Drinking Water Potability

Group 16: Daniel Momeni, Kyle Dinh, Jayoung Kim, Melanie Lue

Statistics 141C Final Project - Spring 2022

Abstract:

Drinking water is a basic human right that has been neglected by governments all over the world. Clean water is a health and development issue at the national, regional, and local level. For instance, in the United States we have seen the effects of contaminated drinking water in Flint Michigan and the islands of Hawaii. This study seeks to find potability of different bodies of water to determine which factors affect the safety of our drinking water infrastructure. This would give us a baseline idea of what factors lead to more dangerous water and how much of each substance can be neglected. For clarity, potability is defined as fit to drink.

We hope that this analysis can be used by respective government agencies so that these substances can be better regulated in our drinking water. Every variable in this study can be regulated by the government so it is fundamental to their understanding to know which ones negatively affect our drinking water. Furthermore, we will find the accuracies of the models we create to find the best model in determining water potability.

Questions we seek to answer in this study include:

1. Which variables / factors have a higher effect on determining water quality?
2. Are there variables that don't have an effect on water quality?
3. Is there any correlation between the variables?
4. How accurate are the classification methods/models chosen?
5. Which classification method is best in determining water potability?

1 About The Data

1.1 General Description

“Water Quality: Drinking Water Potability” [1] is the name of the data set used for this project which was obtained from kaggle. The data set has 10 columns with the first 9 being quantitative variables, and the last column being a binary variable. The data set looks into the water quality metrics of 3,276 unique bodies of water. The variables are as follows:

1. ph: pH of 1. water (0 to 14).
2. Hardness: Capacity of water to precipitate soap in mg/L.
3. Solids: Total dissolved solids in ppm.
4. Chloramines: Amount of Chloramines in ppm.
5. Sulfate: Amount of Sulfates dissolved in mg/L.
6. Conductivity: Electrical conductivity of water in μS/cm.
7. Organic_carbon: Amount of organic carbon in ppm.
8. Trihalomethanes: Amount of Trihalomethanes in μg/L.
9. Turbidity: Measure of light emitting property of water in NTU.
10. Potability: Indicates if water is safe for human consumption (Potable - 1 and Not potable - 0)

1.2 Data Processing

The data set we will be using contains NaN values in the ph, sulfate, and trihalomethanes columns. The ph column contains 491 NaN values. The sulfate column contains 781 NaN values. The trihalomethanes column contains 162 NaN values. We believe that simply removing the rows of data which contained NaN values would deplete our data set too much, thus lowering the quality of our statistical modeling. Instead, we opted to calculate the mean of each column and replace the NaN values of that column with its respective mean. We believe that this will retain the integrity and structure of the data set while not manipulating the results of our statistical analysis.

2 Methodology

2.1 Proposed Methods

This is a classification problem so we will use classification methods to solve the problem. The methods we chose for classification were gradient descent/logistic regression model, K Nearest Neighbors, and decision trees. Unlike linear regression where we can use linear regression methods to compute the coefficients, it is necessary to use gradient descent to estimate the coefficients for logistic regression models for classification. Furthermore, we will use K Nearest Neighbors models as another classification method. For this particular model, we will use parallel computing to compute different K Nearest Neighbors models with different values of K because this method becomes very slow as the dataset increases in size. The final model that we will create is the decision tree, which splits the dataset into smaller and smaller clusters for classification. In this report, we will implement the decision tree from scratch and compare this model to the built in decision tree model. After making the models, we will

create confusion matrices to compare the results and choose the best model based on the highest accuracy.

3 Gradient Descent

3.1 Goal of Gradient Descent

To begin understanding the goal of gradient descent we must first understand what a cost function is. A cost function evaluates the performance of our machine learning algorithm. The loss function computes the error for a training example. The cost function is the average of the loss function. It tells us how good our model is at making predictions. We would want to minimize this value as it would mean we have less error in our algorithm which means it is better at making predictions.

A gradient measures how much the output of a function changes in relation to the change in inputs of a function. A change in all weights in relation to the change in error. Gradient descent is an optimization method used in machine learning to train a model to find the minimum value of a function. It does this by minimizing a cost function to its local minimum.

For our gradient descent model we will be looking at a multiple regression model that takes in potability as our response variable against all of our predictor variables. We want to do this to determine how many iterations are needed in a multiple regression model in order to minimize the cost function which will ultimately minimize the errors in our regression model.

3.2 Mathematical Theory of Gradient Descent

The cost function is defined as:

$$\text{Cost} = \frac{1}{N}\sum_{i=1}^{N}(Y' - Y)^2$$

Cost is equal to 1 over N (our number of data points) multiplied by the summation of Y actual minus Y predicted which is squared. The square differences make it easier to derive a regression line. To compute that line we need to find the derivative of the first cost function. The square difference also increases the error distance which makes the bad predictions more pronounced than the good ones.

To calculate a gradient descent is as follows.

We will find the first derivative cost function in regards to the error and formula y=mx+b. The first derivative below is in regards to m.

$$\frac{\partial}{\partial m}Error = \frac{\partial}{\partial m}(mx + b - Y)$$

Here we see that x, b and y are constants so we end up with:

$$\frac{\partial}{\partial m}Error = x$$

The derivative below is in regards to b.

$$\frac{\partial}{\partial b} Error = \frac{\partial}{\partial b} (mx + b - Y)$$

Here we see that m, x and Y are constants so we end up with:

$$\frac{\partial}{\partial b} Error = 1$$

Now we put the values back in the cost function multiplied by a learning rate. The error determines the direction to minimize the error and the learning rate determines how large a step to take in that direction.

$$\frac{\partial J}{\partial m} = 2Error \cdot X \cdot Learning\ Rate$$

which then equals

$$m^1 = m^0 - error \cdot X \cdot Learning\ Rate$$

$$\frac{\partial J}{\partial m} = 2Error \cdot X \cdot Learning\ Rate$$

which then equals

$$b^1 = b^0 - error \cdot X \cdot Learning\ Rate$$

m^1, b^1 are the next position parameters. $m^1,\ b^0$ are the current position parameters. To solve for the gradient we iterate through our data points using our m and b values. Then we compute the partial derivatives. This gradient tells us the slope of the cost function at our current position and the direction we should move to update our parameters.

3.3 Gradient Descent Conclusion & Analysis

We will plot the cost function of all of our variables in which the x-axis is the number of iterations and the value of the cost function on the y-axis, as shown in Fig. 1. If gradient descent is working properly then the cost function should decrease after every iteration. As mentioned before we applied a multiple regression model to account for the multiple variables we used in our gradient descent model. What we see in our graph is that gradient descent is working properly. Our multiple regression model for which potability was the response variable against all of our predictor variables yields a result in which the cost function is minimized after roughly 200 iterations.

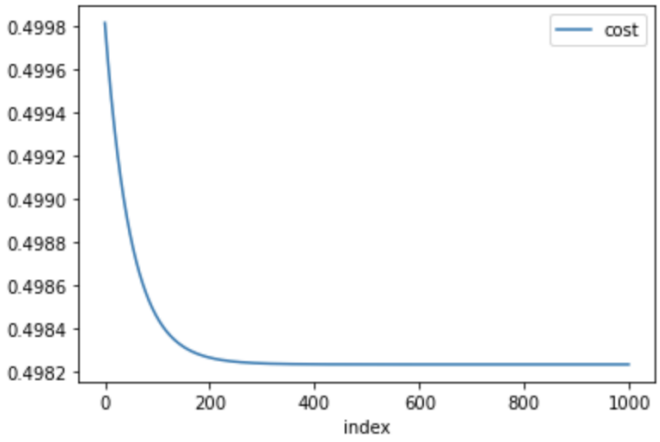

Figure 1. Cost function

4 Logistic Model

4.1 Goal of Logistic Model

We created a logistic regression model in order to model the probability of potable drinking being true against the independent variables of the dataset. Logistic regression measures the goodness-of-fit as a log likelihood function. Knowing the results of gradient descent will also help us here to know how many iterations of the model is needed for the errors to be small. When creating our model we will first begin by making sure that the assumptions of this model are met. We then plan to determine which variables are the most important when it comes to potability. We then want to create a model based off of this to understand the potability of water depending on the variables that make water more dangerous or safe to drink.

4.2 Logistic Regression Assumptions

Unlike linear regression which requires many assumptions such as the errors having a normal distribution and homoscedasticity, logistic regression does not require that many assumptions. Some assumptions of logistic regression are the requirement of the dependent variable to be binary, little or no multicollinearity between variables, and a large sample size. The assumptions of having a binary dependent variable and large sample size are clearly satisfied for this dataset. When we look at the correlation heatmap of the variables in Fig. 2, we notice that there is very low correlation between the variables of our dataset. Thus, the no multicollinearity assumption of logistic regression is also satisfied.

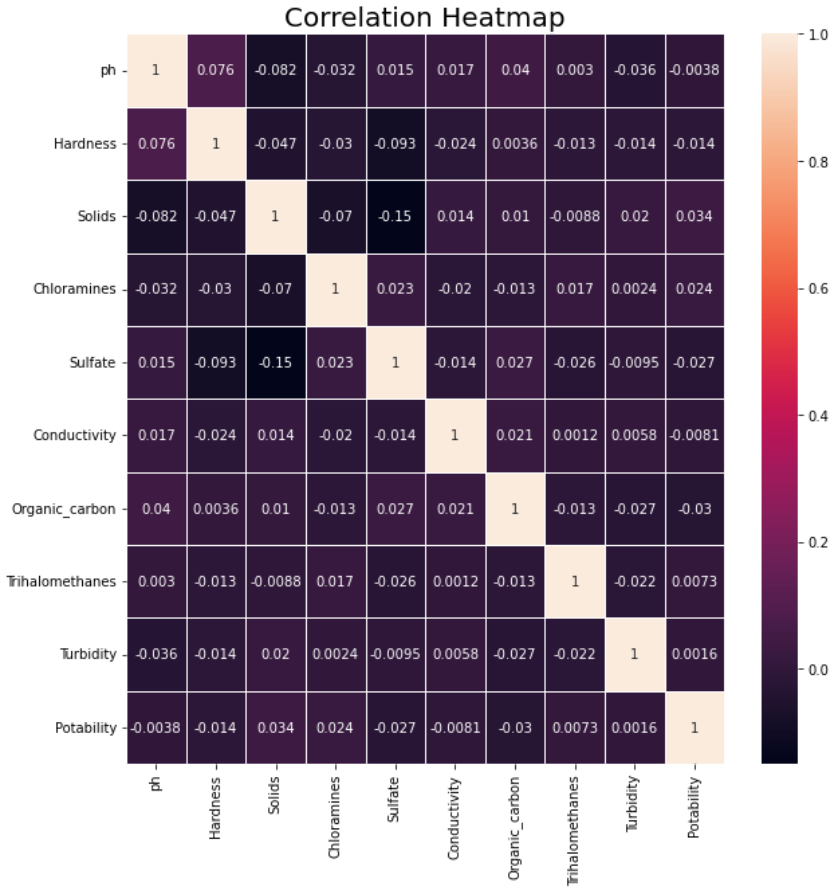

Figure 2. Correlation Heatmap of the Variables

4.3 Feature Importance

We wanted to find the most important features in our data set in regards to potability. This was in order to understand what factors were having a negative and positive impact on our drinking water. These factors could then later be further examined in our logistic model. We obtained the importances of our factors by looking at their coefficients. This was the most practical method as logistic models boil down to an equation in which its coefficients are assigned a value. From Fig. 3, we see that solids, chloramines, and trihalomethanes played a large positive impact on potability. Hardness, sulfate, and organic carbon played a large negative impact on potability. Turbidity and ph played no significant role in feature importance.

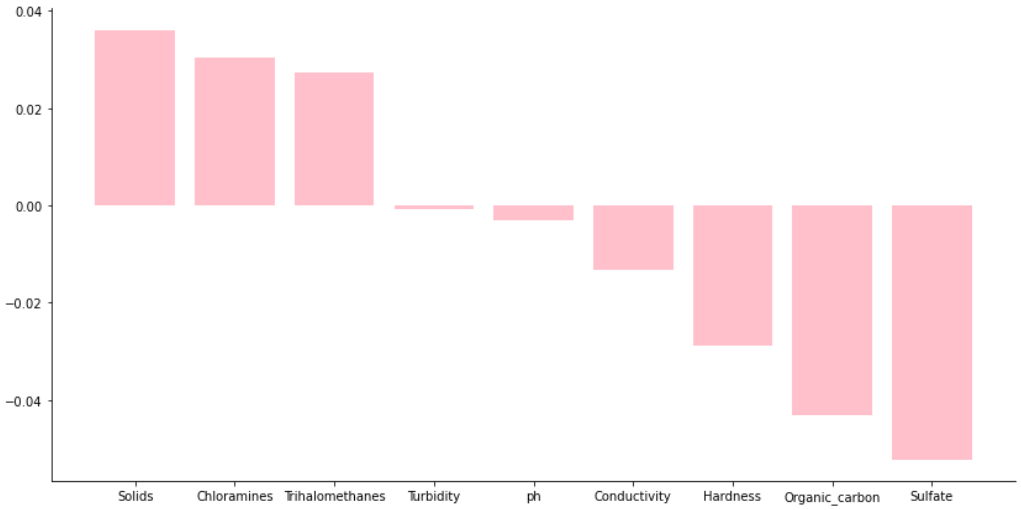

Figure 3. Barplot of Importances from Logistic Regression model

4.4 Classification method performance

From the confusion matrix in Fig. 4, even though its accuracy score is 0.623, we see that our logistic regression model only predicted 0 - classifying the water as not potable. This indicates that logistic regression may not be the best model for predicting water potability.

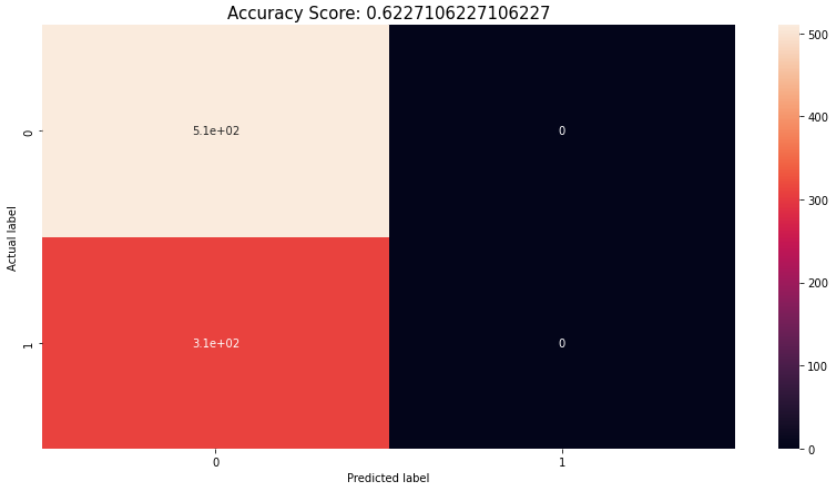

Figure 4. Confusion Matrix of Logistic Regression model

5 K Nearest Neighbors

5.1 Goal of K Nearest Neighbors

In machine learning, K Nearest Neighbors models can be used for classification problems and regression problems. This model relies heavily on the assumption that points close to each other are of the same class or group. The algorithm works by finding the K closest neighbors to a given point. We define the K closest neighbors as the K points where the euclidean distance between these K points and the point of interest is minimized. We can then classify the point of interest based on the classes of the K closest neighbors. Because there are many different K values to choose from, it is reasonable to test the accuracy of different K values and choose the K that results in the model with the highest accuracy.

5.2 Group Assumption

Before we create our K Nearest Neighbors models, we should first determine how reliable the models will be by looking at the grouping of potable water and not potable water. We see in Fig. 5 that there is a lot of overlap between the grouping of potable water and the grouping of not potable water. Thus, we should be cautious of the conclusions we make using this model.

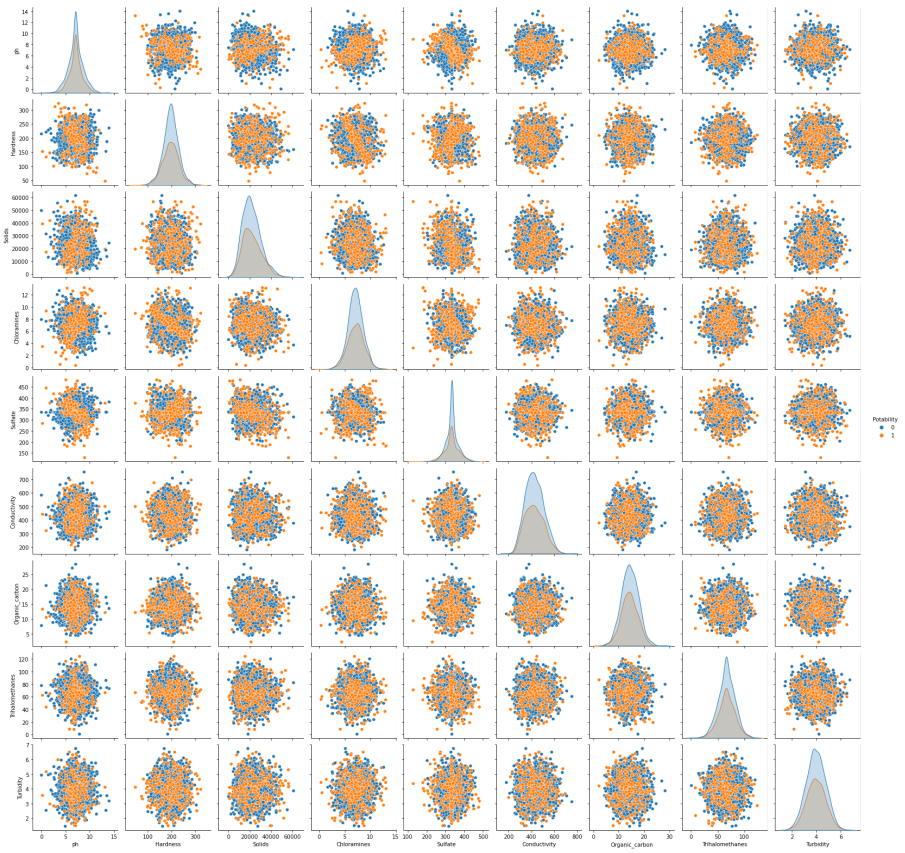

Figure 5. Pairplot of all variables

5.4 Implementing K Nearest Neighbors

In order to find the best K-value for our model, we used parallel computing to find the error rates of k-values 1 to 9, as shown in Fig. 6. From the graph, it is apparent that $K = 2$ had the lowest error rate.

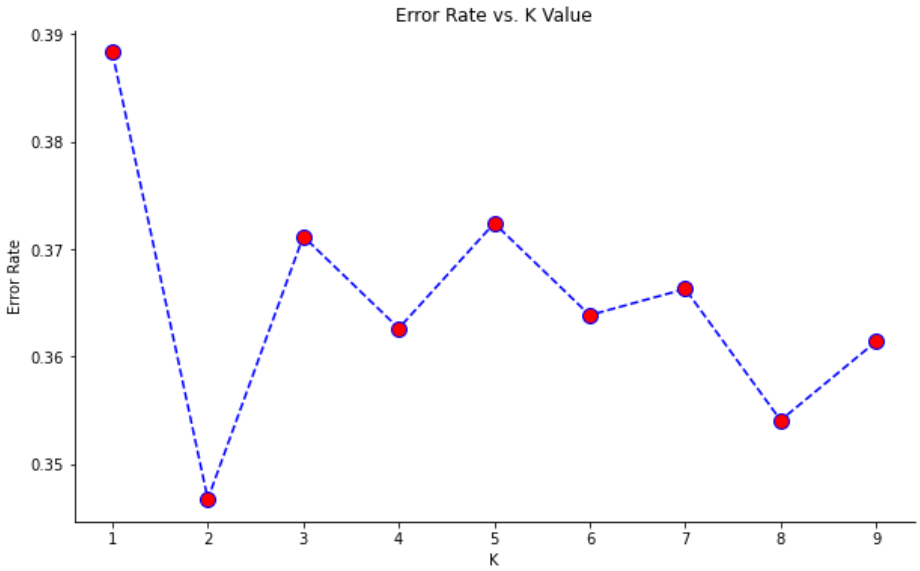

Figure 6. Line Plot of Error Rate vs K-value

5.5 Method Performance

Our model, considering the 2 nearest neighbors, results in an accuracy of 0.66, which is better than the accuracy found in the logistic regression model. We notice from the confusion matrix in Fig. 7, that the model we make tends to classify the water as not potable when in reality it is potable, which is similar to the logistic model. Unlike the logistic model, the model does predict some water samples as potable.

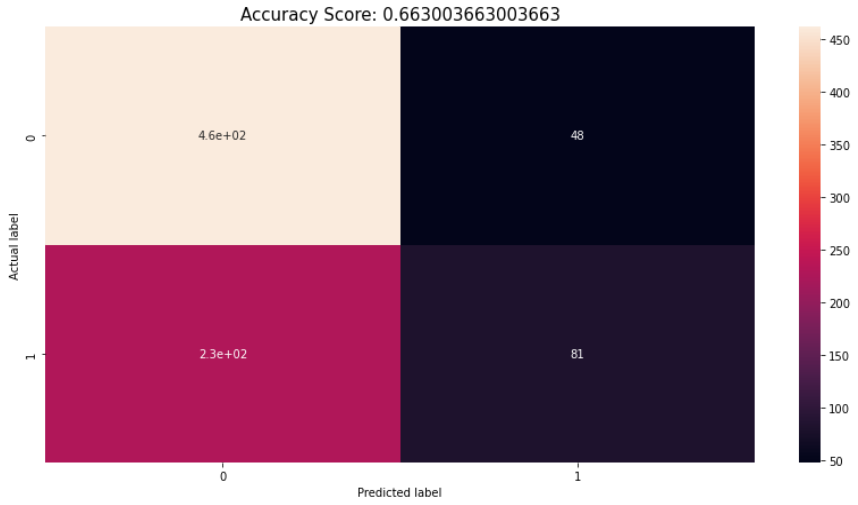

Figure 7. Confusion Matrix for K Nearest Neighbors

6 Decision Tree

6.1 Goal of Decision Tree

To make a good prediction for the water potability and compare the accuracy with other methods we chose above, we decide to use the decision tree. The decision tree is commonly used in the case of supervised learning scenarios because it is pretty immune to missing data and outliers and the most significant variable and the relationship between the variables are easily identified with it. It also has two different types such as regression and classification to handle the different types of data which are numerical and categorical data. In our case, the target variable, water potability is categorical, so we used the classification one to train the variable. In the process of the decision tree, it splits the nodes on all available variables and combinations for the classification case and selects the best split so the purity of the node is really important. Since it should increase with the respect to the target variable after each split, it should be small to get a better prediction.

6.2 Built in Decision Tree

Using a built-in decision tree, we receive an accuracy of 0.63, which is better than the logistic regression one but not better than the K Nearest Neighbors model. In addition, the model seems to struggle with correctly classifying if the water is potable, as shown in the confusion matrix in Fig. 8. When the water is actually potable, the model has almost a 50% accuracy.

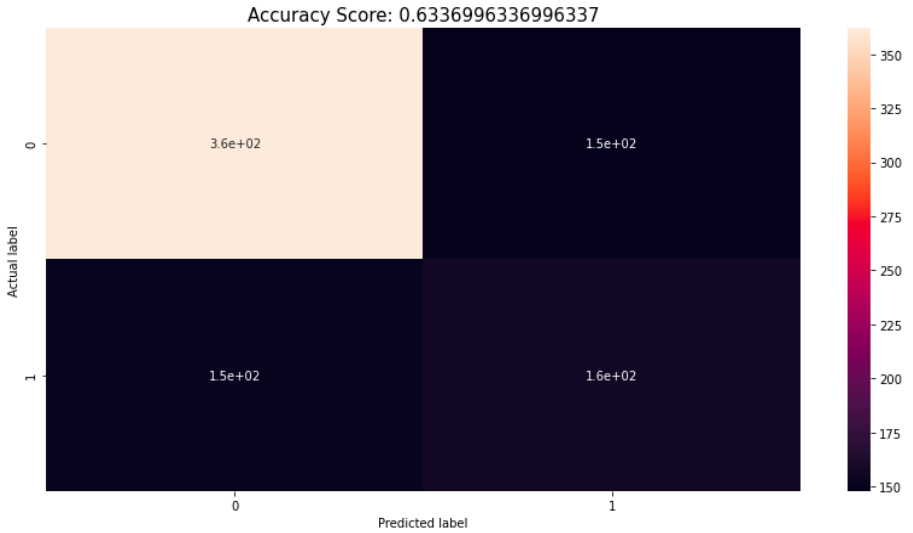

Figure 8. Confusion Matrix for built in Decision Tree

6.3 From Scratch

Creating the decision tree from scratch requires several steps.

6.3.1 Calculate the impurity

In the case of the decision tree, we use the cost functions to find the cuts that minimize impurity which compares cuts for the classification. To calculate the impurity, we use two different cost functions.

i. Using the Gini index

The Gini index calculates the probability of random characteristics that will be incorrectly classified. It ranges from 0 to 0.5 which 0 indicates a pure cut and 0.5 indicates a completely pure cut which means that the variable is very impure so the data is equally divided. The Gini index is calculated as follows:

$$Gini = 1 - \sum_{i=1}^{n} (P_i)^2$$

Where P_i is the probability of having that value.

ii. Using the entropy

Entropy measures impurity and unlike the Gini index it ranges from 0 to 1 where 1 indicates high impurity. Entropy is calculated as the follows:

$$E(S) = \sum_{i=1}^{c} - p_i log_2 p_i$$

6.3.2 Choose the cuts

To choose the cuts for the decision tree, we use entropy and variance to calculate Information Gain which indicates the improvement when making different partitions. Depending on the type of the decision tree, there are two options to calculate the Information Gain.

i. Classification

$$IG = E(d) - \sum \frac{|s|}{|d|} E(s)$$

ii. Regression

$$IG = Variance(d) - \sum \frac{|s|}{|d|} Variance(s)$$

After calculating the Information Gain for all variables, we can choose the split that generates the highest Information Gain.

6.3.3 Calculate the split

Based on the data types, the ways to calculate the split are different.

i. Numeric variable

In order to calculate the split of a numeric variable, first, all values of the variable should be obtained and the Information Gain is used as a filter to check if the value is less or not.

ii. Categorical variable

For a categorical variable, the idea is the same as calculating the split of a numerical variable. Instead of possible values of the variable, we need to calculate the Information Gain for all possible combinations of the variable except for the option that includes all combinations. This step is computationally costly so it is recommended to not have too many categorical variables.

iii. The best split

Once all the splits are calculated for all the variables, now it is time to choose the best split. The best split is the one that generates the highest Information Gain so we just need to calculate the Information Gain for each variable and split the data based on the Information Gain and apply this method recursively until it reaches the end.

6.3.4 Train using the decision tree

In order to train the decision tree, we need some hyperparameters to make the decision tree more efficient. The hyperparameters are the maximum depth of the tree, the minimum number of observations that continually creates new nodes, and the minimum amount the Information Gain that increases for the tree keeps growing. Now, we have all the information to train the decision tree. If one of the conditions is not fulfilled, then we make the prediction.

6.3.5 Predict using the decision tree

Using all the information we obtained above, we can finally make the prediction using the decision tree. We split the decision into several pieces, check the type of decision such as numerical or categorical, and check the decision boundary. If the decision is fulfilled, it will return the result, if it is not, then it will continue with the decision.

6.3.6 Result

We receive an accuracy of 0.73 by using the decision tree from scratch and it is the best result among all the methods we used above. The confusion matrix in Fig. 9 looks pretty similar to the confusion matrices from the logistic regression and K Nearest Neighbors but the confusion matrix of the decision tree from scratch predicts more water samples as potable.

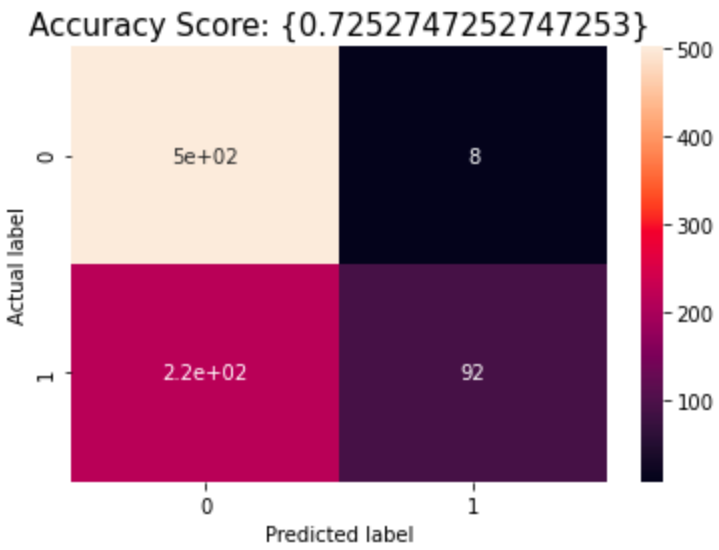

Figure 9. Confusion Matrix for Decision Tree from scratch

7 Conclusion

To review, the questions we want to answer are:

1. Which variables / factors have a higher effect on determining water quality?
2. Are there variables that don't have an effect on water quality?
3. Is there any correlation between the variables?
4. How accurate are the classification methods/models chosen?
5. Which classification method is best in determining water potability?

1.) When we look at Fig. 3, we see the feature importance of the variables. In other words, we see how important each variable is to water quality. For example, solids and chloramines have strong positive effects on water quality while sulfate and organic carbon have strong negative effects on water quality. The exact effects of the variables are shown more clearly in Fig. 3.

2.) Again, Fig. 3 shows not only which variables have positive and negative effects on water quality, but also which variables have little effect on water quality. The two variables with very little effect on water quality are Turbidity and pH.

3.) Referring to Fig. 2, we plotted the correlation heatmap between the variables in our dataset. Interestingly, many of the correlations between the variables are close to 0, which indicates very little correlation between the variables.

4.) Below is a table showing the accuracy scores of each of our models.

Logistic	KNN (k = 2)	Dec. Tree - Built In	Dec. Tree - Scratch
0.62	0.66	0.63	0.73

5.) When we choose the best model for determining water potability, we will quantify the usefulness of each model based on their accuracy. Based on the accuracy in the table above, the best model in determining water potability is the decision tree that we created from scratch which has an accuracy of 0.73.

References

[1] Kadiwal, Aditya (2021) "Water Quality - Drinking Water Potability" Posted on Kaggle.com. Retrieved May 7, 2022 from https://www.kaggle.com/datasets/adityakadiwal/water-potability

[2] https://towardsdatascience.com/introduction-to-logistic-regression-66248243c148

[3] https://towardsdatascience.com/gradient-descent-explained-9b953fc0d2c

[4] https://towardsdatascience.com/introducing-k-nearest-neighbors-7bcd10f938c5

[5] https://anderfernandez.com/en/blog/code-decision-tree-python-from-scratch/

Code Appendix

Gradient Descent Code is below.

```
import pandas as pd
import seaborn as sns
import numpy as np
%matplotlib inline

df = pd.read_csv('water_potability.csv')
df.head()
# gets shape of data
df.shape
# gets summary stats of data
df.describe()
# gets info on the data
df.info()
# Finds sum of null values
df.isnull().sum()
df["Potability"].value_counts()
import matplotlib.pyplot as plt

plt.rcParams['figure.figsize'] = [8, 5]

plt.rcParams['figure.figsize'] = [5, 3]
df.groupby("Potability").size().plot(kind='bar')
plt.show()
# Normalizing the data

water = (df - df.mean())/df.std()
water.head()
# Changes NaN values in a column with the mean of that column

Na_not_accepted = ['ph', 'Hardness', 'Solids', 'Chloramines',
'Sulfate', 'Conductivity', 'Organic_carbon', 'Trihalomethanes',
'Turbidity', 'Potability']
```

```
for column in Na_not_accepted:
    mean = int(water[column].mean(skipna=True))
    water[column] = water[column].replace(np.NaN, mean)

water.head()
XX = water[['ph', 'Hardness', 'Solids', 'Chloramines', 'Sulfate',
'Conductivity', 'Organic_carbon', 'Trihalomethanes', 'Turbidity']]
yy = water['Potability']
XX.head()
XX['intercept'] = 1
XX.head()
XX = np.array(XX)
yy = np.array(yy)
theta = np.matrix(np.array([0,0,0,0,0,0,0,0,0,0]))
alpha = .01
iterations = 1000
#computes the cost function
def compute_cost(X, y, theta):
    return np.sum(np.square(np.matmul(X, theta) - y)) / (2 * len(y))
# Uses linear algebra and gradient descent mathemmatics to interate
through the problem until it is minimized.
# Finds the beta value along with cost from input vectors
def gradient_descent_multi(XX, yy, theta, alpha, iterations):
    theta = np.zeros(XX.shape[1])
    m = len(X)
    gdm_df = pd.DataFrame( columns = ['Bets', 'cost'])

    for i in range(iterations):
        gradient = (1/m) * np.matmul(XX.T, np.matmul(XX, theta) - y)
        theta = theta - alpha * gradient
        cost = compute_cost(XX, yy, theta)
        gdm_df.loc[i] = [theta, cost]
    return gdm_df
gradient_descent_multi(XX, yy, theta, alpha, iterations)
# plots gradient descent
gradient_descent_multi(XX, yy, theta, alpha,
iterations).reset_index().plot.line(x='index', y=['cost'])
```

Code for Feature Importance is below.

```
import matplotlib.pyplot as plt
from matplotlib import rcParams
rcParams['figure.figsize'] = 14,7
rcParams['axes.spines.top'] = False
rcParams['axes.spines.right'] = False
from sklearn.preprocessing import StandardScaler
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler, LabelEncoder

ss = StandardScaler()
label_quality = LabelEncoder()

water['Potability'] =
label_quality.fit_transform(water['Potability'])

X = water.drop('Potability', axis = 1)
y = water['Potability']

# Splits data into train and test data

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size
=.25, random_state = 42)

X_tr_sc = ss.fit_transform(X_train)
X_te_sc = ss.transform(X_test)
# Uses built in logistic regression and coefficents to calculate the
importance value

from sklearn.linear_model import LogisticRegression
model = LogisticRegression()
model.fit(X_tr_sc, y_train)
importances = pd.DataFrame(data={
    'Attribute': X_train.columns,
    'Importance': model.coef_[0]
})

importances = importances.sort_values(by='Importance', ascending =
```

```
False)
# Plots importances.
plt.bar(x=importances["Attribute"], height =
importances['Importance'], color = 'pink')
plt.xticks(rotation = 'vertical')
plt.show()
```

KNN code is below.

```
# normalizing the data
water = (df - df.mean())/df.std()
water.head()

# Correlation of normalized data
cormat = water.corr()
round(water,2)

# Scatter plot
sns.pairplot(df, hue="Potability")

 # uses a function that creates a knn model and returns the error for
that model
%run knn_mp.ipynb

# implements parallel computing
if __name__ == "__main__":
    pool = mp.Pool(processes = 4)
    results = pool.imap(knn_error,range(1,10))

error = []
for i in results:
    error.append(i)

plt.figure(figsize =(10, 6))
plt.plot(range(1, 10), error, color ='blue',
                linestyle ='dashed', marker ='o',
         markerfacecolor ='red', markersize = 10)
```

```
plt.title('Error Rate vs. K Value')
plt.xlabel('K')
plt.ylabel('Error Rate')

# knn_mp.ipynb
def knn_error(x):

    from sklearn.preprocessing import StandardScaler
    from sklearn.model_selection import train_test_split
    from sklearn.preprocessing import StandardScaler, LabelEncoder
    from sklearn.neighbors import KNeighborsClassifier
    import pandas as pd
    import seaborn as sns
    import numpy as np

    df = pd.read_csv('water_potability.csv')
    water = (df - df.mean())/df.std()
    Na_not_accepted = ['ph', 'Hardness', 'Solids', 'Chloramines',
'Sulfate', 'Conductivity', 'Organic_carbon', 'Trihalomethanes',
'Turbidity', 'Potability']

    for column in Na_not_accepted:
        mean = int(water[column].mean(skipna=True))
        water[column] = water[column].replace(np.NaN, mean)

    label_quality = LabelEncoder()
    ss = StandardScaler()

    water['Potability'] =
label_quality.fit_transform(water['Potability'])
    X = water.drop('Potability', axis = 1)
    y = water['Potability']

    X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size =.25, random_state = 42)

    X_tr_sc = ss.fit_transform(X_train)
    X_te_sc = ss.transform(X_test)
```

```
    knn_model = KNeighborsClassifier(n_neighbors = x)
    knn_model.fit(X_tr_sc, y_train)
    pred_i = knn_model.predict(X_te_sc)
    error = np.mean(pred_i != y_test)

    return error


# prints the heat map to visualize correct and incorrect
classifications for k = 2

knn_model = KNeighborsClassifier(n_neighbors=2)
knn_model.fit(X_train,y_train)
knn_pred = knn_model.predict(X_te_sc)
print(confusion_matrix(y_test, knn_pred))
print(classification_report(y_test, knn_pred))
sns.heatmap(confusion_matrix(y_test,knn_pred), annot=True)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
all_sample_title = 'Accuracy Score:
{0}'.format(knn_model.score(X_te_sc, y_test))
plt.title(all_sample_title, size = 15)
```

Decision tree code is below.

```
# Using the built-in function
# Defining the decision tree algorithm
dt_model = DecisionTreeClassifier()
dt_model.fit(X_train,y_train)
# Predicting the values of test data
dt_pred = dt_model.predict(X_te_sc)
print(confusion_matrix(y_test, dt_pred))
print(classification_report(y_test, dt_pred))
sns.heatmap(confusion_matrix(y_test,dt_pred), annot=True)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
all_sample_title = 'Accuracy Score: {0}'.format(dt_model.score(X_te_sc,
y_test))
plt.title(all_sample_title, size = 15)

# Using the function created from scratch
water = (df - df.mean())/df.std()
X = water.iloc[:, :-1].values
y = water.iloc[:, -1].values.reshape(-1,1)

X_train, X_test, y_train, y_test = train_test_split(X, y, test_size =.25,
random_state = 42)

class Node():
    def __init__(self, feature_index=None, threshold=None, left=None,
right=None, info_gain=None, value=None):
        # for decision node
        self.feature_index = feature_index
        self.threshold = threshold
        self.left = left
        self.right = right
        self.info_gain = info_gain
        # for leaf node
        self.value = value

class DecisionTreeScratch():
    def __init__(self, min_samples_split=2, max_depth=2):
        self.root = None
        self.min_samples_split = min_samples_split
        self.max_depth = max_depth

    def build_tree(self, dataset, curr_depth=0):
        X, Y = dataset[:,:-1], dataset[:,-1]
        num_samples, num_features = np.shape(X)
        if num_samples>=self.min_samples_split and curr_depth<=self.max_depth:
```

```
            best_split = self.get_best_split(dataset, num_samples,
num_features)
            if best_split["info_gain"]>0:
                left_subtree = self.build_tree(best_split["dataset_left"],
curr_depth+1)
                right_subtree = self.build_tree(best_split["dataset_right"],
curr_depth+1)
                return Node(best_split["feature_index"],
best_split["threshold"],
                            left_subtree, right_subtree,
best_split["info_gain"])
        leaf_value = self.calculate_leaf_value(Y)
        return Node(value=leaf_value)

    def get_best_split(self, dataset, num_samples, num_features):
        best_split = {}
        max_info_gain = -float("inf")
        for feature_index in range(num_features):
            feature_values = dataset[:, feature_index]
            possible_thresholds = np.unique(feature_values)
            for threshold in possible_thresholds:
                dataset_left, dataset_right = self.split(dataset,
feature_index, threshold)
                if len(dataset_left)>0 and len(dataset_right)>0:
                    y, left_y, right_y = dataset[:, -1], dataset_left[:, -1],
dataset_right[:, -1]
                    curr_info_gain = self.information_gain(y, left_y, right_y,
"gini")
                    if curr_info_gain>max_info_gain:
                        best_split["feature_index"] = feature_index
                        best_split["threshold"] = threshold
                        best_split["dataset_left"] = dataset_left
                        best_split["dataset_right"] = dataset_right
                        best_split["info_gain"] = curr_info_gain
                        max_info_gain = curr_info_gain
        return best_split

    def split(self, dataset, feature_index, threshold):
        dataset_left = np.array([row for row in dataset if
row[feature_index]<=threshold])
        dataset_right = np.array([row for row in dataset if
row[feature_index]>threshold])
        return dataset_left, dataset_right

    def information_gain(self, parent, l_child, r_child, mode="entropy"):
```

```
        weight_l = len(l_child) / len(parent)
        weight_r = len(r_child) / len(parent)
        if mode=="gini":
            gain = self.gini_index(parent) - (weight_l*self.gini_index(l_child)
+ weight_r*self.gini_index(r_child))
        else:
            gain = self.entropy(parent) - (weight_l*self.entropy(l_child) +
weight_r*self.entropy(r_child))
        return gain

    def entropy(self, y):
        class_labels = np.unique(y)
        entropy = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            entropy += -p_cls * np.log2(p_cls)
        return entropy

    def gini_index(self, y):
        class_labels = np.unique(y)
        gini = 0
        for cls in class_labels:
            p_cls = len(y[y == cls]) / len(y)
            gini += p_cls**2
        return 1 - gini

    def calculate_leaf_value(self, Y):
        Y = list(Y)
        return max(Y, key=Y.count)

    def fit(self, X, Y):
        dataset = np.concatenate((X, Y), axis=1)
        self.root = self.build_tree(dataset)

    def predict(self, X):
        preditions = [self.make_prediction(x, self.root) for x in X]
        return preditions

    def make_prediction(self, x, tree):
        if tree.value!=None: return tree.value
        feature_val = x[tree.feature_index]
        if feature_val<=tree.threshold:
            return self.make_prediction(x, tree.left)
        else:
            return self.make_prediction(x, tree.right)
```

```
dts_model = DecisionTreeScratch(min_samples_split=3, max_depth=3)
dts_model.fit(X_train,y_train)
dts_pred = dt_model.predict(X_test)

sns.heatmap(confusion_matrix(y_test, dts_pred), annot=True)
plt.ylabel('Actual label')
plt.xlabel('Predicted label')
all_sample_title = 'Accuracy Score:
{0}'.format(dts_model.score(X_test, y_test))
plt.title(all_sample_title, size = 15)
```