Práctica 1:

Creando la primera red neuronal para clasificación y para regresión

Esta primera práctica tiene como objetivo fundamental el diseño, desarrollo y ejecución de dos modelos neuronales sencillos para resolver dos tipos de tareas: clasificación y regresión. Para el desarrollo se va a hacer uso de la API Keras a alto nivel de Tensorflow.

Aprendizaje técnico:

- Utilizar la librería de algebra lineal Numpy
- Utilizar la librería de tratamiento y análisis de datos Pandas
- Utilizar la librería de Machine Learning Scikit-Learn
- Utilizar la API Keras de Tensorflow

Consideraciones generales en el uso de Notebooks

- Asegúrate de que todo el código necesario para ejecutarlo completamente está disponible.
- Asegúrate de usar un orden lógico de ejecución de celdas para permitir una ejecución completa y automática. El Notebook debe poder funcionar perfectamente simplemente lanzando la opción "Restart & Run All" de la sección "Kernel". Cuidado con instanciar variables o métodos en celdas posteriores a las cuales están utilizadas, una ejecución independiente y manual puede hacer que funcione, pero la ejecución automática fallará.
- Asegúrate de utilizar las celdas de tipo "Markdown" para texto y las de "Code" para código. Combinar distintos tipos de celda ayuda a la explicabilidad de la resolución y a la limpieza en general.
- Puedes crear celdas adicionales si lo necesitas.
- A la hora de la entrega del Notebook definitivo, asegúrate que lo entregas ejecutado y con las salidas de las celdas guardadas. Esto será necesario para la generación del PDF de entrega.
- El código cuanto más comentado mejor. Además, cualquier reflexión adicional siempre será bienvenida.

```
import pandas as pd
import tensorflow as tf
import keras
from keras.models import Sequential
from keras.layers import Dense
from sklearn.preprocessing import StandardScaler
from sklearn.preprocessing import MinMaxScaler
from sklearn.model_selection import train_test_split
```

1. Red Neuronal para problema de clasificación (5 p.)

En este primer apartado el objetivo es crear una red neuronal que funcione como un clasificador. Será importante tener en cuenta todas las consideraciones que supone este objetivo a lo largo del diseño y desarrollo de la red.

1.1. Obtención y análisis previo de los datos (1 p.)

Para este ejercicio, vamos a hacer uso de un dataset real extraído de la plataforma Kaggle. Kaggle es una de las plataformas más conocidas en el mundo del Data Science y del Machine Learning, ya que en ella se publican multitud de datasets para los cuales se organizan competiciones (muchas de ellas organizadas por empresas con remuneración a los ganadores) o se liberan como código libre para practicar o generar modelos propios. El problema a resolver en este ejercicio será la clasificación de móviles en 4 categorías de precio atendiendo a sus características, para lo que utilizaremos el dataset que tenéis disponible aquí

Para enfocar cualquier solución basada en ML es fundamental conocer el problema y analizar detenidamente los datos con los que se cuentan. Por eso, es importante en este punto inicial acudir a la descripción del dataset (link aqui) y leer detenidamente la información con la que contamos: planteamiento del problema a resolver, descripción del dataset, descripción de cada uno de los atributos y tipo de dato de cada uno de ellos. Además, Kaggle genera ciertos análisis fundamentales que permiten ver de forma rápida los distintos valores y distribución de los mismos para cada columna, lo que permite adquirir una visión general del dataset que debería ser completada con un buen proceso de EDA (Exploratory Data Analysis) pero que no es el objetivo del módulo, por lo que vamos a quedarnos con lo fundamental.

Pregunta 1 (0.5 p):

Sin programar ni una línea de código ni abrir el dataset aún, simplemente aprovechando la información que proporciona Kaggle del dataset. ¿Qué porcentaje de móviles del dataset tienen soporte para 3G? ¿Cuántos registros supone eso?

Respuesta aquí

- En el dataset tenemos la columna "four_g" con valores de 0 de 513 registrios y 1 de 487 registros, siendo 0 que no tiene soporte 4 g por consiguiente es 3 g y esto corresponde al 51.3 %

▾ Pregunta 2 (0.5 p):

Sin programar ni una línea de código ni abrir el dataset aún, simplemente aprovechando la información que proporciona Kaggle del dataset. ¿Cuál es el valor de anchura de pantalla del móvil con la pantalla más ancha? ¿Cuál es la media en este atributo?

Respuesta aquí

- No hay información que hable de la anchura de la pantalla

Estas preguntas, aunque aparentemente poco útiles para la creación del modelo, son fundamentales para empezar a familiarizarse con entornos gráficos de análisis de datos como Kaggle y a acostumbrarse a dedicar una buena parte del tiempo a comprender los datos con los que se va a trabajar. Toda esta información (y muchas más) debería de ser recopilada en la fase de EDA, que no se va a acometer en esta práctica, y utilizada para el óptimo diseño de la red neuronal.

▾ 1.2. Adaptación de los datos (1 p.)

Tras haber analizado detenidamente los datos, el siguiente paso será empezar a "jugar" con ellos. Para eso, habrá que proceder a descargar el fichero *.csv* disponible en los archivos de la práctica y guardarlo en una ubicación accesible. Es importante conocer la ruta del fichero para poder leerlo ahora desde el Notebook.

```
# Este PATH (ruta) es válido si el Notebook y el dataset están en la misma carpeta. En caso d
# descargado en otro sitio y no lo puedas mover, simplemente pon el PATH absoluto del fichero
# como por ejemplo: C:\Users\pablo\Documents\data_ej1.csv
PATH_DEL_DATASET = 'train.csv'
data = pd.read_csv(filepath_or_buffer = PATH_DEL_DATASET)
```

```
data.head()
```

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_wt
0	842	0	2.2	0	1	0	7	0.6	188
1	1021	1	0.5	1	0	1	53	0.7	136
2	563	1	0.5	1	2	1	41	0.9	145
3	615	1	2.5	0	0	0	10	0.8	131
4	1821	1	1.2	0	13	1	44	0.6	141

5 rows × 21 columns

Ahora, en la variable *data*, tenemos cargado el dataset en memoria y lo podemos empezar a utilizar. Concretamente, esta variable *data* es de tipo DataFrame, un tipo de objeto que se define en la librería Pandas. Iremos viendo a lo largo de la práctica las ventajas que supone trabajar con este tipo de datos en vez de matrices numéricas directamente.

```
data
```

	battery_power	blue	clock_speed	dual_sim	fc	four_g	int_memory	m_dep	mobile_
0	842	0	2.2	0	1	0	7	0.6	1
1	1021	1	0.5	1	0	1	53	0.7	1
2	563	1	0.5	1	2	1	41	0.9	1
3	615	1	2.5	0	0	0	10	0.8	1
4	1821	1	1.2	0	13	1	44	0.6	1
...	...	...	...	...	...	...	...	...	
1995	794	1	0.5	1	0	1	2	0.8	1
1996	1965	1	2.6	1	0	0	39	0.2	1
1997	1911	0	0.9	1	1	1	36	0.7	1
1998	1512	0	0.9	0	4	1	46	0.1	1
1999	510	1	2.0	1	5	1	45	0.9	1

2000 rows × 21 columns

```
data.columns
```

```
Index(['battery_power', 'blue', 'clock_speed', 'dual_sim', 'fc', 'four_g',
       'int_memory', 'm_dep', 'mobile_wt', 'n_cores', 'pc', 'px_height',
```

```
       'px_width', 'ram', 'sc_h', 'sc_w', 'talk_time', 'three_g',
       'touch_screen', 'wifi', 'price_range'],
      dtype='object')
```

Como se puede observar, el DataFrame *data* contiene tanto los atributos de clasificación como la etiqueta objetivo, es decir, deberemos separar entre X e y para poder trabajar. Esta columna que funciona como "target" es la llamada como "price_range", y lo sabemos por la descripción del problema. Por tanto, el siguiente paso será separar el DataFrame *data* en dos distintos: X e y

```
y = data[['price_range']]
X = data.drop(columns = ["price_range"])
```

▼ Pregunta 3 (0.5 p):

Tras haber separado X e y podemos ir empezando a sacar conclusiones de cara al diseño de la red neuronal. ¿Con cuántos registros contamos en el dataset? ¿Cuántos atributos usaremos para clasificar? ¿Cuántas clases distintas hay?

Es fundamental que se vea el código utilizado para responder cada una de estas preguntas

```
# Código de la respuesta aquí
num_registros = X.shape[0]
print(num_registros)
```

```
2000
```

```
num_atributos = X.shape[1]
print(num_atributos)
```

```
20
```

```
num_clases = y['price_range'].nunique()
print(num_clases)
```

```
4
```

Respuesta aquí

- contamos en el dataset 2000 registros, tenemos 20 a atributos y 4 clases

Si analizamos un poco más detenidamente el DataFrame X que usaremos como entrada de la red neuronal, veremos que se mezclan atributos de distinto tipo. Como recordarás, los atributos pueden ser de tipo categórico, nominal o binario (que realmente es una mezcla de las primeras

dos). Cada uno de estos tipos de datos deben de ser manipulados de una forma concreta para poder ser utilizados y para sacar el máximo provecho de su información. Estas manipulaciones deben de estar basadas en la información recogida en la EDA que tendríamos que haber llevado a cabo, pero para simplificar el proceso de limpieza y preparación de los datos, vamos a llevar a cabo las transformaciones más básicas posibles. Pero antes, el primer paso será identificar de qué tipo es cada uno de los atributos del DataFrame X:

```
COLUMNAS_BINARIAS = ["blue", "dual_sim", "four_g", "three_g", "touch_screen", "wifi"]
COLUMNAS_NUMERICAS = ["battery_power", "clock_speed", "fc", "int_memory", "m_dep", "mobile_wt
```

Una vez identificados los dos tipos de atributos con los que contamos, comenzaremos haciendo las transformaciones necesarias a los de tipo numérico. Como recordarás de módulos anteriores, los atributos de tipo numérico tienen la peculiaridad de que cada uno puede utilizar una escala de medida diferente y estar agrupado más o menos en un rango de valores. Esto, lo que puede provocar, es que aquellos atributos con valores más grandes tengan mayor influencia en las decisiones de la red y que, aquellos con valores más bajos (pero quizás más representativos para tomar una decisión) se vean eclipsados. Es por ello que hay que llevar a cabo un proceso de normalización. Para ello lo primero será separar todos estos atributos en un DataFrame auxiliar.

```
X_numericas = X[COLUMNAS_NUMERICAS]
```

Hay distintas estrategias de normalización, cada una de ellas más apropiada según la naturaleza y rango de valores de cada atributo, que habría que analizar en el proceso de EDA. Para simplificar, en esta práctica aplicaremos una normalización estándar a todos los atributos. Para ello, utilizaremos el objeto StandardScaler de scikit-learn, que aplicará la normalización por columnas. El objeto StandardScaler acepta como entrada y devuelve una matriz Numpy, es por ello que es necesario transformar nuestro DataFrame *X_numericas* a una matriz Numpy con el método **to_numpy()** y después convertir la salida otra vez en un DataFrame con el propio constructor del objeto.

```
sc = StandardScaler()
matriz_normalizada = sc.fit_transform(X_numericas.to_numpy())
X_normalizada = pd.DataFrame(matriz_normalizada, columns = COLUMNAS_NUMERICAS)
```

Ya tenemos en el DataFrame *X_normalizada* todos los atributos nominales normalizados. El siguiente paso será reconstruir el DataFrame original concatenando los atributos nominales normalizados y los atributos binarios, para los cuales no es necesario llevar a cabo ninguna transformación. Para ello, utilizaremos el método **concat()** de Pandas.

```
X_limpia = pd.concat([X[COLUMNAS_BINARIAS],X_normalizada], axis = 1)
```

Por tanto, ya tenemos en X e y los datos necesarios y limpios para poder crear la red neuronal

```
X_limpia.head()
```

	blue	dual_sim	four_g	three_g	touch_screen	wifi	battery_power	clock_speed	
0	0	0	0	0	0	1	-0.902597	0.830779	-0.76
1	1	1	1	1	1	0	-0.495139	-1.253064	-0.99
2	1	1	1	1	1	0	-1.537686	-1.253064	-0.53
3	1	0	0	1	0	0	-1.419319	1.198517	-0.99
4	1	0	1	1	1	0	1.325906	-0.395011	2.00

```
y.head()
```

	price_range
0	1
1	2
2	2
3	2
4	1

- Pregunta 4 (0.5 p):

  Una vez que tenemos los conjuntos X e y listos, el siguiente paso será crear los subconjuntos de entrenamiento, validación y test necesarios en cualquier problema de Machine Learning. Los tamaños de cada uno de definen a continuación:

  - El subconjunto de **entrenamiento** tiene que tener un 80% de los registros totales. Almacenar en variables de nombre X_train, y_train
  - El subconjunto de **test** tiene que tener el 20% de los registros totales. Almacenar en variables de nombre X_test, y_test
  - El subconjunto de **validación** tiene que tener un 20% de los registros del subconjunto de **entrenamiento**. Almacenar en variables de nombre X_val, y_val

*Nota: Hay muchas formas de separar conjuntos, pero quizás la mejor y más sencilla sea utilizar el método **train_test_split()** de scikit-learn*

```
# Respuesta aquí
from sklearn.model_selection import train_test_split

# Dividir en subconjunto de entrenamiento y prueba
X_train, X_test, y_train, y_test = train_test_split(X_limpia, y, test_size=0.2, random_state=

# Dividir el subconjunto de entrenamiento en subconjunto de entrenamiento y validación
X_train, X_val, y_train, y_val = train_test_split(X_train, y_train, test_size=0.2, random_sta
```

Ya tenemos los subconjuntos creados:

```
print("X TRAIN: ", X_train.shape)
print("X TEST shape: ", X_test.shape)
print("X VAL shape: ", X_val.shape)
print("y TRAIN shape: ", y_train.shape)
print("y TEST shape: ", y_test.shape)
print("y VAL shape: ", y_val.shape)
```

```
X TRAIN:  (1280, 20)
X TEST shape:  (400, 20)
X VAL shape:  (320, 20)
y TRAIN shape:  (1280, 1)
y TEST shape:  (400, 1)
y VAL shape:  (320, 1)
```

1.3. Creación, entrenamiento y evaluación de la Red Neuronal (3 p.)

Llegamos a la parte final: el momento de diseñar, entrenar y evaluar el modelo. Para ello, como ya hemos dicho anteriormente, vamos a hacer uso de Keras. Hay multitud de documentación acerca de esta librería, además de mucha comunidad que lo utiliza asiduamente, por lo que es fácil ver ejemplos similares sobre los que trabajar.

Pregunta 5 (1 p): Diseño, creación y compilación del modelo

En esta primera práctica, el diseño de la Red Neuronal va a ser fijo. En la siguiente práctica tendréis más libertad para jugar con los hiperparámetros de la red. Por tanto, crea una red con las siguientes características:

- Tres capas ocultas con tamaños 64, 32 y 16, respectivamente, que utilizen como función de activación **ReLU**
- La capa de salida tiene que usar una función de activación **Softmax**

El modelo tiene que ser compilado con las siguientes propiedades:

- El optimizador a utilizar debe de ser **Stochastic Gradient Descent**
- Las métricas a mostrar serán las de **loss** y **accuracy**
- La función de pérdida debe de ser la más acorde al problema de clasificación y a la codificación de la salida

```
# Código aquí
#librerias
from tensorflow.keras import models
from tensorflow.keras import layers
from tensorflow.keras import optimizers

# Crear modelo
model = models.Sequential()
model.add(layers.Dense(64, activation='relu', input_shape=(X_train.shape[1],)))
model.add(layers.Dense(32, activation='relu'))
model.add(layers.Dense(16, activation='relu'))
model.add(layers.Dense(4, activation='softmax'))

# Compilar modelo
model.compile(optimizer=optimizers.SGD(lr=0.001),
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

```
WARNING:absl:`lr` is deprecated in Keras optimizer, please use `learning_rate` or use th
```

Pregunta 7 (1 p): Entrenamiento del modelo

El modelo tiene que ser entrenado con las siguientes propiedades:

- Debe de ser entrenado durante **20** épocas
- Se debe utilizar un tamaño de batch de **32**
- Se debe utilizar el subconjunto de validación definido anteriormente para extraer métricas

```
# Código aquí
history = model.fit(X_train, y_train, epochs=20, batch_size=32, validation_data=(X_val, y_val
```

```
Epoch 1/20
40/40 [==============================] - 1s 9ms/step - loss: 1.4558 - accuracy: 0.3148 -
Epoch 2/20
40/40 [==============================] - 0s 3ms/step - loss: 1.3558 - accuracy: 0.3898 -
Epoch 3/20
40/40 [==============================] - 0s 3ms/step - loss: 1.3051 - accuracy: 0.4461 -
```

```
Epoch 4/20
40/40 [==============================] - 0s 3ms/step - loss: 1.2576 - accuracy: 0.4906 -
Epoch 5/20
40/40 [==============================] - 0s 3ms/step - loss: 1.2034 - accuracy: 0.5133 -
Epoch 6/20
40/40 [==============================] - 0s 3ms/step - loss: 1.1410 - accuracy: 0.5242 -
Epoch 7/20
40/40 [==============================] - 0s 4ms/step - loss: 1.0739 - accuracy: 0.5367 -
Epoch 8/20
40/40 [==============================] - 0s 3ms/step - loss: 1.0069 - accuracy: 0.5383 -
Epoch 9/20
40/40 [==============================] - 0s 3ms/step - loss: 0.9442 - accuracy: 0.5508 -
Epoch 10/20
40/40 [==============================] - 0s 3ms/step - loss: 0.8875 - accuracy: 0.5711 -
Epoch 11/20
40/40 [==============================] - 0s 3ms/step - loss: 0.8360 - accuracy: 0.5938 -
Epoch 12/20
40/40 [==============================] - 0s 3ms/step - loss: 0.7889 - accuracy: 0.6227 -
Epoch 13/20
40/40 [==============================] - 0s 3ms/step - loss: 0.7446 - accuracy: 0.6695 -
Epoch 14/20
40/40 [==============================] - 0s 3ms/step - loss: 0.7022 - accuracy: 0.7227 -
Epoch 15/20
40/40 [==============================] - 0s 3ms/step - loss: 0.6614 - accuracy: 0.7508 -
Epoch 16/20
40/40 [==============================] - 0s 3ms/step - loss: 0.6213 - accuracy: 0.8000 -
Epoch 17/20
40/40 [==============================] - 0s 3ms/step - loss: 0.5820 - accuracy: 0.8336 -
Epoch 18/20
40/40 [==============================] - 0s 3ms/step - loss: 0.5434 - accuracy: 0.8484 -
Epoch 19/20
40/40 [==============================] - 0s 3ms/step - loss: 0.5059 - accuracy: 0.8766 -
Epoch 20/20
40/40 [==============================] - 0s 3ms/step - loss: 0.4712 - accuracy: 0.8930 -
```

¿Qué métrica es más importante a tener en cuenta en este problema? ¿Por qué? Atendiendo a esta métrica, ¿qué valor ha obtenido el modelo en el subconjunto de entrenamiento? ¿Y en el de validación?

```
train_acc = history.history['accuracy'][-1]
val_acc = history.history['val_accuracy'][-1]
print('Precisión en subconjunto de entrenamiento:', train_acc)
print('Precisión en subconjunto de validación:', val_acc)
```

```
Precisión en subconjunto de entrenamiento: 0.8929687738418579
Precisión en subconjunto de validación: 0.8968750238418579
```

Respuesta aquí:

- la métrica más importante a tener en cuenta es la precisión (accuracy), ya que indica la proporción de muestras clasificadas correctamente. En este caso, interesa que el modelo sea capaz de clasificar correctamente el mayor número posible de móviles en su respectiva categoría de precio.

▼ Pregunta 8 (1 p): Evaluación del modelo

Una vez creado, compilado y entrenado el modelo, es hora de evaluarlo utilizando el conjunto de test. ¿Cuáles son los valores de loss y accuracy en el conjunto de test?

```
# Código aquí
test_loss, test_acc = model.evaluate(X_test, y_test)
print('Loss en el conjunto de test:', test_loss)
print('Precisión en el conjunto de test:', test_acc)
```

```
13/13 [==============================] - 0s 2ms/step - loss: 0.4478 - accuracy: 0.8875
Loss en el conjunto de test: 0.4477619230747223
Precisión en el conjunto de test: 0.887499988079071
```

Una vez tenemos los resultados del modelo en el subconjunto de test, ¿podríamos decir que el modelo es mejor que una clasificación al azar? Justifica tu respuesta

Respuesta aquí

- Sí, podemos decir que el modelo es mejor que una clasificación al azar, ya que su precisión en el conjunto de test es del 88.75%, lo que significa que fue capaz de clasificar correctamente el 88.75% de los móviles en el conjunto de test.

Muestra la salida de la predicción para el primer registro del conjunto de test e interpreta la salida: ¿Qué significa cada uno de los valores de salida? ¿Cuál es la clase de salida del modelo? ¿Cuál es la salida real para ese registro? Teniendo en cuenta estas dos últimas respuestas, ¿es correcta la salida del modelo?

```
# Código aquí
import numpy as np
prediccion = model.predict(X_test[:1])
print('Predicción del modelo:', prediccion)
print('Clase de salida del modelo:', np.argmax(prediccion))
print('Salida real para ese registro:', y_test.iloc[0])
```

```
1/1 [==============================] - 0s 87ms/step
Predicción del modelo: [[7.9570472e-01 2.0085454e-01 3.0988154e-03 3.4188561e-04]]
Clase de salida del modelo: 0
Salida real para ese registro: price_range    0
Name: 1860, dtype: int64
```

- La predicción del modelo muestra cuatro valores, que representan la probabilidad de que el registro pertenezca a cada una de las cuatro categorías de precio posibles. La salida real para ese registro también es 0, lo que indica que el modelo ha clasificado correctamente el registro. Por tanto, podemos afirmar que la salida del modelo es correcta para este registro del conjunto de test.

2. Red Neuronal para problema de regresión (5 p.)

En este segundo apartado, utilizaremos una red neuronal para afrontar un problema de regresión. Es importante darse cuenta de las implicaciones que tiene en el diseño, entrenamiento y evaluación de las redes. Los pasos a seguir en este segundo apartado son muy similares a los del primero, por lo que prácticamente todas las explicaciones se pueden reutilizar, a las que se añadirán las consideraciones particulares de este ejercicio.

2.1. Obtención y análisis previo de los datos (1 p.)

Al igual que en el primer apartado, haremos uso de otro dataset extraído de la plataforma Kaggle. Concretamente, vamos a utilizar un dataset de precios de vuelos en India, cuyo objetivo es dar un precio de billete dadas unas características del vuelo. Tenéis disponible la información del dataset aquí. Por favor, entrad en el link y leed detenidamente el problema, la descripción de los datos, las descripciones de las columnas y los ficheros disponibles. Aunque veáis tres datasets distintos, el fichero que vamos a utilizar (y que os habréis podido descargar desde la plataforma) es el llamado Clean_Dataset.csv

Pregunta 9 (0.5 p):

Sin programar ni una línea de código ni abrir el dataset aún, simplemente aprovechando la información que proporciona Kaggle del dataset. ¿Cuántas aerolíneas se contemplan en el dataset?

Respuesta aquí

- Vistara 43%
- Air_India 27%

- Other (91402) 30%

Pregunta 10 (0.5 p):

Sin programar ni una línea de código ni abrir el dataset aún, simplemente aprovechando la información que proporciona Kaggle del dataset. ¿El atributo **departure_time** de qué tipo es? ¿Y el atributo **stops**?

Respuesta aquí

2.2. Adaptación de los datos (1 p.)

Tras haber analizado detenidamente los datos, el siguiente paso será empezar a "jugar" con ellos. Para eso, habrá que proceder a descargar el fichero *.csv* disponible en los archivos de la práctica y guardarlo en una ubicación accesible. Es importante conocer la ruta del fichero para poder leerlo ahora desde el Notebook.

```
# Este PATH (ruta) es válido si el Notebook y el dataset están en la misma carpeta. En caso d
# descargado en otro sitio y no lo puedas mover, simplemente pon el PATH absoluto del fichero
# como por ejemplo: C:\Users\pablo\Documents\data_ej2.csv
PATH_DEL_DATASET_2 = '/content/Clean_Dataset.csv'
data_2 = pd.read_csv(filepath_or_buffer = PATH_DEL_DATASET_2)
```

En la variable *data_2* tenemos almacenado el dataset que, otra vez, es de tipo *DataFrame*.

```
data_2.head()
```

	Unnamed: 0	airline	flight	source_city	departure_time	stops	arrival_time	desti
0	0	SpiceJet	SG-8709	Delhi	Evening	zero	Night	
1	1	SpiceJet	SG-8157	Delhi	Early_Morning	zero	Morning	
2	2	AirAsia	I5-764	Delhi	Early_Morning	zero	Early_Morning	
3	3	Vistara	UK-995	Delhi	Morning	zero	Afternoon	
4	4	Vistara	UK-963	Delhi	Morning	zero	Morning	

Analizando todas las columnas y con el objetivo de simplificar al máximo posible el trabajo, vamos a no utilizar el atributo de identificación de vuelo (columna **flight**), ya que tienes demasiados valores únicos y requeriría de hacer más trabajo de *Feature Engineering*. Además, la columna llamada *Unnamed:0* se crea automáticamente como índice, por lo que podemos borrarla también.

```
data_2 = data_2.drop(columns = ['Unnamed: 0', 'flight'])
```

```
data_2.head()
```

	airline	source_city	departure_time	stops	arrival_time	destination_city	class
0	SpiceJet	Delhi	Evening	zero	Night	Mumbai	Economy
1	SpiceJet	Delhi	Early_Morning	zero	Morning	Mumbai	Economy
2	AirAsia	Delhi	Early_Morning	zero	Early_Morning	Mumbai	Economy
3	Vistara	Delhi	Morning	zero	Afternoon	Mumbai	Economy
4	Vistara	Delhi	Morning	zero	Morning	Mumbai	Economy

Separaremos los atributos por su tipo, para poder aplicarles las transformaciones que sean necesarias para su tipo en particular. El tipo de un atributo muchas veces está claro, pero en otras ocasiones puede existir cierta libertad que permite tomar decisiones en cuanto a como tratar un atributo. Ejemplos de esto puede ser el campo **days_left** o el caso **stops**

```
COLUMNAS_CATEGORICAS_2 = ['airline', 'source_city', 'departure_time', 'arrival_time', 'destin
COLUMNAS_NUMERICAS_2 = ['duration', 'days_left']
```

Separamos los atributos categóricos en un *DataFrame* independiente y los codificamos utilizando *OneHotEncoding*, gracias a la función **get_dummies()** de *Pandas*.

```
data_2_cat = data_2[COLUMNAS_CATEGORICAS_2]
data_2_cat = pd.get_dummies(data_2_cat)
```

```
data_2_cat
```

	airline_AirAsia	airline_Air_India	airline_GO_FIRST	airline_Indigo	airline_S
0	0	0	0	0	
1	0	0	0	0	
2	1	0	0	0	
3	0	0	0	0	
4	0	0	0	0	
...	...	...	...	...	
300148	0	0	0	0	
300149	0	0	0	0	
300150	0	0	0	0	
300151	0	0	0	0	

La columna **stops** aparentemente es de tipo categórico y se podría utilizar como tal (codificandolo a *OneHotEncoding* también), pero vamos a transformarlo en un valor numérico para reducir el número de columnas y simplificar la preparación de los datos

```
translation_dict = {
    "one": 1,
    "zero": 0,
    "two_or_more": 2
}
```

Reconstruiremos el dataset ya transformado concatenando tres *DataFrame* independientes:

- El *DataFrame* de atributos categóricos transformados a *OneHotEncoding* : **data_2_cat**
- La columna **stops** transformada con el diccionario de traducción definido arriba: **data_2['stops'].apply(lambda x: translation_dict.get(x))**
- Un *DataFrame* auxiliar que recoge únicamente los atributos nominales, sobre los que no vamos a aplicar ninguna transformación: **data_2[COLUMNAS_NUMERICAS_2]**

El resultado lo almacenaremos en el *DataFrame* llamado *X_limpia_2*, que ya tiene todos los atributos que introduciremos como entrada al modelo

```
X_limpia_2 = pd.concat([data_2_cat, data_2['stops'].apply(lambda x: translation_dict.get(x)),
```

Ahora tenemos que tratar con el atributo objetivo, en este caso el llamado **price**, que recuperaremos del *DataFrame* inicial

```
y_2 = data_2[['price']]

y_2.head()
```

	price
0	5953
1	5953
2	5956
3	5955
4	5955

Si nos fijamos, los valores de precio son de tipo numeral y tiene un valor mucho más grande que los valores numéricos de los atributos (en su mayoría binarios por la codificación OneHot o los nominales de las dos columnas numéricas que habíamos identificado). Trabajar con valores pequeños y bastante acotados para predecir valores muy grandes puede suponer problemas, al tener que generarse matrices de pesos con valores muy elevados. Para aliviar este problema, procederemos a normalizar el valor de precio utilizando *MinMaxScaler* de *scikit-learn*, que acota los valores entre 0 y 1, siendo 0 el valor mínimo que se encuentre en la columna y 1 el valor máximo de la misma

```
mmc = MinMaxScaler()

y_2 = pd.DataFrame(mmc.fit_transform(y_2.to_numpy()), columns = ['price'])

y_2.head()
```

	price
0	0.039749
1	0.039749
2	0.039773
3	0.039765
4	0.039765

▾ Pregunta 11 (0.5 p):

Tras haber separado X e y podemos ir empezando a sacar conclusiones de cara al diseño de la red neuronal. ¿Con cuántos registros contamos en el dataset? ¿Cuántos atributos usaremos para clasificar? ¿La salida de la red qué dimensión tiene?

Es fundamental que se vea el código utilizado para responder cada una de estas preguntas

```
# Código de la respuesta aquí
# Obtener el tamaño de X_limpia_2
print("Tamaño de X_limpia_2:", X_limpia_2.shape)

# Obtener el tamaño de y_2
print("Tamaño de y_2:", y_2.shape)
```

```
Tamaño de X_limpia_2: (300153, 35)
Tamaño de y_2: (300153, 1)
```

Respuesta aquí

- El conjunto de datos Flight Price Prediction de Kaggle tiene un total de 300153 registros.
- Para la clasificación se utilizarán 35 atributos.
- La salida de la red neuronal tendrá una dimensión de 1 atributo.

Pregunta 12 (0.5 p):

Una vez que tenemos los conjuntos X_limpia_2 e y_2 listos, el siguiente paso será crear los subconjuntos de entrenamiento, validación y test necesarios en cualquier problema de Machine Learning. Los tamaños de cada uno de definen a continuación:

- El subconjunto de **entrenamiento** tiene que tener un 80% de los registros totales. Almacenar en variables de nombre X_train_2, y_train_2
- El subconjunto de **test** tiene que tener el 20% de los registros totales. Almacenar en variables de nombre X_test_2, y_test_2
- El subconjunto de **validación** tiene que tener un 20% de los registros del subconjunto de **entrenamiento**. Almacenar en variables de nombre X_val_2, y_val_2

*Nota: Hay muchas formas de separar conjuntos, pero quizás la mejor y más sencilla sea utilizar el método **train_test_split()** de scikit-learn*

```
#Código aquí
from sklearn.model_selection import train_test_split

# Dividir el conjunto de datos en subconjuntos de entrenamiento y test
X_train_2, X_test_2, y_train_2, y_test_2 = train_test_split(X_limpia_2, y_2, test_size=0.2, r
```

```
# Dividir el subconjunto de entrenamiento en subconjuntos de entrenamiento y validación
X_train_2, X_val_2, y_train_2, y_val_2 = train_test_split(X_train_2, y_train_2, test_size=0.2
```

Ya tenemos los subconjuntos creados:

```
print("X TRAIN: ", X_train_2.shape)
print("X TEST shape: ", X_test_2.shape)
print("X VAL shape: ", X_val_2.shape)
print("y TRAIN shape: ", y_train_2.shape)
print("y TEST shape: ", y_test_2.shape)
print("y VAL shape: ", y_val_2.shape)
```

```
X TRAIN:  (192097, 35)
X TEST shape:  (60031, 35)
X VAL shape:  (48025, 35)
y TRAIN shape:  (192097, 1)
y TEST shape:  (60031, 1)
y VAL shape:  (48025, 1)
```

2.3. Creación, entrenamiento y evaluación de la Red Neuronal (3 p.)

Llegamos a la parte final: el momento de diseñar, entrenar y evaluar el modelo. Para ello, como ya hemos dicho anteriormente, vamos a hacer uso de Keras. Hay multitud de documentación acerca de esta librería, además de mucha comunidad que lo utiliza asiduamente, por lo que es fácil ver ejemplos similares sobre los que trabajar.

Pregunta 13 (1 p): Diseño, creación y compilación del modelo

En esta primera práctica, el diseño de la Red Neuronal va a ser fijo. En la siguiente práctica tendréis más libertad para jugar con los hiperparámetros de la red. Por tanto, crea una red con las siguientes características:

- Dos capas ocultas con tamaños 32 y 16, respectivamente, que utilizen como función de activación **ReLU**
- La capa de salida tiene que usar una función de activación **ReLU**

El modelo tiene que ser compilado con las siguientes propiedades:

- El optimizador a utilizar debe de ser **Adam**
- Las métricas a mostrar serán la de **loss** y **accuracy**
- La función de pérdida debe de ser la más acorde al problema de clasificación

```
# Código aquí
from keras.models import Sequential
from keras.layers import Dense
```

```
# Crear el modelo
modelo = Sequential()
modelo.add(Dense(32, input_dim=X_train_2.shape[1], activation='relu'))
modelo.add(Dense(16, activation='relu'))
modelo.add(Dense(1, activation='relu'))

# Compilar el modelo
modelo.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy', 'loss'])
```

Pregunta 14 (1 p): Entrenamiento del modelo

El modelo tiene que ser entrenado con las siguientes propiedades:

- Debe de ser entrenado durante **10** épocas
- Se debe utilizar un tamaño de batch de **64**
- Se debe utilizar el subconjunto de validación definido anteriormente para extraer métricas

```
# Código aquí

modelo.compile(loss='mean_squared_error', optimizer='adam', metrics=['accuracy'])

historial = modelo.fit(X_train_2, y_train_2, validation_data=(X_val_2, y_val_2), epochs=10, b
```

```
Epoch 1/10
3002/3002 [==============================] - 8s 2ms/step - loss: 0.0412 - accuracy: 9.89
Epoch 2/10
3002/3002 [==============================] - 7s 2ms/step - loss: 0.0024 - accuracy: 9.94
Epoch 3/10
3002/3002 [==============================] - 7s 2ms/step - loss: 0.0020 - accuracy: 9.94
Epoch 4/10
3002/3002 [==============================] - 7s 2ms/step - loss: 0.0019 - accuracy: 9.99
Epoch 5/10
3002/3002 [==============================] - 7s 2ms/step - loss: 0.0018 - accuracy: 9.99
Epoch 6/10
3002/3002 [==============================] - 7s 2ms/step - loss: 0.0017 - accuracy: 9.99
Epoch 7/10
3002/3002 [==============================] - 7s 2ms/step - loss: 0.0016 - accuracy: 9.99
Epoch 8/10
3002/3002 [==============================] - 7s 2ms/step - loss: 0.0016 - accuracy: 9.99
Epoch 9/10
3002/3002 [==============================] - 7s 2ms/step - loss: 0.0015 - accuracy: 9.99
Epoch 10/10
3002/3002 [==============================] - 7s 2ms/step - loss: 0.0014 - accuracy: 9.99
```

¿Qué métrica es más importante a tener en cuenta en este problema? ¿Por qué? Atendiendo a esta métrica, ¿qué valor ha obtenido el modelo en el subconjunto de entrenamiento? ¿Y en el de validación?

- una métrica adecuada para evaluar el rendimiento del modelo es el error cuadrático medio (mean squared error, MSE). Esta métrica mide la cantidad de error cuadrático promedio entre la variable objetivo real y la predicción del modelo, lo que nos da una idea de qué tan bien el modelo puede predecir los valores reales.

▾ Pregunta 16 (1 p): Evaluación del modelo

Una vez creado, compilado y entrenado el modelo, es hora de evaluarlo utilizando el conjunto de test. ¿Cuáles son los valores de loss y accuracy en el conjunto de test?

```
# Código aquí
# Evaluar el modelo con el conjunto de test
resultado = modelo.evaluate(X_test_2, y_test_2, verbose=0)

# Mostrar los valores de pérdida y precisión en el conjunto de test
print("Pérdida en el conjunto de test:", resultado[0])
print("Precisión en el conjunto de test:", resultado[1])
```

```
Pérdida en el conjunto de test: 0.0014217731077224016
Precisión en el conjunto de test: 0.001016141613945365
```

Respuesta aquí

- el valor de precisión en el conjunto de test es muy bajo, lo que sugiere que el modelo no está haciendo predicciones precisas de los precios de los vuelos en general.

Muestra las predicciones de la red para los 10 primeros registros de test. ¿Qué significan esos valores? ¿Son los valores reales de precio?

```
# Código aquí

# Obtener las predicciones del modelo para los 10 primeros registros del subconjunto de test
predicciones = modelo.predict(X_test_2[:10])

# Mostrar las predicciones y los valores reales de precio
print("Predicciones del modelo:", predicciones)
print("Valores reales de precio:", y_test_2[:10])
```

```
1/1 [==============================] - 0s 61ms/step
Predicciones del modelo: [[0.02524903]
 [0.5393653 ]
 [0.03524019]
 [0.45885462]
 [0.03269888]
 [0.07725823]
 [0.16550241]
 [0.01946789]
 [0.3888037 ]
 [0.6357145 ]]
Valores reales de precio:            price
27131   0.051334
266857  0.522490
141228  0.041733
288329  0.484192
97334   0.044873
135931  0.028287
290630  0.186388
141944  0.022588
244517  0.255194
215903  0.620960
```

Respuesta aquí

- Los valores mostrados en las predicciones son las predicciones de precios escalados del modelo para los 10 primeros registros del subconjunto de test. Para obtener los precios reales, es necesario aplicar la transformación inversa de la escala MinMax a los valores reales. Los precios reales representan los valores reales de la variable objetivo (en este caso, el precio de los vuelos).

Muestra los 10 primeros registros de *y_test_2*. ¿Qué significan esos valores? ¿Son los valores reales de precio?

```
# Código aquí
# Aplicar la transformación inversa de la escala MinMax a los valores reales
precios_reales = mmc.inverse_transform(y_test_2[:10])

# Mostrar los precios reales
print("Precios reales:", precios_reales)
```

```
Precios reales: [[ 7366.]
 [64831.]
 [ 6195.]
 [60160.]
 [ 6578.]
 [ 4555.]
 [23838.]
```

```
[ 3860.]
[32230.]
[76841.]]
```

Recupera los valores reales de los billetes para los 10 primeros registros de test, tanto de las predicciones como las de *y_test_2*.

Pista: Hace rato usamos un MinMaxScaler para algo. Revisa la documentación para ver si algún método puede ayudar

Respuesta aquí

- Los valores que se muestran son los precios reales correspondientes a los 10 primeros registros del subconjunto de test, después de aplicar la transformación inversa de la escala MinMax. Estos precios reales representan los valores reales de la variable objetivo (en este caso, el precio de los vuelos) en la moneda correspondiente.

✓ 0 s se ejecutó 14:28