Universidade Vigo

Trabajo Fin de Máster

Online Machine Learning

Carlos Rodríguez González

Máster en Técnicas Estadísticas

Curso 2021-2022

Propuesta de Trabajo Fin de Máster

Título en galego: Online Machine Learning
Título en español: Online Machine Learning
English title: Online Machine Learning
Modalidad: Modalidad B
Autor/a: Carlos Rodríguez González, Universidade de Santiago de Compostela
Director/a: Marta Sestelo Pérez, Universidade de Vigo
Tutor/a: Bruno Fernández Castro, Gradiant
Breve resumen del trabajo: Investigación y desarrollo de un prototipo basado en Online Machine Learning que permita calcular y actualizar en tiempo real un modelo de regresión o clasificación, mejorando la capacidad de autoaprendizaje de estos métodos durante el procesado de flujos de datos continuos a lo largo del tiempo.
Recomendaciones:
Otras observaciones:

Don/doña Marta Sestelo Pérez, Profesora ayudante doctora de la Universidade de Vigo, y don/doña Bruno Fernández Castro, Responsable Técnico de Machine Learning & Optimización de Gradiant, informan que el Trabajo Fin de Máster titulado

Online Machine Learning

fue realizado bajo su dirección por don/doña Carlos Rodríguez González para el Máster en Técnicas Estadísticas. Estimando que el trabajo está terminado, dan su conformidad para su presentación y defensa ante un tribunal.

En Santiago de Compostela, a 27 de enero de 2022.

El/la director/a:

SESTELO PEREZ MARTA - 76998300G

Firmado digitalmente por SESTELO PEREZ MARTA - 76998300G
Fecha: 2022.01.27 11:11:10 +01'00'

Don/doña Marta Sestelo Pérez

El/la tutor/a:

Firmado por FERNANDEZ CASTRO, BRUNO (FIRMA) el día 28/01/2022 con un certificado emitido por AC DNIE 006

Don/doña Bruno Fernández Castro

El/la autor/a:

Don/doña Carlos Rodríguez González

Índice general

Capítulo 1

Introducción

En este capítulo se realiza una descripción del centro tecnológico Gradiant en el cual se ha realizado este trabajo y se introduce el problema tratado en el mismo.

1.1. Descripción de Gradiant

Gradiant[1], centro tecnológico TIC, tiene como objetivo mejorar la competitividad de las empresas mediante la transferencia de conocimiento y tecnologías en los ámbitos de la conectividad, inteligencia y seguridad. Con más de 100 profesionales y 14 patentes solicitadas, Gradiant ha desarrollado más de 285 proyectos diferentes de I+D+i, convirtiéndose en uno de los principales motores de la innovación en Galicia. Su facturación en 2018 alcanzó los 5 millones de euros, trabajando con más de 170 clientes en 29 países.

El Centro se conforma a partir de un patronato que agrupa a representantes del sector público y privado. Está formado por las Universidades de A Coruña, Santiago de Compostela y Vigo; las empresas Altia, Arteixo Telecom, Egatel, Indra, Plexus, R, Telefónica, Grupo Televés, y la Asociación empresarial INEO. El compromiso del centro con la calidad es una constante desde sus inicios. El Centro cuenta con los siguientes certificados: Sistema de Gestión de Calidad UNE-EN ISO 9001:2015, Sistema de Gestión de Proyectos de I+D+i UNE 166002:2014, Sistemas de Gestión de la Seguridad de la Información UNE-EN ISO/IEC 27001:2013. Además, forma parte del registro estatal de Centros de Innovación Tecnológica (Sello CIT).

[1] https://www.gradiant.org/

Tras diez años de actividad, Gradiant se sitúa como socio tecnológico de la industria orientado a sus necesidades en el ámbito de las TIC, aportando su experiencia nacional e internacional en tecnologías para la seguridad y la privacidad; el procesado de señales multimedia; internet de las cosas; la biometría y analítica de datos y los sistemas de comunicaciones avanzadas.

Por su propio objetivo fundacional, Gradiant ha colaborado en múltiples proyectos orientados a la mejora de la competitividad de las empresas, principalmente en el ámbito de la digitalización, la profesionalización de la producción y la incorporación efectiva de tecnología. En lo que se refiere a las tecnologías del lenguaje, además de su participación en este tipo de proyectos, Gradiant ha participado en competiciones internacionales y nacionales. Estas competiciones generalmente implican el uso de técnicas de aprendizaje automático y recursos lingüísticos para resolver tareas complejas (por ejemplo, análisis de sentimientos, similitud semántica de texto, traducción automática, etc.):

- OKE 2018 Challenge - ESWC 2018. Gradiant consiguió el primer puesto en "Task 3: Relation extraction".
- SemEval 2015 Sentiment Analysis in Twitter: Gradiant terminó en el puesto 16 de 40 en la clasificación general, siendo galardonado con el 1er puesto en uno de los conjuntos de datos (detección de sarcasmo 2014).
- TASS SePLN 2015 Sentiment Analysis of Spanish Tweets: Gradiant logró la 2ª posición de los participantes.

Por otro lado, siguiendo la misma orientación, Gradiant también ofrece servicios de formación. Entre los impartidos en los últimos años, podemos destacar los siguientes:

- Formaciones para equipos de trabajo técnicos:
  - SCRUM.
  - DevOps. Plataforma CI/CD.
  - Docker y Kubernetes.
  - Programa SkillShare (Unidad Mixta Telefónica - Gradiant). Data engineering.
- Formaciones en escuelas de negocio:
  - IESIDE: Programa de BI, Controlling y Big Data.
  - IFFE: Máster de Big Data. Asignatura de introducción a Industria 4.0.

- Formaciones a clústeres:
  - CLUSAGA: PROGRAMA DE CAPACITACIÓN EN INDUSTRIA 4.0 2016 (Sistemas Ciberfísicos, Internet Industrial de las cosas y Ciberseguridad, Big Data. Análisis de datos. Cloud).
  - CLUSAGA: PROGRAMA DE CAPACITACIÓN EN INDUSTRIA 4.0 2018 (Industrial Internet of Things, Big Data y analítica avanzada, Ciberseguridad).
  - APV: PROGRAMA DE CAPACITACIÓN DE PERSONAL DIRECTIVO EN EL ÁMBITO PORTUARIO EN INDUSTRIA 4.0 (Recogida y gestión de información para la Industria 4.0, Cloud computing y ciberseguridad).
  - CLÚSTER TIC: formación de BLOCKCHAIN avanzada.

Además, Gradiant ofrece otros servicios profesionales avanzados: asesoramiento estratégico, experto técnico, vigilancia tecnológica, apoyo a la innovación, laboratorio HW (https://www.gradiant.org/servicios/).

1.1.1. Internacionalización

Gradiant dispone de 11 patentes, principalmente en EEUU y en Europa, pero también se han gestionado patentes a nivel estatal. Además, dentro de su portfolio tecnológico, destacan un amplio abanico de tecnologías directamente licenciables, que están a disposición de las empresas interesadas mediante acuerdos comerciales.

Como resultado de una apuesta estratégica a medio y largo plazo iniciada en 2015, Gradiant se ha encaminado decididamente a la internacionalización de su propiedad intelectual en tecnología con una estrategia decidida de desarrollo de negocio en los principales mercados internacionales. En esta línea, anualmente se hace una revisión del plan comercial internacional para enfocar los esfuerzos de comercialización en mercados con especial interés en sus tecnologías.

También en consonancia con estas apuestas, en los últimos tiempos Gradiant está trabajando también en la identificación de oportunidades de innovación abierta en otros países. Fruto de este trabajo, ha presentado recientemente una propuesta en la convocatoria RIE Industry Day 2019 (Singapur).

En términos cuantitativos, puede dar cuenta del éxito en la estrategia de internacionalización con más de 30 licencias vendidas en Europa y Estados Unidos, más de 10 empresas extranjeras probando tecnología desarrollada en Gradiant, más de 5 pilotos de testeo precomercial internacional y la firma de 3 acuerdos de promoción tecnológica. Enfocando en el año 2018, se puede destacar una facturación internacional del 18 % al cierre y presencia (a diferentes niveles) en 29 países.

En cuanto a proyectos con fondos FP7 y H2020, Gradiant participa y ha participado en numerosos proyectos internacionales.

1.1.2. Algunas referencias

La Figura 1.1 muestra algunas de las principales empresas del sector industrial con las que Gradiant ha colaborado o colabora actualmente en proyectos relacionados con inteligencia artificial y/o aprendizaje automático.

Figura 1.1: Principales empresas del sector industrial colaboradoras con Gradiant.

1.2. Exposición del problema

Los modelos basados en algoritmos de aprendizaje automático (*machine learning* o ML) son elementos estáticos, en cuanto a que se construyen a partir de un conjunto sufi-

ciente de datos de entrenamiento antes de su evaluación y despliegue en producción. A no ser que se realicen reentrenamientos periódicos con nuevos conjuntos de datos (tanto de forma manual como programados periódicamente) estos modelos no suelen ser actualizados.

Los modelos basados en algoritmos de *online machine learning* están diseñados para reaccionar de forma dinámica y en tiempo real a los flujos de datos continuos que reciben como entrada, reentrenándose iterativamente y adaptándose de forma automática a los posibles cambios en sus distribuciones de probabilidad.

Este tipo de algoritmos tienen aplicaciones en diferentes ámbitos, por ejemplo:

- Situaciones en las que el modelo debe adaptarse eficiente y dinámicamente a frecuentes cambios en las propiedades de los datos de entrada.
- Escenarios en las que no se dispone de datos con la cantidad y/o calidad suficiente como para realizar un único entrenamiento del modelo y seguir operando indefinidamente.
- Bases de datos demasiado grandes como para caber en memoria (aprendizaje *out-of-core*).
- Dispositivos IoT embebidos, para aplicaciones de *edge computing*, en los que el *hardware* no es tan potente como para realizar entrenamientos con grandes volúmenes de datos.

Online machine learning (o simplemente *online learning*) es, por todo ello, un nuevo campo de investigación abierto dentro de la línea de proyectos relacionados con la Industria 4.0 en Gradiant. El objetivo de esta investigación, en la que se enmarca el presente trabajo, es indagar en esta clase de métodos (sus prestaciones, características, limitaciones y comportamiento ante diferentes situaciones) con el fin de validar su potencial para el desarrollo y comercialización de diversas aplicaciones basadas en análisis de datos: mantenimiento predictivo de equipos, monitorización de la calidad de manufacturas o sistemas prescriptivos para el control de la producción, entre otras.

Durante este trabajo se han realizado las siguientes tareas principales:

- Breve revisión del estado del arte en relación a la temática propuesta por Gradiant, esto es, métodos de regresión/clasificación basados en técnicas de *online learning*.

- Revisión detallada de un algoritmo concreto, *adaptive random forests*, para la resolución de problemas de regresión/clasificación supervisados.
- Diseño y programación en lenguaje Python de una batería de pruebas y experimentos para la evaluación del citado algoritmo con distintos conjuntos de datos, así como el análisis de resultados obtenidos.

Capítulo 2

Online learning

En este capítulo se hace una introducción de *online learning* así como una breve revisión del estado del arte reciente en relación a los algoritmos y librerías de código abierto disponibles.

2.1. Introducción online learning

Online learning (Hoi et al. 2021) son un subconjunto de técnicas dentro de los métodos de *machine learning* orientados al procesado de flujos continuos de datos. Incluye una importante cantidad de técnicas diseñadas para el desarrollo de modelos de regresión o clasificación capaces de aprender de los datos disponibles para su entrenamiento de forma secuencial, superando las dificultades de los métodos tradicionales a la hora de aprender o actualizarse de forma rápida y eficiente cada vez que llegan datos nuevos. De esta forma, los modelos basados en *online learning* aprenden de forma iterativa ayudándose del conocimiento adquirido en el pasado.

Estas técnicas suelen utilizarse en dos situaciones principales:

- Mejorar la eficiencia y escalabilidad (en términos de recursos de computación) de los modelos de regresión/clasificación, basados en métodos de *machine learning* tradicionales, para el procesado de flujos continuos de datos. Ejemplo de esto pueden ser los métodos de máquinas de soporte vectorial (SVM, *support vector machines*), que han sido explorados desde un punto de vista del aprendizaje *online* con algoritmos de *online learning* (Cauwenberghs y Poggio 2001; Shalev-Shwartz et al. 2011), haciéndolos más eficientes y escalables que los SVMs convencionales.

- Aplicar los algoritmos de *online learning* directamente a tareas de análisis en flujo continuo de datos, donde las observaciones van llegando de manera secuencial y la respuesta puede variar o evolucionar con el tiempo. Ejemplo de esto puede ser la regresión en series temporales, como la predicción del precio de las acciones, donde los datos llegan de forma periódica y hay que tomar decisiones inmediatamente antes de que llegue la siguiente observación.

2.2. Algoritmos online learning

Igual que los métodos de *machine learning* tradicionales, las técnicas de *online learning* pueden ser utilizadas para solucionar una amplia variedad de tareas. Desde la perspectiva de algoritmos específicos, podemos agruparlos en tres categorías principales:

- *Online learning* supervisado. Agrupa aquellos métodos centrados en tareas donde las observaciones (x, y)[1] llegan de forma secuencial y ambas (x e y) están disponibles. Esta categoría incluye tanto las tareas donde x e y están disponibles a la vez (de forma inmediata), y las tareas donde está disponible la x, y la respuesta y se retrasa algunos instantes (pudiendo llegar antes otras observaciones con su respectiva x).

- *Online learning* con respuesta limitada. Agrupa aquellos métodos centrados en tareas donde las observaciones (x, y) llegan de forma secuencial, si bien la respuesta y llega de manera limitada o parcial. Por ejemplo, en un una tarea de multiclasificación, en el caso de que sepamos si hemos acertado o no en nuestra predicción pero nunca lleguemos a saber el verdadero valor o etiqueta de y.

- *Online learning* no supervisado. Agrupa aquellos métodos centrados en tareas donde las observaciones x llegan de forma secuencial y no disponemos de una respuesta y en ningún momento.

En este trabajo nos centraremos en el caso de *online learning* supervisado. A continuación resumiremos brevemente las principales familias de métodos y algoritmos propuestos en la literatura científica reciente relacionada con este campo.

[1] Consideramos la observación (x, y), donde x es un vector $x = (x_1, \ldots, x_M)$ formado por los datos asociados a M variables predictoras, e y es el dato asociado a la variable respuesta (pudiendo ser un valor numérico (regresión) o categórico (clasificación)).

2.2.1. Algoritmos de online learning basados en métodos ensemble

Los métodos *ensemble* son muy empleados en *machine learning* tradicional. Ejemplos de estas técnicas podrían ser el *bagging* (Breiman 1996), *random forests* (Breiman 2001) o *boosting* (Freund y Schapire 1996). A partir de ellos, se han propuesto adaptaciones al contexto del flujo continuo de datos, como pueden ser:

DWM (*dynamic weighted majority*)	Kolter y Maloof (2003)
Online bagging y *online boosting*	Oza (2005)
Streaming random forests	Abdulsalam et al. (2007)
Dynamic streaming random forests	Abdulsalam et al. (2008)
Online coordinate boosting	Pelossof et al. (2009)
Leveraging bagging	Bifet et al. (2010a)
Online smooth-boost	Chen et al. (2012)
SAE2 (*social adaptive ensemble*)	Gomes y Enembreck (2014)
Optimal y *adaptive algorithms* para *online boosting*	Beygelzimer et al. (2015)
OBag (*online bagging*) de árboles *FIMT-DD*	Ikonomovska et al. (2015)
y ORF (*online random forest*) para regresión	
ARF (*adaptive random forests*)	Gomes et al. (2017,2018)

Cuadro 2.1: Algoritmos de *online learning* basados en métodos *ensemble* adaptados al flujo continuo de datos.

2.2.2. Algoritmos de online learning lineales

En este tipo de algoritmos consideramos un dominio de entrada $\mathcal{X}$ y un dominio de salida $\mathcal{Y}$ para una tarea de aprendizaje, y el objetivo es aprender una hipótesis $f : \mathcal{X} \to \mathcal{Y}$, donde suponemos que f es lineal. Podemos agrupar estos algoritmos en aquellos de primer orden y segundo orden.

- Algoritmos de *online learning* lineales de primer orden:

Perceptron	Rosenblatt (1958)
Winnow	Littlestone (1988)
ALMA (*approximate large margin algorithm*)	Gentile (2001)
ROMMA (*relaxed online maximum margin algorithms*)	Li y Long (2002)
OGD (*online gradient descent*)	Zinkevich (2003)
PA (*passive aggressive*)	Crammer et al. (2006)

Cuadro 2.2: Algoritmos de *online learning* lineales de primer orden.

- Algoritmos de *online learning* lineales de segundo orden:

SOP (*second order perceptron*)	Cesa-Bianchi et al. (2005)
CW (algoritmo de aprendizaje *confidence-weighted*)	Dredze et al. (2008)
AROW (*adaptive regularization of weight vectors*)	Crammer et al. (2009)
IELLIP (*improved ellipsoid method* para *online learning*)	Yang et al. (2009)
NHERD (*normal herding method via gaussian herding*)	Crammer y Lee (2010)
NAROW (*new variant of adaptive regularization*)	Orabona y Crammer (2010)
SCW (algoritmos de aprendizaje *soft confidence weighted*)	Wang et al. (2012a)

Cuadro 2.3: Algoritmos de *online learning* lineales de segundo orden.

2.2.3. Algoritmos de predicción con expert advice

La predicción con *expert advice* (asesoramiento de expertos) es un tema importante (Roughgarden y Schrijvers 2017) en *online learning* con numerosas aplicaciones. El funcionamiento general podría seguir el siguiente esquema. Un modelo tiene N expertos donde escoger, denotándolos $1, \ldots, N$. En cada paso t, el modelo decide una distribución $\mathbf{p}_t$ sobre los expertos, donde $p_{t,i}$ es el peso de cada experto i, y $\sum_{i=1}^{N} p_{t,i} = 1$. Cada experto sufre una pérdida $\ell_{t,i}$ debida al entorno. La pérdida total del modelo es $\sum_{i=1}^{N} p_{t,i}\ell_{t,i} = \mathbf{p}_t^\top \ell_t$, es decir la pérdida media ponderada de los expertos con respecto a la distribución $\mathbf{p}_t$ elegida por el modelo. Destacamos los siguientes algoritmos:

WM (algoritmo *weighted majority*)	Littlestone y Warmuth (1994)
Hedge	Freund y Schapire (1997)
EWAF (*exponentially weighted average forecaster*) y GF (*greedy forecaster*)	Cesa-Bianchi y Lugosi (2006)
Parameter-free online learning	Chaudhuri et al. (2009), Chernov y Vovk (2010)
Randomized multiplicative weights	Arora et al. (2012)

Cuadro 2.4: Algoritmos de *online learning* de predicción con *expert advice*.

2.2.4. Algoritmos de online learning con regularización

Los métodos de regularización imponen fuertes restricciones para limitar el número de pesos distintos de cero asociados a las posibles variables predictoras a considerar en el modelo. Entre los algoritmos de *online learning* con regularización destacamos los siguientes:

FOBOS (*forward looking subgradients*)	Duchi y Singer (2009)
TGD (*truncated gradient descent*)	Langford et al. (2009)
RDA (*regularized dual averaging*)	Xiao (2009)
Ada-RDA (*adaptive regularization* de RDA) y Ada-FOBOS (*adaptive regularization* de FOBOS)	Duchi et al. (2011)

Cuadro 2.5: Algoritmos de *online learning* con regularización.

2.2.5. Algoritmos de online learning con kernels

Como algoritmos de *online learning* con *kernels* (núcleos) podemos destacar los siguientes:

Kernelized perceptron	Freund y Schapire (1999)
Kernelized OGD	Kivinen et al. (2004)

Cuadro 2.6: Algoritmos de *online learning* con *kernels*.

Para mejorar la eficiencia y escalabilidad de estos algoritmos con núcleos, se presentan una familia de algoritmos de *online learning* con *kernels* escalables, que se pueden agrupar en aquellos que usan estrategias de *budget maintenance* (mantenimiento del presupuesto) y aquellos que emplean estrategias de *functional approximation* (aproximación funcional).

- Algoritmos de *online learning* con *kernels* escalables vía *budget maintenance*.

  Estos métodos se pueden agrupar según tres tipos de estrategias:

SV Removal	
RBP	Cavallanti et al. (2007)
BPA-S	Wang y Vucetic (2010)
BOGD	Zhao et al. (2012)
SV Projection	
Projectron y *projectron++*	Orabona et al. (2009)
BSGD+project	Wang et al. (2012b)
SV Merging	
BSGD+merge	Wang et al. (2012b)

Cuadro 2.7: Algoritmos de *online learning* con *kernels* escalables vía *budget maintenance*.

- Algoritmos de *online learning* con *kernels* escalables vía *functional approximation*:

FOGD (*Fourier online gradient descent*) y NOGD (*Nyström online gradient descent*)	Wang et al. (2013), Lu et al. (2016)

Cuadro 2.8: Algoritmos de *online learning* con *kernels* escalables vía *functional approximation*.

Por último, dentro de los algoritmos de *online learning* con *kernels*, cabe citar los algoritmos de *online learning* con múltiples *kernels* (OMKL) (Jin et al. 2010; Yang et al. 2012; Hoi et al. 2013), que combinan de forma automática múltiples *kernels* sin fijar ningún *kernel* predeterminado.

2.2.6. Algoritmos de online learning utilizando redes neuronales

Como algoritmo de *online learning* empleando redes neuronales (*neural networks*) destacamos el algoritmo eSNN (*evolving spiking neural networks*) propuesto por Lobo et al. (2018) por ser uno de los más recientes en el estado del arte.

2.3. Librerías online learning

En cuanto a la implementación de los algoritmos de *online learning*, cabe decir que en la actualidad hay escasez de *frameworks* y librerías de código abierto especializadas en la materia. A continuación, destacamos las siguientes implementadas en Python:

`Scikit-learn`	Pedregosa et al. (2011)
`Jubatus`	Hido (2012)
`Tornado`	Pesaranghader (2018)
`River`	Montiel et al. (2021)

Cuadro 2.9: Librerías públicas de *online learning*.

Además, cabe citar los siguientes *frameworks* de código abierto:

`MOA` (*massive online analysis*)	Bifet et al. (2010b)
`LIBOL` (*library for online learning algorithms*)	Hoi et al. (2014)

Cuadro 2.10: *Frameworks* de código abierto de *online learning*.

Capítulo 3

Adaptive random forests

En este capítulo se introducen el método *random forests* para posteriormente presentar el algoritmo de *online learning* ARF (*adaptive random forests*) como adaptación del método *random forests* al flujo continuo de datos.

3.1. Introducción random forests

Los árboles de decisión son métodos de aprendizaje supervisado empleados para realizar predicciones en problemas tanto de clasificación como regresión. La idea es ir particionando el espacio de entrada mediante divisiones sucesivas hasta obtener conjuntos aislados de muestras semejantes que se utilizarán para realizar las predicciones. Estos métodos tienen como ventajas su simplicidad y que son fácilmente interpretables. Sin embargo, su capacidad predictiva es baja (varianza grande).

A partir de la década de 1990 comienzan a usarse los métodos *ensemble*, consistentes en combinar las predicciones de muchos modelos, como pueden ser los árboles de decisión, con poca capacidad predictiva pero sencillos y fáciles de implementar, para obtener predicciones más potentes. De esta manera, se reduce mucho la varianza. El método *bagging* (*bootstrap aggregation*, Breiman 1996) fue uno de los primeros métodos en usarse. Consiste básicamente en generar muchas réplicas *bootstrap* a partir de una muestra de entrenamiento y con cada una de ellas entrenar un modelo para obtener predicciones. Combinando las predicciones de los modelos (por ejemplo promediando o por voto mayoritario) se obtienen las predicciones finales del modelo.

Una variante del *bagging* en el caso de emplear árboles de decisión como método pre-

dictivo es el algoritmo *random forests* (o bosques aleatorios, Breiman 2001). Para tratar de seguir reduciendo la varianza, el algoritmo *random forests* además de emplear *bootstrap* para generar réplicas con las que entrenar los modelos (árboles), en la construcción de estos árboles a la hora de generar las divisiones en el espacio predictor se restringen las posibles variables predictoras a utilizar en el corte, seleccionando m de forma aleatoria, con $m < M$, siendo M el número total de variables predictoras.

El algoritmo *random forests* es en la actualidad uno de los algoritmos más usados en *machine learning* en el contexto tradicional, aquel en el que se dispone del conjunto de datos entero a la vez (sin flujo continuo de datos). Esta preferencia es debida a su alto nivel de aprendizaje y bajas demandas en el aspecto de ajuste de hiperparámetros. Sin embargo, en el contexto del flujo continuo de datos no hay algoritmos de *random forests* que se puedan considerar punteros en comparación con los basados en *bagging* y *boosting*.

Las adaptaciones del *random forests* al flujo continuo de datos anteriores al año 2017 (ver capítulo 2) carecen de métodos de remuestreo y detección de derivas adecuados. Además, ninguno permite la resolución de problemas de regresión. Por ello, en este trabajo se va a examinar el rendimiento de ARF como algoritmo basado en el *random forests* adaptado al flujo continuo de datos.

3.2. ARF

A continuación se presenta ARF (*adaptive random forests*, Gomes et al. 2017) como algoritmo de regresión/clasificación para flujos continuos de datos. Empezaremos explicando ARF para clasificación, siendo análogo el caso de regresión con alguna diferencia que comentaremos más adelante. ARF presenta las siguientes características principales:

- Remuestreo adecuado adaptado al flujo continuo de datos.
- Restricción a un subconjunto aleatorio de tamaño m de las variables predictoras evaluadas para realizar las divisiones en cada nodo de los árboles.
- Incorporación de detectores de avisos y de derivas, con la posibilidad de entrenar árboles en un segundo plano.
- Voto mayoritario ponderado a la hora de realizar las predicciones.

3.2.1. Remuestreo para el flujo continuo de datos

El algoritmo *random forests* pueden hacer crecer muchos árboles sin caer en el sobreajuste usando *bagging* y selección aleatoria de posibles variables predictoras a la hora de realizar divisiones en los nodos. Sin embargo, esto requiere pasadas múltiples sobre los datos para crear una remuestra para cada árbol.

En el flujo continuo de datos no es factible realizar estas pasadas múltiples sobre los datos. Es por ello que una adaptación del método *random forests* al flujo continuo de datos requiere de un proceso *bagging* adecuado. Para entender el procedimiento que se usa en ARF, veamos primero cómo actúa el *bagging* en el caso de disponer de un conjunto de datos (sin flujo continuo de datos). Cada árbol se entrena con una remuestra *bootstrap* de tamaño Z generada escogiendo observaciones con reemplazamiento de forma aleatoria de una muestra de entrenamiento. En cada remuestra *bootstrap*, una observación aparece K veces con una probabilidad $P(K = k)$ siguiendo una distribución binomial[1]. Para valores de Z suficientemente grandes la distribución binomial tiende a una distribución Poisson[2] ($\lambda = 1$). Basándose en esto, Oza (2005) propuso el algoritmo de *online bagging*, de forma que el remuestreo del *bagging* original se sustituye mediante un remuestreo ponderado de manera que una observación aparece K veces con una probabilidad $P(K = k)$ que sigue una distribución Poisson ($\lambda = 1$), lo que equivale a que el árbol se entrene con esa observación K veces.

En ARF, en lugar de una Poisson ($\lambda = 1$), se utiliza una distribución Poisson ($\lambda = 6$), como en el *leveraging bagging* (Bifet et al. 2010a). Esto tiene el efecto práctico de emplear un mayor número de observaciones en el entrenamiento de los árboles, por lo que estos crearán más rápido divisiones en los nodos hoja, pudiendo así crecer de manera más profunda y también adaptarse más rápido a los posibles cambios. Para $\lambda = 1$, aproximadamente un

[1] Sea X una variable discreta que sigue una distribución binomial, $X \sim Bin(n, p)$, entonces su función de probabilidad viene dada por:

$$P(X = k) = \binom{n}{k} p^k (1-p)^{n-k}$$

para $k = 0, 1, 2, \ldots, n$ donde

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}.$$

[2] Sea X una variable discreta que sigue una distribución Poisson, $X \sim Poisson(\lambda)$, entonces su función de probabilidad viene dada por:

$$P(X = k) = \frac{\exp(-\lambda) \cdot \lambda^k}{k!}$$

para $k = 0, 1, 2, \ldots, n$.

37 % de los valores son 0 (por lo que las observaciones asociadas no se escogen), 37 % son 1 (se entrena con esa observación una vez) y 26 % son mayores que 1 (se entrena con esa observación repetidas veces). En cambio, para $\lambda = 6$, aproximadamente solo un 0.25 % de los valores son 0, por lo que el número de observaciones que no se usan en el entrenamiento del árbol es muy inferior.

Algorithm 1 RFTree Train. **Symbols:** λ: Fixed parameter to Poisson distribution; GP: Grace period before recalculating heuristics for split test.

```
1: function RFTreeTrain(m, t, x, y)
2:     k ← Poisson(λ = 6)
3:     if k > 0 then
4:         l ← FindLeaf(t, x)
5:         UpdateLeafCounts(l, x, k)
6:         if InstancesSeen(l) ≥ GP then
7:             AttemptSplit(l)
8:             if DidSplit(l) then
9:                 CreateChildren(l, m)
10:            end if
11:        end if
12:    end if
13: end function
```

Figura 3.1: Función *RFTreeTrain* para el entrenamiento de los árboles de ARF, extraída del artículo de Gomes et al. (2017).

3.2.2. Restricción de variables predictoras

La función empleada en ARF para el entrenamiento de los árboles se puede ver detallada en el algoritmo de la Figura 3.1, extraída del artículo de Gomes et al. (2017). Esta función (*RFTreeTrain*) está basada en el algoritmo del árbol de Hoeffding (Domingos y Hulten 2000), un árbol de decisión muy rápido, pero con algunas diferencias. *RFTreeTrain* no incluye ningún tipo de poda del árbol. Además, incorpora la restricción a un subconjunto aleatorio de tamaño m, con $m < M$, siendo M el número total de variables predictoras, de las variables evaluadas para intentar la división en un nodo hoja del árbol. Uno de los parámetros destacados de la función es GP (*grace period*, es una cantidad que hace referencia al número de observaciones en un nodo hoja, el cual debe superarse para actualizar los criterios de división en el nodo). Valores pequeños de GP provocarán actualizaciones más frecuentes de los criterios de división, lo que tenderá a la construcción de árboles más profundos, algo habitual e incluso deseable en el *random forests* debido a que tiende a aumentar la variabilidad entre árboles, y el combinar múltiples árboles evita caer en el sobreajuste del conjunto.

Por lo tanto, fijado m el número de variables evaluadas para intentar la división en un nodo hoja del árbol, entonces para un árbol t, y para una observación (x, y), la función *RFTreeTrain* sigue los siguientes pasos:

- Se asigna a k un valor aleatorio generado por una distribución Poisson ($\lambda = 6$), que nos indicará si esta observación se utilizará para entrenar el árbol o no. Si $k > 0$, entonces lo hará k veces.
- Se clasifica la observación según su x en el nodo hoja l, y se actualiza el número de observaciones que caen en ese nodo sumando k a la cuenta.
- Si el número de observaciones en ese nodo hoja es mayor que GP, entonces se intentará una división.
- Si se da la división, se crearán nodos hijos del nodo hoja l empleando la variable de entre las m evaluadas que más reduce la entropía (clasificación) o la varianza (regresión) entre el nodo padre y los nuevos nodos hijos creados.

3.2.3. Detección de avisos y derivas

Para lidiar con los flujos continuos de datos que pueden ir evolucionando, ARF incorpora un método de detección de posibles cambios o derivas en la distribución de los datos de entrada. De esta forma, ARF presenta dos detectores: un primer detector de avisos (*warning detector*) más sensible que detecta cambios incipientes en el error de predicción, a partir del cual se crea otro árbol que se empieza a entrenar desde cero en un segundo plano sin ser incorporado al conjunto de árboles y sin la influencia de sus predicciones; y un segundo detector de derivas (*drift detector*) más conservador que detecta una deriva significativa o importante en el error de predicción, a partir de la cual se sustituye el árbol que generó esa deriva, por el respectivo árbol que se estaba entrenando en segundo plano a partir del aviso previo, si hubo un aviso previo en ese árbol. Si no ha habido un aviso previo en ese árbol, se resetea por un árbol nuevo que se empieza a entrenar desde cero en el conjunto.

ARF permite la utilización de distintos métodos de detección, como pueden ser ADWIN (*Adaptive Windowing*, Bifet y Gavaldà 2007) o PHT (*Page Hinkley Test*, Page 1954). Utilizaremos ADWIN ya que nos permite simplificar el código empleando solo dos parámetros, δ_w nivel de confianza para el detector de avisos y δ_d nivel de confianza para el

detector de derivas, mientras que PHT requiere de un conjunto más amplio de parámetros.

Un detector ADWIN utiliza un parámetro $\delta \in (0,1)$, que es un nivel de confianza del detector, mientras que los valores de entrada son una secuencia (posiblemente infinita) $x_1, \ldots, x_t, \ldots$ El valor x_t está disponible en el instante t. Sea W una ventana deslizante que contiene los valores más recientes x_i. Consideramos una partición $W_0 \cdot W_1$ de W formada por dos subventanas disjuntas W_0 y W_1, donde W_1 contiene los valores x_i más recientes. La idea del método es ir calculando en cada instante t las medias muestrales $\hat{\mu}_{W_0}$ y $\hat{\mu}_{W_1}$ y comprobar si difieren lo suficiente, lo que indicaría un cambio. Nótese que cada vez que se añade un valor x_t las subventanas se mueven y se descartan el valor x_i más antiguo incluido en el anterior cálculo. Vamos a expresar esta idea de manera más exacta. Se definen:

$$m := \frac{2}{\frac{1}{|W_0|} + \frac{1}{|W_1|}},$$

$$\epsilon_{cut} := \sqrt{\frac{1}{2m} \cdot \ln \frac{4|W|}{\delta}},$$

donde m es la media armónica de $|W_0|$ y $|W_1|$, y ϵ_{cut} un valor que utilizamos como umbral. De esta forma, un test para el método ADWIN será comprobar si $|\hat{\mu}_{W_0} - \hat{\mu}_{W_1}| \geq \epsilon_{cut}$, que de cumplirse detecta un cambio.

3.2.4. Voto mayoritario ponderado

En ARF (para clasificación) vamos a emplear el voto mayoritario ponderado a la hora de realizar las predicciones. En ARF los árboles cada vez que llega una observación (x, y) siguen el procedimiento de evaluarse (realizar predicciones $\hat{y}$ con x y evaluarse empleando $\hat{y}$ e y) y luego entrenarse (con esa observación (x, y)). Esto permite que al árbol t se le pueda asignar el peso c_t/n_t, que es el cociente de las observaciones correctamente clasificadas c_t por el árbol t entre el total de observaciones vistas n_t por el árbol t desde la última sustitución o reseteo del mismo. Este peso refleja la precisión o rendimiento del árbol, por lo que al realizar las predicciones utilizando el voto mayoritario ponderado (se tomará como predicción la $\hat{y}$ mayoritaria una vez sumados los votos ponderados) los árboles que mejor se comportan tendrán más influencia y aquellos con peor rendimiento pesarán o contarán menos, hasta que no mejoren o sean sustituidos o reseteados.

De manera análoga al caso de los detectores, ARF es compatible con otro esquema de voto, como puede ser el voto mayoritario. En el caso de regresión (Gomes et al. 2018), utilizaremos la mediana (también se puede usar la media, pero usaremos la mediana por su

robustez) de las predicciones individuales para obtener las predicciones finales del conjunto de árboles.

Algorithm 2 Adaptive random forests. **Symbols:** m: maximum features evaluated per split; n: total number of trees ($n = |T|$); δ_w: warning threshold; δ_d: drift threshold; $c(\cdot)$: change detection method; S: Data stream; B: Set of background trees; $W(t)$: Tree t weight; $P(\cdot)$: Learning performance estimation function.

```
1: function ADAPTIVERANDOMFORESTS(m, n, δ_w, δ_d)
2:     T ← CreateTrees(n)
3:     W ← InitWeights(n)
4:     B ← ∅
5:     while HasNext(S) do
6:         (x, y) ← next(S)
7:         for all t ∈ T do
8:             ŷ ← predict(t, x)
9:             W(t) ← P(W(t), ŷ, y)
10:            RFTreeTrain(m, t, x, y)          ▷ Train t on the current instance (x, y)
11:            if C(δ_w, t, x, y) then          ▷ Warning detected?
12:                b ← CreateTree()             ▷ Init background tree
13:                B(t) ← b
14:            end if
15:            if C(δ_d, t, x, y) then          ▷ Drift detected?
16:                t ← B(t)                     ▷ Replace t by its background tree
17:            end if
18:        end for
19:        for all b ∈ B do                     ▷ Train each background tree
20:            RFTreeTrain(m, b, x, y)
21:        end for
22:    end while
23: end function
```

Figura 3.2: Algoritmo de ARF que muestra el funcionamiento del mismo, extraída del artículo de Gomes et al. (2017).

3.2.5. Funcionamiento general de ARF

El algoritmo de ARF (para clasificación) puede verse en la Figura 3.2, extraída del artículo de Gomes et al. (2017). Una vez vistas las principales características de ARF, estamos en disposición de ver el funcionamiento de este cada vez que le llega una nueva observación (x, y).

Se fijan los parámetros m (número de variables evaluadas para una posible división el un nodo hoja), n (número de árboles que componen el conjunto), δ_w (nivel de confianza

para el detector de avisos), y δ_d (nivel de confianza para el detector de derivas).

Para cada árbol $t \in T$ del conjunto:

- El árbol t predice $\hat{y}$ con la x de la observación.
- El árbol t actualiza el peso c_t/n_t, que es el cociente de las observaciones correctamente clasificadas c_t por el árbol t entre el total de observaciones vistas n_t por el árbol t desde la última sustitución o reseteo del mismo.
- El árbol t se entrena mediante la función *RFTreeTrain* con la observación (x, y).
- Si se detecta un aviso en el árbol t una vez actualizado el detector de avisos con la observación (x, y), se inicializa un árbol b en segundo plano B asociado al árbol t (esto es, $B(t) = b$).
- Si se detecta una deriva en el árbol t una vez actualizado el detector de avisos con la observación (x, y), se sustituye el árbol t por el respectivo árbol $B(t)$ que estaba en segundo plano a partir del aviso previo, si hubo un aviso previo en el árbol t, o se resetea por un árbol nuevo si no ha habido un aviso previo en el árbol t.

Para cada árbol $b \in B$ en segundo plano:

- El árbol b se entrena mediante la función *RFTreeTrain* con la observación (x, y).

Capítulo 4

Librería `River`

Este capítulo introduce la librería `River` y sus funciones más importantes para la realización de experimentos con ARF, que se abordarán en el siguiente capítulo.

4.1. Introducción a la librería `River`

`River` (Montiel et al. 2021) es una librería de *online learning* en Python especializada en el flujo continuo de datos. Es el resultado de la fusión entre las librerías `creme` (Halford et al. 2019) y `scikit-multiflow` (Montiel et al. 2018). Esta librería se presenta con las siguientes características:

- Progresiva. Las herramientas de la librería permiten ser actualizadas con una sola observación, y de esta forma ser empleadas en el flujo continuo de datos.

- Adaptativa. Como el flujo continuo de datos puede ir evolucionando, esta librería implementa métodos adaptativos para ofrecer robustez frente a los entornos cambiantes.

- Propósito general. Apta para diferentes problemas de *machine learning*: clasificación, regresión, aprendizaje no supervisado, etc.

- Eficiente y de fácil uso. Las técnicas implementadas para el flujo continuo de datos intentan gestionar de la forma más eficiente posible aspectos como la memoria y el tiempo dada la naturaleza ilimitada de este tipo de datos.

4.2. Funciones principales para la implementación de ARF en River

A continuación presentaremos las funciones principales para la utilización de ARF con esta librería.

4.2.1. Módulo ensemble

El módulo `ensemble` contiene implementadas las funciones de ARF para clasificación y regresión:

- `AdaptiveRandomForestClassifier` . Función de ARF para clasificación. Los parámetros más relevantes son los siguientes:
  - `n_models` . Por defecto `10`. Número de árboles del conjunto.
  - `max_features` . Por defecto `"sqrt"` (raíz cuadrada). Es el valor m, número de variables predictoras evaluadas para intentar la división en un nodo hoja del árbol. `"sqrt"` toma como m la raíz cuadrada del número total de variables predictoras. También permite las siguientes opciones:
    - `"log2"` . Toma el logaritmo (en base 2) del número total de variables predictoras.
    - `None` . Toma el número total de variables predictoras.
    - Un valor `int` (entero). Considera ese número.
    - Un valor `float` (número flotante). Considera la parte entera de multiplicar ese valor por el número total de variables predictoras.
  - `lambda_value` . Por defecto `6`. Valor de λ empleado para el remuestreo de los datos.
  - `metric` . Por defecto `Accuracy`. Métrica empleada para evaluar el rendimiento (las predicciones) de los árboles dentro del conjunto. Si se emplea voto mayoritario ponderado, esta métrica también se usa para asignar los pesos a los árboles.
  - `disable_weighted_vote` . Por defecto `False`. Si `True`, se desactiva el voto mayoritario ponderado, por lo que se usa el voto mayoritario.
  - `drift_detector` . Por defecto `ADWIN`. Método de detección de derivas (en el error de predicción). Si `None`, desactiva el método de detección de derivas.

- `warning_detector` . Por defecto `ADWIN`. Método de detección de avisos (cambios incipientes en el error de predicción). Si `None`, desactiva el método de detección de avisos.
- `grace_period` . Por defecto `50`. GP es una cantidad que hace referencia al número de observaciones en un nodo hoja, el cual debe superarse para actualizar los criterios de división en el nodo.
- `seed` . Por defecto `None`. Semilla. Si se especifica un valor `int` (entero), este se fija como semilla para el generador de números aleatorios.

- `AdaptiveRandomForestRegressor` . Función de ARF para regresión. Como parámetros adicionales a los citados en la función para clasificación, cabe comentar el siguiente:
  - `aggregation_method` Por defecto `median` (mediana). Método usado para realizar las predicciones del conjunto. Permite también `mean` (media).

  Además, para regresión el valor por defecto para `metric` es `MSE`.

4.2.2. Módulo `evaluate`

El módulo `evaluate` contiene implementadas funciones para la evaluación del rendimiento de un modelo. Para el caso de ARF utilizaremos la siguiente función:

- `progressive_val_score` . Función que evalúa el rendimiento de un modelo en un escenario de flujo continuo de datos que se determine. Los parámetros más relevantes son los siguientes:
  - `dataset` . Conjunto de datos que será transformado en un flujo continuo de datos.
  - `model` . Modelo a evaluar.
  - `metric` . Métrica empleada para evaluar el rendimiento (las predicciones) del modelo. El módulo `metrics` contiene implementadas diferentes métricas, de las cuales destacamos las siguientes:
    - `Accuracy`. Precisión o tasa de clasificación correcta (aciertos) del modelo.
    - `KappaM`. Métrica que compara la tasa de aciertos del modelo con la de un clasificador de clase mayoritaria (que predice $\hat{y}$ como la etiqueta y mayoritaria). Un valor de 1 indicaría máxima precisión y 0 una precisión igual a la de un clasificador de clase mayoritaria.

- `MSE`. Error cuadrático medio.
- `RMSE`. Raíz cuadrada del error cuadrático medio.
- `WeightedF1`. Computa la métrica F_1[1] para cada clase, y luego realiza un promedio ponderado de estas según la proporción de etiquetas y de cada clase.
- `ConfusionMatrix` . Matriz de confusión para clasificación, donde las filas corresponden a las etiquetas observadas y las columnas a las predicciones de dichas etiquetas.

- `delay` . Por defecto `None`. Indica el número de observaciones que tienen que llegar antes de que la y de la observación (x, y) actual esté disponible. Dicho de otra forma, el retraso en la disponibilidad de la y de una observación (x, y) entrante.
- `print_every` . Por defecto `0`. Indica la frecuencia con la que se muestra la métrica.
- `show_time` . Por defecto `False`. Indica si mostrar o no el tiempo empleado por el modelo.
- `show_memory` . Por defecto `False`. Indica si mostrar o no la memoria empleada por el modelo.
- `file` . Nombre de un archivo donde se escribirá el progreso de la evaluación del modelo.

4.2.3. Módulo `preprocessing`

El módulo `preprocessing` contiene implementadas funciones para el preprocesado de los datos. Para regresión, vamos a usar algunas de estas funciones para un mejor funcionamiento de ARF. Así, a cada observación (x, y) entrante se escalan las variables predictoras x antes de que sean empleadas en ARF. Destacamos las siguientes:

- `AdaptiveStandardScaler` . Escala los datos en un rango de $[0, 1]$ usando media y varianza móviles ponderadas exponencialmente (EWMA y EWMV[2]). Su único parámetro es:

[1] Dado un clasificador binario (que clasifica en dos clases, 1 (positivos) y 0 (negativos)), la métrica F_1 se define como:

$$F_1 = \frac{2 \cdot TP}{2 \cdot TP + FN + FP},$$

donde TP son los verdaderos positivos, FP los falsos positivos y FN los falsos negativos.

[2] El cálculo de EWMA y EWMV sigue el siguiente esquema:

  - `alpha` . Por defecto `0.3` . Valor α (entre $[0, 1]$) empleado en el cálculo de EWMA y EWMV. Cuanto más cerca esté `alpha` de `1`, mayor peso se le asigna a las observaciones más recientes.

- `MinMaxScaler` . Escala los datos en un rango de $[0, 1]$. Siendo x_{min} y x_{max} los valores mínimo y máximo, entonces cada dato x_i entrante se escala como $\frac{x_i - x_{min}}{x_{max} - x_{min}}$.

Sean $EWMA_1 = x_1; EWMVar_1 = 0; \delta_i = x_i - EWMA_{i-1}$;

$$EWMA_i = \alpha \cdot \delta_i + (1 - \alpha) \cdot EWMA_{i-1};$$
$$EWMVar_i = (1 - \alpha) \cdot \left(EWMVar_{i-1} + \alpha \cdot \delta_i^2\right);$$
$$EWMStd_i = \sqrt{EWMVar_i}.$$

Así, cada dato x_i entrante se escala como $\frac{x_i - EWMA_i}{EWMStd_i}$.

Capítulo 5

Experimentos

En este capítulo se describen las pruebas más destacadas realizadas con ARF a lo largo de la investigación. Para ello, se diseñan y programan experimentos en lenguaje Python (usando la versión 3.8.12) empleando la librería `River` (con la versión 0.8.0).

5.1. Descripción conjuntos de datos

A continuación se describen los conjuntos de datos empleados durante los experimentos llevados a cabo en este trabajo, tanto para la resolución de problemas de clasificación como regresión en flujos continuos de datos.

Para clasificación:

- Conjunto de datos *Steel Plates Faults*.

  El conjunto de datos *Steel Plates Faults* (defectos de placas de acero) se puede encontrar en el repositorio de *Machine Learning* de la UCI (Frank y Asuncion 2010). Cada muestra de este *dataset* representa un defecto superficial durante la fabricación de hojas de acero inoxidable. Contiene 1941 observaciones, y presenta 27 variables predictoras que representan la forma geométrica de cada defecto y su contorno: 25 numéricas y 2 categóricas de tipo *boolean*. El objetivo será predecir la variable categórica *Faults* (tipo de defecto), que presenta las siguientes 7 etiquetas:

Etiqueta	Nº observaciones
Pastry	158
Z_Scratch	190
K_Scatch	391
Stains	72
Dirtiness	55
Bumps	402
Other_Faults	673

Cuadro 5.1: Etiquetas de la variable categórica respuesta *Faults* (tipo de defecto) del conjunto de datos *Steel Plates Faults*, junto con el número de observaciones asociadas a cada etiqueta.

Las observaciones del conjunto de datos están agrupadas según la etiqueta correspondiente de la variable *Faults* y aparecen en el orden dispuesto en el Cuadro 5.1.

Para regresión:

- Conjuntos de datos de Gradiant.

  Durante el desarrollo de este trabajo, Gradiant ha proporcionado dos conjuntos de datos diferentes (Gradiant 1 y Gradiant 2, con 541 y 462 observaciones respectivamente). Ambos, obtenidos como resultado de la monitorización de un proceso de fabricación real, están formados por:

  - una serie de 13 covariables numéricas que registran de forma periódica los valores de una serie de parámetros de producción y máquinas (velocidades, temperaturas, presiones, etc.).
  - una variable respuesta (o *target*) numérica que mide y cuantifica la calidad de las piezas fabricadas.

  El objetivo de analizar estos datasets es el desarrollo de un modelo de regresión que, a partir de los valores de los diferentes parámetros de fabricación, estime la calidad

unitaria de las manufacturas en tiempo real.

Por razones de confidencialidad ha sido necesario anonimizar estos datos, de forma que ninguna información sensible sea expuesta públicamente en la redacción y presentación de este trabajo.

- Conjunto de datos `Bikes`.

  El conjunto de datos `Bikes` del módulo `datasets` de `River` contiene 182470 observaciones con información acerca del número de bicicletas alquiladas cada hora, entre los años 2011 y 2012, por los habitantes de Toulouse (Francia), además de la correspondiente información meteorológica de la ciudad. El objetivo será predecir la variable *bikes* (el número de bicicletas alquiladas) a partir de las siguientes variables predictoras (consideramos solo las 5 variables numéricas): *clouds* (nubosidad), *humidity* (humedad), *pressure* (presión atmosférica), *temperature* (temperatura) y *wind* (viento).

5.2. Descripción de los experimentos

Comenzamos detallando los experimentos realizados para clasificación y posteriormente describimos los desarrollados para el caso de regresión.

Experimentos para clasificación

5.2.1. Experimento 1

Empezamos los experimentos para clasificación empleando el conjunto de datos *Steel Plates Faults* presentado anteriormente. Utilizamos la función para el caso de clasificación `AdaptiveRandomForestClassifier` del módulo `ensemble` (ver capítulo 4, sección 4.2) con los siguientes parámetros : `n_models=50`, `max_features="sqrt"`, `lambda_value=6`, `grace_period=20` y `seed=42`. Implementamos un código que, recibiendo una a una las observaciones (x_i, y_i) con $i \in (1, N)$, siendo N el número total de observaciones (1941), sigue el siguiente esquema:

- Para el conjunto de observaciones x_i con $i \in (1, 10)$ (esto es, las primeras 10 observaciones):

1. Muestra el número de observación i y la verdadera etiqueta y_i.

2. El modelo se entrena con la observación (x_i, y_i).

- Para el conjunto de observaciones x_i con $i \in (11, N)$, con N el número total de muestras del *dataset*:

1. Muestra el número de observación i y su verdadera etiqueta y_i.

2. Muestra las probabilidades estimadas para cada una de las posibles etiquetas, $\hat{y}_i$, obtenidas a partir de la muestra x_i y el modelo entrenado con el conjunto de observaciones vistas hasta el momento (esto es, el conjunto de x_k con $k \in (11, \text{i})$).

3. Muestra la matriz de confusión actualizada con y_i e $\hat{y}_i$.

4. Muestra la precisión (*accuracy*) actualizada con y_i e $\hat{y}_i$.

5. El modelo se entrena con la nueva observación (x_i, y_i).

Este conjunto de datos tiene las observaciones ordenadas y agrupadas según su etiqueta y, por lo que lo interesante del experimento será comprobar cuánto tarda el modelo en ir aprendiendo las nuevas etiquetas a medida que van apareciendo. Vamos a ir viendo cómo se comporta el modelo con la aparición de las etiquetas:

- Hasta la observación 159 tan solo hay etiquetas de *Pastry*, por lo que el modelo predice siempre esta etiqueta.

- En la observación 159 aparece la etiqueta *Z_ Scratch*. El modelo sigue prediciendo la etiqueta *Pastry* hasta la observación 174, que predice por primera vez *Z_ Scratch*, como se puede ver en la Salida 5.1. A partir de aquí, excepto en un par de observaciones, el modelo va a seguir prediciendo correctamente *Z_ Scratch*.

```
-Instance 174 ,True label: Z_Scratch
Prediction: {'Z_Scratch': 0.5383760178669137, 'Pastry':
    0.46162398213308636}
              Pastry  Z_Scratch
     Pastry      148          0
  Z_Scratch       15          1
Accuracy: 90.85%

```

Salida 5.1: Salida de la observación 174 a partir del código realizado para el experimento 1.

- En la observación 349 aparece la etiqueta *K_ Scatch*. El modelo sigue prediciendo la etiqueta *Z_ Scratch* hasta la observación 352, que predice por primera vez *K_ Scatch*, como se puede ver en la Salida 5.2. A partir de aquí, excepto en 8 observaciones, el modelo va a seguir prediciendo correctamente *K_ Scatch*.

```
-Instance 352 ,True label: K_Scatch
Prediction: {'Z_Scratch': 0.2912006308545348, 'Pastry':
    0.018395549674421995, 'K_Scatch': 0.6904038194710432}
              K_Scatch     Pastry  Z_Scratch
   K_Scatch          1          0          3
     Pastry          0        148          0
  Z_Scratch          0         17        173
Accuracy: 94.15%

```

Salida 5.2: Salida de la observación 352 a partir del código realizado para el experimento 1.

- En la observación 740 aparece la etiqueta *Stains*. El modelo sigue prediciendo la etiqueta *K_ Scatch* hasta la observación 743, que predice por primera vez *Stains*, como se puede ver en la Salida 5.3. A partir de aquí, excepto en un par de observaciones, el modelo va a seguir prediciendo correctamente *Stains*.

```
-Instance 743 ,True label: Stains
Prediction: {'Z_Scratch': 0.04283895530615952, 'Pastry': 0.0, '
    K_Scatch': 0.42177596760779307, 'Stains': 0.5353850770860474}
              K_Scatch     Pastry     Stains  Z_Scratch
   K_Scatch        380          0          0         11
     Pastry          0        148          0          0
     Stains          3          0          1          0
  Z_Scratch          0         17          0        173
Accuracy: 95.77%

```

Salida 5.3: Salida de la observación 743 a partir del código realizado para el experimento 1.

- En la observación 812 aparece la etiqueta *Dirtiness*. El modelo predice mal hasta la observación 815, que predice por primera vez *Dirtiness*, como se puede ver en la Salida 5.4. A partir de aquí, salvo en 11 observaciones, el modelo va a seguir prediciendo correctamente *Dirtiness*.

```
-Instance 815 ,True label: Dirtiness
```

```
Prediction: {'Stains': 0.3551497922089512, 'Dirtiness':
    0.3713555131401978, 'Z_Scratch': 0.05846753960636464, 'K_Scatch':
    0.2150271550444862, 'Pastry': 0.0}
             Dirtiness   K_Scatch     Pastry     Stains  Z_Scratch
  Dirtiness          1          1          0          2          0
   K_Scatch          0        380          0          0         11
     Pastry          0          0        148          0          0
     Stains          0          5          0         67          0
  Z_Scratch          0          0         17          0        173
Accuracy: 95.53%

```

Salida 5.4: Salida de la observación 815 a partir del código realizado para el experimento 1.

- En la observación 867 aparece la etiqueta *Bumps*. El modelo predice mal hasta la observación 870, que predice por primera vez *Bumps*, como se puede ver en la Salida 5.5. A partir de aquí, salvo en 9 observaciones, el modelo va a seguir prediciendo correctamente *Bumps*.

```
-Instance 870 ,True label: Bumps
Prediction: {'Stains': 0.23167616323029727, 'Bumps':
    0.36149915945665906, 'Dirtiness': 0.3458550023327209, 'Z_Scratch':
     0.0, 'K_Scatch': 0.060969674980322824, 'Pastry': 0.0}
            Bumps  Dirtiness   K_Scatch     Pastry    Stains  Z_Scratch
    Bumps       1          2          0          0         1          0
Dirtiness       0         41          1          0        13          0
 K_Scatch       0          0        380          0         0         11
   Pastry       0          0          0        148         0          0
   Stains       0          0          5          0        67          0
Z_Scratch       0          0          0         17         0        173
Accuracy: 94.19%

```

Salida 5.5: Salida de la observación 870 a partir del código realizado para el experimento 1.

- En la observación 1269 aparece la etiqueta *Other_ Faults*. El modelo predice mal hasta la observación 1280, que predice por primera vez *Other_ Faults*, como se puede ver en la Salida 5.6. A partir de aquí, salvo en una observación, el modelo va a seguir prediciendo correctamente *Other_ Faults*.

```
-Instance 1280 ,True label: Other_Faults
```

```
Prediction: {'Other_Faults': 0.5260981483836024, 'Bumps':
    0.4493778831781073, 'Dirtiness': 0.02452396843829033, 'Stains':
    0.0, 'K_Scatch': 0.0}
            Bumps  Dirt.  K_Sc. Other_F. Pastry Stains Z_Scratch
       Bumps 390      9      0       0       0      3         0
   Dirtiness   0     41      1       0       0     13         0
    K_Scatch   0      0    380       0       0      0        11
Other_Faults  11      0      0       1       0      0         0
      Pastry   0      0      0       0     148      0         0
      Stains   0      0      5       0       0     67         0
   Z_Scratch   0      0      0       0      17      0       173
Accuracy: 94.49%

```

Salida 5.6: Salida de la observación 1280 a partir del código realizado para el experimento 1.

- Finalmente, utilizado todo el conjunto de datos (1941 observaciones), se obtiene una precisión (*accuracy*) del 96.32 % y la matriz de confusión mostrada en la Salida 5.7.

```
-Instance 1941 ,True label: Other_Faults
Prediction: {'Other_Faults': 1.0, 'Bumps': 0.0, 'Dirtiness': 0.0}
            Bumps  Dirt.  K_Sc. Other_F. Pastry Stains Z_Scratch
       Bumps 390      9      0       0       0      3         0
   Dirtiness   0     41      1       0       0     13         0
    K_Scatch   0      0    380       0       0      0        11
Other_Faults  12      0      0     661       0      0         0
      Pastry   0      0      0       0     148      0         0
      Stains   0      0      5       0       0     67         0
   Z_Scratch   0      0      0       0      17      0       173
Accuracy: 96.32%

```

Salida 5.7: Salida de la observación 1941 a partir del código realizado para el experimento 1.

De esta manera, apreciamos que ARF ofrece un rendimiento considerable con este conjunto de datos, en el cual las observaciones van llegando agrupadas y ordenadas según su etiqueta, y el modelo aprende bastante rápido cuando aparecen nuevas etiquetas.

5.2.2. Experimento 2

En este segundo experimento seguimos utilizando el conjunto de datos *Steel Plates Faults*. Nos centraremos en analizar la relevancia de distintos parámetros del modelo para

intentar detectar cuáles de ellos tienen una mayor influencia en el comportamiento de ARF.

Para esta tarea, vamos a utilizar de nuevo la función `AdaptiveRandomForestClassifier` del módulo `ensemble` (ver capítulo 4, sección 4.2) con los siguientes parámetros: `max_features="sqrt"` y `seed=42`. Analizaremos los parámetros GP (`grace_period`), el número de árboles del conjunto (`n_models`) y λ (`lambda_value`). Para cada uno de ellos, fijando el valor de los otros dos parámetros, compararemos el comportamiento del modelo empleando distintos valores del parámetro analizado. De esta forma, para cada caso, implementamos un código para el cual ARF recibe las muestras u observaciones (x, y) una a una y va actualizando la precisión (*accuracy*), tiempo y memoria del modelo. Representaremos esta información mediante gráficas.

5.2.2.1. Experimentos con GP (`grace_period`)

Para analizar el parámetro GP (`grace_period`), fijamos `n_models=50` y `lambda_value=6`. Para `grace_period`, tomamos 7 valores distintos: 3, 10, 20, 50, 100, 200 y 500. A continuación se muestran las gráficas de la precisión (*accuracy*), tiempo y memoria de los modelos, respectivamente.

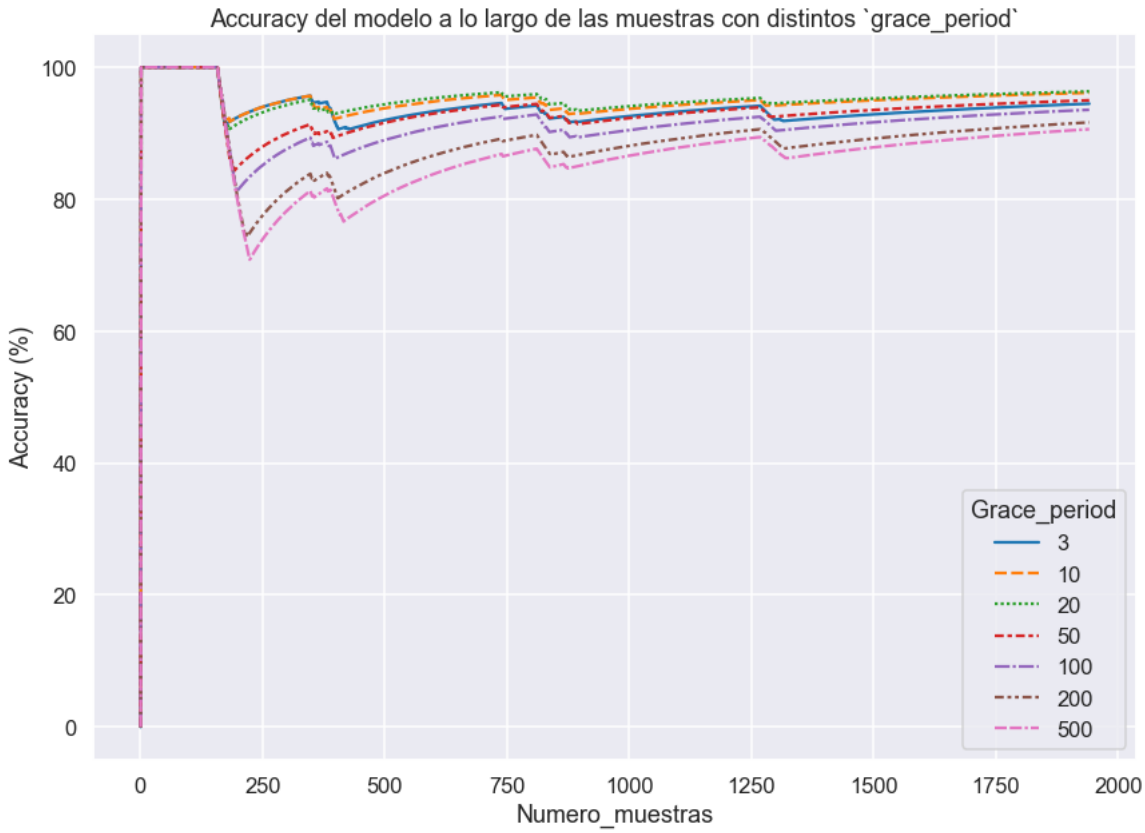

Figura 5.1: Gráfica que muestra la precisión (*accuracy*) del modelo a lo largo de las muestras empleando distintos valores del parámetro GP (`grace_period`).

En la Figura 5.1 podemos ver que en cuanto a la precisión (*accuracy*), los modelos que mejor se comportan son con `grace_period` = 20 y 10. Nótese también las significativas diferencias de precisión obtenidas con algunos valores del parámetro (por ejemplo, GP=200 y GP=50), además de cómo los valores de *accuracy* convergen lentamente (aunque a distintas velocidades) al mismo valor conforme aumenta el número de muestras recibidas.

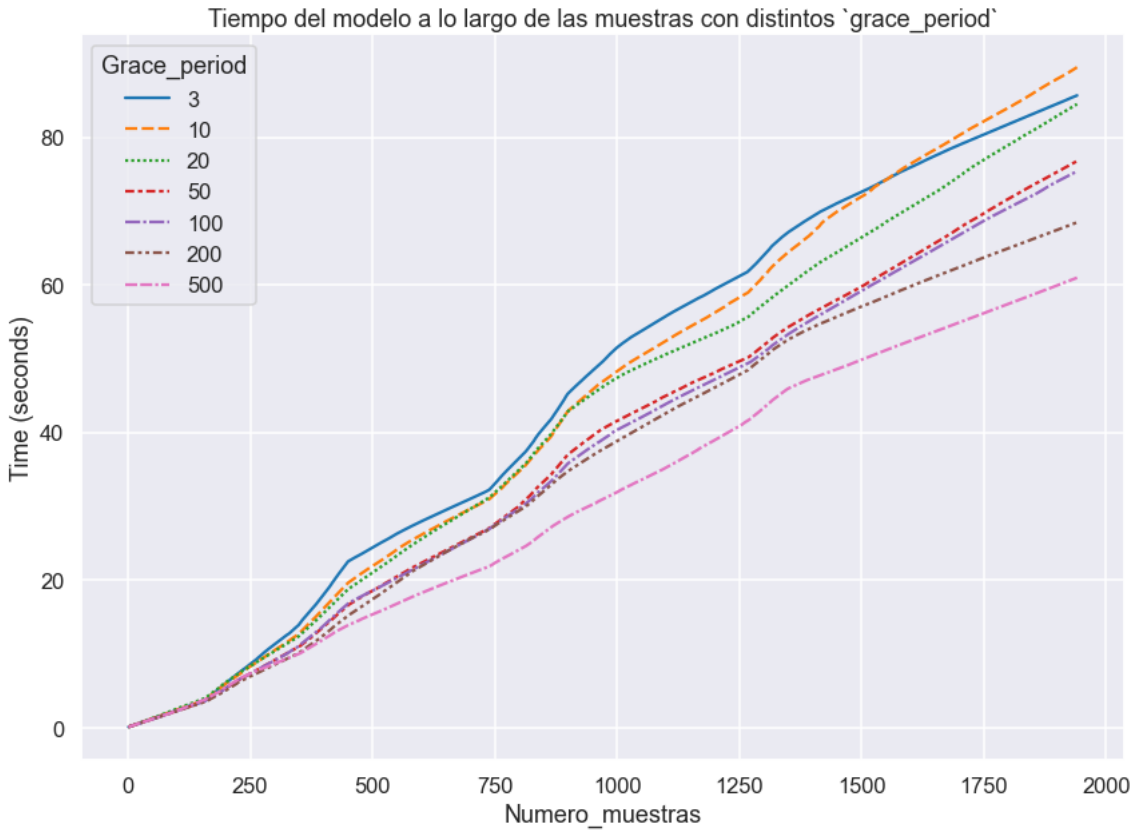

Figura 5.2: Gráfica que muestra el tiempo (*time*) en segundos empleado durante el entrenamiento del modelo a lo largo de las muestras empleando distintos valores del parámetro GP (`grace_period`).

En la Figura 5.2 podemos ver el tiempo empleado para el entrenamiento de los modelos conforme aumenta el número de muestras recibidas. Observamos que a medida que aumentamos el valor de `grace_period`, menos tarda el modelo en entrenarse.

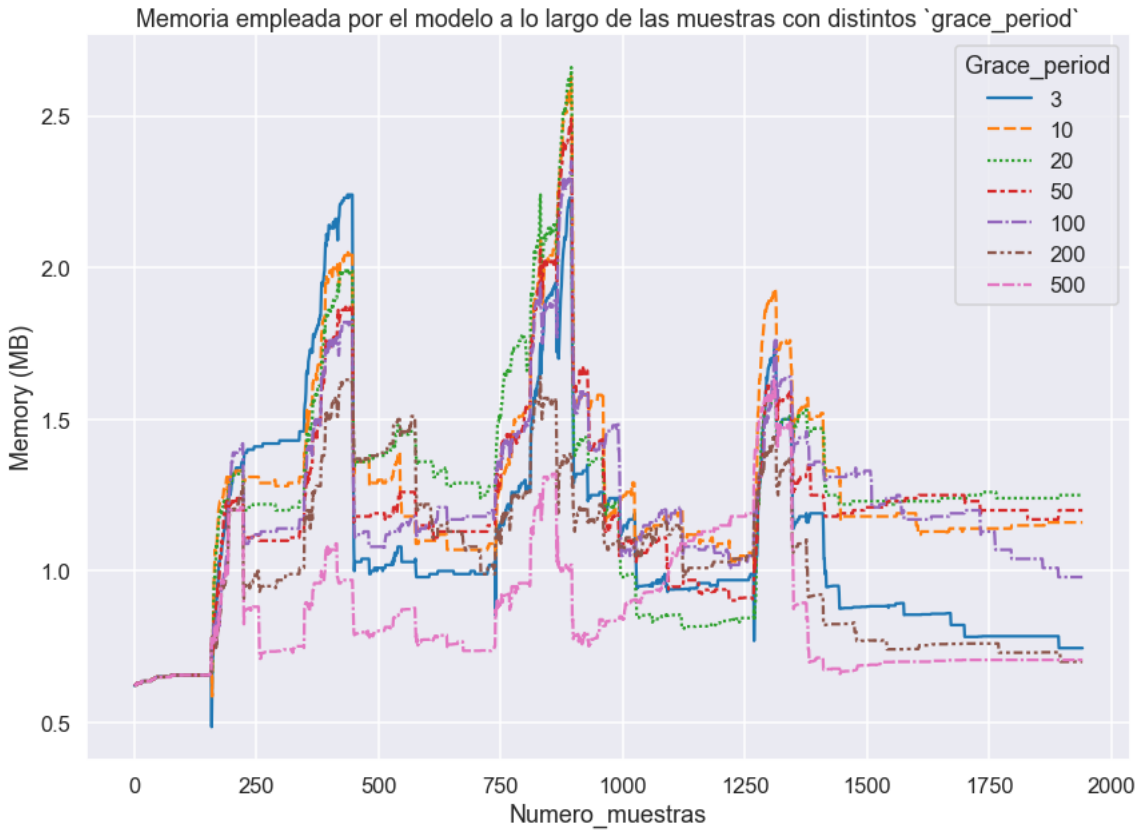

Figura 5.3: Gráfica que muestra la memoria (*memory*) en MB empleada por el modelo a lo largo de las muestras empleando distintos valores del parámetro *GP* (`grace_period`).

En la Figura 5.3 podemos ver la memoria empleada por los modelos a lo largo de las muestras. Observamos que todos los modelos sufren picos de memoria aproximadamente cuando las observaciones de *Steel Plates Faults* cambian de etiqueta. En estos instantes, el modelo se actualiza para aprender los nuevos patrones que le ayuden a predecir la nueva etiqueta, por lo que la memoria que necesita el modelo aumenta.

5.2.2.2. Experimentos con el nº de árboles del conjunto (`n_models`)

Para analizar el parámetro del número de árboles del conjunto (`n_models`), fijamos `grace_period=20` y `lambda_value=6`. Para `n_models`, tomamos 7 valores distintos: 3, 10, 50, 100, 200, 300 y 500. A continuación se muestran las gráficas de la precisión (*accuracy*), tiempo y memoria de los modelos, respectivamente.

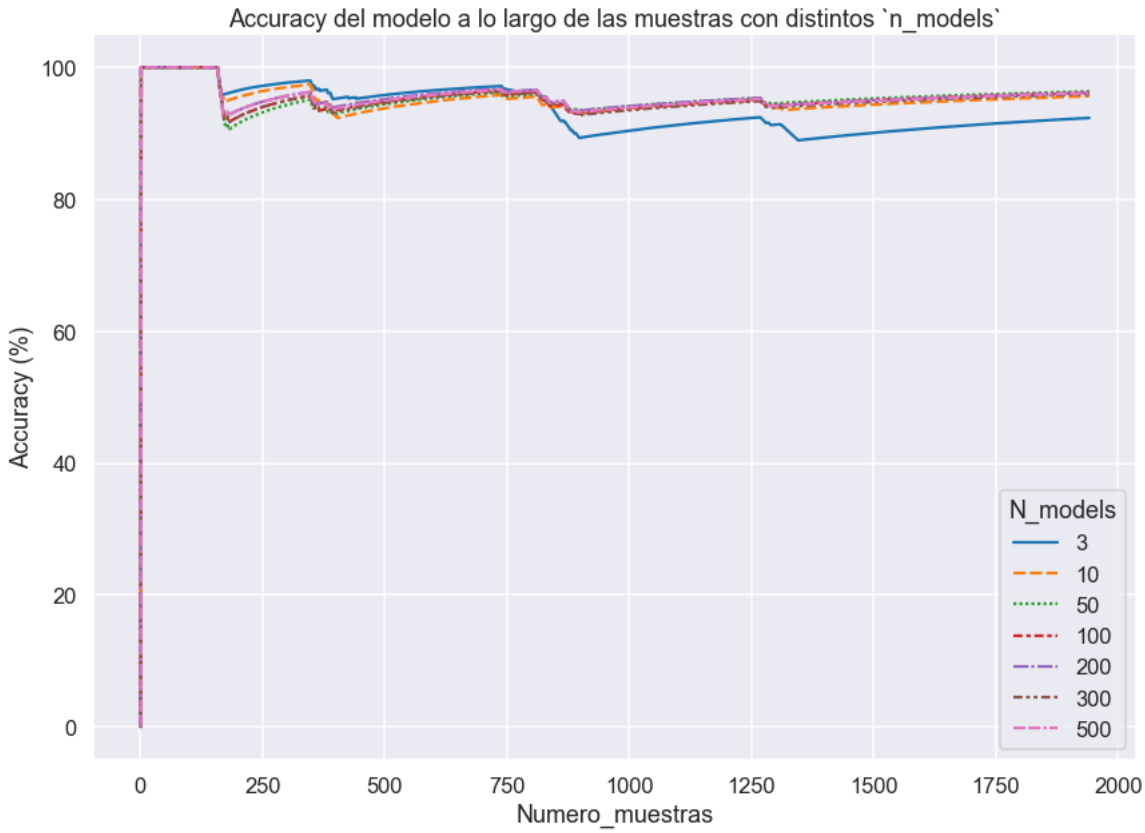

Figura 5.4: Gráfica que muestra la precisión (*accuracy*) del modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.4 podemos ver que en cuanto a la precisión *accuracy*, los modelos se comportan de forma muy similar, salvo en el modelo con `n_models`=3 que se comporta peor. El que mejor precisión (*accuracy*) presenta es el modelo con `n_models`=50.

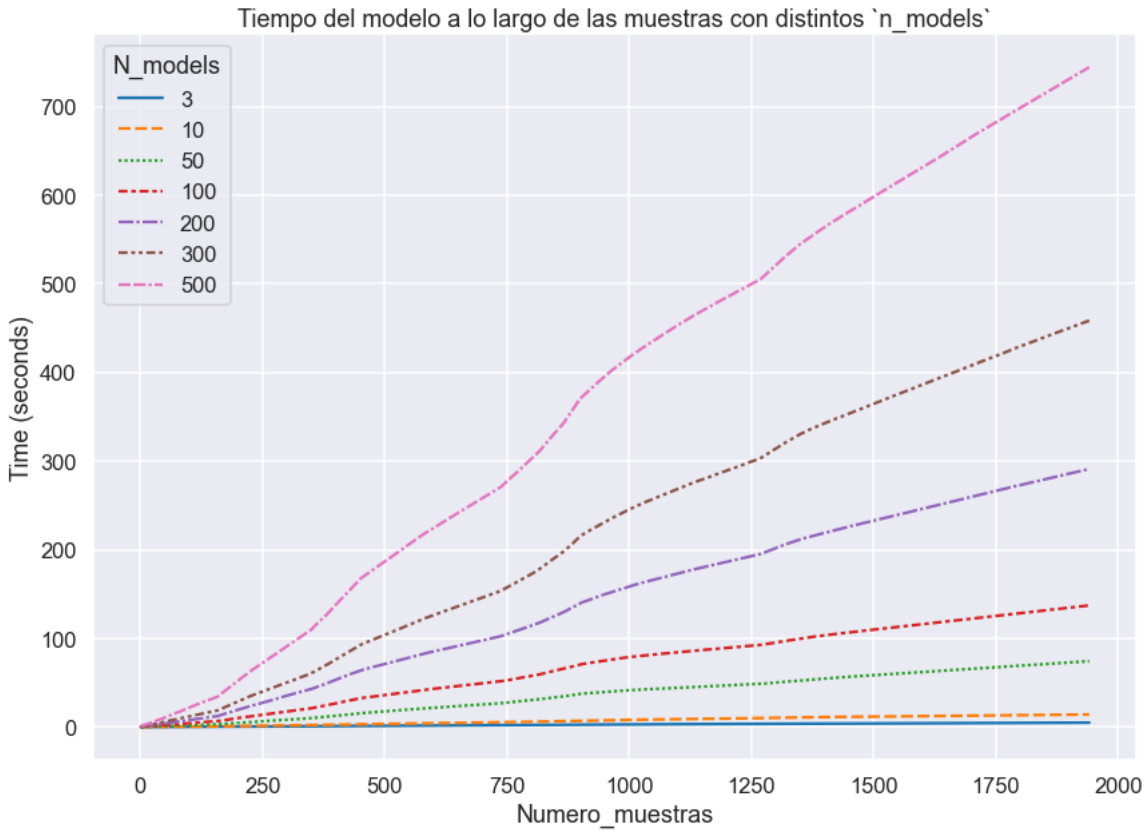

Figura 5.5: Gráfica que muestra el tiempo (*time*) en segundos empleado durante el entrenamiento del modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.5 podemos ver el tiempo empleado para el entrenamiento de los modelos a lo largo de las muestras. Observamos que a medida que aumentamos el valor de `n_models`, aumenta considerablemente el tiempo de entrenamiento.

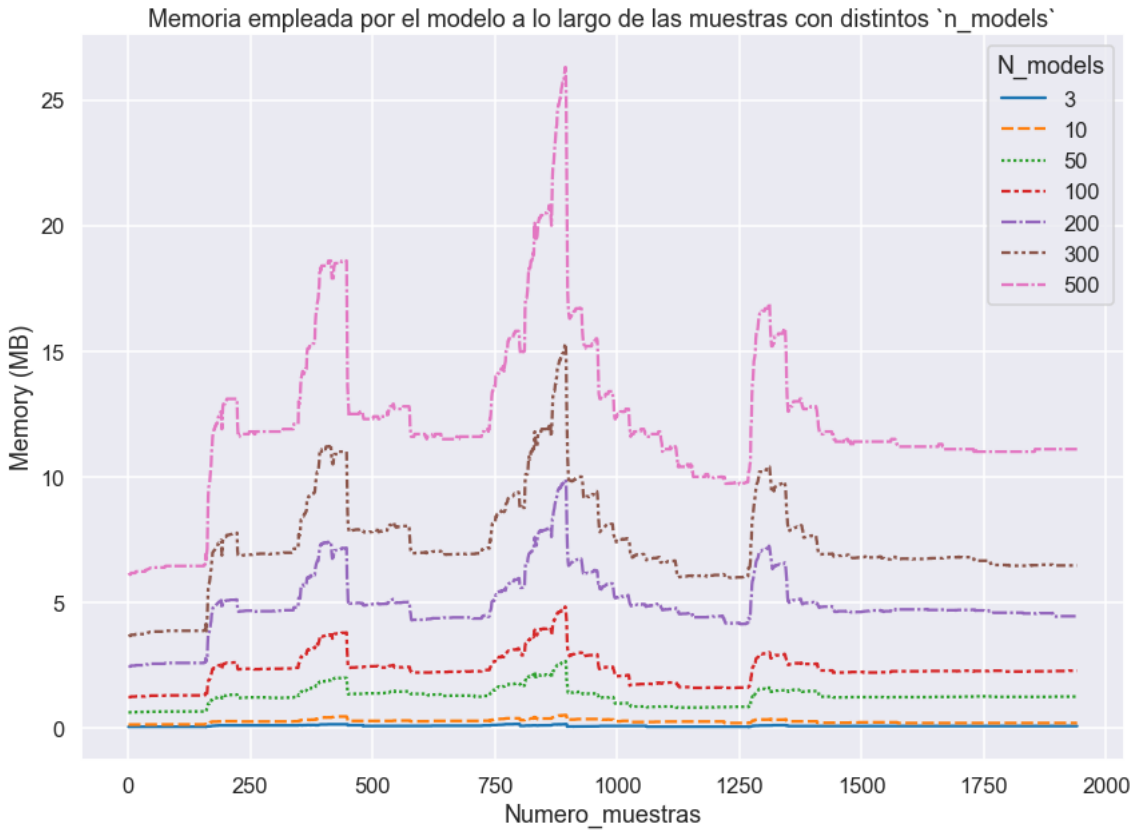

Figura 5.6: Gráfica que muestra la memoria (*memory*) en MB empleada por el modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.6 podemos ver la memoria empleada por los modelos a lo largo de las muestras. Observamos que todos los modelos sufren picos de memoria aproximadamente cuando las observaciones de *Steel Plates Faults* cambian de etiqueta, siendo los modelos con un mayor número de árboles los que siempre emplean una mayor memoria.

5.2.2.3. Experimentos con λ (`lambda_value`)

Para analizar el parámetro λ (`lambda_value`), fijamos `n_models=50` y `grace_period=20`. Para `lambda_value`, tomamos 7 valores distintos: 1, 2, 3, 5, 6, 7 y 8. A continuación se muestran las gráficas de la precisión (*accuracy*), tiempo y memoria de los modelos, respectivamente.

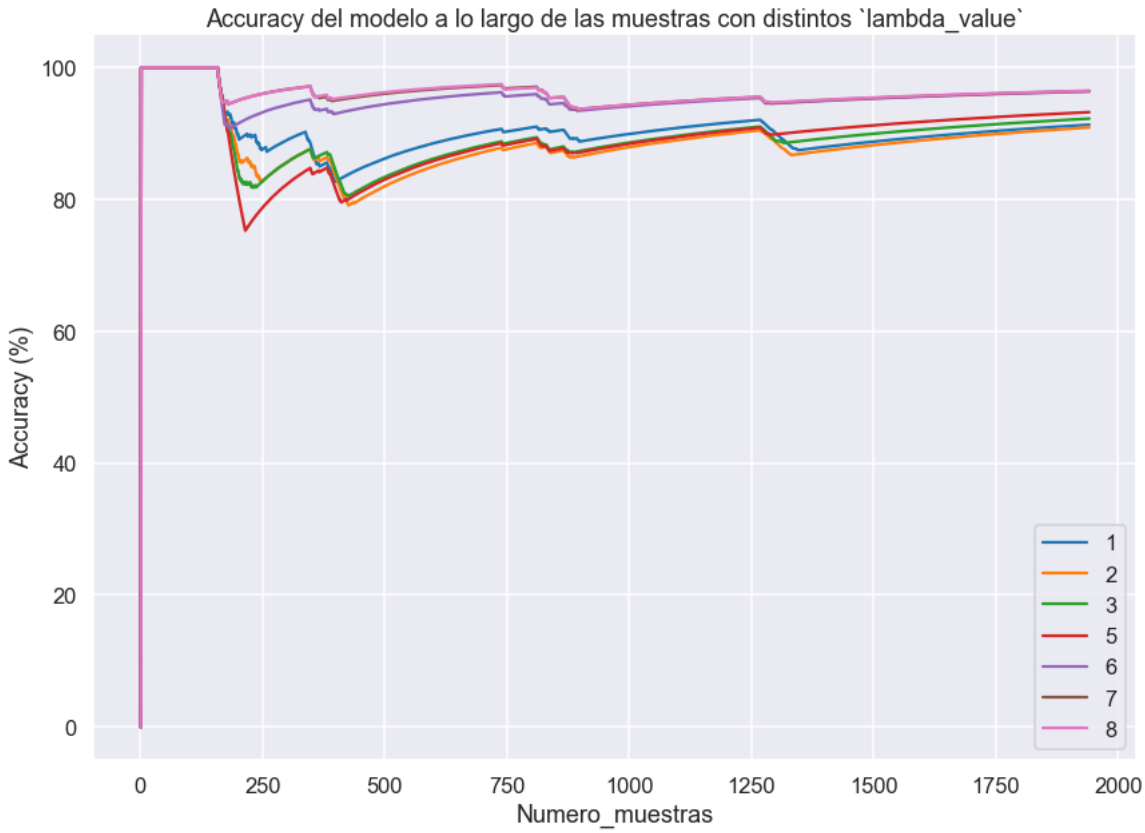

Figura 5.7: Gráfica que muestra la precisión (*accuracy*) del modelo a lo largo de las muestras empleando distintos valores del parámetro λ (`lambda_value`).

En la Figura 5.7 podemos ver que en cuanto a la precisión *accuracy*, los modelos se comportan de forma similar, siendo los modelos con `lambda_value`=8, 6 y 7 los que mejor precisión (*accuracy*) presentan.

Figura 5.8: Gráfica que muestra el tiempo (*time*) en segundos empleado durante el entrenamiento del modelo a lo largo de las muestras empleando distintos valores del parámetro λ (`lambda_value`).

En la Figura 5.8 podemos ver el tiempo empleado para entrenar los modelos a lo largo de las muestras. Observamos que a medida que aumentamos el valor de `lambda_value`, aumenta el tiempo de entrenamiento, si bien el que más tiempo emplea es el modelo con `lambda_value`=5. Nótese como, en general, valores pequeños de `lambda_value` producen tiempos de entrenamiento menores.

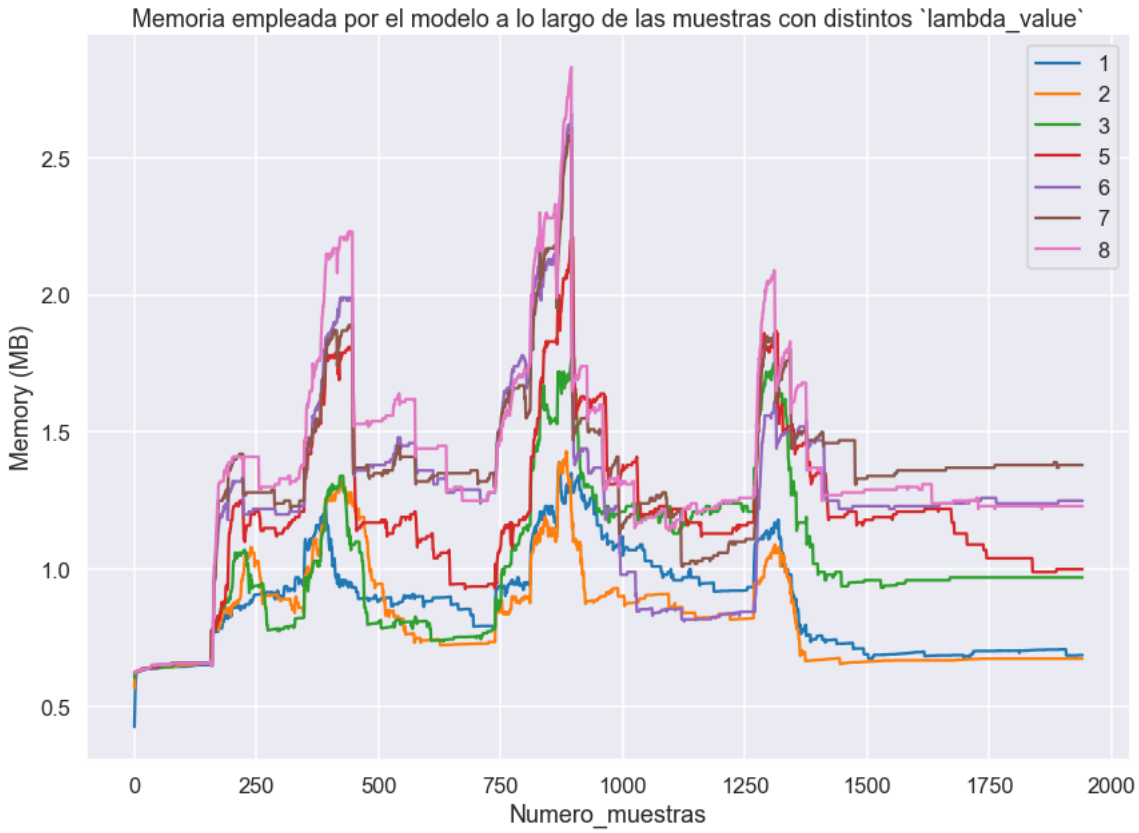

Figura 5.9: Gráfica que muestra la memoria (*memory*) en MB empleada por el modelo a lo largo de las muestras empleando distintos valores del parámetro λ (`lambda_value`).

En la Figura 5.9 podemos ver la memoria empleada por los modelos a lo largo de las muestras. Observamos que todos los modelos sufren picos de memoria aproximadamente cuando las observaciones de *Steel Plates Faults* cambian de etiqueta, siendo los modelos con un mayor valor de λ los que emplean una mayor memoria.

Realizadas estos experimentos, podemos considerar que el valor λ=6 que emplea ARF por defecto (como en *leveraging bagging*, Bifet et al. 2010a) ofrece un comportamiento correcto y no nos centraremos en realizar más experimentos con este parámetro.

5.2.3. Experimento 3

En este tercer experimento seguimos utilizando el conjunto de datos *Steel Plates Faults*. Visto el buen rendimiento de ARF con este conjunto de datos, nos preguntamos si este

comportamiento es por el buen funcionamiento de ARF o si es debido a *Steel Plates Faults* por tener las observaciones agrupadas y ordenadas según su etiqueta, lo que podría simplificar mucho la tarea de clasificación por parte del modelo.

Así, en este experimento emplearemos el conjunto *Steel Plates Faults* con las muestras desordenadas, de forma que las etiquetas de la variable objetivo *Faults* que queremos clasificar vayan apareciendo mezcladas en las observaciones. Para ello, remuestreamos sin reemplazamiento las observaciones, y repetimos con este nuevo conjunto de datos los experimentos realizados en el experimento 2 con los parámetros GP (`grace_period`) y número de árboles del conjunto (`n_models`).

Utilizamos de nuevo la función `AdaptiveRandomForestClassifier` del módulo `ensemble` (ver capítulo 4, sección 4.2) con los siguientes parámetros: `max_features="sqrt"`, `lambda_value=6` y `seed=42`. Analizaremos los parámetros GP (`grace_period`) y el número de árboles del conjunto (`n_models`). Para cada uno de ellos, fijando el valor del otro parámetro, compararemos el comportamiento del modelo empleando distintos valores del parámetro analizado. De esta forma, para cada caso, implementamos un código para el cual ARF recibe las muestras u observaciones (x, y) una a una y va actualizando la precisión (*accuracy*), tiempo y memoria del modelo. Representaremos esta información mediante gráficas.

5.2.3.1. Experimentos con GP (`grace_period`)

Para analizar el parámetro GP (`grace_period`), fijamos `n_models=50`. Para `grace_period`, tomamos 7 valores distintos: 3, 10, 20, 50, 100, 200 y 500. A continuación se muestran las gráficas de la precisión (*accuracy*), tiempo y memoria de los modelos, respectivamente.

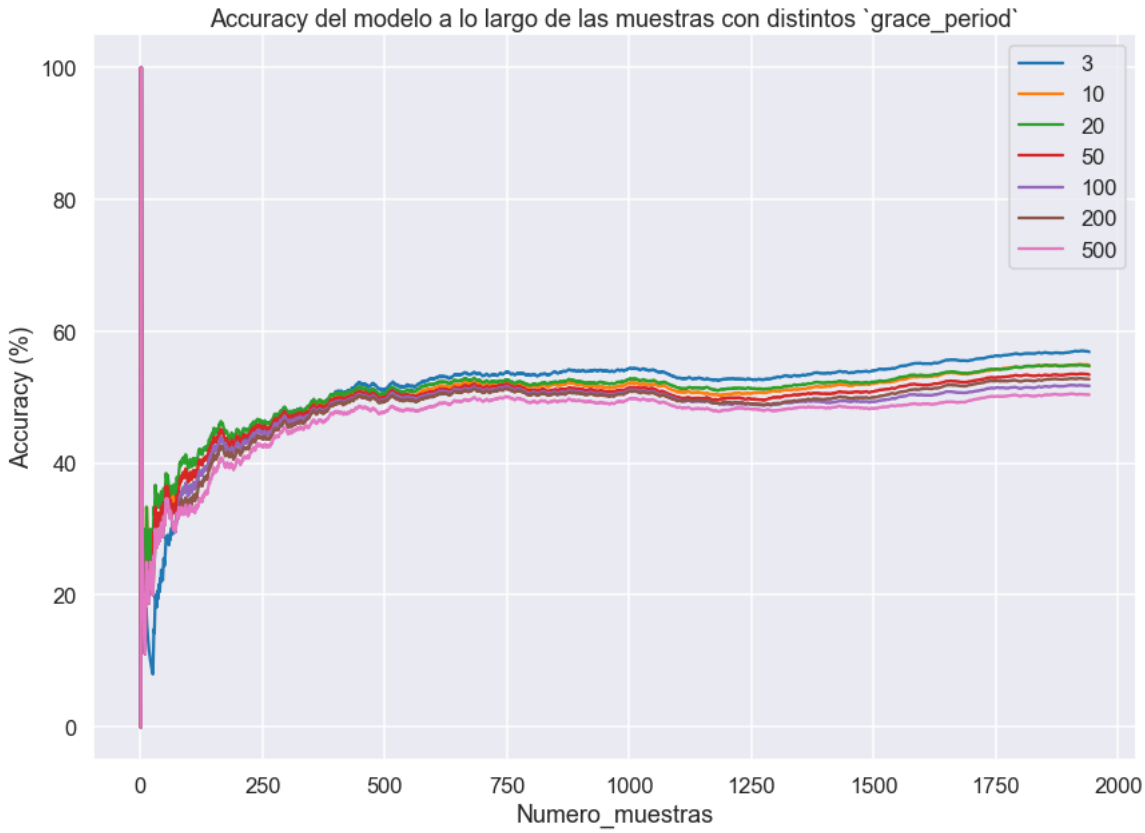

Figura 5.10: Gráfica que muestra la precisión (*accuracy*) del modelo a lo largo de las muestras empleando distintos valores del parámetro *GP* (`grace_period`).

En la Figura 5.10 podemos ver que todos los modelos presentan un comportamiento muy malo en cuanto a la precisión (*accuracy*), apenas superando el 50 % de clasificación correcta.

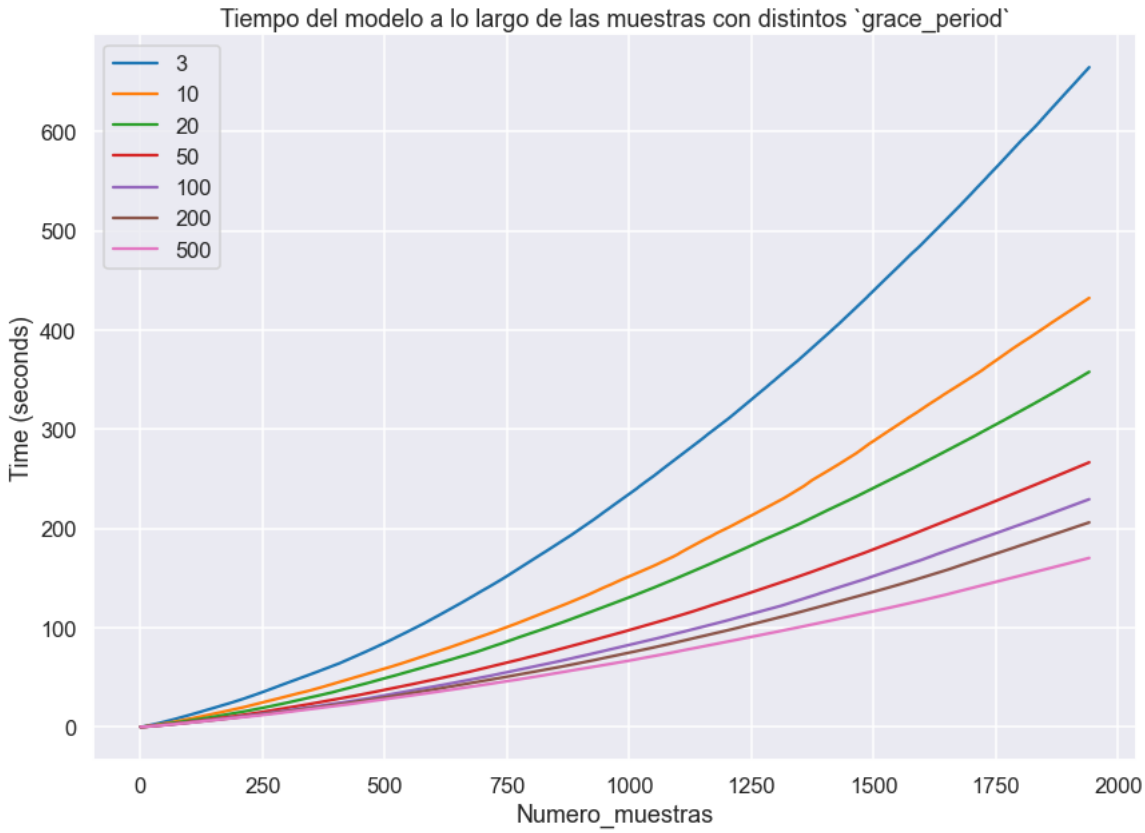

Figura 5.11: Gráfica que muestra el tiempo (*time*) en segundos empleado durante el entrenamiento del modelo a lo largo de las muestras empleando distintos valores del parámetro GP (`grace_period`).

En la Figura 5.11 podemos ver el tiempo empleado durante el entrenamiento de los modelos a lo largo de las muestras. Observamos que a lo largo de las muestras, el tiempo de entrenamiento aumenta de manera muy considerable, si bien a medida que aumentamos el valor de `grace_period`, menos tarda el modelo.

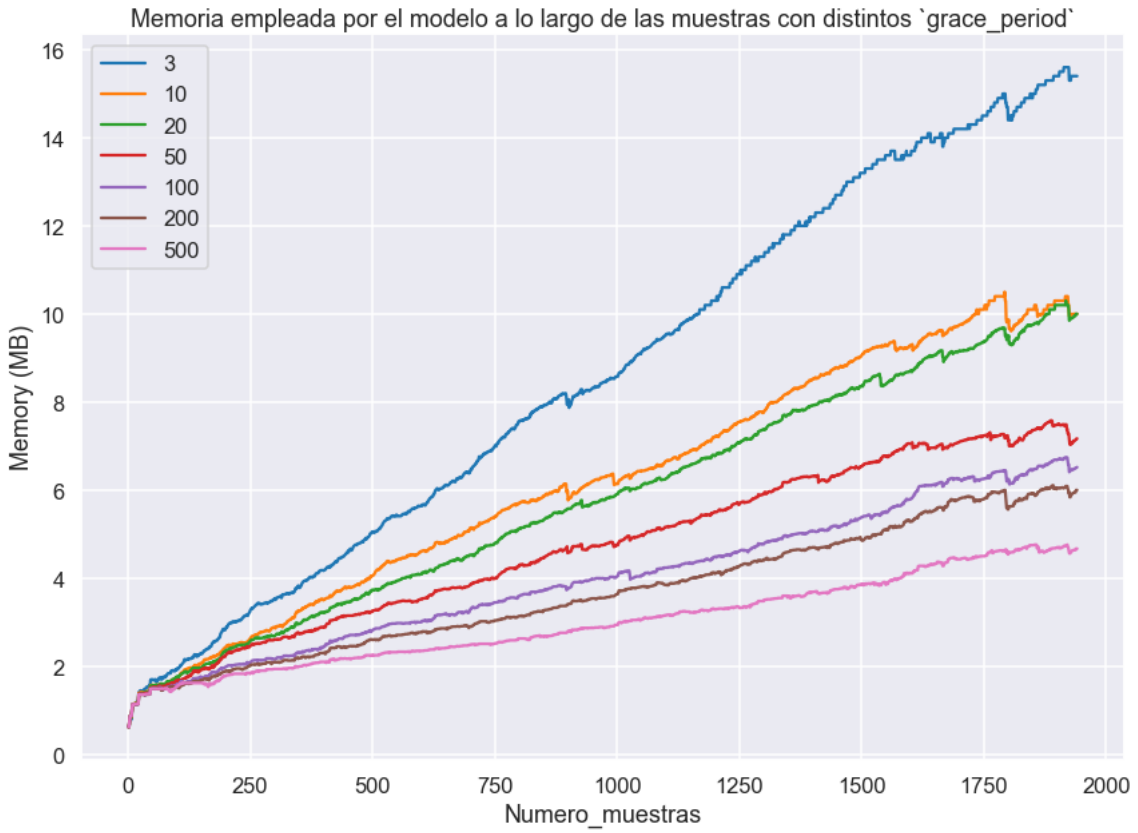

Figura 5.12: Gráfica que muestra la memoria (*memory*) en MB empleada por el modelo a lo largo de las muestras empleando distintos valores del parámetro *GP* (`grace_period`).

En la Figura 5.12 podemos ver la memoria empleada por los modelos a lo largo de las muestras. Observamos que en todos los modelos la memoria no para de aumentar y ya no se estabiliza como sucedía cuando las muestras con la misma etiqueta llegaban de forma consecutiva.

5.2.3.2. Experimentos con el nº de árboles del conjunto (`n_models`)

Para analizar el parámetro del número de árboles del conjunto (`n_models`), fijamos `grace_period=20`. Para `n_models`, tomamos 7 valores distintos: 3, 10, 50, 100, 200, 300 y 500. A continuación se muestran las gráficas de la precisión (*accuracy*), tiempo y memoria de los modelos, respectivamente.

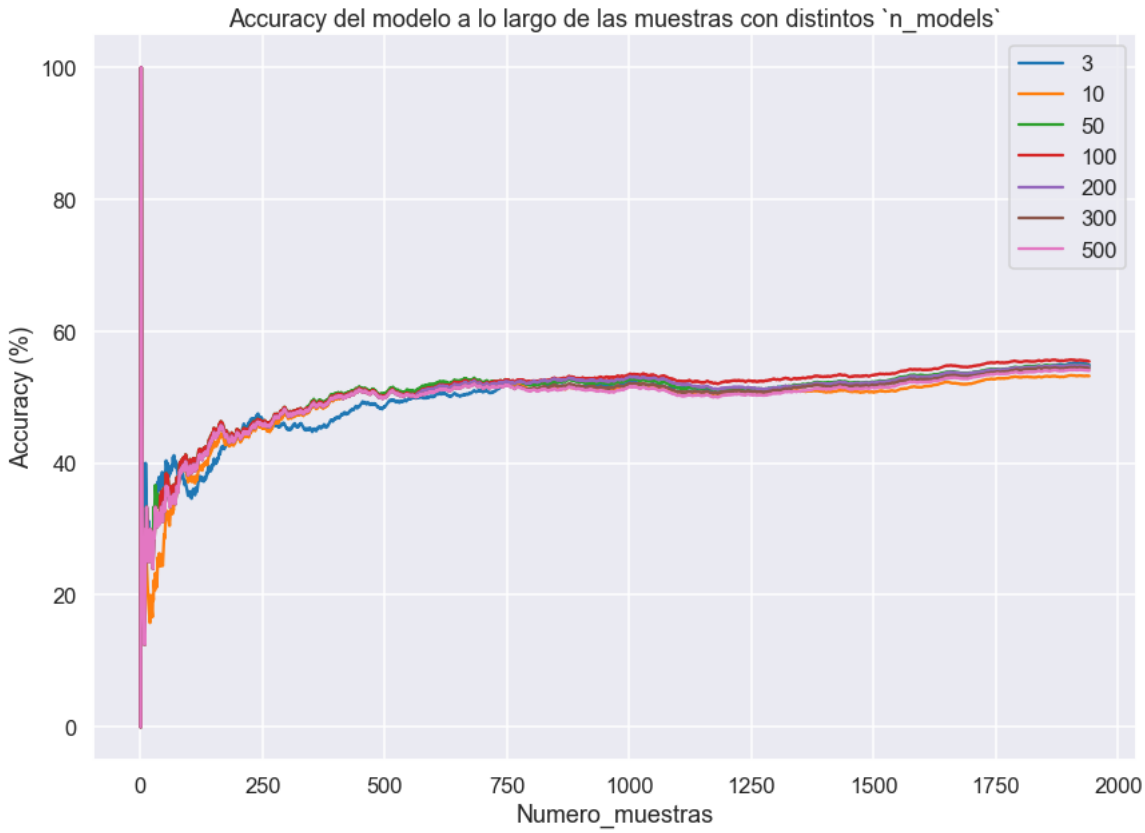

Figura 5.13: Gráfica que muestra la precisión (*accuracy*) del modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.13 podemos ver que todos los modelos presentan un comportamiento muy malo en cuanto a la precisión (*accuracy*), apenas superando el 50 % de clasificación correcta.

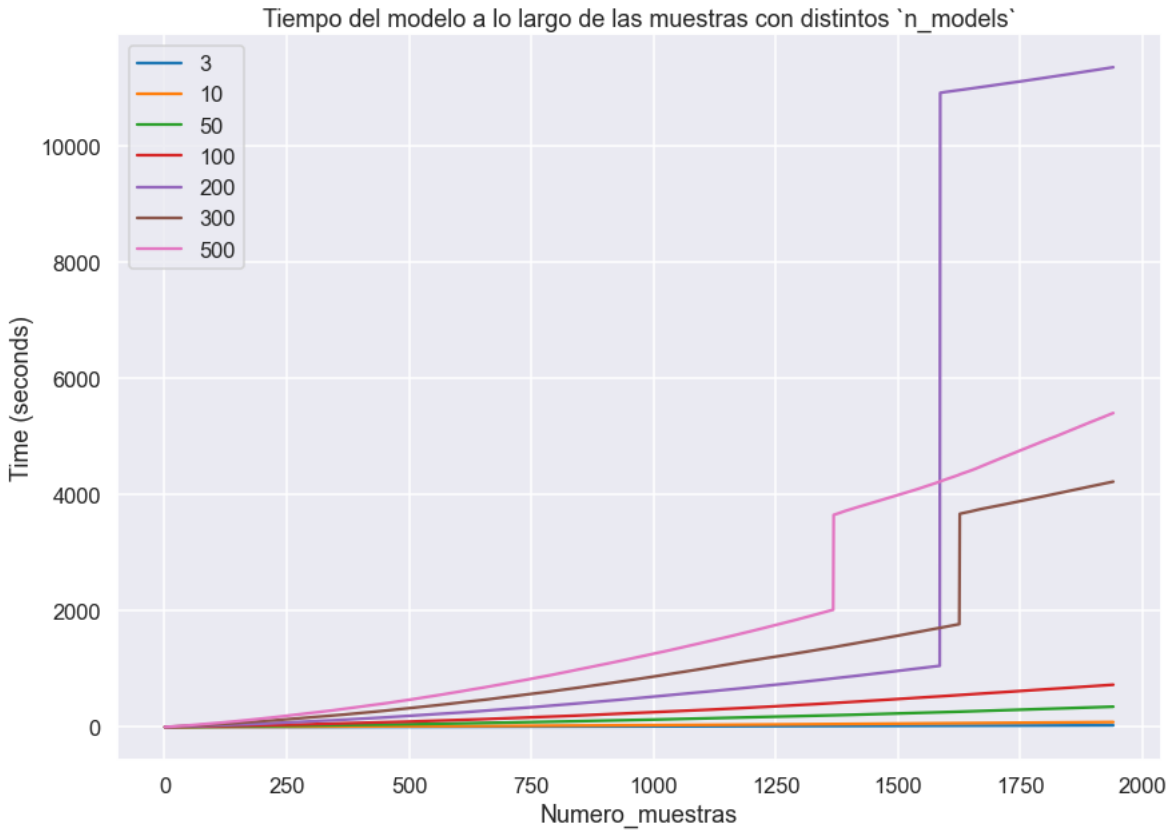

Figura 5.14: Gráfica que muestra el tiempo (*time*) en segundos empleado durante el entrenamiento del modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.14 podemos ver el tiempo de entrenamiento a lo largo de las muestras. Observamos que, con `n_models`=200, 500 y 300, el tiempo de entrenamiento aumenta considerablemente.

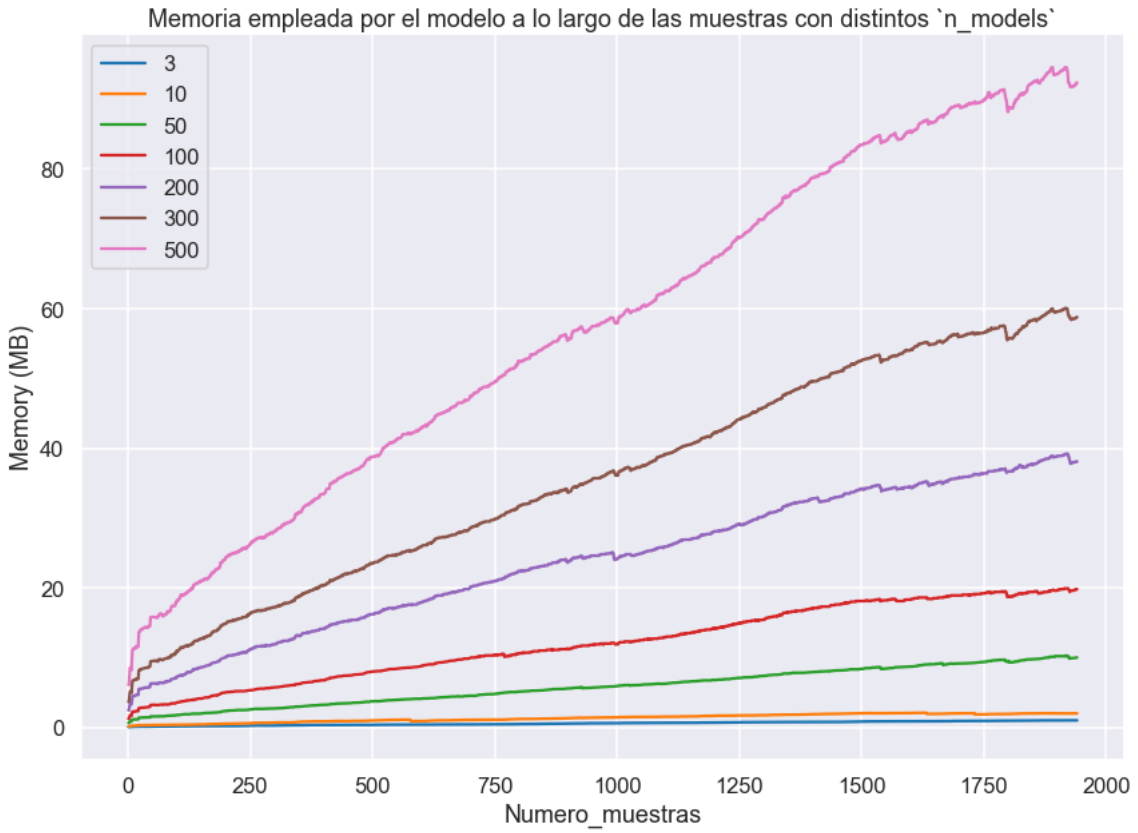

Figura 5.15: Gráfica que muestra la memoria (*memory*) en MB empleada por el modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.15 podemos ver la memoria empleada por los modelos a lo largo de las muestras. Observamos que, salvo con pocos árboles en el conjunto (3 o 10), en el resto de los modelos la memoria no para de aumentar, sobre todo el modelos con muchos árboles, y ya no se estabiliza como sucedía cuando las muestras con la misma etiqueta llegaban de forma consecutiva.

A la vista de estos experimentos, vemos que al desordenar las etiquetas, ARF ya no ofrece un buen rendimiento y se comporta muy mal, apenas superando el 50 % de tasa de clasificación correcta. Además, observamos que la memoria no se estabiliza en ningún momento y no para de aumentar, síntoma de que ARF no termina de aprender correctamente.

Experimentos para regresión

A continuación se detallan los experimentos realizados para regresión.

5.2.4. Experimento 4

Empezamos los experimentos para regresión empleando los conjuntos de datos Gradiant 1 y Gradiant 2 presentados con anterioridad. Utilizamos la función para el caso de regresión `AdaptiveRandomForestRegressor` del módulo `ensemble` (ver capítulo 4, sección 4.2) con los parámetros: `max_features="sqrt"`, `lambda_value=6`, `grace_period=20` y `seed=42`. Además, para un mejor funcionamiento de ARF, emplearemos la función `AdaptiveStandardScaler` con el parámetro `alpha=0.3` (valor por defecto) para el preprocesado de los datos, de manera que las observaciones se escalan en un rango de $[0, 1]$ como paso previo al entrenamiento del modelo.

Para ambos conjuntos de datos, Gradiant 1 y Gradiant 2, analizaremos el parámetro de número de árboles del conjunto (`n_models`), comparando el comportamiento del modelo empleando 7 valores distintos del parámetro: 3, 10, 50, 100, 200, 300 y 500. De esta forma, implementamos un código para el cual ARF recibe las muestras u observaciones (x, y) una a una (tras ser escaladas) y va actualizando el RMSE (*root mean squared error*, raíz cuadrada del error cuadrático medio) y la memoria del modelo. Representaremos esta información mediante gráficas.

5.2.4.1. Experimentos con Gradiant 1

A continuación se muestran, para el conjunto de datos Gradiant 1, las gráficas del RMSE y la memoria del modelo, respectivamente.

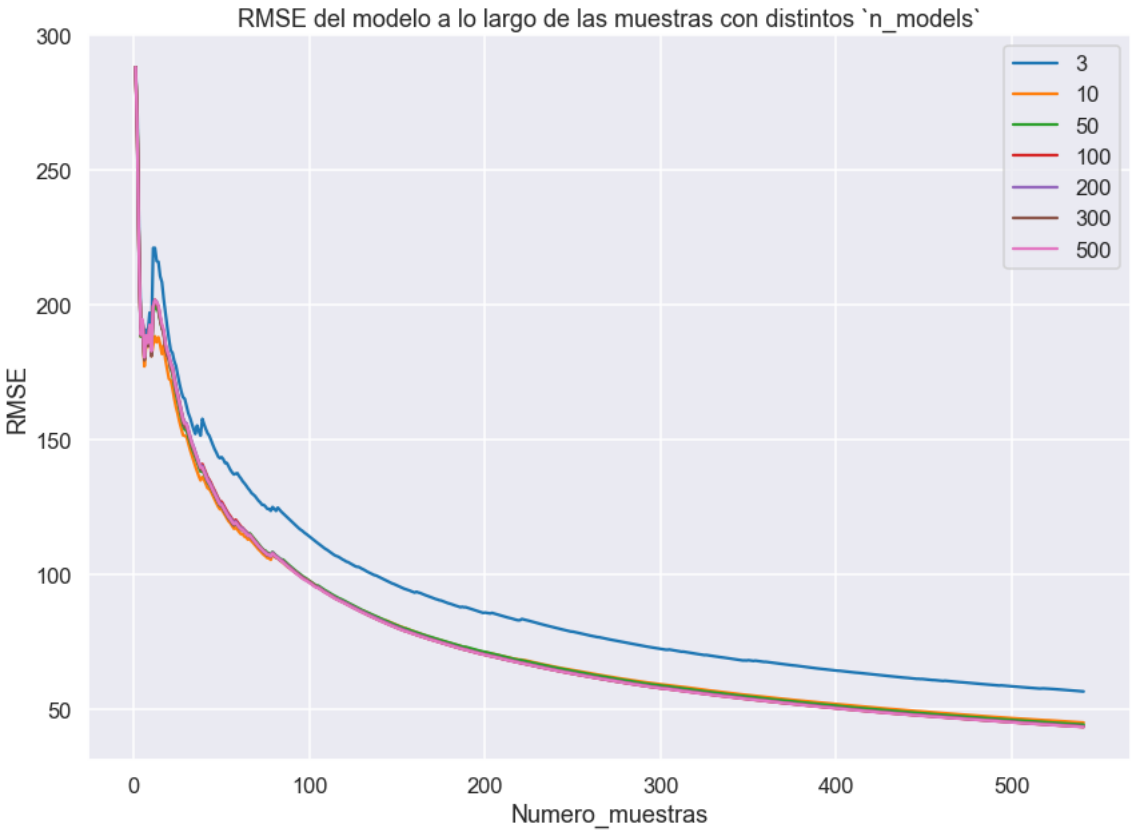

Figura 5.16: Gráfica que muestra el RMSE del modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.16 podemos ver que todos los modelos presentan un comportamiento similar en cuanto al RMSE excepto con `n_models`=3 que se comporta peor.

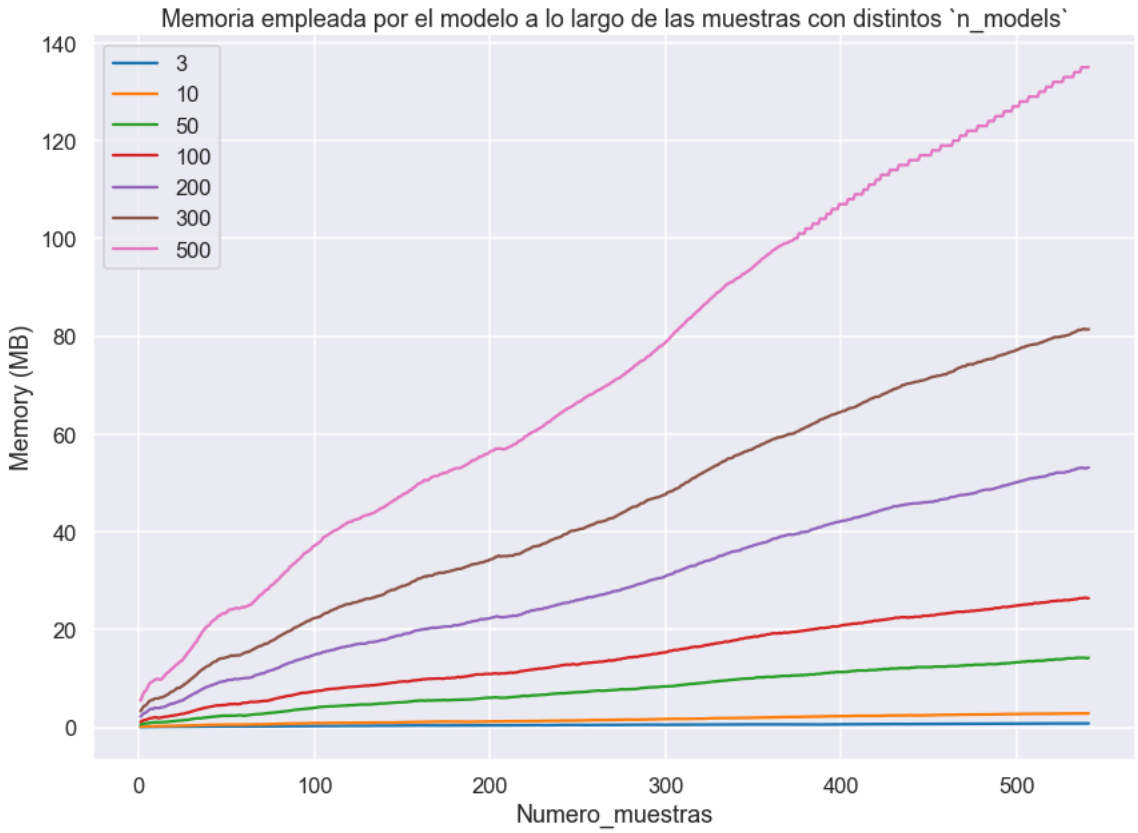

Figura 5.17: Gráfica que muestra la memoria (*memory*) en MB empleada por el modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.17 podemos ver la memoria empleada por los modelos a lo largo de las muestras. Observamos que, salvo con pocos árboles en el conjunto (3 o 10), en el resto de los modelos la memoria no para de aumentar, sobre todo el modelos con muchos árboles, y no se estabiliza en ningún momento.

5.2.4.2. Experimentos con Gradiant 2

A continuación se muestran, para el conjunto de datos Gradiant 2, las gráficas del RMSE y la memoria del modelo, respectivamente.

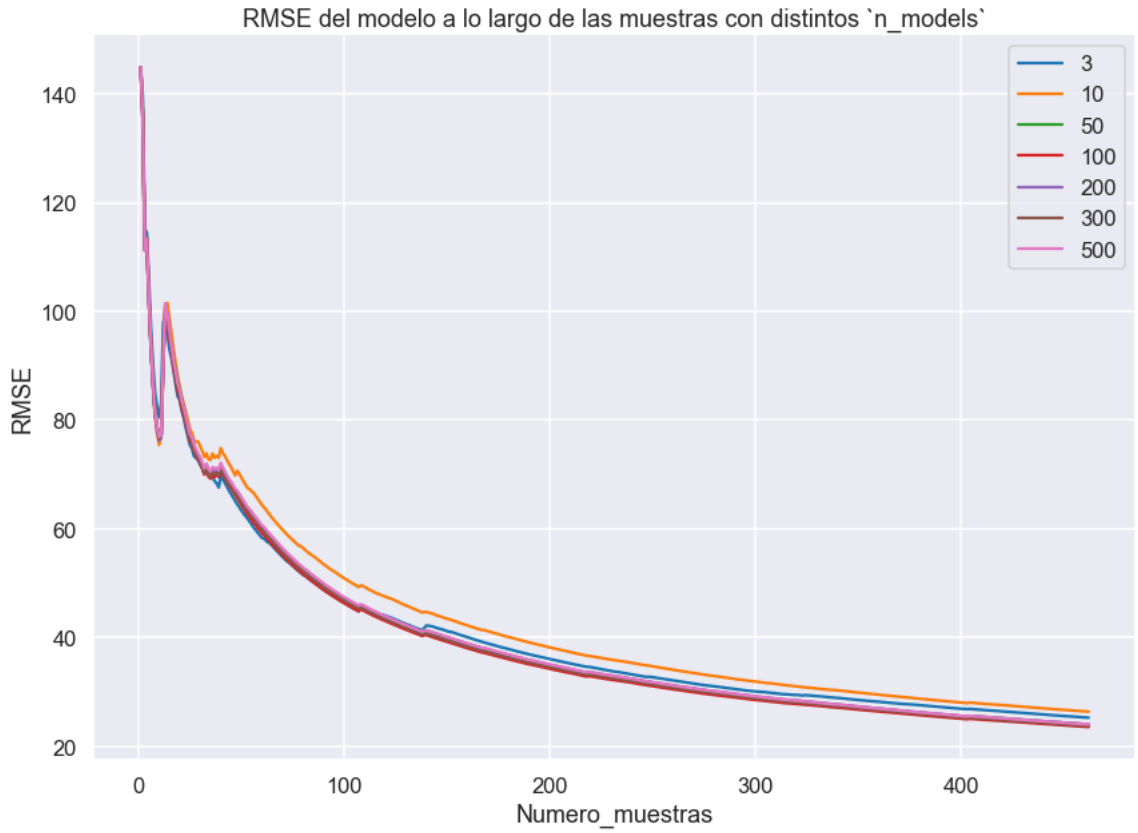

Figura 5.18: Gráfica que muestra el RMSE del modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.18 podemos ver que todos los modelos presentan un comportamiento similar en cuanto al RMSE excepto con `n_models`=10 y 30 que se comportan algo peor.

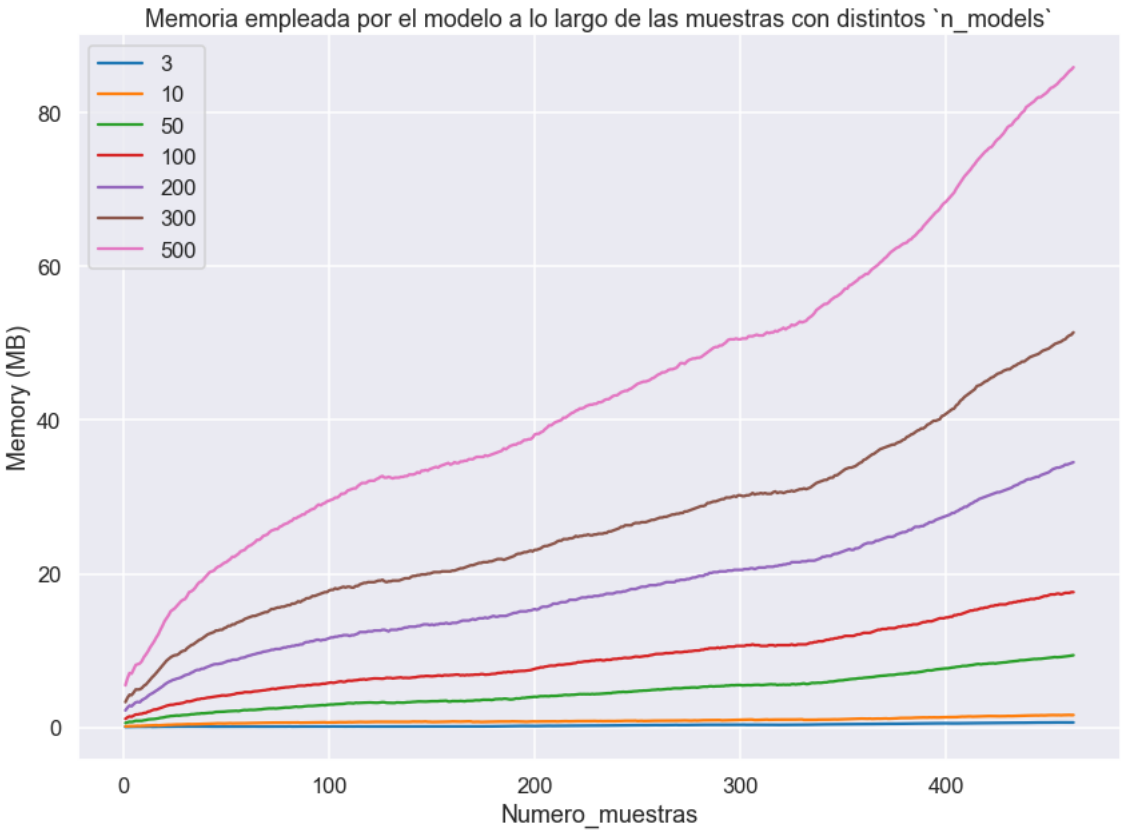

Figura 5.19: Gráfica que muestra la memoria (*memory*) en MB empleada por el modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.19 podemos ver la memoria empleada por los modelos a lo largo de las muestras. Observamos que, salvo con pocos árboles en el conjunto (3 o 10), en el resto de los modelos la memoria no para de aumentar, sobre todo el modelos con muchos árboles, y no se estabiliza en ningún momento.

Una vez realizados estos experimentos, vemos que ARF para regresión tampoco ofrece un gran rendimiento, puesto que la memoria empleada por el modelo no para de aumentar, por lo que ARF parece no aprender de la mejor forma.

5.2.5. Experimento 5

En el experimento anterior hemos realizado pruebas con ARF para regresión, si bien los conjuntos de datos Gradiant 1 y Gradiant 2 contienen tan solo 541 y 462 observaciones, respectivamente. Por lo tanto, quizás necesitemos emplear un conjunto de datos más grande para testear mejor a ARF. De esta manera, en este quinto experimento vamos a utilizar el conjunto de datos `Bikes`, que contiene 182470 observaciones, y vamos a realizar los experimentos análogamente al caso anterior con este conjunto de datos mucho más grande.

Utilizamos de nuevo la función `AdaptiveRandomForestRegressor` del módulo `ensemble` (ver capítulo 4, sección 4.2) con los siguientes parámetros: `max_features="sqrt"`, `lambda_value=6`, `grace_period=50` y `seed=42`. Además, para un mejor funcionamiento de ARF, emplearemos la función `MinMaxScaler` para el preprocesado de los datos, de manera que las observaciones se escalan en un rango de $[0, 1]$ como paso previo al entrenamiento del modelo.

Analizaremos el parámetro de número de árboles del conjunto (`n_models`), comparando el comportamiento del modelo empleando 4 valores distintos del parámetro: 3, 10, 50 y 100. De esta forma, implementamos un código para el cual ARF recibe las muestras u observaciones (x, y) una a una (tras ser escaladas) y va actualizando cada 100 muestras el RMSE y la memoria del modelo (al haber tantas muestras es suficiente recoger las métricas cada 100 y así se agiliza un poco la ejecución). Representaremos esta información mediante gráficas, que se muestran a continuación.

Nótese que para el modelo con `n_models`=100 se obtiene en la ejecución un error de memoria insuficiente a partir de la observación 119200, por lo que en las gráficas el modelo con `n_models`=100 sólo se representa hasta esta observación.

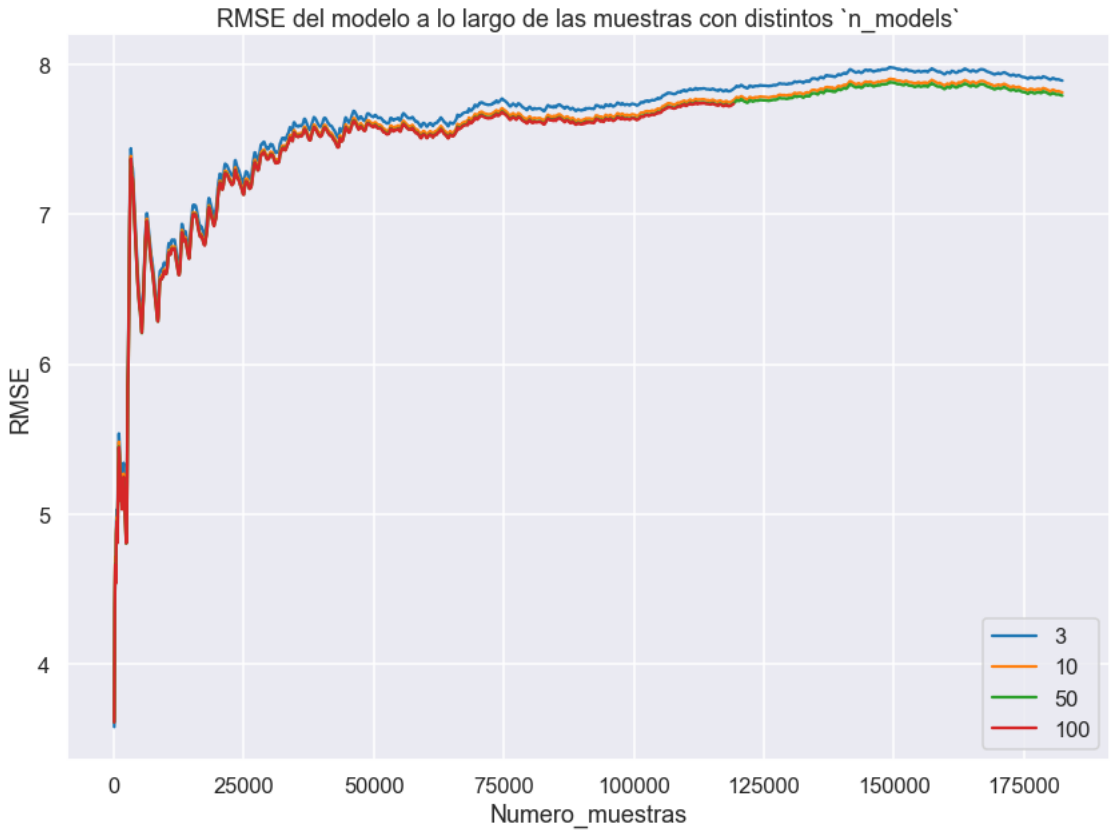

Figura 5.20: Gráfica que muestra el RMSE del modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.20 podemos ver que todos los modelos presentan un comportamiento similar (el modelo con `n_models`=3 algo peor) en cuanto al RMSE. El error no tiende nunca a reducirse a lo largo de las muestras, algo que nos indica un rendimiento deficiente.

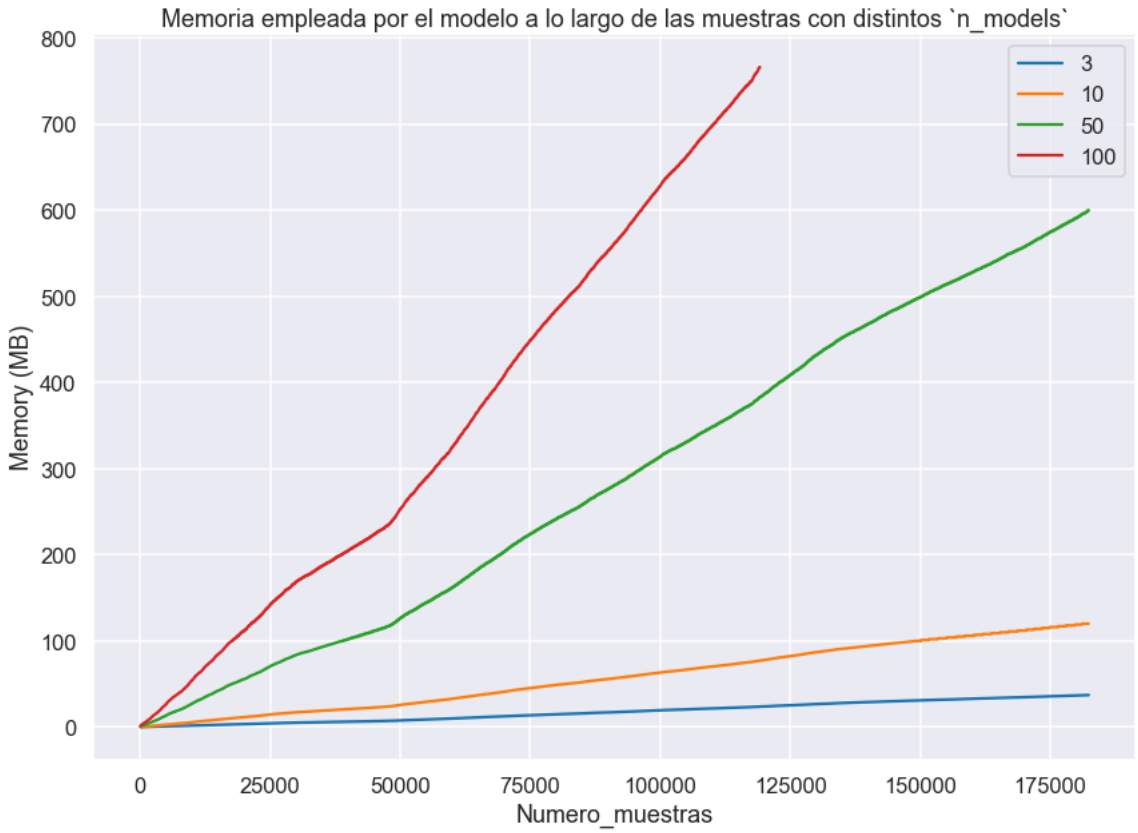

Figura 5.21: Gráfica que muestra la memoria (*memory*) en MB empleada por el modelo a lo largo de las muestras empleando distintos valores para el número de árboles del conjunto (`n_models`).

En la Figura 5.21 podemos ver la memoria empleada por los modelos a lo largo de las muestras. Observamos que, salvo con pocos árboles en el conjunto (`n_models`=3), en el resto de los modelos la memoria empleada por el modelo se dispara a lo largo de las muestras, algo que se agudiza en el modelo con `n_models`=100 dando un error de memoria insuficiente en la ejecución a partir de la observación 119200.

Tras estos experimentos, es evidente el pobre rendimiento de ARF para regresión, que no es capaz de aprender de forma adecuada a lo largo de las muestras mientras la memoria empleada crece indiscriminadamente.

5.2.6. Experimento 6

En esta último experimento continuaremos con el conjunto datos `Bikes`. Trataremos de confirmar el aprendizaje deficiente de ARF comprobando el número medio de nodos hoja del modelo a lo largo de las muestras.

Así, vamos a utilizar de nuevo la función `AdaptiveRandomForestRegressor` del módulo `ensemble` (ver capítulo 4, sección 4.2) con los siguientes parámetros: `n_models=50`, `max_features="sqrt"`, `lambda_value=6`, `grace_period=50` y `seed=42`. Además, para un mejor funcionamiento de ARF, emplearemos la función `MinMaxScaler` para el preprocesado de los datos, de manera que las observaciones se escalan en un rango de $[0, 1]$ como paso previo al entrenamiento del modelo.

De esta forma, implementamos un código para el cual ARF recibe las muestras u observaciones (x, y) una a una (tras ser escaladas) y va calculando el número medio de nodos hoja del modelo (número total de hojas dividido por el número de árboles del conjunto) a lo largo de las muestras. Representaremos esta información mediante una gráfica, que se muestra a continuación.

En la Figura 5.22 podemos ver el número medio de hojas del modelo a lo largo de las muestras. Como se observa, este número no para de aumentar notablemente, llegando finalmente a superar una media de 2800 hojas. Esto deja claro que el modelo no aprende correctamente, sobreajustándose debido a la necesidad de crear nuevos nodos hoja continuamente para intentar reducir el error de predicción.

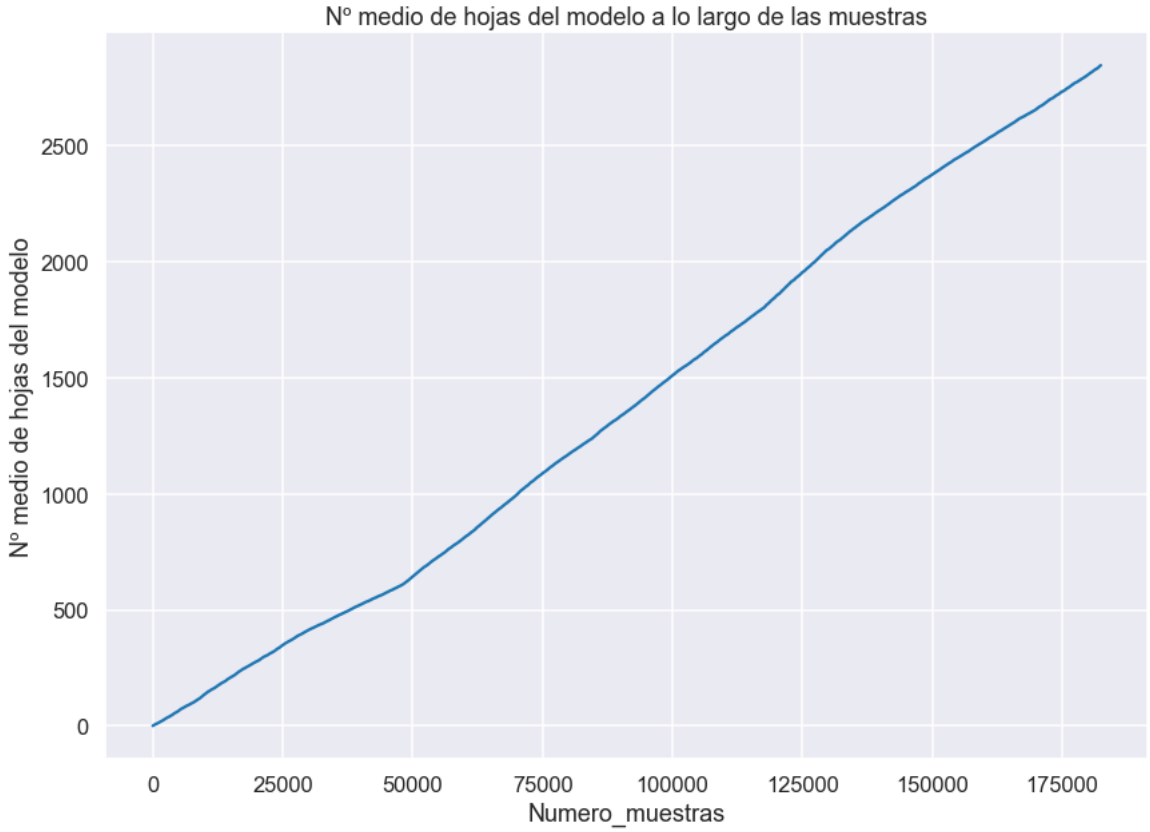

Figura 5.22: Gráfica que muestra el número medio de hojas del modelo a lo largo de las muestras.

Capítulo 6

Comentarios finales

Tras la revisión detallada del algoritmo ARF para la resolución de tareas de regresión/ clasificación supervisadas, la realización de experimentos para la evaluación del algoritmo con distintos conjuntos de datos, así como el análisis de los resultados obtenidos, podemos realizar un par de comentarios.

Como aspecto más relevante, cabe comentar que el algoritmo ARF muestra carencias de aprendizaje con los conjunto de datos empleados en los experimentos. Como solución para intentar combatir el error de predicción, el modelo tiene la necesidad de crear nuevos nodos hoja en sus árboles de forma reiterada, algo que tiende al sobreajuste y que no soluciona el problema de aprendizaje. Así, el modelo no para de crecer debido a estos nuevos nodos hoja y aumentan tanto la memoria como el tiempo empleados por el mismo. Se ha observado este comportamiento durante la resolución de problemas de regresión y clasificación y, aunque crítico de cara a la implantación de este tipo de algoritmos en soluciones comerciales en el ámbito de la industria 4.0, no se cita en Gomes et al. (2017,2018).

Otro aspecto a comentar es que en la realización de los experimentos para tareas de regresión nos hemos encontrado con muchos problemas a la hora de trabajar con ARF. Nos hemos visto en la necesidad de emplear distintas funciones para el preprocesado de los datos, ya que sin este preescalado de los datos ARF no llega ni a dar resultados mínimamente aceptables. Esto es algo que no se cita en Gomes et al. (2018) donde se presenta el algoritmo ARF para tareas de regresión.

Por todo ello, no recomendamos el uso del algoritmo ARF en aplicaciones comerciales.

Como líneas futuras, proponemos realizar un estudio empleando el algoritmo eSNN

(*evolving spiking neural networks*) (citado en la sección 2.2.6), que es uno de los métodos de *online learning* que ha surgido recientemente en la literatura científica de esta temática.

Apéndice A

Código de los experimentos

A continuación, a modo de ejemplo, se muestra el código empleado para la realización de los experimentos 1 y 6. El resto de experimentos fueron realizadas utilizando códigos similares.

A.1. Experimento 1

```
# coding: utf-8
#############################################################################
# Steel plates faults dataset (1941 instances) includes 27 attributes.
# Objetive is to predict the type of Fault (7 types).
from river import stream
from river import ensemble
from river import metrics
from contextlib import redirect_stdout
params = {
      'converters': {
                  'X_Minimum': float,
                  'X_Maximum': float,
                  'Y_Minimum': float,
                  'Y_Maximum': float,
                  'Pixels_Areas': float,
                  'X_Perimeter': float,
                  'Y_Perimeter': float,
                  'Sum_of_Luminosity': float,
                  'Minimum_of_Luminosity': float,
                  'Maximum_of_Luminosity': float,
                  'Length_of_Conveyer': float,
                  'TypeOfSteel_A300': bool,
```

```
                'TypeOfSteel_A400': bool,
                'Steel_Plate_Thickness': float,
                'Edges_Index': float,
                'Empty_Index': float,
                'Square_Index': float,
                'Outside_X_Index': float,
                'Edges_X_Index': float,
                'Edges_Y_Index': float,
                'Outside_Global_Index': float,
                'LogOfAreas': float,
                'Log_X_Index': float,
                'Log_Y_Index': float,
                'Orientation_Index': float,
                'Luminosity_Index': float,
                'SigmoidOfAreas': float
            }
    }

dataset = stream.iter_csv('steel_plates_faults.csv', target='Fault',**
    params)

model = ensemble.AdaptiveRandomForestClassifier(n_models=50,max_features="
    sqrt",disable_weighted_vote=False,lambda_value=6,
                                                    grace_period=20,seed=42)

cm=metrics.ConfusionMatrix()
metric = metrics.Accuracy() #cm=cm


##############################
# Escribe en un archivo el progreso del modelo al ir leyendo los datos
with open('progress.log', 'w') as f:
    with redirect_stdout(f):
        for i,(x,y) in enumerate(dataset, start=1):
            print('############')
            print('-Instance',i,',True label:',y)
            if i>10:
                print('Prediction:',model.predict_proba_one(x))
                cm=cm.update(y,model.predict_one(x))
                print(cm)
                metric=metric.update(y,model.predict_one(x))
                print(metric)
            model.learn_one(x, y)
```

A.2. Experimento 6

```
# -*- coding: utf-8 -*-

# Para que se vean algo mÃ¡s grandes los grÃ¡ficos
# en el notebook
import seaborn as sns
from matplotlib import rcParams
rcParams['figure.figsize'] = 14, 10
sns.set_context('talk')




# # Dataset Bikes de river (`toulouse_bikes`)
# 5 bike stations from the city of Toulouse. The dataset contains 182,470 observations.

# ## Pruebas con nÂº medio de hojas del modelo a lo largo de la muestra

# In[2]:


from river import ensemble
#from river import evaluate
from river import metrics
from river import preprocessing
from river import datasets
from river import compose
import time
dataset = datasets.Bikes()
model = compose.Select('clouds', 'humidity', 'pressure', 'temperature', 'wind')
model |= preprocessing.MinMaxScaler()
model |= ensemble.AdaptiveRandomForestRegressor(n_models=50,max_features="sqrt",aggregation_method="median",
                                                lambda_value=6,grace_period=50,seed=42,metric = metrics.RMSE())
metric = metrics.RMSE()
from contextlib import redirect_stdout
with open('progress_Bikes_n_leaves.log', 'w') as f:
    with redirect_stdout(f):
        tic = time.perf_counter()
        ntrees=50
        for i,(x,y) in enumerate(dataset, start=1):
```

```
            hojas = 0
            media = 0
            print('############')
            print('-Instance',i,',True label:',y)
            print('Prediction:',model.predict_one(x))
            metric=metric.update(y,model.predict_one(x))
            print(metric)
            if i>1:
                for j in range(ntrees):
                    hojas += model.steps['AdaptiveRandomForestRegressor'].
    models[j].model.n_leaves
                media = hojas/ ntrees
                print('NÂº medio de hojas del modelo', media)
            model.learn_one(x, y)
        toc = time.perf_counter()
        print(f"Run time:{toc - tic:0.4f} seconds")


# Vamos a crear un data frame con los resultados. Guardamos `
    Numero_muestras`, `RMSE` y `Media_hojas`.

# In[3]:


# save Numero_muestras
### for selecting elements in a .txt
import regex as re
m=[]
with open('progress_Bikes_n_leaves.log') as f:
    for line in f.read().splitlines():
          #print(line)
          m.append(re.search('(?<=Instance).[-+]?[.]?[\d]+(?:,\d\d\d)
    *[\.]?\d*(?:[eE][-+]?\d+)?', line))

#m.group(0)
#print(float(m[1].group(0)))
from operator import is_not
from functools import partial
m=list(filter(partial(is_not, None), m)) # quitamos los None
#print(m)
#len(m)
Numero_muestras=[]
for i in range(len(m)):
    Numero_muestras.append(int(m[i].group(0).replace('[','').replace(']','')
    ).replace(',','')))  # con replace quitamos los '[' ']' y ','
```

```
print(len(Numero_muestras))


# In[4]:


# save RMSE
### for selecting elements in a .txt
import regex as re
m=[]
with open('progress_Bikes_n_leaves.log') as f:
    for line in f.read().splitlines():
          #print(line)
          m.append(re.search('(?<=RMSE:).[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d
    *(?:[eE][-+]?\d+)?', line))

# m.group(0)
#print(float(m[1].group(0)))
from operator import is_not
from functools import partial
m=list(filter(partial(is_not, None), m)) # quitamos los None
#print(m)
#len(m)
RMSE=[]
for i in range(len(m)):
    RMSE.append(float(m[i].group(0)))
#print(RMSE)
print(len(RMSE))


# In[5]:


# save Media_hojas
### for selecting elements in a .txt
import regex as re
m=[]
with open('progress_Bikes_n_leaves.log') as f:
    for line in f.read().splitlines():
          #print(line)
          m.append(re.search('(?<=NÂº medio de hojas del modelo)
    .[-+]?[.]?[\d]+(?:,\d\d\d)*[\.]?\d*(?:[eE][-+]?\d+)?', line))

# m.group(0)
#print(float(m[1].group(0)))
```

```
from operator import is_not
from functools import partial
m=list(filter(partial(is_not, None), m)) # quitamos los None
#print(m)
#len(m)
Media_hojas=[]
for i in range(len(m)):
    Media_hojas.append(float(m[i].group(0)))
# vamos a introducir NaN en la muestra 1 para que tenga el mismo len
import numpy as np
Media_hojas.insert(0,np.nan)
#print(Media_hojas)
print(len(Media_hojas))


# In[6]:


# Save all as data frame
import pandas as pd
df = pd.DataFrame(
    {
        "Numero_muestras": Numero_muestras,
        "RMSE": RMSE,
        "Media_hojas": Media_hojas
    }
)
#df.head()
#df.tail()
df


# Vamos a representar el `RMSE`:

# In[7]:


import matplotlib.pyplot as plt
import seaborn as sns
sns.set_style('darkgrid')
out=sns.lineplot(data=df,x='Numero_muestras',y='RMSE')#,dashes=False)
out.set_title('RMSE del modelo a lo largo de las muestras')
out.set_ylabel('RMSE')
plt.show()

```

```
169
170 # Vamos a representar el nÂº medio de hojas del modelo a lo largo de las
        muestras ‘Media_hojas‘:
171
172 # In[8]:
173
174
175 import matplotlib.pyplot as plt
176 import seaborn as sns
177 sns.set_style('darkgrid')
178 out=sns.lineplot(data=df,x='Numero_muestras',y='Media_hojas')#,dashes=False
        )
179 out.set_title('NÂº medio de hojas del modelo a lo largo de las muestras')
180 out.set_ylabel('NÂº medio de hojas del modelo')
181 plt.show()
```

Bibliografía

[1] Abdulsalam H, Skillicorn DB, Martin P (2007) Streaming random forests. En: 11th international on database engineering and applications symposium, IDEAS (pp. 225-232). IEEE.

[2] Abdulsalam H, Skillicorn DB, Martin P (2008) Classifying evolving data streams using dynamic streaming random forests. En: Database and expert systems applications (pp. 643-651). Springer.

[3] Arora S, Hazan E, Kale S (2012) The multiplicative weights update method: a meta-algorithm and applications. Theory of Computing, 8(1), 121-164.

[4] Beygelzimer A, Kale S, Luo H (2015) Optimal and adaptive algorithms for online boosting. In International conference in machine learning (pp. 2323-2331).

[5] Bifet A, Gavaldà R (2007) Learning from Time-Changing Data with Adaptive Windowing. En: Proceedings of the 7th SIAM International Conference on Data Mining.

[6] Bifet A, Holmes G, Pfahringer B (2010a) Leveraging bagging for evolving data streams. En: PKDD (pp. 135-150).

[7] Bifet A, Holmes G, Pfahringer B, Kranen P, Kremer H, Jansen T, Seidl T (2010b). Moa: Massive online analysis, a framework for stream classification and clustering. En: Proceedings of the First Workshop on Applications of Pattern Analysis (pp. 44-50). PMLR.

[8] Breiman L (1996) Bagging predictors. Mach Learn 24, 123-140. https://doi.org/10.1007/BF00058655

[9] Breiman L (2001) Statistical modeling: The two cultures (with comments and a rejoinder by the author). Statistical science, 16(3), 199-231.

[10] Cauwenberghs G, Poggio T (2001) Incremental and decremental support vector machine learning. Advances in neural information processing systems, 409-415.

[11] Cavallanti G, Cesa-Bianchi N, Gentile C (2007). Tracking the best hyperplane with a simple budget perceptron. Machine Learning, 69(2), 143-167.

[12] Cesa-Bianchi N, Conconi A, Gentile C (2005) A second-order perceptron algorithm. SIAM Journal on Computing, 34(3), 640-668.

[13] Cesa-Bianchi N, Lugosi G (2006) Prediction, learning, and games. Cambridge university press.

[14] Chaudhuri K, Freund Y, Hsu DJ (2009) A parameter-free hedging algorithm. In Advances in neural information processing systems (pp. 297-305).

[15] Chen ST, Lin HT, Lu CJ (2012) An online boosting algorithm with theoretical justifications. En: Proceedings of the international conference on machine learning (ICML).

[16] Chernov A, Vovk V (2010) Prediction with advice of unknown number of experts.

[17] Crammer K, Dekel O, Keshet J, Shalev-Shwartz S, Singer Y (2006) Online passive-aggressive algorithms. The Journal of Machine Learning Research, 7:551-585.

[18] Crammer K, Kulesza A, Dredze M (2013) Adaptive regularization of weight vectors. Machine learning, 91(2), 155-187.

[19] Crammer K, Lee DD (2010) Learning via gaussian herding. En: Advances in neural information processing systems (pp. 451-459).

[20] Domingos P, Hulten G (2000) Mining high-speed data streams. En: Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 71-80). ACM SIGKDD.

[21] Dredze M, Crammer K, Pereira F (2008) Confidence-weighted linear classification. En: Proceedings of the 25th international conference on Machine learning (pp. 264-271).

[22] Duchi J, Singer Y (2009) Efficient online and batch learning using forward backward splitting. The Journal of Machine Learning Research, 10, 2899-2934.

[23] Duchi J, Hazan E, Singer Y (2011). Adaptive subgradient methods for online learning and stochastic optimization. Journal of machine learning research, 12(7).

[24] Frank A, Asuncion A (2010) UCI Machine Learning Repository. University of California, School of Information and Computer Science, Irvine. http://archive.ics.uci.edu/ml

[25] Freund Y, Schapire RE (1996) Schapire R: Experiments with a new boosting algorithm. En: Thirteenth International Conference on ML.

[26] Freund Y, Schapire RE (1997) A decision-theoretic generalization of on-line learning and an application to boosting. Journal of computer and system sciences, 55(1), 119-139.

[27] Freund Y, Schapire RE (1999) Large margin classification using the perceptron algorithm. Machine learning, 37(3), 277-296.

[28] Gentile C (2001) A new approximate maximal margin classification algorithm. Journal of Machine Learning Research, 2(Dec), 213-242.

[29] Gomes HM, Enembreck F (2014) Sae2: advances on the social adaptive ensemble classifier for data streams. En: Proceedings of the 29th annual ACM symposium on applied computing (SAC), SAC 2014 (pp. 199-206). ACM.

[30] Gomes HM, Bifet A, Read J, Barddal JP, Enembreck F, Pfharinger B, Holmes G, Abdessalem T (2017) Adaptive random forests for evolving data stream classification. Machine Learning 106(9-10), pp. 1469-1495 . https://doi.org/10.1007/s10994-017-5642-8

[31] Gomes HM, Barddal JP, Boiko LE, Bifet A (2018) Adaptive random forests for data stream regression. ESANN 2018.

[32] Halford M, Bolmier G, Sourty R, Vaysse R, Zouitine A (2019) creme, a Python library for online machine learning. https://github.com/MaxHalford/creme

[33] Hido S (2012) Jubatus: Distributed online machine learning framework for big data. Proc. of the 1st Extremely Large Databases (XLDB) Asia. http://jubat.us/

[34] Hoi SC, Jin R, Zhao P, Yang T (2013) Online multiple kernel classification. Machine Learning, 90(2), 289-316.

[35] Hoi SC, Wang J, Zhao P (2014) Libol: A library for online learning algorithms. Journal of Machine Learning Research, 15(1), 495.

[36] Hoi SC, Sahoo D, Lu J, Zhao P (2021) Online learning: A comprehensive survey. Neurocomputing, 459, 249-289.

[37] Ikonomovska E, Gama J, Dẑeroski S (2015) Online tree-based ensembles and option trees for regression on evolving data streams. Neurocomputing, 150, 458-470.

[38] Jin R, Hoi SC, Yang T (2010) Online multiple kernel learning: Algorithms and mistake bounds. En: International conference on algorithmic learning theory (pp. 390-404). Springer, Berlin, Heidelberg.

[39] Kivinen J, Smola AJ, Williamson RC (2004) Online learning with kernels. IEEE transactions on signal processing, 52(8), 2165-2176.

[40] Kolter JZ, Maloof M (2003) Dynamic weighted majority: A new ensemble method for tracking concept drift. En: Third IEEE international conference on data mining, ICDM 2003 (pp. 123-130). IEEE.

[41] Langford J, Li L, Zhang T (2009) Sparse Online Learning via Truncated Gradient. Journal of Machine Learning Research, 10(3).

[42] Li Y, Long PM (2002) The relaxed online maximum margin algorithm. Machine Learning, 46(1), 361-387.

[43] Lobo JL, Laña I, Del Ser J, Bilbao MN, Kasabov N (2018) Evolving spiking neural networks for online learning over drifting data streams. Neural Networks, 108, 1-19.

[44] Lu J, Hoi SC, Wang J, Zhao P, Liu ZY (2016) Large scale online kernel learning. Journal of Machine Learning Research, 17(47), 1.

[45] Montiel J, Read J, Bifet A, Abdessalem T (2018) Scikit-Multiflow: A Multi-output Streaming Framework. En: Journal of Machine Learning Research 19.72 , pp. 1-5. https://github.com/scikit-multiflow/scikit-multiflow

[46] Montiel J, Halford M, Mastelini SM, Bolmier G, Sourty R, Vaysse R, Zouitine A, Gomes HM, Read J, Abdessalem T, Bifet A (2021) River: machine learning for streaming data in Python. https://github.com/online-ml/river

[47] Nick Littlestone (1988) Learning quickly when irrelevant attributes abound: A new linearthreshold algorithm. Machine learning, 2(4):285-318.

[48] Littlestone N, Warmuth MK (1994) The weighted majority algorithm. Information and computation, 108(2), 212-261.

[49] Orabona F, Keshet J, Caputo B (2009) Bounded Kernel-Based Online Learning. Journal of Machine Learning Research, 10(11).

[50] Orabona F, Crammer K (2010) New adaptive algorithms for online classification. Advances in neural information processing systems, 23, 1840-1848.

[51] Oza NC (2005) Online bagging and boosting. IEEE International Conference on Systems, Man and Cybernetics, 3, 2340-2345.

[52] Page ES (1954) Continuous inspection schemes. Biometrika, 41(1/2), 100-115.

[53] Pedregosa F, Varoquaux G, Gramfort A, Michel V, Thirion B, Grisel O, Blondel M, Prettenhofer P, Weiss R, Dubourg V, Vanderplas J, Passos A, Cournapeau D, Brucher M, Perrot M, Duchesnay É (2011) Scikit-learn: Machine learning in Python. the Journal of machine Learning research, 12, 2825-2830.

[54] Pelossof R, Jones M, Vovsha I, Rudin C (2009) Online coordinate boosting. En: IEEE 12th international conference on computer vision workshops (ICCV Workshops) (pp. 1354-1361). IEEE.

[55] Pesaranghader A (2018) A Reservoir of Adaptive Algorithms for Online Learning from Evolving Data Streams (Doctoral dissertation, Université d'Ottawa/University of Ottawa).

[56] Rosenblatt F (1958) The perceptron: a probabilistic model for information storage and organization in the brain. Psychological review, 65(6):386, 1958.

[57] Roughgarden T, Schrijvers O (2017) Online prediction with selfish experts.

[58] Shalev-Shwartz S, Singer Y, Srebro N, Cotter A (2011) Pegasos: Primal estimated sub-gradient solver for svm. Mathematical programming, 127(1),3-30.

[59] Wang Z, Vucetic S (2010) Online passive-aggressive algorithms on a budget. En: Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics (pp. 908-915). JMLR Workshop and Conference Proceedings.

[60] Wang J, Zhao P, Hoi SC (2012a) Exact soft confidence-weighted learning.

[61] Wang Z, Crammer K, Vucetic S (2012b) Breaking the curse of kernelization: Budgeted stochastic gradient descent for large-scale svm training. The Journal of Machine Learning Research, 13(1), 3103-3131.

[62] Wang J, Zhao P, Hoi SC, Zhuang J, Liu ZY (2013) Large scale online kernel classification.

[63] Xiao L (2009) Dual averaging method for regularized stochastic learning and online optimization. Advances in Neural Information Processing Systems, 22, 2116-2124.

[64] Yang L, Jin R, Ye J (2009) Online learning by ellipsoid method. En: Proceedings of the 26th Annual International Conference on Machine Learning (pp. 1153-1160).

[65] Yang T, Mahdavi M, Jin R, Yi J, Hoi S (2012) Online kernel selection: Algorithms and evaluations. En: Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 26, No. 1).

[66] Zhao P, Wang J, Wu P, Jin R, Hoi SC (2012). Fast bounded online gradient descent algorithms for scalable kernel-based online learning.

[67] Zinkevich M (2003) Online convex programming and generalized infinitesimal gradient ascent. En: Proceedings of the 20th international conference on machine learning (ICML) (pp. 928-936).