UNIVERSIDAD CATÓLICA DE MURCIA
ALUMNOS INTERNOS CURSO 2022/2023
(Convocatoria 01/10/2022)

INFORME DE ALUMNOS INTERNOS

APELLIDOS, NOMBRE: Maria Isabel Cornejo Rojas
PASAPORTE: 120013359
TITULACIÓN: Departamento de Ingeniería Informática
GRUPO DE INVESTIGACIÓN: Bioinformatics and High Performance Computing BIO-HPC

1. INTRODUCCIÓN

En el siguiente documento se encontrará el proceso de entrenar una red neuronal, comenzando por adquirir el conocimiento necesario para entenderlo, realizando pruebas iniciales como hacer ejemplos guiados de redes neuronales, entrenamiento de algoritmos de aprendizaje automático, limpieza y análisis de data, hasta finalmente entrenar mi propia red neuronal con el dataset que se escogió en su momento.

Asimismo, se encontrarán los resultados de diferentes pruebas hechas, muestras de los resultados obtenidos en la limpieza de datos, cómo cambió el dataset luego de realizar modificaciones y más.

Finalmente, una pequeña discusión respecto a lo trabajado, los resultados obtenidos, comentarios sobre la dificultad, dedicación y tiempo que requiere, y su relevancia hoy en día.

2. MEMORIA

Este apartado se dividirá en cinco subapartados, que irá de manera cómo se fue avanzando en este proyecto.

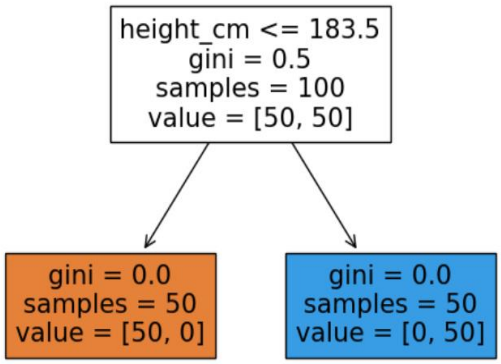
2.1. Entrenamiento de algoritmos de aprendizaje automático

Lo que se realizó aquí fue programar un árbol de decisión, random forest y boosting, el objetivo era comprobar cuál tiene mejor precisión respecto al dataset que se utilizó (proveído por mis docentes)

El dataset constaba de 100 datos con 6 atributos: id, height_cm, age, num tattoos, piercings, category. Cada algoritmo debía precisar de cuál categoría correspondía cada id dependiendo de lo que se había entrenado, en cada uno se utilizaron las métricas de precisión, recall y f1, además se graficó una matriz de confusión para una mejor visualización.

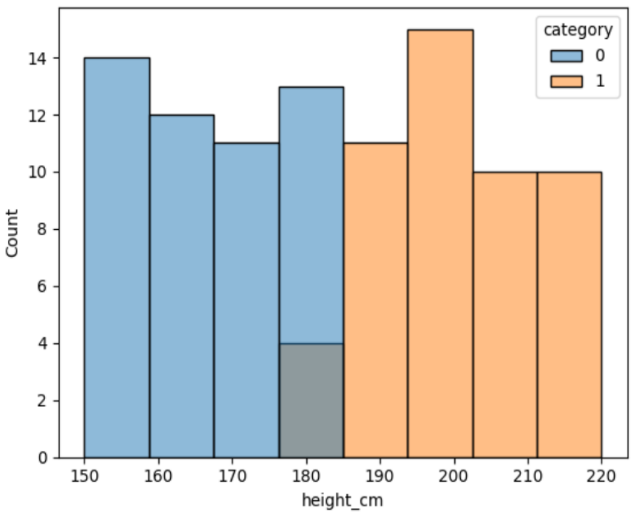
2.1.1. Árbol de decisión

Aquí la experimentación fue más extensa, primero se graficó cuantas ramas creaba el árbol para realizar la predicción.



Al principio no comprendía porque solo era de un nivel, luego de investigar, al ser solo 100 datos los que se están tratando, considerado un dataset pequeño, es mucho más rápido y sencillo realizar un entrenamiento.

Al observar el árbol creado, se entiende que se considera el atributo height_cm como determinante por lo que utilicé un histograma para realizar un mapeo de los datos.



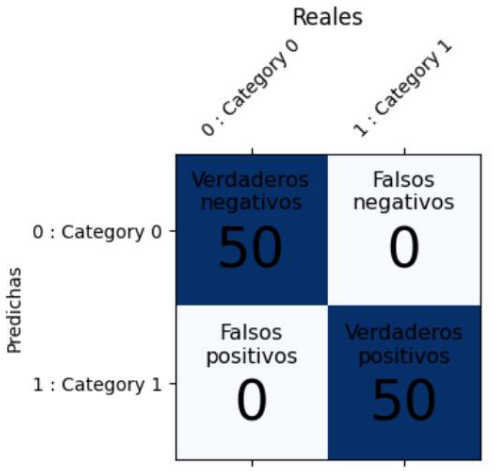
En este histograma lo que entendemos es que hay tantos individuos con tal altura, y que la mitad del dataset es mitad categoría 0 y la otra mitad 1.

Podemos decir que se utiliza la altura para predecir a qué categoría pertenece cada uno.

Lo siguiente que realizamos es una pequeña comparación entre los datos que tenemos en el dataset respecto a categoría, con lo que predecimos y podemos observar que son iguales en todos (al menos los extraídos)

Graficamos una matriz de confusión para comprobar la exactitud y obtenemos un resultado perfecto de 1, al igual que en las métricas utilizadas.

	category	pred
7	1	1
60	0	0
30	0	0
13	0	0
8	0	0
54	1	1
11	0	0
15	0	0
78	1	1
87	0	0



Accuracy: 1.00
Precision: 1.00
Recall: 1.00
F1: 1.00

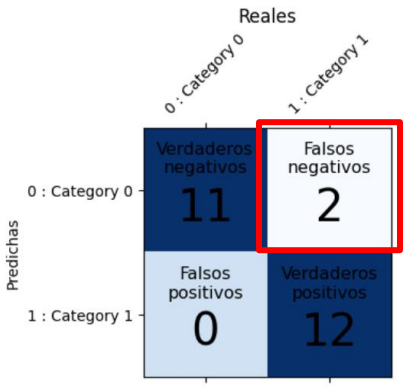
2.1.2. Random forest

Para este algoritmo solo graficamos la matriz, lo interesante de este fue la combinación de parámetros que tuve que realizar para poder obtener resultados altos. Al final se entrenó de la siguiente manera:

```
model = RandomForestClassifier(n_estimators = 5, random_state=25, min_samples_leaf=8,)
```

Y obtuvimos los siguientes resultados:

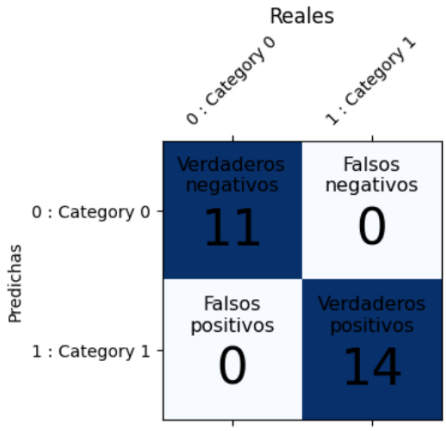
Como observamos, de exactitud se obtuvo un 0.92, sin embargo, con la métrica de precisión se obtuvo un 1, con recall tuvimos el resultado más bajo de 0.86 y de F1, al igual que exactitud, un 0.92.
En la matriz tenemos 2 falsos negativos respecto a la categoría 0 que se clasificaron como categoría 1, pero en realidad son 0.



Accuracy: 0.92
Precision: 1.00
Recall: 0.86
F1: 0.92

2.1.3. Boosting

En este algoritmo no hubo una gran necesidad de cambiar los parámetros, solo se utilizó un número de estimadores de 5 y obtuvimos los siguientes resultados.



Accuracy: 1.00
Precision: 1.00
Recall: 1.00
F1: 1.00

Observamos que obtenemos una exactitud de 1 y lo mismo con las demás métricas.

Podemos concluir que tanto un árbol de decisión como el boosting nos darían resultados de exactitud 1, lo que es excelente, pero por la menor complejidad al momento de entrenar, concluiría que el árbol de decisión respecto a este ejemplo puede ser mejor.

Por otro lado, tal vez el random forest estudiando bien sus parámetros y analizando el dataset y configurando todo y estudiando los resultados, se podría obtener también una exactitud de 1.

2.2. Elección de dataset

Adentrándonos en lo que es el proyecto principal, el dataset seleccionado es en base al estudio de imágenes de cáncer de mama, en este se clasifican los tumores por benignos o malignos. Este dataset consta de 569 filas y 32 columnas, las dos primeras son el id y el diagnóstico (diagnóstico) lo que queremos predecir con nuestra red neuronal. Los atributos que tenemos son todos relacionados con la descripción de imágenes, algunos son la media del perímetro, media del radio, media de la textura y más.

En el apartado de bibliografía se incluye la fuente.

2.3. Preprocesamiento y limpieza de dataset

Se han realizado diversos métodos para conocer más el dataset y también para prepararlo y tenerlo listo cuando se utilice para el entrenamiento de la red neuronal. En este apartado se explicarán los métodos usados para estos procesos.

2.3.1. Cálculo de varianza

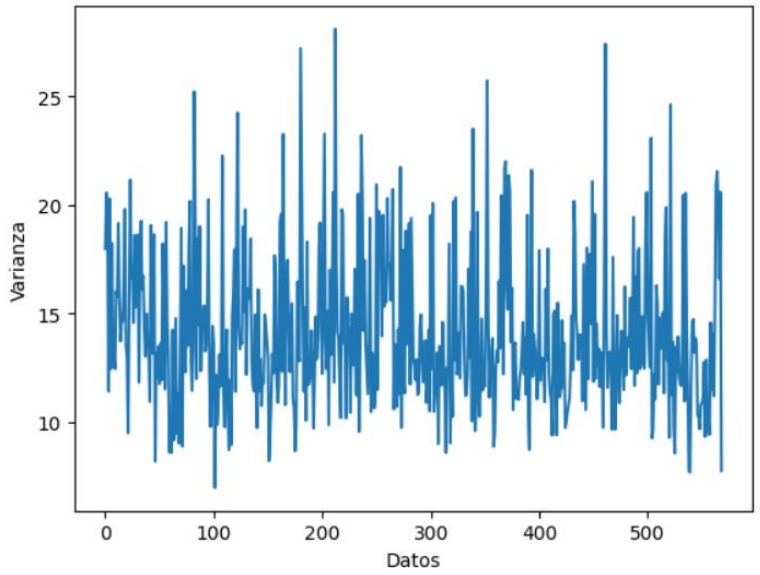
Se realizó una función para calcular la varianza de todos los datos por cada columna.

Luego se llamó a la función para cada columna, asimismo gráfica un plot donde muestra como varían las varianzas.

```
atri = list(data.columns)
```

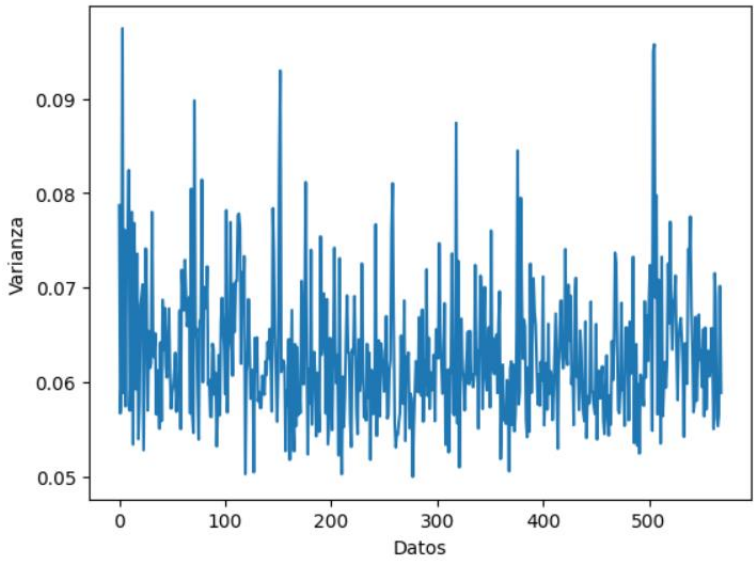
```
i = 0
def pintargrafico(data,atri,i):
    print("Atributo: " + atri[i])
    datitos = data.iloc[:, i].tolist()
    vdat = st.variance(datitos)
    print(f"Varianza: ")
    print("{0:.7f}".format(vdat))
    plt.plot(datitos)
    plt.xlabel("Datos")
    plt.ylabel("Varianza")
```

```
Atributo: radius_mean
Varianza:
12.4189201
```



En esta imagen mostramos el plot de la columna (atributo) radius_mean, cuya varianza es de 12.4189201. Este proceso se repitió 29 veces más, así obtuvimos la varianza por cada atributo, sin embargo, esto solo fue un pasó para reconocer que había algunas columnas que no nos iba a ser de mayor utilidad, tenemos el siguiente ejemplo:

```
Atributo: fractal_dimension_mean
Varianza:
0.0000498
```



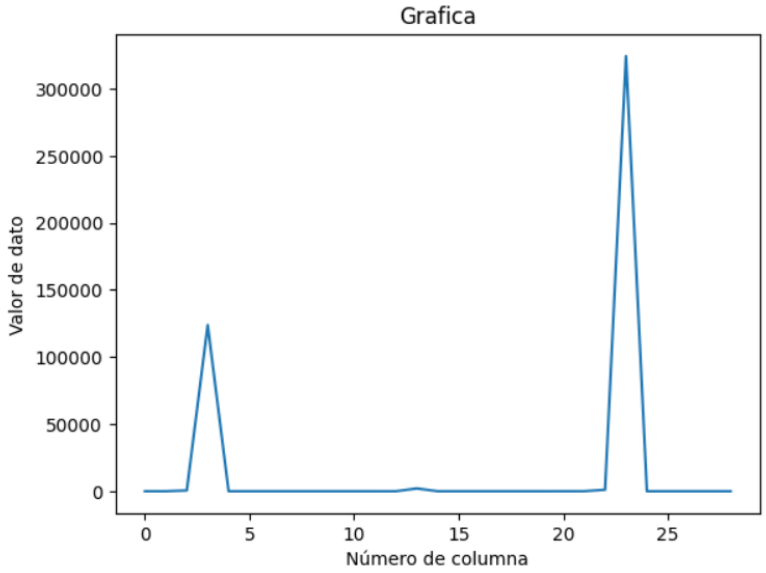
El atributo fractal_dimension_mean muy rara vez llega a más de 0.09 y la varianza es de 0.0000498, al momento de realizar el entrenamiento, la red neuronal no va a utilizar realmente este dato porque llega a no ser relevante.

2.3.2. Normalización de varianzas

La normalización nos sirve para simplificar el análisis y garantizar que los datos se encuentren en una escala comparable y coherente, por ello normalizamos las varianzas obtenidas por cada atributo, obtenemos los siguientes resultados:

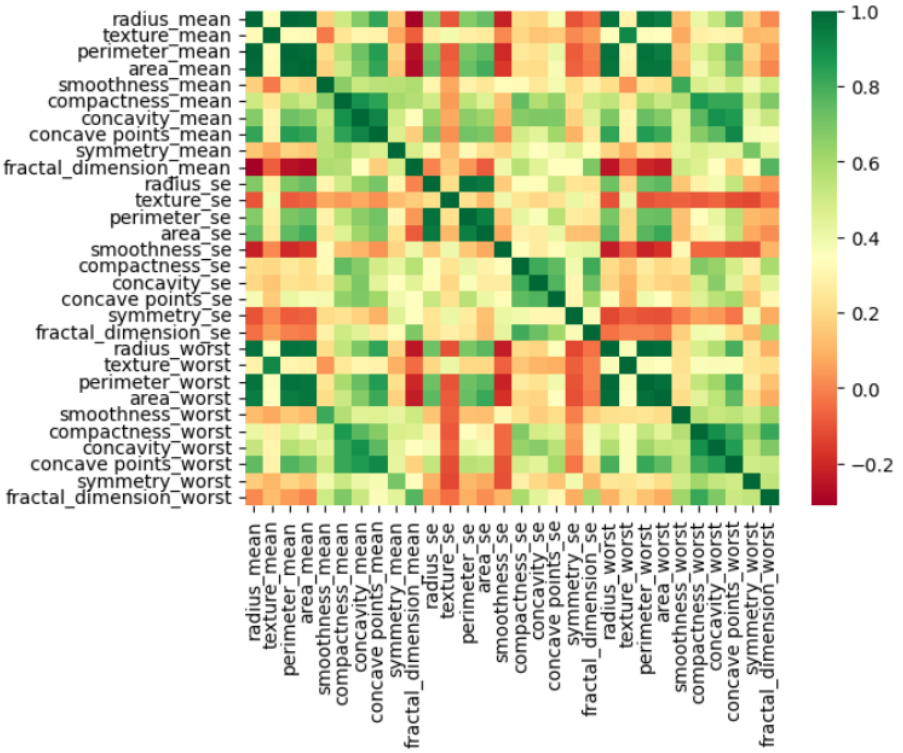
```
Varianzas sin normalizar:
[12.419, 18.499, 590.44, 123843.554, 0.0, 0.003, 0.006, 0.002, 0.001, 0.0, 0.077, 0.304, 4.088, 2069.432, 0.0, 0.0, 0.001,
0.0, 0.0, 0.0, 23.36, 37.776, 1129.131, 324167.385, 0.001, 0.025, 0.044, 0.004, 0.004]

Media de varianzas sin normalizar: 15582.64

Desviación estándar de varianzas sin normalizar: 62528.859

Varianzas normalizadas:
[-0.249, -0.249, -0.24, 1.731, -0.249, -0.249, -0.249, -0.249, -0.249, -0.249, -0.249, -0.249, -0.249, -0.216, -0.249, -0.24
9, -0.249, -0.249, -0.249, -0.249, -0.249, -0.249, -0.231, 4.935, -0.249, -0.249, -0.249, -0.249, -0.249]
```

Lo que observamos en la imagen son todas las varianzas sin normalizar y la media de estas, luego obtenemos las varianzas normalizadas, y graficamos cuál es el comportamiento de estas:



2.3.3. Correlación

Por otro lado, hemos obtenido la correlación entre los atributos, esto nos sirve para evaluar las relaciones o asociaciones entre estos, lo graficamos con un mapa de calor:



Interpretamos que los que son de color más cálido tienen menos correlación que los de color verde, también concluimos que con tener la mitad del mapa es suficiente ya que a partir de la línea diagonal verde del medio este es modo espejo, es decir, todo lo de arriba de la diagonal es espejo con lo de abajo.

2.3.4. Cálculo de outliers

Adentrándonos en como modificaremos nuestro dataset para que tenga más contenido útil, el primer paso que damos es el de eliminación de outliers. Básicamente utilizamos la siguiente función para eliminar todos los datos que se escapan fuera del rango que establecemos:

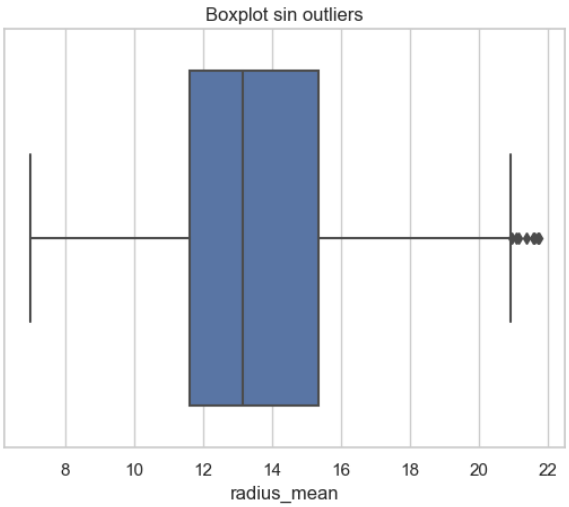
```
def eliminarOutliers(data, atri):
    for i in range(len(atri)):
        sns.set(style="whitegrid")

        print("Atributo: " + atri[i])
        datitos = data.loc[:, atri[i]].tolist()

        # Calculamos los cuartiles
        Q1 = np.percentile(datitos, 25)
        Q3 = np.percentile(datitos, 75)

        # Calcular el rango intercuartil (IQR)
        IQR = Q3 - Q1

        # Calcular los bigotes superior e inferior
        bigote_superior = Q3 + 1.5 * IQR
        bigote_inferior = Q1 - 1.5 * IQR

        print("Bigote superior:", bigote_superior)
        print("Bigote inferior:", bigote_inferior)

        # Eliminar los outliers de la base de datos
        data = data[(data[atri[i]] >= bigote_inferior) & (data[atri[i]] <= bigote_superior)]

        # Graficar el boxplot sin outliers
        sns.boxplot(x=data[atri[i]])
        plt.title("Boxplot sin outliers")
        plt.show()

    return data
```

Al llamar a la función nos da como resultado no solo cada grafica (boxplots) de outliers por atributo, sino también nos devuelve un dataset limpio.

```
columnas = data.columns.tolist()

for columna in columnas[2:]:
    data = eliminarOutliers(data, [columna])
```

Un ejemplo de cómo se grafica el boxplot sería el siguiente, usando el atributo radius_mean:

```
Atributo: radius_mean
Bigote superior: 21.8
Bigote inferior: 5.540000000000002
```



Los datos regularmente se encuentran entre 11 y 16, teniendo los limites marcados en 21.8 y 5.54, interpretamos a los que se encuentran fuera de estos como outliers.

2.4. Modificación de dataset

Luego de haber utilizado la anterior función para la modificación del dataset, este termina siendo de un tamaño de 32 columnas con 233 filas, en un principio eran 569.

```
data
```

	id	diagnosis	radius_mean	texture_mean	perimeter_mean	area_mean	smoothness_mean	compactness_mean	concavity_mean	concave points_mean	...
19	8510426	B	13.540	14.36	87.46	566.3	0.09779	0.08129	0.06664	0.04781	...
20	8510653	B	13.080	15.71	85.63	520.0	0.10750	0.12700	0.04568	0.03110	...
21	8510824	B	9.504	12.44	60.34	273.9	0.10240	0.06492	0.02956	0.02076	...
37	854941	B	13.030	18.42	82.61	523.8	0.08983	0.03766	0.02562	0.02923	...
40	855167	M	13.440	21.58	86.18	563.0	0.08162	0.06031	0.03110	0.02031	...
...	...	...	...	...	...	...	...	...	...	...	...
551	923780	B	11.130	22.44	71.49	378.4	0.09566	0.08194	0.04824	0.02257	...
552	924084	B	12.770	29.43	81.35	507.9	0.08276	0.04234	0.01997	0.01499	...
554	924632	B	12.880	28.92	82.50	514.3	0.08123	0.05824	0.06195	0.02343	...
555	924934	B	10.290	27.61	65.67	321.4	0.09030	0.07658	0.05999	0.02738	...
560	925292	B	14.050	27.15	91.38	600.4	0.09929	0.11260	0.04462	0.04304	...

233 rows × 32 columns

2.5. Entrenamiento de red neuronal

En este apartado se comentará cómo se entrenó a la red neuronal y el código que se utilizó, en el apartado 3 se mostrarán los resultados e interpretabilidad.

```
#Entrenamiento red neuronal
from keras.models import Sequential
from keras.layers import Dense, Dropout
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder, MinMaxScaler

# Preprocesamiento de datos
label_encoder = LabelEncoder()
labels = label_encoder.fit_transform(data['diagnosis'])

scaler = MinMaxScaler()
features = data.drop(['id', 'diagnosis'], axis=1)
features_scaled = scaler.fit_transform(features)

# División de datos en conjunto de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(features_scaled, labels, test_size=0.3, random_state=22)

# Definición del modelo
model = Sequential()
model.add(Dense(256, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dropout(0.5))
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(88, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(64, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(1, activation='sigmoid'))

# Compilación del modelo
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])

# Entrenamiento del modelo
model.fit(X_train, y_train, epochs=30, batch_size=32, verbose=1)

# Evaluación del modelo en el conjunto de prueba
loss, accuracy = model.evaluate(X_test, y_test)
print('Loss:', loss)
print('Accuracy:', accuracy)
```

Procesamiento de datos:

Antes de entrenar la red neuronal, se realiza un proceso de preprocesamiento de los datos. Se utiliza el codificador de etiquetas “LabelEncoder” para transformar las etiquetas de la variable "diagnosis" en valores numéricos. Además, se utiliza la técnica de escalamiento “MinMaxScaler” para normalizar las características del conjunto de datos, lo que asegura que todas las características se encuentren en un rango similar.

División de datos:

Los datos se dividen en conjuntos de entrenamiento y prueba mediante la función “train_test_split” de Scikit-learn. Se utiliza un tamaño de prueba del 30% y una semilla aleatoria (random_state) de 22 para garantizar la reproducibilidad de los resultados.

Arquitectura del modelo:
La red neuronal se construye utilizando el modelo secuencial de Keras. La arquitectura consta de varias capas densas (fully connected) con funciones de activación “ReLU”, también se agregan capas de “Dropout” para mitigar el sobreajuste del modelo al eliminar aleatoriamente conexiones entre las neuronas durante el entrenamiento.

Entrenamiento del modelo:
El modelo se compila utilizando la función de pérdida “binary_crossentropy” y el optimizador “Adam”. Se entrena utilizando el conjunto de entrenamiento con un total de 30 épocas y un tamaño de lote de 32. El proceso de entrenamiento se realiza con verbosidad (verbose) para mostrar el progreso en cada época.

Evaluación del modelo:
Una vez finalizado el entrenamiento, se evalúa el modelo utilizando el conjunto de prueba. Se calcula la pérdida (loss) y la precisión (accuracy) del modelo. La pérdida indica qué tan bien se ajustan las predicciones del modelo a los valores reales, mientras que la precisión representa la proporción de predicciones correctas en relación con el total de predicciones.

Por otro lado, también utilizó la métrica AUC para evaluar el entrenamiento de nuestra red neuronal:

```
#Probando con AuC
from sklearn.metrics import roc_auc_score
import tensorflow as tf

# Definir el modelo
model = Sequential()
model.add(Dense(120, activation='relu', input_shape=(X_train.shape[1],)))
model.add(Dense(100, activation='relu'))
model.add(Dense(80, activation='relu'))
model.add(Dense(20, activation='relu'))
model.add(Dense(1, activation='sigmoid'))

# Compilar el modelo con la métrica AUC
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=[tf.keras.metrics.AUC()])

# Entrenar el modelo
model.fit(X_train, y_train, epochs=11, batch_size=20)

# Realizar la predicción en el conjunto de prueba
y_pred = model.predict(X_test)

# Calcular el AUC
auc = roc_auc_score(y_test, y_pred)
print('AUC:', auc)
```

Definiendo el modelo de la red neuronal, este consta de varias capas densas con funciones de activación ReLU, seguidas de una capa de salida con una función de activación sigmoide. Utilizamos la función de pérdida "binary_crossentropy" y el optimizador "adam" para compilar el modelo.

Luego, entrenamos el modelo utilizando los datos de entrenamiento, ajustamos el modelo a los datos de entrada "X_train" y las etiquetas correspondientes "y_train". Especificamos el número de épocas y el tamaño del lote para el entrenamiento. Después, el modelo ha aprendido a asignar correctamente las características de entrada a las etiquetas de salida.

A continuación, realizamos la predicción en el conjunto de prueba utilizando el modelo entrenado. El resultado de la predicción se almacena en "y_pred".

Finalmente, calculamos el AUC comparando las etiquetas reales del conjunto de prueba "y_test" con las predicciones del modelo "y_pred" utilizando la función "roc_auc_score" de la biblioteca sklearn.metrics. El AUC proporciona una medida de qué tan bien el modelo puede clasificar las muestras positivas y negativas. Cuanto más cercano a 1 sea el valor del AUC, mejor será el rendimiento del modelo.

Finalmente, se realizó una función que nos permite verificar, pasándole cierto ID, si el tipo de benigno “B” o maligno “M”

```python
def predecirMalignidad(id_especifico, model, scaler, data):

    if id_especifico not in data['id'].values:
        return "ID no encontrado"

    datos_id = data.loc[data['id'] == id_especifico, data.columns[2:]]

    if datos_id.empty:
        return "Datos vacíos para el ID específico"

    datos_id = scaler.transform(datos_id)

    prediccion = model.predict(datos_id)

    if prediccion[0] < 0.5:
        resultado = "Benigno"
    else:
        resultado = "Maligno"

    return resultado
```

La función la utilizamos para predecir la malignidad de un caso específico identificado por el ID. Toma como entrada el ID específico, el modelo previamente entrenado, un escalador de datos y el dataset.

En primer lugar, verifica si el ID proporcionado está presente en los datos. Si no se encuentra, se devuelve el mensaje "ID no encontrado". A continuación, se seleccionan los datos correspondientes al ID específico del conjunto de datos.

Si los datos para el ID están vacíos, se devuelve el mensaje "Datos vacíos para el ID específico". Luego, los datos se transforman utilizando el escalador proporcionado y se realiza la predicción utilizando el modelo entrenado sobre los datos transformados. La predicción representa la probabilidad de malignidad.

Si la predicción es menor a 0.5, se asigna el resultado "Benigno"; de lo contrario, se asigna "Maligno". La elección de considerar una predicción menor a 0.5 como "Benigno" se basa en el contexto del problema. En este caso, se utiliza una función de activación sigmoidal en la capa de salida del modelo, que produce valores en el rango de 0 a 1.

Al establecer un umbral de 0.5, se toma una decisión de clasificación: si la probabilidad de la clase positiva (maligno) es menor a 0.5, se considera como clase negativa (benigno), y si es mayor o igual a 0.5, se considera como clase positiva.

Por último, se devuelve el resultado de la predicción, que indica la malignidad del caso específico identificado por el ID.

3. RESULTADOS

En este apartado se presentarán los resultados obtenidos de la red neuronal, los de la métrica AUC y de la función que se ha creado.

3.1. Resultados red neuronal

```
Epoch 29/30
13/13 [==============================] - 0s 4ms/step - loss: 0.0416 - accuracy: 0.9899
Epoch 30/30
13/13 [==============================] - 0s 3ms/step - loss: 0.0652 - accuracy: 0.9774
6/6 [==============================] - 0s 3ms/step - loss: 0.1816 - accuracy: 0.9532
Loss: 0.18161490559577942
Accuracy: 0.9532163739204407
```

Podemos observar que se ha obtenido una exactitud de 0.9532 y una perdida de 0.1816, en general, un accuracy de este nivel demuestra que el modelo es capaz de predecir con precisión la clase correcta en aproximadamente el 95.32% de las muestras de prueba.

Es importante tener en cuenta que el desempeño de un modelo puede variar dependiendo del problema y del conjunto de datos específicos. En este caso particular, alcanzar un accuracy de 0.9532 indica que el modelo tiene un buen rendimiento y es prometedor en términos de su capacidad para realizar predicciones precisas.

En resumen, obtener un accuracy de 0.9532 con Keras es un resultado positivo y muestra que el modelo tiene un buen rendimiento en la tarea de clasificación. Esto respalda la confianza en la capacidad de mi red neuronal entrenada y proporciona una base sólida para continuar explorando y mejorando el modelo en futuros proyectos.

3.2. Resultados AUC

```
Epoch 1/11
20/20 [==============================] - 1s 3ms/step - loss: 0.6271 - auc: 0.7039
Epoch 2/11
20/20 [==============================] - 0s 3ms/step - loss: 0.4728 - auc: 0.9216
Epoch 3/11
20/20 [==============================] - 0s 2ms/step - loss: 0.3368 - auc: 0.9619
Epoch 4/11
20/20 [==============================] - 0s 3ms/step - loss: 0.2282 - auc: 0.9779
Epoch 5/11
20/20 [==============================] - 0s 2ms/step - loss: 0.1481 - auc: 0.9901
Epoch 6/11
20/20 [==============================] - 0s 2ms/step - loss: 0.1059 - auc: 0.9944
Epoch 7/11
20/20 [==============================] - 0s 2ms/step - loss: 0.0972 - auc: 0.9941
Epoch 8/11
20/20 [==============================] - 0s 4ms/step - loss: 0.0693 - auc: 0.9978
Epoch 9/11
20/20 [==============================] - 0s 4ms/step - loss: 0.0705 - auc: 0.9965
Epoch 10/11
20/20 [==============================] - 0s 3ms/step - loss: 0.0565 - auc: 0.9983
Epoch 11/11
20/20 [==============================] - 0s 4ms/step - loss: 0.0517 - auc: 0.9989
6/6 [==============================] - 0s 2ms/step
AUC: 0.9854929577464789
```

Obtener un valor de 0.9854 para el Área bajo la Curva (AUC) al utilizar el modelo entrenado con el métrica AUC es un resultado muy bueno. El AUC es una métrica que evalúa la capacidad del modelo para clasificar correctamente los casos positivos frente a los negativos, y un valor de 0.9854 indica un desempeño muy sólido.

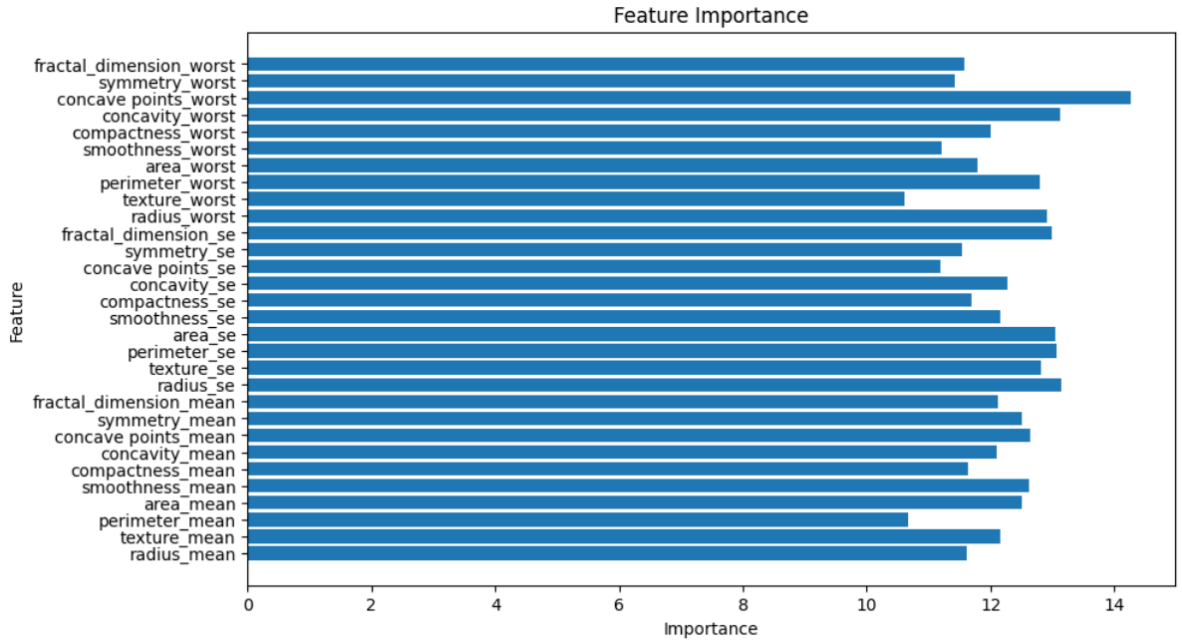
Un AUC alto demuestra que el modelo tiene una capacidad excelente para distinguir entre las clases de manera precisa y consistente. En este caso, el valor de 0.9854 indica que mi modelo tiene una alta probabilidad de asignar una puntuación de probabilidad más alta a las muestras positivas en comparación con las negativas, lo cual es muy deseable en tareas de clasificación.

Es importante destacar que el AUC es una métrica robusta y ampliamente utilizada en la evaluación de modelos de clasificación, ya que considera la relación entre la tasa de verdaderos positivos y la tasa de falsos positivos en diferentes puntos de corte de probabilidad.

Este resultado respalda la confianza en la capacidad de mi modelo para realizar predicciones precisas y confiables, lo cual es fundamental en muchos escenarios de aplicación donde la precisión y la confiabilidad son críticas.

3.3. Interpretabilidad

Finalmente, ¿Cómo podemos entender a raíz de qué el modelo ha aprendido que tipo de tumor es cada uno? Se realizó el siguiente gráfico para ayudar a la comprensión y la interpretabilidad de este problema.



Estudiando y entendiendo la gráfica, el modelo para aprender a predecir tomo cómo el atributo más importante “concave points_worst”, esta es una característica específica de los puntos más cóncavos de un tumor maligno en un estudio de diagnóstico del cáncer de mama.

Los puntos más cóncavos son áreas del contorno del tumor que presentan una curvatura pronunciada hacia adentro. La medida de "concave points worst" proporciona información sobre la forma y la irregularidad del tumor. Un valor alto de "concave points worst" indica la presencia de más puntos cóncavos y una mayor irregularidad en el contorno del tumor.

En el contexto de un modelo de clasificación de cáncer de mama, esta característica puede ser relevante para determinar la probabilidad de malignidad de un tumor. Un valor alto de "concave points worst" puede indicar un mayor riesgo de que el tumor sea maligno, ya que la presencia de puntos cóncavos pronunciados y una forma irregular pueden ser signos de crecimiento y comportamiento maligno del tumor.

Sabiendo esto, tendría mucha lógica que, a raíz de ese atributo, el modelo considerase que es clave para realizar la predicción.

3.4. Resultados función

En la primera imagen vemos el resultado de cuando pasamos un ID y nos devuelve el tipo, esta función el 95% de las veces nos lo predice de manera correcta, lo podemos comparar con la captura del listado de los datos (Segunda imagen)

```
predecirMalignidad(866083, model, scaler, data)
1/1 [==============================] - 0s 122ms/step
'Maligno'
```

125	86561	B	13.850	17.21	88.44	588.7
126	866083	M	13.610	24.69	87.76	572.6
130	866714	B	12.190	13.29	79.08	455.8

Pero también nos lo predice de manera incorrecta, por ejemplo:

Tenemos el ID 855167, que, verificando en el listado de datos, es Maligno, la función nos devuelve lo siguiente:

```
predecirMalignidad(855167, model, scaler, data)
1/1 [==============================] - 0s 56ms/step
'Benigno'
```

Se concluye que ajustando más los hiper parámetros se podría alcanzar una mayor exactitud, sin embargo, se realizó un pequeño experimento en el cual se creó una función para que se auto elija la mejor combinación de hiper parámetros cuando dé la mejor exactitud.

El resultado que se obtuvo es que la media de todas las pruebas se lograba una exactitud del 94% aproximadamente, lo que nos hace pensar que no solo es como ajustemos los parámetros para el entrenamiento, sino también la complejidad del problema y de los datos.

4. DISCUSIÓN

Los resultados obtenidos en la red neuronal entrenada utilizando Keras son bastante alentadores. Se logró un accuracy de 0.9532, lo que significa que el modelo es capaz de predecir correctamente la clase de un 95.32% de las muestras de prueba, ello indica que el modelo tiene una buena capacidad de clasificación precisa.

Además, se obtuvo un valor de 0.9854 para el Área bajo la Curva (AUC). Esta métrica es muy positiva, ya que cuanto más cercano esté el valor de AUC a 1, como ya se ha mencionado, mejor será el rendimiento del modelo en términos de su capacidad para clasificar correctamente los datos positivos y negativos. Con un valor de 0.9854, se puede asegurar de que mi modelo tiene una alta capacidad de discriminación y es muy efectivo en la tarea de clasificación binaria.

En general, estos resultados demuestran que la red neuronal entrenada con Keras tiene un rendimiento sólido y es capaz de realizar predicciones precisas en la tarea específica para la cual fue entrenada. Sin embargo, es importante tener en cuenta que la evaluación del rendimiento de un modelo no debe basarse únicamente en una métrica, como el accuracy o el AuC.

5. BIBLIOGRAFÍA

Dataset de cáncer de mama de Wisconsin (Breast Cancer Wisconsin Data) en Kaggle:
UCI Machine Learning Repository. (2019). Breast Cancer Wisconsin (Diagnostic) Data Set. Recuperado de https://www.kaggle.com/uciml/breast-cancer-wisconsin-data

Documentación de scikit-learn para aprendizaje supervisado:
scikit-learn. (Consultado en 2023). Supervised Learning. Recuperado de https://scikit-learn.org/stable/supervised_learning.html#supervised-learning

Artículo científico sobre detección de cáncer de mama utilizando técnicas de aprendizaje automático:
Alom, M. Z., Yakopcic, C., Taha, T. M., Asari, V. K., & van Essen, H. (2021). A Comprehensive Study on Machine Learning Techniques for Breast Cancer Prediction. Diagnostics, 13(7), 1238. DOI: 10.3390/diagnostics13071238

Artículo de blog sobre la comprensión de la curva AUC-ROC:
Niranjan, V. (2019). Understanding AUC-ROC Curve. Towards Data Science. Recuperado de https://towardsdatascience.com/understanding-auc-roc-curve-68b2303cc9c5

Artículo de blog sobre la comprensión de la matriz de confusión:
Bansal, R. (2021). In-Depth Understanding of Confusion Matrix. Analytics Vidhya. Recuperado de https://www.analyticsvidhya.com/blog/2021/05/in-depth-understanding-of-confusion-matrix/

Artículo científico sobre el uso de técnicas de aprendizaje automático para el diagnóstico de cáncer de mama:
Pathak, P. M., Shingavi, R. M., & Patil, P. M. (2020). Deep Learning Approach for Breast Cancer Diagnosis Using Transfer Learning. 2020 IEEE 17th India Council International Conference (INDICON), 1-6. DOI: 10.1109/INDICON50604.2020.9328789