Loading Data

First step is to load the dataset

In [121…
```
# Importing necessary libraries
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
```

In [122…
```
# Importing data
original_df = pd.read_excel("Bank_loan_data.xlsx", sheet_name=1)

# Making a copy
df = original_df.copy()
```

In [123…
```
# Head of dataframe
df.head()
```

Out[123]:

	ID	Age	Gender	Experience	Income	ZIP Code	Family	CCAvg	Education	Mortgage	Home Ownership	Personal Loan	Securities Account	CD Account	Online
0	1	25	NaN	1	49.0	91107	4	1.6	1	0	NaN	0	1	0	0.0
1	2	45	NaN	19	34.0	90089	3	1.5	1	0	NaN	0	1	0	0.0
2	3	39	NaN	15	11.0	94720	1	1.0	1	0	NaN	0	0	0	0.0
3	4	35	NaN	9	100.0	94112	1	2.7	2	0	NaN	0	0	0	0.0
4	5	35	NaN	8	45.0	91330	4	1.0	2	0	NaN	0	0	0	0.0

In [124…
```
# Shape of DataFrame
df.shape
```

Out[124]: (5000, 16)

Checking the features of DataFrame

In [125…
```
# Checking Null Values
print(df.isna().sum())
```

```
ID                      0
Age                     0
Gender               4520
Experience              0
Income                 67
ZIP Code                0
Family                  0
CCAvg                   0
Education               0
Mortgage                0
Home Ownership       4520
Personal Loan           0
Securities Account      0
CD Account              0
Online                 40
CreditCard              0
dtype: int64
```

There are a lot of missing datas, perhaps it is because of the dulication of data? Lets try to see the duplicate data

In [305…
```
column_list = df.drop(columns=["ID"]).columns
column_list
```

Out[305]:
```
Index(['Age', 'Experience', 'Income', 'ZIP Code', 'Family', 'CCAvg',
       'Education', 'Mortgage', 'Home Ownership', 'Personal Loan',
       'Securities Account', 'CD Account', 'Online', 'CreditCard',
       'Has_Mortgage'],
      dtype='object')
```

In [127…
```
# Checking for duplicate values
df[column_list].duplicated().sum()
```

Out[127]:
```
0
```

There are no such duplicated values, so casually lets do some feature engineering

In [128…
```
# Dropping the gender column because it is missing and we cant impute the data
df.drop(columns=["Gender"], inplace=True)
```

In [129…
```
# Looking at home ownership
df["Home Ownership"].value_counts()
```

Out[129]:
```
Home Mortage    219
Home Owner      138
Rent            123
Name: Home Ownership, dtype: int64
```

Since home ownerhip could be an useful column, lets try to impute the missing value with new value called "None" to preserve the model accuracy for now

In [130…
```
# Filling na values of Home Ownership with new value "None"
df["Home Ownership"].fillna("None", inplace=True)
```

In [131…
```
# Checking value counts
df["Home Ownership"].value_counts()
```

Out[131]:
```
None           4520
Home Mortage    219
Home Owner      138
Rent            123
Name: Home Ownership, dtype: int64
```

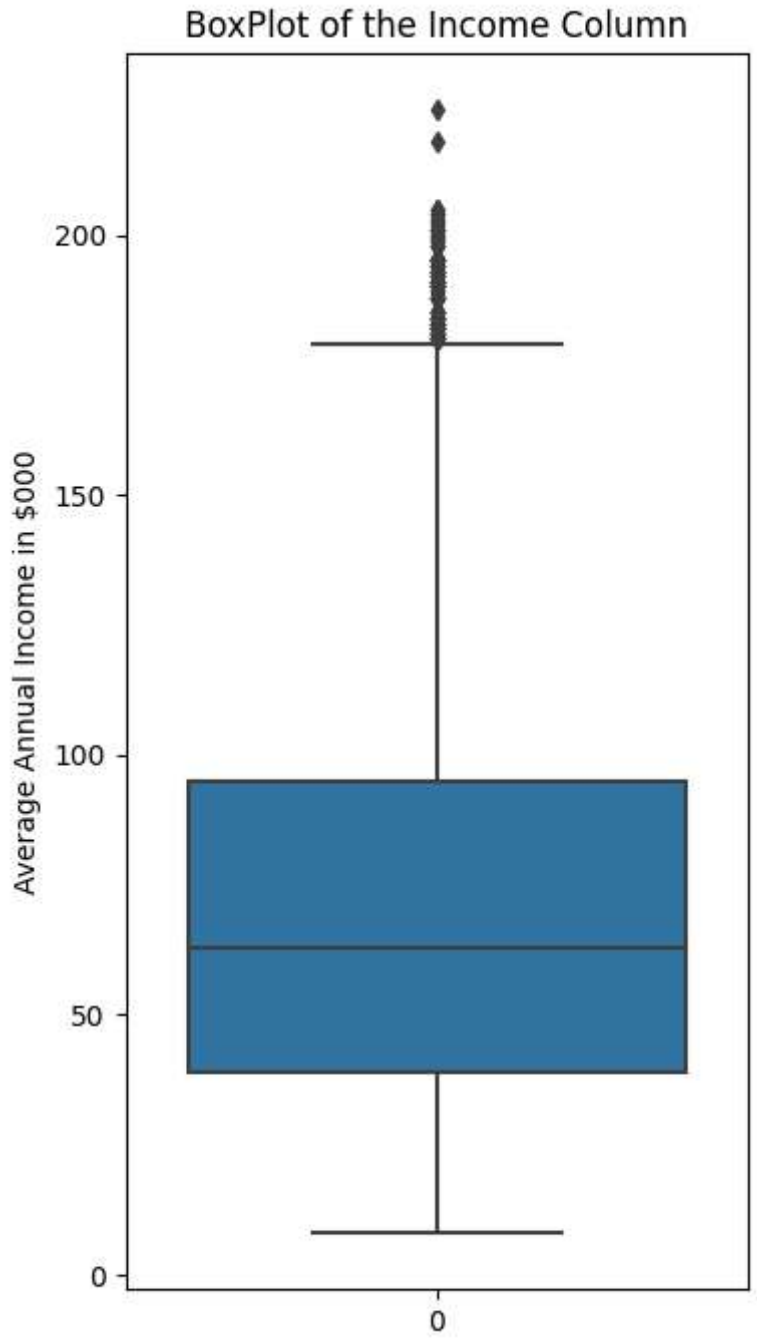
Lets look at the missing value of income groups

In [132…
```
# Summary Statistics of Income column
df["Income"].describe().round(2)
```

Out[132]:
```
count    4933.00
mean       72.76
std        45.43
min         8.00
25%        39.00
50%        63.00
75%        95.00
max       224.00
Name: Income, dtype: float64
```

Lets check for outliers before imputing data

In [133…
```
# Setting size of plot
plt.figure(figsize=(4, 8))

# Plotting a box plot of income column
sns.boxplot(data=df["Income"])

plt.title("BoxPlot of the Income Column")
plt.ylabel("Average Annual Income in $000")
plt.show()
```


BoxPlot of the Income Column
Average Annual Income in $000
200
150
100
50
0
0

There are a significant number of outliers in the dataset, so lets count them to get an idea

In [134…
```
# Determining 1st and 3rd Quartile and interquartile range
q1 = df["Income"].quantile(0.25)
q3 = df["Income"].quantile(0.75)
iqr = q3 - q1
threshold = q3 + 1.5 * iqr
num_outliers = len(df[df["Income"] > threshold])
num_outliers
```

Out[134]: 158

There are 158 outliers, so it is better to impute with median in this case

In [135…
```
# Calculating the median of Income Column
income_median = df["Income"].median()

# Filling null value with the median
df["Income"].fillna(income_median, inplace=True)
```

In [136…
```
display(df["Income"].isna().sum())
display(df["Income"].describe().round(2))
```

```
0
count    5000.00
mean       72.63
std        45.13
min         8.00
25%        39.00
50%        63.00
75%        94.00
max       224.00
Name: Income, dtype: float64
```

Great, now we can move onto fixing our another missing column Online Banking, Since this has relatively fewer amount of missing values, we can impute it with the mode of the column without many adverse effects, before that lets fix the datatype to categorical

In [137…
```
# Changing the datatype to category
df["Online"] = df["Online"].astype("category")
```

In [138…
```
# Extracting the first mode
online_mode = df["Online"].mode().values[0]
online_mode
```

Out[138]:
```
1.0
```

In [139…
```
# Filling the empty value with the extracted mode
df["Online"].fillna(online_mode, inplace=True)
```

In [140…
```
display(df["Online"].isna().sum())
display(df["Online"].describe())
df["Online"].value_counts()
```

```
0
count     5000.0
unique       2.0
top          1.0
freq      3001.0
Name: Online, dtype: float64
```

Out[140]:
```
1.0    3001
0.0    1999
Name: Online, dtype: int64
```

Lets check the new details of our dataframe

In [141…
```
# Checking the null values again
df.isna().sum()
```

Out[141]:
```
ID                    0
Age                   0
Experience            0
Income                0
ZIP Code              0
Family                0
CCAvg                 0
Education             0
Mortgage              0
Home Ownership        0
Personal Loan         0
Securities Account    0
CD Account            0
Online                0
CreditCard            0
dtype: int64
```

In [142…

```
# Summary Statistics of numerical column rounded to 2 decimals
df.describe(include=np.number).round(2).T
```

Out[142]:

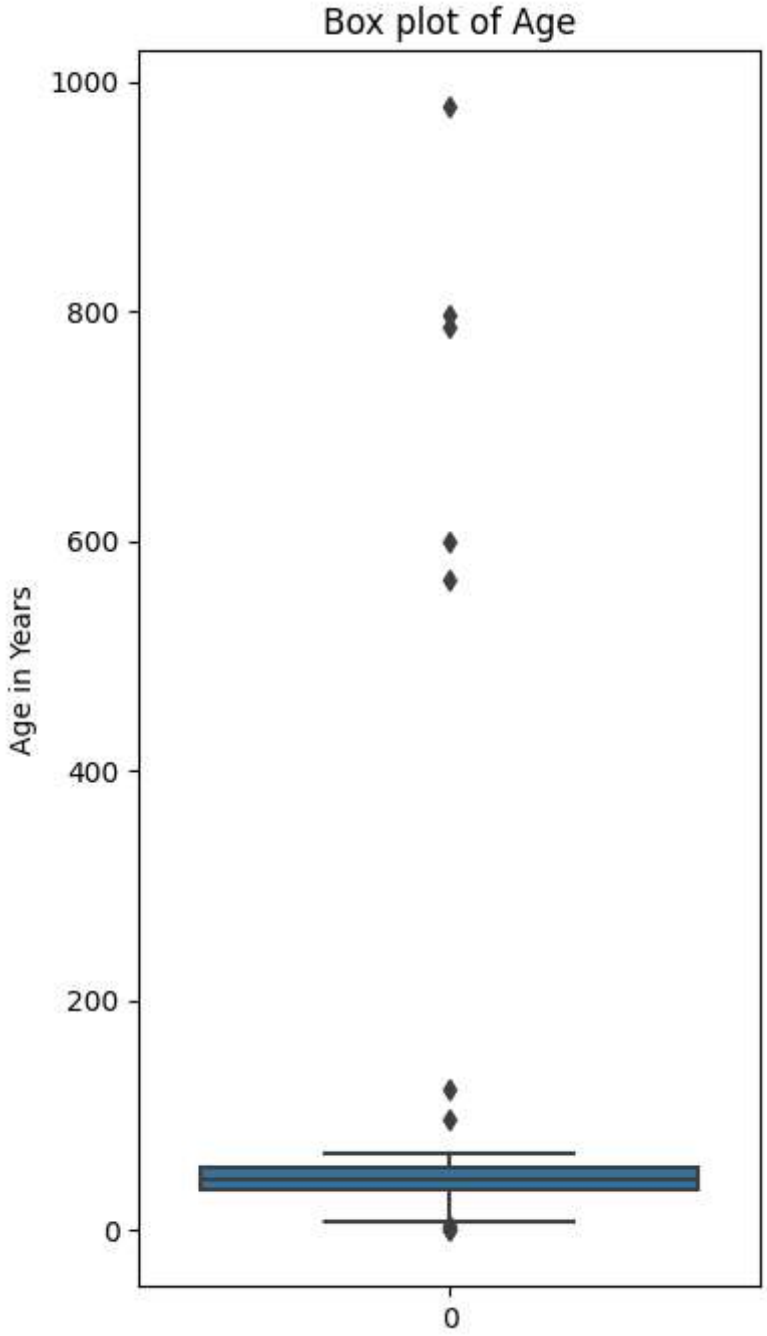
	count	mean	std	min	25%	50%	75%	max
ID	5000.0	2500.50	1443.52	1.0	1250.75	2500.5	3750.25	5000.0
Age	5000.0	46.01	25.44	0.0	35.00	45.0	55.00	978.0
Experience	5000.0	20.10	11.47	-3.0	10.00	20.0	30.00	43.0
Income	5000.0	72.63	45.13	8.0	39.00	63.0	94.00	224.0
ZIP Code	5000.0	93152.50	2121.85	9307.0	91911.00	93437.0	94608.00	96651.0
Family	5000.0	2.40	1.15	1.0	1.00	2.0	3.00	4.0
CCAvg	5000.0	1.94	1.75	0.0	0.70	1.5	2.50	10.0
Education	5000.0	1.88	0.84	1.0	1.00	2.0	3.00	3.0
Mortgage	5000.0	56.50	101.71	0.0	0.00	0.0	101.00	635.0
Securities Account	5000.0	0.10	0.31	0.0	0.00	0.0	0.00	1.0
CD Account	5000.0	0.06	0.24	0.0	0.00	0.0	0.00	1.0
CreditCard	5000.0	0.29	0.46	0.0	0.00	0.0	1.00	1.0

There are a lot of problems with the data, we have to clean it up one column by column

Age Column

In [143…

```
# Setting figure size
plt.figure(figsize=(4, 8))

# Plotting the Age Column
sns.boxplot(df["Age"])

plt.ylabel("Age in Years")
plt.title("Box plot of Age")
plt.show()
```


Box plot of Age
1000
800
600
400
200
0
Age in Years
0

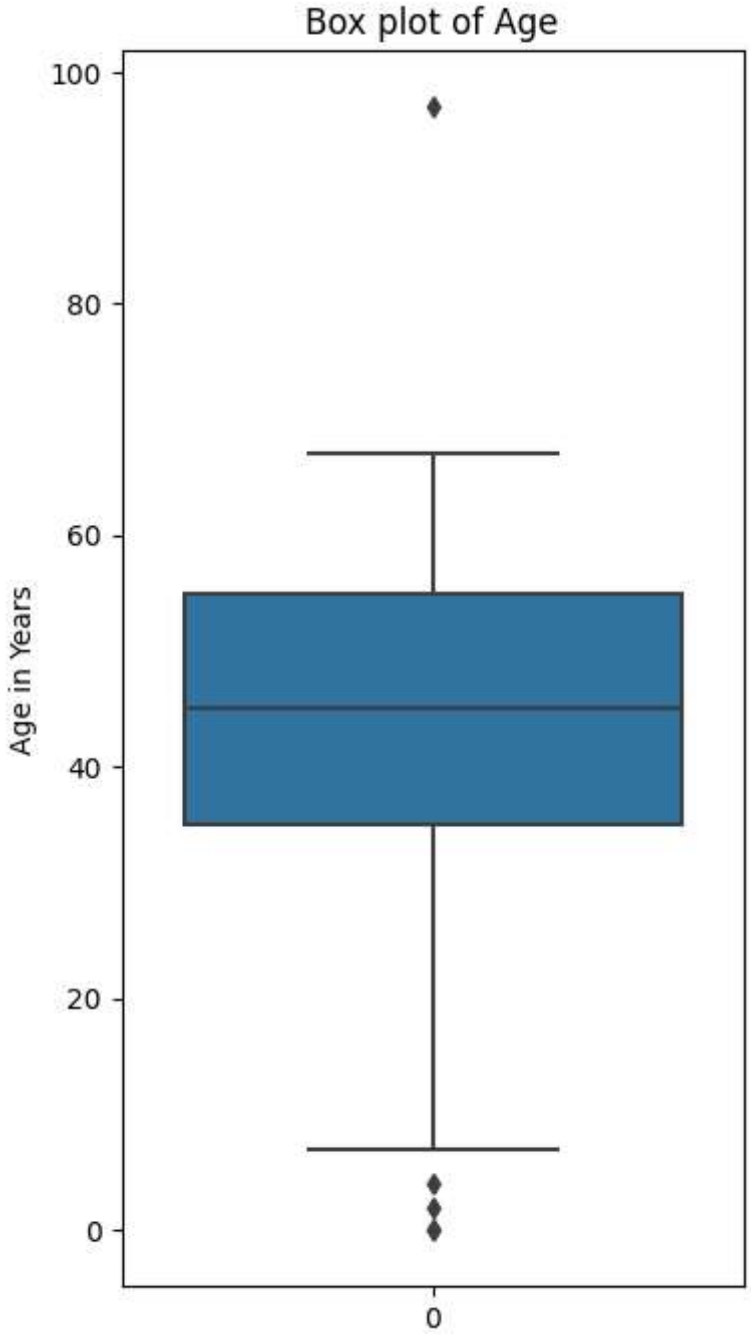
There are 7 outliers which is just impossible, so lets remove those entries

In [306…

```
# Choosing ages above 100 years to drop
rows_to_drop = df[df["Age"] >= 100]

# Dropping those rows
df.drop(rows_to_drop.index, inplace=True)
```

In [145…

```
# Figure Size
plt.figure(figsize=(4, 8))

# Plotting the boxplot of Age column
sns.boxplot(df["Age"])

plt.ylabel("Age in Years")
plt.title("Box plot of Age")
plt.show()
```


Box plot of Age
Age in Years
100
80
60
40
20
0
0

In [146…
```
# Summary Statistics of Age Column
df["Age"].describe().round(2)
```

Out[146]:
```
count    4994.00
mean       45.29
std        11.58
min         0.00
25%        35.00
50%        45.00
75%        55.00
max        97.00
Name: Age, dtype: float64
```

This looks relatively fine, so fixed the age column

Experience Column

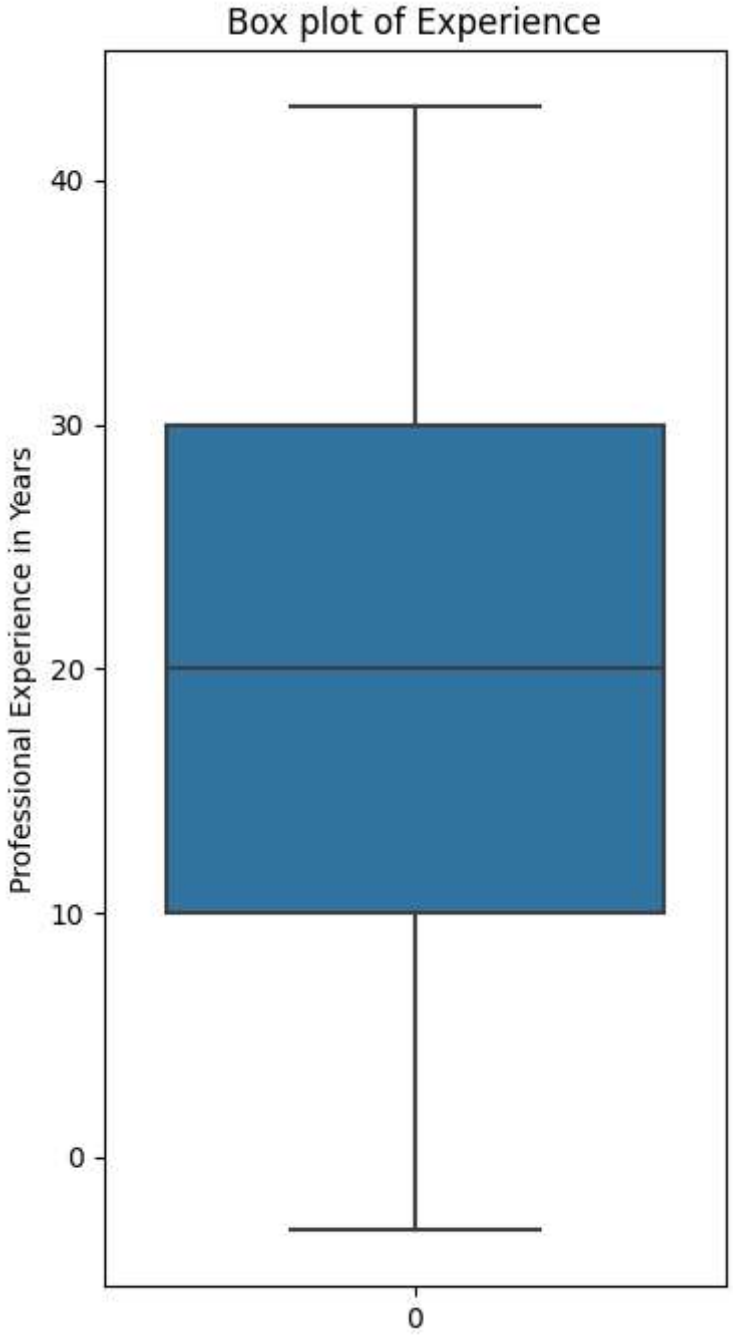
In [147…
```
# Summary Statistics of Experience Column
df["Experience"].describe().round(2)
```

Out[147]:
```
count    4994.00
mean       20.11
std        11.47
min        -3.00
25%        10.00
50%        20.00
75%        30.00
max        43.00
Name: Experience, dtype: float64
```

There are negative years of experience, so something is off, lets check for more errors

In [148…
```
# Figure Size
plt.figure(figsize=(4, 8))

# Plotting a boxplot of Experience Column
sns.boxplot(df["Experience"])
```

```
plt.ylabel("Professional Experience in Years")
plt.title("Box plot of Experience")
plt.show()
```


Box plot of Experience
Professional Experience in Years
40
30
20
10
0
0

In [160…
```
print(
    f"Total number of missing data: {len(df[df['Experience'] < 0])} which is {(((len(df[df['Experience'] < 0]))/len(df))*100):
)
df[df["Experience"] < 0].sample(3)
```

```
Total number of missing data: 52 which is 1.04% of total data
```

Out[160]:

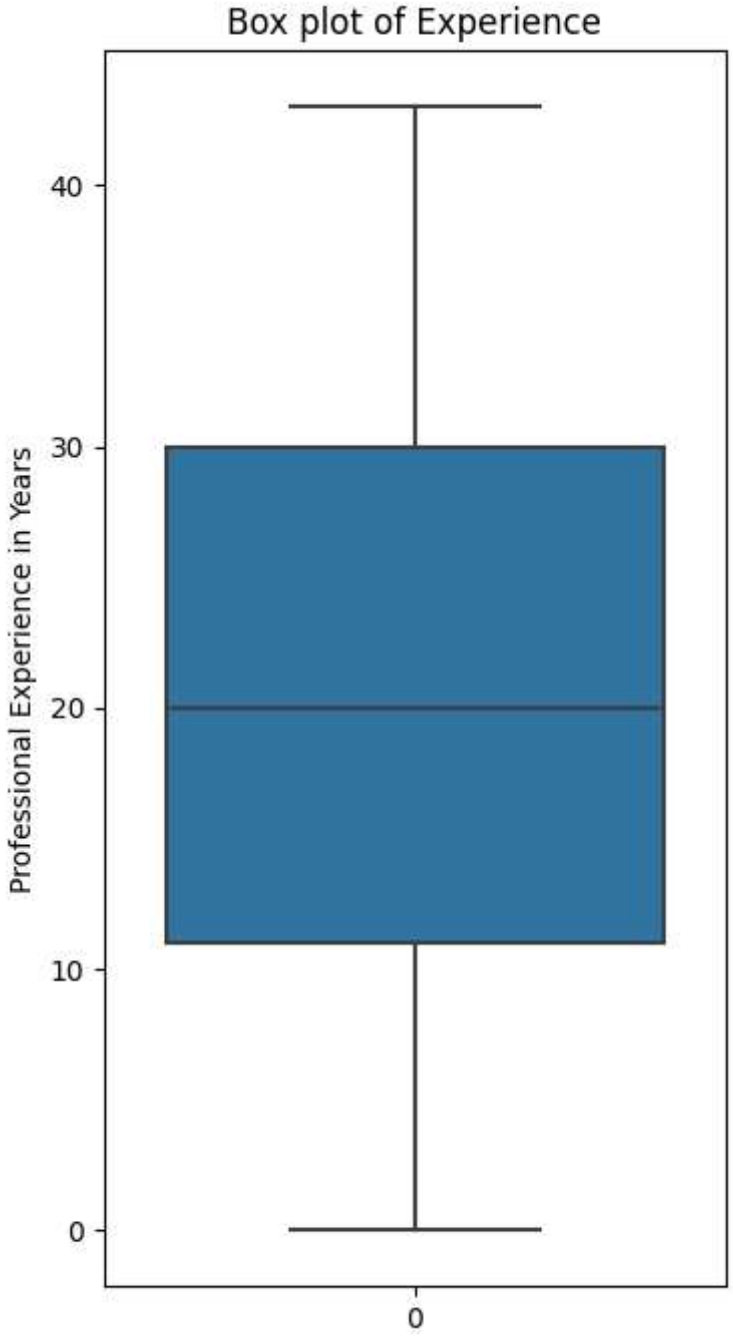
	ID	Age	Experience	Income	ZIP Code	Family	CCAvg	Education	Mortgage	Home Ownership	Personal Loan	Securities Account	CD Account	Online	C
3394	3395	25	-1	113.0	90089	4	2.1	3	0	None	0	0	0	1.0	
4116	4117	24	-2	135.0	90065	2	7.2	1	0	None	0	0	0	1.0	
649	650	25	-1	82.0	92677	4	2.1	3	0	None	0	0	0	1.0	

Since there are 52 rows with negative number of experience, and we do not know what kind of error this is so we can drop the rows with faulty data since the number of missing rows is lower than 5% of the total size of data

In [161…
```
# Selecting rows with Experience below 0 years as it is impossible
rows_to_drop = df[df["Experience"] < 0]

# Dropping those rows
df.drop(rows_to_drop.index, inplace=True)
```

In [164…
```
display(df["Experience"].describe().round(2))
plt.figure(figsize=(4, 8))
sns.boxplot(df["Experience"])
plt.ylabel("Professional Experience in Years")
plt.title("Box plot of Experience")
plt.show()
```

```
count    4942.00
mean       20.33
std        11.31
min         0.00
25%        11.00
50%        20.00
75%        30.00
max        43.00
Name: Experience, dtype: float64
```


Box plot of Experience
Professional Experience in Years
40
30
20
10
0
0

Family and Education Column

We change the datatypes to correct format

In [173…
```
df["Family"] = df["Family"].astype("category")
df["Education"] = df["Education"].astype("category")
```

Mortgage Column

In [174…
```
df["Mortgage"].describe().round(2)
```

Out[174]:
```
count    4942.00
mean       56.54
std       101.58
min         0.00
25%         0.00
50%         0.00
75%       101.00
max       635.00
Name: Mortgage, dtype: float64
```

Lets count the number of people who have mortgages and who dont

In [180…
```
print(f"Number of People with mortgages: {(len(df[df['Mortgage']!=0]))}")
print(f"Number of People without mortgages: {len(df[df['Mortgage']==0])}")
```

```
Number of People with mortgages: 1524
Number of People without mortgages: 3418
```

Lets Create a new column named Has_Mortgage that stores if a person has mortgage or not depending if the value is 0 or more

In [188…
```
# Making a new column that sets value to 0 if mortgage value is equal to 0 and 1 if more than 0
df["Has_Mortgage"] = np.where(df["Mortgage"] > 0, 1, 0)

# Changing to Category dtype
df["Has_Mortgage"] = df["Has_Mortgage"].astype("category")
```

In [189…
```
df["Has_Mortgage"].value_counts()
```

Out[189]:
```
0    3418
1    1524
Name: Has_Mortgage, dtype: int64
```

Securities Account Column

In [192…
```
df["Securities Account"] = df["Securities Account"].astype("category")
```

CD Account Column

In [194…
```
df["CD Account"] = df["CD Account"].astype("category")
```

Online Column

In [195…
```
df["Online"] = df["Online"].astype("category")
```

CreditCard Column

In [196…
```
df["CreditCard"] = df["CreditCard"].astype("category")
```

In [307…
```
# Summary Statistics of numerical columns (Transposed)
display(df.describe(include=np.number).round(2).T)

# Summary Statistics of categorical column (Transposed)
display(df.describe(exclude=np.number).T)
```

	count	mean	std	min	25%	50%	75%	max
ID	4941.0	2503.27	1442.82	1.0	1257.0	2500.0	3752.0	5000.0
Age	4941.0	45.51	11.44	0.0	36.0	46.0	55.0	97.0
Experience	4941.0	20.33	11.31	0.0	11.0	20.0	30.0	43.0
Income	4941.0	72.53	45.10	8.0	39.0	63.0	94.0	224.0
ZIP Code	4941.0	93152.94	2125.97	9307.0	91911.0	93437.0	94608.0	96651.0
CCAvg	4941.0	1.93	1.74	0.0	0.7	1.5	2.6	10.0
Mortgage	4941.0	56.47	101.47	0.0	0.0	0.0	101.0	635.0
Personal Loan	4941.0	0.10	0.29	0.0	0.0	0.0	0.0	1.0

	count	unique	top	freq
Family	4941	4	1	1469
Education	4941	3	1	2079
Home Ownership	4941	4	None	4468
Securities Account	4941	2	0	4425
CD Account	4941	2	0	4641
Online	4941.0	2.0	1.0	2966.0
CreditCard	4941	2	0	3488
Has_Mortgage	4941	2	0	3418

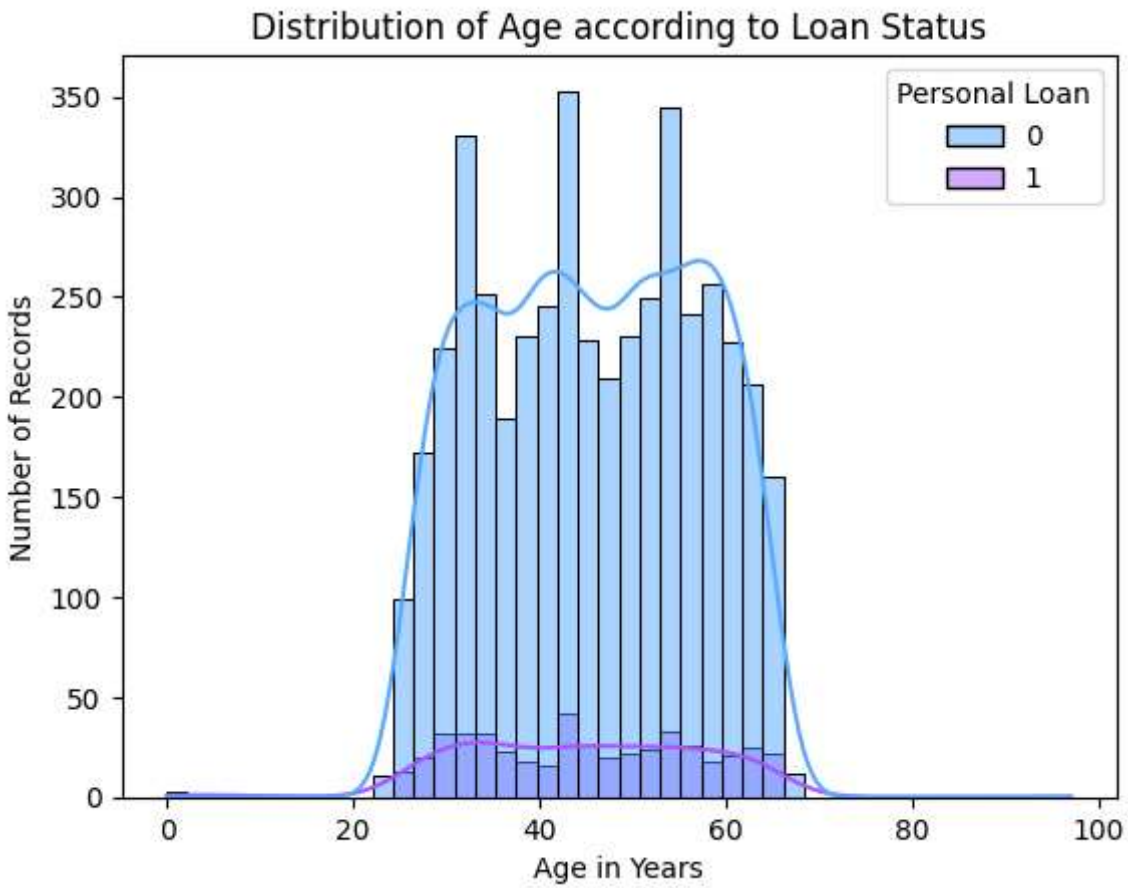
In [215…

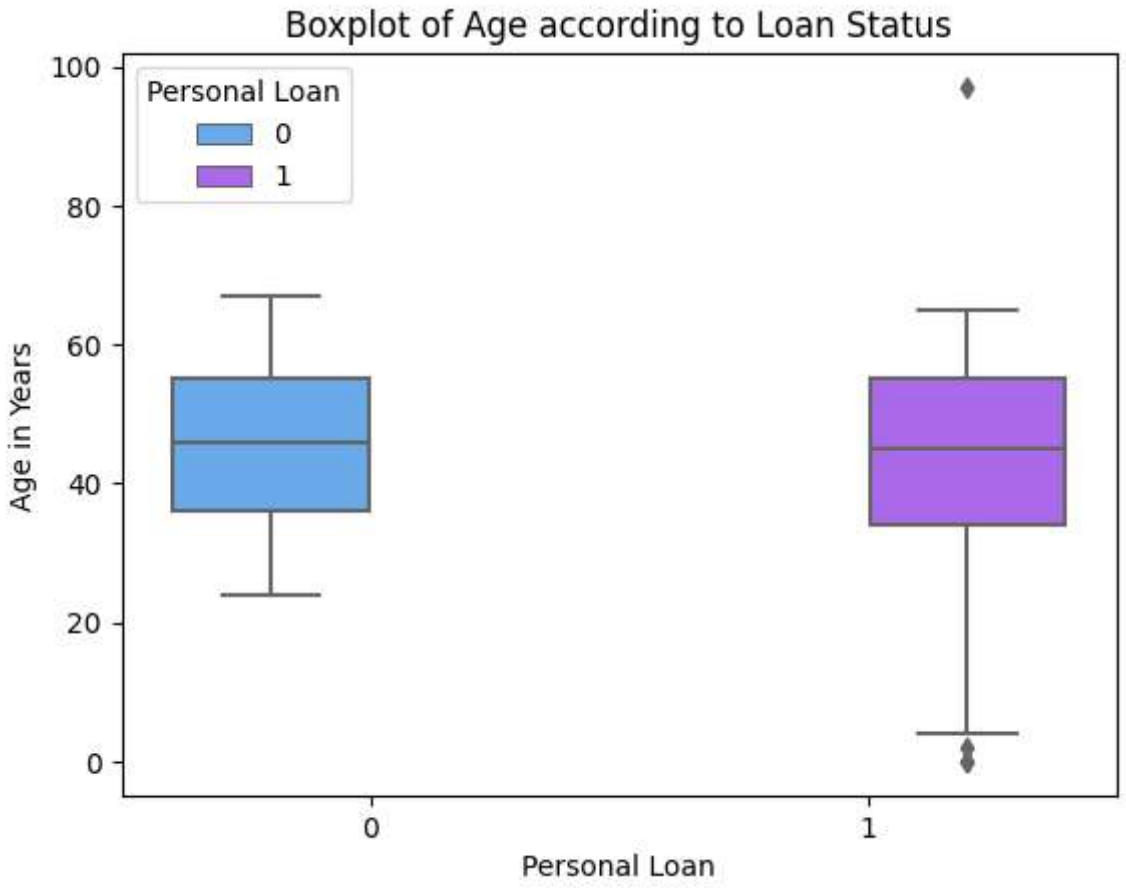
```
# There is a value with no visible character in Personal Loan, so locating it and removing from data
ghost_value_row = df[(df["Personal Loan"] != 0) & (df["Personal Loan"] != 1)]

# Dropping the ghost row
df.drop(ghost_value_row.index, inplace=True)
```

EDA and Graphs

In [236…

```
sns.histplot(data=df, x="Age", kde=True, hue="Personal Loan", palette="cool")
plt.xlabel("Age in Years")
plt.ylabel("Number of Records")
plt.title("Distribution of Age according to Loan Status")
plt.show()

sns.boxplot(data=df, x="Personal Loan", y="Age", hue="Personal Loan", palette="cool")
plt.ylabel("Age in Years")
plt.title("Boxplot of Age according to Loan Status")
plt.show()
```

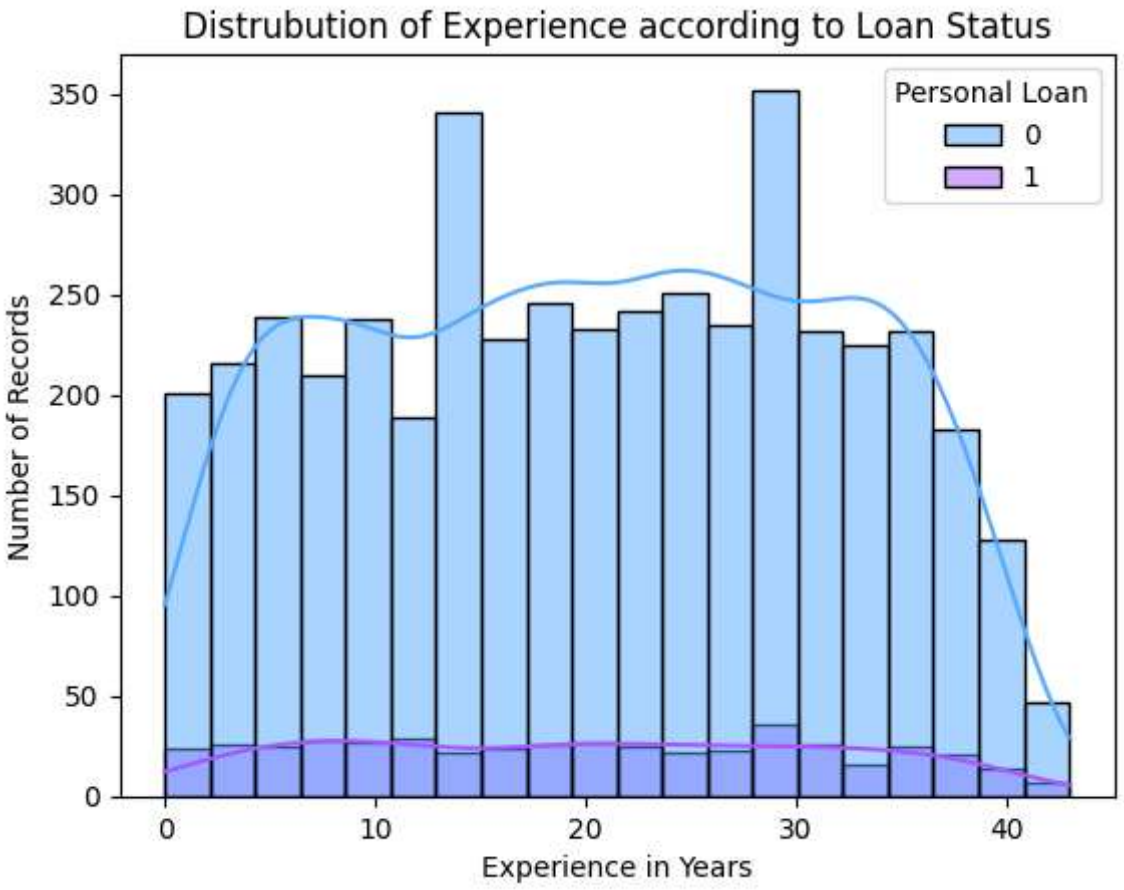

Distribution of Age according to Loan Status
Personal Loan
0
1
Number of Records
0
50
100
150
200
250
300
350
0
20
40
60
80
100
Age in Years

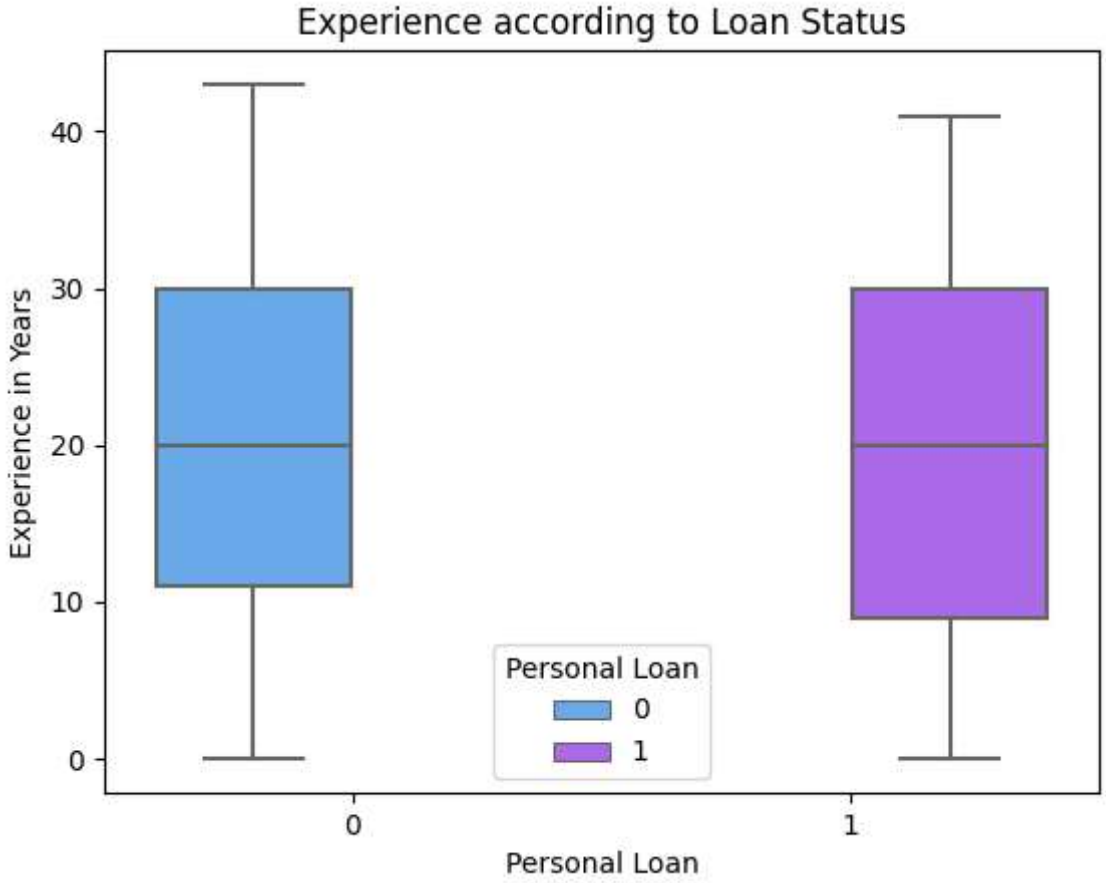


In [235…

```
sns.histplot(data=df, x="Experience", kde=True, hue="Personal Loan", palette="cool")
plt.xlabel("Experience in Years")
plt.ylabel("Number of Records")
plt.title("Distrubution of Experience according to Loan Status")
plt.show()


sns.boxplot(
    data=df, x="Personal Loan", y="Experience", hue="Personal Loan", palette="cool"
)
plt.ylabel("Experience in Years")
```

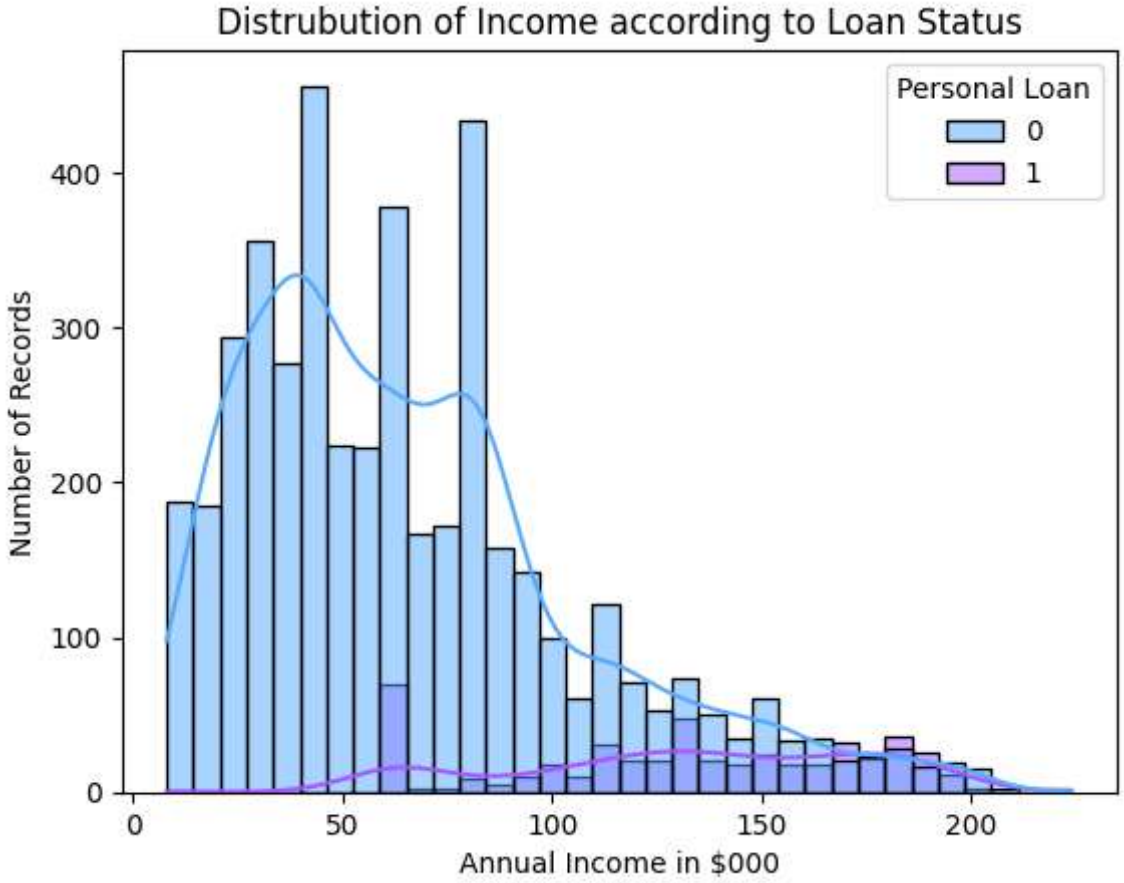
```
plt.title("Experience according to Loan Status")
plt.show()
```

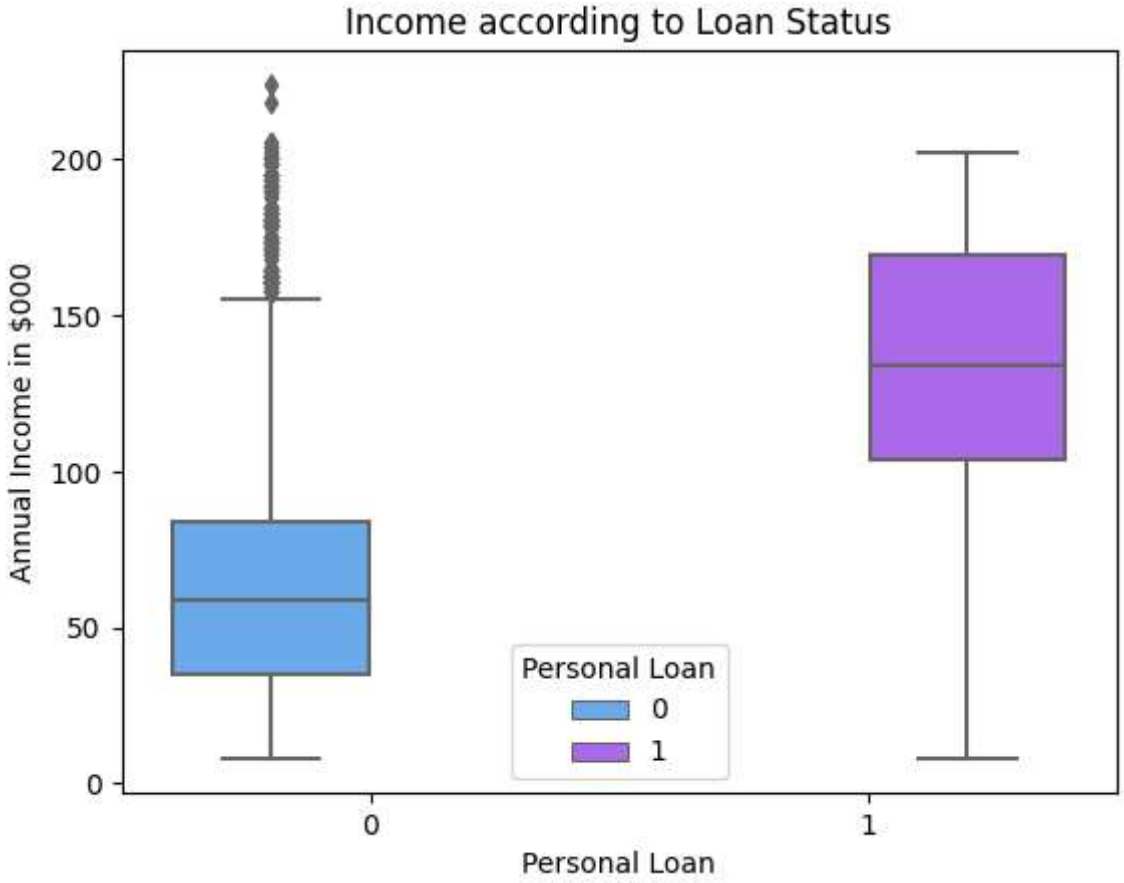




In [234…

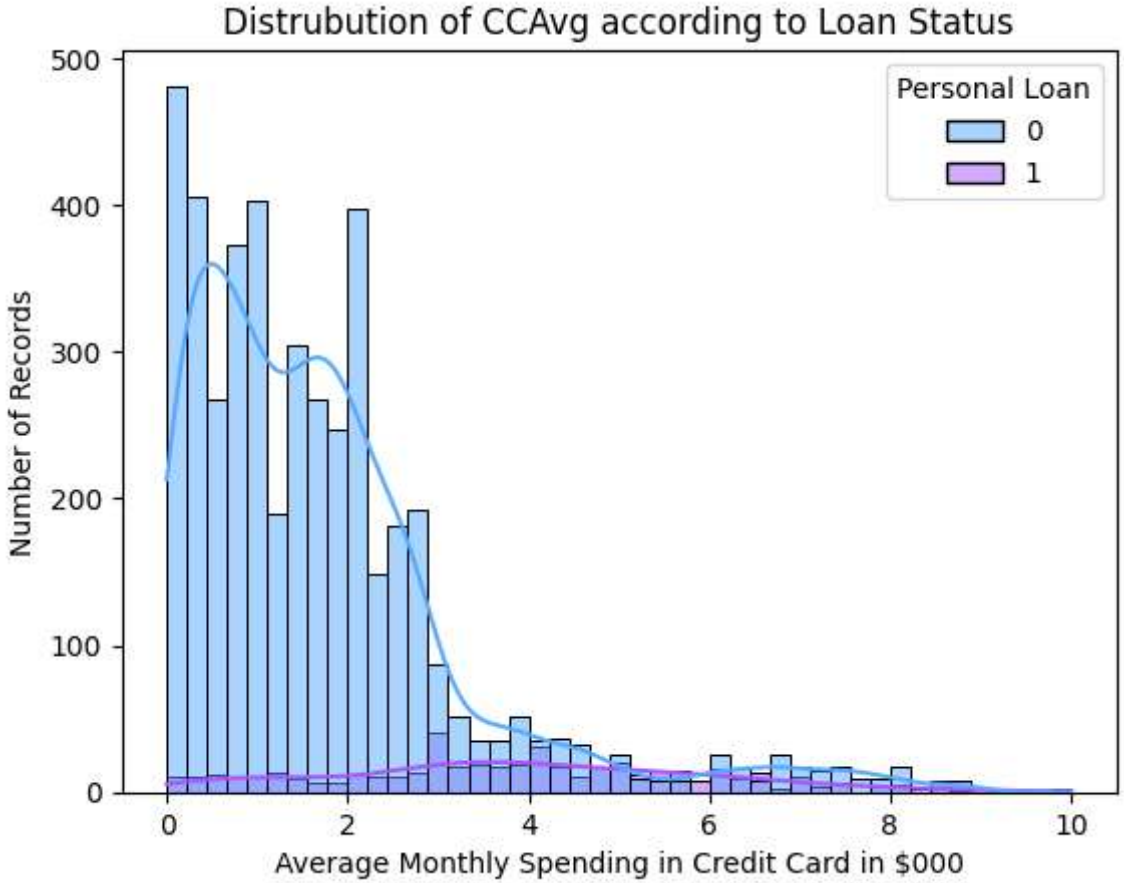
```
sns.histplot(data=df, x="Income", kde=True, hue="Personal Loan", palette="cool")
plt.xlabel("Annual Income in $000")
plt.ylabel("Number of Records")
plt.title("Distrubution of Income according to Loan Status")
plt.show()


sns.boxplot(data=df, x="Personal Loan", y="Income", hue="Personal Loan", palette="cool")
plt.ylabel("Annual Income in $000")
plt.title("Income according to Loan Status")
plt.show()
```

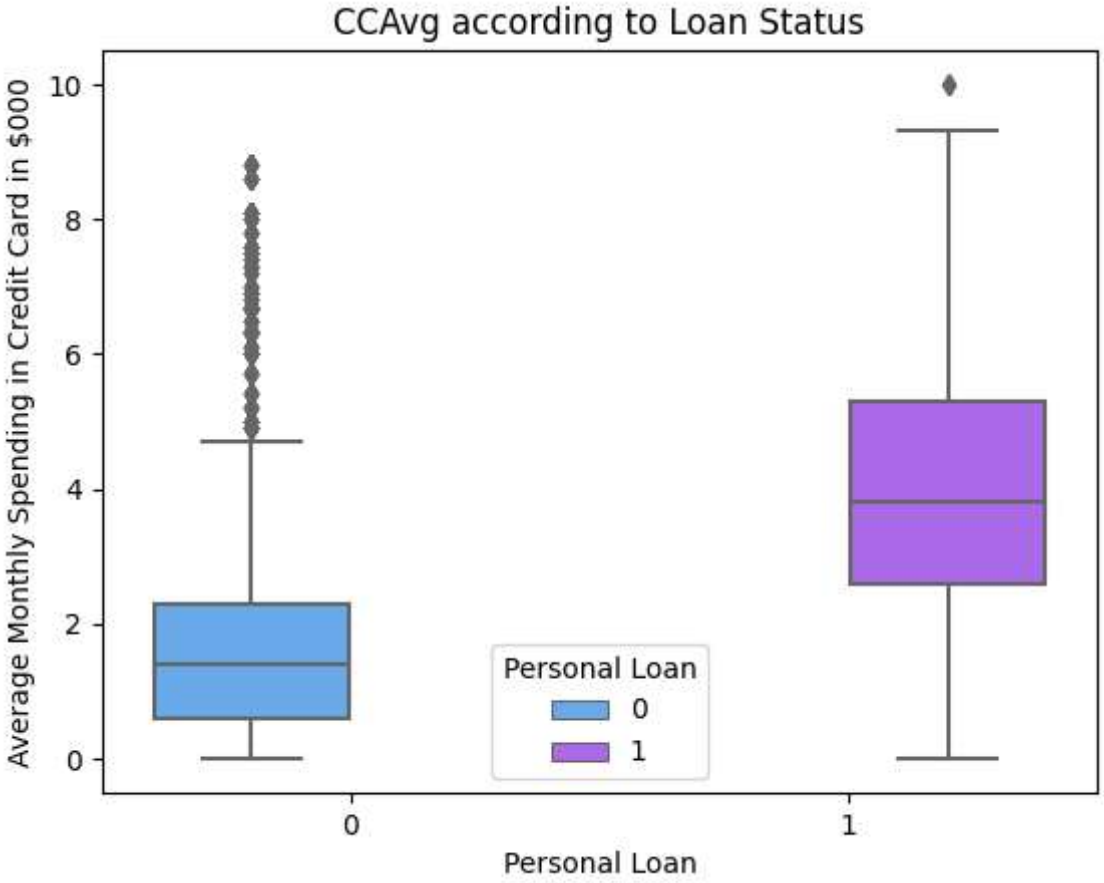

Distrubution of Income according to Loan Status
Personal Loan
0
1
Number of Records
0
100
200
300
400
0
50
100
150
200
Annual Income in $000



In [233…

```
sns.histplot(data=df, x="CCAvg", kde=True, hue="Personal Loan", palette="cool")
plt.xlabel("Average Monthly Spending in Credit Card in $000")
plt.ylabel("Number of Records")
plt.title("Distrubution of CCAvg according to Loan Status")
plt.show()


sns.boxplot(data=df, x="Personal Loan", y="CCAvg", hue="Personal Loan", palette="cool")
plt.ylabel("Average Monthly Spending in Credit Card in $000")
plt.title("CCAvg according to Loan Status")
plt.show()
```

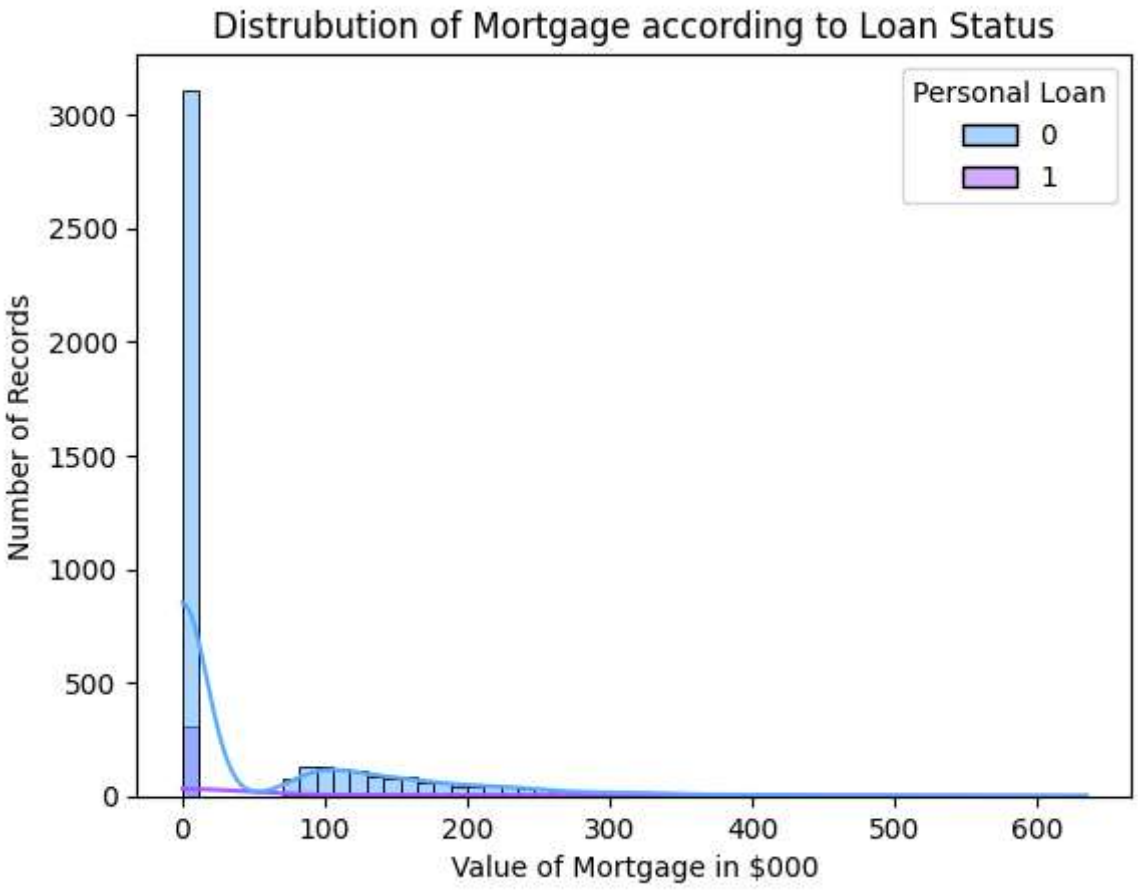

Distrubution of CCAvg according to Loan Status
Personal Loan
0
1
Number of Records
500
400
300
200
100
0
0
2
4
6
8
10
Average Monthly Spending in Credit Card in $000

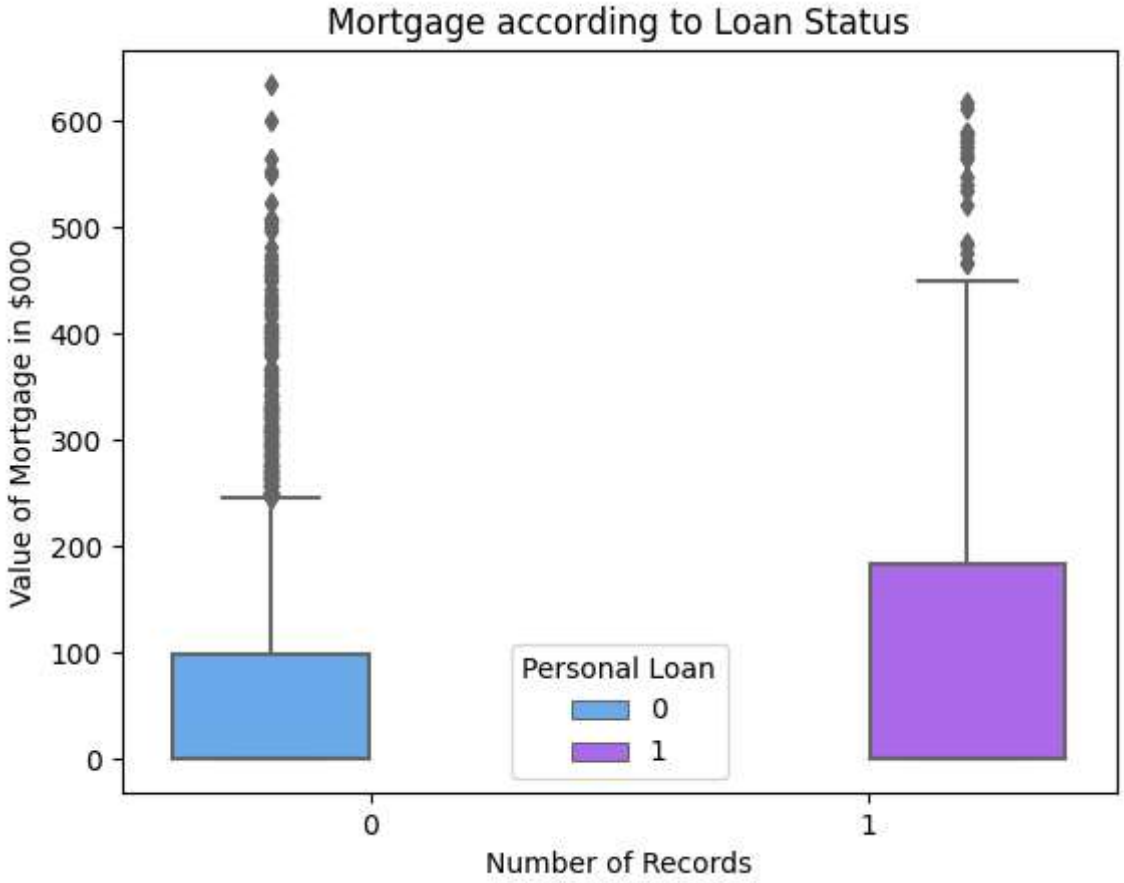


In [232…

```
sns.histplot(data=df, x="Mortgage", kde=True, hue="Personal Loan", palette="cool")
plt.xlabel("Value of Mortgage in $000")
plt.ylabel("Number of Records")
plt.title("Distrubution of Mortgage according to Loan Status")
plt.show()


sns.boxplot(
    data=df, x="Personal Loan", y="Mortgage", hue="Personal Loan", palette="cool"
)
plt.ylabel("Value of Mortgage in $000")
```

```
plt.title("Mortgage according to Loan Status")
plt.show()
```





In [308…

```
# Summary Stats of categorical columns (Transposed)
df.describe(exclude=np.number).T
```

Out[308]:

	count	unique	top	freq
Family	4941	4	1	1469
Education	4941	3	1	2079
Home Ownership	4941	4	None	4468
Securities Account	4941	2	0	4425
CD Account	4941	2	0	4641
Online	4941.0	2.0	1.0	2966.0
CreditCard	4941	2	0	3488
Has_Mortgage	4941	2	0	3418

In [309…

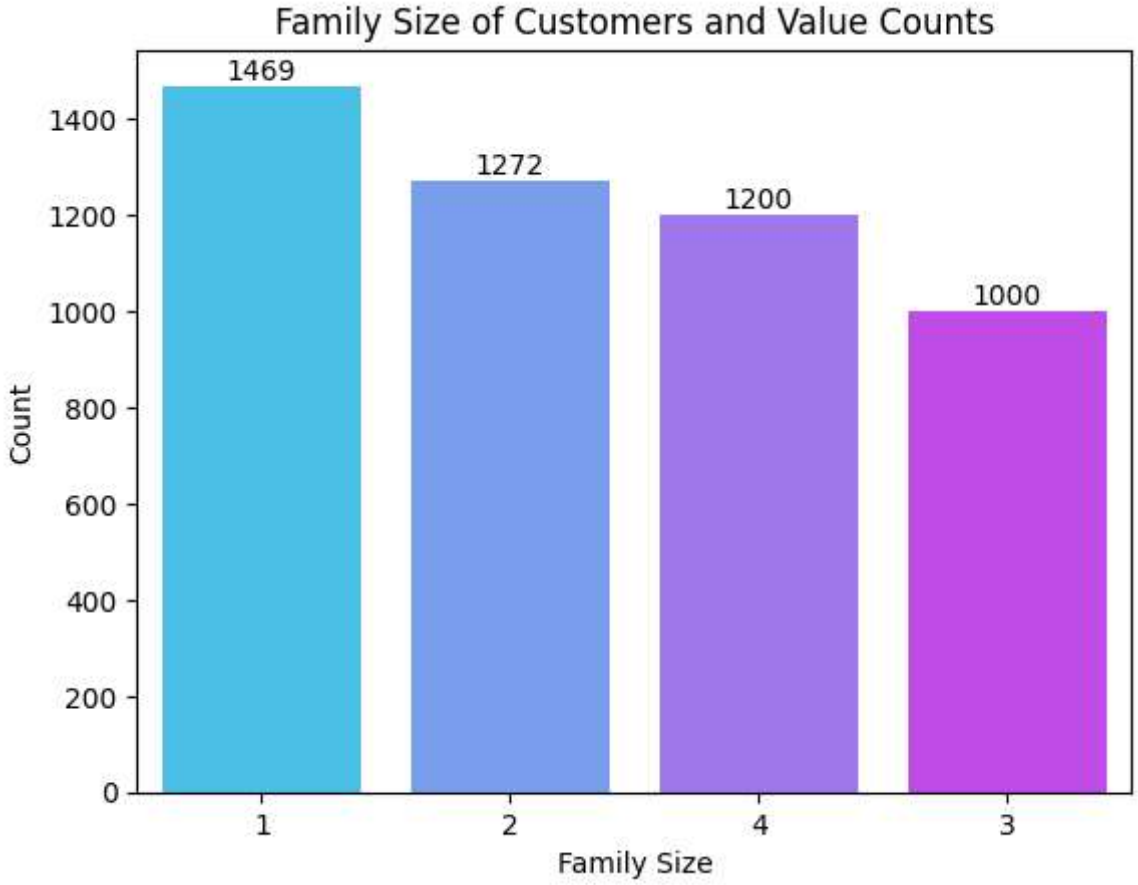
```python
# cst = create sorted dataframe
def cst(column_name: str):
    """
    Function to create a dataframe that stores the unique values in the column along with count and returns a dataframe

    Params: column_name (str) : Name of Categorical Column to Create the sorted dataframe for

    Returns: dataframe(DataFrame)
    """
    value_counts = df[column_name].value_counts()
    dataframe = pd.DataFrame(value_counts).reset_index()
    dataframe = dataframe.rename(columns={"index": column_name, column_name: "Count"})
    dataframe = dataframe.sort_values(by=["Count"], ascending=False)
    return dataframe


cst("Family")
```

Out[309]:

	Family	Count
0	1	1469
1	2	1272
2	4	1200
3	3	1000

In [256…

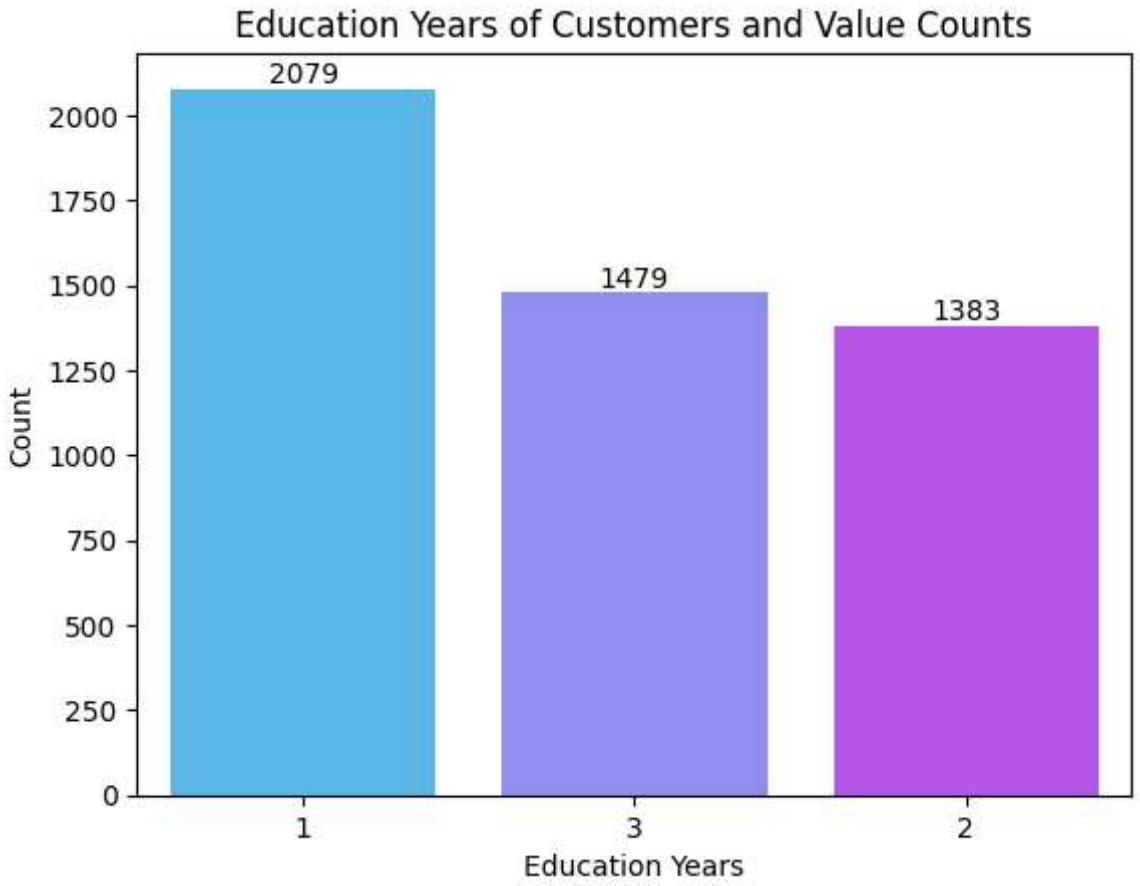
```
# Creating a sorted dataframe to preserve high to low value order
family_count = cst("Family")

# Plotting the count plot
sns.countplot(data=df, x="Family",
              order=family_count["Family"], palette="cool")

# Annotating with values by iterating through rows
for i, count in enumerate(family_count["Count"]):
    plt.annotate(str(count), xy=(i, count), ha="center", va="bottom")

plt.xlabel("Family Size")
plt.ylabel("Count")
plt.title("Family Size of Customers and Value Counts")
plt.show()
```



In [261…

```
# Creating a sorted dataframe to preserve high to low value order
education_count = cst("Education")

# Plotting the count plot
sns.countplot(
    data=df, x="Education", order=education_count["Education"], palette="cool"
)

# Annotating with values by iterating through rows
for i, count in enumerate(education_count["Count"]):
    plt.annotate(str(count), xy=(i, count), ha="center", va="bottom")

plt.xlabel("Education Years")
```

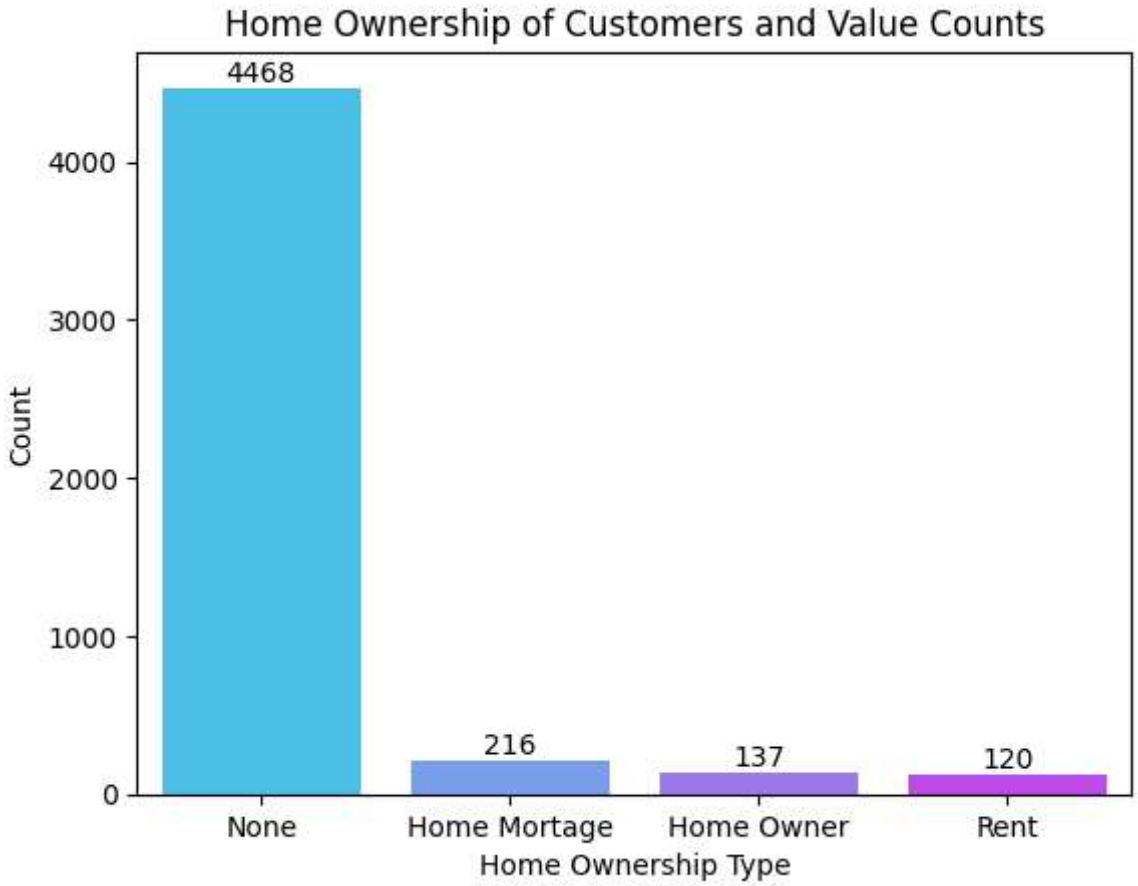
```
plt.ylabel("Count")
plt.title("Education Years of Customers and Value Counts")
plt.show()
```



In [262…

```
# Creating a sorted dataframe to preserve high to low value order
homeownership_count = cst("Home Ownership")

# Plotting the count plot
sns.countplot(
    data=df,
    x="Home Ownership",
    order=homeownership_count["Home Ownership"],
    palette="cool",
```

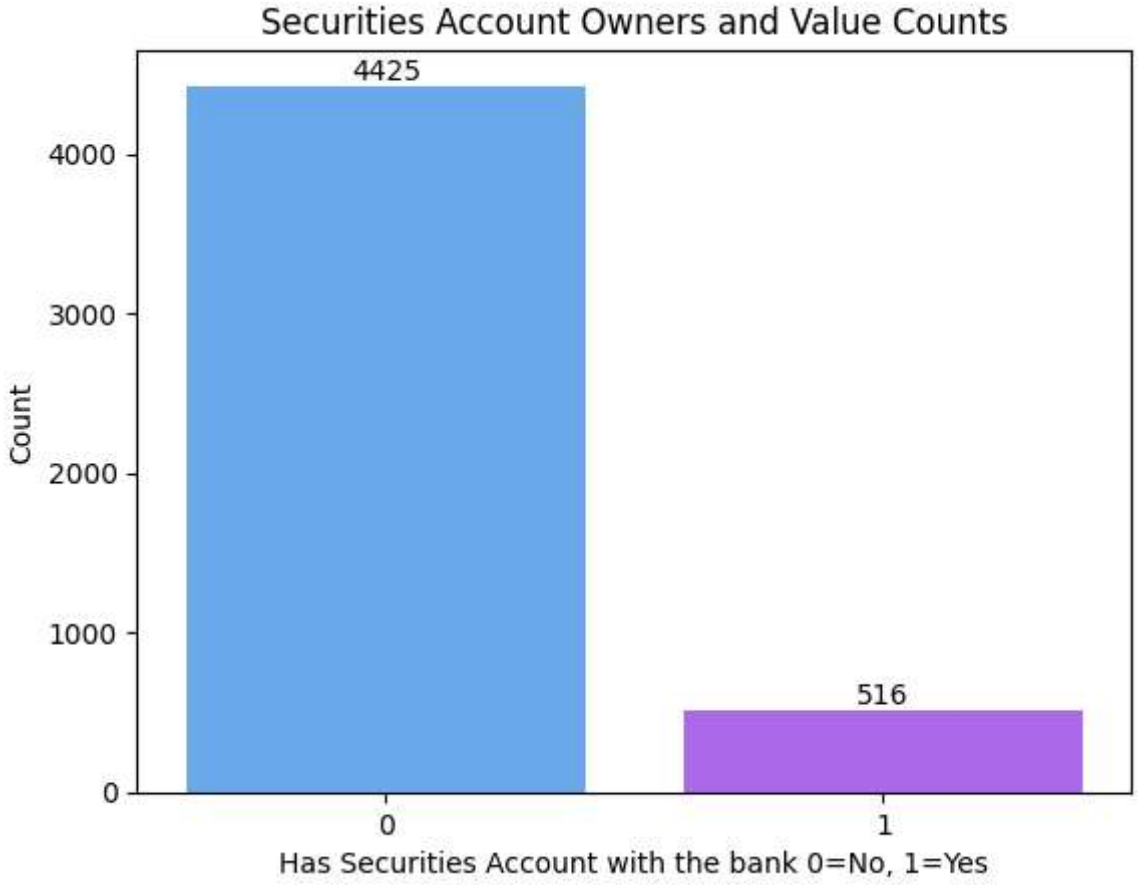
```
)

# Annotating with values by iterating through rows
for i, count in enumerate(homeownership_count["Count"]):
    plt.annotate(str(count), xy=(i, count), ha="center", va="bottom")

plt.xlabel("Home Ownership Type")
plt.ylabel("Count")
plt.title("Home Ownership of Customers and Value Counts")
plt.show()
```



In [267…

```
# Creating a sorted dataframe to preserve high to low value order
securityaccount_count = cst("Securities Account")
```

```
# Plotting the count plot
sns.countplot(
    data=df,
    x="Securities Account",
    order=securityaccount_count["Securities Account"],
    palette="cool",
)

# Annotating with values by iterating through rows
for i, count in enumerate(securityaccount_count["Count"]):
    plt.annotate(str(count), xy=(i, count), ha="center", va="bottom")

plt.xlabel("Has Securities Account with the bank 0=No, 1=Yes")
plt.ylabel("Count")
plt.title("Securities Account Owners and Value Counts")
plt.show()
```

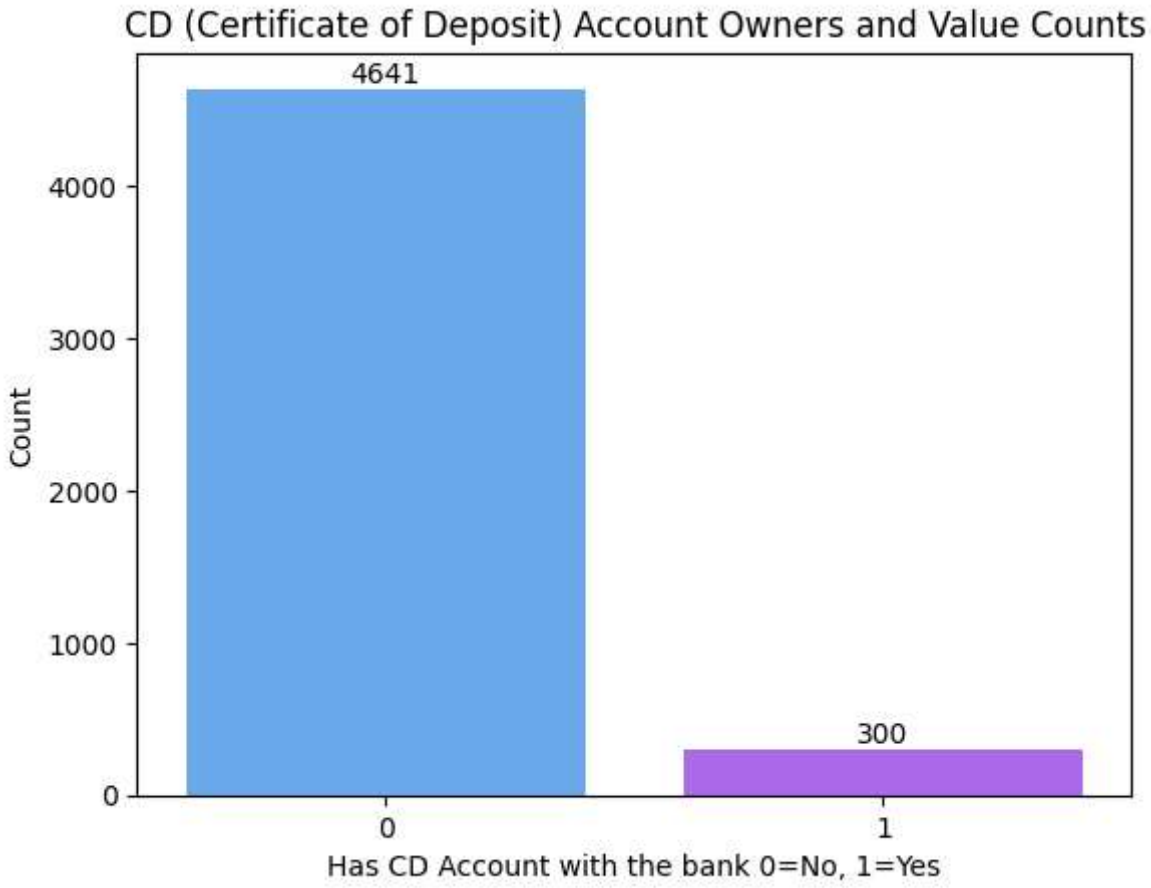


CD Account, Online, Has_Mortgage, Credit Card

In [269…

```
# Creating a sorted dataframe to preserve high to low value order
cd_count = cst("CD Account")

# Plotting the count plot
sns.countplot(data=df, x="CD Account",
              order=cd_count["CD Account"], palette="cool")

# Annotating with values by iterating through rows
for i, count in enumerate(cd_count["Count"]):
    plt.annotate(str(count), xy=(i, count), ha="center", va="bottom")
```

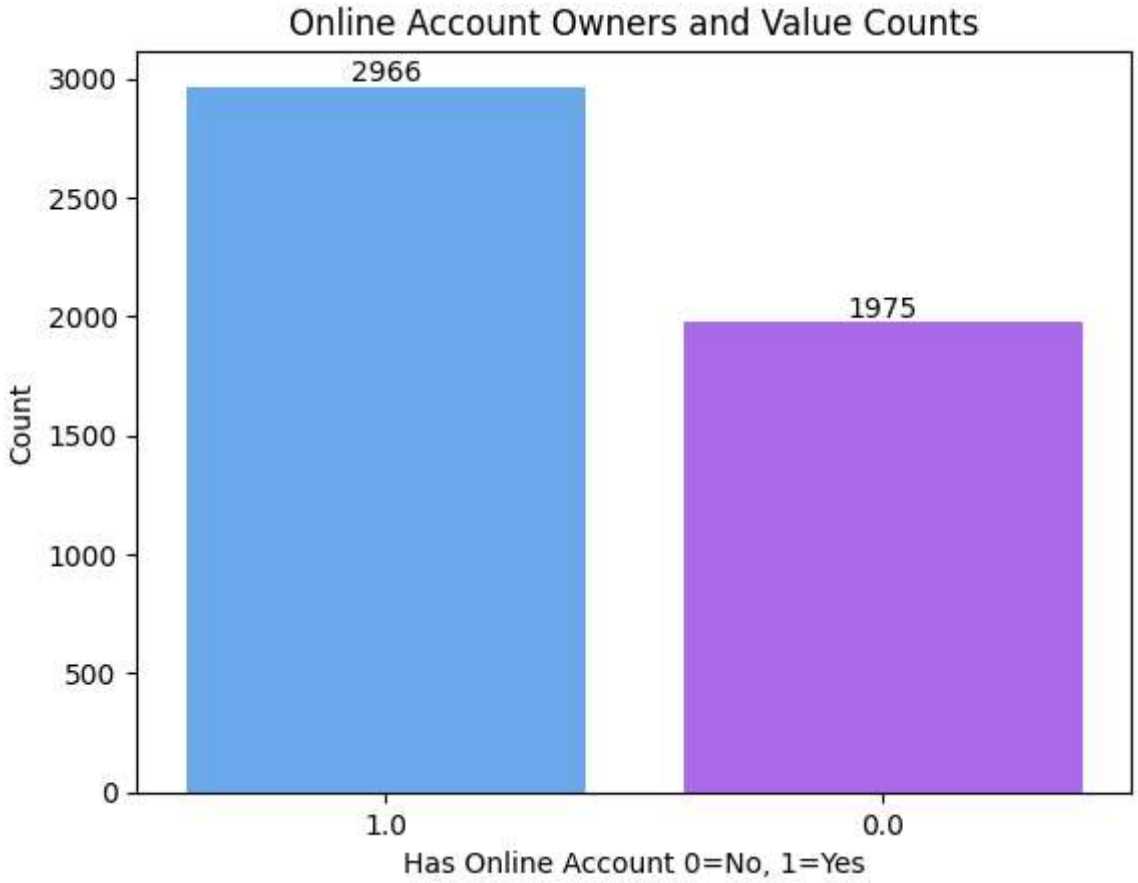
```
plt.xlabel("Has CD Account with the bank 0=No, 1=Yes")
plt.ylabel("Count")
plt.title("CD (Certificate of Deposit) Account Owners and Value Counts")
plt.show()
```



In [304…

```
# Creating a sorted dataframe to preserve high to low value order
count_df = cst("Online")

# Plotting the count plot
sns.countplot(data=df, x="Online", order=count_df["Online"], palette="cool")

# Annotating with values by iterating through rows
```

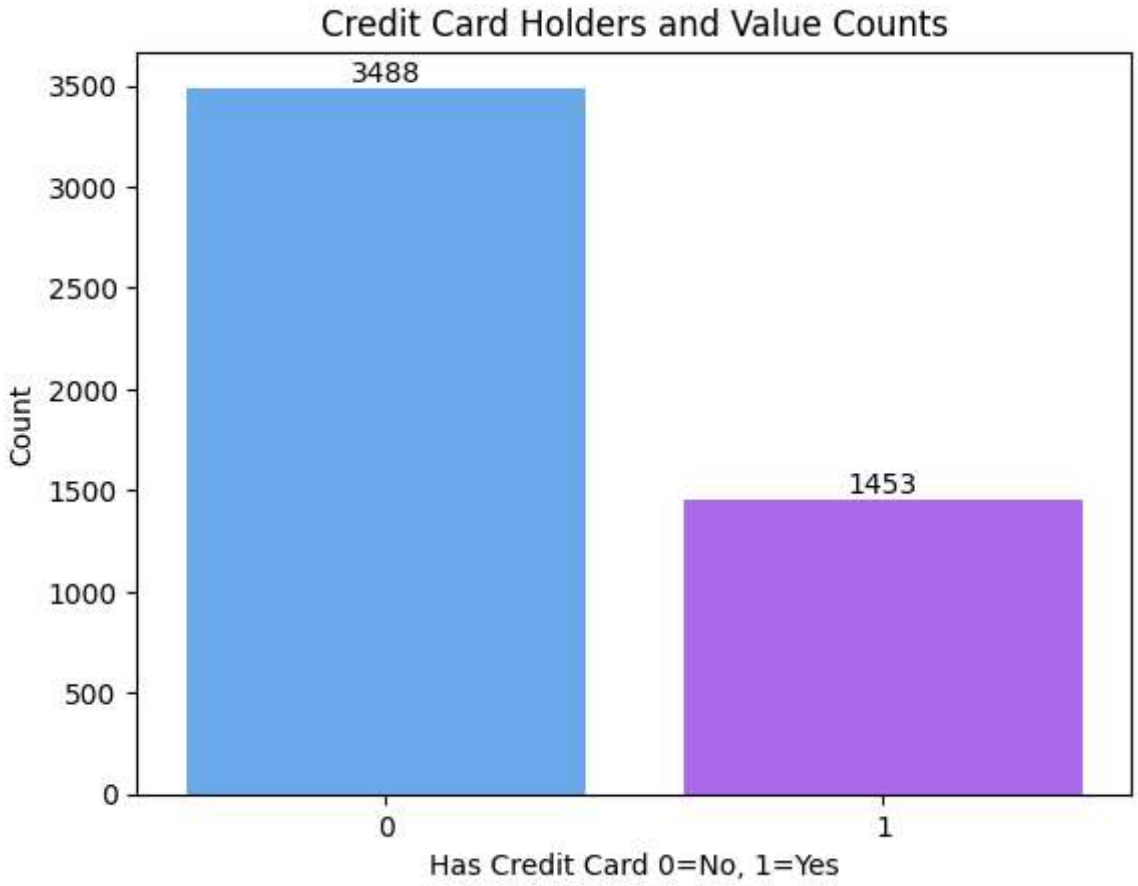
```
for i, count in enumerate(count_df["Count"]):
    plt.annotate(str(count), xy=(i, count), ha="center", va="bottom")
plt.xlabel("Has Online Account 0=No, 1=Yes")
plt.ylabel("Count")
plt.title("Online Account Owners and Value Counts")
plt.show()
```



In [280…

```
# Creating a sorted dataframe to preserve high to low value order
count_df = cst("CreditCard")

# Plotting the count plot
sns.countplot(data=df, x="CreditCard",
              order=count_df["CreditCard"], palette="cool")
```

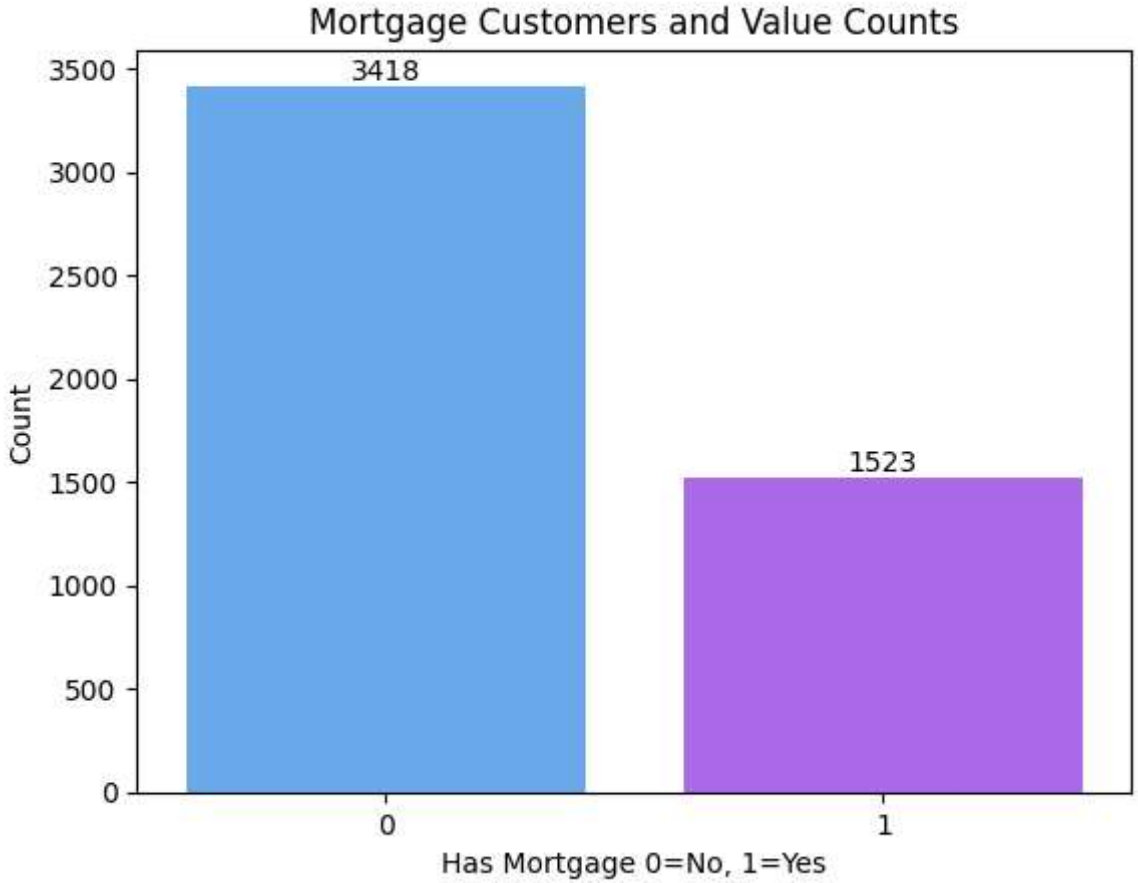
```
# Annotating with values by iterating through rows
for index, row in count_df.iterrows():
    plt.annotate(str(row["Count"]), xy=(
        index, row["Count"]), ha="center", va="bottom")

plt.xlabel("Has Credit Card 0=No, 1=Yes")
plt.ylabel("Count")
plt.title("Credit Card Holders and Value Counts")
plt.show()
```



In [283…

```
# Creating a sorted dataframe to preserve high to low value order
count_df = cst("Has_Mortgage")
```

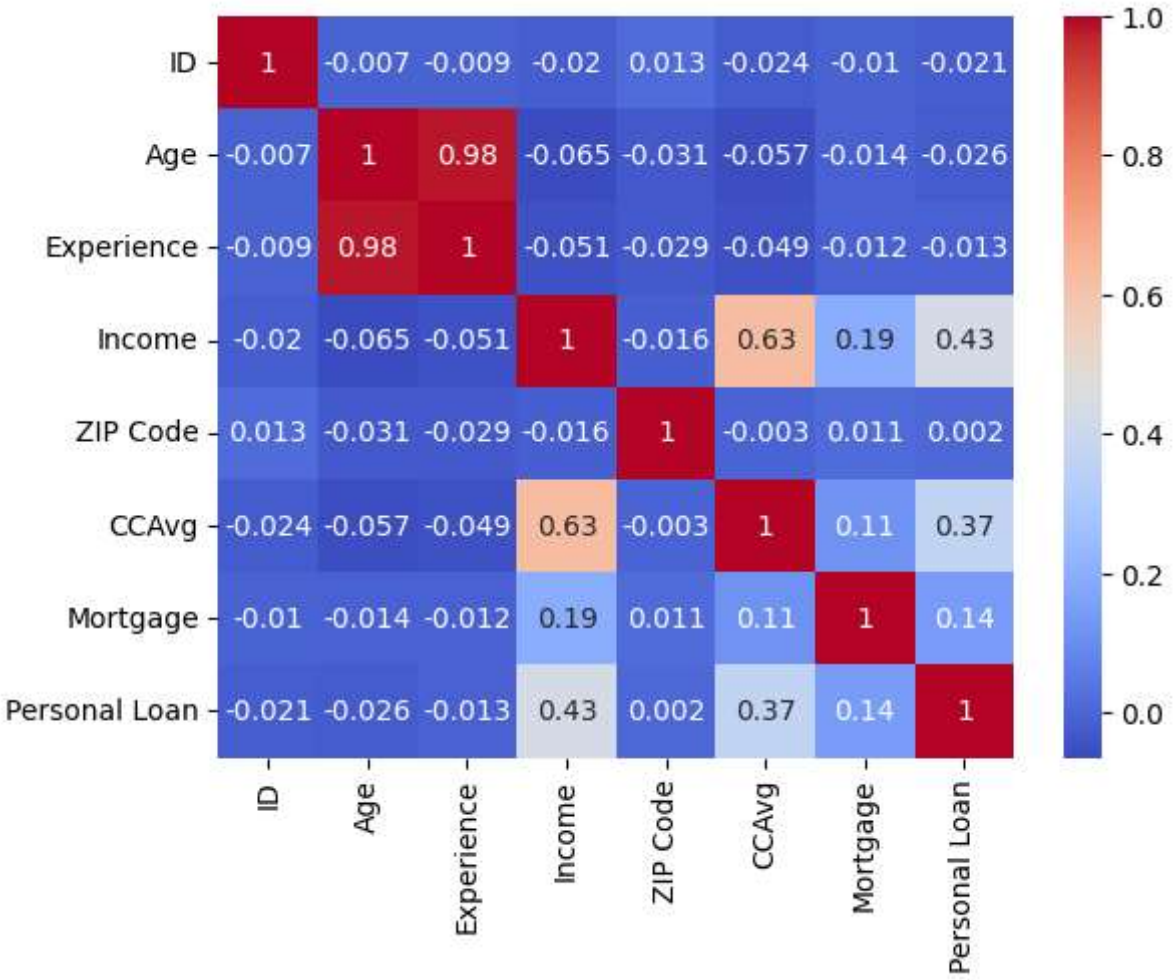
```
# Plotting the count plot
sns.countplot(data=df, x="Has_Mortgage",
              order=count_df["Has_Mortgage"], palette="cool")

# Annotating with values by iterating through rows
for index, row in count_df.iterrows():
    plt.annotate(str(row["Count"]), xy=(
        index, row["Count"]), ha="center", va="bottom")


plt.xlabel("Has Mortgage 0=No, 1=Yes")
plt.ylabel("Count")
plt.title("Mortgage Customers and Value Counts")
plt.show()
```



Looking at the correlation

In [313…

```
# Converting Personal Loan to numeric Column
df["Personal Loan"] = df["Personal Loan"].astype(np.number)

# Making a correlation table
corr_table = df.corr(numeric_only=True).round(3)

# Plotting the heatmap
sns.heatmap(corr_table, annot=True, cmap="coolwarm")
plt.show()
```

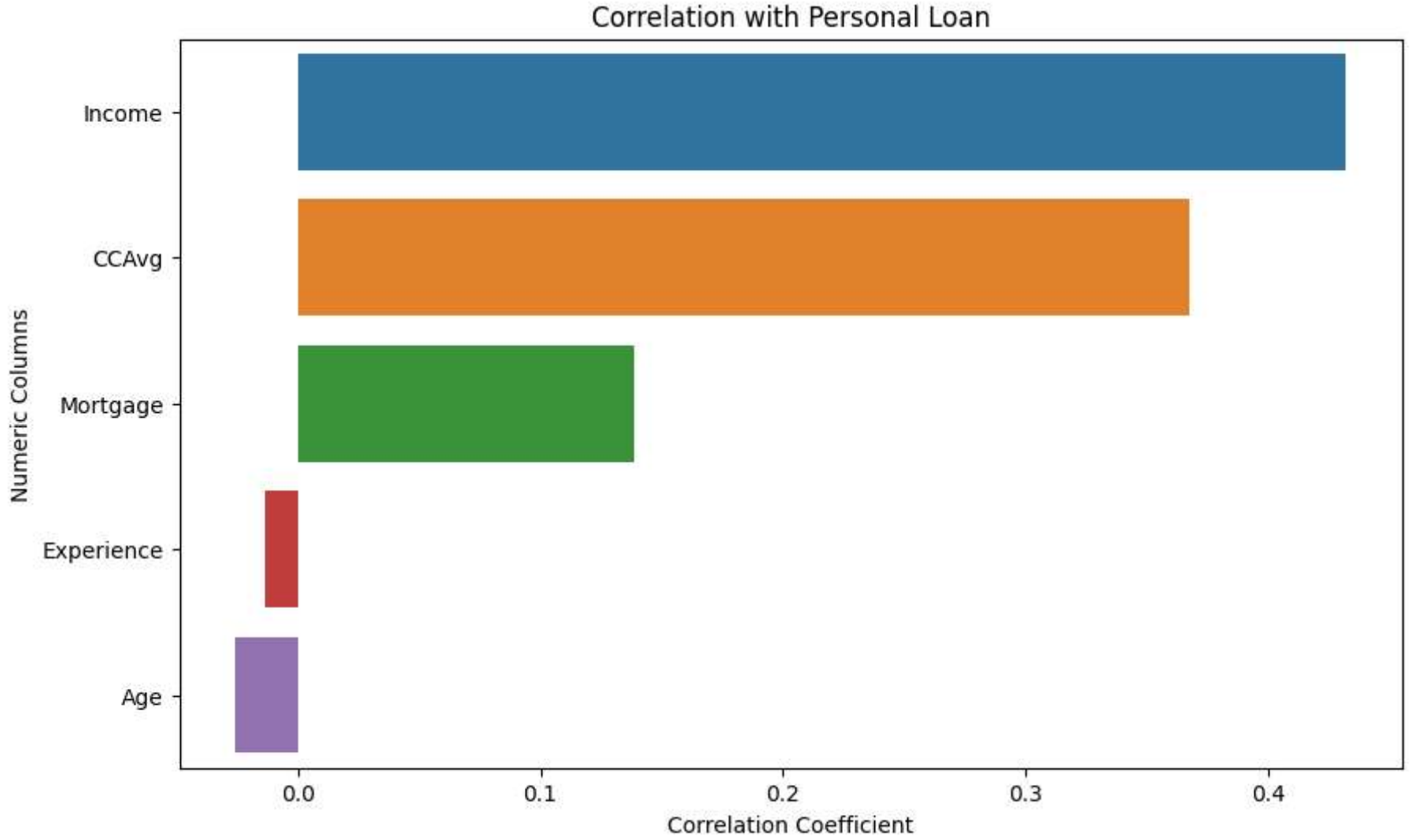


In [303…

```
# Selecting numeric columns
numeric_cols = df.select_dtypes(include=["float64", "int64"]).columns

# Dropping unnecessary and irrevelant columns
numeric_cols = numeric_cols.drop(["Personal Loan", "ID", "ZIP Code"])


# Making a correlation matrix specifically with the Personal Loan Column
correlation_matrix = df[numeric_cols].corrwith(df["Personal Loan"])
display(correlation_matrix)
```

```
# Sorting the columns
sorted_correlation = correlation_matrix.sort_values(ascending=False)

# Plotting
plt.figure(figsize=(10, 6))
sns.barplot(x=sorted_correlation.values,
            y=sorted_correlation.index, orient="h")
plt.xlabel("Correlation Coefficient")
plt.ylabel("Numeric Columns")
plt.title("Correlation with Personal Loan")
plt.show()
```

```
Age          -0.025797
Experience   -0.013216
Income        0.431862
CCAvg         0.367072
Mortgage      0.138826
dtype: float64
```


Correlation with Personal Loan
Income
CCAvg
Mortgage
Experience
Age
Numeric Columns
0.0
0.1
0.2
0.3
0.4
Correlation Coefficient

In [330…

Out[330]:
```
ID                     int64
Age                    int64
Experience             int64
Income               float64
ZIP Code               int64
Family              category
CCAvg                float64
Education           category
Mortgage               int64
Home Ownership        object
Personal Loan        float64
Securities Account  category
CD Account          category
Online              category
CreditCard          category
Has_Mortgage        category
dtype: object
```

Making the Model

We use two machine learning models to predict the Personal Loan column:

1. Random Forest
2. XGBoost

In [334…
```
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import MinMaxScaler, StandardScaler
from sklearn.ensemble import RandomForestClassifier
import xgboost as xgb
from sklearn.metrics import confusion_matrix, precision_score, accuracy_score
```

In [356…
```
# Encoding the Family Column and Dummy Trapping (Reducing dimensions)
encoded_df = pd.get_dummies(df, columns=['Family'], prefix='Family', prefix_sep='-', drop_first=True)

# Encoding the Education Column and Dummy Trapping (Reducing dimensions)
encoded_df = pd.get_dummies(encoded_df, columns=['Education'], prefix='Education', prefix_sep='-', drop_first=True)
```

```
# Encoding the Home Ownership Column and Dummy Trapping (Reducing dimensions)
encoded_df = pd.get_dummies(encoded_df, columns=[
                            'Home Ownership'], prefix='HO', prefix_sep='-', drop_first=True)

# Encoding the Home Online Column and Dummy Trapping (Reducing dimensions)
encoded_df = pd.get_dummies(encoded_df, columns=[
                            'Online'], prefix='Online', prefix_sep='-', drop_first=True)


encoded_df
```

Out[356]:

	ID	Age	Experience	Income	ZIP Code	CCAvg	Mortgage	Personal Loan	Securities Account	CD Account	...	Has_Mortgage	Family-2	Family-3	Fam
0	1	25	1	49.0	91107	1.6	0	0.0	1	0	...	0	0	0	
1	2	45	19	34.0	90089	1.5	0	0.0	1	0	...	0	0	1	
2	3	39	15	11.0	94720	1.0	0	0.0	0	0	...	0	0	0	
3	4	35	9	100.0	94112	2.7	0	0.0	0	0	...	0	0	0	
4	5	35	8	45.0	91330	1.0	0	0.0	0	0	...	0	0	0	
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	
4995	4996	29	3	40.0	92697	1.9	0	0.0	0	0	...	0	0	0	
4996	4997	30	4	15.0	92037	0.4	85	0.0	0	0	...	1	0	0	
4997	4998	63	39	24.0	93023	0.3	0	0.0	0	0	...	0	1	0	
4998	4999	65	40	49.0	90034	0.5	0	0.0	0	0	...	0	0	1	
4999	5000	28	4	83.0	92612	0.8	0	0.0	0	0	...	0	0	1	

4941 rows × 21 columns

In [357…
```
# Splitting the dataset
X_train, X_test, y_train, y_test = train_test_split(
    encoded_df.drop(columns=["Personal Loan", "ID"]), df["Personal Loan"], test_size=0.2, random_state=2
)
```

In [361…
```
rf = RandomForestClassifier()
rf.fit(X_train, y_train)
y_pred = rf.predict(X_test)
confusion_matrix(y_test, y_pred)
```

Out[361]:
```
array([[908,   0],
       [  0,  81]], dtype=int64)
```

Fine Tuning the HyperParameters

We use GridSearchCV to estimate the best hyperparameters to get the best parameters for better accuracy

In [363…
```
from sklearn.model_selection import GridSearchCV


# Define the parameter grid for grid search
param_grid = {
    'n_estimators': [100, 200, 300],  # Number of trees in the forest
    'max_depth': [None, 5, 10],       # Maximum depth of each tree
    # Minimum number of samples required to split an internal node
    'min_samples_split': [2, 5, 10],
    # Minimum number of samples required to be at a leaf node
    'min_samples_leaf': [1, 2, 4],
    # Number of features to consider at each split
    'random_state': [2]              # Random seed for reproducibility
}

# Create a RandomForestClassifier
rf = RandomForestClassifier()

# Perform grid search with cross-validation
grid_search = GridSearchCV(
```

```
    estimator=rf, param_grid=param_grid, cv=5, scoring='accuracy')
grid_search.fit(X_train, y_train)

# Get the best parameters and best score
best_params = grid_search.best_params_
best_score = grid_search.best_score_

# Train the RandomForestClassifier with the best parameters
best_rf = RandomForestClassifier(**best_params)
best_rf.fit(X_train, y_train)

# Make predictions on the test set
y_pred = best_rf.predict(X_test)

# Evaluate the model's performance
accuracy = accuracy_score(y_test, y_pred)
print("Best Parameters:", best_params)
print("Best Score:", best_score)
print("Test Set Accuracy:", accuracy)
```

```
Best Parameters: {'max_depth': None, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100, 'random_state': 2}
Best Score: 1.0
Test Set Accuracy: 1.0
```

In [367…

```
rf_model = RandomForestClassifier(max_depth=None, min_samples_leaf=1, min_samples_split=2, n_estimators=100, random_state=2)
rf_model.fit(X_train, y_train)
y_pred = rf_model.predict(X_test)
confusion_matrix(y_pred, y_test)
```

Out[367]:

```
array([[908,   0],
       [  0,  81]], dtype=int64)
```

This is the model and it seems to doing very great with very high accuracy and very high precision.

In [370…

```
# Exporting the model
import pickle

# Save the trained model to a file
with open('rf_model.pkl', 'wb') as file:
    pickle.dump(rf_model, file)
```

To load and use this model one can use:

```
import pickle

# Load the saved model from file
with open('rf_model.pkl', 'rb') as file:
    loaded_model = pickle.load(file)

# Use the loaded model for prediction
new_predictions = loaded_model.predict(new_data)
```