```
Missing Completely at Random (MCAR):

    Missing values occur randomly and independently of other variables or the observed values.
    The probability of a value being missing is the same for all observations.
    No systematic differences between missing and non-missing values.
    Example: A survey question is accidentally skipped by respondents due to a random error.

Missing at Random (MAR):

    The probability of a value being missing depends on other observed variables, but not on the missing value itself.
    Systematic differences between missing and non-missing values are explained by observed data.
    Example: Income data is missing for unemployed individuals in a survey, but this missingness can be explained by the employment status variable.

Missing Not at Random (MNAR):

    The probability of a value being missing depends on the missing value itself, even after accounting for observed data.
    Systematic differences between missing and non-missing values cannot be explained by observed data.
    Example: Patients with severe health conditions are less likely to report their symptoms accurately in a health survey.
```

Handle Missing Data

```
Deletion
```

In [1]:
```python
import pandas as pd
```

In [2]:
```
df=pd.read_csv("D:/Downloads/archive (22)/loan_data_set.csv")
df
```

Out[2]:

	Loan_ID	Gender	Married	Dependents	Education	Self_Employed	ApplicantIncome	Coappli
0	LP001002	Male	No	0	Graduate	No	5849	
1	LP001003	Male	Yes	1	Graduate	No	4583	
2	LP001005	Male	Yes	0	Graduate	Yes	3000	
3	LP001006	Male	Yes	0	Not Graduate	No	2583	
4	LP001008	Male	No	0	Graduate	No	6000	
...	...	...	...	...	...	...	...	
609	LP002978	Female	No	0	Graduate	No	2900	
610	LP002979	Male	Yes	3+	Graduate	No	4106	
611	LP002983	Male	Yes	1	Graduate	No	8072	
612	LP002984	Male	Yes	2	Graduate	No	7583	
613	LP002990	Female	No	0	Graduate	Yes	4583	

614 rows × 13 columns

In [3]:
```
df.isnull().sum()
```

Out[3]:
```
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
```

In [4]:
```
df.isnull().sum()
```

Out[4]:
```
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
```

In [5]:
```python
#total number of missing value in dataset
df.isnull().sum().sum()
```

Out[5]:
```
149
```

Deleting the entire row

In [6]:
```python
df.dropna(axis=0).isnull().sum()
```

Out[6]:
```
Loan_ID              0
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
```

Deleting the entire column

In [7]:
```python
df.drop(['Dependents'],axis=1)
df.isnull().sum()
```

Out[7]:
```
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
```

Imputing the Missing Value

In [8]:
```python
#using fillna() and mean
df["LoanAmount"]=df["LoanAmount"].fillna(df["LoanAmount"].mean())
df.isnull().sum()
```

Out[8]:
```
Loan_ID               0
Gender               13
Married               3
Dependents           15
Education             0
Self_Employed        32
ApplicantIncome       0
CoapplicantIncome     0
LoanAmount            0
Loan_Amount_Term     14
Credit_History       50
Property_Area         0
Loan_Status           0
dtype: int64
```

In [9]:
```python
# replace missing value with median
df['Loan_Amount_Term'].fillna(df['Loan_Amount_Term'].median()).isna().sum()
```

Out[9]:
```
0
```

Replacing with the previous value - forward fill

In [10]:
```python
df['Married'].ffill().isna().sum()
```

Out[10]:
```
0
```

Replacing with the next value - backward fill

In [11]:
```python
df['Self_Employed'].bfill().isna().sum()
```

Out[11]:
```
0
```

Impute the Most Frequent Value

In [12]:
```python
from sklearn.impute import SimpleImputer
imputer = SimpleImputer(strategy='most_frequent')
imputer.fit_transform(df)
```

Out[12]:
```
array([['LP001002', 'Male', 'No', ..., 1.0, 'Urban', 'Y'],
       ['LP001003', 'Male', 'Yes', ..., 1.0, 'Rural', 'N'],
       ['LP001005', 'Male', 'Yes', ..., 1.0, 'Urban', 'Y'],
       ...,
       ['LP002983', 'Male', 'Yes', ..., 1.0, 'Urban', 'Y'],
       ['LP002984', 'Male', 'Yes', ..., 1.0, 'Urban', 'Y'],
       ['LP002990', 'Female', 'No', ..., 0.0, 'Semiurban', 'N']],
      dtype=object)
```

In [13]:
```python
imputer = SimpleImputer(strategy='constant', fill_value='missing')
imputer.fit_transform(df)
print(df)
```

```
      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   
..        ...     ...     ...        ...           ...           ...   
609  LP002978  Female      No          0      Graduate            No   
610  LP002979    Male     Yes         3+      Graduate            No   
611  LP002983    Male     Yes          1      Graduate            No   
612  LP002984    Male     Yes          2      Graduate            No   
613  LP002990  Female      No          0      Graduate           Yes   

     ApplicantIncome  CoapplicantIncome  LoanAmount  Loan_Amount_Term  \
0               5849                0.0  146.412162             360.0   
1               4583             1508.0  128.000000             360.0   
2               3000                0.0   66.000000             360.0   
3               2583             2358.0  120.000000             360.0   
4               6000                0.0  141.000000             360.0   
```

Interpolation

Missing values can also be imputed using interpolation. Pandas’ interpolate method can be used to replace the missing values with different interpolation methods like ‘polynomial,’ ‘linear,’ and ‘quadratic.’ The default method is ‘linear.’

In [14]:
```python
df.interpolate()
df.isna().sum()
```

```
C:\Users\heman\AppData\Local\Temp\ipykernel_16864\4235256301.py:1: FutureWarnin
g: DataFrame.interpolate with object dtype is deprecated and will raise in a fu
ture version. Call obj.infer_objects(copy=False) before interpolating instead.
  df.interpolate()
```

Out[14]:
```
Loan_ID               0
Gender               13
Married               3
Dependents           15
Education             0
Self_Employed        32
ApplicantIncome       0
CoapplicantIncome     0
LoanAmount            0
Loan_Amount_Term     14
Credit_History       50
Property_Area         0
Loan_Status           0
dtype: int64
```

Multivariate Approach

IterativeImputer in scikit-learn is a powerful imputation technique that estimates missing values in a dataset by modeling each feature with missing values as a function of other features. It iteratively imputes missing values for each feature using predictions from a set of estimators, typically regression models.

In [15]:
```
import pandas as pd
d = pd.read_csv('http://bit.ly/kaggletrain', nrows=6)
cols = ['SibSp', 'Fare', 'Age']
X = d[cols]
A=X
X
```

Out[15]:

	SibSp	Fare	Age
0	1	7.2500	22.0
1	1	71.2833	38.0
2	0	7.9250	26.0
3	1	53.1000	35.0
4	0	8.0500	35.0
5	0	8.4583	NaN

In [16]:
```
from sklearn.experimental import enable_iterative_imputer
from sklearn.impute import IterativeImputer
impute_it = IterativeImputer()
impute_it.fit_transform(X)
```

Out[16]:
```
array([[ 1.        ,  7.25      , 22.        ],
       [ 1.        , 71.2833    , 38.        ],
       [ 0.        ,  7.925     , 26.        ],
       [ 1.        , 53.1       , 35.        ],
       [ 0.        ,  8.05      , 35.        ],
       [ 0.        ,  8.4583    , 28.50639495]])
```

Nearest Neighbors Imputations (KNNImputer)

Missing values are imputed using the k-Nearest Neighbors approach, where a Euclidean distance is used to find the nearest neighbors.

In [17]:
```
from sklearn.impute import KNNImputer
impute_knn = KNNImputer(n_neighbors=2)
impute_knn.fit_transform(A)
```

Out[17]:
```
array([[ 1.    ,  7.25  , 22.    ],
       [ 1.    , 71.2833, 38.    ],
       [ 0.    ,  7.925 , 26.    ],
       [ 1.    , 53.1   , 35.    ],
       [ 0.    ,  8.05  , 35.    ],
       [ 0.    ,  8.4583, 30.5   ]])
```

Handling missing data

Delete rows with missing values (careful, you might lose data!).

Fill in the blanks with averages, midpoints, or common values (but be aware of bias).

Use fancy tricks like KNN to find similar data points to estimate missing values.

Feature Scaling

Feature scaling is a data preprocessing technique used to transform the values of features or variables in a dataset to a similar scale. The purpose is to ensure that all features contribute equally to the model and to avoid the domination of features with larger values
common techniques for feature scaling, including standardization, normalization, and min-max scaling.

Machine learning algorithms like linear regression, logistic regression, neural network, PCA (principal component analysis), etc., that use gradient descent as an optimization technique require data to be scaled.

Distance algorithms like KNN, K-means clustering, and SVM(support vector machines) are most affected by the range of features. This is because, behind the scenes, they are using distances between data points to determine their similarity.

Tree-based algorithms, on the other hand, are fairly insensitive to the scale of the features.

Normalization

Normalization(MinMax) is a scaling technique in which values are shifted and rescaled so that they end up ranging between 0 and 1. It is also known as Min-Max scaling.

Rescales values to a range between 0 and 1
Sensitive to outliers
Retains the shape of the original distribution
May not preserve the relationships between the data points
Scales values between [0, 1] or [-1, 1].

```
(x - min)/(max - min)
```

Standardization

```
Centers data around the mean and scales to a standard deviation of 1
Less sensitive to outliers
Changes the shape of the original distribution
Preserves the relationships between the data points
It is not bounded to a certain range.
```

```
(x - mean)/standard deviation
```

In [18]:
```
import seaborn as sns
import matplotlib.pyplot as plt
```

In [19]:
```
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         614 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
```

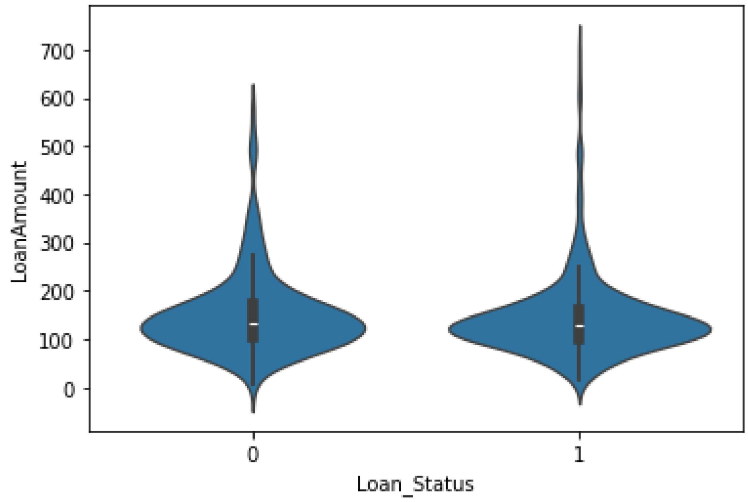
In [23]:
```python
df['Loan_Status'] = pd.Categorical(df['Loan_Status'])
df['Loan_Status'] = df['Loan_Status'].map({'Y':1, 'N':0})
df.head()
```

Out[23]:

e	CoapplicantIncome	LoanAmount	Loan_Amount_Term	Credit_History	Property_Area	Loan_Status
9	0.0	146.412162	360.0	1.0	Urban	1
3	1508.0	128.000000	360.0	1.0	Rural	0
0	0.0	66.000000	360.0	1.0	Urban	1
3	2358.0	120.000000	360.0	1.0	Urban	1
0	0.0	141.000000	360.0	1.0	Urban	1

In [33]:
```python
sns.violinplot(x="Loan_Status",y="LoanAmount",data=df)
```

Out[33]: <Axes: xlabel='Loan_Status', ylabel='LoanAmount'>



In [34]:
```python
import numpy as np
from sklearn.preprocessing import MinMaxScaler, StandardScaler
```

In [35]:
```python
scaler_minmax = MinMaxScaler()
df['LoanAmount_Normalized'] = scaler_minmax.fit_transform(df[['LoanAmount']])

# Standardize LoanAmount to mean=0 and std=1
scaler_standard = StandardScaler()
df['LoanAmount_Standardized'] = scaler_standard.fit_transform(df[['LoanAmount']]
```

In [39]:

```
plt.figure(figsize=(14, 6))

# Plot normalized LoanAmount
plt.subplot(1, 3, 1)
sns.violinplot(x="Loan_Status", y="LoanAmount_Normalized", data=df)
plt.title('Normalized LoanAmount')

# Plot standardized LoanAmount
plt.subplot(1, 3, 2)
sns.violinplot(x="Loan_Status", y="LoanAmount_Standardized", data=df)
plt.title('Standardized LoanAmount')

plt.subplot(1, 3, 3)
sns.violinplot(x="Loan_Status",y="LoanAmount",data=df)
plt.title('Normal LoanAmount')

plt.show()
```