Dirty Water

# Introduction

# Default libraries: pandas, numpy

# import beakerx

import cProfile

import datetime

import math

import os

import pickle

import sys

import time

import matplotlib.pyplot as plt

import numpy as np

import pandas as pd

import seaborn as sns

# import urbangrammar\_graphics as ugg

from IPython.display import Markdown as md

from collections import Counter

from clustergram import Clustergram

from datetime import datetime

from itertools import groupby

from sklearn import \_\_version\_\_ as sklearn\_version

from sklearn.decomposition import PCA

from sklearn.dummy import DummyRegressor

from sklearn.ensemble import RandomForestClassifier, RandomForestRegressor, AdaBoostClassifier, BaggingClassifier

from sklearn.feature\_selection import SelectKBest, f\_regression

from sklearn.impute import SimpleImputer

from sklearn.linear\_model import LinearRegression, LogisticRegression

from sklearn.metrics import mean\_squared\_error, mean\_absolute\_error, accuracy\_score, classification\_report

from sklearn.model\_selection import train\_test\_split, cross\_validate, learning\_curve, GridSearchCV, RandomizedSearchCV, GridSearchCV

from sklearn.neighbors import KNeighborsClassifier

from sklearn.pipeline import make\_pipeline

from sklearn.preprocessing import scale, StandardScaler, MinMaxScaler, RobustScaler

from sklearn.tree import DecisionTreeClassifier

from xgboost import XGBClassifier

pd.options.display.float\_format = "{:.2f}".format

# plt.rcParams.update({'font.size': 18})

<h1>Table of Contents<span class="tocSkip"></span></h1><a id='0'></a>

<!-- <font size="5"> \*\*Questions:\*\* </font> -->

\* [1 Introduction](#1-Introduction)

\* [1.1 Data source](#1.1-Data-source)

\* [1.2 Stakeholders](#1.2-Stakeholders)

\* [1.3 Water Quality](#1.3-Water-Quality)

\* [2 EDA (Exploratory Data Analysis)](#2-EDA-(Exploratory-Data-Analysis))

\* [2.1 Objectives](#2.1-Objectives)

\* [2.2 Load Data](#2.2-Load-Data)

\* [2.3 Missing Values](#2.3-Missing-Values)

\* [2.4 Numeric Data](#2.4-Numeric-Data)

\* [2.4.1 Distributions](#2.4.1-Distributions)

\* [2.4.2 Classification: measured vs. expected](#2.4.2-Classification:-measured-vs.-expected)

\* [2.4.3 Boxplots and Statistical summary grouped by potability](#2.4.3-Boxplots-and-Statistical-summary-grouped-by-potability)

\* [2.4.4 Scatterplots of numeric features against Potability](#2.4.4-Scatterplots-of-numeric-features-against-Potability)

\* [2.4.5 Correlation heatmap](#2.4.5-Correlation-heatmap)

\* [2.4.6 Considerations](#2.4.6-Considerations)

\* [2.5 Principle components analysis (PCA)](#2.5-Principle-components-analysis (PCA))

\* [2.5.1 Scale the data](#2.5.1-Scale-the-data)

\* [2.5.2 PCA transformation](#2.5.2-PCA-transformation)

\* [2.6 EDA Conclusions](#2.6-EDA-Conclusions)

\* [3 Modeling](#3-Modeling)

\* [3.1 Data treatment](#3.1-Data-treatment)

\* [3.1.1 Missing values](#3.1.1-Missing-values)

\* [3.1.2 Scaling](#3.1.2-Scaling)

\* [3.1.3 Train/Test sets](#3.1.3-Train/Test-sets)

\* [3.2 Classifiers](#3.2-Classifiers)

\* [3.2.1 Declaration](#3.2.1-Declaration)

\* [3.2.2 Parameters](#3.2.2-Parameters)

\* [3.2.3 Grid](#3.2.3-Grid)

\* [3.2.4 Training](#3.2.4-Training)

\* [3.2.5 Performance](#3.2.5-Performance)

\* [3.2.6 Best model](#3.2.6-Best-model)

\* [4 Conclusions](#4-Conclusions)

# 1 Introduction

Drinking water is essential for living and should be available to all human beings. But, unfortunately, this is not the reality around the world. Dry places are no the most populated places on the planet, the deserts, for example. However, there are locations with water bodies, but contaminants compromise the water potability making it unsafe for human consumption.

This project will explore the water potability classification of 3276 water bodies by evaluating the water's physical properties.

<!-- https://www.sciencedirect.com/topics/chemical-engineering/potable-water -->

## 1.1 Data source

This data is available on the public data repository Kaggle at https://www.kaggle.com/adityakadiwal/water-potability/activity.

There is no information about the data source or additional details, e.g., the water bodies geographical location.

## 1.2 Stakeholders

- \*\*Governmental or Non-governmental entities\*\* worried about the population public policies of health and sanitation

- \*\*Water treatment companies\*\* that could provide their services on the location of these water bodies and need to know which kind of facilities they would have to use accordingly to the water treatment required

- \*\*Researchers\*\* looking for answers about the water usage versus health indicators in a given location, or the waste disposition along the water body and its treatment, and other water consumption related topics

- \*\*Inhabitants\*\* of the regions where they collected the samples of water.

## 1.3 Water Quality

We will evaluate some water physical properties:

\*\*1. pH value:\*\*

indicates if the water is acidic or alkaline. WHO recommends pH from 6.5 to 8.5.

\*\*2. Hardness:\*\* is the concentration of calcium and magnesium salts.

\*\*3. Solids (Total dissolved solids - TDS):\*\*

inorganic and some organic minerals or salts such as potassium, calcium, sodium, bicarbonates, chlorides, magnesium, sulfates, etc. The desirable limit is 500 mg/l, and the maximum limit is 1000 mg/l.

\*\*4. Chloramines:\*\*

are most commonly formed when ammonia is added to chlorine to treat drinking water. Chlorine levels up to 4 milligrams per liter (mg/L or 4 parts per million (ppm)) are considered safe in drinking water.

\*\*5. Sulfate:\*\*

occurs naturally in minerals, soil, and rocks. Most freshwater supplies measure 3 to 30 mg/L of sulfate, although some locations present much higher concentrations (1000 mg/L). Seawater sulfate concentration is about 2,700 mg/L.

\*\*6. Conductivity:\*\*

electrical conductivity (EC) measures the ionic process of a solution that enables it to transmit current. According to WHO standards, EC value should not exceed 400 μS/cm.

\*\*7. Total Organic Carbon:\*\*

(TOC) measures the total amount of carbon in organic compounds. According to US EPA, < 2 mg/L in treated / drinking water, and < 4 mg/L in source water used for treatment.

\*\*8. Trihalomethanes:\*\*

(THMs) are chemicals found in water treated with chlorine. Drinking water THMs concentration depends on the level of organic material, chlorine amount, and temperature used during the treatment. THM levels up to 80 ppm are considered safe in drinking water.

\*\*9. Turbidity:\*\*

is a measure of light-emitting properties of water and indicates the waste discharge concerning the colloidal matter. WHO recommends turbidity below 5.00 NTU.

\*\*10. Potability:\*\*

is our \*\*\*target variable\*\*\* indicating if the water is safe for human consumption (potable) or not (non-potable):

- 1: potable

- 0: non-potable.

dict\_range = {

'ph': [6.5, 8.5],

'Hardness': [120, 170],

'Solids': [0, 1000],

'Chloramines': [0, 4],

'Sulfate': [3, 30],

'Conductivity': [0, 400],

'Organic\_carbon': [0, 4],

'Trihalomethanes': [0, 80],

'Turbidity': [0, 5]

}

dict\_labels = {

'ph': [[-np.inf, -10e-5, 6.5, 8.5, 14, np.inf],

['out\_range\_below', 'below', 'potable', 'above', 'out\_range\_high']],

'Hardness': [[-np.inf, -10e-5, 120, 170, np.inf],

['out\_range\_below', 'below', 'potable', 'above']],

'Solids': [[-np.inf, -10e-5, 500, 1000, np.inf],

['out\_range\_below', 'below', 'potable', 'above']],

'Chloramines': [[-np.inf, -10e-5, 4, np.inf],

['out\_range\_below', 'potable', 'above']],

'Sulfate': [[-np.inf, -10e-5, 3, 30, np.inf],

['out\_range\_below', 'below', 'potable', 'above']],

'Conductivity': [[-np.inf, -10e-5, 400, np.inf],

['out\_range\_below', 'potable', 'above']],

'Organic\_carbon': [[-np.inf, -10e-5, 2, 4, np.inf],

['out\_range\_below', 'potable', 'needs treatment', 'above']],

'Trihalomethanes': [[-np.inf, -10e-5, 80, np.inf],

['out\_range\_below', 'potable', 'above']],

'Turbidity': [[-np.inf, -10e-5, 5, np.inf],

['out\_range\_below', 'potable', 'above']]

}

# Exploratory Data Analysis

# 2. EDA (Exploratory Data Analysis)

## 2.1 Objectives

Data preparation and integrity check for the classification task:

- target variable definition

- evaluation of the parameters as discriminators

- check for anomalies in the data

## 2.2 Load Data

path = os.getcwd() + '\\Water\_potability\\'

water = pd.read\_csv(path + 'water\_potability.csv',

index\_col=None,

delimiter=',')

type(water)

water.info()

water.head()

- 3276 entries

- Target: `Potability` indicates if the water is safe for human consumption:

- 1: potable

- 0: non-potable

- Remaining columns are the parameters to classify the entry as potable or not.

- Numerical features:

- float64(9) - parameters

- int64(1) - target

- Categorical:

- only the target variable

- Headers: names are related to the problem.

- Missing values: present - further investigation required.

## 2.3 Missing Values

Counting of the number of missing values in each column.

missing = pd.concat([water.isnull().sum(), 100 \* water.isnull().mean()],

axis=1)

missing.columns = ['count', '%']

missing = missing[missing['count'] > 0]

missing.sort\_values(by='%', ascending=False)

water.dropna().info()

- \*\*Columns with missing data\*\*: 3

- \*\*Missing most\*\*: `Sufalte` - 23.8%

- \*\*Target missing\*\*: none

Removing all the entries with some missing values:

- \*\*remove\*\*: 1265

- \*\*\*keep\*\*\*: 2011

## 2.4 Numeric Data

Evaluating the statical parameters and the correlation among them.

### 2.4.1 Classes and the theoretical potable range

Showcasing the frequency distribution for all data. The theoretical "potable" range of each parameter is highlighted.

Each parameter is plotted twice:

1. all data consolidated

2. data split by the class, where the potable coded as dark green and the non-potable a lighter green. The area hatched in blue is the range considered potable for that physical property of the water.

def visualize\_all(df, n\_bins, r):

fig = plt.figure(figsize=(20, 15))

plt.suptitle('Histograms split by classes of all parameters')

rows = math.ceil(math.sqrt(len(water.keys()) - 1))

cols = math.floor(math.sqrt(len(water.keys()) - 1))

ax\_array = fig.subplots(rows, cols)

i = 0

for param in water.keys():

if param != 'Potability':

i = i + 1

potable = df[df['Potability'] == 1][param]

no\_potable = df[df['Potability'] == 0][param]

ax = ax\_array[math.ceil(i / cols) - 1, ((i - 1) % cols)]

ax.title.set\_text(param)

ax.axvline(x=r[param][0], c='b', ls='--', alpha=.5)

ax.axvline(x=r[param][1], c='b', ls='--', alpha=.5)

ax.hist(potable, bins=n\_bins, color='black', alpha=1, label='potable')

ax.hist(no\_potable,

bins=n\_bins,

color='green',

alpha=0.4,

label='non-potable')

ax.legend()

y\_bottom, y\_top = ax.get\_ylim()

ax.fill\_betweenx([0, y\_top + 10],

r[param][0],

r[param][1],

hatch="///",

facecolor="none",

edgecolor="b",

linewidth=0.01,

alpha=.4)

# ax.set\_xlabel('count')

plt.subplots\_adjust(left=0.1,

bottom=0.1,

right=0.9,

top=0.9,

wspace=0.25,

hspace=0.3)

plt.show()

# Histograms of all numeric features

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

parameters.hist(figsize=(20, 15),color='green',grid=False,edgecolor='white',alpha=.6)

plt.subplots\_adjust(left=0.1,

bottom=0.1,

right=0.9,

top=0.9,

wspace=0.25,

hspace=0.3

);

plt.xlabel('count')

plt.suptitle('Histograms of all parameters')

visualize\_all(water, 30, dict\_range)

- \*\*Distribution\*\*:

- All parameters have a normal distribution

- Potable and non-potable categories are pretty similar: distributions are centered on the same region and same spread over the range

- We cannot find a single parameter capable of differentiating between potable and non-potable

- \*\*Constant values\*\*:

- None

- \*\*Theoretical potable range\*\*:

- By WHO, water with pH < 6.5 is not considered drinkable. However, we have 258 water bodies with pH < 6.5 classified as potable. For example, the pH of gastric acid ranges from 1.5 to 3.5, and we have one sample with the pH equals to 0.22 and classified as drinkable. So something is wrong with the data.

- `Solids`, `Sulfates`, and `Organic Carbon` distributions are entirely out of the potable range.

### 2.4.2 Classification: measured \*vs.\* expected

Following, we have each parameter labeled as \*\*expected\*\*, when the class matches the theoretical classification and \*\*non-expected\*\* otherwise.

There is one column for each measured class and the colors represent labels: dark green is expected and light green non-expected.

The ideal scenario is all the measured class matching the theoretical classification and getting the label expected. So all bars in dark green.

def label\_param(df, param, group\_by\_param, label\_values, label\_names):

df = df[[param, group\_by\_param]]

df = df.dropna()

df[param + '\_cat'] = pd.cut(df[param],

bins=label\_values,

labels=label\_names)

clfd = df

conditions = [

(clfd[group\_by\_param] == 1) & (clfd[param + '\_cat'] == 'potable'),

(clfd[group\_by\_param] == 0) & (clfd[param + '\_cat'] != 'potable'),

(clfd[group\_by\_param] == 1) & (clfd[param + '\_cat'] != 'potable'),

(clfd[group\_by\_param] == 0) & (clfd[param + '\_cat'] == 'potable'),

]

choices = ['expected', 'expected', 'non-expected', 'non-expected']

clfd['classification'] = np.select(conditions, choices, default=0)

ranges = clfd.drop(columns=[param + '\_cat'])

ranges = ranges.groupby([group\_by\_param, 'classification']).agg(['count'])

ranges = ranges[param].reset\_index()

ranges = ranges.pivot(index='classification', columns=group\_by\_param, values='count')

ranges = ranges.fillna(0)

return ranges

def visualize\_all\_labeled(df, group\_by\_param):

rows = math.ceil(math.sqrt(len(df.keys()) - 1))

cols = math.floor(math.sqrt(len(df.keys()) - 1))

fig, axes = plt.subplots(ncols=cols, nrows=rows, figsize=(20, 15))

fig.suptitle('Classification - expected vs. non-expected for each class')

i = 0

for param in water.keys():

if param != 'Potability':

i = i + 1

r = int((i - 1) / cols)

c = i - r \* cols - 1

ax = axes[math.ceil(i / cols) - 1, ((i - 1) % cols)]

ax.title.set\_text(param)

data = label\_param(water, param, group\_by\_param,

dict\_labels[param][0], dict\_labels[param][1])

columns = list(data.keys())

# print(param)

# print(data)

# Get some pastel shades for the colors

n\_entries = len(data)

# print(n\_entries)

colors = plt.cm.summer(np.linspace(0, 0.5, len(data)))

index = np.arange(len(columns)) + 0.3

bar\_width = 0.4

# Initialize the vertical-offset for the stacked bar chart.

y\_offset = np.zeros(len(columns))

# Plot bars and create text labels for the table

cell\_text = []

for e in range(n\_entries):

x = list(data.keys())

# print(x)

y = list(data.iloc[e])

# print(y)

ax.bar(x, y, bar\_width, bottom=y\_offset, color=colors[e])

for n, v in enumerate(y):

ax.text(x[n] - .05,

int(v / 2) + y\_offset[n],

str(int(v)),

color='white',

fontweight='bold')

y\_offset = y\_offset + y

cell\_text.append(['%1.0f' % (x) for x in y])

# Reverse colors and text labels to display the last value at the top.

# colors = colors[::-1]

# cell\_text.reverse()

# Add a table at the bottom of the axes

# the\_table = ax.table(cellText=cell\_text,

# rowLabels=list(data.index),

# rowColours=colors,

# colLabels=['non-potable','potable'],

# loc='bottom')

ax.set\_xticks(list(data.keys()))

ax.set\_xticklabels(['non-potable', 'potable'])

# ax.set\_ylabel('Count')

ax.set\_yticks([])

axes[r, c].legend(['expected', 'non-expected'],

loc='upper center',

frameon=False)

# axes[r,c].legend(loc='upper center', frameon=False, ncol=2)

# plt.legend(list(data.keys()))

# axes[r,c].legend(, title=data.keys(), frameon=False, ncol=2)

plt.subplots\_adjust(left=0.1,

bottom=0.1,

right=0.9,

top=0.9,

wspace=0.15,

hspace=0.25)

# Adjust layout to make room for the table:

plt.subplots\_adjust(left=0.2, bottom=0.2)

plt.show()

visualize\_all\_labeled(water, 'Potability')

The ideal scenario is all the bars in dark green. However, there are values classified as potable in the non-potable bar and vice-versa, what could be considered as errors.

- \*\*Classification correctness:\*\* majority of the classification measured is not agreeing with the expected category taking WHO standards into consideration for all parameters.

- \*\*Trend:\*\* the misclassification of potable and non-potable water is common for all parameters with no preference for any class.

- [rephrase] `Chloramines`, the amount of Chlorine in the water, suggests that the data is composed of water heavily treated once the distribution shows a more prominent occurrence of values above what is considered potable. Supposing that, even after intensive treatment, they were classified as potable because there was no other water source available, and they have to ignore WHO criteria. If data come from somewhere following WHO standards strictly, we could confirm a big issue with the classification data.

Once we have so many errors in classification and we do not have further information about the data source, we can imagine a different scenario. The data came from locations without another source of water, and it was classified as potable simply considering if they were consumed by the population or not.

### 2.4.3. Outliers and Statistical summary grouped by potability

- Visualization of each parameter split into potable and non-potable using a box-plot

- Evaluation of the summary of their statistical metrics.

def visualize\_all\_box(df, group):

rows = math.ceil(math.sqrt(len(df.keys()) - 1))

cols = math.floor(math.sqrt(len(df.keys()) - 1))

fig, axes = plt.subplots(ncols=cols, nrows=rows, figsize=(20, 15))

fig.suptitle('Boxplot of each parameter split by potability')

i = 0

for param in water.keys():

if param != 'Potability':

i = i + 1

r = int((i - 1) / cols)

c = i - r \* cols - 1

sns.boxplot(ax=axes[r, c],

data=df,

x=group,

y=param,

hue=group,

palette='summer\_r'

,showmeans=True, meanprops={"marker": "o", "markerfacecolor": "red", "markeredgecolor": "red"}

,flierprops=dict(marker='.', markerfacecolor='steelblue', markeredgecolor='steelblue', alpha=.5)

,boxprops=dict(alpha=.7)

,width=.5

,notch=True

).set\_title(param)

for line in axes[r,c].get\_lines()[6::7]:

# line.set\_mec('purple') # to test that we have the correct Line2Ds

offsets = line.get\_xdata()

# print(y)

line.set\_xdata(offsets + np.random.uniform(-0.05, 0.05, offsets.size))

axes[r,c].legend(loc='upper center', title=group, frameon=False, ncol=2)

axes[r,c].set\_xticklabels(['non-potable', 'potable'])

plt.subplots\_adjust(left=0.1,

bottom=0.1,

right=0.9,

top=0.9,

wspace=0.15,

hspace=0.25)

plt.show()

visualize\_all\_box(water, 'Potability')

# Statistical summary of the numerical columns

summ = water.describe().transpose()

summ

Boxplots:

- \*\*Median:\*\* values for potable and non-potable are very similar for all the parameters.

- \*\*Quartiles:\*\* have the same range for both classes.

- \*\*Max/Min\*\*: no significant differences between the classes.

- \*\*Outliers\*\*: there are outliers, but they are not so numerous.

Statistical summary:

- \*\*Mean x median\*\*: metrics have close values (expected for normal distributions)

- \*\*Skew (symmetry)\*\*: 25% and 75% are consistently almost symmetrical to the median (expected for normal distributions)

- \*\*Range\*\*: no negative values (per the parameter definitions). The range among all parameters is wide.

- \*\*Anomalies\*\* (e.g., outside the range such as -1 and 999): absent

\*Mean and standard deviation with similar values for both classes indicate a poor discriminator in the classification task.\*

### 2.4.4 Correlation heatmap

A great way to gain a high-level view of relationships amongst the parameters.

#Show a seaborn heatmap of correlations in data

plt.subplots(figsize=(15, 10))

sns.heatmap(water.corr(), annot=True)

- We could correlate `Turbidity` with `Solids` and `Organic\_carbon`, but they do not have a higher correlation in the data.

- No strong correlations were found in the data set.

### 2.4.5 Considerations

Data is showing to be an unreliable source of information for the classification of the water as potable or non-potable:

- \*\*Statistical significance\*\*

- The average, shape, and spread of the distribution of all parameters for potable and non-potable water overlaps. It indicates that statistical analysis won't be meaningful due to a poor statistical significance.

- \*\*Wrong classification\*\*

A considerable amount of data is being classified as potable, outside the presumed drinkable range for all the parameters. However, the opposite also is true; many entries are classified as non-potable while belonging to the presumed potable range.

Although the exploratory data analysis suggests that the classification is wrong, we will consider it correct once we don't have further information about the origin of the data. However, we are not expecting the models' good performance once the potable and non-potable are pretty similar and not the best for the modeling.

## 2.5 Principle components analysis [(PCA)](https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html#sklearn.decomposition.PCA)

The basic steps are:

1. scale the data

2. fit the PCA transformation

3. apply the transformation to the data to create the derived features

4. (optionally) use the derived features to look for patterns in the data and explore the coefficients

### 2.5.1 Scale the data

water\_summary = water.groupby('ph').agg(

# count=pd.NamedAgg(column='Hardness', aggfunc='size'), #could pick any column here

Hardness=pd.NamedAgg(column='Hardness', aggfunc='sum'),

Solids=pd.NamedAgg(column='Solids', aggfunc='sum'),

Chloramines=pd.NamedAgg(column='Chloramines', aggfunc='sum'),

Sulfate=pd.NamedAgg(column='Sulfate', aggfunc='sum'),

Conductivity=pd.NamedAgg(column='Conductivity', aggfunc='sum'),

Organic\_carbon=pd.NamedAgg(column='Organic\_carbon', aggfunc='sum'),

Trihalomethanes=pd.NamedAgg(column='Trihalomethanes', aggfunc='sum'),

Turbidity=pd.NamedAgg(column='Turbidity', aggfunc='sum'),

# Potability=pd.NamedAgg(column='Potability', aggfunc='sum'),

).reset\_index()

# water\_summary.head()

#New dataframe, `db\_summary\_scale` from `db\_summary` whilst setting the index to the selected criteria

df\_summary\_scale = water\_summary.set\_index('ph', drop=True)

water\_summary = water\_summary.set\_index('ph', drop=True)

#Save the index's labels (using the index attribute of `db\_summary\_scale`) into the variable 'db\_summary\_index'

df\_summary\_index = df\_summary\_scale.index

#Save the column names (using the `columns` attribute) of `db\_summary\_scale` into the variable 'db\_summary\_columns'

df\_summary\_columns = df\_summary\_scale.columns

# df\_summary\_scale.head()

df\_summary\_scale.dtypes

- Columns are all numeric

# Scale data

df\_summary\_scale = scale(df\_summary\_scale)

#Create a new dataframe from `db\_summary\_scale` using the column names `db\_summary\_columns`

df\_summary\_scaled\_df = pd.DataFrame(df\_summary\_scale,

columns=df\_summary\_columns)

# df\_summary\_scaled\_df.head()

Validating the scaling:

- Check the average, standard deviation, and unbiased standard deviation estimator (ddof = 1)

#Call `db\_summary\_scaled\_df`'s `mean()` method

validate = df\_summary\_scaled\_df.agg(['mean', 'std']).T

validate['ddof\_1'] = df\_summary\_scaled\_df.std(ddof=0)

validate

- average is zero

- all variables with the same standard deviation

- STD(ddof) = 1 as expected in a scaling

### 2.5.2 PCA transformation

Fit the PCA transformation using the scaled data.

df\_pca = PCA().fit(df\_summary\_scale)

Plot the cumulative variance ratio with number of components.

#Call the `cumsum()` method on the 'explained\_variance\_ratio\_' attribute of `db\_pca` and

#create a line plot to visualize the cumulative explained variance ratio with number of components

#Set the xlabel to 'Component #', the ylabel to 'Cumulative ratio variance', and the

#title to 'Cumulative variance ratio explained by PCA components for state/resort summary statistics'

#Hint: remember the handy ';' at the end of the last plot call to suppress that untidy output

plt.subplots(figsize=(10, 6))

plt.plot(df\_pca.explained\_variance\_ratio\_.cumsum())

plt.xlabel('Component #')

plt.ylabel('Cumulative ratio variance')

plt.title('Cumulative variance ratio explained by PCA components for ph summary statistics');

plt.grid()

- There are no relevant components explaining the variance of the data.

#Call `db\_pca`'s `transform()` method, passing in `db\_summary\_scale` as its argument

df\_pca\_x = df\_pca.transform(df\_summary\_scale)

# df\_pca\_x.shape

Plot the first two derived features (the first two principle components) and label each point.

#Create a dataframe containing the values of the first two PCA components

#Remember the first component was given by db\_pca\_x[:, 0],

#and the second by db\_pca\_x[:, 1]

#Call these 'PC1' and 'PC2', respectively and set the dataframe index to `db\_summary\_index`

pca\_df = pd.DataFrame({

'PC1': df\_pca\_x[:, 0],

'PC2': df\_pca\_x[:, 1]

},

index=df\_summary\_index)

# pca\_df.head()

Concatenating both parts on axis 1 and using the indexes.

water\_avg\_potability = water.groupby('ph')['Potability'].mean()

# water\_avg\_potability.sort\_values(ascending=False).head()

#Use pd.concat to concatenate `pca\_df` and `db\_avg` along axis 1

# remember, pd.concat will align on index

pca\_df = pd.concat([pca\_df, water\_avg\_potability], axis=1)

pca\_df.head()

# pca\_df.dtypes

Checking for missing values

pca\_df[pca\_df.isnull().any(axis=1)]

- No missing values

# Create a seaborn scatterplot by calling `sns.scatterplot`

# Specify the dataframe pca\_df as the source of the data,

# specify 'PC1' for x and 'PC2' for y,

# specify 'target' for the pointsize (scatterplot's `size` argument),

# specify 'catergories' foar `hue`

# specify sorter for `hue\_order`

x = pca\_df.PC1

y = pca\_df.PC2

ph = pca\_df.index

plt.subplots(figsize=(12, 10))

# Note the argument below to make sure we get the colours in the ascending

# order we intuitively expect!

sns.scatterplot(

x='PC1',

y='PC2',

hue='Potability',

data=pca\_df,

hue\_order=pca\_df.Potability.sort\_values(ascending=True).unique())

# sns.scatterplot(x='PC1', y='PC2', size='Potability', hue='Quartile',

# hue\_order=pca\_df.Quartile.cat.categories, data=pca\_df)

#if the amount of data allows, annotate with the labels

# for s, x, y in zip(ph, x, y):

# plt.annotate(s, (x, y))

pc\_var = 100 \* df\_pca.explained\_variance\_ratio\_.cumsum()[1]

plt.title(f'Ski states summary PCA, {pc\_var:.1f}% variance explained');

- No clustering by PCA and potability

- No visible patterns

pd.DataFrame(df\_pca.components\_, columns=df\_summary\_columns)

- No remarkable values could be found in the components of the PCA.

## 2.6 EDA Conclusions

So far, we couldn't find classifiers to define the water as potable.

- Numerical: 10

- Categorical: 0

Data seems to be incorrect for the classification and the distribution for each category. We will carry on the analysis once we don't have more details about the data source. We cannot be sure if the data is wrong or if this a really poor situation for the consumers of this water.

# Modeling

During the data assessment we did not remove any column or adjusted any entries. But we have missing values that must be treated.

## 3.1 Data treatment

Cleansing and adjusting data to be used in the model:

- missing values

- scaling

- splitting the data into training and test sets

### 3.1.1 Missing values

Rows with some missing entry will be removed once we have enough rows with all the entries as shown below.

p = water.groupby('Potability')['Potability'].agg(['count'])

p['percentage'] = 100 \* p['count'] / p['count'].sum()

water.Potability.value\_counts().plot(kind ='pie')

print(len(water))

print()

p

p = water.groupby('Potability')['Potability'].agg(['count'])

p['percentage'] = 100 \* p['count'] / p['count'].sum()

# p['labels'] = p.apply(lambda x: "{x['percentage']} ({x['count']})",axis=1)

p['labels'] = "{p['percentage']} ({p['count']})"

# df['combined']=df.apply(lambda x:'%s\_%s' % (x['foo'],x['bar']),axis=1)

p

# labels = p['count'] + p['percentage']

# labels = '{p:.2f}% ({v:d})'.format(p=p['percentage'],v=p['count'])

# labels

# plt.pie(water, labels=labels, autopct=lambda p : '{:.2f}% ({:,.0f})'.format(p,p \* sum(values)/100))

# water.Potability.value\_counts().plot(kind ='pie')

# print(len(water))

# print()

# p

p1 = water.groupby('Potability')['Potability'].agg(['count'])

p1['percentage'] = 100 \* p1['count'] / p1['count'].sum()

# p1['group'] = 'full'

water\_clean = water.dropna()

p2 = water\_clean.groupby('Potability')['Potability'].agg(['count'])

p2['percentage'] = 100 \* p2['count'] / p2['count'].sum()

# p2['group'] = 'no\_na'

p = pd.concat([p1,p2],axis=1,keys=['full','without\_missing'])

p

# p.T.unstack(0)

first = pd.Series({'True':2316, 'False': 64})

second = pd.Series({'True':2351, 'False': 29})

df = pd.concat([first, second], axis=1, keys=['First pie', 'Second pie'])

df

# p.plot(kind ='pie',subplots=True)

# water.Potability.value\_counts().plot(kind ='pie')

# print(len(water))

# print()

# p.columns[[1, 2]]

# p\_count = p.drop(p.columns[[1, 2]], axis=1, inplace=True)

# p\_count = p[['count']]

# axes = p.plot(kind='pie', subplots=True)

# for ax in axes:

# ax.set\_aspect('equal')

# ax.yaxis.set\_label\_coords(-0.15, 0.5)

# plt.subplots\_adjust(wspace=0.5)

# plt.show()

# p\_count

water\_clean = water.dropna()

p = water\_clean.groupby('Potability')['Potability'].agg(['count'])

p['percentage'] = 100 \* p['count'] / p['count'].sum()

water\_clean.Potability.value\_counts().plot(kind ='pie')

print(len(water))

print()

p

Data with missing entries:

- Total rows: \*\*3276\*\*

- Distribution: 61% non-potable / 39% potable

Removing rows with some missing entry:

- Total rows removed: \*\*1265\*\*

\*\*Data without missing entries\*\*:

- Distribution: 60% non-potable / 40% potable

- Total rows: \*\*2011\*\*

### 3.1.2 Scaling

The parameters have different units, varying by orders of magnitude. Scaling will put them all on a consistent range.

To evaluate any effect of the scaling on the the performance of the models, we will test different scaling methods:

1. [StandardScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.StandardScaler.html) scales each feature to zero mean and unit variance.

2. [MinMaxScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html) scales all the data features in the range [0, 1] or else in the range [-1, 1] if there are negative values in the dataset.

3. [RobustScaler](https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.RobustScaler.html) remove the outliers and then use either \*StandardScaler\* or \*MinMaxScaler\*.

x = water\_clean.drop(['Potability'], axis=1)

y = water\_clean['Potability']

cols = x.columns

std\_scl = StandardScaler()

standard\_df = x

standard\_df[cols] = std\_scl.fit\_transform(x[cols])

min\_max = MinMaxScaler()

minmax\_df = x

minmax\_df[cols] = min\_max.fit\_transform(x[cols])

rbt\_scl = RobustScaler()

robust\_df = x

robust\_df[cols] = rbt\_scl.fit\_transform(x[cols])

scaling = [

('StandardScaler', standard\_df, y),

('MinMaxScaler', minmax\_df, y),

('RobustScaler', robust\_df, y)

]

### 3.1.3 Train/Test sets

- 70/30 train/test split

- Random sampling and uniform sampling (stratified)

sampling = ['random','stratified']

train\_sets = []

for name, x, y in scaling:

parameters = x

target = y

for s in sampling:

if s == 'random':

x\_train, x\_test, y\_train, y\_test = train\_test\_split(

parameters,

target,

test\_size=0.3,

random\_state=47

)

else:

x\_train, x\_test, y\_train, y\_test = train\_test\_split(

parameters,

target,

test\_size=0.3,

random\_state=47,

stratify=y

)

train\_sets.append([name,s,x\_train, x\_test, y\_train, y\_test])

# train\_sets

split = [

('Data shape:', water\_clean.shape),

('Train set - parameters:', x\_train.shape),

('Train set - target:', y\_train.shape),

('Test set - parameters:', x\_test.shape),

('Test set - target:', y\_test.shape)

]

split

#Check the `dtypes` attribute of `x\_train` to verify all features are numeric

print()

print('Data train types:'),

print(x\_train.dtypes)

#Repeat this check for the test split in `x\_test`

print()

print('Data test types:'),

print(x\_test.dtypes)

- All nine parameters are numeric attributes for both the training and the test data sets

- Total rows: 2011

- Training set rows: 1407

- Test set rows: 604

## 3.2 Classifiers

We evaluated six supervised classification models:

1. \*\*Logistic Regression (Logit model)\*\*

- It is used to model the probability of well defined classes or events (categorical), that can be binary or linear. This model is widely used for classification.

- \*Parameters\*

- \*random state:\* seed to get the same results when running the model multiple times.

2. \*\*K Nearest Neighbours (KNN)\*\*

- KNN uses the distance between the values to group them assuming that the data is similar when they are close.

- Parameters

- \*n\_neighbors:\* number of neighbors used.

3. \*\*Decision Tree\*\*

- Decision trees create a set of rules to make the decision. These rules are evaluated and based on the decision you move to the next rule (node) till you reach the to the final classification (leaf).

- Parameters

- \*criterion:\* the function to measure the quality of a split. Gini for impurity and Entropy for the information gain.

- \*max\_depth:\* the maximum depth of the tree.

- \*min\_samples\_leaf:\* the minimum number of samples required to split an internal node.

4. \*\*Random Forest\*\*

(https://towardsdatascience.com/understanding-random-forest-58381e0602d2)

- Random forest, consists of a large number of individual decision trees that operate as an ensemble.

- Parameters

- \*n\_estimators:\* the number of trees in the forest.

- \*min\_samples\_leaf:\* the minimum number of samples required to split an internal node.

5. \*\*AdaBoost\*\*

(https://www.datacamp.com/community/tutorials/adaboost-classifier-python)

- Ada-boost or Adaptive Boosting is one of ensemble boosting classifier proposed by Yoav Freund and Robert Schapire in 1996. It combines multiple classifiers to increase the accuracy of classifiers.

- Parameters

- \*n\_estimators:\* the maximum number of estimators at which boosting is terminated. In case of perfect fit, the learning procedure is stopped early.

- \*learning\_rate:\* weight applied to each classifier at each boosting iteration. A higher learning rate increases the contribution of each classifier. There is a trade-off between the `learning\_rate` and `n\_estimators parameters`.

6. \*\*XGBoost\*\*

(https://www.datatechnotes.com/2019/07/classification-example-with.html)

- The XGBoost stands for eXtreme Gradient Boosting, which is a boosting algorithm based on gradient boosted decision trees algorithm.

- Parameters

- \*n\_estimators:\* number of boosting rounds.

- \*learning\_rate:\* boosting learning rate.

<!-- 7. Bagging -->

### 3.2.1 Declaration

Creation of the objects for each model.

## MODEL DECLARATION

lr = LogisticRegression(random\_state=42)

knn = KNeighborsClassifier()

dt = DecisionTreeClassifier()

rf = RandomForestClassifier()

ada = AdaBoostClassifier()

xgb = XGBClassifier(use\_label\_encoder=False, eval\_metric='logloss')

# bagging = BaggingClassifier()

### 3.2.2 Parameters

Selecting a range of the most common parameters to be tested for each model.

## PARAMETERS

# 'parameters' of logistic regression

params\_lr = {'random\_state': [42]}

# parameters of knn

params\_knn = {'n\_neighbors': np.arange(1, 50)}

# parameters for decision tree

# "gini" for the Gini impurity and “entropy” for the information gain.

# min\_samples\_leaf: The minimum number of samples required to be at a leaf node, have the effect of smoothing the model

params\_dt = {

'criterion': ['gini', 'entropy'],

'max\_depth': np.arange(1, 50),

'min\_samples\_leaf': [1, 2, 4, 7, 10, 20, 30, 40, 70, 100]

}

# parameters for Random Forest

# n\_estimators: The number of trees in the forest.

params\_rf = {

'n\_estimators': [100, 200, 350, 500],

'min\_samples\_leaf': [2, 5, 10, 30]

}

# parameters fpr AdaBoost

params\_ada = {

'n\_estimators': [50, 100, 250, 400, 500, 600],

'learning\_rate': [0.2, 0.4, 0.6, 0.8, 1]

}

# parameters for XGBoost

params\_xgb = {

'n\_estimators': [50, 100, 250, 400, 600, 800, 1000],

'learning\_rate': [0.2, 0.4, 0.6, 0.8, 1]

}

### 3.2.3 Grid

Defining a grid with all the parameters to figure out the best ones.

## GRID PREPARATION

grid\_lr = GridSearchCV(lr, param\_grid=params\_lr)

grid\_knn = GridSearchCV(knn, param\_grid=params\_knn, cv=5) #search knn for 5 fold cross validation

grid\_dt = GridSearchCV(dt, param\_grid=params\_dt, cv=5) #grid search decision tree for 5 fold cv

grid\_rf = GridSearchCV(rf, param\_grid=params\_rf, cv=5)

grid\_ada = GridSearchCV(ada, param\_grid=params\_ada, cv=5)

rs\_xgb = RandomizedSearchCV(xgb, param\_distributions=params\_xgb, cv=5)

grids = [

('Logistic Regression', grid\_knn, lr), # grid\_knn to fill the

('K Nearest Neighbours', grid\_knn, knn),

('Decision Tree', grid\_dt, dt),

('Random Forest', grid\_rf, rf),

('AdaBoost', grid\_ada, ada),

('XGBoost', rs\_xgb, xgb)

]

# grids

### 3.2.4 Training

Running the models for all the grid of parameters the parameters with the best accuracy.

## TRAINING TO FIND THE BEST PARAMATERS

c = 0

d = []

for i, item in enumerate(grids):

classifier\_name = item[0]

grid = item[1]

classifier = item[2]

for scaler, pick, x\_train, x\_test, y\_train, y\_test in train\_sets:

c = c + 1

print (str(c) + '/' + str(len(train\_sets)\*len(grids)) + ': ' + classifier\_name + ' -> ' + scaler + ' - ' + pick)

t\_start = time.time()

if classifier\_name != 'XGBoost':

grid.fit(x\_train, y\_train)

else:

grid.fit(x\_train, y\_train, eval\_metric='logloss')

# print("Best parameters for " + classifier\_name + ":", grid.best\_params\_)

t\_end = time.time()

# Once the grid keep the best parameters as default,

# we can evaluate the accuracy of each model checking the accuracy of the model with the test set

t\_start\_test = time.time()

# Evaluate classifier's accuracy on the test set

y\_pred = grid.predict(x\_test)

accuracy = accuracy\_score(y\_test, y\_pred)

# print('{:s} : {:.2f}'.format(classifier\_name, accuracy))

t\_end\_test = time.time()

d.append({

'scaling\_method': scaler,

'sampling\_method': pick,

'classifier': classifier\_name,

'accuracy': accuracy,

'start\_training': datetime.fromtimestamp(t\_start).strftime('%Y-%m-%d %H:%M:%S'),

'end\_training': datetime.fromtimestamp(t\_end).strftime('%Y-%m-%d %H:%M:%S'),

'run\_time': np.abs(t\_start - t\_end),

'start\_test': datetime.fromtimestamp(t\_start\_test).strftime('%Y-%m-%d %H:%M:%S'),

'end\_test': datetime.fromtimestamp(t\_end\_test).strftime('%Y-%m-%d %H:%M:%S'),

'run\_time\_seconds\_test': np.abs(t\_start\_test - t\_end\_test),

'best\_parameters': grid.best\_params\_

})

# if classifier\_name == 'Decision Tree':

# bagging = BaggingClassifier(classifier,

# n\_estimators=100,

# random\_state=42)

# bagging.fit(x\_train, y\_train)

# t\_end\_test = time.time()

# t\_start\_test = time.time()

# y\_pred = bagging.predict(x\_test)

# accuracy = accuracy\_score(y\_test, y\_pred)

# t\_end\_test = time.time()

#

# d.append({

# 'scaling\_method': scaler,

# 'sampling\_method': pick,

# 'classifier': 'Bagging',

# 'accuracy': accuracy,

# 'start\_training': datetime.fromtimestamp(t\_start).strftime('%Y-%m-%d %H:%M:%S'),

# 'end\_training': datetime.fromtimestamp(t\_end).strftime('%Y-%m-%d %H:%M:%S'),

# 'run\_time': np.abs(t\_start - t\_end),

# 'start\_test': datetime.fromtimestamp(t\_start\_test).strftime('%Y-%m-%d %H:%M:%S'),

# 'end\_test': datetime.fromtimestamp(t\_end\_test).strftime('%Y-%m-%d %H:%M:%S'),

# 'run\_time\_seconds\_test': np.abs(t\_start\_test - t\_end\_test),

# 'best\_parameters': grid.best\_params\_

# })

profiling\_grid = pd.DataFrame(d)

### 3.2.5 Performance

Models sorted by accuracy and the best parameters for each model.

# Performance

# from IPython.display import display, HTML

# display(HTML(df.to\_html(index=False)))

final\_results = profiling\_grid.sort\_values('accuracy', ascending=False).drop(columns=['start\_training','end\_training','start\_test','end\_test']).reset\_index(drop=True)

# a = profiling\_grid.loc[:,['scaling\_method','sampling\_method', 'classifier','accuracy','run\_time\_seconds\_training','best\_parameters']].reset\_index(drop=True)

# a.sort\_values('accuracy', ascending=False)

final\_results

### 3.2.6 Modeling results

fig, axes = plt.subplots(ncols=1, nrows=1, figsize=(15, 15))

sns.scatterplot(ax=axes, x='run\_time', y='accuracy', data=profiling\_grid, s=500, alpha =0.7, hue='classifier');

plt.grid()

- `Random Forest` had the best accuracy for all sets of scenarios, and the time to train the model was in the average of all classifiers

- `KNN` was the fastest model to train but also the worst accuracy

- `AdaBoost` was the slowest, but the extra time did not convert in better accuracy, having poor results

- `XGBoost` was faster and slightly better than `Decision Trees`

def add\_line(ax, xpos, ypos):

line = plt.Line2D([xpos, xpos], [ypos + .1, ypos],

transform=ax.transAxes,

color='gray')

line.set\_clip\_on(False)

ax.add\_line(line)

def label\_len(my\_index, level):

labels = my\_index.get\_level\_values(level)

return [(k, sum(1 for i in g)) for k, g in groupby(labels)]

def label\_group\_bar\_table(ax, df, adj):

ypos = -.1

scale = 1. / df.index.size

for level in range(df.index.nlevels)[::-1]:

pos = 0

for label, rpos in label\_len(df.index, level):

r = 0

if rpos == 1 and adj > 0:

r = 90

lxpos = (pos + .5 \* rpos) \* scale

ax.text(lxpos,

ypos - adj,

label,

ha='center',

rotation=r,

transform=ax.transAxes)

add\_line(ax, pos \* scale, ypos)

pos += rpos

add\_line(ax, pos \* scale, ypos)

ypos -= .1

# for tick in ax.get\_xticklabels():

# tick.set\_rotation(0)

def multi\_index\_x\_label(data, adj):

fig = plt.figure(figsize=(20, 15))

ax = fig.add\_subplot(111)

ax.set\_ylabel('Accuracy')

data.plot(kind='bar', ax=fig.gca())

# Remove default labels

ax.set\_xticklabels('')

ax.set\_xlabel('')

label\_group\_bar\_table(ax, df, adj)

fig.subplots\_adjust(bottom=.1 \* df.index.nlevels)

plt.legend(loc='upper center',

title='',

frameon=False,

ncol=len(df.columns))

# plt.legend(np.array(list(df.columns))[:,1])

plt.show()

# multi\_index\_x\_label(df,.05)

performance = profiling\_grid.loc[:,['scaling\_method','sampling\_method', 'classifier','accuracy']]

df = performance.groupby(['sampling\_method','scaling\_method','classifier']).accuracy.sum().unstack(0)

multi\_index\_x\_label(df,.05)

- \*\*Random sampling\*\* performs slightly better than \*\*Stratified sampling\*\* for all classifiers and scaling techniques.

df = performance.groupby(['scaling\_method','classifier','sampling\_method']).accuracy.sum().unstack(0)

multi\_index\_x\_label(df,0)

- \*\*StandardScaler\*\*, \*\*MinMaxScaler\*\*, and \*\*Robust Scaler\*\* do not change the accuracy of the classifiers in general. `Decision Tree`, `Random Forest`, `XGBoost` accuracy were affected by the scaling method adopted, but the differences were not substantial.

df = performance.groupby(['classifier','sampling\_method','scaling\_method']).accuracy.sum().unstack(0)

multi\_index\_x\_label(df,0)

- `Random Forest` had the best accuracy for any sampling or scaling method.

- When using the stratified sampling `Random Forest`, accuracy was on average 10% better than the other classifiers.

# Conclusion

The goal of the project was to classify samples from water bodies as potable or not. We tested six different classifiers exploring a large set of parameters and evaluating three scaling methods and two sampling techniques.

\*\*Modeling\*\*

- The best model was the `Random Forest` adopting the parameters 'min\_samples\_leaf': 2 and 'n\_estimators': 500 achieving an accuracy of \*70%\*.

- The sampling was the \*random\* technique, and the data scaling with \*MinMaxScaler\*.

- This model took approximately 70 seconds to run.

See below the top 5 classifiers.

\*\*Data\*\*

Besides reaching 69% of accuracy, the data quality raises some concerns and could be classified as unreliable data with poor data quality:

- Classification measured is discrepant to the expected classification.

- All parameters have a heavy overlapping distribution for potable and non-potable.

- Absence of correlation between parameters that should be correlated, such as the known correlation between pH and potability.

\*\*Final thoughts\*\*

The physical properties presented are accurate predictors of portability, as can be found in the academic literature. However, the analysis of this data showed us a substantial discrepancy between the expected and the measured. Unless we deal with data from areas without any alternatives, those samples should never be classified as potable. So the only reasonable explanation is that the classification field is wrong.

\*\*Conclusion\*\*

Our solution should not be used as a good predictor for water potability based on the physical properties of the water due to this doubt about the data quality.

\*\*Alternative solution\*\*

Solving this problem would demand reclassifying all the samples properly if the measures are correct and only the classification is incorrect, or redo all the sampling and measurements.

final\_results.loc[:4,:]