Standard Bank

Credit / Home Loans - Auto Exploratory Data Analysis

Standard Bank is embracing the digital transformation wave and intends to use new and exciting technologies to give their customers a complete set of services from the convenience of their mobile devices. As Africa's biggest lender by assets, the bank aims to improve the current process in which potential borrowers apply for a home loan. The current process involves loan officers having to manually process home loan applications. This process takes 2 to 3 days to process upon which the applicant will receive communication on whether or not they have been granted the loan for the requested amount. To improve the process Standard Bank wants to make use of machine learning to assess the credit worthiness of an applicant by implementing a model that will predict if the potential borrower will default on his/her loan or not, and do this such that the applicant receives a response immediately after completing their application.

There are many AutoEDA Python libraries out there which include:

- dtale (https://dtale.readthedocs.io/en/latest/)
- pandas profiling (https://pandas-profiling.ydata.ai/docs/master/index.html)
- sweetviz (https://pypi.org/project/sweetviz/)

and many more. In this task we will use Sweetviz.. You may be required to use bespoke EDA methods.

The Home Loans Department manager wants to know the following:

1. An overview of the data. (HINT: Provide the number of records, fields and their data types. Do for both).
2. What data quality issues exist in both train and test? (HINT: Comment any missing values and duplicates)
3. How do the the loan statuses compare? i.e. what is the distrubition of each?
4. How many of the loan applicants have dependents based on the historical dataset?
5. How do the incomes of those who are employed compare to those who are self employed based on the historical dataset?
6. Are applicants with a credit history more likely to default than those who do not have one?
7. Is there a correlation between the applicant's income and the loan amount they applied for?

Import Libraries

In [ ]:
```
#!pip install sweetviz
#uncomment the above if you need to install the library
#!pip install auto-sklearn
#uncomment the above if you need to install the library
```

In [ ]:
```
#!pip install --upgrade scipy
```

In [ ]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import sweetviz
import autosklearn.classification
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, confusion_matrix
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.impute import SimpleImputer
```

In [ ]:
```
import dtale
import ydata_profiling
import lux
from pandas_profiling import ProfileReport
```

```
/var/folders/s1/sz1sc8z54_z2_vdg29xpfblr0000gn/T/ipykernel_62713/363322112.py:4: DeprecationWarning:

`import pandas_profiling` is going to be deprecated by April 1st. Please use `import ydata_profiling` instead.
```

1. pandas: It provides data structures and functions for efficiently handling and analyzing data. It is commonly used for data manipulation and analysis tasks.
2. numpy: It is a library for numerical computing in Python. It provides support for large, multi-dimensional arrays and various mathematical functions.
3. matplotlib.pyplot: It is a plotting library used for creating visualizations, such as line plots, scatter plots, bar plots, histograms, etc.
4. seaborn: It is a data visualization library that builds on top of matplotlib. It provides a high-level interface for creating informative and visually appealing statistical graphics.
5. sweetviz: It is a library for visualizing and analyzing datasets. It generates detailed exploratory analysis reports, including summary statistics, distribution plots, correlation matrices, and more.
6. autosklearn.classification: It is a library for automated machine learning (AutoML). It provides a simple interface for automatically selecting and optimizing machine learning models for classification tasks.
7. sklearn.linear_model.LogisticRegression: It is a class from scikit-learn (sklearn) library that implements logistic regression, a popular algorithm for binary classification.
8. sklearn.metrics: It provides various evaluation metrics for assessing the performance of machine learning models, such as accuracy, precision, recall, and confusion matrix.
9. sklearn.model_selection.train_test_split: It is a function that splits a dataset into training and testing subsets. It is commonly used for evaluating the performance of machine learning models.
10. sklearn.preprocessing: It provides functions for preprocessing data, including scaling features, encoding categorical variables, and handling missing values.
11. sklearn.impute.SimpleImputer: It is a class from scikit-learn that provides strategies for imputing missing values in a dataset. It is used for filling in missing values with mean, median, or most frequent values.

Import Datasets

In [ ]:
```
train = pd.read_csv('train.csv')
test = pd.read_csv('test.csv')
```

Exploratory Data Analysis

Each row represents an individual's loan application, and each column represents a specific attribute or feature related to the loan application.

- Loan_ID: A unique identifier for each loan application.
- Gender: The gender of the loan applicant (e.g., Male or Female).
- Married: Indicates whether the applicant is married (Yes or No).
- Dependents: The number of dependents the applicant has.
- Education: Indicates the educational background of the applicant (e.g., Graduate or Not Graduate).
- Self_Employed: Indicates whether the applicant is self-employed (Yes or No).
- ApplicantIncome: The income of the applicant.
- CoapplicantIncome: The income of the co-applicant (if any).
- LoanAmount: The amount of the loan requested by the applicant.
- Loan_Amount_Term: The term or duration of the loan in months.
- Credit_History: Indicates the credit history of the applicant (e.g., 1.0 represents a good credit history, 0.0 represents a bad credit history).
- Property_Area: The area or location of the property associated with the loan application (e.g., Urban, Rural, or Semiurban).
- Loan_Status: The final status of the loan application (e.g., Y represents approved, N represents not approved).

The result shows a sample of loan application data, where each row provides information about a specific loan application, including the applicant's characteristics, loan details, and the loan's final status.

In [ ]:
```
train.head()
```

The result provided is a tabular representation of loan application data. Each row represents a specific loan application, and each column represents a different attribute or feature associated with the loan application.

- Loan_ID: A unique identifier for each loan application.
- Gender: The gender of the loan applicant (e.g., Male or Female).
- Married: Indicates whether the applicant is married (Yes or No).
- Dependents: The number of dependents the applicant has.
- Education: Indicates the educational background of the applicant (e.g., Graduate or Not Graduate).
- Self_Employed: Indicates whether the applicant is self-employed (Yes or No).
- ApplicantIncome: The income of the applicant.
- CoapplicantIncome: The income of the co-applicant (if any).
- LoanAmount: The amount of the loan requested by the applicant.
- Loan_Amount_Term: The term or duration of the loan in months.
- Credit_History: Indicates the credit history of the applicant (e.g., 1.0 represents a good credit history, 0.0 represents a bad credit history).
- Property_Area: The area or location of the property associated with the loan application (e.g., Urban, Rural, or Semiurban).

The result shows a subset of loan application data, where each row provides information about a specific loan application, including the applicant's characteristics, loan details, and related information.

In [ ]:
```
test.head()
```

The script below shows the concatenation of two DataFrames, `train` and `test`, along the row axis (axis=0).

The code snippet calculates the number of rows in the `train` DataFrame using the `shape` attribute and accessing the first element of the tuple returned by `shape[0]`. This value is assigned to the variable `n`.

Then, the `pd.concat()` function is used to concatenate the `train` and `test` DataFrames along the row axis (axis=0). The resulting concatenated DataFrame is assigned to the variable `df`.

The `df.head()` function is called to display the first few rows of the concatenated DataFrame (`df`). This provides a preview of the merged dataset, allowing you to inspect the structure and content of the combined data.

In [ ]:
```
n = train.shape[0]
df = pd.concat([train, test], axis=0)
df.head()
```

SWEETVIZ

In [ ]:
```
autoEDA = sweetviz.analyze(train)
autoEDA.show_notebook()
```

In [ ]:
```
autoEDA = sweetviz.analyze(test)
autoEDA.show_notebook()
```

DTALE

In [ ]:
```
# Using dtale
d = dtale.show(train)
d.open_browser()
```

```
/usr/local/lib/python3.9/site-packages/dtale/views.py:756: FutureWarning:['Gender', 'Married', 'Dependents', 'Self_Employed'] did not aggregate successfully. If any error is raised this will raise in a future version of pandas. Drop these columns/ops to avoid this warning.
```

In [ ]:
```
# Using dtale
d = dtale.show(test)
d.open_browser()
```

```
/usr/local/lib/python3.9/site-packages/dtale/views.py:756: FutureWarning:['Gender', 'Dependents', 'Self_Employed'] did not aggregate successfully. If any error is raised this will raise in a future version of pandas. Drop these columns/ops to avoid this warning.
```

PANDAS PROFILING

In [ ]:
```
# Using pandas-profiling
profile = ProfileReport(train)
profile.to_file(output_file='train_report.html')  # Saves the report to an HTML file
```

In [ ]:
```
# Using pandas-profiling
profile = ProfileReport(test)
profile.to_file(output_file='test_report.html')  # Saves the report to an HTML file
```

An overview of the data. (HINT: Provide the number of records, fields and their data types. Do for both).

In [ ]:
```
# 1. Overview of the data
print("Train dataset:")
print("Number of records:", train.shape[0])
print("Number of fields:", train.shape[1])
print("Data types:")
print(train.dtypes)
```

```
Train dataset:
Number of records: 614
Number of fields: 13
Data types:
Loan_ID               object
Gender                object
Married               object
Dependents            object
Education             object
Self_Employed         object
ApplicantIncome        int64
CoapplicantIncome    float64
LoanAmount           float64
Loan_Amount_Term     float64
Credit_History       float64
Property_Area         object
Loan_Status           object
dtype: object
```

The result describes the Train dataset, which contains 614 records and 13 fields. Here are the data types of each field:

- `Loan_ID` : Object (likely a unique identifier for each loan application)
- `Gender` : Object (categorical variable representing the gender of the applicant)
- `Married` : Object (categorical variable indicating whether the applicant is married or not)
- `Dependents` : Object (categorical variable representing the number of dependents the applicant has)
- `Education` : Object (categorical variable indicating the educational background of the applicant)
- `Self_Employed` : Object (categorical variable indicating whether the applicant is self-employed or not)
- `ApplicantIncome` : Integer (numerical variable representing the income of the applicant)
- `CoapplicantIncome` : Float (numerical variable representing the income of the co-applicant)
- `LoanAmount` : Float (numerical variable representing the loan amount)
- `Loan_Amount_Term` : Float (numerical variable representing the term of the loan in months)
- `Credit_History` : Float (numerical variable representing the credit history of the applicant)
- `Property_Area` : Object (categorical variable indicating the property area of the applicant)
- `Loan_Status` : Object (categorical variable indicating the approval status of the loan)

The data types provide information about how each field is represented in the dataset, whether as categorical variables (objects) or numerical variables (integers or floats).

In [ ]:
```
print("\nTest dataset:")
print("Number of records:", test.shape[0])
print("Number of fields:", test.shape[1])
print("Data types:")
print(test.dtypes)
```

```
Test dataset:
Number of records: 367
Number of fields: 12
Data types:
Loan_ID               object
Gender                object
Married               object
Dependents            object
Education             object
Self_Employed         object
ApplicantIncome        int64
CoapplicantIncome      int64
LoanAmount           float64
Loan_Amount_Term     float64
Credit_History       float64
Property_Area         object
dtype: object
```

The result describes the Test dataset, which contains 367 records and 12 fields. Here are the data types of each field:

- `Loan_ID` : Object (likely a unique identifier for each loan application)
- `Gender` : Object (categorical variable representing the gender of the applicant)
- `Married` : Object (categorical variable indicating whether the applicant is married or not)
- `Dependents` : Object (categorical variable representing the number of dependents the applicant has)
- `Education` : Object (categorical variable indicating the educational background of the applicant)
- `Self_Employed` : Object (categorical variable indicating whether the applicant is self-employed or not)
- `ApplicantIncome` : Integer (numerical variable representing the income of the applicant)
- `CoapplicantIncome` : Integer (numerical variable representing the income of the co-applicant)
- `LoanAmount` : Float (numerical variable representing the loan amount)
- `Loan_Amount_Term` : Float (numerical variable representing the term of the loan in months)
- `Credit_History` : Float (numerical variable representing the credit history of the applicant)
- `Property_Area` : Object (categorical variable indicating the property area of the applicant)

The data types provide information about how each field is represented in the dataset, whether as categorical variables (objects) or numerical variables (integers or floats).

What data quality issues exist in both train and test? (HINT: Comment any missing values and duplicates)

In [ ]:
```
# 2. Data quality issues
print("\nData quality issues in train dataset:")
# Check for missing values
print("Missing values:")
print(train.isnull().sum())

# Check for duplicates
print("Duplicates:")
print(train.duplicated().sum())
```

```
Data quality issues in train dataset:
Missing values:
Loan_ID               0
Gender               13
Married               3
Dependents           15
Education             0
Self_Employed        32
ApplicantIncome       0
CoapplicantIncome     0
LoanAmount           22
Loan_Amount_Term     14
Credit_History       50
Property_Area         0
Loan_Status           0
dtype: int64
Duplicates:
0
```

In [ ]:
```
print("\nData quality issues in test dataset:")
# Check for missing values
print("Missing values:")
print(test.isnull().sum())

# Check for duplicates
print("Duplicates:")
print(test.duplicated().sum())
```

```
Data quality issues in test dataset:
Missing values:
Loan_ID               0
Gender               11
Married               0
Dependents           10
Education             0
Self_Employed        23
ApplicantIncome       0
CoapplicantIncome     0
LoanAmount            5
Loan_Amount_Term      6
Credit_History       29
Property_Area         0
dtype: int64
Duplicates:
0
```

The result describes the data quality issues in the Train and Test datasets. Here's the information:

Train Dataset:

- Missing values:
  - `Gender` : 13 missing values
  - `Married` : 3 missing values
  - `Dependents` : 15 missing values
  - `Self_Employed` : 32 missing values
  - `LoanAmount` : 22 missing values
  - `Loan_Amount_Term` : 14 missing values
  - `Credit_History` : 50 missing values
- Duplicates: There are no duplicates in the Train dataset.

Test Dataset:

- Missing values:
  - `Gender` : 11 missing values
  - `Married` : 0 missing values
  - `Dependents` : 10 missing values
  - `Self_Employed` : 23 missing values
  - `LoanAmount` : 5 missing values
  - `Loan_Amount_Term` : 6 missing values
  - `Credit_History` : 29 missing values
- Duplicates: There are no duplicates in the Test dataset.

The information provided highlights the number of missing values for each field in both datasets and indicates whether there are any duplicates present. Missing values may need to be addressed by applying techniques such as imputation or removal, depending on the specific requirements of the analysis or modeling task.

How do the the loan statuses compare? i.e. what is the distrubition of each?

In [ ]:
```
# 3. Loan status distribution
print("\nLoan status distribution:")
print(train['Loan_Status'].value_counts())
```

```
Loan status distribution:
Y    422
N    192
Name: Loan_Status, dtype: int64
```

The result describes the distribution of loan statuses in the dataset. Here's the information:

Loan status distribution:

- Y (Approved): 422 instances
- N (Not Approved): 192 instances

This indicates that out of the total number of loan applicants in the dataset, 422 loans were approved (Y) and 192 loans were not approved (N). The distribution provides an overview of the loan approval outcomes in the dataset.

How many of the loan applicants have dependents based on the historical dataset?

In [ ]:
```
# 4. Number of loan applicants with dependents
print("\nNumber of loan applicants with dependents:")
print(train['Dependents'].value_counts())
```

```
Number of loan applicants with dependents:
0     345
1     102
2     101
3+     51
Name: Dependents, dtype: int64
```

The result describes the number of loan applicants with different numbers of dependents.

Number of loan applicants with dependents:

- 0 dependents: 345 applicants
- 1 dependent: 102 applicants
- 2 dependents: 101 applicants
- 3+ dependents: 51 applicants

This indicates the distribution of loan applicants based on the number of dependents they have. The majority of loan applicants (345) do not have any dependents, followed by 102 applicants with 1 dependent, 101 applicants with 2 dependents, and 51 applicants with 3 or more dependents.

How do the incomes of those who are employed compare to those who are self employed based on the historical dataset?

In [ ]:
```
# 5. Comparison of incomes based on employment type
print("\nComparison of incomes based on employment type:")
print(train.groupby('Self_Employed')['ApplicantIncome'].mean())
```

```
Comparison of incomes based on employment type:
Self_Employed
No     5049.748000
Yes    7380.817073
Name: ApplicantIncome, dtype: float64
```

The result describes the comparison of incomes based on the employment type (self-employed or not self-employed). Here's the information:

Comparison of incomes based on employment type:

- Self_Employed: The average applicant income for individuals who are self-employed is approximately $7,380.82.
- Not Self_Employed: The average applicant income for individuals who are not self-employed is approximately $5,049.75.

This indicates that, on average, self-employed individuals have a higher income compared to those who are not self-employed.

Are applicants with a credit history more likely to default than those who do not have one?

In [ ]:
```
# 6. Default rate based on credit history
print("\nDefault rate based on credit history:")
default_rate = train[train['Loan_Status'] == 'N']['Credit_History'].value_counts(normalize=True)
print(default_rate)
```

```
Default rate based on credit history:
1.0    0.541899
0.0    0.458101
Name: Credit_History, dtype: float64
```

The result describes the default rate based on credit history. Default rate based on credit history:

- Credit_History: The default rate for individuals with a credit history of 1.0 is approximately 54.19%.
- Credit_History: The default rate for individuals with a credit history of 0.0 is approximately 45.81%.

This indicates that individuals with a credit history of 1.0 have a higher default rate compared to those with a credit history of 0.0.

Is there a correlation between the applicant's income and the loan amount they applied for?

In [ ]:
```
# 7. Correlation between income and loan amount
print("\nCorrelation between income and loan amount:")
correlation = train['ApplicantIncome'].corr(train['LoanAmount'])
print("Correlation coefficient:", correlation)
```

```
Correlation between income and loan amount:
Correlation coefficient: 0.5709090389885666
```

```
/usr/local/lib/python3.9/site-packages/dtale/views.py:2010: FutureWarning:The default value of regex will change from True to False in a future version.
/usr/local/lib/python3.9/site-packages/lux/core/frame.py:193: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
```

The result indicates the correlation between income and loan amount. The correlation coefficient is 0.5709090389885666.

The correlation coefficient measures the strength and direction of the linear relationship between two variables. In this case, the correlation coefficient of 0.5709090389885666 suggests a moderate positive correlation between income and loan amount. This means that as the income increases, the loan amount tends to increase as well, and vice versa. However, it's important to note that correlation does not imply causation, and other factors may also influence the relationship between income and loan amount.