

Práctica 2: Aprendizaje Federado y Continuo

Aprendizaje Automático a Gran Escala

Alejandro Rodríguez (alejandro.rodrigueze@udc.es), Pedro de la Fuente (p.de.la.fuente@udc.es) y Jorge Menéndez (j.menendezf@udc.es)

Universidade da Coruña

1 Parte I: Aprendizaje Federado con Flower

1.1 Diseño e Implementación

El objetivo es implementar y analizar un sistema de Aprendizaje Federado para la clasificación de imágenes utilizando el dataset *Fashion-MNIST*. El entorno de simulación se ha construido utilizando el framework Flower (Flwr) y la librería de aprendizaje profundo PyTorch.

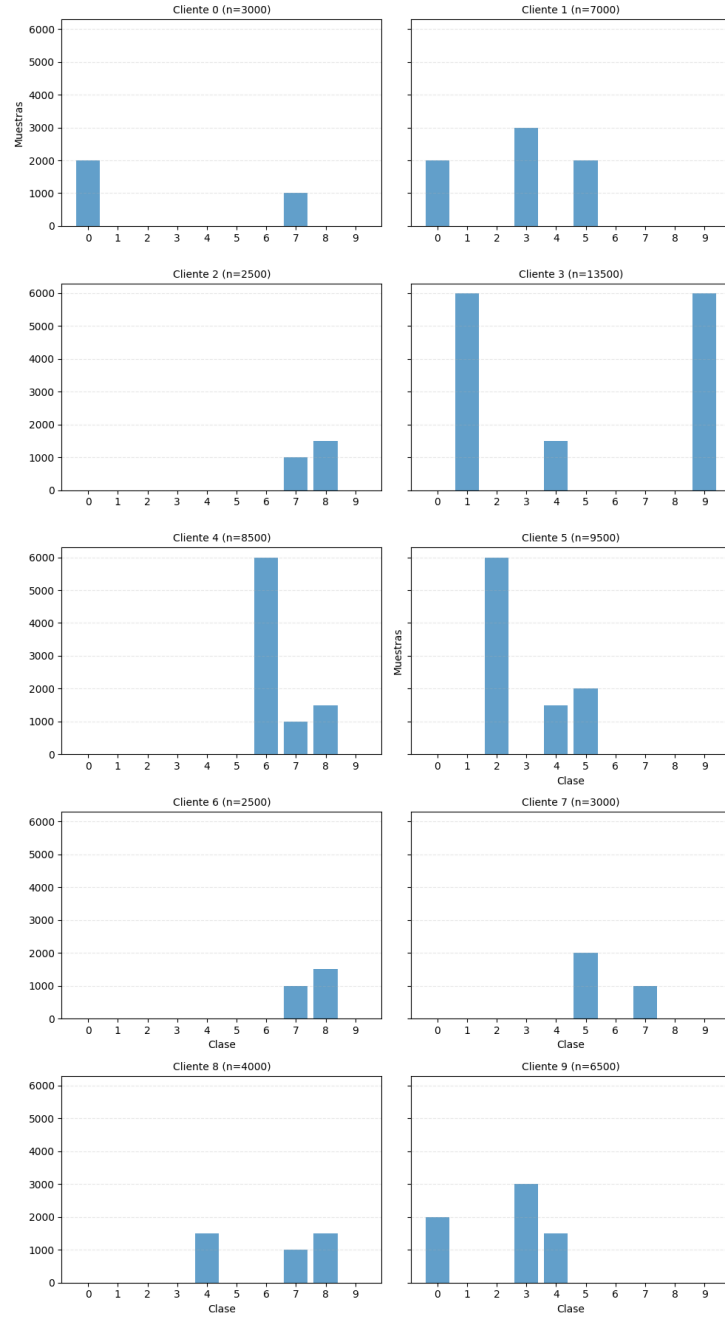
Simulamos un escenario realista Non-IID (Non-Independent and Identically Distributed), donde los datos no están uniformemente distribuidos entre los clientes. Además, se comparan dos estrategias de agregación: FedAvg (el estándar) y Fed-Prox (diseñado para manejar heterogeneidad en los datos).

La arquitectura del proyecto sigue una estructura modular separando la lógica del servidor (`server_app.py`), el cliente (`client_app.py`) y las tareas de aprendizaje automático (`task.py`), orquestadas mediante un archivo de configuración (`pyproject.toml`).

1.1.1 Gestión de Datos y Particionado Non-IID Para cumplir con el requisito de heterogeneidad de datos (Non-IID), se implementó una estrategia de particionado personalizada en el módulo `task.py`.

- **Generación de Particiones:** Se diseñó una función que asigna a cada uno de los 10 clientes un subconjunto de datos limitado a 2 o 3 clases del dataset Fashion-MNIST. La asignación es estocástica: se seleccionan 2 clases base y existe un 50% de probabilidad de añadir una tercera, garantizando que ningún cliente tenga acceso a la distribución global completa.
- **Preprocesamiento:** Antes de generar los DataLoaders, se calculan la media y la desviación típica reales de los píxeles del conjunto de entrenamiento para normalizar las entradas, facilitando la convergencia de los modelos.
- **Validación del Particionado:** Se muestran los histogramas generados para validar que la distribución de clases es efectivamente heterogénea y sesgada en cada cliente en la siguiente figura.

Distribución de Clases por Cliente (Non-IID)

**Fig. 1.** Histogramas de clases de todos los clientes.

1.1.2 Definición de Modelos y Estrategias Se han definido dos arquitecturas neuronales en `task.py`:

- **MLP (Perceptrón Multicapa):** Modelo base obligatorio. Consiste en una capa `Flatten` seguida de capas lineales con funciones de activación `ReLU` y regularización mediante `Dropout` (0.2) para mitigar el sobreajuste en los conjuntos de datos locales reducidos.
- **CNN (Red Neuronal Convolutacional):** Modelo extendido. Consta de dos bloques convolucionales seguidos de un clasificador lineal denso. Busca explotar la correlación espacial de las imágenes.

El entrenamiento local soporta tanto **FedAvg** como **FedProx**. En el caso de FedProx, se incluye el término `proximal_mu` en la función de pérdida para penalizar la divergencia de los pesos locales respecto al modelo global. Para asegurar la reproductibilidad de los experimentos, se ha fijado una semilla global (`SEED=42`).

1.1.3 Lógica del Cliente y Entrenamiento La clase `client_app.py` encapsula la lógica de entrenamiento local. Se ha extendido el bucle de entrenamiento estándar para soportar tanto FedAvg como FedProx.

- **División Train/Val:** Antes del entrenamiento, cada cliente divide su partición local de datos: un 80% se destina al entrenamiento (actualización de gradientes) y el 20% restante se reserva para validación local. Esto permite al cliente evaluar la generalización del modelo recibido antes o después de entrenar, sin contaminar los datos de test del servidor."
- **Entrenamiento y Optimizador:** Se utiliza el optimizador SGD con un momentum de 0.9. En la función `train_one_round` se introduce el cálculo del término proximal. Si `proximal_mu` es > 0 , se calcula la norma L2 entre los pesos locales actuales y los pesos globales recibidos del servidor. Este valor se suma a la función de pérdida, penalizando las desviaciones excesivas del modelo global.
- **Comunicación:** El cliente recibe los parámetros globales, entrena durante un número configurable de `local_epochs` y devuelve los pesos actualizados.

1.1.4 Lógica del Servidor y Estrategias El servidor `server_app.py` orquesta la federación utilizando las estrategias de Flower, seleccionadas dinámicamente según la configuración.

- **Configuración Dinámica:** Se leen los hiperparámetros desde `pyproject.toml`. Este permite ejecutar diferentes escenarios experimentales sin modificar el código fuente.
- **Evaluación Centralizada:** Se implementa una función que testea el modelo agregado al final de cada ronda utilizando un conjunto de test centralizado (distribución IID completa) para medir el rendimiento real del modelo global.
- **Persistencia de los Resultados:** El servidor registra las métricas de Accuracy y Loss ronda a ronda en archivos CSV con nombres descriptivos (ej. `results_fedavg(mlp)_1e10_frac1.0.csv`), lo que facilita la generación posterior de gráficas comparativas.

1.2 Análisis de Hiperparámetros y Resultados

En esta sección se analiza el comportamiento del sistema federado bajo diferentes configuraciones, evaluando el impacto de los hiperparámetros locales y comparando las estrategias de agregación.

1.2.1 Impacto de `local_epochs` Se evaluó el impacto de la carga de trabajo local en la convergencia del modelo global configurando `local_epochs` en $\{1, 5, 10\}$, manteniendo fija la tasa de participación de clientes `fraction_fit=1.0` y utilizando FedAvg.

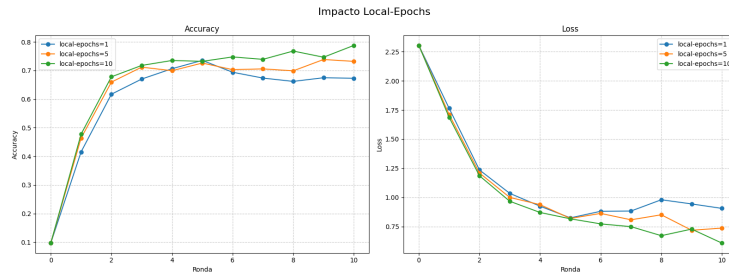


Fig. 2. Evolución de la precisión y pérdida variando el número de `local_epochs`.

Como se observa en la figura 2, el número de `local_epochs` actúa como un fuerte acelerador de la convergencia. Mientras que con `local_epochs=10` el modelo logra una precisión superior al 76% rápidamente, la configuración de `local_epochs=1` muestra un rendimiento significativamente inferior. Tras 10 rondas de comunicación, el modelo con el valor más bajo de `local_epochs` apenas alcanza el rendimiento que el modelo de `local_epochs=10` logra en la segunda ronda, evidenciando una convergencia mucho más lenta.

Este comportamiento se debe a la naturaleza Non-IID de los datos. Con pocas `local_epochs`, los gradientes generados por los clientes son "superficiales" y muy sesgados hacia sus clases específicas. Al promediar estos pesos en el servidor, el modelo global avanza con dificultad hacia una solución general. Por el contrario, al aumentar las `local_epochs`, los clientes son capaces de aprender representaciones más robustas de sus subconjuntos de datos antes de la agregación, lo que permite al modelo global dar "pasos" más grandes y efectivos en el espacio de optimización, reduciendo drásticamente la necesidad de rondas de comunicación.

Dada la superioridad en rendimiento, seleccionamos `local_epochs=10` como el valor óptimo para los siguientes experimentos.

1.2.2 Impacto de `fraction_fit` Manteniendo `local_epochs=10`, se analizó el impacto de la participación de los clientes (`fraction_fit`) probando los valores $\{0.1, 0.4, 1.0\}$. Dado que tenemos 10 clientes en total, esto equivale a seleccionar 1, 4 o 10 clientes por ronda respectivamente.

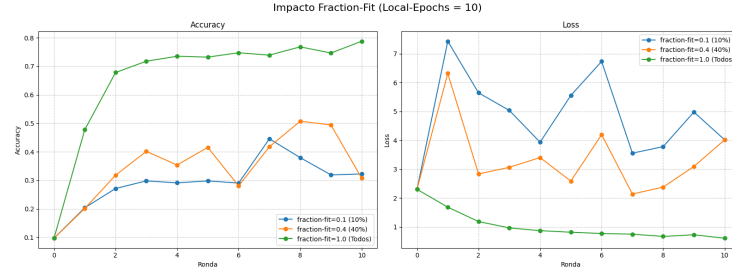


Fig. 3. Evolución de la precisión y pérdida variando el número de `fraction_fit`, una vez fijado `local_epochs=10`.

La figura 3 revela una disparidad crítica en el rendimiento. Las configuraciones con `fraction_fit=0.1` y `0.4` muestran un comportamiento errático e insuficiente, estancándose en una precisión inferior al 40% tras 10 rondas. Las curvas son extremadamente ruidosas, con caídas de rendimiento entre rondas consecutivas. Por el contrario, al aumentar la participación a `fraction_fit=1.0` el sistema se estabiliza notablemente, alcanzando una precisión final cercana al 80% y ofreciendo la curva más suave y la convergencia más rápida.

La inestabilidad observada con los valores más bajos es una consecuencia directa de la heterogeneidad de los datos (Non-IID). Al seleccionar pocos clientes, el modelo global se actualiza basándose exclusivamente en los gradientes de las 2 o 3 clases que poseen esos clientes, "olvidando" lo aprendido sobre otras clases en rondas anteriores (olvido catastrófico). El vector de actualización resultante está muy sesgado y no representa la distribución real de los datos. Al aumentarla, el servidor promedia los pesos de múltiples clientes con distribuciones de clases distintas cancelando los sesgos individuales y produciendo un vector de actualización global que apunta con mayor fidelidad hacia el mínimo de la función de pérdida global, suavizando la curva de aprendizaje.

Para los siguientes experimentos seleccionaremos `fraction_fit=1.0` (participación total). Al eliminar el ruido introducido por el muestreo de clientes, podremos comparar las estrategias de agregación (FedAvg vs. FedProx) en igualdad de condiciones, asegurando que las diferencias de rendimiento se deban al algoritmo y no a la suerte en la selección de clientes.

1.2.3 Comparativa de Estrategias: FedAvg vs FedProx Se comparó el algoritmo estándar FedAvg con FedProx, variando el parámetro de regularización μ en $\{0.01, 0.1\}$ para evaluar la robustez de la agregación frente a la heterogeneidad.

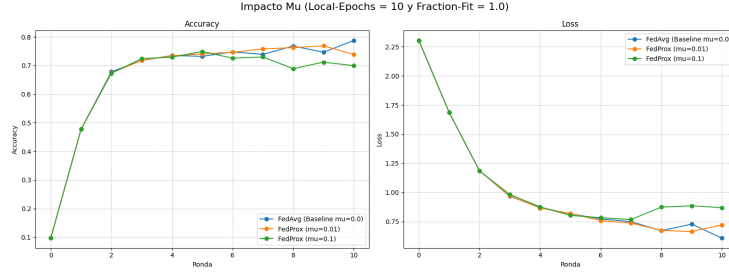


Fig. 4. Evolución de la precisión y pérdida variando el valor de `proximal_mu` (μ) una vez fijados `local_epochs`=10 y `fraction_fit`=1.0.

Los resultados mostrados en la figura 4 indican que la elección de μ es determinante. Con un valor conservador de $\mu = 0.01$, el comportamiento es muy competitivo frente a FedAvg durante la mayoría de las rondas, aunque muestra una ligera inestabilidad final en la ronda 10. Sin embargo, al aumentar la restricción a $\mu = 0.1$, el rendimiento cae visiblemente, ya que la precisión cae y la curva de loss se mantiene consistentemente más alta.

La función de FedProx es añadir un "término proximal" a la función de pérdida local que limita cuánto pueden alejarse los pesos del cliente respecto al modelo global. Es por ello que cuanto menor es el valor menos afecta, ya que apenas modifica el gradiente. Dado que nuestro entorno de simulación no presenta clientes lentos o con fallos ni una heterogeneidad extrema que cause divergencia masiva, la regularización adicional de FedProx no aporta una ventaja competitiva en términos de precisión.

Considerando que FedProx introduce una sobrecarga computacional adicional (cálculo de la norma L2 en cada paso) y que, en este escenario específico sin fallos de sistema, no supera la precisión de FedAvg, seleccionamos FedAvg como la estrategia más eficiente. Aún así, para la comparativa final de modelos, mantendremos la mejor configuración de FedProx ($\mu = 0.01$).

1.2.4 Comparativa General de Arquitecturas: MLP vs CNN Finalmente, se substituyó el perceptrón multicapa (MLP) por una Red Neuronal Convolutiva (CNN) compacta para evaluar el impacto de la arquitectura, manteniendo los hiperparámetros encontrados `local_epochs`=10 y `fraction_fit`=1.0.

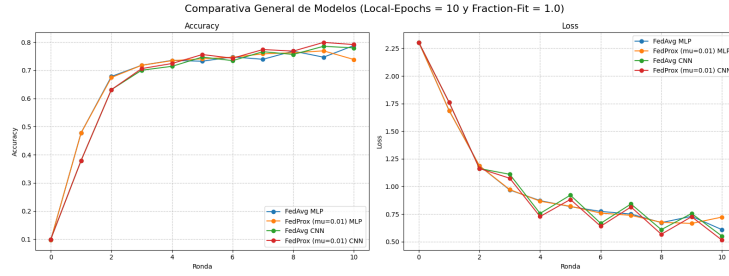


Fig. 5. Evolución de la precisión y pérdida en la comparación general de estrategias FedAvg y FedProx.

Tal y como muestra la figura 5, la arquitectura más compleja (CNN) no logra una mejora drástica en precisión respecto al modelo simple (MLP). Ambos modelos terminan la simulación con un rendimiento muy parejo, oscilando entre el 78% y el 79%. De hecho, bajo FedAvg, el MLP alcanza un 78.78%, superando ligeramente el 77.95% de la CNN.

Sin embargo, el análisis de la Pérdida (Loss) revela una historia diferente. La CNN reduce el error a valores entre 0.51 y 0.55, notablemente inferiores al 0.61 del MLP. Técnicamente, esta discrepancia entre un Loss menor y un Accuracy similar evidencia una brecha de confianza o calibración: aunque ambos modelos cometen un número similar de errores de clasificación, cuando la CNN acierta, lo hace con una probabilidad muy cercana a 1 (alta certeza), mientras que el MLP muestra mayor incertidumbre. Esta robustez interna sugiere que la CNN será mucho más resiliente ante ruido futuro o nuevos datos, aunque la métrica rígida de Accuracy todavía no lo refleje.

Esta situación es común en las primeras etapas del aprendizaje federado heterogéneo. La CNN, al tener mayor capacidad (filtros convolucionales), minimiza mejor la función de coste global, pero requiere más rondas para "alinearse" los conocimientos locales y superar los sesgos Non-IID. El MLP, al ser más simple, converge rápido a una solución 'decente' pero subóptima.

Como conclusión general, en un escenario limitado a 10 rondas, la CNN no justifica su coste computacional extra frente al MLP, que se muestra como una solución muy eficiente ('baseline robusto'). Para materializar la superioridad de la CNN, sería necesario extender la simulación a 20 o 30 rondas. Por tanto, para este experimento corto, el MLP con FedAvg resulta ser la opción más equilibrada.

2 Parte II: Aprendizaje Continuo con River

En esta sección se analiza el aprendizaje sobre flujos de datos (*streaming*) utilizando el conjunto de *Electricity*. Este dataset contiene fluctuaciones en los

precios de la electricidad que dependen de la demanda y la oferta, presentando claros desafíos de cambio de concepto (*concept drift*) a lo largo del tiempo.

El objetivo es comparar el rendimiento de modelos estáticos (Batch), modelos incrementales simples (Naive Bayes) y modelos adaptativos avanzados (HAT y ARF).

2.1 De Batch a Streaming

Se comparó el enfoque tradicional por lotes frente al aprendizaje incremental utilizando el clasificador **GaussianNB**.

2.1.1 Metodología

- **Enfoque Batch (Scikit-learn):** Se dividió el dataset respetando el orden temporal (sin barajar), utilizando los primeros 31,718 registros (70%) para entrenamiento y los restantes para test.
- **Enfoque Streaming (River):** Se utilizó una evaluación *Prequential*. El modelo predice una instancia para contabilizar la métrica e inmediatamente entrena con ella.

2.1.2 Resultados Comparativos La tabla 1 muestra la precisión final obtenida por ambos enfoques. Aunque el modelo en streaming obtiene una precisión ligeramente inferior (-2.8%), su ventaja radica en la eficiencia: procesó el flujo completo en 8.95 segundos con un consumo de memoria constante, algo inviable para modelos batch en flujos infinitos.

Table 1. Comparativa de precisión: Modelo Estático vs. Incremental.

Librería	Estrategia	Precisión (Accuracy)
Scikit-learn	Batch (Hold-out 70/30)	0.7539
River	Streaming (Prequential)	0.7258

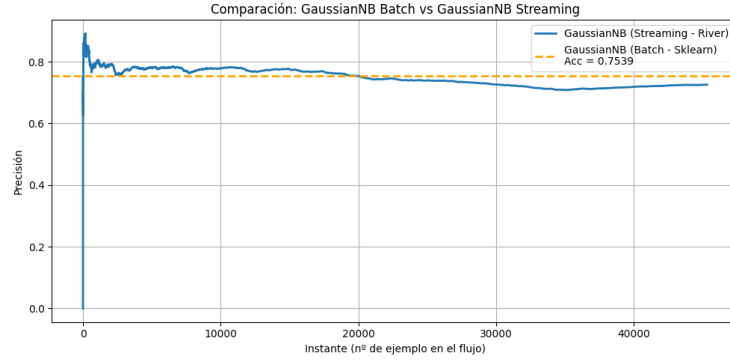


Fig. 6. Evolución de la precisión del modelo en Batch estático frente al modelo Streaming incremental

2.2 Manejo de Concept Drift con ADWIN

Dado que los datos de electricidad varían con el tiempo (cambios de estación, mercado, etc.), se integró el detector de cambios ADWIN (Adaptive Windowing) al clasificador **GaussianNB**.

2.2.1 Estrategia de Adaptación Se monitorizó la precisión (o error) dentro de una ventana adaptativa mediante el algoritmo ADWIN. Al detectar un *drift* significativo, se aplicó una estrategia de reinicio completo del modelo (*model reset*), descartando el conocimiento previo para adaptarse rápidamente a la nueva distribución de datos.

2.2.2 Resultados

- **Drifts detectados:** 59 eventos de cambio.
- **Precisión final:** 0.7998.

La incorporación de ADWIN mejoró la precisión del modelo Naive Bayes incremental de 0.7258 a 0.7998 (+7.4%). Esto confirma la presencia de cambios conceptuales en el dataset y demuestra que "olvidar" datos antiguos obsoletos es beneficioso en este escenario.

Analizando la figura 7, se observa cómo, justo después de cada línea vertical de detección de drift, la precisión sufre una caída pero se recupera con una pendiente casi vertical. Esta recuperación instantánea valida la estrategia de reinicio frente a la adaptación gradual: en escenarios de cambios bruscos como el mercado eléctrico, resulta computacionalmente más eficiente y preciso descartar el conocimiento obsoleto y re-aprender desde cero con datos frescos que intentar corregir los sesgos de un modelo antiguo.

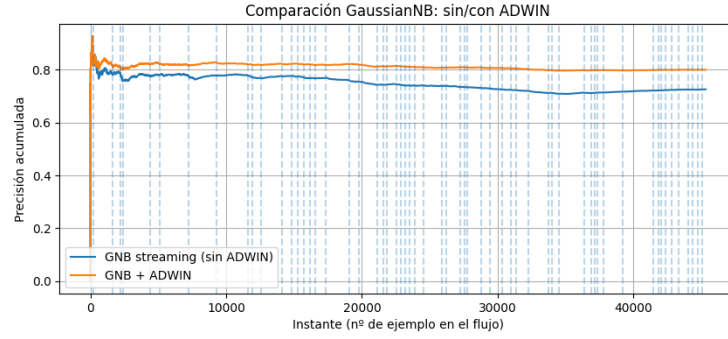


Fig. 7. Detección de Drifts con ADWIN. Las líneas verticales indican los puntos donde se reinició el modelo.

2.3 Modelos Adaptativos: HAT y ARF

Finalmente, se evaluaron dos modelos diseñados nativamente para entornos cambiantes: el Hoeffding Adaptive Tree (HAT) y el Adaptive Random Forest (ARF). Ambos gestionan el drift internamente sin necesidad de forzar reinicios manuales.

2.3.1 Resultados La tabla 2 resume el rendimiento final. El Adaptive Random Forest demostró ser el modelo más robusto.

Table 2. Comparativa de precisión: Modelo Estático vs. Incremental.

Modelo adaptativo	Precisión final
Hoeffding Adaptive Tree (HAT)	0.8318
Adaptive Random Forest (ARF)	0.8946

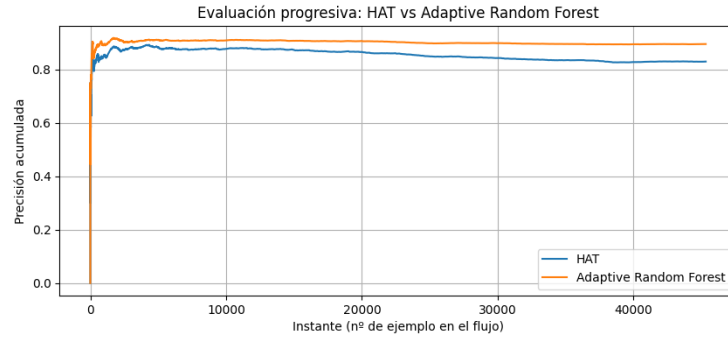


Fig. 8. Evaluación progresiva de modelos adaptativos: HAT vs ARF.

2.4 Análisis y Discusión

2.4.1 Superioridad de los modelos adaptativos (HAT y ARF) frente a GaussianNB ante cambios conceptuales Los resultados experimentales demuestran que tanto HAT como ARF superan consistentemente a Gaussian Naive Bayes (GNB) en entornos de flujo de datos con concept drift. Esta diferencia se basa en las asunciones estadísticas de cada modelo:

- **Rigidez de GNB (Estacionariedad):** GaussianNB es un modelo paramétrico que asume que la distribución de los datos es estacionaria y que los atributos son condicionalmente independientes. El modelo mantiene estadísticas globales (medias y varianzas) calculadas sobre todo el histórico de datos. Cuando ocurre un cambio de concepto, estas estadísticas "antiguas" sesgan la predicción, ya que el modelo arrastra una memoria infinita que deja de ser representativa de la distribución actual. Además, su incapacidad para capturar interacciones complejas no lineales limita su precisión.
- **Flexibilidad de los Modelos Adaptativos:** Por el contrario, HAT y ARF son modelos no paramétricos basados en árboles adaptativos. Estos dividen el espacio en regiones locales, permitiendo modelar fronteras de decisión complejas y no lineales. Su principal ventaja es la capacidad de olvidar información obsoleta y aprender nuevos patrones de forma incremental, ajustándose a la dinámica temporal de los datos sin las restricciones de normalidad e independencia de GNB.

2.4.2 Robustez del ensamblado: Análisis de la superioridad de ARF sobre HAT Al comparar los modelos adaptativos entre sí, el Adaptive Random Forest (ARF) demuestra una mayor robustez y precisión (89.46%) frente al Hoeffding Adaptive Tree (HAT) individual (83.18%). Esta superioridad se explica por el efecto del ensamblado y la gestión avanzada del drift.

1. **El papel de los Árboles Adaptativos y la Reducción de Varianza:** Un HAT individual es sensible al ruido y puede sufrir de alta varianza (overfitting)

a cambios locales). ARF mitiga esto mediante Online Bagging (usando una distribución de Poisson para el remuestreo). Al combinar múltiples árboles adaptativos que observan variaciones del flujo de datos, el voto mayoritario del bosque suaviza los errores individuales y proporciona una predicción más robusta.

2. **Detectores basados en Ventanas (ADWIN):** La capacidad de adaptación de estos modelos reside en el uso de detectores de cambio basados en ventanas, específicamente ADWIN (Adaptive Windowing).
 - En cada nodo del árbol, se mantienen ventanas de datos de tamaño variable que monitorean el error o la precisión.
 - Si las estadísticas de la sub-ventana más reciente difieren significativamente de las del pasado (superando un umbral de confianza), el detector asume que ha ocurrido un cambio. Esto permite una granularidad fina, detectando drifts en regiones específicas del espacio de características.
3. **Mecanismo de Sustitución Dinámica de Modelos:** Este es el factor diferenciador que otorga mayor robustez a ARF frente a HAT:
 - **HAT (Reinicio brusco):** Cuando detecta un cambio, poda la rama obsoleta e inicia una nueva desde cero. Esto puede provocar una caída temporal en el rendimiento mientras la nueva rama aprende.
 - **ARF (Sustitución suave):** Implementa una estrategia de "árboles de fondo" (background trees). Cuando el detector de ventanas de un árbol base lanza una advertencia (pero aún no confirma el cambio), ARF comienza a entrenar un árbol paralelo en segundo plano. Si el cambio se confirma, se produce una sustitución dinámica del modelo: el árbol de fondo (ya entrenado con los datos recientes) reemplaza instantáneamente al árbol principal. Esto garantiza que el ensamblado mantenga una alta precisión sin los "baches" de aprendizaje que sufre un árbol individual.