Caso Práctico: Análisis de los datos

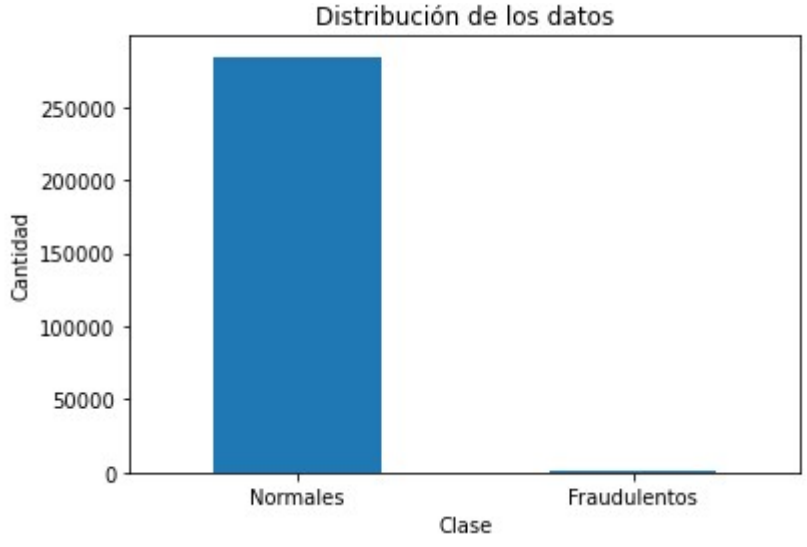
Primero importamos lo necesario y obtenemos los datos del csv.

```
import pandas as pd
import matplotlib.pyplot as plt

datos = pd.read_csv("creditcard.csv")
print(datos.head())

nr_clases = datos['Class'].value_counts(sort=True)
print(nr_clases)
```

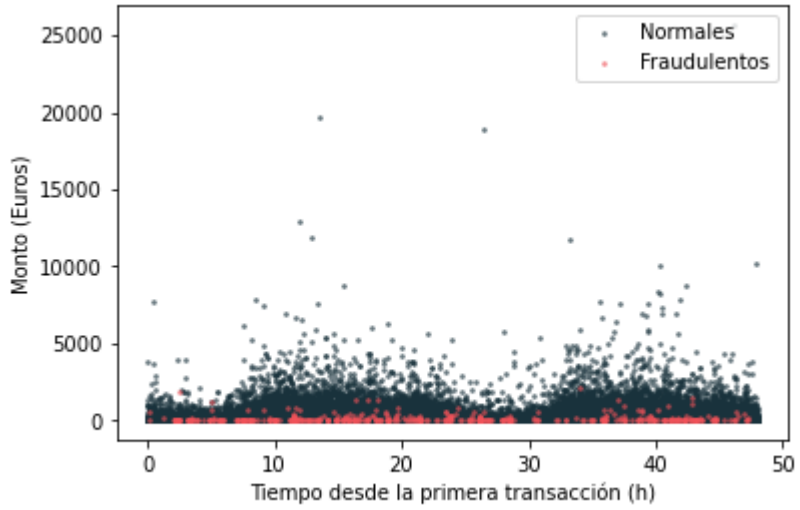
Observamos los primeros elementos de cada valor del dataset y además vemos el número de datos normales y fraudulentos.

```
   Time        V1        V2        V3        V4        V5        V6        V7  \
0   0.0 -1.359807 -0.072781  2.536347  1.378155 -0.338321  0.462388  0.239599
1   0.0  1.191857  0.266151  0.166480  0.448154  0.060018 -0.082361 -0.078803
2   1.0 -1.358354 -1.340163  1.773209  0.379780 -0.503198  1.800499  0.791461
3   1.0 -0.966272 -0.185226  1.792993 -0.863291 -0.010309  1.247203  0.237609
4   2.0 -1.158233  0.877737  1.548718  0.403034 -0.407193  0.095921  0.592941

         V8        V9  ...       V21       V22       V23       V24       V25  \
0  0.098698  0.363787  ... -0.018307  0.277838 -0.110474  0.066928  0.128539
1  0.085102 -0.255425  ... -0.225775 -0.638672  0.101288 -0.339846  0.167170
2  0.247676 -1.514654  ...  0.247998  0.771679  0.909412 -0.689281 -0.327642
3  0.377436 -1.387024  ... -0.108300  0.005274 -0.190321 -1.175575  0.647376
4 -0.270533  0.817739  ... -0.009431  0.798278 -0.137458  0.141267 -0.206010

        V26       V27       V28  Amount  Class
0 -0.189115  0.133558 -0.021053  149.62      0
1  0.125895 -0.008983  0.014724    2.69      0
2 -0.139097 -0.055353 -0.059752  378.66      0
3 -0.221929  0.062723  0.061458  123.50      0
4  0.502292  0.219422  0.215153   69.99      0

[5 rows x 31 columns]
0    284315
1       492
Name: Class, dtype: int64
```

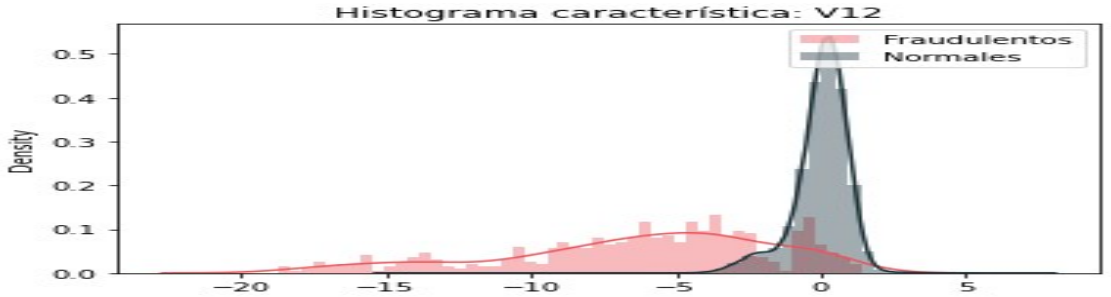
En la siguiente imagen podemos ver en un gráfico la distribución de los datos normales y fraudulentos.



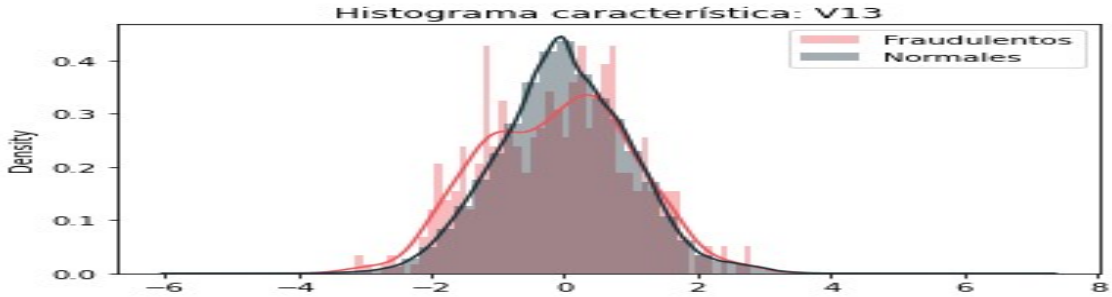
Podemos observar que hay muchas más transacciones normales que fraudulentas.
Ahora miraremos como se comportan a través del tiempo.



En negro podemos observar los datos normales y en rojo los datos fraudulentos, no podemos observar que siga un patrón claro con los que se pueda separar ambos tipos de datos.
Por último, vemos si las distribuciones de las características V1 a V28 presentan alguna diferencia entre los dos tipos de datos:




```
/home/alberto/anaconda3/lib/python3.8/site-package
precated function and will be removed in a future
level function with similar flexibility) or `histp
  warnings.warn(msg, FutureWarning)
/home/alberto/anaconda3/lib/python3.8/site-package
precated function and will be removed in a future
level function with similar flexibility) or `histp
  warnings.warn(msg, FutureWarning)
```



En ninguna característica se puede observar una diferencia clara entre los tipos de datos.
Vamos a separar los datos en train y test, lo haremos con la función train_test_split. Como hay muchos datos validos y muy pocos fraudulentos, lo que haremos para el entrenamiento será entrenar solo con los datos validos para que cuando sea un dato fraudulento del test, el error sea mayor y sepamos que es fraudulento.

```
from sklearn.preprocessing import StandardScaler
datos.drop(['Time'], axis=1, inplace=True)
datos['Amount'] = StandardScaler().fit_transform(datos['Amount'].values.reshape(-1,1))

from sklearn.model_selection import train_test_split

X_train, X_test = train_test_split(datos, test_size=0.2, random_state=42)
X_train = X_train[X_train.Class == 0]
X_train = X_train.drop(['Class'], axis=1)
X_train = X_train.values

Y_test = X_test['Class']
X_test = X_test.drop(['Class'], axis=1)
X_test = X_test.values
```

Ahora, con keras creamos un autoencoder que tendrá tanto de entrada como de salida la dimensión de X_train. Lo entrenamos con la función de error del error medio cuadrático el cuál se minimizará con el método del gradiente descendiente (SGD). Lo entrenamos durante 100 epoch y con un tamaño del lote de 32-

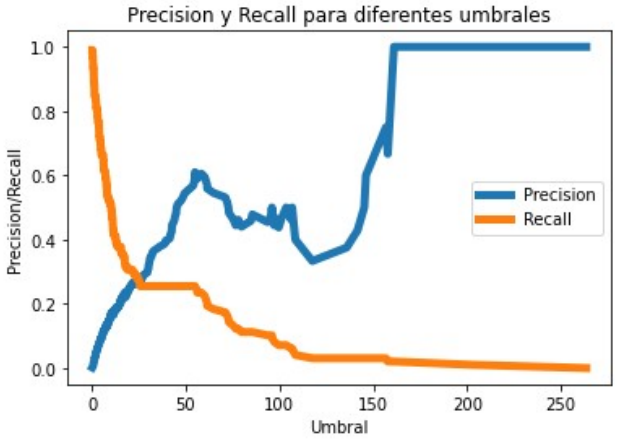
```
import numpy as np
np.random.seed(5)
from keras.models import Model, load_model
from keras.layers import Input, Dense

dim_entrada = X_train.shape[1]          # 29
capa_entrada = Input(shape=(dim_entrada,))

encoder = Dense(20, activation='tanh')(capa_entrada)
encoder = Dense(14, activation='relu')(encoder)

decoder = Dense(20, activation='tanh')(encoder)
decoder = Dense(29, activation='relu')(decoder)

autoencoder = Model(inputs=capa_entrada, outputs=decoder)

from keras.optimizers import SGD
sgd = SGD(learning_rate=0.01)
autoencoder.compile(optimizer='sgd', loss='mse')

nits = 100
tam_lote = 32
autoencoder.fit(X_train, X_train, epochs=nits, batch_size=tam_lote, shuffle=True, validation_data=(X_test,X_test), ve
```

```
/home/alberto/anaconda3/lib/python3.8/site-packages/keras/optimizers/optimizer_v2/gradient_descent.py:114: UserWarni
ng: The `lr` argument is deprecated, use `learning_rate` instead.
  super().__init__(name, **kwargs)
```

```
Epoch 1/100
7108/7108 [==============================] - 27s 4ms/step - loss: 0.9156 - val_loss: 0.8676
Epoch 2/100
7108/7108 [==============================] - 37s 5ms/step - loss: 0.8091 - val_loss: 0.8168
Epoch 3/100
7108/7108 [==============================] - 18s 2ms/step - loss: 0.7757 - val_loss: 0.7953
Epoch 4/100
7108/7108 [==============================] - 17s 2ms/step - loss: 0.7564 - val_loss: 0.7759
Epoch 5/100
7108/7108 [==============================] - 22s 3ms/step - loss: 0.7365 - val_loss: 0.7582
Epoch 6/100
7108/7108 [==============================] - 16s 2ms/step - loss: 0.7204 - val_loss: 0.7438
Epoch 7/100
```

Mostramos como ha ido evolucionando la precisión y el recall dependiendo del umbral.

```
1781/1781 [==============================] - 3s 2ms/step
(56962, 29)
```



Por último fijamos un umbral fijo y vemos la matriz de confusión que aparece.

```
umbral_fijo = 0.75
Y_pred = [1 if e > umbral_fijo else 0 for e in ecm]

conf_matrix = confusion_matrix(Y_test, Y_pred)
print(conf_matrix)
```

```
[[49373  7491]
 [    7    91]]
```

En el documento CasoPracticoCodigoModificado.ipynb podremos ver el código implementando distintas librerías.