DWDM Report

All Roll Numbers of your Team:	17BCD7071, 17BCD7014
Student names in your team :	Snigda Gedela, Tejo Vinay Potti
Contribution by each team member in percentage to complete this project:	Snigda: 50% Tejo Vinay: 50%
Slot (A2/D2):	A2
Number of Techniques implemented in Bucket 1:	5
1. Names of the techniques implemented:	• Imputation Using Forward Fil • Imputation Using Backward Fill • Imputation Using Mean • Imputation Using Median • Imputation Using Mode
2. Data set used and its link:	Heart Disease Dataset https://www.kaggle.com/ronitf/heart-disease-uci
Number of Techniques implemented in Bucket 2:	8
1. Names of the techniques implemented:	• Imputation Using Mean Values (using SimpleImputer module) • Imputation Using Median Values (using SimpleImputer module) • Imputation Using Most Frequent Values • Imputation Using Constant Value Outside the Fixed Value Range • Drop the column with missing values • Imputation Using k-NN (using fast_knn) • Imputation Using Multivariate Imputation by Chained Equation(MICE) • Imputation Using Deep Learning(datawig)
2. Data set used and its link:	Heart Disease Dataset https://www.kaggle.com/ronitf/heart-disease-uci
Number of Techniques implemented in Bucket 3:	2
1. Names of the techniques implemented:	• Imputation Using Linear Regression • Imputation Using k-nearest neighbourhood (using fancyimpute
2. Data set used and its link:	Heart Disease Dataset https://www.kaggle.com/ronitf/heart-disease-uci
Number of Techniques implemented in Bucket 4:	1

1. Names of the techniques implemented:	• Imputation Using the Value Zero
2. Data set used and its link:	Heart Disease Dataset https://www.kaggle.com/ronitf/heart-disease-uci

➔ Take any dataset where you understood about the attributes.
Dataset chosen: 'Heart Disease' dataset has been chosen(no missing values present)

- No of attributes = 14
- **Attributes**:

1. **age** – age in years
2. **sex** –
   Values- 1:male ; 2:female
3. **cp** – chest pain type
   Values- 1: typical angina ; 2: atypical angina ; 3: non-anginal pain ;
   4: asymptomatic
4. **trestbps** - resting blood pressure (in mm Hg on admission to the hospital)
5. **chol** - serum cholestoral (in mg/dl)
6. **fbs** - fasting blood sugar > 120 mg/dl
   Values- 1: true; 0: false
7. **restecg** - resting electrocardiographic results
   Values - 0: normal ; 1: having ST-T wave abnormality (T wave inversions and/or ST elevation or depression of > 0.05 mV) ; 2: showing probable or definite left ventricular hypertrophy by Estes' criteria
8. **thalach** - maximum heart rate achieved
9. **exang** - exercise induced angina
   Values- 1: yes ; 0: no
10. **oldpeak** - ST depression induced by exercise relative to rest
11. **slope** - the slope of the peak exercise ST segment
    Values- 1: upsloping ; 2: flat ; 3: downsloping
12. **ca** - number of major vessels (0-3) colored by flourosopy
13. **thal** –
    Values- 3: normal ; 6: fixed defect ; 7: reversable defect
14. **target** - diagnosis of heart disease (angiographic disease status)
    Values- 0: < 50% diameter narrowing ; 1: > 50% diameter narrowing (in any major vessel: attribute

➔ Read the dataset and print its head(first five rows of the dataset)

In [1]:
```
#import necessary modules
import pandas as pd

#read the csv file - heart dataset
heart_data=pd.read_csv('heart.csv')

#display the first five rows of the data
heart_data.head()
```

Out[1]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2	1
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2	1
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2	1
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2	1

➔ Print tail (last 5 rows of the dataset)

In [2]:
```
#display the first five rows of the data
heart_data.tail()
```

Out[2]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	target
298	57	0	0	140	241	0	1	123	1	0.2	1	0	3	0
299	45	1	3	110	264	0	1	132	0	1.2	1	0	3	0
300	68	1	0	144	193	1	1	141	0	3.4	1	2	3	0
301	57	1	0	130	131	0	1	115	1	1.2	1	1	3	0
302	57	0	1	130	236	0	0	174	0	0.0	1	1	2	0

➔ Understanding the datatypes of each attribute in the dataset.

In [3]:
```
#understanding the data type of each column of the dataset
heart_data.dtypes
```

Out[3]:
```
age           int64
sex           int64
cp            int64
trestbps      int64
chol          int64
fbs           int64
restecg       int64
thalach       int64
exang         int64
oldpeak     float64
slope         int64
ca            int64
thal          int64
target        int64
dtype: object
```

➔ The target column predicts if the patient has a heart disease

In [4]:
```
# select columns other than 'target' because it is the column we want to predict, so it should not be a part of the data
cols = [col for col in heart_data.columns if col not in ['target']]

# dropping the 'target' column from the dataset
data = heart_data[cols]

#assigning the 'target' column as our_target
our_target = heart_data['target']
data.head(n=2)
```

'target' column has been removed from the dataset

Out[4]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2

➔ Finding accuracy of the data using Naive-Bayes before adding missing values and preprocessing.

In [5]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train, data_test, our_target_train, our_target_test = train_test_split(data,our_target, test_size = 0.30, random_state =
```

In [7]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
#create an object of the type GaussianNB
gnb = GaussianNB()
#train the algorithm on training data and predict using the testing data
pred = gnb.fit(data_train, our_target_train).predict(data_test)
#print(pred.tolist())
#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7912087912087912
```

➔ Creating a copy of this data to data_one, so that data without missing values is not disturbed

In [102]:
```
data_one=data.copy()
data_one
```

Out[102]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2	172	199	1	1	162	0	0.5	2	0	3
9	57	1	2	150	168	0	1	174	0	1.6	2	0	2
10	54	1	0	140	239	0	1	160	0	1.2	2	0	2
11	48	0	2	130	275	0	1	139	0	0.2	2	0	2
12	49	1	1	130	266	0	1	171	0	0.6	2	0	2
13	64	1	3	110	211	0	0	144	1	1.8	1	0	2

➔ Creating missing values in one of the columns:

Challenges faced:

Tried different ways to code such that 20 percent of the column is randomly replaced with NaN values. Every time, the code compiles but arises a warning and does not allow the code to be executed.

Figured out why this warning comes up everytime. It is due to the chained assignment mode, read about this mode and understood that using '.loc' instead of putting the assignment in a loop will prevent this warning from blocking the execution.

Also, learned how to stop the warning message being shown in the output.

Finally, using .loc worked and missing values were created.

In [103]:

```
#Randomly replace 20% of the 'cp' column with NaN values
import random
import numpy as np
pd.set_option('mode.chained_assignment', None)
#def input_nan(x,pct):
#    n = int(len(x)*(pct - x.isna().mean()))
#    idxs = np.random.choice(len(x), max(n,0), replace=False, p=x.notna()/x.notna().sum())
#    x.iloc[idxs] = np.nan


#df[['y','x1']].apply(input_nan, pct=.15)
#data[['chol']].apply(input_nan,pct=.2)
int(len(data_one[['cp']])*(.2-data_one[['cp']].isna().mean()))
n=int(len(data_one[['cp']])*(.2-data_one[['cp']].isna().mean()))
n
idxs=np.random.choice(len(data_one[['cp']]),max(n,0),replace=False)
print (idxs)

#data[['chol']].iloc[idxs]=np.nan
#idxs[1]
#data.loc[i,['chol']]=np.nan
#data[['chol']].isna().sum()
for i in idxs:
    data_one.loc[i,['cp']]=np.nan

data_one.loc[:,['cp']]
```

➔ The idxs array consists of the indices in 'cp' to be converted to NaN

```
[ 66 164  39  69 166 105 147 272 129 121 268 243 239  77 170 110  74   4
 289  72 278 233 302  31  17 263 235 161   0 100 103 294  97 236 227 211
 113  16 260  26 240 241 245 279  52  81 127 299 124 213  45 185 222 232
 139 258 297 122  40 298]
```

Out[103]:

	cp
0	NaN
1	2.0
2	1.0
3	1.0
4	NaN
5	0.0
6	1.0
7	1.0
8	2.0
9	2.0
10	0.0
11	2.0
12	1.0

➔ 60 missing values created which is 20% of the column.

```
In [104]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
          data_one
```

Out[104]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	NaN	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	NaN	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	2.0	150	168	0	1	174	0	1.6	2	0	2
10	54	1	0.0	140	239	0	1	160	0	1.2	2	0	2

```
In [105]: #number of missing values present in the 'cp' column of our dataset(data_one)
          #20% of the total_number of rows = 60
          data_one.loc[:,['cp']].isna().sum()
```

```
Out[105]: cp    60
          dtype: int64
```

BUCKET 1

1.1 Imputation Using Forward Fill

In this imputation method, the column containing the missing values (cp) is considered. The missing values are filled with the preceding non missing value of the column.

Code:

```
#for forward-fill
data_one.fillna(method='ffill',inplace=True)
data_one
```

Link referred:
https://towardsdatascience.com/handling-missing-values-in-machine-learning-part-1-dda69d4f88ca

Challenges faced:
The first two values of the column cp were missing values and remained as missing values after using forward fill whereas other missing values were filled.

Later understood that since the first two values cannot have any preceding non missing values, these values cannot be filled, so they remain missing.
Again, created missing values randomly to the dataset so that there is no missing value in the index 0.

- Missing values at index 0 and 1 and 9

```
In [41]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
         data_one
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	NaN	145	233	1	0	150	0	2.3	0	0	1
1	37	1	NaN	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	NaN	150	168	0	1	174	0	1.6	2	0	2
10	54	1	0.0	140	239	0	1	160	0	1.2	2	0	2
11	48	0	2.0	130	275	0	1	139	0	0.2	2	0	2

- Index 0 and 1 was not filled but index 9 filled by its preceding value using forward fill

```
In [45]: #for forward-fill
         data_one.fillna(method='ffill',inplace=True)
         data_one
```

Out[45]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	NaN	145	233	1	0	150	0	2.3	0	0	1
1	37	1	NaN	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	2.0	150	168	0	1	174	0	1.6	2	0	2
10	54	1	0.0	140	239	0	1	160	0	1.2	2	0	2
11	48	0	2.0	130	275	0	1	139	0	0.2	2	0	2
12	49	1	2.0	130	266	0	1	171	0	0.6	2	0	2

Before Preprocessing : Before preprocessing, the ‘cp’ column has 60 missing values, that is, 20% of the total number of rows.

```
In [51]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
         data_one
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	NaN	130	204	0	0	172	0	1.4	2	0	2
3	56	1	NaN	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	2.0	150	168	0	1	174	0	1.6	2	0	2
10	54	1	NaN	140	239	0	1	160	0	1.2	2	0	2
11	48	0	2.0	130	275	0	1	139	0	0.2	2	0	2

After Preprocessing:

```
In [53]: #for forward-fill
         data_one.fillna(method='ffill',inplace=True)
         data_one
```

Out[53]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	2.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	2.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	2.0	150	168	0	1	174	0	1.6	2	0	2
10	54	1	2.0	140	239	0	1	160	0	1.2	2	0	2
11	48	0	2.0	130	275	0	1	139	0	0.2	2	0	2
12	49	1	1.0	130	266	0	1	171	0	0.6	2	0	2

Obervation:

- The missing values of the ‘cp’ column are filled by forward fill i.e., by the preceding non missing value.

- For example, the 3rd , 4rth ,11th value of the column ‘cp’, that is, cp[2] ,cp[3] and cp[10] which were missing values got filled with it’s preceding non missing value, that is, 2.0 (for all three).
- And also, all the integer values in all the columns have been changed to float values.

```
In [55]:
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_one, data_test_one, our_target_train_one, our_target_test_one = train_test_split(data_one,our_target, test_size =
```

```
In [56]:
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_one)
#using the train data (data_train_one) and train of target data(our_target_train_one)
pred = gnb.fit(data_train_one, our_target_train_one).predict(data_test_one)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_one, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7802197802197802
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.7802197802197802

1.2 Imputation Using Backward Fill

In this imputation method, the column containing the missing values (cp) is considered. The missing values are filled with the next non missing value of the column.

Code:

```
#for back fill
data_one.fillna(method='bfill',inplace=True)
data_one
```

Link referred:

https://towardsdatascience.com/handling-missing-values-in-machine-learning-part-1-dda69d4f88ca

Challenges faced:
No challenges faced. As the last value of the column is not a missing value, all the missing values were filled.

Before Preprocessing : Before preprocessing, the ‘cp’ column has 60 missing values, that is, 20% of the total number of rows.

```
In [29]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
         data_one
```

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	NaN	150	168	0	1	174	0	1.6	2	0	2
10	54	1	NaN	140	239	0	1	160	0	1.2	2	0	2
11	48	0	2.0	130	275	0	1	139	0	0.2	2	0	2

After Preprocessing:

```
In [34]: #for back fill
         data_one.fillna(method='bfill',inplace=True)
         data_one
```

Out[34]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	2.0	150	168	0	1	174	0	1.6	2	0	2
10	54	1	2.0	140	239	0	1	160	0	1.2	2	0	2
11	48	0	2.0	130	275	0	1	139	0	0.2	2	0	2

Obervation:

- The missing values of the ‘cp’ column are filled by backward fill i.e., by the next non missing value.
- For example, the 10th ,11th value of the column ‘cp’, that is, cp[9] and cp[10] which were missing values got filled with it’s next non missing value, that is, 2.0
- And also, all the integer values in all the columns have been changed to float values.

```
In [37]: #DIVIDING TEST DATA AND TRAIN DATA

         #importing train_test_split module
         from sklearn.model_selection import train_test_split

         #split data set into train and test sets using train_test_split() method
         #test_size = 0.30 :30% of the data is take as test data and 70% as train data
         #random_state ensures that we get a reproducible result every time
         data_train_one, data_test_one, our_target_train_one, our_target_test_one = train_test_split(data_one,our_target, test_size =
```

```
In [38]: # import the necessary module
         from sklearn.naive_bayes import GaussianNB
         from sklearn.metrics import accuracy_score

         #create an object of the type GaussianNB
         gnb = GaussianNB()

         #train the algorithm on training data and predict using the testing data
         #predict the test data (test_data_one)
         #using the train data (data_train_one) and train of target data(our_target_train_one)
         pred = gnb.fit(data_train_one, our_target_train_one).predict(data_test_one)

         #print the accuracy score of the model
         print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_one, pred, normalize = True))

         Naive-Bayes accuracy :  0.7802197802197802
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.7802197802197802

1.3 Imputation Using Mean

In this imputation method, the column containing the missing values (cp) is considered. The missing values are filled with the mean of the column containing the missing values.

Code:

```
#replace missing values with mean
data_one.cp.fillna(data_one.cp.mean(),inplace=True)
data_one
```

Link referred:
https://towardsdatascience.com/handling-missing-values-in-machine-learning-part-1-dda69d4f88ca

Challenges faced:
No challenges faced.

Before Preprocessing : Before preprocessing, the 'cp' column has 60 missing values, that is, 20% of the total number of rows.

```
In [83]:
#data_one is the dataset without the target column and with 20% missing values in the 'cp' column
data_one
```

Out[83]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	NaN	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	2.0	150	168	0	1	174	0	1.6	2	0	2
10	54	1	NaN	140	239	0	1	160	0	1.2	2	0	2

After Preprocessing:

```
In [86]:
#replace missing values with mean
data_one.cp.fillna(data_one.cp.mean(),inplace=True)
data_one
```

Out[86]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.00000	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.00000	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.00000	130	204	0	0	172	0	1.4	2	0	2
3	56	1	0.99177	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.00000	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.00000	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.00000	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.00000	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.00000	172	199	1	1	162	0	0.5	2	0	3
9	57	1	2.00000	150	168	0	1	174	0	1.6	2	0	2
10	54	1	0.99177	140	239	0	1	160	0	1.2	2	0	2
11	48	0	2.00000	130	275	0	1	139	0	0.2	2	0	2
12	49	1	1.00000	130	266	0	1	171	0	0.6	2	0	2
13	64	1	0.99177	110	211	0	0	144	1	1.8	1	0	2

Obervation:

- The missing values of the ‘cp’ column are filled by the mean of the non-missing values i.e, mean=0.99177
- For example, the 4th, 11th value of the column ‘cp’, that is, cp[3] and cp[10] which were missing values got filled with the mean, that is, 0.99177
- And also, all the integer values in all the columns have been changed to float values.

```
In [88]:
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_one, data_test_one, our_target_train_one, our_target_test_one = train_test_split(data_one,our_target, test_size =
```

```
In [89]:
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_one)
#using the train data (data_train_one) and train of target data(our_target_train_one)
pred = gnb.fit(data_train_one, our_target_train_one).predict(data_test_one)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_one, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7692307692307693
```

The Naïve-Bayes accuracy after preprocessing (0.7692307692307693) is approximately 3 percent less than the accuracy before preprocessing and removing missing values in the data. The other techniques gave a better accuracy and closer accuracy to the actual accuracy, so this preprocessing method is not advisable for this dataset.

1.4 Imputation Using Median

In this imputation method, the column containing the missing values (cp) is considered. The missing values are filled with the median of the column containing the missing values.

Code:

```
#replace missing values with median
data_one.cp.fillna(data_one.cp.median(),inplace=True)
data_one
```

Link referred:
https://towardsdatascience.com/handling-missing-values-in-machine-learning-part-1-dda69d4f88ca

Challenges faced:
No challenges faced.

Before Preprocessing : Before preprocessing, the 'cp' column has 60 missing values, that is, 20% of the total number of rows.

```
In [94]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
         data_one
```

Out[94]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	NaN	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2

After Preprocessing:

```
In [96]: #replace missing values with median
         data_one.cp.fillna(data_one.cp.median(),inplace=True)
         data_one
```

Out[96]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	1.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3

Obervation:

- The missing values of the 'cp' column are filled by the median of the non-missing values i.e, median=1.0
- For example, the 6th value of the column 'cp', that is, cp[5] which were missing values got filled with the median, that is, 1.0

- And also, all the integer values in all the columns have been changed to float values.

```
In [98]:
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_one, data_test_one, our_target_train_one, our_target_test_one = train_test_split(data_one,our_target, test_size =
```

```
In [99]:
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_one)
#using the train data (data_train_one) and train of target data(our_target_train_one)
pred = gnb.fit(data_train_one, our_target_train_one).predict(data_test_one)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_one, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7912087912087912
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing(0.7912087912087912) is very close to the accuracy before preprocessing and removing missing values in the data.

1.5 Imputation Using Mode

In this imputation method, the column containing the missing values (cp) is considered. The missing values are filled with the mode of the column containing the missing values.

Code:

```
#replace missing values with mode
data_one.cp.fillna(data_one.cp.mode()[0],inplace=True)
data_one
```

Link referred:
https://towardsdatascience.com/handling-missing-values-in-machine-learning-part-1-dda69d4f88ca

Challenges faced:
No challenges faced.

Before Preprocessing : Before preprocessing, the ‘cp’ column has 60 missing values, that is, 20% of the total number of rows.

```
In [104]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
          data_one
```

Out[104]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	NaN	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	NaN	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3

After Preprocessing:

```
In [111]: #replace missing values with mode
          data_one.cp.fillna(data_one.cp.mode()[0],inplace=True)
          data_one
```

Out[111]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	0.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3

Obervation:

- The missing values of the ‘cp’ column are filled by the median of the non-missing values i.e, mode=0.0
- For example, the 1st and 5fth values of the column ‘cp’, that is, cp[0] and cp[4] which were missing values got filled with the mode, that is, 0.0
- And also, all the integer values in all the columns have been changed to float values.

In [113]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_one, data_test_one, our_target_train_one, our_target_test_one = train_test_split(data_one,our_target, test_size =
```

In [114]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_one)
#using the train data (data_train_one) and train of target data(our_target_train_one)
pred = gnb.fit(data_train_one, our_target_train_one).predict(data_test_one)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_one, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7802197802197802
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.7802197802197802

BUCKET 2

2.1 Imputation Using Mean Values (using SimpleImputer module)

In this imputation method, each column is considered separately and then the missing values are filled with the mean of the non-missing values present in the column.

For this Heart Dataset, the column (cp) having missing values is filled with mean of the non-missing values of the column that is 0.987

Code:

```
#IMPUTATION USING MEAN VALUES

#Impute the values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer

#strategy is taken as'mean' because imputation using mean value should
be applied
imp_mean = SimpleImputer( strategy='mean')

#'fit' fits the imputer on data
```

```
imp_mean.fit(data_one)

#'transform' imputes all missing values with mean values in data_one an
d copies to imputed_data_one
imputed_data_one = imp_mean.transform(data_one)

#imputed_data_one is converted into a DataFrame
imputed_data_one=pd.DataFrame(imputed_data_one)

#Printing imputed_data_one
imputed_data_one
```

Link referred:

https://towardsdatascience.com/6-different-ways-to-compensate-for-missing-values-data-imputation-with-examples-6022d9ca0779

Challenges faced:

- Upgrading sklearn module was not an easy job. Sklearn module was already pre-installed but we required 0.21.3 version. While upgrading it, some packages like pandas got downgraded. Because of this issue, we had to uninstall and then again reinstall anaconda.

Before Preprocessing : Before preprocessing, the 'cp' column has 60 missing values, that is, 20% of the total number of rows.

```
In [11]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
         data_one
```

Out[11]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	NaN	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0.0	140	241	0	1	123	1	0.2	1	0	3
299	45	1	3.0	110	264	0	1	132	0	1.2	1	0	3
300	68	1	NaN	144	193	1	1	141	0	3.4	1	2	3
301	57	1	NaN	130	131	0	1	115	1	1.2	1	1	3
302	57	0	NaN	130	236	0	0	174	0	0.0	1	1	2

```
In [12]: #number of missing values present in the 'cp' column of our dataset(data_one)
         #20% of the total_number of rows = 60
         data_one.loc[:,['cp']].isna().sum()
```

```
Out[12]: cp    60
         dtype: int64
```

After Preprocessing:

In [13]:
```
#IMPUTATION USING MEAN VALUES

#Impute the values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer

#strategy is taken as'mean' because imputation using mean value should be applied
imp_mean = SimpleImputer( strategy='mean')

#'fit' fits the imputer on data
imp_mean.fit(data_one)

#'transform' imputes all missing values with mean values in data_one and copies to imputed_data_on
imputed_data_one = imp_mean.transform(data_one)

#imputed_data_one is converted into a DataFrame
imputed_data_one=pd.DataFrame(imputed_data_one)

#Printing imputed_data_one
imputed_data_one
```

Out[13]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	63.0	1.0	3.000000	145.0	233.0	1.0	0.0	150.0	0.0	2.3	0.0	0.0	1.0
1	37.0	1.0	2.000000	130.0	250.0	0.0	1.0	187.0	0.0	3.5	0.0	0.0	2.0
2	41.0	0.0	1.000000	130.0	204.0	0.0	0.0	172.0	0.0	1.4	2.0	0.0	2.0
3	56.0	1.0	0.987654	120.0	236.0	0.0	1.0	178.0	0.0	0.8	2.0	0.0	2.0
4	57.0	0.0	0.000000	120.0	354.0	0.0	1.0	163.0	1.0	0.6	2.0	0.0	2.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57.0	0.0	0.000000	140.0	241.0	0.0	1.0	123.0	1.0	0.2	1.0	0.0	3.0
299	45.0	1.0	3.000000	110.0	264.0	0.0	1.0	132.0	0.0	1.2	1.0	0.0	3.0
300	68.0	1.0	0.987654	144.0	193.0	1.0	1.0	141.0	0.0	3.4	1.0	2.0	3.0
301	57.0	1.0	0.987654	130.0	131.0	0.0	1.0	115.0	1.0	1.2	1.0	1.0	3.0
302	57.0	0.0	0.987654	130.0	236.0	0.0	0.0	174.0	0.0	0.0	1.0	1.0	2.0

Obervation:

- The missing values of the ‘cp’ column are filled by the mean of the non-missing values i.e, mean=0.987654
- For example, the 4th, 300th, 301th, 302th value of the column ‘cp’, that is, cp[3] ,cp[299],cp[300] and cp[301] which were missing values got filled with the mean, that is, 0.987654
- And also, all the integer values in all the columns have been changed to float values.

In [14]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_one, data_test_one, our_target_train_one, our_target_test_one = train_test_split(imputed_data_one,our_target
```

In [15]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score
#create an object of the type GaussianNB
gnb = GaussianNB()
#train the algorithm on training data and predict using the testing data
pred = gnb.fit(data_train_one, our_target_train_one).predict(data_test_one)
#print(pred.tolist())
#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_one, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7802197802197802
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.780219780219780

2.2 Imputation Using Median Values (using SimpleImputer module)

In this imputation method, each column is considered separately and then the missing values are filled with the median of the non-missing values present in the column.

For this Heart Dataset, the column (cp) having missing values is filled with median of the non-missing values of the column that is 1.0

Code:

```
#IMPUTATION USING MEDIAN VALUE

#Impute the values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer

#strategy is taken as'median' because imputation using median value sho
uld be applied
imp_median = SimpleImputer( strategy='median')

#'fit' fits the imputer on data
#data_one is our dataset without the target column
imp_median.fit(data_one)

#'transform' imputes all missing values with median values in data_one
and copies to imputed_data_two
imputed_data_two = imp_median.transform(data_one)

#imputed_data_two is converted into a DataFrame
```

```
imputed_data_two=pd.DataFrame(imputed_data_two)

#printing imputed_data_two
imputed_data_two
```

Link referred:

https://towardsdatascience.com/6-different-ways-to-compensate-for-missing-values-data-imputation-with-examples-6022d9ca0779

Challenges faced:
Not many challenges faced as this preprocessing method was similar to the imputation using mean.

Before Preprocessing : Before preprocessing, the 'cp' column has 60 missing values, that is, 20% of the total number of rows.

In [11]:
```
#data_one is the dataset without the target column and with 20% missing values in the 'cp' column
data_one
```

Out[11]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	NaN	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0.0	140	241	0	1	123	1	0.2	1	0	3
299	45	1	3.0	110	264	0	1	132	0	1.2	1	0	3
300	68	1	NaN	144	193	1	1	141	0	3.4	1	2	3
301	57	1	NaN	130	131	0	1	115	1	1.2	1	1	3
302	57	0	NaN	130	236	0	0	174	0	0.0	1	1	2

In [12]:
```
#number of missing values present in the 'cp' column of our dataset(data_one)
#20% of the total_number of rows = 60
data_one.loc[:,['cp']].isna().sum()
```

Out[12]:
```
cp    60
dtype: int64
```

After Preprocessing:

In [16]:
```
#IMPUTATION USING MEDIAN VALUE

#Impute the values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer

#strategy is taken as'median' because imputation using median value should be applied
imp_median = SimpleImputer( strategy='median')

#'fit' fits the imputer on data
#data_one is our dataset without the target column
imp_median.fit(data_one)

#'transform' imputes all missing values with median values in data_one and copies to imputed_data_two
imputed_data_two = imp_median.transform(data_one)

#imputed_data_two is converted into a DataFrame
imputed_data_two=pd.DataFrame(imputed_data_two)

#printing imputed_data_two
imputed_data_two
```

Out[16]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	63.0	1.0	3.0	145.0	233.0	1.0	0.0	150.0	0.0	2.3	0.0	0.0	1.0
1	37.0	1.0	2.0	130.0	250.0	0.0	1.0	187.0	0.0	3.5	0.0	0.0	2.0
2	41.0	0.0	1.0	130.0	204.0	0.0	0.0	172.0	0.0	1.4	2.0	0.0	2.0
3	56.0	1.0	1.0	120.0	236.0	0.0	1.0	178.0	0.0	0.8	2.0	0.0	2.0
4	57.0	0.0	0.0	120.0	354.0	0.0	1.0	163.0	1.0	0.6	2.0	0.0	2.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57.0	0.0	0.0	140.0	241.0	0.0	1.0	123.0	1.0	0.2	1.0	0.0	3.0
299	45.0	1.0	3.0	110.0	264.0	0.0	1.0	132.0	0.0	1.2	1.0	0.0	3.0
300	68.0	1.0	1.0	144.0	193.0	1.0	1.0	141.0	0.0	3.4	1.0	2.0	3.0
301	57.0	1.0	1.0	130.0	131.0	0.0	1.0	115.0	1.0	1.2	1.0	1.0	3.0
302	57.0	0.0	1.0	130.0	236.0	0.0	0.0	174.0	0.0	0.0	1.0	1.0	2.0

303 rows × 13 columns

Obervation:

- The missing values of the ‘cp’ column are filled by the mean of the non-missing values i.e, median=1.0
- For example, the 4th, 300th, 301th, 302th value of the column ‘cp’, that is, cp[3] ,cp[299],cp[300] and cp[301] which were missing values got filled with the mean, that is, 1.0
- And also, all the integer values in all the columns have been changed to float values.
- It was also observed that , how many ever times, the missing values indices are changed, the accuracy of both mean imputation and median imputation remained equal.

In [17]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_two, data_test_two, our_target_train_two, our_target_test_two = train_test_split(imputed_data_two,our_targe
t, test_size = 0.30, random_state = 10)
```

In [18]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()
#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_two)
#using the train data (data_train_two) and train of target data(our_target_train_two)
pred = gnb.fit(data_train_two, our_target_train_two).predict(data_test_two)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_two, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7802197802197802
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.7802197802

2.3 Imputation Using Most Frequent Values

In this imputation method, each column is considered separately and then the missing values are filled with the most frequently repeated values of the non-missing values present in the column.

For this Heart Dataset, the column (cp) having missing values is filled with the most frequent values of the non-missing values of the column that is 1.0

Code:

```
#IMPUTATION USING MOST FREQUENT VALUE

#Impute the values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer

#strategy is taken as'median' because imputation using median value sho
uld be applied
imp_median = SimpleImputer( strategy='median')

#'fit' fits the imputer on data
#data_one is our dataset without the target column
imp_median.fit(data_one)

#'transform' imputes all missing values with mean values in data_one an
d copies to imputed_data_two
imputed_data_two = imp_median.transform(data_one)
```

```
#imputed_data_two is converted into a DataFrame
imputed_data_two=pd.DataFrame(imputed_data_two)

#printing imputed_data_two
imputed_data_two
```

Link referred:

https://towardsdatascience.com/6-different-ways-to-compensate-for-missing-values-data-imputation-with-examples-6022d9ca0779

Challenges faced:
Not many challenges faced as this preprocessing method was similar to the imputation using mean, just had to change the strategy to 'most_frequent'.

Before Preprocessing : Before preprocessing, the 'cp' column has 60 missing values, that is, 20% of the total number of rows.

```
In [11]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
         data_one
```

Out[11]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	NaN	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0.0	140	241	0	1	123	1	0.2	1	0	3
299	45	1	3.0	110	264	0	1	132	0	1.2	1	0	3
300	68	1	NaN	144	193	1	1	141	0	3.4	1	2	3
301	57	1	NaN	130	131	0	1	115	1	1.2	1	1	3
302	57	0	NaN	130	236	0	0	174	0	0.0	1	1	2

```
In [12]: #number of missing values present in the 'cp' column of our dataset(data_one)
         #20% of the total_number of rows = 60
         data_one.loc[:,['cp']].isna().sum()
```

```
Out[12]: cp    60
         dtype: int64
```

After Preprocessing:

In [16]:
```
#IMPUTATION USING MEDIAN VALUE

#Impute the values using scikit-learn SimpleImpute Class
from sklearn.impute import SimpleImputer

#strategy is taken as'median' because imputation using median value should be applied
imp_median = SimpleImputer( strategy='median')

#'fit' fits the imputer on data
#data_one is our dataset without the target column
imp_median.fit(data_one)

#'transform' imputes all missing values with median values in data_one and copies to imputed_data_two
imputed_data_two = imp_median.transform(data_one)

#imputed_data_two is converted into a DataFrame
imputed_data_two=pd.DataFrame(imputed_data_two)

#printing imputed_data_two
imputed_data_two
```

Out[16]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	63.0	1.0	3.0	145.0	233.0	1.0	0.0	150.0	0.0	2.3	0.0	0.0	1.0
1	37.0	1.0	2.0	130.0	250.0	0.0	1.0	187.0	0.0	3.5	0.0	0.0	2.0
2	41.0	0.0	1.0	130.0	204.0	0.0	0.0	172.0	0.0	1.4	2.0	0.0	2.0
3	56.0	1.0	1.0	120.0	236.0	0.0	1.0	178.0	0.0	0.8	2.0	0.0	2.0
4	57.0	0.0	0.0	120.0	354.0	0.0	1.0	163.0	1.0	0.6	2.0	0.0	2.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57.0	0.0	0.0	140.0	241.0	0.0	1.0	123.0	1.0	0.2	1.0	0.0	3.0
299	45.0	1.0	3.0	110.0	264.0	0.0	1.0	132.0	0.0	1.2	1.0	0.0	3.0
300	68.0	1.0	1.0	144.0	193.0	1.0	1.0	141.0	0.0	3.4	1.0	2.0	3.0
301	57.0	1.0	1.0	130.0	131.0	0.0	1.0	115.0	1.0	1.2	1.0	1.0	3.0
302	57.0	0.0	1.0	130.0	236.0	0.0	0.0	174.0	0.0	0.0	1.0	1.0	2.0

303 rows × 13 columns

Obervation:

- The missing values of the ‘cp’ column are filled by the most frequent value of the non-missing values i.e, 1.0
- For example, the 4th, 300th, 301th, 302th value of the column ‘cp’, that is, cp[3] ,cp[299],cp[300] and cp[301] which were missing values got filled with the mean, that is, 1.0
- And also, all the integer values in all the columns have been changed to float values.
- It was also observed that , how many ever times, the missing values indices are changed, the accuracy of both mean imputation and median imputation remained equal.

```
In [17]: #DIVIDING TEST DATA AND TRAIN DATA

         #importing train_test_split module
         from sklearn.model_selection import train_test_split

         #split data set into train and test sets using train_test_split() method
         #test_size = 0.30 :30% of the data is take as test data and 70% as train data
         #random_state ensures that we get a reproducible result every time
         data_train_two, data_test_two, our_target_train_two, our_target_test_two = train_test_split(imputed_data_two,our_targe
         t, test_size = 0.30, random_state = 10)
```

```
In [18]: # import the necessary module
         from sklearn.naive_bayes import GaussianNB
         from sklearn.metrics import accuracy_score

         #create an object of the type GaussianNB
         gnb = GaussianNB()
         #train the algorithm on training data and predict using the testing data
         #predict the test data (test_data_two)
         #using the train data (data_train_two) and train of target data(our_target_train_two)
         pred = gnb.fit(data_train_two, our_target_train_two).predict(data_test_two)

         #print the accuracy score of the model
         print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_two, pred, normalize = True))

         Naive-Bayes accuracy :  0.7802197802197802
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.7802197802

2.4 Imputation Using Constant Value Outside the Fixed Value Range

In this imputation method, the column containing the missing values (cp) is considered. The missing values are filled with a constant value outside the fixed value range of the column. Example: -99

Code:

```
#Replace with some constant value outside fixed value range-999,-1 etc
data_one.cp.fillna(-99,inplace=True)
data_one
 imputed_data_one=pd.DataFrame(imputed_data_one)

#Printing imputed_data_one
imputed_data_one
```

Link referred:
https://towardsdatascience.com/handling-missing-values-in-machine-learning-part-1-dda69d4f88ca

Challenges faced:
No challenges faced.

Before Preprocessing : Before preprocessing, the ‘cp’ column has 60 missing values, that is, 20% of the total number of rows.

```
In [59]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
         data_one
```

15	50	0	2.0	120	219	0	1	158	0	1.6	1	0	2
16	58	0	NaN	120	340	0	1	172	0	0.0	2	0	2
17	66	0	3.0	150	226	0	1	114	0	2.6	0	0	2
18	43	1	0.0	150	247	0	1	171	0	1.5	2	0	2
19	69	0	3.0	140	239	0	1	151	0	1.8	2	2	2
20	59	1	0.0	135	234	0	1	161	0	0.5	1	0	3
21	44	1	2.0	130	233	0	1	179	1	0.4	2	0	2
22	42	1	NaN	140	226	0	1	178	0	0.0	2	0	2
23	61	1	2.0	150	243	1	1	137	1	1.0	1	0	2
24	40	1	NaN	140	199	0	1	178	1	1.4	2	0	3
25	71	0	1.0	160	302	0	1	162	0	0.4	2	2	2
26	59	1	NaN	150	212	1	1	157	0	1.6	2	0	2
27	51	1	2.0	110	175	0	1	123	0	0.6	2	0	2

After Preprocessing:

```
In [62]: #Replace with some constant value outside fixed value range-999,-1 etc
         data_one.cp.fillna(-9,inplace=True)
         data_one
```

Out[62]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	-9.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	2.0	150	168	0	1	174	0	1.6	2	0	2
10	54	1	0.0	140	239	0	1	160	0	1.2	2	0	2
11	48	0	2.0	130	275	0	1	139	0	0.2	2	0	2
12	49	1	1.0	130	266	0	1	171	0	0.6	2	0	2
13	64	1	3.0	110	211	0	0	144	1	1.8	1	0	2

14	58	0	3.0	150	283	1	0	162	0	1.0	2	0	2
15	50	0	2.0	120	219	0	1	158	0	1.6	1	0	2
16	58	0	-9.0	120	340	0	1	172	0	0.0	2	0	2
17	66	0	3.0	150	226	0	1	114	0	2.6	0	0	2
18	43	1	0.0	150	247	0	1	171	0	1.5	2	0	2
19	69	0	3.0	140	239	0	1	151	0	1.8	2	2	2
20	59	1	0.0	135	234	0	1	161	0	0.5	1	0	3
21	44	1	2.0	130	233	0	1	179	1	0.4	2	0	2
22	42	1	-9.0	140	226	0	1	178	0	0.0	2	0	2
23	61	1	2.0	150	243	1	1	137	1	1.0	1	0	2
24	40	1	-9.0	140	199	0	1	178	1	1.4	2	0	3
25	71	0	1.0	160	302	0	1	162	0	0.4	2	2	2
26	59	1	-9.0	150	212	1	1	157	0	1.6	2	0	2
27	51	1	2.0	110	175	0	1	123	0	0.6	2	0	2
28	65	0	2.0	140	417	1	0	157	0	0.8	2	1	2
29	53	1	-9.0	130	197	1	0	152	0	1.2	0	0	2

Obervation:

- The missing values of the ‘cp’ column are filled by the value -9.
- For example, the 17th, 23rd, 25th and 30th values of the column ‘cp’, that is, cp[16] ,cp[22],cp[24] and cp[29] which were missing values got replaced with -9.
- And also, all the integer values in all the columns have been changed to float values.

In [64]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_one, data_test_one, our_target_train_one, our_target_test_one = train_test_split(data_one,our_target, test_size =
```

In [65]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_one)
#using the train data (data_train_one) and train of target data(our_target_train_one)
pred = gnb.fit(data_train_one, our_target_train_one).predict(data_test_one)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_one, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7692307692307693
```

The Naïve-Bayes accuracy after preprocessing (0.7692307692307693) is approximately 3 percent less than the accuracy before preprocessing and removing missing values in the data. The other techniques gave a better

accuracy and closer accuracy to the actual accuracy, so this preprocessing method is not advisable for this dataset.

2.5 Drop the column with missing values

In this imputation method, the column containing the missing values (cp) is dropped from the dataset.

Code:

```
#drop a column (cp) as it has some missing values.
data_one.dropna(axis=1,inplace=True)
data_one
```

Link referred:
https://towardsdatascience.com/handling-missing-values-in-machine-learning-part-1-dda69d4f88ca

Challenges faced:
No challenges faced.

Before Preprocessing : Before preprocessing, the 'cp' column has 60 missing values, that is, 20% of the total number of rows.

```
In [76]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
         data_one
```

Out[76]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	NaN	120	354	0	1	163	1	0.6	2	0	2
5	57	1	NaN	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	1.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	2.0	150	168	0	1	174	0	1.6	2	0	2
10	54	1	0.0	140	239	0	1	160	0	1.2	2	0	2

After Preprocessing:

In [78]:
```
#drop a column (cp) as it has some missing values.
data_one.dropna(axis=1,inplace=True)
data_one
```

Out[78]:

	age	sex	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	145	233	1	0	150	0	2.3	0	0	1
1	37	1	130	250	0	1	187	0	3.5	0	0	2
2	41	0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	120	236	0	1	178	0	0.8	2	0	2
4	57	0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	140	192	0	1	148	0	0.4	1	0	1
6	56	0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	120	263	0	1	173	0	0.0	2	0	3

Obervation:

- The ‘cp’ column containing missing values is removed.

In [79]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_one, data_test_one, our_target_train_one, our_target_test_one = train_test_split(data_one,our_target, test_size =
```

In [80]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_one)
#using the train data (data_train_one) and train of target data(our_target_train_one)
pred = gnb.fit(data_train_one, our_target_train_one).predict(data_test_one)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_one, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7692307692307693
```

The Naïve-Bayes accuracy after preprocessing (0.7692307692307693) is approximately 3 percent less than the accuracy before preprocessing and removing missing values in the data. The other techniques gave a better accuracy and closer accuracy to the actual accuracy, so this preprocessing method is not advisable for this dataset.

2.6 Imputation Using k-NN (using fast_knn)

This implementation creates a basic mean impute then uses the resulting complete list to construct a KDTree. Then, it uses the resulting KDTree to compute nearest neighbours (NN). After it finds the k-NNs, it takes the weighted average of them.

Code:

```
#IMPUTATION USING k-NN

#sys module is imported
import sys

#fast_knn module is imported from impyute.imputation.cs library
from impyute.imputation.cs import fast_knn

#recursion doubles the amount of memory that is allocated by the functi
on and a stack is used to store the data.
#Increase the recursion limit of the OS to 100000 as python has a recur
ison depth of < 1000
sys.setrecursionlimit(100000)

#-------Finding the value of k ---  k = sqrt(total number of rows in ou
r data)
#A small value of k means that noise will have a higher influence on th
e result
#A large value make it computationally expensive

#Find the number of rows in th'data_one' dataframe
no_of_rows=data_one.shape[0]

#find the k value
import math
k_value=int(math.sqrt(no_of_rows))

# start the KNN training
imputed_data_four=fast_knn(data_one.values, k=k_value)

#imputed_data_four is converted into a dataframe
imputed_data_four=pd.DataFrame(imputed_data_four)

#print imputed_data_four, that is, the preprocessed dataset
imputed_data_four
imputed_data_three=pd.DataFrame(imputed_data_three)

#Printing imputed_data_three
imputed_data_three
```

Link referred:

https://towardsdatascience.com/6-different-ways-to-compensate-for-missing-values-data-imputation-with-examples-6022d9ca0779

https://discuss.analyticsvidhya.com/t/how-to-choose-the-value-of-k-in-knn-algorithm/2606/2

Challenges faced:

- Understanding the k-NN imputation method was quite challenging and interesting.
- Impyute module had to be installed as it was not present in the jupyter notebook
- Understanding which value that should be given to k parameter was not easy.

Before Preprocessing : Before preprocessing, the 'cp' column has 60 missing values, that is, 20% of the total number of rows.

In [131]:
```
#data_one is the dataset without the target column and with 20% missing values in the 'cp' column
data_one
```

Out[131]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	NaN	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0.0	140	241	0	1	123	1	0.2	1	0	3
299	45	1	3.0	110	264	0	1	132	0	1.2	1	0	3
300	68	1	0.0	144	193	1	1	141	0	3.4	1	2	3
301	57	1	0.0	130	131	0	1	115	1	1.2	1	1	3
302	57	0	NaN	130	236	0	0	174	0	0.0	1	1	2

303 rows × 13 columns

In [132]:
```
#number of missing values present in the 'cp' column of our dataset(data_one)
#20% of the total_number of rows = 60
data_one.loc[:,['cp']].isna().sum()
```

Out[132]:
```
cp    60
dtype: int64
```

After Preprocessing:

In [142]:
```
#IMPUTATION USING k-NN

#sys module is imported
import sys

#fast_knn module is imported from impyute.imputation.cs library
from impyute.imputation.cs import fast_knn

#recursion doubles the amount of memory that is allocated by the function and a stack is used to store the data.
#Increase the recursion limit of the OS to 100000 as python has a recurison depth of < 1000
sys.setrecursionlimit(100000)

#-------Finding the value of k ---  k = sqrt(total number of rows in our data)
#A small value of k means that noise will have a higher influence on the result
#A large value make it computationally expensive

#Find the number of rows in th'data_one' dataframe
no_of_rows=data_one.shape[0]

#find the k value
import math
k_value=int(math.sqrt(no_of_rows))

# start the KNN training
imputed_data_four=fast_knn(data_one.values, k=k_value)

#imputed_data_four is converted into a dataframe
imputed_data_four=pd.DataFrame(imputed_data_four)

#print imputed_data_four, that is, the preprocessed dataset
imputed_data_four
```

Out[142]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	63.0	1.0	3.000000	145.0	233.0	1.0	0.0	150.0	0.0	2.3	0.0	0.0	1.0
1	37.0	1.0	2.000000	130.0	250.0	0.0	1.0	187.0	0.0	3.5	0.0	0.0	2.0

Out[142]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	63.0	1.0	3.000000	145.0	233.0	1.0	0.0	150.0	0.0	2.3	0.0	0.0	1.0
1	37.0	1.0	2.000000	130.0	250.0	0.0	1.0	187.0	0.0	3.5	0.0	0.0	2.0
2	41.0	0.0	1.124502	130.0	204.0	0.0	0.0	172.0	0.0	1.4	2.0	0.0	2.0
3	56.0	1.0	1.000000	120.0	236.0	0.0	1.0	178.0	0.0	0.8	2.0	0.0	2.0
4	57.0	0.0	0.000000	120.0	354.0	0.0	1.0	163.0	1.0	0.6	2.0	0.0	2.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57.0	0.0	0.000000	140.0	241.0	0.0	1.0	123.0	1.0	0.2	1.0	0.0	3.0
299	45.0	1.0	3.000000	110.0	264.0	0.0	1.0	132.0	0.0	1.2	1.0	0.0	3.0
300	68.0	1.0	0.000000	144.0	193.0	1.0	1.0	141.0	0.0	3.4	1.0	2.0	3.0
301	57.0	1.0	0.000000	130.0	131.0	0.0	1.0	115.0	1.0	1.2	1.0	1.0	3.0
302	57.0	0.0	1.222635	130.0	236.0	0.0	0.0	174.0	0.0	0.0	1.0	1.0	2.0

303 rows × 13 columns

Obervation:

- Depending on the dataset, it can be much more accurate than the mean, median or most frequent imputation methods. But for this dataset, accuracy after the k-NN imputation is ~0.02 less than that of the accuracy after the mean/median imputation.
- And also, all the integer values in all the columns have been changed to float values.

In [143]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_four, data_test_four, our_target_train_four, our_target_test_four = train_test_split(imputed_data_four,our_
target, test_size = 0.30, random_state = 10)
```

In [144]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_four)
#using the train data (data_train_four) and train of target data(our_target_train_four)
pred = gnb.fit(data_train_four, our_target_train_four).predict(data_test_four)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_four, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7692307692307693
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.769230769

2.7 Imputation Using Multivariate Imputation by Chained Equation(MICE)

This type of imputation works by filling the missing data multiple times. Firstly, the data is imputed m times and stored separately. The only change in all the imputed datasets is the imputed values as non-missing values remain same. And then, regression coefficient is calculated on each dataset. And finally all the m Q-values are pooled up to Q' and variance is calculated. If the Q values are normally distributed(approx.), then we calculate the mean over all the Q values and sum the within- and between-imputation variance.

Code:

```
#import mice module from impyute library
from impyute.imputation.cs import mice

# start the MICE training
#mice is an in-built function in which our dataset(data_one) should be
given as input
imputed_data_five=mice(data_one.values)

#Converting imputed_data_five array into imputed_data_five dataframe(im
puted dataset)
imputed_data_five=pd.DataFrame(imputed_data_five)
```

```
#Printing the dataset after imputing(imputed_data_five)
imputed_data_five
```

Links referred:

https://towardsdatascience.com/6-different-ways-to-compensate-for-missing-values-data-imputation-with-examples-6022d9ca0779

https://www.jstatsoft.org/article/view/v045i03/v45i03.pdf

Challenges faced:

- Implementation of the MICE imputation method was straight-forward as we don't even have to learn about any parameters that should be given as input. 'mice' is the only in-built in which the dataset has to be given as the input.
- But understanding how the algorithm works internally was challenging as well as interesting.

Before Preprocessing : Before preprocessing, the 'cp' column has 60 missing values, that is, 20% of the total number of rows.

In [387]:
```
#data_one is the dataset without the target column and with 20% missing values in the 'cp' column
data_one
```

Out[387]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	NaN	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	NaN	140	241	0	1	123	1	0.2	1	0	3
299	45	1	3.0	110	264	0	1	132	0	1.2	1	0	3
300	68	1	0.0	144	193	1	1	141	0	3.4	1	2	3
301	57	1	NaN	130	131	0	1	115	1	1.2	1	1	3
302	57	0	1.0	130	236	0	0	174	0	0.0	1	1	2

303 rows × 13 columns

In [388]:
```
#number of missing values present in the 'cp' column of our dataset(data_one)
#20% of the total_number of rows = 60
data_one.loc[:,['cp']].isna().sum()
```

Out[388]:
```
cp    60
dtype: int64
```

After Preprocessing:

In [401]:
```
#import mice module from impyute library
from impyute.imputation.cs import mice

# start the MICE training
#mice is an in-built function in which our dataset(data_one) should be given as input
imputed_data_five=mice(data_one.values)

#Converting imputed_data_five array into imputed_data_five dataframe(imputed dataset)
imputed_data_five=pd.DataFrame(imputed_data_five)

#Printing the dataset after imputing(imputed_data_five)
imputed_data_five
```

Out[401]:

	0	1	2	3	4	5	6	7	8	9	10	11	12
0	63.0	1.0	1.611837	145.0	233.0	1.0	0.0	150.0	0.0	2.3	0.0	0.0	1.0
1	37.0	1.0	2.000000	130.0	250.0	0.0	1.0	187.0	0.0	3.5	0.0	0.0	2.0
2	41.0	0.0	1.000000	130.0	204.0	0.0	0.0	172.0	0.0	1.4	2.0	0.0	2.0
3	56.0	1.0	1.000000	120.0	236.0	0.0	1.0	178.0	0.0	0.8	2.0	0.0	2.0
4	57.0	0.0	0.000000	120.0	354.0	0.0	1.0	163.0	1.0	0.6	2.0	0.0	2.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57.0	0.0	0.355805	140.0	241.0	0.0	1.0	123.0	1.0	0.2	1.0	0.0	3.0
299	45.0	1.0	3.000000	110.0	264.0	0.0	1.0	132.0	0.0	1.2	1.0	0.0	3.0
300	68.0	1.0	0.000000	144.0	193.0	1.0	1.0	141.0	0.0	3.4	1.0	2.0	3.0
301	57.0	1.0	0.127936	130.0	131.0	0.0	1.0	115.0	1.0	1.2	1.0	1.0	3.0
302	57.0	0.0	1.000000	130.0	236.0	0.0	0.0	174.0	0.0	0.0	1.0	1.0	2.0

303 rows × 13 columns

Observation:

In [402]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_five, data_test_five, our_target_train_five, our_target_test_five = train_test_split(imputed_data_five,our_
target, test_size = 0.30, random_state = 10)
```

In [403]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_five)
#using the train data (data_train_five) and train of target data(our_target_train_five)
pred = gnb.fit(data_train_five, our_target_train_five).predict(data_test_five)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_five, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7802197802197802
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.780219

2.8 Imputation Using Deep Learning(datawig)

In this imputation method, Deep learning is used and implemented using datawig module. It learns Machine Learning models using Deep Neural Networks to impute missing values in a dataframe. It also supports both CPU and GPU for training.

Code:

```
#IMPUTATION USING DEEP LEARNING

#import datawig module
import datawig

#divding the dataset into train and test data randomly random_split inb
uilt function of the datawig module
#(data_one) dataset is given as the parameter
df_train, df_test = datawig.utils.random_split(data_one)

# Initialize a SimpleImputer model:
#input_columns - columns containing information about the column we wan
t to impute
#output_column - the column we'd like to impute values for
#output_path - stores model data and metrics

imputer = datawig.SimpleImputer(

    input_columns=['age','sex','trestbps','chol','fbs','restecg','thala
ch','exang','oldpeak','slope','ca','thal'],
    output_column= 'cp',

    output_path = 'imputer_model'

    )

#'fit' fits an imputer model on the train data
imputer.fit(train_df=df_train, num_epochs=50)

#Impute missing values and return original dataframe with predictions
imputed = imputer.predict(df_test)
```

Links referred:

https://towardsdatascience.com/6-different-ways-to-compensate-for-missing-values-data-imputation-with-examples-6022d9ca0779

For resolving datawig installation issue -

https://github.com/cvxgrp/cvxpy/issues/503

https://github.com/cvxgrp/cvxpy/issues/727

Challenges faced:

- We faced problems while installing datawig module initially as ‘the building wheel for scs’ failed several times. Huge amount of time had to be invested to dig down the problem.
- Finally, it was understood that an internal C code was not getting compiled , hence XCode had to be installed.

Before Preprocessing : Before preprocessing, the ‘cp’ column has 60 missing values, that is, 20% of the total number of rows.

In [387]:
```
#data_one is the dataset without the target column and with 20% missing values in the 'cp' column
data_one
```

Out[387]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	NaN	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	NaN	140	241	0	1	123	1	0.2	1	0	3
299	45	1	3.0	110	264	0	1	132	0	1.2	1	0	3
300	68	1	0.0	144	193	1	1	141	0	3.4	1	2	3
301	57	1	NaN	130	131	0	1	115	1	1.2	1	1	3
302	57	0	1.0	130	236	0	0	174	0	0.0	1	1	2

303 rows × 13 columns

In [388]:
```
#number of missing values present in the 'cp' column of our dataset(data_one)
#20% of the total_number of rows = 60
data_one.loc[:,['cp']].isna().sum()
```

Out[388]:
```
cp    60
dtype: int64
```

After Preprocessing:

In [404]:
```
#IMPUTATION USING DEEP LEARNING

#import datawig module
import datawig

#divding the dataset into train and test data randomly random_split inbuilt function of the datawig module
#(data_one) dataset is given as the parameter
df_train, df_test = datawig.utils.random_split(data_one)

# Initialize a SimpleImputer model:
#input_columns - columns containing information about the column we want to impute
#output_column - the column we'd like to impute values for
#output_path - stores model data and metrics

imputer = datawig.SimpleImputer(

    input_columns=['age','sex','trestbps','chol','fbs','restecg','thalach','exang','oldpeak','slope','ca','thal'],

    output_column= 'cp',

    output_path = 'imputer_model'

    )

#'fit' fits an imputer model on the train data
imputer.fit(train_df=df_train, num_epochs=50)

#Impute missing values and return original dataframe with predictions
imputed = imputer.predict(df_test)
```

```
In [405]: #printing imputed dataframe(dataset after imputation using Deep Learning)
          imputed
```

Out[405]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal	cp_imputed
85	67	0	2.0	115	564	0	0	160	0	1.6	1	0	3	0.931184
134	41	0	1.0	126	306	0	1	163	0	0.0	2	0	2	1.229499
143	67	0	0.0	106	223	0	1	142	0	0.3	2	2	2	0.842939
253	67	1	0.0	100	299	0	0	125	1	0.9	1	2	2	0.101906
205	52	1	0.0	128	255	0	1	161	1	0.0	2	1	3	0.310248
266	55	0	NaN	180	327	0	2	117	1	3.4	1	0	2	0.930230
104	50	1	2.0	129	196	0	1	163	0	0.0	2	0	2	0.869311
34	51	1	NaN	125	213	0	0	125	1	1.4	2	1	2	0.389835
182	61	0	0.0	130	330	0	0	169	0	0.0	2	0	2	0.962328
108	50	0	1.0	120	244	0	1	162	0	1.1	2	0	2	1.091191
50	51	0	2.0	130	256	0	0	149	0	0.5	2	0	2	1.106455
254	59	1	3.0	160	273	0	0	125	0	0.0	2	0	2	0.869754
88	54	0	2.0	110	214	0	1	158	0	1.6	1	0	2	1.210933
263	63	0	0.0	108	269	0	1	169	1	1.8	1	2	2	-0.318404
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2	0.159982
133	41	1	1.0	110	235	0	1	153	0	0.0	2	0	2	1.078815
44	39	1	2.0	140	321	0	0	182	0	0.0	2	0	2	0.931169
250	51	1	0.0	140	298	0	1	122	1	4.2	1	3	3	0.265805
203	68	1	2.0	180	274	1	0	150	1	1.6	1	0	3	1.516696
135	49	0	0.0	130	269	0	1	163	0	0.0	2	0	2	1.214099
145	70	1	1.0	156	245	0	0	143	0	0.0	2	0	2	0.919383
71	51	1	2.0	94	227	0	1	154	1	0.0	2	1	3	0.077375
118	46	0	1.0	105	204	0	1	172	0	0.0	2	0	2	1.126441
274	47	1	0.0	110	275	0	0	118	1	1.0	1	1	2	0.102866
300	68	1	0.0	144	193	1	1	141	0	3.4	1	2	3	0.971255
216	62	0	2.0	130	263	0	1	97	0	1.2	1	1	3	1.253349
74	43	0	2.0	122	213	0	1	165	0	0.2	1	0	2	1.465734
298	57	0	NaN	140	241	0	1	123	1	0.2	1	0	3	0.425907

Observation:

- After this imputation method is applied, the ‘cp’ column, the only column containing missing values has not be modified. Instead, a new column, that is, cp_imputed has been added to the dataset with the imputed values.
- Unlike the above imputation methods, other integer value columns have not been changed to float values. The entire dataset remains exactly same except for thr addition of new column that is imputed.

In [402]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_five, data_test_five, our_target_train_five, our_target_test_five = train_test_split(imputed,our_target, tes
```

In [403]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_five)
#using the train data (data_train_five) and train of target data(our_target_train_five)
pred = gnb.fit(data_train_five, our_target_train_five).predict(data_test_five)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_five, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7802197802197802
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.780219780

BUCKET 3

3.1 Imputation Using Linear Regression

In this imputation method, a correlation matrix is used to identify several predictors of the variable with missing values. The best predictors are selected and used as independent variables in a regression equation. The variable with missing data is used as the dependent variable. And then the cases with complete data for the predictor variables are used to generate the regression equation; the equation is then used to predict missing values for incomplete cases. In an iterative process, values for the missing variable are inserted and then all cases are used to predict the dependent variable. These steps are repeated until there is little difference between the predicted values from one step to the next.

Code:

```
#import LinearRegression from sklearn library
 from sklearn.linear_model import LinearRegression
linreg = LinearRegression()
data_LR = data_one[['age','sex','cp','trestbps','chol','fbs','restecg',
'thalach','exang','oldpeak','slope','ca','thal']]

#Step-1: Spliting the dataset that contains the missing values and no m
issing values into test and train dataframe respectively.
```

```
x_train = data_LR[data_LR['cp'].notnull()].drop(columns='cp')
y_train = data_LR[data_LR['cp'].notnull()]['cp']
x_test = data_LR[data_LR['cp'].isnull()].drop(columns='cp')
y_test = data_LR[data_LR['cp'].isnull()]['cp']

#Step-2: Training the algorithm
linreg.fit(x_train, y_train)

#Step-3: Predict the missing values in the attribute of the test data.
predicted = linreg.predict(x_test)

#Step-4: Let's obtain the complete dataset by combining with the target
attribute.
data_one.cp[data_one.cp.isnull()] = predicted
data_one
```

Links referred:

https://stackoverflow.com/questions/44097633/imputing-missing-values-using-a-linear-regression-in-python

Challenges faced:

- No mentionable challenge was faced while imputation using linear regression.

Before Preprocessing : Before preprocessing, the 'cp' column has 60 missing values, that is, 20% of the total number of rows.

```
In [507]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
          data_one
```

Out[507]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	NaN	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	NaN	120	354	0	1	163	1	0.6	2	0	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	NaN	140	241	0	1	123	1	0.2	1	0	3
299	45	1	3.0	110	264	0	1	132	0	1.2	1	0	3
300	68	1	0.0	144	193	1	1	141	0	3.4	1	2	3
301	57	1	0.0	130	131	0	1	115	1	1.2	1	1	3
302	57	0	NaN	130	236	0	0	174	0	0.0	1	1	2

```
In [508]: #number of missing values present in the 'cp' column of our dataset(data_one)
          #20% of the total_number of rows = 60
          data_one.loc[:,['cp']].isna().sum()
```

```
Out[508]: cp    60
          dtype: int64
```

After Preprocessing:

In [495]:
```
#import LinearRegression from sklearn library
from sklearn.linear_model import LinearRegression
linreg = LinearRegression()
data_LR = data_one[['age','sex','cp','trestbps','chol','fbs','restecg','thalach','exang','oldpeak','slope','ca','thal'
]]

#Step-1: Spliting the dataset that contains the missing values and no missing values into test and train dataframe res
pectively.
x_train = data_LR[data_LR['cp'].notnull()].drop(columns='cp')
y_train = data_LR[data_LR['cp'].notnull()]['cp']
x_test = data_LR[data_LR['cp'].isnull()].drop(columns='cp')
y_test = data_LR[data_LR['cp'].isnull()]['cp']

#Step-2: Training the algorithm
linreg.fit(x_train, y_train)

#Step-3: Predict the missing values in the attribute of the test data.
predicted = linreg.predict(x_test)

#Step-4: Let's obtain the complete dataset by combining with the target attribute.
data_one.cp[data_one.cp.isnull()] = predicted
data_one
```

Out[495]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.000000	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.000000	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.000000	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.000000	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.000000	120	354	0	1	163	1	0.6	2	0	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0.223713	140	241	0	1	123	1	0.2	1	0	3
299	45	1	3.000000	110	264	0	1	132	0	1.2	1	0	3
300	68	1	0.000000	144	193	1	1	141	0	3.4	1	2	3
301	57	1	0.000000	130	131	0	1	115	1	1.2	1	1	3
302	57	0	1.000000	130	236	0	0	174	0	0.0	1	1	2

303 rows × 13 columns

Observation:

In [496]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_six, data_test_six, our_target_train_six, our_target_test_six = train_test_split(data_one,our_target, test_
size = 0.30, random_state = 10)
```

In [497]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_six)
#using the train data (data_train_six) and train of target data(our_target_train_six)
pred = gnb.fit(data_train_six, our_target_train_six).predict(data_test_six)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_six, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7912087912087912
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.7912087912087912

3.2 Imputation Using k-nearest neighbourhood (using fancyimpute)

In this imputation method, a correlation matrix is used to identify several predictors of the variable with missing values. The best predictors are selected and used as independent variables in a regression equation. The variable with missing data is used as the dependent variable. And then the cases with complete data for the predictor variables are used to generate the regression equation; the equation is then used to predict missing values for incomplete cases. In an iterative process, values for the missing variable are inserted and then all cases are used to predict the dependent variable. These steps are repeated until there is little difference between the predicted values from one step to the next.

Code:

```
#import KNN module from fancyimpute library
from fancyimpute import KNN

#We use the data_one dataframe from heart dataset
#fancy impute removes column names.
train_cols = list(data_one)

# Use 5 nearest rows which have a feature to fill in each row's missing
features
data_one = pd.DataFrame(KNN(k=5).fit_transform(data_one))

#train_cols is inserted into data_one column-wise
data_one.columns = train_cols
data_one
```

Links referred:

https://www.analyticsvidhya.com/blog/2018/03/introduction-k-neighbours-algorithm-clustering/

Challenges faced:

- No mentionable challenge was faced while implementing imputation using KNN

Before Preprocessing : Before preprocessing, the ‘cp’ column has 60 missing values, that is, 20% of the total number of rows.

In [533]:
```
#data_one is the dataset without the target column and with 20% missing values in the 'cp' column
data_one
```

Out[533]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	NaN	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	1.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57	0	0.0	140	241	0	1	123	1	0.2	1	0	3
299	45	1	3.0	110	264	0	1	132	0	1.2	1	0	3
300	68	1	0.0	144	193	1	1	141	0	3.4	1	2	3
301	57	1	NaN	130	131	0	1	115	1	1.2	1	1	3
302	57	0	NaN	130	236	0	0	174	0	0.0	1	1	2

303 rows × 13 columns

In [534]:
```
#number of missing values present in the 'cp' column of our dataset(data_one)
#20% of the total_number of rows = 60
data_one.loc[:,['cp']].isna().sum()
```

Out[534]:
```
cp    60
dtype: int64
```

After Preprocessing:

In [551]:
```
#import KNN module from fancyimpute library
from fancyimpute import KNN

#We use the data_one dataframe from heart dataset
#fancy impute removes column names.
train_cols = list(data_one)

# Use 5 nearest rows which have a feature to fill in each row's missing features
data_one = pd.DataFrame(KNN(k=5).fit_transform(data_one))

#train_cols is inserted into data_one column-wise
data_one.columns = train_cols
data_one
```

```
Imputing row 1/303 with 1 missing, elapsed time: 0.030
Imputing row 101/303 with 0 missing, elapsed time: 0.031
Imputing row 201/303 with 0 missing, elapsed time: 0.033
Imputing row 301/303 with 0 missing, elapsed time: 0.034
```

Out[551]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63.0	1.0	1.990083	145.0	233.0	1.0	0.0	150.0	0.0	2.3	0.0	0.0	1.0
1	37.0	1.0	2.000000	130.0	250.0	0.0	1.0	187.0	0.0	3.5	0.0	0.0	2.0
2	41.0	0.0	1.000000	130.0	204.0	0.0	0.0	172.0	0.0	1.4	2.0	0.0	2.0
3	56.0	1.0	1.000000	120.0	236.0	0.0	1.0	178.0	0.0	0.8	2.0	0.0	2.0
4	57.0	0.0	0.000000	120.0	354.0	0.0	1.0	163.0	1.0	0.6	2.0	0.0	2.0
...	...	...	...	...	...	...	...	...	...	...	...	...	...
298	57.0	0.0	0.000000	140.0	241.0	0.0	1.0	123.0	1.0	0.2	1.0	0.0	3.0
299	45.0	1.0	3.000000	110.0	264.0	0.0	1.0	132.0	0.0	1.2	1.0	0.0	3.0
300	68.0	1.0	0.000000	144.0	193.0	1.0	1.0	141.0	0.0	3.4	1.0	2.0	3.0
301	57.0	1.0	0.719560	130.0	131.0	0.0	1.0	115.0	1.0	1.2	1.0	1.0	3.0
302	57.0	0.0	1.394522	130.0	236.0	0.0	0.0	174.0	0.0	0.0	1.0	1.0	2.0

303 rows × 13 columns

Observation:

In [552]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_six, data_test_six, our_target_train_six, our_target_test_six = train_test_split(data_one,our_target, test_
size = 0.30, random_state = 10)
```

In [553]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_six)
#using the train data (data_train_six) and train of target data(our_target_train_six)
pred = gnb.fit(data_train_six, our_target_train_six).predict(data_test_six)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_six, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7692307692307693
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing is nearer to the accuracy before preprocessing and removing missing values in the data, that is, 0.769230769230

BUCKET 4

4.1 Imputation Using the Value Zero

In this imputation method, the column containing the missing values (cp) is considered. The missing values are filled with zero.

- Trying to understand if filling the missing values with zero is useful for this dataset.

Code:

```
#Imputation by zero i.e., by filling the missing values with 0
data_one.cp.fillna(0,inplace=True)
data_one
```

Before Preprocessing : Before preprocessing, the ‘cp’ column has 60 missing values, that is, 20% of the total number of rows.

```
In [68]: #data_one is the dataset without the target column and with 20% missing values in the 'cp' column
         data_one
```

Out[68]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	NaN	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	NaN	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	NaN	150	168	0	1	174	0	1.6	2	0	2
10	54	1	0.0	140	239	0	1	160	0	1.2	2	0	2

After Preprocessing:

```
In [70]: #Imputation by zero i.e., by filling the missing values with 0
         data_one.cp.fillna(0,inplace=True)
         data_one
```

Out[70]:

	age	sex	cp	trestbps	chol	fbs	restecg	thalach	exang	oldpeak	slope	ca	thal
0	63	1	3.0	145	233	1	0	150	0	2.3	0	0	1
1	37	1	2.0	130	250	0	1	187	0	3.5	0	0	2
2	41	0	0.0	130	204	0	0	172	0	1.4	2	0	2
3	56	1	1.0	120	236	0	1	178	0	0.8	2	0	2
4	57	0	0.0	120	354	0	1	163	1	0.6	2	0	2
5	57	1	0.0	140	192	0	1	148	0	0.4	1	0	1
6	56	0	1.0	140	294	0	0	153	0	1.3	1	0	2
7	44	1	0.0	120	263	0	1	173	0	0.0	2	0	3
8	52	1	2.0	172	199	1	1	162	0	0.5	2	0	3
9	57	1	0.0	150	168	0	1	174	0	1.6	2	0	2
10	54	1	0.0	140	239	0	1	160	0	1.2	2	0	2
11	48	0	2.0	130	275	0	1	139	0	0.2	2	0	2
12	49	1	0.0	130	266	0	1	171	0	0.6	2	0	2
13	64	1	3.0	110	211	0	0	144	1	1.8	1	0	2

Obervation:

- The missing values of the ‘cp’ column are replaced with 0.
- For example,here the 3rd , 8th ,10th value of the column ‘cp’, that is, cp[2] ,cp[7] and cp[9] which were missing values got filled with 0.
- And also, all the integer values in all the columns have been changed to float values.

In [72]:
```
#DIVIDING TEST DATA AND TRAIN DATA

#importing train_test_split module
from sklearn.model_selection import train_test_split

#split data set into train and test sets using train_test_split() method
#test_size = 0.30 :30% of the data is take as test data and 70% as train data
#random_state ensures that we get a reproducible result every time
data_train_one, data_test_one, our_target_train_one, our_target_test_one = train_test_split(data_one,our_target, test_size =
```

In [73]:
```
# import the necessary module
from sklearn.naive_bayes import GaussianNB
from sklearn.metrics import accuracy_score

#create an object of the type GaussianNB
gnb = GaussianNB()

#train the algorithm on training data and predict using the testing data
#predict the test data (test_data_one)
#using the train data (data_train_one) and train of target data(our_target_train_one)
pred = gnb.fit(data_train_one, our_target_train_one).predict(data_test_one)

#print the accuracy score of the model
print("Naive-Bayes accuracy : ",accuracy_score(our_target_test_one, pred, normalize = True))
```

```
Naive-Bayes accuracy :  0.7912087912087912
```

Yes, this preprocessing is useful to the dataset as the Naïve-Bayes accuracy after preprocessing(0.7912087912087912) is very close to the accuracy before preprocessing and removing missing values in the data.