Water Potability Prediction - Using Logistic Regression

1. Water potability refers to the safety of water for human consumption. 2. Potable water is free from harmful contaminants and bacteria and is safe for drinking and food preparation. 3. The World Health Organization considers access to safe drinking water a basic human right. 4. There are various methods to ensure water potability, including filtration and treatment processes such as UV filtration and reverse osmosis. 5. In developed countries, tap water meets drinking water quality standards, although only a small proportion is actually consumed or used in food preparation. 6. If you are concerned about the potability of your water, you can have it tested

In [59]:
```
#Libararies used for data manipulation and visualization

import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')
```

In [60]:
```
#Load the data and accessing first fe columns from of it

water_data=pd.read_csv('water_potability.csv')
water_data.head()
```

Out[60]:

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalo
0	NaN	204.890455	20791.318981	7.300212	368.516441	564.308654	10.379783	
1	3.716080	129.422921	18630.057858	6.635246	NaN	592.885359	15.180013	
2	8.099124	224.236259	19909.541732	9.275884	NaN	418.606213	16.868637	
3	8.316766	214.373394	22018.417441	8.059332	356.886136	363.266516	18.436524	1
4	9.092223	181.101509	17978.986339	6.546600	310.135738	398.410813	11.558279	

In [5]:
```
water_data.columns  #Accessing columns from the dataset
```

Out[5]:
```
Index(['ph', 'Hardness', 'Solids', 'Chloramines', 'Sulfate', 'Conductivity',
       'Organic_carbon', 'Trihalomethanes', 'Turbidity', 'Potability'],
      dtype='object')
```

Discription of the columns present: ph:pH of water Hardness : Capacity of water to precipitate soap in mg/L Solids : Total dissolved solids in ppm Chloramines : Amount of Chloramines in ppm Sulfate : Amount of Sulfates dissolved in mg/L Conductivity : Electrical conductivity of water in ps/cm Organic_carbon : Amount of organic carbon in ppm Trihalomethanes : Amount of Trihalomethanes in ug/L Turbidity : Measure of light emiting property of water in NTU (Nephelometric Turbidity Units) Potability : Indicates if water is safe for human consumption

In [6]:
```
water_data.shape #Checking no. of rows and columns present
```

Out[6]:
```
(3276, 10)
```

In [61]:
```
water_data.describe()   #Checking for statistical information
```

Out[61]:

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbo
count	2785.000000	3276.000000	3276.000000	3276.000000	2495.000000	3276.000000	3276.00000
mean	7.080795	196.369496	22014.092526	7.122277	333.775777	426.205111	14.28497
std	1.594320	32.879761	8768.570828	1.583085	41.416840	80.824064	3.30816
min	0.000000	47.432000	320.942611	0.352000	129.000000	181.483754	2.20000
25%	6.093092	176.850538	15666.690297	6.127421	307.699498	365.734414	12.06580
50%	7.036752	196.967627	20927.833607	7.130299	333.073546	421.884968	14.21833
75%	8.062066	216.667456	27332.762127	8.114887	359.950170	481.792304	16.55765
max	14.000000	323.124000	61227.196008	13.127000	481.030642	753.342620	28.30000

In [62]:
```
#Check for insights from the data

water_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3276 entries, 0 to 3275
Data columns (total 10 columns):
 #   Column           Non-Null Count  Dtype  
---  ------           --------------  -----  
 0   ph               2785 non-null   float64
 1   Hardness         3276 non-null   float64
 2   Solids           3276 non-null   float64
 3   Chloramines      3276 non-null   float64
 4   Sulfate          2495 non-null   float64
 5   Conductivity     3276 non-null   float64
 6   Organic_carbon   3276 non-null   float64
 7   Trihalomethanes  3114 non-null   float64
 8   Turbidity        3276 non-null   float64
 9   Potability       3276 non-null   int64  
dtypes: float64(9), int64(1)
memory usage: 256.1 KB
```

In [8]:
```
#Checking for duplicates value

water_data.duplicated().any()
```

Out[8]:
```
False
```

In [9]:
```
#Checking for null values

water_data.isnull().sum()
```

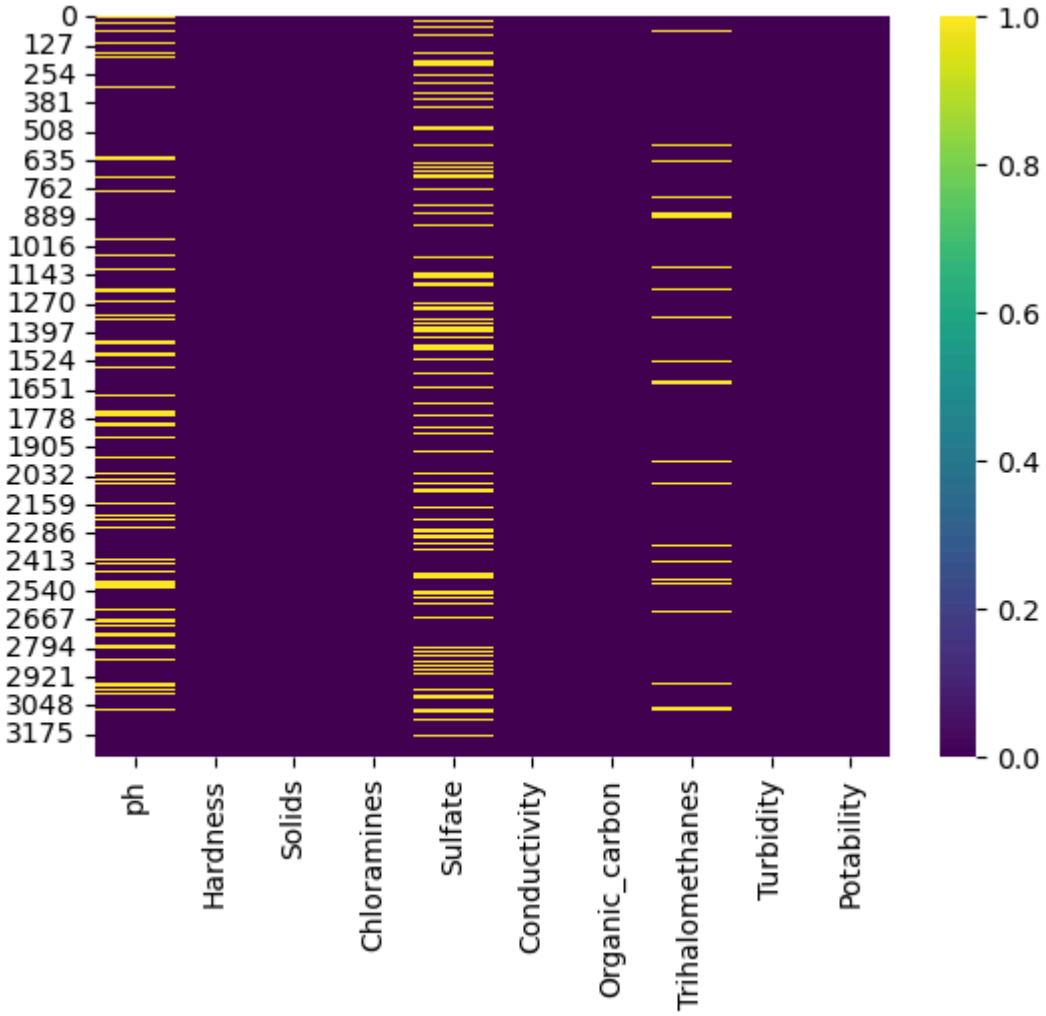
Out[9]:
```
ph                491
Hardness            0
Solids              0
Chloramines         0
Sulfate           781
Conductivity        0
Organic_carbon      0
Trihalomethanes   162
Turbidity           0
Potability          0
dtype: int64
```

In [63]:
```
#this code creates a DataFrame null_df that contains information about the number and
#in each column of the water_data DataFrame.

null_df = water_data.isnull().sum().reset_index()
null_df.columns = ['column', 'Null_count']
null_df["%miss_value"] = round(null_df['Null_count']/len(water_data),2)*100   #roundir
null_df
```

Out[63]:

	column	Null_count	%miss_value
0	ph	491	15.0
1	Hardness	0	0.0
2	Solids	0	0.0
3	Chloramines	0	0.0
4	Sulfate	781	24.0
5	Conductivity	0	0.0
6	Organic_carbon	0	0.0
7	Trihalomethanes	162	5.0
8	Turbidity	0	0.0
9	Potability	0	0.0

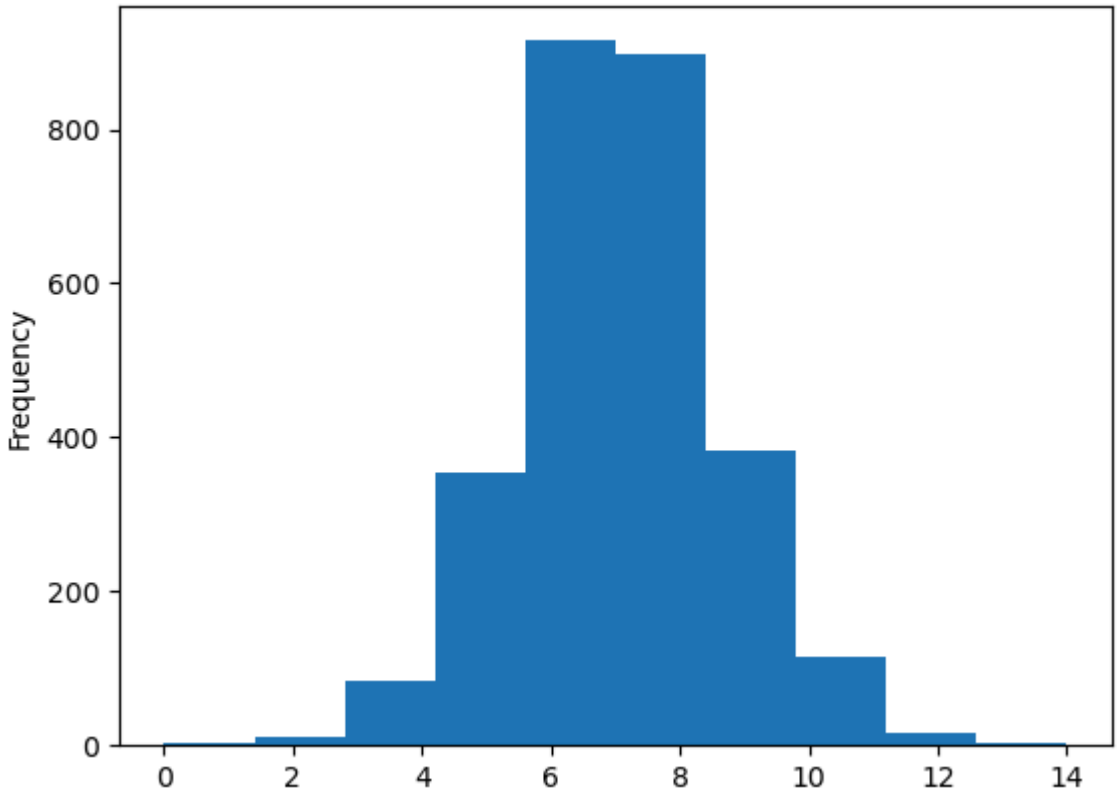
In [66]:
```
#Checking for missing value by heatmap by using seaborn

sns.heatmap(water_data.isnull(), cmap='viridis')
plt.show()
```



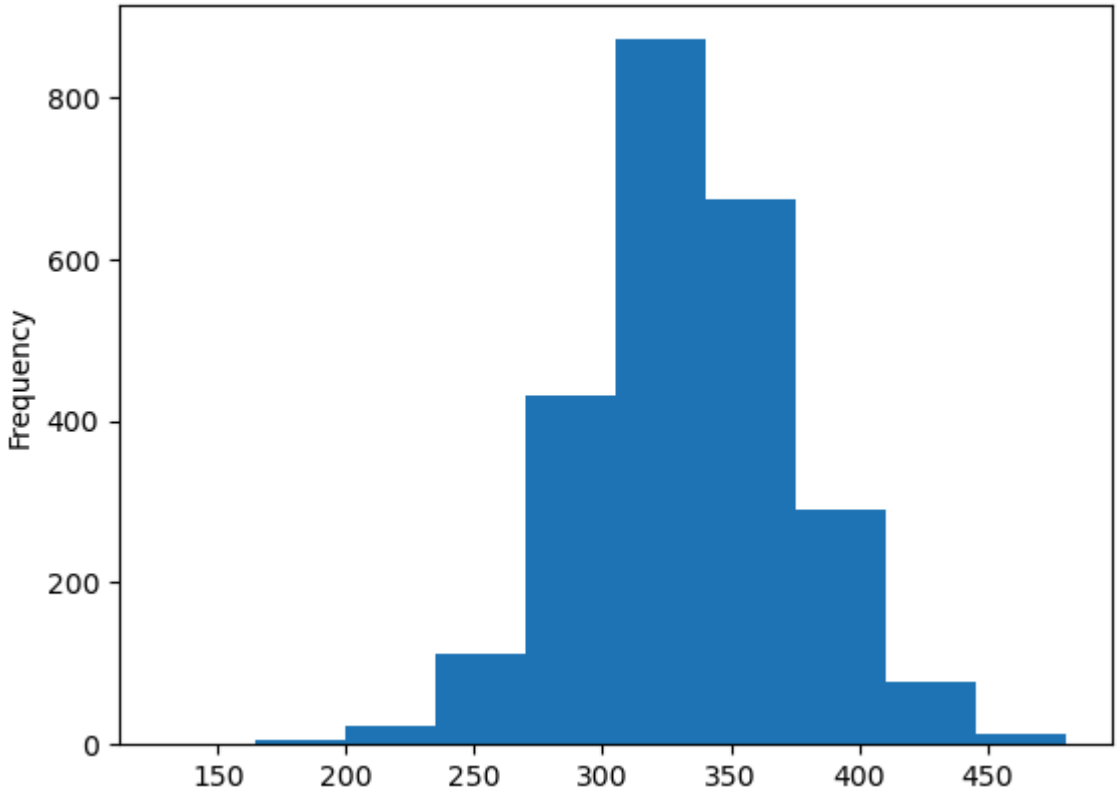
Handling Missing Values :

In [12]:
```
#To check if data is normally distributes

water_data['ph'].plot(kind='hist')
```

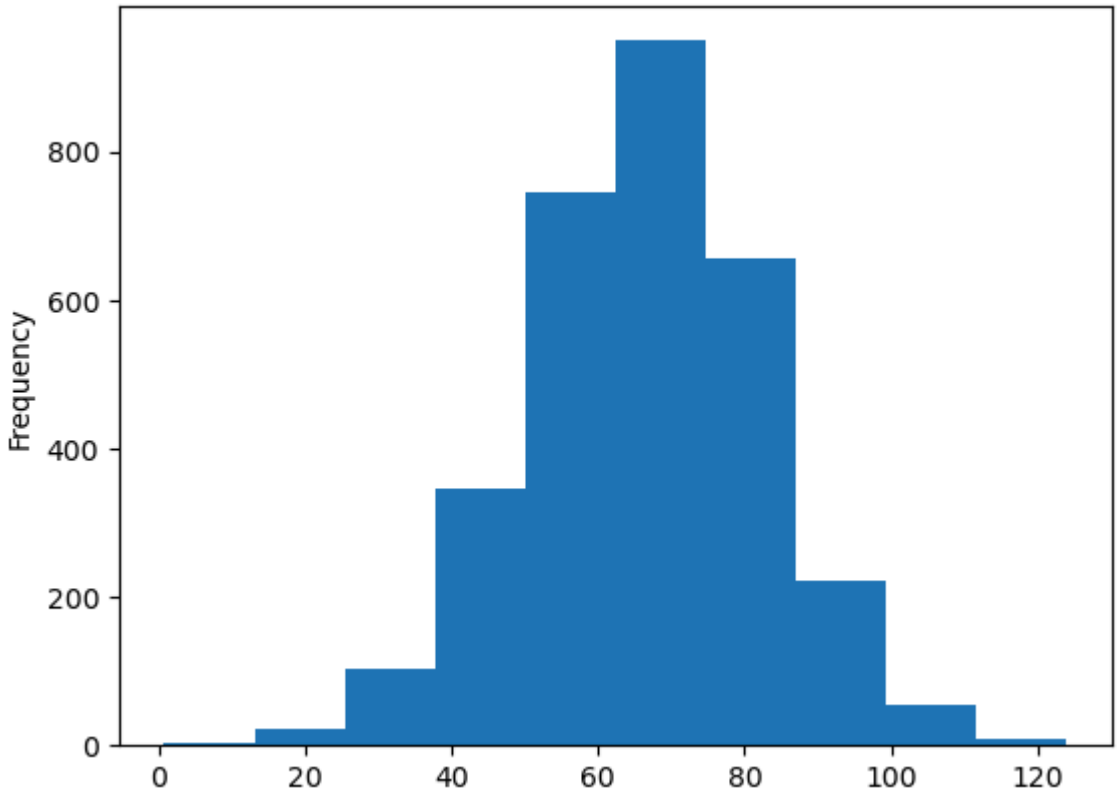
Out[12]:
```
<Axes: ylabel='Frequency'>
```



In [13]:
```
water_data['Sulfate'].plot(kind = 'hist')
plt.show()
```

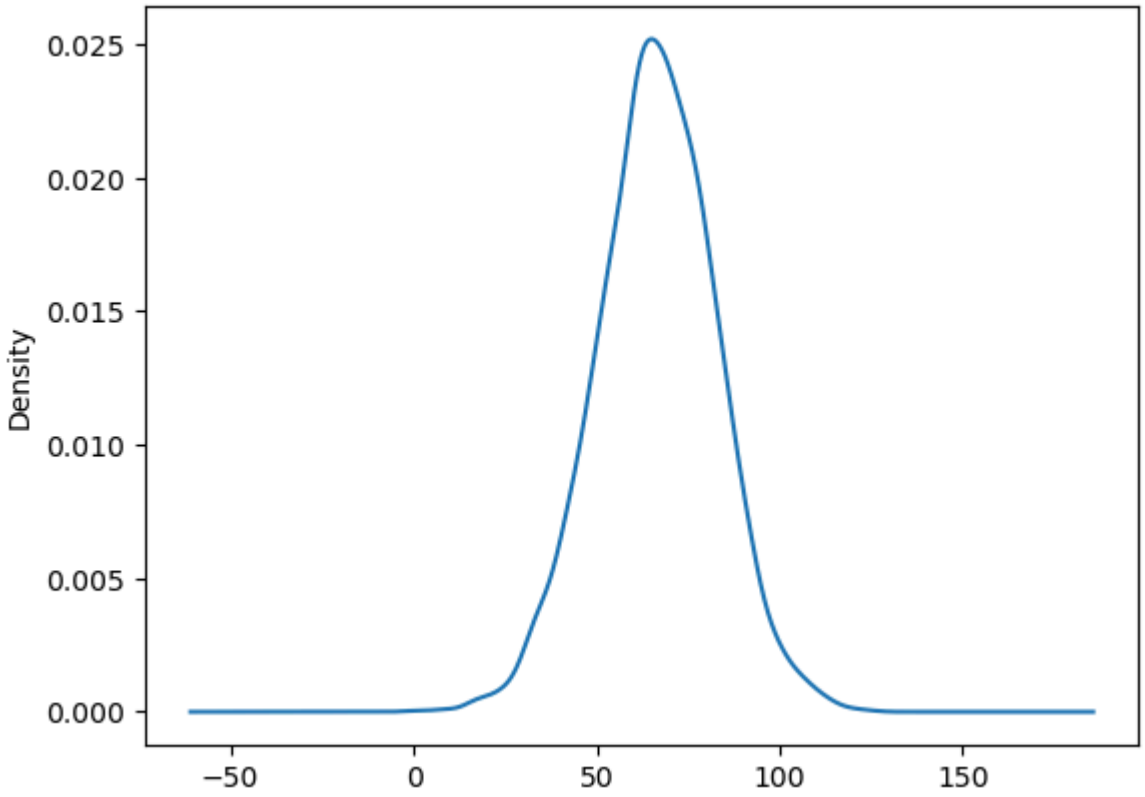


In [14]:
```
water_data['Trihalomethanes'].plot(kind = 'hist')
plt.show()
```



From the above cases, we can conclude that the data is similar to normaaly distributed.

In [15]:
```
# sns.histplot(data = water_data[ 'TrihaLomethanes '), kde = True, hue = water_data[ '
# Kdeplot for Trihalomethanes

fig = plt. figure()
ax = fig.add_subplot(111)
water_data['Trihalomethanes'].plot(kind = 'kde',ax=ax)
plt.show()
```



the below code ensures that missing values in the specified columns are replaced with the average value of each respective column. This approach helps to maintain the overall integrity of the dataset

In [16]:
```
# Filling mean value at missing values

water_data[ 'ph'] = water_data[ 'ph'].fillna(water_data[ 'ph'].mean())
water_data[ 'Trihalomethanes'] = water_data[ 'Trihalomethanes'].fillna(water_data['Tri
water_data['Sulfate'] = water_data[ 'Sulfate'].fillna(water_data[ 'Sulfate'].mean())
```

In [17]:
```
water_data.isnull().sum()
```

Out[17]:
```
ph                0
Hardness          0
Solids            0
Chloramines       0
Sulfate           0
Conductivity      0
Organic_carbon    0
Trihalomethanes   0
Turbidity         0
Potability        0
dtype: int64
```

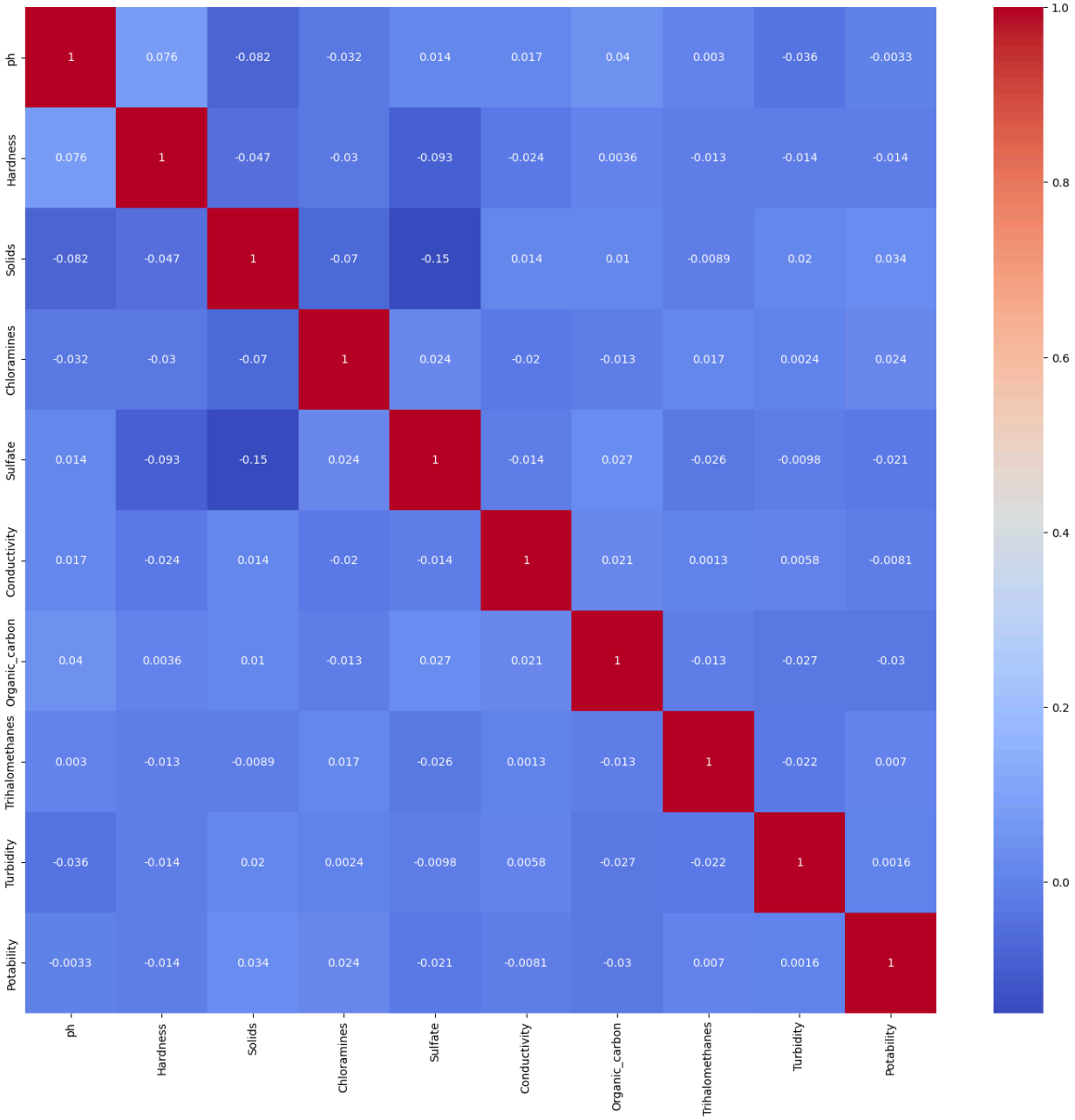
In [18]:
```
# Check for correlation
corr_matrix = water_data.corr()
corr_matrix
```

Out[18]:

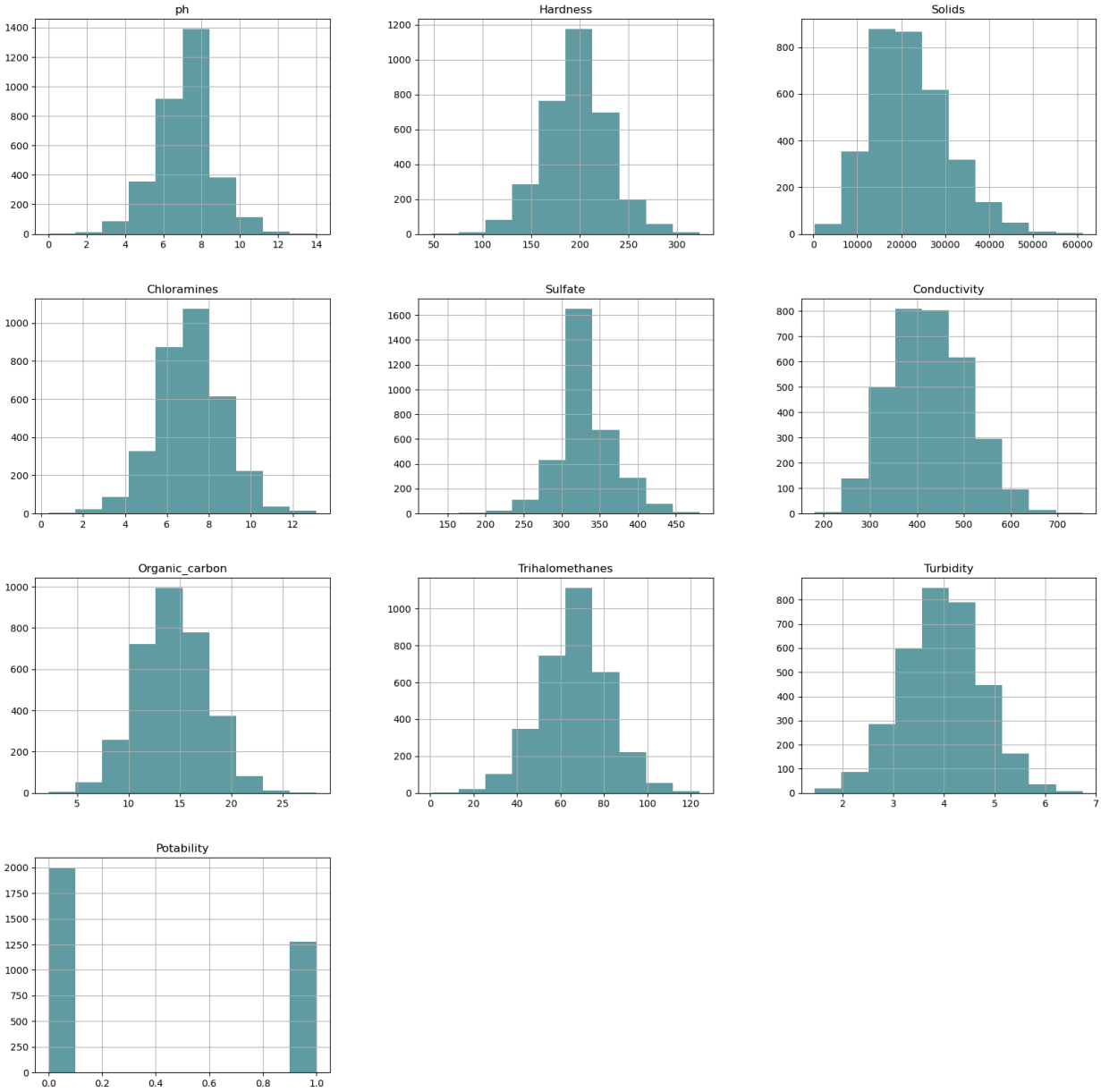
	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbo
ph	1.000000	0.075833	-0.081884	-0.031811	0.014403	0.017192	0.04006
Hardness	0.075833	1.000000	-0.046899	-0.030054	-0.092766	-0.023915	0.00361
Solids	-0.081884	-0.046899	1.000000	-0.070148	-0.149840	0.013831	0.01024
Chloramines	-0.031811	-0.030054	-0.070148	1.000000	0.023791	-0.020486	-0.01265
Sulfate	0.014403	-0.092766	-0.149840	0.023791	1.000000	-0.014059	0.02690
Conductivity	0.017192	-0.023915	0.013831	-0.020486	-0.014059	1.000000	0.02096
Organic_carbon	0.040061	0.003610	0.010242	-0.012653	0.026909	0.020966	1.00000
Trihalomethanes	0.002994	-0.012690	-0.008875	0.016627	-0.025605	0.001255	-0.01297
Turbidity	-0.036222	-0.014449	0.019546	0.002363	-0.009790	0.005798	-0.02730
Potability	-0.003287	-0.013837	0.033743	0.023779	-0.020619	-0.008128	-0.03000

In [35]:

```
# plotting Heatmap for visualizing correlattion

plt.figure(figsize=(18, 16))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm')
plt.show()
```



In [39]:
```
# Plotting Histogram of each and evry column

data_hist_plot = water_data.hist(figsize = (20,20), color = "#5F9EA0")
```



the code below computes the average values of each feature (such as pH, hardness, etc.) for potable and non-potable water separately and presents them in a transposed format. This data shows the average values of various water quality parameters for two groups based on potability: potable (labeled as 0) and non-potable (labeled as 1).

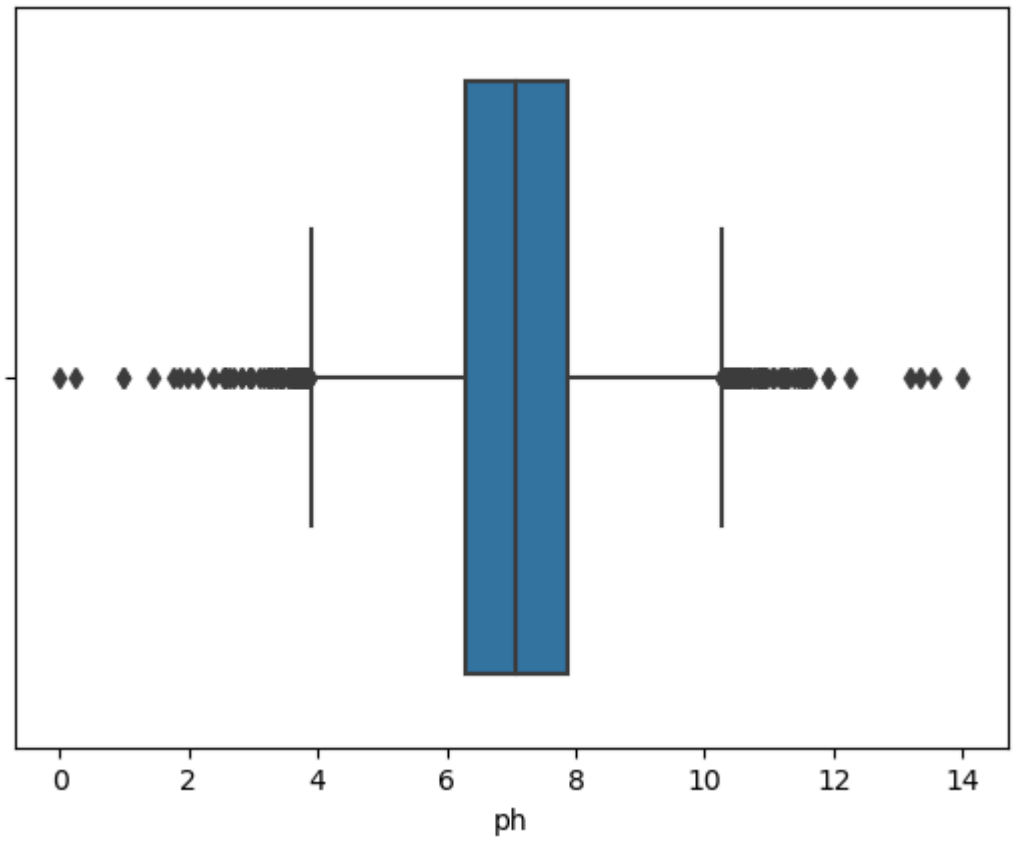
In [21]:
```
water_data.groupby('Potability').mean().T
```

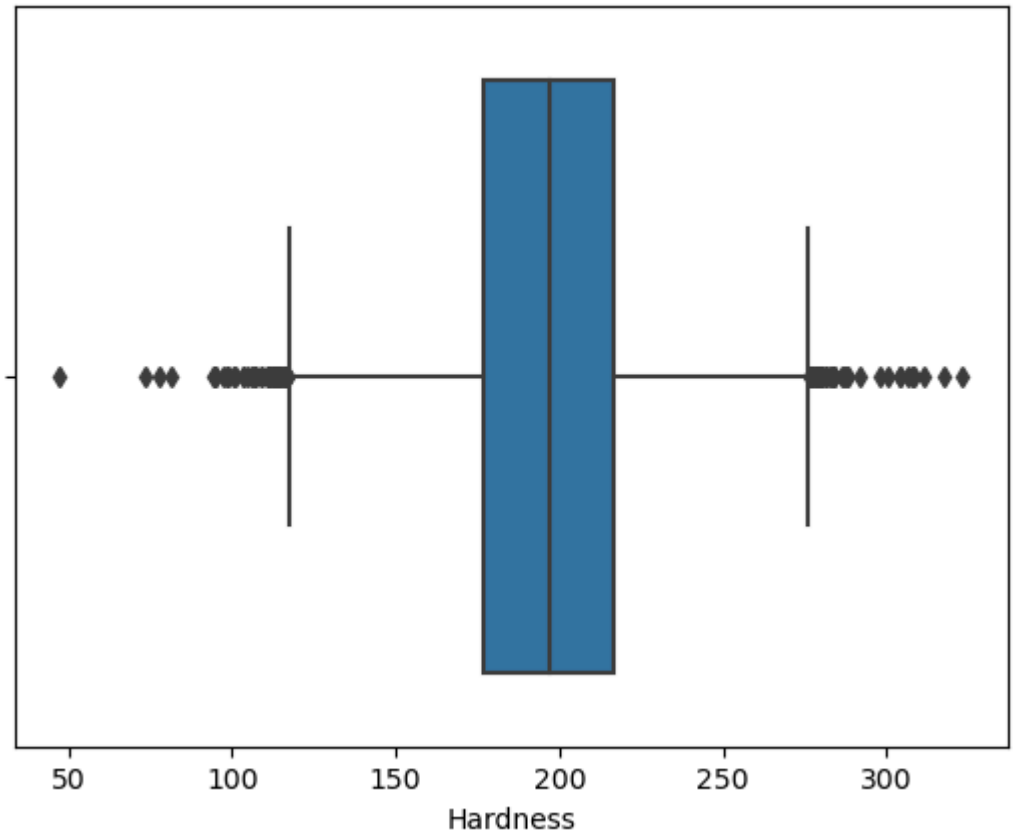
Out[21]:

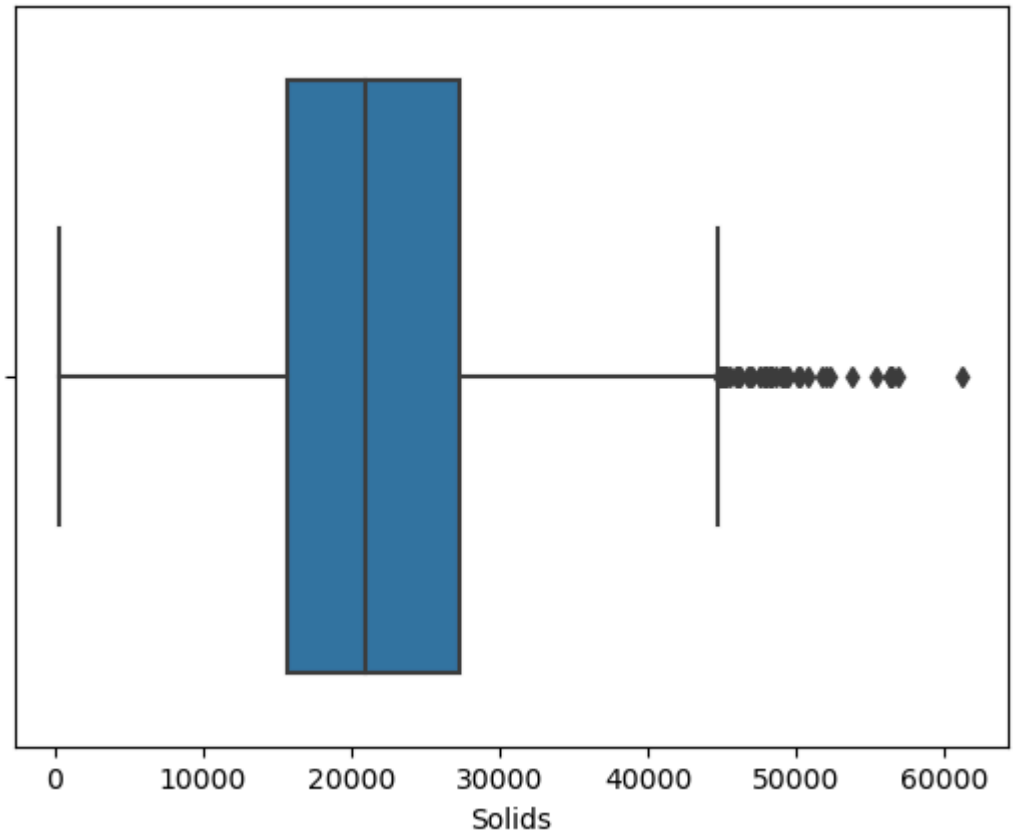
Potability	0	1
ph	7.084658	7.074754
Hardness	196.733292	195.800744
Solids	21777.490788	22383.991018
Chloramines	7.092175	7.169338
Sulfate	334.371700	332.844122
Conductivity	426.730454	425.383800
Organic_carbon	14.364335	14.160893
Trihalomethanes	66.308522	66.533513
Turbidity	3.965800	3.968328

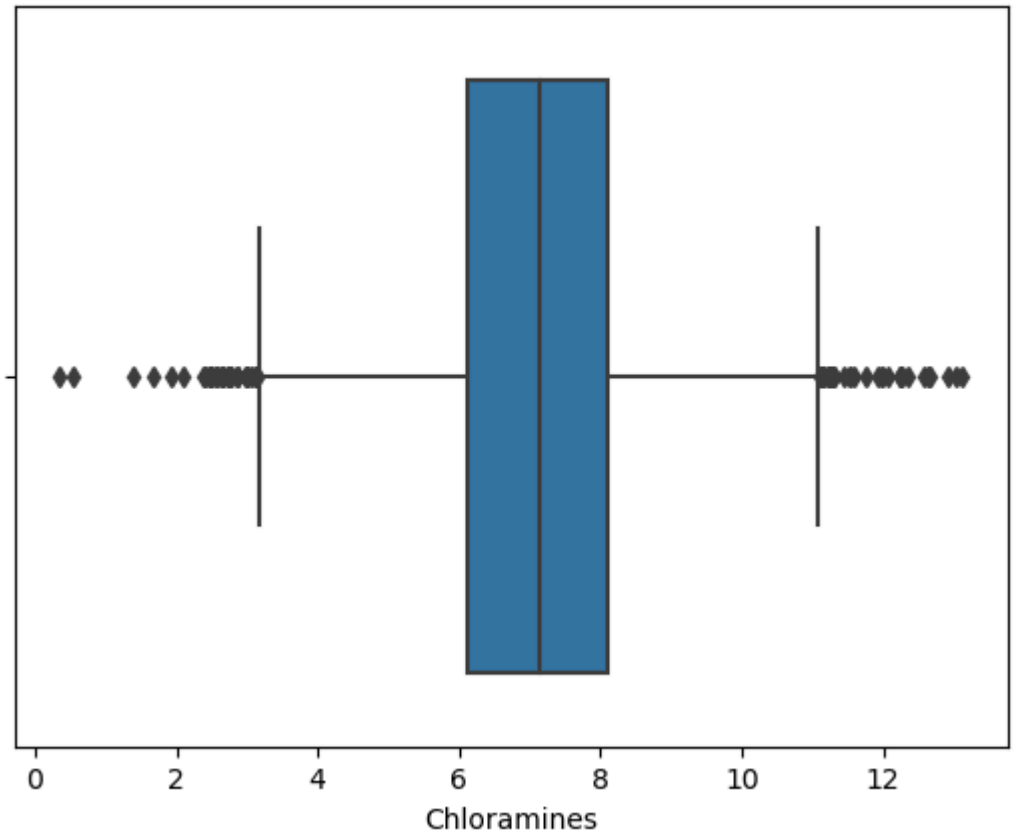
In [22]:

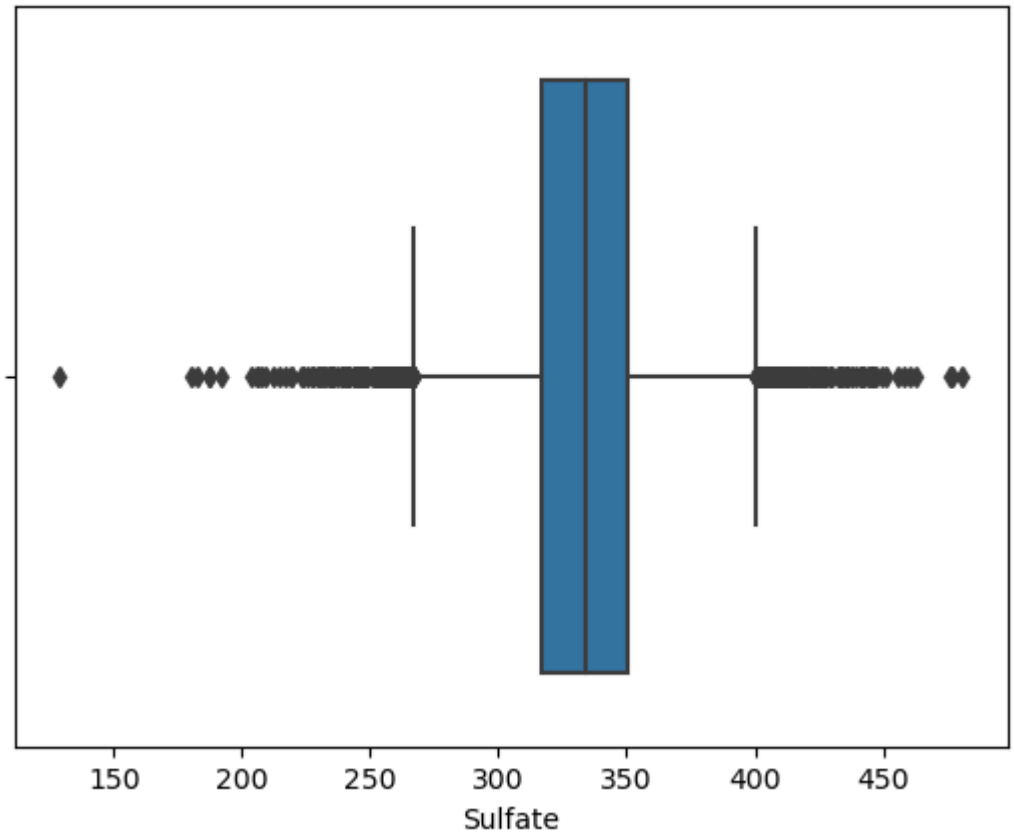
```
for col in water_data. columns:
    sns.boxplot(data=water_data, x=col)
    plt. show()
```

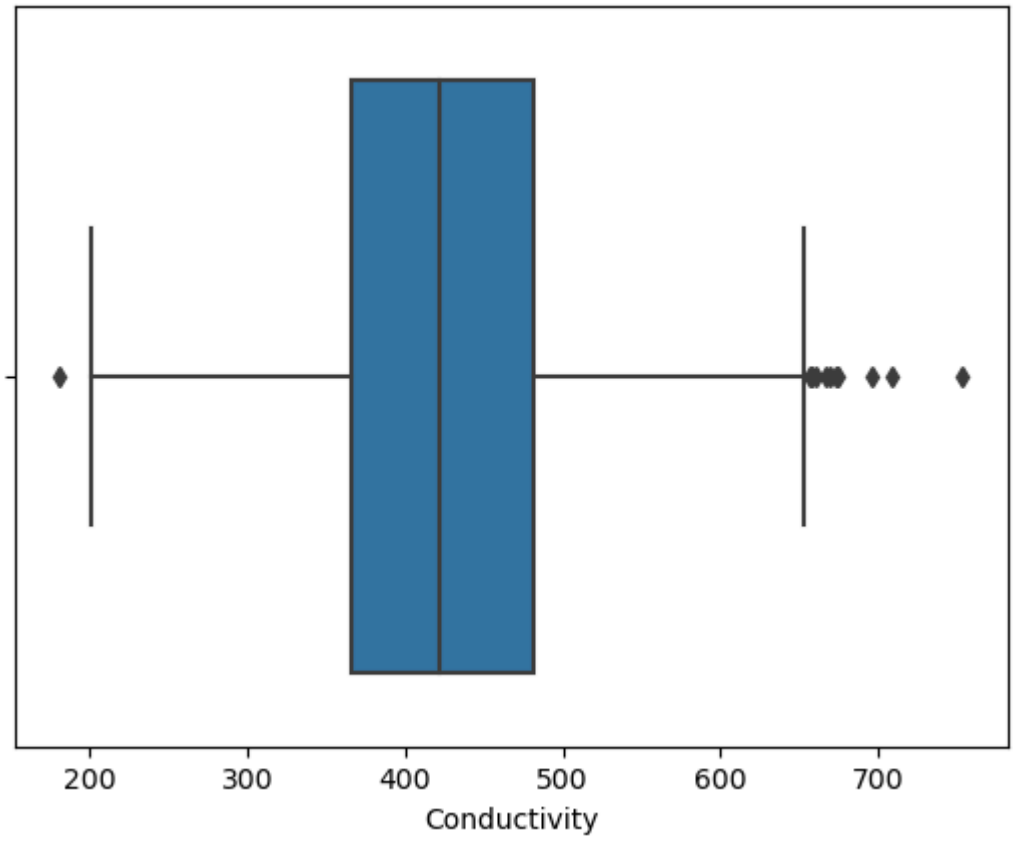


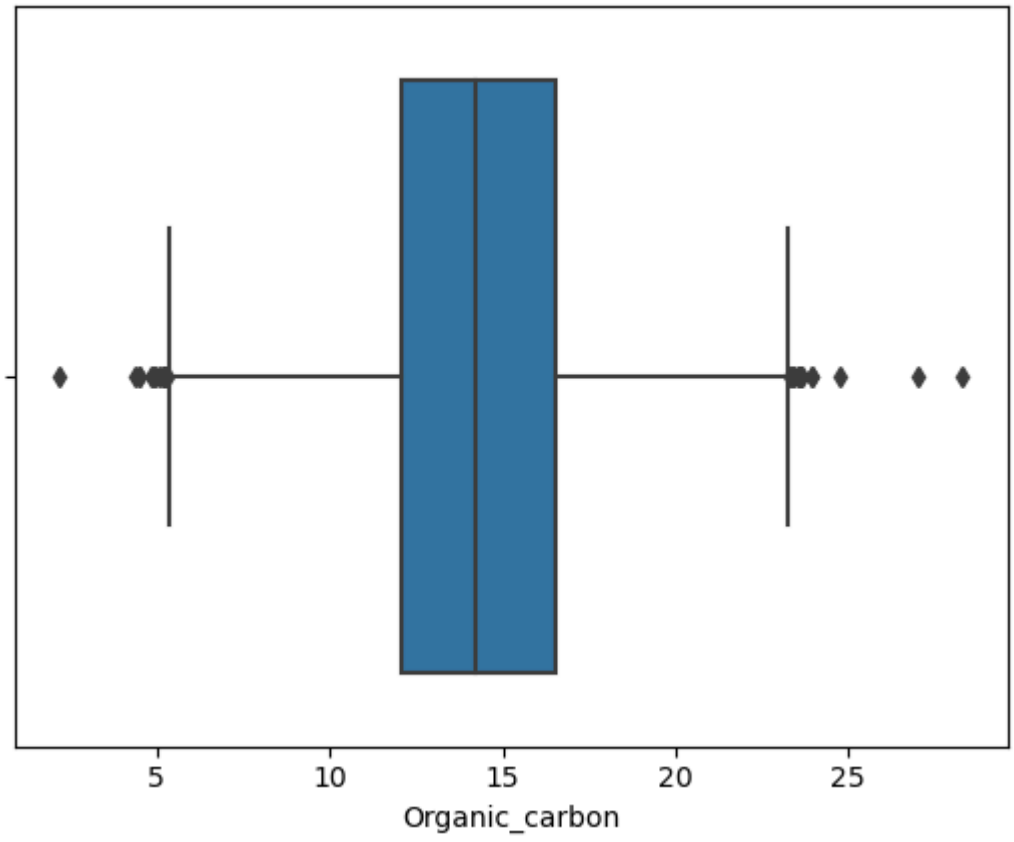

50
100
150
200
250
300
Hardness

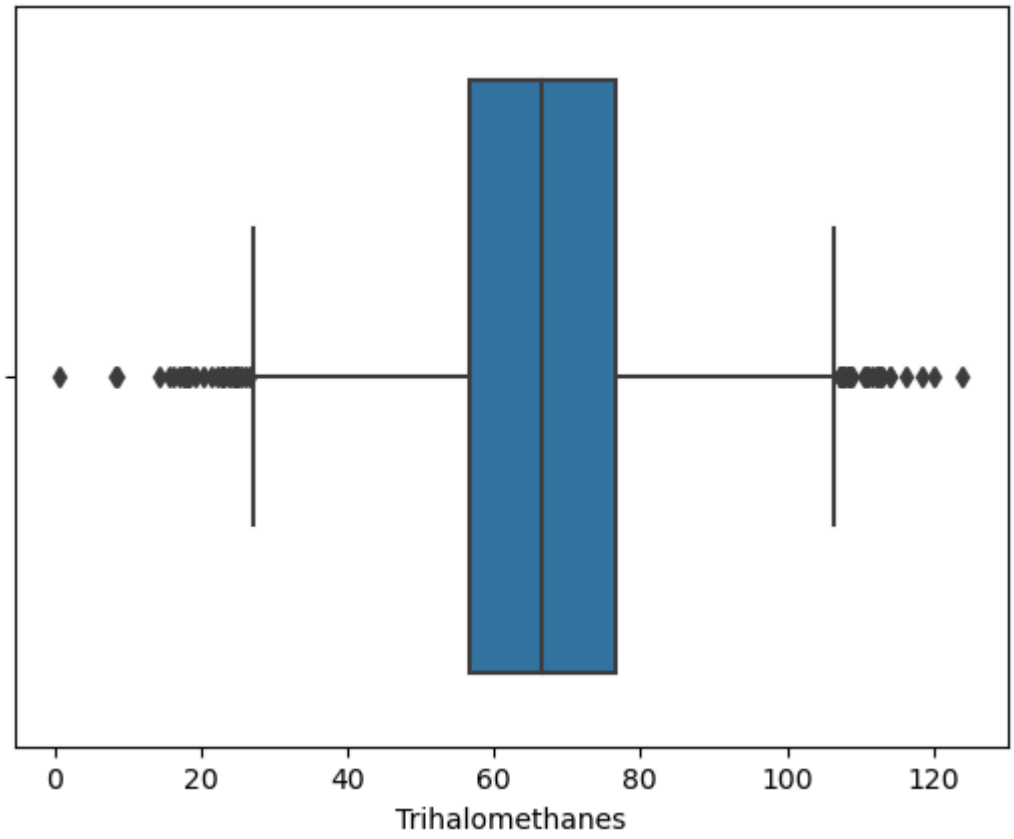

0
10000
20000
30000
40000
50000
60000
Solids

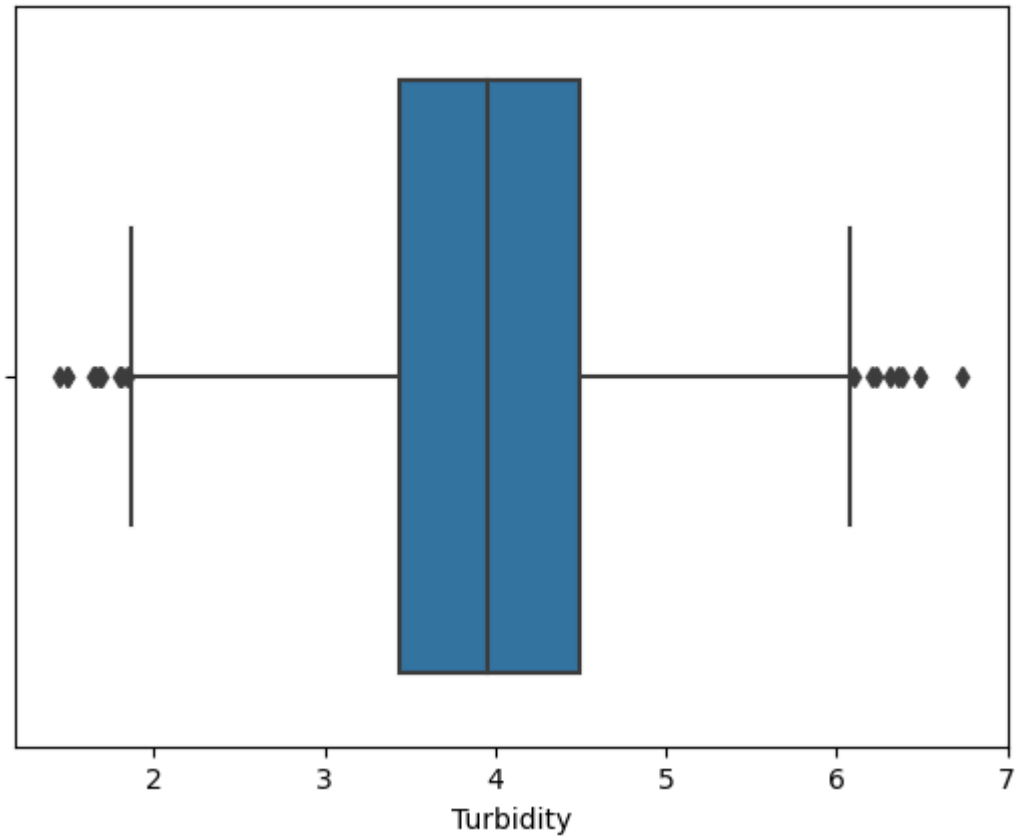

0
2
4
6
8
10
12
Chloramines

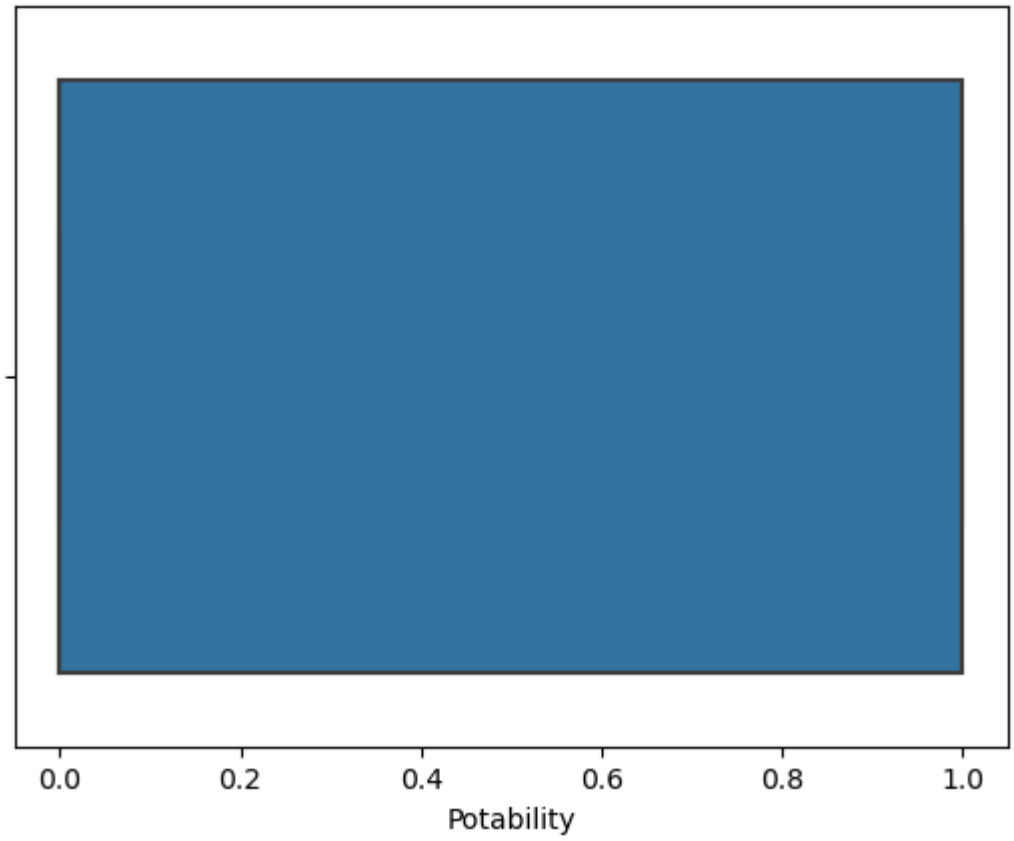

150
200
250
300
350
400
450
Sulfate


200
300
400
500
600
700
Conductivity


5
10
15
20
25
Organic_carbon


0
20
40
60
80
100
120
Trihalomethanes


2
3
4
5
6
7
Turbidity

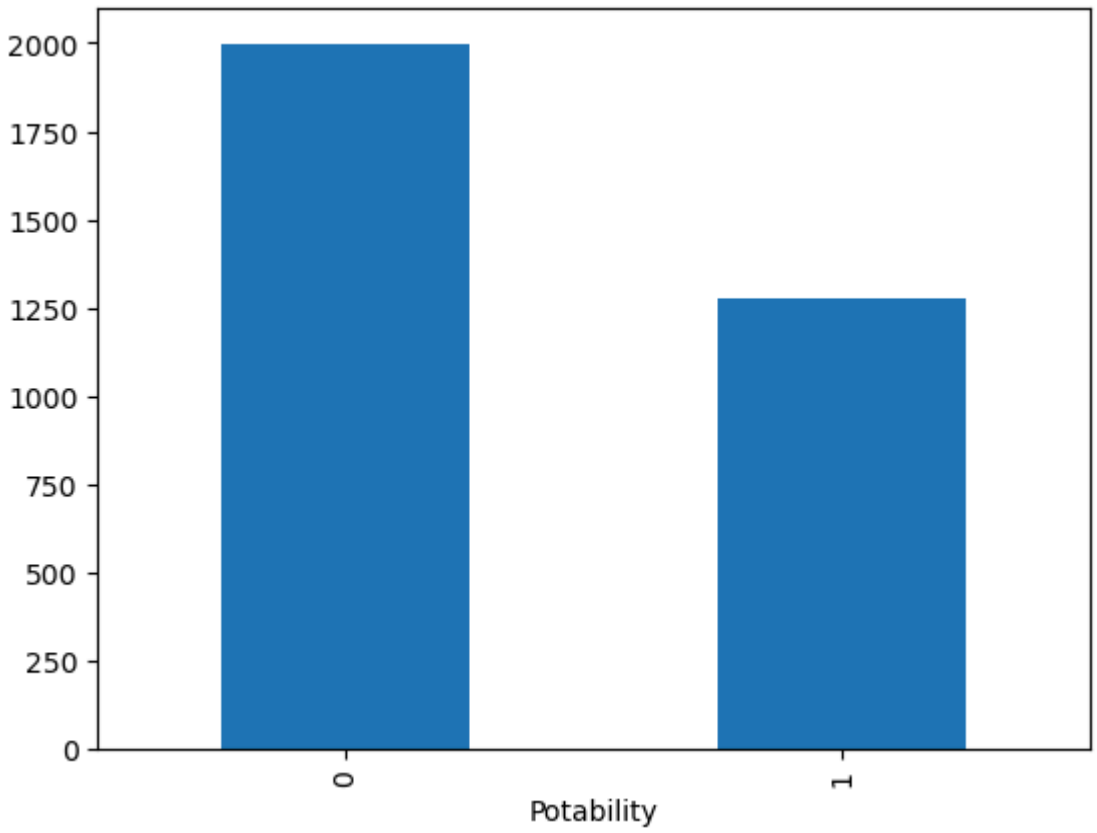


In [23]:
```
# counting the value for potable and non-potable records
water_data[ 'Potability'].value_counts()
```

Out[23]:
```
Potability
0    1998
1    1278
Name: count, dtype: int64
```

In [24]:
```
# sns.countplot(water_data[ 'Potability')
water_data['Potability'].value_counts().plot(kind = 'bar')
```

Out[24]:
```
<Axes: xlabel='Potability'>
```



Data Preprocessing:

In [25]:
```
X = water_data.drop('Potability', axis = 1)   #independent feature
y = water_data[ 'Potability']    #dependent feature
```

In [26]:
```
X.head()
```

Out[26]:

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalo
0	7.080795	204.890455	20791.318981	7.300212	368.516441	564.308654	10.379783	
1	3.716080	129.422921	18630.057858	6.635246	333.775777	592.885359	15.180013	
2	8.099124	224.236259	19909.541732	9.275884	333.775777	418.606213	16.868637	
3	8.316766	214.373394	22018.417441	8.059332	356.886136	363.266516	18.436524	1
4	9.092223	181.101509	17978.986339	6.546600	310.135738	398.410813	11.558279	

In [27]:
```
y.head()
```

Out[27]:
```
0    0
1    0
2    0
3    0
4    0
Name: Potability, dtype: int64
```

Feature Engeineering:

In [28]:
```
from sklearn.preprocessing import StandardScaler
std_scaler = StandardScaler()
```

In [29]:
```
X_scaled = std_scaler.fit_transform(X)
X_scaled
```

Out[29]:
```
array([[-6.04313345e-16,  2.59194711e-01, -1.39470871e-01, ...,
        -1.18065057e+00,  1.30614943e+00, -1.28629758e+00],
       [-2.28933938e+00, -2.03641367e+00, -3.85986650e-01, ...,
         2.70597240e-01, -6.38479983e-01,  6.84217891e-01],
       [ 6.92867789e-01,  8.47664833e-01, -2.40047337e-01, ...,
         7.81116857e-01,  1.50940884e-03, -1.16736546e+00],
       ...,
       [ 1.59125368e+00, -6.26829230e-01,  1.27080989e+00, ...,
        -9.81329234e-01,  2.18748247e-01, -8.56006782e-01],
       [-1.32951593e+00,  1.04135450e+00, -1.14405809e+00, ...,
        -9.42063817e-01,  7.03468419e-01,  9.50797383e-01],
       [ 5.40150905e-01, -3.85462310e-02, -5.25811937e-01, ...,
         5.60940070e-01,  7.80223466e-01, -2.12445866e+00]])
```

Training and Testing Data

In [30]:
```
# Spitting data into two sets for training and testing

from sklearn.model_selection import train_test_split
x_train, x_test, y_train, y_test = train_test_split(X, y, test_size= 0.2, random_state
```

In [31]:
```
x_train.shape, x_test.shape
```

Out[31]:
```
((2620, 9), (656, 9))
```

Model Development:

List of Models LogisticRegression DecisionTreeClassifier RandomForestClassifier

Importing Models:

In [32]:
```
from sklearn. linear_model import LogisticRegression
from sklearn. tree import DecisionTreeClassifier
from sklearn. ensemble import RandomForestClassifier
```

Model Training using Logistic Regression

In [33]:
```
# Creating the object of the model

LR = LogisticRegression()
```

cross-validation : We split our dataset into folds, train the model on some folds, and test it on others. This helps us get a more accurate estimate of how well our model will perform on new, unseen data.

In [34]:
```
from sklearn.model_selection import cross_val_score
from sklearn.metrics import classification_report
```

The code below efficiently evaluates the performance of different machine learning models using cross-validation and presents the results in a tabular format for comparison.

In [35]:
```
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import cross_val_score
import pandas as pd

models = [LogisticRegression(), DecisionTreeClassifier(), RandomForestClassifier()]
features = X_scaled
labels = y
CV = 5   #divided into 5 folds
accu_list = []
ModelName = [] # Model Name List

for model in models:
    model_name = model.__class__.__name__
    accuracies = cross_val_score(model, features, labels, scoring='accuracy', cv=CV)
    accu_list.append(accuracies.mean() * 100)
    ModelName.append(model_name)

model_acc_df = pd.DataFrame({"Model": ModelName, "Cross_Val_Accuracy": accu_list})
print(model_acc_df)
```

```
                   Model  Cross_Val_Accuracy
0     LogisticRegression           61.019549
1  DecisionTreeClassifier           57.418358
2  RandomForestClassifier           63.828337
```

In [36]:
```
model_acc_df = pd.DataFrame({"Model": ModelName, "cross_val_Accuracy": accu_list})
model_acc_df
```

Out[36]:

	Model	cross_val_Accuracy
0	LogisticRegression	61.019549
1	DecisionTreeClassifier	57.418358
2	RandomForestClassifier	63.828337

In [ ]:
```
From the above value of Cross-Validation accuracy, we can conclude that Random Forest
```

In [37]:
```
from sklearn. metrics import classification_report
```

In [38]:
```
# Logistic Regression
LR = LogisticRegression()
LR.fit(x_train, y_train)

# Decision Tree
DT = DecisionTreeClassifier()
DT.fit(x_train, y_train)

# Random Forest
```

```
RF = RandomForestClassifier()
RF.fit(x_train, y_train)

# Making predictions
y_pred_lr = LR.predict(x_test)
y_pred_dt = DT.predict(x_test)
y_pred_rf = RF.predict(x_test)
```

Precision: Precision is the ratio of true positive predictions to the total number of positive predictions. It measures the accuracy of positive predictions made by the model. Recall: Recall, also known as sensitivity or true positive rate, is the ratio of true positive predictions to the total number of actual positive instances in the data. It measures the model's ability to correctly identify positive instances. F1-score: The F1-score is the harmonic mean of precision and recall. It provides a single metric that balances both precision and recall.

In [39]:
```
print(classification_report(y_test, y_pred_rf))
```

```
              precision    recall  f1-score   support

           0       0.67      0.87      0.75       400
           1       0.61      0.32      0.42       256

    accuracy                           0.66       656
   macro avg       0.64      0.60      0.59       656
weighted avg       0.65      0.66      0.63       656
```

In [40]:
```
print(classification_report(y_test, y_pred_dt))
```

```
              precision    recall  f1-score   support

           0       0.65      0.67      0.66       400
           1       0.46      0.44      0.45       256

    accuracy                           0.58       656
   macro avg       0.56      0.56      0.56       656
weighted avg       0.58      0.58      0.58       656
```

In [41]:
```
print(classification_report(y_test, y_pred_lr))
```

```
              precision    recall  f1-score   support

           0       0.61      1.00      0.76       400
           1       0.00      0.00      0.00       256

    accuracy                           0.61       656
   macro avg       0.30      0.50      0.38       656
weighted avg       0.37      0.61      0.46       656
```

By the score of precision, we can conclude that Random Forest would be the best model. min_samples_split: The minimum number of samples required to split an internal node. min_samples_leaf: The minimum number of samples required to be at a leaf node. n_estimators: The number of trees in the forest. criterion: The function to measure the quality of a split. "gini" for Gini impurity and "entropy" for information gain. This code sets up the parameters for hyperparameter tuning for a Random Forest classifier. Hyperparameter tuning is the process of finding the best set of hyperparameters for a machine learning algorithm to perform optimally. GridSearchCV helps you to find the best combination of these parameters automatically by searching through a grid of possible values that you specify beforehand.

In [42]:
```
from sklearn.model_selection import GridSearchCV, RandomizedSearchCV
from sklearn.model_selection import StratifiedKFold
```

```
params_RF = {"min_samples_split":[2, 6],
             "min_samples_leaf": [11, 4],
              "n_estimators" :[100, 200, 300],
              "criterion":["gini", 'entropy']}
```

The code beow sets up a grid search for hyperparameter tuning of a Random Forest classifier using cross-validation and specifies various parameters for the grid search process.

In [43]:
```
cv_method = StratifiedKFold(n_splits=3)
GridSearchCV_RF = GridSearchCV(estimator = RandomForestClassifier(),
                               param_grid=params_RF,
                               cv=cv_method,
                               verbose=1,
                               n_jobs=2,
                               scoring= "accuracy",
                              return_train_score= True )
```

In [44]:
```
GridSearchCV_RF.fit(x_train, y_train)
best_params_RF = GridSearchCV_RF.best_params_
print ("Best Hyperparameters for Random Forest are = ", best_params_RF)
```

```
Fitting 3 folds for each of 24 candidates, totalling 72 fits
Best Hyperparameters for Random Forest are =  {'criterion': 'entropy', 'min_samples_l
eaf': 4, 'min_samples_split': 6, 'n_estimators': 300}
```

In [45]:
```
from sklearn.metrics import classification_report

best_estimator = GridSearchCV_RF.best_estimator_
best_estimator.fit(x_train, y_train)
y_pred_best = best_estimator.predict(x_test)
print(classification_report(y_test, y_pred_best))
```

```
              precision    recall  f1-score   support

           0       0.67      0.93      0.78       400
           1       0.71      0.29      0.41       256

    accuracy                           0.68       656
   macro avg       0.69      0.61      0.59       656
weighted avg       0.69      0.68      0.63       656
```

In [47]:
```
best_estimator = GridSearchCV_RF.best_estimator_
best_estimator
```

Out[47]:
```
RandomForestClassifier(criterion='entropy', min_samples_leaf=4,
                       min_samples_split=6, n_estimators=300)
```

In [46]:
```
from sklearn.metrics import accuracy_score
print(f"Accuracy of Random Forest Model = {round(accuracy_score(y_test, y_pred_best)*1
```

```
Accuracy of Random Forest Model = 67.68 %
```

Predictive System:

In [48]:
```
water_data.columns
```

Out[48]:
```
Index(['ph', 'Hardness', 'Solids', 'Chloramines', 'Sulfate', 'Conductivity',
       'Organic_carbon', 'Trihalomethanes', 'Turbidity', 'Potability'],
      dtype='object')
```

In [49]:
```
list1 = water_data.iloc[2:3, 0:9].values.flatten().tolist()
list1
```

Out[49]:
```
[8.099124189298397,
 224.23625939355776,
 19909.541732292397,
 9.275883602694089,
 333.7757766108135,
 418.6062130644815,
 16.868636929550973,
 66.42009251176368,
 3.0559337496641685]
```

In [50]:
```
ph = float(input("Enter the pH Value: "))
Hardness = float(input("Enter the Hardness Value: "))
Solids = float(input("Enter the Solids Value: "))
Chloramines = float(input("Enter the Chloramines Value: "))
Sulfate = float(input("Enter the Sulfate Value: "))
Conductivity = float(input("Enter the Conductivity Value: "))
Organic_carbon = float(input("Enter the Organic Carbon Value: "))
Trihalomethanes = float(input("Enter the Trihalomethanes Value: "))
Turbidity = float(input("Enter the Turbidity Value: "))
```

```
Enter the pH Value: 8.1
Enter the Hardness Value: 224.5
Enter the Solids Value: 19000
Enter the Chloramines Value: 9.5
Enter the Sulfate Value: 343
Enter the Conductivity Value: 420
Enter the Organic Carbon Value: 17
Enter the Trihalomethanes Value: 66.5
Enter the Turbidity Value: 3.5
```

In [51]:
```
input_data = [ph, Hardness, Solids, Chloramines, Sulfate, Conductivity, Organic_carbor
```

In [52]:
```
water_data_input = std_scaler.transform([[ph, Hardness, Solids, Chloramines, Sulfate,
                                          Organic_carbon, Trihalomethanes, Turbidity]]
water_data_input
```

Out[52]:
```
array([[ 0.69346369,  0.85568742, -0.34379065,  1.50218488,  0.25525636,
        -0.07678478,  0.8208317 ,  0.00657728, -0.59824187]])
```

In [53]:
```
model_prediction = best_estimator.predict(water_data_input)
model_prediction
```

Out[53]:
```
array([1], dtype=int64)
```

In [54]:
```
if model_prediction[0] == 0:
    print("water is Not SAFE for Consumption")
else:
    print("water is SAFE for Consumption")
```

```
water is SAFE for Consumption
```

In [55]:
```
def water_quality_prediction(input_data):
    scaled_data = std_scaler.transform([input_data])  # Assuming std_scaler is defined
    model_prediction = best_estimator.predict(scaled_data)
    if model_prediction[0] == 0:
        return "Water is 'NOT SAFE' for Consumption"
    else:
        return "Water is 'SAFE' for Consumption"
```

In [56]:
```
ph = float(input("Enter the pH Value: "))
Hardness = float(input("Enter the Hardness Value: "))
Solids = float(input("Enter the Solids Value: "))
Chloramines = float(input("Enter the Chloramines Value: "))
Sulfate = float(input("Enter the Sulfate Value: "))
Conductivity = float(input("Enter the Conductivity Value: "))
Organic_carbon = float(input("Enter the Organic Carbon Value: "))
Trihalomethanes = float(input("Enter the Trihalomethanes Value: "))
Turbidity = float(input("Enter the Turbidity Value: "))

input_data = [ph, Hardness, Solids, Chloramines, Sulfate, Conductivity, Organic_carbon

water_quality_prediction(input_data)
```

```
Enter the pH Value: 7.65
Enter the Hardness Value: 240
Enter the Solids Value: 14245
Enter the Chloramines Value: 6.28
Enter the Sulfate Value: 373
Enter the Conductivity Value: 416
Enter the Organic Carbon Value: 10.46
Enter the Trihalomethanes Value: 85.85
Enter the Turbidity Value: 2.43
```

Out[56]:
```
"Water is 'SAFE' for Consumption"
```

In [70]:
```
ph = float(input("Enter the pH Value: "))
Hardness = float(input("Enter the Hardness Value: "))
Solids = float(input("Enter the Solids Value: "))
Chloramines = float(input("Enter the Chloramines Value: "))
Sulfate = float(input("Enter the Sulfate Value: "))
Conductivity = float(input("Enter the Conductivity Value: "))
Organic_carbon = float(input("Enter the Organic Carbon Value: "))
Trihalomethanes = float(input("Enter the Trihalomethanes Value: "))
Turbidity = float(input("Enter the Turbidity Value: "))

input_data = [ph, Hardness, Solids, Chloramines, Sulfate, Conductivity, Organic_carbon

water_quality_prediction(input_data)
```

```
Enter the pH Value: 3.7
Enter the Hardness Value: 129
Enter the Solids Value: 18630
Enter the Chloramines Value: 6.63
Enter the Sulfate Value: 368.5
Enter the Conductivity Value: 592.88
Enter the Organic Carbon Value: 15
Enter the Trihalomethanes Value: 56
Enter the Turbidity Value: 4.5
```

Out[70]:
```
"Water is 'SAFE' for Consumption"
```

The model can help in early detection of contaminated water sources by predicting whether water is potable or not based on chemical and physical parameters. This early detection can prevent the consumption of unsafe water, reducing the risk of waterborne diseases and contribute to sustainable development goals related to clean water and sanitation.