Project Title:

[Water Quality Prediction- Machine Learning]

Author: Sahriar Wahid Galib

Table of content

Introduction

Access to safe drinking water is both a fundamental human right and essential for safeguarding the health and well-being of communities. Be that as it may, water quality may differ significantly due to some factors, namely pH levels, hardness, solids, and also common contaminants like Chloramines & Sulfates that it contains. This 'Water Quality Prediction' project aims to predict water potability using a dataset of more than three thousand water bodies. Our goal is to determine whether water from a particular water body is drinkable or not by analyzing the parameters that determine the potability of water using machine learning algorithms.

In this project, we aim to train and test different data models to predict the potability of water with decent accuracy and make comparisons among various models for our dataset. Our methodology includes data analyzing, feature analysis, correlation analysis , data preprocessing e.g. imputing missing values, feature scaling, under sampling etc. Then we proceed to train and test our dataset using different machine learning models to predict water potability and make comparisons among several models to achieve better results with higher accuracy. The ultimate purpose of this project is to assist policymakers and water management entities in safeguarding potable water sources.

Dataset description

Source:

Google driveLink:

https://drive.google.com/file/d/1hxpOKcs2T1HeJcodQCahmPFKIvPNp_sV/view?usp=sharing

Kaggle Dataset Reference:

http://www.kaggle.com/datasets/adityakadiwal/water-potability

Description:

This dataset contains water quality metrics for 3276 different water bodies and has different parameters eg. ph, hardness, solids etc that determine the drinkability of water. In our dataset, "Potability" is the target variable. The value of the target variable indicates whether water from a particular water source is safe for drinking or not.

Importing Dataset:

The dataset was downloaded from Kaggle, it was saved in Google Drive. Subsequently, the CSV file was accessed and read from the specified path in Google Drive.

Importing Libraries

```
[694] import math
      import numpy as np
      import pandas as pd
      import seaborn as sns
      import matplotlib.pyplot as plt
      from sklearn.metrics import confusion_matrix, accuracy_score, classification_report
```

Importing Dataset

```
[ ] water_data= pd.read_csv('/content/422-Project/water_potability.csv')
    water_data.head()
```

	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

Features:

This dataset contains 10 features. 'ph', 'Hardness', 'Solids', 'Chloramines', 'Sulfate', 'Conductivity', 'Organic_carbon', 'Trihalomethanes', 'Turbidity' and 'Potability'. Nine of them are parameters that determine the target variable and ‘Potability’ is the target variable.

```
features= water_data.columns
print("Features of the dataset: ",'\n', features)
```

```
Features of the dataset:
 Index(['ph', 'Hardness', 'Solids', 'Chloramines', 'Sulfate', 'Conductivity',
       'Organic_carbon', 'Trihalomethanes', 'Turbidity', 'Potability'],
      dtype='object')
```

Problem type:

This is a classification problem because the target variable "potability" has values of either 0 or 1, which makes this problem a binary classification problem. In this case the goal is to classify instances into one of two classes based on input features. In this case, the classes are "potable" (1) and "not potable" (0).

```
[ ] target_variable= water_data['Potability']
    print("Target variable data type:")
    print(target_variable)
```

```
Target variable data type:
0       0
1       0
2       0
3       0
4       0
       ..
3271    1
3272    1
3273    1
3274    1
3275    1
Name: Potability, Length: 3276, dtype: int64
```


Datapoints:

This dataset has 3276 data points because it has 3276 rows and 10 columns. The shape of the data-frame is (3276, 10) which means that the dataset has 3276 data points and 10 features.

```
rows_and_columns= water_data.shape
print("Shape of Data: ", rows_and_columns)
print("Number of rows (Datapoints): ", rows_and_columns[0])
print("Number of colums (Features): ", rows_and_columns[1])
```

```
Shape of Data:  (3276, 10)
Number of rows (Datapoints):  3276
Number of colums (Features):  10
```

Features Type:

The dataset has 10 features. Nine of them are ‘float’ type data and one of them is ‘int’ type data. Therefore all the features are Quantitative. No categorical feature present in this dataset.

```
print("Feature Data Types: ")
feature_data_types= water_data.info()
```

```
Feature Data Types:
<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
```

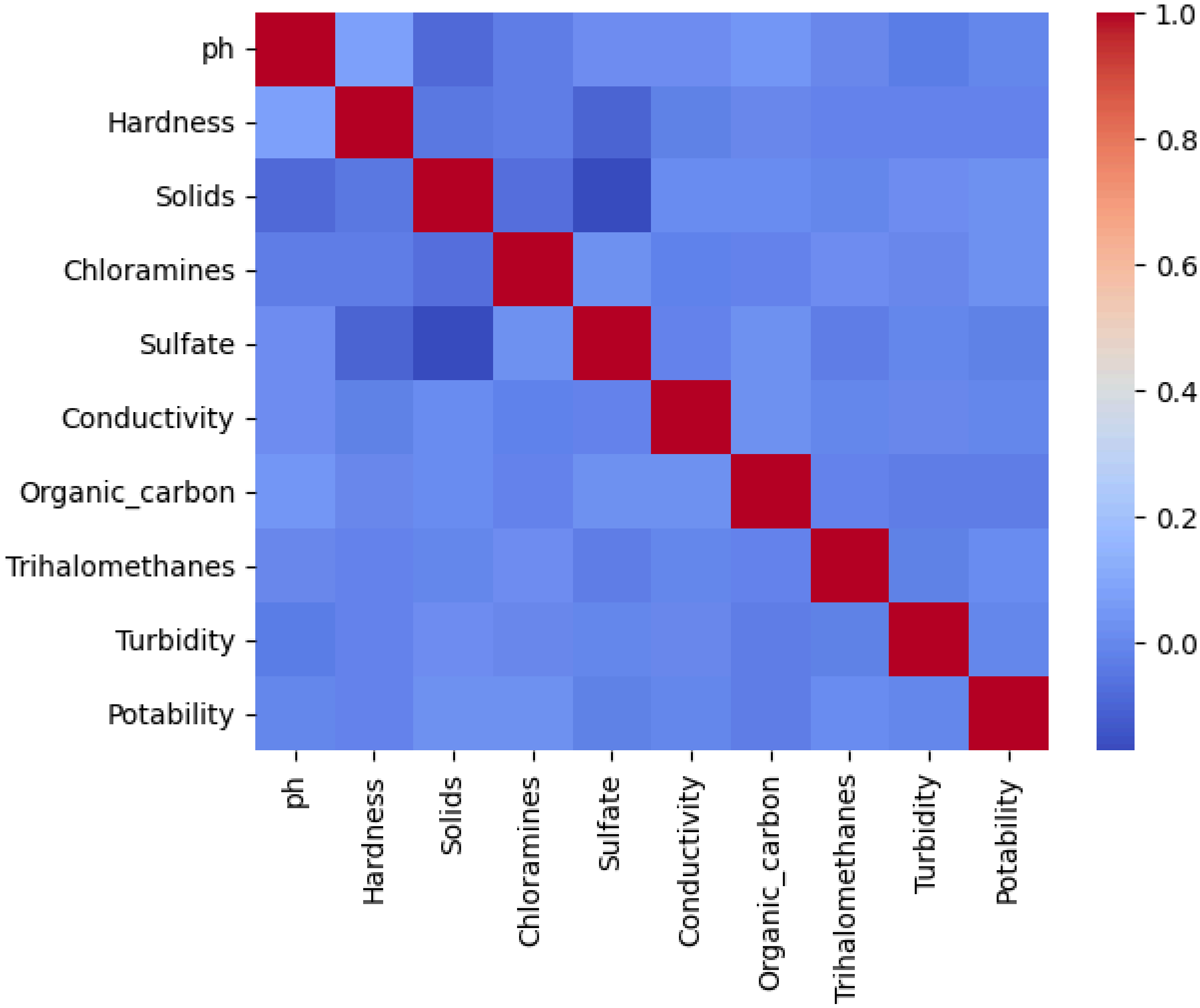

Correlation of the Features:

The correlation among the features is illustrated in the table below. Additionally, utilizing the Seaborn library to generate a heatmap allows us to implement a visual representation of the correlation between all the features, making it easier to interpret the relationships between them.

```
corr_of_features= water_data.corr()
corr_of_features
```

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity	Potability
ph	1.000000	0.082096	-0.089288	-0.034350	0.018203	0.018614	0.043503	0.003354	-0.039057	-0.003556
Hardness	0.082096	1.000000	-0.046899	-0.030054	-0.106923	-0.023915	0.003610	-0.013013	-0.014449	-0.013837
Solids	-0.089288	-0.046899	1.000000	-0.070148	-0.171804	0.013831	0.010242	-0.009143	0.019546	0.033743
Chloramines	-0.034350	-0.030054	-0.070148	1.000000	0.027244	-0.020486	-0.012653	0.017084	0.002363	0.023779
Sulfate	0.018203	-0.106923	-0.171804	0.027244	1.000000	-0.016121	0.030831	-0.030274	-0.011187	-0.023577
Conductivity	0.018614	-0.023915	0.013831	-0.020486	-0.016121	1.000000	0.020966	0.001285	0.005798	-0.008128
Organic_carbon	0.043503	0.003610	0.010242	-0.012653	0.030831	0.020966	1.000000	-0.013274	-0.027308	-0.030001
Trihalomethanes	0.003354	-0.013013	-0.009143	0.017084	-0.030274	0.001285	-0.013274	1.000000	-0.022145	0.007130
Turbidity	-0.039057	-0.014449	0.019546	0.002363	-0.011187	0.005798	-0.027308	-0.022145	1.000000	0.001581
Potability	-0.003556	-0.013837	0.033743	0.023779	-0.023577	-0.008128	-0.030001	0.007130	0.001581	1.000000

```
import seaborn as sns
sns.heatmap(corr_of_features, cmap='coolwarm')
```



Imbalance of the Dataset:

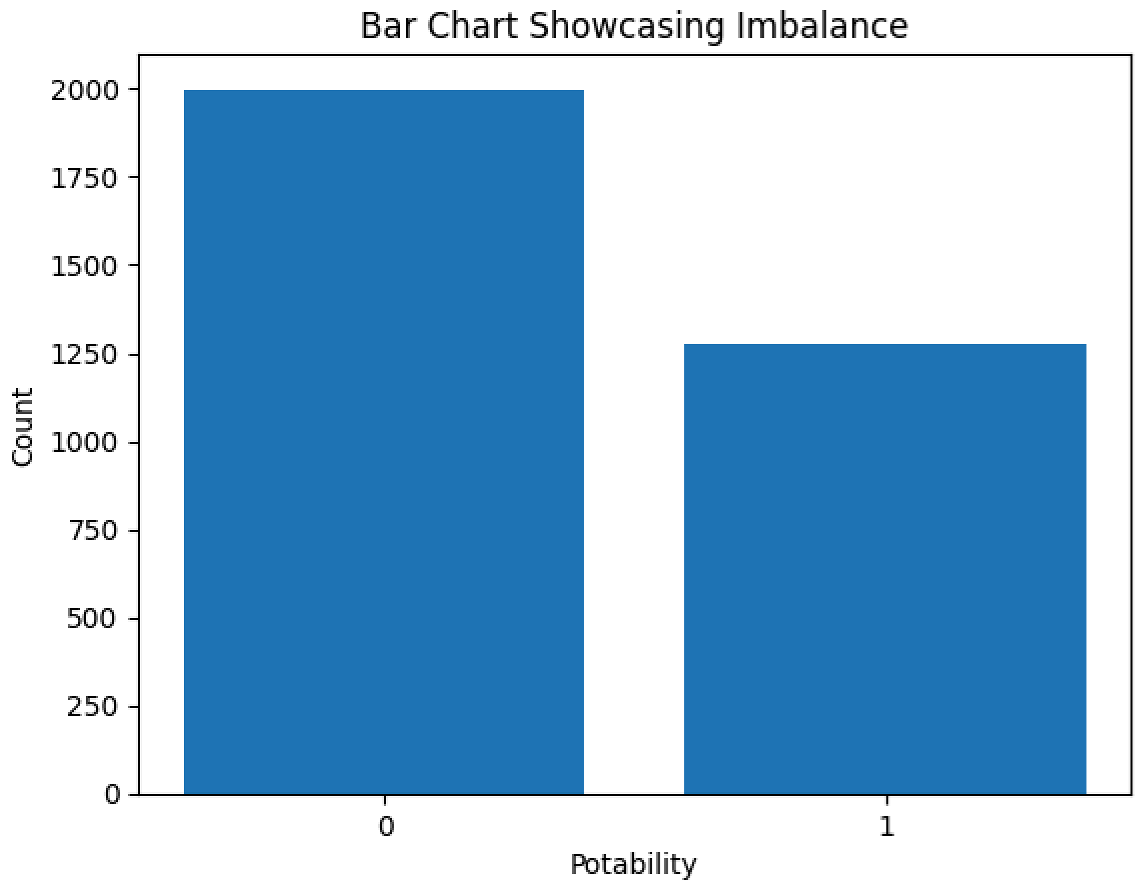
In our dataset, for the target variable 'Potability' there are two values '0' and '1' indicating not drinkable and drinkable respectively. We can see from the number of instances for both outputs that we have 1278 data points for output '1' and 1998 data points for output '0'. So, our dataset is clearly not balanced. This fact is also visible in the bar chart.

Checking Imbalanced Dataset

```
Drinkable_Water = water_data[water_data["Potability"]==1]

Not_Drinkable_Water = water_data[water_data["Potability"]==0]

print('Rows with Drinkable_Water: ',Drinkable_Water.shape[0])
print('Rows with Not_Drinkable_Water: ',Not_Drinkable_Water.shape[0])
```

```
Rows with Drinkable_Water:  1278
Rows with Not_Drinkable_Water:  1998
```

```
potability_counts = water_data['Potability'].value_counts()

x_values = [0, 1]
y_values = [potability_counts[0], potability_counts[1]]
plt.xticks([0, 1])

plt.bar(x_values, y_values)
plt.xlabel('Potability')
plt.ylabel('Count')
plt.title('Bar Chart Showcasing Imbalance')

plt.show()
```


Bar Chart Showcasing Imbalance
2000
1750
1500
1250
1000
750
500
250
0
Count
0
1
Potability

Dataset pre-processing

Faults:

Null Values:

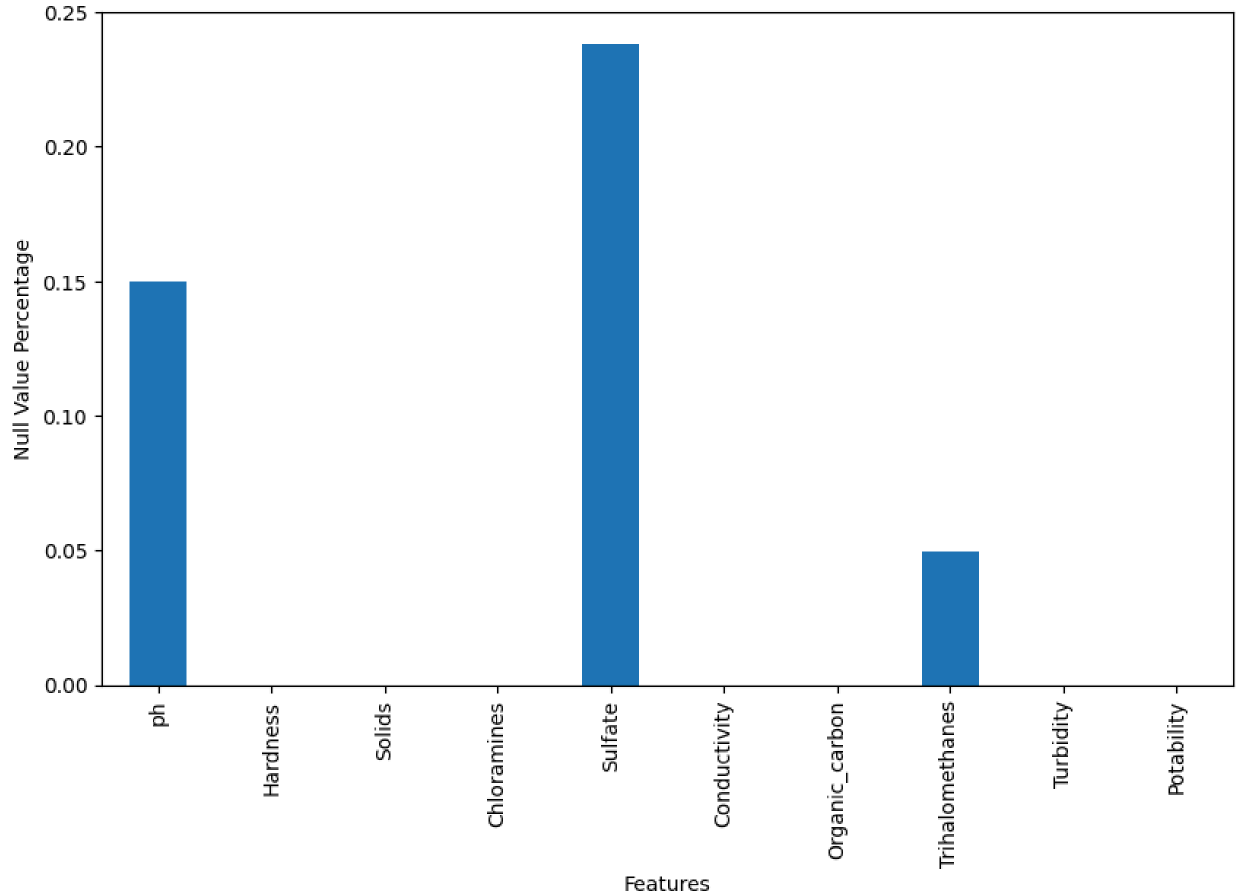
We have lots of null values in our dataset. The features which contain null values are ‘ph’, ‘Sulfate’ and ‘Trihalomethanes’. In the following bar chart we can see what are the percentages of null values and which of the mentioned features have the highest percentage of null values.

Checking Null Values

```
[ ] check_null= water_data.isnull().sum()
    check_null
```

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

```
water_data.isnull().mean().plot.bar(figsize=(10,6))
plt.xlabel("Features")
plt.ylabel('Null Value Percentage')
```



Solution:

Imputing mean values

A solution for the null values can be dropping rows and columns but proceeding with this method will reduce our data points which will affect the accuracy of our prediction. Therefore, we solved the null value problem by imputing mean values for the null values of the features.

Impute Mean Values For Null Values

```
fill_values = {
    "ph": water_data["ph"].mean(),
    "Sulfate": water_data["Sulfate"].mean(),
    "Trihalomethanes": water_data["Trihalomethanes"].mean()}

water_data.fillna(value=fill_values, inplace=True)
```

Result after Imputing Mean Values

```
[ ] check_null_new=water_data.isnull().sum()
    check_null_new

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

Balancing Dataset

Since our dataset is not balanced, we can balance out the dataset using under-sampling which will make sure we have an equal number of instances for both outputs.

Before Under Sampling:

```
print("Before Under Sampling: ")
print("Features shape: ", x.shape)
print("Target shape: ", y.shape)

Before Under Sampling:
Features shape:  (3276, 9)
Target shape:  (3276,)
```

After Under Sampling:

```
from imblearn.under_sampling import RandomUnderSampler
x, y = RandomUnderSampler().fit_resample(x, y)
print("After Under Sampling: ")
print("Features shape: ", x.shape)
print("Target shape: ", y.shape)

After Under Sampling:
Features shape:  (2556, 9)
Target shape:  (2556,)
```

Feature scaling

For the dataset we performed standard scaling, which minimizes discrepancies between data points that have widely different values. This method ensures that the data is uniformly scaled before going through the models, promoting better consistency and accuracy in the analysis.

```
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()
x = scaler.fit_transform(x)
x
```

```
array([[ 0.06753017,  0.14236981,  0.35354123, ..., -0.54098266,
        -2.70452761, -0.37097456],
       [-2.2075458 ,  1.11361106,  1.13750032, ..., -1.37156078,
        -0.16270276, -0.06663323],
       [ 0.7317972 ,  0.763544  ,  0.87242489, ..., -1.04702343,
        -0.42749039,  0.96119566],
       ...,
       [ 1.61797391, -0.61832789,  1.25216981, ..., -0.95679948,
         0.2192826 , -0.84072756],
       [-1.36056829,  1.029506  , -1.14969129, ..., -0.91790422,
         0.6976965 ,  0.96605619],
       [ 0.54608041, -0.03722124, -0.53477512, ...,  0.57093074,
         0.77345295, -2.10916512]])
```

Dataset splitting

Splitting Train and Test Data:

We divided our dataset into training and testing subsets, with 70% allocated for training and 30% for testing purposes.

Train and Test data split

```
[713] from sklearn.model_selection import train_test_split
      x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.3, random_state = 0)
      print("Train Data Shape: ",x_train.shape)
      print("Test Data Shape: ",x_test.shape)
```

```
Train Data Shape:  (1789, 9)
Test Data Shape:  (767, 9)
```

Model training & testing

We have applied 4 models for the training and testing purposes for our dataset. These models are:

- Logistic regression
- Naive Bayes
- KNN
- Support Vector Machine

Logistic regression

Logistic regression is a statistical technique utilized to determine the likelihood of a binary result for a target variable by considering predictor variables or features. It is used for classification problems where the output variable is binary. This means the target variable can only have two possible values, often represented as 0 and 1. In our case, the target variable 'Potability' belongs to this criteria. Therefore we used logistic regression.

Logistic Regression

```
[714] from sklearn.linear_model import LogisticRegression
      LR_model= LogisticRegression()
```

```
[715] LR_model.fit(x_train,y_train)
```

```
LogisticRegression
LogisticRegression()
```

```
[716] LR_pred = LR_model.predict(x_test)
      LR_pred
```

```
array([0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0,
```

Naive Bayes

Naive Bayes is a probabilistic machine learning model that is widely used for classification tasks. It makes a strong assumption of independence between features. It assumes that the presence of a particular feature in a class is unrelated to the presence of any other feature. This assumption simplifies the calculation of probabilities and makes the model computationally efficient. Naive Bayes calculates the probability of each class given the input features and then selects the class with the highest probability as the predicted class. We used this model for our classification problem.

Naive Bayes

```
[720] from sklearn.naive_bayes import GaussianNB
```

```
[721] NV_model = GaussianNB()
```

```
[722] NV_model.fit(x_train, y_train)
```

```
GaussianNB
GaussianNB()
```

```
[723] NV_pred= NV_model.predict(x_test)
      NV_pred
```

```
array([0, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0,
       0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 0,
```

KNN

K-Nearest Neighbors (KNN) is a machine learning algorithm used for classification and regression tasks. This model finds the class of a new data point by looking at the classes of its closest neighbors (nearest points) in the training data. It chooses the most common class among these neighbors as the prediction for the new point. Although KNN is computationally expensive for large datasets, it works well for comparatively small datasets. Since our dataset is relatively small it gives better results for our classification problem.

KNN Classifier

```
[727] from sklearn.neighbors import KNeighborsClassifier

[728] KN_model = KNeighborsClassifier(n_neighbors=11, leaf_size=20)

[729] KN_model.fit(x_train, y_train)

KNeighborsClassifier
KNeighborsClassifier(leaf_size=20, n_neighbors=11)

[730] KN_pred = KN_model.predict(x_test)
      KN_pred

array([0, 1, 1, 1, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0, 1, 1, 1, 0, 0, 1, 1,
       1, 0, 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 1,
```

Support Vector Machine

Support Vector Machine (SVM) is a powerful supervised learning algorithm used for both classification and regression tasks. SVM works by finding the best line or boundary to separate different groups of data points. It maximizes the space between the groups, called the margin, which helps it make accurate predictions. For our classification problem we used SVM and got higher accuracy than the other models.

Support Vector Machine

```
[734] from sklearn.svm import SVC
```

```
[735] SVM_model = SVC(kernel="rbf")
```

```
[736] SVM_model.fit(x_train, y_train)
```

```
SVC
SVC()
```

```
[737] SVM_pred = SVM_model.predict(x_test)
      SVM_pred
```

```
array([0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 1, 0, 1, 1, 0, 1, 0,
       0, 0, 0, 0, 1, 0, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 0,
```

Model Comparison analysis

Prediction accuracy Comparison for all models:

Logistic Regression (51%):

```
[717] LG_accuracy = math.ceil(accuracy_score(y_test, LR_pred)*100)
      print("Logistic Regression Model Accuracy:",LG_accuracy,"%")

      Logistic Regression Model Accuracy: 51 %
```

Naive Bayes (60%):

```
[724] NV_accuracy = math.ceil(accuracy_score(y_test, NV_pred)*100)
      print("Naive Bayes Model Accuracy:",NV_accuracy,"%")

      Naive Bayes Model Accuracy: 60 %
```

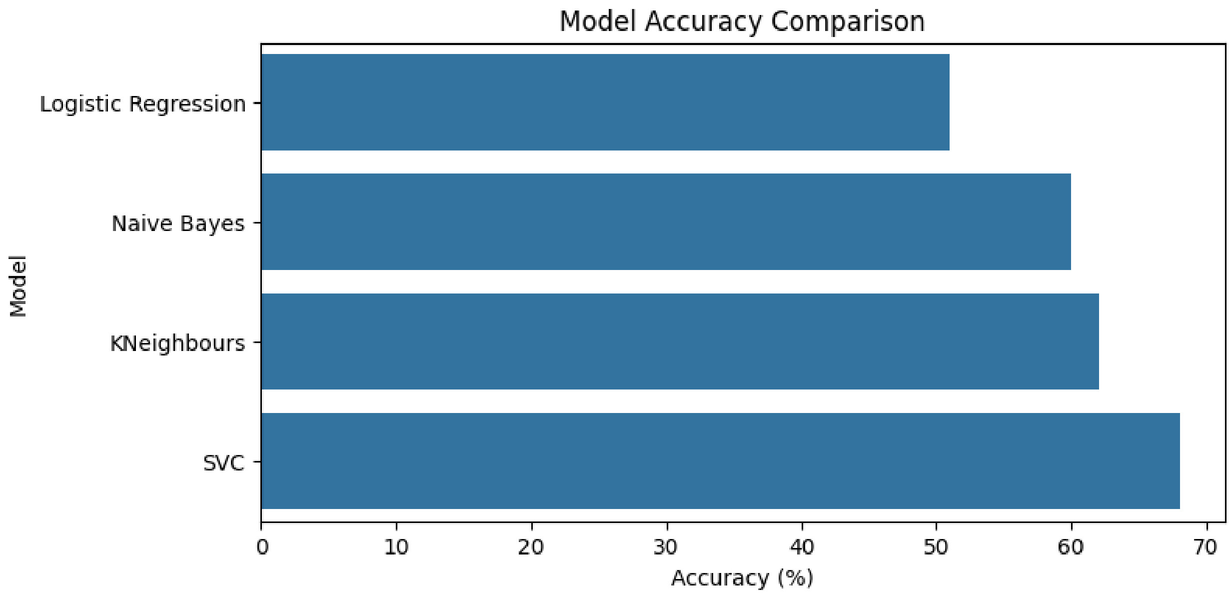
KNN (62%):

```
[731] KN_accuracy = math.ceil(accuracy_score(y_test, KN_pred)*100)
      print("KNN Model Accuracy:",KN_accuracy,"%")

      KNN Model Accuracy: 62 %
```

Support Vector Machine (68%):

```
[738] SVM_accuracy = math.ceil(accuracy_score(y_test, SVM_pred)*100)
      print("SVM Model Accuracy:",SVM_accuracy,"%")

      SVM Model Accuracy: 68 %
```

Bar Chart (Prediction Accuracy Comparison):



Among the four models we used for training and testing our dataset, SVC gives the best result with 68% accuracy. On the other hand, Logistic Regression provides the lowest accuracy of 51%. The accuracy of KNN model is 62% which is very close to Naive Bayes. Naive Bayes provides 60 % accuracy.

<u>Precision and recall comparison</u>

Logistic Regression:

Logistic regression shows relatively balanced precision and recall scores for both classes. However, the recall for class 0 is lower compared to class 1, indicating that the model might miss some instances of class 0.

```
[ ] print(classification_report(y_test,LR_pred))

              precision    recall  f1-score   support

           0       0.52      0.42      0.46       394
           1       0.49      0.58      0.53       373

    accuracy                           0.50       767
   macro avg       0.50      0.50      0.50       767
weighted avg       0.50      0.50      0.50       767
```

Naive Bayes:

Naive Bayes shows higher precision for class 1 and higher recall for class 0. This suggests that the model is better at correctly identifying instances of class 0 but struggles with precision for class 1.

```
[ ] print(classification_report(y_test,NV_pred))

              precision    recall  f1-score   support

           0       0.59      0.70      0.64       394
           1       0.61      0.49      0.54       373

    accuracy                           0.60       767
   macro avg       0.60      0.59      0.59       767
weighted avg       0.60      0.60      0.59       767
```

KNN:

KNN shows balanced precision and recall scores for both classes, indicating a relatively stable performance across both classes. However, the scores are not significantly higher compared to other classifiers.

```
print(classification_report(y_test,KN_pred))

              precision    recall  f1-score   support

           0       0.62      0.64      0.63       394
           1       0.61      0.60      0.60       373

    accuracy                           0.62       767
   macro avg       0.62      0.62      0.62       767
weighted avg       0.62      0.62      0.62       767
```

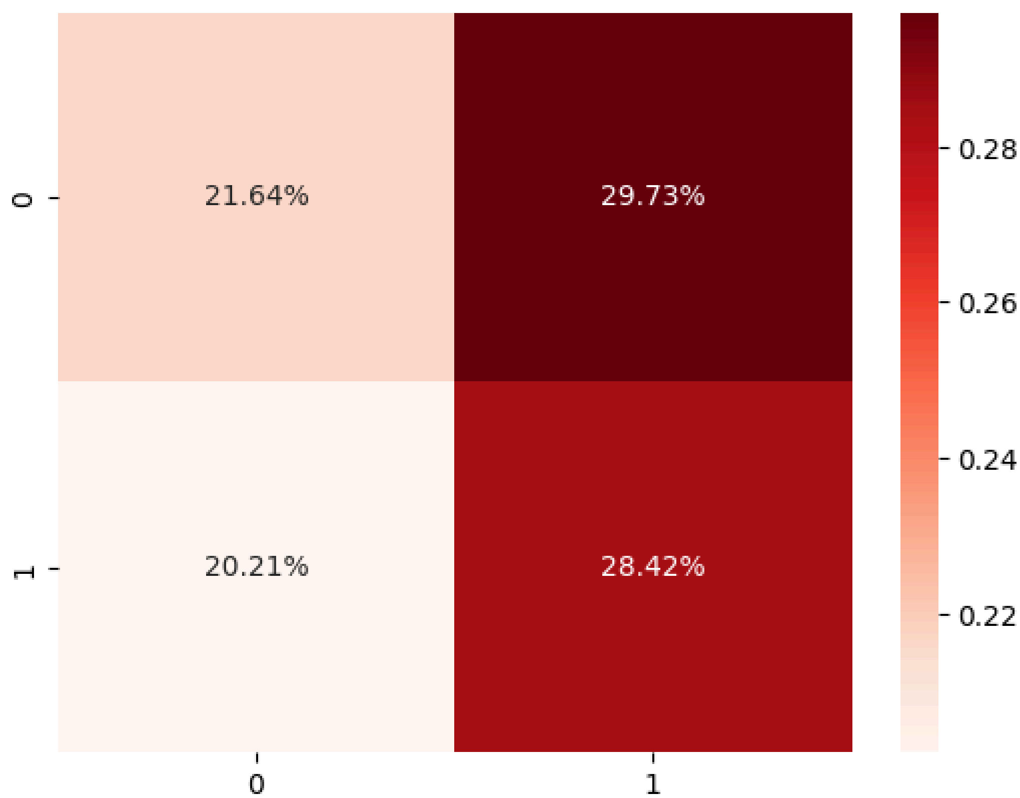
Support Vector Machine:

SVM shows higher precision and recall for both the classes than the other three models used. The model seems to be better at correctly identifying instances of both the classes with higher precision and recall scores. That's why this model has higher accuracy than the others.

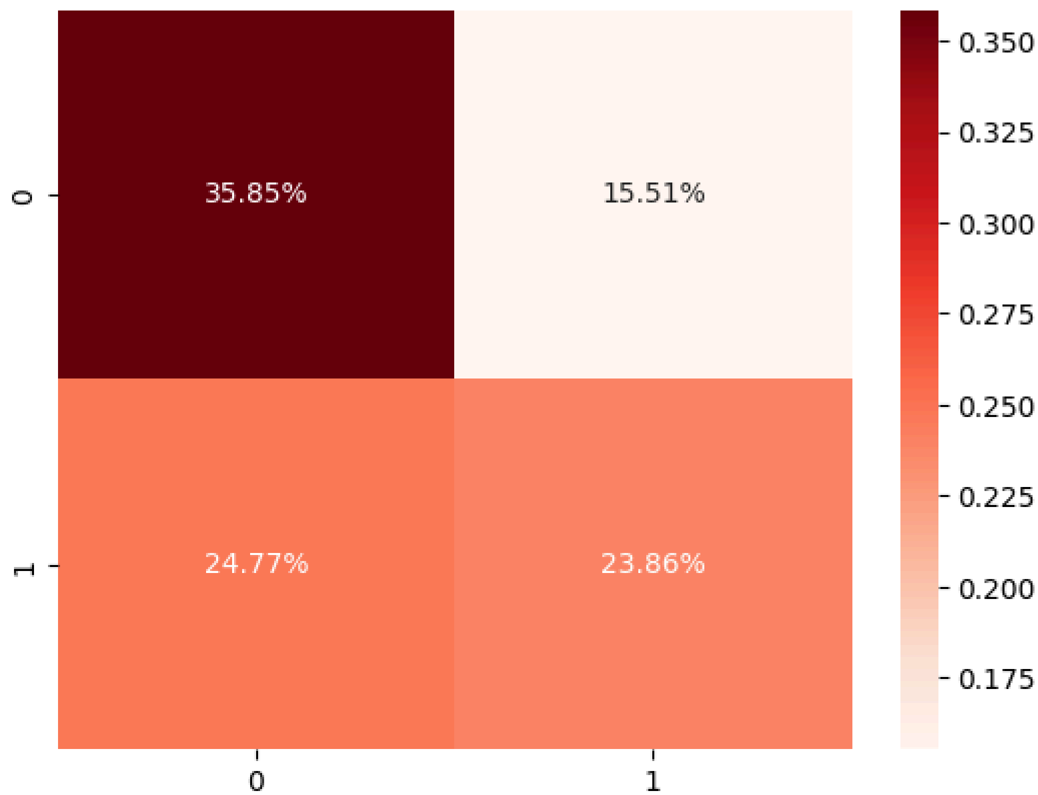
```
print(classification_report(y_test,SVM_pred))

              precision    recall  f1-score   support

           0       0.67      0.73      0.70       394
           1       0.69      0.62      0.65       373

    accuracy                           0.68       767
   macro avg       0.68      0.68      0.68       767
weighted avg       0.68      0.68      0.68       767
```

Confusion Matrix for each model

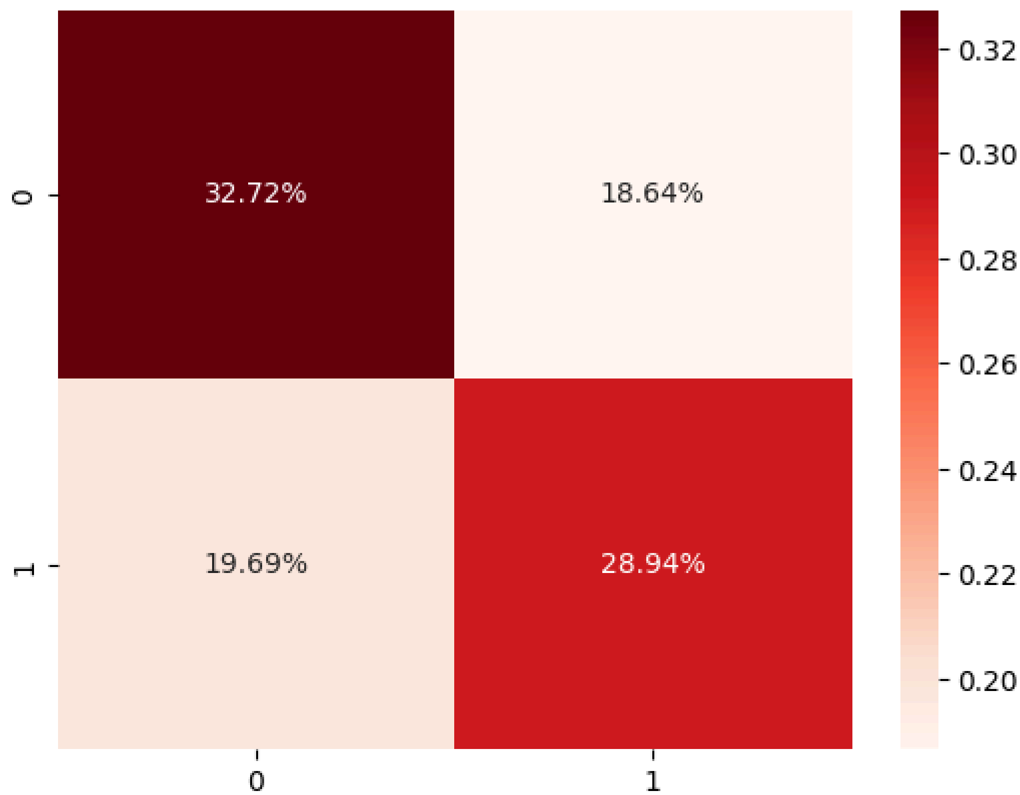
Logistic Regression:



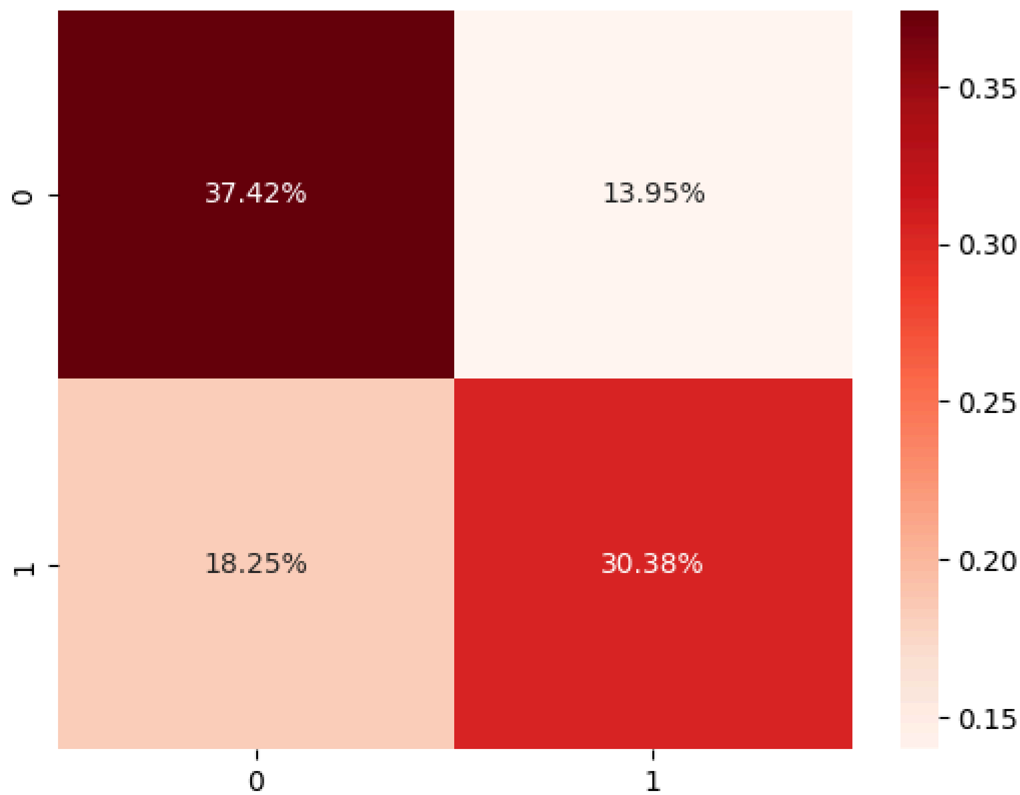
Naive Bayes:


35.85%
15.51%
24.77%
23.86%
0
1
0
1
0.350
0.325
0.300
0.275
0.250
0.225
0.200
0.175

KNN:



Support Vector Machine:



Conclusion

In conclusion, through extensive data and feature analysis, model training and testing we aimed to achieve accurate predictions and provide valuable insights in this 'Water Quality Prediction' project. The highest accuracy we achieved in terms of predicting the potability of water is 68%. That means we can correctly predict whether water from a source is drinkable or not 68% of the time. SVM model works best for our dataset which gave the highest accuracy among the four models we used. Logistic regression provided the lowest accuracy of 51%. Naive Bayes and KNN were close in terms of accuracy which is around 60%. Furthermore, we saw precision and recall comparison and confusion matrix for each model which gave us additional insights about their accuracy for our dataset.