Missing Values

Missing values occurs in dataset when some of the informations is not stored for a variable There are 3 mechanisms

1 Missing Completely at Random, MCAR:

Missing completely at random (MCAR) is a type of missing data mechanism in which the probability of a value being missing is unrelated to both the observed data and the missing data. In other words, if the data is MCAR, the missing values are randomly distributed throughout the dataset, and there is no systematic reason for why they are missing.

- For example, in a survey about the prevalence of a certain disease, the missing data might be MCAR if the survey participants with missing values for certain questions were selected randomly and their missing responses are not related to their disease status or any other variables measured in the survey.

2. Missing at Random MAR:

Missing at Random (MAR) is a type of missing data mechanism in which the probability of a value being missing depends only on the observed data, but not on the missing data itself. In other words, if the data is MAR, the missing values are systematically related to the observed data, but not to the missing data. Here are a few examples of missing at random:

- Income data: Suppose you are collecting income data from a group of people, but some participants choose not to report their income. If the decision to report or not report income is related to the participant's age or gender, but not to their income level, then the data is missing at random.

- Medical data: Suppose you are collecting medical data on patients, including their blood pressure, but some patients do not report their blood pressure. If the patients who do not report their blood pressure are more likely to be younger or have healthier lifestyles, but the missingness is not related to their actual blood pressure values, then the data is missing at random.

3. Missing data not at random (MNAR)

It is a type of missing data mechanism where the probability of missing values depends on the value of the missing data itself. In other words, if the data is MNAR, the missingness is not random and is dependent on unobserved or unmeasured factors that are associated with the missing values.

- For example, suppose you are collecting data on the income and job satisfaction of employees in a company. If employees who are less satisfied with their jobs are more likely to refuse to report their income, then the data is not missing at random. In this case, the missingness is dependent on job satisfaction, which is not directly observed or measured.

Examples

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

In [2]:
```
df=sns.load_dataset('titanic')
```

In [3]:
```
df.head() # top 5 data
```

Out[3]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True

In [4]:
```
df.tail() # bottom 5 data
```

Out[4]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
886	0	2	male	27.0	0	0	13.00	S	Second	man	True	NaN	Southampton	no	True
887	1	1	female	19.0	0	0	30.00	S	First	woman	False	B	Southampton	yes	True
888	0	3	female	NaN	1	2	23.45	S	Third	woman	False	NaN	Southampton	no	False
889	1	1	male	26.0	0	0	30.00	C	First	man	True	C	Cherbourg	yes	True
890	0	3	male	32.0	0	0	7.75	Q	Third	man	True	NaN	Queenstown	no	True

In [6]:

```
df.shape
```

Out[6]: (891, 15)

In [7]:

```
df.describe()
```

Out[7]:

	survived	pclass	age	sibsp	parch	fare
count	891.000000	891.000000	714.000000	891.000000	891.000000	891.000000
mean	0.383838	2.308642	29.699118	0.523008	0.381594	32.204208
std	0.486592	0.836071	14.526497	1.102743	0.806057	49.693429
min	0.000000	1.000000	0.420000	0.000000	0.000000	0.000000
25%	0.000000	2.000000	20.125000	0.000000	0.000000	7.910400
50%	0.000000	3.000000	28.000000	0.000000	0.000000	14.454200
75%	1.000000	3.000000	38.000000	1.000000	0.000000	31.000000
max	1.000000	3.000000	80.000000	8.000000	6.000000	512.329200

In [8]:

```
## Check missing values

df.isnull()
```

Out[8]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone
0	False	False	False	False	False	False	False	False	False	False	False	True	False	False	False
1	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False	False	False	True	False	False	False
3	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False	False	False	True	False	False	False
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
886	False	False	False	False	False	False	False	False	False	False	False	True	False	False	False
887	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
888	False	False	False	True	False	False	False	False	False	False	False	True	False	False	False
889	False	False	False	False	False	False	False	False	False	False	False	False	False	False	False
890	False	False	False	False	False	False	False	False	False	False	False	True	False	False	False

891 rows × 15 columns

In [5]:

```
## Check total missing values

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

Out[5]:

	0
survived	0
pclass	0
sex	0
age	177
sibsp	0
parch	0
fare	0
embarked	2
class	0
who	0
adult_male	0
deck	688
embark_town	2
alive	0
alone	0

dtype: int64

In [9]:

```
## Delete the rows or data point to handle missing values

df.shape
```

Out[9]: (891, 15)

In [10]:

```
df.dropna().shape
```

Out[10]: (182, 15)

In [11]:
```
## Column wise deletion
df.dropna(axis=1)
```

Out[11]:

	survived	pclass	sex	sibsp	parch	fare	class	who	adult_male	alive	alone
0	0	3	male	1	0	7.2500	Third	man	True	no	False
1	1	1	female	1	0	71.2833	First	woman	False	yes	False
2	1	3	female	0	0	7.9250	Third	woman	False	yes	True
3	1	1	female	1	0	53.1000	First	woman	False	yes	False
4	0	3	male	0	0	8.0500	Third	man	True	no	True
...	...	...	...	...	...	...	...	...	...	...	...
886	0	2	male	0	0	13.0000	Second	man	True	no	True
887	1	1	female	0	0	30.0000	First	woman	False	yes	True
888	0	3	female	1	2	23.4500	Third	woman	False	no	False
889	1	1	male	0	0	30.0000	First	man	True	yes	True
890	0	3	male	0	0	7.7500	Third	man	True	no	True

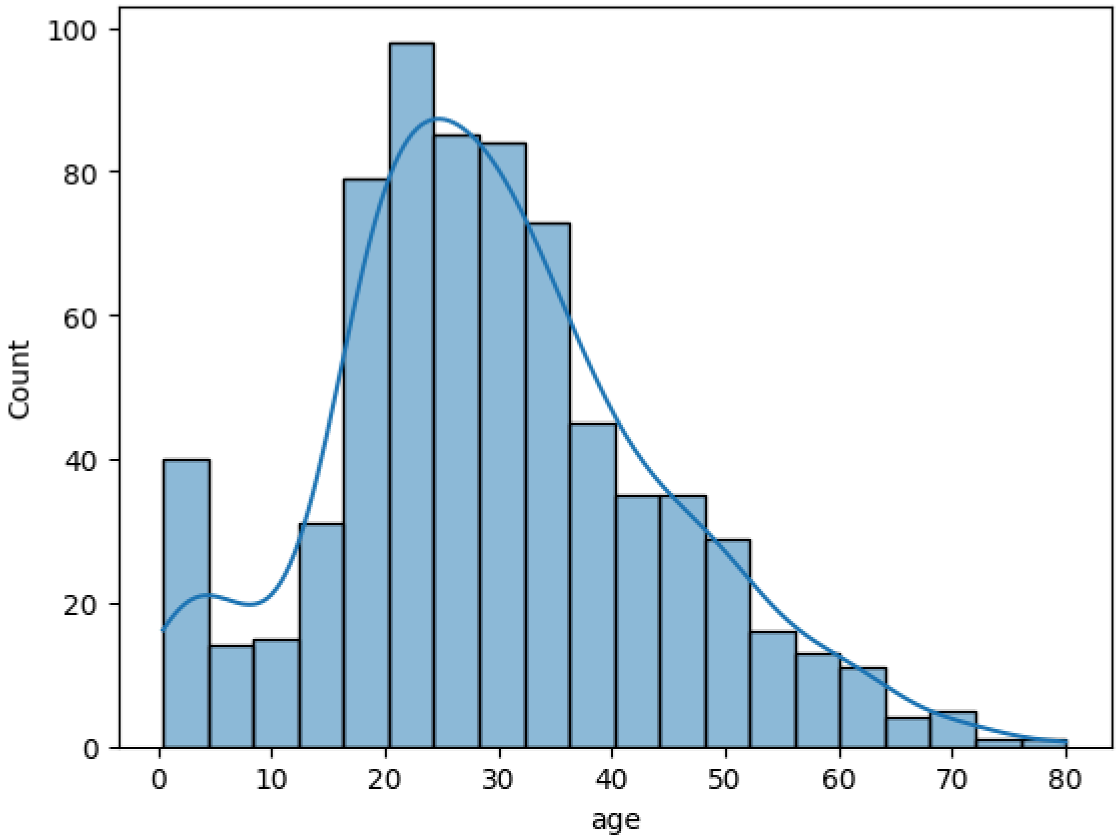
891 rows × 11 columns

Imputation Missing Values

Mean Value Imputation

In [12]:
```
sns.histplot(df['age'],kde=True)
```

Out[12]:
```
<Axes: xlabel='age', ylabel='Count'>
```



In [13]:
```
# column wise check the null value

df['age'].isnull().sum()
```

Out[13]: 177

Mean Imputation works when we have normally distributed data.

In [14]:
```
# Impute the mean value to the age column and create new column

df['Age_mean']=df['age'].fillna(df['age'].mean())
```

In [15]:
```
df[['Age_mean','age']]
```

Out[15]:

	Age_mean	age
0	22.000000	22.0
1	38.000000	38.0
2	26.000000	26.0
3	35.000000	35.0
4	35.000000	35.0
...	...	...
886	27.000000	27.0
887	19.000000	19.0
888	29.699118	NaN
889	26.000000	26.0
890	32.000000	32.0

891 rows × 2 columns

In [16]:

```
df.head()
```

Out[16]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone	Age_mean
0	0	3	male	22.0	1	0	7.2500	S	Third	man	True	NaN	Southampton	no	False	22.0
1	1	1	female	38.0	1	0	71.2833	C	First	woman	False	C	Cherbourg	yes	False	38.0
2	1	3	female	26.0	0	0	7.9250	S	Third	woman	False	NaN	Southampton	yes	True	26.0
3	1	1	female	35.0	1	0	53.1000	S	First	woman	False	C	Southampton	yes	False	35.0
4	0	3	male	35.0	0	0	8.0500	S	Third	man	True	NaN	Southampton	no	True	35.0

2. Median Value Imputation- If we have outliers in the dataset

In [17]:

```
df['age_median']=df['age'].fillna(df['age'].median())
```

In [18]:
```
df[['age_median','Age_mean','age']]
```

Out[18]:

	age_median	Age_mean	age
0	22.0	22.000000	22.0
1	38.0	38.000000	38.0
2	26.0	26.000000	26.0
3	35.0	35.000000	35.0
4	35.0	35.000000	35.0
...	...	...	...
886	27.0	27.000000	27.0
887	19.0	19.000000	19.0
888	28.0	29.699118	NaN
889	26.0	26.000000	26.0
890	32.0	32.000000	32.0

891 rows × 3 columns

3. Mode Imputation Technqiue--Categorical values

In [19]:
```
df['embarked'].isnull()
```

Out[19]:

	embarked
0	False
1	False
2	False
3	False
4	False
...	...
886	False
887	False
888	False
889	False
890	False

891 rows × 1 columns

dtype: bool

In [20]:
```
df[df['embarked'].isnull()]
```

Out[20]:

	survived	pclass	sex	age	sibsp	parch	fare	embarked	class	who	adult_male	deck	embark_town	alive	alone	Age_mean	age_median
61	1	1	female	38.0	0	0	80.0	NaN	First	woman	False	B	NaN	yes	True	38.0	38.0
829	1	1	female	62.0	0	0	80.0	NaN	First	woman	False	B	NaN	yes	True	62.0	62.0

In [21]:
```
df['embarked'].unique()
```

Out[21]:
```
array(['S', 'C', 'Q', nan], dtype=object)
```

In [22]:
```
mode_value=df[df['embarked'].notna()]['embarked'].mode()[0]
```

In [23]:
```
df['embarked_mode']=df['embarked'].fillna(mode_value)
```

In [24]:
```
df[['embarked_mode','embarked']]
```

Out[24]:

	embarked_mode	embarked
0	S	S
1	C	C
2	S	S
3	S	S
4	S	S
...	...	...
886	S	S
887	S	S
888	S	S
889	C	C
890	Q	Q

891 rows × 2 columns

In [25]:
```
df['embarked_mode'].isnull().sum()
```

Out[25]:
```
0
```

In [26]:
```
df['embarked'].isnull().sum()
```

Out[26]:
```
2
```

Handling Imbalanced Dataset

1. Up Sampling

2. Down Sampling

In [27]:
```
# Set the random seed for reproducibility

np.random.seed(123)

# Create a dataframe with two classes

n_samples = 1000
class_0_ratio = 0.9
n_class_0 = int(n_samples * class_0_ratio)
n_class_1 = n_samples - n_class_0
```

In [28]:
```
n_class_0
```

Out[28]:
```
900
```

In [29]:
```
n_class_1
```

Out[29]:
```
100
```

In [30]:
```
## Create DataFrame with Imbalanced Dataset

class_0 = pd.DataFrame({
    'feature_1': np.random.normal(loc=0, scale=1, size=n_class_0),
    'feature_2': np.random.normal(loc=0, scale=1, size=n_class_0),
    'target': [0] * n_class_0
})
```

In [31]:
```
class_1 = pd.DataFrame({
    'feature_1': np.random.normal(loc=2, scale=1, size=n_class_1),
    'feature_2': np.random.normal(loc=2, scale=1, size=n_class_1),
    'target': [1] * n_class_1
})
```

In [32]:
```
# Combine both data within one dataframe

df=pd.concat([class_0,class_1]).reset_index(drop=True)
```

In [33]:
```
df.head()
```

Out[33]:

	feature_1	feature_2	target
0	-1.085631	0.551302	0
1	0.997345	0.419589	0
2	0.282978	1.815652	0
3	-1.506295	-0.252750	0
4	-0.578600	-0.292004	0

In [34]:
```
df.tail()
```

Out[34]:

	feature_1	feature_2	target
995	1.376371	2.845701	1
996	2.239810	0.880077	1
997	1.131760	1.640703	1
998	2.902006	0.390305	1
999	2.697490	2.013570	1

In [35]:
```
# total target count

df["target"].value_counts()
```

Out[35]:

target	count
0	900
1	100

dtype: int64

In [36]:
```
## upsampling
```

```
df_minority=df[df['target']==1]

df_majority=df[df['target']==0]
```

In [ ]:

In [37]:

```
from sklearn.utils import resample
df_minority_upsampled=resample(df_minority,replace=True, #Sample With replacement
         n_samples=len(df_majority),
         random_state=42
        )
```

In [38]:

```
df_minority_upsampled.shape
```

Out[38]:

```
(900, 3)
```

In [39]:

```
df_minority_upsampled.head()
```

Out[39]:

	feature_1	feature_2	target
951	1.125854	1.843917	1
992	2.196570	1.397425	1
914	1.932170	2.998053	1
971	2.272825	3.034197	1
960	2.870056	1.550485	1

In [40]:

```
df_upsampled=pd.concat([df_majority,df_minority_upsampled])
```

In [41]:

```
df_upsampled['target'].value_counts()
```

Out[41]:

target	count
0	900
1	900

dtype: int64

Down Sampling

In [42]:
```
# Set the random seed for reproducibility
np.random.seed(123)

# Create a dataframe with two classes
n_samples = 1000
class_0_ratio = 0.9
n_class_0 = int(n_samples * class_0_ratio)
n_class_1 = n_samples - n_class_0

class_0 = pd.DataFrame({
    'feature_1': np.random.normal(loc=0, scale=1, size=n_class_0),
    'feature_2': np.random.normal(loc=0, scale=1, size=n_class_0),
    'target': [0] * n_class_0
})

class_1 = pd.DataFrame({
    'feature_1': np.random.normal(loc=2, scale=1, size=n_class_1),
    'feature_2': np.random.normal(loc=2, scale=1, size=n_class_1),
    'target': [1] * n_class_1
})
```

In [43]:
```
df = pd.concat([class_0, class_1]).reset_index(drop=True)
```

In [44]:
```
# Check the class distribution
print(df['target'].value_counts())
```

```
target
0    900
1    100
Name: count, dtype: int64
```

In [45]:
```
## downsampling
df_minority=df[df['target']==1]
df_majority=df[df['target']==0]
```

In [46]:
```
from sklearn.utils import resample
df_majority_upsampled=resample(df_minority,replace=True, #Sample With replacement
         n_samples=len(df_majority),
         random_state=42
        )
```

In [47]:
```
df_minority_upsampled.shape
```

Out[47]:
```
(900, 3)
```

In [48]:
```
df_upsampled=pd.concat([df_majority,df_minority_upsampled])
```

In [49]:
```
df_upsampled['target'].value_counts()
```

Out[49]:

target	count
0	900
1	900

dtype: int64

SMOTE (Synthetic Minority Oversampling Technique)

- SMOTE (Synthetic Minority Over-sampling Technique) is a technique used in machine learning to address imbalanced datasets where the minority class has significantly fewer instances than the majority class.
- SMOTE involves generating synthetic instances of the minority class by interpolating between existing instances.

In [50]:
```
from sklearn.datasets import make_classification
```

In [51]:
```
X,y=make_classification(n_samples=1000,n_redundant=0,n_features=2,n_clusters_per_class=1,
                   weights=[0.90],random_state=12)
```

In [52]:
```
df1=pd.DataFrame(X,columns=['f1','f2'])
df2=pd.DataFrame(y,columns=['target'])
```

In [53]:
```
final_df=pd.concat([df1,df2],axis=1)
```

In [54]:
```
final_df.head()
```

Out[54]:

	f1	f2	target
0	-0.762898	-0.706808	0
1	-1.075436	-1.051162	0
2	-0.610115	-0.909802	0
3	-2.023284	-0.428945	1
4	-0.812921	-1.316206	0

In [55]:
```
final_df['target'].value_counts()
```

Out[55]:

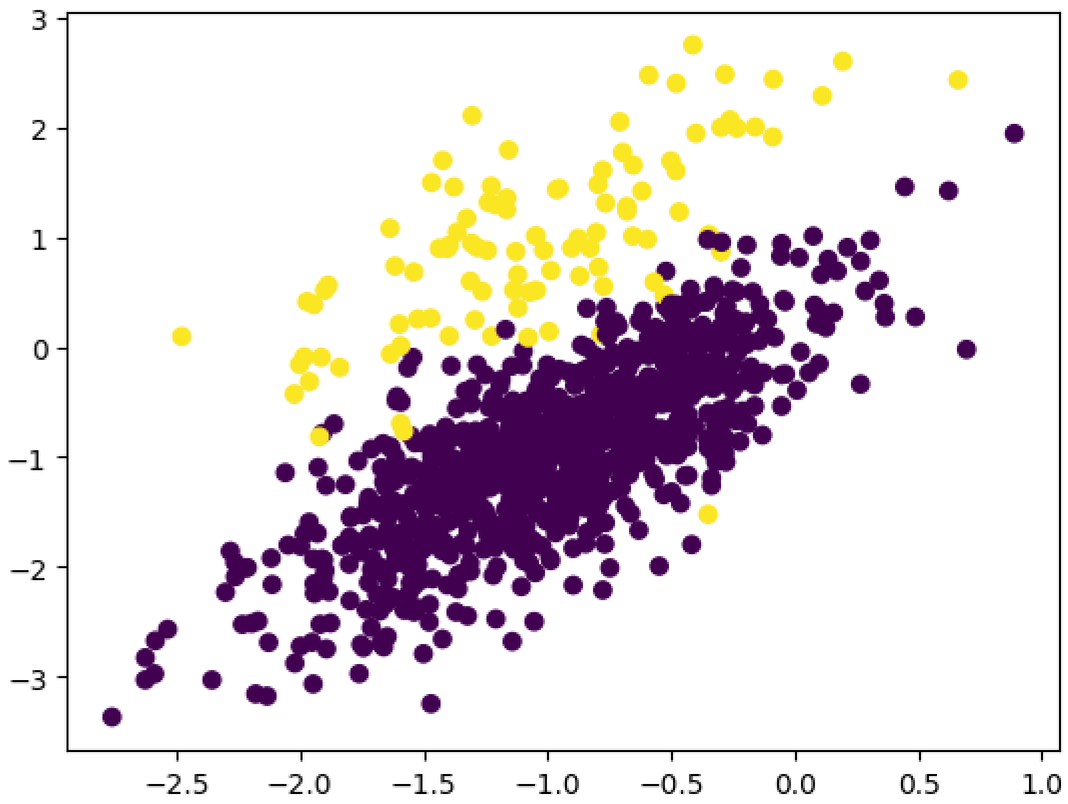
	count
target	
0	900
1	100

dtype: int64

In [56]:
```
import matplotlib.pyplot as plt

plt.scatter(final_df['f1'],final_df['f2'],c=final_df['target'])
```

Out[56]: <matplotlib.collections.PathCollection at 0x7ee6d13afc10>



In [57]:
```
!pip install imblearn
```

```
Collecting imblearn
  Downloading imblearn-0.0-py2.py3-none-any.whl.metadata (355 bytes)
Requirement already satisfied: imbalanced-learn in /usr/local/lib/python3.10/dist-packages (from imblearn) (0.12.4)
Requirement already satisfied: numpy>=1.17.3 in /usr/local/lib/python3.10/dist-packages (from imbalanced-learn->imblearn) (1.26.
4)
Requirement already satisfied: scipy>=1.5.0 in /usr/local/lib/python3.10/dist-packages (from imbalanced-learn->imblearn) (1.13.
1)
Requirement already satisfied: scikit-learn>=1.0.2 in /usr/local/lib/python3.10/dist-packages (from imbalanced-learn->imblearn)
(1.5.2)
Requirement already satisfied: joblib>=1.1.1 in /usr/local/lib/python3.10/dist-packages (from imbalanced-learn->imblearn) (1.4.
2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python3.10/dist-packages (from imbalanced-learn->imblearn)
(3.5.0)
Downloading imblearn-0.0-py2.py3-none-any.whl (1.9 kB)
Installing collected packages: imblearn
Successfully installed imblearn-0.0
```

In [58]:
```
from imblearn.over_sampling import SMOTE
```

In [59]:
```
## transform the dataset
oversample=SMOTE()
X,y=oversample.fit_resample(final_df[['f1','f2']],final_df['target'])
```

In [60]:
```
X.shape
```

Out[60]:
```
(1800, 2)
```

In [61]:
```
y.shape
```

Out[61]:
```
(1800,)
```

In [62]:
```
len(y[y==0])
```

Out[62]:
```
900
```

In [63]:
```
len(y[y==1])
```

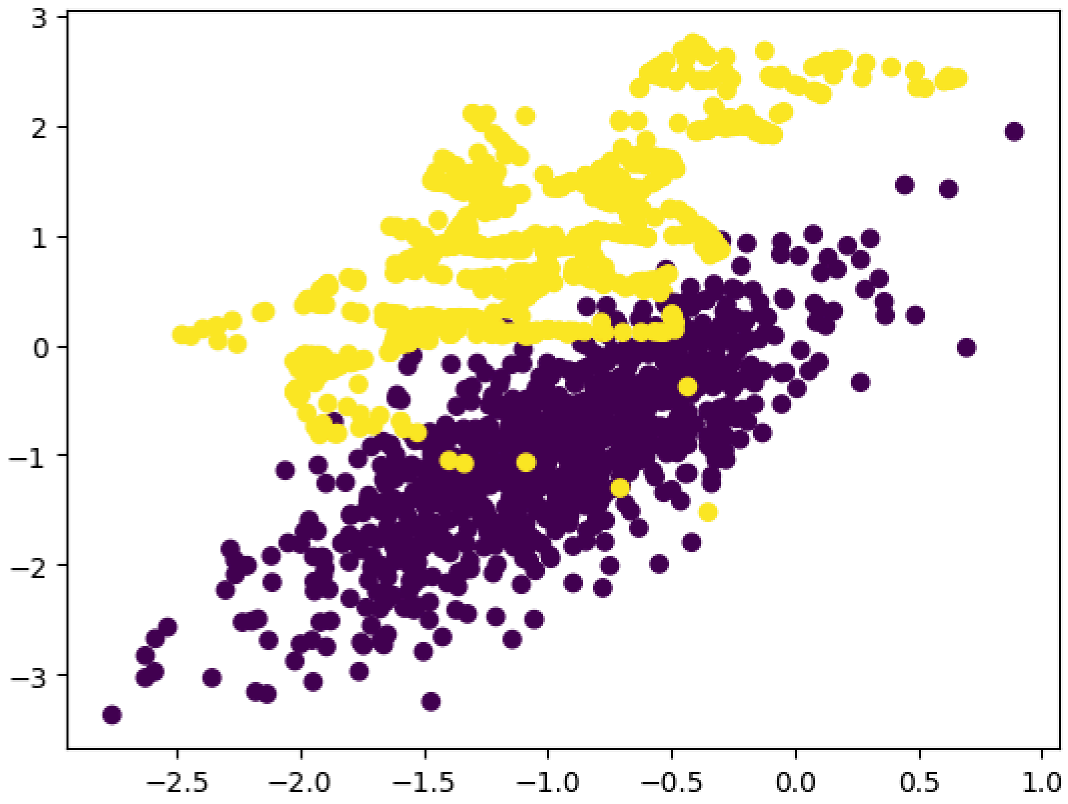
Out[63]:
```
900
```

In [64]:
```
df1=pd.DataFrame(X,columns=['f1','f2'])
df2=pd.DataFrame(y,columns=['target'])
```

In [65]:
```
oversample_df=pd.concat([df1,df2],axis=1)
```

In [66]:
```
plt.scatter(oversample_df['f1'],oversample_df['f2'],c=oversample_df['target'])
```

Out[66]:
```
<matplotlib.collections.PathCollection at 0x7ee6cf2fc3a0>
```



5 number Summary And Box Plot

In [67]:
```
## Minimum,MAximum,Median,Q1,Q3,IQR
```

In [68]:
```
marks=[45,32,56,75,89,54,32,89,90,87,67,54,45,98,99,67,74]
minimum,Q1,median,Q3,maximum=np.quantile(marks,[0,0.25,0.50,0.75,1.0])
```

In [69]:
```
minimum,Q1,median,Q3,maximum
```

Out[69]:
```
(32.0, 54.0, 67.0, 89.0, 99.0)
```

In [70]:
```
IQR=Q3-Q1
print(IQR)
```

```
35.0
```

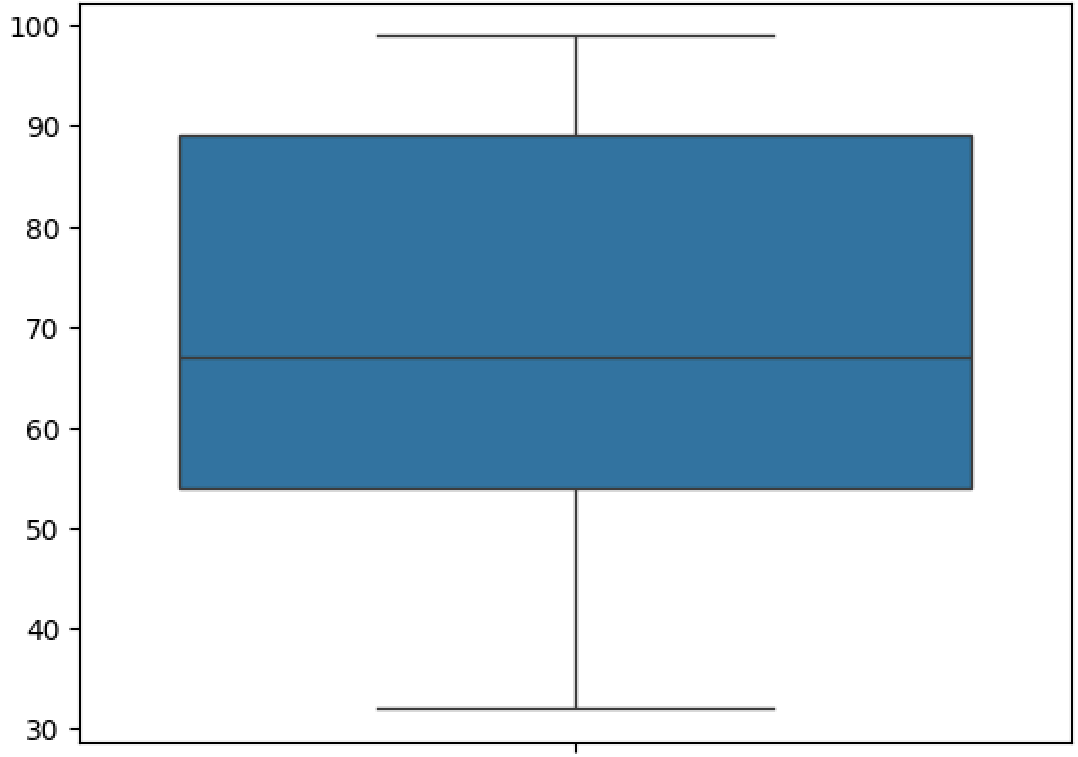
In [71]:
```
lower_fence=Q1-1.5*(IQR)
higher_fence=Q3+1.5*(IQR)
```

In [72]:
```
lower_fence, higher_fence
```

Out[72]:
```
(1.5, 141.5)
```

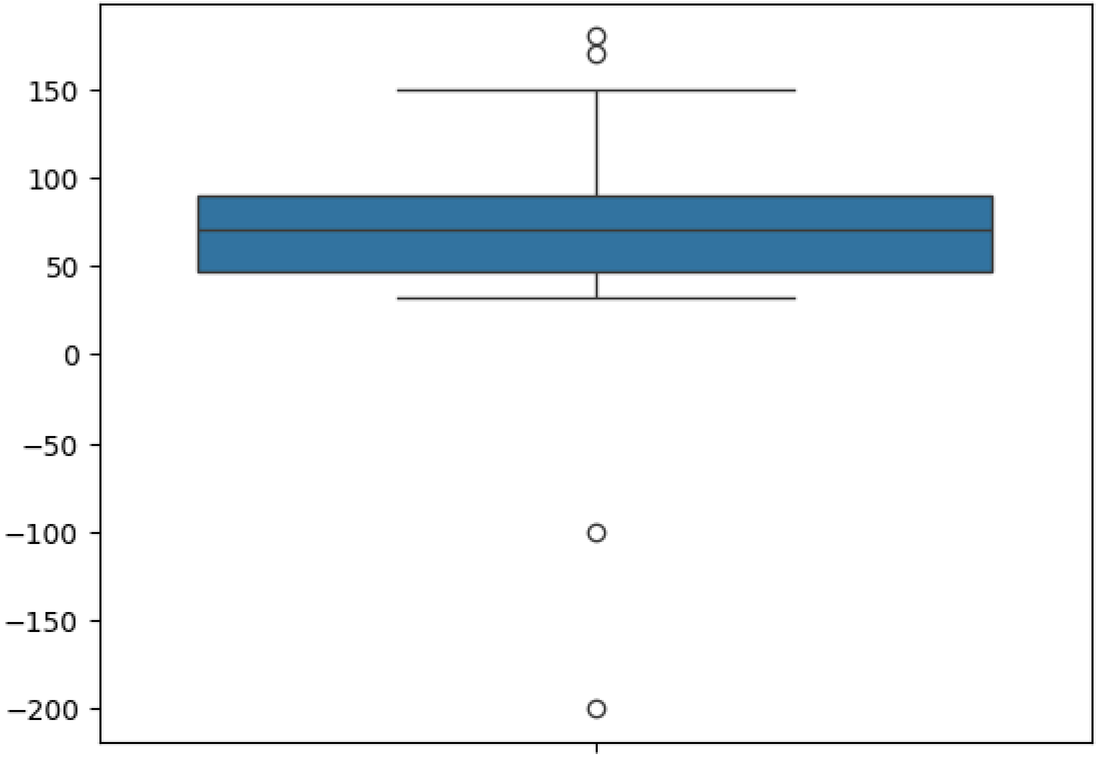
In [73]:
```
sns.boxplot(marks)
```

Out[73]:
```
<Axes: >
```



In [75]:
```
lst_marks=[-100,-200,45,32,56,75,89,54,32,89,90,87,67,54,45,98,99,67,74,150,170,180]
sns.boxplot(lst_marks)
```

Out[75]:
```
<Axes: >
```



Data Encoding

1. Nominal/OHE Encoding
2. Label and Ordinal Encoding
3. Target Guided Ordinal Encoding

Nominal/OHE Encoding

- One hot encoding, also known as nominal encoding, is a technique used to represent categorical data as numerical data, which is more suitable for machine learning algorithms.
- In this technique, each category is represented as a binary vector where each bit corresponds to a unique category.

- For example, if we have a categorical variable "color" with three possible values (red, green, blue), we can represent it using one hot encoding as follows:

1. Red: [1, 0, 0]
2. Green: [0, 1, 0]
3. Blue: [0, 0, 1]

In [76]:
```
from sklearn.preprocessing import OneHotEncoder
```

In [77]:
```
## Create a simple dataframe
df = pd.DataFrame({
    'color': ['red', 'blue', 'green', 'green', 'red', 'blue']
})
```

In [78]:
```
df.head()
```

Out[78]:

	color
0	red
1	blue
2	green
3	green
4	red

In [79]:
```
##create an instance of Onehotencoder

encoder=OneHotEncoder()
```

In [80]:
```
## perform fit and transform

encoded=encoder.fit_transform(df[['color']]).toarray()
```

In [81]:
```
encoder_df=pd.DataFrame(encoded,columns=encoder.get_feature_names_out())
```

In [82]:
```
encoder_df
```

Out[82]:

	color_blue	color_green	color_red
0	0.0	0.0	1.0
1	1.0	0.0	0.0
2	0.0	1.0	0.0
3	0.0	1.0	0.0
4	0.0	0.0	1.0
5	1.0	0.0	0.0

In [83]:

```
## for new data

encoder.transform([['blue']]).toarray()
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:493: UserWarning: X does not have valid feature names, but OneHotEncoder was fitted with feature names
  warnings.warn(
```

Out[83]:

```
array([[1., 0., 0.]])
```

In [84]:

```
pd.concat([df,encoder_df],axis=1)
```

Out[84]:

	color	color_blue	color_green	color_red
0	red	0.0	0.0	1.0
1	blue	1.0	0.0	0.0
2	green	0.0	1.0	0.0
3	green	0.0	1.0	0.0
4	red	0.0	0.0	1.0
5	blue	1.0	0.0	0.0

In [85]:

```
import seaborn as sns
sns.load_dataset('tips')
```

Out[85]:

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4
...	...	...	...	...	...	...	...
239	29.03	5.92	Male	No	Sat	Dinner	3
240	27.18	2.00	Female	Yes	Sat	Dinner	2
241	22.67	2.00	Male	Yes	Sat	Dinner	2
242	17.82	1.75	Male	No	Sat	Dinner	2
243	18.78	3.00	Female	No	Thur	Dinner	2

244 rows × 7 columns

Label Encoding

- Label encoding and ordinal encoding are two techniques used to encode categorical data as numerical data.
- Label encoding involves assigning a unique numerical label to each category in the variable. The labels are usually assigned in alphabetical order or based on the frequency of the categories.
- For example, if we have a categorical variable "color" with three possible values (red, green, blue), we can represent it using label encoding as follows:

1. Red: 1
2. Green: 2
3. Blue: 3

In [86]:
```
df.head()
```

Out[86]:

	color
0	red
1	blue
2	green
3	green
4	red

In [87]:

```
from sklearn.preprocessing import LabelEncoder
lbl_encoder=LabelEncoder()
```

In [88]:

```
lbl_encoder.fit_transform(df[['color']])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_label.py:114: DataConversionWarning: A column-vector y was passed
when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, warn=True)
```

Out[88]:

```
array([2, 0, 1, 1, 2, 0])
```

In [89]:

```
lbl_encoder.transform([['red']])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_label.py:132: DataConversionWarning: A column-vector y was passed
when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, dtype=self.classes_.dtype, warn=True)
```

Out[89]:

```
array([2])
```

In [90]:

```
lbl_encoder.transform([['blue']])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_label.py:132: DataConversionWarning: A column-vector y was passed
when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, dtype=self.classes_.dtype, warn=True)
```

Out[90]:

```
array([0])
```

In [91]:

```
lbl_encoder.transform([['green']])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/preprocessing/_label.py:132: DataConversionWarning: A column-vector y was passed when a 1d array was expected. Please change the shape of y to (n_samples, ), for example using ravel().
  y = column_or_1d(y, dtype=self.classes_.dtype, warn=True)
```

Out[91]:
```
array([1])
```

Ordinal Encoding

- It is used to encode categorical data that have an intrinsic order or ranking.
- In this technique, each category is assigned a numerical value based on its position in the order.
- For example, if we have a categorical variable "education level" with four possible values (high school, college, graduate, post-graduate), we can represent it using ordinal encoding as follows:

1. High school: 1
2. College: 2
3. Graduate: 3
4. Post-graduate: 4

In [92]:
```python
## Ordinal Encoding

from sklearn.preprocessing import OrdinalEncoder
```

In [93]:
```python
# create a sample dataframe with an ordinal variable


df = pd.DataFrame({
    'size': ['small', 'medium', 'large', 'medium', 'small', 'large']
})
```

In [94]:
```python
df
```

Out[94]:

	size
0	small
1	medium
2	large
3	medium
4	small
5	large

In [95]:
```
## create an instance of ORdinalEncoder and then fit_transform


encoder=OrdinalEncoder(categories=[['small','medium','large']])
```

In [96]:
```
encoder.fit_transform(df[['size']])
```

Out[96]:
```
array([[0.],
       [1.],
       [2.],
       [1.],
       [0.],
       [2.]])
```

In [97]:
```
encoder.transform([['small']])
```

```
/usr/local/lib/python3.10/dist-packages/sklearn/base.py:493: UserWarning: X does not have valid feature names, but OrdinalEncode
r was fitted with feature names
  warnings.warn(
```

Out[97]:
```
array([[0.]])
```

Target Guided Ordinal Encoding

- It is a technique used to encode categorical variables based on their relationship with the target variable.

- This encoding technique is useful when we have a categorical variable with a large number of unique categories, and we want to use this variable as a feature in our machine learning model.
- In Target Guided Ordinal Encoding, we replace each category in the categorical variable with a numerical value based on the mean or median of the target variable for that category.
- This creates a monotonic relationship between the categorical variable and the target variable, which can improve the predictive power of our model.

In [98]:
```
# create a sample dataframe with a categorical variable and a target variable
df = pd.DataFrame({
    'city': ['New York', 'London', 'Paris', 'Tokyo', 'New York', 'Paris'],
    'price': [200, 150, 300, 250, 180, 320]
})
```

In [99]:
```
df
```

Out[99]:

	city	price
0	New York	200
1	London	150
2	Paris	300
3	Tokyo	250
4	New York	180
5	Paris	320

In [100…
```
df.size
```

Out[100]: 12

In [101…
```
df.shape
```

Out[101]: (6, 2)

In [102…
```
df.ndim
```

Out[102]: 2

In [104…
```
df.describe()
```

Out[104]:

	price
count	6.000000
mean	233.333333
std	68.019605
min	150.000000
25%	185.000000
50%	225.000000
75%	287.500000
max	320.000000

In [105…
```
mean_price=df.groupby('city')['price'].mean().to_dict()
```

In [106…
```
mean_price
```

Out[106]:
```
{'London': 150.0, 'New York': 190.0, 'Paris': 310.0, 'Tokyo': 250.0}
```

In [107…
```
df['city_encoded']=df['city'].map(mean_price)
```

In [108…
```
df[['price','city_encoded']]
```

Out[108]:

	price	city_encoded
0	200	190.0
1	150	150.0
2	300	310.0
3	250	250.0
4	180	190.0
5	320	310.0

In [ ]:

```
!jupyter nbconvert --to html /content/Complete_Python_for_Data_Analysis.ipynb
```