Proyecto CPIA:
Detección de transacciones fraudulentas

Miguel Ausejo Gallego, Andreu Solà i Dagas

Diciembre 2025

1 Introducción

El objetivo de este proyecto es diseñar un sistema que nos permita detectar transacciones bancarias fraudulentas a partir de datos numéricos anonimizados. Concretamente, a partir de una transacción X (que se trata de un vector de d dimensiones porque estas operaciones tienen distintas características) queremos predecir un valor y que será 0 o 1, donde 1 significa que la transacción es fraudulenta y 0 que es legítima.

El problema tiene varios retos. Primero de todo, en el *dataset* que utilizamos hay un gran desbalance de clases: los fraudes son una minoría en comparación con las transacciones normales. Concretamente, en el *dataset* que utilizamos para entrenar los modelos hay 1289169 transacciones de las cuáles solo 7506 son fraudes, mientras que en el *dataset* con el que evaluamos los modelos solo hay 2145 transacciones fraudulentas entre 555719 totales. Además, no todos los errores tienen la misma importancia: es peor no detectar un fraude (falso negativo) que marcar como fraude una operación que en realidad no lo es (falso positivo).

Para medir si el sistema funciona bien debemos utilizar distintas métricas, no solo la precisión. Otras medidas muy importantes para tener en cuenta son la completitud, la F1-score, etc., que explicaremos en detalle más adelante. También es necesario elegir un umbral de decisión que permita ajustar el equilibrio entre falsos positivos y falsos negativos según lo que se quiera priorizar.

2 Experimentos

2.1 Entorno de Ejecución y Software

Hemos llevado a cabo el proyecto al completo en el entorno de desarrollo proporcionado por **Visual Studio Code**, ya que contamos con mucha experiencia con él. El lenguaje utilizado para el proyecto ha sido **Python**, concretamente la versión **3.13**. Este lenguaje ha sido escogido debido a su amplio uso y facilidades que ofrece para proyectos de aprendizaje automático. Además, el formato de los archivos no es *.py*, sinó que hemos usado *notebooks* con **.ipynb**, ya que este formato facilita trabajos de ML o análisis de datos al poder ejecutar celdas por separado y estructurar mejor cada parte del código.
Nuestro proyecto hace uso de distintas librerías científicas y de *Machine Learning* (ML), muy robustas y usadas. La gestión de recursos ha sido optimizada mediante procesos de paralelización, usando librerías, asignando un porcentaje de los CPU disponibles en la ejecución, no el total de CPUs porque podemos provocar que nuestro ordenador se cuelgue. Las principales librerías y módulos utilizados son, según su uso:

- **Manipulación de Datos: pandas** y **numpy** para la carga, limpieza y vectorización de operaciones.
- **Procesamiento Paralelo: multiprocessing.pool.ThreadPool** y **os** para la carga simultánea de datasets y tareas de preprocesamiento. Con **os** podemos calcular el número de CPUs disponibles para asignar un porcentaje de ellas o su totalidad.
- **Machine Learning Clásico: scikit-learn** (**sklearn**) para modelos base (**Random Forest**, **MLP**), métricas de evaluación (**accuracy**, **recall**, **f1-score**, **confusion_matrix**) y preprocesamiento (**OrdinalEncoder**, **OneHotEncoder**, **StandardScaler**, **RobustScaler**).

- **Deep Learning: torch** (**PyTorch**) y **torch.nn** para la implementación de redes neuronales profundas (**Autoencoder**) y **pytorch_tabnet** para el modelo **TabNet**.
- **Optimización de Hiperparámetros: optuna** para la búsqueda bayesiana de hiperparámetros, utilizando **TPESampler** (Tree-structured Parzen Estimator) y **MedianPruner** para la eficiencia computacional.
- **Manejo de Desbalanceo: imblearn** (**imbalanced-learn**) para técnicas de sobremuestreo sintético (**SMOTE**) en pipelines de entrenamiento.

2.2 Descripción del Dataset

A lo largo del proyecto se han utilizado los datos del *dataset* de *Kaggle* **Credit Card Fraud Detection**, el cual contiene transacciones bancarias simuladas que replican patrones de fraude del mundo real. Las características de nuestro *dataset* son las siguientes:

- **Estructura:** El dataset consta de dos archivos principales: `fraudTrain.csv` (entrenamiento, con más de 1 millón de transacciones) y `fraudTest.csv` (test, con más de medio millón).
- **Características (Features):** Originalmente cuenta con 23 columnas, incluyendo información temporal (`trans_date_trans_time`), datos del titular (`dob`, `gender`, `job`, `city`), detalles de la transacción (`category`, `merchant`, etc.) e identificadores únicos.
- **Variable Objetivo:** `is_fraud`, una variable binaria donde **0** indica una transacción legítima y **1** indica fraude. El dataset presenta un fuerte desbalance de clases, con una mayoría abrumadora de transacciones legítimas frente a una minoría de fraudes, lo que condiciona la estrategia de modelado.

2.3 Preprocesamiento

Antes de pasar a la creación de nuestros modelos de ML, es importante visualizar, estudiar y pulir los datos. Nuestros datasets tienen variables numéricas y categóricas (de texto). Además, había columnas que presentaban muchas anomalías. Por eso, el preprocesamiento consta de distintas partes, específicas según los requerimientos de los datos y también según la arquitectura de cada modelo.

2.3.1 Limpieza y Transformación General

Gradias a que hemos usado celdas, podemos limpiar y transformar de forma sencilla los datos de nuestro dataset. Utilizando algunas de las librerías ya mencionadas, especialmente de **pandas** y **scikit-learn**, hemos hecho las siguientes operaciones:

- **Eliminación de Ruido:** Se descartaron columnas que no aportaban información para mejorar predicciones e identificadores únicos que pueden causar sobreajuste (`Unnamed: 0`, `cc_num`, `trans_num`).
- ***Features* Temporales:** Las columnas de fecha (`trans_date_trans_time` y `dob`) se transformaron a formato numérico **Unix Timestamp** (segundos desde el 1 de enero de 1970), permitiendo a los modelos interpretar la temporalidad y la edad de forma numérica.

Esto es solo la primera parte del preprocesamiento, que es igual para todos los distintos modelos que hemos entrenado.

2.3.2 Estrategias de Codificación (*Encoding*)

Además de pasar fechas a formato numérico, el *dataset* también contiene variables categóricas como por ejemplo el nombre del comercio o del emisor, el apellido, calle, etc. Dependiendo del modelo, se emplean distinas estrategias para codificar estas columnas, que también variarion ligeramente dependiendo del algoritmo entrenado:

- **Random Forest y MLP:** Utilizamos el `OrdinalEncoder` de `sklearn`. Se implementó una estrategia para manejar valores desconocidos en el conjunto de test (`handle_unknown='use_encoded_value'`), asignándoles el valor `-1`. Esto implica que, si un valor categórico no ha sido visto en el conjunto de entrenamiento, se considera como desconocido.

- **Para TabNet:** Aplicamos el método `get_dummies` (modo de One-Hot Encoding) a variables seleccionadas (`gender`, `category`, `state`) para crear representaciones binarias de estos valores. Se alinearon las columnas entre train y test para asegurar consistencia dimensional, ya que en este tipo de codificación las dimensiones pueden variar y provocar problemas en el entrenamiento.
- **Para Autoencoder:** Diseñamos un **enfoque híbrido**:
  - **OneHotEncoder:** Para variables de baja cardinalidad, es decir, con pocos valores únicos (`gender`, `category`).
  - **OrdinalEncoder:** Para variables de alta cardinalidad, con muchos valores únicos (`merchant`, `job`, `city`, etc.).

Todas estas operaciones de codificación se explican en detalle más adelante, en la implementación del código. Además, estan explicadas igualmente en los *notebooks* comentados del código. También se aplicó el escalado de los datos en todos los modelos, utilizando las librerías importadas para escalado ya mencionadas.

2.4 Creación de Modelos y Algoritmos

2.4.1 Random Forest (RF)

Entrenamos un clasificador **Random Forest** optimizado con la librería de **Optuna**. Para la optimización, realizamos 50 intentos (aunque este valor se puede cambiar como queramos) buscando maximizar la métrica de la completitud, o recall, que es la cantidad de fraudes totales (la clase minoritaria) que detectamos. Cuántos más detectemos, mejor, aunque esto suele ir acompañado de un incremento en el número de falsos positivos.
El espacio de parámetros en el cual realizamos la búsqueda consistía en una combinación entre distintos valores de estos mismos. Se exploraron hiperparámetros como `n_estimators` (20–50), `max_depth` (10–50 o `None`), `min_samples_split`, `min_samples_leaf` y `max_features`.
Una vez encontrada la combinación de parámetros óptimos de entre los que habíamos propuesto, podemos tener una aproximación de cuál es el mejor modelo que podemos crear, que viene definido por los parámetros.

2.4.2 Multi-Layer Perceptron (MLP)

También implementamos una red neuronal **MLP** en una *Pipeline* con el escalado y balanceo de datos, usando **SMOTE** (Synthetic Minority Over-sampling Technique). Esta última técnica consiste en generar muestras sintéticas de la clase minoritaria, los fraudes, para entrenar el modelo con mayor exposición a ellos.
Para hallar la estructura más óptima de la red neuronal utilizamos también **Optuna**, probando distintos números de capas ocultas con distintos números de neuronas, diferentes funciones de activación **ReLu** y **tanh**, **learning_rate_init** y regularicación **alpha**. El solver utilizado fue **adam** con **early stopping**, que hace que si el entrenamiento no mejora en un cierto número de épocas, lo ature inmediatamente para evitar *overfitting* y reducir el tiempo necesario para la ejecución.
Probando combinaciones de estos valores obtuvimos los más óptimos para entrenar un **MLP** en nuestro problema de detección de fraudes.

2.4.3 Autoencoder

Decidimos probar también con un modelo de aprendizaje no supervisado (o semi-supervisado), aunque como mostramos más adelante no es el más óptimo. El **Autoencoder** se basa en reconstruir las transacciones a partir de los patrones aprendidos anteriormente, y se implementa con la librería **PyTorch**.
Este modelo se centra en transacciones normales solamente, y se espera que sea capaz de aprender a como comprimir y descomprimir transacciones legítimas mientras que cometa más error de reconstrucción en fraudes, ya que presentan más anomalías y no los habrá visto nunca.
La arquitectura es ligeramente distinta en este caso: es una red profunda que comprima las *features* hasta un cierto cuello de botella, con activación **ReLu** y **Dropout**. Esto es el *Encoder*. Luego, el *Decoder* se encarga de pasar del cuello de botella a las dimensiones originales intentando no cometer ningún error. Dependiendo del cuello de botella que definamos podemos ajustar el modelo para mejorar en algunas métricas u otras.

2.4.4 Tabnet

Para el TabNet, un modelo de Deep learning, debemos utilizar la librería de **PyTorch** ya mencionada. Este modelo está especializado para tratar con datos tabulares, como es nuestro caso.
Usamos el optimizador **Adam**, con **lr=0.02**. Empleamos también la máscara **entmax** para la selección de características dispersas, para aprender relaciones más complejas entre las características de las transacciones. Se entrenó el modelo con hasta 100 épocas en lotes de datos de 1024, aunque este valor se puede cambiar, y con un *early stopping* de hasta 10 épocas sin mejora.

Todos los conceptos técnicos vistos hasta ahora, como la codificación y decodificación, se explicaran en detalle en apartados posteriores, como en la implementación.

2.5 Hardware y Tiempos de Ejecución

En cuanto al *hardware*, para ejecutar nuestro código no disponemos de ninguna GPU en nuestros ordenadores, por lo que se han utilizado puramente CPUs. Esto hace que el tiempo de ejecución sea más largo, ya que las GPUs son más potentes para hacer este tipo de cálculos y procesos. Mediante librerías se puede saber si el ordenador que utilizamos tiene GPU o CPU, así como el número de ellas.

Finalmente, la cantidad específica de ejecuciones y tiempos de cómputo tanto en paralelo como en secuencial de nuestros modelos se explicarán en detalle en la parte de resultado. No obstante, hemos podido ver como en algunos casos la paralelización sí disminuía el tiempo necesario de ejecuión, mientras que en otros no era óptimo aplicarlo porque el secuencial tarda incluso menos.

3 Descripción detallada de la implementación

En esta sección de describirá la implementación en Python llevada a cabo en este proyecto, con los recursos y librerías anteriormente mencionadas.

3.1 Carga, visualización de los datos

En cualquier trabajo de Machine Learning lo más importante es entender el *dataset* con el que se trabaja. Por lo tanto, el primer paso que hicimos fue importar las librerías de **numpy** y **pandas** para estudiar los datos. Luego, cargamos los archivos *csv* de entrenamiento y test que, como son muy grandes, utilizamos paralelización en este proceso con la librería de **multiprocessing**. Después, estudiamos los datos con métodos como **.info()** y **.head()**, entre otros. Con este estudio, nos dimos cuenta que muchas *features* que teníamos eran categóricos, es decir, texto (*strings*), lo que muchas veces puede causar problemas a la hora de enternar modelos de ML que solo aceptan valores numéricos.

3.2 Preprocesamiento de los datos

Con el estudio realizado de los datos, empezamos su preprocesamiento y limpieza eliminando la primera columna que es solo un índice de las distintas transacciones y, por lo tanto, no aporta ninguna información relevante. También eliminamos las columnas com el identificador y el número de la transacción, por el mismo motivo. Además, transformamos las columnas de fechas de la transación y de nacimiento del emisor a formato *timestamp*, ya que son categóricas. Este formato son los segundos pasados desde el 1 de enero de 1970. Evidentemente, esto lo hacemos para tanto los datos de entrenamiento como de test.

No obstante, hay más columnas categóricas, como las de *'gender', 'category', 'merchant', 'job', 'city', 'state', 'street', 'first', 'last'*, que nos encargaremos de transformarlas a formato numérico también. Para estas columnas, utilizamos el Ordinal Encoder, One-Hot Encoder o una mezcla híbrida de los dos, tal y como veremos dependiendo del caso. Estos módulos de la librería de sklearn se encargan de transformar de texto a números. El ordinal encoder trabaja asignando un número a cada valor de texto que encuentra. Por ejemplo, consideremos una variable categórica *Sexo* con dos categorías: Masculino y Femenino.

Mediante un **Ordinal Encoder**, a cada categoría se le asigna un valor entero:

$$\text{Masculino} \to 0, \quad \text{Femenino} \to 1$$

Este método introduce implícitamente un orden alfabético entre las categorías.

En cambio, con un **One-Hot Encoder**, cada categoría se representa mediante una columna binaria:

$$\text{Masculino} \to (1,0), \quad \text{Femenino} \to (0,1)$$

De este modo, no se asume ninguna relación de orden entre las categorías. No obstante, el One-Hot Encoder consume más recursos que el ordinal encoder al necesitar más memoria para almacenar toda la información. Finalmente, lo que obtenemos son dos datasets, una para el entrenamiento y otro para la evaluación del modelo, con todos los elementes en formato numérico. Este proceso ha sido paralelizado usando la librería de **multiprocessing** también.

Cabe destacar que la forma como transformamos de variables categóricas a numéricas puede variar ligeramente entre los distintos algoritmos y métodos planteados. Esto se debe a que algunas columnas pueden ser más o menos relevantes para ciertos algoritmos y cambiar la cantidad de memoria que se necesita, o que algunos modelos mejoren su rendimiento al usar un codificador One-Hot Encoder u Ordinal Encoder. Concretamente, en los primeros modelos más simples que ahora explicaremos (**Random Forest** y **Multi-Layer Perceptron (MLP)**, hemos usado Ordinal Encoder, mientras que para el **Autoencoder** hemos utilizado un modelo híbrido, y para el **TabNet** solamente One-Hot Encoder con el método de **.get_dummies()**.
Finalmente, una vez tenemos los datos tal y como deseamos, procedimos a separar las *features* de entrenamiento o test (todas las variables que dan información sobre la transacción) y la columna que da la respuesta de si ese transacción es legítima o fraudulenta, llamada **is_fraud**. Separamos el *dataset* de entrenamiento en **X_train**, que tiene las columnas con información para entrenar, y **y_train**, que tiene las respuestas de si es fraude, indicado con un 1, o si es legítima, con un 0. Hicimos exactamente lo mismo para el dataset de test.

A partir de aquí, estudiaremos los distintos modelos que hemos implementado, que son diferentes entre ellos pero cuentan con el mismo preprocesamiento y limpieza de los datos que hemos mencionado, de forma muy similar entre todos ellos.

3.3 Evaluación de Modelos Simples: Random Forest y MLP

Para tener una referencia sobre qué precisión podemos conseguir con modelos de ML en nuestro *dataset*, primero entrenamos dos modelos simples directamente importades desde la librería de **scikit-learn**: el **Random Forest** y el **MLP**.

3.3.1 Random Forest

El **Random Forest** es un método de ML que se basa en la combinación de muchos árboles de decisión, de aquí su nombre. Los distintos árboles que conforman el bosque se entrenan sobre una muestra aleatoria de los datos. Un árbol está compuesto por distintos nodos, y en cada uno de ellos solo se consideran un subconjunto aleatorio de las variables para hacer la división. Esto hace que los árboles cada vez vean los datos desde perspectivas ligeramente diferentes, tras cada división, para determinar y trazar relaciones entre los datos y llegar a la respuesta de si se trata de un fraude o no. Luego, cada árbol hace su predicción y, al final, se hace como una votacióni entre todos los árboles de decisión que conforman el bosque. Como si de un voto por mayoría se tratase, la predición final es la respuesta más votada por los árboles.

Entendido entonces como funciona un árbol de decisión, entrenamos el classificador que hemos importado directamente de **scikit-learn**. Los parámetros con los que podemos entrenar el modelo se llaman **hiperparámetros**, y podemos escoger su valor. Por ejemplo, podemos escoger el número de árboles que queremos en nuestro bosque con **n_estimators**, o la profundidad de cada árbol con **max_depth**. En esta parte, también importamos la librería **os**, que nos permite calcular el número de CPUs disponibles en nuestro ordinador. Así, podemos usar un porcentaje de estas para optimizar el entrenamiento del

modelo. No los usamos todos porque esto puede hacer que se cuelgue el ordenador y pare el entrenamiento porque hay otros procesos corriendo de fondo que necesitan también de estas CPUs.

Inicialmente escogemos valores completamente aleatorios de los hiperparémtros que, obviamente, acaban dando un resultado mediocre, detectando poquísimos fraudes. Este modelo así es inviable. Para solucionar este problema, importamos la librería de **optuna** son distintos módulos explicados en detalle en los comentarios del código. Esta librería nos permite buscar cuáles son los mejores hiperparámetres del **Random Forest** para encontrar el máximo de fraudes posibles en nuestro *dataset*. Nosotros mismos definimos una lista de parámetros sobre los que vamos a probar distintas combinaciones, es decir, distinto número de árboles de decisión, distintos valores para la profundidad, distintos mínimos valores para cada hoja del árbol, etc.
Optuna se encarga de estudiar distintas combinaciones de estos hiperparámetros y nos permite especificar qué métrica queremos maximizar: precisión (si clasifica las transacciones correctamente), completitud o recall (qué cantidad de fraudes detecta de entre los totales) o F1-Score (balance entre precisión y completitud), entre otras. Estas métricas estan explicadas con más detalle en posteriores secciones. Así, evaluamos distintos modelos intentando maximizar la completitud, por encima de la precisión, ya que es más importante detectar la máxima cantidad posible de fraudes que intentar predecir correctamente si se trata de un fraude o una transacción legítima. No obstante, maximizando el recall podemos obtener muchos falsos positivos (transacciones normales marcadas como fraudes), así que tenemos que encontrar un balance entre las métricas, aunque priorizando el recall. Debemos mencionar que en este entrenamiento hemos aplicado el **SMOTE**, importando el módulo, que permite crear muestras de fraudes para aumentar estas muestras en nuestro *dataset* desbalanceado.

Los detalles de cómo se construye esta implementación en Python estan especificados en el código, con comentarios en las líneas de código.

El estudio llevado a cabo por **Optuna** se puede paralelizar de la misma manera que hemos hecho anteriormente, especificando un porcentaje elevado (pero nunca el 100%) de CPUs que puede utilizar para su trabajo. El código necesario es el siguiente:

```
import os
n_cores_mlp = os.cpu_count()
safe_cores_rf = max(1, int(n_cores_mlp * 0.75))
```

Finalmente, obtenemos los mejores parámetros que ha encontrado **Optuna** según nuestras especificaciones, y con estos parámetros optimizados podemos entrenar lo que será el mejor modelo de **Random Forest**. Entonces, mirando la completitud obtenida, sabremos que porcentaje de fraudes totales puede detectar este modelo. Más adelante, en la sección de resultados, comentaremos el resultado obtenido para la completitud y como este nos servirá como referencia para evaluar el resto de modelos. Por último, incluimos en la siguiente lista la combinación de hiperparámetros determinados como los mejores:

- `n_estimators = 78`: nombre total de árboles de decisión que conforman el bosque.
- `max_depth = 16`: profundidad máxima de cada árbol para limitar su complejidad y evitar *overfitting*.
- `min_samples_split = 6`: número mínimo de muestras necesarias para dividir un nodo niterno.
- `min_samples_leaf = 3`: número mínimo de muestras que ha de tener cada hoja del árbol.
- `max_features = sqrt`: número de variables consideradas en cada división, correspondientes a la raíz quadrada del total de características.

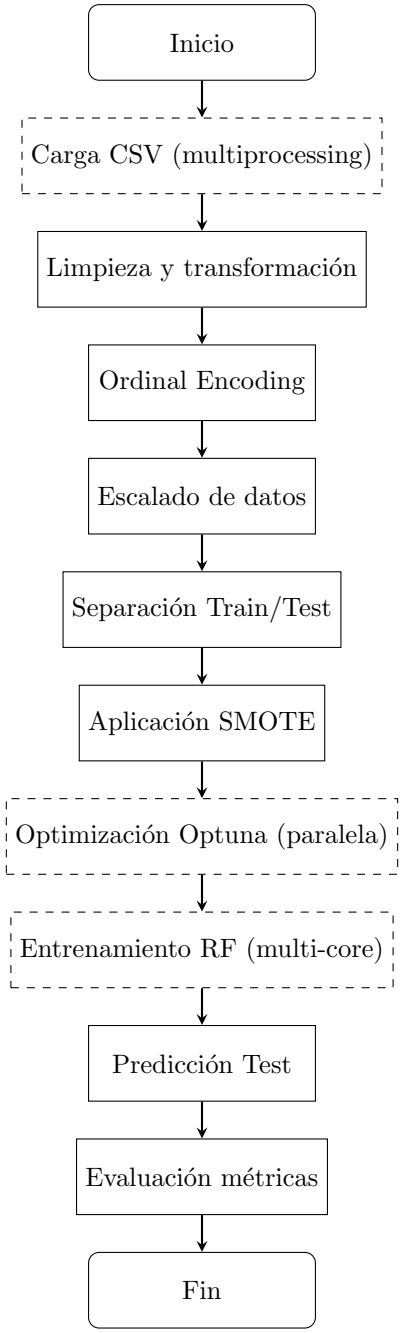

Figure 1: Diagrama de ejecución del modelo Random Forest

3.3.2 Multi-Layer Perceptron (MLP)

El siguiente modelo simple que estudiamos fue el **MLP**, un tipo de red neuronal relativamente simple. Puede tener distinas capas de neuronas, pero como mínimo siempre debe tener una capa de entrada por la cual entran los datos, y una de salida que da la predicción final. En medio, pueden haber capas ocultas que procesan la información a través de ir ajustando los pesos de las conexiones entre neuronas (perceptrones) y funciones de activación. Cada neurona hace una combinación ponderada de la entrada de datos y aplica una función lineal, ajustando sus pesos en cada iteración, cosa que permite al modelo aprender las relaciones complejas entre distintas variables. Así, puede aprender patrones no lineales.

Tal y como hemos hecho con el **Random Forest**, importamos el modelo de **scikit-learn**, así como las métricas, y funciones como el **SMOTE** para poder entrenar el modelo. Además, volvemos a utilizar **Optuna**. El proceso en este caso es muy similar al anterior, así que no vamos a entrar tan en detalle.

Primero de todo, especificamos el porcentaje seguro de CPUs que vamos a utilizar para paralelizar la ejecución. Entonces, creamos la función del **MLP** con todos los hiperparámetros con los cuales va a probar posibles combinaciones para maximizar el recall. Se pueden ver los parámetros escogidos, con explicación, en el código. De igual manera que antes, utilizamos **Optuna** para que encuentra la combinación óptima para maximizar el recall con este modelo. Una vez hemos encontrado la mejor combinación de hiperparámetros, ya tendremos el mejor modelo lista para entrenarlo y hacer las predicciones. Al final, calculamos las métricas ya mencionadas, y incluimos los resultados obtenidos en la sección de resultados de este proyecto.

Los hiperparámetros óptimas son:

- `hidden_layer_sizes = (50,)`: arquitectura interna, una sola capa oculta con 50 neuronas.
- `activation = tanh`: función de activación hiperbólica que introduce no linearidad al modelo.
- `solver = adam`: algorismo de optimización basado en un gradiente descendente adaptativo.
- `alpha` = 1.36×10^{-5}: factor de regularización L2 para reducir el *overfitting*.
- `learning_rate_init` = 1.26×10^{-4}: tasa de aprendizaje inicial del proceso de entrenamiento.
- `batch_size = 256`: número de meustras utilizadas en cada actualización de los pesos internos.

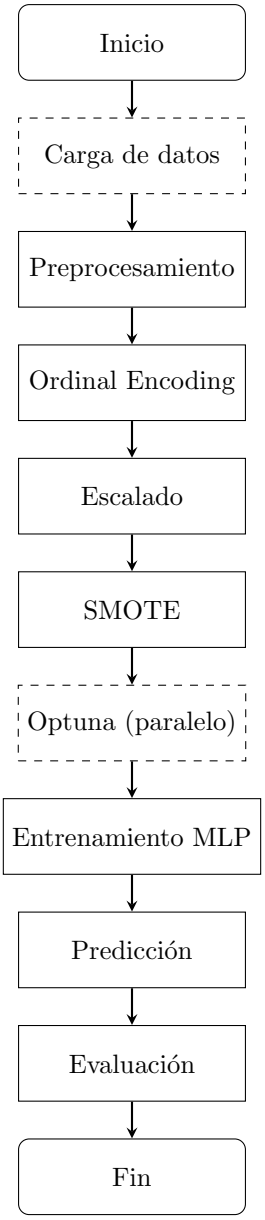

Figure 2: Diagrama de ejecución del modelo MLP

3.4 Autoencoder

El primer modelo que evaluamos para intentar superar los resultados obtenidos con los algoritmos simples anteriores es el **Autoencoder**.

Un **Autoencoder** es un tipo de red neuronal que aprende a comprimir los datos de las transacciones y luego a reconstruirlas a los datos originales a partir de los comprimidos. Funciona de la siguiente manera: primera, codifica los datos de entrada convirtiendolos a un vector de dimensiones reducidas, como si combinara los datos de las transacciones. Esto es la parte del *Encoder*, y se repite varias veces. Por ejemplo, si la transacción de entrada tiene 64 columnas, puede comprimirlo a 48, luego 32, 16 y al final 8. Tendrá la información esencial de las 64 columnas iniciales comprimidas en solo 8. Después viene la parte de la decodificación, el *Decoder*. En esta parte, coge el vector de dimensiones reducidas e intenta reconstruir la versíon original de este. Vuelve a pasar de 8 dimensiones a 64 intentando que la salida sea lo más similar posible a la entrada original.
En este proceso, el modelo aprendre patrones y estructuras clave de los datos sin necesidad de etiquetarlos. Es decir, los datos de entrada son solo transacciones legítimas. Se puede calcular el error de la reconstrucción de salida con la entrada. Por lo tanto, el **Autoencoder** aprenderá los patrones y estructuras de transacciones legítimas, provocando que el error de reconstrucción de fraudes sea mucho mayor

que el de transacciones normales. Entonces, aplicando un valor umbral del error, podemos clasificar las transacciones como fraudes o legítimas.

En resumen:

Entrada → Encoder → Vector reducido → Decoder → Salida reconstruida con las dimensiones originales

Una vez entendido como funciona el **Autoencoder**, hacemos el mismo preprocesamiento de datos que hemos explicado al inicio de esta sección, cuyos detalles se pueden ver en el código de este modelo. En este caso, para el entrenamiento necesitamos más librerías de las que ya hemos usado hasta ahora. Debemos importar la librería de **pytorch** que nos proporcionará las herramientas necesarias para construir la arquitectura de la red neuronal, definir la función de pérdida y optimizar los pesos del modelo durante el entrenamiento. Este modelo tiene un paso más de preprocesamiento de los dates, ya que debemos coger solamente las transacciones normales, sin fraudes, tal y como hemos explicado. Aplicamos también el escalado con el **Robust Scaler** importado desde la librería de **scikit-learn**, proceso que también hemos paralelizado. Luego definimos la clase base para todos los modelos de Pytorch y heredamos **nn.Module**. Para más detalles se puede consultar el código del proyecto. Entonces, definimoso el *Encoder*, con sus parámetros y funciones (de activación, *dropout*, linealidades, etc.). En esta parte, pasaremos de transacciones con más de 70 columnas a reducirlas hasta 8 (o las dimensoines que queramos), capturando los patrones y estructuras clave de estas.
Luego, el *Decoder* se encargará de reconstruir desde 8 dimensions hasta el número original de *features*, intentando reconstruir la transacción original. El modelo, que solo habrá visto transacciones legítimas, cometerá errores notables al intentar reconstruir las fraudulentas. Entonces, si separamos las reconstrucciones con mayor error, obtendremos teóricamente las que son fraudes, ya que el modelo no las habrá visto nunca y seran anómalas para él. El error que calculamos es el Error Cuadrático Medio (MSE por sus siglas en inglés). Para llevar a cabo el entrenamiento, es necesario transformar los vectores de *numpy* a *tensores*.
Para cargar los datos en este modelo de forma más rápida y eficiente utilizamos la paralelización en el método de **DataLoader**, asignando 4 procesos para esta tarea.

Entonces, una vez comprimidas y descomprimidas todas las transacciones del *dataset* en un total de 100 épocas, calculamos el MSE de todas las transacciones. Vamos comparando el MSE medio en las distintas épocas del entrenamiento y, si mejora, seguimos entrenando. No obstante, si en 15 épocas no mejora, se para el entrenamiento.
Finalizado el entrenamiento, evaluamos el *dataset* de tensores de test, codificandolo y decodificandolo como hemos explicado. Guardamos el MSE cometido en cada transacción, teniendo en cuenta que en este *dataset* hay tanto transacciones normales como legítimas, porque es el de test. Así pues, si miramos qué transacciones estan por encima del error de un cierto porcentaje de transaccoines legítimas (estudiamos 90%, 95% y 99%), deberíamos encontrar la gran mayoría de fraudes. Evidentemente, si miramos a partir del 90%, cuyo error umbral es menor que a 95% o 99%, encontraremos más fraudes. Esto queda claramente comprobado con los resultados obtenidos, los cuales comentaremos y evaluaremos en la sección de resultados.

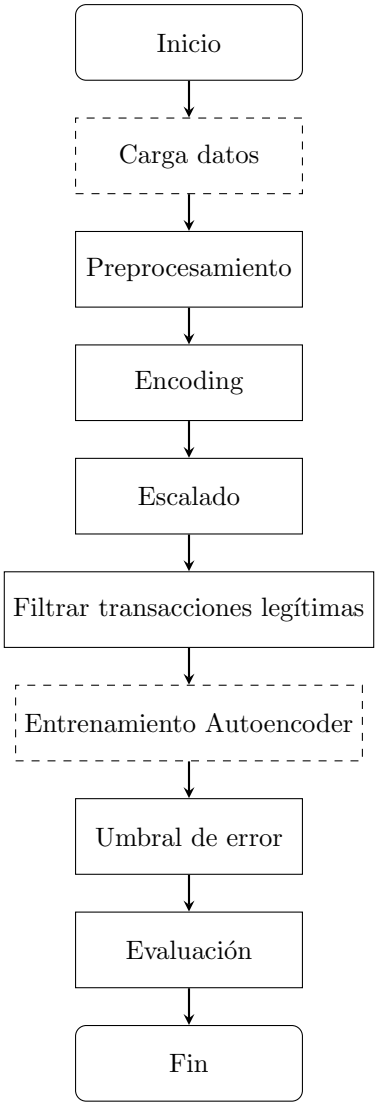

Figure 3: Diagrama de ejecución del Autoencoder

3.5 TabNet: Attentive Interpretable Tabular Learning

En esta sección, la última, estudiamos el modelo TabNet. TabNet es una arquitectura de red neuronal diseñada específicamente para datos tabulares, como los CSV que tenemos en este proyecte. Este método combina la eficiencia de los árboles de decisión, que ya hemos visto que son los más óptimos en nuestro problema, con la potencia de las redes neuronales para aprender patrones y no linearidades.
El TabNet funciona de la siguiente manera: en vez de analizar todos los datos a la vez, lo hace en lotes más pequeños y así puede aprender cuáles son las columnas más importantes en cada paso, como un árbol de decisión. El modelo no toma decisiones de golpe, sinó que mira primero unas variables, saca una conclusión parcial, y luego mira otras. Esto le permite aprender relaciones muy complejas entre datos.
Así pues, en nuestro caso de detección de fraudes donde algunas variables pueden ser mucho más importantes que otras, que son básicamente ruido, este sistema puede proporcionar mucha más precisión. Además, este modelo está optimizado para archivos tabulares gigantes donde algunos modelos tradicionales se pueden quedar cortos.

Con el preprocesamiento ya explicado, utilizamos el One-Hot Encoder (a través del método **.get_dummies()** para transformar las variables categóricas a numéricas. También tenemos que escalar los datos con el Robust Scaler que ya hemos utilizado. Sin embargo, el TabNet requiere de una pequeña muestra de datos para llevar a cabo una evaluación en el entrenamiento, así que usamos el módulo **train_test_split** de **scikit-learn** para obtener una muestra de los datos de entrenamiento como evaluación.

Una vez hecho todo esto, creamos el modelo TabNet con los siguientes hiperparámetros, algunos parecidos a los que ya hemos visto y optimizado:

- `optimizer_fn = torch.optim.Adam`: se escoge el optimizador *Adam* por su buen rendimiento y capacidad de converger rápido.
- `optimizer_params = {lr=2e-2}`: tasa de aprendizaje inicial de 0.02, que determina el tamaño de los pasos durante la actualización de pesos.
- `scheduler_params = {"step_size":10, "gamma":0.9}`: configuración del scheduler que reduce la tasa de aprendizaje cada 10 *epochs*, multiplicándola por 0.9, permitiendo un ajuste progresivo del aprendizaje.
- `scheduler_fn = torch.optim.lr_scheduler.StepLR`: se utiliza *StepLR* como scheduler para disminuir la tasa de aprendizaje de forma escalonada durante el entrenamiento.
- `mask_type = 'entmax'`: tipo de máscara usada por TabNet para enfocar la atención en las características más importantes de los datos.

Una vez definidos, procedemos a entrenar el modelo, que se entrena con la función `fit` utilizando los siguientes parámetros:

- `X_train, y_train`: datos de entrenamiento.
- `eval_set = [(X_val, y_val)]`, `eval_name = ['val']`: conjunto de validación y su nombre para evaluar el modelo durante el entrenamiento.
- `eval_metric = ['auc']`: métrica de evaluación, *AUC*, adecuada para clasificación binaria con clases desbalanceadas.
- `max_epochs = 100`: número máximo de épocas de entrenamiento.
- `patience = 10`: detención temprana si la métrica de validación no mejora durante 10 épocas.
- `batch_size = 1024`, `virtual_batch_size = 128`: tamaño de los lotes y lotes virtuales para normalización.
- `num_workers = 4`: número de trabajadores para cargar los datos en paralelo.
- `drop_last = False`: no descarta el último lote aunque sea más pequeño que el tamaño definido.
- `weights = 1`: no se aplican pesos de clase, ya que TabNet maneja bien el desbalanceo.

Tal y como vemos, para que este proceso sea más rápido utilizamos **num_workers**=4, paralelizandolo. Luego, ya podemos hacer las predicciones sobre el *dataset* de test. Finalmente, evaluamos los resultados con las métricas ya vistas. En este caso, las predicciones se han hecho asignando una cierta probabilidad a que sea fraude o legítima, con el método **.predict_proba**, no asignando solo una posibilidad booleana (0 o 1). Por ejemplo, puede decir que esta un 70% seguro de que una transacción sea fraude y 30% que no. Entonces, podemos definir un threshold que solo permita clasificar como fraude transacciones de las que esté un cierto porcentaje seguro de que es fraude o al revés.
Finalmente, calculamos las métricas usando la matriz de confusión, que hemos importando anteriormente. La matriz de confusión nos permite saber el número de negativos verdaderos (TN), falsos positivos (FP), falsos negativos (FN) y positivos verdaderos (TP). Al final, con el TabNet, hemos conseguido un recall del 94%, clasificando correctamente 2023 de 2145, todo un éxito y superando los anteriores modelos.
Así pues, vemos que combinando las ventajas de una red neuronal con árboles de decisión, podemos obtener mejores resultados que los modelos por separado, con el modelo TabNet.

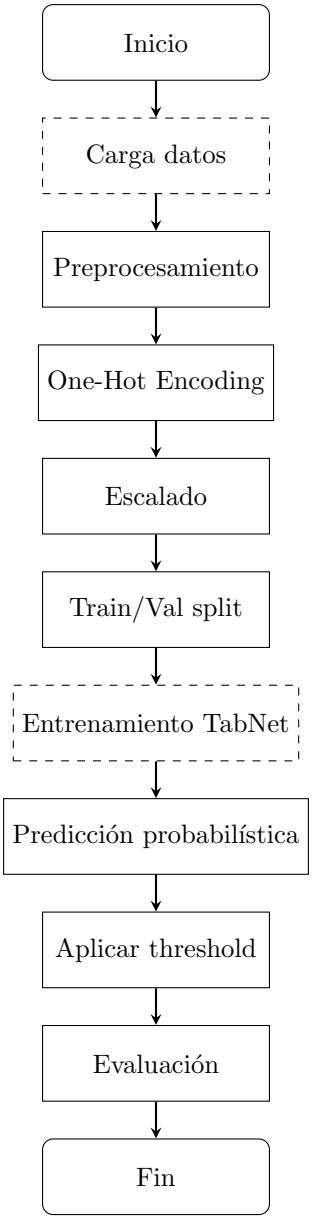

Figure 4: Diagrama de ejecución del modelo TabNet

4 Resultados

Para evaluar los resultados de los modelos que hemos implementado (Random Forest, Red Neuronal (MLP), Autoencoder y TabNet), debemos escoger un conjunto de métricas que nos ayuden a definir tanto la calidad de predicción que tienen cada uno de nuestros modelos como la eficiencia computacional de estos. Teniendo en cuenta que estamos trabajando en la detección de fraudes, donde los conjuntos de datos son muy desbalanceados, no podemos basarnos unícamente en métricas más globales como la accuracy, sino que debemos evaluar la capacidad del modelo para detectar la mayor cantidad de fraudes.

Por un lado tenemos las métricas que mediran el rendimiento de predicción de cada uno de nuestros modelos. En nuestro caso hemos escogido las siguientes métricas:

1. **Recall (sensibilidad):** esta es nuestra métrica de mayor importancia para el proyecto. En el contexto del fraude, es mucho más importante ser capaz de detectar todos los fraudes (es decir, aumentar al máximo el número de verdaderos positivos) aunque eso implique aumentar el número de falsas alarmas (es decir, puede haber un aumento de falsos positivos como consecuencia). Por este motivo, la métrica del recall, la cual mide cuántos fraudes reales fuimos capaces de detectar

respecto el total de fraudes, nos ayuda a conseguir este objetivo. Como podemos ver en la siguiente ecuación (1), el recall se calcula diviendo el nombre de transacciones fraudulentas detectadas (TP) respecto al número total de fraudes detectados y no detectados (TP + FN).

$$Recall = \frac{TP}{TP + FN} \tag{1}$$

2. **F1_score y precision:** estas métricas no son tan prioritarias en nuestro proyecto, pero aun así queremos calcularlas para evaluar el rendimiento más global de nuestro proyecto. La precision nos ayuda a medir la relación que tenemos entre las predicciones de fraudes correctas del total de detecciones marcadas como fraude que hemos encontrado. En la ecuación (2) podemos observar que es el ratio de fraudes detectados correctamente (TP) respecto al total de predicciones marcadas como fraude, tanto acertadas como erróneas (TP y FP). Por otro lado, la f1_score es un equilibrio entre la precision y el recall y nos ayuda a entender como de balanceado es nuestro modelo.

$$Precision = \frac{TP}{TP + FP} \tag{2}$$

$$F1\text{-}Score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{3}$$

3. **Matriz de confusión:** esta representación nos permite identificar de manera visual como clasifica nuestro modelo cada una de las predicciones. En nuestro caso, la hemos incluido en formato de tablas donde podemos ver el número de fraudes detectados, fraudes perdidos y falsas alarmas respecto al total de fraudes de nuestro dataset.

Por otro lado tenemos las métricas que se encargar de evaluar la eficiencia computacional de nuestro proyecto. Una parte importante de nuestro trabajo también se trata de intentar implementar técnicas de paralelización como las que hemos visto en clase con el objetivo de mejorar el rendimiento y velocidad de nuestro código. Para ello, mediremos su desempeño con las siguientes métricas:

1. **Tiempo de ejecución y Speedup:** estas métricas nos servirán para cuantificar la mejora que representan nuestras técnicas de paralelización. Mediremos el tiempo de ejecución de nuestro código de manera secuencial y después de manera paralelizada y los compararemos mediante el cálculo del Speedup. Como hemos visto en clase, el Speedup se calcula dividiendo el tiempo que tarda en ejecutarse el código en secuencial respecto al que tarda en hacerlo en paralelo, como se muestra en la ecuación (4).

$$S = \frac{T_{secuencial}}{T_{paralelo}} \tag{4}$$

2. **Ancho de banda:** nos perite evaluar la capacidad que tiene nuestro código para aprovechar el hardware disponible. En nuestro caso, el ancho de banda nos permitirá medir la tasa de transferencia de datos durante el entrenamiento, de manera que podremos evaluar como mejora la cantidad de muestras por segundo que carga utilizando las técnicas de paralelización.

A continuación, vamos a analizar los resultados específicos obtenidos para cada uno de los modelos desarrollados bajo estos criterios.

4.1 Random Forest & MLP

Como hemos comentado anteriormente, los dos modelos más simples con los que empezamos a elaborar este proyecto fueron el Random Forest y la red neuronal MLP. En estos casos nos limitamos a obtener el valor de recall (o completitud) más alto posible, con el objetivo de sentar una referencia de resultados para el resto de modelos. Este paso es muy importante, ya que nos permite saber los resultados que podemos obtener sin la necesidad de utilizar modelos más complejos que quizá no suponen una mejora significativa respecto a estos modelos más sencillos.

El modelo del Random Forest consiguió un valor del recall de :

$$\text{Recall}_{RF} = 0.853$$

Esto indica una buena capacidad del modelo para identificar correctamente que transacciones son fraudes, ya que detecta el 85.3% de ellas, aproximadamente 1830 de 2145. El resultado era esperable, ya que el Random Forest es un modelo que trabaja bien con datasets desbalanceados, ya que al trabajar con muchos árboles de decisión puede detectar la clase minoritaria. Además, puede captar relaciones no lineales como los fraudes, que no suelen seguir patrones lineales de comportamiento. También es un modelo robusto al ruido al basarse en muchas votaciones de distintos árboles. Este umbral a superar presenta un reto, ya que es bastante alto.

Por otro lado, el resultado obtenido para el MLP fue:

$$\text{Recall}_{MLP} = 0.709$$

Aunque el recall obtenido es inferior al del modelo anterior, sigue siendo un buen modelo con un buen equilibrio, lo cual también eran esperable. Como el dataset está bastante desbalanceado, hay pocos fraudes y el modelo MLP tiende a aprender los patrones de la clase mayoritaria. El Random Forest, en cambio, es más capaz de captar patrones de la clase minoritaria. La red neuronal necesita más datos para poder aprender mejor. Además, es sensible al ruido y a la escala de los datos, cosa que también le puede perjudicar si hay muchas anomalías. No obstante, cambiando el umbral, o threshold, con el que la red neuronal clasifica las transacciones podemos obtener una mayor completitud a cambio de un aumento en el número de falsos positivos. Así pues, aunque el MLP es un buen modelo, no es tan eficiente como un modelo basado en árboles de decisión, que para este problema en concreto es mejor.

En conclusión, ambos modelos, aunque son sencillos, ofrecen un buen valor de la completitud que nos planteamos superar a la vez que mantenemos un buen equilibrio con el número de falsos positivos de nuestros modelos. Además, hemos podido comprovar como algoritmos basados en árboles de decisión ofrecen un mejor rendimiento en problemas como el nuestro.

4.2 Autoenconder

En esta sección vamos a evaluar los resultados obtenidos para el modelo del Autoencoder. Como hemos comentado previamente, para el autoencoder hemos utilizados diferentes umbrales de error, los cuales son 99%, 95% y 90%, de más restrictivo a menos respectivamente. Podemos observar los resultados obtenidos en las siguientes tablas:

Estrategia	**Threshold**	**TP**	**FP**	**Recall**	**Precision**	**F1-Score**
P99 (Muy estricta)	26.83	11	5,536	0.0051	0.0020	0.0029
P95 (Intermedia)	2.33	97	27,679	0.0452	0.0035	0.0065
P90 (Seleccionada)	**0.35**	**1,121**	**55,358**	**0.5226**	**0.0198**	**0.0382**

Table 1: Comparativa de los diferentes umbrales de error del Autoencoder

Tal y como vemos, para el error por encima del 90% de transacciones (es decir, marcar como fraude solo el 10% de transacciones con mayor error de reconstrucción) obtenemos la detección de fraudes más alta, con el 52.26% de ellos encontrados. También podemos observar como esto conlleva una Precisión pésima de tan solo el 2%. Esto demuestra la dificultad de separar anomalías sutiles utilizando únicamente el error de reconstrucción sin etiquetas supervisadas. En los casos con umbrales más altos, vemos como los resultados son peores, los cuáles son especialmente notables para el caso del 99%, motivo por los cuales deberíamos descartar esas estrategias con umbrales tan altos. Centrándonos en los resultados específicos de la estrategia del 90%, obtenemos estos resultados:

Clase	**Precision**	**Recall**	**F1-Score**	**Muestras**
0 (Legítima)	1.00	0.90	0.95	553,574
1 (Fraude)	0.02	0.52	0.04	2,145

Table 2: Informe completo para la estrategia 90%

En conclusión, vemos que el recall del autoencoder no es demasiado bueno, solo es del 52.26%. Tal y como hemos visto antes con el MLP, esto se debe a que el dataset está muy desbalanceado pero, además,

Métrica	Valor
True Positives (TP) - Fraudes detectados	1,121
False Positives (FP) - Falsas alarmas	55,358
False Negatives (FN) - Fraudes perdidos	1,024
Total de casos de Fraude	2,145
Recall (Sensibilidad)	**0.5226**

Table 3: Métricas de completas de detección para el Autoencoder con estrategia 90%)

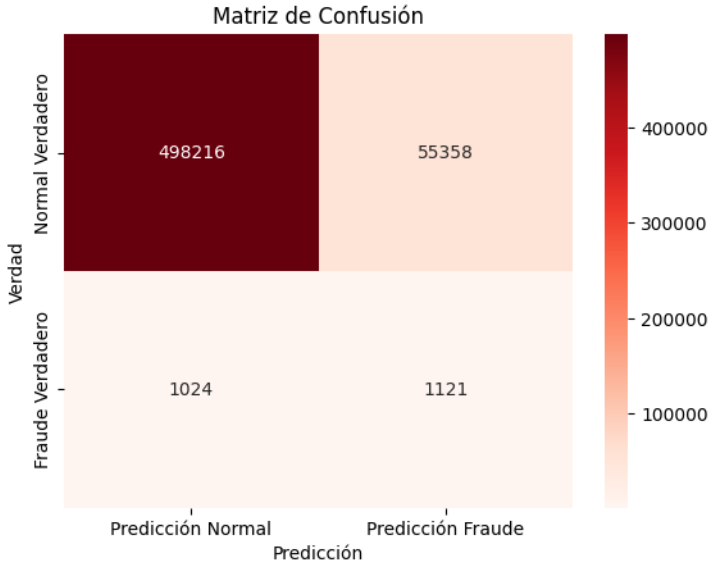

Figure 5: Matriz de Confusión que ilustra los resultados del Autoencoder.

el error de las transacciones fraudulentas no suele ser lo suficientemente alto como para superar el umbral y pasan desapercibidos. El Autoencoder, que es un método no supervisado, funciona bien para anomalías claras y regulares, pero en bases de datos muy desbalanceadas y con fraudes sutiles, es menos efectivo que modelos como los vistos al principio. Por este motivo, en este tipo de contextos deberíamos buscar otro tipo de modelos que sse adecuén más a las características de nuestro dataset.

4.3 TabNet

Para este último modelo, hicimos un análisi similar al del modelo anterior, por tal de poder comparar con mayor certeza los resultados obtenidos, los cuáles podemos observar en las siguientes tablas:

Clase	Precision	Recall	F1-Score	Muestras medidas
0 (Legítima)	1.00	0.94	0.97	553,574
1 (Fraude)	0.06	0.94	0.11	2,145

Table 4: Informe de Clasificación (Threshold > 0.3)

Como podemos comprobar con los resultados que hemos obtenido, hemos conseguido optimizar el valor de Recall a un valor incluso más grande que el que habíamos encontrado para el Random Forest inicial y, por un margen aún mayor, mayor también que el resto de modelos. Tal y como hemos explicado en el apartado de implementación, el TabNet asigna valores de probabilidad de fraude a cada transacción. Cuánto mayor sea este valor, mayor es la probabilidad de que este sea fraude según nuestra predicción. En este parte es dónde entra en juega el valor de threshold que asignamos, que no deja de ser el punto que marcamos como umbral a partir del cual clasificamos a todas las transacciones con un valor superior como fraudulentas. Si incrementamos este valor, reducimos la cantidad de fraudes detectados (tanto verídicos como falsos). Es importante por eso encontrar un buen equilibrio en nuestro valor de threshold. En el

Métrica	Valor
True Positives (TP) - Fraudes detectados	2,023
False Positives (FP) - Falsas alarmas	34,093
False Negatives (FN) - Fraudes perdidos	122
Total de casos de Fraude	2,145
Recall (Sensibilidad)	**0.9431**

Table 5: Elementos de la Matriz de Confusión (Threshold > 0.3)

Clase	Precision	Recall	F1-Score	Muestras medidas
0 (Legítima)	1.00	0.97	0.99	553,574
1 (Fraude)	0.11	0.92	0.20	2,145

Table 6: Informe de Clasificación (Threshold > 0.5)

Métrica	Valor
True Positives (TP) - Fraudes detectados	1,968
False Positives (FP) - Falsas alarmas	15,725
False Negatives (FN) - Fraudes perdidos	177
Total de casos de Fraude	2,145
Recall (Sensibilidad)	**0.9175**

Table 7: Elementos de la Matriz de Confusión (Threshold > 0.5)

caso de querer obtener el menor número de falsos negativos (es decir fraudes no detectados) debemos reducir el threshold. No obstante, a su vez estamos aumentado la cantidad de falsos positivos detectados.

Para el caso del threshold de valor 0.3 obtenemos el mayor recall, del 94,31%. En el caso del threshold de valor 0.5, podemos observar como el número de falsos positivos se reduce considerablemente, pasando de 34,093 a 15,725. No obstante, dejamos de detectar 55 casos de fraude. De esta manera, podemos entender claramente el precio a pagar para detectar un mayor número de fraudes.

En conclusión, el modelo TabNet ha demostrado ser una solución robusta para el problema de desbalances de clases, consiguiendo detectar un gran número de fraudes y consiguiendo un recall significativemente mejor. Por desgracia, esta mejora ha tenido un coste en el número de falsos positivos detectados.

4.4 Resultados de eficiencia

Una vez evaluados los resultados de las predicciones, debemos evaluar también las mejoras que han supuesto nuestros implimentaciones de paralelización. Para ello, corrimos el código primero en secuencial y posteriormente en paralelo, y medimos el tiempo que tardó en cada ocasión. La mayoría de las celdas de nuestros códigos son casi immediatas. No obstante, las partes que requieren un mayor tiempo de ejecución (como la optimización de hiperparámetros) si que necesitan mucho tiempo, por lo que implementar medidas de paralelización puede resultar en una gran ventaja.

Los resultados que obtuvimos fueron los siguientes:

Modelo	Tiempo Secuencial	Tiempo Paralelo	Speedup (S)
Random Forest	224m 57s	40m 52s	**5.50**
MLP	217m 27s	60m 55s	**3.57**
Autoencoder	88m 03s	97m 25s	**0.90**
TabNet	28m 48s	17m 13s	**1.67**

Table 8: Tiempos de Ejecución y Speedup por modelo

Como podemos observar en la tabla anterior, en la mayoría de modelos hemos conseguido una mejora

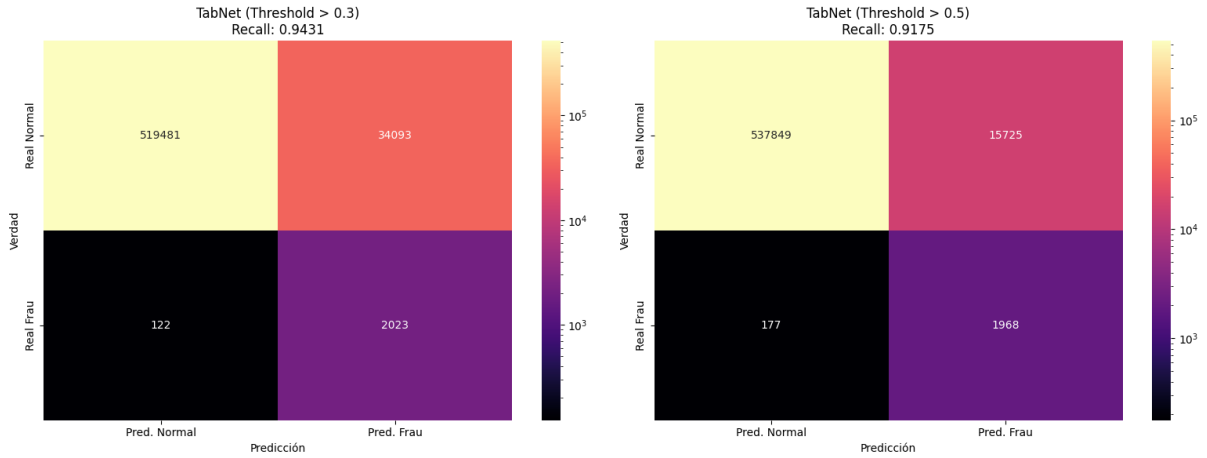

Figure 6: Matriz de Confusión que ilustra los resultados del TabNet para ambos casos de threshold

con las implementaciones. El mejor resultado lo encontramos en el Random Forest, donde el Speedup es de un valor superior a 5. Esto es debido a que la optimización de hiperparámetros tiene un coste computacional alto, y al hacerlo en secuencial gastamos mucho tiempo. No obstante, es un proceso fácilmente paralelizable, ya que cada árbol es independiente de otro y esto nos permite dividir procesos sin demasiada dificultad. Uno de los factores que causan esta mejora es utilizar un mayor número de CPUs, lo cual nos permite distribuir la construcción de los estimadores.

Otros modelos con un buen resultado han sido el TabNet y la red neuronal MLP. En el caso del TabNet, cambiar la paralelización en la carga de datos con el num_workers ha sido de gran ayuda. Sin embargo, la mejora no es tan notable porque el proceso en secuencial no era demasiado grande, y por lo tanto el margen de mejora no era tan elevado.

Por último, el único caso en el que hemos visto un empeoramiento en el rendimiento ha sido el Autoencoder. Este problema puede estar relacionado con el overhead de la paralelización que vimos en otras prácticas de clase, donde orquestar todos los procesos genera un coste mayor que la mejora que otorga la paralelización en si. Quizá se pudiera mejorar estos resultados si implementaramos un algoritmo más complejo o técnicas específicas para este tipo de modelos.

5 Limitaciones del trabajo y posibles mejoras

A pesar de los resultados satisfactorios obtenidos, sobretodo en el modelo TabNet con un recall de más del 94%, el trabajo tiene algunas limitaciones. Estas limitaciones provienen mayoritariamente de las características de los propios algoritmos seleccionados así como de las técnicas utilizadas.

Una posible mejora relacionada con el hardware sería disponer de unidades de procesamiento potentes (GPU) para el entrenamiento de los modelos. Aunque el coste computacional ha disminuido en la mayoría de modelos utilizando los métodos de paralelización, el entrenamiento de muchos de los modelos han sido muy extensos, hecho que no nos permetía una cómoda adaptación ni ejecutar muchas veces el código para hacer pruebas. El uso de entornos con GPUs nos permetiría reducir el tiempo de ejecución significativamente, así como experimentar con arquitecturas más complejas, profundas y con un mayor número de epochs que quizá nos darían mejores resultados.

Otro de los desafíos que hemos encontrado ha sido el modelo Autoencoder, el cual no ha sido muy eficaz, con un recall de tan solo el 52,26%. El principal problema de este modelo ha sido que los fradues en este dataset no siempre presentan un error de reconstrucción significativamente mayor que las transacciones legítimas complejas. Además las técnicas de paralelización tampoco han conseguido implementarse de manera eficaz, haciendo que el overhead de la gestión de procesos fuera mayor a la mejora que aportaba. Para solucionar esto, podríamos analizar la posibilidad de utilizar otros algoritmos similares al Autoencoder pero con algunas diferencias que le permiten trabajar mejor con este tipo de datasets. También podríamos intentar implementar técnicas de paralelización más complejas que permitan superar al overhead y suponer una mejora real en el tiempo de ejecución.

Una posible alternativa relacionada con el Autoencoder sería utilizar los errores de reconstrucción que calcula nuestro modelo como una nueva columna de features de nuestro dataset con la cual alimentar un segundo modelo (como por ejemplo el Random Forest). De esta manera, el Random Forest tendría acceso a los errores calculados por el Autoencoder y a su vez a todo el resto de columnas del dataset, permitiéndole así encontrar nuevos patrones más complejos para obtener unos resultados aún mejores.

Por último, cabe destacar también el gran número de falsos positivos obtenidos. Aunque hayamos conseguido unos buenos valores de recall en algunos modelos como el TabNet, la cantidad de falsos positivos detectados no permetirían aplicar estos modelos en el mundo real, ya que la cantidad de falsas alarmas envíadas al cliente serían demasiado elevadas. Una ruta de mejora en este caso sería combinar varios modelos, como por ejemplo un primer TabNet que hiciera una primera criba y pasará los casos sospechosos por un Random Forest secundario para filtrar falsos positivos, mejorando así la precisión sin sacrificar el recall.

6 Conclusiones

El objetivo principal de este proyecto era diseñar un sistema eficiente para la detección de fraude bancario capaz de operar sobre conjuntos de datos altamente desbalanceados. Tras haber implementaod y evaluado cuatro modelos distintos (Random Forest, MLP, Autoencoder y TabNet), podemos extraer las siguientes conclusiones:

1. **Superioridad del modelo TabNet:** el modelo TabNet ha resultado ser la solución más robusta, alcanzando un Recall del 94.31% y superando significativamente a los modelos clásicos como Random Forest (85.3%) y MLP (70.9%). Esto confirma la idea de que las arquitecturas como la del TabNet que combinan la interpretabilidad de los árboles de decisión con la capacidad de representación de las redes neuronales son idóneas para datos complejos.
2. **Eficiencia de la paralelización:** se ha podido demostrar que la paralelización es una herramienta potente que permite reducir los costes de computación, pero dependiendo del contexto y manera en que se utilizan. Hemos logrado un gran Speedup en algunos modelos como en el Random Forest (Speedup de 5.50), validando así el uso de Python multiprocessing para tareas independientes como lo son en este caso cada uno de los árboles. Sin embargo, el fracaso del caso del Autoencoder (Speedup 0.90) evidencia que la sobrecarga de gestión de procesos puede ser contraproducente en modelos donde el cuello de botella no está siendo gestionado adecuadamente, como ya vimos en algunas secciones de las prácticas de paralelización hechas en clase.
3. **Las consecuencias de un recall muy alto:** el análisis de los resultados evidencia el inevitable precio a pagar en la detección de fraudes: maximizar la seguridad (Recall) implica aceptar un mayor número de falsas alarmas (Falsos Positivos). La elección del umbral de decisión (como el visto en el threshold de 0.3 o 0.5 en TabNet) permite ajustar este equilibrio según las necesidades del negocio, demostrando que no existe una configuración "perfecta", sino configuraciones adecuadas a distintas estrategias de riesgo.

En definitiva, el proyecto ha logrado desarrollar una solución funcional y optimizada que no solo detecta la gran mayoría de los casos de fraude, sino que también hace un uso eficiente de los recursos computacionales disponibles mediante técnicas de programación paralela.

7 Contenerización y despliegue con Docker y Hugging Face

Para este proyecto, hemos decidido añadir una parte extra final donde hemos hecho una aplicación web interactiva en **Hubbing Face** para poder evaluar más fácilmente nuevas transacciones sobre el modelo ya entrenado. Para ello, hemos guardado los pesos del TabNet (con threshold de 0.5), el mejor modelo, para poder hacer predicciones nuevas sin tener que entrenar el modelo desde 0 otra vez.

En un nuevo *Notebook* del modelo, exactamente igual que el código mostrado, hemos guardado los pesos y la arquitectura del modelo TabNet utilizando el método $.save_model, quegenerael$

```
clf.save_model('tabnet_model')
```

y también guardamos el escalado para que las nuevas features tengan el mismo escalado.

```
joblib.dump(scaler, 'scaler.pkl')
```

Guardamos la lista de columnas para mantener la estructura de 72 features:

```
model_columns = list(train_data.drop(columns='is_fraud').columns)
joblib.dump(model_columns, 'model_columns.pkl')
```

Entonces, con los archivos generados por estos métodos en Python, creamos una nueva carpeta llamada $\mathbf{hf}_{d}eploy, dondeincluimoslossiguientesarchivos$:

tabnet_model.zip: fichero que contiene los pesos y parámetros del modelo entrenado.

scaler.pkl: el RobutScaler que normaliza y escala los datos de entrada.

model_columns.pkl: lista que garantiza que los datos de entrada se transformen exactamente en el formato que el modelo espera y sobre el que se ha entrenado.

app.py: código Python que utiliza Streamlit para crear la interífice de la web y ejecuta la lógica de predicción.

requirements.txt: archivo de texto con las librerías que el servidor de Hugging Face tiene que instalar en el container (*pandas, torch, pytorch-tabnet, etc.*.

Dockerfile: archivo que le dice a Hugging Face cómo tiene que crear el contenedor (como una máquina virtual) en el que se ejecuta el código y la aplicación.

Para publicar la aplicación, hemos utilizado **Docker**, ya que llevamos un tiempo aprendiendo a utilizarlo y hemos creído que esto era una buena oportunidad. Docker permite que la aplicación que hemos creado en nuestro ordenador funcione de igual manera en el servidor de Hugging Face, "contenerizando" el código y evitando errores de diferentes versiones o configuración. Hugging Face permite crear un espacio que detecta el Dockerfile creado, construye el sistema operativo virtual, instala las depedencias y librerías especificadas en el archivo y el .txt y ejecuta la aplicación creada con Streamlit.

El link a la aplicación es el siguiente:
https://huggingface.co/spaces/aandreeeuu/Fraud-Detector-Specialist

En resumen, hemos empaquetado el modelo con el .zip, el escalador .pkl y la web .py dentro de un contenedor, Docker, y lo hemos subido a la nube porque cualquiera que desee analizar una transacción pueda hacerlo, utilizando el modelo que hemos entrenado anteriormente.