```
import pandas as pd
import numpy as np
import matplotlib
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix
from sklearn import preprocessing
from sklearn.model_selection import train_test_split
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense, Dropout
from keras.optimizers import SGD
from pathlib import Path
```

```
# Read the data
train = pd.read_csv("train.csv")
test = pd.read_csv("test.csv")
test_original = test.copy()
```

Exploring the data

From a first look at the data we can see that there are different types of variables, which include categorical and numerical variables. That means we will have to preprocess the data before we can use it to train the model.

```
# Check the data
train.head()
```

	id	FRAUDE	VALOR	HORA_AUX	Dist_max_NAL	Canal1	FECHA	COD_PAIS	CANAL	DIASE
0	9000000001	1	0.0	13	659.13	ATM_INT	20150501	US	ATM_INT	
1	9000000002	1	0.0	17	594.77	ATM_INT	20150515	US	ATM_INT	
2	9000000003	1	0.0	13	659.13	ATM_INT	20150501	US	ATM_INT	
3	9000000004	1	0.0	13	659.13	ATM_INT	20150501	US	ATM_INT	
4	9000000005	1	0.0	0	1.00	ATM_INT	20150510	CR	ATM_INT	

5 rows × 26 columns

Check for any null values present

Also, we can see that there are missing values in the data. We will have to deal with them before training the model.

```
train.isnull().sum()
```

```
id                   0
FRAUDE               0
VALOR                0
HORA_AUX             0
Dist_max_NAL         0
Canal1               0
FECHA                0
COD_PAIS             0
CANAL                0
DIASEM               0
DIAMES               0
FECHA_VIN           24
OFICINA_VIN         24
SEXO                55
SEGMENTO            24
EDAD                24
INGRESOS            24
EGRESOS             24
NROPAISES            0
Dist_Sum_INTER    1547
Dist_Mean_INTER   1547
Dist_Max_INTER    1547
NROCIUDADES          0
Dist_Mean_NAL      457
Dist_HOY             0
Dist_sum_NAL         0
dtype: int64
```

Now, in order to get a better understanding of the data, we can plot the distribution of the data.

```
train.describe()
```

	id	FRAUDE	VALOR	HORA_AUX	Dist_max_NAL	FECHA	DIASEM
count	2.965000e+03	2965.000000	2.965000e+03	2965.000000	2965.000000	2.965000e+03	2965.00000C
mean	6.890938e+09	0.246543	5.035695e+05	14.960877	314.656739	2.015051e+07	3.143002
std	9.739700e+09	0.431071	9.859497e+05	6.348607	295.142673	9.134641e+00	2.092284
min	2.364560e+06	0.000000	0.000000e+00	0.000000	1.000000	2.015050e+07	0.00000C
25%	2.552997e+09	0.000000	9.016001e+04	12.000000	24.830000	2.015050e+07	1.00000C
50%	6.142884e+09	0.000000	2.435912e+05	16.000000	243.620000	2.015052e+07	3.00000C
75%	9.000000e+09	0.000000	5.058190e+05	20.000000	594.770000	2.015052e+07	5.00000C
max	9.330050e+10	1.000000	2.001406e+07	23.000000	1310.460000	2.015053e+07	6.00000C

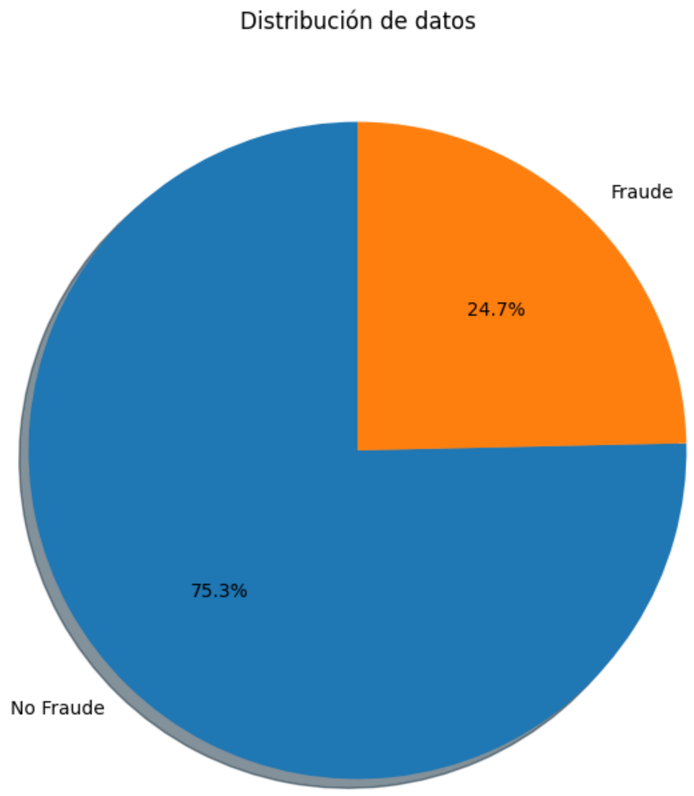
8 rows × 21 columns

Ploting the Fraud column

Upon plotting the data, we can see that the data is imbalanced, which means that there are more values of one class than the other.

```
values = train['FRAUDE'].value_counts()
values = np.array(values)
fig, ax = plt.subplots(1,1, figsize=(8,8))
plt.pie(values, labels=['No Fraude', 'Fraude'], autopct='%1.1f%%', shadow=True, startangle=90)
plt.title('Distribución de datos')
```

```
    Text(0.5, 1.0, 'Distribución de datos')
```



Data preprocessing

We will have to preprocess the data before we can use it to train the model. The preprocessing steps include:

1. Drop the columns that are not needed for the model, such as id, FECHA, FECHA_VIN, OFICINA_VIN, and the columns related to distance.
2. Normalize the numerical variables using the standard scaler by removing the mean and scaling to unit variance.

3. Drop the rows that have missing values in the column SEGMENTO.

```
train = train.drop(['id','Dist_max_NAL','Dist_Sum_INTER','Dist_Mean_INTER','Dist_Max_INTER','Dist_Mean_NAL','Dist_HOY','Dist_sum_NAL','FECHA','FECHA_VIN',
cols_to_norm = ['VALOR','INGRESOS','EGRESOS']
train[cols_to_norm] = train[cols_to_norm].apply(lambda x: (x - x.mean()) / (x.std()))
train = train.dropna(subset=['SEGMENTO'])
```

4. Encode the categorical variables using the label encoder from sklearn.

```
types = train.dtypes
obj = train.select_dtypes(include = "object").columns
# Convert the categorical variables to numeric
label_encoder = preprocessing.LabelEncoder()
for label in obj:
    train[label] = label_encoder.fit_transform(train[label].astype(str))
```

Defining training data

For the model to be able to learn, we will have to define the input and output variables. The input variables are all the columns except the output variable, which is FRAUDE.

Then we will split the data into train and test sets. We will use 80% of the data to train the model and 20% to test it.

```
# Define the output and input variables
y = train['FRAUDE']
x = train.drop(['FRAUDE'],axis=1)

# Split the data into train and test sets (80% train, 20% test)
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size = 0.2, random_state = 0)
```

Building the model

The model is built using Keras' Sequential API. This allows for the easy creation of a linear stack of layers. The model consists of five dense (fully connected) layers and one dropout layer for regularization. The dense layers use the Rectified Linear Unit (ReLU) activation function, except for the last layer, which uses the sigmoid activation function because it is a binary classification problem.

The dropout layer helps prevent overfitting by randomly setting a fraction of the input units to zero during training. Overfitting occurs when a model becomes too specialized for the training data and performs poorly on new, unseen data. The dropout layer randomly sets a fraction of input units to zero during training, which helps prevent the model from relying too heavily on any single feature or neuron.

```
model = Sequential([
    Dense(input_dim = x_train.shape[1], units = 100, activation = 'relu'),
    Dense(units = 60, activation = 'relu'),
    Dropout(0.5),
    Dense(units = 30, activation = 'relu'),
    Dense(units = 10, activation = 'relu'),
    Dense(units = 1, activation = 'sigmoid')
])
```

The model is built using the Adam optimizer and the binary cross-entropy loss function, which are appropriate for binary classification problems. The model is then trained on the training data with a batch size of 10 and for 100 epochs.

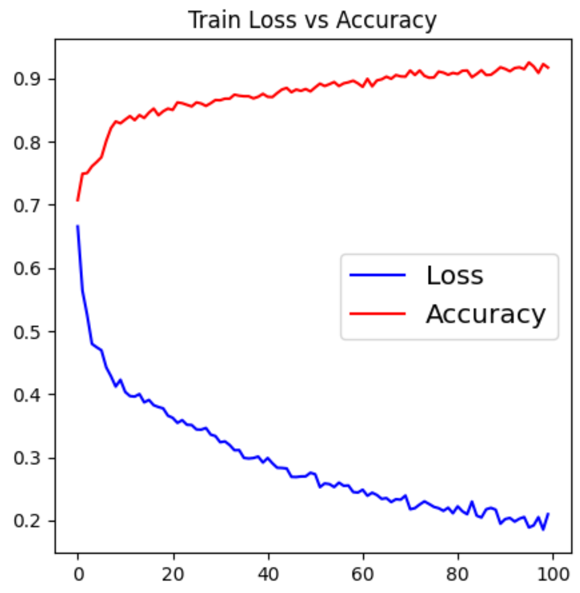
```
optimizador = keras.optimizers.Adam(learning_rate=(0.001))
model.compile(optimizer = optimizador, loss = 'binary_crossentropy', metrics = ['accuracy'])
history = model.fit(x_train, y_train, batch_size = 10, epochs = 100)
```

```
Epoch 80/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2113 - accuracy: 0.9090
Epoch 81/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2223 - accuracy: 0.9073
Epoch 82/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2144 - accuracy: 0.9124
Epoch 83/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2098 - accuracy: 0.9128
Epoch 84/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2298 - accuracy: 0.9022
Epoch 85/100
236/236 [==============================] - 1s 3ms/step - loss: 0.2076 - accuracy: 0.9073
Epoch 86/100
236/236 [==============================] - 1s 3ms/step - loss: 0.2045 - accuracy: 0.9133
Epoch 87/100
236/236 [==============================] - 1s 3ms/step - loss: 0.2175 - accuracy: 0.9056
Epoch 88/100
236/236 [==============================] - 1s 4ms/step - loss: 0.2199 - accuracy: 0.9060
Epoch 89/100
236/236 [==============================] - 1s 4ms/step - loss: 0.2169 - accuracy: 0.9116
Epoch 90/100
236/236 [==============================] - 1s 2ms/step - loss: 0.1947 - accuracy: 0.9179
Epoch 91/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2021 - accuracy: 0.9154
Epoch 92/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2039 - accuracy: 0.9116
Epoch 93/100
236/236 [==============================] - 1s 2ms/step - loss: 0.1981 - accuracy: 0.9167
Epoch 94/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2027 - accuracy: 0.9179
Epoch 95/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2053 - accuracy: 0.9150
Epoch 96/100
236/236 [==============================] - 1s 2ms/step - loss: 0.1888 - accuracy: 0.9256
Epoch 97/100
236/236 [==============================] - 1s 2ms/step - loss: 0.1921 - accuracy: 0.9192
Epoch 98/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2051 - accuracy: 0.9090
Epoch 99/100
236/236 [==============================] - 1s 2ms/step - loss: 0.1854 - accuracy: 0.9230
Epoch 100/100
236/236 [==============================] - 1s 2ms/step - loss: 0.2098 - accuracy: 0.9175
```

The loss and accuracy of the training process are plotted to visualize the performance of the model during training. It is expected that the loss will decrease and the accuracy will increase with each epoch.

```
# Plot the training loss and accuracy
fig, bx = plt.subplots(1,1, figsize=(5,5))
bx.plot(history.history['loss'],color='b')
bx.plot(history.history['accuracy'],color='r')
bx.set_title('Train Loss vs Accuracy', fontsize=12)
bx.legend(['Loss', 'Accuracy'], loc='best', fontsize='x-large')
```

```
<matplotlib.legend.Legend at 0x7dd756b75450>
```



The final score from loss and accuracy

```
# Evaluate the model
score = model.evaluate(x_test,y_test)
```

```
    19/19 [==============================] - 0s 2ms/step - loss: 0.3523 - accuracy: 0.8846
```

Make predictions on the test set

```
y_prediction = model.predict(x_test)
test_array = np.array(y_test)
unique, counts = np.unique(test_array, return_counts=True)
dict(zip(unique, counts))
```

```
    19/19 [==============================] - 0s 2ms/step
    {0: 449, 1: 140}
```

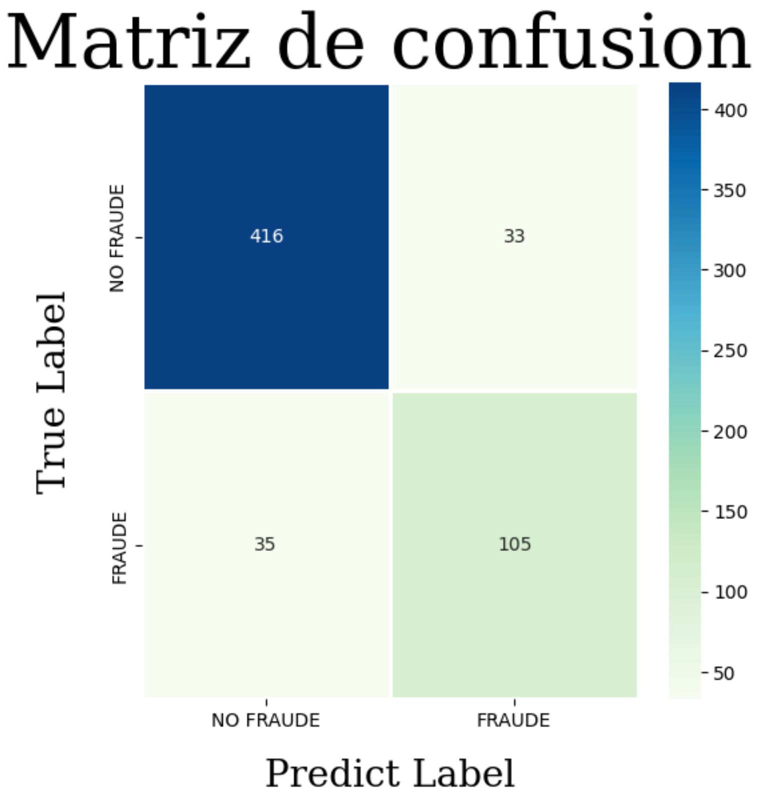
Evaluation of the model

The confusion matrix is a table often used to describe the performance of a classification model on a set of test data for which the true values are known. The confusion matrix shows the number of correct and incorrect predictions made by the model compared to the actual results. The results are accuracy, precision, recall and F1 score. Accuracy is the proportion of the total number of predictions that were correct. Precision is the proportion of positive predictions that were actually correct. Recall is the proportion of positive predictions that were actually classified correctly. The F1 score is the harmonic mean of precision and recall.

```
cm = confusion_matrix(test_array,y_prediction.round())
tn, fp, fn, tp = confusion_matrix(test_array,y_prediction.round()).ravel()
result = pd.DataFrame(cm, index=['NO FRAUDE','FRAUDE'],columns=['NO FRAUDE','FRAUDE'])

# Plot the confusion matrix
fig = plt.figure(figsize= (6,6))
ax = fig.add_subplot(1,1,1)
sns.heatmap(result, annot=True, linewidths=1, linecolor= 'white',fmt= 'd', cmap="GnBu")
font= {'family': 'serif',
         'color': 'darkred',
        'weight': 'normal',
        'size':14}
ax.set_title("Matriz de confusion ",fontdict ={'family': 'serif','color': 'black','weight': 'normal','size':40})
ax.set_ylabel("True Label", labelpad=20, fontdict={'family': 'serif','color': 'black','weight': 'normal','size':20})
ax.set_xlabel("Predict Label", labelpad=15, fontdict={'family': 'serif','color': 'black','weight': 'normal','size':20})
```

```
    Text(0.5, 36.72222222222221, 'Predict Label')
```



```
precision=tp/(tp+fp)
recall=tp/(tp+fn)
F1 = (2*precision*recall)/(precision+recall)
accuracy=tp/(tp+fn+fp)
print('\n Accuracy:%f \n Precision:%f  \n Recall:%f  \n F1-Score:%f '%(accuracy,precision,recall,F1))
```

```
Accuracy:0.606936
Precision:0.760870
Recall:0.750000
F1-Score:0.755396
```

Verification of the test values

To evaluate the model on the test data, we need to preprocess the data in the same way as we did for the training data. First, we will select the columns used for training the model, and then we will normalize the numerical variables using the standard scaler.

```
test = test[train.columns]
test = test.drop(['FRAUDE'],axis=1)

cols_to_norm = ['VALOR','INGRESOS','EGRESOS']
test[cols_to_norm] = test[cols_to_norm].apply(lambda x: (x - x.mean()) / (x.std()))
obj = test.select_dtypes(include = "object").columns
label_encoder = preprocessing.LabelEncoder()

for label in obj:
    test[label] = label_encoder.fit_transform(test[label].astype(str))
```

Make predictions on the test data

```
test_evaluado = model.predict(test)
# Add the predictions to the test dataframe
test_original['FRAUDE'] = test_evaluado.round()
```

```
4/4 [==============================] - 0s 3ms/step
```

```
test_original.head()
```

	id	FRAUDE	VALOR	HORA_AUX	Dist_max_COL	Dist_max_INTER	Canal1	FECHA_FRAUDE	COD_PAIS	CANAL	...	Dist_Mean_INTER	Dist_Max_INTER
0	98523068	1.0	42230.09	18	1.00	1.00	POS	20150515	US	POS	...	NaN	NaN
1	300237898	1.0	143202.65	20	614.04	7632.97	POS	20150506	US	MCI	...	6092.69	7632.97
2	943273308	1.0	243591.25	2	286.84	2443.14	ATM_INT	20150517	EC	ATM_INT	...	1743.52	2443.14
3	951645809	1.0	238267.40	20	1.00	1.00	ATM_INT	20150508	EC	ATM_INT	...	NaN	NaN
4	963797516	1.0	490403.58	13	1.00	1.00	ATM_INT	20150501	US	ATM_INT	...	NaN	NaN

5 rows × 32 columns

```
# Save the predictions to an Excel file
# test_original.to_excel(base / "output/test_evaluado.xlsx", index = False)
```