Ciencia de Datos 2023 - Entrega de Trabajo Final Integrador

Sambrana Ivan

Conjunto de datos para la clasificación de la calidad del agua

Acerca del Conjunto de Datos

Contexto

Este es un conjunto de datos creado a partir de datos imaginarios sobre la calidad del agua en un entorno urbano. Recomiendo utilizar este conjunto de datos para fines educativos, para practicar y adquirir los conocimientos necesarios.

Contenido

Lo que hay dentro es más que simplemente filas y columnas. Puedes ver los ingredientes del agua listados como nombres de las columnas.

Descripción

Todos los atributos son variables numéricas y se enumeran a continuación:

aluminio - peligroso si es mayor a 2.8

amoníaco - peligroso si es mayor a 32.5

arsénico - peligroso si es mayor a 0.01

bario - peligroso si es mayor a 2

cadmio - peligroso si es mayor a 0.005

cloroamina - peligroso si es mayor a 4

cromo - peligroso si es mayor a 0.1

cobre - peligroso si es mayor a 1.3

fluoruro - peligroso si es mayor a 1.5

bacterias - peligroso si es mayor a 0

virus - peligroso si es mayor a 0

plomo - peligroso si es mayor a 0.015

nitratos - peligroso si es mayor a 10

nitritos - peligroso si es mayor a 1

mercurio - peligroso si es mayor a 0.002

perclorato - peligroso si es mayor a 56

radio - peligroso si es mayor a 5

selenio - peligroso si es mayor a 0.5

plata - peligroso si es mayor a 0.1

uranio - peligroso si es mayor a 0.3

es_seguro - atributo de clase {0 - no seguro, 1 - seguro}

https://www.kaggle.com/datasets/mssmartypants/water-quality

Recolección de Datos

[1]
```
from google.colab import drive

drive.mount("/content/drive")
```

```
Mounted at /content/drive
```

[2]
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

[3]
```
df = pd.read_csv('/content/drive/MyDrive/MTI-ciencia de datos/waterQuality1.csv')
df.head()
```

	aluminium	ammonia	arsenic	barium	cadmium	chloramine	chromium	copper	flouride	bacteria	...	lead	nitrates	nitrites	mercury	perchlorate	radium	selenium	silver	uranium	is_safe
0	1.65	9.08	0.04	2.85	0.007	0.35	0.83	0.17	0.05	0.20	...	0.054	16.08	1.13	0.007	37.75	6.78	0.08	0.34	0.02	1
1	2.32	21.16	0.01	3.31	0.002	5.28	0.68	0.66	0.90	0.65	...	0.100	2.01	1.93	0.003	32.26	3.21	0.08	0.27	0.05	1
2	1.01	14.02	0.04	0.58	0.008	4.24	0.53	0.02	0.99	0.05	...	0.078	14.16	1.11	0.006	50.28	7.07	0.07	0.44	0.01	0
3	1.36	11.33	0.04	2.96	0.001	7.23	0.03	1.66	1.08	0.71	...	0.016	1.41	1.29	0.004	9.12	1.72	0.02	0.45	0.05	1
4	0.92	24.33	0.03	0.20	0.006	2.67	0.69	0.57	0.61	0.13	...	0.117	6.74	1.11	0.003	16.90	2.41	0.02	0.06	0.02	1

5 rows × 21 columns

Analisis exploratorio

```
[4] df.shape
```

```
(7999, 21)
```

Verificamos que no tenga nulos

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

```
aluminium      0
ammonia        0
arsenic        0
barium         0
cadmium        0
chloramine     0
chromium       0
copper         0
flouride       0
bacteria       0
viruses        0
lead           0
nitrates       0
nitrites       0
mercury        0
perchlorate    0
radium         0
selenium       0
silver         0
uranium        0
is_safe        0
dtype: int64
```

```
[6] df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 7999 entries, 0 to 7998
Data columns (total 21 columns):
 #   Column       Non-Null Count  Dtype
---  ------       --------------  -----
 0   aluminium    7999 non-null   float64
 1   ammonia      7999 non-null   object
 2   arsenic      7999 non-null   float64
 3   barium       7999 non-null   float64
 4   cadmium      7999 non-null   float64
 5   chloramine   7999 non-null   float64
 6   chromium     7999 non-null   float64
 7   copper       7999 non-null   float64
 8   flouride     7999 non-null   float64
 9   bacteria     7999 non-null   float64
 10  viruses      7999 non-null   float64
 11  lead         7999 non-null   float64
 12  nitrates     7999 non-null   float64
 13  nitrites     7999 non-null   float64
 14  mercury      7999 non-null   float64
 15  perchlorate  7999 non-null   float64
 16  radium       7999 non-null   float64
 17  selenium     7999 non-null   float64
 18  silver       7999 non-null   float64
 19  uranium      7999 non-null   float64
 20  is_safe      7999 non-null   object
dtypes: float64(19), object(2)
memory usage: 1.3+ MB
```

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

admium	chloramine	chromium	copper	flouride	bacteria	viruses	lead	nitrates	nitrites	mercury	perchlorate	radium	selenium	silver	uranium
000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000	7999.000000
042806	2.176831	0.247226	0.805857	0.771565	0.319665	0.328583	0.099450	9.818822	1.329961	0.005194	16.460299	2.920548	0.049685	0.147781	0.044673
036049	2.567027	0.270640	0.653539	0.435373	0.329485	0.378096	0.058172	5.541331	0.573219	0.002967	17.687474	2.323009	0.028770	0.143551	0.026904
000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
008000	0.100000	0.050000	0.090000	0.405000	0.000000	0.002000	0.048000	5.000000	1.000000	0.003000	2.170000	0.820000	0.020000	0.040000	0.020000
040000	0.530000	0.090000	0.750000	0.770000	0.220000	0.008000	0.102000	9.930000	1.420000	0.005000	7.740000	2.410000	0.050000	0.080000	0.050000
070000	4.240000	0.440000	1.390000	1.160000	0.610000	0.700000	0.151000	14.610000	1.760000	0.008000	29.480000	4.670000	0.070000	0.240000	0.070000
130000	8.680000	0.900000	2.000000	1.500000	1.000000	1.000000	0.200000	19.830000	2.930000	0.010000	60.010000	7.990000	0.100000	0.500000	0.090000

```
[8] df.fillna(df.mean(), inplace=True)
    df.isnull().sum()
```

```
<ipython-input-8-c6455a908190>:1: FutureWarning: The default value of numeric_only in DataFrame.mean is deprecated. In a future version, it will default to False. In addition, specifying 'numeri
  df.fillna(df.mean(), inplace=True)
aluminium      0
ammonia        0
arsenic        0
barium         0
cadmium        0
chloramine     0
chromium       0
copper         0
flouride       0
bacteria       0
viruses        0
lead           0
nitrates       0
nitrites       0
mercury        0
perchlorate    0
radium         0
selenium       0
silver         0
uranium        0
is_safe        0
dtype: int64
```

verificamos que la columna "is_safe" atributo de clase, solo tenga valores 0,1.

```
[9] df.is_safe.value_counts()
```

```
0        7084
1         912
#NUM!       3
Name: is_safe, dtype: int64
```

El dataset contiene registros corruptos, la columan dataset tiene registos con el valor string '#NUM!', a continuacion limpiaremos esos registros

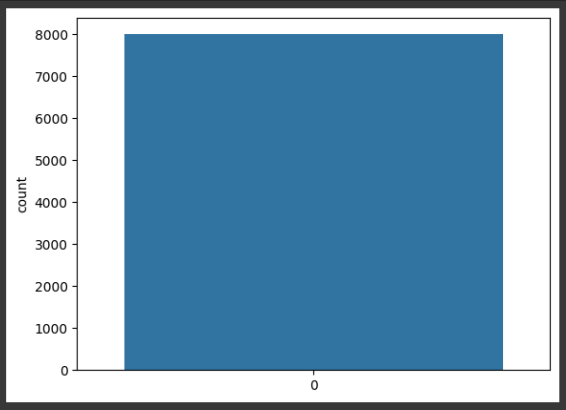
```
[22] df = df[df.is_safe != '#NUM!']
```

Verificamos nuevamente

```
[23] df.is_safe.value_counts()
```

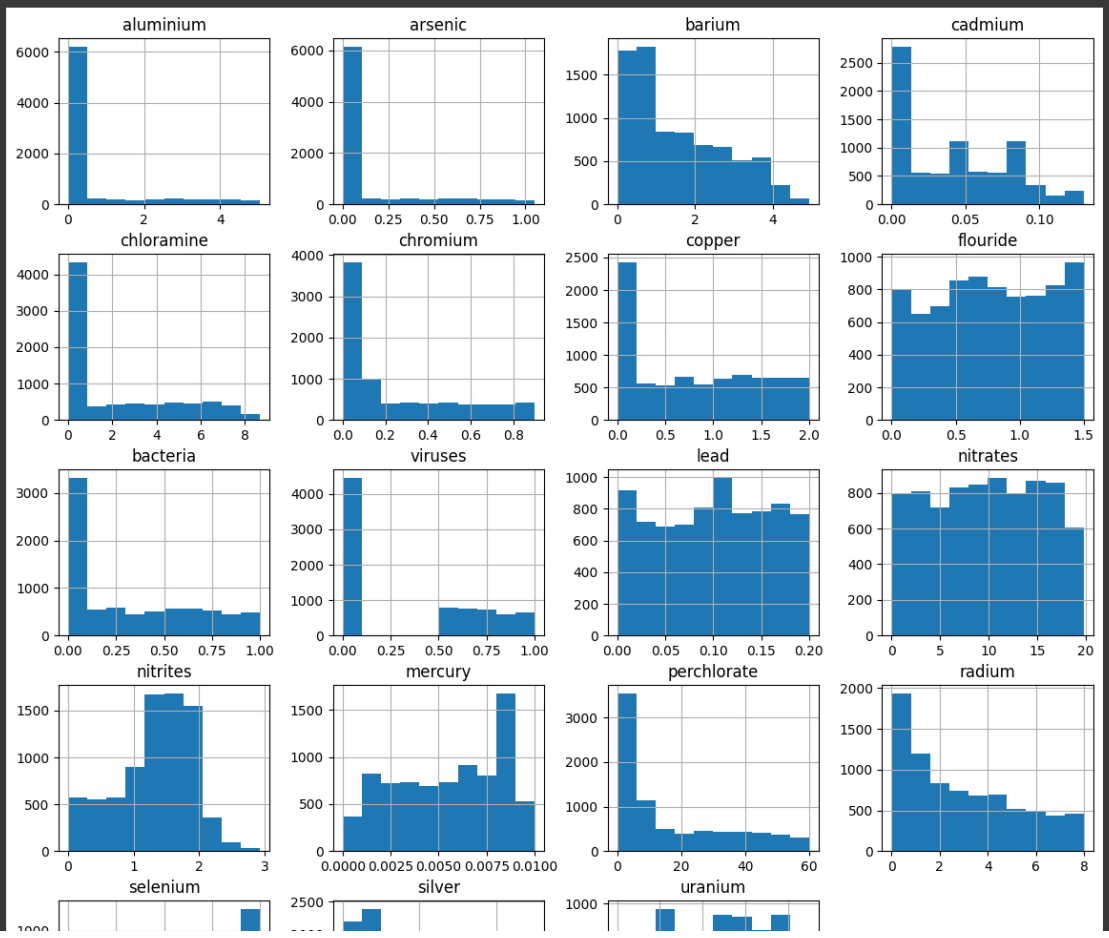
```
0    7084
1     912
Name: is_safe, dtype: int64
```

```
[44] sns.countplot(df['is_safe'])
     plt.show()
```



Visualiza todas las características del conjunto de datos como puedes ver a continuación.

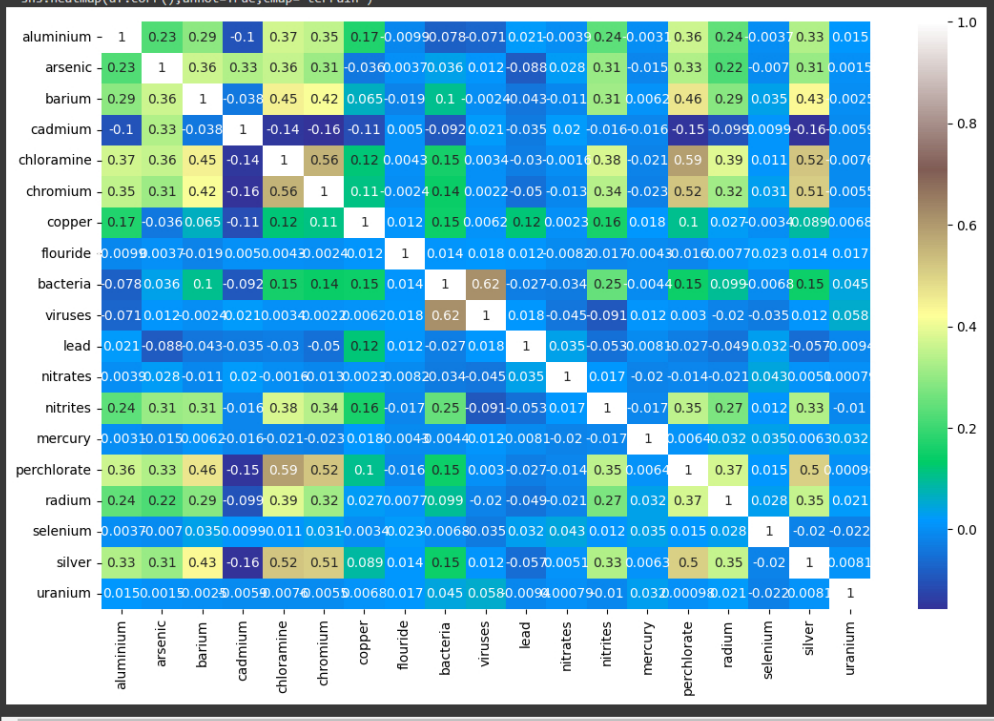
```
[26] df.hist(figsize=(14,14))
     plt.show()
```



Visualiza la correlación de todas las características utilizando una función de mapa de calor de seaborn.

```
[27] plt.figure(figsize=(13,8))
     sns.heatmap(df.corr(),annot=True,cmap='terrain')
     plt.show()
```

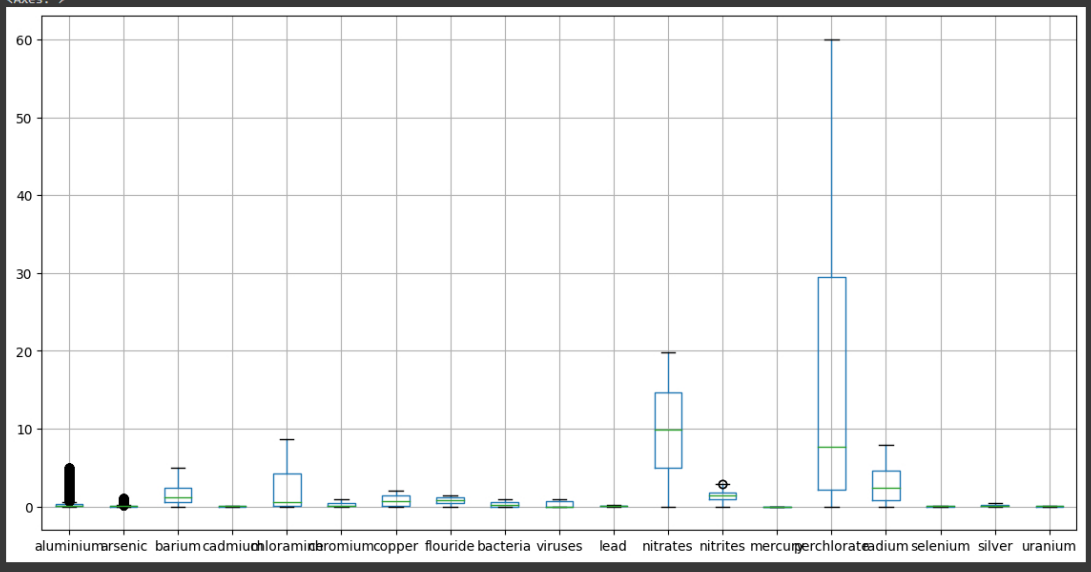
```
<ipython-input-27-6450b8fbb5cc>:2: FutureWarning: The default value of numeric_only in DataFrame.corr is deprecated. In a future version, it will default to False. Select only valid columns or s
  sns.heatmap(df.corr(),annot=True,cmap='terrain')
```



Ahora observa los valores atípicos utilizando una función de diagrama de caja (boxplot).

```
[28] df.boxplot(figsize=(14,7))
```

```
<Axes: >
```



Preparacion

Divide los datos en características independientes y dependientes. Todas son características independientes excepto "is_safe", porque "is_safe" es nuestra característica dependiente.

```
[29] X = df.drop('is_safe',axis=1)
     Y= df['is_safe']
```

```
[30] df.head()
```

	aluminium	ammonia	arsenic	barium	cadmium	chloramine	chromium	copper	flouride	bacteria	...	lead	nitrates	nitrites	mercury	perchlorate	radium	selenium	silver	uranium	is_safe
0	1.65	9.08	0.04	2.85	0.007	0.35	0.83	0.17	0.05	0.20	...	0.054	16.08	1.13	0.007	37.75	6.78	0.08	0.34	0.02	1
1	2.32	21.16	0.01	3.31	0.002	5.28	0.68	0.66	0.90	0.65	...	0.100	2.01	1.93	0.003	32.26	3.21	0.08	0.27	0.05	1
2	1.01	14.02	0.04	0.58	0.008	4.24	0.53	0.02	0.99	0.05	...	0.078	14.16	1.11	0.006	50.28	7.07	0.07	0.44	0.01	0

3	1.36	11.33	0.04	2.96	0.001	7.23	0.03	1.66	1.08	0.71	...	0.016	1.41	1.29	0.004	9.12	1.72	0.02	0.45	0.05	1
4	0.92	24.33	0.03	0.20	0.006	2.67	0.69	0.57	0.61	0.13	...	0.117	6.74	1.11	0.003	16.90	2.41	0.02	0.06	0.02	1

5 rows × 21 columns

Divide el conjunto de datos en entrenamiento y prueba utilizando la función train_test_split, la cual retorna cuatro conjuntos de datos

```
[31] from sklearn.model_selection import train_test_split
     X_train, X_test, Y_train, Y_test = train_test_split(X,Y, test_size= 0.2, random_state=101, shuffle=True)

     X_train.shape,X_test.shape,Y_train.shape,Y_test.shape
```

```
((6396, 20), (1600, 20), (6396,), (1600,))
```

Normalizacion de los conjuntos

```
[32] from sklearn.preprocessing import StandardScaler
     standard_x = StandardScaler()
     X_train = standard_x.fit_transform(X_train)
     X_test = standard_x.fit_transform(X_test)
```

Prueba de Modelo Regresion Logistica

```
[47] from sklearn.linear_model import LogisticRegression

     clf = LogisticRegression(random_state=0)

     clf.fit(X_train, Y_train)
     pred = clf.predict(X_test)
```

```
[48] X_train, pred
```

```
(array([[-0.48609022, -0.47763684, -0.28128987, ..., -0.69177987,
         -0.67801047,  0.93666606],
        [-0.49409669,  1.58745883, -0.47955771, ..., -1.38769431,
         -0.60845309,  1.68250063],
        [-0.45406433,  0.97524518, -0.43990414, ...,  0.70004902,
         -0.88668262, -0.55500307],
        ...,
        [ 1.96389009,  1.53118919, -0.08302202, ...,  1.39596346,
         -0.12155142,  1.68250063],
        [ 1.19526882,  0.2910064 ,  1.58242786, ...,  0.00413458,
         -0.81712524, -0.18208578],
        [-0.4620708 , -1.57489476, -0.55886485, ..., -0.69177987,
         -0.88668262, -0.55500307]]),
 array(['0', '0', '0', ..., '0', '0', '0'], dtype=object))
```

```
[49] print(f"Accuracy Score = {accuracy_score(Y_test,prediction)*100}")
     print(f"Confusion Matrix =\n {confusion_matrix(Y_test,prediction)}")
     print(f"Classification Report =\n {classification_report(Y_test,prediction)}")
```

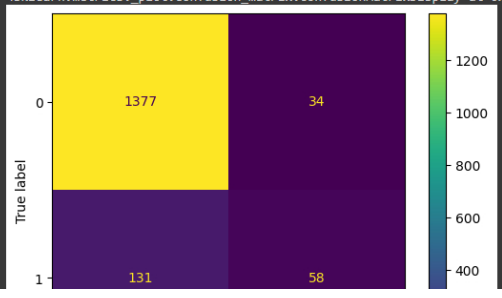
```
Accuracy Score = 94.125
Confusion Matrix =
 [[1377   34]
 [  60  129]]
Classification Report =
              precision    recall  f1-score   support

           0       0.96      0.98      0.97      1411
           1       0.79      0.68      0.73       189

    accuracy                           0.94      1600
   macro avg       0.87      0.83      0.85      1600
weighted avg       0.94      0.94      0.94      1600
```

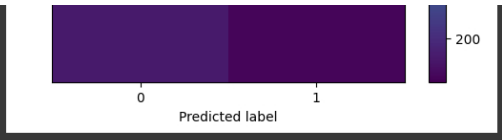
94.125%

Matriz de Confusion

```
[50] #Matriz de confusión
     from sklearn.metrics import confusion_matrix
     #conf_mat = confusion_matrix(Y_test,pred)
     #conf_mat

     from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay
     cm = confusion_matrix(Y_test, pred, labels=clf.classes_)
     disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
     disp.plot()
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f52705adc00>
```





Entrenar un Clasificador de Árbol de Decisión y verificar la precisión

```
[36] from sklearn.tree import DecisionTreeClassifier
     from sklearn.metrics import accuracy_score,confusion_matrix,classification_report
     dt=DecisionTreeClassifier(criterion= 'gini', min_samples_split= 10, splitter= 'random')
     dt.fit(X_train,Y_train)
```

```
DecisionTreeClassifier(min_samples_split=10, splitter='random')
```

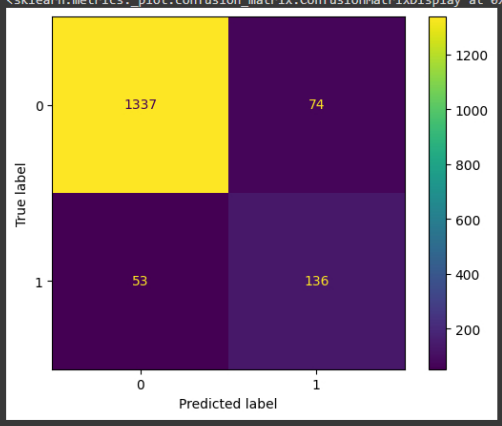
```
[37] prediction=dt.predict(X_test)
     print(f"Accuracy Score = {accuracy_score(Y_test,prediction)*100}")
     print(f"Confusion Matrix =\n {confusion_matrix(Y_test,prediction)}")
     print(f"Classification Report =\n {classification_report(Y_test,prediction)}")
```

```
Accuracy Score = 92.0625
Confusion Matrix =
 [[1337   74]
 [  53  136]]
Classification Report =
              precision    recall  f1-score   support

           0       0.96      0.95      0.95      1411
           1       0.65      0.72      0.68       189

    accuracy                           0.92      1600
   macro avg       0.80      0.83      0.82      1600
weighted avg       0.92      0.92      0.92      1600
```

92.0625%

```
[38] from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay
     cm = confusion_matrix(Y_test, prediction, labels=clf.classes_)
     disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
     disp.plot()
```

```
<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f526d98e260>
```




```
[40] res = dt.predict([[0.07 ,0.52,0.01,0.31,0.04,0.01,0.03,0.77,0.46,0,0,0.059,13.44,1.93,0.005,0.03,0.91,0.01,0.03,0.03]])[0]
     res
```

```
'0'
```

Aplicar Ajuste de Hiper Parametros

```
[41] from sklearn.model_selection import RepeatedStratifiedKFold
     from sklearn.model_selection import GridSearchCV

     # define models and parameters
     model = DecisionTreeClassifier()
     criterion = ["gini", "entropy"]
     splitter = ["best", "random"]
     min_samples_split = [2,4,6,8,10,12,14]

     # define grid search
     grid = dict(splitter=splitter, criterion=criterion, min_samples_split=min_samples_split)
     cv = RepeatedStratifiedKFold(n_splits=10, n_repeats=3, random_state=1)
     grid_search_dt = GridSearchCV(estimator=model, param_grid=grid, n_jobs=-1, cv=cv,
                                   scoring='accuracy',error_score=0)
     grid_search_dt.fit(X_train, Y_train)
```

```
GridSearchCV
estimator: DecisionTreeClassifier
DecisionTreeClassifier
```

```
[42] print(f"Best: {grid_search_dt.best_score_:.3f} using {grid_search_dt.best_params_}")
     means = grid_search_dt.cv_results_['mean_test_score']
```

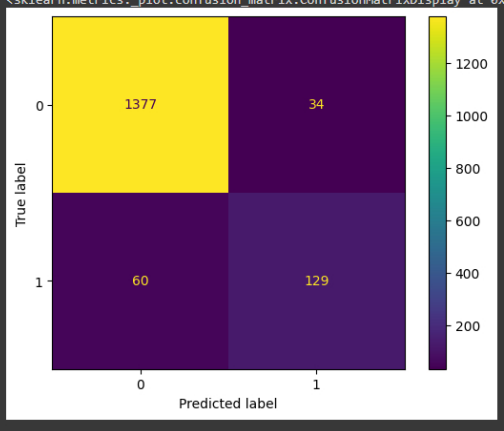
```
    stds = grid_search_dt.cv_results_['std_test_score']
    params = grid_search_dt.cv_results_['params']

    for mean, stdev, param in zip(means, stds, params):
        print(f"{mean:.3f} ({stdev:.3f}) with: {param}")

    print("Training Score:",grid_search_dt.score(X_train, Y_train)*100)
    print("Testing Score:", grid_search_dt.score(X_test, Y_test)*100)
```

```
Best: 0.958 using {'criterion': 'entropy', 'min_samples_split': 14, 'splitter': 'best'}
0.951 (0.009) with: {'criterion': 'gini', 'min_samples_split': 2, 'splitter': 'best'}
0.927 (0.009) with: {'criterion': 'gini', 'min_samples_split': 2, 'splitter': 'random'}
0.952 (0.008) with: {'criterion': 'gini', 'min_samples_split': 4, 'splitter': 'best'}
0.931 (0.011) with: {'criterion': 'gini', 'min_samples_split': 4, 'splitter': 'random'}
0.953 (0.009) with: {'criterion': 'gini', 'min_samples_split': 6, 'splitter': 'best'}
0.933 (0.008) with: {'criterion': 'gini', 'min_samples_split': 6, 'splitter': 'random'}
0.952 (0.008) with: {'criterion': 'gini', 'min_samples_split': 8, 'splitter': 'best'}
0.934 (0.010) with: {'criterion': 'gini', 'min_samples_split': 8, 'splitter': 'random'}
0.953 (0.008) with: {'criterion': 'gini', 'min_samples_split': 10, 'splitter': 'best'}
0.937 (0.010) with: {'criterion': 'gini', 'min_samples_split': 10, 'splitter': 'random'}
0.953 (0.009) with: {'criterion': 'gini', 'min_samples_split': 12, 'splitter': 'best'}
0.935 (0.009) with: {'criterion': 'gini', 'min_samples_split': 12, 'splitter': 'random'}
0.954 (0.009) with: {'criterion': 'gini', 'min_samples_split': 14, 'splitter': 'best'}
0.937 (0.011) with: {'criterion': 'gini', 'min_samples_split': 14, 'splitter': 'random'}
0.954 (0.008) with: {'criterion': 'entropy', 'min_samples_split': 2, 'splitter': 'best'}
0.928 (0.010) with: {'criterion': 'entropy', 'min_samples_split': 2, 'splitter': 'random'}
0.955 (0.008) with: {'criterion': 'entropy', 'min_samples_split': 4, 'splitter': 'best'}
0.933 (0.009) with: {'criterion': 'entropy', 'min_samples_split': 4, 'splitter': 'random'}
0.956 (0.008) with: {'criterion': 'entropy', 'min_samples_split': 6, 'splitter': 'best'}
0.935 (0.011) with: {'criterion': 'entropy', 'min_samples_split': 6, 'splitter': 'random'}
0.956 (0.007) with: {'criterion': 'entropy', 'min_samples_split': 8, 'splitter': 'best'}
0.934 (0.008) with: {'criterion': 'entropy', 'min_samples_split': 8, 'splitter': 'random'}
0.956 (0.008) with: {'criterion': 'entropy', 'min_samples_split': 10, 'splitter': 'best'}
0.936 (0.009) with: {'criterion': 'entropy', 'min_samples_split': 10, 'splitter': 'random'}
0.958 (0.007) with: {'criterion': 'entropy', 'min_samples_split': 12, 'splitter': 'best'}
0.938 (0.009) with: {'criterion': 'entropy', 'min_samples_split': 12, 'splitter': 'random'}
0.958 (0.008) with: {'criterion': 'entropy', 'min_samples_split': 14, 'splitter': 'best'}
0.940 (0.008) with: {'criterion': 'entropy', 'min_samples_split': 14, 'splitter': 'random'}
Training Score: 98.8899120700438
Testing Score: 95.125
```

Volvemos a aplicar con la recomendacion Best: 0.958 using {'criterion': 'entropy', 'min_samples_split': 14, 'splitter': 'best'}

```
[43] # define grid search
    from sklearn.tree import DecisionTreeClassifier
    from sklearn.metrics import accuracy_score,confusion_matrix,classification_report
    from sklearn.metrics import classification_report, confusion_matrix, ConfusionMatrixDisplay

    dt=DecisionTreeClassifier(criterion= 'entropy', min_samples_split= 14, splitter= 'best')
    dt.fit(X_train,Y_train)

    prediction=dt.predict(X_test)
    print(f"Accuracy Score = {accuracy_score(Y_test,prediction)*100}")
    print(f"Confusion Matrix =\n {confusion_matrix(Y_test,prediction)}")
    print(f"Classification Report =\n {classification_report(Y_test,prediction)}")


    cm = confusion_matrix(Y_test, prediction, labels=clf.classes_)
    disp = ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=clf.classes_)
    disp.plot()
```

```
Accuracy Score = 94.125
Confusion Matrix =
 [[1377   34]
 [  60  129]]
Classification Report =
              precision    recall  f1-score   support

           0       0.96      0.98      0.97      1411
           1       0.79      0.68      0.73       189

    accuracy                           0.94      1600
   macro avg       0.87      0.83      0.85      1600
weighted avg       0.94      0.94      0.94      1600

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f5270ca2590>
```



se mejora a un 94.125%

Conclusion

Se Aplicaron dos modelos de aprendizaje supervisado - la Regresión Logística y el Clasificador de Árbol de Decisión - al conjunto de datos de calidad del agua.

Ambos modelos demostraron ser eficaces en la predicción de la seguridad del agua, un atributo de clase binario, a partir de los datos de varios componentes del agua. La Regresión Logística, que se utiliza ampliamente para predecir variables dependientes binarias, alcanzó una

precisión del **94.125%.** Este método no solo proporciona una predicción binaria, sino que también ofrece una estimación de la probabilidad de cada resultado, lo que añade un nivel adicional de información y permite una evaluación más matizada de la seguridad del agua.

El Clasificador de Árbol de Decisión, por otro lado, es un modelo no paramétrico que puede manejar tanto características numéricas como categóricas, lo que lo hace ideal para este conjunto de datos que contiene una mezcla de tales características. Este modelo alcanzó una precisión del **92.0625%**.

Además, el refinamiento de los hiperparámetros en nuestro modelo de Árbol de Decisión resultó en una mejora de la precisión, alcanzando un puntaje igual al de la Regresión Logística,** 94.125%**. Este proceso, que involucra el ajuste de la profundidad del árbol, el número mínimo de muestras requerido para dividir un nodo interno, y otros parámetros, puede ayudar a mejorar la eficacia del modelo y a evitar el sobreajuste o el infraajuste.

En conclusión, ambos modelos resultaron ser altamente efectivos para predecir la seguridad del agua en este conjunto de datos. Por último, se ha demostrado la utilidad del refinamiento de hiperparámetros para mejorar la precisión de las predicciones