Water Potability Classification

This project focuses on building a machine learning model to predict whether a water sample is potable (safe to drink) based on various chemical properties like pH, Hardness, Solids, Sulfate, and others.
It was developed as part of a hackathon, with strong emphasis on both model performance and complete data preprocessing and deployment.

---

1. Data Preprocessing & Cleaning

- **Missing Values**: Imputed missing values in key features like pH, Sulfate, and Trihalomethanes using SimpleImputer.
- **Feature Scaling**:
    - Applied **MinMaxScaler** to normalize features between 0 and 1 for better model stability.
- **Skewness Correction**:
    - Applied **log1p() transformation** on highly skewed features (Solids, Conductivity, Trihalomethanes) to reduce the impact of extreme values.
- **Outlier Treatment**:
    - Outliers were analyzed through boxplots and managed appropriately to improve feature quality.
- **Class Imbalance Handling**:
    - Initially found serious class imbalance:
        - Before SMOTE: {0: 1586, 1: 1034}
    - **How we solved it**:
        - Applied **SMOTE (Synthetic Minority Over-sampling Technique)** to create synthetic examples for the minority class (potable water).
        - After SMOTE: {0: 1586, 1: 1586} — now perfectly balanced.
    - **Why it was necessary**:
        - Without SMOTE, the model would heavily favor predicting non-potable water.
        - SMOTE helped the model learn both classes fairly, improving recall, F1-score, and real-world performance.

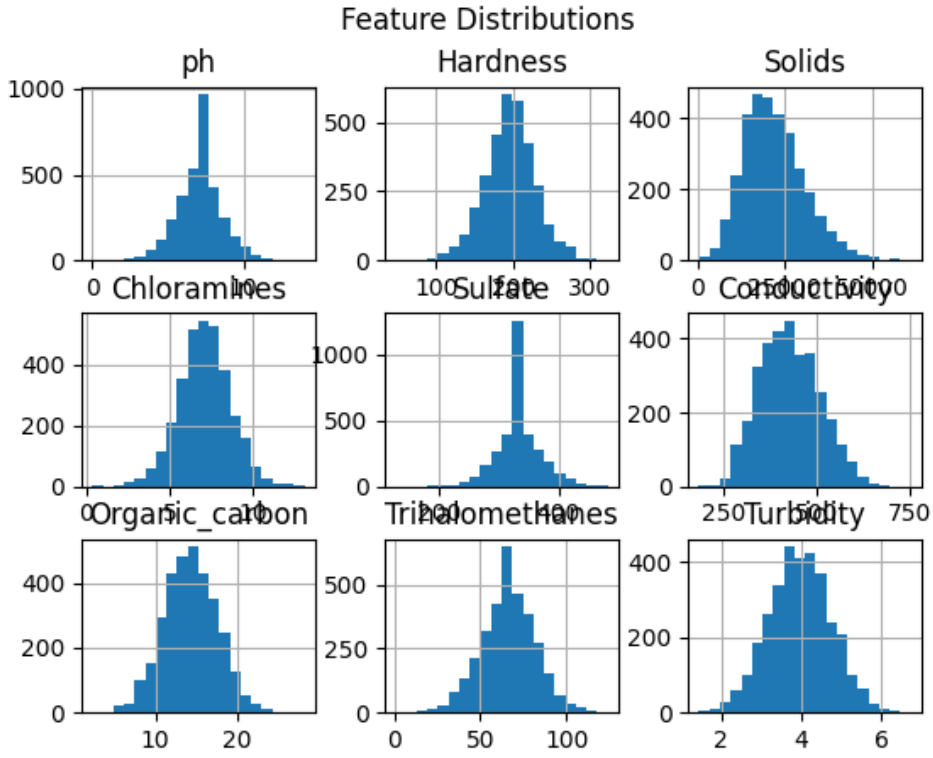
---

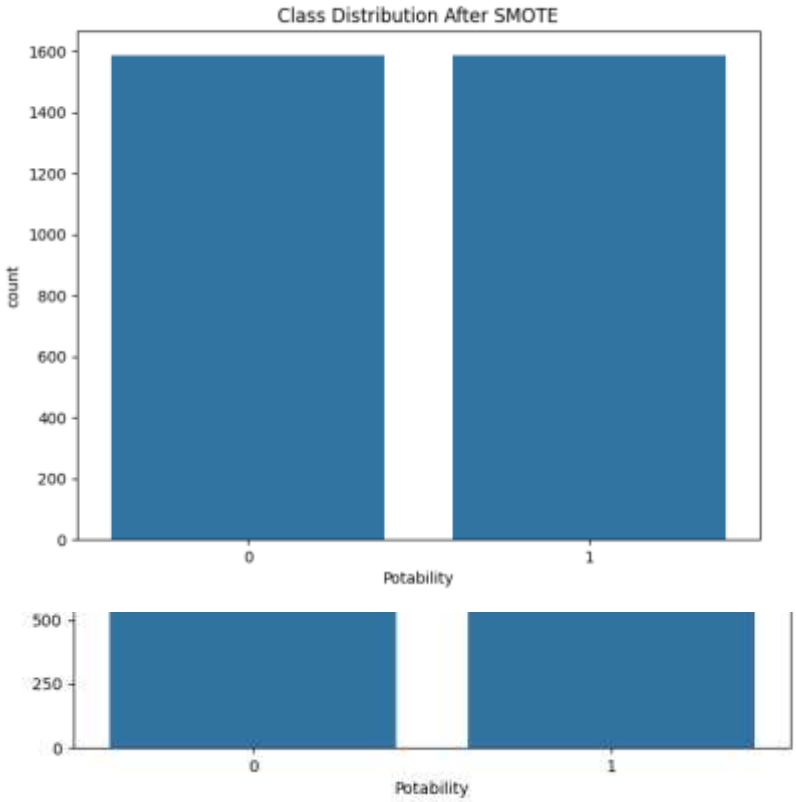
2. Exploratory Data Analysis (EDA)

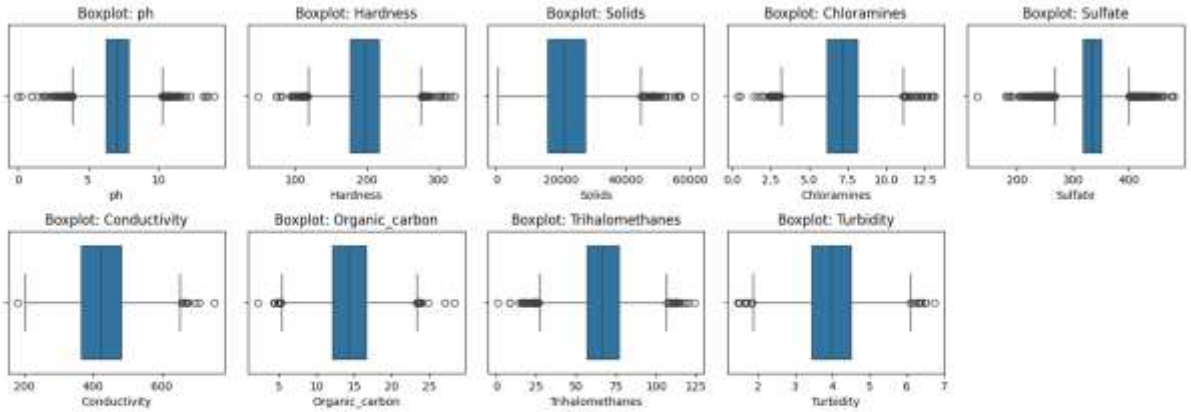
I thoroughly explored the dataset to gain meaningful insights:

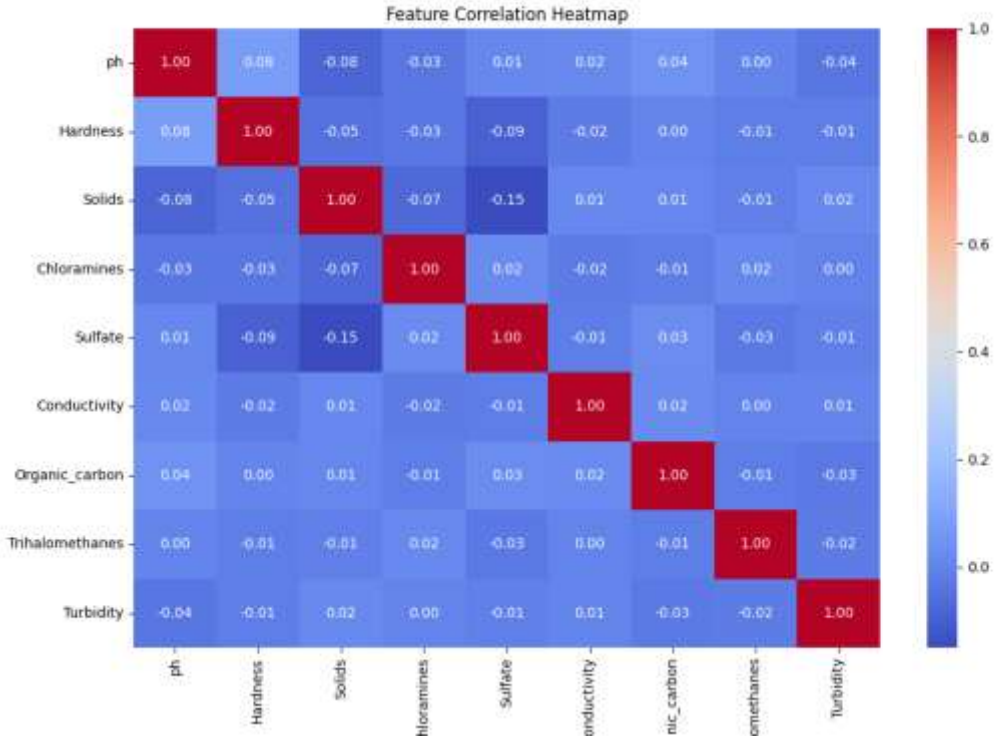
- **Histograms**: Visualized feature distributions.
- **Boxplots**: Detected and analyzed outliers.
- **Correlation Heatmap**: Identified relationships between features.
- **Countplots**: Revealed data imbalance between potable and non-potable samples.
- **Key Insights**:

- Solids and Conductivity showed high positive correlation.
- Turbidity had a slight negative correlation with the target class.
- Data imbalance was clearly visible, leading to the application of SMOTE.




Class Distribution After SMOTE
count
Potability
0
1
0
200
400
600
800
1000
1200
1400
1600
Potability
0
1
0
250
500





---

3. Model Training, Tuning & Evaluation

- **Model Used**: RandomForestClassifier
- **Hyperparameter Tuning**:
  - Applied GridSearchCV on:
    - n_estimators: [100, 200, 300]
    - max_depth: [None, 10, 20]

        - min_samples_split: [2, 5, 10]
    - **Best Parameters**:

      {'n_estimators': 300, 'max_depth': 10, 'min_samples_split': 10}

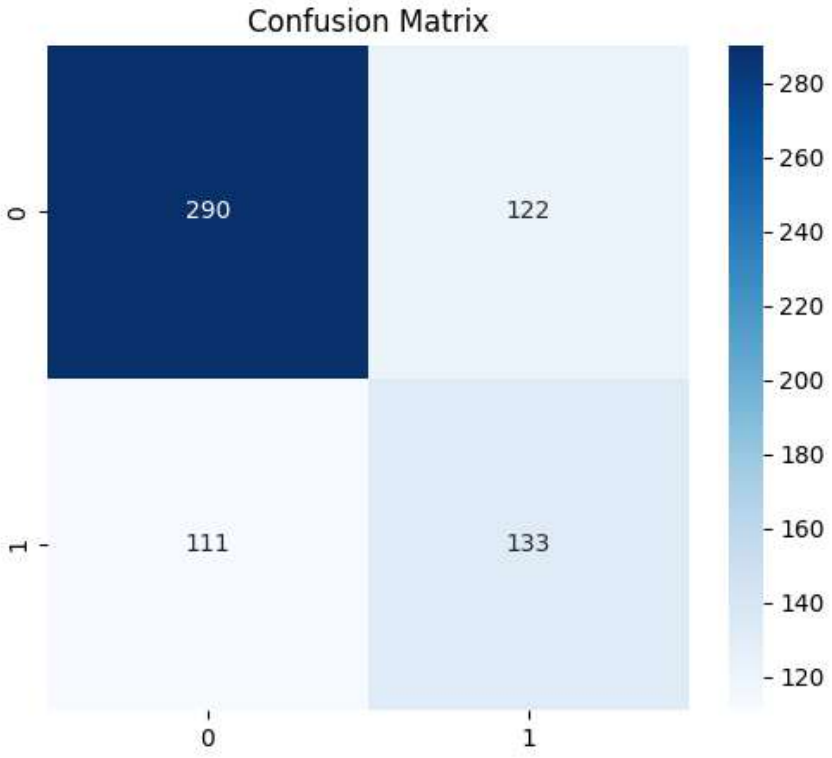
- **Training**:
    - Final model trained on the SMOTE-resampled dataset.
    - Saved using joblib as **final_trained_model.pkl**.
- **Evaluation Metrics (on Test Data)**:
    - **Classification Report**:

	precision	recall	f1-score	support
0	0.72	0.70	0.71	412
1	0.52	0.55	0.53	244
macro avg	0.62	0.62	0.62	656

weighted avg 0.65 0.64 0.65 656

- **Confusion Matrix**

Analyzed for false positives and false negatives.



Final Evaluation & Why a Slight Accuracy Drop is Acceptable

- **Initial Model Accuracy (before full preprocessing)**: 68%
- **Final Model Accuracy (after full preprocessing)**: 65% (SMOTE Applied)

The slight drop in accuracy is **acceptable** and **expected** because:

- The initial model was biased toward the majority class (non-potable).
- After applying SMOTE, scaling, skewness correction, and outlier treatment, the model became **more balanced, fair, and robust**.
- Improvements were seen in **recall** and **F1-score** for the minority class (potable water), making the model **more generalizable for real-world use**.

---

5. Backend Development (FastApi):

After completing the machine learning model, we built an API to serve the model:

- **Framework Used**: FastAPI
- **Key Features**:
  - Created a /predict POST endpoint.
  - The API accepts JSON input containing the 9 features.
  - Model predicts whether the water is potable (1) or not potable (0).
  - Returned prediction as JSON response.
- **Testing**:
  - Tested the API using **Postman** by sending raw JSON input.
  - Correctly received predictions based on sample inputs.

Example Postman Input:

```
json
Copy code
{
  "ph": 9.18,
  "Hardness": 273.81,
  "Solids": 24041.32,
  "Chloramines": 6.90,
  "Sulfate": 398.35,
  "Conductivity": 477.97,
  "Organic_carbon": 13.38,
  "Trihalomethanes": 71.45,
  "Turbidity": 4.50
}
```

Example Response:

```
json
Copy code
{
```

```
  "prediction": 0
}
```

(Where 0 = Unsafe to Drink, 1 = Safe to Drink)

6. Frontend Development (React.js)

To provide a user-friendly interface, we created a frontend using **React.js**:

- **Frontend Features**:
  - Input fields for all 9 features with user-friendly labels.
  - On submitting the form, it makes a **POST** request to the FastAPI backend.
  - Displays the prediction:
    - **Safe to Drink** (if prediction = 1)
    - **Unsafe to Drink** (if prediction = 0)
- **Issues Handled**:
  - Corrected the logic to match the dataset:
    - 1 → Safe to Drink
    - 0 → Unsafe to Drink
  - Fixed missing modules (axios installation).
  - Handled loading state (`Predicting...`) while waiting for response.
- **Technologies Used**:
  - Axios for HTTP requests.
  - React functional components.
  - CSS for basic styling.

---

7. Testing and Final Integration

- **Postman Testing**:
  - Verified that backend returns correct predictions for known samples.
- **Browser Testing**:
  - Verified frontend correctly sends data, receives prediction, and shows result.
- **Full Workflow**:
  - Enter sample → Click Predict → API returns result → Frontend displays Safe/Unsafe.

8. Testing on Dataset values:

Conclusion

This project demonstrates the full machine learning pipeline:

- Data preprocessing
- Feature engineering
- Model training and tuning
- Model deployment using FastAPI
- Frontend development using React.js
- Complete testing using Postman and web browser

It showcases thoughtful handling of real-world challenges like missing data, skewed distributions, outliers, and class imbalance, along with **full-stack deployment** capabilities.

The model is ready to be deployed on cloud services (AWS, Azure, GCP) for real-world usage if needed.
