## **WATER QUALITY ANALYSIS**

**TEAM MEMBER**

**621721243043 : SASMITHA D S**

**Phase 3 Submission Project**

## **Introduction:**

* Access to safe drinking-water is essential to health, a basic human right and a component of effective policy for health protection. This is important as a health and development issue at a national, regional and local level. In some regions, it has been shown that investments in water supply and sanitation can yield a net economic benefit, since the reductions in adverse health effects and health care costs outweigh the costs of undertaking the interventions.
* Drinking water and staying hydrated is associated with a reduced incidence of urinary tract infections (UTIs), lower blood pressure and heart disease. Therefore, drinking water is essential for good heart health.
* Water is the most important nutrient for the body. It has many benefits for your health and helps to protect you from illness and disease. Water is also an essential part of a healthy lifestyle.



**Content of the Project:**

Building a water quality analysis involves several steps, starting with preprocessing the data and performing exploratory data analysis (EDA). Below, I'll outline a step-by-step guide to help you get started:

**Step 1: Data Collection**

- The first step in any data analysis project is to collect the relevant data. You may obtain water quality data from sources such as government agencies, environmental organizations, or research institutions. Ensure that the data includes parameters like pH, turbidity, dissolved oxygen, temperature, and various chemical concentrations.

**Step 2: Data Preprocessing**

- Data preprocessing is essential to clean and prepare the data for analysis. Some common tasks include:

- Handling missing values: Identify and handle missing data points, either by imputation or removal.

- Data encoding: Convert categorical data into a numerical format if necessary.

- Outlier detection and treatment: Identify and deal with outliers that may skew the analysis.

- Data scaling/normalization: Scale numerical features to ensure they have a similar range.

- Date and time formatting: If your data includes timestamps, format them appropriately.

**Step 3: Exploratory Data Analysis (EDA)**

EDA is a crucial phase in understanding your data. It helps you uncover patterns, relationships, and potential insights. Here are some EDA techniques:

- Descriptive Statistics:

- Calculate basic statistics like mean, median, standard deviation, and percentiles for each parameter.

- Data Visualization:

- Create various types of plots to visualize your data. Common plots include histograms, box plots, scatter plots, and time series plots.

- Use tools like Matplotlib, Seaborn, or Plotly in Python for creating visualizations.

- Correlation Analysis:

- Explore the correlation between different water quality parameters using techniques like correlation matrices and scatter plots.

- Identify which parameters have strong relationships and might influence each other.

- Time Series Analysis:

- If your data has a time component, analyze trends and seasonality in water quality parameters.

- Use time series decomposition techniques to separate trends, seasonality, and noise.

- Geospatial Analysis:

- If your data includes geospatial information, create maps to visualize how water quality varies across different locations.

**Step 4: Hypothesis Testing**

- If you have specific questions or hypotheses about your data, conduct hypothesis tests to confirm or reject these hypotheses. Common tests include t-tests and ANOVA for comparing groups, and regression analysis to model relationships.

**Step 5: Feature Selection**

- Based on your EDA and hypothesis testing, identify which features (water quality parameters) are most relevant to your analysis. You can use feature importance techniques like mutual information or recursive feature elimination.

**Step 6: Data Splitting**

- Split your data into training and testing sets if you plan to build predictive models. This ensures that you can evaluate the model's performance.

**Step 7: Model Building (if applicable)**

- If your goal is to predict water quality or make future projections, build appropriate machine learning models. This could include regression, time series forecasting, or classification models, depending on your objectives.

**Step 8: Evaluation and Validation**

- Evaluate the model's performance using appropriate metrics. For regression, you can use metrics like mean squared error, and for classification, use accuracy, precision, recall, or F1-score.

**Step 9: Interpretation and Reporting**

- Finally, interpret your findings and report your analysis, including any insights, patterns, or recommendations based on the data.

Remember that the specific techniques and tools you use can vary depending on the nature of your water quality data and your research goals. The above steps provide a general framework for getting started with a water quality analysis.

**What is Potabible water**

At its most basic level, potabible water relates to the safety of water.

Many questions begin to emerge.

* Are we able to consume all fresh water types?
* What percentage of the worlds fresh water can be accessed?
* Has the water table increased as sea levels have rised?

In [1]:

import numpy as np

import pandas as pd

import seaborn as sns

import matplotlib.pyplot as plt

import os

import sys

for dirname, \_, filenames **in** os.walk('/kaggle/input'):

for filename **in** filenames:

print(os.path.join(dirname, filename))

/kaggle/input/water-potability/water\_potability.csv

In [2]:

print(sys.version)

3.7.10 | packaged by conda-forge | (default, Feb 19 2021, 16:07:37)

[GCC 9.3.0]

EDA

In [3]:

df = pd.read\_csv("../input/water-potability/water\_potability.csv")

In [4]:

df.head()

Out[4]:

|  | ph | Hardness | Solids | Chloramines | Sulfate | Conductivity | Organic\_carbon | Trihalomethanes | Turbidity | Potability |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | NaN | 204.890455 | 20791.318981 | 7.300212 | 368.516441 | 564.308654 | 10.379783 | 86.990970 | 2.963135 | 0 |
| 1 | 3.716080 | 129.422921 | 18630.057858 | 6.635246 | NaN | 592.885359 | 15.180013 | 56.329076 | 4.500656 | 0 |
| 2 | 8.099124 | 224.236259 | 19909.541732 | 9.275884 | NaN | 418.606213 | 16.868637 | 66.420093 | 3.055934 | 0 |
| 3 | 8.316766 | 214.373394 | 22018.417441 | 8.059332 | 356.886136 | 363.266516 | 18.436524 | 100.341674 | 4.628771 | 0 |
| 4 | 9.092223 | 181.101509 | 17978.986339 | 6.546600 | 310.135738 | 398.410813 | 11.558279 | 31.997993 | 4.075075 | 0 |

In [5]:

df.info(memory\_usage="deep")

<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 [6]:

print(df.shape)

print(len(df))

print(f'Number of rows: **{**df.shape[0]**}** **\n**Number of columns: **{**df.shape[1]**}**')

(3276, 10)

3276

Number of rows: 3276

Number of columns: 10

Out[7]:

|  | ph | Hardness | Solids | Chloramines | Sulfate | Conductivity | Organic\_carbon | Trihalomethanes | Turbidity | Potability |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| count | 2785.000000 | 3276.000000 | 3276.000000 | 3276.000000 | 2495.000000 | 3276.000000 | 3276.000000 | 3114.000000 | 3276.000000 | 3276.000000 |
| mean | 7.080795 | 196.369496 | 22014.092526 | 7.122277 | 333.775777 | 426.205111 | 14.284970 | 66.396293 | 3.966786 | 0.390110 |
| std | 1.594320 | 32.879761 | 8768.570828 | 1.583085 | 41.416840 | 80.824064 | 3.308162 | 16.175008 | 0.780382 | 0.487849 |
| min | 0.000000 | 47.432000 | 320.942611 | 0.352000 | 129.000000 | 181.483754 | 2.200000 | 0.738000 | 1.450000 | 0.000000 |
| 25% | 6.093092 | 176.850538 | 15666.690297 | 6.127421 | 307.699498 | 365.734414 | 12.065801 | 55.844536 | 3.439711 | 0.000000 |
| 50% | 7.036752 | 196.967627 | 20927.833607 | 7.130299 | 333.073546 | 421.884968 | 14.218338 | 66.622485 | 3.955028 | 0.000000 |
| 75% | 8.062066 | 216.667456 | 27332.762127 | 8.114887 | 359.950170 | 481.792304 | 16.557652 | 77.337473 | 4.500320 | 1.000000 |
| max | 14.000000 | 323.124000 | 61227.196008 | 13.127000 | 481.030642 | 753.342620 | 28.300000 | 124.000000 | 6.739000 | 1.000000 |

In [8]:

df.describe().T

Out[8]:

|  | count | mean | std | min | 25% | 50% | 75% | max |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ph | 2785.0 | 7.080795 | 1.594320 | 0.000000 | 6.093092 | 7.036752 | 8.062066 | 14.000000 |
| Hardness | 3276.0 | 196.369496 | 32.879761 | 47.432000 | 176.850538 | 196.967627 | 216.667456 | 323.124000 |
| Solids | 3276.0 | 22014.092526 | 8768.570828 | 320.942611 | 15666.690297 | 20927.833607 | 27332.762127 | 61227.196008 |
| Chloramines | 3276.0 | 7.122277 | 1.583085 | 0.352000 | 6.127421 | 7.130299 | 8.114887 | 13.127000 |
| Sulfate | 2495.0 | 333.775777 | 41.416840 | 129.000000 | 307.699498 | 333.073546 | 359.950170 | 481.030642 |
| Conductivity | 3276.0 | 426.205111 | 80.824064 | 181.483754 | 365.734414 | 421.884968 | 481.792304 | 753.342620 |
| Organic\_carbon | 3276.0 | 14.284970 | 3.308162 | 2.200000 | 12.065801 | 14.218338 | 16.557652 | 28.300000 |
| Trihalomethanes | 3114.0 | 66.396293 | 16.175008 | 0.738000 | 55.844536 | 66.622485 | 77.337473 | 124.000000 |
| Turbidity | 3276.0 | 3.966786 | 0.780382 | 1.450000 | 3.439711 | 3.955028 | 4.500320 | 6.739000 |
| Potability | 3276.0 | 0.390110 | 0.487849 | 0.000000 | 0.000000 | 0.000000 | 1.000000 | 1.000000 |

In [9]:

df.sample(5)

Out[9]:

|  | ph | Hardness | Solids | Chloramines | Sulfate | Conductivity | Organic\_carbon | Trihalomethanes | Turbidity | Potability |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1715 | NaN | 193.577157 | 19671.351229 | 7.568897 | 342.673409 | 322.735685 | 18.511586 | 77.710307 | 3.481517 | 0 |
| 57 | 4.999414 | 190.287050 | 24323.865903 | 7.230164 | 324.893038 | 405.330482 | 8.236558 | 99.427385 | 4.460684 | 0 |
| 2296 | 6.369974 | 128.616868 | 13689.592073 | 9.066830 | NaN | 319.785076 | 8.001415 | 36.007524 | 3.427066 | 0 |
| 457 | 7.240351 | 201.997196 | 14462.674308 | 6.737176 | 314.043137 | 534.800988 | 14.213794 | 82.945817 | 3.782972 | 0 |
| 1363 | 5.117914 | 179.701677 | 26215.401103 | 5.502903 | 303.257592 | 369.618616 | 14.761878 | 82.831554 | 3.806700 | 0 |

In [10]:

df.describe?

1b. Missing values

In [11]:

df.isnull().sum()

Out[11]:

ph 491

Hardness 0

Solids 0

Chloramines 0

Sulfate 781

Conductivity 0

Organic\_carbon 0

Trihalomethanes 162

Turbidity 0

Potability 0

dtype: int64

In [12]:

def isnull\_prop(df):

total\_rows = df.shape[0]

missing\_val\_dict = {}

for col **in** df.columns:

missing\_val\_dict[col] = [df[col].isnull().sum(), (df[col].isnull().sum() / total\_rows)]

return missing\_val\_dict

null\_dict = isnull\_prop(df)

print(null\_dict.items())

dict\_items([('ph', [491, 0.14987789987789987]), ('Hardness', [0, 0.0]), ('Solids', [0, 0.0]), ('Chloramines', [0, 0.0]), ('Sulfate', [781, 0.23840048840048841]), ('Conductivity', [0, 0.0]), ('Organic\_carbon', [0, 0.0]), ('Trihalomethanes', [162, 0.04945054945054945]), ('Turbidity', [0, 0.0]), ('Potability', [0, 0.0])])

In [13]:

df\_missing = pd.DataFrame.from\_dict(null\_dict, orient="index", columns=['missing', 'miss\_percent'])

df\_missing

Out[13]:

|  | missing | miss\_percent |
| --- | --- | --- |
| ph | 491 | 0.149878 |
| Hardness | 0 | 0.000000 |
| Solids | 0 | 0.000000 |
| Chloramines | 0 | 0.000000 |
| Sulfate | 781 | 0.238400 |
| Conductivity | 0 | 0.000000 |
| Organic\_carbon | 0 | 0.000000 |
| Trihalomethanes | 162 | 0.049451 |
| Turbidity | 0 | 0.000000 |
| Potability | 0 | 0.000000 |

In [14]:

plt.figure(figsize=(15,8))

sns.heatmap(df.isnull());



In [15]:

sns.displot(df["ph"], kde=False)

plt.axvline(x=df.ph.mean(), linewidth=3, color='g', label="mean", alpha=0.5)

plt.axvline(x=df.ph.median(), linewidth=3, color='y', label="median", alpha=0.5)

plt.xlabel("ph")

plt.ylabel("Count")

plt.title("Distribution of ph", size=14)

plt.legend(["mean", "median"]);

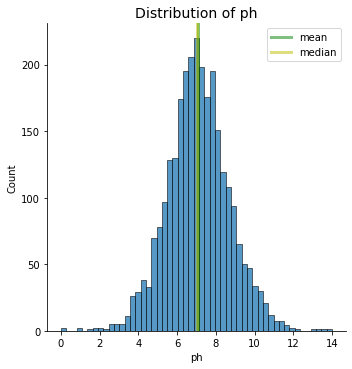
print(f'Mean pH value **{**df.ph.mean()**}** **\n** Median pH value **{**df.ph.median()**}** **\n** Min pH value **{**df.ph.min()**}** **\n** Max pH value **{**df.ph.max()**}**')

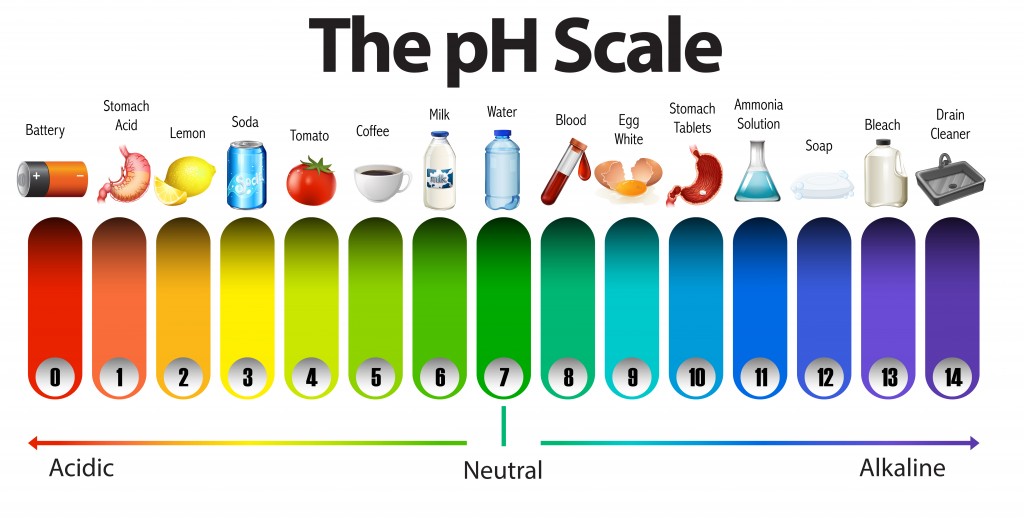
Mean pH value 7.080794504276819

Median pH value 7.036752103833548

Min pH value 0.0

Max pH value 13.999999999999998



Do these values of pH relate to actual water or are there a wider range of sources being supplied?Predict Potability

In [16]:

from sklearn.preprocessing import scale

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import StandardScaler

from sklearn.pipeline import Pipeline

from sklearn.model\_selection import RepeatedStratifiedKFold

from sklearn.feature\_selection import RFE

from sklearn.dummy import DummyClassifier

from sklearn.neighbors import KNeighborsClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import GridSearchCV

from sklearn.model\_selection import RandomizedSearchCV

from scipy.stats import randint

from sklearn.model\_selection import cross\_val\_score

from sklearn.metrics import classification\_report

from sklearn.metrics import confusion\_matrix

from sklearn.metrics import roc\_auc\_score

In [17]:

df1 = (

df

.assign(ph=lambda df\_:df\_.ph.fillna(df\_.ph.mean()),

Sulfate=lambda df\_:df\_.Sulfate.fillna(df\_.Sulfate.mean()),

Trihalomethanes=lambda df\_:df\_.Trihalomethanes.fillna(df\_.Trihalomethanes.mean())

)

)

df1.head()

Out[17]:

|  | ph | Hardness | Solids | Chloramines | Sulfate | Conductivity | Organic\_carbon | Trihalomethanes | Turbidity | Potability |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 7.080795 | 204.890455 | 20791.318981 | 7.300212 | 368.516441 | 564.308654 | 10.379783 | 86.990970 | 2.963135 | 0 |
| 1 | 3.716080 | 129.422921 | 18630.057858 | 6.635246 | 333.775777 | 592.885359 | 15.180013 | 56.329076 | 4.500656 | 0 |
| 2 | 8.099124 | 224.236259 | 19909.541732 | 9.275884 | 333.775777 | 418.606213 | 16.868637 | 66.420093 | 3.055934 | 0 |
| 3 | 8.316766 | 214.373394 | 22018.417441 | 8.059332 | 356.886136 | 363.266516 | 18.436524 | 100.341674 | 4.628771 | 0 |
| 4 | 9.092223 | 181.101509 | 17978.986339 | 6.546600 | 310.135738 | 398.410813 | 11.558279 | 31.997993 | 4.075075 | 0 |

In [18]:

*# # Apply mean value to the missing values*

*# df['ph'].fillna(df['ph'].mean(), inplace=True)*

*# df['Sulfate'].fillna(df['Sulfate'].mean(), inplace=True)*

*# df['Trihalomethanes'].fillna(df['Trihalomethanes'].mean(), inplace=True)*

*# df.isnull().sum()*

In [19]:

*# Separate into X and y variables*

X, y = df1.drop(['Potability'], axis=1), df1['Potability'].values

In [20]:

X.head()

Out[20]:

|  | ph | Hardness | Solids | Chloramines | Sulfate | Conductivity | Organic\_carbon | Trihalomethanes | Turbidity |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 7.080795 | 204.890455 | 20791.318981 | 7.300212 | 368.516441 | 564.308654 | 10.379783 | 86.990970 | 2.963135 |
| 1 | 3.716080 | 129.422921 | 18630.057858 | 6.635246 | 333.775777 | 592.885359 | 15.180013 | 56.329076 | 4.500656 |
| 2 | 8.099124 | 224.236259 | 19909.541732 | 9.275884 | 333.775777 | 418.606213 | 16.868637 | 66.420093 | 3.055934 |
| 3 | 8.316766 | 214.373394 | 22018.417441 | 8.059332 | 356.886136 | 363.266516 | 18.436524 | 100.341674 | 4.628771 |
| 4 | 9.092223 | 181.101509 | 17978.986339 | 6.546600 | 310.135738 | 398.410813 | 11.558279 | 31.997993 | 4.075075 |

In [21]:

X\_scaled = scale(X)

print("Mean of Unscaled Features: **{}**".format(np.mean(X)))

print("Standard Deviation of Unscaled Features: **{}**".format(np.std(X)))

print("Mean of Scaled Features: **{}**".format(np.mean(X\_scaled)))

print("Standard Deviation of Scaled Features: **{}**".format(np.std(X\_scaled)))

Mean of Unscaled Features: ph 7.080795

Hardness 196.369496

Solids 22014.092526

Chloramines 7.122277

Sulfate 333.775777

Conductivity 426.205111

Organic\_carbon 14.284970

Trihalomethanes 66.396293

Turbidity 3.966786

dtype: float64

Standard Deviation of Unscaled Features: ph 1.469732

Hardness 32.874743

Solids 8767.232421

Chloramines 1.582843

Sulfate 36.137095

Conductivity 80.811727

Organic\_carbon 3.307657

Trihalomethanes 15.767474

Turbidity 0.780263

dtype: float64

Mean of Scaled Features: 3.1955625682332546e-16

Standard Deviation of Scaled Features: 1.0

In [22]:

(

df1

.Potability

.value\_counts(normalize=True) *# display frequencies as a percentage*

)

Out[22]:

0 0.60989

1 0.39011

Name: Potability, dtype: float64

In [23]:

dummy\_clf = DummyClassifier(strategy='most\_frequent')

dummy\_clf.fit(X, y)

dummy\_clf.predict(X)

score = dummy\_clf.score(X, y)

print(score)

0.6098901098901099

In [24]:

Out[24]:

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

In [25]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.4, random\_state=2, stratify=y)

knn = KNeighborsClassifier(n\_neighbors=7)

knn.fit(X\_train, y\_train)

print(knn.score(X\_test, y\_test))

0.5659801678108314

In [26]:

neighbors = np.arange(1, 12)

train\_accuracy = np.empty(len(neighbors))

test\_accuracy = np.empty(len(neighbors))

for i, k **in** enumerate(neighbors):

knn = KNeighborsClassifier(n\_neighbors=k)

knn.fit(X\_train, y\_train)

train\_accuracy[i] = knn.score(X\_train, y\_train)

test\_accuracy[i] = knn.score(X\_test, y\_test)

plt.title('k-NN: Varying Number of Neighbors')

plt.plot(neighbors, test\_accuracy, label = 'Testing Accuracy')

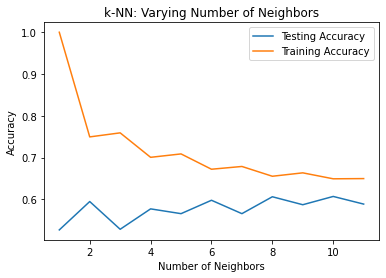
plt.plot(neighbors, train\_accuracy, label = 'Training Accuracy')

plt.legend()

plt.xlabel('Number of Neighbors')

plt.ylabel('Accuracy')

plt.show()



In [27]:

steps = [('scaler', StandardScaler()),

('knn', KNeighborsClassifier())]

pipeline = Pipeline(steps)

knn\_scaled = pipeline.fit(X\_train, y\_train)

knn\_unscaled = KNeighborsClassifier().fit(X\_train, y\_train)

print('Accuracy with Scaling: **{}**'.format(knn\_scaled.score(X\_test, y\_test)))

print('Accuracy without Scaling: **{}**'.format(knn\_unscaled.score(X\_test, y\_test)))

Accuracy with Scaling: 0.6247139588100686

Accuracy without Scaling: 0.5659801678108314

In [28]:

param\_dist = {"max\_depth": [3, None],

"max\_features": randint(1, 9),

"min\_samples\_leaf": randint(1, 9),

"criterion": ["gini", "entropy"]}

tree = DecisionTreeClassifier()

tree\_cv = RandomizedSearchCV(tree, param\_dist, cv=5)

tree\_cv.fit(X, y)

print("Tuned Decision Tree Parameters: **{}**".format(tree\_cv.best\_params\_))

print("Best score is **{}**".format(tree\_cv.best\_score\_))

Tuned Decision Tree Parameters: {'criterion': 'gini', 'max\_depth': 3, 'max\_features': 8, 'min\_samples\_leaf': 4}

Best score is 0.616912586110594

In [29]:

class **ModelBuild**():

def \_\_init\_\_(self, X, y, model=DecisionTreeClassifier(criterion='gini', max\_depth=3, min\_samples\_leaf=8)):

self.X = X

self.y = y

self.model = model

def \_train\_test\_split(self):

X\_train, X\_test, y\_train, y\_test = train\_test\_split(self.X, self.y, test\_size=0.3, random\_state=42)

return X\_train, X\_test, y\_train, y\_test

def \_pipeline(self):

steps = [('scaler', StandardScaler()),

('model\_name', self.model)]

return Pipeline(steps)

def model\_build(self):

if \_\_name\_\_ == "\_\_main\_\_":

X\_train, X\_test, y\_train, y\_test = self.\_train\_test\_split()

pipeline = self.\_pipeline()

fit = pipeline.fit(X\_train, y\_train)

return print("Accuracy: **{}**".format(pipeline.score(X\_test, y\_test)))

In [30]:

ModelBuild(X, y).model\_build()

Accuracy: 0.6378433367243134

In [31]:

class **FeatureSelection**(ModelBuild):

def \_\_init\_\_(self, X, y, model=RandomForestClassifier()):

super().\_\_init\_\_(X, y, model=RandomForestClassifier())

self.X = X

self.y = y

self.model = model

def rfe\_model(self):

model\_dict = dict()

for i **in** range(2, len(self.X.columns)):

rfe = RFE(estimator=self.model, n\_features\_to\_select=i)

model = DecisionTreeClassifier()

model\_dict[str(i)] = Pipeline(steps=[('rfe', rfe), ('mod', model)])

return model\_dict

def eval\_model(self, model):

cv = RepeatedStratifiedKFold(n\_splits=10, n\_repeats=3, random\_state=6)

scores = cross\_val\_score(model, self.X, self.y, scoring='accuracy', cv=cv, n\_jobs=-1, error\_score='raise')

return scores

def feature\_select(self, n\_feature):

rfe = RFE(estimator=self.model, n\_features\_to\_select=n\_feature)

rfe.fit(self.X, self.y)

*for i in range(X.shape[1]):*

for i, col **in** enumerate(X.columns):

print('Column: **%s**, Selected **%s**, Rank: **%.3f**' % (col, rfe.support\_[i], rfe.ranking\_[i]))

def feature\_selection(self):

if \_\_name\_\_ == "\_\_main\_\_":

models = self.rfe\_model()

results, names = list(), list()

for name, model **in** models.items():

scores = self.eval\_model(model)

results.append(scores)

names.append(name)

print(f'**{**name**}**, mean\_score: **{**np.mean(scores)**}**, std\_score: **{**np.std(scores)**}**')

box\_plt = plt.boxplot(results, labels=names, showmeans=True)

return box\_plt

In [32]:

box = FeatureSelection(X, y, model=DecisionTreeClassifier(criterion='gini', max\_depth=3, min\_samples\_leaf=8)).feature\_selection()

plt.show()

2, mean\_score: 0.5645039282961636, std\_score: 0.02339914989185637

3, mean\_score: 0.5499611521344572, std\_score: 0.022912477766004778

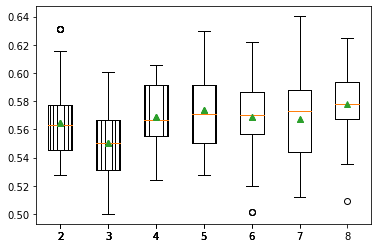
4, mean\_score: 0.5690848686009796, std\_score: 0.02395412036612033

5, mean\_score: 0.5734665970512918, std\_score: 0.026497320847046976

6, mean\_score: 0.5686746351408468, std\_score: 0.026467739214973585

7, mean\_score: 0.5670498620123816, std\_score: 0.030790547204677197

8, mean\_score: 0.5780360383879067, std\_score: 0.023313092263193808



In [33]:

features = FeatureSelection(X, y, model=DecisionTreeClassifier(criterion='gini', max\_depth=3, min\_samples\_leaf=8)).feature\_select(5)

Column: ph, Selected True, Rank: 1.000

Column: Hardness, Selected False, Rank: 5.000

Column: Solids, Selected True, Rank: 1.000

Column: Chloramines, Selected False, Rank: 4.000

Column: Sulfate, Selected True, Rank: 1.000

Column: Conductivity, Selected False, Rank: 3.000

Column: Organic\_carbon, Selected False, Rank: 2.000

Column: Trihalomethanes, Selected True, Rank: 1.000

Column: Turbidity, Selected True, Rank: 1.000

In [34]:

from lightgbm import LGBMClassifier

In [35]:

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size = 0.4, random\_state=2, stratify=y)

lgbm = LGBMClassifier()

lgbm.fit(X\_train, y\_train)

y\_pred = lgbm.predict(X\_test)

print(lgbm.score(X\_test, y\_test))

print(classification\_report(y\_test, y\_pred))

print(confusion\_matrix(y\_test, y\_pred))

0.6468344774980931

precision recall f1-score support

0 0.68 0.80 0.73 800

1 0.57 0.41 0.47 511

accuracy 0.65 1311

macro avg 0.62 0.60 0.60 1311

weighted avg 0.63 0.65 0.63 1311

[[641 159]

[304 207]]

In [36]:

lgbm.get\_params()

Out[36]:

{'boosting\_type': 'gbdt',

'class\_weight': None,

'colsample\_bytree': 1.0,

'importance\_type': 'split',

'learning\_rate': 0.1,

'max\_depth': -1,

'min\_child\_samples': 20,

'min\_child\_weight': 0.001,

'min\_split\_gain': 0.0,

'n\_estimators': 100,

'n\_jobs': -1,

'num\_leaves': 31,

'objective': None,

'random\_state': None,

'reg\_alpha': 0.0,

'reg\_lambda': 0.0,

'silent': True,

'subsample': 1.0,

'subsample\_for\_bin': 200000,

'subsample\_freq': 0}

In [37]:

steps = [('scaler', StandardScaler()),

('lgbm', LGBMClassifier())]

pipeline = Pipeline(steps)

parameters = {

'lgbm\_\_learning\_rate':[0.03, 0.05, 0.1],

'lgbm\_\_objective':['binary'],

'lgbm\_\_metric':['binary\_logloss'],

'lgbm\_\_max\_depth':[10],

'lgbm\_\_n\_estimators':[100, 200, 300]

}

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.3, random\_state=42)

cv = GridSearchCV(pipeline, parameters, cv=3)

cv.fit(X\_train, y\_train)

y\_pred = cv.predict(X\_test)

In [38]:

print(f'Best score : **{**cv.best\_score\_**}**')

print(f'Best params : **{**cv.best\_params\_**}**')

print("Accuracy: **{}**".format(cv.score(X\_test, y\_test)))

print(classification\_report(y\_test, y\_pred))

Best score : 0.6489317774812078

Best params : {'lgbm\_\_learning\_rate': 0.03, 'lgbm\_\_max\_depth': 10, 'lgbm\_\_metric': 'binary\_logloss', 'lgbm\_\_n\_estimators': 100, 'lgbm\_\_objective': 'binary'}

Accuracy: 0.6734486266531028

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | precision | recall | f1-score | support |
| 0 | 0.69 | 0.89 | 0.77 | 617 |
| 1 | 0.62 | 0.31 | 0.42 | 366 |
| accuracy |  |  | 0.67 | 983 |
| macro avg | 0.65 | 0.60 | 0.60 | 983 |
| weighted avg | 0.66 | 0.67 | 0.64 | 983 |