Universidade Vigo

Trabajo Fin de Máster

Incremental Learning para aplicaciones de edge computing distribuido en redes IoT

Ángel Pablo González López

Máster en Técnicas Estadísticas

Curso 2022-2023

Propuesta de Trabajo Fin de Máster

Título en galego: Incremental Learning para aplicacións de edge computing distribuído en redes IoT

Título en español: Incremental Learning para aplicaciones de edge computing distribuido en redes IoT

English title: Incremental Learning for edge computing applications distributed in IoT networks

Modalidad: Modalidad B

Autor/a: Ángel Pablo González López, Universidad de Santiago de Compostela

Director/a: Marta Sestelo Pérez, Universidad de Vigo

Tutor/a: Bruno Fernández Castro, Gradiant

Breve resumen del trabajo:

Investigación del estado del arte del incremental learning y desarrollo en Python de un algoritmo incremental de regresión multivariante a partir de otro de clasificación, de acuerdo a los estándares de la librería River.

Recomendaciones:

Conocimiento y experiencia trabajando con Python.
Valorables pero no imprescindibles:

- Interés o conocimientos en técnicas de ML

Doña Marta Sestelo Pérez, Profesora Ayudante Doctora de la Universidad de Vigo y don Bruno Fernández Castro, Responsable Técnico de Machine Learning & Optimización de Gradiant informan que el Trabajo Fin de Máster titulado

Incremental Learning para aplicaciones de edge computing distribuido en redes IoT

fue realizado bajo su dirección por don Ángel Pablo González López 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 2023.

SESTELO PEREZ MARTA - 76998300G
Firmado digitalmente por SESTELO PEREZ MARTA - 76998300G
Fecha: 2023.01.26 16:55:52 +01'00'

Firmado por FERNANDEZ CASTRO, BRUNO (FIRMA) el día 27/01/2023 con un certificado emitido por AC DNIE 006

La directora:
Doña Marta Sestelo Pérez

El tutor:
Don Bruno Fernández Castro

Ángel GL

El autor:
Don Ángel Pablo González López

Declaración responsable. Para dar cumplimiento a la Ley 3/2022, de 24 de febrero, de convivencia universitaria, referente al plagio en el Trabajo Fin de Máster (Artículo 11, Disposición 2978 del BOE núm. 48 de 2022), **el/la autor/a declara** que el Trabajo Fin de Máster presentado es un documento original en el que se han tenido en cuenta las siguientes consideraciones relativas al uso de material de apoyo desarrollado por otros/as autores/as:

- Todas las fuentes usadas para la elaboración de este trabajo han sido citadas convenientemente (libros, artículos, apuntes de profesorado, páginas web, programas,...)
- Cualquier contenido copiado o traducido textualmente se ha puesto entre comillas, citando su procedencia.
- Se ha hecho constar explícitamente cuando un capítulo, sección, demostración... sea una adaptación casi literal de alguna fuente existente.

Y, acepta que, si se demostrara lo contrario, se le apliquen las medidas disciplinarias que correspondan.

Índice general

Resumen IX

1. Introducción 1
1.1. Conceptos previos . . . 1
1.1.1. IoT . . . 1
1.1.2. Cloud y edge computing . . . 1
1.2. Gradiant . . . 2
1.3. Estructura del trabajo . . . 3

2. Estado del arte 5
2.1. Contexto general y notación . . . 5
2.1.1. Prototipos . . . 5
2.1.2. Notación . . . 5
2.2. Algoritmos de aprendizaje incremental . . . 6
2.3. Algoritmos basados en prototipos . . . 7
2.3.1. Algoritmo LVQ . . . 7
2.3.2. Algoritmo ILVQ . . . 8
2.3.3. Algoritmo XuILVQ . . . 11
2.3.4. Otros algoritmos incrementales de regresión . . . 11

3. Adaptación del algoritmo XuILVQ a regresión multivariante 15
3.1. Librerías empleadas . . . 15
3.1.1. Librería River . . . 16
3.2. Adaptación a regresión . . . 16
3.2.1. Convertir regresión en clasificación . . . 17
3.2.2. Modificación del método de actualización de prototipos . . . 19
3.3. Algoritmo XuRILVQ . . . 19
3.3.1. Primera y segunda versión del algoritmo XuRILVQ . . . 19
3.3.2. Tercera versión del algoritmo XuRILVQ . . . 19

4. Resultados 23
4.1. Herramientas de validación . . . 23
4.1.1. Medida de error . . . 23
4.1.2. Utilidades de River . . . 24
4.1.3. *Datasets* empleados . . . 24
4.2. Primera comparación entre versiones . . . 25
4.3. Comparación con *datasets* de River . . . 26
4.4. Comportamiento con respecto a la dimensión . . . 29

5. Conclusiones **33**
5.1. Posibles cambios a realizar en el algoritmo 33
5.1.1. Cambios relacionados con el *clustering* 33
5.1.2. Posibles añadidos 34
5.2. Líneas de trabajo futuras 34

Bibliografía **35**

Resumen

Resumen en español

Los modelos de regresión/clasificación basados en algoritmos de *incremental learning* están diseñados para reaccionar de forma dinámica y en tiempo real a flujos de datos continuos que reciben como entrada, re-entrenándose automática e iterativamente, adaptándose a los posibles cambios en sus distribuciones de probabilidad.

Este tipo de algoritmos tienen aplicaciones en diferentes ámbitos, por ejemplo, en sistemas distribuidos de *edge computing* sobre dispositivos IoT embebidos, en los que el *hardware* no es tan potente como para almacenar y realizar entrenamientos programados con grandes volúmenes de datos.

Este trabajo investiga la aplicación de este tipo de algoritmos, aplicados para la resolución de problemas de regresión multivariante, en el ámbito de la Industria 4.0 y las redes de dispositivos IoT interconectados.

Para la realización del trabajo, el alumno se integró en un equipo y colaboró en el diseño e implementación del algoritmo de regresión, así como en el desarrollo de un conjunto de pruebas o tests para su evaluación.

English abstract

Regression and classification models based on incremental learning algorithms are specifically designed to react dynamically and in real time to continuous data streams received as input, retraning themselves automatically and iteratively, adapting to possible changes in their probability distributions.

Such algorithms have applications in different domains, e.g. in distributed edge computing systems on embedded IoT devices, where the hardware is not powerful enough to store and perform scheduled training with large volumes of data.

This thesis investigates the application of this type of algorithms, applied to solve multivariate regression problems, in the field of Industry 4.0 and networks of interconnected IoT devices.

To carry out the work, the student joined a team and collaborated in the design and implementation of the regression algorithm, as well as in the development of a set of tests for its evaluation.

Capítulo 1

Introducción

En este primer capítulo se procederá, en la Sección 1.1, a explicar los conceptos que dan nombre a este trabajo así como a explicar su interés como motivadores de esta investigación. A continuación, en la Sección 1.2, se explicará el desempeño de Gradiant como empresa así como sus objetivos de cara al desarrollo de este trabajo. Finalmente, en la Sección 1.3, se detallarán la estructura que sigue este trabajo.

1.1. Conceptos previos

Dentro del gran campo que son los algoritmos de *machine learning* (ML o en español, aprendizaje automático) los algoritmos de ***incremental learning*** (en ocasiones erróneamente llamado *online learning*, pues son conceptos distintos[1]) surgen como respuesta a situaciones en las que, por alguna causa, en lugar de disponer de todos los datos desde un inicio, los datos son recibidos en forma de flujo de datos. Esto obliga a estos algoritmos a adaptarse constantemente y de manera dinámica a la información recibida.

1.1.1. IoT

Esta clase de algoritmos resulta especialmente apropiada para dispositivos de IoT (***Internet of Things***). El IoT, o Internet de las cosas en español, pese a no tener una definición rigurosa, engloba a todos aquellos dispositivos con sensores, capacidad de procesamiento de información, *software* y capacidad de conexión e intercambio de información con otros dispositivos. Con la reciente tendencia a la domotización de hogares y fábricas, es cada vez mayor el número de dispositivos IoT que hay en circulación, desde enchufes inteligentes hasta "smart fridges" pasando por termostatos, sensores de humedad, detectores de movimiento, etc. Se estima que para 2025 en el mundo habrá más de 75 mil millones de estos dispositivos, lo que equivaldría a una media de más de 9 dispositivos por persona (Ketsu y Mishra 2021).

1.1.2. Cloud y edge computing

La capacidad de los algoritmos de *incremental learning* de actualizarse constantemente según nuevos datos son recibidos es una propiedad deseable para los dispositivos IoT por dos motivos fundamentales:

- Generalmente, los datos recibidos son extraídos de manera continua de redes masivas de sensores y dispositivos IoT.

[1] La diferencia radica en que *online learning* se asocia a muestras que se reciben como un flujo continuo de datos, mientras que en *incremental learning* los datos pueden ser recibidos en lotes (o *batches* de datos) no necesariamente en tiempo real. Así, todo algoritmo de *online learning* es también de *incremental learning*, siendo falso el recíproco.

- La cantidad de datos que se recibe es tal que el uso de arquitecturas de computación basadas exclusivamente en la nube (*cloud computing*) ralentizaría en gran medida el cálculo. Para paliar este problema, se opta por soluciones donde son los propios dispositivos los encargados de realizar parte de las operaciones en una arquitectura de computación híbrida nube-frontera donde los dispositivos IoT son los encargados de la parte de computación frontera (***edge computing***).

Por *cloud computing* nos referimos a una red de servidores en remoto a los que se accede a través de Internet con el fin de almacenar y procesar datos de todo tipo. Una de sus principales ventajas radica en que este tipo de computación evita la necesidad de disponer de un *hardware* en concreto o de instalar cualquier tipo de *software*. A cambio, sus principales desventajas son que esta centralización de las operaciones crea una dependencia del proveedor del servicio así como una necesidad obligatoria de conexión a Internet. En casos como el mencionado, donde se disponen de datos del orden de millones de TB, el proceso de subirlos todos a la nube puede actuar como cuello de botella, ralentizando sobremanera hasta los cálculos más simples.

El *edge computing* (a veces también llamado *mist computing*) surge con el fin de poner remedio a los mayores inconvenientes del *cloud computing*. Así, en esta arquitectura los propios dispositivos IoT serían los encargados de, además de estar conectados al servidor en la nube, realizar un preprocesamiento y filtrado inicial de los datos, almacenarlos de manera local y operar con ellos (siempre y cuando se requiera poca potencia de computación). El nombre de *edge computing* se deriva pues del hecho de desplazar gran parte de la carga de trabajo a la frontera de la red, estando así lo más cerca posible de usuarios, dispositivos y, más importante, las fuentes de los datos recibidos.

Para una revisión más detallada de las principales ventajas e inconvenientes de ambas arquitecturas de computación así como su adaptación al contexto IoT se puede consultar Ketsu y Mishra (2021).

1.2. Gradiant

Gradiant, centro de Tecnologías de la Información y las Comunicaciones (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 140 profesionales y 14 patentes solicitadas, Gradiant ha desarrollado más de 340 proyectos diferentes de I+D+i, convirtiéndose en uno de los principales motores de la innovación en Galicia.

El Centro fue creado en 2008 y 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 Abanca, Altia, Arteixo Telecom, Egatel, Indra, Plexus, R, Telefónica, 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).

Tras 14 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, IoT, la biometría y analítica de datos y los sistemas de comunicaciones avanzadas.

En cuanto a la línea de Industria del centro, donde se engloba este trabajo, su misión se encuentra totalmente alineada con la de Gradiant, pero aplicada al sector manufacturero. Por tanto, se puede definir como "contribuir al dinamismo innovador, el crecimiento y la mejora competitiva del tejido industrial gallego a través del desarrollo tecnológico y la innovación en el uso de las TIC".

Su filosofía es ayudar a las empresas a generar negocio a través del aumento de su competitividad mediante la transformación digital, generando mejoras en sus procesos y facilitando la creación de

nuevos modelos de negocio. Integran la transferencia de conocimiento y diseñan soluciones especializadas para la industria, sumando el valor añadido de los profesionales del centro. Además, siendo completamente independientes, por lo que aportan una visión neutral acerca de las mejores soluciones para cubrir los objetivos de las empresas.

La idea propuesta inicialmente por Gradiant consiste en que, partiendo de un algoritmo incremental de clasificación basado en prototipos, se realice una adaptación de este al contexto de la regresión multivariante (manteniendo el ser incremental y basado en prototipos). Posteriormente, el objetivo es realizar una implementación de este algoritmo en Python empleando la librería de código abierto River (Montiel et al. 2021) para, finalmente, comprobar la efectividad del algoritmo comparándolo con otros algoritmos de regresión incrementales del estado del arte esperando que, a cambio de un incremento controlado en el error del modelo de regresión, se observe un menor consumo de memoria que en los otros algoritmos. Eventualmente, la adaptación creada podría ser incorporada al catálogo de algoritmos incrementales de River.

Se ha seguido en todo momento la hoja de ruta de la propuesta, comenzando por una revisión del estado del arte del tema para, posteriormente, proponer varias adaptaciones distintas del algoritmo y realizar una comparación con otros regresores. Como se comprobará posteriormente, los objetivos marcados han sido cumplidos obteniendo en última instancia un algoritmo con resultados prometedores.

1.3. Estructura del trabajo

Habiendo introducido el marco de los conceptos y la empresa en que se sitúa la realización de este trabajo, se procede a detallar la estructura seguida en este trabajo.

En el Capítulo 2 se hace una revisión del estado del arte en lo referente a algoritmos incrementales de ML y algoritmos basados en el uso de prototipos, concepto que se introduce previamente junto con notación recurrente a lo largo del trabajo. A continuación, en el Capítulo 3 se introducen las herramientas y conceptos empleados a la hora de realizar la adaptación del algoritmo para, finalmente, concluir formulando el pseudocódigo de las adaptaciones propuestas. El Capítulo 4 engloba todo lo relacionado con las pruebas realizadas a las mencionadas adaptaciones, desde medidas de error y *datasets* empleados en la comparación hasta las propias comparaciones. Por último, en el Capítulo 5 se desarrollan posibles cambios que podrían probarse en el algoritmo así como futuras líneas de trabajo con las que continuar la investigación.

Capítulo 2

Estado del arte

Este capítulo consiste en una revisión sobre el estado actual del tema a tratar en este trabajo, es decir, algoritmos incrementales y más concretamente aquellos basados en prototipos. En primer lugar, en la Sección 2.1, se introduce el concepto de prototipo así como notación general que se empleará con regularidad en los siguientes capítulos. A continuación, en la Sección 2.2, se reseñan brevemente algunos tipos de algoritmos de aprendizaje incremental. Por último, en la Sección 2.3, se hará una breve revisión de algoritmos basados en prototipos, destacando sobre todo el algoritmo ILVQ (Xu et al. 2012) y su modificación, el algoritmo XuILVQ (González-Soto et al. 2022).

2.1. Contexto general y notación

2.1.1. Prototipos

Un concepto fundamental para el desarrollo de este trabajo es el de modelos de ML basados en **prototipos**. En estos modelos la información aportada por los datos es retenida a través de los prototipos, un conjunto representativo de los datos. Estos prototipos pueden ser tanto un subconjunto de los datos como elementos totalmente ajenos a los datos. Es entonces posible emplear estos prototipos, junto con una medida que cuantifique la similitud entre datos y prototipos, para desarrollar modelos de aprendizaje tanto supervisado como no supervisado. Estos modelos presentan ciertas propiedades que los hacen muy interesantes en la práctica, por ejemplo:

- En general, para obtener un conjunto representativo de los datos se necesita un número menor de prototipos, lo cual ayuda a reducir el empleo de memoria.
- Los prototipos ayudan a encontrar estructuras subyacentes en los datos, como puede ser el agrupamiento en *clusters*.
- Al trabajar con prototipos que representan a los datos en lugar de los propios datos, estos modelos son más seguros en términos de privacidad.

Son estas propiedades, en especial las dos primeras, las que hacen que el uso de modelo de aprendizaje basados en prototipos sea deseable en el contexto que nos encontramos, i. e., redes de dispositivos de IoT distribuidos en una arquitectura de *edge computing*. Interesa tanto ocupar la menor capacidad de memoria posible como encontrar posibles estructuras de los datos que ayuden a simplificarlos en el preprocesado, facilitando posteriormente el operar con ellos (Biehl et al. 2016).

2.1.2. Notación

Denotaremos por $\mathbf{x} = (x_1, \ldots, x_d) \in \mathbb{R}^d$ al vector cuyos componentes son las variables empleadas para la predicción, también llamadas **variables explicativas**. En principio, no hay imposiciones sobre

estas variables y pueden ser tanto categóricas como numéricas, si bien es cierto que, generalmente, los algoritmos están diseñados para tratar con un tipo concreto de variable. En nuestro caso, estaremos interesados en variables explicativas numéricas.

Denotaremos por $y \in \mathbb{R}$ a la etiqueta o respuesta, a la cual llamaremos **clase** cuando hablemos de clasificación y **variable respuesta** cuando hablemos de regresión. En caso de tratarse de un problema de clasificación, la pertenencia a $\mathbb{R}$ siempre la podremos asegurar sin más que establecer una biyección entre las distintas clases y tantos números naturales como clases haya.

El objetivo de estos problemas consiste en, partiendo de una muestra

$$\{(x_{1i}, \dots, x_{di}, y_i) \in \mathbb{R}^{d+1} : i = 1, \dots, n\},$$

obtener predicciones $\hat{y}(\mathbf{x})$ de la respuesta en función del valor de $\mathbf{x} = (x_1, \dots, x_d)$.

Como ya se ha mencionado previamente, en nuestro caso resulta interesante el hecho de trabajar con una muestra representativa de los datos en lugar de hacerlo con la totalidad de estos, haciendo uso de prototipos. Por lo tanto, denotaremos por $\mathbf{w}_i \in \mathbb{R}^d$ al vector de valores que toma el prototipo i en las distintas variables explicativas, con su correspondiente clase o valor de la variable respuesta asociada $y_i \in \mathbb{R}$. Al conjunto de todos los prototipos lo denotaremos por G y al cardinal de dicho conjunto lo denotaremos por N_G. Así, nuestro objetivo será el obtener una predicción $\hat{y}(\mathbf{x})$ de la clase o variable respuesta en función de la relación de las variables explicativas $\mathbf{x}$ con el conjunto de prototipos $G = \{\mathbf{w}_1, \mathbf{w}_2, \dots\}$.

2.2. Algoritmos de aprendizaje incremental

Como ya se ha mencionado en el anterior capítulo, los algoritmos de aprendizaje incremental resultan fundamentales en situaciones donde la información es recibida en forma de un flujo constante de datos, siendo necesario que se adapten de manera rápida y constante a los nuevos datos recibidos.

Un término relacionado con esta forma de adaptación constante es el conocido como *concept drift*, el cual se usa para describir el hecho de que la distribución de los datos varíe con el tiempo (equivalentemente, los datos no son i.i.d.). En el contexto de algoritmos de aprendizaje incremental esto supone un problema, pues el algoritmo se adapta rápidamente a los datos más recientes recibidos pero es precisamente esta velocidad de adaptación la que hace que también olvide información antigua importante con igual rapidez.

Para paliar este problema, junto con otras consideraciones que se deben tener en cuenta sobre el aprendizaje incremental (como puede ser la elección de parámetros adaptativos o el uso eficiente de la memoria), se han desarrollado multitud de algoritmos, entre los que podemos destacar:

- **SVM incrementales**: Existen diversos algoritmos incrementales basados en SVM (*support-vector machine*). Estos pueden, entre otros enfoques, basarse en una heurística como sería reentrenar el modelo cada ciertas iteraciones con los vectores de soporte junto con un *batch* de datos o incorporar modificaciones en la función de coste para facilitar el hacerla incremental. Se puede consultar más al respecto de esta clase de algoritmos en Xiao et al. (2000) o Domeniconi y Gunopulos (2001).

- **Modelos conexionistas**: Con este nombre se conoce a aquellos modelos complejos formados por redes de modelos simples interconectados, como son las redes neuronales artificiales. Dentro de este tipo de algoritmos destacan el *multi-layer perceptron* (MLP) y sus adaptaciones. Puede consultarse un ejemplo de adaptación propuesta en Polikar et al. (2000).

- **Métodos de ensamblaje**: Combinan distintos modelos usando alguna ponderación adecuada. En esta clase de algoritmos destacan los del tipo *Random Forest*, como es, por ejemplo, el propuesto en Wang et al. (2009).

- **Algoritmos basados en prototipos**: Emplean muestras representativas de la población y se verán más en detalle en la siguiente sección.

Se puede consultar también Gepperth y Hammer (2016) para obtener más información sobre estos algoritmos, al ser una muy buena revisión de los algoritmos incrementales.

2.3. Algoritmos basados en prototipos

Dentro de la familia de algoritmos basados en el uso de prototipos, podemos hacer distinción entre algoritmos de aprendizaje supervisado y algoritmos de aprendizaje no supervisado.

Entre los de aprendizaje no supervisado destacan algoritmos como *Competitive Vector Quantization* (VQ), *Neural Gas Algorithm* (NG) o los *Self-Organizing Maps* (SOM) sobre los que se puede encontrar más información en Biehl et al. (2016). De estos, es especialmente interesante el SOM, al haber sido parte de una de las adaptaciones realizadas, por lo que referimos al artículo original de Kohonen (1990) para una explicación más detallada.

Entre los algoritmos de aprendizaje supervisado basados en prototipos destacan principalmente dos tipos, ambos enfocados generalmente en problemas de clasificación. Por un lado, se encuentran las diversas adaptaciones de algoritmos de *k-Nearest Neighbors* (k-NN), las cuales emplean prototipos en lugar de los propios datos (Biehl et al. 2016). Por el otro lado, los algoritmos de *Learning Vector Quantization* (LVQ), en sus distintas versiones, son una alternativa a estos algoritmos enfocada principalmente en mantener un número pequeño de elementos en memoria.

2.3.1. Algoritmo LVQ

El algoritmo LVQ fue propuesto por Kohonen (1995) para resolver, en su primera versión, problemas de clasificación. Su comportamiento se basa en la actualización de los prototipos existentes cuando un nuevo vector de variables explicativas $\mathbf{x} \in \mathbb{R}^d$ con su correspondiente etiqueta $y \in \mathbb{R}$ son tomados de la muestra.

Llamaremos **ganador** (y denotaremos su índice por s_1) al prototipo con menor distancia euclidiana al dato tomado de la muestra. Así, el prototipo ganador es actualizado en base a la coincidencia o no de las clases del prototipo ganador y del dato de la muestra, la distancia entre ambos y el ratio de aprendizaje, η, según la siguiente fórmula:

$$\mathbf{w}_{s_1} \leftarrow \mathbf{w}_{s_1} + \eta \cdot \psi(y_{s_1}, y_u) \cdot (\mathbf{x}_u - \mathbf{w}_{s_1}), \tag{2.1}$$

donde la función ψ definida por

$$\psi(x, y) = \begin{cases} +1 & \text{si } x = y, \\ -1 & \text{en otro caso,} \end{cases}$$

nos indica la dirección en la que se actualiza el prototipo ganador según si la clase de este coincide con la del nuevo dato o no. Explicado en términos coloquiales, el prototipo se moverá en la dirección del nuevo dato si su etiqueta coincide, mientras que se alejará de este si su etiqueta es distinta. Así, el LVQ clásico se resume en el Algoritmo 1.

El algoritmo LVQ expuesto sienta, mediante su método de actualización de prototipos, las bases de toda una serie de algoritmos y variaciones que comparten, en mayor o menor medida, este mismo formato de actualización.

Los primeros ejemplos de variaciones del algoritmo LVQ surgen de manera natural al reemplazar la distancia euclidiana por otra. Entre las posibles distancias sustitutas se encuentran las distancias de Minkowski (de las que la distancia euclidiana es un caso concreto), la distancia de Mahalanobis (la cual tiene la ventaja de tener en cuenta la covarianza de las variables y ayuda a reducir problemas de escala) o distancias "kernelizadas" (que pueden transformar problemas no lineales de clasificación en problemas más simples en un espacio de mayor dimensión). Para un mayor desarrollo se puede consultar Biehl et al. (2016).

Algoritmo 1 Learning vector quantization (LVQ).

1: Fijar el número de prototipos N_G.
2: Inicializar los N_G prototipos.
3: Introducir un nuevo dato $\mathbf{x} \in \mathbb{R}^d$ (con clase $y \in \mathbb{R}$).
4: Buscar dentro de G al prototipo ganador: $s_1 = \arg\min_{c \in G} \|\mathbf{x} - \mathbf{w}_c\|$.
5: **if** $y = y_{s_1}$ **then**
6: $\quad \mathbf{w}_{s_1} \leftarrow \mathbf{w}_{s_1} + \eta(\mathbf{x} - \mathbf{w}_{s_1})$
7: **else** ▷ Ecuación (2.1)
8: $\quad \mathbf{w}_{s_1} \leftarrow \mathbf{w}_{s_1} - \eta(\mathbf{x} - \mathbf{w}_{s_1})$
9: **end if**
10: Ir al paso 3 para procesar un nuevo dato hasta que no queden datos de entrenamiento.

Existen modificaciones del LVQ más complejas con multitud de diversos objetivos, como pueden ser el mejorar la velocidad de convergencia o la capacidad de adaptación de este a nuevas clases. En nuestro caso, estaremos especialmente interesados en las modificaciones que conviertan el algoritmo LVQ en un algoritmo de aprendizaje incremental. Además, también resulta interesante suprimir, en primer lugar, la necesidad del algoritmo LVQ de que el experimentador establezca previamente el número de prototipos empleados e inicialice estos con algún valor (en la práctica ocasiona que el número de prototipos por clase se asigne de manera arbitraria o a criterio del experimentador), así como introducir un método dentro del algoritmo que lidie con la presencia de ruido y datos atípicos (*outliers*) en la muestra.

2.3.2. Algoritmo ILVQ

El *Incremental Learning Vector Quantization* (ILVQ) es un algoritmo propuesto por Xu et al. (2012), que realiza la adaptación al caso incremental mediante principalmente tres innovaciones con respecto al algoritmo LVQ original:

- **Mecanismo de inserción de nuevos prototipos**, buscando que asegure el aprendizaje del algoritmo tanto entre distintas clases como dentro de cada clase. Así, datos con una nueva clase serán añadidos al conjunto de prototipos, mientras que para datos cuya clase está compartida con algún prototipo, estos solo serán añadidos al conjunto de prototipos cuando sean suficientemente distintos de los ya almacenados (**condición de la distancia umbral**).

- La **distancia umbral** propuesta para la inclusión de prototipos debe ser **adaptativa**, de forma que no crezca indefinidamente con el tiempo.

- **Mecanismo de eliminación de prototipos obsoletos**, el cual penaliza prototipos que no han sido recientemente empleados como el prototipo ganador para los datos más recientes.

Con estas modificaciones, el algoritmo ILVQ se puede resumir mediante la Tabla 2.1 y los Algoritmos 2 y 3.

Con el fin de suprimir la necesidad de que el experimentador determine cuántos prototipos emplear y con qué valor inicializarlos tomamos los dos primeros datos recibidos como prototipos. Tras esto, procesamos un nuevo dato $\mathbf{x}$. En caso de que pertenezca a una nueva clase, es inmediatamente añadido al conjunto de prototipos G, con el fin de mejorar el aprendizaje entre clases. Si no pertenece a una nueva clase, se decidirá su entrada como nuevo prototipo si verifica las condiciones de distancia umbral con respecto a los dos prototipos más cercanos, lo que mejorará en este caso el aprendizaje dentro de la clase (líneas 1 a 8).

En caso de que la condición de entrada de nuevos prototipos no se verifique se procede a actualizar los elementos del algoritmo que guardan las relaciones topológicas de los prototipos (líneas 9 a 16):

Algoritmo 2 Incremental learning vector quantization (ILVQ).

1: Inicializar G para que contenga los 2 primeros datos de la muestra de entrenamiento.
2: Inicializar $E = \emptyset$.
3: Introducir un nuevo dato $\mathbf{x} \in \mathbb{R}^d$ (con etiqueta $y \in \mathbb{R}$).
4: Buscar dentro de G al prototipo ganador ($s_1 = \arg\min_{c \in G} \|\mathbf{x} - \mathbf{w}_c\|$) y al subcampeón ($s_2 = \arg\min_{c \in G \setminus \{s_1\}} \|\mathbf{x} - \mathbf{w}_c\|$).
5: **if** ($\mathbf{x} \in$ una nueva clase) $\vee \|\mathbf{x} - \mathbf{w}_{s_1}\| > T_{s_1} \vee \|\mathbf{x} - \mathbf{w}_{s_2}\| > T_{s_2}$ **then**
6: $\quad G \leftarrow G \cup \{\mathbf{x}\}$
7: $\quad$ Volver al paso 3 para procesar un nuevo dato.
8: **end if**
9: **if** $(s_1, s_2) \notin E$ **then**
10: $\quad E \leftarrow E \cup \{(s_1, s_2)\}$
11: $\quad \text{age}(s_1, s_2) \leftarrow 0$
12: **end if**
13: **for** $s_i \in N_{s_1}$ **do** ▷ Itera por los prototipos que comparten una arista con s_1
14: $\quad \text{age}(s_1, s_i) \leftarrow \text{age}(s_1, s_i) + 1$
15: **end for**
16: $M_{s_1} \leftarrow M_{s_1} + 1$
17: **if** $\text{clase}(s_1) = \text{clase}(\mathbf{x})$ **then**
18: $\quad \mathbf{w}_{s_1} \leftarrow \mathbf{w}_{s_1} + \eta_1(\mathbf{x} - \mathbf{w}_{s_1})$ ▷ Actualización de prototipos en la dirección de $\mathbf{x}$
19: $\quad$ **for** $s_i \in N_{s_1}$ y $\text{clase}(s_i) \neq \text{clase}(\mathbf{x})$ **do**
20: $\quad\quad \mathbf{w}_{s_i} \leftarrow \mathbf{w}_{s_i} - \eta_2(\mathbf{x} - \mathbf{w}_{s_i})$ ▷ Aparta los prototipos con clase distinta a $\mathbf{x}$
21: $\quad$ **end for**
22: **else**
23: $\quad \mathbf{w}_{s_1} \leftarrow \mathbf{w}_{s_1} - \eta_1(\mathbf{x} - \mathbf{w}_{s_1})$
24: $\quad$ **for** $s_i \in N_{s_1}$ y $\text{clase}(s_i) = \text{clase}(\mathbf{x})$ **do**
25: $\quad\quad \mathbf{w}_{s_i} \leftarrow \mathbf{w}_{s_i} + \eta_2(\mathbf{x} - \mathbf{w}_{s_i})$
26: $\quad$ **end for**
27: **end if**
28: $E \leftarrow \{(s_1, s_2) \in E \mid \text{age}(s_1, s_2) < \mathit{AgeOld}\}$
29: **if** el número de iteración es múltiplo entero del parámetro λ **then**
30: $\quad$ Eliminar los prototipos s_i del conjunto G que no tienen prototipo vecino.
31: $\quad$ Eliminar los prototipos s_i cuyo con un solo prototipo vecino y $M_{s_i} < 0{,}5 \sum_{j=1}^{N_G} \frac{M_j}{N_G}$.
32: **end if**
33: Volver al paso 3 para procesar un nuevo dato.

Algoritmo 3 Cálculo de la distancia umbral T_i.

1: $T_{i_{\text{within}}} = \dfrac{1}{N_{\text{clase}(i)}} \sum_{(i,j) \in E \wedge \text{clase}(i) = \text{clase}(j)} \|\mathbf{w}_i - \mathbf{w}_j\|$
2: $T_{i_{\text{between}}} = \min_{(k_1, i) \in E \wedge \text{clase}(i) \neq \text{clase}(k_1)} \|\mathbf{w}_i - \mathbf{w}_{k_1}\|$
3: **if** $T_{i_{\text{between}}} < T_{i_{\text{within}}}$ **then**
4: $\quad T_{i_{\text{between}}} = \min_{(k_2, i) \in E \wedge \text{clase}(i) \neq \text{clase}(k_2) \wedge k_1 \neq k_2} \|\mathbf{w}_i - \mathbf{w}_{k_2}\|$
5: **end if**
6: Volver al paso 2 hasta que $T_{i_{\text{between}}} \geq T_{i_{\text{within}}}$
7: $T_i = T_{i_{\text{between}}}$
8: Devolver T_i

Tabla 2.1: Notación empleada en el algoritmo ILVQ.

Notación	Descripción
G	Conjunto de prototipos
N_G	Número de prototipos en G
E	Conjunto de aristas
$\mathbf{w}_i$	Valores del prototipo i
M_i	Recuento de "victorias" del prototipo i
T_i	Distancia umbral del prototipo i
N_i	Conjunto de vecinos del prototipo i
age(i, j)	Edad de la arista que une los prototipos i y j
η_1, η_2	Ratios de aprendizaje

- El conjunto E almacena las aristas que unen al prototipo ganador y al subcampeón de cada iteración (a los que consideraremos "vecinos").
- El parámetro age(i, j) lleva el recuento de la última iteración en que ha intervenido la arista (i, j).
- El parámetro M_c lleva el recuento de veces que el prototipo c ha sido el prototipo ganador.

Tras esto, se produce la actualización de los prototipos ya existentes siguiendo una regla similar a la del algoritmo LVQ. Si $\mathbf{x}$ y el prototipo ganador pertenecen a la misma clase, el prototipo ganador se mueve en la dirección de $\mathbf{x}$, mientras que los vecinos del ganador que tienen clase distinta a $\mathbf{x}$ son apartados en dirección contraria. En caso de que $\mathbf{x}$ y el ganador pertenezcan a clases distintas, la actualización se produce al revés, es decir, el ganador es alejado de $\mathbf{x}$ y sus vecinos con misma clase que $\mathbf{x}$ se mueven en su dirección (líneas 17 a 27).

En último lugar, se eliminan todas aquellas aristas cuya edad supere un parámetro llamado *AgeOld* previamente fijado (línea 28). Además, cada cierto número de iteraciones se procede a la depuración de prototipos. Se eliminan los prototipos que no tienen vecino y los que, teniendo un solo vecino, tienen un recuento de victorias pobre. En la práctica, esto se corresponde con eliminar muestras aisladas y prototipos en clases de baja densidad, respectivamente (líneas 29 a 32).

Cálculo de la distancia umbral T_i

La distancia umbral juega un papel fundamental en el algoritmo ILVQ, al determinar cuándo un nuevo dato debe ser considerado como prototipo, aun perteneciendo a la misma clase que otros prototipos ya almacenados. Como ya se ha mencionado, es importante que esta distancia sea adaptativa en lugar de fija, para evitar que el conjunto G crezca de forma indefinida, lo que ocasionaría sobreajustes así como echar por tierra el objetivo de reducir el consumo de memoria que conlleva tener muchos datos

almacenados. Así, la distancia umbral se actualiza cada vez que un prototipo i se convierte en el ganador o el subcampeón.

Lógicamente, la distancia umbral debe ser menor que la distancia entre clases del prototipo i para conseguir distinguir prototipos de diferentes clases. Además, también debe ser mayor que la distancia dentro de la clase del prototipo i para evitar discriminar prototipos potencialmente útiles. Para asegurar esto, la distancia entre clases se toma como el mínimo de las distancias entre i y sus vecinos con distinta clase, y la distancia dentro de la clase se toma como la media de las distancias entre i y sus vecinos con misma clase.

El Algoritmo 3 detalla de manera precisa los pasos empleados para el cálculo de las distancias umbral.

2.3.3. Algoritmo XuILVQ

Es sencillo ver que el algoritmo ILVQ presentado solo aporta información sobre el aprendizaje del algoritmo, no especificando nada sobre el método empleado para la predicción. Como se explicará cuando se presente la librería River, esta parte del algoritmo es la que se corresponde con la función `learn_one()`. A efectos de la implementación de este algoritmo en River, se acuña en González-Soto et al. (2022) el nombre de algoritmo XuILVQ para referirse al híbrido nacido de combinar el algoritmo ILVQ de clasificación con la estructura de aprendizaje y predicción requerida por River.

Para la tarea de predicción, estos autores deciden emplear un algoritmo de k-NN ponderado. Así, dado un vector $\mathbf{x}$ para el que se desea obtener una predicción, se tomarán los k prototipos más cercanos al vector y se les asignará un peso en función de su distancia a $\mathbf{x}$, asignando el vector a la clase con mayor peso. Así, el algoritmo de predicción (que como veremos en el siguiente capítulo se corresponde con la función `predict_one()` de River) se puede resumir mediante la Tabla 2.2 y el Algoritmo 4, donde la función **Softmax**[1] se usa para representar la distribución de probabilidad sobre las distintas clases.

Además de añadir el algoritmo de predicción, el algoritmo XuILVQ presenta dos ligeras modificaciones a nivel de código en el cálculo de la distancia umbral con respecto al algoritmo ILVQ original con el fin de evitar casos patológicos (González-Soto et al. 2022).

2.3.4. Otros algoritmos incrementales de regresión

En comparación con su contraparte para problemas de clasificación, no es tanta la literatura que hay sobre algoritmos incrementales de regresión. A continuación, vamos a presentar los cuatro algoritmos con los que será comparada la adaptación del algoritmo XuILVQ. En el caso de querer consultar más algoritmos incrementales de regresión puede ser de utilidad acudir a River (Montiel et al. 2021).

k-NN Regressor (k-NN)

En su implementación en River, este algoritmo de los k vecinos más próximos opera mediante un parámetro llamado **window_size** que controla el número de muestras de entrenamiento que almacena en memoria, siendo dichas muestras los prototipos en este caso.

[1] La función **Softmax** es una generalización de la función logística, empleada para comprimir un vector n-dimensional de valores reales arbitrarios en un vector n-dimensional de valores reales acotados en el rango $[0, 1]$. La función viene dada por

$$\begin{aligned} \sigma\colon \mathbb{R}^n &\longrightarrow [0,1]^n \\ \mathbf{z} &\longmapsto \sigma(\mathbf{z}) = (\sigma_1(\mathbf{z}), \ldots, \sigma_n(\mathbf{z})), \end{aligned}$$

con $\sigma_i(\mathbf{z}) = \dfrac{e^{z_i}}{\sum_{k=1}^{n} e^{z_k}}$ para cada $i \in \{1, \ldots, n\}$.

Tabla 2.2: Notación empleada en el algoritmo de predicción de XuILVQ

Notación	Descripción
$\mathbf{x}$	Vector de entradas
K	Conjunto de los k prototipos más cercanos a $\mathbf{x}$
$\mathbf{w}_i$	Valores del prototipo i
y_i	Clase del prototipo i
$\hat{\mathbf{y}}$	Conjunto de probabilidades de pertenencia a cada clase
$\hat{\mathbf{y}}_{y_i}$	Probabilidad de que $\mathbf{x}$ pertenezca a la clase y_i

Algoritmo 4 Algoritmo de predicción de XuILVQ.

1: Encontrar los k prototipos más cercanos al vector $\mathbf{x}$.
2: Inicializar $\hat{\mathbf{y}}$ a 0 para todo y en K.
3: **for** $(\mathbf{w}_i, y_i) \in K$ **do**
4: **if** $\|\mathbf{x} - \mathbf{w}_i\| = 0$ **then**
5: Actualizar $\hat{\mathbf{y}}_{y \neq y_i} = 0$ y $\hat{\mathbf{y}}_{y=y_i} = 1$.
6: Ir al paso 12.
7: **else**
8: $\hat{\mathbf{y}}_{y_i} \leftarrow \hat{\mathbf{y}}_{y_i} + \frac{1}{\|\mathbf{x}-\mathbf{w}_i\|}$
9: **end if**
10: **end for**
11: Actualizar $\hat{\mathbf{y}} \leftarrow \mathbf{Softmax}(\hat{\mathbf{y}})$.
12: Devolver $\arg\max_{y_i} \hat{\mathbf{y}}_{y_i}$.

Es importante resaltar que este algoritmo funciona con un presupuesto de memoria prefijado, con lo que, en un principio, su comparación con nuestra adaptación no resultará excesivamente ilustrativa. Aún así, se ha comprobado que la adaptación mejora a este algoritmo en términos de memoria.

Adaptive Random Forest Regressor (ARFR)

Como ya se ha mencionado, los *Random Forests* son métodos de ensamblaje ampliamente usados en tareas de clasificación y regresión. La implementación de River se basa en el trabajo de Gomes et al. (2017). En él, se presentan los *Random Forests* adaptativos como adaptación del algoritmo clásico para lidiar con flujos de datos y con el *concept drift*. Esta adaptación se basa principalmente en un remuestreo *bootstrap* apropiado para suprimir el no disponer de toda la información así como en limitar la división de cada hoja a un subconjunto de las variables explicativas. Es recomendable consultar Gomes et al. (2017) para un desarrollo más detallado.

Hoeffding Adaptive Tree Regressor (HATR)

Este algoritmo se basa en el árbol de Hoeffding, un árbol de decisión incremental para flujos de datos, y lo usa como pilar sobre el que construir un algoritmo capaz de lidiar con el *concept drift*.

Esto puede realizarse de dos maneras distintas, mediante una ventana móvil o mediante una ventana adaptativa, siendo esta última la empleada en la implementación en River. Puede consultarse Bifet y Gavaldà (2009) para un desarrollo más detallado así como ciertos resultados teóricos que garantizan su buen funcionamiento.

***Passive Agressive Regressor* (PAR)**

Este algoritmo se basa en la resolución de un problema de optimización. Así, en cada iteración el algoritmo recibe un nuevo dato y trata de predecir su valor en base al modelo lineal previo. Tras hacer la predicción, el algoritmo recibe el auténtico valor de la respuesta y sufre un aumento en la función de coste según si la predicción se desvía del valor real más que una cantidad prefijada ε. Tras esto, el algoritmo genera un nuevo modelo lineal que se empleará en la siguiente iteración. Para un desarrollo más exhaustivo, así como varios resultados sobre cotas de error puede consultarse Crammer et al. (2006).

Capítulo 3

Adaptación del algoritmo XuILVQ a regresión multivariante

En este capítulo se detallan los pasos seguidos en el desarrollo de la adaptación del algoritmo y su posterior implementación en Python. En la Sección 3.1 se introducen las librerías empleadas en la adaptación del algoritmo, centrándose sobre todo en la librería River, por ser esta fundamental en la implementación. La Sección 3.2 se centra en explicar los mecanismos empleados en la adaptación del algoritmo XuILVQ. Por último, en la Sección 3.3, se presenta el pseudocódigo de las tres adaptaciones del algoritmo XuILVQ propuestas.

3.1. Librerías empleadas

En la Tabla 3.1 puede verse un breve resumen de las principales librerías empleadas en Python a lo largo del trabajo.

Numpy es una librería clásica de Python centrada en la creación y el manejo de estructuras llamadas *arrays*. No obstante, en este trabajo la estructura principal con la que se opera son los diccionarios de Python, con lo que el empleo de esta librería quedó principalmente relegado al uso de algunas de sus funciones en diversos pasos del algoritmo.

Matplotlib es una librería de generación de todo tipo de gráficas y representaciones visuales de los datos. En nuestro caso, ha sido empleada de forma posterior a la adaptación para obtener gráficas que permitiesen comparar el comportamiento del algoritmo con respecto a otros o su evolución según el tamaño del *dataset*, entre otras gráficas.

Tabla 3.1: Librerías de Python empleadas.

Librería	**Referencia**	**Breve descripción**
River	Montiel et al. (2021)	Algoritmos de *online* e *incremental learning*
Scikit-learn	Pedregosa et al. (2011)	Tareas de aprendizaje supervisado y no supervisado
Numpy	Harris et al. (2020)	Creación y procesado de *arrays*
Matplotlib	Hunter (2007)	Representación gráfica y visualización de datos

Scikit-learn podría ser considerada uno de los pilares del ML en Python, al poseer un amplio catálogo de algoritmos y herramientas para las principales tareas tanto de aprendizaje supervisado

como no supervisado: clasificación, regresión, *clustering*, reducción de dimensiones, etc. En nuestro caso, se ha empleado esta librería en un paso concreto del algoritmo consistente en el agrupamiento de los datos en *clusters* así como en la posterior validación del algoritmo, generando un *dataset* de regresión lineal sintético para comprobar su correcto funcionamiento.

La librería River, al ser el pilar sobre el que se sustenta toda la implementación, se comenta más en detalle a continuación.

3.1.1. Librería River

La librería River (Montiel et al. 2021) surge como fusión de las librerías Creme (Halford et al. 2019) y Scikit-multiflow (Montiel et al. 2018), con el objetivo principal de ser la librería insignia para la realización de todo tipo de tareas de ML con flujos de datos (de las que el *incremental learning* forma parte). Hoy en día es la librería más completa en lo referente a algoritmos de *online* e *incremental learning*, además de ser de código abierto.

Dada la naturaleza del problema a tratar, resulta lógico tomar River como base sobre la que construir nuestro algoritmo, más aún sabiendo que alberga multitud de métodos y métricas de evaluación de algoritmos. Además, aunque la mayoría de los *datasets* que contiene son para problemas de clasificación, también contiene varios *datasets* para regresión con los que comprobar la eficiencia del algoritmo.

Por tanto, resulta vital comprender la forma en que son implementados los algoritmos en River. La mayoría de los algoritmos de clasificación y regresión de River tienen en común dos funciones, llamadas `learn_one()` y `predict_one()`, respectivamente. La función `learn_one()` se centra únicamente en la parte del algoritmo relacionada con el aprendizaje. Es empleada para actualizar el modelo con un nuevo dato. Por otro lado, la función `predict_one()` es la encargada de, partiendo del último modelo disponible, proporcionar una predicción sobre la respuesta asociada a un dato $\mathbf{x}$.

3.2. Adaptación a regresión

Habiendo introducido previamente el algoritmo XuILVQ y la librería River, estamos ya en condiciones de explicar cómo se ha adaptado este algoritmo de clasificación a regresión, al que nos referiremos a partir de ahora como XuRILVQ. Con el fin de mantener las buenas propiedades que presenta el algoritmo XuILVQ, la adaptación busca realizar la menor cantidad posible de cambios en el algoritmo original.

Para la función `predict_one()` se ha decidido emplear un algoritmo de k-NN basado en prototipos, de manera similar al algoritmo de predicción del XuILVQ. Se ha optado por la versión más básica de este, en lugar de uno ponderado o kernelizado, por ser el más sencillo de implementar. Así, el algoritmo de predicción XuRILVQ queda determinado por la Tabla 3.2 y el Algoritmo 5, donde $\mathbb{1}$ denota la función indicadora[1].

Algoritmo 5 Algoritmo de predicción de XuRILVQ.

1: Encontrar los k prototipos más cercanos al vector $\mathbf{x}$.
2: Devolver $\hat{y}(\mathbf{x}) = \dfrac{1}{k}\displaystyle\sum_{i=1}^{n} \mathbb{1}_{\{\mathbf{w}_i \in K\}} \cdot y_i$.

[1]Dados un conjunto X y un subconjunto, $A \subset X$, llamamos función indicadora (o función característica) a la función que indica la pertenencia de los elementos de X a A. Esta función viene dada por

$$\begin{aligned} \mathbb{1}_A \colon X &\longrightarrow \{0,1\} \\ x &\longmapsto \begin{cases} 1, & x \in A, \\ 0, & x \notin A. \end{cases} \end{aligned}$$

Tabla 3.2: Notación empleada en el algoritmo de predicción de XuRILVQ

Notación	Descripción
$\mathbf{x}$	Vector de entradas
K	Conjunto de los k prototipos más cercanos a $\mathbf{x}$
$\mathbf{w}_i$	Valores del prototipo i
y_i	Valor de la variable respuesta del prototipo i
$\hat{y}(\mathbf{x})$	Predicción de la variable respuesta para $\mathbf{x}$

Se ha optado por mantener el algoritmo de predicción lo más simple posible, al ser el aspecto novedoso del trabajo el uso de prototipos en el algoritmo de aprendizaje (función `learn_one()`). Siempre sería posible cambiar posteriormente el algoritmo de predicción por otro método de regresión con mejores propiedades.

Para el algoritmo de aprendizaje se ha decidido tener en cuenta dos enfoques distintos, sobre los cuales se comentarán a continuación ciertos aspectos relevantes:

- Convertir el problema de regresión en un problema de clasificación.
- Emplear una modificación del método de actualización de prototipos.

3.2.1. Convertir regresión en clasificación

Los problemas de clasificación se pueden ver como casos concretos de problemas de regresión, para los que ya existen algoritmos concretos. El ejemplo más común en clasificación binaria es la regresión logística, sobre la que se puede consultar más al respecto de su origen y formulación en Cramer (2002). Además, también cuenta con su adaptación para problemas de clasificación multiclase, que conocemos como regresión logística multinomial (Greene 2012).

En cambio, no es tanto ni tan óptimo lo que está escrito en la otra dirección, es decir, convirtiendo problemas de regresión en clasificación. En Salman y Kecman (2012) se aborda precisamente este tema al convertir un SVM de regresión en uno de clasificación.

La base detrás de este enfoque radica en considerar el problema de regresión como un problema de clasificación multiclase. Para ello, se realiza una discretización del dominio de la variable respuesta en una cantidad razonable de clases (intervalos) para, posteriormente, agrupar los datos según estas nuevas clases y ejecutar el algoritmo. Una consecuencia negativa de esto es que favorece la consideración de datos atípicos como clases en lugar de despreciarlos. Dado que el XuILVQ ya tiene incorporado un mecanismo de eliminación de datos atípicos, este enfoque parece idóneo para formular una adaptación del algoritmo. Aún así, resulta vital tener en cuenta ciertos aspectos al realizar esta discretización para convertir un problema de regresión en uno de clasificación.

¿Cuántas clases se crean?

Para dar una respuesta a esta pregunta que sea satisfactoria es necesario tener dos consideraciones en cuenta.

En primer lugar, el algoritmo ILVQ presentaba como una de sus ventajas frente al LVQ que el experimentador no necesitaba hacer suposiciones sobre la cantidad de prototipos a emplear ni su

posición inicial, al incorporar mecanismos automáticos de entrada, salida y actualización de estos. De igual modo, que sea el propio algoritmo el que decida en cada iteración el número de clases en los que dividir los datos según el número de prototipos resulta igualmente deseable.

La segunda consideración viene con respecto al consumo de memoria. Como ya se comentó, el objetivo final es producir un algoritmo capaz de correr en dispositivos de IoT con baja capacidad de computación y memoria. Por lo tanto, nuestra adaptación debe consumir la menor memoria posible.

Con estos dos objetivos en mente, se propone que sea la **fórmula de Sturges** (Sturges 1926) la que decida de manera automática en cada iteración el número de clases en las que dividir los datos.

La fórmula de Sturges fue originalmente propuesta para la estimación de densidades. Se deriva de la distribución binomial y viene dada por

$$c = \lceil \log_2 n \rceil + 1,$$

donde c denota el número de clases y n el número de datos (en nuestro caso, prototipos).

Esta fórmula se comporta de manera muy satisfactoria cuando los datos son aproximadamente normales. Además, para una cantidad de datos en torno a $n = 100$ la fórmula de Sturges proporciona un valor muy similar a la fórmula de Scott (Scott 1979), otra fórmula clásica de la estimación de densidades basada en el ancho de banda que optimiza el IMSE (error cuadrático medio integrado). Se puede consultar Scott (2009) para una comparación más detallada entre ambas fórmulas.

Como única modificación a la fórmula de Sturges se ha cambiado la función techo por el redondeo clásico (que denotaremos por $[\cdot]$) para evitar errores en las primeras iteraciones donde se obtenían más clases de los prototipos que había.

¿Qué criterio se emplea para agrupar en clases?

Al igual que al decidir el número de clases a emplear, resulta deseable nuevamente que sea el propio algoritmo el que agrupe los datos en la cantidad de clases previamente calculada según la regla de Sturges propuesta. Para esta tarea se ha propuesto un método clásico del aprendizaje no supervisado como es el ***clustering***.

Los algoritmos de *clustering* son procedimientos de agrupación de datos según algún criterio previamente establecido de forma que los datos del mismo grupo (llamado *cluster*) son más similares entre sí que los pertenecientes a otros *clusters* en términos del criterio previamente mencionado.

Estos algoritmos pueden dividirse a su vez en diversos grupos según el criterio y modo en que los *clusters* son creados, entre los que destacan los métodos particionales y jerárquicos. Nos centraremos en el *clustering* jerárquico al ser el método empleado para la adaptación del algoritmo.

Los algoritmos jerárquicos se basan en la construcción sucesiva de *clusters* usando *clusters* construidos previamente. Estos algoritmos pueden, a su vez, ser aglomerativos o divisivos. Los algoritmos aglomerativos comienzan considerando cada elemento como un *cluster* separado y los van uniendo de manera sucesiva, mientras que los algoritmos divisivos comienzan empleando todo el conjunto de datos como un *cluster* para posteriormente dividirlo sucesivamente en *clusters* más pequeños.

Un paso clave en la aplicación de estos algoritmos es la selección del criterio de formación de nuevos *clusters*. Este criterio acostumbra a ser una medida de distancia (como pueden ser la euclidiana, Manhattan o del coseno) acompañada de una medida de disimilitud entre *clusters*. Esta puede ser expresada mediante la **fórmula de actualización de disimilitud de Lance-Williams** (Murtagh y Contreras 2012). Así, si los datos (o *clusters*) i y j son aglomerados en $i \cup j$, la medida de disimilitud entre este nuevo *cluster* y otro dato (o *cluster*) k viene dada por

$$\alpha_i d(i,k) + \alpha_j d(j,k) + \beta d(i,j) + \gamma \|d(i,k) - d(j,k)\|,$$

donde $d(\cdot,\cdot)$ es la función de distancia empleada y $\alpha_i, \alpha_j, \beta$ y γ definen la medida concreta de disimilitud. En Murtagh y Contreras (2012) pueden consultarse distintas medidas de disimilitud según los valores de estas variables, como pueden ser *single-link* o *complete-link* entre otras.

Para nuestra adaptación, se ha decidido emplear como distancia la euclidiana y como medida de disimilitud la proporcionada por el método de Ward, formulado en Ward (1963), el cual es un criterio

que minimiza la suma residual de cuadrados (RSS). En términos de la fórmula de Lance-Williams, sus parámetros son

$$\alpha_i = \frac{|i| + |k|}{|i| + |j| + |k|}, \quad \beta = \frac{|k|}{|i| + |j| + |k|}, \quad \gamma = 0,$$

donde $|i|$ denota el número de elementos en el *cluster* i.

Para la implementación en Python se ha empleado la función `AgglomerativeClustering` de la librería Scikit-learn, la cual en su módulo sobre *clusters* incorpora una gran variedad de distintos algoritmos. La medida de Ward es la empleada por defecto, pudiendo emplearse también otras como el *single-link* o el *complete-link* sin más que variar el parámetro `linkage` de la función.

3.2.2. Modificación del método de actualización de prototipos

En Grbovic y Vucetic (2009) se introduce una modificación del algoritmo de LVQ para adaptarlo a regresión. Esta modificación surge de considerar el problema de regresión como uno de optimización con distintas funciones de coste posibles.

Centrándose en el último apartado de este artículo y tomando el método de actualización empleado, el otro enfoque que se ha seguido consiste en la creación de un híbrido entre este algoritmo y el XuILVQ, adoptando también los estándares de aprendizaje y predicción de la librería River. Así, del algoritmo propuesto por Grbovic y Vucetic se tomaría el criterio de actualización de prototipos, mientras que del algoritmo XuILVQ se mantendría todo lo referente a los criterios de entrada y salida de estos.

3.3. Algoritmo XuRILVQ

Habiendo dado respuesta a las principales preguntas surgidas sobre los distintos enfoques tomados, estamos en condiciones de presentar las distintas versiones propuestas para el algoritmo XuRILVQ con el correspondiente pseudocódigo. En todos los pseudocódigos de las versiones propuestas, el color rojo indicará que esas líneas se han modificado o añadido con respecto al algoritmo XuILVQ original.

3.3.1. Primera y segunda versión del algoritmo XuRILVQ

El algoritmo XuRILVQ en su primera versión (XuRILVQ-1) queda definido por la Tabla 2.1 ya empleada en el algoritmo ILVQ original y el Algoritmo 6, además del Algoritmo 3 para el cálculo de las distancias umbral. Esta versión es la más semejante al algoritmo XuILVQ, al únicamente incorporar como novedad los pasos referentes al *clustering* para convertir el problema de regresión en uno de clasificación.

El algoritmo XuILVQ, al ser un algoritmo de clasificación, emplea el valor numérico de la respuesta únicamente como indicador de las distintas clases mientras que en un problema de regresión la respuesta es una variable con algún tipo de significado. En ese sentido, es razonable pensar que quizás sería deseable actualizar también el valor de la respuesta cuando se actualizan los prototipos. Esta actualización se realizaría de igual manera que se actualizan los valores de las variables explicativas, acercando los valores del mismo *cluster* y alejando los de *clusters* distintos.

Así, incluyendo también la actualización de la variable respuesta de acuerdo a lo antes mencionado, se tiene la versión 2 del algoritmo XuRILVQ (XuRILVQ-2). Esta queda definida por el Algoritmo 7, junto con, de nuevo, la Tabla 2.1 y el Algoritmo 3.

3.3.2. Tercera versión del algoritmo XuRILVQ

Esta tercera y última versión es la que incorpora el método de actualización de prototipos propuesto en Grbovic y Vucetic (2009). Al contrario que en el algoritmo XuILVQ y en sus dos primeras adaptaciones propuestas, aquí solamente se actualizan los dos prototipos más cercanos, el ganador y el subcampeón, siguiendo la que se conoce como *window-rule* formulada en Kohonen (1990) para el

algoritmo SOM previamente mencionado. Esta regla determina cuando se produce la actualización en base a las distancias entre el vector $(\mathbf{x}, y)$ y los prototipos $(\mathbf{w}_{s_1}, y_{s_1})$ y $(\mathbf{w}_{s_2}, y_{s_2})$. Así, si denotamos por $e_i = (y - y_i)^2$ a la diferencia entre la variable respuesta del nuevo dato y la del prototipo i esta última versión (XuRILVQ-3) queda resumida mediante el Algoritmo 8 acompañado nuevamente de la Tabla 2.1 y el Algoritmo 3.

Algoritmo 6 Primera versión del algoritmo XuRILVQ (XuRILVQ-1).

1: Inicializar G para que contenga los 2 primeros datos de la muestra de entrenamiento.
2: Inicializar $E = \emptyset$.
3: Introducir un nuevo dato $\mathbf{x} \in \mathbb{R}^d$ (con variable respuesta $y \in \mathbb{R}$).
4: Calcular $n_{\text{cluster}} = 1 + [\log_2 N_G]$.
5: Realizar un *clustering* aglomerativo de los prototipos y $\mathbf{x}$ con respecto a la variable respuesta.
6: Sincronizar el *cluster* asignado a los prototipos con respecto a pasadas iteraciones.
7: Buscar dentro de G al prototipo ganador ($s_1 = \arg\min_{c \in G} \|\mathbf{x} - \mathbf{w}_c\|$) y al subcampeón ($s_2 = \arg\min_{c \in G \setminus \{s_1\}} \|\mathbf{x} - \mathbf{w}_c\|$).
8: **if** ($\mathbf{x}$ es un nuevo *cluster*) $\vee \|\mathbf{x} - \mathbf{w}_{s_1}\| > T_{s_1} \vee \|\mathbf{x} - \mathbf{w}_{s_2}\| > T_{s_2}$ **then**
9: $\quad G \leftarrow G \cup \{\mathbf{x}\}$
10: $\quad$ Volver al paso 3 para procesar un nuevo dato.
11: **end if**
12: **if** $(s_1, s_2) \notin E$ **then**
13: $\quad E \leftarrow E \cup \{(s_1, s_2)\}$
14: $\quad \text{age}(s_1, s_2) \leftarrow 0$
15: **end if**
16: **for** $s_i \in N_{s_1}$ **do**
17: $\quad \text{age}(s_1, s_i) \leftarrow \text{age}(s_1, s_i) + 1$
18: **end for**
19: $M_{s_1} \leftarrow M_{s_1} + 1$
20: **if** $\textit{cluster}(s_1) = \textit{cluster}(\mathbf{x})$ **then** ▷ Solo actualiza las variables explicativas
21: $\quad \mathbf{w}_{s_1} \leftarrow \mathbf{w}_{s_1} + \eta_1(\mathbf{x} - \mathbf{w}_{s_1})$
22: $\quad$ **for** $s_i \in N_{s_1}$ y $\textit{cluster}(s_i) \neq \textit{cluster}(\mathbf{x})$ **do**
23: $\quad\quad \mathbf{w}_{s_i} \leftarrow \mathbf{w}_{s_i} - \eta_2(\mathbf{x} - \mathbf{w}_{s_i})$
24: $\quad$ **end for**
25: **else**
26: $\quad \mathbf{w}_{s_1} \leftarrow \mathbf{w}_{s_1} - \eta_1(\mathbf{x} - \mathbf{w}_{s_1})$
27: $\quad$ **for** $s_i \in N_{s_1}$ y $\textit{cluster}(s_i) = \textit{cluster}(\mathbf{x})$ **do**
28: $\quad\quad \mathbf{w}_{s_i} \leftarrow \mathbf{w}_{s_i} + \eta_2(\mathbf{x} - \mathbf{w}_{s_i})$
29: $\quad$ **end for**
30: **end if**
31: $E \leftarrow \{(s_1, s_2) \in E \mid \text{age}(s_1, s_2) < \textit{AgeOld}\}$
32: **if** el número de iteración es múltiplo entero del parámetro λ **then**
33: $\quad$ Eliminar los prototipos s_i del conjunto G que no tienen prototipo vecino.
34: $\quad$ Eliminar los prototipos s_i cuyo con un solo prototipo vecino y $M_{s_i} < 0{,}5 \sum_{j=1}^{N_G} \frac{M_j}{N_G}$.
35: **end if**
36: Volver al paso 3 para procesar un nuevo dato.

Algoritmo 7 Segunda versión del algoritmo XuRILVQ (XuRILVQ-2).

1: Inicializar G para que contenga los 2 primeros datos de la muestra de entrenamiento.
2: Inicializar $E = \emptyset$.
3: Introducir un nuevo dato $\mathbf{x} \in \mathbb{R}^d$ (con variable respuesta $y \in \mathbb{R}$).
4: Calcular $n_{\text{cluster}} = 1 + [\log_2 N_G]$.
5: Realizar un *clustering* aglomerativo de los prototipos y $\mathbf{x}$ con respecto a la variable respuesta.
6: Sincronizar el *cluster* asignado a los prototipos con respecto a pasadas iteraciones.
7: Buscar dentro de G al prototipo ganador ($s_1 = \arg\min_{c \in G} \|\mathbf{x} - \mathbf{w}_c\|$) y al subcampeón ($s_2 = \arg\min_{c \in G \setminus \{s_1\}} \|\mathbf{x} - \mathbf{w}_c\|$).
8: **if** ($\mathbf{x}$ ∈ un nuevo *cluster*) $\vee \|\mathbf{x} - \mathbf{w}_{s_1}\| > T_{s_1} \vee \|\mathbf{x} - \mathbf{w}_{s_2}\| > T_{s_2}$ **then**
9: $G \leftarrow G \cup \{\mathbf{x}\}$
10: Volver al paso 3 para procesar un nuevo dato.
11: **end if**
12: **if** $(s_1, s_2) \notin E$ **then**
13: $E \leftarrow E \cup \{(s_1, s_2)\}$
14: $\text{age}(s_1, s_2) \leftarrow 0$
15: **end if**
16: **for** $s_i \in N_{s_1}$ **do**
17: $\text{age}(s_1, s_i) \leftarrow \text{age}(s_1, s_i) + 1$
18: **end for**
19: $M_{s_1} \leftarrow M_{s_1} + 1$
20: **if** $\mathit{cluster}(s_1) = \mathit{cluster}(\mathbf{x})$ **then** ▷ Actualiza también la variable respuesta
21: $(\mathbf{w}_{s_1}, y_{s_1}) \leftarrow (\mathbf{w}_{s_1}, y_{s_1}) + \eta_1(\mathbf{x} - \mathbf{w}_{s_1}, y - y_{s_1})$
22: **for** $s_i \in N_{s_1}$ y $\mathit{cluster}(s_i) \neq \mathit{cluster}(\mathbf{x})$ **do**
23: $(\mathbf{w}_{s_i}, y_{s_i}) \leftarrow (\mathbf{w}_{s_i}, y_{s_i}) - \eta_2(\mathbf{x} - \mathbf{w}_{s_i}, y - y_{s_i})$
24: **end for**
25: **else**
26: $(\mathbf{w}_{s_1}, y_{s_1}) \leftarrow (\mathbf{w}_{s_1}, y_{s_1}) + \eta_1(\mathbf{x} - \mathbf{w}_{s_1}, y - y_{s_1})$
27: **for** $s_i \in N_{s_1}$ y $\mathit{cluster}(s_i) = \mathit{cluster}(\mathbf{x})$ **do**
28: $(\mathbf{w}_{s_i}, y_{s_i}) \leftarrow (\mathbf{w}_{s_i}, y_{s_i}) + \eta_2(\mathbf{x} - \mathbf{w}_{s_i}, y - y_{s_i})$
29: **end for**
30: **end if**
31: $E \leftarrow \{(s_1, s_2) \in E \mid \text{age}(s_1, s_2) < \mathit{AgeOld}\}$
32: **if** el número de iteración es múltiplo entero del parámetro λ **then**
33: Eliminar los prototipos s_i del conjunto G que no tienen prototipo vecino.
34: Eliminar los prototipos s_i cuyo con un solo prototipo vecino y $M_{s_i} < 0{,}5 \sum_{j=1}^{N_G} \frac{M_j}{N_G}$.
35: **end if**
36: Volver al paso 3 para procesar un nuevo dato.

Algoritmo 8 Tercera versión del algoritmo XuRILVQ (XuRILVQ-3).

1: Inicializar G para que contenga los 2 primeros datos de la muestra de entrenamiento
2: Inicializar $E = \emptyset$
3: Introducir un nuevo dato $\mathbf{x} \in \mathbb{R}^d$ (con variable respuesta $y \in \mathbb{R}$)
4: Calcular $n_{\text{cluster}} = 1 + [\log_2 N_G]$
5: Realizar un *clustering* aglomerativo de los prototipos y $\mathbf{x}$ con respecto a la variable respuesta
6: Sincronizar el *cluster* asignado a los prototipos con respecto a pasadas iteraciones
7: Buscar dentro de G al prototipo ganador ($s_1 = \arg\min_{c \in G} \|\mathbf{x} - \mathbf{w}_c\|$) y al subcampeón ($s_2 = \arg\min_{c \in G \setminus \{s_1\}} \|\mathbf{x} - \mathbf{w}_c\|$)
8: **if** ($\mathbf{x} \in$ un nuevo *cluster*) $\vee \|\mathbf{x} - \mathbf{w}_{s_1}\| > T_{s_1} \vee \|\mathbf{x} - \mathbf{w}_{s_2}\| > T_{s_2}$ **then**
9: $\quad G \leftarrow G \cup \{\mathbf{x}\}$
10: $\quad$ Volver al paso 3 para procesar un nuevo dato
11: **end if**
12: **if** $(s_1, s_2) \notin E$ **then**
13: $\quad E \leftarrow E \cup \{(s_1, s_2)\}$
14: $\quad \text{age}(s_1, s_2) \leftarrow 0$
15: **end if**
16: **for** $s_i \in N_{s_1}$ **do**
17: $\quad \text{age}(s_1, s_i) \leftarrow \text{age}(s_1, s_i) + 1$
18: **end for**
19: $M_{s_1} \leftarrow M_{s_1} + 1$
20: **if** $\min\left(\frac{\|(\mathbf{x}, y) - (\mathbf{w}_{s_1}, y_{s_1})\|}{\|(\mathbf{x}, y) - (\mathbf{w}_{s_2}, y_{s_2})\|}, \frac{\|(\mathbf{x}, y) - (\mathbf{w}_{s_2}, y_{s_2})\|}{\|(\mathbf{x}, y) - (\mathbf{w}_{s_1}, y_{s_1})\|}\right) > s$ **then** ▷ s parámetro entre 0,4 y 0,8
21: $\quad \mathbf{w}_{s_1} \leftarrow \mathbf{w}_{s_1} - \eta(e_{s_1} - e_{s_2})(\mathbf{x} - \mathbf{w}_{s_1})$
22: $\quad \mathbf{w}_{s_2} \leftarrow \mathbf{w}_{s_2} + \eta(e_{s_1} - e_{s_2})(\mathbf{x} - \mathbf{w}_{s_2})$
23: $\quad y_{s_1} \leftarrow y_{s_1} + \eta(y - y_{s_1})$
24: $\quad y_{s_2} \leftarrow y_{s_2} + \eta(y - y_{s_2})$
25: **else**
26: $\quad y_{s_1} \leftarrow y_{s_1} + \eta(y - y_{s_1})$
27: $\quad y_{s_2} \leftarrow y_{s_1} + \eta(y - y_{s_2})$
28: **end if**
29: $E \leftarrow \{(s_1, s_2) \in E \mid \text{age}(s_1, s_2) < \mathit{AgeOld}\}$
30: **if** el número de iteración es múltiplo entero del parámetro λ **then**
31: $\quad$ Eliminar los prototipos s_i del conjunto G que no tienen prototipo vecino.
32: $\quad$ Eliminar los prototipos s_i cuyo con un solo prototipo vecino y $M_{s_i} < 0{,}5 \sum_{j=1}^{N_G} \frac{M_j}{N_G}$.
33: **end if**
34: Volver al paso 3 para procesar un nuevo dato.

Capítulo 4

Resultados

Habiendo introducido en el capítulo anterior las distintas versiones del algoritmo XuRILVQ como adaptación del algoritmo XuILVQ, en este capítulo se procede a comentar los resultados obtenidos de la implementación en Python del algoritmo. En primer lugar, en la Sección 4.1 se introducirán brevemente las herramientas usadas en la validación y comparación del modelo. Tras esto, en la Sección 4.2 se realiza una primera comparativa gráfica del comportamiento de las 3 versiones.

A continuación, en la Sección 4.3 se realizará una comparación de las dos primeras versiones del algoritmo, entre ellas y con otros algoritmos. A partir de aquí se optará por el empleo de una única versión XuRILVQ (la primera) para estudiar más en detalle. Por último, en la Sección 4.4 se valorará el comportamiento del algoritmo XuRILVQ escogido con respecto al incremento del número de dimensiones del conjunto de datos.

4.1. Herramientas de validación

4.1.1. Medida de error

Como medida del error cometido por el algoritmo se ha optado por emplear la **raíz del error cuadrático medio**. El RMSE (por sus siglas en inglés, *root-mean-squared error*) es frecuentemente usado como medida de la calidad de un modelo o estimador. En este caso, el RMSE medirá la diferencia entre los valores predichos por el modelo y los valores reales observados.

Dada una muestra de observaciones $\mathbf{x}_i \in \mathbb{R}^d$ con respuesta asociada $y_i \in \mathbb{R}$, $i \in \{1, \dots, n\}$, y un modelo que a cada vector $\mathbf{x}_i$ le asigna la predicción $\hat{y}(\mathbf{x}_i)$, definimos el error cuadrático medio del modelo como

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}(\mathbf{x}_i))^2,$$

y el RMSE vendrá entonces dado por

$$\text{RMSE} = \sqrt{\text{MSE}}.$$

Como consecuencia de esta definición resulta inmediato comprobar que el RMSE es siempre no negativo y, en caso de ser 0, esto equivaldría a un ajuste perfecto de los datos. El efecto de cada error de predicción en el RMSE es proporcional a la diferencia entre la predicción y el valor real. A causa de esto, el RMSE es especialmente sensible a la presencia de datos atípicos. Aunque es algo importante a tener en cuenta, no resulta preocupante en nuestro caso, pues el algoritmo XuRILVQ ya incorpora un mecanismo de eliminación de prototipos obsoletos fruto de datos atípicos.

4.1.2. Utilidades de River

River trae incorporado un sencillo e intuitivo sistema de creación de *pipelines*, una manera simple de encadenar transformaciones y estimadores que faciliten el entrenamiento y la validación de modelos. La estructura general de estas *pipelines* acostumbra a ser una serie de transformaciones realizadas sobre los *datasets* seguida de un estimador (que en este caso sería el algoritmo XuRILVQ). En esta ocasión, las únicas transformaciones que se han realizado sobre los *datasets* han sido la selección de las variables continuas, su estandarización y, en *datasets* donde hay alguna variable categórica, la agrupación de datos según estas categorías.

Además de este sistema de creación de *pipelines*, River también incorpora sus propias herramientas de validación de modelos. Para este fin, la función canónica de River es `progressive_val_score()`, cuyos principales parámetros son:

- **dataset**: El *dataset* sobre el que se evaluará el modelo.
- **model**: El modelo a evaluar, en nuestro caso el XuRILVQ.
- **metric**: La métrica empleada para evaluar las predicciones del modelo, en nuestro caso el RMSE.

Así, el *dataset* es convertido en un flujo de datos constante que pasa a través del modelo. En cada iteración, al modelo se le ordena que dé una predicción sobre una observación o se le actualiza con el valor real de la variable respuesta de la observación. El tiempo que pasa hasta que al modelo se le da el valor real de la variable respuesta puede ser controlado mediante el parámetro **delay**, que por defecto es 0. Ajustando este parámetro podríamos obtener estimaciones en un escenario de producción más realista donde los valores reales de la variable respuesta suelen obtenerse tras un cierto tiempo. Además, la función `progressive_val_score()` también dispone de los parámetros **show_time** y **show_memory** para llevar un seguimiento de la memoria y tiempo de ejecución consumidos por el modelo.

4.1.3. *Datasets* empleados

Para la validación del modelo se han empleado dos tipos de *datasets* distintos. En primer lugar se ha comparado el comportamiento de las tres versiones del algoritmo XuRILVQ sobre un *dataset* sintético de regresión lineal, descartando la versión XuRILVQ-3. Tras esto, se han empleado algunos de los *datasets* de regresión de los que dispone River para una primera comparación con otros algoritmos de regresión, donde se descarta la versión XuRILVQ-2 al ser más inestable que la XuRILVQ-1. Finalmente, se han creado de nuevo *datasets* sintéticos para evaluar el comportamiento del XuRILVQ-1 frente al aumento del número de dimensiones así como para compararlo con otros algoritmos de River.

Datasets de River

Los *datasets* de River sobre los que se han probado los algoritmos de regresión son:

- **TrumpApproval**: Contiene 5 variables explicativas correspondientes a 5 índices de aprobación de Donald Trump recolectados por distintas agencias de sondeo. La variable respuesta es el índice de aprobación del modelo de "FiveThirtyEight", un blog estadounidense centrado en análisis político, economía y deportes.
- **Bikes**: Contiene información sobre las estaciones de alquiler de bicicletas de la ciudad de Toulouse.
- **Taxis**: Contiene información sobre viajes en taxi en la ciudad de Nueva York.
- **ChickWeights**: Contiene datos sobre la evolución en el tiempo del peso de pollos según el tipo de dieta.

Dado que los *datasets* de TrumpApproval y el de ChickWeights poseen un tamaño muestral pequeño (1001 y 578, respectivamente) se han empleado todos los datos en la comparación. Para los *datasets* de Bikes y Taxis, al ser el tamaño muestral del orden de cientos de miles e incluso millones, se ha optado por tomar submuestras de tamaño 10000.

Datasets sintéticos

Para la comparación de las distintas versiones del algoritmo XuRILVQ así como para su comparación con otros algoritmos de regresión, se ha optado por usar la función `make_regression()` de la librería Scikit-learn, la cual genera un *dataset* sintético donde la variable respuesta y las distintas explicativas mantienen una relación lineal. Las variables explicativas son tomadas de manera aleatoria de una distribución normal estándar y la variable respuesta es generada como una combinación lineal aleatoria de las variables explicativas junto con un cierto error gaussiano, donde los coeficientes del modelo son generados a partir de una distribución uniforme en el intervalo $[0, 1]$ y multiplicados por 100. Para más información al respecto de la generación del *dataset* puede ser de utilidad consultar la documentación de Scikit-learn (Pedregosa et al. 2011). Sus principales parámetros son:

- **n_samples**: Tamaño de la muestra creada.
- **n_features**: Número de variables explicativas.
- **n_targets**: Número de variables respuesta, por defecto 1.
- **noise**: Desviación típica del error gaussiano que es aplicado a los datos generados.
- **random_state**: Semilla que determina la generación aleatoria del *dataset*. Se fija para reproducibilidad de los resultados.
- **coef**: Parámetro booleano que determina si se devuelven los valores de los coeficientes del modelo, por defecto *False*.

Empleando esta función en reiteradas ocasiones para distintas semillas se han generado diversas réplicas de los *datasets*. El objetivo de esto es que los valores de RMSE o de prototipos almacenados sean promediados para que los resultados obtenidos sean menos sensibles a perturbaciones.

4.2. Primera comparación entre versiones

En primer lugar, resulta interesante el ver de manera gráfica cómo se comportan las distintas versiones del algoritmo XuRILVQ. Para ello, generamos un *dataset* bidimensional sintético de 3000 datos mediante la función `make_regression()` y hacemos que las tres versiones iteren sobre todos los datos. En la Figura 4.1 se puede ver el *dataset* antes de ser procesado por las distintas versiones, donde cada punto morado se corresponde con un dato sintético. En las próximas figuras, los puntos morados de nuevo van a representar los datos mientras que en verde se representarán los prototipos almacenados tras iterar el correspondiente algoritmo.

Iterando el XuRILVQ-1 sobre el *dataset* se obtiene el resultado de la Figura 4.2. Como podemos comprobar, los prototipos se ajustan bien a la recta de regresión. Es importante observar, eso sí, como el algoritmo parece tender a suprimir los puntos de los extremos, lo que podría agravar el efecto frontera. Sobre este efecto, presente en problemas de estimación no paramétrica de curvas, se pueden consultar Cheng et al. (1997) o Jones (1993). Esta versión almacena 106 prototipos, lo cual supone un gran ahorro de memoria comparado con los 3000 datos iniciales.

Al iterar el algoritmo XuRILVQ-2 sobre el *dataset* obtenemos los resultados de la Figura 4.3. Aunque nuevamente los datos se ajustan aceptablemente bien a la recta de regresión, es cierto que hay un empeoramiento con respecto a la primera versión del algoritmo. Aunque aquí no se puede apreciar, en otras pruebas realizadas, este algoritmo parecía ser inicialmente más sensible a datos atípicos que el

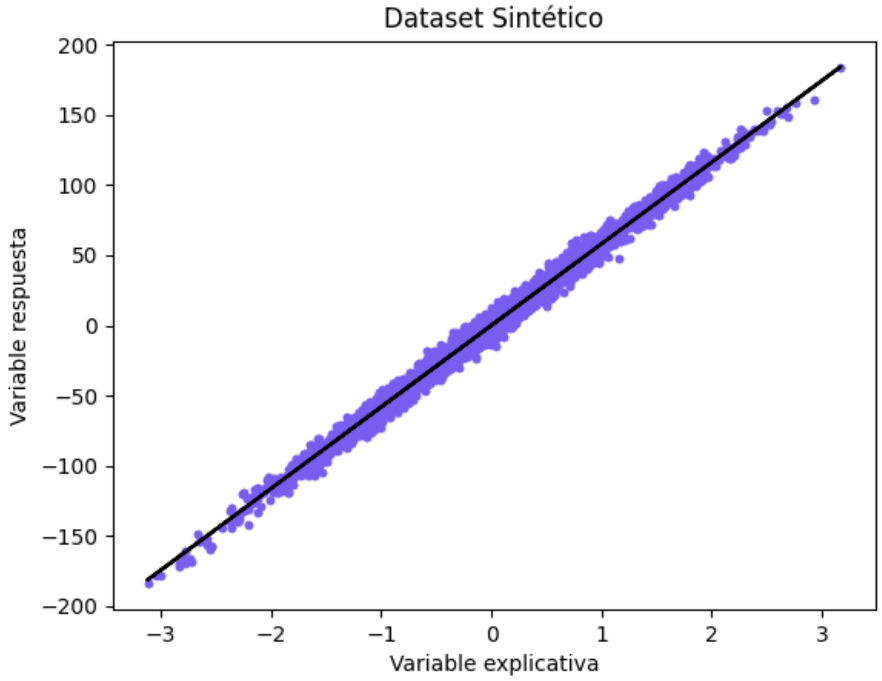

Figura 4.1: *Dataset* sintético empleado para comparar las versiones del algoritmo XuRILVQ.

XuRILVQ-1, aunque permitiendo al algoritmo iterar lo suficiente terminaba solventando el problema al eliminar el prototipo atípico. Esta versión almacena 84 prototipos lo cual resulta una mejora con respecto a la primera versión.

Por último, en la Figura 4.4 podemos ver el desastroso resultado de iterar la versión XuRILVQ-3 por el *dataset*. Para que puedan aparecer representados los prototipos no es posible mantener la misma escala en el eje X que en las otras gráficas. Aunque ese problema se debe en parte a un dato en concreto extremadamente alejado (y que eventualmente desaparecería), el resto de prototipos sigue encontrándose muy lejos de la recta de regresión en comparación con las otras dos versiones del algoritmo XuRILVQ (valores del orden de 10^4 para la variable explicativa cuando esta debería oscilar entre -3 y $+3$). En cantidad de prototipos almacenados, esta versión resulta ser la que menos almacena, con un total de 76. Ante los malos resultados de esta versión, se decide seguir únicamente con las dos primeras versiones.

4.3. Comparación con *datasets* de River

Se han comparado las dos primeras versiones del algoritmo XuRILVQ (correspondientes a los Algoritmos 6 y 7) con diversos regresores de River, comparando el RMSE y la memoria ocupada. Esta comparación puede observarse en la Tabla 4.1.

Comparando ambas versiones del algoritmo podemos comprobar como, en general, es la primera (XuRILVQ-1) la que mejores resultados arroja. Aunque en uso de memoria la segunda versión mejora a la primera, el RMSE de la primera versión es siempre mejor y se encuentra más en sintonía con los resultados de los otros regresores. Por ejemplo, para el *dataset* de Taxis el RMSE de la segunda versión se dispara hasta 24151, mientras que el resto de algoritmos arrojan un RMSE en un rango entre 3800 y 4800. De igual forma, aunque en menor magnitud, ocurre también con el *dataset* de

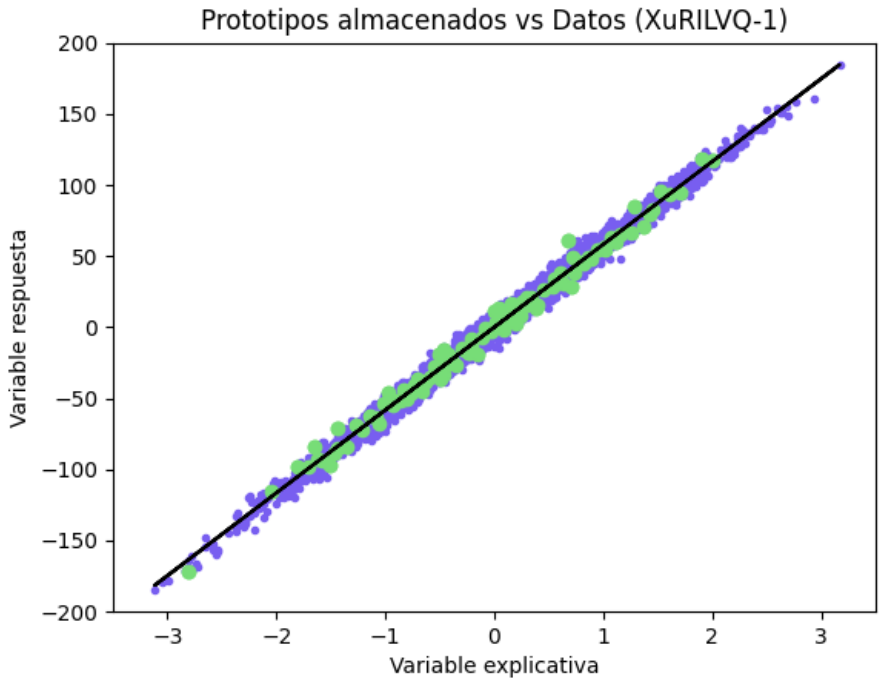

Figura 4.2: Prototipos almacenados del XuRILVQ-1.

Bikes, arrojando un RMSE de 57,4537 mientras el resto oscila entre 6 y 9. Para el *dataset* de Trump Approval sorprendentemente son ambas versiones del XuRILVQ las que mejores valores de RMSE y memoria arrojan.

Centrando únicamente la atención en el apartado de la memoria, resulta obvio que ninguno de los algoritmos es capaz de competir con el PAR. Por el lado contrario, el ARFR es siempre el que mayor memoria consume. En medio se encuentran, variando el orden según el *dataset*, los algoritmos de HATR, k-NN y ambas versiones del XuRILVQ. En tres de los cuatro *datasets*, ambos algoritmos XuRILVQ ocupan menos memoria que los otros dos, aunque siendo distintos los *datasets* para cada uno.

En general, consideramos que es la primera versión del algoritmo la que mejor se ajusta al objetivo original del trabajo y, de ahora en adelante, será a la que nos refiramos cuando hablemos de algoritmo XuRILVQ. Aún así, siempre sería posible intercambiar una por otra según el foco de atención se encuentre en minimizar la memoria consumida al mínimo o tratar de encontrar un equilibrio entre el aumento de RMSE y la reducción de memoria.

Habiendo visto que el algoritmo XuRILVQ tiene un comportamiento en términos de RMSE similar a los algoritmos de regresión de River, se realiza una comparación más detallada entre ellos en términos de memoria (ver Figura 4.5). En esta figura el eje X indica la dimensión del *dataset* sintético (es decir, el número de variables explicativas) y el eje Y representa la memoria consumida y, sobre este eje de coordenadas, cada recta representa la evolución del promedio sobre 6 réplicas de *datasets* sintéticos de la memoria consumida por cada algoritmo.

Al igual que se comprobaba en la Tabla 4.1, el algoritmo que menos memoria consume es siempre el PAR. Seguido de este se encuentra el algoritmo XuRILVQ. Aunque a partir de 500/600 dimensiones presenta un crecimiento en la memoria consumida, esto no representa algo preocupante, al seguir estando por debajo de algoritmos de eficacia probada como son k-NN, ARFR y HATR. De estos tres, el que más memoria consume es precisamente este último mientras que k-NN consume menos memoria

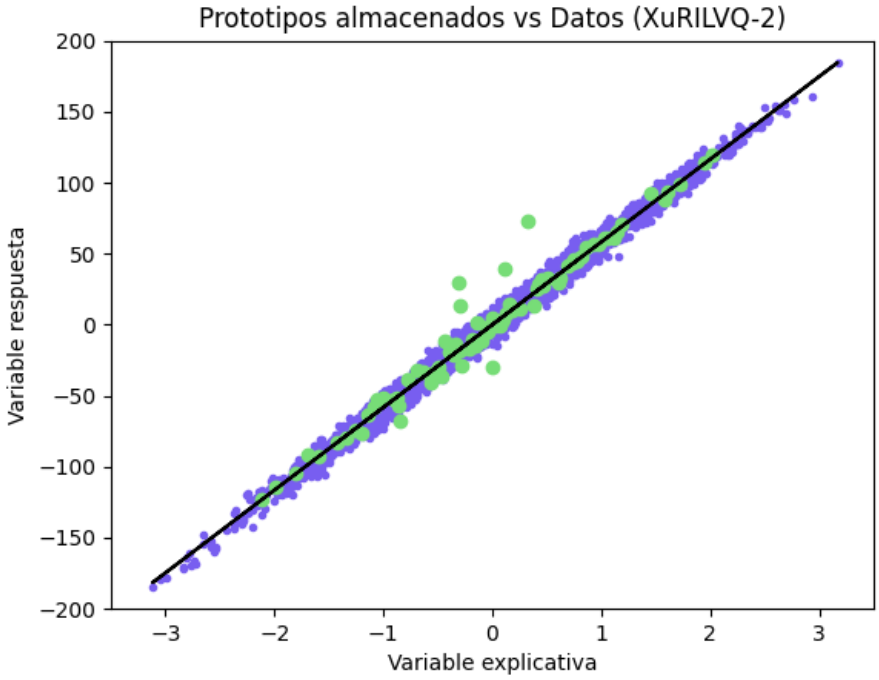

Figura 4.3: Prototipos almacenados del XuRILVQ-2.

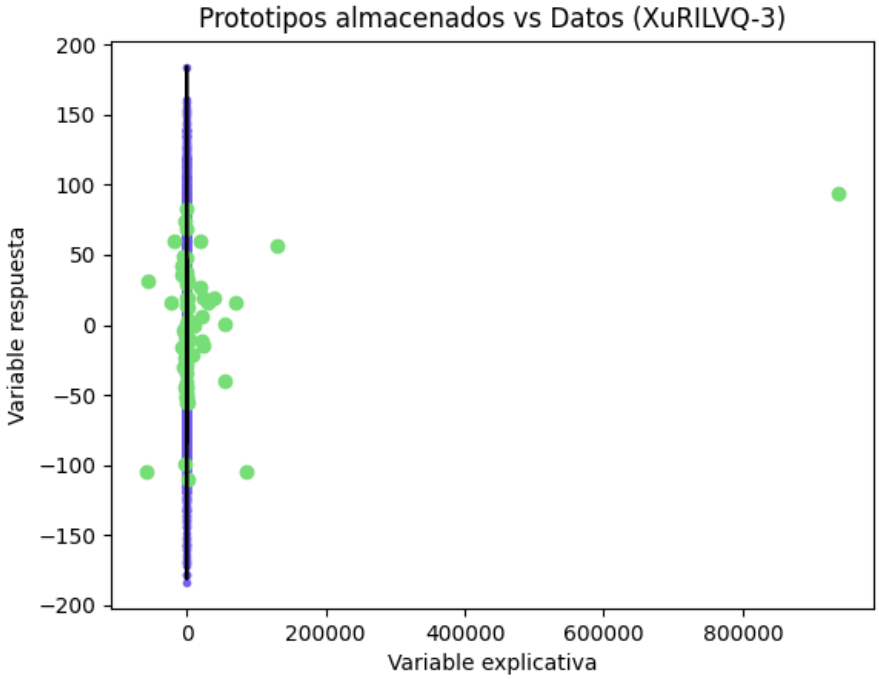

Figura 4.4: Prototipos almacenados del XuRILVQ-3.

Tabla 4.1: Relación de RMSE y memoria para distintos *datasets* y regresores.

	Trump Approval		**Bikes**		**Taxis**		**Chick Weights**	
	RMSE	Memoria (KB)	RMSE	Memoria (KB)	RMSE	Memoria (KB)	RMSE	Memoria (KB)
ARF Regressor	3,030421	2744,32	6,349207	2088,96	3824,67	38451,2	28,655663	2713,6
HAT Regressor	3,905844	822,51	6,369236	76,47	3946,5	5570,56	21,36078	213,05
***k*-NN Regressor**	1,596333	360,01	6,54099	294,79	4234,36	286,29	35,482041	241,85
PA Regressor	6,662732	4,54	9,045693	4,87	3839,8	4,24	22,806265	27,71
XuRILVQ - 1	0,823372	275,05	6,562256	260,18	4710,76	597,66	29,392108	113,52
XuRILVQ - 2	0,892451	136,47	57,453722	75,89	24151,05	398,04	36,573635	115,66

que ARFR para pocas dimensiones aunque eventualmente se igualan.

4.4. Comportamiento con respecto a la dimensión

Aunque al aumentar el número de variables explicativas se podría pensar que mejora la capacidad predictiva del modelo, esto en general no es cierto. Esto se debe a que al aumentar el número de dimensiones el volumen del espacio crece tan rápido que todos los datos disponibles se encuentran dispersos y aislados. Este problema no está presente en espacios de bajas dimensiones y es lo que se conoce con el nombre de **maldición de la dimensionalidad**. Este término se acuñó por primera vez en Bellman (1957) para englobar todos estos fenómenos que ocurren en campos como son el análisis numérico, la combinatoria, el muestreo o el ML al trabajar en espacios de alta dimensión.

Resulta por tanto realmente importante que el algoritmo XuRILVQ sea capaz de comportarse bien en espacios de alta dimensionalidad pues en general, en un contexto de IoT, serán muchas las variables explicativas de las que se dispondrá. Aunque ya se ha visto que no destaca negativamente en este aspecto, con el fin de comprobar el comportamiento del algoritmo para *datasets* de diversas dimensiones se ha realizado la gráfica de la Figura 4.6. En este caso el eje X indica el índice de iteración, el eje Y representa la memoria consumida y cada curva representa el algoritmo XuRILVQ aplicado sobre datasets sintéticos de 5, 10, 50, 100 y 500 dimensiones, respectivamente.

Lo primero que salta a la vista al observar la gráfica son las caídas que presentan las curvas. Esas caídas repentinas en el consumo de memoria se corresponden con los pasos del algoritmo correspondientes a la depuración y eliminación de prototipos obsoletos, que se producen en este caso concreto cada 150 iteraciones del algoritmo.

Otro aspecto a tener en cuenta tiene que ver con que para *datasets* de 5 y 10 dimensiones y las suficientes iteraciones, el algoritmo XuRILVQ consume más memoria que para un *dataset* de 50 dimensiones (y casi más incluso que el de 100 dimensiones). Esto no deja de ser un comportamiento extraño pues, en general, la memoria consumida para almacenar prototipos va a ser mayor cuantas más variables tenga el prototipo.

Para tratar de comprobar a qué se debe este comportamiento, se ha calculado, al cabo de las 1000 iteraciones, el número medio de prototipos que almacena el algoritmo XuRILVQ para cada uno de los *datasets*, así como el promedio de RMSE. Los resultados se pueden ver en la Tabla 4.2.

Como se puede comprobar, para *datasets* de baja dimensión el número medio de prototipos almacenado es considerablemente mayor que para los *datasets* de alta dimensión. Esto explica el mayor consumo de memoria de los *datasets* de 5 y 10 variables explicativas. Para los *datasets* de 50, 100 y 500 dimensiones el número medio de prototipos almacenado resulta sorprendentemente bajo. Esto deja en claro que el consumo de memoria en estos casos se debe a lo costoso de almacenar datos con tantas variables y no a la cantidad en sí de datos almacenados. Este comportamiento parece indicar la presencia de problemas relacionados con la maldición de la dimensionalidad, ya que si todos los datos

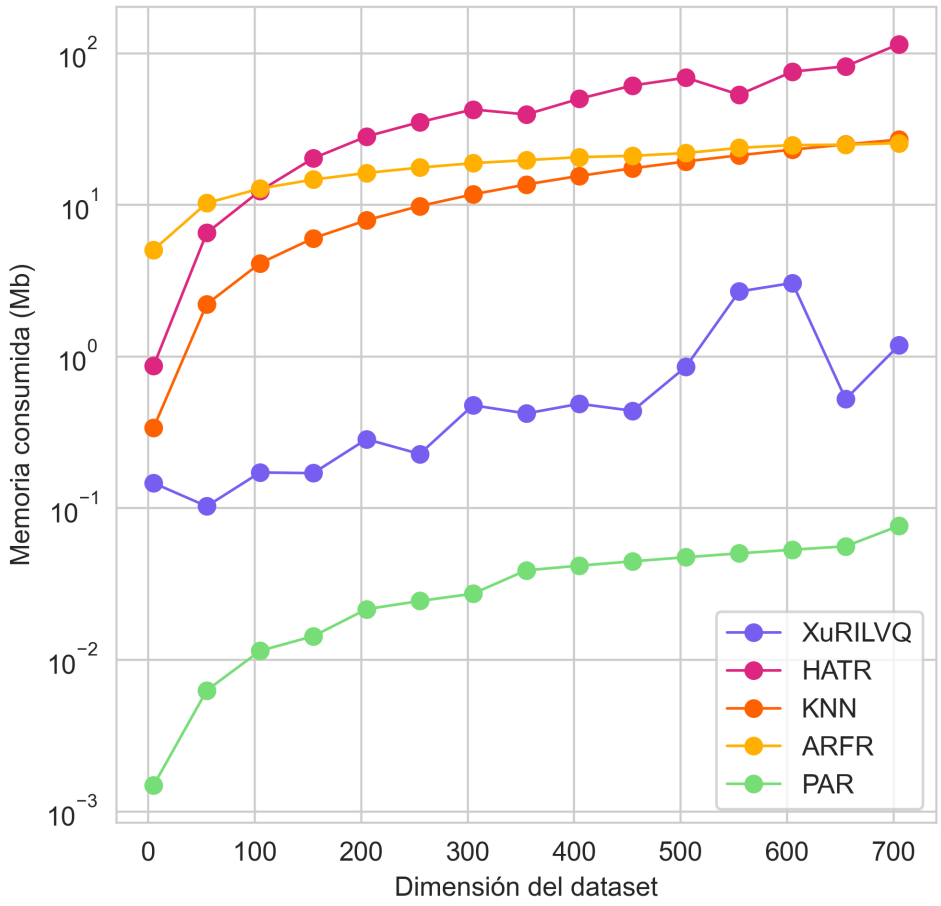

Figura 4.5: Comparación del consumo de memoria de los distintos algoritmos para 6 réplicas promediadas.

se encuentran dispersos en el espacio, el algoritmo va a considerarlos datos atípicos y eventualmente los eliminará. Sería interesante realizar alguna modificación para evitar este comportamiento en *datasets* de alta dimensión.

En lo referente a RMSE, se puede comprobar como, efectivamente, este aumenta al añadir dimensiones al *dataset*. No obstante, no lo hace tanto como el bajo número de prototipos almacenados invitaría a pensar. Esto puede deberse, en parte, al hecho de que el efecto que produce la maldición de la dimensionalidad no escala de manera lineal con el número de dimensiones. En Verleysen y François (2005) se puede comprobar, entre otros fenómenos, que el volumen de una hiperesfera d-dimensional de radio 1 alcanza su valor máximo para $d = 5$ para, posteriormente, decrecer hacia 0 rápidamente al seguir aumentando d, o que la proporción entre el volumen de una hiperesfera de radio 1 y un hipercubo de lado 2 se hace prácticamente nulo para $d \geq 10$. En nuestro caso concreto, se puede comprobar como, al pasar de 5 a 10 dimensiones, el RMSE prácticamente se duplica pasando de 68,83 a 127,25 mientras que al pasar de 50 a 500 dimensiones el aumento de RMSE es de apenas 16 puntos, de 181,01 a 197,11. Sin embargo, aunque el aumento en el RMSE no parece ser preocupante, sería necesario realizar un

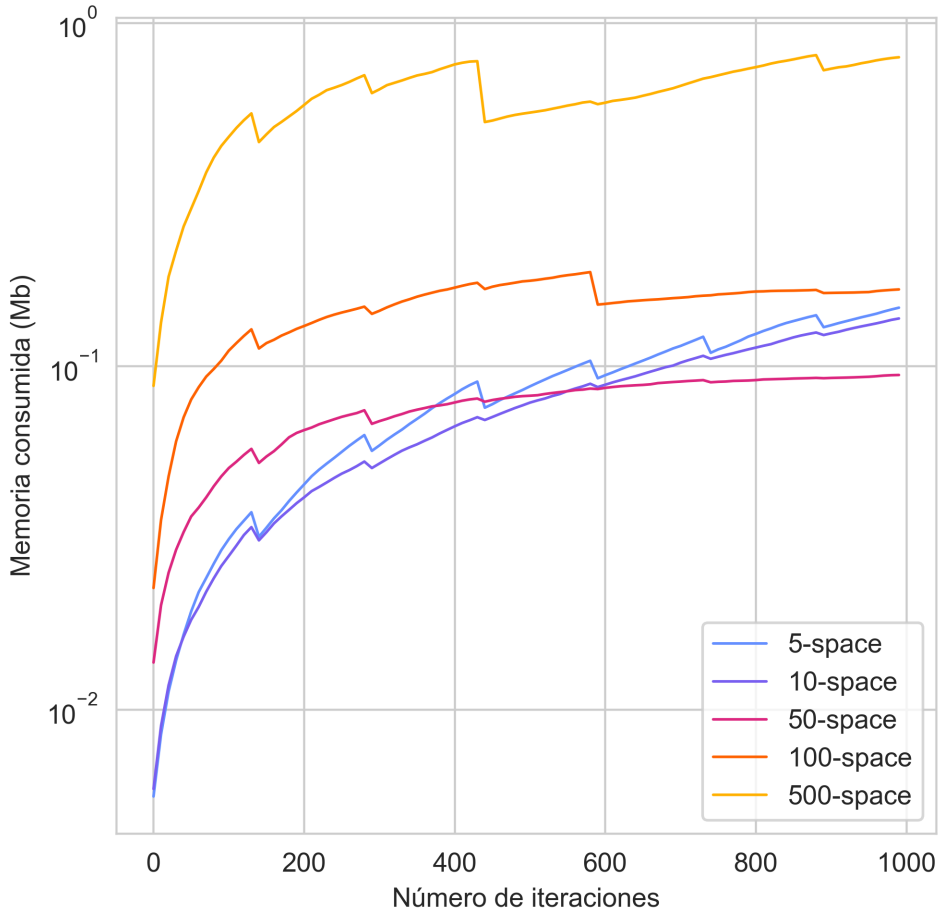

Figura 4.6: Comparación de memoria según dimensión para 50 réplicas promediadas.

estudio más en profundidad sobre estos resultados.

Tabla 4.2: Relación del número medio de prototipos almacenados según dimensión del *dataset*.

	Dimensión del dataset				
	5	10	50	100	500
Media de prototipos	134,32	98,1	29,08	32,66	39,48
Media de RMSE	68,83	127,25	181,01	183,86	197,11

Capítulo 5

Conclusiones

Tras haber detallado en el anterior capítulo todo lo referente a los resultados obtenidos para las distintas versiones del algoritmo, este último capítulo finaliza el trabajo con comentarios sobre qué conclusiones se han extraído del desarrollo y la implementación de esta adaptación del algoritmo XuILVQ. En la Sección 5.1 se mencionan posibles cambios que podrían hacerse al algoritmo con el fin de comparar su comportamiento con el que se ha observado hasta ahora. Por último, en la Sección 5.2 se comentan posibles líneas futuras de trabajo a desarrollar en relación con el algoritmo propuesto.

5.1. Posibles cambios a realizar en el algoritmo

Aunque los resultados obtenidos han resultado ser prometedores y dan a entender que el algoritmo XuRILVQ es competitivo con otros algoritmos incrementales de regresión, es probable que el algoritmo presente margen de mejora en diversos aspectos. Es por eso precisamente que, en esta sección, se detallan algunos de los cambios que podrían ser probados en el algoritmo de aprendizaje del XuRILVQ para tratar de obtener un mejor algoritmo o con un mejor desempeño en un escenario concreto. Con respecto al algoritmo de predicción, este siempre podría ser reemplazado por otro método de regresión de manera sencilla, dependiendo de lo que se busque con dicha regresión. Para el algoritmo de aprendizaje se sugieren dos cambios posibles.

5.1.1. Cambios relacionados con el *clustering*

Como ya se ha mencionado, es deseable que el mecanismo que decide la cantidad de *clusters* en la que se van a dividir los datos sea automático. Aún manteniendo esto en mente, son muchos los mecanismos que se podrían emplear para calcular el valor de n_{cluster}.

Dado que los resultados obtenidos empleando la fórmula de Sturges han sido satisfactorios, podría resultar interesante probar con otros valores clásicos de la estimación de densidades como son los dados por la fórmula de Scott (Scott 1979) anteriormente mencionada o la fórmula de Freedman-Diaconis (Freedman y Diaconis 1981). Esta última parte de la fórmula de Scott, pero hace uso del rango intercuartílico de los datos para hacerla menos sensible a datos atípicos. Otro posible enfoque sería el de escoger un n_{cluster} que minimice el error de validación cruzada, aunque esto resulta computacionalmente más costoso y podría no ser óptimo en este sentido.

Al ser el consumo de memoria algo que se tuvo en cuenta a la hora de desarrollar el algoritmo, resultaría igualmente deseable que el algoritmo de *clustering* sea lo más eficiente posible. Dentro de los algoritmos de *clustering* jerárquico, aunque los aglomerativos suelen ser los más eficientes computacionalmente, el coste en memoria de estos algoritmos es cúbico con el número de datos, n, y en el mejor de los casos puede ser reducido a $O(n^2 \log n)$ o incluso $O(n^2)$ (Béjar 2013).

Para mejorar este aspecto, se podría emplear el algoritmo Ckmeans.1d.dp, el cual está diseñado para agrupar datos univariantes asegurando optimalidad, eficiencia y reproducibilidad. Presentado

originalmente en Wang y Song (2011), el algoritmo asegura un coste en memoria de $O(kn)$, donde k denota el número de *clusters* y n el número de datos. Además, eliminando el tiempo empleado en ordenar los valores de entrada, el orden del algoritmo con respecto al tiempo también es lineal. Este algoritmo está originalmente implementado en R aunque en la actualidad ya cuenta con adaptaciones a JavaScript, Python, Matlab, etc.

5.1.2. Posibles añadidos

Como se ha mencionado, el *concept drift* puede llegar a ser un problema en el contexto de algoritmos de aprendizaje incremental. Como remedio a este problema, un posible acercamiento sería el uso de un modelo dual de almacenamiento consistente en un modelo de memoria a corto plazo y otro de memoria a largo plazo (que se conoce como **arquitectura de memoria autoajustable**, SAM por las siglas en inglés). Para un ejemplo de empleo de este sistema puede consultarse Jakob et al. (2022).

Con este sistema, por ejemplo, se almacenarían modelos de regresión cada cierto número de iteraciones y tras recibir un nuevo *batch* de datos el modelo que dichos datos generasen podría ser comparado con el almacenado para ayudar a detectar la presencia de *concept drift* y distinguirlo de datos atípicos. Tras esto, el mejor modelo pasaría a la memoria a largo plazo mientras que la memoria a corto plazo se vaciaría para dar paso a nuevos datos.

No obstante, el principal inconveniente de este enfoque está en el consumo extra de memoria que requeriría esta característica. Sería quizás deseable el incluirlo como algo opcional que se decida emplear según la capacidad computacional del dispositivo.

5.2. Líneas de trabajo futuras

Habiendo ya comprobado que el algoritmo XuRILVQ se desempeña de manera aceptable con *datasets* de la librería River y *datasets* de regresión lineal generados sintéticamente, resulta claro que una buena primera línea de trabajo en el futuro sería el **empleo del algoritmo en flujos de datos reales**. En la vida real los datos usualmente presentan un mayor volumen de valores atípicos, por lo que sería interesante ver cómo el algoritmo XuRILVQ se desempeña frente a ellos, al haber sido diseñado con un mecanismo de eliminación de datos atípicos.

Una ventaja de los algoritmos basados en prototipos (y que el algoritmo ILVQ y todos sus descendientes mantiene) está en que estos algoritmos operan con un extra de seguridad frente a aquellos que mantienen en memoria los datos. Esto se debe a que los datos almacenados en forma de prototipos no permanecen inalterados sino que evolucionan en cada iteración, haciendo que eventualmente solo sean una mera representación de la población. Esta propiedad resulta interesante cuando se opera con datos sensibles y permite una mayor privacidad, lo cual permitiría una comunicación entre modelos conectados a una red a través de sus prototipos sin violar la privacidad de los datos.

En este sentido, otra posible línea de trabajo sería la **creación e implementación de un esquema de aprendizaje incremental colaborativo entre dispositivos IoT**. Así, los distintos dispositivos solo analizarían y operarían con los datos obtenidos por ellos mismos pero recibirían la información de los datos obtenidos por otros dispositivos en forma de prototipos. Al no violarse la privacidad de los datos, la escalabilidad de esto podría ser tan grande como se desease, sin más que aumentar el número de dispositivos que aprenden y comparten sus prototipos.

Por último, y como ya se ha comentado, se podría trabajar en la **incorporación del algoritmo XuRILVQ al catálogo de River** de algoritmos incrementales de regresión. Para esto sería necesario que el algoritmo XuRILVQ se ajuste completamente a la estructura estándar de estimadores de River, siguiendo el ejemplo de González-Soto et al. (2022).

Bibliografía

[1] Bellman RE (1957) Dynamic programming. Princeton University Press

[2] Bejar J (2013) Strategies and Algorithms for Clustering Large Datasets: A Review

[3] Biehl M, Hammer B, Villmann T (2016) Prototype-based models in machine learning. Wiley Interdisciplinary Reviews: Cognitive Science 7(2): 92-111

[4] Bifet A, Gavaldà R (2009) Adaptive learning from evolving data streams. Advances in Intelligent Data Analysis VIII: 249-260

[5] Cheng MY, Fan J, Marron JS (1997) On automatic boundary corrections. The Annals of Statistics 25(4): 1691-1708

[6] Cramer JS (2002) The Origins of Logistic Regression. Tinbergen Institute Working Paper 119(4): 167-178

[7] Crammer K, Dekel O et al. (2006) Online Passive-Aggressive Algorithms. Journal of Machine Learning Research 7: 551-585

[8] Domeniconi C, Gunopulos D (2001) Incremental support vector machine construction. Proceedings 2001 IEEE International Conference on Data Mining: 589-592

[9] Freedman D, Diaconis P (1981) On the histogram as a density estimator: L_2 theory. Zeitschrift für Wahrscheinlichkeitstheorie und Verwandte Gebiete 57(4): 453-476

[10] Geppeth A, Hammer B (2016) Incremental learning algorithms and applications. European Symposium on Artificial Neural Networks (ESANN)

[11] Gomes HM, Bifet A, Read J et al. (2017) Adaptive random forests for evolving data stream classification. Mach Learn 106: 1469-1495

[12] González-Soto M, Fernández-Castro B, Díaz-Redondo RP, Fernández-Veiga M (2022) XuILVQ: A River Implementation of the Incremental Learning Vector Quantization for IoT. PE-WASUN 2022: Proceedings of the 19th ACM International Symposium on Performance Evaluation of Wireless Ad Hoc, Sensor & Ubiquitous Networks: 1-8

[13] Grbovic M, Vucetic S (2009) Regression Learning Vector Quantization. 2009 Ninth IEEE International Conference on Data Mining: 788-793

[14] Greene WH (2012) Econometric Analysis (Seventh Edition). Pearson Education, Boston: 803-806

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

[16] Harris CR, Millman KJ, van der Walt SJ et al. (2020) Array programming with NumPy. Nature 585: 357-362

[17] Hunter JD (2007) Matplotlib: A 2D Graphics Environment. Computing in Science & Engineering 9(3): 90-95

[18] Jakob J, Artelt A, Hasenjäger M, Hammer B (2022) SAM-kNN Regressor for ONline Learning in Water Distribution. Artifical Neural Networks and Machine Learning - ICANN 2022: 752-762

[19] Jones MC (1993) Simple boundary correction for kernel density estimation. Statistics and Computing 3: 135-146

[20] Ketsu S, Mishra PK (2021): Cloud, Fog and Mist Computing in IoT: An Indication of Emerging Opportunities. IETE Technical Review 39(3): 713-724

[21] Kohonen T (1990) The self-organizing map. Proceedings of the IEEE 78(9): 1464-1480

[22] Kohonen T (1995) Learning Vector Quatization. Springer Series in Information Sciences: Self-organizing maps 30: 175-189

[23] 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

[24] Montiel J, Read J, Bifet A, Abdessalem T (2018) Scikit-multiflow: A multi-output streaming framework. The Journal of Machine Learning Research 19(72): 1-5. https://github.com/scikit-multiflow/scikit-multiflow

[25] Murtagh F, Contreras P (2012) Algorithms for hierarchical clustering: an overview. WIREs Data Mining and Knowledge Discovery 2(1): 86-97

[26] 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. https://github.com/scikit-learn/scikit-learn

[27] Polikar R, Udpa L, Udpa SS, Honavar V (2000) LEARN++: an incremental learning algorithm for multilayer perceptron networks. 2000 IEEE International Conference on Acoustics, Speech, and Signal Processing Proceedings: 3414-3417

[28] Salman R, Kecman V (2012) Regression as classification. 2012 Proceedings of IEEE Southeastcon: 1-8

[29] Scott DW (1979) On optimal and data-based histograms. Biometrika 66(3): 605-610

[30] Scott DW (2009) Sturges' rule. WIREs Computational Statistics 1(3): 303-306

[31] Sturges, HA (1926) The choice of a class interval. Journal of the American Statistical Association 21(153): 65-66

[32] Verleysen M, François D (2005) The Curse of Dimensionality in Data Mining and Time Series Prediction. Computational Intelligence and Bioinspired Systems: 758-770

[33] Wang A, Wan G, Cheng Z, Li S (2009) An incremental extremely random forest classifier for online learning and tracking. 2009 16th IEEE International Conference on Image Processing: 1449-1452

[34] Wang H, Song M (2011) Ckmeans.1d.dp: Optimal k-means clustering in one dimension by dynamic programming. The R Journal 3(2): 29-33

[35] Ward JH (1963) Hierarchical Grouping to Optimize an Objective Function. Journal of the American Association 58(301): 236-244

[36] Xiao R, Wang J, Zhang F (2000) An approach to incremental SVM learning algorithm. Proceedings 12th IEEE Internationals Conference on Tools with Artificial Intelligence: 268-273

[37] Xu Y, Shen F, Zhao J (2012) An incremental learning vector quantization algorithm for pattern classification. Neural Computing and Applications 21(6): 1205-1215