In [1]:
```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import numpy as np
```

In [2]:
```
Credit_df = pd.read_csv('Credit_Data.csv')
```

In [3]:
```
Credit_df
```

Out[3]:

	ID	Income	Limit	Rating	Cards	Age	Education	Gender	Student	Married	Ethnicity	Balan
0	1	14.891	3606	283	2	34	11	Male	No	Yes	Caucasian	3
1	2	106.025	6645	483	3	82	15	Female	Yes	Yes	Asian	9
2	3	104.593	7075	514	4	71	11	Male	No	No	Asian	5
3	4	148.924	9504	681	3	36	11	Female	No	No	Asian	9
4	5	55.882	4897	357	2	68	16	Male	No	Yes	Caucasian	3
...	...	...	...	...	...	...	...	...	...	...	...	
395	396	12.096	4100	307	3	32	13	Male	No	Yes	Caucasian	5
396	397	13.364	3838	296	5	65	17	Male	No	No	African American	4
397	398	57.872	4171	321	5	67	12	Female	No	Yes	Caucasian	1
398	399	37.728	2525	192	1	44	13	Male	No	Yes	Caucasian	
399	400	18.701	5524	415	5	64	7	Female	No	No	Asian	9

400 rows × 12 columns

EDA

In [4]:
```
#Check for missing variables
Credit_df.isnull().sum()
```

Out[4]:

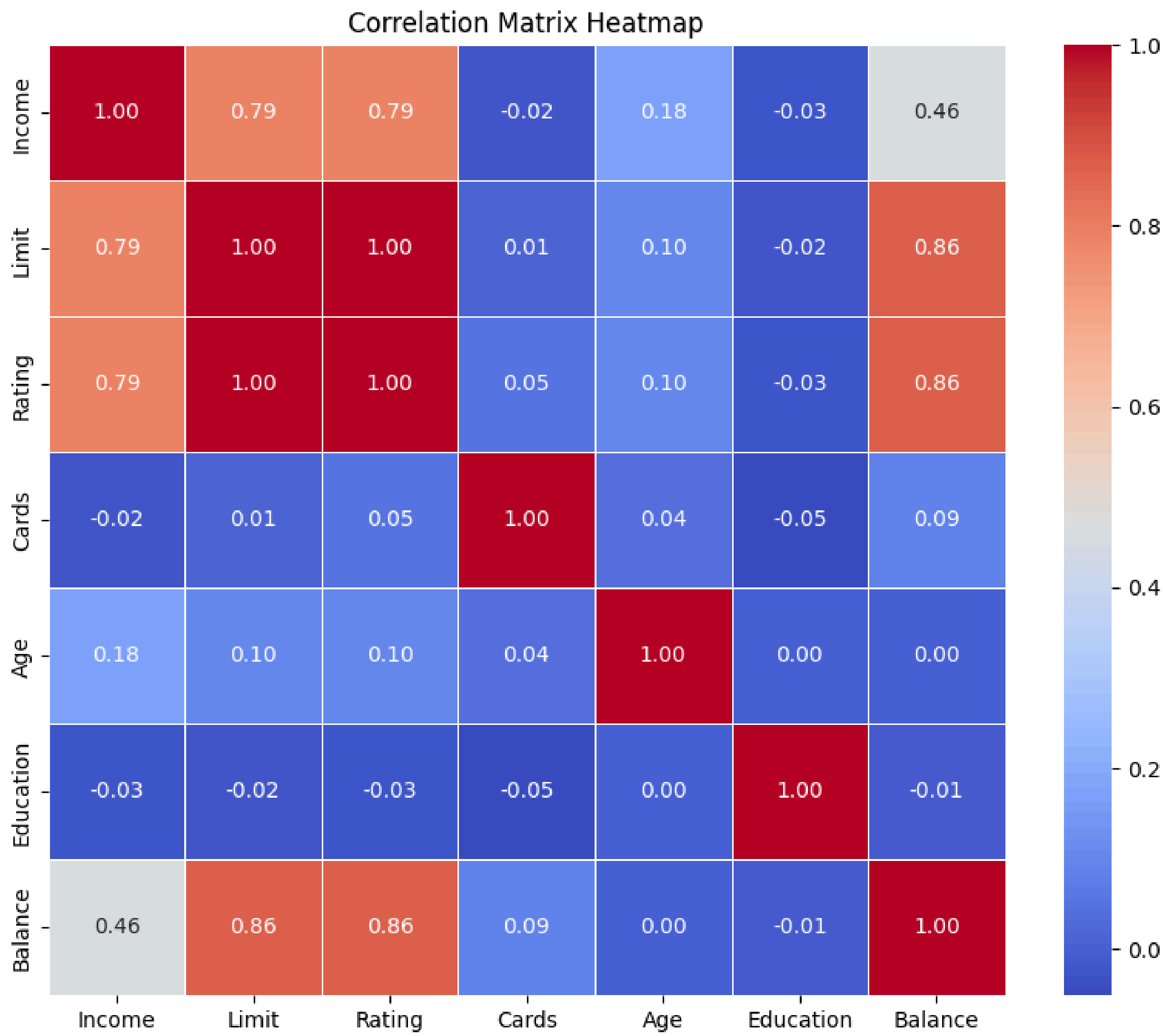
	0
ID	0
Income	0
Limit	0
Rating	0
Cards	0
Age	0
Education	0
Gender	0
Student	0
Married	0
Ethnicity	0
Balance	0

dtype: int64

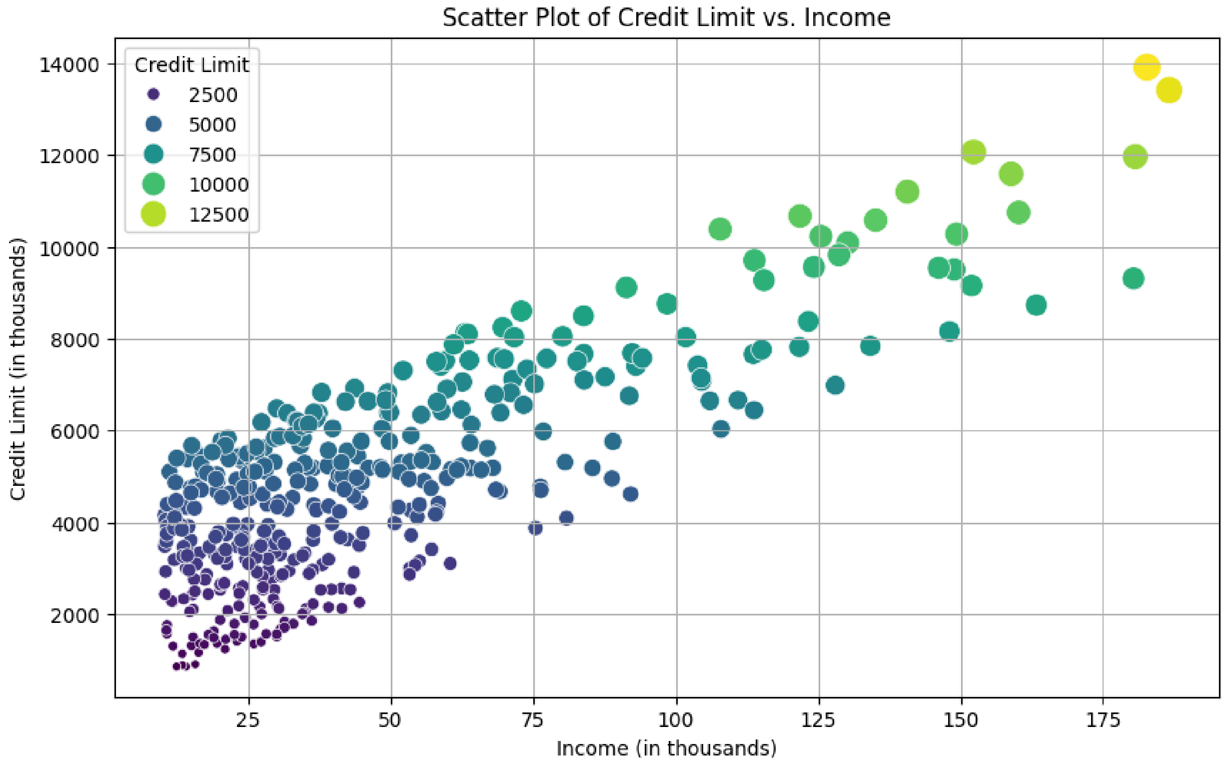
We see that there are no missing variables.

In [5]:
```
#Checking out correlation_matrix
correlation_matrix = Credit_df[['Income','Limit','Rating','Cards','Age','Education','E
```

In [6]:
```
#Creating the heatmap
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt='.2f', linewidths=0.5
plt.title('Correlation Matrix Heatmap')
plt.show()
```



In [7]:
```
plt.figure(figsize=(10, 6))
sns.scatterplot(data = Credit_df, x='Income', y='Limit', hue='Limit', palette='viridis
plt.title('Scatter Plot of Credit Limit vs. Income')
plt.xlabel('Income (in thousands)')
plt.ylabel('Credit Limit (in thousands)')
plt.legend(title='Credit Limit')
plt.grid(True)
plt.show()
```



Above as expected from our heatmap we can see within the scatter plot the relationship between income and credit limit. As income increases so does the indivduals credit limit.

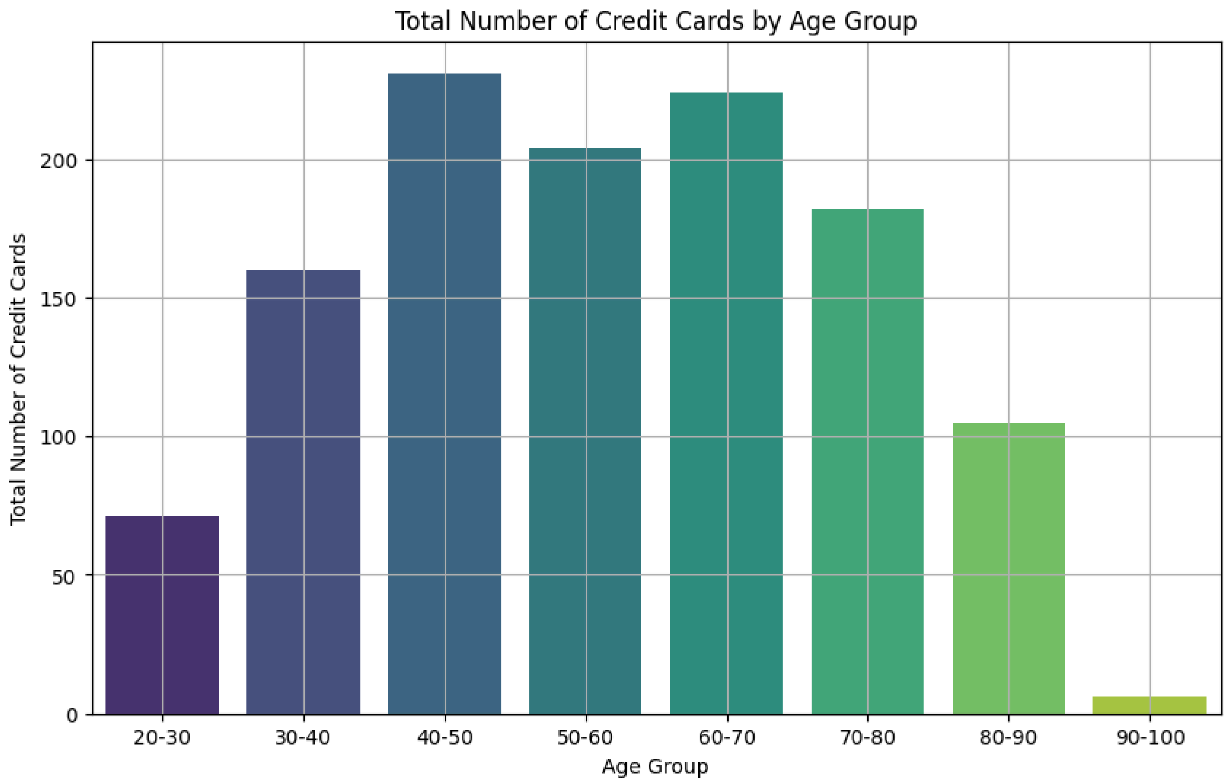
In [8]:
```
# Define age bins
bins = [20, 30, 40, 50, 60, 70, 80, 90, 100]
labels = ['20-30', '30-40', '40-50', '50-60', '60-70', '70-80', '80-90', '90-100']
Credit_df['Age Group'] = pd.cut(Credit_df['Age'], bins=bins, labels=labels, right=Fals
```

In [9]:
```
# Aggregate data
age_group_summary = Credit_df.groupby('Age Group')['Cards'].sum().reset_index()
```

```
<ipython-input-9-928c967493c8>:2: FutureWarning: The default of observed=False is dep
recated and will be changed to True in a future version of pandas. Pass observed=Fals
e to retain current behavior or observed=True to adopt the future default and silence
this warning.
  age_group_summary = Credit_df.groupby('Age Group')['Cards'].sum().reset_index()
```

In [10]:
```
# Plotting
plt.figure(figsize=(10, 6))
sns.barplot(x='Age Group', y='Cards', data=age_group_summary, palette='viridis')
plt.title('Total Number of Credit Cards by Age Group')
plt.xlabel('Age Group')
plt.ylabel('Total Number of Credit Cards')
plt.grid(True)
plt.show()
```

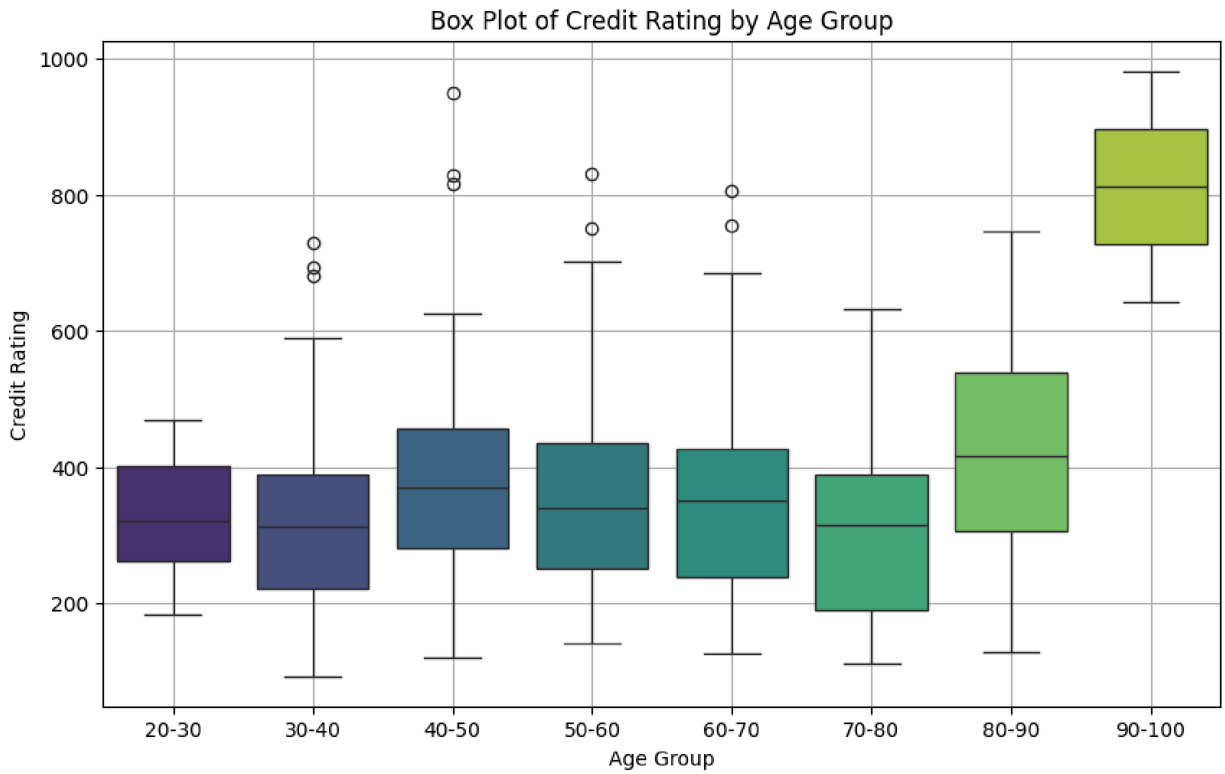
```
<ipython-input-10-77ca609a7033>:3: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.
0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.barplot(x='Age Group', y='Cards', data=age_group_summary, palette='viridis')
```



In the barplot we can observe that age groups 40-50, 50-60, 60-70, & 70-80 have the higher avg amount of credit cards owned.

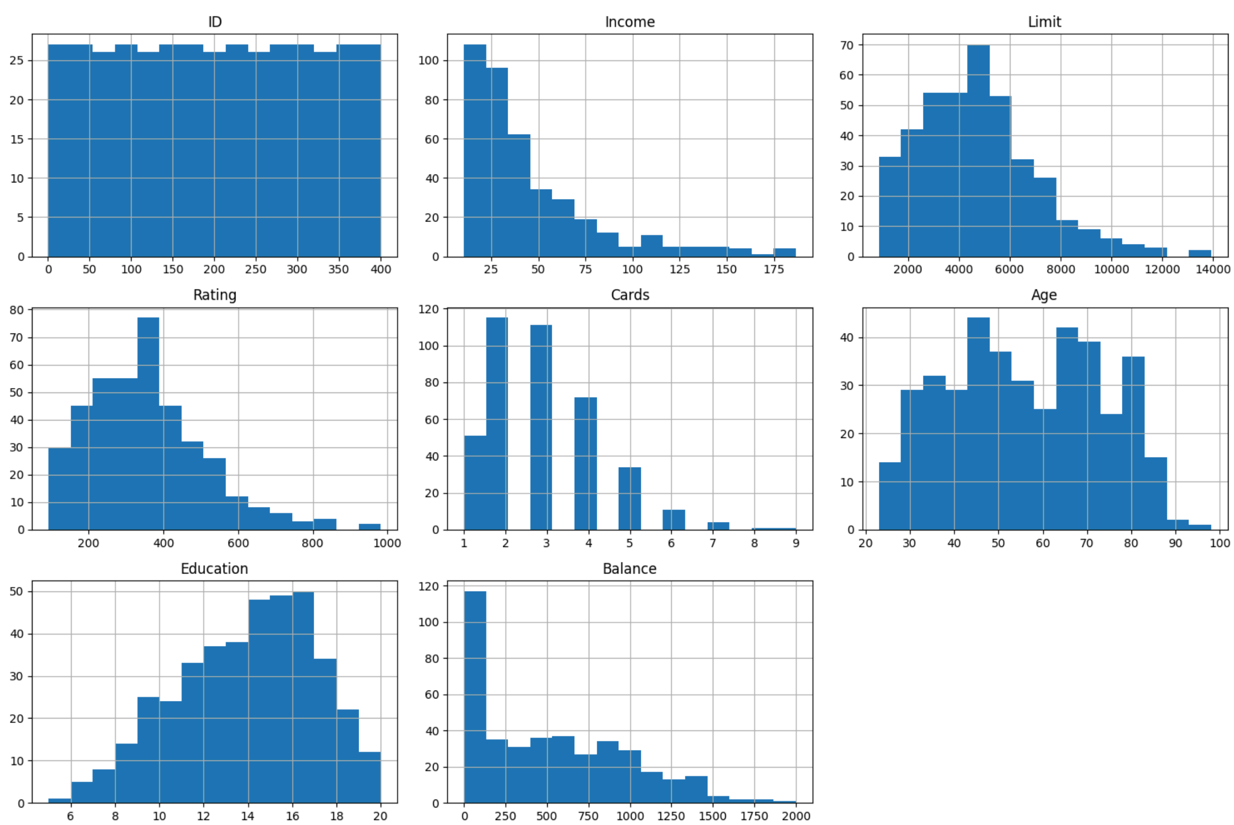
In [11]:
```
plt.figure(figsize=(10, 6))
sns.boxplot(x='Age Group', y='Rating', data=Credit_df, palette='viridis')
plt.title('Box Plot of Credit Rating by Age Group')
plt.xlabel('Age Group')
plt.ylabel('Credit Rating')
plt.grid(True)
plt.show()
```

```
<ipython-input-11-82fd5a160a45>:2: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.
0. Assign the `x` variable to `hue` and set `legend=False` for the same effect.

  sns.boxplot(x='Age Group', y='Rating', data=Credit_df, palette='viridis')
```



In [12]:

```
# Plot histograms for each numerical column
numerical_cols = Credit_df.select_dtypes(include=[np.number])
numerical_cols.hist(bins=15, figsize=(15, 10), layout=(3, 3))
plt.tight_layout()
plt.show()
```



In [13]:

```
skewness = numerical_cols.skew()
print(skewness)
```

```
ID           0.000000
Income       1.742117
Limit        0.837493
Rating       0.865394
Cards        0.791928
Age          0.011496
Education   -0.329212
Balance      0.584595
dtype: float64
```

Checking for Outliers

In [16]:
```
# Only select numerical columns
numerical_cols = Credit_df.select_dtypes(include=[np.number])

# Calculate IQR for each column
Q1 = numerical_cols.quantile(0.25)
Q3 = numerical_cols.quantile(0.75)
IQR = Q3 - Q1

# Find outliers: data points that are below Q1 - 1.5*IQR or above Q3 + 1.5*IQR
outliers_iqr = (numerical_cols < (Q1 - 1.5 * IQR)) | (numerical_cols > (Q3 + 1.5 * IQR

# Count the number of outliers in each column
outlier_counts = outliers_iqr.sum()
print(outlier_counts)
```

```
ID            0
Income       29
Limit        13
Rating       11
Cards         2
Age           0
Education     0
Balance       0
dtype: int64
```

We're capping outliers to reduce their impact on the model while preserving the data. Outliers, especially in skewed features like Income, Limit, and Rating, can distort the model's predictions. By capping the extreme values at the 5th and 95th percentiles, we ensure that these values stay within a reasonable range without entirely removing important data. This helps improve model performance and stability.

In [17]:
```
columns_to_cap = ['Income', 'Limit', 'Rating']
```

In [18]:
```
# Define the capping thresholds (5th and 95th percentiles)
for col in columns_to_cap:
    lower_bound = Credit_df[col].quantile(0.05)  # 5th percentile
    upper_bound = Credit_df[col].quantile(0.95)  # 95th percentile

    # Cap values below 5th percentile and above 95th percentile
    Credit_df[col] = Credit_df[col].clip(lower=lower_bound, upper=upper_bound)

# Verify that the outliers have been capped
print(Credit_df[columns_to_cap].describe())
```

```
          Income        Limit      Rating
count  400.000000   400.000000  400.000000
mean    43.908875  4672.327500  349.995000
std     31.075357  2086.354111  138.963833
min     12.066150  1483.150000  138.000000
25%     21.007250  3088.000000  247.250000
50%     33.115500  4622.500000  344.000000
75%     57.470750  5872.750000  437.250000
max    124.349500  9161.800000  642.700000
```

In [20]:

```python
# Only select numerical columns
numerical_cols = Credit_df.select_dtypes(include=[np.number])

# Calculate IQR for each column
Q1 = numerical_cols.quantile(0.25)
Q3 = numerical_cols.quantile(0.75)
IQR = Q3 - Q1

# Find outliers: data points that are below Q1 - 1.5*IQR or above Q3 + 1.5*IQR
outliers_iqr = (numerical_cols < (Q1 - 1.5 * IQR)) | (numerical_cols > (Q3 + 1.5 * IQF

# Count the number of outliers in each column
outlier_counts = outliers_iqr.sum()
print(outlier_counts)
```

```
ID            0
Income       29
Limit         0
Rating        0
Cards         2
Age           0
Education     0
Balance       0
dtype: int64
```

It seems that Income still shows 29 outliers because the capping process only adjusts values that exceed the 5th and 95th percentiles, but doesn't remove the rows entirely. To handle this we will adjust the capping thresholds to the 1st and 99th percentiles instead of the 5th and 95th percentiles.

In [21]:

```python
columns_to_cap = ['Income', 'Limit', 'Rating']
```

In [22]:

```python
# Adjust the capping thresholds to 1st and 99th percentiles
for col in columns_to_cap:
    lower_bound = Credit_df[col].quantile(0.01)  # 1st percentile
    upper_bound = Credit_df[col].quantile(0.99)  # 99th percentile

    # Cap values below 1st percentile and above 99th percentile
    Credit_df[col] = Credit_df[col].clip(lower=lower_bound, upper=upper_bound)

# Recalculate outliers using IQR after capping
Q1 = Credit_df['Income'].quantile(0.25)
Q3 = Credit_df['Income'].quantile(0.75)
IQR = Q3 - Q1

# Find outliers for 'Income' after new capping
outliers_iqr = (Credit_df['Income'] < (Q1 - 1.5 * IQR)) | (Credit_df['Income'] > (Q3 +
```

```
# Count outliers in 'Income' after adjusting capping
outliers_count = outliers_iqr.sum()
```

In [24]:
```
outliers_count
```

Out[24]:
```
29
```

Even after implementing the capping process it appears that the income column still has some outliers. Moving forward I would like to implement the model phase as is with the column being capped. Reasoning is because if the extreme Income values are genuine and represent a segment of the population, they could provide valuable insights into high-income behavior and its relationship with the credit card balance.

Feature Engineering Applying One-Hot Encoding

In [32]:
```
# Use pd.get_dummies to convert categorical variables to dummy/indicator variables
X_encoded = pd.get_dummies(X, drop_first=True)  # drop_first=True to avoid multicollir
```

Splitting The Data into Test and Train

In [35]:
```
X_train_encoded = pd.get_dummies(X_train, drop_first=True)
X_test_encoded = pd.get_dummies(X_test, drop_first=True)

# Ensure that both training and test sets have the same columns
X_train_encoded, X_test_encoded = X_train_encoded.align(X_test_encoded, join='left', a
```

In [36]:
```
from sklearn.linear_model import LinearRegression

model = LinearRegression()
model.fit(X_train_encoded, y_train)
```

Out[36]:
```
▾ LinearRegression
LinearRegression()
```

Model Evaluation

In [39]:
```
from sklearn.metrics import mean_squared_error, r2_score

# Predict on the test set
y_pred = model.predict(X_test_encoded)

# Calculate performance metrics
mse = mean_squared_error(y_test, y_pred)
r2 = r2_score(y_test, y_pred)

print(f'Mean Squared Error: {mse}')
print(f'R² Score: {r2}')
```

```
Mean Squared Error: 33789.151748188175
R² Score: 0.7977588806694244
```

In [40]:
```
mean_balance = Credit_df['Balance'].mean()
print(f'Mean Balance: {mean_balance}')
```

```
Mean Balance: 520.015
```

Our mean Squared Error shows us that there is a high relative error because the avg mean balance is $500 now if the avg balance was much higher than our m2 would be accepted instead it is not. On the other hand our R2 is not too bad as the number is closer to 1. Overall the high MSE relative to the mean balance suggests that the current model may not be performing well and needs improvement. Therefore we will explore another model: Random Forest

Random Forest Model/Hyper Tuning

In [41]:
```
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import GridSearchCV

# Define the model
rf_model = RandomForestRegressor()

# Define the parameter grid
param_grid = {
    'n_estimators': [50, 100, 200],
    'max_depth': [10, 20, 30],
    'min_samples_split': [2, 5, 10]
}

# Initialize GridSearchCV
grid_search = GridSearchCV(estimator=rf_model, param_grid=param_grid, cv=5, scoring='r

# Fit GridSearchCV
grid_search.fit(X_train_encoded, y_train)
```

Out[41]:
```
▸ GridSearchCV
▸ estimator: RandomForestRegressor
  ▸ RandomForestRegressor
```

In [42]:
```
# Best parameters and best score
print("Best Parameters:", grid_search.best_params_)
print("Best Score:", -grid_search.best_score_)
```

```
Best Parameters: {'max_depth': 30, 'min_samples_split': 5, 'n_estimators': 50}
Best Score: 20645.428226954064
```

In [43]:
```
# Fit the model with the best hyperparameters
best_rf_model = grid_search.best_estimator_

# Predict and evaluate
y_pred_rf = best_rf_model.predict(X_test_encoded)
mse_rf = mean_squared_error(y_test, y_pred_rf)
print(f'Optimized Random Forest MSE: {mse_rf}')
```

```
Optimized Random Forest MSE: 24106.967578796543
```

In [44]:
```
from sklearn.metrics import r2_score

# Predict using the best model
```

```
y_pred_rf = best_rf_model.predict(X_test_encoded)

# Calculate R² score
r2_rf = r2_score(y_test, y_pred_rf)
print(f'Optimized Random Forest R² Score: {r2_rf}')
```

```
Optimized Random Forest R² Score: 0.8557104912506974
```

Interpretation:

MSE: 24,106.97 is an improvement over the linear regression model's MSE of 33,789.15. This lower MSE indicates that the Random Forest model is better at predicting the credit card balance with fewer prediction errors.

R^2 Score: 0.8557 means that approximately 85.57% of the variance in the credit card balance is explained by the model. This is a high R^2 score, suggesting that the Random Forest model explains a large portion of the variability in the target variable.