Water Potability Prediction

The Battleship Team

DigitalSkola

Final Project

Our Team

Muhammad Nurfalah Rohmawan

Sandy Febrian

Samsul Rizal

Renaldi Ega

Table of Content

A. Background of Study

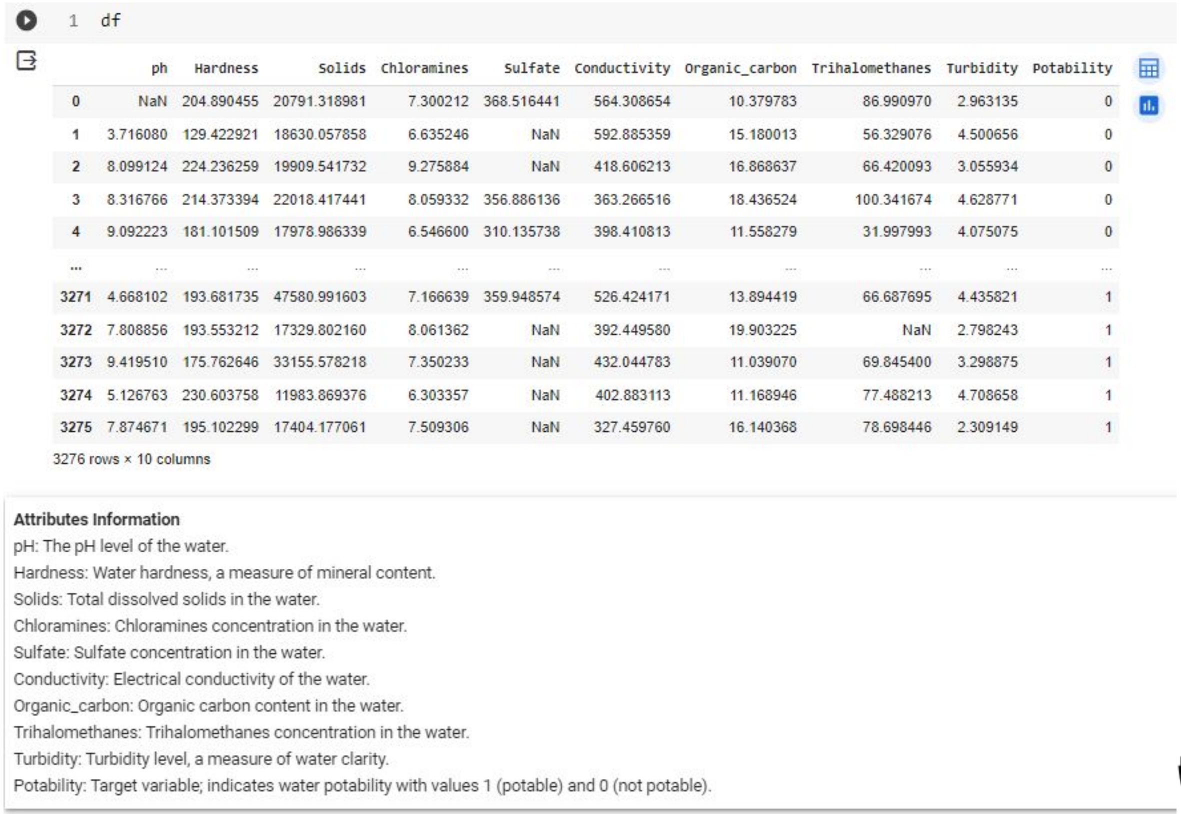
- Human existence relies on water, but both surface and groundwater sources are often polluted by various contaminants generated from rapid population growth and urban development.
- Globally, 20% of people lack clean drinking water, and 50% lack safe sanitation systems, as per the United Nations Environment Program (2000).
- In recent years, predicting water quality has become a significant research area due to its critical importance to address any issue related to clean water in the future.
- Predictive models can provide early warnings for water quality issues, such as contamination events or natural disasters. This information is invaluable for emergency response and safeguarding public health.

B. Material and Method

In this project, we aim to predict the water potability based on a set of features. To train our machine learning model, we will use Scikit Learn and deploy to Streamlit

Source: https://www.kaggle.com/datasets/uom190346a/water-quality-and-potability/data

Evaluating Data Structure



```
df
```

	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
...	...	...	...	...	...	...	...	...	...	...
3271	4.668102	193.681735	47580.991603	7.166639	359.948574	526.424171	13.894419	66.687695	4.435821	1
3272	7.808856	193.553212	17329.802160	8.061362	NaN	392.449580	19.903225	NaN	2.798243	1
3273	9.419510	175.762646	33155.578218	7.350233	NaN	432.044783	11.039070	69.845400	3.298875	1
3274	5.126763	230.603758	11983.869376	6.303357	NaN	402.883113	11.168946	77.488213	4.708658	1
3275	7.874671	195.102299	17404.177061	7.509306	NaN	327.459760	16.140368	78.698446	2.309149	1

3276 rows × 10 columns

Attributes Information

pH: The pH level of the water.

Hardness: Water hardness, a measure of mineral content.

Solids: Total dissolved solids in the water.

Chloramines: Chloramines concentration in the water.

Sulfate: Sulfate concentration in the water.

Conductivity: Electrical conductivity of the water.

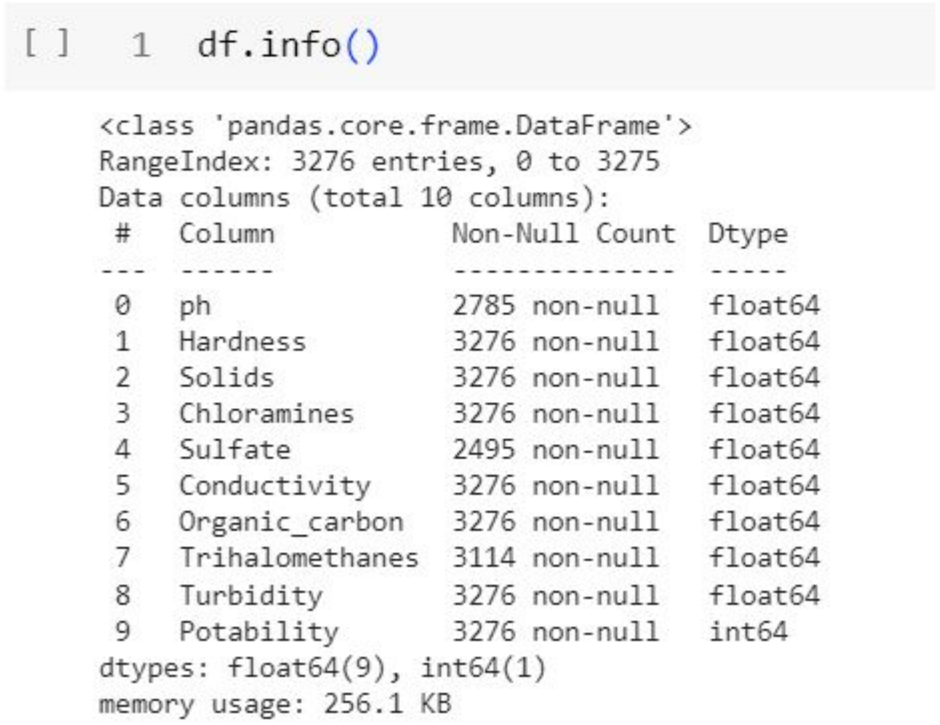
Organic_carbon: Organic carbon content in the water.

Trihalomethanes: Trihalomethanes concentration in the water.

Turbidity: Turbidity level, a measure of water clarity.

Potability: Target variable; indicates water potability with values 1 (potable) and 0 (not potable).

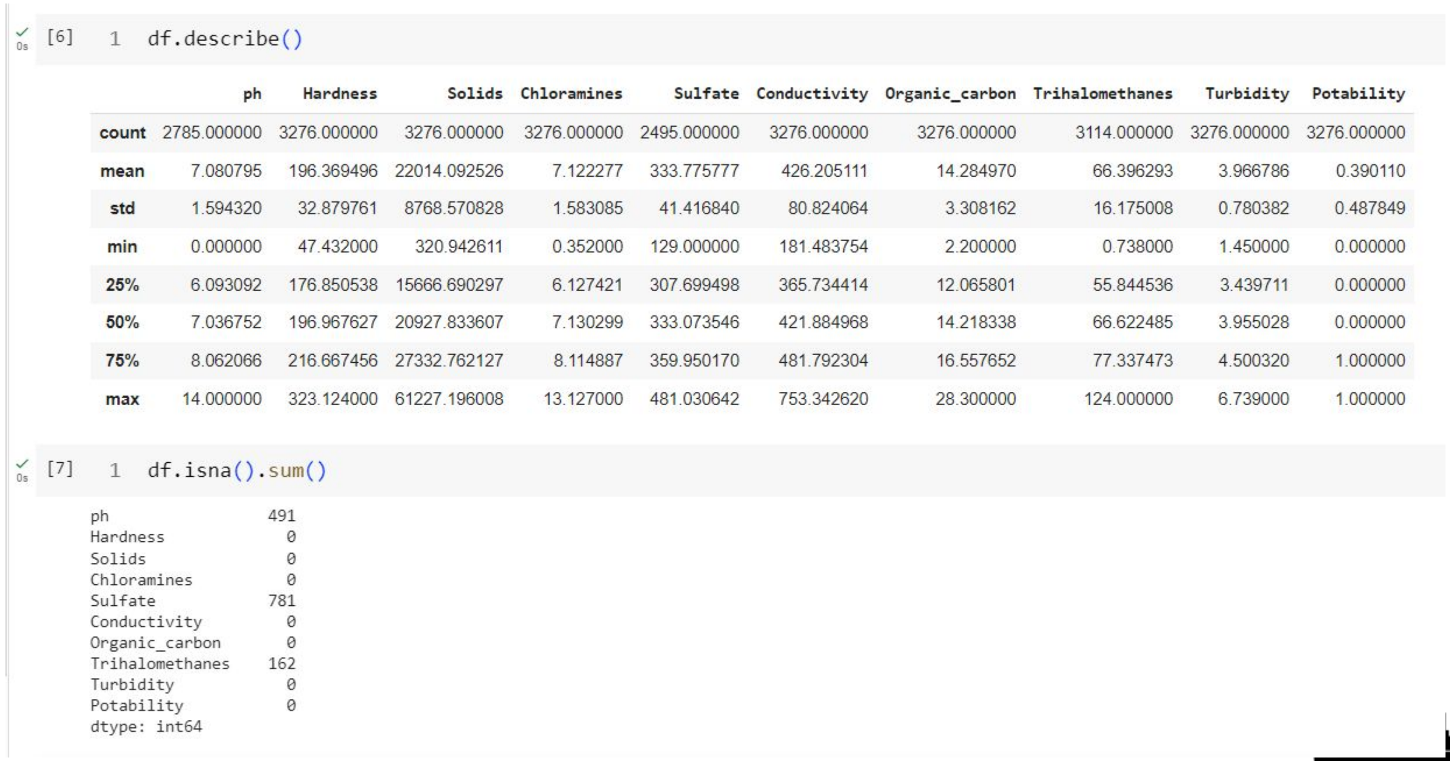
Evaluating Data Structure



```
[ ]   df.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
```

Evaluating Data Structure



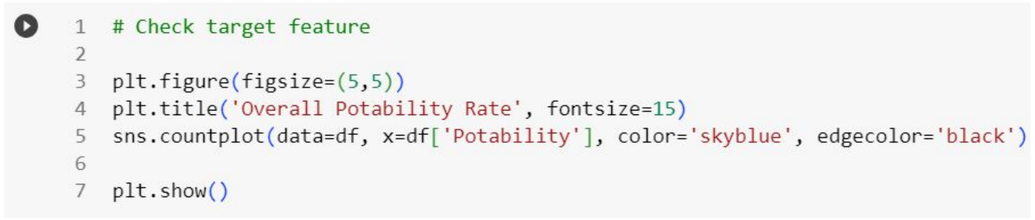
```
[6] df.describe()
```

	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

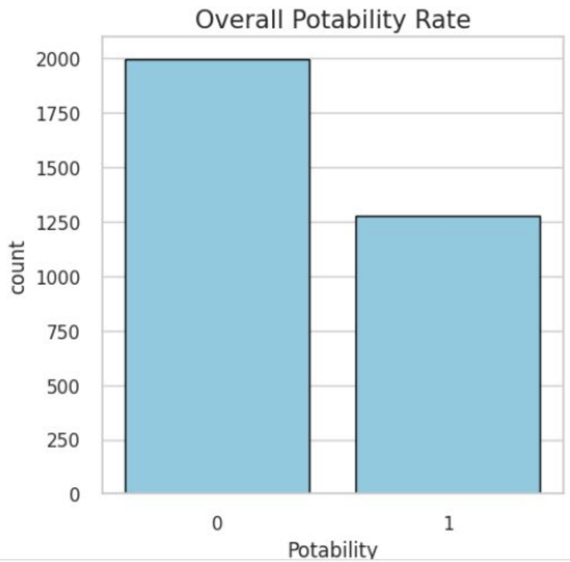
```
[7] df.isna().sum()
```

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

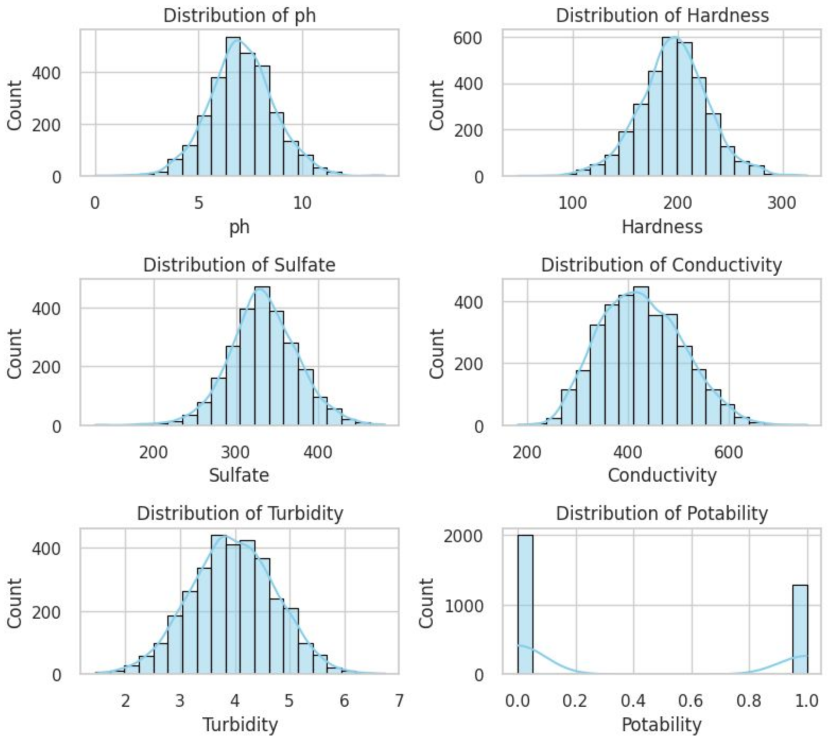
Checking Target Feature



```
# Check target feature

plt.figure(figsize=(5,5))
plt.title('Overall Potability Rate', fontsize=15)
sns.countplot(data=df, x=df['Potability'], color='skyblue', edgecolor='black')

plt.show()
```

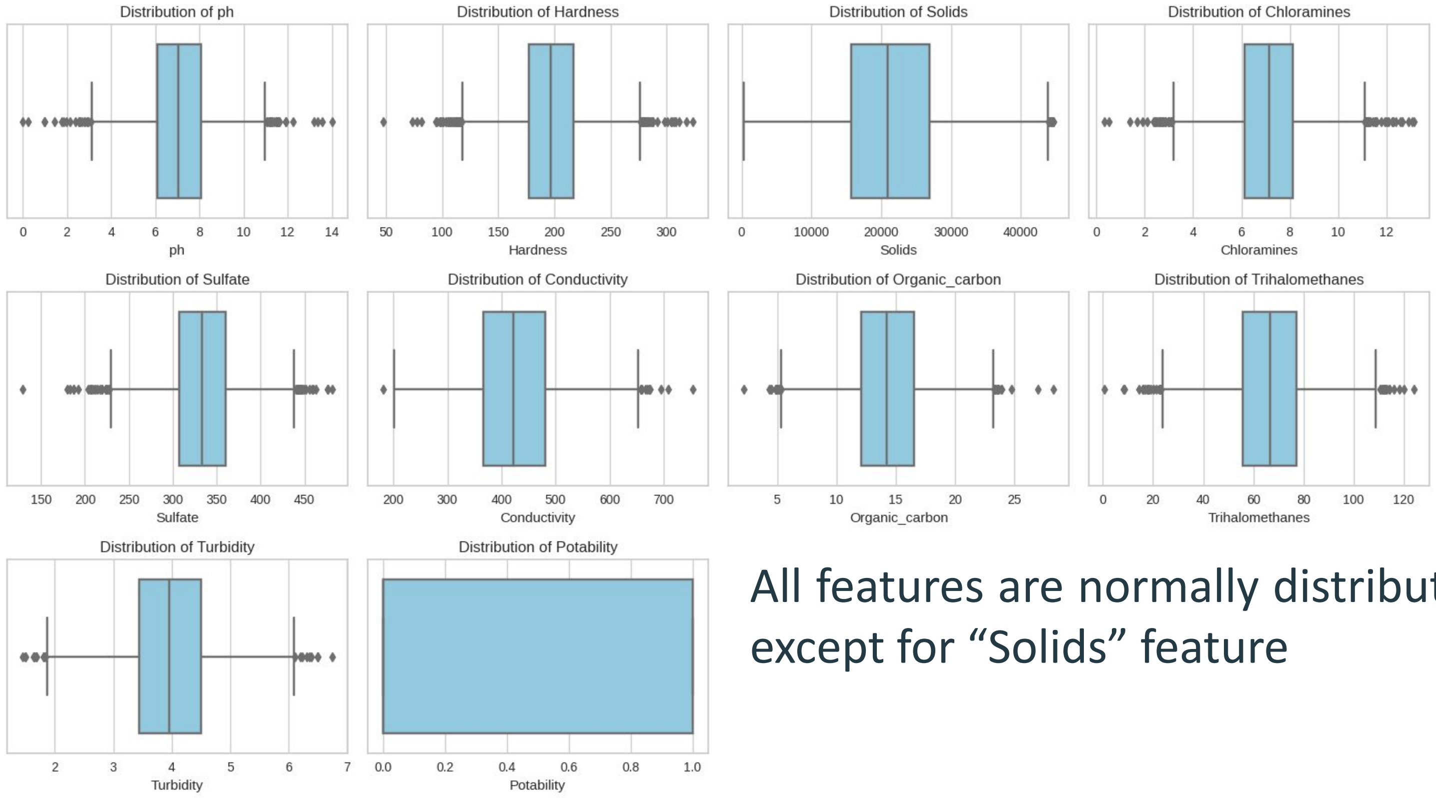


Distribution with Histplot



All features are normally distributed except for “Solids” feature

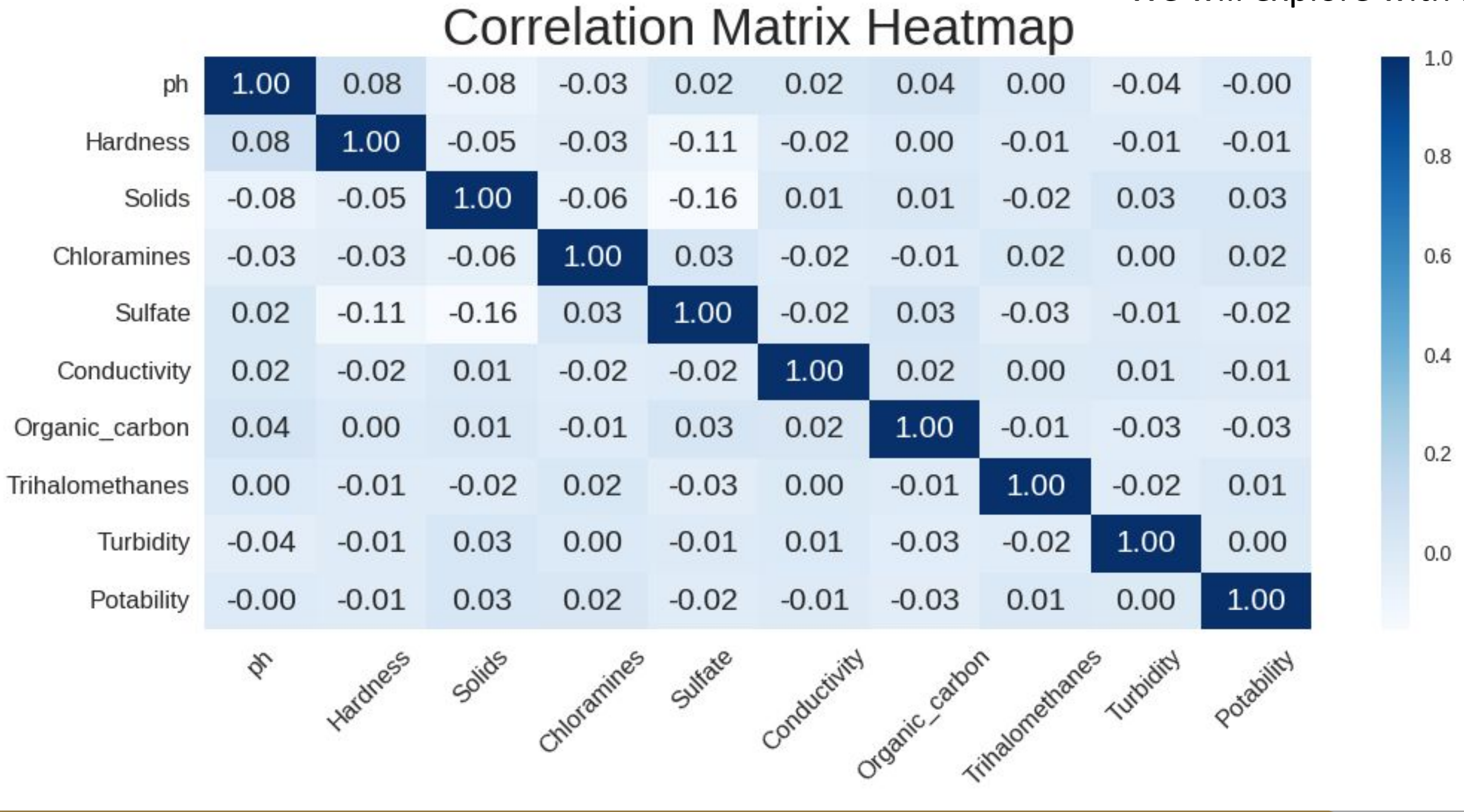
Distribution with Boxplot



All features are normally distributed except for “Solids” feature

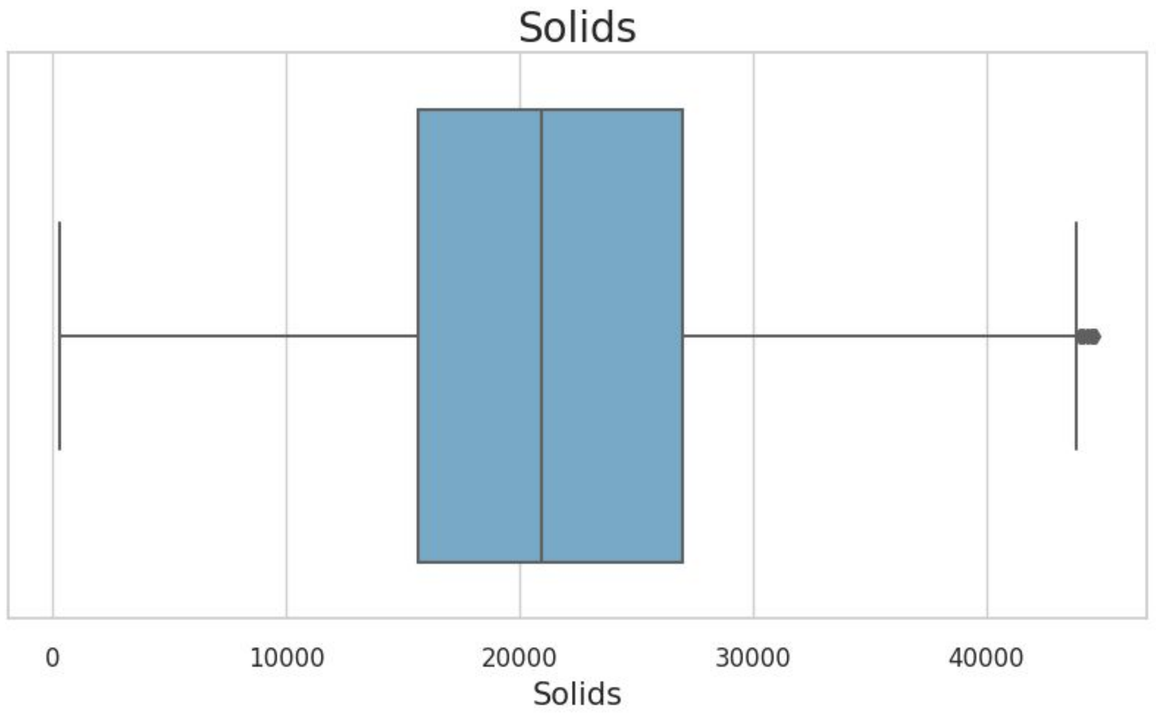
Correlation Matrix

All features seems to have low correlation with Potability.
We will explore with Non-Linear model

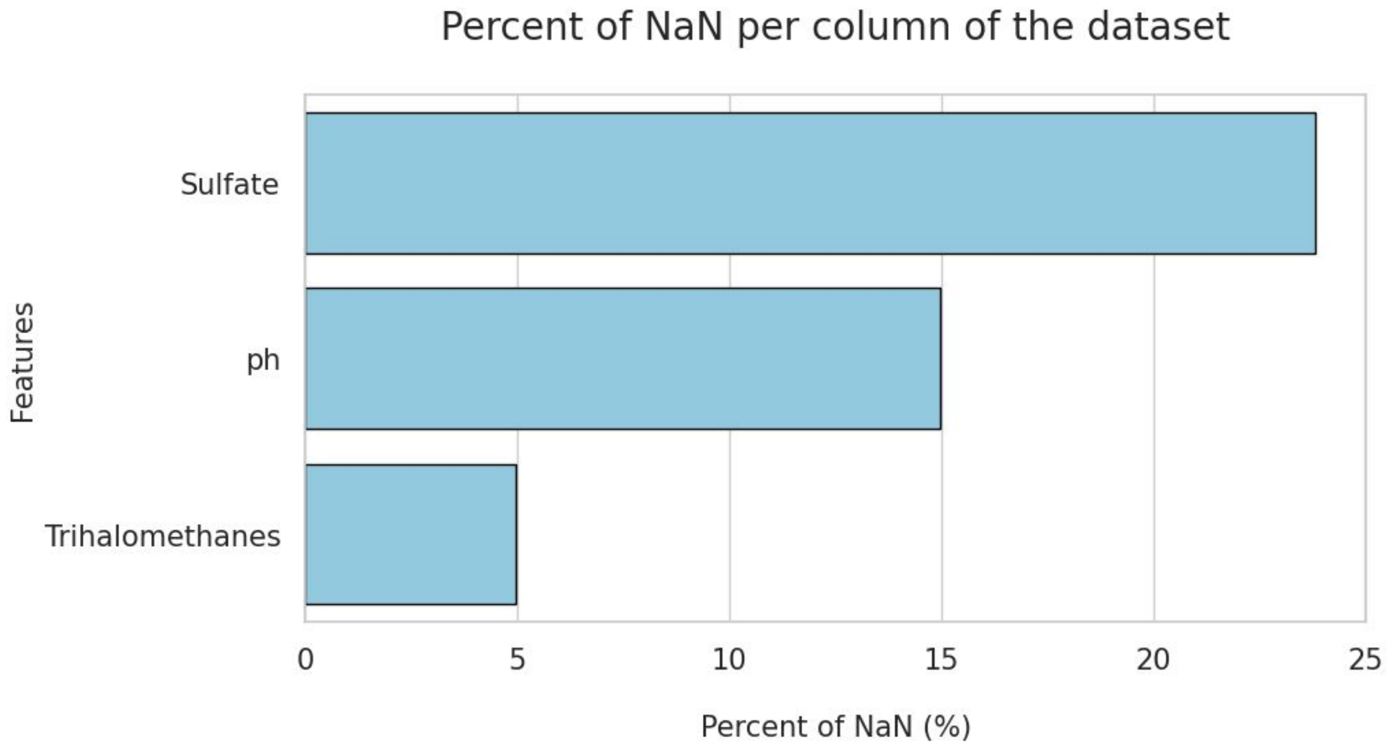


	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity	Potability
ph	1.00	0.08	-0.08	-0.03	0.02	0.02	0.04	0.00	-0.04	-0.00
Hardness	0.08	1.00	-0.05	-0.03	-0.11	-0.02	0.00	-0.01	-0.01	-0.01
Solids	-0.08	-0.05	1.00	-0.06	-0.16	0.01	0.01	-0.02	0.03	0.03
Chloramines	-0.03	-0.03	-0.06	1.00	0.03	-0.02	-0.01	0.02	0.00	0.02
Sulfate	0.02	-0.11	-0.16	0.03	1.00	-0.02	0.03	-0.03	-0.01	-0.02
Conductivity	0.02	-0.02	0.01	-0.02	-0.02	1.00	0.02	0.00	0.01	-0.01
Organic_carbon	0.04	0.00	0.01	-0.01	0.03	0.02	1.00	-0.01	-0.03	-0.03
Trihalomethanes	0.00	-0.01	-0.02	0.02	-0.03	0.00	-0.01	1.00	-0.02	0.01
Turbidity	-0.04	-0.01	0.03	0.00	-0.01	0.01	-0.03	-0.02	1.00	0.00
Potability	-0.00	-0.01	0.03	0.02	-0.02	-0.01	-0.03	0.01	0.00	1.00

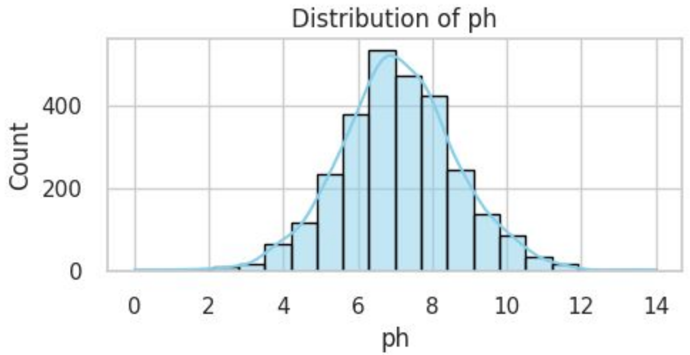
Cleaning Outliers

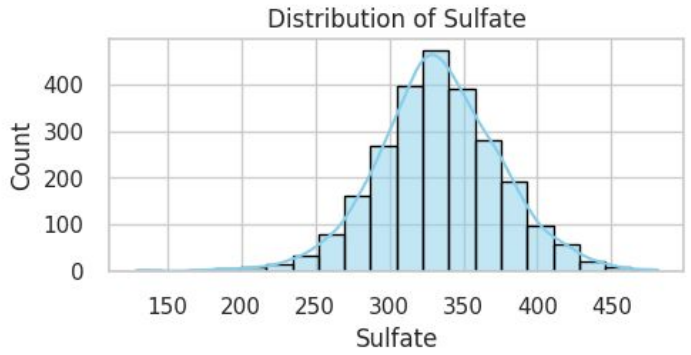


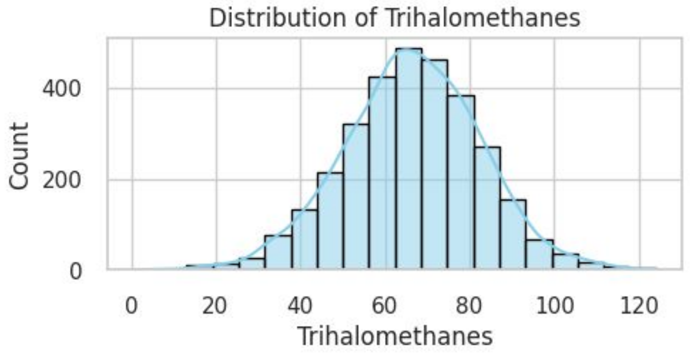
Check NaN Values



Check NaN Values





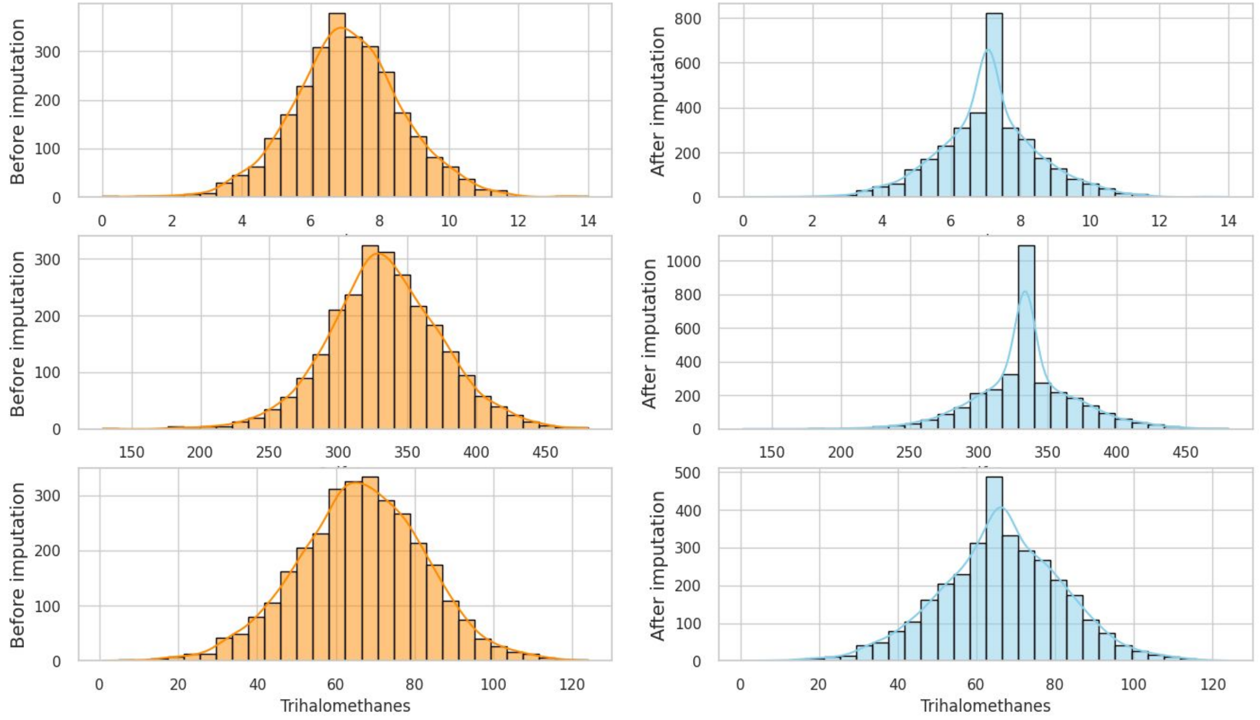


ph and Trihalomethanes has normal distribution = we use mean or median
Sulfate has left skew = we use mean

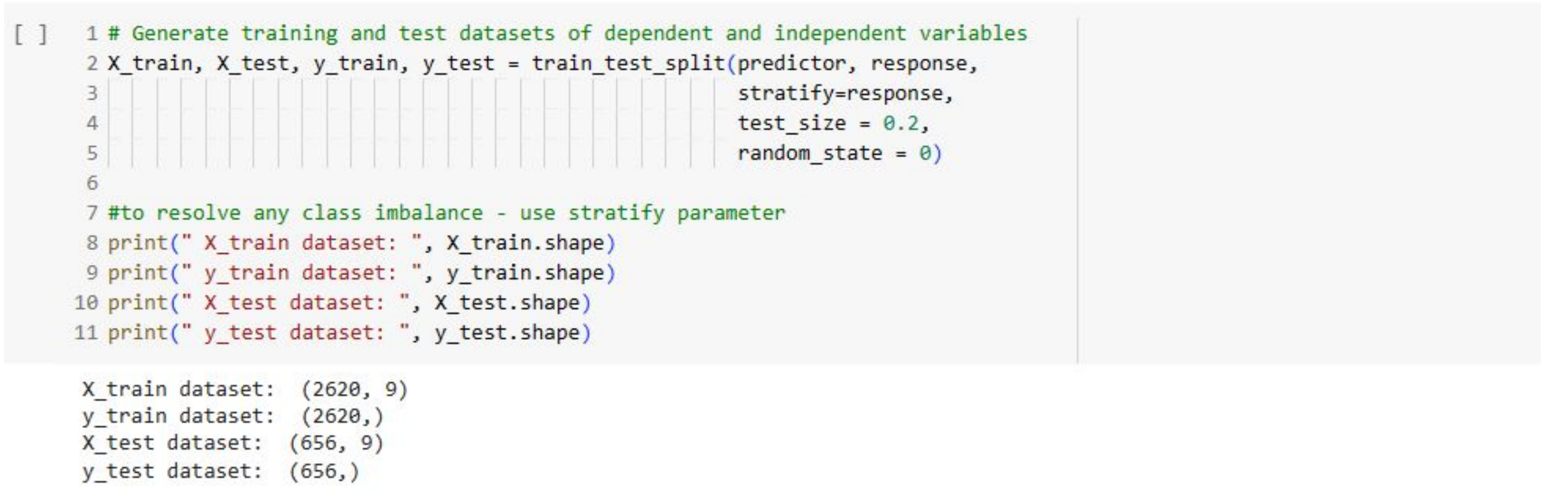
To simplify, all feature will use mean
SimpleImputer

Compare Before after Imputer

Distribution is not affected after SimpleImputer



Split data set into training and testing

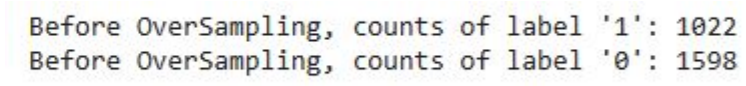


```
[ ] # Generate training and test datasets of dependent and independent variables
    X_train, X_test, y_train, y_test = train_test_split(predictor, response,
                                                         stratify=response,
                                                         test_size = 0.2,
                                                         random_state = 0)

    #to resolve any class imbalance - use stratify parameter
    print(" X_train dataset: ", X_train.shape)
    print(" y_train dataset: ", y_train.shape)
    print(" X_test dataset: ", X_test.shape)
    print(" y_test dataset: ", y_test.shape)
```

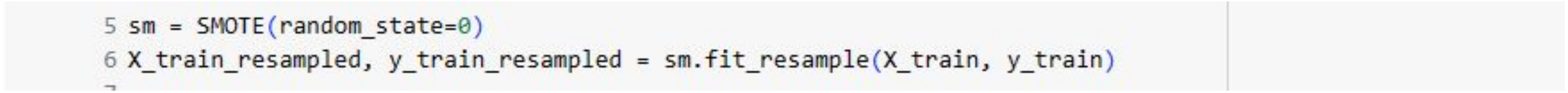
```
X_train dataset:  (2620, 9)
y_train dataset:  (2620,)
X_test dataset:  (656, 9)
y_test dataset:  (656,)
```

Generate training and testing datasets from **df_imputed.** First, define a predictor (X) and response (y) variable. And then split with the proportion are 80% for training and 20% for testing.

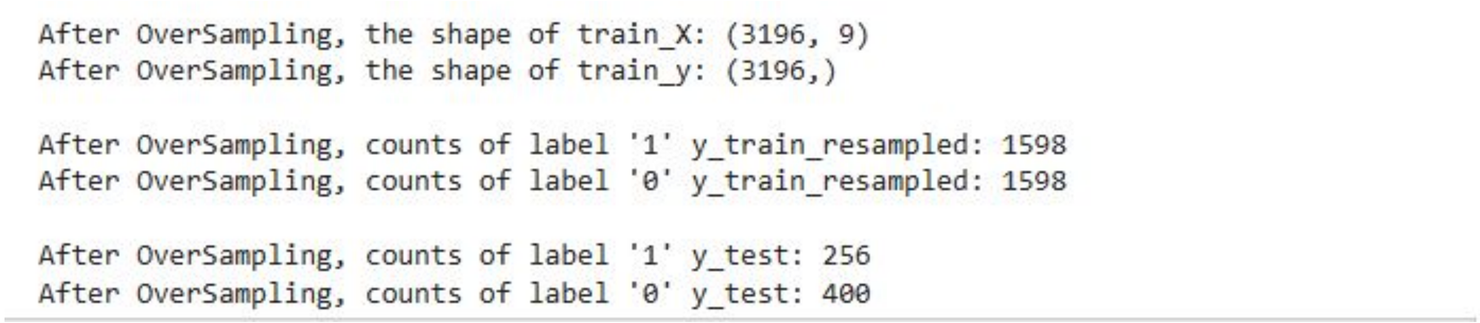
Handling Imbalanced Data with SMOTE



```
Before OverSampling, counts of label '1': 1022
Before OverSampling, counts of label '0': 1598
```

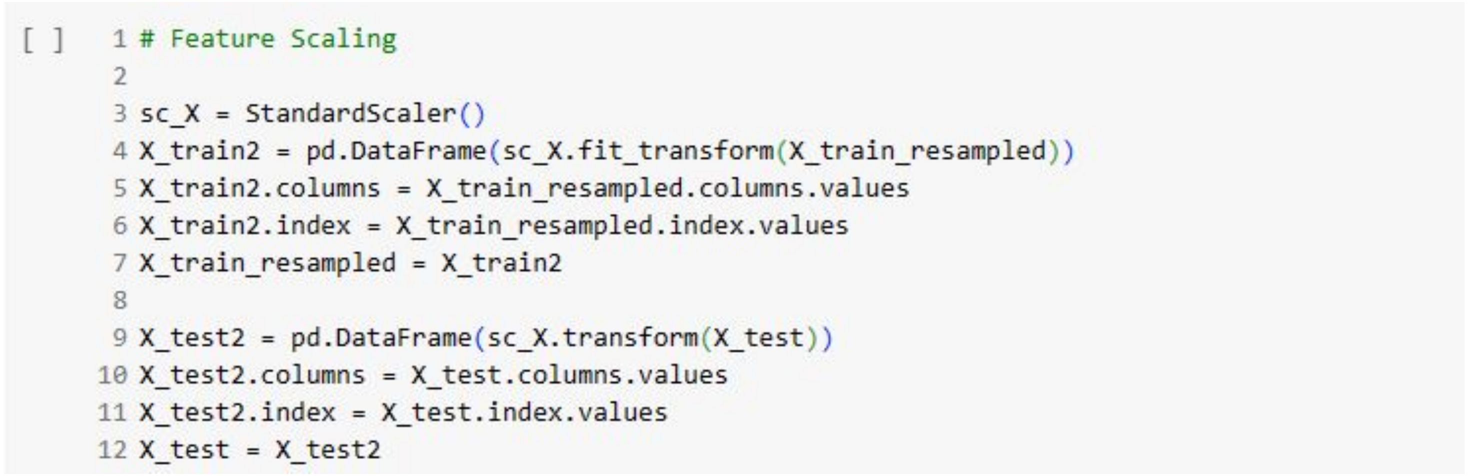


```
sm = SMOTE(random_state=0)
X_train_resampled, y_train_resampled = sm.fit_resample(X_train, y_train)
```



```
After OverSampling, the shape of train_X: (3196, 9)
After OverSampling, the shape of train_y: (3196,)

After OverSampling, counts of label '1' y_train_resampled: 1598
After OverSampling, counts of label '0' y_train_resampled: 1598

After OverSampling, counts of label '1' y_test: 256
After OverSampling, counts of label '0' y_test: 400
```

Feature Scaling



```
[ ]  # Feature Scaling

     sc_X = StandardScaler()
     X_train2 = pd.DataFrame(sc_X.fit_transform(X_train_resampled))
     X_train2.columns = X_train_resampled.columns.values
     X_train2.index = X_train_resampled.index.values
     X_train_resampled = X_train2

     X_test2 = pd.DataFrame(sc_X.transform(X_test))
     X_test2.columns = X_test.columns.values
     X_test2.index = X_test.index.values
     X_test = X_test2
```

Feature scaling using **StandardScaler(),** it makes mean = 0 and scales the data to unit variance.

Model

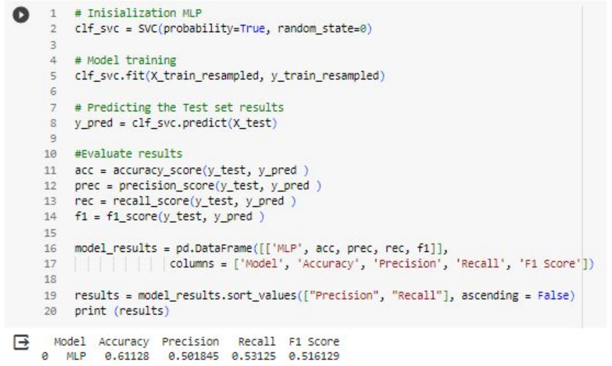
To enhance the accuracy and efficacy of water quality classification, this analysis uses the **Scikit-learn library** to implement six (6) Classifiers as below:

1. Support Vector Classifier (SVC)
2. Multi-Layer Perceptron Classifier (MPL)
3. XGBoost Classifier (XGB)
4. Decision Tree Classifier (DT
5. Random Forest Classifier (RF)
6. K-Neighbor Classifier (KNN)

Which one is the best?

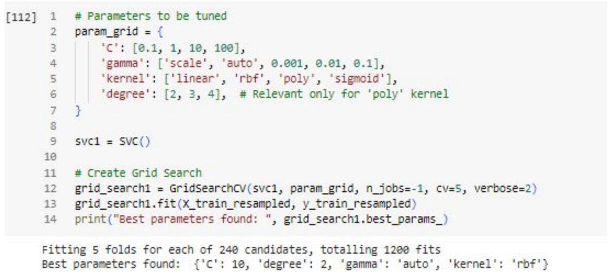
Model

Support Vector Classifier (SVC)



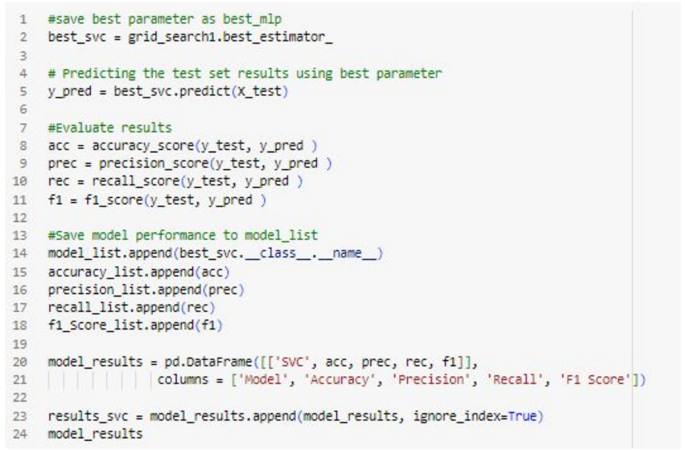
```
# Inisialization MLP
clf_svc = SVC(probability=True, random_state=0)

# Model training
clf_svc.fit(X_train_resampled, y_train_resampled)

# Predicting the Test set results
y_pred = clf_svc.predict(X_test)

#Evaluate results
acc = accuracy_score(y_test, y_pred )
prec = precision_score(y_test, y_pred )
rec = recall_score(y_test, y_pred )
f1 = f1_score(y_test, y_pred )

model_results = pd.DataFrame([['MLP', acc, prec, rec, f1]],
                  columns = ['Model', 'Accuracy', 'Precision', 'Recall', 'F1 Score'])

results = model_results.sort_values(["Precision", "Recall"], ascending = False)
print (results)
```

```
  Model  Accuracy  Precision   Recall  F1 Score
0   MLP   0.61128   0.501845  0.53125  0.516129
```

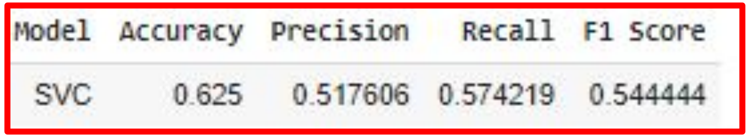


```
[112]
# Parameters to be tuned
param_grid = {
    'C': [0.1, 1, 10, 100],
    'gamma': ['scale', 'auto', 0.001, 0.01, 0.1],
    'kernel': ['linear', 'rbf', 'poly', 'sigmoid'],
    'degree': [2, 3, 4],  # Relevant only for 'poly' kernel
}

svc1 = SVC()

# Create Grid Search
grid_search1 = GridSearchCV(svc1, param_grid, n_jobs=-1, cv=5, verbose=2)
grid_search1.fit(X_train_resampled, y_train_resampled)
print("Best parameters found: ", grid_search1.best_params_)
```

```
Fitting 5 folds for each of 240 candidates, totalling 1200 fits
Best parameters found:  {'C': 10, 'degree': 2, 'gamma': 'auto', 'kernel': 'rbf'}
```



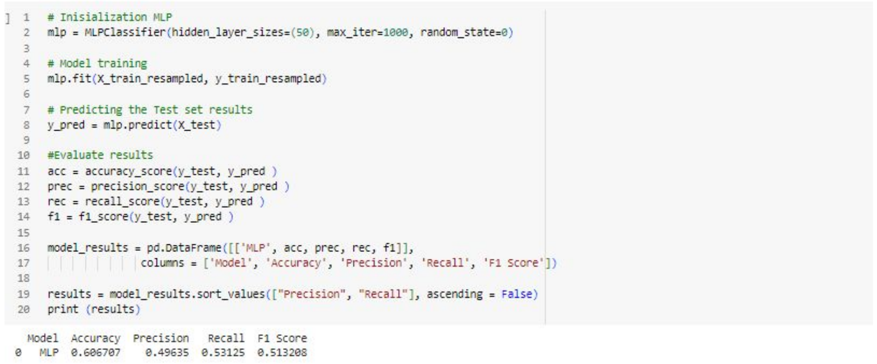
```
#save best parameter as best_mlp
best_svc = grid_search1.best_estimator_

# Predicting the test set results using best parameter
y_pred = best_svc.predict(X_test)

#Evaluate results
acc = accuracy_score(y_test, y_pred )
prec = precision_score(y_test, y_pred )
rec = recall_score(y_test, y_pred )
f1 = f1_score(y_test, y_pred )

#Save model performance to model_list
model_list.append(best_svc.__class__.__name__)
accuracy_list.append(acc)
precision_list.append(prec)
recall_list.append(rec)
f1_Score_list.append(f1)

model_results = pd.DataFrame([['SVC', acc, prec, rec, f1]],
                  columns = ['Model', 'Accuracy', 'Precision', 'Recall', 'F1 Score'])

results_svc = model_results.append(model_results, ignore_index=True)
model_results
```



Model	Accuracy	Precision	Recall	F1 Score
SVC	0.625	0.517606	0.574219	0.544444

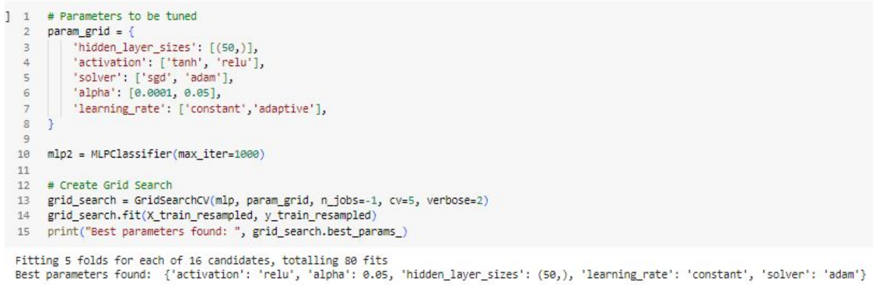
Model

Multi-Layer Perceptron Classifier (MPL)



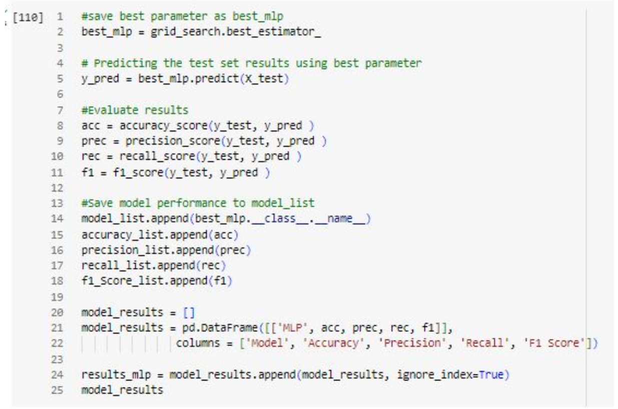
```
# Inisialization MLP
mlp = MLPClassifier(hidden_layer_sizes=(50), max_iter=1000, random_state=0)

# Model training
mlp.fit(X_train_resampled, y_train_resampled)

# Predicting the Test set results
y_pred = mlp.predict(X_test)

#Evaluate results
acc = accuracy_score(y_test, y_pred )
prec = precision_score(y_test, y_pred )
rec = recall_score(y_test, y_pred )
f1 = f1_score(y_test, y_pred )

model_results = pd.DataFrame([['MLP', acc, prec, rec, f1]],
                             columns = ['Model', 'Accuracy', 'Precision', 'Recall', 'F1 Score'])

results = model_results.sort_values(["Precision", "Recall"], ascending = False)
print (results)
```

```
  Model  Accuracy  Precision   Recall  F1 Score
0   MLP  0.606707    0.49635  0.53125  0.513208
```

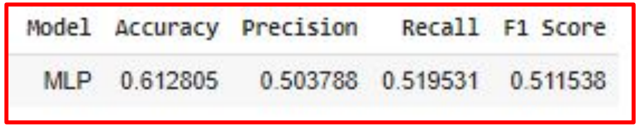


```
# Parameters to be tuned
param_grid = {
    'hidden_layer_sizes': [(50,)],
    'activation': ['tanh', 'relu'],
    'solver': ['sgd', 'adam'],
    'alpha': [0.0001, 0.05],
    'learning_rate': ['constant','adaptive'],
}

mlp2 = MLPClassifier(max_iter=1000)

# Create Grid Search
grid_search = GridSearchCV(mlp, param_grid, n_jobs=-1, cv=5, verbose=2)
grid_search.fit(X_train_resampled, y_train_resampled)
print("Best parameters found: ", grid_search.best_params_)
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits
Best parameters found:  {'activation': 'relu', 'alpha': 0.05, 'hidden_layer_sizes': (50,), 'learning_rate': 'constant', 'solver': 'adam'}
```



```
[110]
#save best parameter as best_mlp
best_mlp = grid_search.best_estimator_

# Predicting the test set results using best parameter
y_pred = best_mlp.predict(X_test)

#Evaluate results
acc = accuracy_score(y_test, y_pred )
prec = precision_score(y_test, y_pred )
rec = recall_score(y_test, y_pred )
f1 = f1_score(y_test, y_pred )

#Save model performance to model_list
model_list.append(best_mlp.__class__.__name__)
accuracy_list.append(acc)
precision_list.append(prec)
recall_list.append(rec)
f1_Score_list.append(f1)

model_results = []
model_results = pd.DataFrame([['MLP', acc, prec, rec, f1]],
                             columns = ['Model', 'Accuracy', 'Precision', 'Recall', 'F1 Score'])

results_mlp = model_results.append(model_results, ignore_index=True)
model_results
```

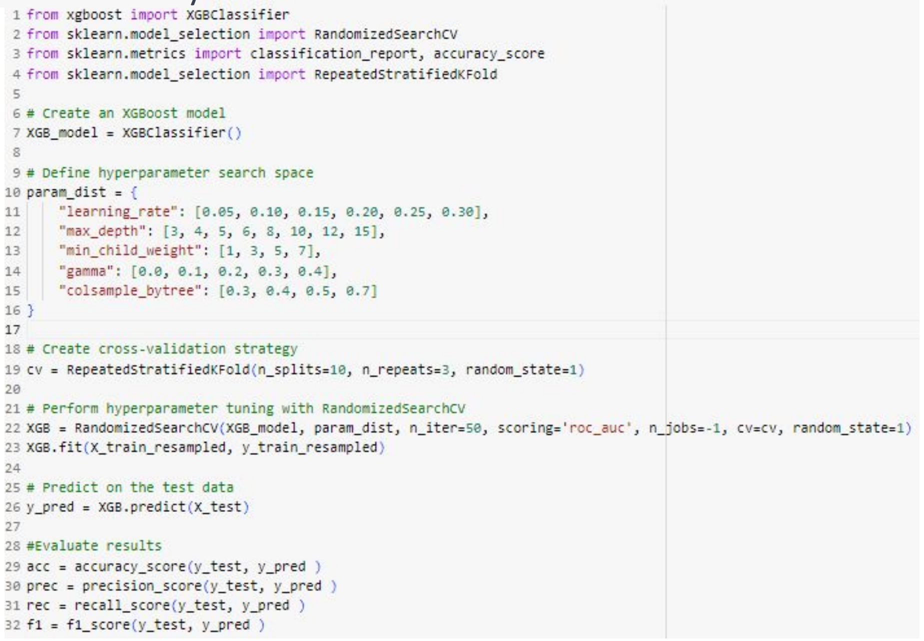


Model	Accuracy	Precision	Recall	F1 Score
MLP	0.612805	0.503788	0.519531	0.511538

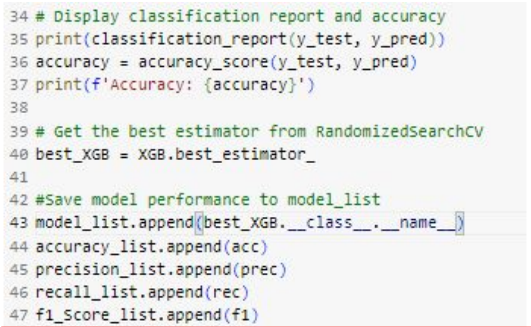
Model

XG Boost Classifier :

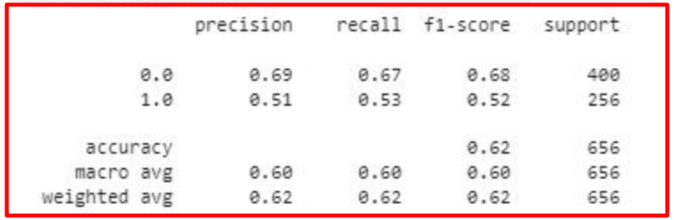
It's an extension of the traditional gradient boosting algorithm and is known for its high performance and efficiency



```python
from xgboost import XGBClassifier
from sklearn.model_selection import RandomizedSearchCV
from sklearn.metrics import classification_report, accuracy_score
from sklearn.model_selection import RepeatedStratifiedKFold

# Create an XGBoost model
XGB_model = XGBClassifier()

# Define hyperparameter search space
param_dist = {
    "learning_rate": [0.05, 0.10, 0.15, 0.20, 0.25, 0.30],
    "max_depth": [3, 4, 5, 6, 8, 10, 12, 15],
    "min_child_weight": [1, 3, 5, 7],
    "gamma": [0.0, 0.1, 0.2, 0.3, 0.4],
    "colsample_bytree": [0.3, 0.4, 0.5, 0.7]
}

# Create cross-validation strategy
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

# Perform hyperparameter tuning with RandomizedSearchCV
XGB = RandomizedSearchCV(XGB_model, param_dist, n_iter=50, scoring='roc_auc', n_jobs=-1, cv=cv, random_state=1)
XGB.fit(X_train_resampled, y_train_resampled)

# Predict on the test data
y_pred = XGB.predict(X_test)

#Evaluate results
acc = accuracy_score(y_test, y_pred )
prec = precision_score(y_test, y_pred )
rec = recall_score(y_test, y_pred )
f1 = f1_score(y_test, y_pred )
```



```python
# Display classification report and accuracy
print(classification_report(y_test, y_pred))
accuracy = accuracy_score(y_test, y_pred)
print(f'Accuracy: {accuracy}')

# Get the best estimator from RandomizedSearchCV
best_XGB = XGB.best_estimator_

#Save model performance to model_list
model_list.append(best_XGB.__class__.__name__)
accuracy_list.append(acc)
precision_list.append(prec)
recall_list.append(rec)
f1_Score_list.append(f1)
```

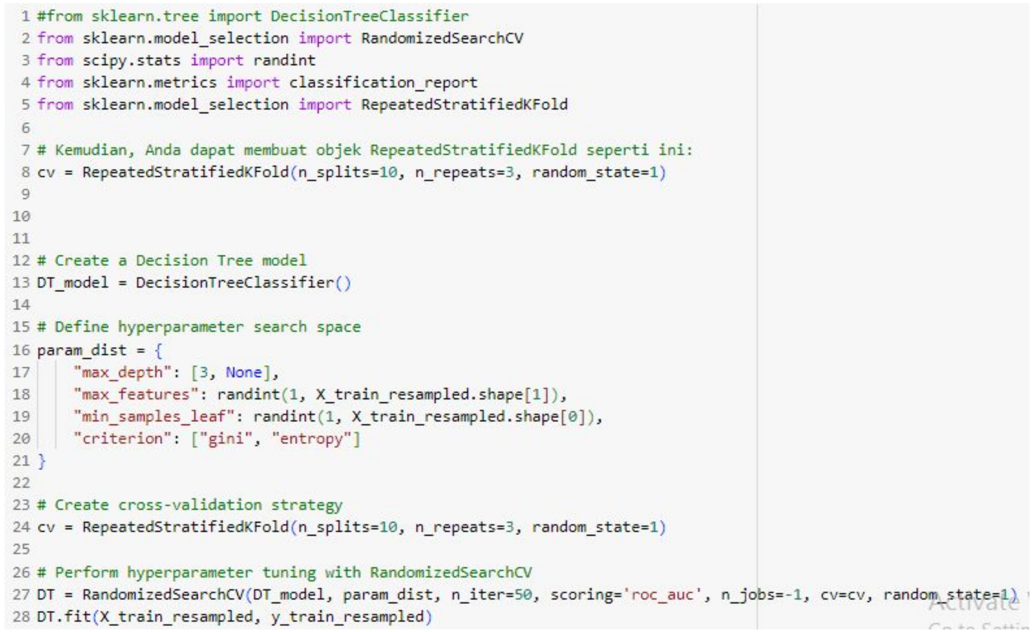


```
              precision    recall  f1-score   support

         0.0       0.69      0.67      0.68       400
         1.0       0.51      0.53      0.52       256

    accuracy                           0.62       656
   macro avg       0.60      0.60      0.60       656
weighted avg       0.62      0.62      0.62       656
```

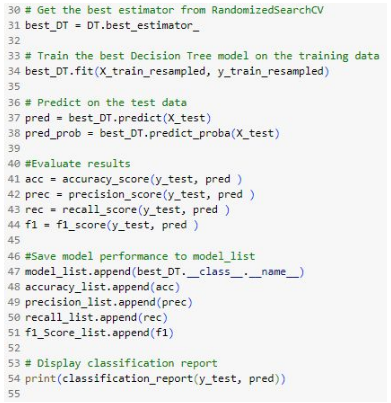
Model

Decision Tree

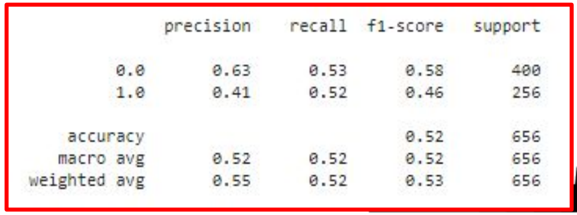
A decision tree is a popular supervised machine learning algorithm used for both classification and regression tasks..



```
#from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint
from sklearn.metrics import classification_report
from sklearn.model_selection import RepeatedStratifiedKFold

# Kemudian, Anda dapat membuat objek RepeatedStratifiedKFold seperti ini:
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)



# Create a Decision Tree model
DT_model = DecisionTreeClassifier()

# Define hyperparameter search space
param_dist = {
    "max_depth": [3, None],
    "max_features": randint(1, X_train_resampled.shape[1]),
    "min_samples_leaf": randint(1, X_train_resampled.shape[0]),
    "criterion": ["gini", "entropy"]
}

# Create cross-validation strategy
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

# Perform hyperparameter tuning with RandomizedSearchCV
DT = RandomizedSearchCV(DT_model, param_dist, n_iter=50, scoring='roc_auc', n_jobs=-1, cv=cv, random_state=1)
DT.fit(X_train_resampled, y_train_resampled)
```



```
# Get the best estimator from RandomizedSearchCV
best_DT = DT.best_estimator_

# Train the best Decision Tree model on the training data
best_DT.fit(X_train_resampled, y_train_resampled)

# Predict on the test data
pred = best_DT.predict(X_test)
pred_prob = best_DT.predict_proba(X_test)

#Evaluate results
acc = accuracy_score(y_test, pred )
prec = precision_score(y_test, pred )
rec = recall_score(y_test, pred )
f1 = f1_score(y_test, pred )

#Save model performance to model_list
model_list.append(best_DT.__class__.__name__)
accuracy_list.append(acc)
precision_list.append(prec)
recall_list.append(rec)
f1_Score_list.append(f1)

# Display classification report
print(classification_report(y_test, pred))
```

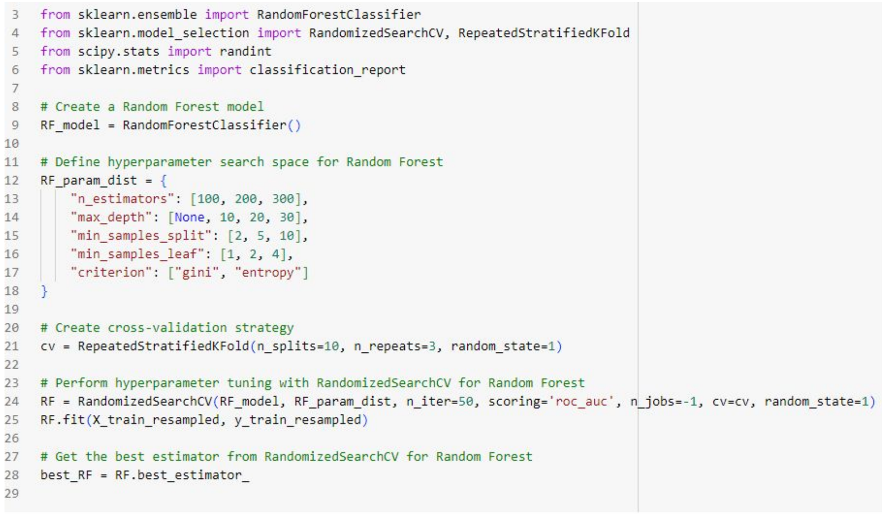


```
              precision    recall  f1-score   support

         0.0       0.63      0.53      0.58       400
         1.0       0.41      0.52      0.46       256

    accuracy                           0.52       656
   macro avg       0.52      0.52      0.52       656
weighted avg       0.55      0.52      0.53       656
```

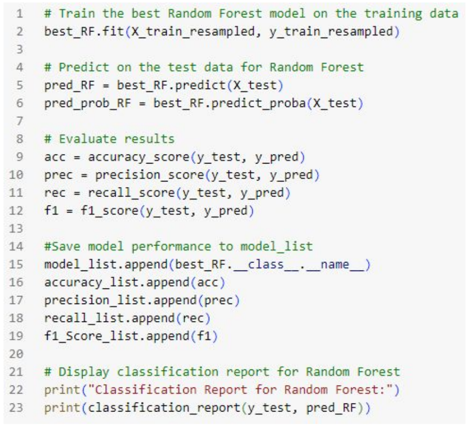
Model

Random Forest :

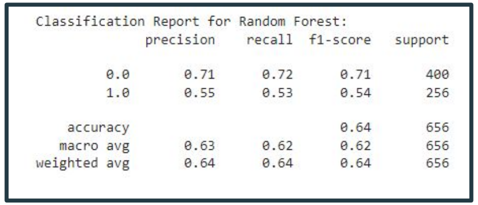
Random Forests are made out of decision trees composed of different bootstrapped data.



```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV, RepeatedStratifiedKFold
from scipy.stats import randint
from sklearn.metrics import classification_report

# Create a Random Forest model
RF_model = RandomForestClassifier()

# Define hyperparameter search space for Random Forest
RF_param_dist = {
    "n_estimators": [100, 200, 300],
    "max_depth": [None, 10, 20, 30],
    "min_samples_split": [2, 5, 10],
    "min_samples_leaf": [1, 2, 4],
    "criterion": ["gini", "entropy"]
}

# Create cross-validation strategy
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

# Perform hyperparameter tuning with RandomizedSearchCV for Random Forest
RF = RandomizedSearchCV(RF_model, RF_param_dist, n_iter=50, scoring='roc_auc', n_jobs=-1, cv=cv, random_state=1)
RF.fit(X_train_resampled, y_train_resampled)

# Get the best estimator from RandomizedSearchCV for Random Forest
best_RF = RF.best_estimator_

```



```python
# Train the best Random Forest model on the training data
best_RF.fit(X_train_resampled, y_train_resampled)

# Predict on the test data for Random Forest
pred_RF = best_RF.predict(X_test)
pred_prob_RF = best_RF.predict_proba(X_test)

# Evaluate results
acc = accuracy_score(y_test, y_pred)
prec = precision_score(y_test, y_pred)
rec = recall_score(y_test, y_pred)
f1 = f1_score(y_test, y_pred)

#Save model performance to model_list
model_list.append(best_RF.__class__.__name__)
accuracy_list.append(acc)
precision_list.append(prec)
recall_list.append(rec)
f1_Score_list.append(f1)

# Display classification report for Random Forest
print("Classification Report for Random Forest:")
print(classification_report(y_test, pred_RF))
```

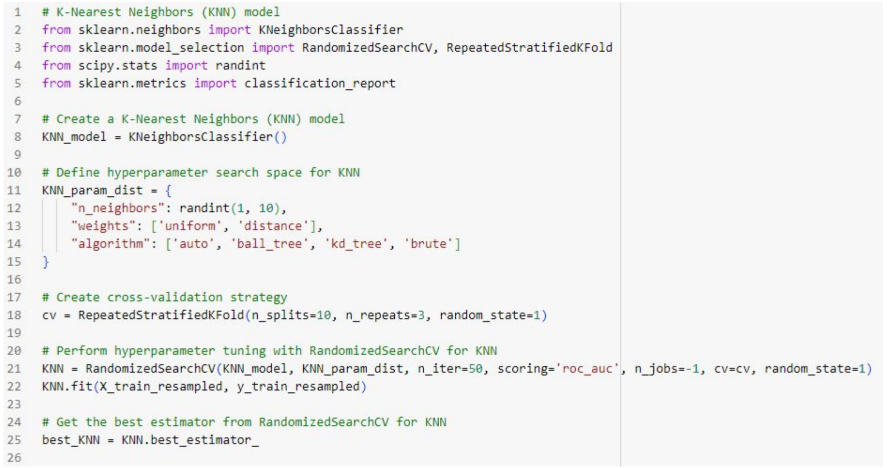


```
Classification Report for Random Forest:
              precision    recall  f1-score   support

         0.0       0.71      0.72      0.71       400
         1.0       0.55      0.53      0.54       256

    accuracy                           0.64       656
   macro avg       0.63      0.62      0.62       656
weighted avg       0.64      0.64      0.64       656
```

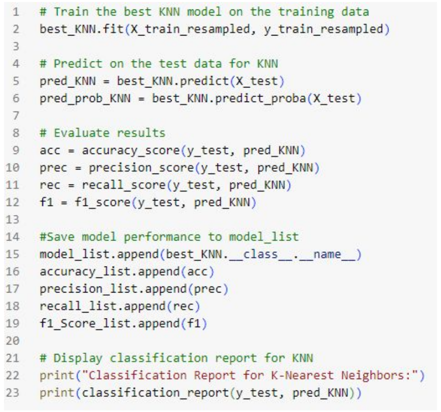
Model

K-Nearest Neighbour :

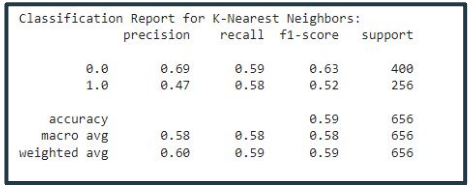
K-NN makes predictions by finding the **K-nearest data** points in the training dataset to a given data point and then classifying or regressing based on the majority or average of the neighbors' labels or values.



```python
# K-Nearest Neighbors (KNN) model
from sklearn.neighbors import KNeighborsClassifier
from sklearn.model_selection import RandomizedSearchCV, RepeatedStratifiedKFold
from scipy.stats import randint
from sklearn.metrics import classification_report

# Create a K-Nearest Neighbors (KNN) model
KNN_model = KNeighborsClassifier()

# Define hyperparameter search space for KNN
KNN_param_dist = {
    "n_neighbors": randint(1, 10),
    "weights": ['uniform', 'distance'],
    "algorithm": ['auto', 'ball_tree', 'kd_tree', 'brute']
}

# Create cross-validation strategy
cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)

# Perform hyperparameter tuning with RandomizedSearchCV for KNN
KNN = RandomizedSearchCV(KNN_model, KNN_param_dist, n_iter=50, scoring='roc_auc', n_jobs=-1, cv=cv, random_state=1)
KNN.fit(X_train_resampled, y_train_resampled)

# Get the best estimator from RandomizedSearchCV for KNN
best_KNN = KNN.best_estimator_

```



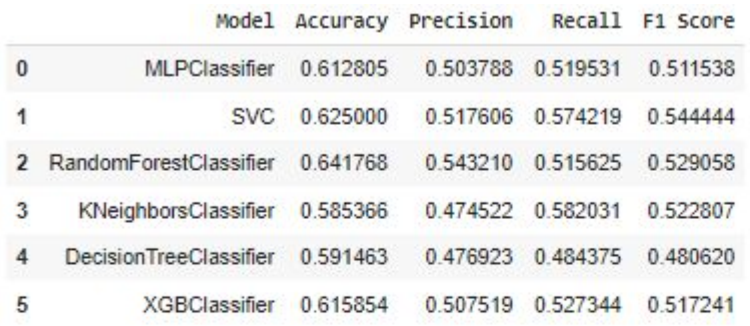
```python
# Train the best KNN model on the training data
best_KNN.fit(X_train_resampled, y_train_resampled)

# Predict on the test data for KNN
pred_KNN = best_KNN.predict(X_test)
pred_prob_KNN = best_KNN.predict_proba(X_test)

# Evaluate results
acc = accuracy_score(y_test, pred_KNN)
prec = precision_score(y_test, pred_KNN)
rec = recall_score(y_test, pred_KNN)
f1 = f1_score(y_test, pred_KNN)

#Save model performance to model_list
model_list.append(best_KNN.__class__.__name__)
accuracy_list.append(acc)
precision_list.append(prec)
recall_list.append(rec)
f1_Score_list.append(f1)

# Display classification report for KNN
print("Classification Report for K-Nearest Neighbors:")
print(classification_report(y_test, pred_KNN))
```



```
Classification Report for K-Nearest Neighbors:
              precision    recall  f1-score   support

         0.0       0.69      0.59      0.63       400
         1.0       0.47      0.58      0.52       256

    accuracy                           0.59       656
   macro avg       0.58      0.58      0.58       656
weighted avg       0.60      0.59      0.59       656
```

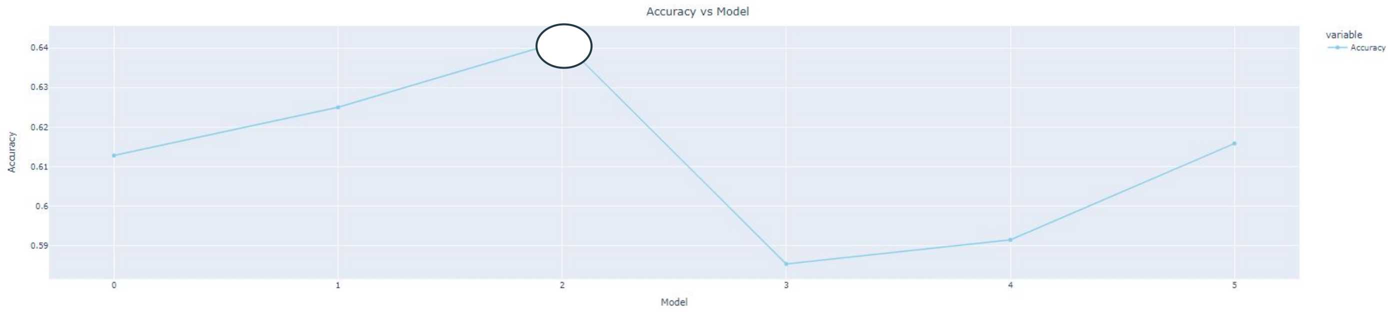
Model Evaluation

Accuracy Model



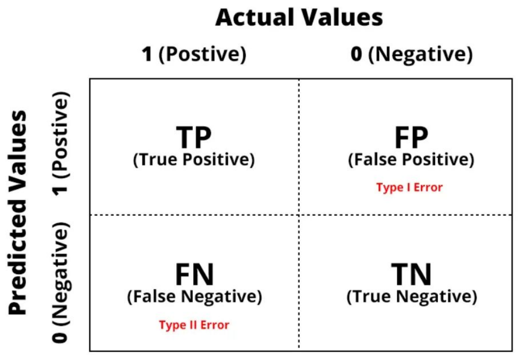
	Model	Accuracy	Precision	Recall	F1 Score
0	MLPClassifier	0.612805	0.503788	0.519531	0.511538
1	SVC	0.625000	0.517606	0.574219	0.544444
2	RandomForestClassifier	0.641768	0.543210	0.515625	0.529058
3	KNeighborsClassifier	0.585366	0.474522	0.582031	0.522807
4	DecisionTreeClassifier	0.591463	0.476923	0.484375	0.480620
5	XGBClassifier	0.615854	0.507519	0.527344	0.517241

"Random Forest Classifier (RF) with accuracy of 0.64 is the best one

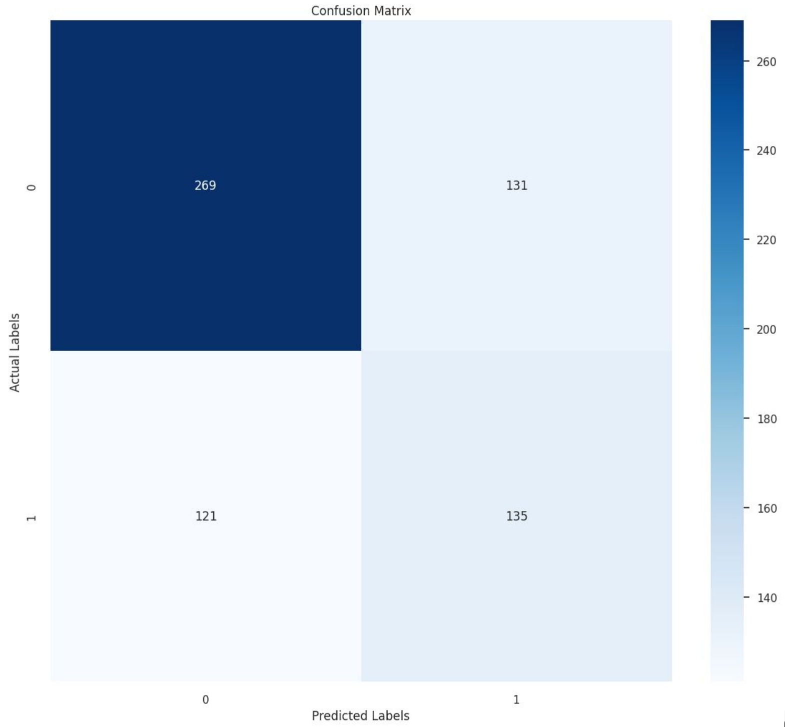


Model Evaluation

Confusion-Matrix

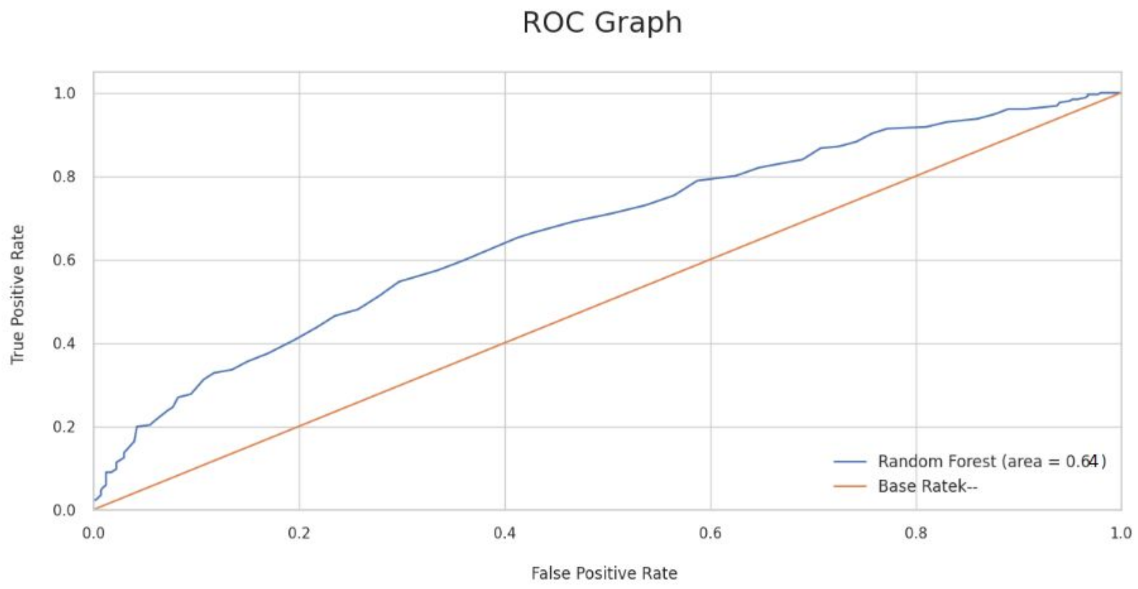


Confusion Matrix



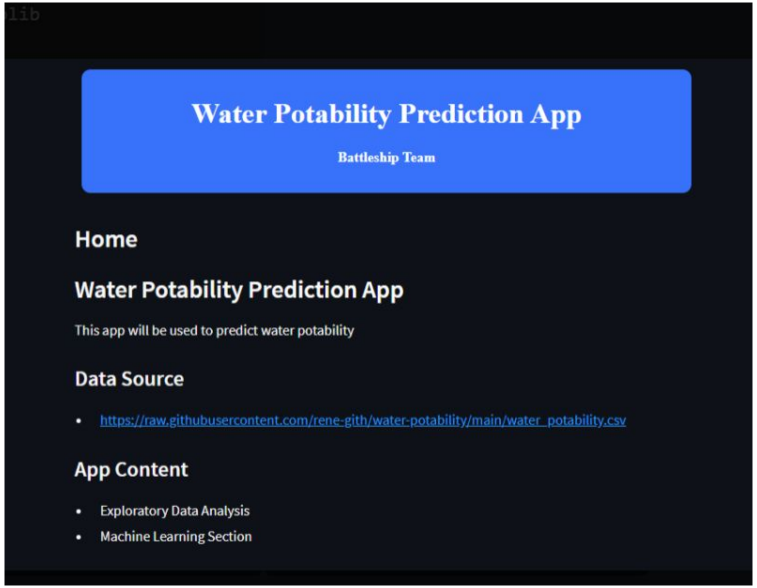
Model Evaluation

Evaluate the model using ROC Graph



Model Deployment

Streamlit



lib

Water Potability Prediction App

Battleship Team

Home

Water Potability Prediction App

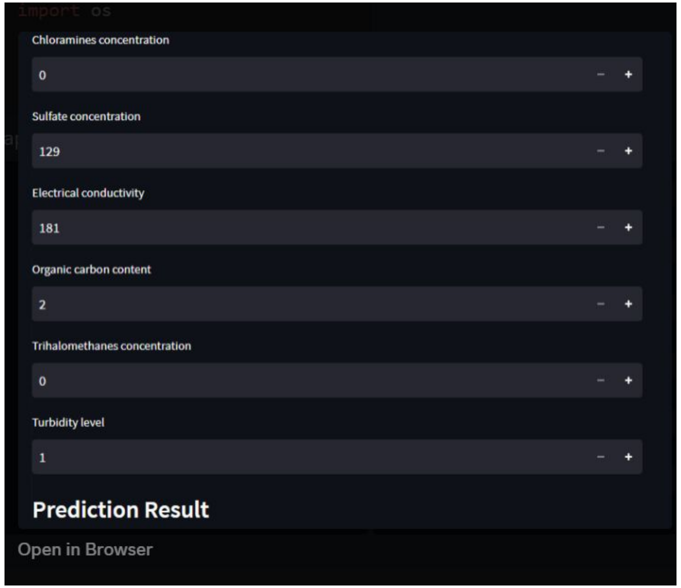
This app will be used to predict water potability

Data Source

- https://raw.githubusercontent.com/rene-gith/water-potability/main/water_potability.csv

App Content

- Exploratory Data Analysis
- Machine Learning Section



import os

Chloramines concentration

0

Sulfate concentration

129

Electrical conductivity

181

Organic carbon content

2

Trihalomethanes concentration

0

Turbidity level

1

Prediction Result

Open in Browser

C. Conclusion

"Random Forest Classifier (RF) with accuracy of 0.64 is the best one to other models in accuracy and robustness, making it the optimal choice for classification."