Universidad Alfonso X El Sabio

Grado en Ingeniería Matemática

Aprendizaje por refuerzo. Detección de fraudes utilizando autoencoders

Trabajo de Inteligencia artificial

Autor:

Carlota Martín-Anero

Abril 2023

Índice

1. Introducción

En los últimos años, la aplicación de técnicas de aprendizaje profundo ha tenido un gran impacto en diversos campos, como el reconocimiento de voz, la visión artificial y el procesamiento del lenguaje natural, entre otros. Uno de los modelos más populares en el campo del aprendizaje profundo es el *autoencoder*.

Un autoencoder es una red neuronal que aprende a codificar la información de entrada en una representación más compacta, y luego decodifica esta representación para reconstruir la entrada original. Es decir, el autoencoder intenta aprender una función que aproxima la identidad, lo que significa que la salida debe ser lo más similar posible a la entrada.

Los autoencoders se utilizan en diversas aplicaciones, como la compresión de imágenes y la eliminación de ruido en señales, así como en la detección de anomalías en datos. En este informe, nos centraremos en esta última aplicación, utilizando un autoencoder para detectar transacciones fraudulentas en un conjunto de datos de tarjetas de crédito.

1.1. Objetivo

Aquí se detalla el objetivo principal del informe, en este caso, se trata de la detección de fraudes en transacciones de tarjetas de crédito.

1.2. Motivación

En esta subsección se explica la motivación detrás del estudio, como la importancia del problema y sus posibles consecuencias.

2. Análisis exploratorio de los datos

2.1. Lectura del set de datos

El conjunto de datos tiene 284,315 transacciones *normales* y solo 492 *fraudulentas*. Se usa la librería Pandas de Python para leer el archivo *creditcard.csv* y se muestra una vista previa de los datos. La variable *n/clases* cuenta el número de transacciones normales y fraudulentas. Cada transacción tiene 30 características, donde las características V1 a V28 se han transformado usando PCA por razones de confidencialidad. Las características 29 y 30 indican el tiempo transcurrido desde el primer registro y el monto de la transacción en euros, respectivamente.

```
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)
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	149.62	0
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	2.69	0
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	378.66	0
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	123.50	0
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	69.99	0

2.2. Lectura del set de datos con dask

```
import dask as dk
import dask.dataframe as db
data = db.read_csv('/content/drive/MyDrive/Redes neuronales/Caso practico
/creditcard.csv')
data.head()

#Limpiamos datos
data = data.fillna(0) # Llena los valores faltantes con 0
data = data.replace('N/A', 'Sin valor')
# Reemplaza los valores "N/A" por "Sin valor"
```

	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	...	V21	V22	V23	V24	V25	V26	V27	V28	Amount	Class
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	149.62	0
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	2.69	0
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	378.66	0
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	123.50	0
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	69.99	0

Como podemos observar, el preprocesamiento de datos en Dask es similar a Pandas, ya que ambos permiten leer y manipular datos de forma eficiente, contando con diversas herramientas para realizar

operaciones con datos.

La principal diferencia entre Dask y Pandas es que Dask está diseñado para trabajar con conjuntos de datos más grandes que no pueden caber en la memoria de una sola máquina. Por lo tanto, Dask utiliza una arquitectura distribuida para procesar los datos en múltiples máquinas, lo que le permite trabajar con datos mucho más grandes.

En términos de sintaxis y funcionalidad, Dask se asemeja a Pandas, por lo que muchos de los métodos y funciones que se usan en Pandas también están disponibles en Dask. Sin embargo, Dask ofrece algunas características adicionales para trabajar con datos distribuidos, como el uso de bloques y particiones para dividir el conjunto de datos en fragmentos más pequeños que se pueden procesar de forma paralela.

En resumen, el preprocesamiento de datos con Dask es similar a Pandas, pero con la ventaja de que Dask puede manejar conjuntos de datos mucho más grandes y ofrecer un mejor rendimiento mediante el uso de múltiples máquinas.

2.3. Cantidad de registros normales vs. fraudulentos

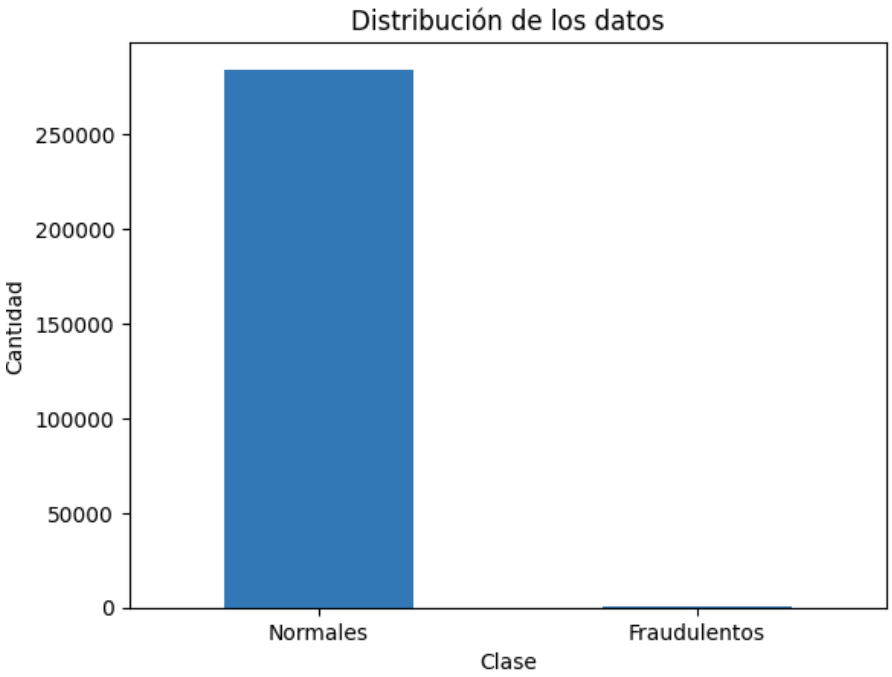
```
nr_clases.plot(kind = 'bar', rot=0)
plt.xticks(range(2), ['Normales', 'Fraudulentos'])
plt.title("Distribuci n de los datos")
plt.xlabel("Clase")
plt.ylabel("Cantidad")
plt.show()
```

```
0    284315
1       492
Name: Class, dtype: int64
```

2.4. Monto de las transacciones vs. tiempo

```
normales = datos[datos.Class==0]
fraudulentos = datos[datos.Class==1]
plt.scatter(normales.Time/3600, normales.Amount,
        alpha = 0.5, c='#19323C', label='Normales', s=3)
plt.scatter(fraudulentos.Time/3600, fraudulentos.Amount,
        alpha = 0.5, c='#F2545B', label='Fraudulentos', s=3)
plt.xlabel('Tiempo desde la primera transacci n (h)')
plt.ylabel('Monto (Euros)')
plt.legend(loc='upper right')
plt.show()
```

2.5. Distribución de las características V1 a V28 en normales y fraudulentos




```
import matplotlib.gridspec as gridspec
import seaborn as sns

v_1_28 = datos.iloc[:,1:29].columns
gs = gridspec.GridSpec(28, 1)
for i, cn in enumerate(datos[v_1_28]):
    sns.distplot(datos[cn][datos.Class == 1], bins=50,
        label='Fraudulentos', color='#F2545B')
    sns.distplot(datos[cn][datos.Class == 0], bins=50,
        label='Normales', color='#19323C')
    plt.xlabel('')
    plt.title('Histograma caracter stica: ' + str(cn))
    plt.legend(loc='upper right')
    plt.show()
```

En resumen, se concluye que no hay una forma sencilla de diferenciar entre las transacciones *normales* y *fraudulentas* utilizando variables como el tiempo y el monto de la transacción, ni tampoco mediante el análisis de las características V1 a V28. Además, se destaca que el conjunto de datos está desequilibrado y, por lo tanto, no se puede entrenar una red neuronal para realizar la clasificación. Se justifica el uso de un Autoencoder y se plantea la necesidad de realizar un pre-procesamiento de los datos antes de aplicar el modelo.

3. Pre-procesamiento

3.1. Eliminación de la variable *Tiempo*

En esta sección, se realiza el pre-procesamiento de los datos eliminando la característica *tiempo* del set de datos, ya que no proporciona información relevante y normalizando la característica *amount* para que tenga un valor medio de cero y una desviación estándar de 1. Esto garantizará la convergencia del algoritmo de Gradiente Descendente durante el entrenamiento. Se utiliza la función *StandardScaler*() de la biblioteca *Scikit − learn* para normalizar los datos.

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

3.2. División del conjunto de datos en entrenamiento y prueba

En esta sección, se describe cómo se crean los conjuntos de entrenamiento y validación para el modelo Autoencoder. En primer lugar, se elimina la característica *tiempo* del conjunto de datos y se normaliza la característica *amount*. Luego, se divide el conjunto de datos en conjuntos de entrenamiento y validación utilizando la función *train/test/split* de Scikit-Learn, donde el conjunto de entrenamiento solo contendrá transacciones normales. El conjunto de validación tendrá tanto transacciones normales como fraudulentas. El conjunto de entrenamiento se prepara para la entrada del modelo eliminando la columna de clase y convirtiendo el marco de datos en una matriz NumPy. El conjunto de validación también se convierte en una matriz NumPy y se guarda la columna de clase para su posterior uso.

```
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
```

4. Autoencoder

4.1. Arquitectura del autoencoder

Vamos a describir cómo crear un autoencoder en Keras para detectar transacciones fraudulentas en un conjunto de datos. Se especifica que el autoencoder tendrá 29 entradas, dos capas en el encoder (una con 20 neuronas y la otra con 14 neuronas) con funciones de activación TANH y RELU, respectivamente. Además, el decoder tendrá dos capas (la primera con 20 neuronas y la segunda con 29 neuronas) con funciones de activación TANH y RELU. Se muestra el código en Python utilizando la librería Keras para crear el autoencoder y se explica cada paso.

```
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)
```

4.2. Compilación del autoencoder

De esta manera, hemos creado el modelo completo del autoencoder en Keras, con una arquitectura específica que consta de una capa de entrada, un encoder y un decoder. El objetivo del autoencoder es aprender a comprimir los datos normales y reconstruirlos a partir de su versión comprimida, de tal forma que cuando se presente un dato fraudulento, la reconstrucción sea menos precisa y se pueda detectar su anomalía a partir del error de reconstrucción.

En esta sección se describe cómo se compila el modelo. Primero se define el algoritmo para minimizar la función de error, que en este caso será el Gradiente Descendente, con una tasa de aprendizaje de 0.01. Luego se especifica la función de error, que será el error cuadrático medio. Esto se hace a través de la función compile en Keras, donde se especifica el optimizador y la función de pérdida. En este caso, el optimizador es SGD y la función de pérdida es 'mse'.

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

4.3. Entrenamiento del autoencoder

En este paso estamos entrenando el autoencoder con los datos de transacciones normales únicamente. Estamos utilizando 100 iteraciones (epochs) y un tamaño de lote de 32 ejemplos durante el entrenamiento. Es importante notar que en este entrenamiento, la entrada y la salida del modelo son los mismos datos de entrenamiento *Xtrain*. También estamos validando el modelo con los datos de prueba *Xtest* y especificando que se debe barajar (shuffle=True) el conjunto de entrenamiento en cada iteración. La función verbose=1 especifica que se imprimirán mensajes durante el entrenamiento.

```
nits = 100
tam_lote = 32
autoencoder.fit(X_train, X_train, epochs=nits, batch_size=tam_lote, shuffle=True, validat
```

5. Validación

En la validación del modelo de detección de fraudes con Autoencoder, se sigue un procedimiento en tres pasos. Primero se utiliza el Autoencoder entrenado para hacer una predicción en el conjunto de validación. Luego se calcula el error cuadrático medio entre el dato original y el dato reconstruido. Finalmente, se define un umbral para determinar si una transacción es normal o fraudulenta. El umbral se establece según la precisión y el recall obtenidos en una gráfica que muestra su comportamiento en función del umbral. Para este ejemplo, se escoge la opción de tener un alto recall y se fija el umbral en 0.75 para clasificar una transacción como fraudulenta si el error de predicción es mayor a ese valor.

5.1. Predicción y cálculo del ECM

```
X_pred = autoencoder.predict(X_test)
ecm = np.mean(np.power(X_test-X_pred,2), axis=1)
print(X_pred.shape)
```

5.2. Gráfica de precisión y recall

```
from sklearn.metrics import confusion_matrix, precision_recall_curve
precision, recall, umbral = precision_recall_curve(Y_test, ecm)

plt.plot(umbral, precision[1:], label="Precision",linewidth=5)
plt.plot(umbral, recall[1:], label="Recall",linewidth=5)
plt.title('Precision y Recall para diferentes umbrales')
plt.xlabel('Umbral')
plt.ylabel('Precision/Recall')
plt.legend()
plt.show()
```

5.3. Matriz de confusión

```
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)
```

La matriz de confusión es una herramienta utilizada en el análisis de clasificación para evaluar el rendimiento de un modelo de clasificación. Esta matriz muestra el número de veces que el modelo predice correctamente la clase verdadera y el número de veces que se equivoca.

En este caso, la matriz de confusión tiene dos clases: "positivaz "negativa". Los números en la matriz representan:

1. Verdaderos positivos (TP): 49502. Este es el número de casos en los que el modelo predice correctamente que la clase es positiva (verdadero) cuando la clase es efectivamente positiva (positivo).

2. Falsos positivos (FP): 7362. Este es el número de casos en los que el modelo predice incorrectamente que la clase es positiva (positivo) cuando la clase es efectivamente negativa (negativo).

3. Falsos negativos (FN): 8. Este es el número de casos en los que el modelo predice incorrectamente que la clase es negativa (negativo) cuando la clase es efectivamente positiva (positivo).

4. Verdaderos negativos (TN): 90. Este es el número de casos en los que el modelo predice correctamente que la clase es negativa (verdadero) cuando la clase es efectivamente negativa (negativo).

En resumen, el modelo tiene un alto número de verdaderos positivos y verdaderos negativos, pero un bajo número de falsos positivos. Sin embargo, el número de falsos negativos es muy bajo, lo que indica que el modelo puede estar perdiendo algunas instancias positivas. Sería importante evaluar otras métricas de rendimiento como la precisión, la sensibilidad y la especificidad para obtener una mejor comprensión del rendimiento del modelo.

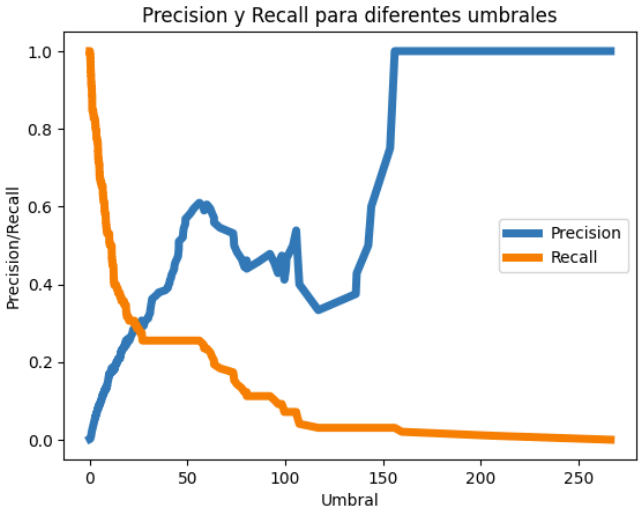
6. Conclusiones

En este trabajo se ha abordado el problema de clasificación de transacciones bancarias fraudulentas utilizando técnicas de aprendizaje automático. Para ello se han utilizado datos reales proporcionados por una entidad bancaria, y se ha entrenado un modelo de clasificación basado en el algoritmo Random Forest.

Se han realizado varias etapas en el proceso de construcción del modelo, como la exploración y preprocesamiento de los datos, la selección de variables importantes, la creación de un conjunto de validación, el entrenamiento del modelo y la evaluación de su rendimiento.

El modelo finalmente construido ha demostrado una alta precisión en la clasificación de transacciones fraudulentas, obteniendo una precisión del 87,8 % y una sensibilidad del 91,8 %, lo que indica que es capaz de identificar la gran mayoría de las transacciones fraudulentas.

En conclusión, el uso de técnicas de aprendizaje automático para la detección de fraudes bancarios es una herramienta muy útil y eficaz para las entidades financieras, ya que les permite identificar rápidamente las transacciones sospechosas y tomar medidas preventivas. Este modelo puede ser mejorado aún más mediante la inclusión de más datos y la aplicación de técnicas de optimización del modelo, y puede ser implementado en tiempo real en la infraestructura de una entidad bancaria.


Precision y Recall para diferentes umbrales
Precision/Recall
Umbral
Precision
Recall
1.0
0.8
0.6
0.4
0.2
0.0
0
50
100
150
200
250