Introduction

Technology has captured most of the fields in present and same goes for analyzing, modelling of data, with the advancement of the technology nowadays we can also predict the future results of our datasets using several machine learning algorithms and high level programming languages has made the job easier for example using several libraries in Python we can easily solve the problems of the huge datasets. The very first method of preserving the data was writing and it was Sumerians who invented writing around 3400 BC, imagine how difficult the world would be without the breakthrough of Sumerians with the only storage device “human brain” which for me is very unreliable storage for storing large data because human brain may have consciousness but its memory is very unreliable as compared to the writing or other form of modern storage devices. Keeping the following knowledge in mind, I became excited to solve the problem of “Loan approval dataset” which I found fascinating because of its importance in financial market and it is also helpful in analyzing which variable is highly correlated with the approval of loan and it is also fun to predict the future approval or disapproval of the loans. The dataset which I have used in this project is about the loan prediction, the goal of employing AI in this situation is to increase the speed and precision of the loan approval process. The manual assessment of applications used in traditional loan approval procedures can be time-consuming and error-prone. It may be able to automate some of the approval process and develop more precise predictions about which applications are most likely to be approved by utilising AI and machine learning techniques. To minimize bias and ensure fairness in the loan approval procedure is another justification for employing AI in this situation. Large datasets can be used to train AI models so they can find trends and make predictions based on data rather than opinions or biases. Overall, this dataset offers an exciting and timely challenge for the use of AI methods. It might be able to increase the efficiency and equity of lending by creating fair and accurate predictive models for loan acceptance, which would be advantageous to both lenders and borrowers.

In [222…
```
import numpy as np
import pandas as pd
from fancyimpute import IterativeImputer
import matplotlib.pyplot as plt
import seaborn as sns
from impyute.imputation.cs import mice
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LogisticRegression
from sklearn.metrics import accuracy_score, classification_report
from sklearn.metrics import accuracy_score, mean_squared_error
from sklearn.neighbors import KNeighborsClassifier
from sklearn.ensemble import RandomForestClassifier
```

In [223…
```
# Load CSV file into a DataFrame
df = pd.read_csv('loan_data.csv')
```

In [224…
```
# Display the first 5 rows of the DataFrame
print(df.head())
```

```
    Loan_ID Gender Married Dependents     Education Self_Employed  \
0  LP001002   Male      No          0      Graduate            No
1  LP001003   Male     Yes          1      Graduate            No
2  LP001005   Male     Yes          0      Graduate           Yes
3  LP001006   Male     Yes          0  Not Graduate            No
4  LP001008   Male      No          0      Graduate            No

   ApplicantIncome  CoapplicantIncome  LoanAmount  Loan_Amount_Term  \
0             5849                0.0         NaN             360.0
1             4583             1508.0       128.0             360.0
2             3000                0.0        66.0             360.0
3             2583             2358.0       120.0             360.0
4             6000                0.0       141.0             360.0

   Credit_History Property_Area Loan_Status
0             1.0         Urban           Y
1             1.0         Rural           N
2             1.0         Urban           Y
3             1.0         Urban           Y
4             1.0         Urban           Y
```

In [225…
```
df.head()
```

Out[225…

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantInc
0	LP001002	Male	No	0	Graduate	No	5849	
1	LP001003	Male	Yes	1	Graduate	No	4583	15
2	LP001005	Male	Yes	0	Graduate	Yes	3000	
3	LP001006	Male	Yes	0	Not Graduate	No	2583	23
4	LP001008	Male	No	0	Graduate	No	6000	

In [226…
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 13 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Loan_ID            614 non-null    object
 1   Gender             601 non-null    object
 2   Married            611 non-null    object
 3   Dependents         599 non-null    object
 4   Education          614 non-null    object
 5   Self_Employed      582 non-null    object
 6   ApplicantIncome    614 non-null    int64
 7   CoapplicantIncome  614 non-null    float64
 8   LoanAmount         592 non-null    float64
 9   Loan_Amount_Term   600 non-null    float64
 10  Credit_History     564 non-null    float64
 11  Property_Area      614 non-null    object
 12  Loan_Status        614 non-null    object
dtypes: float64(4), int64(1), object(8)
memory usage: 62.5+ KB
```

Dataset

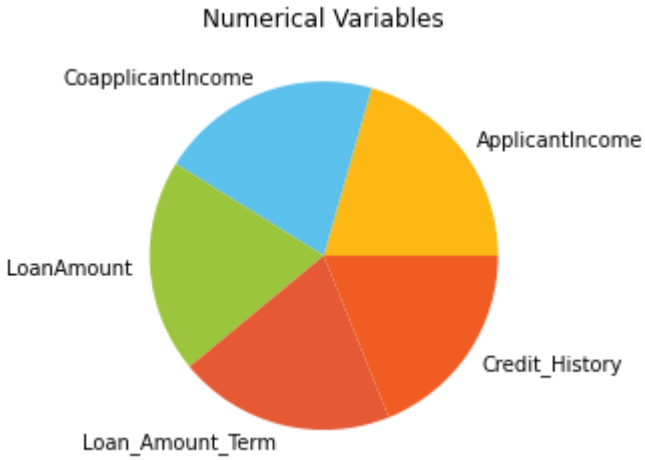
The dataset which I have used in this project is sourced from the famous data science competition website namely “Kaggle”, this website provides several data science challenges for its users, the problem addressed in this data set is to predict the future status of whether the loan will be approved or not.

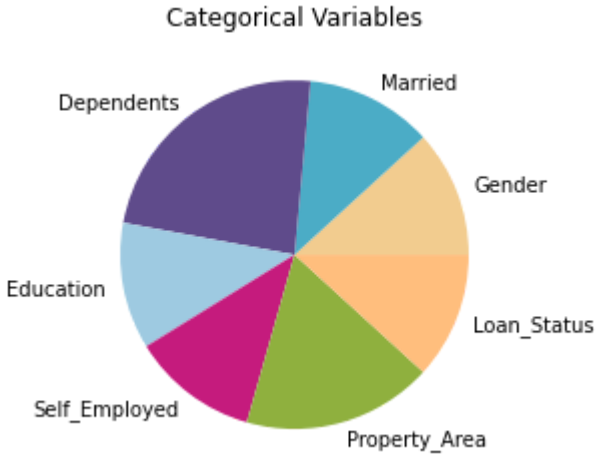
Description of Data

The total rows or observations in this dataset are 614 and it has 13 features or 13 columns. • Loan_ID: A unique ID for each applicant for the loan. • Gender: The sex of the applicant. • Married: The marital status of the applicant. • Dependents: Number of dependents of the applicant. • Education: The level of education of the applicant. • Self_Employed: Employability status, whether the applicant is self-employed or not. • ApplicantIncome: The income of the application or how much the applicant earns. • CoapplicantIncome: The income of the co-applicant. • LoanAmount: the loan amount requested by the applicant • Loan_Amount_Term: the term (in months) of the loan • Credit_History: a binary variable indicating whether the applicant has a credit history or not (1 = Yes, 0 = No) • Property_Area: the type of property for which the loan is being applied (Urban/Semiurban/Rural) • Loan_Status: the binary target variable indicating whether the loan was approved or not (Y = Yes, N = No).

In [227…

```
#figure for the categorical and non categorical data
# Define categorical and numerical columns
cat_cols = ['Gender', 'Married', 'Dependents', 'Education', 'Self_Employed', 'Property_
num_cols = ['ApplicantIncome', 'CoapplicantIncome', 'LoanAmount', 'Loan_Amount_Term', '

# Set colors for numerical and categorical variables
num_colors = ["#FDB813", "#5BC0EB", "#9BC53D", "#E55934", "#F15C22"]
cat_colors = ["#F2CC8F", "#4BACC6", "#5F4B8B", "#9ECAE1", "#C51B7D", "#8FB03E", "#FFBE7

# Plot pie chart for numerical variables
fig, ax = plt.subplots()
num_vals = [df[col].count() for col in num_cols]
num_labels = [col for col in num_cols]
ax.pie(num_vals, labels=num_labels, colors=num_colors)
ax.set_title("Numerical Variables")

# Plot pie chart for categorical variables
fig, ax = plt.subplots()
cat_vals = [df[col].nunique() for col in cat_cols]
cat_labels = [col for col in cat_cols]
ax.pie(cat_vals, labels=cat_labels, colors=cat_colors)
ax.set_title("Categorical Variables")

plt.show()
```





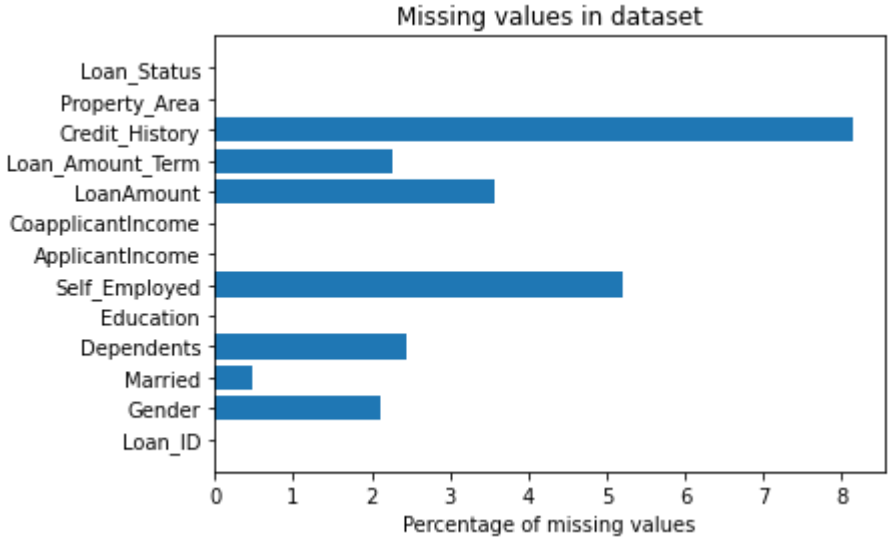
Like most of the datasets this dataset also contains the both categorical and numerical data. Above pie charts show the division of the numerical and categorical variables in our respective dataset. From the above figures we can see that we have five numeric variables namely CoapplicantIncome, ApplicantIncome, Credit_History, Loan_Amount_Term, Loan_Amount and CoapplicantIncome and we have seven categorical variables namely Dependents, Education, Self_Employed, Property_Area, Loan_Status, Genger and Married.

In [228…
```
#finding the percentage of missing values
missing_percentages = df.isnull().mean() * 100
print(missing_percentages)
```

```
Loan_ID              0.000000
Gender               2.117264
Married              0.488599
Dependents           2.442997
Education            0.000000
Self_Employed        5.211726
ApplicantIncome      0.000000
CoapplicantIncome    0.000000
LoanAmount           3.583062
Loan_Amount_Term     2.280130
Credit_History       8.143322
Property_Area        0.000000
Loan_Status          0.000000
dtype: float64
```

In [229…
```
# Plot missing values
# Create a bar chart to visualize the missing values
```

```
fig, ax = plt.subplots()
ax.barh(missing_percentages.index, missing_percentages.values)
ax.set_xlabel("Percentage of missing values")
ax.set_title("Missing values in dataset")
plt.show()
```



Missing Values

Most of the datasets around the globe are not completely fill with the require values and presence of the missing values have been a problem to be solved for most of the data scientists around the world and like most of the datasets our dataset also contains the missing values and to get the best results it is necessary for every data scientist to deal with the missing values therefore it is necessary for the data scientist to deal with the missing values and the missing values could be dealt by filling the empty spaces or delete the columns which contains for small dataset more than 50% missing values and for very large datasets more than 70% missing values, figure above shows the percentage of the missing values in each feature of our dataset. The figure above shows the number of missing values in each feature according to percentages, Gender contains 2.11% of missing values, Married contains 0.488% of missing values, Dependents contains 2.44% of missing values, Self_Employed contains 5.211% of missing values, LoanAmount contains 3.58% of missing values, Loan_Amount_Term contains 2.280% of missing values and Credit_History contains 8.14% of missing values in the dataset.

Imputation of the Missing Values

As mentioned before most of the datasets around the world contain missing values and this may be because of the human error while data entry and many other reasons, for every dataset containing missing values, it is the job of data scientist to deal with those missing, for columns containing more than 50% of the missing values is usually removed from the dataset. There are several methods for imputing the missing values in the dataset and there are different methods for categorical and numerical data, for our categorical data we have used mode imputation which means we have replaced the null with the most frequent values of that specific feature and we have used mode

imputation because it is one of the most frequently used imputation method for categorical data, the implementation of mode imputation is same simple and it is most suitable form of imputation for small datasets and for our numerical variables we have used MICE(Multiple imputation by chained equation) because using mice we can preserve the variation of the dataset by creating multiple imputed datasets, both of these imputation proved to be useful because the accuracy in the end was good.

In [230…
```
df = df.drop('Loan_ID',axis=1)
```

In [ ]:
```
```

In [231…
```
# looking for the percentage in Loan ID column using percentage
df['Gender'].isnull().mean() * 100
```

Out[231…
```
2.1172638436482085
```

In [232…
```
#imputing missing values in gender using mode imputation
gender_mode = df['Gender'].mode()[0]
df['Gender'] = df['Gender'].fillna(gender_mode)
```

In [233…
```
#imputing missing values in Married column using mode imputation
married_mode = df['Married'].mode()[0]  #
df['Married'] = df['Married'].fillna(married_mode)
```

In [234…
```
#imputing missing values in dependent column using mode imputation
dependents_mode = df['Dependents'].mode()[0]
df['Dependents'] = df['Dependents'].fillna(gender_mode)
```

In [235…
```
#imputing missing values in Self_Employed using mode imputation
Self_Employed_mode = df['Self_Employed'].mode()[0]
df['Self_Employed'] = df['Self_Employed'].fillna(Self_Employed_mode)
```

In [236…
```
# Impute missing values in the "LoanAmount" column using MICE
imputer = IterativeImputer()
df['LoanAmount'] = imputer.fit_transform(df[['LoanAmount']])
```

In [237…
```
# Impute missing values in the "Loan_Amount-Term" column using MICE
imputer = IterativeImputer()
df['Loan_Amount_Term'] = imputer.fit_transform(df[['Loan_Amount_Term']])
```

In [238…
```
# Impute missing values in the "Credit_History" column using MICE
imputer = IterativeImputer()
df['Credit_History'] = imputer.fit_transform(df[['Credit_History']])
```

In [239…
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Gender             614 non-null    object
 1   Married            614 non-null    object
 2   Dependents         614 non-null    object
```

```
 3   Education          614 non-null    object
 4   Self_Employed      614 non-null    object
 5   ApplicantIncome    614 non-null    int64
 6   CoapplicantIncome  614 non-null    float64
 7   LoanAmount         614 non-null    float64
 8   Loan_Amount_Term   614 non-null    float64
 9   Credit_History     614 non-null    float64
 10  Property_Area      614 non-null    object
 11  Loan_Status        614 non-null    object
dtypes: float64(4), int64(1), object(7)
memory usage: 57.7+ KB
```

In [240…
```
df.isnull().sum()
```

Out[240…
```
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History       0
Property_Area        0
Loan_Status          0
dtype: int64
```

In [241…
```
# imputing missing values in Loan_Amount_Term and credit_history using mice because its
imputer = SimpleImputer(strategy='mean')
df['Loan_Amount_Term'] = imputer.fit_transform(df[['Loan_Amount_Term']])

df.isnull().sum()
```

Out[241…
```
Gender               0
Married              0
Dependents           0
Education            0
Self_Employed        0
ApplicantIncome      0
CoapplicantIncome    0
LoanAmount           0
Loan_Amount_Term     0
Credit_History       0
Property_Area        0
Loan_Status          0
dtype: int64
```

In [242…
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 614 entries, 0 to 613
Data columns (total 12 columns):
 #   Column             Non-Null Count  Dtype
---  ------             --------------  -----
 0   Gender             614 non-null    object
 1   Married            614 non-null    object
 2   Dependents         614 non-null    object
 3   Education          614 non-null    object
 4   Self_Employed      614 non-null    object
 5   ApplicantIncome    614 non-null    int64
 6   CoapplicantIncome  614 non-null    float64
 7   LoanAmount         614 non-null    float64
 8   Loan_Amount_Term   614 non-null    float64
```

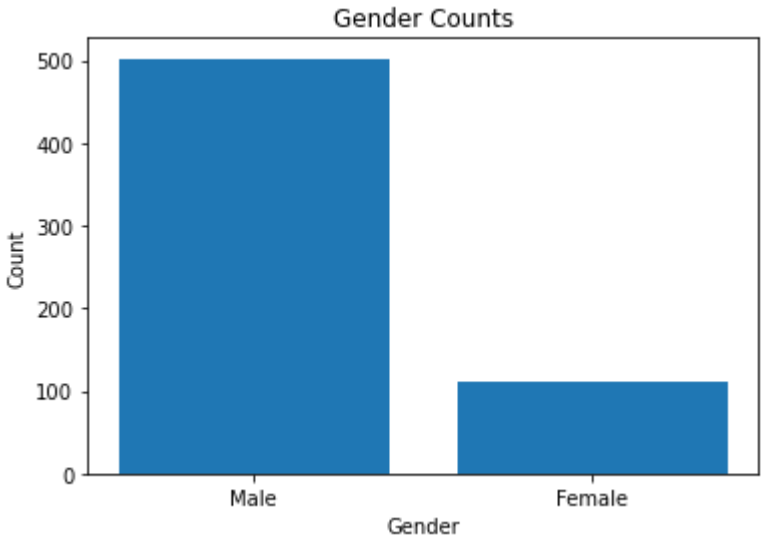
```
 9   Credit_History     614 non-null    float64
 10  Property_Area      614 non-null    object
 11  Loan_Status        614 non-null    object
dtypes: float64(4), int64(1), object(7)
memory usage: 57.7+ KB
```

Mapping

It is obvious that to perform machine learning algorithm we cannot use the categorical data therefore we will perform the mapping for our categorical data using panda's method called map(), mapping is a process in which we convert our categorical data into numerical data, .astype('int') performs this operation of converting the strings into int, we cannot use categorical data for machine learning algorithms because they require numerical data for input. As we know most of the features in our dataset are numerical so they need to be converted into numerical data for example for feature gender we have converted male into 1 and female into 0, likewise for the column marital status we have converted yes into 1 and No into 0.

In [243…
```
# Data visualization and mapping of the categorical dat
gender_counts = df['Gender'].value_counts()
print(gender_counts)
# Create a bar plot of the gender counts
plt.bar(gender_counts.index, gender_counts.values)
# Set the plot title and axis labels
plt.title('Gender Counts')
plt.xlabel('Gender')
plt.ylabel('Count')

# Display the plot
plt.show()
```

```
Male      502
Female    112
Name: Gender, dtype: int64
```

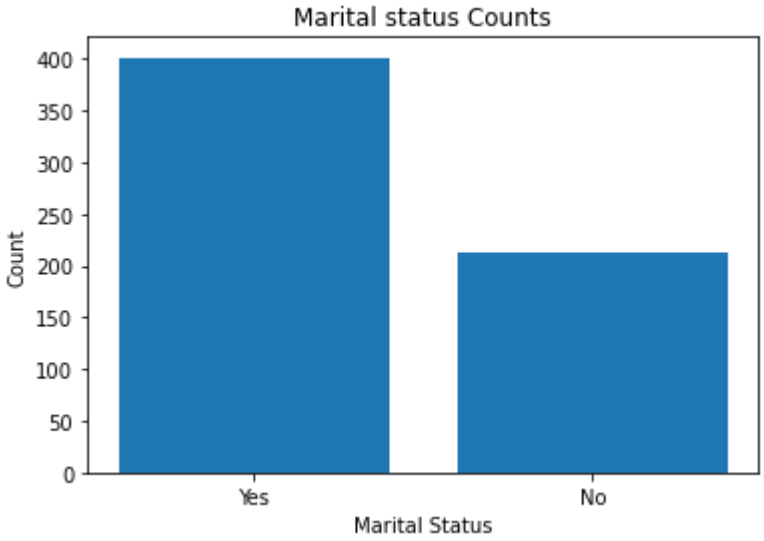


In [244…
```
#converting the values in gender column to numerical
df['Gender'] = df['Gender'].map({'Male':1, 'Female': 0}).astype('int')
```

In [245…
```
# barplot for married column
married_counts = df['Married'].value_counts()
```

```
print(married_counts)
# Create a bar plot of the married counts
plt.bar(married_counts.index, married_counts.values)
# Set the plot title and axis labels
plt.title('Marital status Counts')
plt.xlabel('Marital Status')
plt.ylabel('Count')

# Display the plot
plt.show()
```

```
Yes    401
No     213
Name: Married, dtype: int64
```



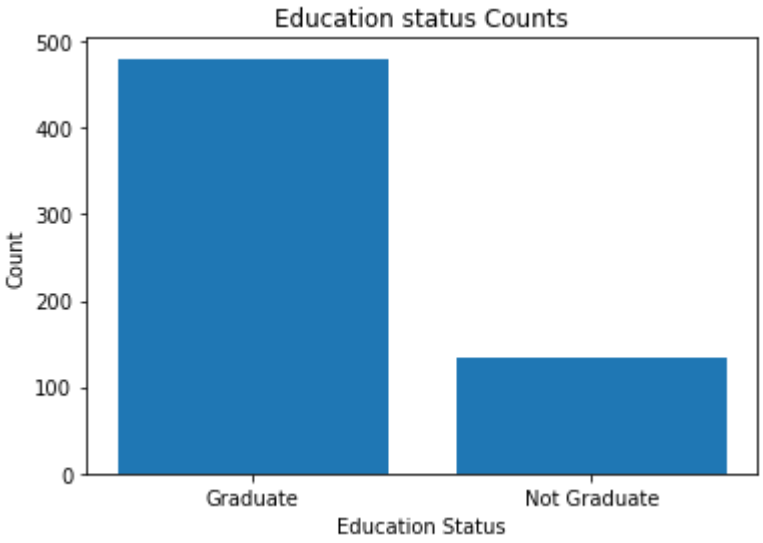
In [246…

```
#converting the values in married column to integer
df['Married'] = df['Married'].map({'Yes':1, 'No': 0}).astype('int')
```

In [247…

```
# barplot for Education column
Education_counts = df['Education'].value_counts()
print(Education_counts)
# Create a bar plot of the Education counts
plt.bar(Education_counts.index, Education_counts.values)
# Set the plot title and axis labels
plt.title('Education status Counts')
plt.xlabel('Education Status')
plt.ylabel('Count')

# Display the plot
plt.show()
```

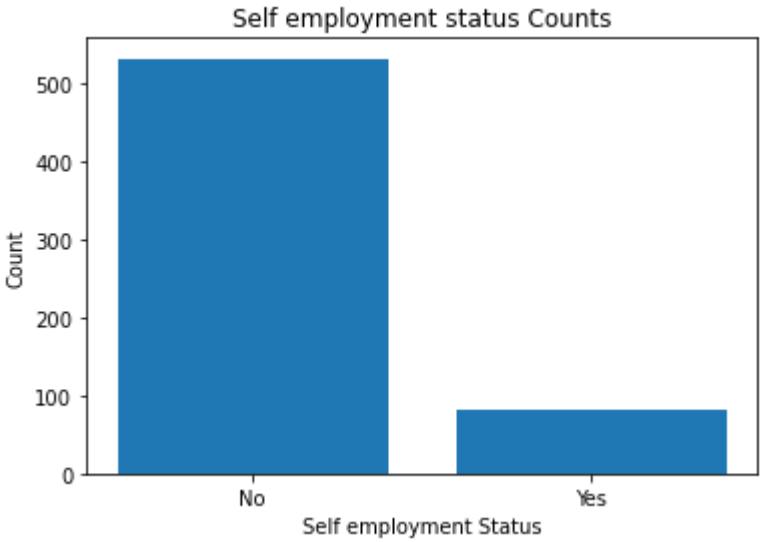
```
Graduate        480
Not Graduate    134
Name: Education, dtype: int64
```



In [248…
```
#converting the values in Education column to integer
df['Education'] = df['Education'].map({'Graduate':1, 'Not Graduate': 0}).astype('int')
```

In [249…
```
# Barplot for self_employed column
selfemp_counts = df['Self_Employed'].value_counts()
print(selfemp_counts)
# Create a bar plot of the employment counts
plt.bar(selfemp_counts.index, selfemp_counts.values)
# Set the plot title and axis labels
plt.title('Self employment status Counts')
plt.xlabel('Self employment Status')
plt.ylabel('Count')

# Display the plot
plt.show()
```

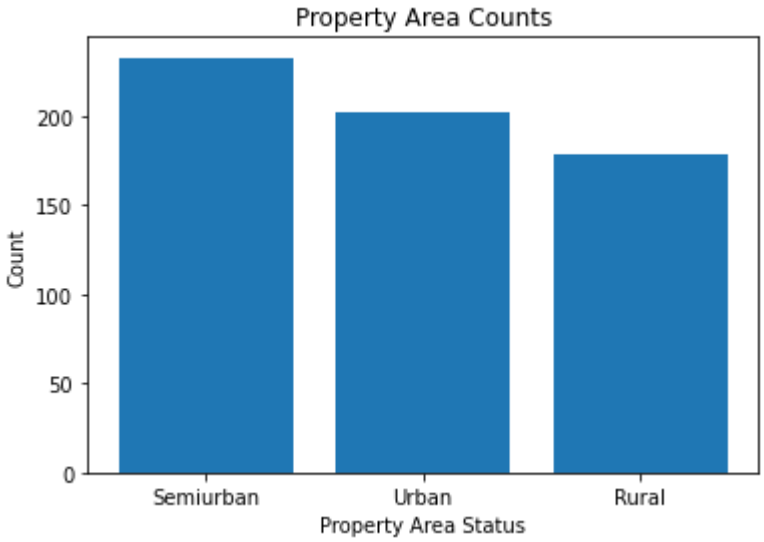
```
No     532
Yes     82
Name: Self_Employed, dtype: int64
```



In [250…
```
#converting the values in Self_Employed column to integer
df['Self_Employed'] = df['Self_Employed'].map({'Yes':1, 'No': 0}).astype('int')
```

In [251…
```
# Barplot for Property_Area column
PropArea_counts = df['Property_Area'].value_counts()
print(PropArea_counts)
# Create a bar plot of the PropArea counts
plt.bar(PropArea_counts.index, PropArea_counts.values)
# Set the plot title and axis labels
plt.title('Property Area Counts')
plt.xlabel('Property Area Status')
plt.ylabel('Count')

# Display the plot
plt.show()
```

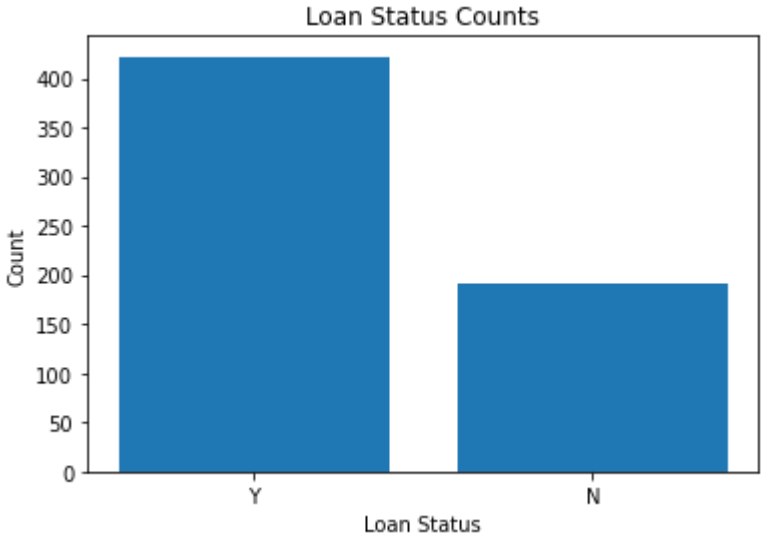
```
Semiurban    233
Urban        202
Rural        179
Name: Property_Area, dtype: int64
```



In [252…
```
# converted property area to numerical
df['Property_Area'].unique()
df['Property_Area'] = df['Property_Area'].map({'Urban':1, 'Semiurban': 2, 'Rural': 0}).
```

In [253…
```
# Barplot for Loan_Status column
loan_counts = df['Loan_Status'].value_counts()
print(loan_counts)
# Create a bar plot of the Loan_status counts
plt.bar(loan_counts.index, loan_counts.values)
# Set the plot title and axis labels
plt.title('Loan Status Counts')
plt.xlabel('Loan Status')
plt.ylabel('Count')

# Display the plot
plt.show()
```

```
Y    422
N    192
Name: Loan_Status, dtype: int64
```



In [254…
```
#converting loan status variable to numerical
df['Loan_Status'] = df['Loan_Status'].map({'Y':1, 'N': 0}).astype('int')
```

In [255…
```
df['Dependents'] = df['Dependents'].replace(to_replace = '3+', value = '4')
```

In [256…
```
df['Dependents'] = df['Dependents'].replace('Male', '0')
df['Dependents'] = df['Dependents'].astype('float')
```

In [257…
```
print(df['Dependents'].unique())
```

```
[0. 1. 2. 4.]
```

X and Y variable

For applying machine learning algorithms on any dataset it is common to split our dataset into x and y, which means that x contains all of the dataset except y or "Loan_Status" which is our response variable. This is mostly done to prepare the data for the machine learning algorithms which divides the variable between input and output, in our case x is our input and y is our output.

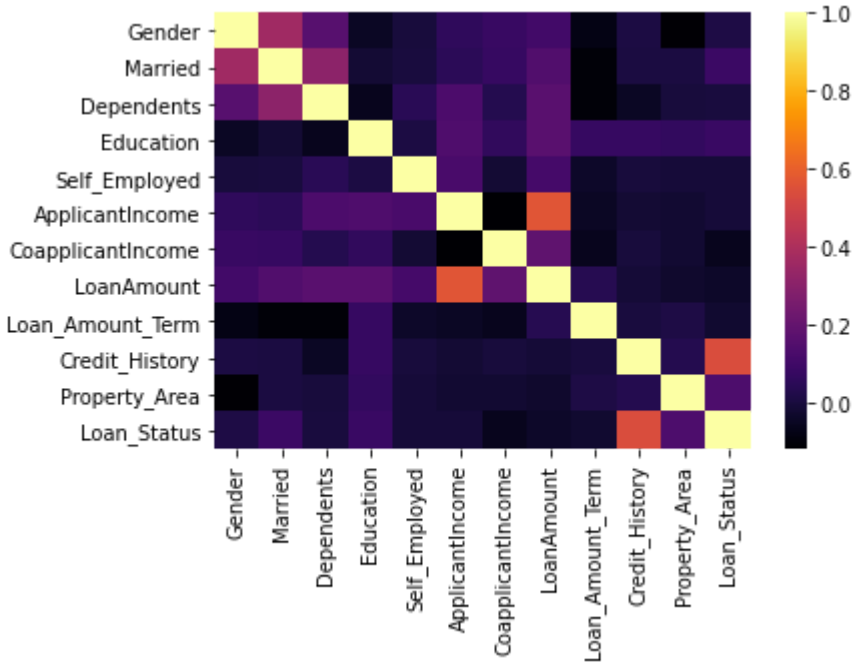
In [258…
```
# storing dependent variables in x and response or target variable in y
x = df.drop('Loan_Status',axis=1)
y = df['Loan_Status']
```

Correlation

Correlation in dataset is the description of how the variables are related to each other for instance if the value of one feature is increased then how it effects the other features, figure below shows the correlation in our dataset. From the figure below we can examine that variables Loan_Status is highly correlated with Credit_History, ApplicantIncome is highly correlated with LoanAmount.

In [259…
```
#Making a correlation plot
corr = df.corr()
```

```
sns.heatmap(corr, cmap='inferno')
plt.show()
```



In [260…
```
x.head()
```

Out[260…

	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	CoapplicantIncome	Loan
0	1	0	0.0	1	0	5849	0.0	146
1	1	1	1.0	1	0	4583	1508.0	128
2	1	1	0.0	1	1	3000	0.0	66
3	1	1	0.0	0	0	2583	2358.0	120
4	1	0	0.0	1	0	6000	0.0	141

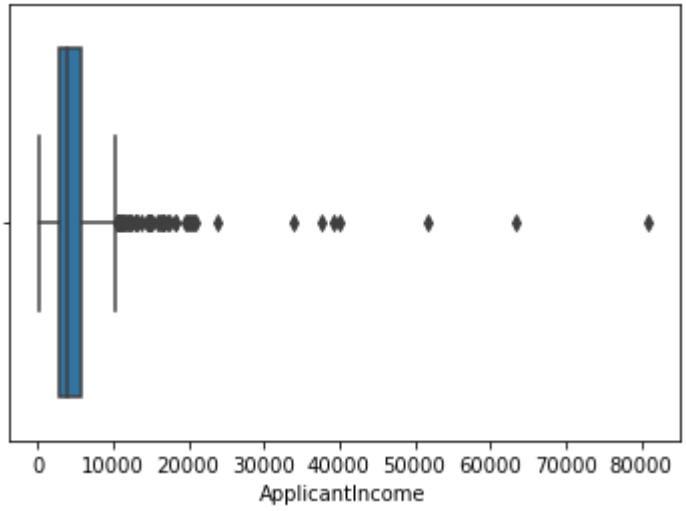
In [261…
```
#do deal with the outlier we will use scalling method for variables which may contain o
#first we will examine the outliers in these columns
#we store variables with outliers in list
my_list = ['ApplicantIncome','CoapplicantIncome','LoanAmount','Loan_Amount_Term']
```

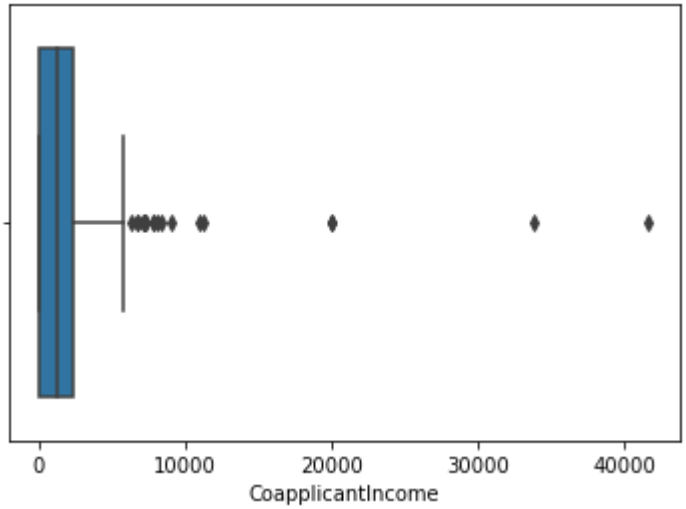
Outliers

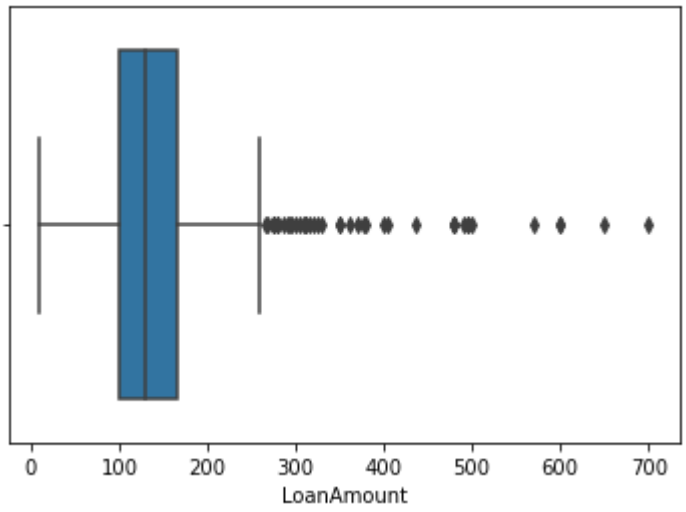
Outliers in any dataset has a very significant effect on our dataset because they are minority of the values in any variable which are different from other data point in a dataset, they are the observations which are very different from the overall pattern of the data, outlier can be really large or really small than the majority of the values in column and can have a significant impact on the results, below are the figures to visualize the outliers in our dataset.

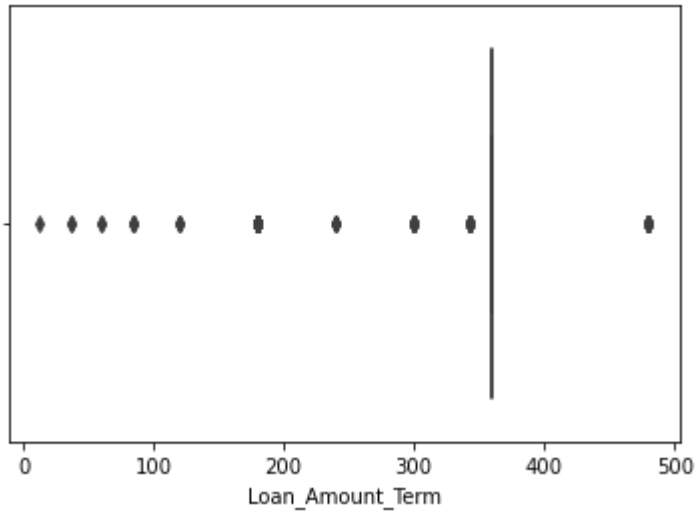
In [262…
```
for i in my_list:
    sns.boxplot(x=df[i])
```

```
# Show the plot
    plt.show()
```









Overcoming the Outliers

To overcome the outliers for our dataset we have used scaling or normalization technique which means that all of the datapoints in our dataset will be changed to values between 1 and 0, this can help us in overcoming outliers because if we have a very large datapoint or a very small datapoint it would not effect our dataset much now because now every value in our dataset is converted into a value between 0 and 1. Some algorithms may prioritise variables with greater values when variables in a dataset have varied scales or ranges, this might lead to issues if there are any outliers. We can somewhat prevent this issue by scaling the variables so that they have the same range.

In [263…
```
#above figures show that these varibles contain outlier, so scalling values in these va
scale = StandardScaler()
x[my_list]=scale.fit_transform(x[my_list])
```

Splitting the dataset

It is important to split the data into training and testing so that we can measure how well a machine learning algorithm performs on brand new, untested data. A machine learning model aims to accurately predict new information. It is important to evaluate the performance of the model with unprecedented data. When the model is trained and tested with the same data, it may be difficult to determine whether the model is actually learning to generalize from the data or is only memorizing it. We can train the model on the training set and evaluate its performance on the test set by partitioning the data into training and test sets. Overall, splitting the data into training and testing sets is a best practice in machine learning, and it allows us to evaluate the performance of a model on new, unseen data and avoid overfitting, there below we have splitted our data into 80% train and 20% test data using train_test_split which is one of the module of the sklearn package.

In [264…
```
# Split the data into training and test sets
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.2, random_state=4
```

In [265…
```
print("Shape of X_train: ", x_train.shape)
print("Shape of y_train: ", y_train.shape)
```

```
print("Shape of X_test: ", x_test.shape)
print("Shape of y_test: ", y_test.shape)
```

```
Shape of X_train:  (491, 11)
Shape of y_train:  (491,)
Shape of X_test:  (123, 11)
Shape of y_test:  (123,)
```

Methods

According to the results below we have used LogisticRegression, KNN and RandomForest algorithms we have used the following algorithms because,

• LogisticRegression: For binary classification situations, where the objective is to predict one of two possible outcomes, the common approach is logistic regression. A binary classification problem, the goal of which is to predict a binary outcome (approved or not approved), the loan approval problem is to determine whether a loan application will be approved or rejected based on various input features, such as the applicant's income, credit score, and employment status. This kind of problem is ideally suited for the popular approach for binary classification problems, logistic regression.

• KNN: KNN is an popular machine learning algorithms for classification and regression issues include K-Nearest Neighbours (KNN). It is an algorithm with no parameters that bases its predictions on the training dataset's k-nearest data points. KNN can be used to predict whether a loan application will be granted or rejected in the case of a loan approval dataset based on multiple input parameters such the applicant's income, credit score, and job status.KNN is a non-parametric algorithm capable of handling relationships between the input and output variables that are not linear. This is beneficial in the loan approval problem since there may not be a linear relationship between the input variables and the output variable.

• RandomForest: Popular machine learning algorithm Random Forest is used to solve classification and regression issues. Multiple decision trees are combined in this ensemble method to produce predictions.

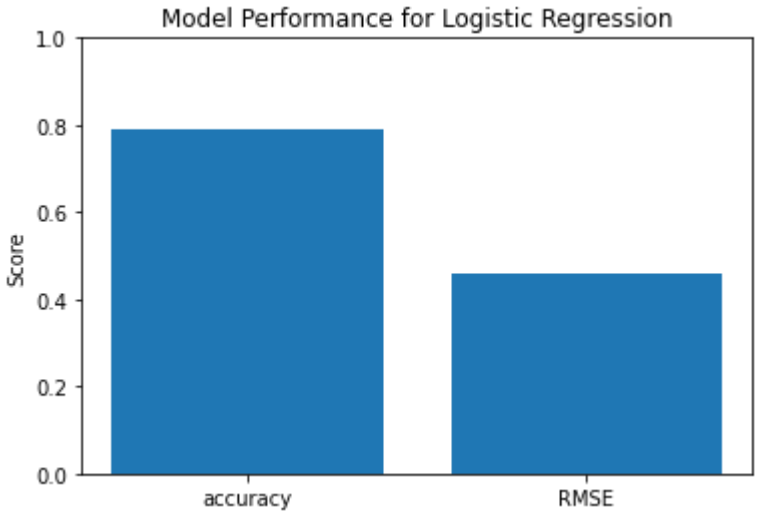
In [275…
```
# Fit the logistic regression model on the training data
lr = LogisticRegression()
lr.fit(x_train, y_train)
# Make predictions on the test data
y_pred = lr.predict(x_test)
# Calculate accuracy and root mean square error
accuracy = accuracy_score(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```

In [276…
```
# Print the results
print("Accuracy for Logistic Regression: ", accuracy)
print("Root mean square error Logistic Regression: ", rmse)
```

```
Accuracy for Logistic Regression:  0.7886178861788617
Root mean square error Logistic Regression:  0.4597631061983315
```

In [277…
```
# Plot accuracy and root mean square error
fig, ax = plt.subplots()
ax.bar(['accuracy', 'RMSE'], [accuracy, rmse])
```

```
ax.set_ylim(0, 1)
ax.set_title('Model Performance for Logistic Regression')
ax.set_ylabel('Score')
plt.show()
```



In [278…

```
# Fit the KNN model on the training data
knn = KNeighborsClassifier(n_neighbors=5)
knn.fit(x_train, y_train)
# Make predictions on the test data
y_pred = knn.predict(x_test)
# Calculate accuracy and root mean square error
accuracy = accuracy_score(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```

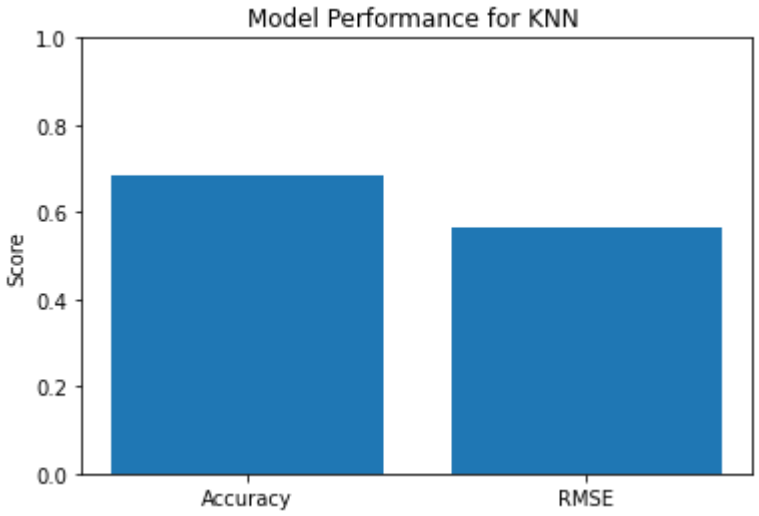
In [279…

```
# Print the results
print("Accuracy for KNN: ", accuracy)
print("Root mean square error for KNN: ", rmse)
```

```
Accuracy for KNN:  0.6829268292682927
Root mean square error for KNN:  0.563092506371473
```

In [280…

```
# Plot accuracy and root mean square error
fig, ax = plt.subplots()
ax.bar(['Accuracy', 'RMSE'], [accuracy, rmse])
ax.set_ylim(0, 1)
ax.set_title('Model Performance for KNN ')
ax.set_ylabel('Score')
plt.show()
```

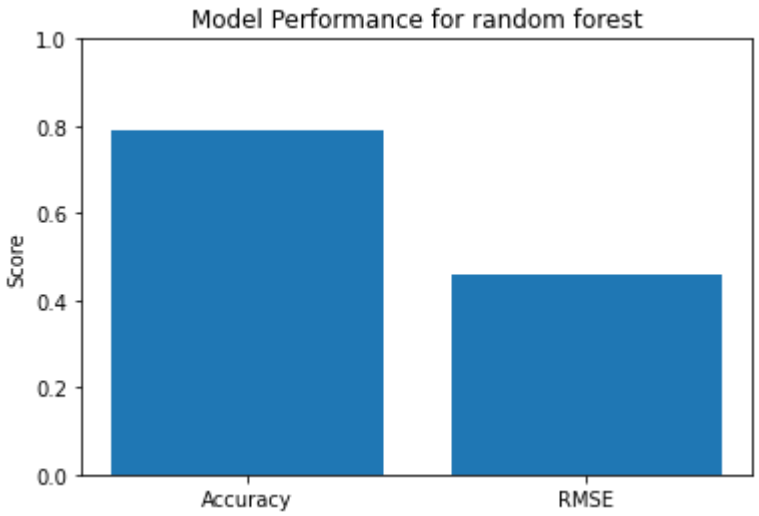


In [281…
```
# Fit the Random Forest model on the training data
rfc = RandomForestClassifier(n_estimators=100, max_depth=5, random_state=42)
rfc.fit(x_train, y_train)
# Make predictions on the test data
y_pred = rfc.predict(x_test)
# Calculate accuracy and root mean square error
accuracy = accuracy_score(y_test, y_pred)
rmse = np.sqrt(mean_squared_error(y_test, y_pred))
```

In [282…
```
# Print the results
print("Accuracy for RandomForest: ", accuracy)
print("Root mean square error RandomForest: ", rmse)
```

```
Accuracy for RandomForest:  0.7886178861788617
Root mean square error RandomForest:  0.4597631061983315
```

In [283…
```
# Plot accuracy and root mean square error
fig, ax = plt.subplots()
ax.bar(['Accuracy', 'RMSE'], [accuracy, rmse])
ax.set_ylim(0, 1)
ax.set_title('Model Performance for random forest')
ax.set_ylabel('Score')
plt.show()
```



Results

From the outcomes presented above we can be sure that for this kind of problem in Machine learning, LogisticRegression or RandomForest are the best algorithms because we can see that they have got the best accuracy score which is almost 80% and the root mean square error for both of these algorithms is 0.45 whereas K Nearest Neighbors algorithm in our problem has given us the worse results which is almost 70% for accuracy and 0.56 score for root mean square error.

Conclusion

The main goal of our project was to determine or predict the loan approval status, to accomplish the goal of this project we first analyzed the data and tried various algorithms for imputations of missing values and prediction of the loan status and for the prediction we found that RandomForest and LogisticRegression worked the best with the accuracy of almost 80%. This kind of work if done manually can be very time taking and requires a lot capital to employee humans for this task and humans can be easily prone to discrimination which means that they can produce results based on biasness so if these kinds of task are done through machine learning could give us fair results.

In [ ]: