Tecnológico de Monterrey

Instituto Tecnológico y de Estudios Superiores de Monterrey

Reto. Evidencia 3. Solución del Reto

TC3004B.104 Inteligencia Artificial Avanzada para la Ciencia de Datos I

Profesores:

Ivan Mauricio Amaya Contreras

Blanca Rosa Ruiz Hernandez

Antonio Carlos Bento

Frumencio Olivas Alvarez

Hugo Terashima Marín

Equipo 2

Integrantes:

Luis Ángel Guzmán Iribe - A01741757

Julian Lawrence Gil Soares - A00832272

Alberto H Orozco Ramos - A00831719

13 de Septiembre de 2023

Introducción

Preámbulo

Este documento consiste de una síntesis del trabajo realizado durante la materia Inteligencia artificial avanzada para la ciencia de datos I (TC3006C.101), la cual sirve como curso introductorio a temas de analitica de datos, construcción y uso de modelos de machine learning y evaluación de los mismos.

Este documento contiene únicamente únicamente nuestros resultados, para encontrar el código y leer en detalle sobre las metodologías utilizadas, consulte los archivos en este repositorio: https://github.com/4lb3rt0r/TC3006_Equipo2/tree/main/final

Reto

Como proyecto final del curso, se trabaja sobre la competencia de Kaggle Titanic - Machine Learning from Disaster, la cual es conocida en círculos de machine learning como un ejercicio clásico en la introducción al tema. El reto consiste en desarrollar un modelo de machine learning capaz de predecir si un determinado pasajero del Titanic sobrevivió o falleció en el undimiento del mismo tomando en cuenta información como su sexo, edad del pasajero, número de familiares a bordo, entre otras variables. El modelo generado debe de ser capaz de clasificar a los pasajeros como falso o positivo si dependiendo de las características del mismo, este hubiera sobrevivido el hundimiento del Titanic.

La competencia tiene como objetivo introducir a los participantes al desarrollo de modelos de machine learning por medio de la experimentación con diferentes algoritmos buscando encontrar la solución más adecuada para el reto, en el contexto del curso, sirve como una aplicación práctica de los temas vistos en clase.

Objetivos

- Comprender y explorar el conjunto de datos de la competencia en búsqueda de las variables más relevantes para la generación del modelo predictivo.
- Limpiar el conjunto de datos a fin de eliminar impurezas como datos atípicos, valores faltantes o realizar transformaciones de variables donde se considere necesario.
- Implementar diferentes algoritmos de machine learning de clasificación.
- Analizar el rendimiento de cada modelo generado y compararlo con los demás para encontrar un modelo preliminar que refinar.
- Refinar el modelo seleccionado para mejorar el desempeño del mismo.

Dataset

En este proyecto se trabaja con el dataset de pasajeros del Titanic, este dataset contiene las variables: *survived*, *pclass*, *sex*, *age*, *sibsp*, *parch*, *ticket*, *fare*, *cabin* y *embarked*; puede leer más a detalle sobre el significado de cada variable en el dataset en la ligra proporcionada anteriormente, pero a modo de resumen, estas variables representan las caracteristicas de los pasajeros embarcados en el Titanic, su clase, sexo, edad, numero de familiares a bordo, el precio de su boleto, etc, meintras que la variable *survided* indica con un 0 o 1 si el pasajero sobrevivió. Es pues el objetivo del reto desarrollar un modelo que

tome en cuenta las características del pasajero para predecir el valor de la variable survived.

Metodología

Limpieza de datos

Para seleccionar variables y limpiar consideramos solo la información relevante para la tendencia de supervivencia. Por ejemplo, mujeres, niños y personas de mejor estatus socioeconómico tenían más probabilidades de sobrevivir. Esto guió nuestro análisis de variables en este documento.

También es importante aclarar que para todos los entrenamientos y pruebas se dividió al conjunto de datos resultado de esta limpieza en 2 subconjuntos correspondientes al 80% y 20% del conjunto de datos original, con la finalidad de evitar situaciones de overfitting y falta de capacidad del modelo de generalizar a datos no vistos.

Análisis de variables

Variables seleccionadas

Las variables que **tomamos en cuenta** para la limpieza de los datos son los siguientes:

- **Survived:** Variable objetivo del dataset, es indispensable para el entrenamiento del modelo.

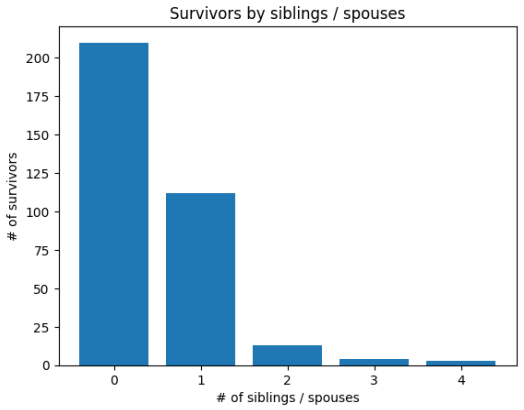
- **PClass**: Este atributo representa la clase en la que viajaba el pasajero. Se cree que a las personas de clases superiores se les dio prioridad para acceder a navíos salvavidas, sin embargo, puede ser que la realidad fuera que al estar más cerca de la cubierta, hubiera sido más fácil escapar de las cabinas de primera clase. Independientemente de la causa, consideramos a esta variable relevante para el análisis.

- **Age:** La edad del pasajero fue otro de los factores determinantes para asignar prioridades en el abordaje de botes salvavidas, por lo tanto, incluir este dato en nuestro conjunto de entrenamiento es fundamental. Es importante destacar que la edad es una de las variables con más valores faltantes. Para abordar este problema, utilizamos el prefijo del nombre para estimar la edad del pasajero basándonos en la mediana de otros pasajeros con el mismo prefijo. Por ejemplo, a los hombres adultos se les asigna el prefijo "Mr.", a las mujeres "Miss." y a los niños "Master.". Para los pasajeros sin ningún prefijo, empleamos la mediana general de los datos.

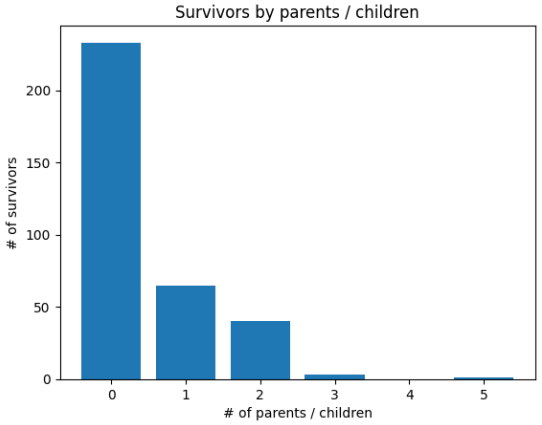
- **Sex:** Al igual que la edad, fue uno de los principales factores para decidir quien subía a los barcos salvavidas, dando preferencia a mujeres sobre hombres.Se puede convertir en un dato booleano para simplificar el trabajo con el parámetro en el en el modelo.

- **SibSp:** Esta variable hace referencia a la cantidad de hermanos y/o cónyuges presentes en el navío. Si bien la importancia de esta variable no fue evidente en un principio, realizamos un gráfico que compara el número de hermanos/conyugues contra el la cantidad de supervivientes. Los resultados revelaron una tendencia

interesante: aquellos con más familiares cercanos tendían a tener menores probabilidades de sobrevivir, y viceversa.



- **Parch:** Similar a Sibsp, esta variable indica el número de hijos y/o padres en el navío que tenía un pasajero. Se realizó un análisis similar al de SibSp y se encontraron los mismos resultados: los pasajeros con más familiares a bordo tenían menos probabilidades de sobrevivir



- **Fare:** Se refiere al precio del boleto pagado. Si bien ya tenemos un indicador de clase socioeconómica (PClass), encontramos que incluso entre pasajeros de la misma clase existían considerables diferencias de precio. Inferimos que aquellos que pagaron más por sus boletos tenían un estatus social más elevado y, por lo tanto, es probable que recibieron un trato preferencial en situaciones críticas.

Variables transformadas

Las variables que se tomaron en cuenta para el análisis pero se **transformaron** o usaron para **inferir valores** de otras variables que, por ejemplo, cuentan con valores nulos.

- **Cabin:** Aunque el número de cabina por sí solo no determina las posibilidades de supervivencia, tener una cabina asignada puede indicar ciertos factores clave. Primero, una cabina privada sugiere una clase más alta, asociada principalmente con la primera clase, lo que es un indicador fuerte de supervivencia. Segundo, tener una

cabina podría facilitar el acceso a la cubierta, lo que aumentaría las probabilidades de escape. Por tanto, creamos la columna booleana *'Has_Cabin'* para considerar estos aspectos.

Variables rechazadas

En cuanto a las variables que **no tomamos en cuenta** para el análisis son:

- **PassengerId**: La ID del pasajero es un identificador útil en muchos contextos para diferentes pasajeros, pero en un análisis estadístico como este, no resulta necesaria.

- **Ticket**: Igual que la ID, sirve para diferenciar pasajeros, pero no proporciona información útil acerca del mismo.

- **Embarked:** El puerto de embarque no proporciona ninguna información útil sobre el pasajero, resulta irrelevante saber desde qué puerto zarpó cada pasajero cuando una vez en el barco resulta más influyente el tratamiento que recibe el pasajero por sus características.

- **Name**: El nombre de los pasajeros no es relevante en este contexto, sin embargo, cabe recalcar que el título del nombre se utiliza en algunos casos para estimar la edad de los pasajeros que no cuentan con dicho registro.

Selección de modelo

Teniendo lista la limpieza y dejado en claro qué tipo y qué variables conforman el set de datos a trabajar para desarrollar el modelo predictivo, es momento de establecer algunos objetivos a cumplir para seleccionar el mejor candidato entre cuatro modelos a probar:

1. *Probar diferentes modelos de Machine Learning que consideramos aptos para llevar a cabo un procedimiento de análisis de los datos de tipo clasificación binaria.*
2. *Evaluar los modelos con la finalidad de seleccionar el que demuestre los mejores resultados con respecto a los demás.*
3. *Definir un punto de referencia con el fin de establecer un objetivo a superar durante la concepción, desarrollo e implementación de estos modelos.*
4. *Proveer el código y metodologías aplicadas con el fin de facilitar su replicación y posteriormente su evaluación.*

Modelos de Machine Learning

Se seleccionaron 4 algoritmos para realizar pruebas de hiper parámetros para encontrar un candidato a modelo final. A continuación se describen los algoritmos a probar.

Decision Tree (Árbol de Decisiones)

El algoritmo de Decision Tree (o árboles de decisión) es un a técnica de machine learning que consiste en el uso de reglas de decisión simple, de forma similar a un diagrama de flujo, aprendidas de un conjunto de datos con la finalidad de predecir valores tomando en cuenta esos mismos parámetros. Consideramos que este es un algoritmo apto para este problema dado que podemos aprovechar de las marcadas divisiones categoricas que tenemos en

nuestro conjunto de datos, como sexo, edad, clase, etc... De forma intuitiva, el modelo preguntaría ¿el pasajero era hombre o mujer? ¿Era niño o adulto? ¿Viajaba en primera clase? y con esto llegaría a una respuesta tomando en cuenta estos valores, asociando un peso diferente a cada variable.

Random Forests (Bosque Aleatorio)

Random Forest (o bosques aleatorios) es un algoritmo que consiste en el uso de *n* árboles de decisión con la finalidad de reducir el ruido presente en un solo arbol, evitar el overfitting, y tomar en cuenta una mayor variabilidad entre los datos; ofreciendo una mayor calidad de los datos a costa de un mayor precio computacional, pero esto no se trata de un problema para fines de este reto dado que trabajamos con un conjunto de datos relativamente pequeño (menor a 1000 filas de datos).

Neural Networks (Redes Neuronales)

Las redes neuronales son modelos computacionales inspirados en el comportamiento del cerebro biológico. Consisten en valores de entrada, que en este caso son variables del archivo de entrenamiento de pasajeros del Titanic. Estos valores pasan por neuronas, a las cuales se les asignan pesos para cada variable. La suma ponderada se procesa con una función matemática, llamada función de activación, para producir la salida final de la neurona.

Este proceso puede repetirse en múltiples capas, potencialmente obteniendo mejores resultados que un enfoque de una sola capa y neurona. Sin embargo, contar con suficientes datos de entrada es crucial para predicciones o clasificaciones precisas.

En el contexto del reto, se eligen las redes neuronales debido a su capacidad de clasificación con datos binarios (0, 1), donde 1 representa supervivencia y 0 no supervivencia. Entre varios modelos, las redes neuronales muestran una mayor precisión debido a su comprensión de la jerarquía de variables, aprendizaje más rápido y configuraciones versátiles. En última instancia, el éxito depende del contexto del problema y encontrar configuraciones óptimas.

Support Vector Machine (Máquina de Vectores de Apoyo)

Support Vector Classification es uno de los algoritmos de clasificación más simples, pero que puede llegar a tener muy buenos resultados. La manera en la que funciona es toma un plano donde se grafica 2 variables, en nuestro caso si el pasajero sobrevivió o murió, luego genera una línea para clasificar las variables. En nuestro caso se toman en cuenta más de dos features para hacer la clasificación, así que se grafica en un plano de múltiples dimensiones y el hiperplano es un plano que divide los puntos entre las dos clases que queremos encontrar.

Métricas de desempeño

Para evaluar el rendimiento de nuestro modelo, hemos decidido elegir las métricas o indicadores más convencionales para medir el rendimiento y la fiabilidad del mismo, que son las siguientes:

- **Matriz de Confusión (*Confusion Matrix*)**: La matriz de confusión es una tabla que provee una descripción detallada del rendimiento del modelo al proyectar las cuentas de verdaderos positivos, falsos positivos, verdaderos negativos y falsos negativos. Además, permite identificar errores específicos que el modelo presenta. Este recurso resulta ser muy valioso, especialmente cuando se necesita evaluar y mejorar el rendimiento del modelo.

$$TP = Verdaderos\ positivos$$
$$FP = Falsos\ positivos$$
$$TN = Verdaderos\ negativos$$
$$FN = Falsos\ negativos$$

- **Exactitud (*Accuracy*)**: Este mide la proporción de instancias que fueron clasificadas correctamente del resto por el modelo. Básicamente, proporciona una medida general de cómo es el modelo, en términos de clasificar correctamente tanto verdaderos positivos y verdaderos negativos.

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}$$

- Precisión (*Precision*): Mide la precisión del modelo con respecto a predicciones positivas. Da respuesta a la pregunta "De todas las instancias estimadas por el modelo ¿cuántas de ellas fueron positivas?" Entre mayor sea este valor, indica que el modelo es bueno al evitar falsos positivos.

$$Precision = \frac{TP}{TP + FP}$$

- **Exhaustividad (*Recall*)**: Este mide la habilidad del modelo para identificar todas las instancias relevantes dentro del dataset. Un valor de "recall" alto indica que el modelo raramente se equivoca en casos positivos. Esta métrica es importante cuando los casos positivos faltantes son bastante costosos, un ejemplo serían los diagnósticos médicos.

$$Precision = \frac{TP}{TP + FN}$$

- **Valor F1 (*F1-Score*)**: La métrica F1 es la media armónica de la precisión con respecto a la exhaustividad. Esto condensa el intercambio de precisión y recall en una sola métrica. Un alto valor de "F1-Score" indica que el modelo demuestra un rendimiento destacado en términos de las variables que considera (precision y recall) Resulta muy útil cuando se busca balancear la precisión y la exhaustividad o bien para identificar una distribución de métricas desigual.

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

Métrica de desempeño para pruebas

Decidimos emplear el **valor F1 del conjunto de predicciones de prueba** como métrica definitiva de desempeño para nuestras pruebas. Tomamos esta decisión porque este valor suele ser utilizado cuando los resultados falsos tienen implicaciones pesadas, en este caso, considerando que declarar o muerto a un pasajero sería una afirmación de gran peso en caso de que se utilizara un modelo similar a éste en caso de un incidente similar al del Titanic. En adición a esto, su uso es recomendado en casos donde una categoría es considerablemente más frecuente que la otra, en este caso, el número de pasajeros fallecidos supera en número al de supervivientes en una relación 2 a 1, siendo que casi 2 tercios de los pasajeros fallecieron durante el hundimiento.

Se utiliza el conjunto de **Prueba** para tomar la decisión ya que consideramos que representa mejor la capacidad de generalización de datos de los modelos, esto evita situaciones en los que la varianza del modelo sea alta y el modelo resulte tener *overfitting* con datos observados previamente y no pueda adaptarse a datos del conjunto de testing de la competencia.

Planeación de pruebas

Con la finalidad de acercarnos a una versión más refinada de cada modelo, se generarán diferentes variaciones de los mismos variando sus hiper parámetros, buscando acercarnos a aquellos que proporcionan mejores resultados. En esta sección, puede encontrar los hiper parámetros que se emplean en cada prueba, y en la sección correspondiente a cada modelo puede encontrar los resultados de dichas pruebas.

Decision Tree

Prueba	max_depth	min_samples_split
Prueba 1	None	2
Prueba 2	5	5
Prueba 3	10	10
Prueba 4	15	10

Random Forest

Prueba	n_estimators
Prueba 1	5
Prueba 2	10
Prueba 3	50
Prueba 4	100

Redes neuronales

En el caso de redes neuronales, debido a la cantidad de hiper parámetros modificables, se optó por modificar solamente la cantidad de neuronas en la primera capa. Los demás

parámetros se mantienen constantes a lo largo de todas las pruebas. La función de activación de la primera y segunda capa es **relu** y de la tercera es la **función sigmoide**; la segunda capa cuenta con **5 neuronas** y la tercera con **10 neuronas**, además, todos los modelos se entrenaron usando **50 iteraciones**.

Prueba	Primer Capa
Prueba 1	1
Prueba 2	2
Prueba 3	3
Prueba 4	4
Prueba 5	10
Prueba 6	50

Support Vector Machine

Prueba	Tipo de Kernel
Prueba 1	Lineal
Prueba 2	Polinomial
Prueba 3	Sigmoidal
Prueba 4	rbf

Resultados

Decision Tree

Prueba	Exactitud	Precisión	Sensibilidad	F1
Prueba 1	77.65%	72.97%	72.97%	72.97%
Prueba 2	80.45%	83.05%	66.22%	73.68%
Prueba 3	82.12%	80.88%	74.32%	77.46%
Prueba 4	81.56%	78.08%	77.02%	77.55%

Random Forest

Prueba	Exactitud	Precisión	Sensibilidad	F1
Prueba 1	80.44%	76.71%	75.67%	76.19%
Prueba 2	79.89%	77.14%	72.97%	75.00%
Prueba 3	79.88%	77.14%	72.97%	75.00%

Prueba 4	79.33%	76.06%	72.98%	74.48%

Redes neuronales

Prueba	Accuracy	Precision	Recall	F1 Score
Prueba 1	75.98%	68.67%	77.03%	72.61%
Prueba 2	81.01%	74.39%	82.43%	78.21%
Prueba 3	78.77%	75.71%	71.62%	73.61%
Prueba 4	79.33%	78.46%	68.92%	73.38%
Prueba 5	79.33%	78.46%	68.92%	73.38%
Prueba 6	82.12%	82.81%	71.62%	76.81%

Support Vector Machine

Prueba	Exactitud	Precisión	Sensibilidad	F1
Prueba 1	78.21%	75.36%	70.27%	72.73%
Prueba 2	62.01%	80.00%	10.81%	19.05%
Prueba 3	62.01%	54.41%	50.00%	52.11%
Prueba 4	65.36%	75.00%	24.32%	36.73%

Comparativa

Conjunto de prueba

Modelo	Prueba	Exactitud	Precisión	Sensibilidad	F1
Decision Tree	Prueba 3	82.12%	80.88%	74.32%	77.46%
Random Forest	Prueba 1	80.44%	76.71%	75.67%	76.19%
Redes Neuronales	Prueba 2	81.01%	74.39%	82.43%	78.21%
SVM	Prueba 1	78.21%	75.36%	70.27%	72.72%

Conjunto de entrenamiento

Modelo	Prueba	Exactitud	Precisión	Sensibilidad	F1
Decision Tree	Prueba 3	82.12%	80.88%	74.32%	77.46%

Random Forest	Prueba 1	94.94%	96.03%	90.30%	93.08%
Redes Neuronales	Prueba 2	84.13%	84.44%	70.90%	77.08%
SVM	Prueba 1	78.79%	73.88%	67.54%	70.57%

Modelo preliminar

Teniendo en cuenta los diferentes modelos de Machine Learning puestos a prueba y habiendo seleccionado el que mejor resultados proporciona con respecto a la base de datos utilizada de los supervivientes del Titanic, ahora se busca mejorar el rendimiento, procesamiento, interpretación y resultados del modelo predilecto como el mejor entre los implementados, es decir, el modelo de **redes neuronales**. Retomando la metrica establecida de F1 podemos apreciar la siguiente tabla:

Pruebas	Desempeño (Puntaje F1)
Prueba 3 de Árbol de Decisión	0,7746
Prueba 1 de Bosques Aleatorios	0,7619
Prueba 2 de Redes Neuronales	0,7821
Prueba 1 de Support Vector Machine	0,7272

Si comparamos los cuatro puntajes obtenidos, podremos observar que las **redes neuronales** poseen el mejor desempeño, por este motivo, se toma la decisión de que resultaría oportuno desarrollar la capacidad de este modelo con el fin de encontrar la mejor versión de las redes neuronales y si es posible obtener mejores resultados buscando un desempeño óptimo mediante su refinamiento, ajuste de hiper parámetros y aplicación de una técnica de regularización.

Refinamiento del Modelo

A partir de la definición del modelo a utilizar para resolver el reto, optamos por elegir una técnica de refinamiento muy utilizada y bastante común en algoritmos de Machine Learning en general, y con ello conseguir la optimización de los hiper parámetros de la red neuronal; esta es conocida como *Grid Search*.

Grid Search es una técnica de optimización de modelos que consiste básicamente en una herramienta capaz de buscar y mejorar el rendimiento de modelos de Machine Learning por medio de una búsqueda exhaustiva en base a un conjunto de posibles valores que pueden adquirir los hiper parámetros. Es decir, en base a una lista de posibles valores por hiper parámetro, se busca probar con múltiples combinaciones que se puedan generar, entrenar el modelo con cada una de estas variaciones, comparar las métricas y resultados de todos y finalmente se queda con el más destacado en desempeño de entre todos. Respecto a

nuestro modelo de redes neuronales, hipe parámetros como las muestras de entrenamiento (*batch_size*), tasa de aprendizaje (*learning_rate*), cantidad de neuronas y de capas (*layers*), optimizadores (*optimizer*) y funciones de activación (*activation*) pueden ser modificados para conseguir el rendimiento y resultados deseados, logrando una regularización óptima y mejor ajuste a los datos, claro que se deben prevenir casos tanto de overfitting como de underfitting del modelo.

Aplicación de Grid Search

Para comenzar con la búsqueda de la mejor configuración de nuestro modelo, definimos una función que será llamada para crear los modelos de redes neuronales necesarios (dependiendo de la cantidad de combinaciones a probar), y dicha función no es diferente del modelo que definimos dentro de la selección de modelos, simplemente se adaptó su definición a una función que recibe los hiper parámetros como variables, conservando 4 capas (2 ocultas que variarán su cantidad de neuronas), alternando las funciones de activación por las 3 primeras capas y el optimizador de las redes neuronales como se muestra a continuación:

```
def create_model(optimizer='adam', activationL1='relu', activationL2='relu', activationL3='relu', neurons1=50, neurons2=5, neurons3=10):
    caracteristic_number = train_features_scaled.shape[1]  # Calculate the number of input features
    model = keras.Sequential([
        keras.layers.Dense(caracteristic_number, activation=activationL1, input_shape=(caracteristic_number,)),
        keras.layers.Dense(neurons2, activation=activationL2),
        keras.layers.Dense(neurons3, activation=activationL3),  # Adding another hidden layer
        keras.layers.Dense(1, activation='sigmoid')
    ])
    model.compile(optimizer=optimizer, loss='mean_squared_error', metrics=['accuracy'])
    return model
```

En cuanto a la distribución de los datos, se ha planteado que se trabaje con 3 subsets distintos, esto con la finalidad de comprobar la evolución del comportamiento del modelo en etapas distintas de la implementación. Esta suele ser una muy buena práctica debido a que nos permite evaluar la forma en cómo cambia el error cuando la complejidad del algoritmo va aumentando. Es por ello, que la separación de los datos se define de la siguiente manera:

Training Set

El set de entrenamiento está compuesto de las variables *train_features* y *y_train*. Estas dos son utilizadas dentro de la sección de entrenamiento, que sirve para que el modelo tenga la oportunidad de probar algunos valores iniciales, entienda un poco del contexto de la situación y en base a ello pueda empezar a generar algunas cuántas predicciones que, con cada iteración, reduzca el valor del error lo más posible y esto le permita posteriormente proporcionar resultados coherentes y predicciones certeras.

Validation Set

El set de validación este compuesto de las variables *val_features* y *val_targets*. Estas dos son utilizadas dentro de la sección de entrenamiento, que sirve para que el modelo pueda ser evaluado con una sección de los datos originales y medir qué tan bueno es su rendimiento.

Test Set

El set de pruebas está compuesto de las variables *test_features* y *test_targets*. Estas dos son utilizadas dentro de la sección de predicciones, la cual en base al entrenamiento y a la validación realizados previamente, el modelo empiece a recibir valores de entrada (*test_features*) sin recibir los valores de salida (*test_targets*), generando predicciones por sí mismo, claro que se espera que teniendo un buen entrenamiento y un MSE pequeño, el rango de error de las pruebas sea de igual forma pequeño y muy preciso en comparación con los datos reales (*test_targets*).

En cuanto a las configuraciones que se consideraron a probar por el algoritmo, se definió una variable *"param_grid",* la cual posee las siguientes combinaciones a probar:
Dicha variable considera 4 combinaciones posibles del optimizador, 2 tipos de funciones de

```
param_grid = {
    'optimizer': ['adam', 'sgd', 'adagrad', 'rmsprop'],
    'activationL1': ['relu', 'sigmoid'],
    'activationL2': ['relu', 'sigmoid'],
    'activationL3': ['relu', 'sigmoid'],
    'neurons2': [15, 10, 5],
    'neurons3': [2, 5, 10],
    'epochs': [30],
    'batch_size': [15, 20, 30, 60]
}
```

activación para las 3 primeras capas, 3 combinaciones distintas de cantidad de neuronas para la segunda y tercera capa, 30 iteraciones fijas y 4 opciones para las muestras de entrenamiento. Teniendo en cuenta todas las posibles combinaciones que se pueden generar en base a esta variable, se espera que el método de Grid Search logre encontrar una buena configuración de hiper parámetros que reflejen una mejora significativa en el rendimiento del modelo, tanto en disminución de errores como en el aumento de verdaderos positivos y negativos, y de métricas como accuracy, que durante la selección de los modelos el mejor resultado arrojado fue de 82%.

La idea principal de variar tantos parámetros es que se obtenga una configuración lo más cercana posible a un modelo refinado confiable, al que posteriormente se pueda aplicar

Finalmente, aplicamos la búsqueda exhaustiva para encontrar la mejor configuración de nuestro modelo. La razón por la cual no se utilizó librerías como *keras* o *SciKit Learn* para llevar a cabo *GridSearch* es debido a que ya se había implementado esta búsqueda con dichas librerías, sin embargo, a la fecha que se está desarrollando este reporte se presentaron algunas complicaciones al respecto, debido a que algunas funciones se volvieron obsoletas con la actualización de esas librerías. Buscando una alternativa, se buscó implementar *GridSearch* de forma convencional con un *for loop* que prueba cada una de las configuraciones en el modelo y elige el mejor de todos:

```python
# Inicializa las variables para hacer un seguimiento de los mejores hiperparámetros y la mejor puntuación
mejores_hiperparametros = None
mejor_puntuacion = float('inf')  # Inicializa con un valor alto para problemas de minimización
mse_values = []

# Itera sobre todas las combinaciones posibles de hiperparámetros
combinaciones_parametros = list(itertools.product(*param_grid.values()))

for params in combinaciones_parametros:
    optimizer, activationL1, activationL2, activationsL3, neurons2, neurons3, epochs, batch_size = params

    # Crea el modelo con los hiperparámetros actuales
    modelo = create_model(optimizer, activationL1, activationL2, activationsL3, neurons2, neurons3)

    # Entrena tu modelo (deberías tener tus datos de entrenamiento y validación)
    historial = modelo.fit(train_features_scaled, train_targets, epochs=epochs, batch_size=batch_size, validation_data=(val_features_scaled, val_targets), verbose=0)

    # Calculamos el MSE
    val_loss = historial.history['val_loss']
    mse_values.append(val_loss)

    # Evalúa tu modelo (deberías tener tu métrica de evaluación)
    perdida_val = modelo.evaluate(val_features_scaled, val_features)[0]

    # Actualiza los mejores hiperparámetros y la mejor puntuación si el modelo actual es mejor
    if perdida_val < mejor_puntuacion:
        best_score = perdida_val
        best_params = params

print("Parameter Grid Configuration:")
for key, value in param_grid.items():
    print(f"{key}: {value}")

best_mse_index = mse_values.index(min(mse_values))

# Imprime los mejores hiperparámetros y la mejor puntuación
print("Mejores Hiperparámetros:", best_params)
print("Mejor Pérdida en Validación:", best_score)
print("MSE durante el entrenamiento:", mse_values[best_mse_index])
```

Para terminar con el refinamiento, se le aplicó una técnica de regularización conocida como **L2**, la cual permite al modelo reducir algunas de las irregularidades que este pueda presentar, en específico, overfitting y alta varianza del modelo. Esta técnica añade un término de penalización a la función de pérdida que desalienta al modelo de asignar pesos excesivamente altos a cualquier característica en particular, lo que obliga al modelo a distribuir los pesos más equitativamente a través de todas las variables proporcionadas.

Para implementar la regularización **L2** se utilizó una función de la librería de *keras*:

```
kernel_regularizer=keras.regularizers.l2()
```

Dicha función se aplica en las capas a las cuales se desea implementar la regularización, en nuestro caso las colocamos dentro de las 2 capas ocultas del modelo:

```python
caracteristic_number = train_features_scaled.shape[1]  # Calculate the number of input features
model = keras.Sequential([
  keras.layers.Dense(caracteristic_number, activation='relu', input_shape=(caracteristic_number,)),
  keras.layers.Dense(15, activation='relu', kernel_regularizer=keras.regularizers.l2(0.5)),
  keras.layers.Dense(2, activation='sigmoid', kernel_regularizer=keras.regularizers.l2(0.4)),
  keras.layers.Dense(1, activation='relu')
])

# Compilamos la red neuronal
model.compile(optimizer='adagrad', loss='mean_squared_error', metrics=['accuracy'])

# Entrenamos el modelo con el set de pruebas
history = model.fit(train_features_scaled, train_targets, epochs=30, batch_size=60)
```

Resultados

En cuanto a los resultados arrojados por el modelo refinado y regularizado, se presentaron algunos cambios ligeros con respecto al original. En algunos casos, se obtuvieron resultados que otorgaban valores más bajos que el 82% de accuracy obtenido al inicio,

siendo estos 80% u 81%, en ocasiones llegaba a bajar a 79%, pero no llegó a bajar tanto de este rango. En otras palabras, el refinamiento que realizamos no arrojó resultados satisfactorios con los que ya teníamos desde un principio. No logramos sobrepasar esa barrera que teníamos del 82%, y en realidad como equipo pensamos que de tener un poco más de tiempo para dedicarle al refinamiento, podríamos haber encontrar un mejor ajuste tanto de hiper parámetros como de regularización, con el fin de encontrar la mejor versión del modelo de redes neuronales. A continuación, se muestra una tabla en la que se proyectan algunos de los resultados obtenidos por el *GridSearch*:

Configuración Original de la Red Neuronal	**Hyperparameter GridSearch Results 1**	**Hyperparameter GridSearch Results 2**	**Hyperparameter GridSearch Results 3**
Best Score: 0.8212290502793296	Best Score: 0.8160243000153647	Best Score: 0.8188313300003546	Best Score: 0.8230448297462445
Best Parameters:	**Best Parameters:**	**Best Parameters:**	**Best Parameters:**
- 'activation': 'relu'	- 'activation': 'relu'	- 'activation': 'relu'	- 'activation': 'relu'
- 'batch_size': 32	- 'batch_size': 64	- 'batch_size': 32	- 'batch_size': 16
- 'epochs': 50	- 'epochs': 50	- 'epochs': 30	- 'epochs': 20
- 'neurons1': 50	- 'neurons1': 50	- 'neurons1': 10	- 'neurons_per_layer': 20
- 'neurons2': 15	- 'neurons2': 15	- 'neurons2': 15	- 'num_layers': 2
- 'neurons3': 75	- 'neurons3': 75	- 'neurons3': 6	- 'optimizer': 'adam'
- 'optimizer': 'adam'	- 'optimizer': 'adam'	- 'optimizer': 'adam'	- 'activation': 'relu'
Parameter Grid:	**Parameter Grid:**	**Parameter Grid:**	**Parameter Grid:**
- optimizer: ['adam']	- optimizer: ['adam', 'sgd']	- optimizer: ['adam', 'sgd']	- optimizer: ['adam', 'sgd']
- activation: ['relu', 'sigmoid']	- activation: ['relu', 'sigmoid']	- activation: ['relu', 'sigmoid', 'softmax', 'tanh']	- activation: ['relu', 'sigmoid', 'softmax', 'tanh']
- neurons1: [50]	- neurons1: [50, 100]	- neurons1: [5, 10]	- num_layers: [1, 2, 3]
- neurons2: [5]	- neurons2: [30, 15]	- neurons2: [10, 15]	- neurons_per_layer: [5, 10, 15, 20]
- neurons3: [10]	- neurons3: [75, 40]	- neurons3: [6, 12]	- epochs: [5, 10, 20, 30]

- epochs: [50]	- epochs: [50, 100]	- epochs: [5, 10, 20, 30]	- batch_size: [16, 32, 64]
- batch_size: [32]	- batch_size: [32, 64]	- batch_size: [16, 32, 64]	
Test Accuracy: 81.01%	Test Accuracy: 79.89%	Test Accuracy: 79.89%	Test Accuracy: 79.89%

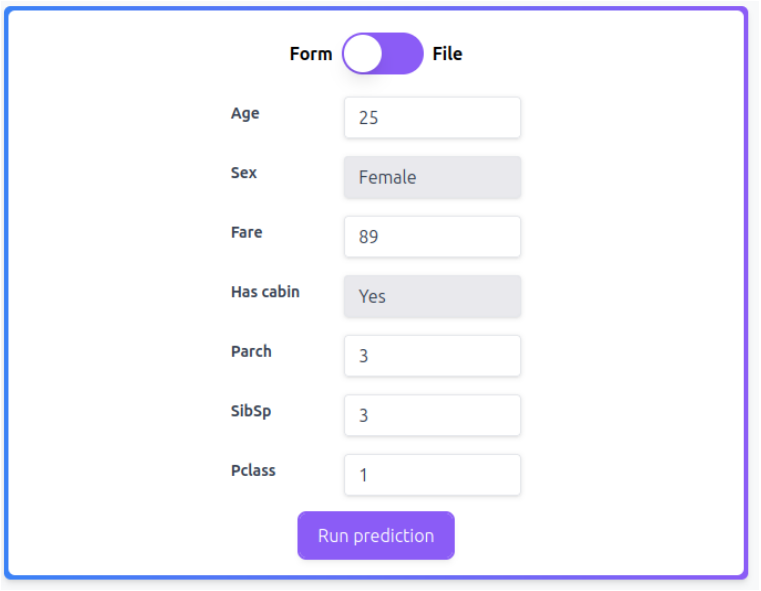
Primero se proyectan los resultados obtenidos con el modelo original, después se muestran las distintas pruebas que se realizaron variando las combinaciones de los hiper parámetros. Cada una de las columnas de los modelos posee sus respectivos mejores hiper parámetros, las combinaciones de valores contenidos en *param_grid* y sus resultados.

En un principio se esperaba que con este refinamiento y regularización se encontrara dicha configuración, sin embargo, no nos fue posible mejorar más el modelo, más bien lo mantuvimos con un rendimiento constante con algunas variaciones un poco más bajas. A pesar de esto, creemos firmemente que esta configuración existe, y se puede maximizar el rendimiento de las redes neuronales para este caso, inclusive para llegar al menos a un 85% de *accuracy* o tal vez si somos optimistas acercarnos al 90%. Al final de todo, se trata del uso de un algoritmo básico de Machine Learning que busca generar una clasificación binaria no tan compleja como podrían serlo otros algoritmos que necesitan de una cantidad exorbitante de datos para realizar predicciones certeras según su contexto. Lo importante ante esta situación es tener en cuenta todo el procedimiento que llevamos a cabo y aprender cómo es que se lleva a cabo la construcción de un modelo de aprendizaje máquina desde la limpieza de datos, investigación, identificación de los modelos, hasta la implementación, selección, refinamiento y regularización.

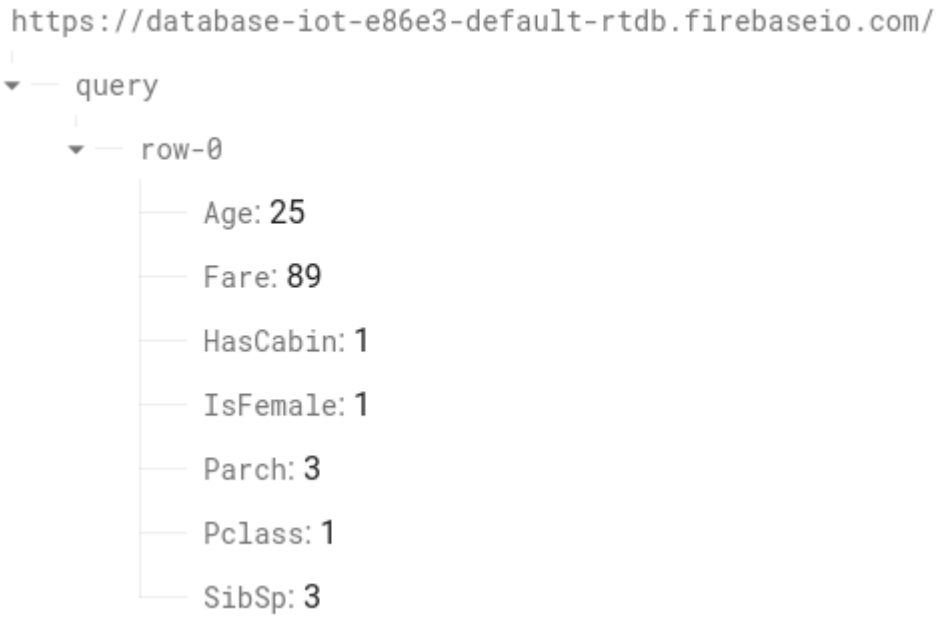
Interfaz

Interfaz web

Respecto a nuestra solución de hardware implementamos una interfaz web usando vue.js y firebase para nuestra base de datos. El primer paso en nuestro proyecto fue implementar la interfaz web y base de datos así como conectar ambos. Desde nuestra interfaz web podemos subir los datos de un pasajero individualmente o podemos subir todo un .csv de pasajeros para generar múltiples predicciones.



Una vez enviada la información desde la interfaz web nuestra base de datos los almacena y genera una liga para dichos datos, esta liga es clave para todo el proceso ya que nuestro microcontrolador depende del internet para poder conectarse a la base de datos para extraer información así como enviar datos de respuesta.



Conección con base de datos

Después de implementar nuestra interfaz web y base de datos tuvimos que buscar una manera de conectar nuestro esp-32 al internet. Para esto utilizamos las librerías de Wifi.h y

para generar la conexión a Firebase utilizamos un cliente creado con la librería de HTTPClient.h. El primer paso en este proceso fue definir la configuración de nuestro wifi y las ligas donde nuestro chip puede recibir y enviar información de la base de datos de firebase. Después de establecer estos parámetros de configuración inicializamos nuestra conexión al internet usando el nombre del wifi y la contraseña y un cliente utilizando la liga de donde nuestro chip recibe información.

```
const char* ssid = "Tec";
const char* password = "*******";

// Define the Firebase URLs
const char* queryUrl = "https://database-iot-e86e3-default-rtdb.firebaseio.com/query/.json";
const char* resultsTestUrl = "https://database-iot-e86e3-default-rtdb.firebaseio.com/results/.json";
```

Después de esto establecimos un cliente nuevo utilizando la liga de resultados para poder publicar nuestros resultados de regreso a la base de datos.

Entrenamiento de modelo

Debido a conflictos de compatibilidad usando micro python decidimos emplear el arduino IDE como nuestro ambiente de programación, debido a esto tuvimos que encontrar la manera de convertir nuestro algoritmo de python a c++. Para esto empleamos una librería llamada tinymlgen. Reducimos el código original de nuestra red neuronal para que solamente utilice la librería de tensor flow, luego le enviamos el modelo a una función de tinymlgen llamada port ésta la convierte a hexadecimal para que se pudiera leer desde un archivo de c++.

```
from tinymlgen import port
# Convertir modelo en un array C
c_code = port(model, pretty_print=True)
print(c_code)
```

```
const unsigned char model_data[] DATA_ALIGN_ATTRIBUTE = {
        0x1c, 0x00, 0x00, 0x00, 0x54, 0x46, 0x4c, 0x33, 0x14, 0x00, 0x20, 0x00,
        0x1c, 0x00, 0x18, 0x00, 0x14, 0x00, 0x10, 0x00, 0x0c, 0x00, 0x00, 0x00,
        0x08, 0x00, 0x04, 0x00, 0x14, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00,
        0x88, 0x00, 0x00, 0x00, 0xe0, 0x00, 0x00, 0x00, 0x94, 0x07, 0x00, 0x00,
        0xa4, 0x07, 0x00, 0x00, 0xd0, 0x0e, 0x00, 0x00, 0x03, 0x00, 0x00, 0x00,
        0x01, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00, 0x1a, 0xf8, 0xff, 0xff,
        0x0c, 0x00, 0x00, 0x00, 0x1c, 0x00, 0x00, 0x00, 0x38, 0x00, 0x00, 0x00,
        0x0f, 0x00, 0x00, 0x00, 0x73, 0x65, 0x72, 0x76, 0x69, 0x6e, 0x67, 0x5f,
        0x64, 0x65, 0x66, 0x61, 0x75, 0x6c, 0x74, 0x00, 0x01, 0x00, 0x00, 0x00,
        0x04, 0x00, 0x00, 0x00, 0x94, 0xff, 0xff, 0xff, 0x0d, 0x00, 0x00, 0x00,
        0x04, 0x00, 0x00, 0x00, 0x07, 0x00, 0x00, 0x00, 0x64, 0x65, 0x6e, 0x73,
        0x65, 0x5f, 0x37, 0x00, 0x01, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00,
        0xee, 0xf8, 0xff, 0xff, 0x04, 0x00, 0x00, 0x00, 0x0d, 0x00, 0x00, 0x00,
        0x64, 0x65, 0x6e, 0x73, 0x65, 0x5f, 0x34, 0x5f, 0x69, 0x6e, 0x70, 0x75,
        0x74, 0x00, 0x00, 0x00, 0x02, 0x00, 0x00, 0x00, 0x34, 0x00, 0x00, 0x00,
        0x04, 0x00, 0x00, 0x00, 0xdc, 0xff, 0xff, 0xff, 0x10, 0x00, 0x00, 0x00,
        0x04, 0x00, 0x00, 0x00, 0x13, 0x00, 0x00, 0x00, 0x43, 0x4f, 0x4e, 0x56,
        0x45, 0x52, 0x53, 0x49, 0x4f, 0x4e, 0x5f, 0x4d, 0x45, 0x54, 0x41, 0x44,
        0x41, 0x54, 0x41, 0x00, 0x08, 0x00, 0x0c, 0x00, 0x08, 0x00, 0x04, 0x00,
        0x08, 0x00, 0x00, 0x00, 0x0f, 0x00, 0x00, 0x00, 0x04, 0x00, 0x00, 0x00,
        0x13, 0x00, 0x00, 0x00, 0x6d, 0x69, 0x6e, 0x5f, 0x72, 0x75, 0x6e, 0x74,
        0x69, 0x6d, 0x65, 0x5f, 0x76, 0x65, 0x72, 0x73, 0x69, 0x6f, 0x6e, 0x00,
        0x11, 0x00, 0x00, 0x00, 0xb0, 0x06, 0x00, 0x00, 0xa8, 0x06, 0x00, 0x00,
        0x94, 0x06, 0x00, 0x00, 0x54, 0x06, 0x00, 0x00, 0x08, 0x06, 0x00, 0x00,
        0xdc, 0x05, 0x00, 0x00, 0x08, 0x05, 0x00, 0x00, 0x54, 0x03, 0x00, 0x00,
        0xec, 0x00, 0x00, 0x00, 0xb4, 0x00, 0x00, 0x00, 0xac, 0x00, 0x00, 0x00,
        0xa4, 0x00, 0x00, 0x00, 0x9c, 0x00, 0x00, 0x00, 0x94, 0x00, 0x00, 0x00,
```

Importación del modelo a microcontrolador ESP32

Después de esto solo quedaba implementar nuestro algoritmo. La manera en la que funciona es que recibe los datos de firebase en un formato JSON y los separa y almacena en variables diferentes. Estos variables son enviados a nuestro modelo. Para nuestro modelo en c++ empleamos una librería de arduino llamada EloquentTinyML.
Después de crear el modelo y recibir los valores de las variables se los enviamos a nuestro modelo y generamos una predicción.
Una vez generada la predicción se lo enviamos de regreso a nuestra base de datos para poder luego ser desplegada de regreso en una tabla en nuestra interfaz web.

```
DynamicJsonDocument doc(1024); // Adjust the buffer size as needed
deserializeJson(doc, payload);

  JsonObject row0 = doc["row-0"];

  int Pclass = row0["Pclass"];
  Serial.println("Pclass");
  Serial.println(Pclass);
  int Age = row0["Age"];
  Serial.println("Age");
  Serial.println(Age);
  int SibSp = row0["SibSp"];
  Serial.println("SibSp");
  Serial.println(SibSp);
  int Parch = row0["Parch"];
  Serial.println("Parch");
  Serial.println(Parch);
  int Fare = row0["Fare"];
  Serial.println("Fare");
  Serial.println(Fare);
  int HasCabin = row0["HasCabin"];
  Serial.println("HasCabin");
  Serial.println(HasCabin);
  int IsFemale = row0["IsFemale"];
  Serial.println("IsFemale");
  Serial.println(IsFemale);
```

Conclusiones

Este se trató de un importante y fructífero ejercicio didáctico que nos permitió aprender sobre análitica y limpieza de datos, diferentes modelos de machine learning, evaluación del desempeño de los mismos y metodologías de optimización de parámetros. Para el final, se logró entrenar un modelo que arroja resultados satisfactorios sobre el conjunto de datos original, y aunque no se logró mejorar su desempeño durante la última etapa en el modo o escala que nos hubiera gustado en un principio, consideramos que el valor educativo que brinda conocer dichas metodologías y enfrentar los desafíos que surgieron en el proceso ha sido invaluable. Este proyecto nos ha proporcionado una comprensión más profunda de las complejidades y sutilezas del machine learning, así como la importancia de la preparación y selección adecuada de datos. Logramos aplicar metodologías para limpieza de bases de datos, realizar análisis profundos a los modelos implementados, trabajar con sistemas de desarrollo web y componentes de hardware con los cuales integramos nuestro modelo y entender cómo se realiza el refinamiento de los mismos.

Lo más destacable que podemos mencionar sobre este proyecto es la forma en la cual implementamos, no solo el modelo final seleccionado referente a las redes neuronales, sino también a la implementación de modelos como Decision Trees, Random Forests o Support Vector Machines, siendo que no nos conformamos con simplemente elegir el primer modelo que otorgara los mejores resultados en primeras instancias, al contrario decidimos seguir intentando con varias de las posibilidades que teníamos al alcance para probar una clasificación binaria con diversas metodologías y lógicas muy distintas, por que en realidad al momento de elaborar las entregas correspondientes a la selección del mejor modelo no teníamos mucha experiencia aplicándolos, claro que teníamos alguna noción de cómo trabajan y la lógica que siguen estos modelos para generar predicciones, pero faltaban algunos conceptos por revisar y profundizar tanto en clase como por nuestra cuenta, lo cual supuso un reto bastante significativo. Aunado a esto, hemos aprendido muchas técnicas de refinado y regularización de modelos, qué es lo que se necesita para empezar a refinarlos, que puede demorar demasiado tiempo encontrar la mejor configuración, o bien que no siempre lograrás obtener los resultados que esperas. Inclusive podrías entorpecer el algoritmo sin darte cuenta, ya sea porque no se filtran los datos de forma correcta, tienes campos vacíos en el dataset, datos erróneos, no se secciona en los subconjuntos necesarios, o por el lado del modelo, la configuración no es la correcta, no tiene ningún tipo de regularización, no se ha probado el modelo con más de un set o distintos conjuntos de datos, y por consecuente el modelo presenta overfitting, underfitting, los errores no se reducen, entre otras muchos indicadores que hemos aprendido a considerar para mejorar nuestras habilidades al momento de definir, comprender, analizar e implementar modelos de aprendizaje máquina.