INFORME PROYECTO

CONTEXTUALIZACION DEL PROBLEMA

El agua potable y segura destinada al consumo humano debe cumplir una calidad sanitaria apta, tanto después de un proceso de tratamiento como también debe mostrar una estabilidad biológica en la red de distribución. Por lo anterior, es necesario una constante vigilancia de la calidad del agua para el abastecimiento de la población desde el origen de la misma hasta que llega al sitio de consumo.

La base de datos contiene información recopilada en diferentes estaciones locales en un área específica. Estos datos muestran el porcentaje de concentración de algunas variables químicas como el porcentaje de aluminio, de amoniaco, bario, cadmio, cloramina, cromo, cobre, fluoruros, plomo, nitratos, nitritos, mercurio, perclorato, radio, plata, uranio y también mide variables biológicas como la presencia de bacterias y virus.

Además presenta una variable dependiente que indica si la calidad de agua es segura o no para el consumo humano y lo muestra con un 1 si es adecuada y con un 0 si no lo es.

Cada una de las variables anteriores contienen unos rangos en sus concentraciones que me indican hasta qué punto todavía es benéfico consumir agua en esas proporciones.

Variable	Umbral de concentración (%)
Aluminio	2.8
Amoníaco	32.5
Arsénico	0.01
Bario	2
Cadmio	0.005
Cloramina	4
Cromo	0.1
Cobre	1.3
Fluoruro	1.5
Bacteria	0
Virus	0
Plomo	0.015
Nitratos	10
Nitritos	1
Mercurio	0.002
Perclorato	56
Radio	5
Selenio	0.5
Plata	0.1
Uranio	0.3

OBJETIVO

Predecir si el estado del agua es seguro para el consumo humano a partir del porcentaje de concentración de las variables químicas (aluminio, amoniaco, arsénico, bario, cadmio, cloramina, cromo, cobre, fluoruro, plomo, nitratos, nitritos, mercurio, perclorato, radio, selenio, plata y uranio) y las variables biológicas (bacterias y virus) utilizando una red neuronal convolucional.

EXPLORACIÓN DE LOS DATOS

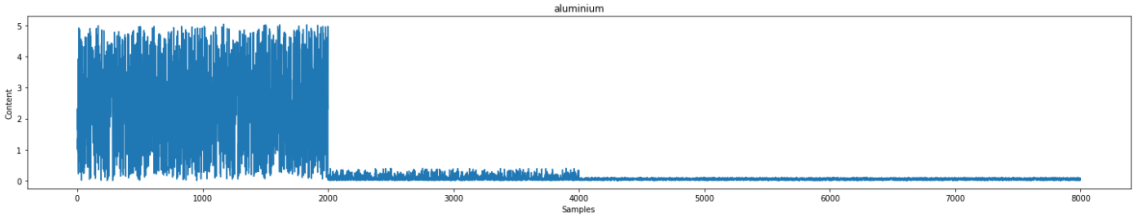
Se trabajaron 7999 datos contenidos en 21 variables (variables químicas, biológicas y variable respuesta).

	aluminium	ammonia	arsenic	barium	cadmium	chloramine	chromium	copper	flouride	bacteria	...	lead	nitrates	nitrites	mercury
0	1.65	9.08	0.04	2.85	0.007	0.35	0.83	0.17	0.05	0.2	...	0.054	16.08	1.13	0.007

1 rows × 21 columns

perchlorate	radium	selenium	silver	uranium	is_safe
0.629062	0.848561	0.8	0.68	0.222222	1.0

Inicialmente se graficaron cada una de las variables empezando desde la variable 0 que corresponde al aluminio y cada vez que se ejecuta se pasa a la siguiente posición, es decir, a la siguiente variable.



1

Cuando se examinan detenidamente los datos de cada una de las variables se observa que no parecen reales sino que alguna persona con experiencia los habría establecido o hizo algunos cálculos matemáticos para determinar esas concentraciones previamente.

Se ha evidenciado que algunos datos de la variable respuesta no corresponden al 1 o al 0 y presentan otro valor.

```
print(df["is_safe"].value_counts())
```

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

A continuación se eliminaron los valores diferentes a 0 y 1 a través de la función drop.

```
# remove values other than 1 and 0 in the response variable
df.drop(df.loc[df['is_safe']=='#NUM!'].index, inplace=True)
```

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

Luego se estandarizaron los datos entre 0 y 1 con la instrucción MinMaxScaler para luego poder usar los datos en las redes neuronales.

Se tuvo el inconveniente que con esta instrucción se eliminaron los nombres de las columnas.

A continuación se crea otro dataset con los datos escalados entre cero y uno y se procedió a asignarle el nombre a las columnas que se habían perdido.

	aluminium	ammonia	arsenic	barium	cadmium	chloramine	chromium	copper	flouride	bacteria	...	lead	nitrates	nitrites	mercury	perch
0	0.326733	0.30615	0.038095	0.576923	0.053846	0.040323	0.922222	0.085	0.033333	0.2	...	0.27	0.810893	0.385666	0.7	0.

1 rows × 21 columns

PREPROCESAMIENTO DE LOS DATOS

Se utilizaron las mismas librerías y por la naturaleza de los datos se puede observar que con un árbol de decisión sencillo o una red neuronal de capas densas podría funcionar el modelo, pero el objetivo es trabajar los datos en redes neuronales convolucionales.

Inicialmente se crea un dataset sin la variable respuesta y otro que contiene solo la variable respuesta

```
# create dataset without response column
df_x=df.iloc[:,0:20]
# create dataset with only response column (is_safe)
df_y = df.iloc[:,-1]
```

Posteriormente se separó el conjunto de datos para entrenamiento y para prueba en un 30%

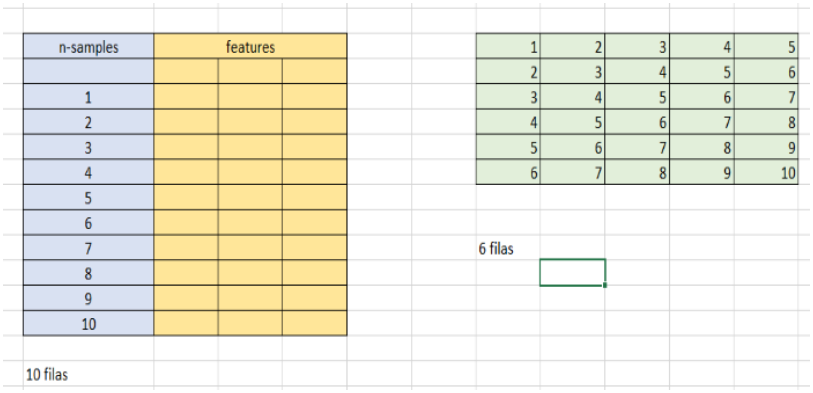
```
from sklearn.model_selection import train_test_split
df_x_train, df_x_test, y_train, y_test = train_test_split(df_x, df_y, test_size=.30)
```

Los resultados muestran una distribución de clases para datos de entrenamiento y datos de prueba.

```
distribution of train classes
0.0    4861
1.0     636
Name: is_safe, dtype: int64

distribution of test classes
0.0    2050
1.0     249
Name: is_safe, dtype: int64
```

Inicialmente los datos están en una sola dimensión por lo que deben convertirse en forma matricial para poder trabajar redes neuronales convolucionales.



Se toman 5 datos de la columna para transponerlos a la primera fila y se incrementa en una posición el valor inicial (1) y se transpone horizontalmente para formar la siguiente fila y así sucesivamente para formar la matriz, pero al llegar a la última posición no se puede continuar porque los valores no son suficientes para una siguiente fila lo que hace que la matriz finalice y no coincida el número de filas por lo que se hace necesario eliminar los últimos datos.

Todo este proceso se hace para todas las variables y se seleccionan matrices de 100 elementos.

Lo anterior se realiza tanto para los datos de entrenamiento como para los datos de prueba. El siguiente algoritmo realiza este proceso y luego se hace el llamado de la función tanto para los datos de entrenamiento como los datos de prueba.

```python
# Generate training sequence
def create_sequences(values, n_samples, n_times, n_features):
    output = np.zeros( (n_samples-n_times, n_times, n_features) )
    for j in range(1, n_features):
        features = []
        for i in range(n_samples - n_times):
            output[i,:,j]=values[i : (i + n_times),j]
    return output
```

```
df_training = np.array(df_x_train)
N_SAMPLES= df_training.shape[0]
N_TIMES = 100
N_FEATURES = df_training.shape[1]
x_train= create_sequences(df_training, N_SAMPLES, N_TIMES, N_FEATURES)
x_train = x_train.astype(np.float32)
x_train.shape
```

El algoritmo realiza la transformación y es lo que se entrega a la red neuronal convolucional

Red convolucional

Se trabaja la red convolucional de una sola dirección porque nos interesan los datos que se encuentran en una sola fila.

La variable respuesta permanece como estaba ya que se encuentra en una sola columna.

Posteriormente se llama a la función create_sequences para transformar los datos de entrenamiento. A continuación se generan los x_train y los x_test

Samples son las filas, n_times son las ventanas y n_features son las variables y se utiliza el tipo de datos float.

```
# Generate x_train sequence so that the model can determine the shape of the input and output
df_training = np.array(df_x_train)
N_SAMPLES= df_training.shape[0]
N_TIMES = 100
N_FEATURES = df_training.shape[1]
x_train= create_sequences(df_training, N_SAMPLES, N_TIMES, N_FEATURES)
x_train = x_train.astype(np.float32)
x_train.shape
```

El mismo procedimiento se utiliza para los datos de prueba.

```
# Generate x_test sequence
df_testing = np.array(df_x_test)
N_SAMPLES= df_testing.shape[0]
N_TIMES = 100
N_FEATURES = df_testing.shape[1]
x_test= create_sequences(df_testing, N_SAMPLES, N_TIMES, N_FEATURES)
x_test = x_test.astype(np.float32)
x_test.shape
```

RED NEURONAL CONVOLUCIONAL

Se estructura una red convolucional de una dimensión con 32 neuronas usando un kernel o filtro de 3 unidades con un padding= “same” para que la entrada y la salida sean iguales y no se reduzca la dimensionalidad de los datos, un stride=1 para que el desplazamiento sea de 1 en 1. También presenta un input_shape con el fin de saber la forma de recibir los datos. Además contiene capas de dropout de 0.2 para reducir el overfitting, una capa de flatten para arreglar los datos en una única dimensión y así poder realizar una clasificación normal con las redes neuronales e incluye una capa densa con una activación sigmoid ya que la clasificación es de uno o cero.

```
model = keras.Sequential(
    [
        layers.Conv1D( filters=32, kernel_size=3, padding="same", strides=1, activation="relu", input_shape=(x_train.shape[1], x_train.shape
        layers.Dropout(rate=0.2),
        layers.Conv1D( filters=16, kernel_size=3, padding="same", strides=1, activation="relu" ),
        layers.Dropout(rate=0.2),
        layers.Conv1D( filters=8, kernel_size=3, padding="same", strides=1, activation="relu" ),
        layers.Dropout(rate=0.2),
        layers.Flatten(),
        layers.Dense( 1, activation= 'sigmoid')
    ]
```

```
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv1d_3 (Conv1D)           (None, 100, 32)           1952

 dropout_3 (Dropout)         (None, 100, 32)           0

 conv1d_4 (Conv1D)           (None, 100, 16)           1552

 dropout_4 (Dropout)         (None, 100, 16)           0

 conv1d_5 (Conv1D)           (None, 100, 8)            392

 dropout_5 (Dropout)         (None, 100, 8)            0

 flatten_1 (Flatten)         (None, 800)               0

 dense_1 (Dense)             (None, 1)                 801

=================================================================
Total params: 4,697
Trainable params: 4,697
Non-trainable params: 0
```

Compilación

La función de pérdida (loss) es el error cuadrático medio (MSE) y la métrica es el accuracy. Para el entrenamiento no se le incluyeron los datos de validación y y_train permanece igual.

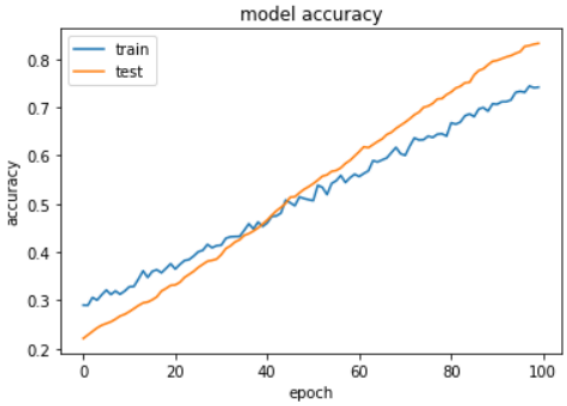
Se realizaron 100 iteraciones por lotes de 100 registros.

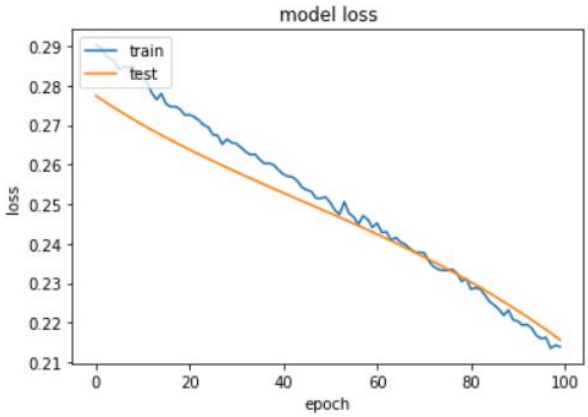
De estos datos se extrajo un porcentaje de 20% para validación dentro de los lotes de entrenamiento.

```
history = model.fit(
            x_train,
            y_train,
            epochs = 100,
            batch_size=100,
            validation_split=0.2,
            callbacks=[keras.callbacks.EarlyStopping(monitor="val_loss", patience=5, mode="min")]
        )
```

RESULTADOS

Comienza con una función de loss de 28.12% y un accuracy de 39.07% y con cada iteración el accuracy se incrementa un poco más y la pérdida se reduce. Finalmente termina con un loss de 13.34% y un accuracy de 88.11%.





Se puede notar que tanto el train y el test tienen un patrón similar en la función accuracy y en la función loss.

Al evaluar el modelo, la pérdida (loss) total equivale a 21.51% y el accuracy fue de 83.04% lo que significa que el modelo es muy bueno.

```
model.evaluate(x_test,y_test)
```

```
72/72 [==============================] - 1s 9ms/step - loss: 0.2151 - accuracy: 0.8304
[0.2150510847568512, 0.830361008644104]
```

La matriz de confusión compara y_test con las predicciones y muestra que no se equivocó en ninguno de los datos.

```
test_preds = model.predict(x_test).argmax(axis=1)
confusion = tf.math.confusion_matrix(labels=y_test, predictions=test_preds, num_classes=2)
print(confusion)
```

```
tf.Tensor(
[[2040    0]
 [ 259    0]], shape=(2, 2), dtype=int32)
```

La conclusión es que la red neuronal es demasiado compleja para la naturaleza de estos datos.