

AAGE - Práctica 2: Aprendizaje federado y continuo

Grupo 2: Andrés Lires Saborido, Ángel Vilariño García

Curso 2025-2026

Parte I: Aprendizaje Federado con Flower

Implementación

Para llevar a cabo los ejercicios de la Parte I de la práctica, se ha partido de los archivos utilizados en el seminario de Aprendizaje Federado con Flower y PyTorch.

A partir de estos archivos se han realizado las siguientes modificaciones, por orden:

1. Establecido el número de clientes en 10.
2. Adaptada la función de transformación de las imágenes del archivo `task.py` según las estadísticas concretas del dataset FashionMNIST.
3. Editado el particionado para utilizar un particionado basado en distribución Dirichlet con $\alpha \leq 0.1$ en la función `load_data` del archivo `task.py`.
4. Creado un nuevo archivo `histogramas.py` para generar un histograma que muestre la distribución de clases en cada cliente.

El histograma resultante se puede ver en la Figura 1. En la entrega se pueden encontrar histogramas individuales para cada cliente.

5. Construida una nueva clase `MLPSimple` en el archivo `task.py`, que define un modelo de red neuronal simple.
6. Implementados los métodos FedAvg y FedProx en el archivo `server_app.py`.
 - Para el caso de FedProx, se ha considerado un valor base de $\mu = 0.01$ y se ha adaptado el archivo `pyproject.toml` para despues poder experimentar con esta variable (*Paper de referencia*).

7. Creada una nueva función `plot_metrics` en el archivo `metrics.py` para generar gráficas comparativas de las métricas obtenidas en las distintas experimentaciones (que se habían almacenado en formato CSV en la carpeta `metrics`).

A continuación, se han ejecutado los entrenamientos siguiendo ambas estrategias (FedAvg y FedProx).

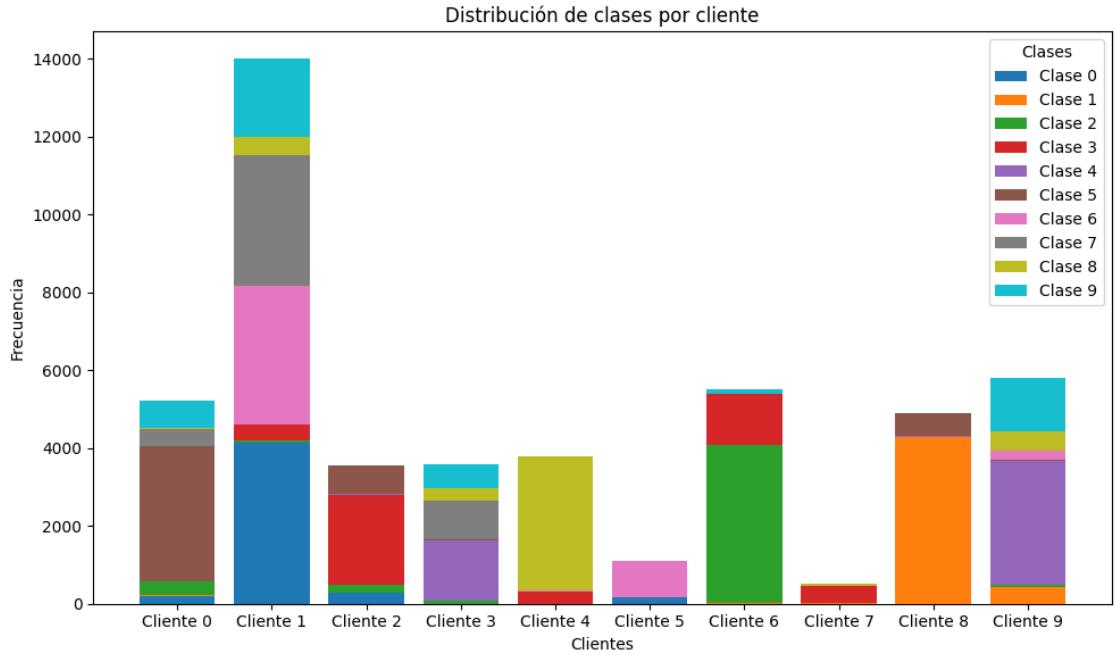


Figure 1: Histograma de distribución de clases entre clientes

8. El apartado 2.4 ha requerido la creación de un nuevo modelo CNN, el cual se ha definido en el archivo `task.py` bajo la clase `CNNModel`.

Se ha entrenado este modelo utilizando tanto FedAvg como FedProx, y se han recogido las métricas correspondientes para su posterior análisis.

Nota:

Tal y como se menciona en el *README*, se ha llevado a cabo un proceso importante de optimización de los códigos de manera que no sea necesario adaptar los scripts principales para cada experimento.

Esto ha conllevado la creación de diferentes archivos `.toml` para gestionar la ejecución de las dos estrategias de aprendizaje federado, así como, más adelante, para variar el modelo utilizado o los hiperparámetros propios del aprendizaje federado.

Experimentación

Los experimentos base realizados son los siguientes:

- **Experimento 1:** Modelo MLPSimple con FedAvg y distribución Dirichlet $\alpha = 0.1$.
- **Experimento 2:** Modelo MLPSimple con FedProx ($\mu = 0.01$) y distribución Dirichlet $\alpha = 0.1$.

A continuación, se ha pasado a utilizar redes neuronales convolucionales (CNN). La red neuronal final utilizada se ha definido en la clase `CNNModel` del archivo `task.py`. Se han probado diferentes redes para llegar a esta arquitectura final,

cambiando el número de capas convolucionales, añadiendo capas de *dropout* o de *batch normalization*. Finalmente se han realizado los siguientes experimentos con el modelo final:

- **Experimento 3:** Modelo CNN con FedAvg y distribución Dirichlet $\alpha = 0.1$.
- **Experimento 4:** Modelo CNN con FedProx ($\mu = 0.01$) y distribución Dirichlet $\alpha = 0.1$.

Las gráficas comparativas de precisión y pérdida para los cuatro experimentos se pueden ver en las Figuras 2 y 3 respectivamente. El modelo que alcanza una mejor precisión es el Experimento 3 (CNN con FedAvg), alcanzando una precisión final del **83.69%**.

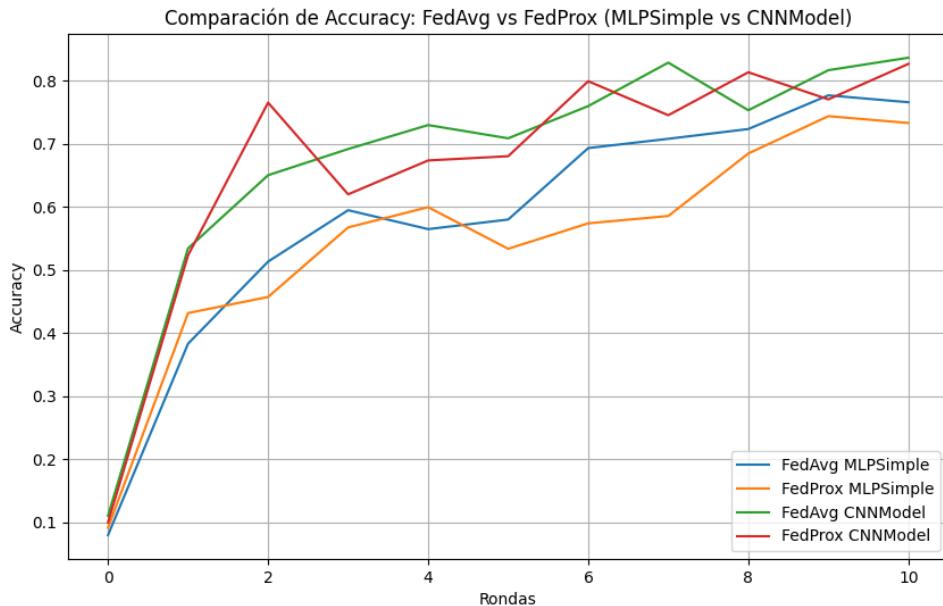


Figure 2: Comparativa de precisión entre los distintos experimentos

Para responder a las preguntas de análisis, se han realizado experimentos adicionales variando los hiperparámetros propios del aprendizaje federado. Estos experimentos adicionales se han gestionado mediante archivos `.toml` específicos para cada configuración.

Los resultados de estas experimentaciones adicionales se presentan en el apartado siguiente.

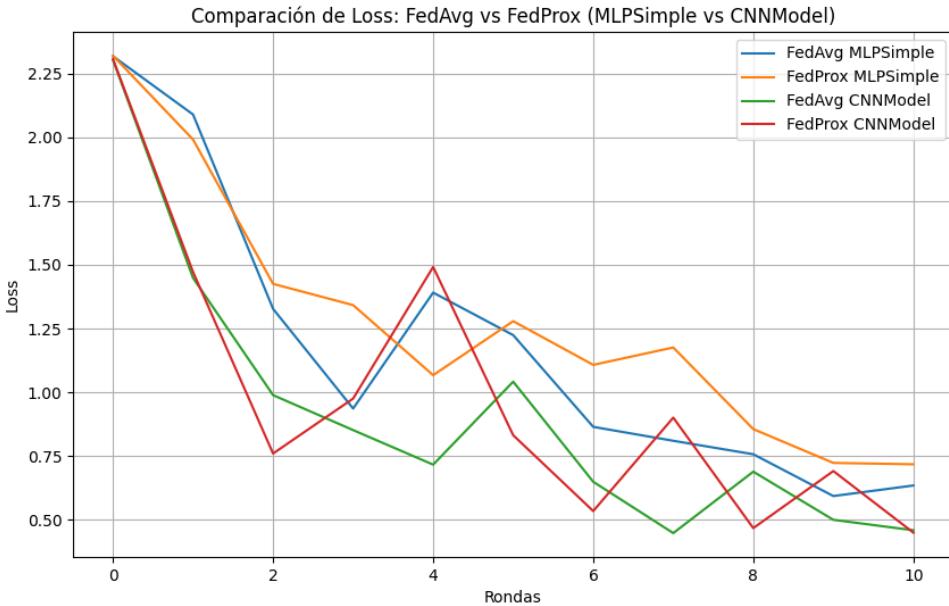


Figure 3: Comparativa de pérdida entre los distintos experimentos

Preguntas de análisis

1. Diferencias observadas entre los métodos. Explicar por qué aparecen.

Como se puede observar en las Figuras 2 y 3, FedAvg tiende a obtener mejores resultados en términos de precisión y pérdida en comparación con FedProx en los experimentos realizados, debido a la naturaleza de ambos métodos. FedAvg promedia los modelos locales sin restricciones adicionales, lo que permite una mayor flexibilidad en la actualización del modelo global, especialmente efectivo cuando los datos entre clientes son relativamente homogéneos o IID. En contraste, FedProx introduce un término de penalización ($\mu = 0.01$) que limita la desviación de los modelos locales respecto al modelo global, lo cual es útil en entornos con datos muy heterogéneos o No-IID para evitar divergencias.

En este caso, al ser los datos más uniformes (aproximadamente 3 o 4 clases de 10 por cliente) o al configurarse un μ demasiado restrictivo, se ha limitado la capacidad de los modelos locales para adaptarse a sus datos específicos, resultando en un rendimiento ligeramente inferior del método FedProx tanto en MLP y CNN.

2. Analizar el impacto de hiperparámetros propios de FL: número de épocas locales, proporción de clientes seleccionados por ronda, etc.

En el aprendizaje federado, más allá de los que existen en el entrenamiento clásico, existen hiperparámetros específicos que influyen en el rendimiento del modelo. Entre ellos, podemos destacar el número de épocas locales, que determina cuántas veces cada cliente entrena su modelo local antes de enviar los parámetros al servidor o la proporción de clientes seleccionados por ronda.

Para responder a esta pregunta, se ha decidido partir del Experimento 3 (CNN con FedAvg y distribución Dirichlet $\alpha = 0.1$) ya que fue el que obtuvo mejores resultados en la experimentación inicial. A partir de este experimento base, se han realizado las siguientes variaciones:

- **local-epochs:** Se han probado valores de 1 (ejemplo base), 3 y 5. Como se puede observar en la Figura 4, aumentar el número de épocas locales no mejora la precisión del modelo global. De hecho, entrenar más épocas locales puede llevar a un sobreajuste en los datos locales de cada cliente (algo observable en la figura 5), lo que resulta en una menor capacidad de generalización cuando se combinan los modelos locales en el servidor. En este caso, el mejor rendimiento se obtiene con 1 época local, alcanzando una precisión cercana al 80%, mientras que con 3 y 5 épocas locales la precisión disminuye ligeramente, situándose alrededor del 75%.

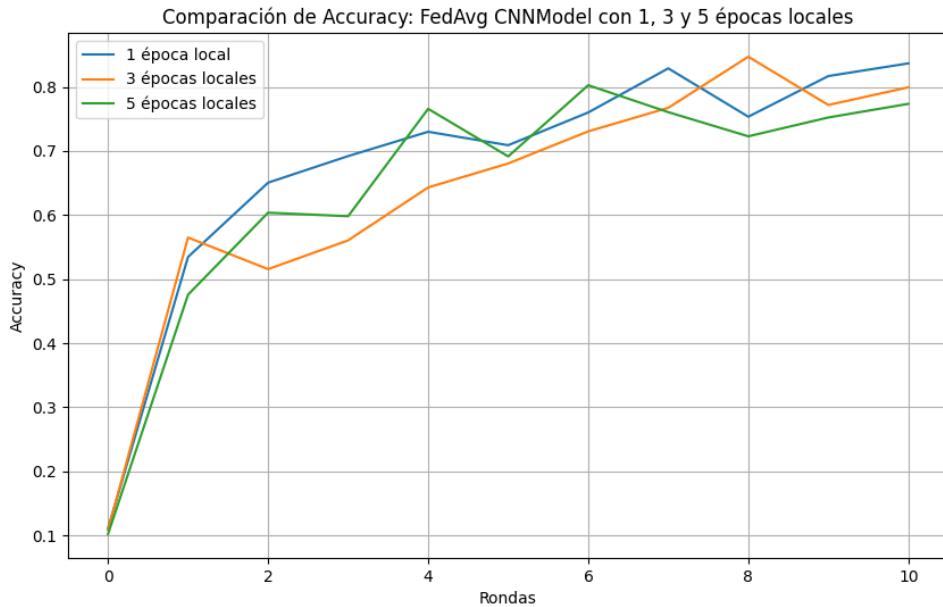


Figure 4: Comparativa de precisión variando local-epochs

- **fraction-train:** Se han probado valores de 0.5 (ejemplo base), 0.1 y 0.9. La Figura 6 muestra que seleccionar una fracción muy baja de clientes (0.1) por ronda puede llevar a una representación insuficiente de la diversidad de datos, afectando negativamente la precisión del modelo global (en torno a 50%). Por otro lado, seleccionar una fracción muy alta (0.9) puede mejorar la representatividad pero a costa de un mayor costo computacional y de comunicación, con una mejora significativa en la precisión (> 80%), obteniendo una prácticamente igual que usando 0.5.

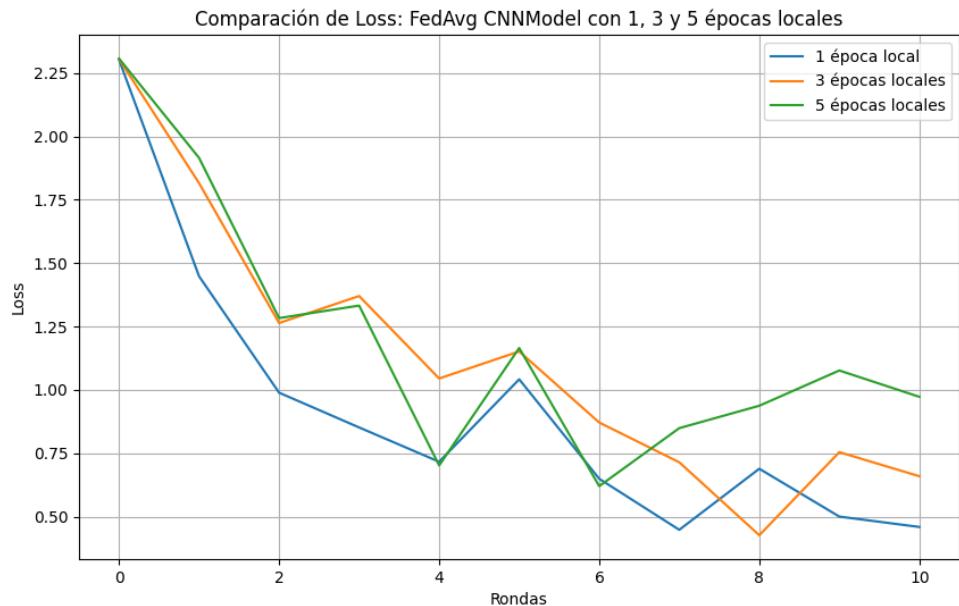


Figure 5: Comparativa de pérdida variando local-epochs

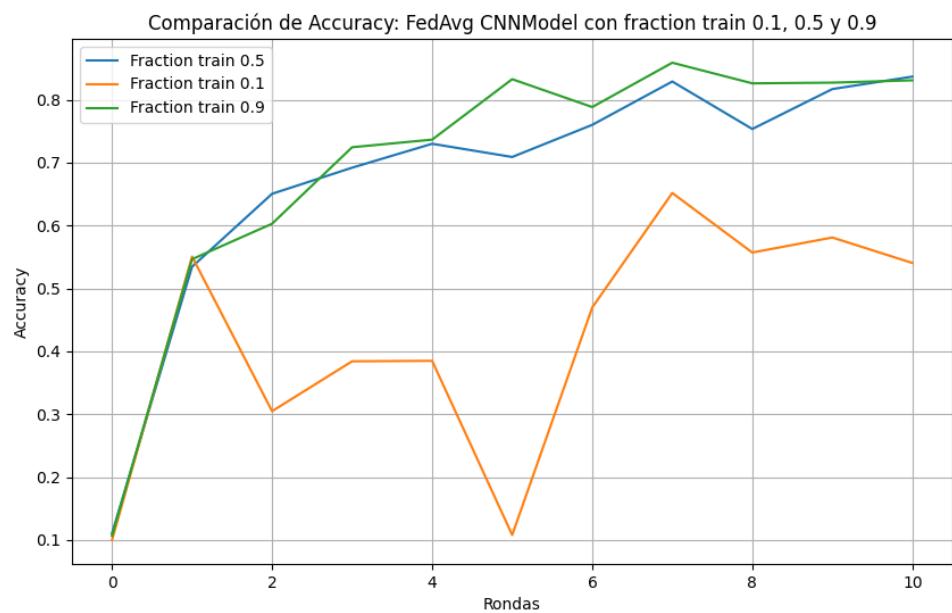


Figure 6: Comparativa de precisión variando fraction-train

Extra: Red preentrenada

Una vez finalizada toda la experimentación requerida para las preguntas de análisis, se ha decidido realizar el experimento adicional de utilizar una red preentrenada. Concretamente, se ha recurrido a la red MobileNet_v3_small, ya que es una red ligera y eficiente, adecuada para dispositivos con recursos limitados, y disponible en la librería `torchvision.models`.

Para utilizar esta red con el conjunto de datos Fashion-MNIST, ha sido necesario adaptar el archivo `task.py` para definir una nueva clase `MobileNet` que carga el modelo preentrenado y hace los siguientes ajustes:

- Modifica la primera capa convolucional para aceptar imágenes de un solo canal (en lugar de tres canales RGB). Se ha cambiado aquí también el parámetro `stride` a 1 para evitar errores de reducción de tamaño de imagen, ya que las imágenes de Fashion-MNIST son de 28x28 píxeles y la red está preentrenada para imágenes 224x224 píxeles.
- Reemplaza la capa final para que tenga 10 salidas, correspondientes a las 10 clases del dataset Fashion-MNIST.

Los resultados no han sido tan buenos como se esperaba, obteniéndose una precisión final del **64.58%** utilizando FedAvg con distribución Dirichlet $\alpha = 0.1$. Tal y como se puede observar en la figura 7, a partir de la ronda 5, la red comienza a sobreajustarse a los datos locales, de forma que a partir de esa ronda la precisión oscila entre el 60% y el 65%, sin mostrar una tendencia clara de mejora.

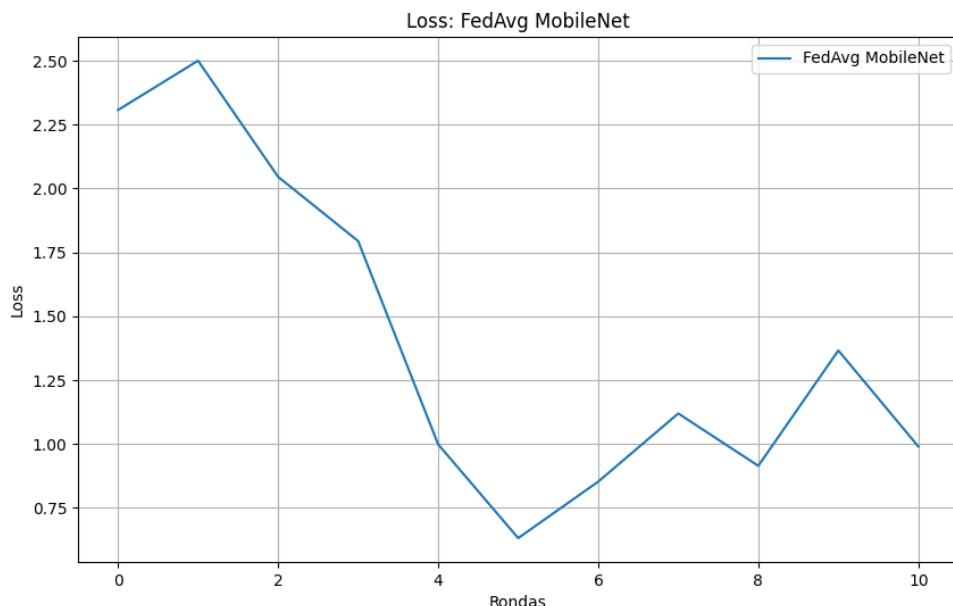


Figure 7: Pérdida del modelo MobileNet preentrenado con FedAvg

Por falta de tiempo no se ha profundizado más en este experimento, pero sería

interesante probar el método FedProx, ajustar los hiperparámetros del aprendizaje federado o incluso cambiar la estrategia de particionado de los datos entre clientes para intentar mejorar los resultados.

Nota

En la entrega final se incluyen todas las gráficas generadas durante la experimentación, así como los archivos .csv con las métricas obtenidas en cada experimento. No todas han sido incluidas en este documento pero están disponibles para su consulta en la carpeta 1-AprendizajeFederado/graficas y 1-AprendizajeFederado/metrics respectivamente.

Parte II: Aprendizaje Continuo con River

En esta parte de la práctica, se trabajará con el dataset de *Electricity*, el cual tiene 45312 instancias y 8 atributos. El objetivo es predecir si el precio de la electricidad subirá o bajará en función de los atributos disponibles. A continuación, se describen las distintas experimentaciones realizadas con modelos de aprendizaje para *data streaming*.

De *batch* a *streaming*

En primer lugar, se entrenará el modelo ***Gaussian Naive Bayes*** utilizando la librería `scikit-learn` en un enfoque de *batch learning*. Pero antes, fue necesario convertir el dataset a un *array* de NumPy y dividir en conjunto de entrenamiento (70%) y de test (30%). La evaluación del modelo tuvo como resultado una precisión del **75.39%**.

A continuación, se implementó el mismo modelo utilizando la librería `River`, que está diseñada para el aprendizaje continuo. El modelo fue entrenado de manera incremental, procesando una instancia a la vez y evaluando su precisión después de cada instancia. El modelo alcanzó una precisión final del **72.87%**, ligeramente inferior al enfoque de *batch learning*, posiblemente debido a la naturaleza secuencial del aprendizaje continuo, como se puede observar en la Figura 8.

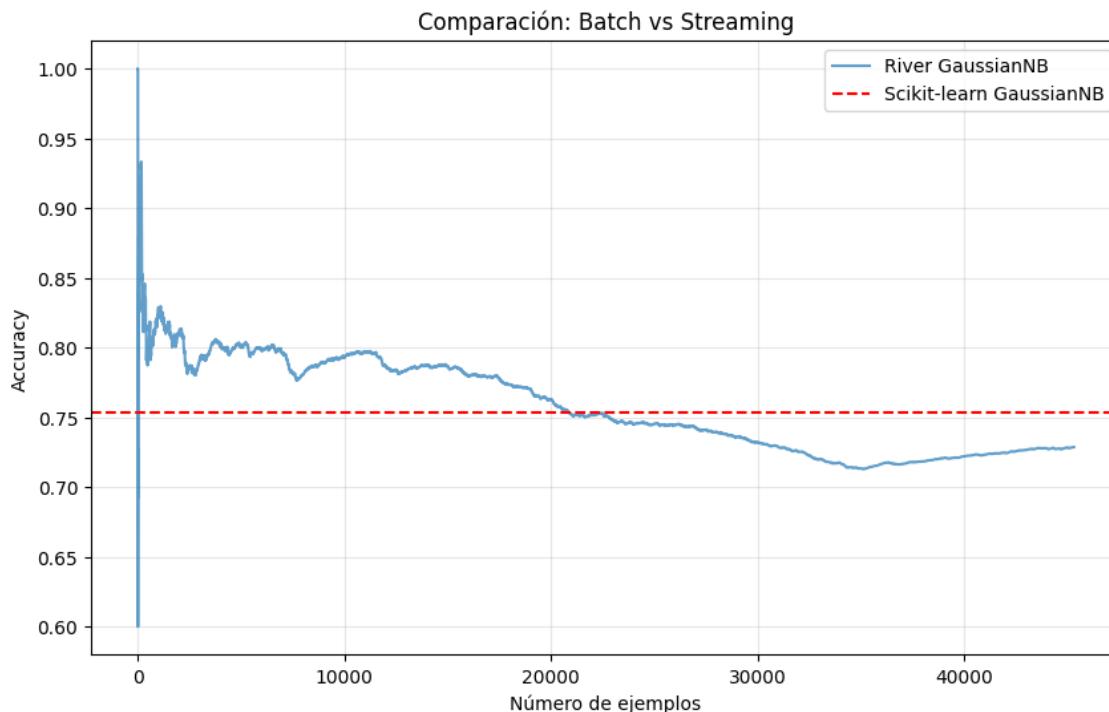


Figure 8: Comparación de precisión entre aprendizaje batch y streaming

Manejo de *concept drift*

Para abordar el *concept drift*, se creó un detector de cambios utilizando el **método ADWIN** (Adaptive Windowing). Este detector monitorea el rendimiento del modelo y ajusta su ventana de datos cuando detecta un cambio significativo en la distribución de los datos. A medida que van llegando los datos, se evalúa el rendimiento del modelo y si la predicción del dato actual es incorrecta, se actualiza el detector ADWIN con un valor de 1; si es correcta, se actualiza con un valor de 0. Cuando ADWIN detecta un cambio, se reinicia el modelo para adaptarse a la nueva distribución de datos.

En la Figura 9 se puede observar cómo el modelo maneja el *concept drift*, detectando 54 cambios a lo largo del flujo de datos y estabilizándose a lo largo del flujo, al contrario que el modelo anterior que presentaba más fluctuaciones en su precisión. El modelo con manejo de *concept drift* alcanzó una precisión final del **80.37%**.

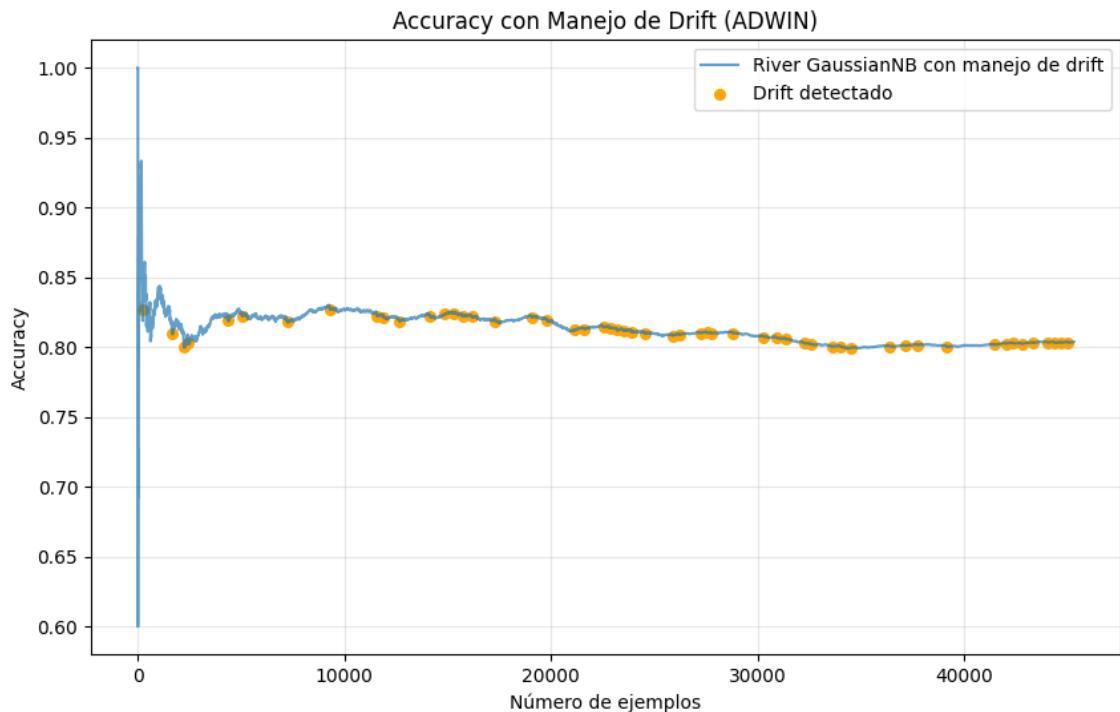


Figure 9: Accuracy con manejo de drift

Modelos adaptativos

Finalmente, se implementaron dos modelos adaptativos: un **Hoeffding Adaptive Tree** (HAT) y un **Adaptive Random Forest** (ARF). Ambos modelos están diseñados para adaptarse automáticamente a los cambios en la distribución de los datos sin necesidad de reiniciar el modelo manualmente. Los dos modelos fueron entrenados y evaluados de la misma manera que el modelo **Gaussian Naive Bayes** de streaming.

El modelo HAT obtuvo una precisión final del **81.50%**, mientras que el modelo ARF alcanzó una precisión del **89.43%**. La Figura 10 muestra la comparación de

precisión entre los tres modelos de aprendizaje continuo, destacando la superioridad del modelo ARF en este caso.

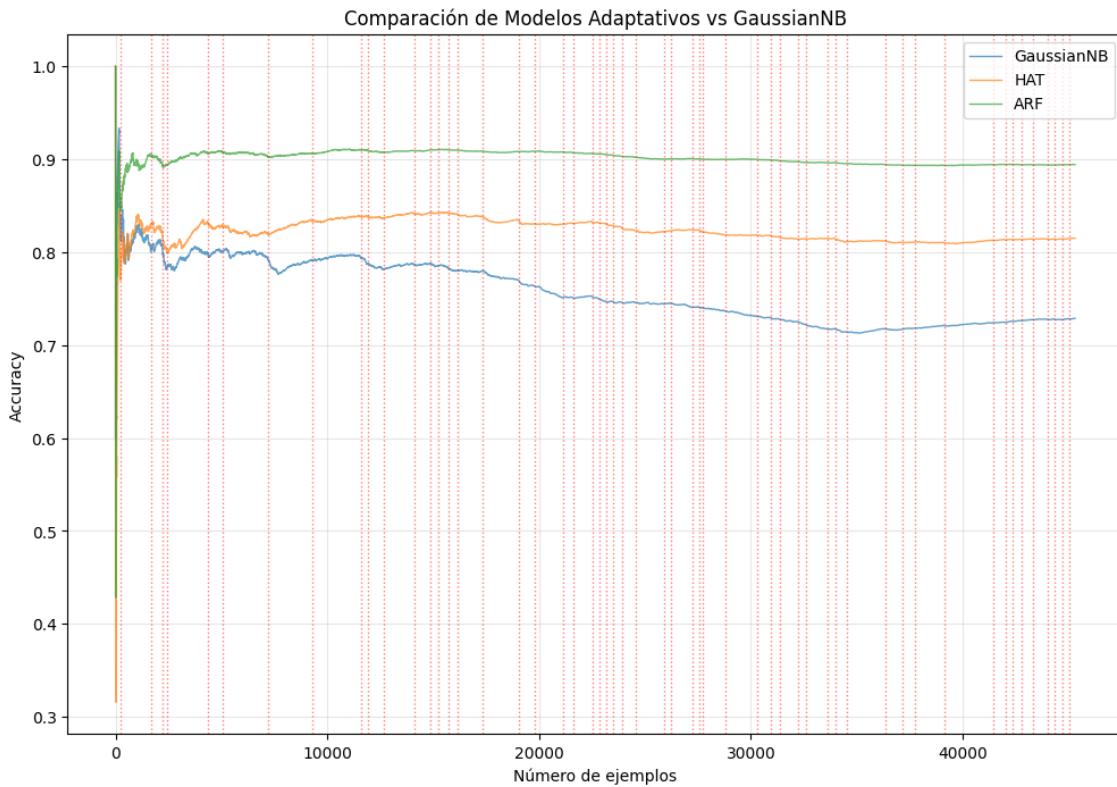


Figure 10: Comparación de precisión entre modelos adaptativos

Preguntas de análisis

1. Justificar por qué un modelo adaptativo como HAT o ARF tiende a superar a GNB en presencia de cambios conceptuales.

Los modelos adaptativos como HAT o ARF superan a GNB en presencia de cambios conceptuales debido a que tienen enfoques distintos en su diseño. La principal diferencia es que GNB se entrena suponiendo estacionariedad en los datos, es decir, que la distribución de probabilidad $P(X|Y)$ no cambia con el tiempo. Por lo tanto, cuando ocurre un cambio conceptual, no será capaz de detectarlo, por lo que su rendimiento se irá degradando a medida que los datos cambien.

Por otro lado, los modelos adaptativos incorporan mecanismos de detección y adaptación que permiten darle más importancia a los datos recientes. En la pregunta siguiente se entrará en más detalles sobre como funcionan HAT y ARF, pero en resumen, ambos modelos pueden ajustar su estructura y parámetros en respuesta a los cambios en la distribución de los datos, lo que les permite mantener un rendimiento más alto en presencia de *concept drift*, como se puede apreciar en la Figura 3.

2. Explicar por qué ARF suele ser más robusto que HAT. Comentar el papel de los árboles adaptativos, las ventanas y la sustitución dinámica de modelos.

ARF suele ser más robusto que HAT debido a que HAT es un solo árbol de decisión, mientras que ARF es un *ensemble* (*random forest*) compuesto por múltiples árboles adaptativos. HAT implementa detectores de *drift* como ADWIN a nivel de nodo del árbol, reemplazando subárboles cuando dejan de ser relevantes. Aunque esto le permite adaptarse a cambios locales en la distribución de los datos, sigue siendo vulnerable a cambios globales que afectan a toda la estructura del árbol.

En contraste, ARF utiliza múltiples árboles adaptativos, cada uno entrenado en una muestra diferente de los datos, con su propio detector de *drift* y con diferentes configuraciones. Esto introduce diversidad en el *ensemble*, lo que mejora la capacidad de generalización y robustez frente a cambios conceptuales. En este caso, las ventanas deslizantes operan a múltiples escalas temporales, permitiendo que cada árbol se especialice en patrones a corto, medio o largo plazo.

Por último, la sustitución dinámica de modelos en ARF permite reemplazar árboles completos que se vuelven obsoletos debido a cambios significativos en los datos, manteniendo así un conjunto de modelos actualizados y relevantes. Todas estas características combinadas hacen que ARF sea más robusto y efectivo en comparación con HAT, teniendo en nuestro caso casi un 10% de mejora en precisión.