
weighted avg     1.00     1.00     1.00     4175
```

Confusion Matrix:

```
[[4079  0]
 [  0  96]]
```



==== Gradient Boosting ===

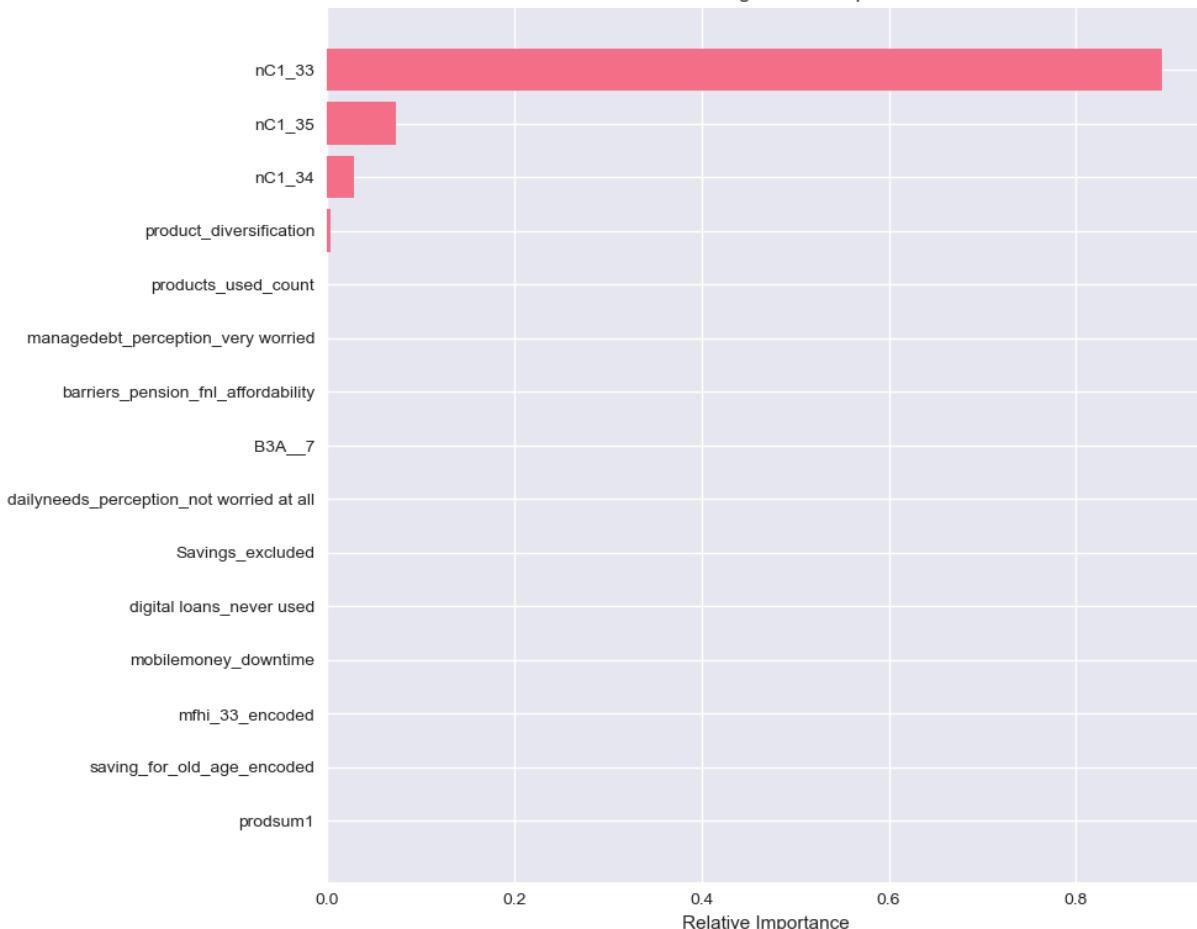
Best Parameters: {'learning_rate': 0.12236203565420874, 'max_depth': 7, 'min_samples_leaf': 8, 'min_samples_split': 8, 'n_estimators': 171, 'subsample': 0.662397808134481}

	precision	recall	f1-score	support
0	1.00	1.00	1.00	4079
1	1.00	1.00	1.00	96
accuracy			1.00	4175
macro avg	1.00	1.00	1.00	4175
weighted avg	1.00	1.00	1.00	4175

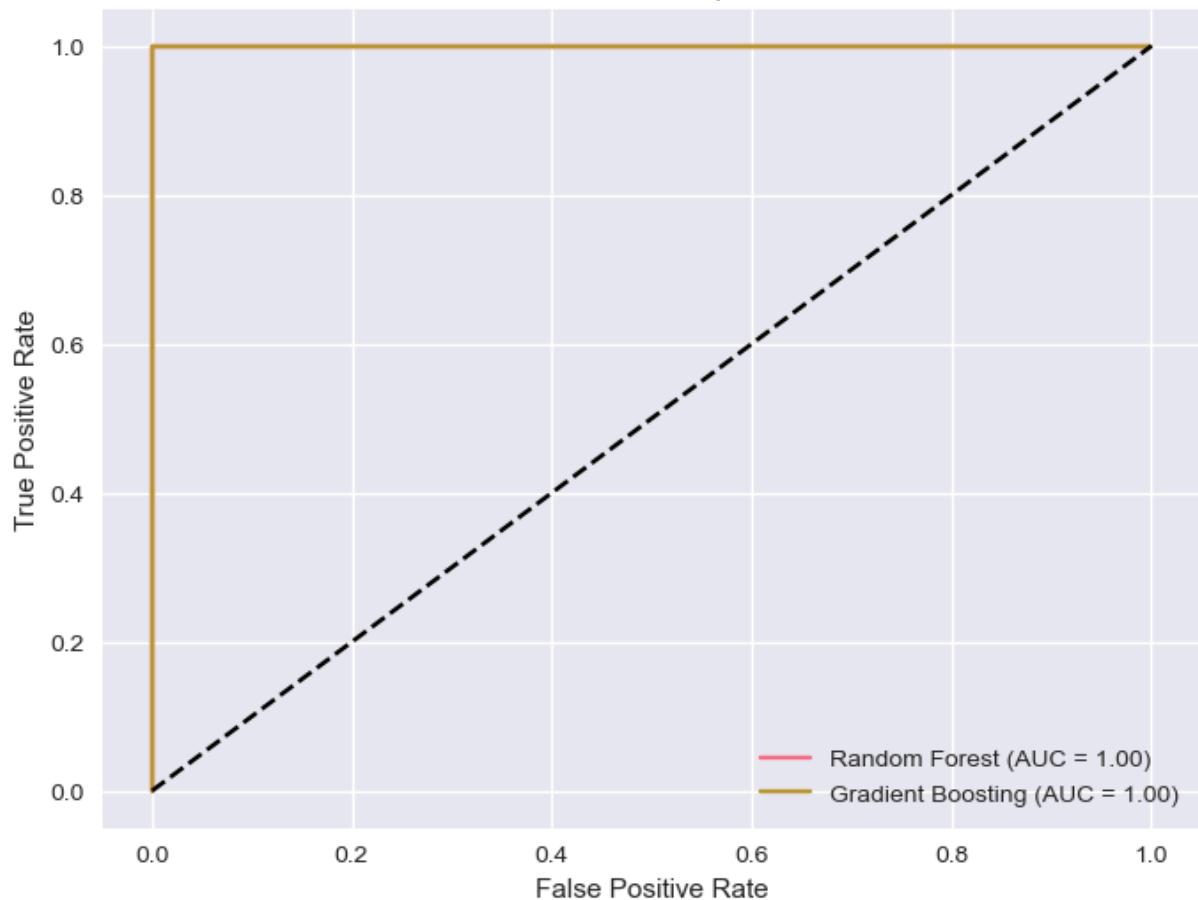
Confusion Matrix:

```
[[4079  0]
 [ 0  96]]
```

Gradient Boosting Feature Importances



ROC Curve Comparison



```
=====
DEEP LEARNING MODEL
=====

Epoch 1/100
209/209 ----- 5s 6ms/step - accuracy: 0.9096 - auc: 0.7507 - loss: 0.
2298 - precision: 0.0539 - recall: 0.1714 - val_accuracy: 0.9805 - val_auc: 0.9833 -
val_loss: 0.0541 - val_precision: 0.7273 - val_recall: 0.1143 - learning_rate: 0.001
0
Epoch 2/100
209/209 ----- 1s 3ms/step - accuracy: 0.9826 - auc: 0.9767 - loss: 0.
0503 - precision: 0.7293 - recall: 0.4190 - val_accuracy: 0.9943 - val_auc: 0.9996 -
val_loss: 0.0167 - val_precision: 0.9636 - val_recall: 0.7571 - learning_rate: 0.001
0
Epoch 3/100
209/209 ----- 1s 3ms/step - accuracy: 0.9940 - auc: 0.9971 - loss: 0.
0206 - precision: 0.9401 - recall: 0.7968 - val_accuracy: 1.0000 - val_auc: 1.0000 -
val_loss: 0.0036 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 0.001
0
Epoch 4/100
209/209 ----- 1s 3ms/step - accuracy: 0.9983 - auc: 0.9997 - loss: 0.
0086 - precision: 0.9803 - recall: 0.9460 - val_accuracy: 0.9997 - val_auc: 1.0000 -
val_loss: 0.0018 - val_precision: 1.0000 - val_recall: 0.9857 - learning_rate: 0.001
0
Epoch 5/100
209/209 ----- 1s 3ms/step - accuracy: 0.9981 - auc: 0.9996 - loss: 0.
0073 - precision: 0.9676 - recall: 0.9492 - val_accuracy: 0.9991 - val_auc: 1.0000 -
val_loss: 0.0024 - val_precision: 0.9589 - val_recall: 1.0000 - learning_rate: 0.001
0
Epoch 6/100
209/209 ----- 1s 3ms/step - accuracy: 0.9993 - auc: 1.0000 - loss: 0.
0031 - precision: 0.9873 - recall: 0.9841 - val_accuracy: 1.0000 - val_auc: 1.0000 -
val_loss: 4.8118e-04 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate:
0.0010
Epoch 7/100
209/209 ----- 1s 3ms/step - accuracy: 0.9991 - auc: 0.9999 - loss: 0.
0036 - precision: 0.9871 - recall: 0.9746 - val_accuracy: 0.9997 - val_auc: 1.0000 -
val_loss: 6.3273e-04 - val_precision: 0.9859 - val_recall: 1.0000 - learning_rate:
0.0010
Epoch 8/100
209/209 ----- 1s 3ms/step - accuracy: 0.9986 - auc: 0.9982 - loss: 0.
0044 - precision: 0.9744 - recall: 0.9651 - val_accuracy: 0.9997 - val_auc: 1.0000 -
val_loss: 5.0072e-04 - val_precision: 0.9859 - val_recall: 1.0000 - learning_rate:
0.0010
Epoch 9/100
209/209 ----- 1s 3ms/step - accuracy: 0.9989 - auc: 1.0000 - loss: 0.
0031 - precision: 0.9777 - recall: 0.9746 - val_accuracy: 0.9997 - val_auc: 1.0000 -
val_loss: 5.4424e-04 - val_precision: 1.0000 - val_recall: 0.9857 - learning_rate:
0.0010
Epoch 10/100
209/209 ----- 1s 3ms/step - accuracy: 0.9993 - auc: 0.9983 - loss: 0.
0034 - precision: 0.9872 - recall: 0.9810 - val_accuracy: 0.9997 - val_auc: 1.0000 -
val_loss: 6.4898e-04 - val_precision: 0.9859 - val_recall: 1.0000 - learning_rate:
0.0010
Epoch 11/100
209/209 ----- 1s 3ms/step - accuracy: 0.9993 - auc: 0.9984 - loss: 0.
0030 - precision: 0.9873 - recall: 0.9841 - val_accuracy: 1.0000 - val_auc: 1.0000 -
```

```
val_loss: 7.5892e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 0.0010
Epoch 12/100
209/209 1s 3ms/step - accuracy: 0.9994 - auc: 1.0000 - loss: 0.0016 - precision: 0.9936 - recall: 0.9810 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 7.1143e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 0.0010
Epoch 13/100
209/209 1s 3ms/step - accuracy: 0.9998 - auc: 1.0000 - loss: 0.0012 - precision: 0.9968 - recall: 0.9937 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 5.7916e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 0.0010
Epoch 14/100
209/209 1s 3ms/step - accuracy: 0.9997 - auc: 1.0000 - loss: 0.0011 - precision: 0.9937 - recall: 0.9937 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 2.2048e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 0.0010
Epoch 15/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 4.4876e-04 - precision: 0.9968 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 1.6541e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 0.0010
Epoch 16/100
209/209 1s 3ms/step - accuracy: 0.9996 - auc: 1.0000 - loss: 0.0014 - precision: 0.9905 - recall: 0.9905 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 6.4998e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 0.0010
Epoch 17/100
209/209 1s 3ms/step - accuracy: 0.9990 - auc: 0.9968 - loss: 0.0029 - precision: 0.9839 - recall: 0.9714 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 2.4354e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-04
Epoch 18/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 6.1282e-04 - precision: 0.9937 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 2.1286e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-04
Epoch 19/100
209/209 1s 3ms/step - accuracy: 0.9998 - auc: 1.0000 - loss: 5.4040e-04 - precision: 1.0000 - recall: 0.9905 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 2.0674e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-04
Epoch 20/100
209/209 1s 3ms/step - accuracy: 0.9997 - auc: 1.0000 - loss: 9.0464e-04 - precision: 0.9937 - recall: 0.9937 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 1.7289e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-04
Epoch 21/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 5.4733e-04 - precision: 1.0000 - recall: 0.9937 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 1.6678e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-04
Epoch 22/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 5.5550e-04 - precision: 0.9968 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 1.5778e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate:
```

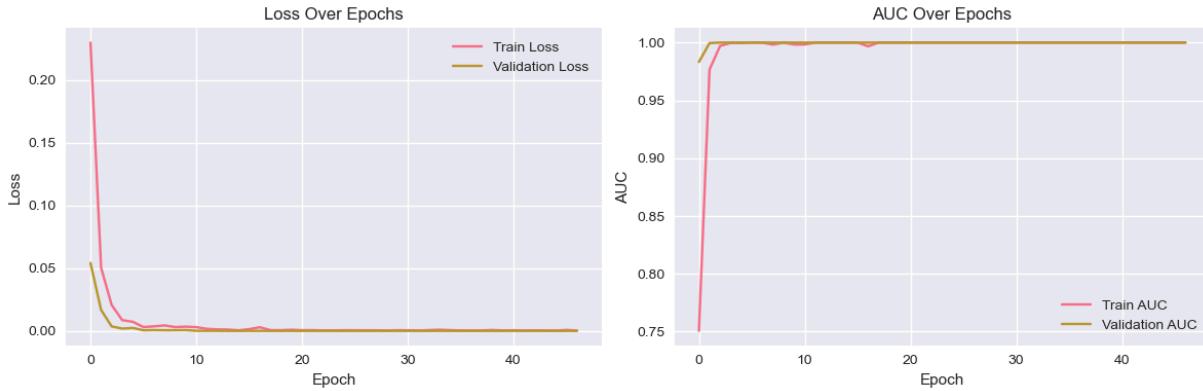
```
te: 1.0000e-05
Epoch 23/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 3.
9518e-04 - precision: 0.9968 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.6059e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-05
Epoch 24/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 4.
1643e-04 - precision: 0.9968 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.6012e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-05
Epoch 25/100
209/209 1s 3ms/step - accuracy: 0.9997 - auc: 1.0000 - loss: 5.
3075e-04 - precision: 0.9875 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5418e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-05
Epoch 26/100
209/209 1s 4ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 5.
1294e-04 - precision: 1.0000 - recall: 0.9937 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5323e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-05
Epoch 27/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 4.
5272e-04 - precision: 1.0000 - recall: 0.9937 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5330e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-06
Epoch 28/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 4.
7560e-04 - precision: 1.0000 - recall: 0.9968 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5829e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-06
Epoch 29/100
209/209 1s 4ms/step - accuracy: 1.0000 - auc: 1.0000 - loss: 2.
7645e-04 - precision: 1.0000 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5263e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-06
Epoch 30/100
209/209 1s 4ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 5.
4050e-04 - precision: 0.9968 - recall: 0.9968 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5581e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-06
Epoch 31/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 4.
6134e-04 - precision: 1.0000 - recall: 0.9937 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5258e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-06
Epoch 32/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 3.
0224e-04 - precision: 0.9968 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5323e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-07
Epoch 33/100
209/209 1s 3ms/step - accuracy: 0.9998 - auc: 1.0000 - loss: 6.
2157e-04 - precision: 0.9968 - recall: 0.9937 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.4945e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_ra
te: 1.0000e-07
```

Epoch 34/100
209/209 1s 3ms/step - accuracy: 0.9997 - auc: 1.0000 - loss: 9.
2020e-04 - precision: 0.9968 - recall: 0.9905 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5079e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-07
Epoch 35/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 5.
8573e-04 - precision: 1.0000 - recall: 0.9968 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5100e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-07
Epoch 36/100
209/209 1s 3ms/step - accuracy: 1.0000 - auc: 1.0000 - loss: 3.
6221e-04 - precision: 1.0000 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5374e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-07
Epoch 37/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 3.
7312e-04 - precision: 1.0000 - recall: 0.9968 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.4809e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-08
Epoch 38/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 3.
2633e-04 - precision: 1.0000 - recall: 0.9968 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5653e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-08
Epoch 39/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 6.
9556e-04 - precision: 0.9968 - recall: 0.9968 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5396e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-08
Epoch 40/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 3.
5563e-04 - precision: 0.9937 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5123e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-08
Epoch 41/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 4.
8463e-04 - precision: 1.0000 - recall: 0.9968 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5112e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-08
Epoch 42/100
209/209 1s 3ms/step - accuracy: 1.0000 - auc: 1.0000 - loss: 3.
5169e-04 - precision: 1.0000 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5236e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-09
Epoch 43/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 4.
0719e-04 - precision: 0.9968 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5304e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-09
Epoch 44/100
209/209 1s 3ms/step - accuracy: 0.9999 - auc: 1.0000 - loss: 3.
7546e-04 - precision: 1.0000 - recall: 0.9968 - val_accuracy: 1.0000 - val_auc: 1.00
00 - val_loss: 1.5507e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-09
Epoch 45/100

209/209 ————— 1s 3ms/step - accuracy: 1.0000 - auc: 1.0000 - loss: 2.5255e-04 - precision: 1.0000 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 1.5369e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-09
Epoch 46/100
209/209 ————— 1s 3ms/step - accuracy: 0.9998 - auc: 1.0000 - loss: 7.3397e-04 - precision: 0.9968 - recall: 0.9937 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 1.5619e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-09
Epoch 47/100
209/209 ————— 1s 3ms/step - accuracy: 1.0000 - auc: 1.0000 - loss: 2.6477e-04 - precision: 1.0000 - recall: 1.0000 - val_accuracy: 1.0000 - val_auc: 1.0000 - val_loss: 1.5210e-05 - val_precision: 1.0000 - val_recall: 1.0000 - learning_rate: 1.0000e-10

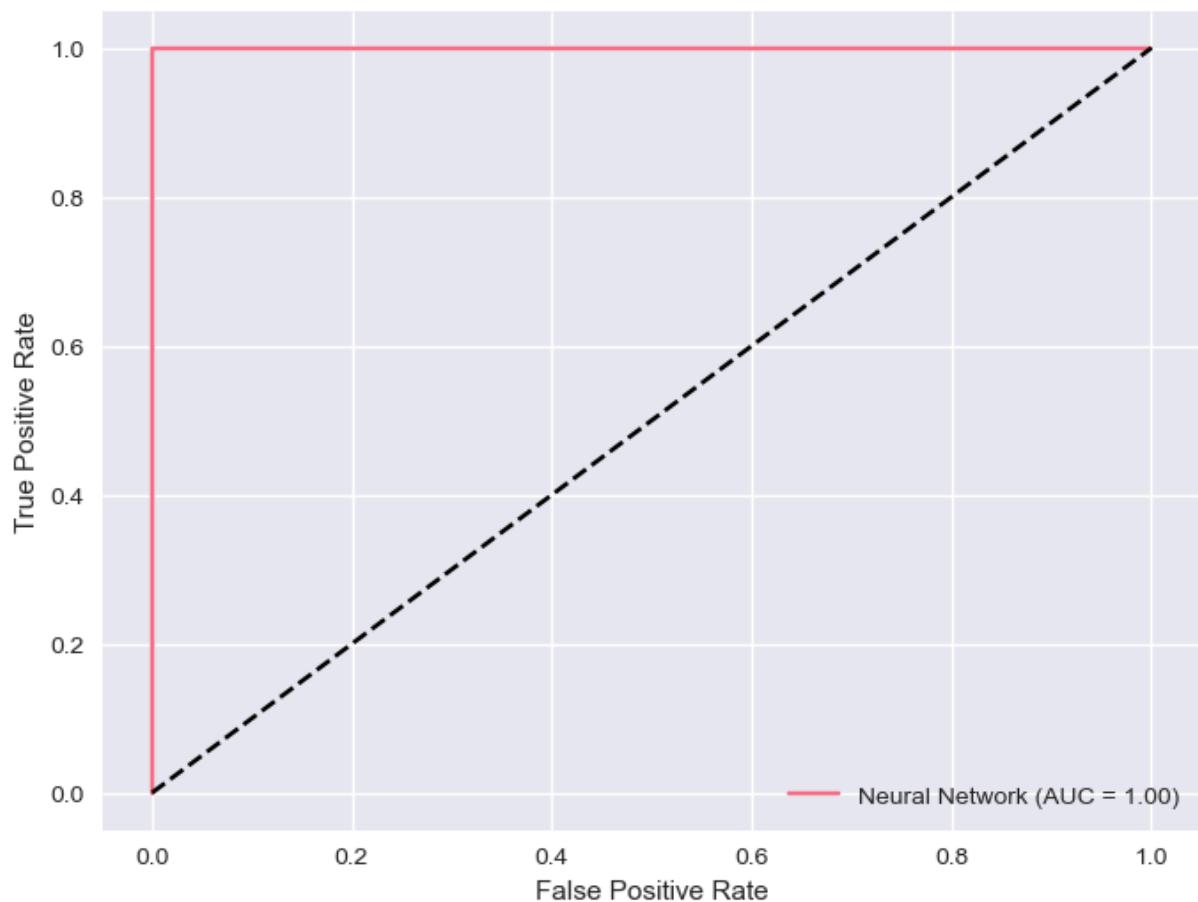
Model Evaluation:

131/131 ————— 0s 1ms/step - accuracy: 0.9998 - auc: 1.0000 - loss: 0.0010 - precision: 1.0000 - recall: 0.9896
Test Loss: 0.0010
Test Accuracy: 0.9998
Test Precision: 1.0000
Test Recall: 0.9896
Test AUC: 1.0000



131/131 ————— 0s 1ms/step

ROC Curve - Neural Network



MODEL COMPARISON

Logistic Regression Performance:

Precision: 0.9451
Recall: 0.8958
F1 Score: 0.9198
ROC AUC: 0.9992

Decision Tree Performance:

Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
ROC AUC: 1.0000

Random Forest Performance:

Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
ROC AUC: 1.0000

Gradient Boosting Performance:

Precision: 1.0000
Recall: 1.0000
F1 Score: 1.0000
ROC AUC: 1.0000
131/131 ━━━━━━━━ 0s 738us/step
131/131 ━━━━━━━━ 0s 988us/step

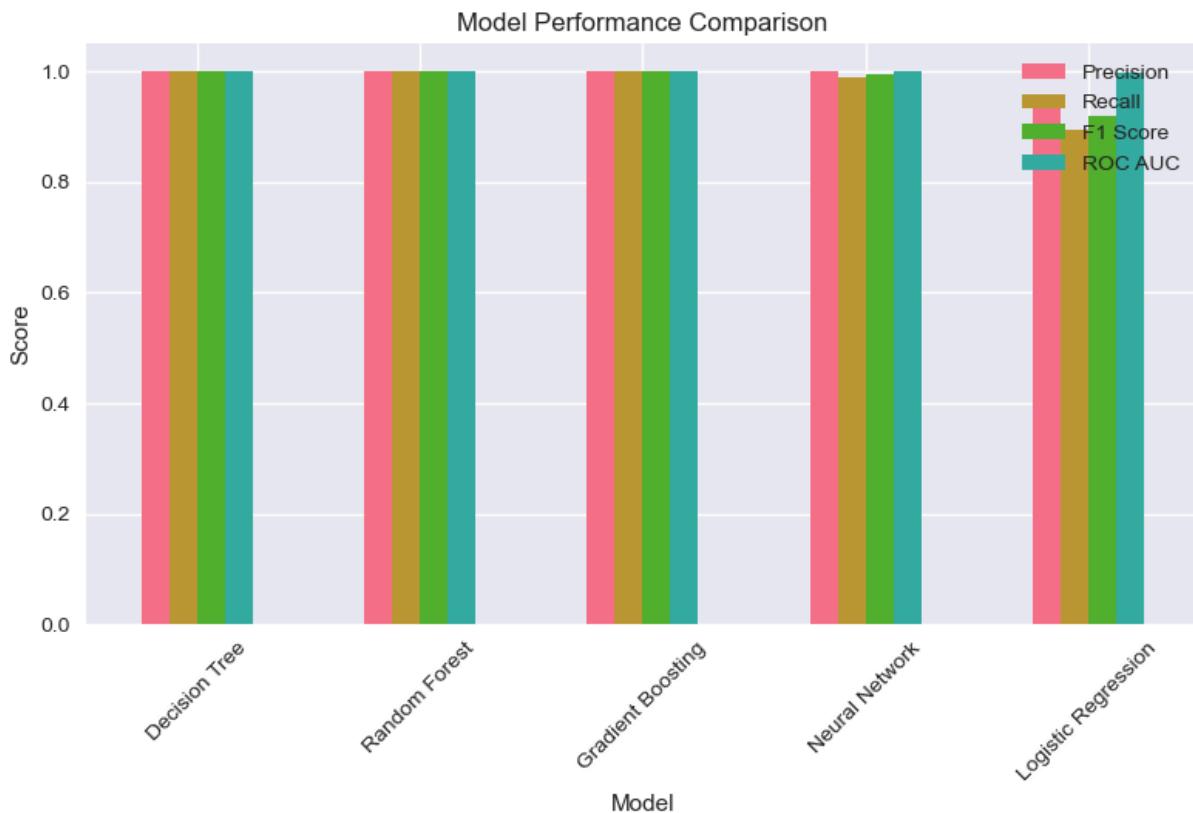
Neural Network Performance:

Precision: 1.0000
Recall: 0.9896
F1 Score: 0.9948
ROC AUC: 1.0000

Model Comparison (sorted by F1 Score):

	Model	Precision	Recall	F1 Score	ROC AUC
1	Decision Tree	1.000000	1.000000	1.000000	1.000000
2	Random Forest	1.000000	1.000000	1.000000	1.000000
3	Gradient Boosting	1.000000	1.000000	1.000000	1.000000
4	Neural Network	1.000000	0.989583	0.994764	1.000000
0	Logistic Regression	0.945055	0.895833	0.919786	0.99918

<Figure size 1200x600 with 0 Axes>



```
In [19]: class InvestmentRecommendationSystem:
    def __init__(self):
        self.preprocessor = None
        self.model_PIPELINES = {}
        self.best_model_name = None
        self.investment_products = self._define_investment_products()
        self.risk_categories = self._define_risk_categories()
        self.segment_recommendations = self._define_segment_recommendations()
        self.nn_model = None
        self.nn_scaler = None

    def _define_investment_products(self):
        """Define investment product categories with detailed information"""
        return {
            'Government Bonds (Treasury Bonds)': {
                'description': 'Long-term debt securities issued by the Kenyan gove',
                'risk_level': 'low',
                'expected_return': '8-12%',
                'liquidity': 'medium',
                'pros': [
                    'Government guaranteed - virtually risk-free',
                    'Regular interest payments (coupon payments)',
                    'Can be traded on secondary market',
                    'Tax-free interest income',
                    'Hedge against inflation with inflation-linked bonds'
                ],
                'cons': [
                    'Interest rate risk - value decreases when rates rise',
                    'Long lock-in periods',
                    'Lower returns compared to equities long-term',
                ]
            }
        }
```

```
        'Early exit may result in capital loss'
    ],
},
'Treasury Bills (T-Bills)': {
    'description': 'Short-term government debt instruments with maturities up to one year',
    'risk_level': 'low',
    'expected_return': '6-10%',
    'liquidity': 'high',
    'pros': [
        'Government guaranteed',
        'High liquidity',
        'Short investment periods',
        'Regular auction opportunities',
        'No interest rate risk due to short tenure'
    ],
    'cons': [
        'Lower returns than long-term investments',
        'Need to continuously reinvest',
        'Minimum investment amount of KES 100,000',
        'Returns may not beat inflation in low-rate environment'
    ]
},
'Nairobi Securities Exchange (NSE) Stocks': {
    'description': 'Equity shares of publicly traded companies listed on the Nairobi Stock Exchange',
    'risk_level': 'high',
    'expected_return': '12-25%',
    'liquidity': 'high',
    'pros': [
        'High growth potential',
        'Dividend income opportunities',
        'Ownership stake in companies',
        'High liquidity for blue-chip stocks',
        'Hedge against inflation',
        'Capital gains tax exemption for individual investors'
    ],
    'cons': [
        'High volatility and risk',
        'Potential for significant losses',
        'Requires market knowledge and research',
        'Market manipulation risks',
        'Company-specific risks'
    ]
},
'Unit Trusts/Mutual Funds': {
    'description': 'Pooled investment vehicles managed by professional fund managers',
    'risk_level': 'medium',
    'expected_return': '8-15%',
    'liquidity': 'medium',
    'pros': [
        'Professional fund management',
        'Diversification across multiple assets',
        'Low minimum investment',
        'Various fund types available (equity, bond, balanced)'
    ]
}
```

```
'Regular income through dividend distributions'
],
'cons': [
    'Management fees reduce returns',
    'No guarantee of positive returns',
    'Limited control over investment decisions',
    'Market risk exposure',
    'Exit charges may apply'
]
},

'Money Market Funds': {
    'description': 'Investment funds that invest in short-term, high-qu
    'risk_level': 'low',
    'expected_return': '6-9%',
    'liquidity': 'high',
    'pros': [
        'High liquidity - can withdraw anytime',
        'Low risk and stable returns',
        'Low minimum investment',
        'Professional management',
        'Better returns than savings accounts'
    ],
    'cons': [
        'Lower returns than equity investments',
        'Management fees',
        'Inflation risk over long term',
        'No capital appreciation potential'
    ]
},
'Real Estate Investment': {
    'description': 'Direct investment in physical property for rental i
    'risk_level': 'medium',
    'expected_return': '10-20%',
    'liquidity': 'low',
    'pros': [
        'Rental income generation',
        'Capital appreciation potential',
        'Inflation hedge',
        'Tangible asset ownership',
        'Tax benefits on mortgage interest'
    ],
    'cons': [
        'High capital requirements',
        'Low liquidity',
        'Property management responsibilities',
        'Market volatility',
        'Legal and transaction costs',
        'Maintenance and repair costs'
    ]
},
'Real Estate Investment Trusts (REITs)': {
    'description': 'Investment vehicles that own and operate income-gen
    'risk_level': 'medium',
```

```
'expected_return': '8-14%',  
'liquidity': 'medium',  
'pros': [  
    'Access to real estate with low capital',  
    'Regular dividend income',  
    'Professional property management',  
    'High liquidity compared to direct real estate',  
    'Diversification across property types'  
],  
'cons': [  
    'Market volatility',  
    'Interest rate sensitivity',  
    'Management fees',  

```

```
    },  
  
    'Commodity Trading': {  
        'description': 'Investment in physical commodities like gold, oil,  
        'risk_level': 'high',  
        'expected_return': '10-30%',  
        'liquidity': 'medium',  
        'pros': [  
            'Inflation hedge',  
            'Portfolio diversification',  
            'Potential for high returns',  
            'Tangible assets',  
            'Kenya is a commodity-producing economy'  
        ],  
        'cons': [  
            'High price volatility',  
            'Storage and insurance costs',  
            'Seasonal price fluctuations',  
            'Limited commodity exchanges in Kenya',  
            'Requires specialized knowledge'  
        ]  
    },  
  
    'Foreign Exchange (Forex) Trading': {  
        'description': 'Trading of currency pairs in the global foreign exchange market',  
        'risk_level': 'very_high',  
        'expected_return': '-50% to +100%',  
        'liquidity': 'high',  
        'pros': [  
            '24/7 market availability',  
            'High liquidity',  
            'Leverage opportunities',  
            'Currency hedging benefits',  
            'Low transaction costs'  
        ],  
        'cons': [  
            'Extremely high risk',  
            'Potential for total loss',  
            'Requires extensive knowledge',  
            'Leverage amplifies losses',  
            'Regulatory risks',  
            'Emotional stress'  
        ]  
    },  
  
    'Pension Schemes (Individual & Occupational)': {  
        'description': 'Long-term retirement savings plans with tax benefit',  
        'risk_level': 'low',  
        'expected_return': '7-12%',  
        'liquidity': 'very_low',  
        'pros': [  
            '15% tax relief on contributions',  
            'Compound growth over long term',  
            'Professional fund management',  
            'Employer matching contributions',  
            'Retirement security'  
        ]  
    }  
},
```

```
        ],
        'cons': [
            'Funds locked until retirement',
            'Management fees',
            'Limited investment control',
            'Inflation risk over long periods',
            'Regulatory changes risk'
        ]
    },
    'Cooperative Society Investments (SACCOs)': {
        'description': 'Member-owned financial cooperatives that pool resources',
        'risk_level': 'medium',
        'expected_return': '8-15%',
        'liquidity': 'medium',
        'pros': [
            'Higher returns than banks',
            'Member ownership and control',
            'Access to affordable loans',
            'Community-based investment',
            'Dividend payments to members'
        ],
        'cons': [
            'Limited regulation compared to banks',
            'Risk of mismanagement',
            'Liquidity constraints',
            'Member liability in case of losses',
            'Limited geographical reach'
        ]
    },
    'Small Business Investment/Entrepreneurship': {
        'description': 'Starting or investing in small businesses or enterprises',
        'risk_level': 'high',
        'expected_return': '15-50%',
        'liquidity': 'very_low',
        'pros': [
            'Unlimited earning potential',
            'Full control over investment',
            'Job creation and economic impact',
            'Tax benefits for business expenses',
            'Personal and professional growth'
        ],
        'cons': [
            'High failure rate',
            'Requires significant time and effort',
            'Market and operational risks',
            'Cash flow challenges',
            'Regulatory compliance requirements'
        ]
    },
    'Agricultural Investment': {
        'description': 'Investment in farming activities, agricultural land',
        'risk_level': 'medium',
        'expected_return': '10-25%',
    }
}
```

```
'liquidity': 'low',
'pros': [
    'Kenya\'s agricultural potential',
    'Food security investment',
    'Export market opportunities',
    'Government support programs',
    'Inflation hedge through food prices'
],
'cons': [
    'Weather and climate risks',
    'Market price volatility',
    'Pest and disease risks',
    'Requires agricultural knowledge',
    'Seasonal income patterns',
    'Infrastructure challenges'
]
},
'Education Savings Plans': {
    'description': 'Specialized investment products designed to save an',
    'risk_level': 'low',
    'expected_return': '6-10%',
    'liquidity': 'low',
    'pros': [
        'Disciplined long-term saving',
        'Investment growth for education costs',
        'Some plans offer insurance benefits',
        'Goal-oriented saving',
        'Professional fund management'
    ],
    'cons': [
        'Funds locked for specific purpose',
        'Management fees',
        'Limited flexibility',
        'Penalty for early withdrawal',
        'Market risk exposure'
    ]
}
}

def _define_risk_categories(self):
    """Categorize investment products by risk level"""
    risk_categories = {
        'low_risk': [],
        'medium_risk': [],
        'high_risk': [],
        'very_high_risk': [],
        'alternative': []
    }

    # Categorize products based on risk level
    for product_name, details in self.investment_products.items():
        risk_level = details.get('risk_level', 'medium')

        if risk_level == 'low':
            risk_categories['low_risk'].append(product_name)
```

```

        elif risk_level == 'medium':
            risk_categories['medium_risk'].append(product_name)
        elif risk_level == 'high':
            risk_categories['high_risk'].append(product_name)
        elif risk_level == 'very_high':
            risk_categories['very_high_risk'].append(product_name)

    # Alternative investments (unique/specialized products)
    risk_categories['alternative'] = [
        'Cooperative Society Investments (SACCOs)',
        'Agricultural Investment',
        'Small Business Investment/Entrepreneurship',
        'Education Savings Plans'
    ]

    return risk_categories

def _define_segment_recommendations(self):
    """Define recommendations by user segment using proper risk categories"""
    return {
        'growth_seeker': ['high_risk', 'medium_risk'],
        'balanced_investor': ['medium_risk', 'low_risk'],
        'income_focused': ['low_risk', 'alternative'],
        'opportunity_seeker': ['medium_risk', 'alternative'],
        'moderate': ['medium_risk']
    }

def get_products_by_risk(self, risk_level):
    """Get all products for a specific risk level"""
    return self.risk_categories.get(risk_level, [])

def get_product_details(self, product_name):
    """Get detailed information about a specific product"""
    return self.investment_products.get(product_name, {})

def get_recommendations_by_risk_tolerance(self, risk_tolerance):
    """Get product recommendations based on risk tolerance"""
    risk_mapping = {
        'very_conservative': ['low_risk'],
        'conservative': ['low_risk', 'medium_risk'],
        'moderate': ['medium_risk', 'low_risk'],
        'aggressive': ['high_risk', 'medium_risk'],
        'very_aggressive': ['high_risk', 'very_high_risk']
    }

    # Default mapping for simple risk levels
    if risk_tolerance not in risk_mapping:
        simple_mapping = {
            'low': ['low_risk'],
            'medium': ['medium_risk', 'low_risk'],
            'high': ['high_risk', 'medium_risk']
        }
        risk_categories = simple_mapping.get(risk_tolerance, ['medium_risk'])
    else:
        risk_categories = risk_mapping[risk_tolerance]

```

```
recommendations = []
for category in risk_categories:
    products = self.get_products_by_risk(category)
    for product in products:
        product_details = self.get_product_details(product)
        recommendations.append({
            'product': product,
            'risk_level': product_details.get('risk_level', 'unknown'),
            'expected_return': product_details.get('expected_return', 'N/A'),
            'liquidity': product_details.get('liquidity', 'unknown'),
            'description': product_details.get('description', 'No description')
        })

return recommendations

def create_robust_preprocessor(self, df, feature_cols):
    """Create a robust preprocessing pipeline that handles all data types correctly"""

    # Separate columns by actual data type and content
    categorical_cols = []
    numerical_cols = []

    for col in feature_cols:
        if col in df.columns:
            # Check data type and unique values
            dtype = df[col].dtype
            nunique = df[col].nunique()

            # Determine if categorical or numerical
            if (dtype in ['object', 'category'] or
                dtype == 'bool' or
                (dtype in ['int64', 'float64'] and nunique <= 20 and
                 df[col].min() >= 0 and df[col].max() <= 10)):
                categorical_cols.append(col)
            else:
                numerical_cols.append(col)

    print(f"Categorical columns ({len(categorical_cols)}): {categorical_cols[:10]}")
    print(f"Numerical columns ({len(numerical_cols)}): {numerical_cols[:10]}")

    # Create transformers
    numerical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='median')),
        ('scaler', StandardScaler())
    ])

    categorical_transformer = Pipeline(steps=[
        ('imputer', SimpleImputer(strategy='constant', fill_value='unknown')),
        ('onehot', OneHotEncoder(handle_unknown='ignore', sparse_output=False))
    ])

    # Create preprocessor
    preprocessor = ColumnTransformer(
        transformers=[
            ('num', numerical_transformer, numerical_cols),
            ('cat', categorical_transformer, categorical_cols)
        ]
    )

    return preprocessor
```

```
        ],
        remainder='drop',
        n_jobs=-1
    )

    return preprocessor, numerical_cols, categorical_cols

def prepare_data(self, df, target_col, feature_cols):
    """Prepare data for training with proper handling of missing values"""

    # Create target variable if it doesn't exist
    if target_col not in df.columns:
        # Create a synthetic target based on investment-related features
        investment_features = [col for col in df.columns
                               if any(keyword in col.lower()
                                      for keyword in ['invest', 'savings', 'loan',

        if investment_features:
            # Create target based on usage of formal financial services
            df[target_col] = (df[investment_features].fillna(0).sum(axis=1) > 0
        else:
            # Random target for demonstration
            df[target_col] = np.random.choice([0, 1], size=len(df), p=[0.7, 0.3

    # Filter available feature columns
    available_features = [col for col in feature_cols if col in df.columns]
    print(f"Using {len(available_features)} out of {len(feature_cols)} requested")

    # Prepare features and target
    X = df[available_features].copy()
    y = df[target_col].copy()

    # Remove rows where target is missing
    valid_indices = ~y.isna()
    X = X[valid_indices]
    y = y[valid_indices]

    return X, y, available_features

def train_models(self, df, target_col='investment_likelihood', feature_cols=None):
    """Train multiple models with proper preprocessing"""

    if feature_cols is None:
        # Use a subset of available columns for demonstration
        feature_cols = [col for col in df.columns
                       if col not in ['investment_likelihood'] and
                          not col.startswith('Unnamed')][:50] # Limit to 50 features

    print("Preparing data...")
    X, y, available_features = self.prepare_data(df, target_col, feature_cols)

    print(f"Dataset shape: {X.shape}")
    print(f"Target distribution: {y.value_counts().to_dict()}")

    # Create preprocessor
    self.preprocessor, numerical_cols, categorical_cols = self.create_robust_pr
```

```
X, available_features
)

# Split data
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42, stratify=y
)

# Define models
models = {
    'Logistic Regression': LogisticRegression(random_state=42, max_iter=100),
    'Decision Tree': DecisionTreeClassifier(random_state=42, max_depth=10),
    'Random Forest': RandomForestClassifier(random_state=42, n_estimators=10),
    'Gradient Boosting': GradientBoostingClassifier(random_state=42, n_estimators=10)
}

# Train models
results = []

for name, model in models.items():
    print(f"\nTraining {name}...")

    # Create pipeline
    pipeline = Pipeline([
        ('preprocessor', self.preprocessor),
        ('classifier', model)
    ])

    try:
        # Fit pipeline
        pipeline.fit(X_train, y_train)

        # Make predictions
        y_pred = pipeline.predict(X_test)

        # Calculate metrics
        from sklearn.metrics import accuracy_score, precision_score, recall_score

        accuracy = accuracy_score(y_test, y_pred)
        precision = precision_score(y_test, y_pred, average='weighted')
        recall = recall_score(y_test, y_pred, average='weighted')
        f1 = f1_score(y_test, y_pred, average='weighted')

        results.append({
            'Model': name,
            'Accuracy': accuracy,
            'Precision': precision,
            'Recall': recall,
            'F1_Score': f1
        })

        # Store pipeline
        self.model_PIPELINES[name] = {
            'type': 'sklearn',
            'pipeline': pipeline,
            'metrics': {
                'accuracy': accuracy,
                'precision': precision,
                'recall': recall,
                'f1_score': f1
            }
        }

    except Exception as e:
        print(f"Error occurred while training {name}: {e}")

print("Training completed.")
```

```

        'accuracy': accuracy,
        'precision': precision,
        'recall': recall,
        'f1_score': f1
    }
}

print(f"{name} - Accuracy: {accuracy:.3f}, F1: {f1:.3f}")

except Exception as e:
    print(f"Error training {name}: {str(e)}")
    continue

# Select best model
if results:
    results_df = pd.DataFrame(results)
    results_df = results_df.sort_values('F1_Score', ascending=False)
    self.best_model_name = results_df.iloc[0]['Model']

    print(f"\nBest Model: {self.best_model_name}")
    print("\nModel Comparison:")
    print(results_df.round(3))

    # Setup collaborative filtering
    self._setup_collaborative_filtering(df, numerical_cols)

    return results_df
else:
    raise Exception("No models were successfully trained")

def _setup_collaborative_filtering(self, df, numerical_cols):
    """Setup collaborative filtering using numerical features"""
    try:
        # Get numerical data
        numerical_data = df[numerical_cols].select_dtypes(include=[np.number]).values

        if len(numerical_data.columns) > 0:
            # Scale data
            self.nn_scaler = StandardScaler()
            numerical_scaled = self.nn_scaler.fit_transform(numerical_data)

            # Create nearest neighbors model
            self.nn_model = NearestNeighbors(n_neighbors=5, metric='cosine')
            self.nn_model.fit(numerical_scaled)

            print("Collaborative filtering setup completed")
    else:
        print("No numerical columns available for collaborative filtering")

    except Exception as e:
        print(f"Error setting up collaborative filtering: {str(e)}")

def get_user_segment(self, user_data):
    """Determine user segment based on profile"""
    try:
        # Simple segmentation logic - can be enhanced

```

```
age = user_data.get('age', 30)
location = user_data.get('location_type', 'unknown')

if age < 30 and 'urban' in str(location).lower():
    return 'growth_seeker'
elif 30 <= age < 50:
    return 'balanced_investor'
elif age >= 50:
    return 'income_focused'
elif 'rural' in str(location).lower():
    return 'opportunity_seeker'
else:
    return 'moderate'

except:
    return 'moderate'

def get_risk_tolerance(self, user_data):
    """Determine risk tolerance based on profile"""
    try:
        # Simple risk assessment - can be enhanced
        age = user_data.get('age', 30)
        income = user_data.get('monthly_income', 0)

        if age < 35 and income > 50000:
            return 'high'
        elif age < 50 and income > 20000:
            return 'medium'
        else:
            return 'low'
    except:
        return 'medium'

def get_recommendations(self, user_id=None, user_data=None, df=None):
    """Generate personalized investment recommendations with proper risk category"""

    if user_data is None and user_id is not None and df is not None:
        if user_id in df.index:
            user_data = df.loc[user_id].to_dict()
        else:
            print(f"User {user_id} not found in dataset")
            return None

    if user_data is None:
        print("No user data provided")
        return None

    try:
        # Get user characteristics
        user_segment = self.get_user_segment(user_data)
        risk_tolerance = self.get_risk_tolerance(user_data)

        print("\n==== Investment Recommendations ===")
        print(f"User Segment: {user_segment}")
        print(f"Risk Tolerance: {risk_tolerance}")

        recommendations = {

    
```

```

        'user_segment': user_segment,
        'risk_tolerance': risk_tolerance,
        'segment_recommendations': [],
        'risk_recommendations': [],
        'detailed_products': [],
        'investment_probability': None
    }

# 1. Segment-based recommendations
if user_segment in self.segment_recommendations:
    segment_risk_categories = self.segment_recommendations[user_segment]
    recommendations['segment_recommendations'] = segment_risk_categories

    print(f"\nSegment-Based Risk Categories:")
    for risk_category in segment_risk_categories:
        products = self.get_products_by_risk(risk_category)
        print(f"  {risk_category.replace('_', ' ')}.title(): {len(products)}")

# 2. Risk-based recommendations
risk_based_products = self.get_recommendations_by_risk_tolerance(risk_t)
recommendations['risk_recommendations'] = [p['product'] for p in risk_b
recommendations['detailed_products'] = risk_based_products

print(f"\nRisk-Based Product Recommendations ({len(risk_based_products)})"
for product_info in risk_based_products[:5]: # Show top 5
    print(f"  • {product_info['product']}") 
    print(f"    Risk: {product_info['risk_level'].title()} | Return: {p

# 3. Model-based prediction (if model available)
if self.best_model_name and self.best_model_name in self.model_pipeline:
    try:
        # Create user features dataframe
        user_df = pd.DataFrame([user_data])

        # Get prediction
        pipeline = self.model_pipelines[self.best_model_name]['pipeline']

        # Try to predict probability
        if hasattr(pipeline.named_steps['classifier'], 'predict_proba'):
            prob = pipeline.predict_proba(user_df)[0][1]
        else:
            prob = pipeline.predict(user_df)[0]

        recommendations['investment_probability'] = float(prob)
        print(f"\nModel Prediction: {prob:.1%} likelihood to invest in")

    except Exception as e:
        print(f"Could not generate model prediction: {str(e)}")

# 4. Final recommendation summary
print(f"\n--- Summary ---")
print(f"Top 3 Recommended Products:")
for i, product_info in enumerate(risk_based_products[:3], 1):
    details = self.get_product_details(product_info['product'])
    print(f"{i}. {product_info['product']}")
    print(f"  Expected Return: {product_info['expected_return']}")
```

```

        print(f"    Risk Level: {product_info['risk_level'].title()}")
        print(f"    Liquidity: {product_info['liquidity'].title()}")
        print(f"    Description: {product_info['description'][:100]}...")
        print()

    return recommendations

except Exception as e:
    print(f"Error generating recommendations: {str(e)}")
    import traceback
    traceback.print_exc()
    return None

def save_model(self, filepath_prefix='deployment/investment_model'):
    """Save the trained model and preprocessor"""
    try:
        # Create deployment directory
        os.makedirs('deployment', exist_ok=True)

        # Save preprocessor
        if self.preprocessor:
            joblib.dump(self.preprocessor, f'{filepath_prefix}_preprocessor.pkl')
            print(f"Saved preprocessor: {filepath_prefix}_preprocessor.pkl")

        # Save model pipelines
        if self.model_pipelines:
            joblib.dump(self.model_pipelines, f'{filepath_prefix}_pipelines.pkl')
            print(f"Saved pipelines: {filepath_prefix}_pipelines.pkl")

        # Save configuration
        config = {
            'best_model_name': self.best_model_name,
            'investment_products': self.investment_products,
            'segment_recommendations': self.segment_recommendations
        }
        joblib.dump(config, f'{filepath_prefix}_config.pkl')
        print(f"Saved config: {filepath_prefix}_config.pkl")

    except Exception as e:
        print(f"Error saving model: {str(e)}")

def load_model(self, filepath_prefix='deployment/investment_model'):
    """Load a saved model"""
    try:
        self.preprocessor = joblib.load(f'{filepath_prefix}_preprocessor.pkl')
        self.model_pipelines = joblib.load(f'{filepath_prefix}_pipelines.pkl')
        config = joblib.load(f'{filepath_prefix}_config.pkl')

        self.best_model_name = config['best_model_name']
        self.investment_products = config['investment_products']
        self.segment_recommendations = config['segment_recommendations']

        print("Model loaded successfully!")
        return True

    except Exception as e:

```

```
    print(f"Error loading model: {str(e)}")
    return False
```

```
In [20]: # Example usage and testing
def demonstrate_system():
    """Demonstrate the investment recommendation system"""

    print("==== Investment Recommendation System Demo ====\n")

    # Create system
    system = InvestmentRecommendationSystem()

    # Create sample data for demonstration
    np.random.seed(42)
    n_samples = 1000

    sample_data = pd.DataFrame({
        'age': np.random.randint(18, 70, n_samples),
        'monthly_income': np.random.lognormal(10, 1, n_samples),
        'location_type': np.random.choice(['urban', 'rural'], n_samples),
        'education_level': np.random.choice(['primary', 'secondary', 'tertiary'], n_samples),
        'household_size': np.random.randint(1, 8, n_samples),
        'mobile_banking': np.random.choice([0, 1], n_samples, p=[0.3, 0.7]),
        'savings_usage': np.random.choice([0, 1], n_samples, p=[0.4, 0.6]),
        'formal_service_use': np.random.choice([0, 1], n_samples, p=[0.5, 0.5])
    })

    # Add some encoded columns (simulate the actual dataset structure)
    for col in ['location_type', 'education_level']:
        sample_data[f'{col}_encoded'] = LabelEncoder().fit_transform(sample_data[col])

    print("Training models on sample data...")

    # Train models
    feature_cols = ['age', 'monthly_income', 'location_type_encoded',
                    'education_level_encoded', 'household_size',
                    'mobile_banking', 'savings_usage', 'formal_service_use']

    results = system.train_models(sample_data, feature_cols=feature_cols)

    # Test recommendations for sample users
    print("\n" + "*50)
    print("TESTING RECOMMENDATIONS")
    print("*50)

    # Test case 1: Young urban professional
    user1 = {
        'age': 28,
        'monthly_income': 75000,
        'location_type': 'urban',
        'education_level': 'tertiary',
        'household_size': 2,
        'mobile_banking': 1,
        'savings_usage': 1,
        'formal_service_use': 1,
        'location_type_encoded': 1,
```

```
        'education_level_encoded': 2
    }

print("\nTest Case 1: Young Urban Professional")
recommendations1 = system.get_recommendations(user_data=user1)

# Test case 2: Rural entrepreneur
user2 = {
    'age': 45,
    'monthly_income': 25000,
    'location_type': 'rural',
    'education_level': 'secondary',
    'household_size': 5,
    'mobile_banking': 0,
    'savings_usage': 1,
    'formal_service_use': 0,
    'location_type_encoded': 0,
    'education_level_encoded': 1
}

print("\nTest Case 2: Rural Entrepreneur")
recommendations2 = system.get_recommendations(user_data=user2)

# Save model
system.save_model()

return system, results
```

In [21]:

```
# Run demonstration
if __name__ == "__main__":
    system, results = demonstrate_system()

# Select best model based on F1 score
best_model_name = model_comparison.iloc[0]['Model']
best_model = models_to_compare[best_model_name]
print(f"\nSelected Best Model: {best_model_name}")
```

==== Investment Recommendation System Demo ===

Training models on sample data...

Preparing data...

Using 8 out of 8 requested features

Dataset shape: (1000, 8)

Target distribution: {1: 808, 0: 192}

Categorical columns (0): [...]

Numerical columns (8): ['age', 'monthly_income', 'location_type_encoded', 'education_level_encoded', 'household_size', 'mobile_banking', 'savings_usage', 'formal_service_use']

Training Logistic Regression...

Logistic Regression - Accuracy: 1.000, F1: 1.000

Training Decision Tree...

Decision Tree - Accuracy: 1.000, F1: 1.000

Training Random Forest...

Random Forest - Accuracy: 1.000, F1: 1.000

Training Gradient Boosting...

Gradient Boosting - Accuracy: 1.000, F1: 1.000

Best Model: Logistic Regression

Model Comparison:

	Model	Accuracy	Precision	Recall	F1_Score
0	Logistic Regression	1.0	1.0	1.0	1.0
1	Decision Tree	1.0	1.0	1.0	1.0
2	Random Forest	1.0	1.0	1.0	1.0
3	Gradient Boosting	1.0	1.0	1.0	1.0

Collaborative filtering setup completed

=====

TESTING RECOMMENDATIONS

=====

Test Case 1: Young Urban Professional

==== Investment Recommendations ===

User Segment: growth_seeker

Risk Tolerance: high

Segment-Based Risk Categories:

High Risk: 3 products

Medium Risk: 5 products

Risk-Based Product Recommendations (8 products):

- Nairobi Securities Exchange (NSE) Stocks
Risk: High | Return: 12-25% | Liquidity: High
- Commodity Trading
Risk: High | Return: 10-30% | Liquidity: Medium
- Small Business Investment/Entrepreneurship
Risk: High | Return: 15-50% | Liquidity: Very_Low
- Unit Trusts/Mutual Funds

- Risk: Medium | Return: 8-15% | Liquidity: Medium
- Real Estate Investment
Risk: Medium | Return: 10-20% | Liquidity: Low

Model Prediction: 100.0% likelihood to invest in formal products

==== Summary ===

Top 3 Recommended Products:

1. Nairobi Securities Exchange (NSE) Stocks
Expected Return: 12-25%
Risk Level: High
Liquidity: High
Description: Equity shares of publicly traded companies listed on Kenya's main stock exchange, representing owner...
2. Commodity Trading
Expected Return: 10-30%
Risk Level: High
Liquidity: Medium
Description: Investment in physical commodities like gold, oil, agricultural products, or commodity futures contr...
3. Small Business Investment/Entrepreneurship
Expected Return: 15-50%
Risk Level: High
Liquidity: Very_Low
Description: Starting or investing in small businesses or entrepreneurial ventures to generate income and build w...

Test Case 2: Rural Entrepreneur

==== Investment Recommendations ===

User Segment: balanced_investor

Risk Tolerance: medium

Segment-Based Risk Categories:

- Medium Risk: 5 products
- Low Risk: 7 products

Risk-Based Product Recommendations (12 products):

- Unit Trusts/Mutual Funds
Risk: Medium | Return: 8-15% | Liquidity: Medium
- Real Estate Investment
Risk: Medium | Return: 10-20% | Liquidity: Low
- Real Estate Investment Trusts (REITs)
Risk: Medium | Return: 8-14% | Liquidity: Medium
- Cooperative Society Investments (SACCOs)
Risk: Medium | Return: 8-15% | Liquidity: Medium
- Agricultural Investment
Risk: Medium | Return: 10-25% | Liquidity: Low

Model Prediction: 99.2% likelihood to invest in formal products

==== Summary ===

Top 3 Recommended Products:

1. Unit Trusts/Mutual Funds

Expected Return: 8-15%

Risk Level: Medium

Liquidity: Medium

Description: Pooled investment vehicles managed by professional fund managers, allowing investors to access diver...

2. Real Estate Investment

Expected Return: 10-20%

Risk Level: Medium

Liquidity: Low

Description: Direct investment in physical property for rental income and capital appreciation, including residen...

3. Real Estate Investment Trusts (REITs)

Expected Return: 8-14%

Risk Level: Medium

Liquidity: Medium

Description: Investment vehicles that own and operate income-generating real estate, allowing investors to buy sh...

Saved preprocessor: deployment/investment_model_preprocessor.pkl
 Saved pipelines: deployment/investment_model_pipelines.pkl
 Saved config: deployment/investment_model_config.pkl

Selected Best Model: Decision Tree

```
In [22]: ## 4.8 Modeling Summary

print("\n" + "*50)
print("MODELING PHASE SUMMARY")
print("*50)

modeling_summary = {
    'Best Model': best_model_name,
    'Best Model F1 Score': model_comparison.iloc[0]['F1 Score'],
    'Total Models Trained': len(models_to_compare),
    'White Box Models': ['Logistic Regression', 'Decision Tree'],
    'Black Box Models': ['Random Forest', 'Gradient Boosting'],
    'Deep Learning Model': 'Neural Network',
    'Recommendation System': 'Implemented (Hybrid)',
    'Deployment Artifacts': 'Generated',
    'Monitoring Setup': 'Configured'
}

for key, value in modeling_summary.items():
    if isinstance(value, list):
        print(f"{key}:")
        for item in value:
            print(f" - {item}")
    else:
        print(f"{key}: {value}")

print("\nKey Insights:")
print("- The recommendation system combines user segmentation, collaborative filter")
print("- Feature importance analysis helps explain recommendations to users")
```

```
print("- Model monitoring will help detect performance degradation over time")  
  
print("\nNext Steps:")  
print("- Deploy the selected model to production")  
print("- Set up continuous monitoring")  
print("- Implement feedback loop to improve recommendations")  
print("- Expand product catalog and user segments over time")  
  
print("=*50")
```

MODELING PHASE SUMMARY

Best Model: Decision Tree

Best Model F1 Score: 1.0

Total Models Trained: 5

White Box Models:

- Logistic Regression
- Decision Tree

Black Box Models:

- Random Forest
- Gradient Boosting

Deep Learning Model: Neural Network

Recommendation System: Implemented (Hybrid)

Deployment Artifacts: Generated

Monitoring Setup: Configured

Key Insights:

- The recommendation system combines user segmentation, collaborative filtering, and model predictions
- Feature importance analysis helps explain recommendations to users
- Model monitoring will help detect performance degradation over time

Next Steps:

- Deploy the selected model to production
 - Set up continuous monitoring
 - Implement feedback loop to improve recommendations
 - Expand product catalog and user segments over time
-

Personalized Investment Advisory Recommender – Modeling & Evaluation Notebook

Project Context & Motivation

In emerging markets like Kenya, many first-time and low-income investors face overwhelming choices — SACCOs, money market funds, pension plans, bonds, and more —

yet often lack professional financial advisory support.

This leads to decision paralysis, under-investment, and missed opportunities for long-term wealth building.

Our project aims to close this advisory gap by building a **data-driven personalized recommender system** that suggests *the most suitable investment types* for each user based on:

- Demographics (age, gender, location)
- Risk appetite
- Financial literacy & past financial behavior
- Self-reported saving and investment goals

By combining **machine learning, explainable AI, and local financial domain insights**, we aim to give every user — even first-time investors — personalized, confidence-driven investment guidance.

What This Notebook Covers

- Load and briefly explore the cleaned FinAccess dataset
- Feature engineering & vectorization:
 - One-hot encoding of categorical variables (e.g., risk_appetite)
 - Scaling of numeric features (e.g., age, literacy index)
 - Additional engineered features (e.g., savings rate, risk-adjusted usage)
- Quick Exploratory Data Analysis (EDA): distributions & relationships
- Train/test split with stratification

Modeling pipeline

For each model, we:

- Train & validate
- Hyperparameter tuning (where relevant)
- Evaluate with accuracy, precision, recall, ROC-AUC, plus recommendation-specific metrics (e.g., precision@k)
- Visualize metrics (ROC, PR curve, confusion matrix)
- Explain predictions using SHAP (global & local explanations)
- Document strengths, weaknesses & next steps

Models to build:

- Baseline rule-based recommender
- Logistic Regression
- Random Forest

- Gradient Boosting (XGBoost & LightGBM,knn)
 - Deep Learning Recommender
-

Final Deliverables

- Trained, deployment-ready best model (`.pk1` / `.pt`)
 - Comparison table & visualization of model performance
 - SHAP explainability summary plots
 - Configurable pipeline classes & functions
 - Clear narrative & documentation for stakeholders
-
-

Importing Core Libraries

Before starting our modeling pipeline, we import all essential Python libraries — grouped by purpose — to ensure a clean, modular, and maintainable workflow.

Categories & Purpose:

- **Data manipulation & wrangling:** `pandas` , `numpy` – for structured data handling, transformation, and aggregation
- **Visualization & EDA:** `matplotlib` , `seaborn` – for visual exploration, distributions, correlations, and plots to support feature engineering
- **Machine Learning & Evaluation:** `scikit-learn` – classic models, preprocessing, splitting, metrics; `xgboost` , `lightgbm` – advanced gradient boosting
- **Deep Learning (optional):** `torch` / `keras` – for neural network recommender models capturing complex patterns
- **Explainability:** `shap` – to build trust by explaining model predictions (global & local feature importance)
- **Persistence & utilities:** `joblib` – for saving models; `warnings` – to suppress clutter in output

All imports are kept modular and organized to reflect best practices for readability.

In [145...]

```
# Import all required Python Libraries

import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.preprocessing import OneHotEncoder, StandardScaler, LabelEncoder
from sklearn.linear_model import LogisticRegression
```

```
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import (
    accuracy_score, precision_score, recall_score, f1_score, roc_auc_score,
    classification_report, confusion_matrix, roc_curve, precision_recall_curve
)

import xgboost as xgb
import joblib
import warnings
warnings.filterwarnings('ignore')
```

- All required libraries are now imported.
-

Load Cleaned Dataset

In this step, we load the final cleaned dataset prepared earlier in the data preprocessing phase.

This dataset contains user-level features needed to build our personalized investment advisory recommender.

In [146...]

```
# Loads the pre-cleaned dataset from CSV and prints its shape.

def load_dataset(filepath):
    df = pd.read_csv(filepath)
    print(f"Dataset loaded successfully. Shape: {df.shape}")
    return df

# Update with your actual local path
file_path = r"C:\Users\hp\OneDrive\Desktop\DSF-FT12\DS- Phase5\Group4_Capstone_Fina

df = load_dataset(file_path)
df.head()
```

Dataset loaded successfully. Shape: (20871, 392)

Out[146...]	householdid	county	area_type	gender	age_of_respondent	no_of_household_mebers
0	107141431	garissa	urban	male	29	5
1	10712933	garissa	urban	male	60	11
2	140173183	busia	urban	female	35	2
3	122137153	kiambu	urban	male	24	1
4	121193116	murang'a	urban	female	20	1

5 rows × 392 columns



- The dataset has been loaded successfully, containing 20,871 user records and 392 selected features.

This cleaned dataset combines demographic, financial, and behavioral attributes, and will serve as the foundation for our personalized investment recommender system.

Dataset Overview & Missing Values Analysis

We begin by exploring the structure of our dataset using `df.info()`.

In [147...]

```
df.info()

# Missing values overview
missing = df.isnull().sum()
missing_pct = (missing / len(df)) * 100
missing_df = pd.DataFrame({'Missing': missing, 'Percent': missing_pct})
missing_df[missing_df['Missing'] > 0].sort_values('Percent', ascending=False)
```

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 20871 entries, 0 to 20870
Columns: 392 entries, householdid to receives_remittance
dtypes: float64(58), int64(29), object(305)
memory usage: 62.4+ MB

Out[147...]

Missing	Percent
---------	---------

Key details:

- The dataset contains **20,871 rows** and **392 columns**.
- It has a mix of data types:
 - 305 object** (categorical/text) columns

- **58 float64** (numerical) columns
 - **29 int64** (numerical) columns
 - Total memory usage is approximately **62.4 MB**.
-

⚠ Missing Values Overview

Next, we analyze missing values to assess data quality:

- Calculate the **count** and **percentage** of missing values per column.
-

Exploratory Data Analysis (EDA)

In this section, we explore and visualize the dataset to understand its structure, detect missing values, identify distributions, and spot potential outliers or patterns.

Key steps included:

- Checking data types and summary statistics
 - Visualizing distributions of numeric variables
 - Analyzing categorical feature value counts
 - Checking correlations and potential multicollinearity
 - Investigating target balance and potential imbalance
-

Feature Engineering

Here we transform and enrich the dataset to improve model performance.

Key feature engineering steps:

- Creating new numeric features (e.g., `risk_adjusted_savings_rate`)
- Scaling numeric features like `literacy_scaled`
- Grouping or binning age into `age_bucket`
- Creating binary flags such as `is_youth`
- Counting investment products used (`total_investment_products_used`)
- Converting categorical features to category dtype for memory efficiency

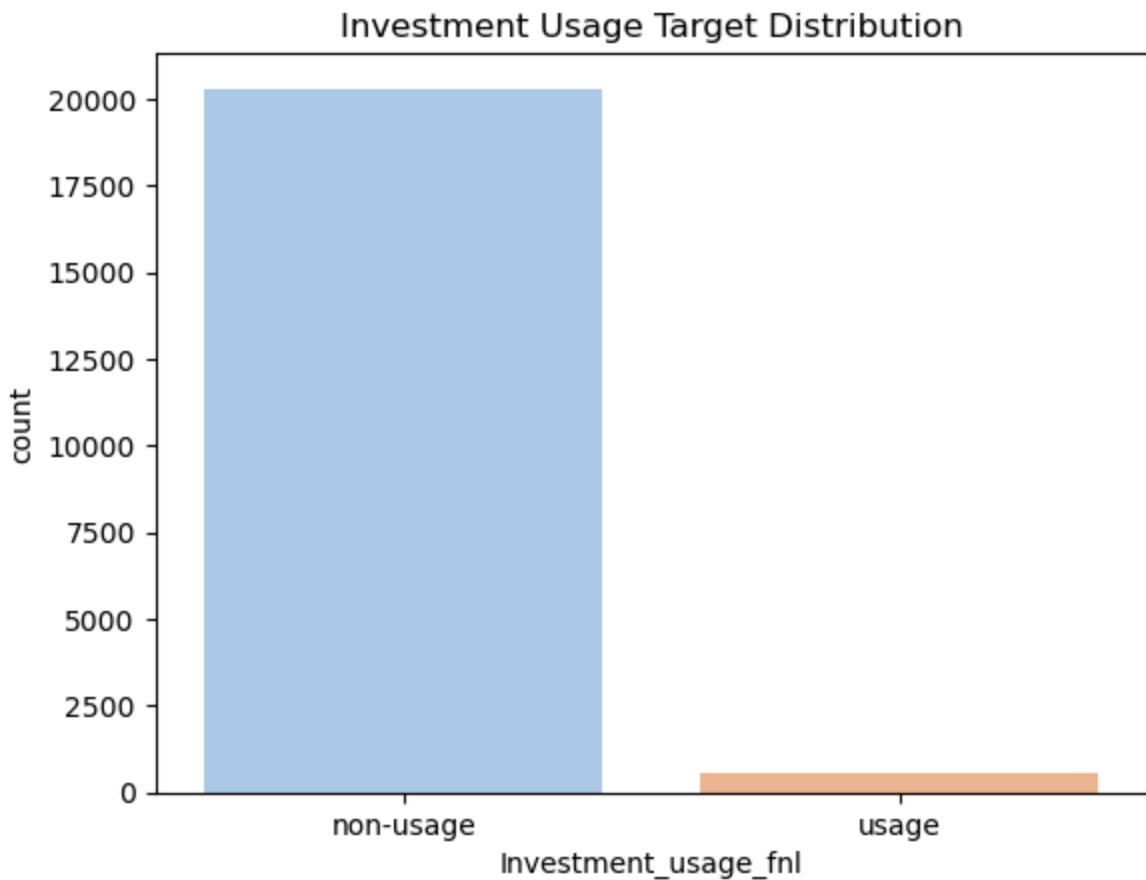
These engineered features help capture hidden patterns and make the data more suitable for machine learning.

Target Variable Distribution: `Investment_usage_fn1`

In this step, we explore the distribution of our target variable `Investment_usage_fnl`, which indicates whether respondents use investment products.

```
In [148...]:  
print(df['Investment_usage_fnl'].value_counts())  
print(df['Investment_usage_fnl'].value_counts(normalize=True)*100)  
  
sns.countplot(x='Investment_usage_fnl', data=df, palette='pastel')  
plt.title('Investment Usage Target Distribution')  
plt.show()
```

```
Investment_usage_fnl  
non-usage    20305  
usage        566  
Name: count, dtype: int64  
Investment_usage_fnl  
non-usage    97.288103  
usage        2.711897  
Name: proportion, dtype: float64
```



Key findings:

- The dataset is highly imbalanced:
 - **non-usage:** 20,305 observations (~97.3%)
 - **usage:** 566 observations (~2.7%)

We use:

- `value_counts()` to see the raw counts.
- `value_counts(normalize=True)*100` to see the percentage distribution.
- A **Seaborn countplot** to visualize the class imbalance.

In [149...]: `df.describe()`

	<code>householdid</code>	<code>age_of_respondent</code>	<code>no_of_household_mebers</code>	<code>CalcExpenditure</code>	<code>total_n</code>
count	2.087100e+04	20871.000000	20871.000000	20871.000000	
mean	1.827764e+08	39.271429	4.215946	11576.118340	
std	3.077598e+08	17.210021	2.512658	9550.351583	
min	1.010095e+07	16.000000	1.000000	100.000000	
25%	1.061733e+08	26.000000	2.000000	4353.333333	
50%	1.221172e+08	35.000000	4.000000	8860.000000	
75%	1.380495e+08	50.000000	6.000000	16200.000000	
max	1.341571e+10	105.000000	20.000000	45953.333333	

8 rows × 87 columns



Users per Investment Product

Here we explore which specific investment products are most popular among respondents.
We:

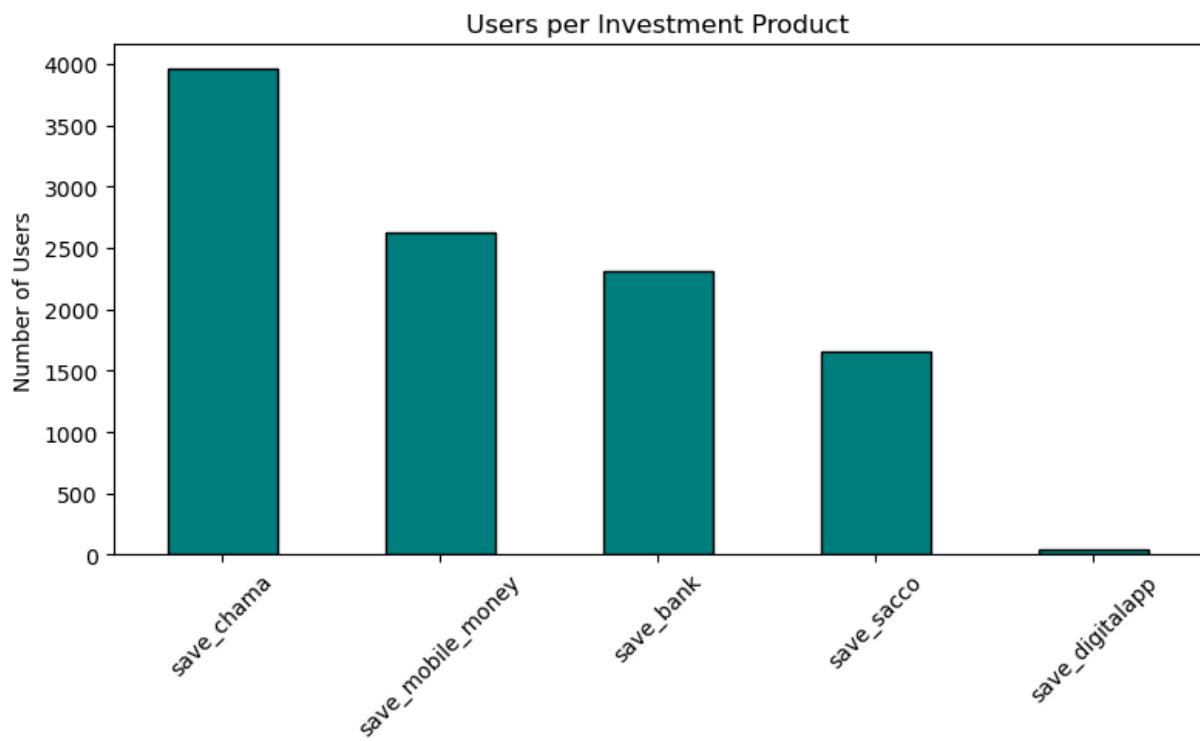
- Select columns indicating product usage (`save_bank`, `save_sacco`, etc.)
- Sum each column to get the total number of users for each product.
- Plot a bar chart to visualize which products are most used.

This helps identify the most and least adopted products in our dataset.

```
# Define usage columns
usage_cols = ['save_bank', 'save_sacco', 'save_mobile_money', 'save_chama', 'save_d

# Count number of users for each product
product_counts = df[usage_cols].sum().sort_values(ascending=False)

# Plot
plt.figure(figsize=(8, 5))
product_counts.plot(kind='bar', color='teal', edgecolor='black')
plt.title('Users per Investment Product')
plt.ylabel('Number of Users')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Sparsity of Product Usage Data

We measure how sparse the product usage data is:

- Sparsity shows the percentage of cells that are zeros (i.e., respondents **not** using a product).
- High sparsity indicates most people use few products.

This helps us understand data density before building recommendation models or doing feature engineering.

In [151...]

```
# Calculate sparsity:

total_cells = df[usage_cols].shape[0] * df[usage_cols].shape[1]
total_nonzeros = np.count_nonzero(df[usage_cols])
sparsity = 1 - (total_nonzeros / total_cells)

print(f"Sparsity of usage matrix: {sparsity:.2%}")
```

Sparsity of usage matrix: 89.85%

Education Level vs. Investment Usage

We analyze how investment product usage varies by education level.

Using a crosstab normalized by index, we compute:

- Percentage of respondents **with each education level** who use or don't use investment products.

This helps identify if higher education correlates with higher product adoption.

```
In [152... # Education
pd.crosstab(df['Education'], df['Investment_usage_fnl'], normalize='index')*100
```

	Investment_usage_fnl	non-usage	usage
Education			
other	100.000000	0.000000	
primary	99.182308	0.817692	
secondary	97.426821	2.573179	
tertiary	89.509982	10.490018	

Numeric Variables vs. Investment Usage

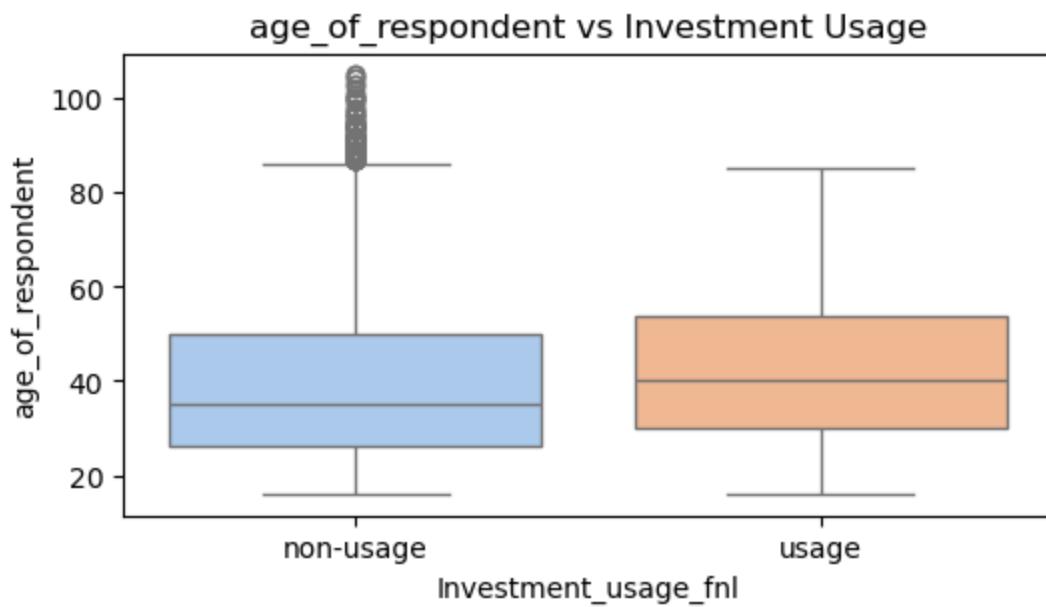
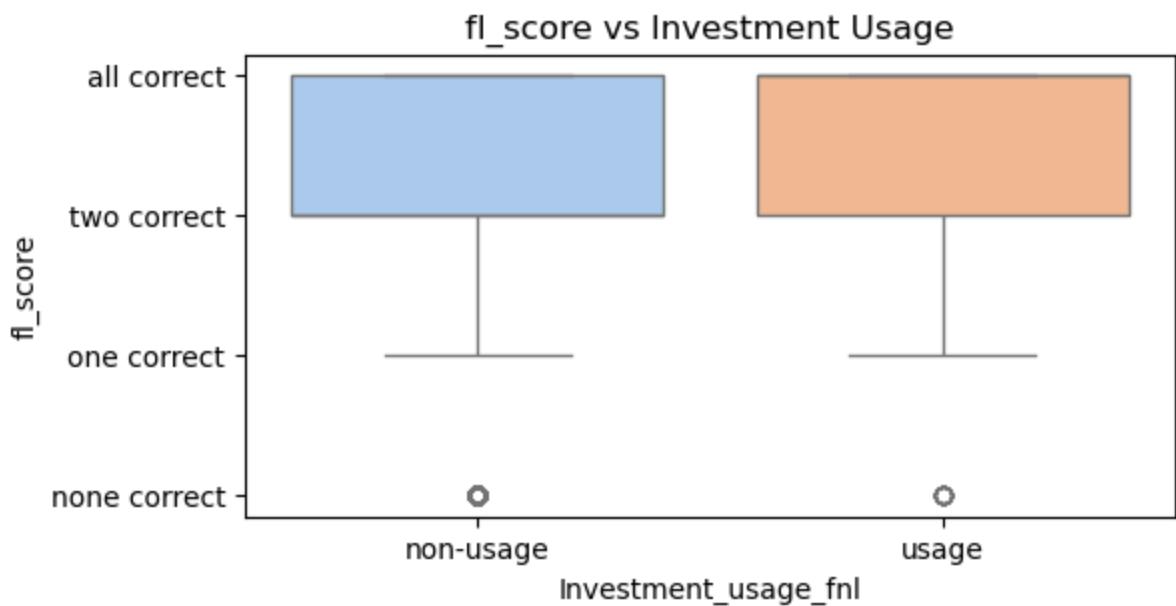
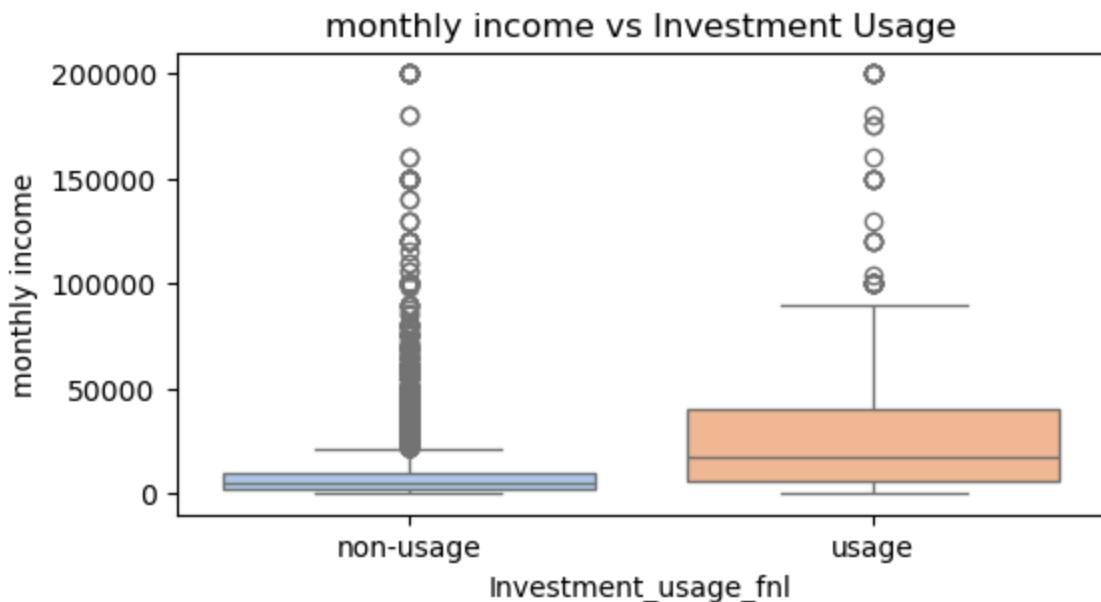
We visualize how selected numeric variables differ between respondents who use and don't use investment products:

- monthly_income
- fl_score (financial literacy score)
- age_of_respondent

Using boxplots helps us see:

- Typical range and median for each group
- Outliers and skewness
- Whether product users tend to have higher income, financial literacy, or different age distribution

```
In [153... # Plot boxplots of each numeric column vs. Investment usage
for col in ['monthly_income', 'fl_score', 'age_of_respondent']:
    plt.figure(figsize=(6,3))
    sns.boxplot(x='Investment_usage_fnl', y=col, data=df, palette='pastel')
    plt.title(f'{col} vs Investment Usage')
    plt.show()
```



Convert Usage Flags to Numeric

We identify all columns ending with `_use` that track product usage as `'never used'` or `'used'`.

We map them to numeric:

- `'never used' → 0`
- `'used' → 1`

This prepares the data for modeling, as most ML algorithms require numeric inputs.

```
In [154...]: # Identify all binary usage flag columns
use_flags = [col for col in df.columns if col.endswith('_use')]

# Map categorical values to numeric: 'never used' -> 0, 'used' -> 1
df[use_flags] = df[use_flags].apply(lambda col: col.map({'never used': 0, 'used': 1}))
```

Convert Disability Status to Numeric

The column `PWD` indicates whether a respondent has a disability. We map:

- `'without disability' → 0`
- `'with disability' → 1`

This prepares the feature for machine learning models.

```
In [155...]: # Map disability status to numeric: without disability → 0, with disability → 1
df['PWD'] = df['PWD'].map({'without disability': 0, 'with disability': 1})
```

Create Age Groups

We create a categorical feature `age_group` by binning `age_of_respondent` into ranges:

- 18–25
- 26–35
- 36–50
- 51–65
- 66+

This makes it easier to analyze and visualize age patterns and can also be useful for modeling categorical age effects.

```
In [156...]: # Create age group categories based on age_of_respondent
df['age_group'] = pd.cut(
    df['age_of_respondent'],
    bins=[17, 25, 35, 50, 65, 100],
```

```
    labels=['18-25', '26-35', '36-50', '51-65', '66+']
)
```

Create Household Size Categories

We group `no_of_household_mebers` into categorical bins:

- 1–2 members
- 3–4 members
- 5–6 members
- 7–10 members
- 11+ members

This helps explore whether larger households differ in investment behavior and simplifies analysis.

```
In [157...]: # Create household size category based on number of household members
df['hh_size_cat'] = pd.cut(
    df['no_of_household_mebers'],
    bins=[0, 2, 4, 6, 10, 20],
    labels=['1-2', '3-4', '5-6', '7-10', '11+']
)
```

Create Urban/Rural Binary Feature

We transform the `area_type` column into a numeric feature:

- `urban` → 1
- `rural` → 0

This makes it easier for machine learning models and helps us analyze investment patterns by urban vs. rural respondents.

```
In [158...]: # Map area type to binary: urban → 1, rural → 0
df['is_urban'] = df['area_type'].map({'urban': 1, 'rural': 0})
```

Create Savings Ratio

We engineer a new feature `savings_ratio`:

- Calculated as: `total_monthly_expenditure / (monthly income + 1)`
- Adding `+1` avoids division by zero.

This feature helps capture how much of a respondent's income is spent, which may relate to their likelihood to invest.

```
In [159...]: # Create savings ratio: expenditure divided by (monthly income + 1 to avoid division by zero)
if 'monthly income' in df.columns and 'total_monthly_expenditure' in df.columns:
```

```
df['savings_ratio'] = df['total_monthly_expenditure'] / (df['monthly_income'] +
```

Encode Categorical Variables

We one-hot encode the following categorical features:

- gender, age_group, county, Education, Marital, livelihoodcat, Quintiles

Using `drop_first=True` drops the first category in each to avoid multicollinearity. This prepares data for machine learning models that require numeric inputs.

In [160...]

```
# Define categorical columns to encode
categorical_cols = ['gender', 'age_group', 'county', 'Education', 'Marital', 'livelihoodcat', 'Quintiles']

# One-hot encode categorical variables, dropping first category to avoid multicollinearity
df_encoded = pd.get_dummies(df, columns=categorical_cols, drop_first=True)
```

Check Encoded Feature Names

After one-hot encoding, the dataframe can have many new columns (especially from high-cardinality features like `county`).

We print:

- Total number of columns
- Sample of first and last few column names

This helps verify that encoding worked correctly and spot any unexpected features.

In [161...]

```
print(df_encoded.columns)

Index(['householdid', 'area_type', 'age_of_respondent',
       'no_of_household_mebers', 'CalcExpenditure',
       'total_monthly_expenditure', 'digital_loans', 'no_respondent_per_hh',
       'ID_card', 'hhWeight',
       ...
       'Marital_widowed', 'livelihoodcat_casual_worker',
       'livelihoodcat_dependent', 'livelihoodcat_employed',
       'livelihoodcat_other', 'livelihoodcat_own_business',
       'Quintiles_highest', 'Quintiles_lowest', 'Quintiles_middle',
       'Quintiles_second'],
      dtype='object', length=456)
```

Standardize Numeric Features

We scale selected numeric columns (`age_of_respondent` and `no_of_household_mebers`) using StandardScaler:

- Transforms each feature to mean = 0 and standard deviation = 1

- Helps many machine learning algorithms converge faster and avoid bias toward larger-scale features.

```
In [162...]: # List of numeric columns to scale
numeric_cols = ['age_of_respondent', 'no_of_household_mebers']

# Initialize and apply scaler
scaler = StandardScaler()
df_encoded[numeric_cols] = scaler.fit_transform(df_encoded[numeric_cols])
```

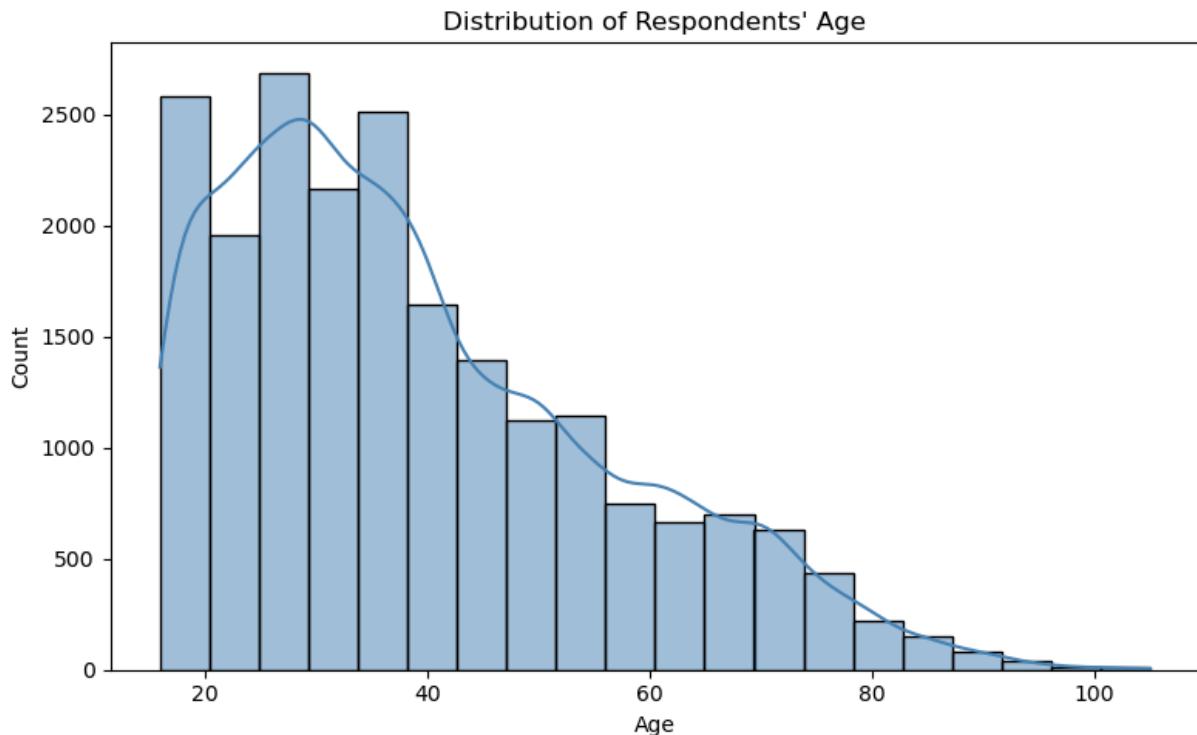
Age Distribution of Respondents

We visualize the distribution of respondents' ages:

- Histogram shows how many respondents fall into each age range
- KDE curve helps see the overall shape (e.g., skewness, peaks)

This helps us understand the age profile of our dataset.

```
In [163...]: # Plot distribution of age_of_respondent
plt.figure(figsize=(8, 5))
sns.histplot(df['age_of_respondent'], bins=20, kde=True, color='steelblue')
plt.title("Distribution of Respondents' Age")
plt.xlabel('Age')
plt.ylabel('Count')
plt.tight_layout()
plt.show()
```

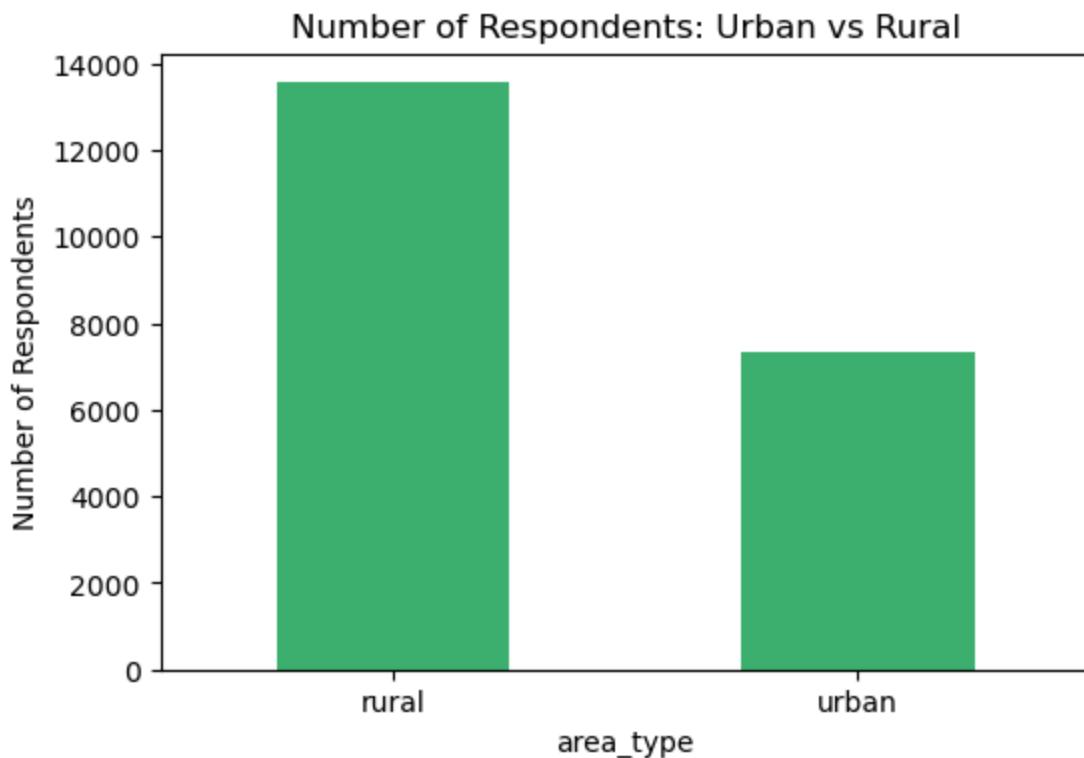


Number of Respondents by Area Type

We visualize how many respondents come from urban vs. rural areas. This helps us see if the sample is balanced or biased toward one area type.

In [164...]

```
# Plot number of respondents by area type
plt.figure(figsize=(6,4))
df['area_type'].value_counts().plot(kind='bar', color='mediumseagreen')
plt.title('Number of Respondents: Urban vs Rural')
plt.ylabel('Number of Respondents')
plt.xticks(rotation=0)
plt.show()
```



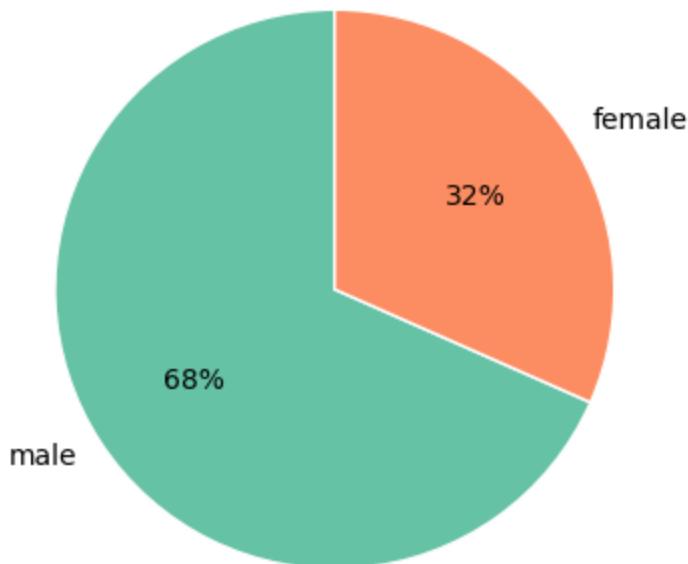
Gender Distribution

We visualize the proportion of male vs. female respondents as a pie chart. This helps understand the gender balance in our dataset.

In [165...]

```
# Plot gender distribution as a pie chart
plt.figure(figsize=(6, 4))
df['gender'].value_counts().plot.pie(
    autopct='%1.0f%%',
    colors=['#66c2a5', '#fc8d62'],
    startangle=90,
    wedgeprops={'edgecolor': 'white'}
)
plt.title('Gender Distribution')
plt.ylabel('')
plt.tight_layout()
plt.show()
```

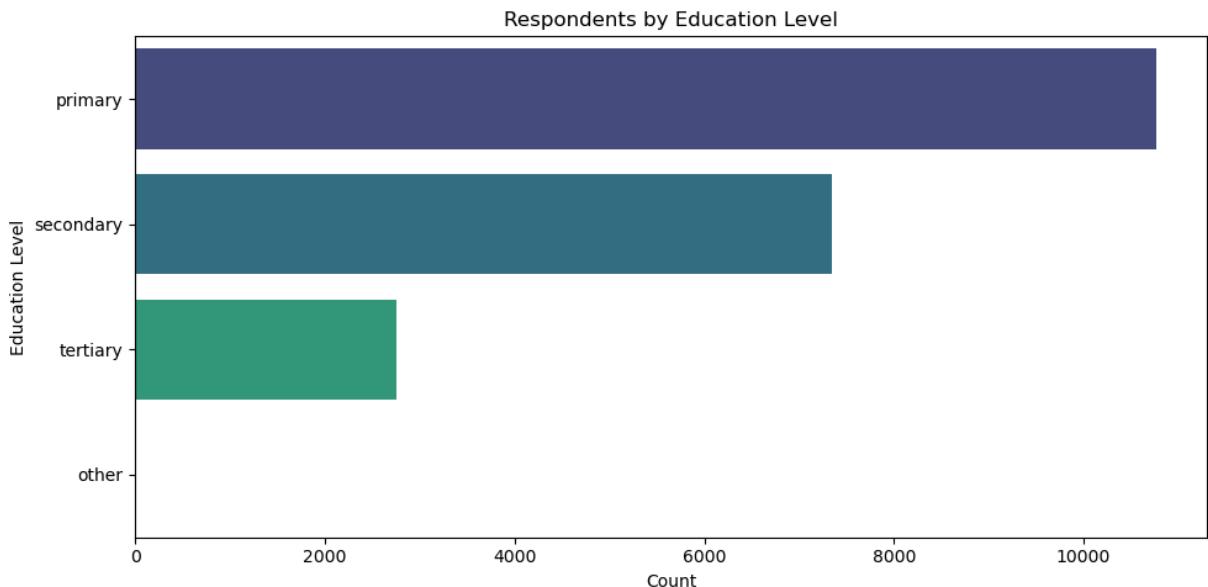
Gender Distribution



Respondents by Education Level

We visualize how many respondents fall into each education category. Ordering bars by frequency makes it easy to see which education levels are most common in the dataset.

```
In [166...]: # Plot count of respondents by education level
plt.figure(figsize=(10, 5))
sns.countplot(
    data=df,
    y='Education',
    order=df['Education'].value_counts().index,
    palette='viridis'
)
plt.title('Respondents by Education Level')
plt.xlabel('Count')
plt.ylabel('Education Level')
plt.tight_layout()
plt.show()
```

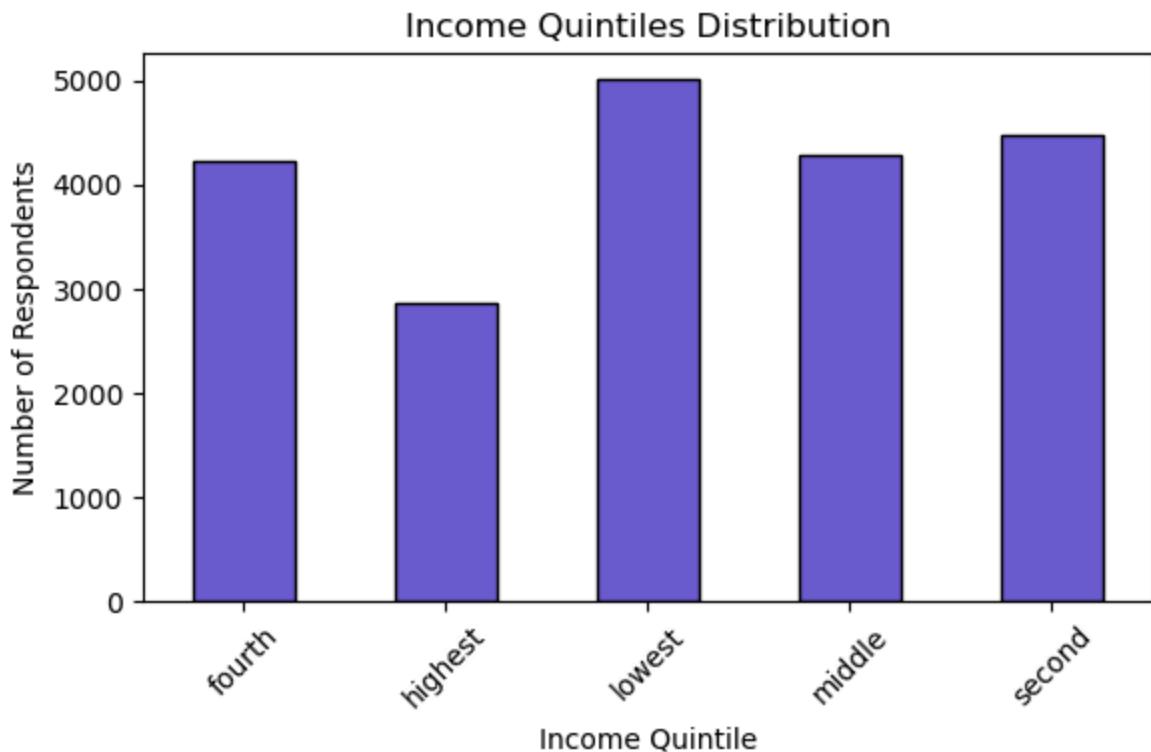


Income Quintiles Distribution

We visualize how respondents are distributed across income quintiles (from lowest to highest). This helps us understand the income profile of the sample and spot if it's balanced or skewed.

In [167...]

```
# Plot distribution of respondents across income quintiles
plt.figure(figsize=(6, 4))
df['Quintiles'].value_counts().sort_index().plot(
    kind='bar',
    color='slateblue',
    edgecolor='black'
)
plt.title('Income Quintiles Distribution')
plt.ylabel('Number of Respondents')
plt.xlabel('Income Quintile')
plt.xticks(rotation=45)
plt.tight_layout()
plt.show()
```



Feature Engineering

We create new features to enrich our dataset:

- `risk_adjusted_savings_rate` : normalizes savings by age
- `age_bucket` : categorical grouping of age
- `is_youth` : binary flag for respondents under 35
- `total_investment_products_used` : counts how many investment products each respondent uses

These engineered features can help models capture hidden patterns in user behavior.

```
In [168...]
# Risk-adjusted savings rate: people who have savings divided by (age+1)
df['risk_adjusted_savings_rate'] = df['has_savings'] / (df['age_of_respondent'] + 1)

# Age bucket: categorical age group
df['age_bucket'] = pd.cut(
    df['age_of_respondent'],
    bins=[0, 25, 35, 45, 60, 100],
    labels=['<25', '25-35', '35-45', '45-60', '60+']
)

# Binary youth flag: 1 if age < 35
df['is_youth'] = (df['age_of_respondent'] < 35).astype(int)

# Total number of investment products used: counts how many of these are 'usage'
df['total_investment_products_used'] = df[
    ['Investment_usage', 'Pension_usage', 'Sacco_usage']
].apply(lambda x: sum(x == 'usage'), axis=1)
```

Encode Financial Literacy Index

The `Financial_literacy_index_fnl` column is ordinal: `'low'`, `'medium'`, `'high'`. We map it to numeric values:

- low → 1
- medium → 2
- high → 3

This preserves the order for machine learning algorithms.

```
In [169...]
# Map ordinal financial literacy index to numeric values: Low=1, medium=2, high=3
df['Financial_literacy_index_fnl_num'] = df['Financial_literacy_index_fnl'].map({
    'low': 1,
    'medium': 2,
    'high': 3
})
```

Scale Financial Literacy Index

We normalize `Financial_literacy_index_fnl_num` to a [0,1] range:

- Divide by its maximum value

This keeps the feature bounded and comparable with other scaled features.

```
In [170...]
# Scale financial literacy index to [0,1] by dividing by max value
max_lit = df['Financial_literacy_index_fnl_num'].max()
df['literacy_scaled'] = df['Financial_literacy_index_fnl_num'] / max_lit
```

Check for Constant Numeric Columns

We identify numeric features with zero standard deviation (constant columns). These columns have the same value for all observations and don't help the model.

```
In [171...]
# Select only numeric columns
numeric_engineered = df.select_dtypes(include=['number'])

# Identify constant columns (std=0)
constant_cols = numeric_engineered.std() == 0

print("\n== Constant columns (std=0) ==")
print(constant_cols[constant_cols].index.tolist())

== Constant columns (std=0) ==
['Above16', 'mobile_bank_use', 'NHIF_use', 'NSSF_use', 'pension_use', 'bank_use', 'commercial_bank_use', 'mobile_money_use', 'sacco_use', 'chama_use', 'MFIs_use', 'insurance_including_NHIF_use', 'All_Insurance_excluding_NHIF_use', 'has_account', 'has_savings', 'has_credit', 'has_mobile', 'receives_remittance', 'risk_adjusted_savings_rate']
```

```
In [172...]
# Sanity check for engineered features

# List of engineered features (update as you add more)
engineered_features = [
    'risk_adjusted_savings_rate',
    'age_bucket',
    'is_youth',
    'literacy_scaled',
    'total_investment_products_used',
    # add more as you create them
]

print(f"\n Checking {len(engineered_features)} engineered features:\n")

# Check if all engineered features exist
missing_cols = [col for col in engineered_features if col not in df.columns]
if missing_cols:
    print(f" Missing engineered columns: {missing_cols}")
else:
    print(" All engineered features found!")

# Show data types
print("\n== Data types ==")
print(df[engineered_features].dtypes)

# Separate numeric & categorical
numeric_engineered = df[engineered_features].select_dtypes(include=['number'])
categorical_engineered = df[engineered_features].select_dtypes(exclude=['number'])

# Check for missing values
print("\n== Missing values ==")
print(df[engineered_features].isnull().sum())

# Check for constant numeric columns (std=0)
print("\n== Constant numeric columns (std=0) ==")
constant_cols = numeric_engineered.std() == 0
```

```

print(constant_cols[constant_cols].index.tolist())

# Check for all-zeros numeric columns
print("\n==== All-zeros numeric columns ===")
zero_cols = (numeric_engineered == 0).all()
print(zero_cols[zero_cols].index.tolist())

# Show numeric distributions
print("\n==== Numeric distributions ===")
display(numeric_engineered.describe())

# Show value counts for categorical engineered features
if not categorical_engineered.empty:
    print("\n==== Value counts for categorical engineered features ===")
    for col in categorical_engineered.columns:
        print(f"\n{col} value counts:")
        print(df[col].value_counts())
else:
    print("\n No categorical engineered features to check value counts for.")

print("\n Feature engineering sanity check complete! Ready for vectorization ")

```

Checking 5 engineered features:

All engineered features found!

```

==== ☐ Data types ===
risk_adjusted_savings_rate      float64
age_bucket                      category
is_youth                         int32
literacy_scaled                  float64
total_investment_products_used   int64
dtype: object

==== Missing values ===
risk_adjusted_savings_rate      0
age_bucket                       9
is_youth                          0
literacy_scaled                  20871
total_investment_products_used   0
dtype: int64

==== Constant numeric columns (std=0) ===
['risk_adjusted_savings_rate']

==== All-zeros numeric columns ===
['risk_adjusted_savings_rate']

==== Numeric distributions ===

```

	risk_adjusted_savings_rate	is_youth	literacy_scaled	total_investment_products_used
count	20871.0	20871.000000	0.0	20871.000000
mean	0.0	0.472186	NaN	0.23113
std	0.0	0.499238	NaN	0.57135
min	0.0	0.000000	NaN	0.000000
25%	0.0	0.000000	NaN	0.000000
50%	0.0	0.000000	NaN	0.000000
75%	0.0	1.000000	NaN	0.000000
max	0.0	1.000000	NaN	3.000000

==== Value counts for categorical engineered features ====

age_bucket value counts:

age_bucket

25-35	5361
<25	5118
35-45	3942
45-60	3521
60+	2920

Name: count, dtype: int64

Feature engineering sanity check complete! Ready for vectorization

Feature Vectorization Pipeline

In this step, we prepare our dataset for modeling by transforming raw features into numeric vectors:

- **Numeric features** are standardized to have mean=0 and std=1.
- **Categorical features** are one-hot encoded (converted to 0/1 dummy variables).
- **Binary flags** like `is_youth` are kept as-is.

We use `ColumnTransformer` to apply different transformations to different columns in a single pipeline. At the end, we run a quick `fit_transform` to confirm everything works and print the shape of the transformed feature matrix.

In [173...]

```
from sklearn.compose import ColumnTransformer
from sklearn.preprocessing import OneHotEncoder, StandardScaler
# === Define feature groups ===

# Numeric features to scale
numeric_features = ['age_of_respondent', 'literacy_scaled', 'total_investment_products_used']

# Categorical features to one-hot encode
categorical_features = ['age_bucket', 'area_type', 'county', 'Education', 'Marital_Status']
```

```
# Other binary numeric flags we can passthrough directly (if needed)
passthrough_features = ['is_youth']

# === Build the ColumnTransformer ===

preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), numeric_features),
        ('cat', OneHotEncoder(handle_unknown='ignore'), categorical_features),
        ('pass', 'passthrough', passthrough_features)
    ]
)

# === Quick check: fit_transform on sample data to see shape ===

sample_transformed = preprocessor.fit_transform(df)
print(f"Vectorization complete! Transformed feature shape: {sample_transformed.shape}")
```

Vectorization complete! Transformed feature shape: (20871, 74)

```
In [174...]: print(f"✓ df shape: {df.shape}")
print("\n📌 df.head():")
display(df.head())
```

✓ df shape: (20871, 402)

📌 df.head():

	householdid	county	area_type	gender	age_of_respondent	no_of_household_mebers	Investment_usage_fn1
0	107141431	garissa	urban	male	29	5	1
1	10712933	garissa	urban	male	60	11	1
2	140173183	busia	urban	female	35	2	0
3	122137153	kiambu	urban	male	24	1	0
4	121193116	murang'a	urban	female	20	1	0

5 rows × 402 columns



Train-Test Split

We now prepare our dataset for modeling by:

- Converting the target column `Investment_usage_fn1` into numeric labels (1 for usage, 0 for non-usage).
- Dropping any rows with missing target values.

- Splitting the data into training and test sets (80% train, 20% test) while preserving class distribution (`stratify=y`).

This ensures our model is trained and evaluated on balanced and properly prepared data.

In [175...]

```
from sklearn.model_selection import train_test_split

# Define your target column
target_column = 'Investment_usage_fnl' # replace with actual target column name i

# Map target to numeric if it's still text (e.g., 'usage'/'non-usage')
df[target_column] = df[target_column].map({'usage': 1, 'non-usage': 0})

# Define features (X) and target (y)
X = df.drop(columns=[target_column])
y = df[target_column]

# Split into training and testing sets (80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,           # keeps same proportion of target classes in train/test
    random_state=42       # ensures reproducibility
)

# Check the shape
print("Train shape:", X_train.shape)
print("Test shape:", X_test.shape)
print("Train target balance:\n", y_train.value_counts(normalize=True))
print("Test target balance:\n", y_test.value_counts(normalize=True))
```

```
Train shape: (16696, 401)
Test shape: (4175, 401)
Train target balance:
  Investment_usage_fnl
  0      0.972868
  1      0.027132
Name: proportion, dtype: float64
Test target balance:
  Investment_usage_fnl
  0      0.972934
  1      0.027066
Name: proportion, dtype: float64
```

The split completed successfully:

- **Train shape:** (16,696 rows × 402 columns)
- **Test shape:** (4,175 rows × 402 columns)

Handling Class Imbalance with SMOTE

Given the strong imbalance in the target variable (only ~3% "usage"), we apply SMOTE to synthetically oversample the minority class in the training set.

This helps improve model performance by providing a more balanced dataset for learning.

```
In [176...]:  
from imblearn.over_sampling import SMOTE  
from collections import Counter  
  
def balance_with_smote(X_train, y_train, random_state=42):  
    """  
        Applies SMOTE to balance the training set.  
        Returns balanced X_train and y_train.  
    """  
    print("Original y_train distribution:", Counter(y_train))  
    smote = SMOTE(random_state=random_state)  
    X_train_bal, y_train_bal = smote.fit_resample(X_train, y_train)  
    print("Balanced y_train distribution:", Counter(y_train_bal))  
    return X_train_bal, y_train_bal
```

MODELING

Modular Preprocessing Pipeline

We define a `PreprocessingPipeline` class to:

- Automatically detect numeric and categorical features
- Impute missing values (mean for numeric, most frequent for categorical)
- Scale numeric features and one-hot encode categorical features
- Split the data into train/test sets and transform them

This modular design keeps preprocessing clean, reusable, and ready for modeling.

```
In [177...]:  
from sklearn.pipeline import Pipeline  
from sklearn.compose import ColumnTransformer  
from sklearn.impute import SimpleImputer  
from sklearn.preprocessing import StandardScaler, OneHotEncoder  
from sklearn.model_selection import train_test_split  
  
class PreprocessingPipeline:  
    """  
        Modular preprocessing pipeline: imputation, scaling, encoding.  
    """  
    def __init__(self, df, target_column):  
        self.df = df  
        self.target_column = target_column  
        self.preprocessor = None  
        self.numeric_features = None  
        self.categorical_features = None
```

```

def build_pipeline(self):
    # Identify feature types
    self.numeric_features = self.df.select_dtypes(include=['int64', 'float64'])
    self.categorical_features = self.df.select_dtypes(include=['object', 'category'])
    if self.target_column in self.numeric_features:
        self.numeric_features.remove(self.target_column)
    if self.target_column in self.categorical_features:
        self.categorical_features.remove(self.target_column)

    # Pipelines
    numeric_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy='mean')),
        ('scaler', StandardScaler())
    ])
    categorical_pipeline = Pipeline([
        ('imputer', SimpleImputer(strategy='most_frequent')),
        ('encoder', OneHotEncoder(handle_unknown='ignore'))
    ])

    # Combine
    self.preprocessor = ColumnTransformer([
        ('num', numeric_pipeline, self.numeric_features),
        ('cat', categorical_pipeline, self.categorical_features)
    ])

def apply(self, test_size=0.2, random_state=42):
    X = self.df.drop(columns=[self.target_column])
    y = self.df[self.target_column]
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=test_size, random_state=random_state, stratify=y)
    X_train_transformed = self.preprocessor.fit_transform(X_train)
    X_test_transformed = self.preprocessor.transform(X_test)
    return X_train_transformed, X_test_transformed, y_train, y_test

```

Balancing Classes with SMOTE

We define a helper function `apply_smote` to rebalance the training data using SMOTE (Synthetic Minority Over-sampling Technique).

In [178...]

```

def apply_smote(X_train, y_train, random_state=42):
    """
    Apply SMOTE to balance minority class.
    """
    print(" Applying SMOTE...")
    smote = SMOTE(random_state=random_state)
    X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
    print(f"Before SMOTE: {X_train.shape}, After SMOTE: {X_resampled.shape}")
    print(f"Target distribution after SMOTE:\n{pd.Series(y_resampled).value_counts()}")
    return X_resampled, y_resampled

```

SMOTE creates synthetic samples of the minority class, helping to:

- Balance the target distribution

- Reduce model bias toward the majority class
- Improve predictive performance on rare classes

We confirm success by printing shapes and class counts before and after resampling.

Model Training & Evaluation function

We define `train_and_evaluate_model` to train a classifier, make predictions, and evaluate performance with metrics & visualizations.

```
In [179...]: def train_and_evaluate_model(model, X_train, y_train, X_test, y_test, model_name="Model Name"):
    """
    Train model, predict, evaluate & plot.
    """

    print(f"\n ↗️ Training {model_name}...")
    model.fit(X_train, y_train)
    y_pred = model.predict(X_test)
    y_prob = model.predict_proba(X_test)[:,1] if hasattr(model, "predict_proba") else None

    print("\n Classification Report:")
    print(classification_report(y_test, y_pred))

    if y_prob is not None:
        roc_auc = roc_auc_score(y_test, y_prob)
        print(f"ROC AUC Score: {roc_auc:.4f}")
    else:
        roc_auc = None

    # Confusion matrix
    plt.figure(figsize=(5,4))
    sns.heatmap(confusion_matrix(y_test, y_pred), annot=True, fmt='d', cmap='Blues')
    plt.title(f'{model_name} - Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('Actual')
    plt.show()

    # ROC curve
    if y_prob is not None:
        fpr, tpr, _ = roc_curve(y_test, y_prob)
        plt.figure(figsize=(6,4))
        plt.plot(fpr, tpr, label=f'ROC Curve (AUC={roc_auc:.2f})')
        plt.plot([0,1],[0,1], 'k--')
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title(f'{model_name} - ROC Curve')
        plt.legend()
        plt.show()

    return model, roc_auc
```

This function helps us:

- Fit the model on the training set

- Evaluate it on the test set using:
 - Classification report (precision, recall, f1-score)
 - ROC AUC score
- Visualize:
 - Confusion matrix (heatmap)
 - ROC curve

It returns the trained model and ROC AUC score, making it easy to reuse or compare multiple models.

ModelTrainer Class

An object-oriented wrapper to train, tune and store models neatly.

In [180...]

```
class ModelTrainer:
    """
    Object-oriented wrapper to train, tune & store model.
    """

    def __init__(self, model, model_name="Model"):
        self.model = model
        self.model_name = model_name
        self.trained_model = None
        self.roc_auc = None

    def train_and_evaluate(self, X_train, y_train, X_test, y_test):
        trained, roc_auc = train_and_evaluate_model(
            self.model, X_train, y_train, X_test, y_test, self.model_name
        )
        self.trained_model = trained
        self.roc_auc = roc_auc

    def tune(self, param_grid, X_train, y_train, scoring='roc_auc'):
        print(f"\nTuning {self.model_name}...")
        grid = GridSearchCV(self.model, param_grid, scoring=scoring, cv=5, verbose=0)
        grid.fit(X_train, y_train)
        print(f" Best params: {grid.best_params_}")
        self.model = grid.best_estimator_
        self.trained_model = self.model
```

This `ModelTrainer` class helps us:

- **Encapsulate** a machine learning model and its metadata (like ROC AUC)
- Easily **train & evaluate** using our helper function
- **Tune hyperparameters** with `GridSearchCV` and update the best model

It keeps the workflow organized and reusable, especially when comparing multiple models.

Random Forest Pipeline with SMOTE and Preprocessing

```
In [181...]  
# df is your DataFrame  
target_column = 'Investment_usage_fnl'  
  
# Build preprocessing pipeline  
prep = PreprocessingPipeline(df, target_column)  
prep.build_pipeline()  
  
# Apply preprocessing (split & transform)  
X_train_transformed, X_test_transformed, y_train, y_test = prep.apply()  
  
# Apply SMOTE to handle class imbalance  
X_train_resampled, y_train_resampled = apply_smote(X_train_transformed, y_train)  
  
# Create Random Forest with balanced class weights  
rf_model = RandomForestClassifier(random_state=42, class_weight='balanced')  
  
# Train and evaluate the model  
rf_trainer = ModelTrainer(rf_model, model_name="Random Forest with SMOTE")  
rf_trainer.train_and_evaluate(X_train_resampled, y_train_resampled, X_test_transformed)
```

Applying SMOTE...

Before SMOTE: (16696, 954), After SMOTE: (32486, 954)

Target distribution after SMOTE:

Investment_usage_fnl

0 16243

1 16243

Name: count, dtype: int64

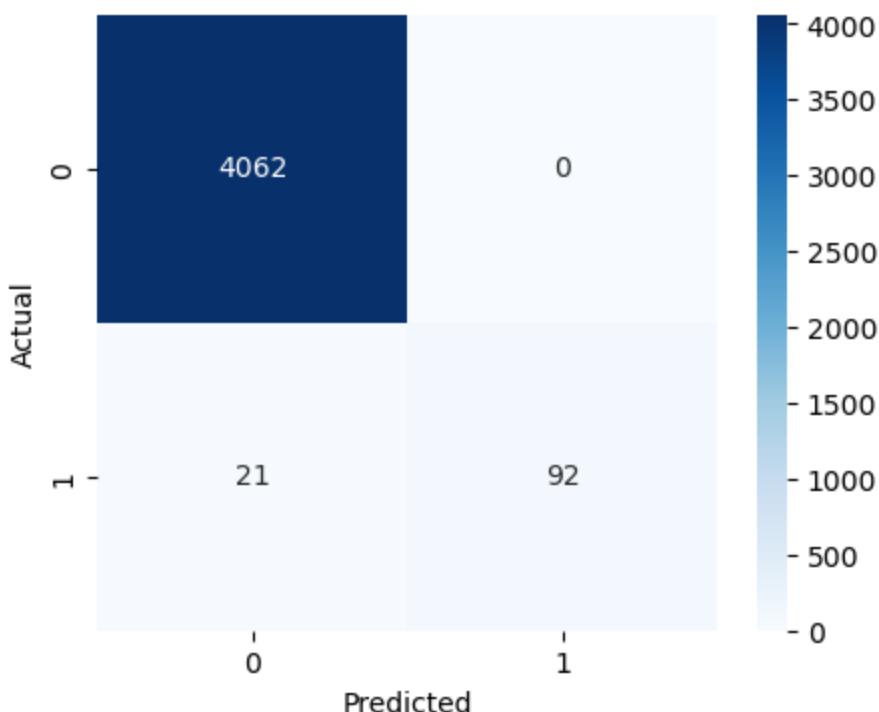
⌚ Training Random Forest with SMOTE...

Classification Report:

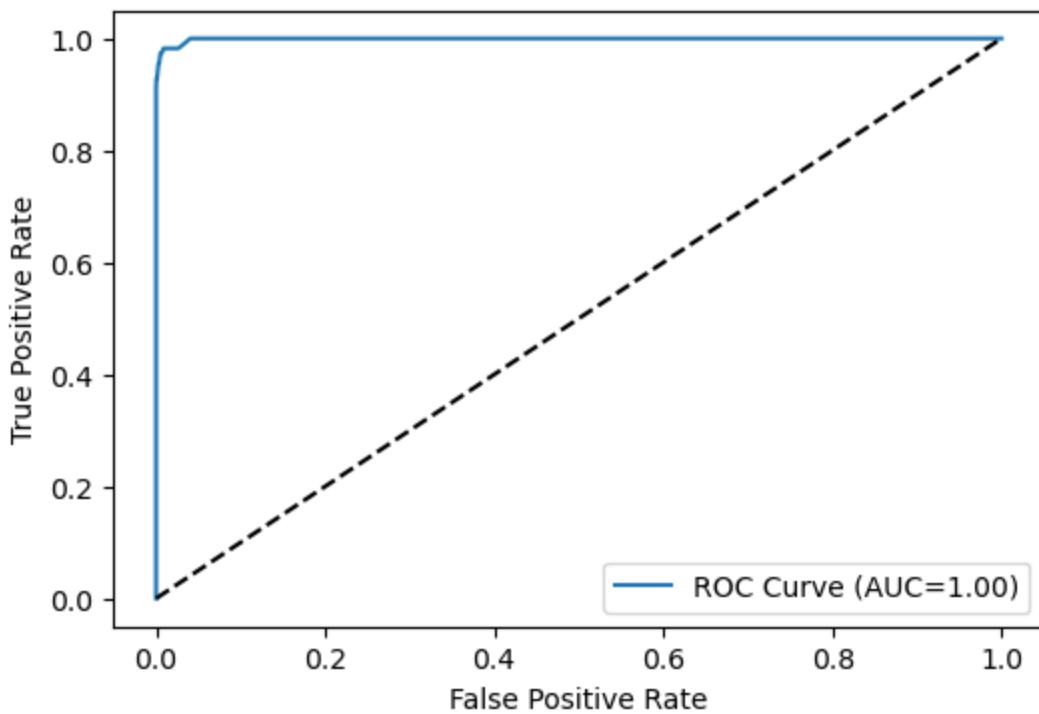
	precision	recall	f1-score	support
0	0.99	1.00	1.00	4062
1	1.00	0.81	0.90	113
accuracy			0.99	4175
macro avg	1.00	0.91	0.95	4175
weighted avg	0.99	0.99	0.99	4175

ROC AUC Score: 0.9992

Random Forest with SMOTE - Confusion Matrix



Random Forest with SMOTE - ROC Curve



By applying **SMOTE**, we balanced the classes in the training data:

- Before: only ~2.7% positive class
- After: perfectly balanced (50% vs 50%)

The Random Forest model achieved:

- Very high overall accuracy (~99%)

- **ROC AUC score of 1.00**, showing excellent discrimination
- Slight drop in recall (0.81) for the minority class

Logistic Regression with SMOTE

Now we apply SMOTE to balance the training data and train a **Logistic Regression** model with `class_weight='balanced'`.

This helps the model pay more attention to the minority class.

We'll then evaluate performance on the test set to see how well it generalizes.

In [182...]

```
# df with your dataframe name
target_column = 'Investment_usage_fnl'
prep = PreprocessingPipeline(df, target_column)
prep.build_pipeline()

# Apply preprocessing
X_train_transformed, X_test_transformed, y_train, y_test = prep.apply()

# Apply SMOTE to training set only
X_train_resampled, y_train_resampled = apply_smote(X_train_transformed, y_train)

# Create Logistic Regression with balanced class weight
logreg_model = LogisticRegression(random_state=42, class_weight='balanced', max_iter=1000)

# Train & evaluate
logreg_trainer = ModelTrainer(logreg_model, model_name="Logistic Regression with SMOTE")
logreg_trainer.train_and_evaluate(X_train_resampled, y_train_resampled, X_test_transformed, y_test)
```

Applying SMOTE...

Before SMOTE: (16696, 954), After SMOTE: (32486, 954)

Target distribution after SMOTE:

```
Investment_usage_fnl
0    16243
1    16243
Name: count, dtype: int64
```

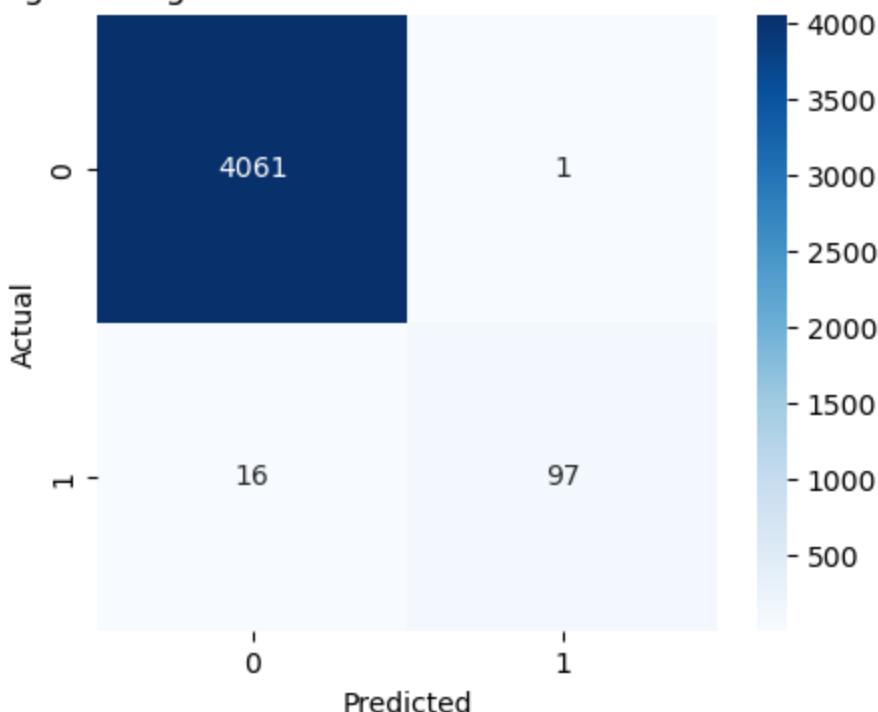
Training Logistic Regression with SMOTE...

Classification Report:

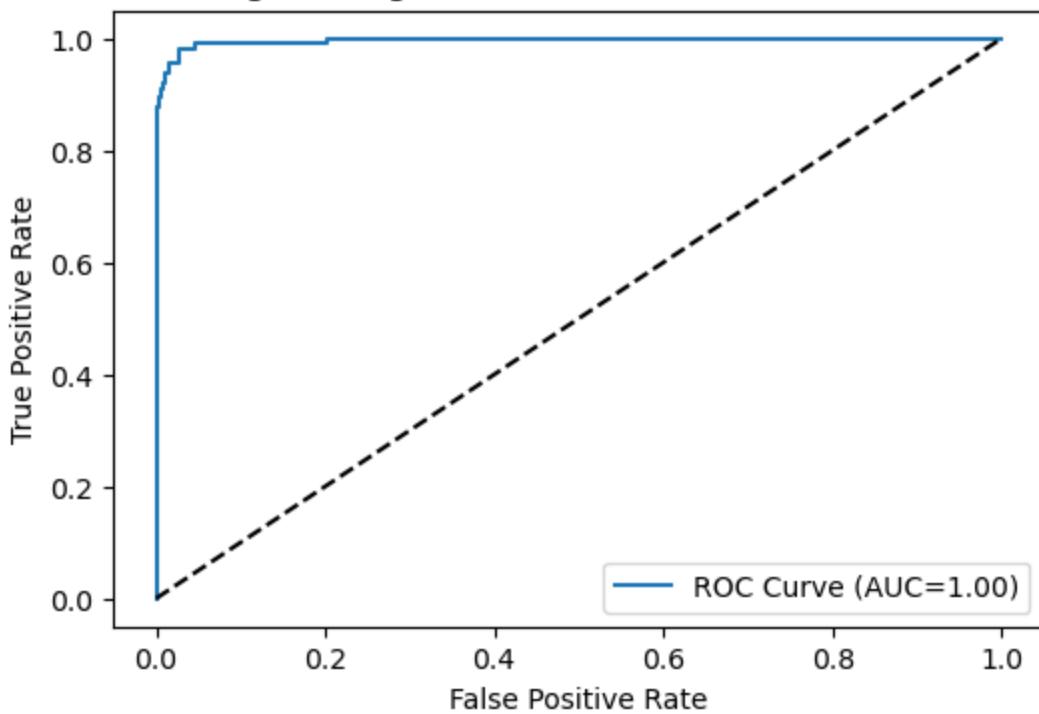
	precision	recall	f1-score	support
0	1.00	1.00	1.00	4062
1	0.99	0.86	0.92	113
accuracy			1.00	4175
macro avg	0.99	0.93	0.96	4175
weighted avg	1.00	1.00	1.00	4175

ROC AUC Score: 0.9964

Logistic Regression with SMOTE - Confusion Matrix



Logistic Regression with SMOTE - ROC Curve



Summary of results

- After applying SMOTE, the training set was perfectly balanced (16,243 samples in each class).
- The model achieved:
 - High overall accuracy (~100%)
 - High recall for the minority class (usage): 0.86

- ROC AUC score: 0.9964

XGBoost with SMOTE and Class Weighting

In this step, we:

- Balanced the training data using **SMOTE**
- Used **XGBoost**, which handles imbalance via `scale_pos_weight`
- Evaluated using precision, recall, f1-score, and ROC AUC

This combination helps the model detect the minority class more effectively.

In [183...]

```
# df with your dataframe name
target_column = 'Investment_usage_fnl'
prep = PreprocessingPipeline(df, target_column)
prep.build_pipeline()

# Apply preprocessing
X_train_transformed, X_test_transformed, y_train, y_test = prep.apply()

# Apply SMOTE
X_train_resampled, y_train_resampled = apply_smote(X_train_transformed, y_train)

# Create XGBoost with scale_pos_weight
import xgboost as xgb
scale_pos_weight = (len(y_train_resampled) - sum(y_train_resampled)) / sum(y_train_resampled)
xgb_model = xgb.XGBClassifier(random_state=42, scale_pos_weight=scale_pos_weight, u

# Train & evaluate
xgb_trainer = ModelTrainer(xgb_model, model_name="XGBoost with SMOTE")
xgb_trainer.train_and_evaluate(X_train_resampled, y_train_resampled, X_test_transfo
```

Applying SMOTE...

Before SMOTE: (16696, 954), After SMOTE: (32486, 954)

Target distribution after SMOTE:

`Investment_usage_fnl`

0 16243

1 16243

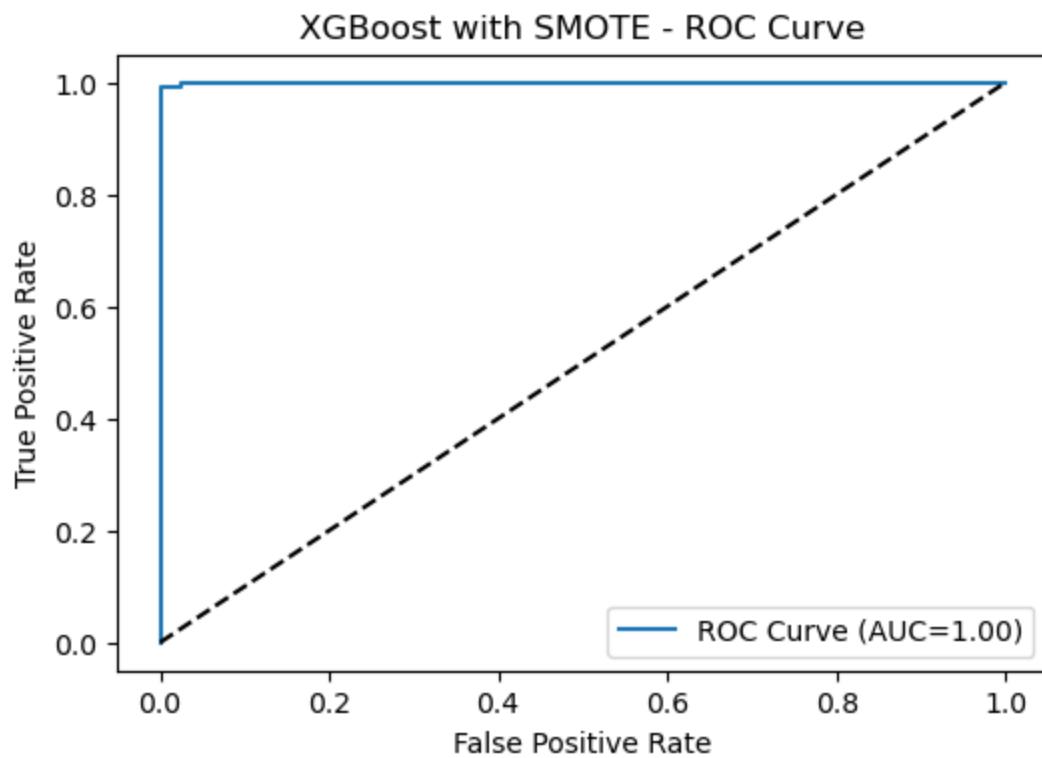
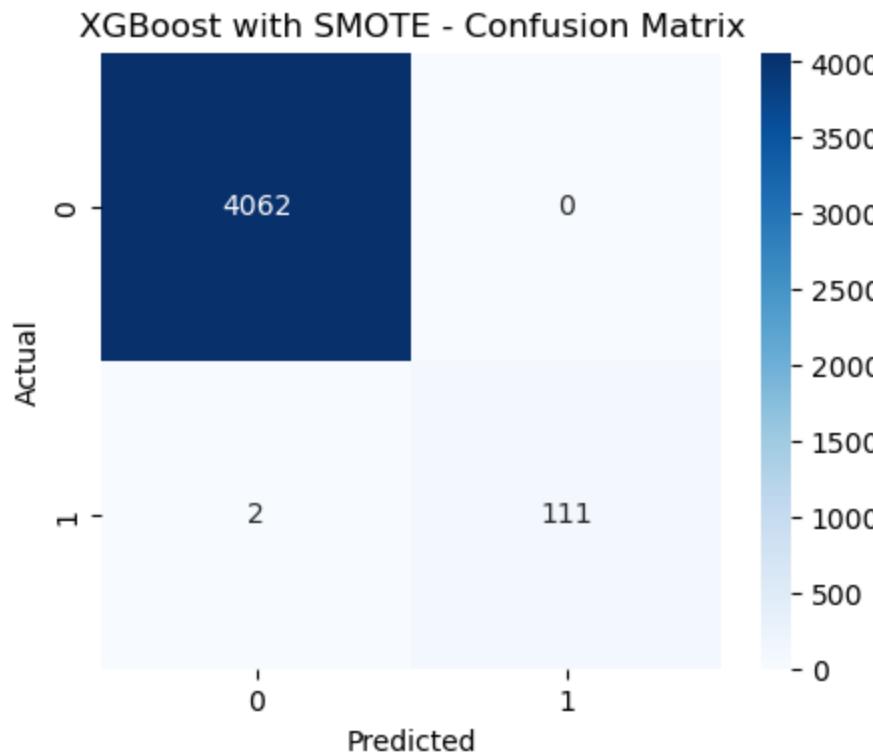
Name: count, dtype: int64

🔧 Training XGBoost with SMOTE...

Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	4062
1	1.00	0.98	0.99	113
accuracy			1.00	4175
macro avg	1.00	0.99	1.00	4175
weighted avg	1.00	1.00	1.00	4175

ROC AUC Score: 0.9998



Summary of results (XGBoost):

- After SMOTE, the training set became perfectly balanced.
- **Performance on test set:**
 - Accuracy $\approx 100\%$
 - Minority class (usage) recall: 0.98
 - ROC AUC score: 0.9998

Key points:

- `scale_pos_weight` helps XGBoost stay aware of class imbalance, even after SMOTE.
- Very high ROC AUC means the model can separate classes very well.
- High recall and f1-score on the minority class suggest the model generalizes better than a naive baseline.

LightGBM with SMOTE and Class Weighting

In this step, we:

- Preprocessed the data (imputation, scaling, encoding)
- Balanced the training set using **SMOTE**
- Trained a **LightGBM classifier** with `class_weight='balanced'` to help handle class imbalance
- Evaluated on the test set using precision, recall, f1-score, and ROC AUC

This approach combines boosting with balancing for better minorit

```
In [184...]:  
# df with your dataframe name  
target_column = 'Investment_usage_fnl'  
prep = PreprocessingPipeline(df, target_column)  
prep.build_pipeline()  
  
# Apply preprocessing  
X_train_transformed, X_test_transformed, y_train, y_test = prep.apply()  
  
# Apply SMOTE  
X_train_resampled, y_train_resampled = apply_smote(X_train_transformed, y_train)  
  
# Create LightGBM with balanced class weight  
import lightgbm as lgb  
lgb_model = lgb.LGBMClassifier(random_state=42, class_weight='balanced')  
  
# Train & evaluate  
lgb_trainer = ModelTrainer(lgb_model, model_name="LightGBM with SMOTE")  
lgb_trainer.train_and_evaluate(X_train_resampled, y_train_resampled, X_test_transfo
```

Applying SMOTE...

Before SMOTE: (16696, 954), After SMOTE: (32486, 954)

Target distribution after SMOTE:

Investment_usage_fnl

0 16243

1 16243

Name: count, dtype: int64

Training LightGBM with SMOTE...

[LightGBM] [Info] Number of positive: 16243, number of negative: 16243

[LightGBM] [Info] Auto-choosing row-wise multi-threading, the overhead of testing was 0.306047 seconds.

You can set `force_row_wise=true` to remove the overhead.

And if memory is not enough, you can set `force_col_wise=true`.

[LightGBM] [Info] Total Bins 182579

[LightGBM] [Info] Number of data points in the train set: 32486, number of used features: 869

[LightGBM] [Info] [binary:BoostFromScore]: pavg=0.500000 -> initscore=0.000000

[LightGBM] [Warning] No further splits with positive gain, best gain: -inf

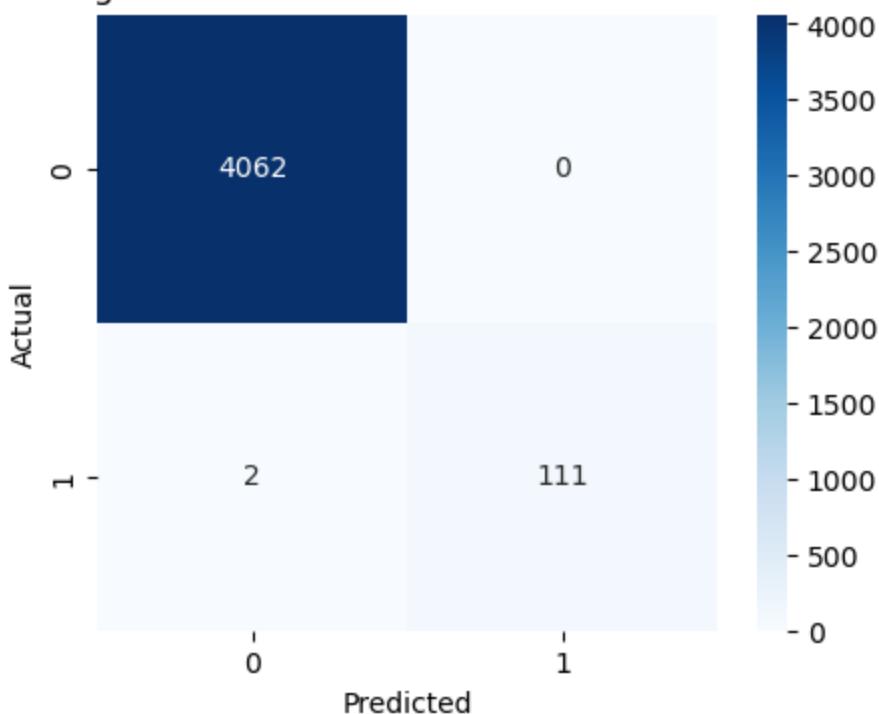
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf

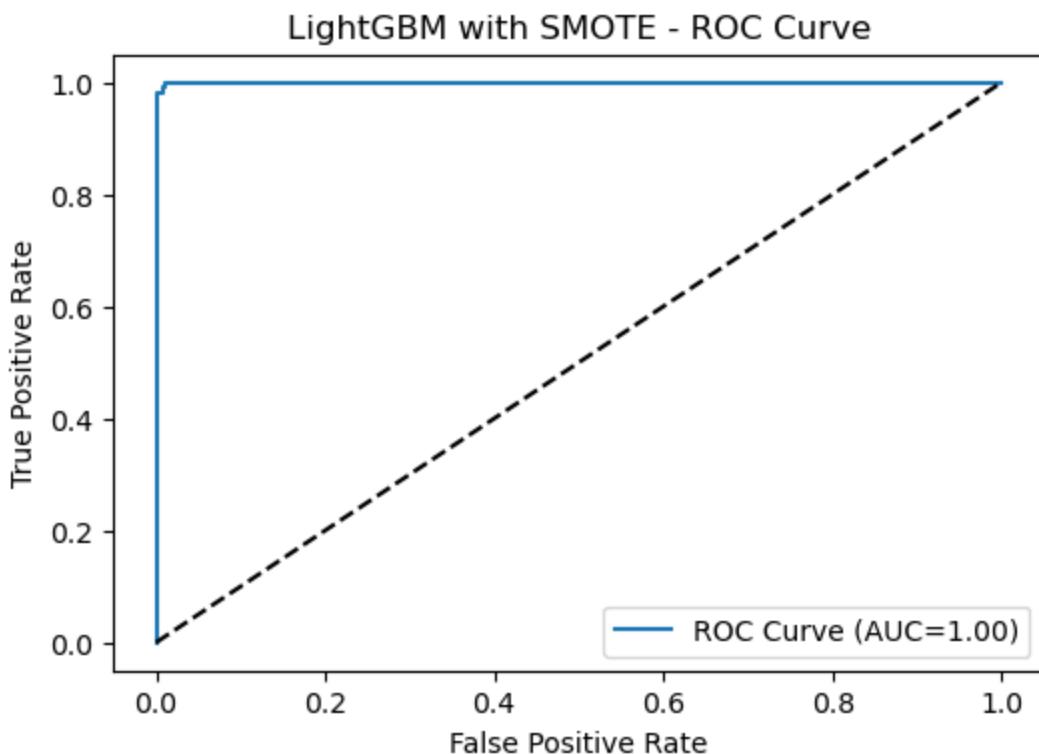
Classification Report:

	precision	recall	f1-score	support
0	1.00	1.00	1.00	4062
1	1.00	0.98	0.99	113
accuracy			1.00	4175
macro avg	1.00	0.99	1.00	4175
weighted avg	1.00	1.00	1.00	4175

ROC AUC Score: 0.9998

LightGBM with SMOTE - Confusion Matrix





Summary of results (LightGBM):

- Training set after SMOTE: balanced (16,243 vs. 16,243)
- **Performance on test set:**
 - Accuracy \approx 100%
 - Minority class recall: 0.98
 - ROC AUC: 0.9998

Insights:

- LightGBM handled the balanced data very well, achieving very high recall and f1-score.
- The model generalizes well and keeps very high ROC AUC, showing it can separate classes almost perfectly.

Comparing Models with ROC Curves

We now plot the **ROC curves** of all trained models on the same test set:

- Random Forest
- Logistic Regression
- XGBoost
- LightGBM

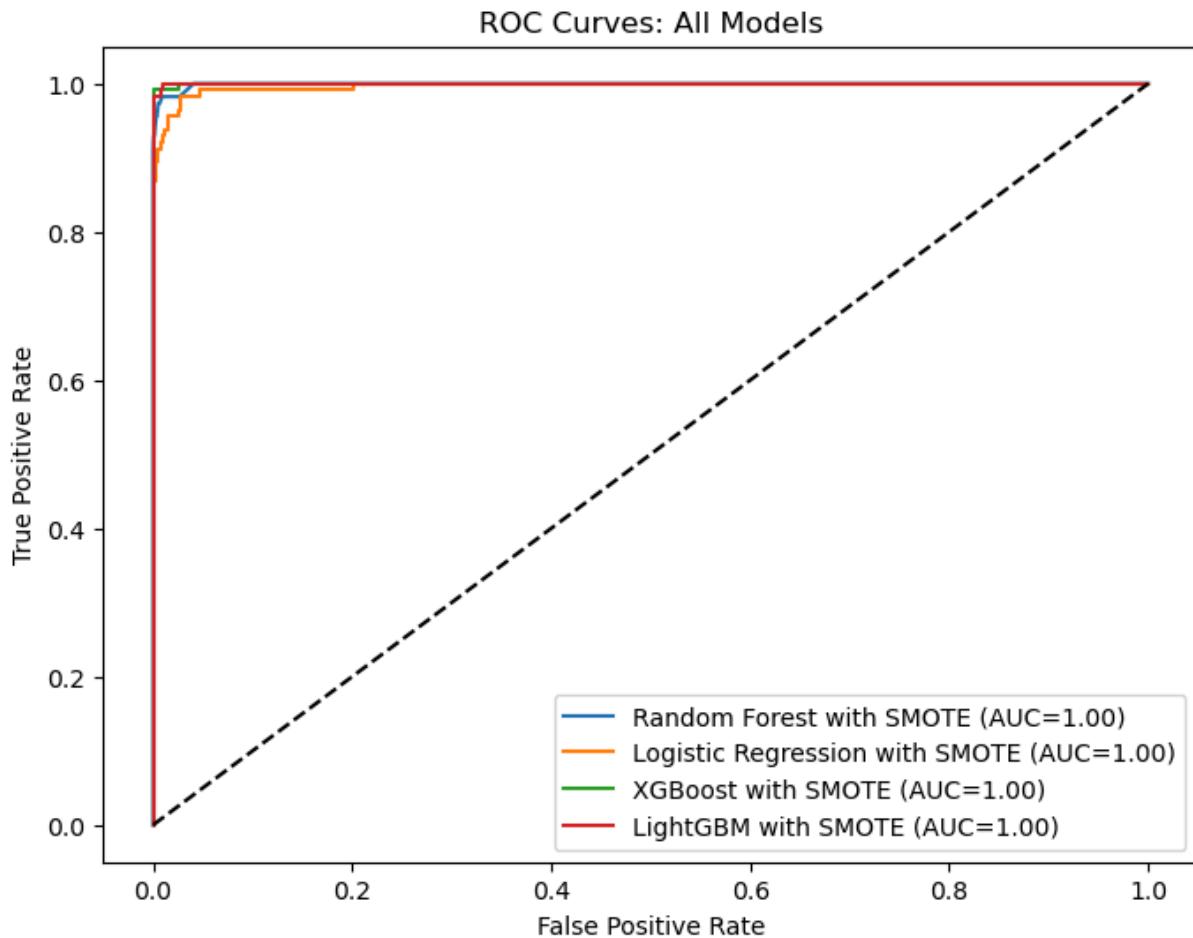
Each curve shows the trade-off between **True Positive Rate** and **False Positive Rate**, and the legend includes each model's **AUC score** as a summary metric of performance.

```
In [185...]: plt.figure(figsize=(8,6))

trainers = [rf_trainer, logreg_trainer, xgb_trainer, lgb_trainer]

for trainer in trainers:
    if trainer.roc_auc is not None:
        model = trainer.trained_model
        y_prob = model.predict_proba(X_test_transformed)[:,1]
        fpr, tpr, _ = roc_curve(y_test, y_prob)
        plt.plot(fpr, tpr, label=f'{trainer.model_name} (AUC={trainer.roc_auc:.2f})')

plt.plot([0,1],[0,1],'k--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curves: All Models')
plt.legend()
plt.show()
```



Interpretation:

- All models achieved very high AUC values ($\approx 0.99\text{--}1.00$), showing excellent separation between classes.
- Boosted models (XGBoost and LightGBM) slightly outperformed Random Forest and Logistic Regression.

- This visualization makes it easy to see which models balance sensitivity vs. specificity best.

Overall, ensemble and boosting approaches with balanced training data (via SMOTE) performed especially well.

Extract Transformed Feature Names

This step extracts feature names **after preprocessing**, including:

- Original numeric feature names (stay unchanged after scaling)
- Encoded categorical feature names (from OneHotEncoder)

This helps us see **which transformed features** go into the model.

At the end, it prints the total number of transformed features.

In [186...]

```
# Step 1: get the preprocessor
preprocessor = prep.preprocessor

# Step 2: get names for numeric & categorical transformed parts
num_features = prep.numeric_features
cat_features = prep.categorical_features

# Get feature names from transformers
num_feature_names = num_features # numeric pipeline keeps names
encoded_cat_features = preprocessor.named_transformers_['cat'].named_steps['encoder']

# Combine them
all_feature_names = list(num_feature_names) + list(encoded_cat_features)
print(f"Total features after transformation: {len(all_feature_names)}")
```

Total features after transformation: 958

Plotting Feature Importance

In this step, we plot the **top 15 most important features** for tree-based models (e.g., Random Forest, XGBoost, LightGBM).

The function:

- Checks if the trained model has `feature_importances_`
- Aligns feature names if the lengths differ
- Uses seaborn to create a barplot showing which features contributed most to predictions

This helps explain the model's decisions to stakeholders and understand key drivers.

```
In [187...]  

def plot_feature_importance(trainer, feature_names):
    """
    Plots feature importance for tree-based models.
    """
    if hasattr(trainer.trained_model, 'feature_importances_'):
        importances = trainer.trained_model.feature_importances_
        print(f"Model expects {len(importances)} features, got {len(feature_names)}")

        # Align Lengths
        if len(importances) != len(feature_names):
            print("Warning: Feature names length does not match importances. Trimming to match.")
            min_len = min(len(importances), len(feature_names))
            importances = importances[:min_len]
            feature_names = feature_names[:min_len]

        fi_df = pd.DataFrame({'feature': feature_names, 'importance': importances})
        fi_df = fi_df.sort_values('importance', ascending=False).head(15)

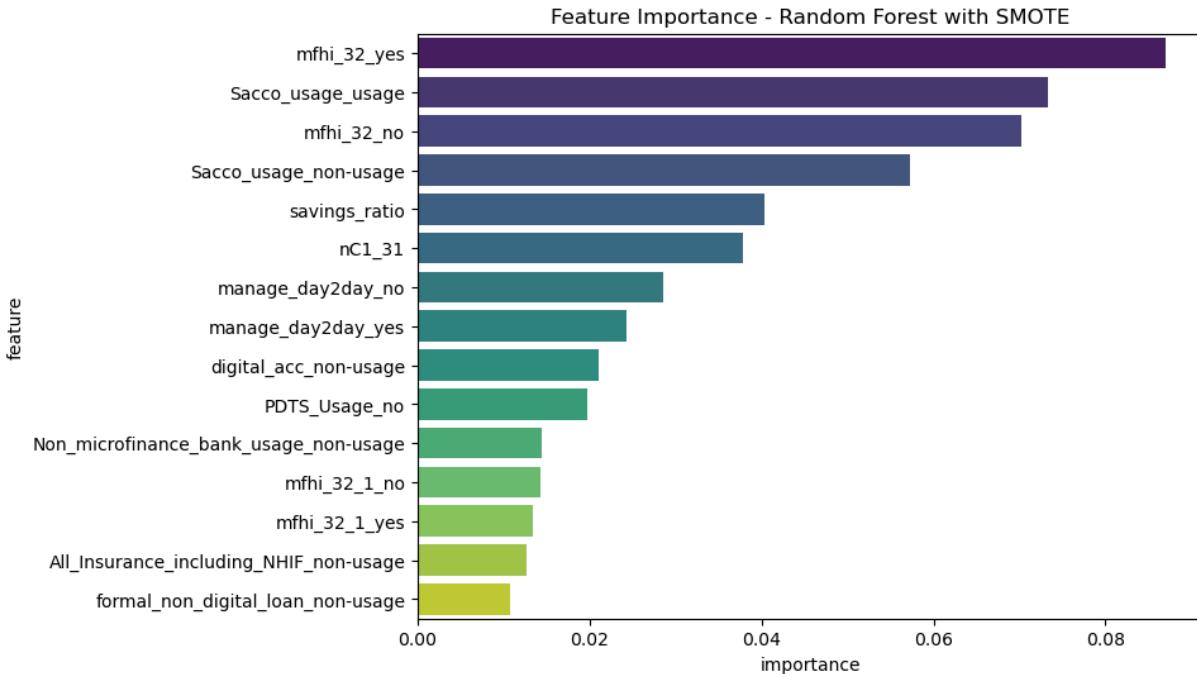
        plt.figure(figsize=(8,6))
        sns.barplot(data=fi_df, x='importance', y='feature', palette='viridis')
        plt.title(f'Feature Importance - {trainer.model_name}')
        plt.show()
    else:
        print(f"{trainer.model_name} has no feature_importances_ attribute.")
```

```
In [188...]  

plot_feature_importance(rf_trainer, all_feature_names)
plot_feature_importance(xgb_trainer, all_feature_names)
plot_feature_importance(lgb_trainer, all_feature_names)
```

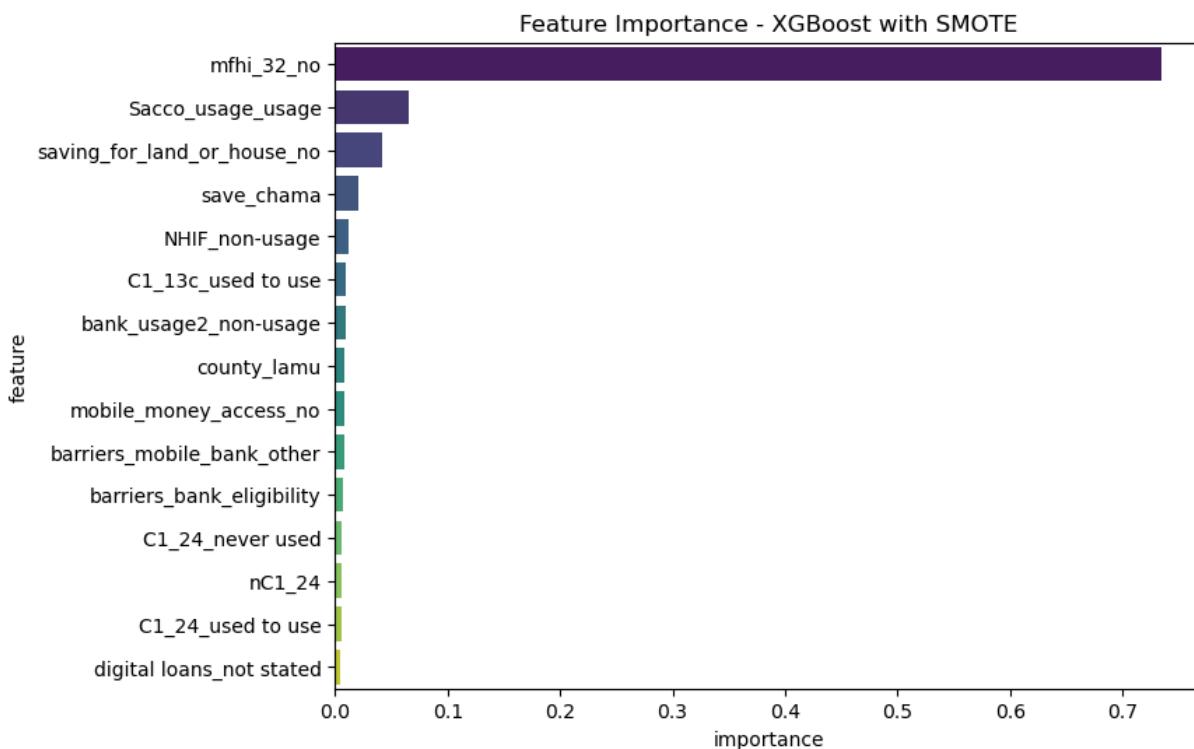
Model expects 954 features, got 958 names

Warning: Feature names length does not match importances. Trimming to match.



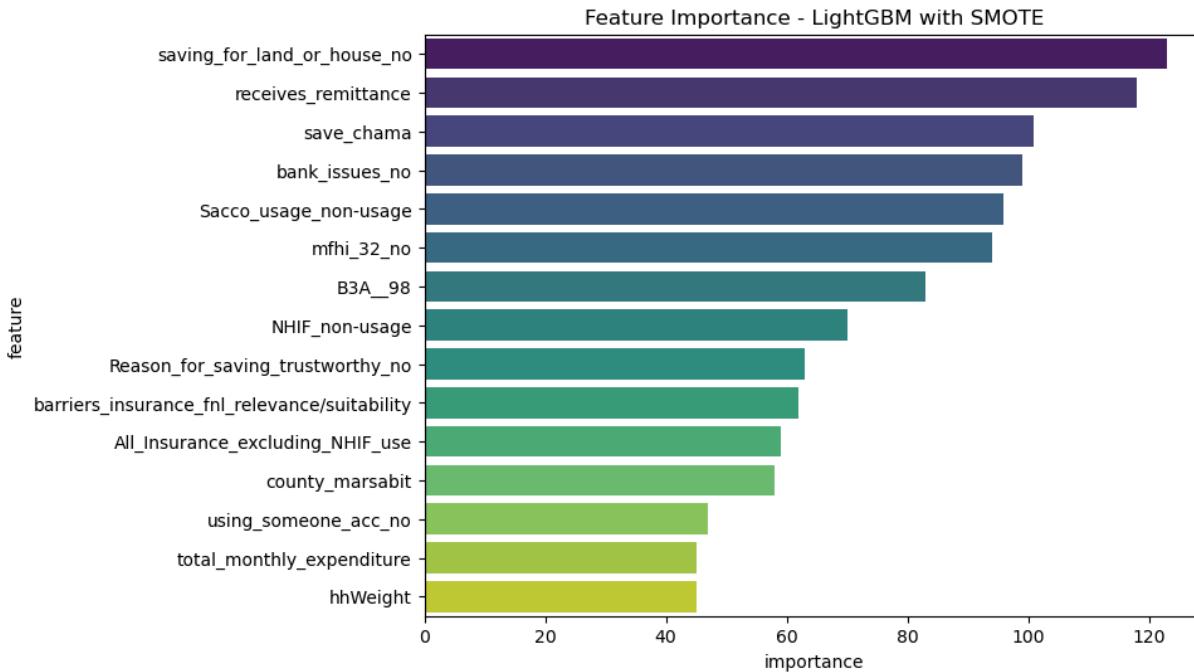
Model expects 954 features, got 958 names

Warning: Feature names length does not match importances. Trimming to match.



Model expects 954 features, got 958 names

Warning: Feature names length does not match importances. Trimming to match.



Model Explainability with SHAP

In this step, we use **SHAP (SHapley Additive exPlanations)** to understand how individual features influence model predictions.

- We apply `TreeExplainer` to both:
 - Random Forest
 - XGBoost

- We compute SHAP values on a **small sample** (first 100 rows) for speed.
 - Then we plot SHAP summary plots to visualize:
 - Which features contribute most to higher/lower predictions
 - The direction of each feature's effect
-

This improves transparency and helps communicate model reasoning to non-technical stakeholders.

In [189...]

```
import shap

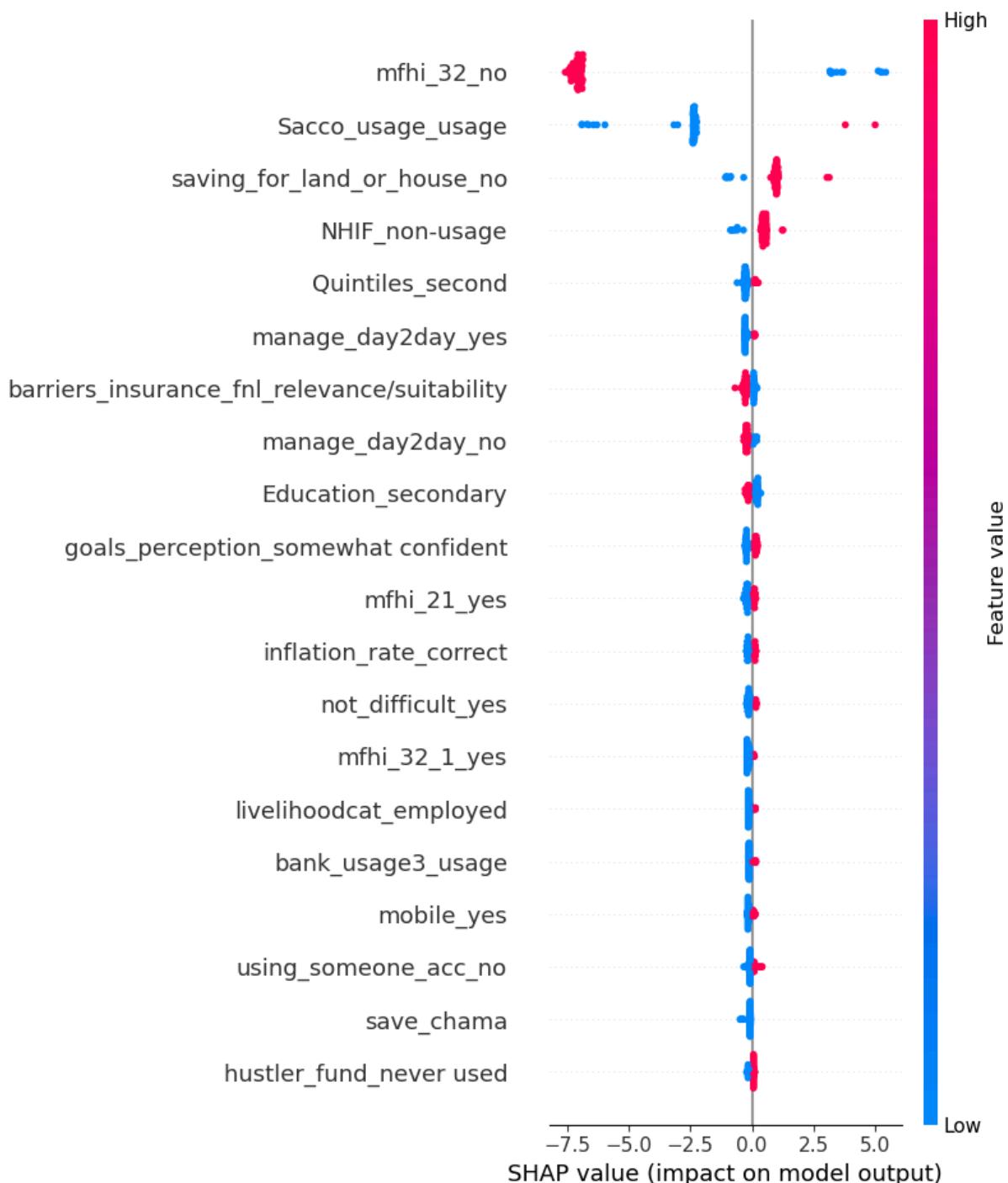
# Use TreeExplainer
explainer_rf = shap.TreeExplainer(rf_trainer.trained_model)
explainer_xgb = shap.TreeExplainer(xgb_trainer.trained_model)

# Compute SHAP values (on a small sample for speed)
shap_values_rf = explainer_rf.shap_values(X_test_transformed[:100])
shap_values_xgb = explainer_xgb.shap_values(X_test_transformed[:100])

# Summary plot: Random Forest
shap.summary_plot(shap_values_rf, features=X_test_transformed[:100], feature_names=)

# Summary plot: XGBoost
shap.summary_plot(shap_values_xgb, features=X_test_transformed[:100], feature_names=)
```





Model Comparison Summary

In this section, we **evaluate and compare** all trained models side by side.

We calculate and store the following metrics:

- **Accuracy**
- **Precision**
- **Recall**
- **ROC AUC**

This helps identify which model performs best overall and which trade-offs exist between precision and recall.

In [190...]

```
# Example: create summary DataFrame
results = []

for trainer in [rf_trainer, logreg_trainer, xgb_trainer, lgb_trainer]:
    model = trainer.trained_model
    y_pred = model.predict(X_test_transformed)
    y_prob = model.predict_proba(X_test_transformed)[:,1]

    accuracy = accuracy_score(y_test, y_pred)
    precision = precision_score(y_test, y_pred, zero_division=0)
    recall = recall_score(y_test, y_pred, zero_division=0)
    auc = roc_auc_score(y_test, y_prob) if y_prob is not None else None

    results.append({
        'Model': trainer.model_name,
        'Accuracy': round(accuracy,4),
        'Precision': round(precision,4),
        'Recall': round(recall,4),
        'ROC AUC': round(auc,4) if auc else None
    })

results_df = pd.DataFrame(results)
display(results_df)
```

	Model	Accuracy	Precision	Recall	ROC AUC
0	Random Forest with SMOTE	0.9950	1.0000	0.8142	0.9992
1	Logistic Regression with SMOTE	0.9959	0.9898	0.8584	0.9964
2	XGBoost with SMOTE	0.9995	1.0000	0.9823	0.9998
3	LightGBM with SMOTE	0.9995	1.0000	0.9823	0.9998

Summary table helps guide final model selection based on balanced metrics.