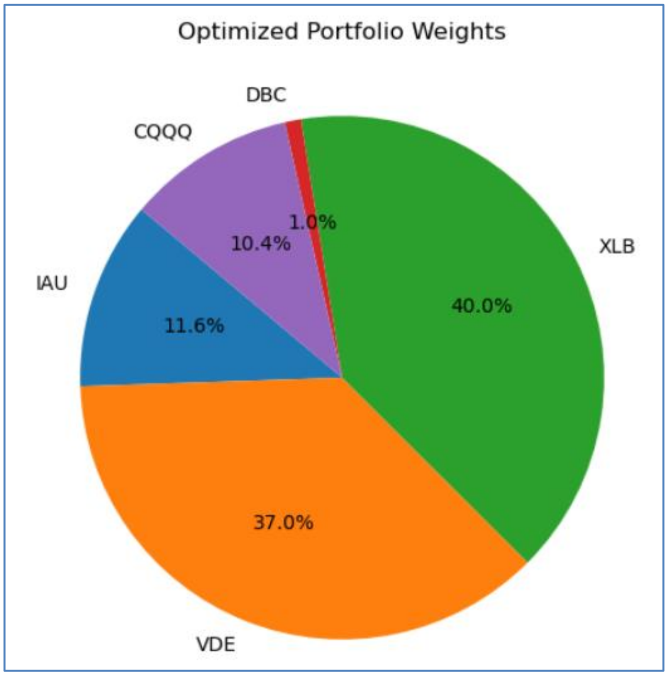
Part 1: Optimal Portfolio Allocation

Construction of a pie chart with optimal portfolio weights

- ETFs: IAU, VDE, XLB, DBC, CQQQ
- Weight Constraints: 1% <= weights <= 40%
- Objective: Minimize volatility based on the historical var-cov data
- Data range: Historical data from Dec-2018 to Dec-2021
- To be calculated:
  - Average return
  - Variance and covariance of the portfolio components.

```
# Downloading historical data
tickers = ['IAU', 'VDE', 'XLB', 'DBC', 'CQQQ']
data = yf.download(tickers, start='2018-12-01', end='2021-12-31', interval='1mo')['Adj Close']

# Calculating monthly returns using end-of-month prices
monthly_prices = data.resample('M').last()
monthly_returns = monthly_prices.pct_change().dropna()
average_returns = monthly_returns.mean()
cov_matrix = monthly_returns.cov()

# Defining the objective function to minimize portfolio volatility
def portfolio_volatility(weights, cov_matrix):
   return np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))

# Defining the objective function to minimize portfolio volatility
def portfolio_volatility(weights, cov_matrix):
   return np.sqrt(np.dot(weights.T, np.dot(cov_matrix, weights)))

# Optimization to minimize portfolio volatility
opt_result = minimize(portfolio_volatility, initial_guess, args=(cov_matrix,),
                      method='SLSQP', bounds=bounds, constraints=constraints)
weights = opt_result.x
```

Pie Chart:



<u>Average monthly returns for each ETF:</u>

Ticker
CQQQ 1.590261
DBC 1.259653
IAU 1.058658
VDE 1.131806
XLB 1.997917

<u>Variance of Each ETF:</u>

IAU 0.005813
VDE 0.003445
XLB 0.001826
DBC 0.015297
CQQQ 0.003836

<u>Covariance Matrix of Portfolio Components:</u>

Ticker	CQQQ	DBC	IAU	VDE	XLB
Ticker					
CQQQ	0.005813	0.001912	0.000439	0.003994	0.001726
DBC	0.001912	0.003445	-0.000006	0.005059	0.002246
IAU	0.000439	-0.000006	0.001826	-0.000237	0.000730
VDE	0.003994	0.005059	-0.000237	0.015297	0.005624
XLB	0.001726	0.002246	0.000730	0.005624	0.003836

Performance of the optimal portfolio since Jan-2022 with a time-series chart. Present the performance statistics. (Cumulative return, annualized return, annualized volatility)

Data time range: January 2022 till October 2024

 # Download data from Jan 2022 to present

future_data = yf.download(tickers, start='2022-01-01', end='2024-10-31', interval='1d')['Adj Close']

 # Monthly portfolio returns using optimized weights:

future_monthly_prices = future_data.resample('M').last()
future_monthly_returns = future_monthly_prices.pct_change().dropna()
future_portfolio_returns = future_monthly_returns.dot(weights)

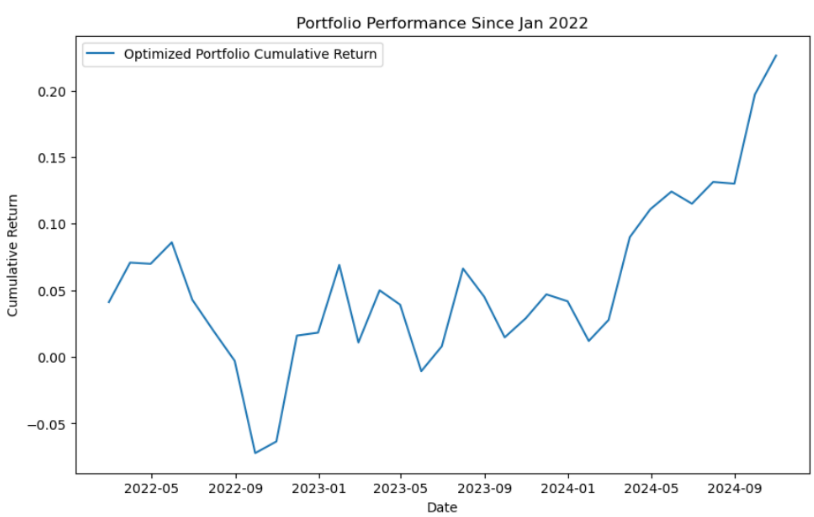
 # Performance Statistics:

cumulative_return = (1 + future_portfolio_returns).prod() - 1
*annualized_return = (1 + cumulative_return) ** (12 / len(future_portfolio_returns)) - 1*
*annualized_volatility = future_portfolio_returns.std() * np.sqrt(12)*

- Cumulative Return: 22.62%. If I had invested in this Portfolio at the beginning of January 2022, my investment would have grown by 22.62% until now.
- Annualized Return: 7.70%. The portfolio has grown by 7.70% every year from January 2022 to October 2024.
- Annualized Volatility: 12.30%. This is the indication of risk or annual fluctuation of 12.30 for this portfolio.

Graphical representation of the Portfolio's performance since Jan-2022:

cumulative_returns = (1 + future_portfolio_returns).cumprod() – 1



Key takeaways from the above time-series performance for Jan-2022 to Oct-2024:

- 2022: In early 2022, the portfolio started with an upward trend, but then started to drop significantly starting from Q2 2022. The most possible impact was due to the Ukraine was which started in Feb 2022. That was the most volatile period. The war might have impacted the energy (VDE), commodities (DBC), Gold (IAU) and Tech (CQQQ) due to supply chain disruptions and restricted commodities' export from Russia and Ukraine.
- 2023 had several upward and downward trends.
- 2024: The portfolio had a very strong upward trend since Q1 2024, indicating favorable market conditions.

Actual portfolio performance relative to S&P 500 over Jan-2022 to Oct-2024 period. Provide charts and numbers as in (b)

 # Downloading S&P 500 data

sp500_data = yf.download('^GSPC', start='2022-01-01', end='2024-10-31', interval='1d')['Adj Close']

 # Calculating monthly returns for S&P 500

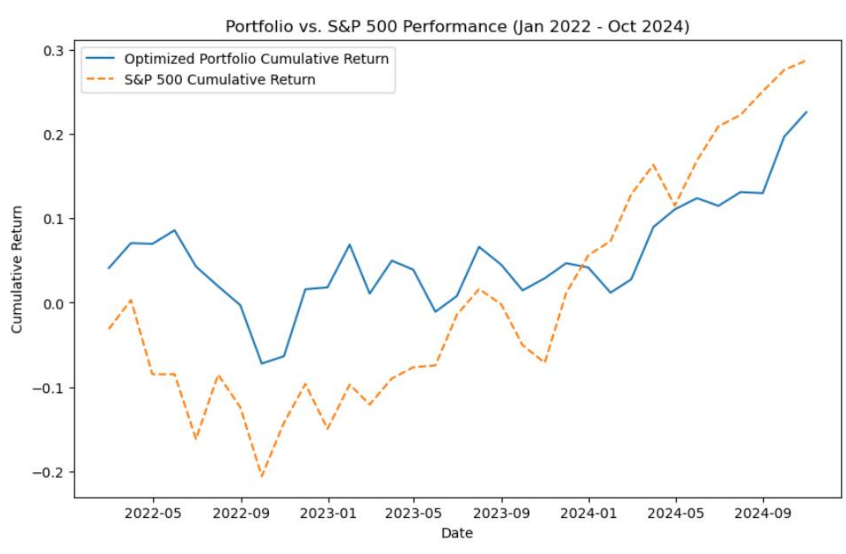
sp500_monthly_prices = sp500_data.resample('M').last()
sp500_monthly_returns = sp500_monthly_prices.pct_change().dropna()

 # Calculating cumulative returns for the portfolio

portfolio_cumulative_returns = (1 + future_portfolio_returns).cumprod() – 1

 # Calculating cumulative returns for S&P 500

sp500_cumulative_returns = (1 + sp500_monthly_returns).cumprod() - 1



 # performance metrics for portfolio

```
portfolio_cumulative_return = portfolio_cumulative_returns.iloc[-1]  # I have used '.iloc[-1]' to get the last value safely
portfolio_annualized_return = (1 + portfolio_cumulative_return) ** (12 / len(future_portfolio_returns)) - 1
portfolio_annualized_volatility = future_portfolio_returns.std() * np.sqrt(12)
```

 # performance metrics for S&P 500

```
sp500_cumulative_return = sp500_cumulative_returns.iloc[-1]
sp500_annualized_return = (1 + sp500_cumulative_return) ** (12 / len(sp500_monthly_returns)) - 1
sp500_annualized_volatility = sp500_monthly_returns.std() * np.sqrt(12)
```

 # Ensuring metrics are scalars by converting them to float if they are Series

```
sp500_cumulative_return = float(sp500_cumulative_return.iloc[0]) if isinstance(sp500_cumulative_return, pd.Series) else float(sp500_cumulative_return)
sp500_annualized_return = float(sp500_annualized_return.iloc[0]) if isinstance(sp500_annualized_return, pd.Series) else float(sp500_annualized_return)
sp500_annualized_volatility = float(sp500_annualized_volatility.iloc[0]) if isinstance(sp500_annualized_volatility, pd.Series) else float(sp500_annualized_volatility)

portfolio_cumulative_return = float(portfolio_cumulative_return.iloc[0]) if isinstance(portfolio_cumulative_return, pd.Series) else float(portfolio_cumulative_return)
portfolio_annualized_return = float(portfolio_annualized_return.iloc[0]) if isinstance(portfolio_annualized_return, pd.Series) else float(portfolio_annualized_return)
portfolio_annualized_volatility = float(portfolio_annualized_volatility.iloc[0]) if isinstance(portfolio_annualized_volatility, pd.Series) else float(portfolio_annualized_volatility)
```

Optimized Portfolio Performance (Jan 2022 - Oct 2024):
Cumulative Return: 22.62%
Annualized Return: 7.70%
Annualized Volatility: 12.30%

S&P 500 Performance (Jan 2022 - Oct 2024):
Cumulative Return: 28.75%
Annualized Return: 9.62%
Annualized Volatility: 17.37%

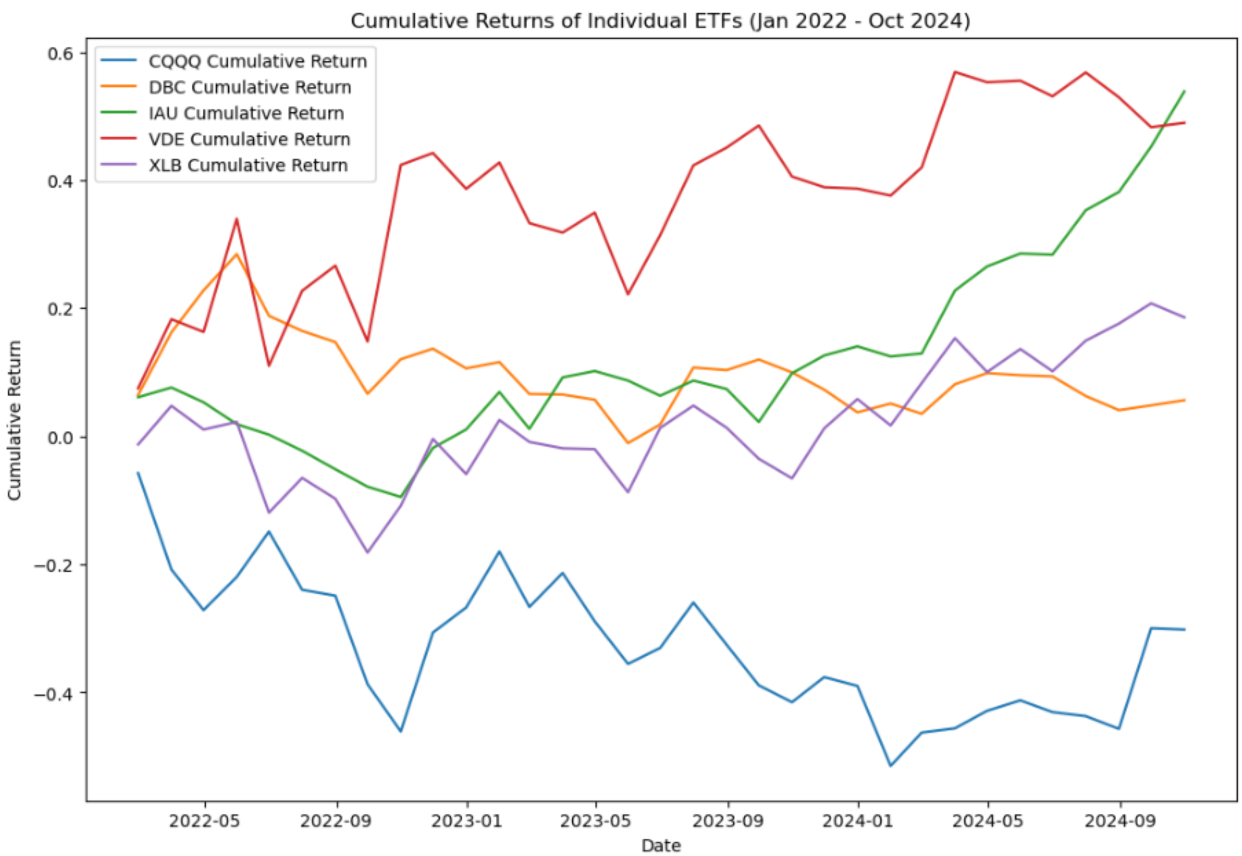
Performance of each component (ETF) in the portfolio separately. Show performance charts and statistics

 # Cumulative returns for each ETF

cumulative_returns = (1 + future_monthly_returns).cumprod() – 1

 # Plotting cumulative returns for each ETF

plt.figure(figsize=(12, 8))
for ticker in cumulative_returns.columns:
plt.plot(cumulative_returns[ticker], label=f'{ticker} Cumulative Return')



Key takeaways:

- High cumulative return: VDE (Energy) and IAU (Gold/ Precious Metals) show upward trend with some volatility over the period and most cumulative return
- Moderate cumulative return: XLB (Materials)
- Low cumulative return: DBC (Commodities)
- Poor cumulative return: CQQQ (China Technology)

```
# Performance statistics for each ETF
performance_stats = {}
for ticker in future_monthly_returns.columns:
    cumulative_return = cumulative_returns[ticker].iloc[-1]  # Last value in cumulative return
    annualized_return = (1 + cumulative_return) ** (12 / len(future_monthly_returns)) - 1
    annualized_volatility = future_monthly_returns[ticker].std() * np.sqrt(12)

    performance_stats[ticker] = {
        'Cumulative Return': cumulative_return,
        'Annualized Return': annualized_return,
        'Annualized Volatility': annualized_volatility
    }

# Converting performance stats to a DataFrame for better display
performance_df = pd.DataFrame(performance_stats).T
performance_df.index.name = 'ETF'
performance_df = performance_df.applymap(lambda x: f"{x:.2%}")  # Format as percentages
```

Performance Statistics for Each ETF (Jan-2022 to Oct-2024):

ETF	Cumulative Return	Annualized Return	Annualized Volatility
CQQQ	-30.17%	-12.24%	39.71%
DBC	5.64%	2.02%	14.15%
IAU	53.86%	16.96%	13.83%
VDE	48.96%	15.59%	26.52%
XLB	18.59%	6.40%	21.16%

Cumulative Return:

- Top Performers: IAU (53.86%) and VDE (48.96%).
- Worst Performer: CQQQ (-30.17%), indicating poor performance in Chinese technology.

Annualized Return:

- Highest Annualized Returns: IAU (16.96%) and VDE (15.59%), consistent with their high cumulative returns.
- Lowest Annualized Return: CQQQ (-12.24%), showing that Chinese technology has been in a downtrend over this period.

Annualized Volatility:

- Most Volatile: CQQQ (39.71%), reflecting the high volatility in Chinese technology stocks.
- Least Volatile: IAU (13.83%) and DBC (14.15%), indicating that gold and commodities are relatively stable assets in this portfolio

Part 2: Portfolio Simulation

Histogram of simulated End values

- Simulation period: Jan-2022 to Jan-2024
- Frequency (step size): Weekly
- Number of Simulation Paths: 1000
- Initial Portfolio Value: 100

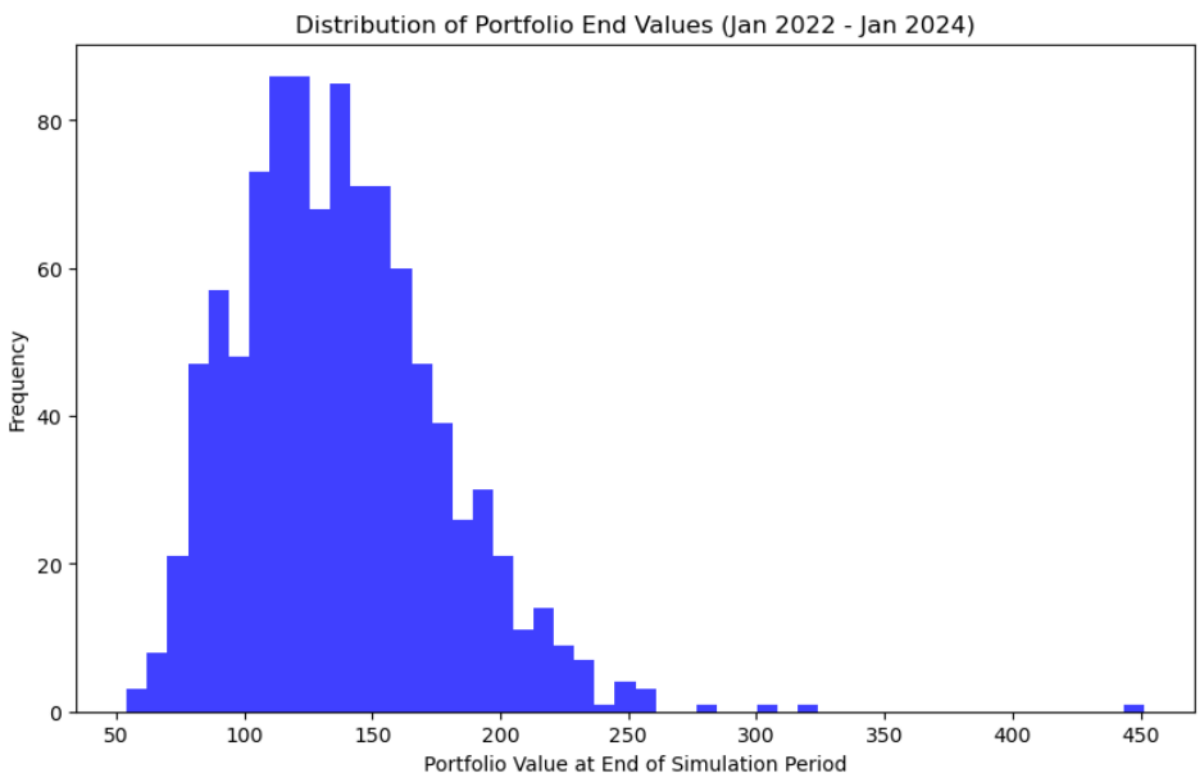
```
# Defining the ETFs and downloading historical data
tickers = ['IAU', 'VDE', 'XLB', 'DBC', 'CQQQ']
data = yf.download(tickers, start="2018-12-01", end="2021-12-31", interval="1d")['Adj Close']

# Calculating weekly returns, average returns, and covariance matrix
weekly_prices = data.resample('W').last()
weekly_returns = weekly_prices.pct_change().dropna()
average_weekly_returns = weekly_returns.mean()
cov_matrix = weekly_returns.cov()
```

Variance-Covariance Matrix of Portfolio Components:

Ticker	CQQQ	DBC	IAU	VDE	XLB
Ticker					
CQQQ	0.001729	0.000520	0.000193	0.000793	0.000645
DBC	0.000520	0.000675	0.000141	0.001073	0.000609
IAU	0.000193	0.000141	0.000448	0.000147	0.000267
VDE	0.000793	0.001073	0.000147	0.002926	0.001423
XLB	0.000645	0.000609	0.000267	0.001423	0.001305

```
optimal_weights = [0.4, 0.37, 0.116, 0.104, 0.01]  # XLB, VDE, IAU, CQQQ, DBC

# Simulation Parameters
num_simulations = 1000          # Number of simulation paths
weeks_in_simulation = 104        # Weekly steps from Jan 2022 to Jan 2024 (2 years)
initial_portfolio_value = 100    # Initial portfolio value

# Cholesky decomposition of the covariance matrix
cholesky_decomp = np.linalg.cholesky(cov_matrix)

# Running simulations
for _ in range(num_simulations):
    portfolio_value = initial_portfolio_value
    weekly_values = np.ones(weeks_in_simulation) * initial_portfolio_value  # Track weekly portfolio
values
```

```
for week in range(weeks_in_simulation):
    # Generate random vector for correlated returns
    random_vector = np.random.normal(0, 1, len(tickers))
    correlated_shocks = np.dot(cholesky_decomp, random_vector)

    # Calculating weekly return using mean returns and correlated shocks
    weekly_return = np.dot(optimal_weights, average_weekly_returns + correlated_shocks)

    # Updating portfolio value using GBM formula
    portfolio_value *= np.exp(weekly_return)
    weekly_values[week] = portfolio_value

# Storing the final portfolio value for the simulation
final_values.append(portfolio_value)
```



- **Central Tendency:** The mode of the portfolio is between 100-150, which is close to the initial portfolio value.
- **Distribution Spread:** Right-skewed, with a long tail extending towards higher portfolio values. Shows potential for higher end values in some scenarios.

- **Range of Outcomes:** Majority of the simulated end values fall within the range of about 50 to 200, showing both downside risk and upside potential. The presence of values exceeding 200 (and even reaching up to 300) indicates that under favorable market conditions, the portfolio has the potential for significant gains.
- **Risk and Variability:** The spread of values around the mean suggests variability in outcomes, with both downside and upside risks. The broader distribution indicates moderate risk in the portfolio, as there is potential for returns both above and below the initial investment.

Comparison of the Actual portfolio performance with the simulated scenarios. Is the Actual return a tail event according to the simulated values?

I compared the actual end value of the portfolio against the distribution of the simulated end values. A tail event typically refers to an outcome that is in the extreme ends (tails) of the distribution, generally in the lowest or highest 5% of outcomes.

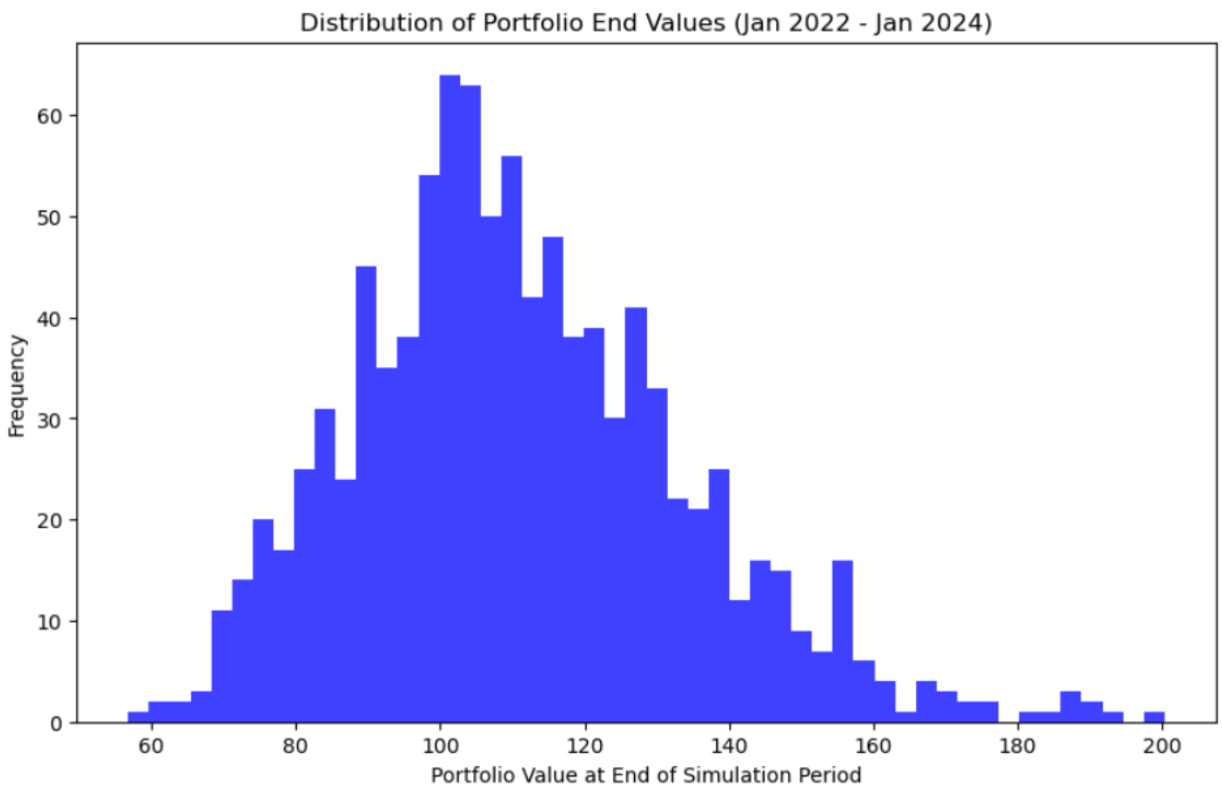
Date range: January 2022 to January 2024
Weight constraints: 1% <= weights <= 40%

Optimized Weights (Weekly Frequency for Jan 2022 - Jan 2024): [0.05962751 0.26948757 0.4 0.01 0.26088493]

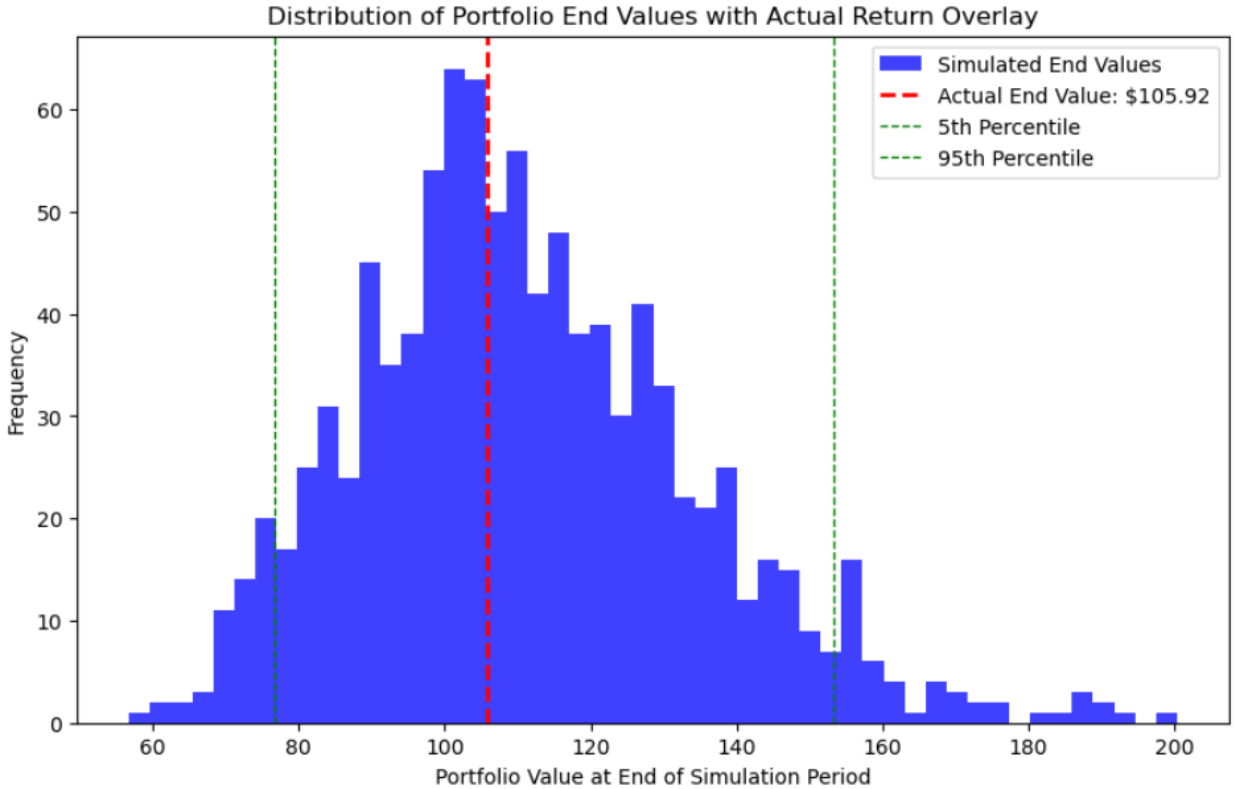
I used these actual weights to build the comparison with simulation

```
# Running GBM Simulations
# Simulation Parameters
num_simulations = 1000        # Number of simulation paths
weeks_in_simulation = 104      # Weekly steps for 2 years (Jan 2022 - Jan 2024)
initial_portfolio_value = 100   # Initial portfolio value

# Cholesky decomposition of the covariance matrix
cholesky_decomp = np.linalg.cholesky(cov_matrix_weekly)

# Running simulations
for _ in range(num_simulations):
   portfolio_value = initial_portfolio_value
   weekly_values = np.ones(weeks_in_simulation) * initial_portfolio_value  # Track weekly portfolio
values

   for week in range(weeks_in_simulation):
      # Generate random vector for correlated returns
      random_vector = np.random.normal(0, 1, len(tickers))
```

```
        correlated_shocks = np.dot(cholesky_decomp, random_vector)

        # Calculate weekly return using mean returns and correlated shocks
        weekly_return = np.dot(optimal_weights, average_weekly_returns + correlated_shocks)

        # Update portfolio value using GBM formula
        portfolio_value *= np.exp(weekly_return)
        weekly_values[week] = portfolio_value

    # Store the final portfolio value for the simulation
    final_values.append(portfolio_value)
```




```
# Calculating the actual weekly portfolio return using optimized weights
weekly_portfolio_returns_actual = weekly_returns.dot(optimal_weights)

# Calculate the actual portfolio end value over the period
actual_end_value = initial_portfolio_value * (1 + weekly_portfolio_returns_actual).cumprod().iloc[-1]

# Calculating percentiles for the simulated distribution
lower_percentile = np.percentile(final_values, 5)  # 5th percentile (lower tail)
upper_percentile = np.percentile(final_values, 95) # 95th percentile (upper tail)
```




```
# Determining if the actual end value is a tail event
if actual_end_value < lower_percentile:
    print("The actual portfolio end value is in the lower 5% tail, indicating a negative tail event.")
elif actual_end_value > upper_percentile:
    print("The actual portfolio end value is in the upper 5% tail, indicating a positive tail event.")
else:
    print("The actual portfolio end value falls within the central 90% of simulated outcomes, indicating it is not a tail event.")
```

The actual portfolio end value falls within the central 90% of simulated outcomes, indicating it is not a tail event.

- The actual portfolio end value is 105.92 dollars (shown as the red dashed line). This is slightly above the initial investment of 100 dollars but falls within the central region of the simulated outcomes.
- The distribution of simulated end values is centered around 100, with most values falling between approximately 50 and 150.
- Since the actual portfolio end value of 105.92 dollars lies within the 5th and 95th percentiles, it is not considered a tail event.

Part 3: Logit Regression for Credit Default

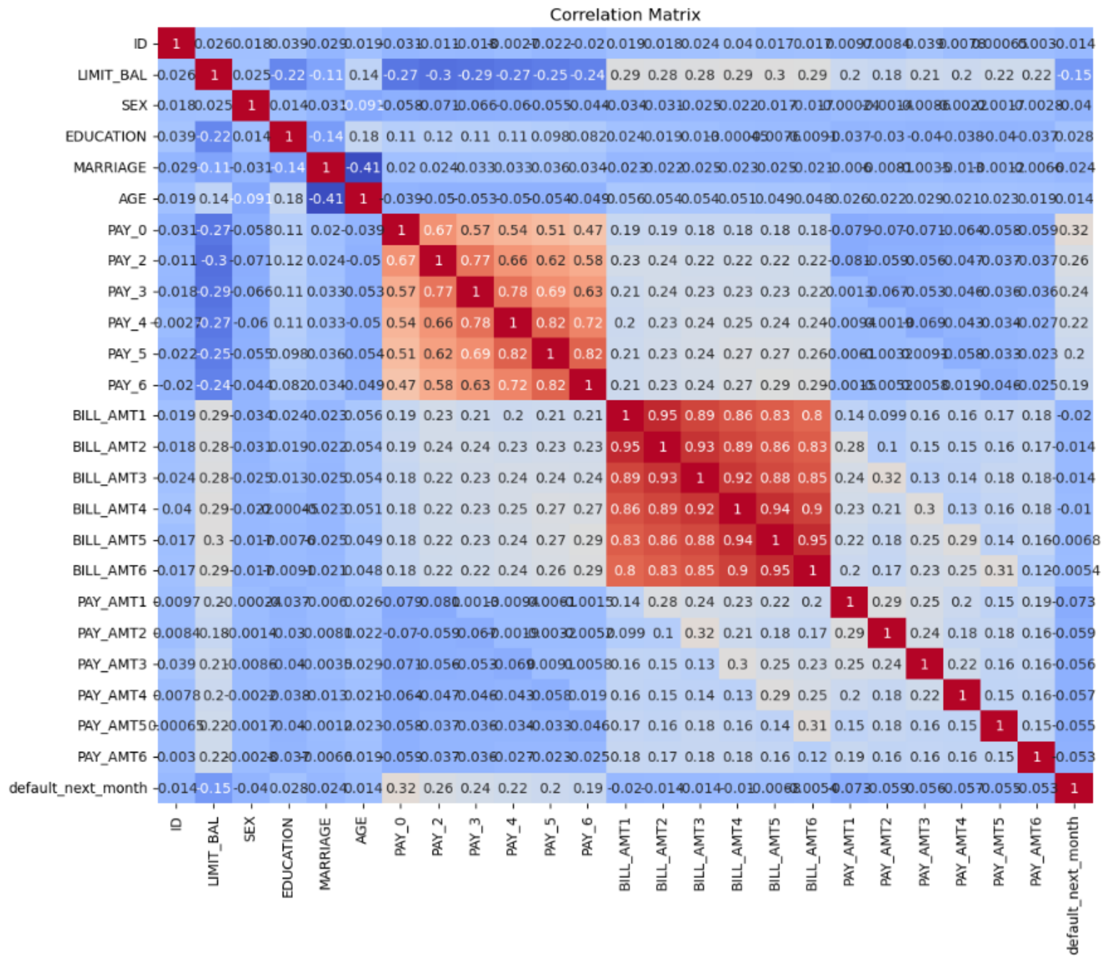
Applying the Logit regression for Train Sample: First 20000 rows, Test Sample: The rest of the sample (next 10000)

Exploratory Data Analysis:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 30000 entries, 0 to 29999
Data columns (total 25 columns):
 #   Column              Non-Null Count  Dtype
---  ------              --------------  -----
 0   ID                  30000 non-null  int64
 1   LIMIT_BAL           30000 non-null  int64
 2   SEX                 30000 non-null  int64
 3   EDUCATION           30000 non-null  int64
 4   MARRIAGE            30000 non-null  int64
 5   AGE                 30000 non-null  int64
 6   PAY_0               30000 non-null  int64
 7   PAY_2               30000 non-null  int64
 8   PAY_3               30000 non-null  int64
 9   PAY_4               30000 non-null  int64
 10  PAY_5               30000 non-null  int64
 11  PAY_6               30000 non-null  int64
 12  BILL_AMT1           30000 non-null  int64
 13  BILL_AMT2           30000 non-null  int64
 14  BILL_AMT3           30000 non-null  int64
 15  BILL_AMT4           30000 non-null  int64
 16  BILL_AMT5           30000 non-null  int64
 17  BILL_AMT6           30000 non-null  int64
 18  PAY_AMT1            30000 non-null  int64
 19  PAY_AMT2            30000 non-null  int64
 20  PAY_AMT3            30000 non-null  int64
 21  PAY_AMT4            30000 non-null  int64
 22  PAY_AMT5            30000 non-null  int64
 23  PAY_AMT6            30000 non-null  int64
 24  default_next_month  30000 non-null  int64
dtypes: int64(25)
```

```
Current columns in the dataset:
Index(['ID', 'LIMIT_BAL', 'SEX', 'EDUCATION', 'MARRIAGE', 'AGE', 'PAY_0',
       'PAY_2', 'PAY_3', 'PAY_4', 'PAY_5', 'PAY_6', 'BILL_AMT1', 'BILL_AMT2',
       'BILL_AMT3', 'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6', 'PAY_AMT1',
       'PAY_AMT2', 'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6',
       'default_next_month'],
      dtype='object')
```

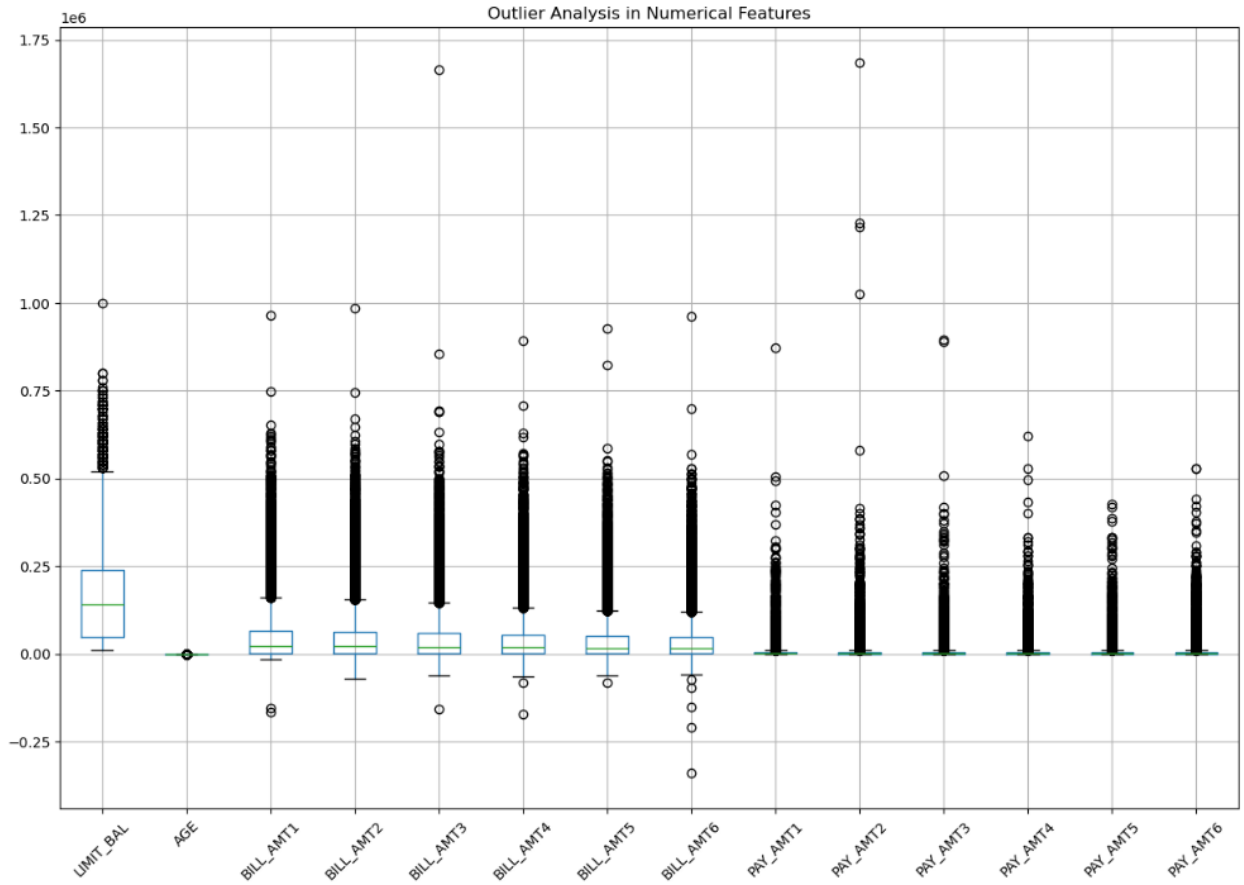
```
                 ID        LIMIT_BAL           SEX     EDUCATION      MARRIAGE  \
count  30000.000000     30000.000000  30000.000000  30000.000000  30000.000000
mean   15000.500000    167484.322667      1.603733      1.853133      1.551867
std     8660.398374    129747.661567      0.489129      0.790349      0.521970
min        1.000000     10000.000000      1.000000      0.000000      0.000000
25%     7500.750000     50000.000000      1.000000      1.000000      1.000000
50%    15000.500000    140000.000000      2.000000      2.000000      2.000000
75%    22500.250000    240000.000000      2.000000      2.000000      2.000000
max    30000.000000   1000000.000000      2.000000      6.000000      3.000000

                AGE         PAY_0         PAY_2         PAY_3         PAY_4  \
count  30000.000000  30000.000000  30000.000000  30000.000000  30000.000000
mean      35.485500     -0.016700     -0.133767     -0.166200     -0.220667
std        9.217904      1.123802      1.197186      1.196868      1.169139
min       21.000000     -2.000000     -2.000000     -2.000000     -2.000000
25%       28.000000     -1.000000     -1.000000     -1.000000     -1.000000
50%       34.000000      0.000000      0.000000      0.000000      0.000000
75%       41.000000      0.000000      0.000000      0.000000      0.000000
max       79.000000      8.000000      8.000000      8.000000      8.000000

       ...      BILL_AMT4      BILL_AMT5      BILL_AMT6      PAY_AMT1  \
count  ...   30000.000000   30000.000000   30000.000000  30000.000000
mean   ...   43262.948967   40311.400967   38871.760400   5663.580500
std    ...   64332.856134   60797.155770   59554.107537  16563.280354
min    ... -170000.000000  -81334.000000 -339603.000000      0.000000
25%    ...    2326.750000    1763.000000    1256.000000   1000.000000
50%    ...   19052.000000   18104.500000   17071.000000   2100.000000
75%    ...   54506.000000   50190.500000   49198.250000   5006.000000
max    ...  891586.000000  927171.000000  961664.000000 873552.000000

           PAY_AMT2      PAY_AMT3       PAY_AMT4       PAY_AMT5  \
count  3.000000e+04   30000.00000   30000.000000   30000.000000
mean   5.921163e+03    5225.68150    4826.076867    4799.387633
std    2.304087e+04   17606.96147   15666.159744   15278.305679
min    0.000000e+00       0.00000       0.000000       0.000000
25%    8.330000e+02     390.00000     296.000000     252.500000
50%    2.009000e+03    1800.00000    1500.000000    1500.000000
75%    5.000000e+03    4505.00000    4013.250000    4031.500000
max    1.684259e+06  896040.00000  621000.000000  426529.000000

            PAY_AMT6  default_next_month
count   30000.000000        30000.000000
mean     5215.502567            0.221200
std     17777.465775            0.415062
min         0.000000            0.000000
25%       117.750000            0.000000
50%      1500.000000            0.000000
75%      4000.000000            0.000000
max    528666.000000            1.000000
```



Key Observation on the Heatmap:

- **Payment Status (PAY_0 to PAY_6):** Most significant predictors of default_next_month, with recent payment behavior (especially PAY_0) showing a stronger association with the target variable.
- **LIMIT_BAL:** Shows a modest negative association. Indicating that individuals with higher credit limits are less likely to default
- **Bill Amounts:** Although these amounts are highly correlated with each other. It suggests that individuals who have higher bills in one month often have higher bills in other months. This pattern may indicate consistent spending behavior over time.
- **Demographic features** (Age, Sex, Education, Marriage) has less correlation with the default_next_month.

This heatmap suggests that PAY_0 (and possibly other PAY features) are essential for predicting defaults, while LIMIT_BAL and some demographic information might contribute but are not primary indicators.



The outliers observed in Balance Limit, or various Bill Amounts are from diversified financial behaviors. I will not remove them at this moment to keep the critical information intact.

 # Encoding Categorical Variables

```
categorical_features = ['SEX', 'EDUCATION', 'MARRIAGE']
data = pd.get_dummies(data, columns=categorical_features, drop_first=True)
```

 # Normalizing Numerical Columns

```
numerical_features = [
    'LIMIT_BAL', 'AGE', 'BILL_AMT1', 'BILL_AMT2', 'BILL_AMT3',
    'BILL_AMT4', 'BILL_AMT5', 'BILL_AMT6', 'PAY_AMT1', 'PAY_AMT2',
    'PAY_AMT3', 'PAY_AMT4', 'PAY_AMT5', 'PAY_AMT6'
]
```

 # Scaling

```
scaler = StandardScaler()
data[numerical_features] = scaler.fit_transform(data[numerical_features])
```

 # Splitting the Data into Training and Test Sets

```
train_data = data.iloc[:20000]
test_data = data.iloc[20000:]

X_train = train_data.drop(columns=['ID', 'default_next_month'])
y_train = train_data['default_next_month']
X_test = test_data.drop(columns=['ID', 'default_next_month'])
y_test = test_data['default_next_month']
```

 # Training the Logistic Regression Model

```
logit_model = LogisticRegression(max_iter=1000, random_state=42)
logit_model.fit(X_train, y_train)
```

```
LogisticRegression
LogisticRegression(max_iter=1000, random_state=42)
```

```
Accuracy for Original Train-Test split: 0.8207
Classification Report for Original Train-Test split:
              precision    recall  f1-score   support

           0       0.83      0.98      0.90      7922
           1       0.73      0.22      0.33      2078

    accuracy                           0.82     10000
   macro avg       0.78      0.60      0.61     10000
weighted avg       0.81      0.82      0.78     10000
```

 # Making Predictions and Evaluating the Model for the original train-test split

```
y_pred = logit_model.predict(X_test)
accuracy = accuracy_score(y_test, y_pred)
classification_rep = classification_report(y_test, y_pred)
```

```
Accuracy for Original Train-Test split: 0.8207
Classification Report for Original Train-Test split:
              precision    recall  f1-score   support

           0       0.83      0.98      0.90      7922
           1       0.73      0.22      0.33      2078

    accuracy                           0.82     10000
   macro avg       0.78      0.60      0.61     10000
weighted avg       0.81      0.82      0.78     10000
```

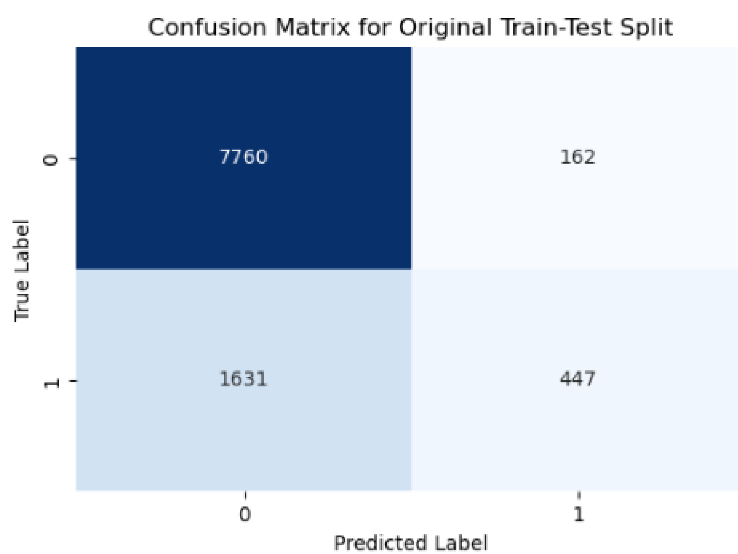
Key Takeaways:

Overall Model Accuracy: The model achieved an accuracy of 82.07% on the test data. This means that about 82% of the predictions (defaults and non-defaults) were correct.

Class-wise Performance:

- **Class 0 (Non-defaults):**
  - **Precision: 0.83** – The model is correct on predicting 'non-defaults' 83% of the time.
  - **Recall: 0.98** – Among all actual non-defaults, the model correctly identifies 98% of them.
  - **F1-score: 0.90** – This combines precision and recall, indicating strong performance in predicting non-defaults.

- **Class 1 (Defaults):**
  - **Precision: 0.73** – When the model predicts a default, it's correct 73% of the time.
  - **Recall: 0.22** – The model identifies only 22% of actual defaults, meaning it misses a significant portion.
  - **F1-score: 0.33** – This low score suggests that while the model can predict defaults, it struggles to capture all actual defaults.
- **Averages:**
  - **Macro Avg:** This is the unweighted average of the scores for each class. The recall score (0.60) shows a significant imbalance in the model's ability to detect each class, with better performance on non-defaults than defaults.
  - **Weighted Avg:** This average is weighted by the number of instances in each class, and it's closer to the model's accuracy, reflecting those non-defaults are the majority.



True Positive Rate (Recall for Defaults):

- Recall for defaults (class 1): TP/ (TP+FN) ≈ 447/ (447+1631) ≈ 21.5%
- This recall indicates that the model only correctly identifies about 21.5% of actual defaulters.

True Negative Rate (Recall for Non-defaults):

- The recall for non-defaults (class 0): TN/ (TN+FP) ≈ 7760/ (7760+162) ≈ 97.9%
- The model correctly identifies nearly all non-defaulters.

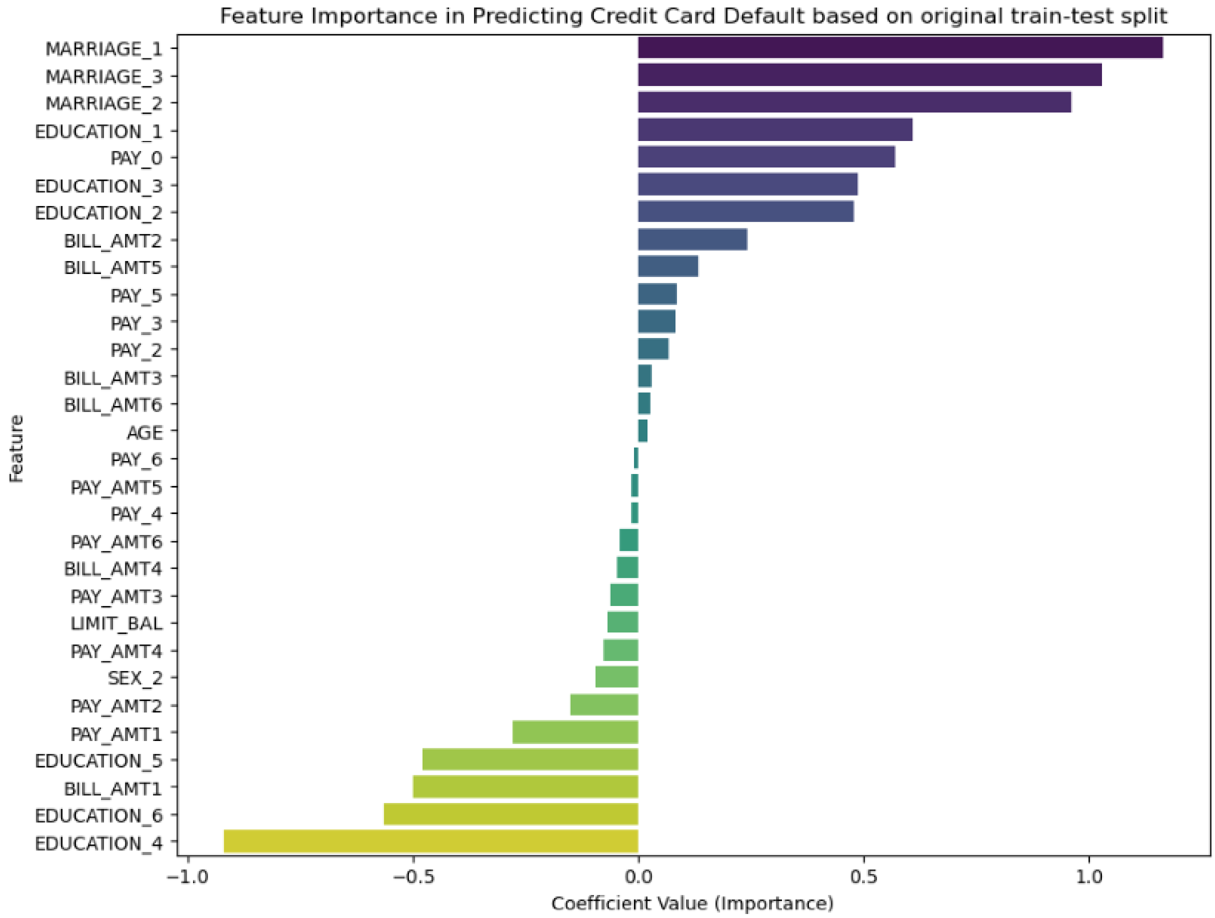
False Positive Rate:

- The false positive rate: FP/ (TN+FP) ≈ 162/ (7760+162) ≈ 2.1
- Seems like the model does not often mistakenly flag non-defaulters as defaults.

False Negative Rate:

- The false negative rate: FN/ (TP+FN) ≈ 78.5%
- This is concerning because the model is missing a significant portion of true defaulters.

While the model does well in predicting non-defaulters; it struggles with predicting true defaulters which could be a challenge in a credit risk situation when many defaulters might go undetected.



- **Positive coefficients**: Marital Status (1, 2, 3), certain Education levels (1, 2, 3), Late Repayment Status (Pay_0), and Bill Amounts (2, 5) increase the probability of default.

- **Negative coefficients:** Certain Educational levels (4, 5, 6), Bill Amount 1, Amounts paid in previous months (1, 2), and higher credit limit decrease the probability of default.

<u>Repeating with a different Train-Test sample. Make the last 20000 rows the Train Sample and the first 10000 the Test set. Highlight the differences in results if any.</u>

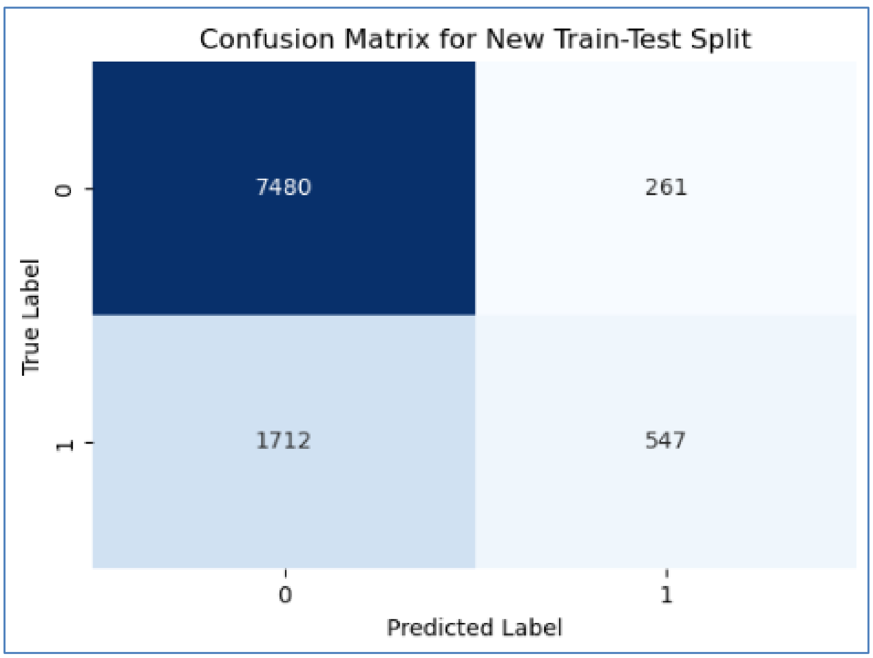
 # New Train-Test Split (Last 20,000 rows for training, first 10,000 rows for testing)

```
train_data_new = data.iloc[10000:]
test_data_new = data.iloc[:10000]
```

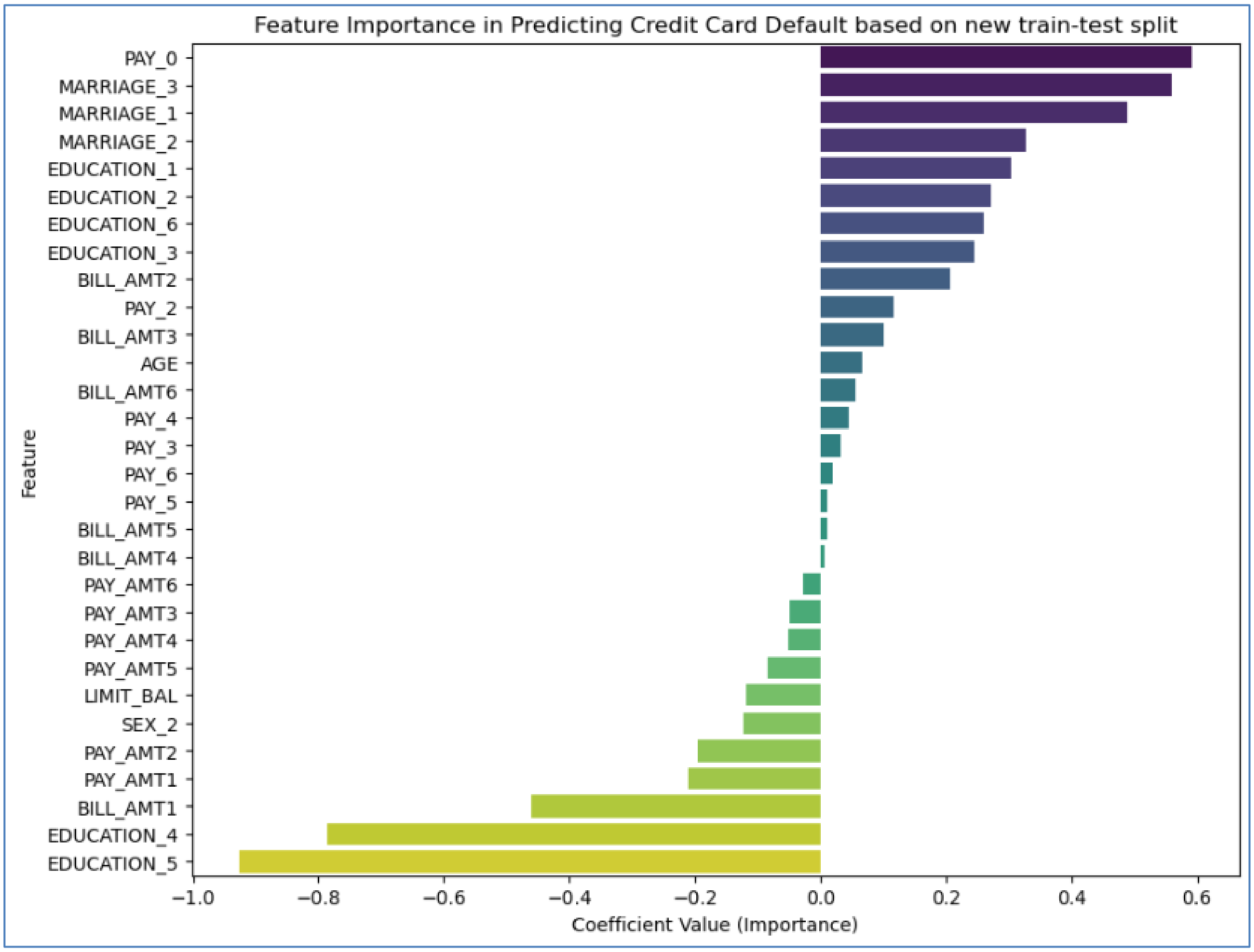
 # Separating features (X) and target (y)

```
X_train_new = train_data_new.drop(columns=['ID', 'default_next_month'])
y_train_new = train_data_new['default_next_month']
X_test_new = test_data_new.drop(columns=['ID', 'default_next_month'])
y_test_new = test_data_new['default_next_month']
```

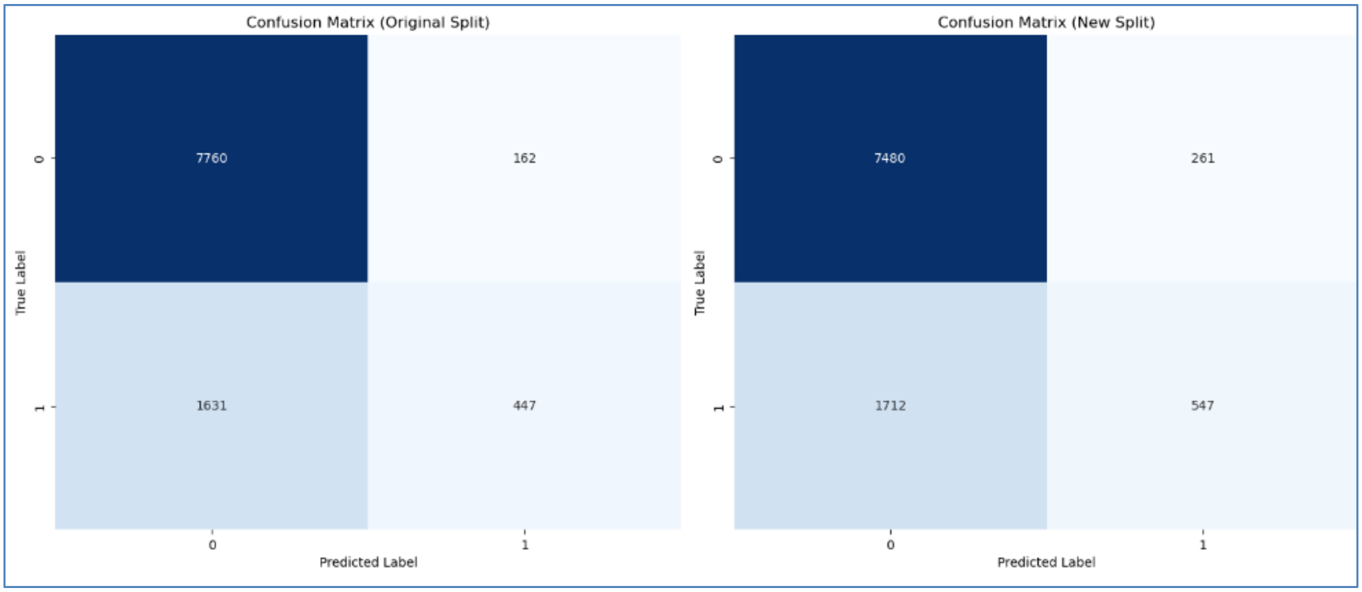
```
Accuracy for new Train-Test split: 0.8027
Classification Report for new Train-Test split:
              precision    recall  f1-score   support

           0       0.81      0.97      0.88      7741
           1       0.68      0.24      0.36      2259

    accuracy                           0.80     10000
   macro avg       0.75      0.60      0.62     10000
weighted avg       0.78      0.80      0.76     10000
```



Feature importance:



Comparing two models side-by-side

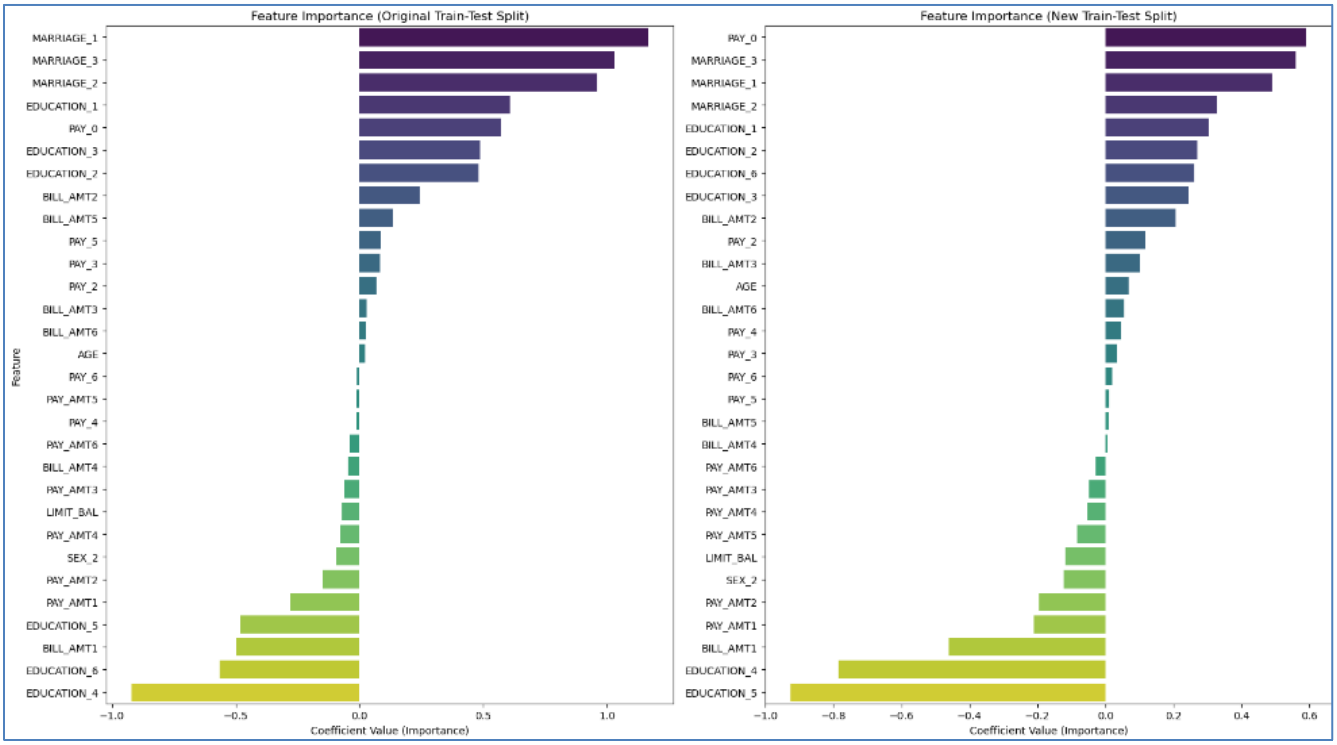


Accuracy: 0.8207

	precision	recall	f1-score	support
0	0.83	0.98	0.90	7922
1	0.73	0.22	0.33	2078
accuracy			0.82	10000
macro avg	0.78	0.60	0.61	10000
weighted avg	0.81	0.82	0.78	10000

Accuracy: 0.8027

	precision	recall	f1-score	support
0	0.81	0.97	0.88	7741
1	0.68	0.24	0.36	2259
accuracy			0.80	10000
macro avg	0.75	0.60	0.62	10000
weighted avg	0.78	0.80	0.76	10000



<u>Observation from two Feature Importance</u>

- **Coefficients that stay strong positive in both models:** Positive coefficients: PAY_0 (very strong in both splits), Marital Status (1, 2, 3), Education (1, 2, 3), BILL_AMT2 and BILL_AMT3
- **Coefficients that stay strong negative in both models:** EDUCATION_4, EDUCATION_5, LIMIT_BAL, BILL_AMT1, PAY_AMT1 to PAY_AMT5.
- **Significant change of coefficient:** EDUCATION_6 has now moved from strong negative to strong positive, and PAY_4 has moved from slight negative to slight positive.

<u>Summary of differences between two models:</u>

- **Slight Decrease in Overall Accuracy and Non-default Precision:** The new split resulted in a minor drop in the model's ability to predict non-defaults accurately.

- **Slight Improvement in Default Recall and F1-score:** The new split marginally improved the model's ability to identify defaults, as seen in the recall and F1-score for class1.

- **True Negatives:** Both models can identify non-defaults, however there is a slight decrease (from 7760 to 7480) in the new model.

- **False Positives:** The new model incorrectly flagged more non-defaulters (from 162 to 261) as defaulters which could be an issue in a practical scenario if too many clients are flagged as high-risk when they are not.

- **False Negatives:** This slight increase (from 1631 to 1712) in false negatives with the new split indicates that the model is missing more defaults which is concerning. This rate is already high, and a further increase is definite concerning.

- **True Positives:** The new split improved in identifying more defaults (from 447 to 547) which is very beneficial.

- **Overall Accuracy:** The accuracy decreased slightly (by ~1.8%) when using the new train-test split. This indicates a slight drop in overall prediction correctness for the new split.

Original Train-Test Split: 82.07%
New Train-Test Split: 80.27%

Class 0 (Non-default) Performance:

- Precision: Decreased by 2% for non-defaults, meaning the model was slightly less accurate when predicting non-defaults in the new split.
  - Original Split: 0.83
  - New Split: 0.81

- Recall: Remains very high for non-defaults in both splits, with a minor decrease in the new split.
  - Original Split: 0.98
  - New Split: 0.97

- F1-score: Decreased slightly, reflecting the small drop in precision and recall
  - Original Split: 0.90
  - New Split: 0.88

Class 1 (Default) Performance:

- Precision: Precision dropped by 5% in the new split, indicating that the model was less precise when predicting defaults.
  - Original Split: 0.73
  - New Split: 0.68

- Recall: Improved slightly for defaults, meaning the new split helped the model identify a marginally higher percentage of actual defaults (24% vs. 22%).
  - Original Split: 0.22
  - New Split: 0.24

- F1-score: Improved slightly for defaults, suggesting a minor improvement in the model's ability to balance precision and recall for defaults.
    - Original Split: 0.33
    - New Split: 0.36

Averages:

- Macro Avg: Quite similar across splits, though the F1-score for the new split is marginally higher, indicating a small improvement in balance across classes.
    - Original Split: 0.60 recall, 0.61 F1-score
    - New Split: 0.60 recall, 0.62 F1-score

- Weighted Avg: Decreased slightly in the new split, reflecting the overall drop in performance for non-default predictions.
    - Original Split: 0.82 recall, 0.78 F1-score
    - New Split: 0.80 recall, 0.76 F1-score

My suggested approach: Overall, the dataset is imbalanced with more non-defaults than defaults (which is expected), therefore the model tends to favor the majority class.

I have used Class Weight and Hyper Parameter Tuning to see if those help improve the performance.

 # Logistic Regression with Class Weight Balancing

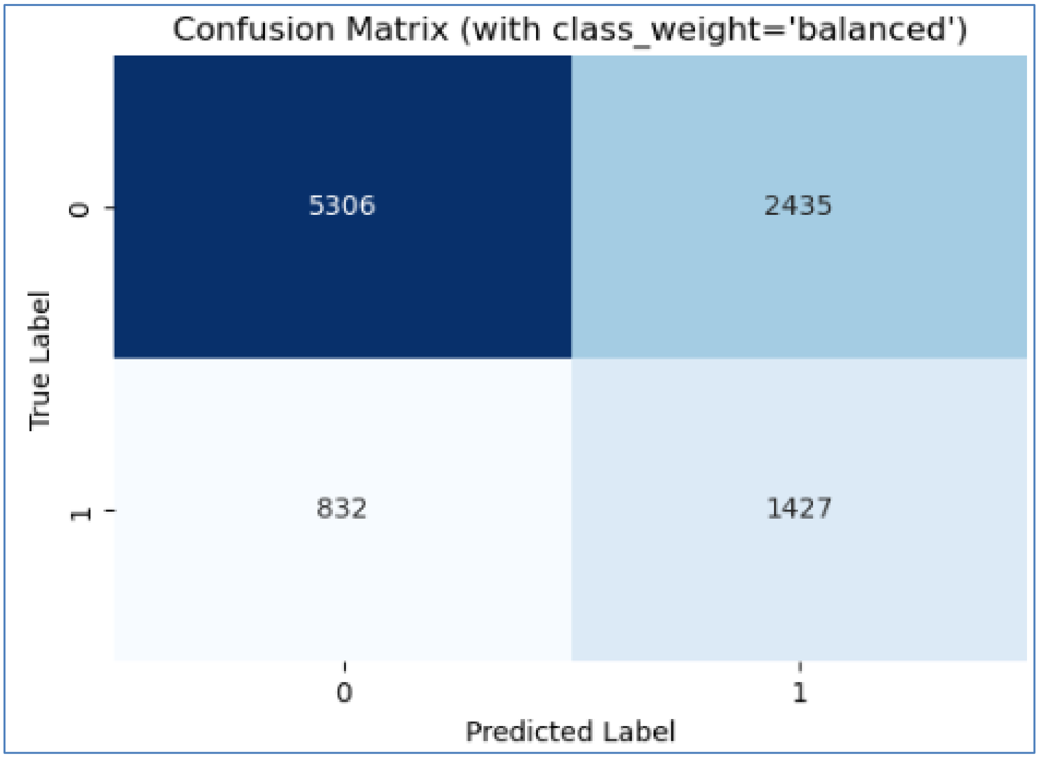
logit_model_weighted = LogisticRegression(max_iter=1000, random_state=42, class_weight='balanced')
logit_model_weighted.fit(X_train_new, y_train_new)

 # Making Predictions and Evaluating the Model

y_pred_weighted = logit_model_weighted.predict(X_test_new)
accuracy_weighted = accuracy_score(y_test_new, y_pred_weighted)
classification_rep_weighted = classification_report(y_test_new, y_pred_weighted)
conf_matrix_weighted = confusion_matrix(y_test_new, y_pred_weighted)

Result:

```
Accuracy (with class_weight='balanced'): 0.6733
Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.69      0.76      7741
           1       0.37      0.63      0.47      2259

    accuracy                           0.67     10000
   macro avg       0.62      0.66      0.62     10000
weighted avg       0.75      0.67      0.70     10000
```



- **True Positive Rate (Recall for Defaults):** The true positive rate (recall for class 1) improved with class_weight='balanced'. Out of all actual defaults (1,427 + 832 = 2,259), the model successfully identified 1,427, which is approximately 63.2%. This is a significant improvement over previous results without balanced class weights.

- False Positive Rate: The number of false positives increased to 2,435, indicating that the model is now predicting more non-defaulters as defaulters. This increase in false positives is a trade-off for achieving better recall for defaults.

- False Negatives: The false negatives have decreased compared to the unweighted model. By reducing the number of missed defaults, the model is better suited for applications where identifying potential defaulters is critical.

Practical Implications: In a credit risk scenario, this model is now more sensitive to defaults, catching more high-risk cases (higher TP). However, the increase in false positives (FP) means more clients are incorrectly flagged as high-risk, which might lead to additional scrutiny or intervention for these clients. This balance might be acceptable if the business prioritizes reducing missed defaults (FN) and is willing to accept more false positives as a trade-off.