1.Problem to be solved

a. What is the prediction problem to be solved?

My idea is to train a classification model that can analyze key indicators like pH level, hardness, and the concentration of dissolved solids and chemicals to predict whether a water sample is potable (safe to drink) or non-potable. By doing this, I would like to speed up the process of water quality assessment, which can be especially useful in places where lab testing is slow, expensive, or unavailable. I would like to highlight that this model doesn't replace sensors or testing kits, it reduces the dependency on the need for an expert to interpret those readings. If basic measurements can be collected with affordable field kits, the model can offer a quick and reliable prediction about potability, helping people make safer decisions without waiting on a lab result

b. Who are the real (or hypothetical) users / beneficiaries of a solution?

The primary beneficiaries or users of this project are individuals and communities that do not have immediate access to professional laboratory testing but still need a reliable way to monitor the safety of their drinking water. This includes public health officials, NGOs focused on water and sanitation, rural communities, water treatment facilities, and disaster recovery teams.

2.Data set properties

a. What is the source of your data set? Include a citation that specifies at least the author(s) and URL.

I got this dataset from Kaggle dataset repository from below URL

Name of the dataset: water_potability.csv
URL: https://www.kaggle.com/datasets/rithikkotha/water-portability-dataset

Author of the repository: RITHIK KOTHA

b. Data set profile: number of items, class distribution, type of features, min/max/mean/mean or distribution for each feature, etc

The data set "*water_potability.csv*" contains quality metrics of 3276 different water bodies, with 9 physicochemical features and a target variable "potability" which has two classes (0=nonpotable, 1=potable). Here, potable means water is safe to drink.

Column Name	Type	Description
ph	Float	Measures the acidity or alkalinity of water. WHO recommends a range between 6.5 and 8.5 for safe drinking water.
Hardness	Float	Represents the concentration of calcium and magnesium salts; affects soap efficiency and taste.
Solids	Float	Total dissolved solids (TDS) in water, including minerals and salts. High TDS indicates high mineral content.
Chloramines	Float	Disinfectant formed from ammonia and chlorine; used in water treatment. Safe level is up to 4 mg/L.
Sulfate	Float	Naturally occurring substance in minerals; high concentrations may occur depending on geography.
Conductivity	Float	Indicates the water's ability to conduct electricity, related to the concentration of dissolved ions.
Organic_carbon	Float	Measures the amount of organic carbon compounds; important in assessing organic contamination.
Trihalomethanes	Float	Byproduct of chlorine disinfection; levels below 80 ppm are considered safe.
Turbidity	Float	Measures the cloudiness of water due to suspended solids; high values indicate poor water quality.
Potability	Categorical	Target variable. Indicates whether the water is safe to drink (1) or not (0).

Potability is a categorical variable which as two distinct values
0 – Not potable (water is not safe to drink)
1 – Potable (water is safe for drinking)

Min/Max/Mean/Median distribution of features

Features	Min	Max	Mean	Median	Missing Values
ph	0	14	7.0808	7.0368	491
Hardness	47.432	323.124	196.3695	196.9676	0
Solids	320.9426	61227.19601	22014.0925	20927.8336	0
Chloramines	0.352	13.127	7.1223	7.1303	0
Sulfate	129	481.0306	333.7758	333.0735	781
Conductivity	181.4838	753.3426	426.2051	421.885	0
Organic_carbon	2.2	28.3	14.285	14.2183	0
Trihalomethanes	0.738	124	66.3963	66.6225	162
Turbidity	1.45	6.739	3.9668	3.955	0
Potability	0	1	0.3901	0	0

Explanation of Missing values:

The dataset contains missing values in three key columns: ph**,** Sulfate, and Trihalomethanes**.** Based on the nature of the data, the missingness is most likely Missing At Random (MAR). This means that the fact a value is missing is related to other observed or unobserved variables, but not to the value of the missing data itself. For example, missing pH levels might be more common under certain sampling conditions or at specific locations where testing equipment was unavailable. These measurements were likely skipped due to logistical reasons, not because the pH values were extreme or unusual.

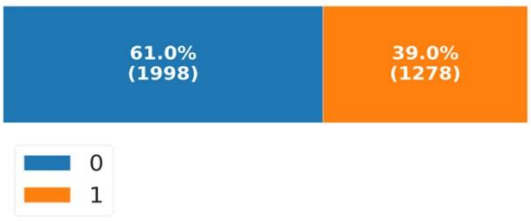
Class Distribution:

Not potable (0): 1,998 samples (≈ 61%)

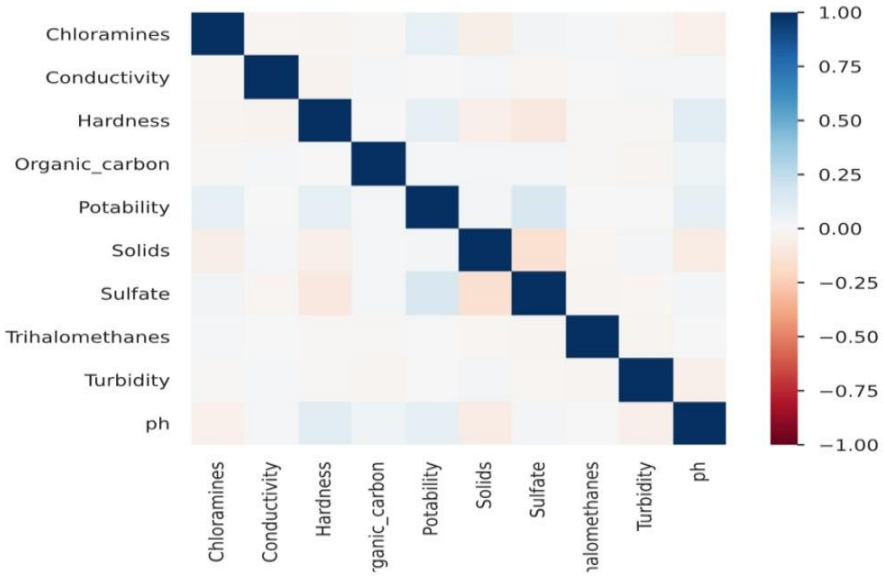
Potable (1): 1,278 samples (≈ 39%)

This concludes that the dataset is having class imbalance, which may impact model performance

Common Values (Plot)



Heat Map: (Collinearity)

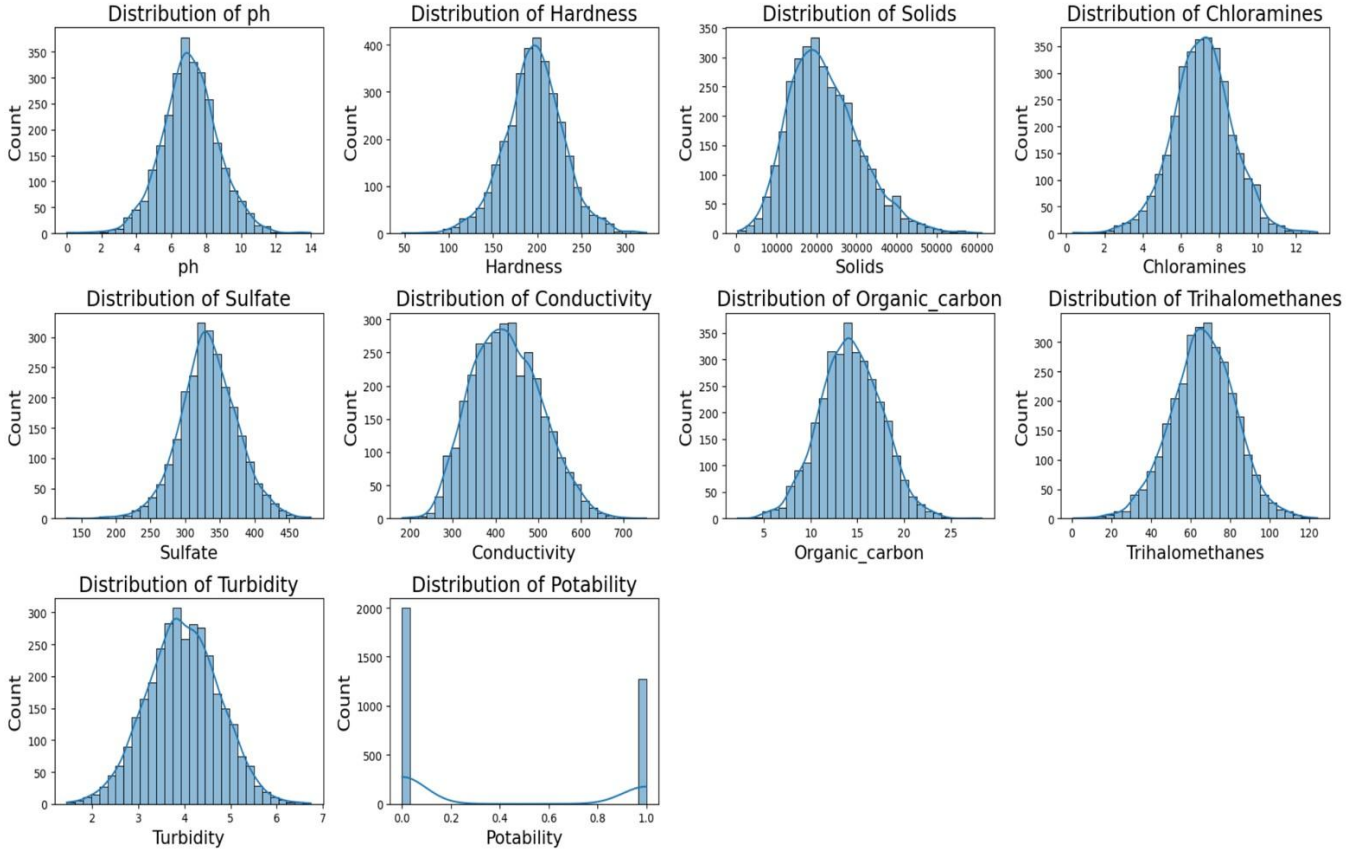


Most features show weak correlations with Potability. The heatmap is mostly light in color, either bluish or reddish, indicating that the correlation values generally fall between –0.1 and +0.1, which is considered very weak.

The only slightly noticeable relationships are:

pH: very light blue (~+0.1 to +0.15)
Sulfate and Chloramines: also weakly positive (~+0.05 to +0.1)

This suggests that no single feature is a strong predictor of potability. Instead, the model will need to learn patterns from combinations of features to make accurate predictions **Distribution of each feature in dataset**



3. Machine learning models

a. What type of model did you use? Why did you choose this model type?

For this project, I used a combination of three supervised classification models:

Logistic Regression, K-Nearest Neighbors (KNN), and Random Forest. I also included a Dummy Classifier to serve as a simple baseline for comparison.

I chose Logistic Regression because it's easy to interpret and tends to perform well when the data is linearly separable. It's often a reliable starting point for binary classification problems like predicting water potability.

KNN was included since it's a non-parametric method that can capture more complex patterns without making strong assumptions about the data. It's useful for seeing how a model based on instance similarity performs in this context.

Random Forest was selected because it's robust, handles non-linear relationships well, and is less prone to overfitting due to its ensemble nature. It also works effectively with imbalanced datasets, especially when using settings like *class_weight='balanced'*.

Using these three different types of models, a linear model, a distance-based model, and an ensemble tree-based model, helped me compare and better understand how each one responds to real-world challenges such as missing data, feature scaling, and class imbalance

Out of all of them, Random Forest consistently gave me the best accuracy, F1 score, recall values across different challenges, so I chose it as my final model. I also picked Random Forest because it's robust to outliers and overfitting, which is important since my dataset is relatively small and has some non-linear patterns. It also handles missing values and noisy features better than most models.

b. What are some strengths and weaknesses of this model type?

Logistic Regression: One of the biggest strengths of logistic regression is its simplicity. It's easy to interpret, quick to train, and works well when the data has a linear relationship. That makes it a great starting point. However, it doesn't do so well when the data has complex, non-linear patterns. It can also be thrown off by outliers or when features are highly correlated.

K-Nearest Neighbors (KNN): KNN is really intuitive and doesn't require a lot of assumptions about the data. Since it doesn't actually train a model (it's a lazy learner), it's great at picking up on non-linear patterns just based on the data itself. The downside is that it can be pretty slow during prediction time, especially as the dataset grows. It's also quite sensitive to features that aren't scaled properly or aren't relevant.

Random Forest: Random Forest tends to be very accurate and handles complex, non-linear relationships really well. It's also great at dealing with missing values and doesn't overfit easily thanks to its ensemble approach. On the flip side, it's not as easy to interpret as simpler models like logistic regression. It also takes longer to train and can use more memory, especially with larger trees and datasets.

c. What hyperparameters must be specified, what values did you choose, and how did you choose them and/or what range of values did you explore for each hyperparameter?

I used *GridSearchCV* to tune hyperparameters for the three main models. Here's how I approached it:

- **Logistic Regression:**

- Tuned parameters: *C (regularization strength), solver, penalty*.
- Explored values: *C = [0.01, 0.1, 1, 10]; solver = ['liblinear', 'lbfgs']; penalty = ['l2']*.
- Reason: To balance between overfitting and underfitting, and to test different optimization algorithms.

- **KNN:**
- Tuned parameters: *n_neighbors, weights, p.*
- Explored values: *n_neighbors = [3, 5, 7, 9]; weights = ['uniform', 'distance']; p = [1 (Manhattan), 2 (Euclidean)]*.
- Reason: To find the best neighborhood size and distance metric for decision boundaries.
- **Random Forest:**
- Tuned parameters: *n_estimators, max_depth, min_samples_split.*
- Explored values: *n_estimators = [100, 150, 200]; max_depth = [None, 5, 10]; min_samples_split = [2, 4, 6]*.
- Reason: To improve generalization and reduce overfitting while maintaining strong predictive power.

Hyperparameters were evaluated using 5-fold cross-validation and optimized for **F1 score**, since the potability dataset has some class imbalance and F1 is better at balancing precision and recall for the minority class.

I got better results for classifiers with below hyperparameters, which I have used throughout the project.

```
CLASSIFIERS = {
    "Dummy classifier (baseline)": DummyClassifier(
strategy="stratified", random_state=42
    ),
    "LogisticRegression": LogisticRegression(
        C=0.1,        penalty='l2',
solver='lbfgs',
max_iter=2000,
random_state=42,
class_weight='balanced',
    ),
    "KNN": KNeighborsClassifier(
        n_neighbors=3,
        p=1,
        weights='distance',
    ),
    "RandomForest": RandomForestClassifier(
n_estimators=150,
min_samples_split=2,
max_depth=None,        random_state=42,
class_weight='balanced',
    ),
}
```

4. **Evaluation**
    a. **What metrics did you use to measure performance?**

To measure model performance, I used accuracy on test data, precision, recall, and F1 score for class 1 (potable water). These metrics together give a more complete picture, especially when the data might be imbalanced. In this project, I considered class 1, which represents potable (drinkable) water, as the positive class, since the main goal was to correctly identify safe drinking water and also it is minority class in this dataset.

In water potability classification, both false positives and false negatives are important, but false negatives can be especially frustrating depending on the situation. A false negative happens when the model predicts that water isn't safe to drink when it actually is. While this doesn't directly harm anyone, it can lead to clean water being wasted, more resources spent on unnecessary treatment, or reduced access to usable water, especially in places where safe water is already scarce. That's why it's important to improve recall for the potable class so we don't accidentally throw out good water and make the system more efficient overall.

    b. **What experimental methodology did you employ to construct training, testing, and validation (if used) data sets?**

To build my training and testing datasets, I used *train_test_split* from scikit-learn to split the original data into 90% for training and 10% for testing set. I used *stratify=y* to make sure both sets had a similar class distribution, and I set the *random state* to 42 to keep results consistent across runs. I didn't create a separate validation set.

Instead, I used *GridSearchCV* on the training set to tune hyperparameters and improve model performance. This way, the test set stayed untouched during tuning, so I could get a fair estimate of how well the model generalizes. After tuning, I evaluated the final model on the test set using accuracy, precision, recall, and F1 score, with class 1 (potable water) as the positive class for all the strategies.

c. What baseline approach(es) did you compare your model to?

To set a baseline, I used scikit-learn's *DummyClassifier* with the "*stratified*" strategy, which assigns labels based on class distribution without learning from the data. This baseline essentially does nothing meaningful; it's a reference point that doesn't try to find patterns. I compared my actual models: Logistic Regression, KNN, and Random Forest against it to ensure they were learning something valuable or not.

5. Challenges

a. Describe at least three challenges you've found with this data set (outliers, missing values, range/scale/units for features, sampling bias, correlations that make the data not i.i.d., etc.) and/or the application area (deployment, maintenance, etc.). Explain how the challenge manifests in your data set (e.g., for missing values, what fraction

of values are missing and for which features? For class imbalance, what is the class distribution?)]

Three challenges with this dataset

i. Class Imbalance:

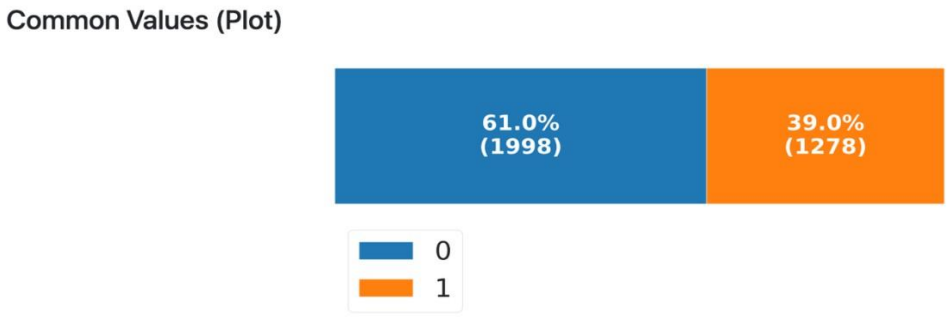
One of the main challenges with this dataset is that it has more non-drinkable water samples than drinkable ones. This uneven distribution creates what's known as class imbalance. When a model is trained on imbalanced data, it tends to favor the majority class. In this case, that means labeling water as not safe to drink, simply because it sees more of those examples. As a result, the model might struggle to correctly identify the smaller group of drinkable water samples, even though they're just as important.

Class Distribution:

Not potable (0): 1,998 samples (≈ 61%)

Potable (1): 1,278 samples (≈ 39%)

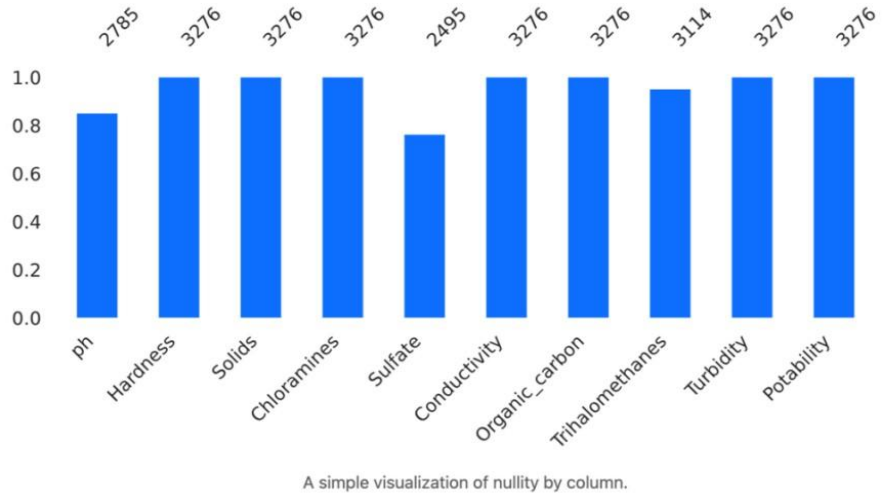
This concludes that the dataset is having class imbalance, which may impact model performance



ii. Missing Values

Dataset contains missing values in three key columns: ph, Sulfate, and Trihalomethanes. Based on the nature of this data, the missingness is most likely MAR (Missing At Random). This means the fact that a value is missing is related to some other observed or unobserved variables, but not to the value of the missing data itself. For example, missing pH levels might be more common in certain sampling conditions or locations where testing equipment wasn't available, or measurements were skipped not because the pH value was extreme or unusual

ph (491 missing, ~15%), Sulfate (781 missing, ~23%), and Trihalomethanes (162 missing, ~5%) have missing values, which is significant and may impact model performance or require imputation.



A simple visualization of nullity by column.

Fig: Bar Chart Showing Null Value Distribution Across Dataset Columns

iii. Feature Scaling and Unit Disparities

Some features in the dataset are measured on very different scales. For example, pH ranges from 0 to 14, Hardness usually falls between 50 and 300, while Solids can exceed 50,000. This large variation creates challenges for models like KNN that rely on distance calculations, as features with larger numerical values such as Solids can disproportionately influence the model, even if they aren't the most important. Moreover, the Solids feature is heavily right-skewed, with a maximum value around 61,000 and high cardinality, which makes it even more likely to overpower other features if not properly scaled.

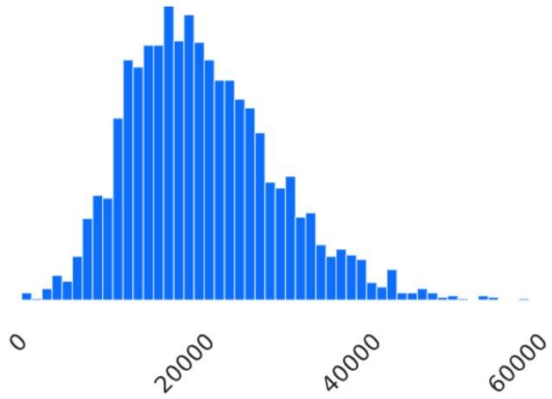


Fig: Histogram of Solids Feature Showing Right Skew and High Cardinality

b. For each challenge, describe three alternative strategies that you investigated. Explain in enough detail that someone else could use this strategy.

Challenge 1 - Class Imbalance:

To handle class imbalance, I employed below three different resampling strategies:

1. ***Random Oversampling***: Before applying oversampling, the training set has 2948 samples with below distribution

```
Original training set class counts:
Potability
0    1798
1    1150
```

By applying oversampling, we duplicate examples from the minority class (class 1: potable water) until both classes had the same number of samples. I used RandomOverSampler to achieve it. This method is straightforward and helped improve recall for the positive class, but it also increased the

risk of overfitting since it relies on repeating the same data points. In my case, it added 648 samples for minority class in my training set. So, I ended up with 3,596 samples in my training set.

```
After Oversampling:
Potability
1    1798
0    1798
```

2.<u>Random Undersampling</u>*:* Here, we took the opposite approach. We reduce the number of majority class samples to match the size of the minority class. Using *RandomUnderSampler*, I randomly dropped many non-potable samples so the two classes would be balanced. While this helped prevent the model from being biased toward the majority class(class 0: non-potable), it came at the cost of discarding a lot of data, which sometimes hurt overall performance. Still, it was a useful baseline to understand how the model behaves with perfectly balanced data.

In my case, it discarded 648 samples which belongs to the majority class and my training set has 2,330 samples after undersampling.

```
After Undersampling:
Potability
0    1150
1    1150
```

3. <u>SMOTE</u> *<u>(Synthetic Minority Over-sampling Technique):</u>* Unlike basic oversampling, SMOTE creates new synthetic examples for the minority class instead of just duplicating existing ones. It works by looking at feature space similarities between neighboring data points in the minority class and generating new, slightly varied data points between them. I applied *SMOTE* with 5 nearest neighbors, which is default value.

For instance, if two potable samples had similar pH and Solids values, SMOTE might create a new data point with values that fall in between. This helped the model generalize better and was especially useful for classifiers like KNN, which are sensitive to data distribution in feature space.

After applying SMOTE, it added 648 synthetic samples of minority class to my training set.

```
After SMOTE:
Potability
1    1798
0    1798

Synthetic samples generated by SMOTE: 648
```

Citation: Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). *SMOTE: Synthetic Minority Over-sampling Technique*. Journal of Artificial Intelligence Research, 16, 321–357. <u>https://doi.org/10.1613/jair.953</u>

Challenge 2 - Missing Values:

In the water potability dataset, several features such as ph, Sulfate, and Trihalomethanes contain missing values. To address this, I explored three different imputation strategies to fill in the missing data before training my models. Each method had its own trade-offs in terms of simplicity, realism, and impact on model performance.

1.Mean Imputation*:* This is the most straightforward method, where I replaced missing values in each column with that column's mean. For example, if the average pH across all non-missing values was 7.2, any missing value in the *ph* column would be filled with 7.2. I used *SimpleImputer* from scikit-learn with *strategy="mean".* This approach is simple and fast, but it may not capture the underlying data patterns well, especially when the distribution is skewed.

2.Random Sample Imputation: Instead of using a fixed value like the mean, it fills in missing values by randomly selecting one of the observed (non-missing) values from the same column, with replacement.

Unlike mean imputation, this approach preserves the original distribution of the data. It's especially helpful when a feature is skewed or has outliers because it keeps the diversity intact. Let's say there are 10 missing values in the *Sulfate* column. This method randomly picks 10 values from the existing non-missing *Sulfate* data (with replacement) and uses them to fill the gaps. It maintains the spread and shape of the original data distribution, which can lead to better model generalization. Since it's random, the results can vary from run to run unless you set a random seed

3. *HotDeck Imputation*: Hot deck imputation is another custom method I implemented to handle missing values. The idea behind it is pretty straightforward, for any row with missing data, I find a donor row from the dataset that has no missing values and use its values to fill in the blanks. This approach helps preserve the natural relationships between features. In my case, samples with high solids and low pH usually have certain sulfate levels, hot deck is likely to pick a donor that reflects that same pattern, which keeps the imputed data more realistic. I chose this method because it doesn't just treat each feature in isolation, it looks at the data more holistically. That said, it does come with some trade-offs. It's slower than other methods, especially as the dataset gets bigger, and if there aren't enough complete rows available, the options for good donor matches become limited.

Challenge 3 - Feature Scaling and Unit Disparities

To address differences in feature scales and distributions, especially for variables like *Solids* that have large magnitudes and skewed distributions, I explored three different preprocessing strategies before training the models. These techniques helped improve model performance, particularly for distance-based algorithms like KNN, and ensured fairer comparisons between features.

***Min-Max Normalization*:** I applied Min-Max scaling using *MinMaxScaler*, which transforms all features into a 0 - 1 range. This is particularly useful when features have bounded ranges, and the data isn't heavily skewed. By rescaling values like *ph*, *Turbidity*, and *Sulfate*, this method helps models converge faster and can sometimes improve performance when used with algorithms that are sensitive to the absolute scale of input values.

***Log Transformation + Standardization**:* The *Solids* feature in the dataset had extreme right skew and very high values (up to 61,000), which could disproportionately affect models. To reduce skew and improve distribution symmetry, I first applied a log transform using *np.log1p()* (which handles zeros safely) to the *Solids* column. Then I standardized the transformed dataset. This combination helped reduce the impact of outliers and made the distribution of Solids more normal-like, improving both model stability and fairness during training.

PowerTransformer (Yeo-Johnson): One of the scaling strategies I explored was the PowerTransformer using the Yeo-Johnson method. I chose this because several features in the dataset weren't normally distributed, they were skewed or heavily lopsided, which can negatively affect the performance of models like Logistic Regression that work best when the input data is bell-shaped or Gaussian. By applying PowerTransformer(method='yeo-johnson'), I aimed to make the data more symmetric and easier for the model to learn from. This method is particularly useful because it handles both positive and negative values, unlike alternatives like Box-Cox. However, one downside is that it can make the transformed data harder to interpret, especially when trying to explain feature importance later. It also changes the original data distribution, which might not be ideal if some features were already close to normal. Still, it was a valuable technique for reducing skew and helping the models generalize better.

6. Results

a. For each challenge: (use placeholders for any in-progress results and explicitly mention that they are in progress). Show your experimental results (probably a table comparing the strategies for this challenge with one or more metrics)

Here are the results from testing different strategies, using metrics like accuracy on test data, F1 score, precision, and recall on class 1.

I focused on evaluating metrics specifically for Class 1 (potable water) because it represents the minority class in the dataset and also critical outcome. In real-world scenarios, especially those involving public health and safety, correctly identifying safe drinking water is far more important than simply achieving high overall accuracy. Metrics like precision, recall, and F1 score for Class 1 give a clearer and more meaningful picture of the model's ability to detect potable water reliably, which is essential for making informed and practical decisions.

Challenge 1 – Class Imbalance:

=== Accuracy (test) ===

	Dummy classifier (baseline)	LogisticRegression	KNN	RandomForest
No Resampling	53.35	53.96	54.57	70.43
Oversampling	49.7	50.61	49.09	68.29
Undersampling	49.7	50.3	50.3	62.2
SMOTE	49.7	51.22	51.83	66.77

=== F1 Score (Class 1) ===

	Dummy classifier (baseline)	LogisticRegression	KNN	RandomForest
No Resampling	0.4093	0.4702	0.3766	0.4641
Oversampling	0.417	0.4527	0.3971	0.4747
Undersampling	0.417	0.4437	0.4399	0.5373
SMOTE	0.417	0.4521	0.4234	0.524

=== Precision (Class 1) ===

	Dummy classifier (baseline)	LogisticRegression	KNN	RandomForest
No Resampling	0.4046	0.4268	0.4054	0.7925
Oversampling	0.3806	0.3988	0.3691	0.6714
Undersampling	0.3806	0.3939	0.3926	0.5143
SMOTE	0.3806	0.4024	0.3973	0.5941

=== Recall (Class 1) ===

	Dummy classifier (baseline)	LogisticRegression	KNN	RandomForest
No Resampling	0.4141	0.5234	0.3516	0.3281
Oversampling	0.4609	0.5234	0.4297	0.3672
Undersampling	0.4609	0.5078	0.5	0.5625
SMOTE	0.4609	0.5156	0.4531	0.4688

To establish a benchmark, a Dummy Classifier was used. This model makes predictions based on class distribution without learning any real patterns. As expected, its performance was poor, with accuracy around 49%, F1 scores near 0.41, and low precision and recall values. These results confirm that any classifier exceeding this baseline is effectively learning from the data.

Before I could run Challenges 1 and 3, I had to handle missing values in the dataset, since many models can't work with incomplete data. For consistency and simplicity across those experiments, I used **mean imputation**. This means I filled in any missing values in each column with the **average value** of that column from the training set. I chose this approach because it's quick, doesn't introduce any new patterns or assumptions, and keeps the dataset easy to work with for the resampling and scaling strategies I was testing.

Classifier Performance by Resampling Strategy

Logistic Regression:

- **Best F1 Score**: *No Resampling* (0.4702)
- **Best Recall**: *No Resampling / Oversampling* (0.5234)
- **Best Precision**: *No Resampling* (0.4268)

Logistic Regression performed best when I didn't apply any resampling. That's likely because I used class_weight='balanced', which already tells the model to pay extra attention to the minority class. When I added synthetic data through SMOTE or duplicated data with Random Oversampling, performance actually dropped a bit, probably because it introduced some noise. In this case, the model's built-in balancing was enough, and adding more resampling just made things more complicated than they needed to be. **K-Nearest Neighbors (KNN):**

- **Best F1 Score**: *Undersampling* (0.4399)
- **Best Recall**: *Undersampling* (0.5)
- **Best Precision**: *No Resampling* (0.4054)

KNN benefited significantly from undersampling. Since it relies on distance calculations, an imbalanced dataset can bias predictions toward the majority class. By undersampling, the class distribution became more balanced, allowing KNN to better identify Class 1 (potable water).

Random Forest

- **Best F1 Score**: *Undersampling* (0.5373)
- **Best Recall**: *Undersampling* (0.5625)
- **Best Precision**: *No Resampling* (0.7925)

Random Forest turned out to be the best overall performer. When I didn't apply any resampling, it reached the highest precision, meaning it was very confident when it labeled water as potable. However, it also missed a lot of actual positive cases, so recall was low. When I used undersampling, recall improved significantly without hurting the F1 score, which provided a much better balance between catching true positives and keeping false positives in check. Overall, undersampling helped Random Forest make more useful and balanced predictions, making it the most effective strategy for this model.

Precision vs Recall trade off:

When comparing all the models, some clear patterns stood out.

- 'No resampling strategy' often gave higher precision, meaning fewer false alarms, but it also missed a lot of true positives, resulting in lower recall.
- Undersampling ended up striking the best balance in most cases, especially with KNN and Random Forest. It did not overly favor either precision or recall, which led to stronger F1 scores, a good indicator that the model was both catching enough positives and being cautious with its predictions. It helps choosing models that make more meaningful and reliable predictions.

For this challenge, out of all the strategies I tried, using Random Forest with undersampling ended up best for me to improve recall on class 1.

Challenge 2 – Missing values:

=== Accuracy (test) ===

	Dummy classifier (baseline)	LogisticRegression	KNN	RandomForest
Mean Imputation	49.7	50.91	57.01	62.2
Random Sample Imputation	49.7	50.91	55.79	63.11
Hot Deck Imputation	49.7	50	55.18	57.32

=== F1 Score (Class 1) ===

	Dummy classifier (baseline)	LogisticRegression	KNN	RandomForest
Mean Imputation	0.417	0.4505	0.4797	0.5373
Random Sample Imputation	0.417	0.4542	0.4528	0.5568
Hot Deck Imputation	0.417	0.4533	0.4615	0.4891

=== Precision (Class 1) ===

	Dummy classifier (baseline)	LogisticRegression	KNN	RandomForest
Mean Imputation	0.3806	0.4	0.4545	0.5143
Random Sample Imputation	0.3806	0.4012	0.438	0.5241
Hot Deck Imputation	0.3806	0.3953	0.4345	0.4589

=== Recall (Class 1) ===

	Dummy classifier (baseline)	LogisticRegression	KNN	RandomForest
Mean Imputation	0.4609	0.5156	0.5078	0.5625
Random Sample Imputation	0.4609	0.5234	0.4688	0.5938
Hot Deck Imputation	0.4609	0.5312	0.4922	0.5234

Before handling missing values, I paired each imputation strategy with undersampling, since that had already shown the best results for handling class imbalance. This helped to keep the comparison between classifiers fair and consistent.

As expected, the Dummy Classifier (used as a baseline) gave the lowest performance overall. Its accuracy stayed fixed at 49.7% across all imputation methods, with F1 scores around 0.417 and no noticeable difference in precision or recall. This confirms that the real classifiers were actually learning meaningful patterns in the data, rather than just guessing based on class distribution.

Classifier performance by imputation strategy:

Logistic Regression

- **Best F1 Score**: Random Sample Imputation (0.4542)
- **Best Recall**: Hot Deck Imputation (0.5312)
- **Best Precision**: Random Sample Imputation (0.4012)

Logistic Regression showed fairly steady performance across all three imputation methods, but the differences were minimal. Random Sample Imputation gave a slight boost, striking a better balance between precision and recall. But F1 and precision suggests, Random sample imputation was overall more balanced. Logistic Regression is more sensitive to the extra noise introduced during imputation, especially since it depends heavily on clean, well-structured feature relationships.

K-Nearest Neighbors (KNN)

- **Best F1 Score**: Mean Imputation (0.4797)
- **Best Recall**: Mean Imputation (0.5078)
- **Best Precision**: Mean Imputation (0.4545)

KNN performed best when paired with Mean Imputation, which makes sense since the algorithm is based on distance calculations and works better with clean, consistently scaled data. The Random Sample and Hot Deck methods might have introduced small inconsistencies that threw it off. Interestingly, KNN actually did slightly better here than it did in Challenge 1. Overall, it was most stable with Mean Imputation, reinforcing that simpler, more uniform imputation methods tend to work best for distance-based models like KNN.

Random Forest

- **Best F1 Score**: Random Sample Imputation (0.5568)
- **Best Recall**: Random Sample Imputation (0.5938)
- **Best Precision**: Random Sample Imputation (0.5241)

Random Forest once again outperformed the other models. Random Sample Imputation led to the strongest F1 score, combining high recall (capturing most positive cases) with solid precision (minimizing false positives) making it best strategy for this challenge. Also, The trade-off here was minimal, which speaks to Random Forest's robustness.

Precision vs Recall tradeoff:

Random Forest consistently showed the best recall across all imputations (especially with Random Sample), meaning it retrieved more potable samples correctly. However, the precision scores were still on the lower side, which means the model often predicted potable water when it wasn't. This trade-off between catching more positives and making more mistakes suggests that class imbalance is still a problem, it's making it harder for the models to reliably identify potable water (Class 1).

Challenge 3 - Feature Scaling and Unit Disparities

```
=============== Challenge 3: Feature Scaling & Unit Disparities ===============

=== Accuracy (test) ===
+------------------------------+------------------------------+---------------------+-------+----------------+
|                              |   Dummy classifier (baseline) |   LogisticRegression |   KNN |   RandomForest |
+==============================+==============================+=====================+=======+================+
| No Scaling                   |                         49.7 |               50.3  | 50.3  |          63.41 |
+------------------------------+------------------------------+---------------------+-------+----------------+
| MinMaxScaler                 |                         49.7 |               50.3  | 56.71 |          63.41 |
+------------------------------+------------------------------+---------------------+-------+----------------+
| PowerTransformer (Yeo-Johnson) |                         49.7 |               50.61 | 57.01 |          63.41 |
+------------------------------+------------------------------+---------------------+-------+----------------+
| Log1p+StandardScaler         |                         49.7 |               50    | 57.62 |          63.41 |
+------------------------------+------------------------------+---------------------+-------+----------------+

=== F1 Score (Class 1) ===
+------------------------------+------------------------------+---------------------+--------+----------------+
|                              |   Dummy classifier (baseline) |   LogisticRegression |    KNN |   RandomForest |
+==============================+==============================+=====================+========+================+
| No Scaling                   |                        0.417 |              0.4437 | 0.4399 |         0.5385 |
+------------------------------+------------------------------+---------------------+--------+----------------+
| MinMaxScaler                 |                        0.417 |              0.4512 | 0.4779 |         0.5385 |
+------------------------------+------------------------------+---------------------+--------+----------------+
| PowerTransformer (Yeo-Johnson) |                        0.417 |              0.4564 | 0.4719 |         0.5385 |
+------------------------------+------------------------------+---------------------+--------+----------------+
| Log1p+StandardScaler         |                        0.417 |              0.4497 | 0.4755 |         0.5385 |
+------------------------------+------------------------------+---------------------+--------+----------------+

=== Precision (Class 1) ===
+------------------------------+------------------------------+---------------------+--------+----------------+
|                              |   Dummy classifier (baseline) |   LogisticRegression |    KNN |   RandomForest |
+==============================+==============================+=====================+========+================+
| No Scaling                   |                       0.3806 |              0.3939 | 0.3926 |         0.5303 |
+------------------------------+------------------------------+---------------------+--------+----------------+
| MinMaxScaler                 |                       0.3806 |              0.3964 | 0.4514 |         0.5303 |
+------------------------------+------------------------------+---------------------+--------+----------------+
| PowerTransformer (Yeo-Johnson) |                       0.3806 |              0.4    | 0.4532 |         0.5303 |
+------------------------------+------------------------------+---------------------+--------+----------------+
| Log1p+StandardScaler         |                       0.3806 |              0.3941 | 0.4599 |         0.5303 |
+------------------------------+------------------------------+---------------------+--------+----------------+
```

=== Recall (Class 1) ===

	Dummy classifier (baseline)	LogisticRegression	KNN	RandomForest
No Scaling	0.4609	0.5078	0.5	0.5469
MinMaxScaler	0.4609	0.5234	0.5078	0.5469
PowerTransformer (Yeo-Johnson)	0.4609	0.5312	0.4922	0.5469
Log1p+StandardScaler	0.4609	0.5234	0.4922	0.5469

Since scaling methods can't handle missing data, I started by filling in the gaps using mean imputation. It's a quick and straightforward approach that works well with all the different scalers I wanted to test. While it's not the most sophisticated method for preserving the underlying patterns in the data, it was a reasonable choice here because the main goal was to focus on how scaling affects model performance, not on imputation quality.

Dummy Classifier got about 49.7% accuracy no matter what scaling I used, which makes sense because it's just guessing. Its F1 score, precision, and recall were all low and didn't change much either. It's clear that scaling generally improved model performance compared to using no scaling at all, except in the case of KNN with PowerTransformer and Log1p combined with StandardScaler, where the results didn't improve

Classifier wise effectiveness of Feature scaling

Logistic Regression

Best Scaling Strategy: *PowerTransformer (Yeo-Johnson)*

- Accuracy: **50.61%**
- F1 Score: **0.4564**
- Precision: **0.4000**
- Recall: **0.5312**

Logistic Regression showed some improvement with PowerTransformer, especially in **recall**, which jumped to over 53%. That means it got better at identifying true positive cases (potable water). However, precision remained lower, suggesting it also produced more false positives. This tradeoff is pretty typical in imbalanced data settings.

K-Nearest Neighbors (KNN)

Best Scaling Strategy: *Log1p + StandardScaler*

- Accuracy: **57.62%**

- F1 Score: **0.4755**
- Precision: **0.4599**
- Recall: **0.4922**

KNN got a lot better after scaling, especially in precision, which jumped up quite a bit. That means when it said the water was safe to drink, it was right more often. Its recall was still decent, though not as high as Logistic Regression. Overall, the balance between the two made KNN one of the stronger models once scaling was applied.

Random Forest

Best Scaling Strategy: *All strategies performed well, gave same results for metrics. Gave best overall performance*

- Accuracy: **63.41%**
- F1 Score: **0.5385**
- Precision: **0.5303**
- Recall: **0.5469**

Random Forest didn't care about scaling at all, its performance stayed exactly the same across every method. That makes sense, since decision trees aren't affected by differences in feature scale. It also had the best **overall recall and precision balance**, meaning it was not only good at finding the potable samples, but also didn't misclassify too many non-potable ones as potable.

Precision vs Recall Tradeoffs:

- Precision and recall don't always move together, and that really showed in this case.
- Logistic Regression did a better job of catching the safe water samples when using PowerTransformer, but it also made more mistakes by calling unsafe water safe. So recall went up, but precision dropped.
- KNN, on the other hand, got a lot more precise after scaling. It was more accurate when it said the water was potable, but it didn't improve much in actually finding all the safe samples.
- Random Forest did the best overall. It had both high precision and recall, and the cool part is it didn't even need any scaling to get there.