Name = Harsh Kumar Jha

Batch = 12

Sap id = 500122473

Roll no. = R2142230376

Course = Elements of AI ML

ASSIGNMENT 1

Identify with an SDG close to your interest and identify a problem that ML / Data Science can solve. Follow the following steps to devise an ML-based solution:

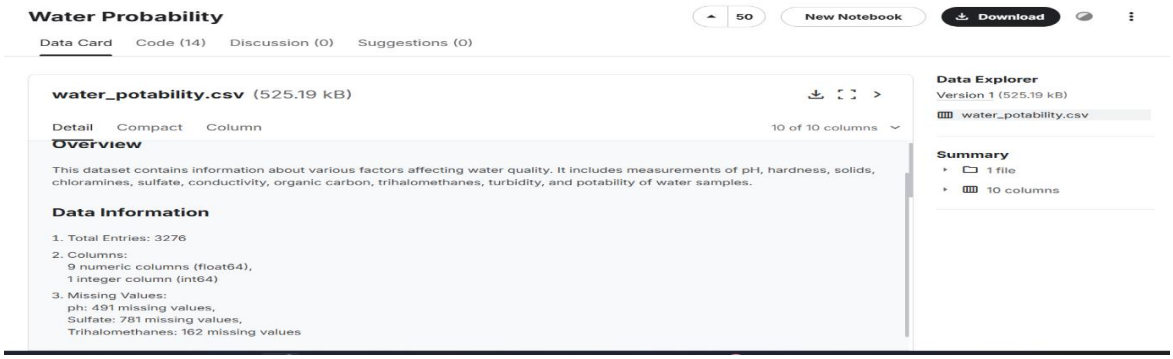
Step 1: Data Acquisition.

• Clean Water and Sanitation: Predicting water quality based on environmental and chemical factors helps prevent waterborne diseases and ensures safer access to drinking water.

Got this **Water Probability** dataset from Kaggle which has attributes related to water quality required for this project. **(water_potability.csv)**

The dataset consists of 3,276 entries and 10 columns, with the following features:

- ph: pH level of water (missing values)
- Hardness: Measure of water hardness
- Solids: Total dissolved solids in water
- Chloramines: Chloramines concentration in water
- Sulfate: Sulfate concentration (missing values)
- Conductivity: Water conductivity
- Organic_carbon: Organic carbon levels
- Trihalomethanes: Concentration of trihalomethanes (missing values)
- Turbidity: Water turbidity level
- Potability: Target variable indicating if water is potable (1) or not (0)



water_potability.csv

1 to 10 of 3276 entries Filter

ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity	Potability
	204.8904554713363	20791.318980747026	7.300211873184757	368.51644134980336	564.3086541722439	10.3797830780847	86.9909704615088	2.9631353806316407	0
3.71608007538699	129.42292051494425	18630.057857970347	6.635245883862		592.8853591348523	15.180013116357259	56.32907628451764	4.500656274942408	0
8.099124189298397	224.23625939355776	19909.541732292393	9.275883602694089		418.6062130644815	16.868636929550973	66.42009251176368	3.0559337496641685	0
8.316765884214679	214.37339408562252	22018.417440775294	8.05933237743854	356.88613564305666	363.2665161642437	18.436524495493302	100.34167436508008	4.628770536837084	0
9.092223456290965	181.10150923612525	17978.98633892625	6.546599974207941	310.13573752420444	398.41081338184466	11.558279443446395	31.997992727424737	4.075075425430034	0
5.584086638456089	188.3133237696164	28748.68773904612	7.54486878877965	326.6783629116736	280.4679159334877	8.399734640152758	54.917861841994466	2.5597082275565217	0
10.223862164528773	248.07173527013992	28749.716543528233	7.5134084658313025	393.66339551509645	283.6516335078445	13.789695317519886	84.60355617402357	2.672988736934779	0
8.635848718500734	203.36152258457054	13672.091763901635	4.563008685599703	303.3097711592812	474.60764494244853	12.36381669870525	62.798308962925155	4.401424715445482	0
	118.98857909025189	14285.583854224515	7.804173553073094	268.646940746221	389.3755658712614	12.70604896865791	53.928845767512236	3.5950171809576155	0
11.180284470721592	227.23146923797458	25484.50849098786	9.077200016914393	404.04163468408996	563.8854814810949	17.92780641128502	71.97660103221915	4.370561936655497	0

Show 10 per page

1 2 10 100 300 320 328

Step 2: Define the methodology and the objectives of your work.

Objectives:

1. Predict water potability (binary classification problem)
2. Create a reliable model with good performance metrics
3. Handle missing values and class imbalance effectively

Methodology:

1. Data preprocessing (handling missing values, scaling)
2. Address class imbalance using SMOTE
3. Evaluate model performance

Step 3: Data Preprocessing Preprocess or clean the dataset.

1. Importing all necessary library

IMPORTING THE LIBRARIES

```
[2] import pandas as pd
    import numpy as np
    from sklearn.model_selection import train_test_split, KFold, cross_val_score
    from sklearn.preprocessing import StandardScaler
    from sklearn.impute import SimpleImputer
    from sklearn.ensemble import RandomForestClassifier
    from sklearn.linear_model import LogisticRegression
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.svm import SVC
    from sklearn.neighbors import KNeighborsClassifier
    from imblearn.over_sampling import SMOTE
    from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
```

2. Loading the Dataset

IMPORTING THE DATASET

```
df = pd.read_csv('water_potability.csv')
df.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

Next steps: Generate code with df | View recommended plots | New interactive sheet

```
# number of rows and Columns in this dataset
df.shape
```

```
(3276, 10)
```

DROP THE DUPLICATE DATA

```
df.drop_duplicates()
```

	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

3. Getting the total no. of portable and nonportable rows

```
[7] df.groupby('Potability').mean()
```

Potability	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity
0	7.085378	196.733292	21777.490788	7.092175	334.56429	426.730454	14.364335	66.303555	3.965800
1	7.073783	195.800744	22383.991018	7.169338	332.56699	425.383800	14.160893	66.539684	3.968328

```
df['Potability'].value_counts()
```

Potability	count
0	1998
1	1278

dtype: int64

0 --> Non Potabable

1 --> Potabable

4. Handling Missing Values

HANDLE MISSING DATA

```
[9] #count the no of missing value
    print(df.isnull().sum())
```

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

```
[10] # Handle missing values using median imputation
     imputer = SimpleImputer(strategy='mean')
     df = pd.DataFrame(imputer.fit_transform(df), columns=df.columns)
```

```
print("Missing values after imputation:")
print(df.isnull().sum())
```

```
Missing values after imputation:
ph                0
Hardness          0
Solids            0
Chloramines       0
Sulfate           0
Conductivity      0
Organic_carbon    0
Trihalomethanes   0
Turbidity         0
Potability        0
dtype: int64
```

5. Encoding Categorical Variables

ENCODING CATEGORICAL DATA

```
from sklearn.preprocessing import LabelEncoder

# Check for categorical columns in the dataset
categorical_cols = df.select_dtypes(include=['object']).columns
print("Categorical columns:", categorical_cols)

# Apply Label Encoding or One-Hot Encoding
for col in categorical_cols:
    # Example with Label Encoding for binary or ordinal categorical features
    if df[col].nunique() <= 2:
        label_encoder = LabelEncoder()
        df[col] = label_encoder.fit_transform(df[col])
    else:
        # Example with One-Hot Encoding for non-binary features
        df = pd.get_dummies(df, columns=[col], prefix=col)

# Verify the result
df.head()
```

```
Categorical columns: Index([], dtype='object')
```

	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	0.0
1	3.716080	129.422921	18630.057858	6.635246	333.775777	592.885359	15.180013	56.329076	4.500656	0.0
2	8.099124	224.236259	19909.541732	9.275884	333.775777	418.606213	16.868637	66.420093	3.055934	0.0
3	8.316766	214.373394	22018.417441	8.059332	356.886136	363.266516	18.436524	100.341674	4.628771	0.0
4	9.092223	181.101509	17978.986339	6.546600	310.135738	398.410813	11.558279	31.997993	4.075075	0.0

Next steps: Generate code with df | View recommended plots | New interactive sheet

6. Check for class imbalance and feature scaling

```
[14] # Check for class imbalance
     print("Class distribution:\n", df['Potability'].value_counts())

     # Apply SMOTE if there is class imbalance
     X = df.drop(columns=['Potability'])
     y = df['Potability']
     if y.value_counts(normalize=True)[0] > 0.6:
         smote = SMOTE()
         X, y = smote.fit_resample(X, y)
         print("After applying SMOTE:\n", pd.Series(y).value_counts())
```

```
Class distribution:
 Potability
0.0    1998
1.0    1278
Name: count, dtype: int64
After applying SMOTE:
 Potability
0.0    1998
1.0    1998
Name: count, dtype: int64
```

```
[15] # Standardize the features
     scaler = StandardScaler()
     X = scaler.fit_transform(X)
```

7. Test and train spilt dataset

Train & Test Split data

```
[19] X_train_resampled, X_test, y_train_resampled, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

print(X_train_resampled.shape)
print(y_train_resampled.shape)
print(X_test.shape)
print(y_test.shape)

(3196, 9)
(3196,)
(800, 9)
(800,)
```

8. Dataset after data preprocessing

Suggested code may be subject to a license | 5DcOOKIE/FaceGenius

```
df.describe()
```

	ph	Hardness	Solids	Chloramines	Sulfate	Conductivity	Organic_carbon	Trihalomethanes	Turbidity	Potability
count	3276.000000	3276.000000	3276.000000	3276.000000	3276.000000	3276.000000	3276.000000	3276.000000	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	1.469956	32.879761	8768.570828	1.583085	36.142612	80.824064	3.308162	15.769881	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%	6.277673	176.850538	15666.690297	6.127421	317.094638	365.734414	12.065801	56.647656	3.439711	0.000000
50%	7.080795	196.967627	20927.833607	7.130299	333.775777	421.884968	14.218338	66.396293	3.955028	0.000000
75%	7.870050	216.667456	27332.762127	8.114887	350.385756	481.792304	16.557652	76.666609	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

```
[20] print(X)
```

Results:

- Clean dataset ready for modelling.
- Balanced class distribution
- Normalized feature scales
- Structured training and testing sets

This preprocessing ensures data quality and prepares the dataset for optimal model performance in predicting water potability.

Step 4: Use multiple ML methods and validate them using K-Fold Cross Validation.

Models to evaluate

```
[27] models = {
         "Logistic Regression": LogisticRegression(),
         "Decision Tree": DecisionTreeClassifier(),
         "Random Forest": RandomForestClassifier(),
         "Support Vector Machine": SVC(),
         "K-Nearest Neighbors": KNeighborsClassifier()
     }
```

```
#Use K-Fold Cross Validation to evaluate each model
kfold = KFold(n_splits=5, shuffle=True, random_state=42)
results = {}
```

```
for model_name, model in models.items():
    # Cross-validate the model
    cv_results = cross_val_score(model, X, y, cv=kfold, scoring='accuracy')
    results[model_name] = {
        "Accuracy Mean": np.mean(cv_results),
        "Accuracy Std": np.std(cv_results)
    }
    print(f"{model_name} - Accuracy: {np.mean(cv_results):.4f} ± {np.std(cv_results):.4f}")
```

```
# Training and evaluation on a holdout set
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
for model_name, model in models.items():
    # Train the model
    model.fit(X_train, y_train)
    # Make predictions
    y_pred = model.predict(X_test)
    # Evaluate the model
    print(f"\n{model_name} Evaluation:")
    print("Accuracy:", accuracy_score(y_test, y_pred))
    print("Precision:", precision_score(y_test, y_pred))
    print("Recall:", recall_score(y_test, y_pred))
    print("F1 Score:", f1_score(y_test, y_pred))
```

```
Logistic Regression - Accuracy: 0.5005 ± 0.0087
Decision Tree - Accuracy: 0.6134 ± 0.0190
Random Forest - Accuracy: 0.6922 ± 0.0123
Support Vector Machine - Accuracy: 0.6592 ± 0.0169
K-Nearest Neighbors - Accuracy: 0.6446 ± 0.0133

Logistic Regression Evaluation:
Accuracy: 0.515
Precision: 0.5252808988764045
Recall: 0.4605911330049261
F1 Score: 0.49081364829396323

Decision Tree Evaluation:
Accuracy: 0.61875
Precision: 0.6199524940617577
Recall: 0.6428571428571429
F1 Score: 0.6311970979443773

Random Forest Evaluation:
Accuracy: 0.68375
Precision: 0.6946564885496184
Recall: 0.6724137931034483
F1 Score: 0.6833541927409261

Support Vector Machine Evaluation:
Accuracy: 0.66125
Precision: 0.6573426573426573
Recall: 0.6945812807881774
F1 Score: 0.6754491017964072

K-Nearest Neighbors Evaluation:
Accuracy: 0.67
Precision: 0.6666666666666666
Recall: 0.6995073891625616
F1 Score: 0.6826923076923077
```

Step 5: Comparing the results using suitable performance metrics such as accuracy, precision, recall, f1-score, kappa statistics, AUC score, confusion matrix, etc. (for classification), mean squared error, mean absolute error, residuals, etc. (for regression), within-cluster sum of squares, silhouette score (for clustering).

Model	Accuracy	Precision	Recall	F1 Score
Logistic Regression	**0.515**	**0.525**	**0.460**	**0.490**
Decision Tree	**0.618**	**0.619**	**0.642**	**0.631**
Random Forest	**0.683**	**0.694**	**0.672**	**0.683**
Support Vector Machine	**0.661**	**0.657**	**0.694**	**0.675**
K-Nearest Neighbours	**0.67**	**0.666**	**0.699**	**0.682**

- **Random Forest** performed the best, with an accuracy of **68.3**% and balanced precision, recall, and F1 scores. This robust model handles the potability classification and any class imbalance effectively, making it a suitable choice for this dataset.
- **K- Nearest Neighbours** also performed well, with an accuracy of 67%, but was slightly less effective than Random Forest.
- **Logistic Regression** provided poor results with 51.5% accuracy, making it useful for simpler alternatives if computational efficiency is needed.

Conclusion:

For predicting water **potability**, the **Random Forest** model is recommended due to its high accuracy and balanced performance across evaluation metrics, suggesting reliable results for classifying potable and non-potable water.