**RELAZIONE del progetto di "INGEGNERIA DELLA CONOSCENZA"**

**Realizzato da:**
**Rigante Chiara (matricola 708209)**
**Scalzo Andrea (matricola 705830)**

## Sommario:



Repository Github: https://github.com/andreascalzo99/Water

# Contesto

L'accesso all'acqua potabile è essenziale per la salute, infatti è riconosciuto come un diritto umano fondamentale. Questo è importante per la salute e lo sviluppo a livello nazionale, regionale e locale. In alcune regioni, è stato dimostrato che gli investimenti nell'approvvigionamento idrico e nei servizi igienico-sanitari possono produrre un beneficio economico netto, poiché le riduzioni degli effetti negativi sulla salute e dei costi sanitari superano i costi di attuazione degli interventi.

Per questo motivo, l’obiettivo del progetto è stato cercare un classificatore, abbastanza performante in base al dataset scelto, per fare in modo che in futuro, inserendo dei dati, sia possibile ottenere un risultato di potabilità, partendo dalle caratteristiche del corpo idrico scelto.

Rigante Chiara (matricola 708209)
Scalzo Andrea (matricola 705830)

# Descrizione dataset

Per analizzare i differenti modelli di apprendimento supervisionato, si è scelto il dataset *Water Potability*, riguardante informazioni circa la potabilità dell'acqua. Quest'ultimo raccoglie informazioni in forma numerica per la risoluzione di un problema di classificazione binaria, quindi classificando l'acqua come potabile e non potabile.
Il file ***"water_potability.csv"*** contiene metriche sulla qualità dell'acqua per 3276 diversi corpi idrici.
Le feature coinvolte nel dataset sono:

- Ph ;
- Durezza;
- Solidi totali disciolti;
- Clorammine;
- Solfati;
- Conducibilità;
- Carbonio organico;
- Trialometani;
- Potabilità (Target Feature).

```
[ ]
    df = pd.read_csv('/content/water_potability.csv')
    df.head()
```

| | ph | Hardness | Solids | Chloramines | Sulfate | Conductivity | Organic_carbon | Trihalomethanes | Turbidity | Potability |
|---|---|---|---|---|---|---|---|---|---|---|
| **0** | NaN | 204.890455 | 20791.318981 | 7.300212 | 368.516441 | 564.308654 | 10.379783 | 86.990970 | 2.963135 | 0 |
| **1** | 3.716080 | 129.422921 | 18630.057858 | 6.635246 | NaN | 592.885359 | 15.180013 | 56.329076 | 4.500656 | 0 |
| **2** | 8.099124 | 224.236259 | 19909.541732 | 9.275884 | NaN | 418.606213 | 16.868637 | 66.420093 | 3.055934 | 0 |
| **3** | 8.316766 | 214.373394 | 22018.417441 | 8.059332 | 356.886136 | 363.266516 | 18.436524 | 100.341674 | 4.628771 | 0 |
| **4** | 9.092223 | 181.101509 | 17978.986339 | 6.546600 | 310.135738 | 398.410813 | 11.558279 | 31.997993 | 4.075075 | 0 |

# Analisi esplorativa del dataset

Dopo aver effettuato il caricamento del dataset in memoria, mediante alcuni comandi è stato possibile effettuare delle osservazioni.

- Comando: describe( )
  Per ottenere informazioni statistiche inerenti ciascuna feature a disposizione, tale comando si occupa del calcolo.

```
[ ] df.describe()
```

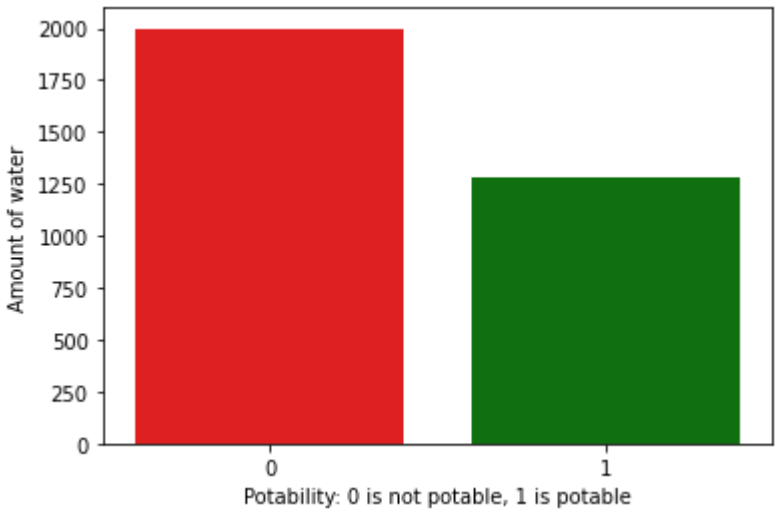
| | ph | Hardness | Solids | Chloramines | Sulfate | Conductivity | Organic_carbon | Trihalomethanes | Turbidity | Potability |
|---|---|---|---|---|---|---|---|---|---|---|
| **count** | 2785.000000 | 3276.000000 | 3276.000000 | 3276.000000 | 2495.000000 | 3276.000000 | 3276.000000 | 3114.000000 | 3276.000000 | 3276.000000 |
| **mean** | 7.080795 | 196.369496 | 22014.092526 | 7.122277 | 333.775777 | 426.205111 | 14.284970 | 66.396293 | 3.966786 | 0.390110 |
| **std** | 1.594320 | 32.879761 | 8768.570828 | 1.583085 | 41.416840 | 80.824064 | 3.308162 | 16.175008 | 0.780382 | 0.487849 |
| **min** | 0.000000 | 47.432000 | 320.942611 | 0.352000 | 129.000000 | 181.483754 | 2.200000 | 0.738000 | 1.450000 | 0.000000 |
| **25%** | 6.093092 | 176.850538 | 15666.690297 | 6.127421 | 307.699498 | 365.734414 | 12.065801 | 55.844536 | 3.439711 | 0.000000 |
| **50%** | 7.036752 | 196.967627 | 20927.833607 | 7.130299 | 333.073546 | 421.884968 | 14.218338 | 66.622485 | 3.955028 | 0.000000 |
| **75%** | 8.062066 | 216.667456 | 27332.762127 | 8.114887 | 359.950170 | 481.792304 | 16.557652 | 77.337473 | 4.500320 | 1.000000 |
| **max** | 14.000000 | 323.124000 | 61227.196008 | 13.127000 | 481.030642 | 753.342620 | 28.300000 | 124.000000 | 6.739000 | 1.000000 |

- Comando: isnull( )
  Per ottenere informazioni circa i valori nulli presenti nel dataset, infatti nelle feature ph, trialometani e solfati erano presenti.

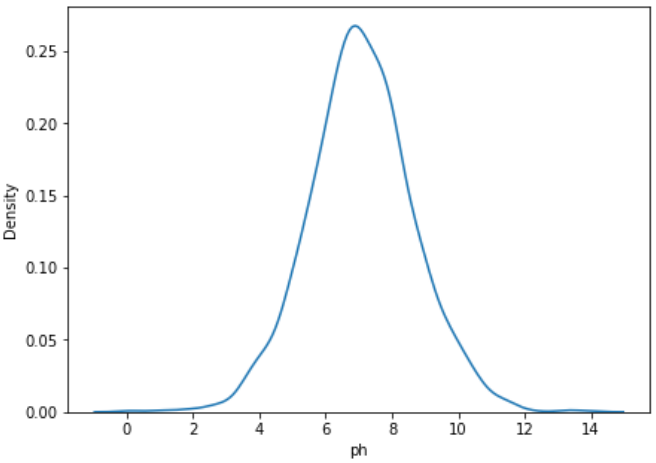
```
[ ] #Visualizzazione dei valori nulli
    df.isnull().sum()
```

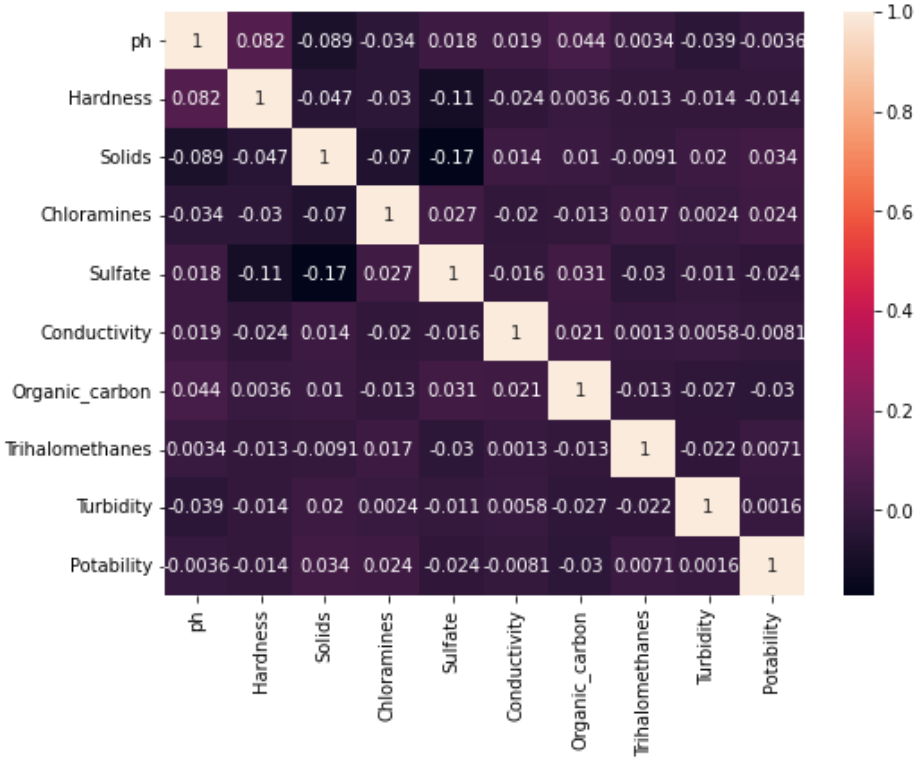
```
ph                491
Hardness            0
Solids              0
Chloramines         0
Sulfate           781
Conductivity        0
Organic_carbon      0
Trihalomethanes   162
Turbidity           0
Potability          0
dtype: int64
```

Inoltre, mediante la libreria “plotly”, si è rappresentato il dataset in riferimento alla potabilità, target feature, mostrando un dataset sbilanciato, in quanto ci sono molti corpi idrici non potabili.




Continuando con l’analisi delle feature, è stato opportuno rappresentare le loro funzioni di densità di probabilità, in quanto sappiamo che la funzione di densità, non è una probabilità, è però una funzione legata ad essa, perché se si vuole calcolare la probabilità che la variabile casuale continua X appartenga ad un intervallo, basta che si faccia l'integrale della funzione di densità.






E' importante comprendere che genere di correlazione ci sia tra le feature, per questo è stata utilizzata una matrice di correlazione con indice r di Pearson. L'indice di correlazione di Pearson è un numero che fornisce informazioni sia sulla forza sia sulla direzione della correlazione tra due variabili quantitative. I valori coinvolti nella nostra matrice sono quasi equivalenti a 0, questo implica che non ci siano forti correlazioni, sia negativamente che positivamente. Per questo è necessario che tutte le variabili siano presenti.

# Pre-processing del dataset

Avendo individuato valori nulli all'interno del nostro dataset, è stato scelto di sostituire tali valori, utilizzando il "Knn Imputer". Prima di poter proseguire con l'algoritmo, si è ritenuto opportuno standardizzare i valori numerici delle feature mediante l'apposito metodo **StandardScaler( )**.
KNN Imputer è stato supportato per la prima volta da Scikit-Learn nel dicembre 2019 quando ha rilasciato la sua versione 0.22. Questo imputer utilizza il metodo k-Nearest Neighbors per sostituire i valori mancanti nei dataset con il valore medio del parametro 'n_neighbors' più vicino trovato nel training set. Per impostazione predefinita, utilizza una metrica della distanza euclidea per imputare i valori mancanti. Nel nostro caso è stato scelto un parametro **n_neighbors** pari a 5.

```
[ ] from sklearn.preprocessing import StandardScaler

    #Standardizing The Data
    sc = StandardScaler()
    X = df.drop(['Potability'],axis=1)
    X = sc.fit_transform(X)
```

```
[ ] # Use KNN Imputer to impute NaN Values
    #per i valori nulli
    from sklearn.impute import KNNImputer

    imputer = KNNImputer(n_neighbors = 5)
    df[['ph','Sulfate','Trihalomethanes']] = imputer.fit_transform(df[['ph','Sulfate','Trihalomethanes']])
```

```
[ ] df.isnull().sum()

    ph                0
    Hardness          0
    Solids            0
    Chloramines       0
    Sulfate           0
    Conductivity      0
    Organic_carbon    0
    Trihalomethanes   0
    Turbidity         0
    Potability        0
    dtype: int64
```

Rigante Chiara (matricola 708209)
Scalzo Andrea (matricola 705830)

# Classificazione

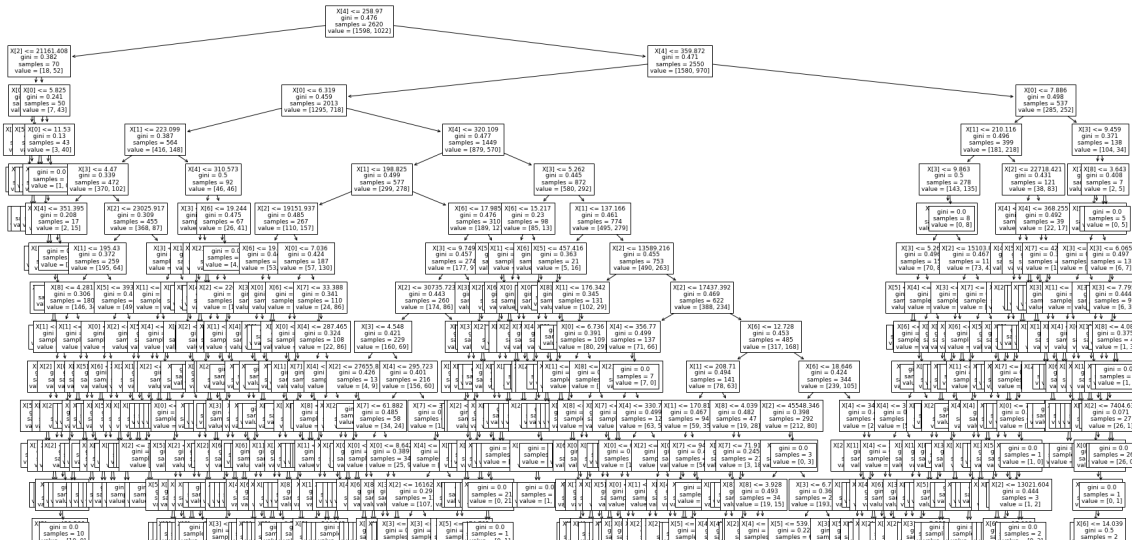
Per la classificazione sono stati scelti diversi algoritmi, in modo tale da offrire una panoramica abbastanza completa per la predizione della potabilità dei corpi idrici.
In primis, però, il nostro dataset è stato suddiviso in Train e in Test, per poter successivamente ottenere dei risultati dai vari classificatori. Il test è rappresentato dal 20% del dataset, in più è stato impostato un **random_state a 42**. Nella parte di train è stata rimossa la feature “Potabilità”, in quanto rappresenta la target feature, quindi costituente del test. Successivamente, è iniziata la parte di classificazione con i vari algoritmi.

### Decision Tree Classifier

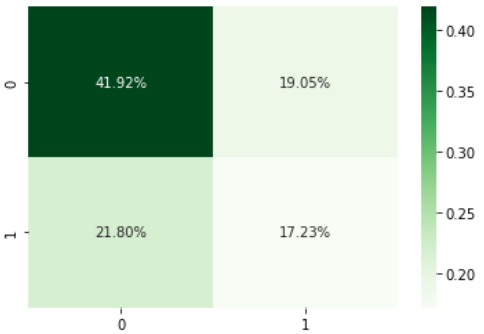
Il primo algoritmo utilizzato è Decision Tree Classifier, essenzialmente è un albero in cui:

- ogni nodo interno (non foglia) è etichettato con una condizione, una funzione booleana degli esempi ;
- ogni nodo interno ha due figli, uno etichettato con *TRUE* e l'altro con *FALSE* ;
- ogni foglia dell'albero è etichettata con una stima puntuale sulla classe.

Ogni condizione incontrata nell'albero viene valutata e viene seguito l'arco corrispondente al risultato. Quando viene raggiunta una foglia, viene restituita la classificazione corrispondente a quella foglia. Un albero decisionale corrisponde a una struttura if-then-else annidata in un linguaggio di programmazione.

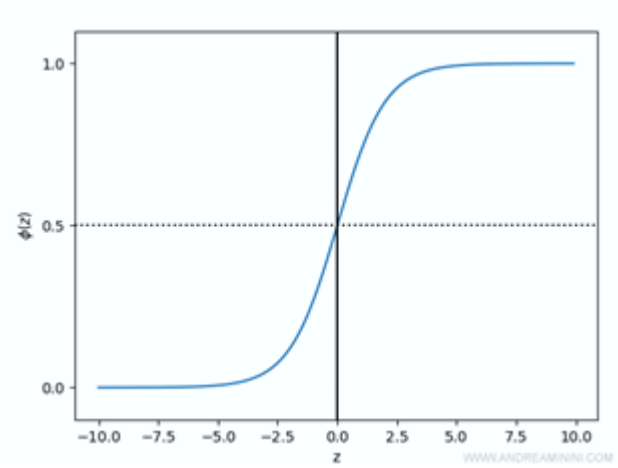


I valori riportati sono stati un training score pari a 100%, significato di un addestramento ben riuscito, ma un’accuratezza pari al 59%. Con la matrice di correlazione, infatti, è evidente come l’algoritmo riesca a riconoscere corpi idrici non potabili, ma ha difficoltà a riconoscere elementi potabili, in quanto c’è una percentuale maggiore di falsi negativi, rispetto ai veri negativi.



## Logistic Regression

Proseguendo, è stato applicato l'algoritmo di classificazione denominato "Logistic Regression": un algoritmo di apprendimento supervisionato che costruisce un modello probabilistico lineare di classificazione dei dati. Nella fase di addestramento, l'algoritmo di regressione logistica prende in input n esempi da un insieme di training (X). Ogni singolo esempio è composto da *m* attributi e dalla classe corretta di appartenenza. Durante l'addestramento l'algoritmo elabora una distribuzione di pesi (W) che permetta di classificare correttamente gli esempi con le classi corrette. Poi calcola la combinazione lineare *z* del vettore dei pesi *W* e degli attributi xm. La combinazione lineare z viene passata alla funzione logistica (sigmoid) che calcola la probabilità di appartenenza del campione alle classi del modello. È la tipica curva a S della distribuzione delle probabilità.




Sono stati ottenuti valori pari a Training score 61% e Accuracy score del 61%.
Per poter ottimizzare i valori ottenuti, si è optato, mediante Grid Search Cv, una ricerca esaustiva sui valori degli iperparametri passati, di trovarne migliori. Gli iperparametri presi in considerazione sono stati

- **penalty:** specifica la norma della sanzione di un termine ;
- **c:** inverso della forza di regolarizzazione;
- **solver:** Algoritmo da utilizzare nel problema di ottimizzazione;
- **max_iter:** Numero massimo di iterazioni necessarie per far convergere i solutori.

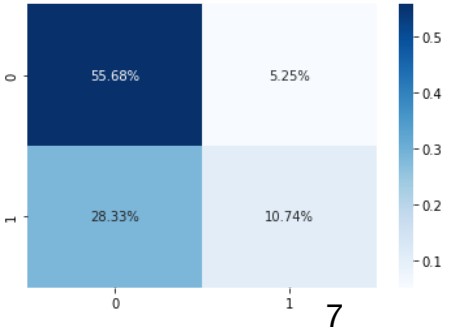
Con i nuovi iperparametri, si è ottenuto un leggero aumento del training score, ma l'accuratezza è rimasta invariata.

## Random Forest Classifier

Il prossimo algoritmo è Random Forest Classifier, facente parte degli algoritmi Ensemble, poiché ottenuto dall'aggregazione tramite bagging di alberi di decisione. Le foreste casuali si pongono come soluzione che minimizza l'overfitting del training set rispetto agli alberi di decisione. Il classificatore della foresta casuale crea un insieme di alberi decisionali da un sottoinsieme selezionato casualmente del set di addestramento. È fondamentalmente *un insieme di alberi decisionali (DT) da un sottoinsieme selezionato casualmente del training set e quindi raccoglie i voti da diversi alberi decisionali per decidere la previsione finale.*
I dati risultanti: training score= 87% e accuratezza=66%. Questo modello mostra delle prestazioni migliori rispetto agli altri modelli presentati finora.
Anche in questo caso, è stata utilizzata la tecnica di Grid Search Cv per ottenere i migliori iperparametri. Gli iperparametri passati per la ricerca sono stati :

- **n_estimators:** Il numero di alberi nella foresta;
- **max_features:** Il numero di feature da considerare quando si cerca la migliore suddivisione.

I valori ottenuti in questo caso sono leggermente diversi e migliorati, rispetto al caso precedente, infatti si ha un training score pari al 93% e un'accuratezza del 67%.



## Support Vector Machine

Successivamente, è stato adoperato il Support Vector Classifier, nell'apprendimento automatico, le macchine vettoriali di supporto (SVM, anche reti vettoriali di supporto) sono modelli di apprendimento supervisionato con algoritmi di apprendimento associati che analizzano i dati utilizzati per la classificazione e l'analisi di regressione.
Una Support Vector Machine (SVM) è un classificatore discriminativo formalmente definito da un iperpiano di separazione. In pratica, dati i dati di addestramento etichettati (apprendimento supervisionato), l'algoritmo produce un iperpiano ottimale che categorizza nuovi esempi.
Questo algoritmo, ha prodotto valori uguali per accuratezza e training score, pari al 61%.
Per permettere la sintonizzazione degli iperparametri, mediante Grid Search CV, sono stati scelti:

- **C:** Parametro di regolarizzazione ;
- **gamma:** Coefficiente del kernel ;
- **kernel:** Specifica il tipo di kernel da utilizzare nell'algoritmo.

I risultati hanno riportato un aumento dell'accuratezza al 67%, mentre il training score è migliorata di qualche decimo, restando equivalente al 61%.

## K-Nearest Neighbors

K-Nearest Neighbors è uno degli algoritmi di classificazione più basilari ma essenziali nell'apprendimento automatico. Appartiene al dominio dell'apprendimento supervisionato e trova un'applicazione intensa. I valori restituiti sono stati 61% per entrambe le metriche.
Passando al miglioramento, alla Grid Search CV, sono stati utilizzati:

- **n_neighbors:** Numero di vicini da considerare ;
- **weights:** Funzione peso utilizzata nella previsione;
- **matric:** La metrica della distanza da utilizzare per l'albero.

Dopo una ricerca esaustiva, i migliori iperparametri hanno riportato valori nettamente migliorati, ovvero 92% di training score e 67% di accuratezza.

## Gaussian NB

L'algoritmo successivo è Gaussian NB, facente parte dei classificatori Naive Bayes, ovvero una raccolta di algoritmi di classificazione basati sul Teorema di Bayes. Nel Gaussian Naive Bayes, si assume che i valori continui associati a ciascuna caratteristica siano distribuiti secondo una distribuzione gaussiana. Una distribuzione gaussiana è anche chiamata distribuzione normale. Quando viene tracciato, fornisce una curva a campana che è simmetrica rispetto alla media dei valori delle feature. Si presume che la probabilità delle caratteristiche sia gaussiana, quindi la probabilità condizionata è data da: $P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} exp\left(-\frac{(x_i-\mu_y)^2}{2\sigma_y^2}\right)$
Per quanto riguarda la predizione dell'algoritmo, ha ottenuto il 63% del training score e il 62% di accuratezza. Al fine di migliorare, l'iperparametro passato alla Grid Search CV, è stato var_smoothing, porzione della varianza maggiore di tutte le feature che viene aggiunta alle varianze per la stabilità del calcolo. La sintonizzazione di questo valore, ha portato una diminuzione nel punteggio del training score a 62,5%, anche nell'accuratezza si verifica lo stesso fenomeno arrivando a 61,1%.

## Algoritmi Ensemble - Boosting

I successivi algoritmi fanno parte degli algoritmi ensemble, ma in particolare utilizzano la tecnica di boosting, è una tecnica di modellazione che tenta di costruire un classificatore forte dal numero di classificatori deboli. Ciascun classificatore influisce sulla votazione finale con un certo peso. Tale peso sarà calcolato in base all'errore di accuratezza che ciascun modello commetterà in fase di learning. In primo luogo, viene creato un modello dai dati di addestramento. Quindi viene costruito il secondo modello che cerca di correggere gli errori presenti nel primo modello. Questa procedura viene continuata e i modelli vengono aggiunti fino a quando non viene previsto correttamente il set di dati di addestramento completo o viene aggiunto il numero massimo di modelli.

### Gradient Boosting Classifier

Il primo modello è Gradient Boosting Classifier, ogni predittore cerca di migliorare il suo predecessore riducendo gli errori. Ma l'idea affascinante alla base del Gradient Boosting è che invece di adattare un predittore sui dati ad ogni iterazione, in realtà si adatta un nuovo predittore *agli errori residui fatti dal predittore precedente.* Per fare previsioni iniziali sui dati, l'algoritmo otterrà il *log delle probabilità* della target feature. Di solito è il numero di valori Veri (valori uguali a 1) diviso per il numero di valori Falsi (valori uguali a 0).

- Per ogni istanza nel training set, calcola i *residui* per quell'istanza, o, in altre parole, il valore osservato meno il valore previsto;
- Una volta fatto ciò, costruisce un nuovo Albero decisionale che cerca effettivamente di prevedere i residui calcolati in precedenza.

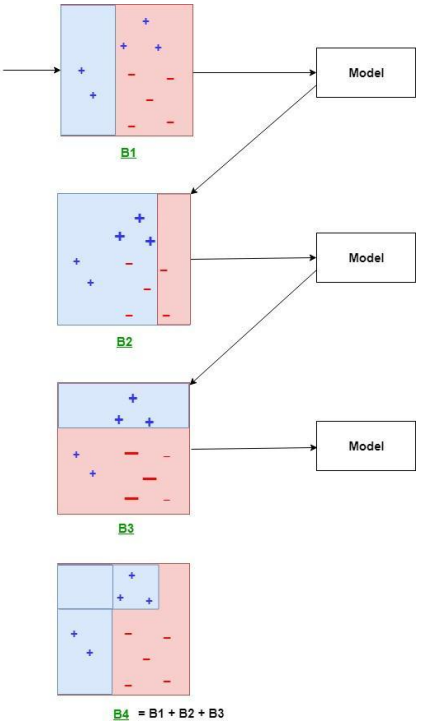
I valori ottenuti da tale modello sono 75% per il training score e 65% per l'accuratezza. Anche qui si è cercato di migliorare i valori, passando i seguenti iperparametri alla Grid Search CV:

- **learning_rate :** questo parametro scala il contributo di ogni albero ;
- **max_depth :** la profondità massima di ogni albero;
- **n_estimators :** il numero di alberi da costruire.

I risultati hanno mostrato una diminuzione del training score, ottenendo 74%, mentre l'accuratezza è migliorata ottenendo il 67%.

### AdaBoost

Il secondo modello osservato è Adaboost, ogni qual volta un modello viene addestrato, ci sarà una fase di ripesaggio delle istanze. L'algoritmo di boosting tenderà a dare un peso maggiore alle istanze misclassificate, nella speranza che il successivo modello sia più esperto su queste ultime. Sostanzialmente, ad ogni iterata calcola il tasso di errore ponderato dell'albero decisionale, ovvero il numero di predizioni sbagliate sul totale delle predizioni, che dipende dai pesi associati agli esempi nel dataset; successivamente in base all'errore calcola il learning rate dell'albero decisionale. Maggiore è il tasso di errore di un albero, minore sarà il potere decisionale che l'albero avrà durante la predizione successiva. Minore è il tasso di errore di un albero, maggiore sarà il potere decisionale assegnato all'albero durante la predizione successiva. Questo modello ha avuto risultati nella media, infatti training score 66% e accuratezza 61%.



Per poter migliorare, sono stati sintonizzati i seguenti iperparametri:

- **base_estimator:** lo stimatore di base da cui viene costruito l'ensemble potenziato ;
- **n_estimators:** Il numero massimo di stimatori al quale viene terminato il potenziamento. In caso di adattamento perfetto, la procedura di apprendimento viene interrotta anticipatamente;
- **learning_rate:** Peso applicato a ciascun classificatore ad ogni iterazione di potenziamento. Un tasso di apprendimento più elevato aumenta il contributo di ciascun classificatore.

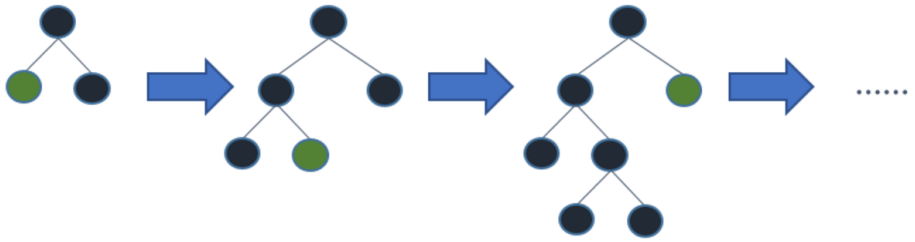
Gli iperparametri sintonizzati di tali valori, hanno portato un training score del 92% e un'accuratezza del 67%.

**XGB**

Il terzo algoritmo è XGB, un potente algoritmo di apprendimento automatico soprattutto per quanto riguarda velocità e precisione. XGBoost è una libreria di potenziamento del gradiente distribuita ottimizzata progettata per essere altamente efficiente, flessibile e portatile . Implementa algoritmi di machine learning nell'ambito del framework Gradient Boosting .
In questo algoritmo, gli alberi decisionali vengono creati in forma sequenziale. I pesi giocano un ruolo importante in XGBoost. I pesi vengono assegnati a tutte le variabili indipendenti che vengono poi inserite nell'albero decisionale che prevede i risultati. Il peso delle variabili predette erroneamente dall'albero viene aumentato e queste variabili vengono quindi inviate al secondo albero decisionale. Questi classificatori/predittori individuali poi si uniscono per dare un modello forte e più preciso.
Come metrica di valutazione per i classificatori interni è stato utilizzato l'errore quadratico medio.
Come score del modello si ha 73%, mentre come accuratezza il 66%. Migliorando i seguenti iperparametri **n_estimators, max-depth, learning rate** si ottengono training score pari all'80% e accuratezza pari al 66%.

**Light GBM**

Il quarto algoritmo è il Light GBM, differisce dagli altri algoritmi basati su alberi. Quest'ultimo fa crescere l'albero verticalmente mentre un altro algoritmo fa crescere gli alberi orizzontalmente, il che significa che Light GBM cresce in base alle foglie mentre l'altro algoritmo cresce a livello. Sceglierà la foglia con la massima perdita delta per crescere. Quando si coltiva la stessa foglia, l'algoritmo Leaf-wise può ridurre più perdite rispetto a un algoritmo level-wise.



Leaf-wise tree growth

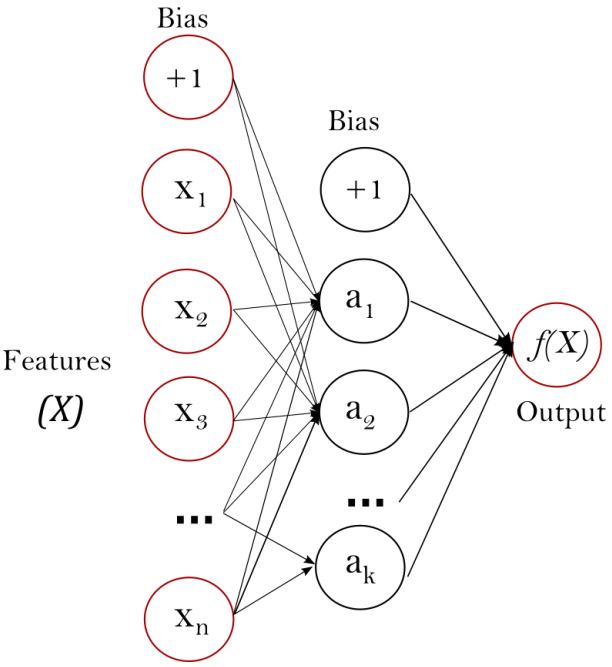
E' l'algoritmo che ha ottenuto i valori migliori, con training score pari al 95% e accuratezza pari al 67%. Andando ulteriormente a sintonizzare gli iperparametri

- **n_estimator:** Numero di alberi potenziati da adattare ;
- **subsample:** Rapporto di sotto campione dell'istanza di training ;
- **max_depth:** Profondità massima dell'albero per i learner di base;
- **learning_rate:** Aumenta il tasso di apprendimento;
- **min_child_samples:** Numero minimo di dati necessari in un figlio (foglia).

Questa procedura ha portato a un miglioramento decimale dell'accuratezza, pari al 68%, mentre ha comportato una diminuzione del training score, diminuito al 86%.

**MLP Classifier**

L'ultimo algoritmo osservato è il Multi-Layer perceptron, Il perceptron multistrato (MLP) è un modello di rete neurale artificiale feedforward che mappa i set di dati di input su un insieme di output appropriati. Un MLP è costituito da più livelli e ogni livello è completamente connesso a quello successivo. I nodi degli strati sono neuroni con funzioni di attivazione non lineari, ad eccezione dei nodi dello strato di input. Tra il livello di input e quello di output possono esserci uno o più livelli nascosti non lineari.




Questo algoritmo ha prodotto un training score del 61% e un'accuratezza del 61%.

Sintonizzando i seguenti iperparametri:

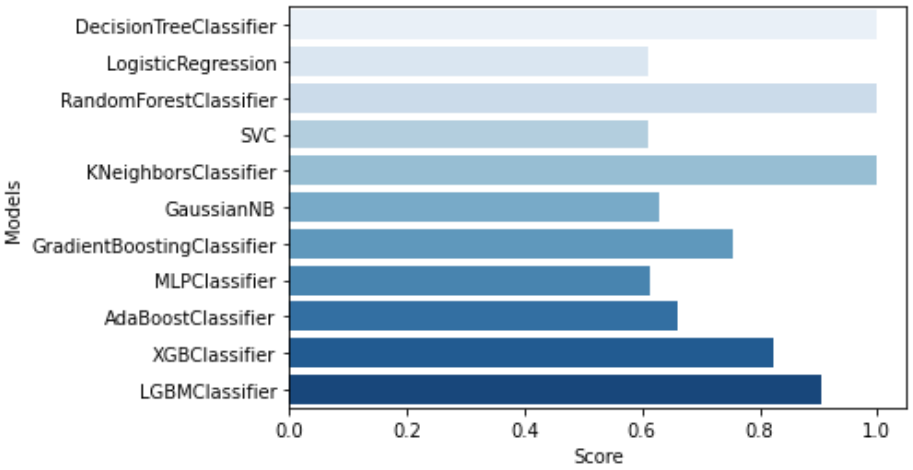
- **hidden_layer_size:** L'iesimo elemento rappresenta il numero di neuroni nell'iesimo strato nascosto;
- **activation:** Funzione di attivazione per il livello nascosto;
- **solver:** Il risolutore per l'ottimizzazione del peso;
- alpha:termine di regolarizzazione;
- **learning_rate:** Programma del tasso di apprendimento per gli aggiornamenti del peso.

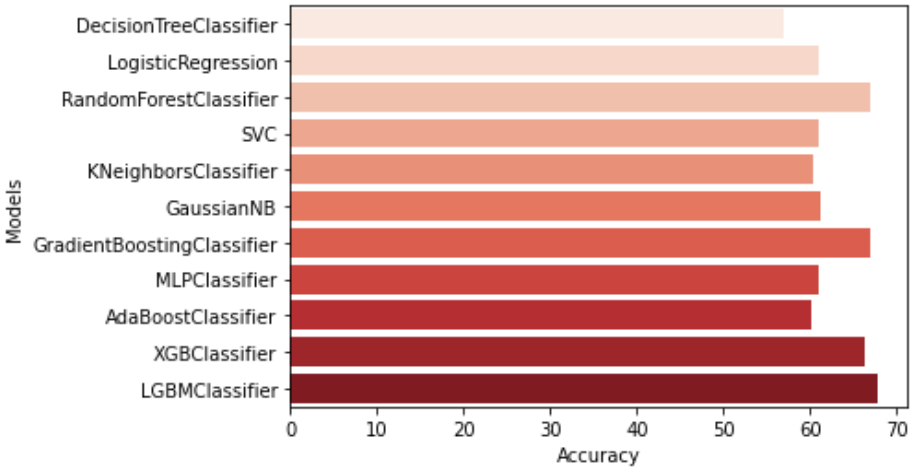
Questi iperparametri hanno portato un training score del 61% e un'accuratezza del 67%.

Rigante Chiara (matricola 708209)
Scalzo Andrea (matricola 705830)

# Risultati finali- Classificazione

In conclusione, si può affermare che l'algoritmo migliore con iperparametri sintonizzati è il Random Forest con training score pari al 100% e accuratezza al 67%.

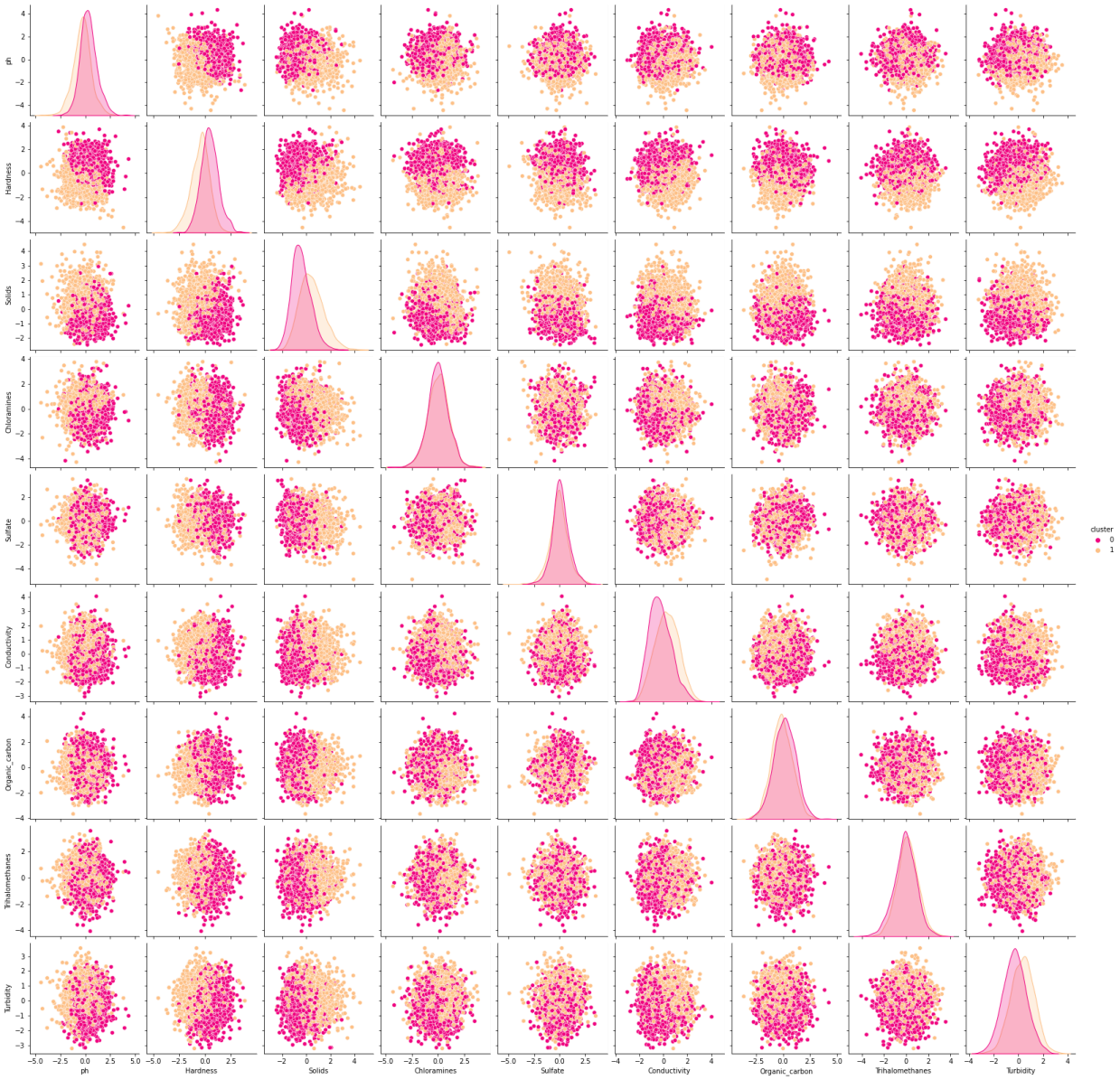


Rigante Chiara (matricola 708209)
Scalzo Andrea (matricola 705830)

# Apprendimento Non-Supervisionato

Per proseguire la nostra analisi sul dataset, avendo rimosso la feature “potability”, è stato scelto un algoritmo di Clustering, consiste nella ricerca di strutture e *pattern* all'interno di un dataset. Si fa **clustering** ogni qualvolta si vogliono trovare gruppi (o appunto, *cluster*) nei dati. E’ una tecnica esplorativa che ci consente di estrarre informazioni dal *dataset* senza avere informazioni a priori su di esso, in più tale algoritmo appartiene alla famiglia dell’hard clustering, ovvero ogni esempio assegnato ha una classe di appartenenza.
L’algoritmo utilizzato è il K-Means, è stato definito un numero random di cluster e successivamente è stato avviato il fitting del modello. Durante la fase di *fitting* il modello si preoccupa di muovere i **k centroidi, punti istanziati randomicamente all’inizio, successivamente avvicinati** verso il **baricentro** di ogni *cluster*.
Uno dei principali svantaggi di questo modello risiede proprio nel fatto che bisogna **definire a priori** il numero di *cluster* (k) presenti. E’ stato scelto un k=2, in maniera casuale.
Dopo aver eseguito il fitting del modello, abbiamo ottenuto le coordinate dei nostri centroidi, precisamente 9 valori, in quanto lo spazio è di dimensione R^9, dove 9 sono le colonne del dataset.
Abbiamo ottenuto una visualizzazione dei cluster mettendo in relazione le feature:



Per far sì che la nostra analisi fosse completa, abbiamo utilizzato l'indice di silhouette per ottenere la scelta migliore di numero di cluster, ovvero l'iperparametro del k-means. In particolare questa metrica è in grado di quantificare quanto un *cluster* è "*puro*" o meno.
Tale indice si calcola:

$$s = \frac{b - a}{max(a, b)} [-1, 1]$$

Dove:

- a= distanza media **intracluster**: distanza media tra i campioni e i punti dello **stesso** cluster (minore è meglio)
- b= distanza media **extracluster**: distanza media tra i campioni e i punti degli **altri** cluster (maggiore è meglio)

Maggiore l'indice di **silhouette**, migliore la **clusterizzazione**.

Per calcolare l'**indice di silhouette** su un *range* di diversi K al fine di trovare il *miglior* numero di *cluster*, si è scelto un **K da 2 a 25**. Per ognuno di questi si è *fittato* un *K-means* e salvato il corrispondente **indice di silhouette**.

```
Tested kMeans with k = 2        SS: 0.0825
Tested kMeans with k = 3        SS: 0.0785
Tested kMeans with k = 4        SS: 0.0748
Tested kMeans with k = 5        SS: 0.0754
Tested kMeans with k = 6        SS: 0.0754
Tested kMeans with k = 7        SS: 0.0762
Tested kMeans with k = 8        SS: 0.0780
Tested kMeans with k = 9        SS: 0.0760
Tested kMeans with k = 10       SS: 0.0766
Tested kMeans with k = 11       SS: 0.0767
Tested kMeans with k = 12       SS: 0.0789
Tested kMeans with k = 13       SS: 0.0782
Tested kMeans with k = 14       SS: 0.0763
Tested kMeans with k = 15       SS: 0.0790
Tested kMeans with k = 16       SS: 0.0801
Tested kMeans with k = 17       SS: 0.0785
Tested kMeans with k = 18       SS: 0.0793
Tested kMeans with k = 19       SS: 0.0767
Tested kMeans with k = 20       SS: 0.0771
Tested kMeans with k = 21       SS: 0.0779
Tested kMeans with k = 22       SS: 0.0753
Tested kMeans with k = 23       SS: 0.0771
Tested kMeans with k = 24       SS: 0.0751
```

In più, visualizzando graficamente, siamo giunti a conclusione che K=2, risulta essere la scelta migliore per il nostro dataset. Aumentando l'iperparametro K, otteniamo *cluster* sempre più *sporchi*.