# Water Quality Analysis – Phase 4

## Understanding data analysis, visualization techniques

In this article, we will go on a journey with a small water quality dataset. From the data, we will seek to find hidden insights with data analysis techniques using pandas and numpy. For the data visualizations, the matplotlib and seaborn libraries will be used. A range of exploratory data analysis (EDA) techniques will be employed to provide further clarity for the data quality.

Each data visualization will aim to highlight different characteristics of the data. They will also provide the user with templates to apply to other challenges.

## ****Dataset****

For this piece of analysis, the Water Quality dataset has been taken from Kaggle¹.

<https://www.kaggle.com/datasets/adityakadiwal/water-potability?source=post_page-----ebc1cf553081-------------------------------->

A jupyter notebook instance with Python code was used for processing.

import sys  
print(sys.version) # displays the version of python installed

After running the script above an output would show that version 3.7.10 of Python was used. To be able to replicate the results that follow, users should ensure that Python 3 is used within the working environment.

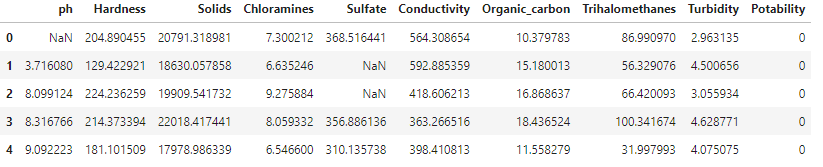
## Understanding the data

Firstly, we need to understand the data that we are working with. As the file format is a csv file, the standard pandas import statement using read\_csv will be used.

# Import the dataset for review as a DataFrame  
df = pd.read\_csv("../input/water-potability/water\_potability.csv")  
# Review the first five observations  
df.head()

Having imported the data, the code assigns the variable df with the DataFrame output results from the pandas method.

As with any dataset that you will process, reviewing a sample of records will help you to gain comfort. A DataFrame has a large number of methods associated with it, with the pandas API a great resource to use. Within the API a head method can be used. Output 1.1 shows the first 5 rows of the DataFrame by default. In order to produce a larger number of rows to be displayed a numeric value would be required inside the parenthesis. Two alternatives could be applied to sample the DataFrame with i) sample (df.sample()) selecting random rows from the index, or ii) tail (df.tail()) selecting the last n rows from the index.

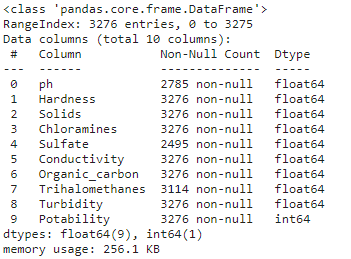
 First five record details from the DataFrame

When running any method, the parenthesis is included after the method name allowing the Python interpreter to produce the result.

Displaying the memory of a DataFrame can be a common task, particularly when memory constraints are involved. An example is where the dataset to import is potentially larger than the memory available within the Python session. By using the pandas library a DataFrame is created in-memory so users should understand what memory can be used when performing these processing steps.

# Display information about the DataFrame - contains memory details  
df.info(memory\_usage="deep")

The code above can be used as a method to display output 1.2. With the inclusion of the keyword memory\_usage, the Python interpreter is forced to do a deeper search to understand the memory usage that is displayed below. A default option would perform a general search to understand, so if accuracy in your assessment is required then ensure that the keyword phrase from above is applied.



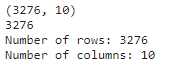
From the results shown in output 1.2, it can show a range of details, from the column names and data types, to also confirming the class of the variable and number of non-null values. We can see that 3,276 rows are shown within the entire table. However, for the column Sulfate, there are only 2,495 non-null values present. Therefore, a number of missing values can be reviewed to understand if there is a pattern for these missing entries with other columns. We will review a data visualization technique later in the article that can help with pattern recognition.

Following the earlier import statement, users could have adjusted the Dtype of a column if the default options were not what was expected. The results above display that for decimal numbers the float Dtype is applied, with the whole number showing int. Also, the largest byte memory type for these numeric columns has been included in order to provide the full coverage of potential input values. Many times users should assess if these Dtypes are holding the correct range of values and if a smaller range is expected going forward then a smaller byte value could be assigned. Applying this logic would help to increase the memory efficiency of the DataFrame and aid with performance when processing.

One feature shown by the info method above that can be reviewed by a number of other methods is the structure of the DataFrame. Such metadata can allow programmers to review basic components of the number of rows and columns.

# Shape of the DataFrame - shows tuple of (#Rows, #Columns)  
print(df.shape)  
# Find the number of rows within a DataFrame  
print(len(df))  
# Extracting information from the shape tuple  
print(f'Number of rows: {df.shape[0]} \nNumber of columns: {df.shape[1]}')

When calling an attribute in Python such as shape, there will be no parenthesis required. An attribute is a data result that can be accessed by both a class and its object. Earlier we reviewed a method which is a function that is contained within a class. For further insights on the smaller details a deep dive into how Python class statements function would be required. However, we can continue with the code that is used and show that with output 1.3 a number of values have been displayed.



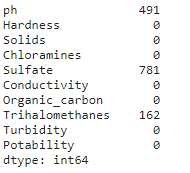
The first row shows the shape output which is a tuple, that is represented by a parenthesis with two values. From the code shown above we are able to access the relative positions within this tuple to display the first and second position values.

## Missing values

As discussed earlier from the metadata and summary statistics there are a number of missing values within the DataFrame. To confirm if this is correct we can apply the code block below.

# Check for the missing values by column  
df.isnull().sum()

The code chained the first isnull method with the sum method to create the number of missing values per column. An isnull assessment will review for non-null values in a column. The sum method is used to perform the count. Output 1.7 highlights that three columns display missing values.



Having the total count of rows with missing values is a great starting point. However, it would be better to review the proportion of missing values within a column.

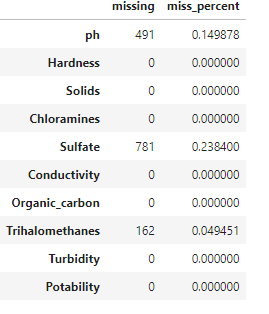
# Proportion of missing values by column  
def isnull\_prop(df):  
 total\_rows = df.shape[0]  
 missing\_val\_dict = {}  
 for col in df.columns:  
 missing\_val\_dict[col] = [df[col].isnull().sum(), (df[col].isnull().sum() / total\_rows)]  
 return missing\_val\_dict  
  
# Apply the missing value method  
null\_dict = isnull\_prop(df)  
print(null\_dict.items())

Creating the isnull\_prop user-defined function enables us to create a dictionary of values for each column. With this function, we have produced the count value from above, as well as using the shape attribute to understand the percentage of missing values.



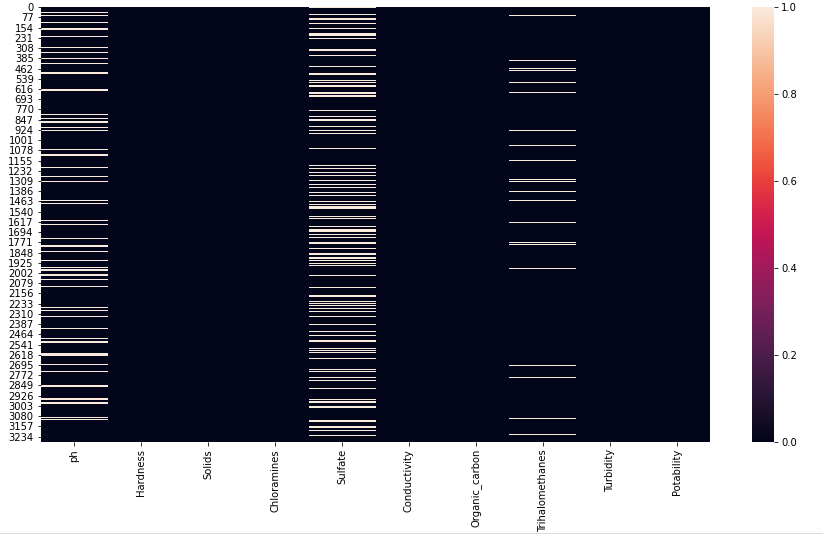
# Create a dataframe of the missing value information  
df\_missing = pd.DataFrame.from\_dict(null\_dict,   
 orient="index",   
 columns=['missing', 'miss\_percent'])  
df\_missing

Applying the dictionary variable to the pandas DataFrame method will make it easier to understand the differences for each column. Output 1.9 now includes the miss\_percent column. We could now apply a threshold value to assess if the percentage of missing values is within our expected range to use the column. If the value is too high e.g., Sulfate value greater than 20%, a user-defined control could be in place that highlights this column needs to either be excluded from future use or reviewed in more detail.



An alternative method to review if any patterns are present by missing values is to apply the heatmap method from the seaborn data visualization library.

# Display missing values using a heatmap to understand any patterns  
plt.figure(figsize=(15,8))  
sns.heatmap(df.isnull());



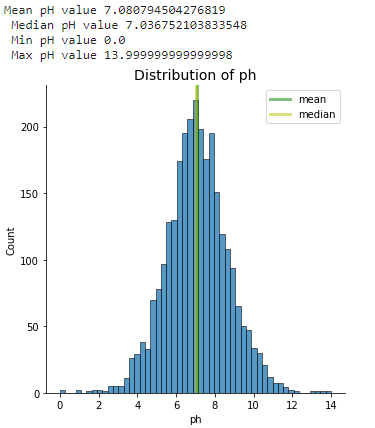
## Understanding the pH variable distribution

One final assessment would be to perform a review of a variable that we have prior external knowledge about. With the seaborn library, we are able to produce a histogram of the pH variable.

# set the histogram, mean and median  
sns.displot(df["ph"], kde=False)  
plt.axvline(x=df.ph.mean(), linewidth=3, color='g', label="mean", alpha=0.5)  
plt.axvline(x=df.ph.median(), linewidth=3, color='y', label="median", alpha=0.5)  
  
# set title, legends and labels  
plt.xlabel("ph")  
plt.ylabel("Count")  
plt.title("Distribution of ph", size=14)  
plt.legend(["mean", "median"]);  
  
print(f'Mean pH value {df.ph.mean()}   
 \n Median pH value {df.ph.median()}   
 \n Min pH value {df.ph.min()}

\n Max pH value {df.ph.max()}')

Similar to the print statement from earlier, the f string statement allows us to add the mean, median, min, and max values to make it easier to review the distribution.

\n Max pH value {df.ph.max()}')

Output shows that the majority of pH values are close to the middle. With a distribution similar to a normal distribution, we could use this insight to help when presenting details to external users.

## ****Conclusion****

Throughout this article, we have aimed to review the early stages of an EDA assessment. Metadata on the imported data was initially reviewed to display early insights. A deeper dive into the summary statistics allowed us to focus on the missing values. Finally, we were able to review the histogram of the pH variable to ensure that the variable followed external expectations.