Universidad de Valladolid

ESCUELA DE INGENIERÍA INFORMÁTICA (SG)

Grado en Ingeniería Informática de Servicios y Aplicaciones

Predicción del avance de la enfermedad de Parkinson mediante técnicas de *Machine Learning*

Alumna: Silvia Muñoz Nogales

Tutor: José Vicente Álvarez Bravo

Índice

Capítulo 1. Introducción

La presente memoria aborda un desafío en la intersección entre la tecnología y la medicina: la predicción del avance de la enfermedad de Parkinson mediante técnicas de Machine Learning. Este proyecto nace de la necesidad de desarrollar herramientas más precisas y explicables que permitan mejorar los tratamientos y la gestión de esta enfermedad neurodegenerativa. En las siguientes secciones, se describen los antecedentes, objetivos y metodología que guían este trabajo.

1.1. Motivación

La enfermedad de Parkinson (EP) es una condición neurodegenerativa progresiva que afecta predominantemente a adultos mayores, aunque también puede presentarse en etapas más tempranas. Se estima que más de 10 millones de personas en el mundo viven con esta enfermedad, y su prevalencia está en aumento debido al envejecimiento global de la población. Los síntomas principales incluyen temblores, rigidez muscular, y dificultad para mantener el equilibrio, pero la enfermedad también afecta aspectos no motores, como el sueño, el estado de ánimo y la cognición, lo que agrava significativamente la calidad de vida de los pacientes y sus cuidadores.

La capacidad de predecir la progresión de la EP es crucial tanto para los pacientes como para los médicos tratantes. Al anticipar la evolución de los síntomas, se pueden diseñar planes de tratamiento más efectivos, prevenir complicaciones, y optimizar los recursos del sistema de salud. Sin embargo, la progresión de la enfermedad varía ampliamente entre los pacientes, influenciada por factores como la edad, el género, las comorbilidades y el acceso a tratamientos. Esta variabilidad presenta un desafío significativo para los modelos actuales de predicción, que a menudo no logran capturar la complejidad de la enfermedad.

Las herramientas actuales basadas en análisis estadísticos o modelos de aprendizaje automático tradicionales, como ARIMA y Support Vector Machines (SVM), han demostrado cierto éxito en tareas relacionadas, pero enfrentan limitaciones importantes. Estos métodos suelen requerir supuestos rígidos sobre la naturaleza de los datos y carecen de capacidad para modelar relaciones no lineales complejas, características de la progresión de la EP. Además, su aplicabilidad se ve reducida en casos de datos faltantes o ruidosos, una situación común en registros médicos.

Los avances recientes en técnicas de aprendizaje profundo, como los modelos Seq2Seq (Sequence-to-Sequence, es decir secuencia a secuencia), abren nuevas posibilidades en este ámbito. Estas arquitecturas, originalmente diseñadas para tareas de traducción automática, han mostrado un potencial prometedor para analizar series temporales clínicas. Al aprovechar su capacidad para capturar dependencias temporales de largo alcance y manejar datos heterogéneos, los modelos Seq2Seq podrían ofrecer predicciones más precisas y robustas. Este trabajo se centra en evaluar el uso de esta tecnología emergente en un contexto crítico como el de la progresión de la EP, contribuyendo al avance del estado del arte en la intersección de la salud y el aprendizaje automático.

1.2. Desafío en Kaggle

Un factor determinante que motivó el desarrollo de este proyecto fue el desafío de Kaggle «AMP Parkinson's Disease Progression Prediction» [1]. Este reto propone utilizar datos clínicos y herramientas de aprendizaje automático para predecir la progresión de la enfermedad en pacientes de forma precisa y reproducible.

1.2.1. La plataforma de Kaggle

Kaggle es una plataforma en línea especializada en ciencia de datos y aprendizaje automático, que funciona como un espacio colaborativo para la resolución de problemas mediante el uso de modelos predictivos y técnicas estadísticas. Fundada en 2010 y adquirida por Google en 2017, Kaggle alberga competiciones en las que empresas, instituciones o investigadores proponen retos basados en conjuntos de datos reales o sintéticos. La comunidad participante, compuesta por científicos de datos, ingenieros y estudiantes de todo el mundo, diseña modelos que compiten por mejorar una métrica de evaluación determinada.

Figura 1: Logo de Kaggle

Más allá de las competiciones, Kaggle ofrece recursos educativos, notebooks interactivos, conjuntos de datos públicos y foros activos que fomentan el aprendizaje continuo y el intercambio de conocimiento. Gracias a su entorno de ejecución basado en la nube, los participantes pueden entrenar modelos directamente en la plataforma sin necesidad de configuración local, lo que favorece la inclusión y la accesibilidad. Kaggle se ha consolidado así como una referencia clave en el ámbito del aprendizaje automático aplicado, especialmente en contextos de evaluación comparativa y desarrollo de prototipos.

1.2.2. *AMP Parkinson's Disease Progression Prediction*

El reto «AMP Parkinson's Disease Progression Prediction» fue organizado en Kaggle, durante 2023, por la iniciativa Accelerating Medicines Partnership® Parkinson's Disease (AMP®PD), una colaboración público-privada liderada por la Foundation for the National Institutes of Health (FNIH) de Estados Unidos, junto con diversas entidades de la industria biofarmacéutica y centros académicos.

El objetivo principal del reto era desarrollar modelos capaces de predecir la progresión de la enfermedad de Parkinson en base a datos longitudinales de pacientes. En concreto, se pedía estimar la evolución de las puntuaciones en la escala MDS-UPDRS (Movement Disorder Society-Sponsored Revision of the Unified Parkinson's Disease Rating Scale), una medida clínica estandarizada que evalúa la gravedad de los síntomas motores y no motores de la enfermedad.

Los datos proporcionados incluían medidas proteómicas y peptídicas obtenidas mediante espectrometría de masa en líquido cefalorraquídeo (CSF), junto con otra información clínica. Las observaciones estaban organizadas en varias visitas generalmente cada 6 meses lo que permitía la modelización de la progresión como un problema de serie temporal multivariable.

Para el desarrollo de este trabajo no se han seguido de forma estricta los requisitos particulares establecidos en la competición original, dado que esta ya había finalizado en el momento de realizar el proyecto. Sin embargo, sí se ha mantenido como referencia el objetivo general del reto.

1.2.3. Soluciones propuestas (Estado del arte)

Las soluciones con mejor evaluación combinaban enfoque del aprendizaje automático clásico, utilizando, práticamente de manera exclusiva, metadatos como mes de visita, horizonte de predicción, frecuencia de visitas y frecuencia de análisis clínicos.

Estos metadatos, que reflejan la evaluación y el seguimiento realizados por profesionales médicos, mejoraron la precisión del modelo. Sin embargo, resultan insatisfactorio desde el punto de vista clínico, ya que no modelan la progresión biológica de la enfermedad, sino aspectos relacionados con la atención sanitaria y la frecuencia de controles.

Los organizadores reconocieron esta limitación y promovieron una segunda fase del estudio centrada en datos más objetivos, como perfiles proteómicos y peptídicos, buscando establecer relaciones más directas con la fisiopatología del Parkinson.

Cabe destacar que ninguna de las soluciones ganadoras aplicó modelos Seq2Seq ni arquitecturas específicas para series temporales complejas, centrándose en modelos tabulares e híbridos. Esto deja un espacio claro para explorar enfoques que aprovechen mejor la naturaleza longitudinal de los datos clínico-proteómicos.

1.3. Objetivos

El objetivo general de este trabajo es evaluar el potencial de los modelos Seq2Seq en la predicción del avance de la enfermedad de Parkinson a partir de datos clínicos secuenciales. La hipótesis de partida es que estas arquitecturas, debido a su capacidad para modelar dependencias temporales complejas, podrían superar a los enfoques clásicos tanto en precisión como en adaptabilidad.

Este objetivo se alinea con la necesidad creciente de desarrollar sistemas predictivos más robustos, que sean útiles no sólo desde un punto de vista técnico, sino también clínico. La progresión del Parkinson es altamente variable entre pacientes, y un modelo capaz de anticipar esta evolución de forma individualizada podría tener un impacto significativo en la toma de decisiones médicas, el diseño de tratamientos personalizados y la mejora general del seguimiento del paciente.

Para lograr este objetivo general, se plantean los siguientes objetivos específicos:

- Comprender y analizar en profundidad el conjunto de datos proporcionado en el reto de Kaggle, identificando las variables clínicas más relevantes para el pronóstico.
- Diseñar un flujo de preprocesamiento que permita transformar los datos en secuencias temporales adecuadas, aplicando técnicas como normalización, imputación de valores faltantes, y estructuración en ventanas temporales (necesario para el modelo elegido).
- Implementar y entrenar diferentes configuraciones de modelos Seq2Seq, incluyendo variantes con mecanismos de atención (*attention*), distintas capas recurrentes (LSTM, GRU) y otros hiperparámetros.

- Comparar el rendimiento de los modelos Seq2Seq con una red neuronal de referencia más simple, empleando los mismos datos de entrada y misma evaluación, para analizar mejoras reales atribuibles al diseño Seq2Seq.
- Evaluar los resultados utilizando la métrica SMAPE (Symmetric Mean Absolute Percentage Error), así como el valor de esta función de pérdida sobre un conjunto de datos, de validación, separado para este fin. (*validation loss*).
- Documentar los hallazgos obtenidos y elaborar recomendaciones sobre futuras líneas de trabajo, incluyendo la integración de más datos (particularmente datos demográficos), el uso de técnicas de *transfer learning*, o la mejora de la transparencia del modelo.

1.4. Herramientas utilizadas

Para el desarrollo del presente trabajo se ha utilizado un conjunto de herramientas tecnológicas que permiten tanto el procesamiento de datos como el entrenamiento y evaluación de modelos de aprendizaje automático. A continuación se describen las principales herramientas y entornos empleados, agrupados por su funcionalidad principal. La selección de estas herramientas está respaldada por estudios recientes sobre el stack de tecnología estándar en ciencia de datos y Machine Learning, como S. Raschka, J. Patterson, y C. Nolet [2], que identifica Python y sus principales bibliotecas como el ecosistema dominante, tanto en investigación como en producción.

1.4.1. Lenguaje: Python

Python ha sido el lenguaje de programación principal del proyecto, elegido por su versatilidad, legibilidad y amplia adopción en la comunidad científica. Permite implementar tanto el flujo de preprocesamiento como los modelos de aprendizaje profundo. Dentro del ecosistema de Python se han utilizado las siguientes bibliotecas:

- **NumPy y Pandas**: Bibliotecas fundamentales para la manipulación y análisis de datos estructurados. Se han utilizado intensamente para explorar los datos, gestionar valores faltantes, transformar variables y construir secuencias temporales.
- **PyTorch**: Framework de desarrollo de redes neuronales utilizado para la implementación de los modelos Seq2Seq y la red neuronal de referencia. Se ha preferido frente a otras alternativas como TensorFlow por su flexibilidad en la construcción de arquitecturas personalizadas y su integración con herramientas de depuración y visualización. Pytorch permite además la aceleración por GPU, a pesar de que en este caso en concreto dada la cantidad limitada de datos no ha sido relevante, de cara a escalar esta solución reduciría los tiempos de entrenamiento incluso con grandes volúmenes.
- **scikit-learn**: Biblioteca de referencia para tareas de aprendizaje automático tradicional y evaluación de modelos. Se ha utilizado para tareas complementarias como la división del conjunto de datos en entrenamiento y validación, así como para cálculos estadísticos de referencia.
- **Matplotlib y Seaborn**: Herramientas de visualización utilizadas para representar gráficamente tanto las características de los datos como los resultados obtenidos en el proceso de modelado.

1.4.2. Entorno: Jupyter Notebooks

Jupyter Notebooks ha sido el entorno de desarrollo interactivo empleado para el análisis exploratorio, diseño y prueba de modelos. Su formato permite una documentación clara y estructurada del proceso de trabajo, facilitando tanto la reproducibilidad como la presentación de resultados. Esta herramienta ha sido recomendada como estándar en entornos de investigación por su integración con Python y librerías científicas.

1.4.3. Composición: Typst

Typst ha sido el sistema de composición tipográfica empleado para la redacción de la memoria. Su sintaxis clara y capacidad para integrar fácilmente referencias bibliográficas, ecuaciones y formato técnico lo convierten en una alternativa moderna y eficaz frente a otras herramientas como LaTeX. Destaca especialmente por su velocidad de compilación, lo que permite iterar de forma mucho más ágil durante la redacción y maquetación del documento, una ventaja clave en procesos iterativos como la elaboración de documentos técnicos complejos.

Estas herramientas han sido seleccionadas no sólo por su idoneidad técnica, sino también por su compatibilidad entre sí y por contar con una amplia comunidad de soporte, lo que ha facilitado su integración a lo largo del desarrollo del proyecto.

Capítulo 2. Planificación

La ejecución exitosa de un proyecto de investigación en Machine Learning requiere una planificación cuidadosa que abarque tanto los aspectos metodológicos como los recursos necesarios. Este capítulo detalla la estrategia seguida para el desarrollo del presente trabajo, desde la metodología de trabajo adoptada hasta la gestión temporal y económica del proyecto. La planificación inicial se fundamenta en las mejores prácticas establecidas en proyectos de ciencia de datos, siguiendo un enfoque iterativo que permite adaptarse a los hallazgos y desafíos que surgen durante el proceso de investigación. Asimismo, se incluye un análisis retrospectivo que contrasta la planificación inicial con la realidad del desarrollo, identificando las desviaciones más significativas y las lecciones aprendidas que pueden ser de utilidad para futuros proyectos similares en el ámbito del aprendizaje automático aplicado a la medicina.

2.1. Metodología de trabajo

Para el desarrollo de este proyecto se ha adoptado la metodología **CRISP-DM** (*Cross-Industry Standard Process for Data Mining*), considerada el estándar de facto para proyectos de minería de datos [3], y que sigue encontrando utilidad en el paradigma actual enfocado a ML [4].

Esta metodología, desarrollada específicamente para proyectos de ciencia de datos, resulta especialmente adecuada para el contexto de este trabajo por su naturaleza iterativa y su capacidad para adaptarse a los desafíos propios del análisis de datos clínicos y la construcción de modelos predictivos.

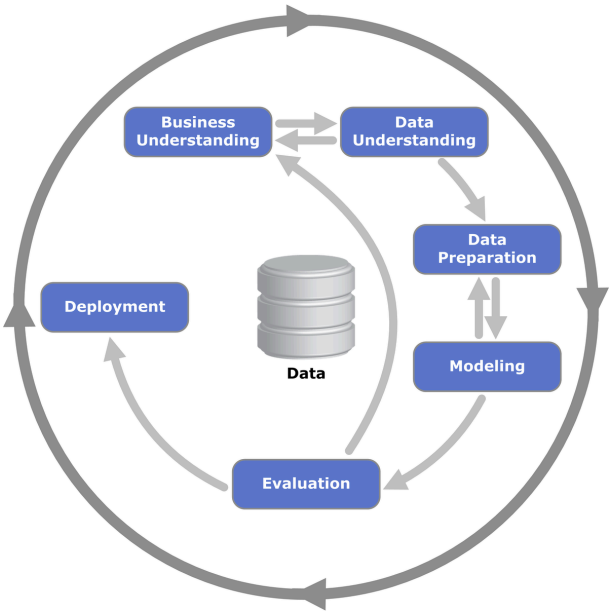

Figura 2: Diagrama de la relación entre las distintas fases en CRISP-DM. Se observa que es una metodología fluida e iterativa, que permite revisitar fases previas ante nueva información. Se observa además el papel central del dato. [5]

CRISP-DM estructura el proceso en seis fases interconectadas que permiten un desarrollo sistemático y reproducible:

1. ***Business Understanding*** (Comprensión del negocio):

Esta fase inicial tiene como objetivo comprender los objetivos y requerimientos del proyecto desde una perspectiva de negocio, para luego convertir este conocimiento en una definición del problema de minería de datos y un plan preliminar diseñado para lograr los objetivos.

En el contexto médico, esto implica entender las necesidades clínicas reales y su impacto en la atención al paciente. En este proyecto específico, se definió el problema médico a resolver estableciendo los objetivos clínicos de la predicción del avance del Parkinson y su traducción a objetivos de Machine Learning medibles. Se analizó el contexto del desafío de Kaggle, identificando las limitaciones de las soluciones previas basadas en metadatos clínicos, y se establecieron los criterios de éxito del proyecto:

- Desde una perspectiva técnica (métricas de evaluación). El modelo implementado deberá superar a los enfoques empleados en soluciones previas, para ello se medirá su desempeño frente a un modelo de referencia (o *baseline*).
- Desde una perspetiva clínica (relevancia de las predicciones). El modelo deberá utilizar los datos de proteinas y péptidos para mejorar la relevancia con respecto a las soluciones previas.

2. ***Data Understanding*** (Comprensión de los datos)

   En esta fase se recolecta el conjunto de datos inicial y se procede a familiarizarse con los datos, identificar problemas de calidad, descubrir primeras percepciones sobre los datos o detectar subconjuntos interesantes para formar hipótesis de información oculta. La comprensión profunda de los datos es fundamental para el éxito de cualquier proyecto de minería de datos, ya que determina tanto las técnicas aplicables como las limitaciones del análisis posterior. En este trabajo se realizó una exploración inicial exhaustiva del conjunto de datos proporcionado, incluyendo un análisis estadístico descriptivo, el análisis de la distribución de variables y la detección de valores faltantes.

3. ***Data Preparation*** (Preparación de los datos):

   Esta fase cubre todas las actividades necesarias para construir el conjunto de datos final que será alimentado a las herramientas de modelado. Las tareas de preparación de datos son propensas a ser realizadas múltiples veces y no en un orden prescrito, incluyendo la selección de tablas, registros y atributos, así como la transformación y limpieza de datos para las herramientas de modelado. Frecuentemente, esta es la fase más intensiva en tiempo del proyecto de minería de datos. En este proyecto, esta fase abarcó la limpieza y transformación de los datos clínicos y proteómicos, implementando técnicas de imputación de valores faltantes adaptadas a la naturaleza temporal de los datos, normalización de variables para garantizar la convergencia de los modelos de deep learning, construcción de secuencias temporales estructuradas para alimentar los modelos Seq2Seq, y división estratificada del conjunto de datos en entrenamiento y validación, manteniendo la coherencia temporal y la representatividad de los diferentes perfiles de progresión de la enfermedad.

4. ***Modeling*** (Modelado):

En esta fase se seleccionan y aplican varias técnicas de modelado, calibrando sus parámetros a valores óptimos. Típicamente, hay varias técnicas para el mismo tipo de problema de minería de datos, y algunas técnicas tienen requerimientos específicos sobre la forma de los datos. Por lo tanto, es común regresar a la fase de preparación de datos durante esta etapa. La experimentación iterativa es clave en esta fas , ya que diferentes algoritmos pueden revelar distintos aspectos de los datos. En este trabajo se implementaron y entrenaron los modelos Seq2Seq propuestos, incluyendo variantes con diferentes tipos de capas recurrentes (LSTM, GRU) y mecanismos de atención, junto con la red neuronal de referencia para establecer una línea base de comparación. Se experimentó con diferentes arquitecturas, hiperparámetros (learning rate, batch size, número de capas) y técnicas de regularización, documentando el proceso de entrenamiento y los resultados obtenidos en cada iteración.

5. ***Evaluation*** (Evaluación):

   En esta etapa del proyecto se ha construido un modelo que parece tener alta calidad desde una perspectiva de análisis de datos. Sin embargo, es importante evaluar a fondo el modelo y revisar los pasos ejecutados para construirlo, para asegurar que el modelo logre apropiadamente los objetivos del negocio. La evaluación debe considerar tanto métricas técnicas como criterios de aplicabilidad práctica. En este proyecto se evaluó el rendimiento de los modelos utilizando la métrica SMAPE (Symmetric Mean Absolute Percentage Error) como medida principal, complementada con el análisis del *validation loss* durante el entrenamiento. Se realizaron comparaciones sistemáticas entre los modelos Seq2Seq y la red neuronal de referencia, analizando no solo la precisión predictiva sino también la estabilidad del entrenamiento, la capacidad de generalización y la robustez frente a diferentes configuraciones de datos de entrada.

6. ***Deployment*** (Despliegue):

   La creación del modelo no es generalmente el final del proyecto. Aún si el propósito del modelo es incrementar el conocimiento de los datos, el conocimiento ganado necesita ser organizado y presentado de una manera que el cliente pueda usar. Esta fase se centra en la operacionalización del modelo y la transferencia de conocimiento.

   Aunque este proyecto tiene un enfoque académico y no implica un despliegue productivo, esta fase se ha interpretado como la documentación sistemática y comunicación de los resultados obtenidos, incluyendo la elaboración de esta memoria técnica, la presentación de hallazgos clave sobre el potencial de los modelos Seq2Seq en el contexto clínico, y la formulación de recomendaciones específicas para futuros trabajos para facilitar su eventual adopción en entornos clínicos.

2.1.1. Comparación con otras metodologías

Para justificar la elección de CRISP-DM en este proyecto, es esencial analizar las características y limitaciones de otras metodologías ampliamente utilizadas en el desarrollo de proyectos tecnológicos. A continuación se presenta una comparación detallada con las metodologías Waterfall y Agile, examinando su aplicabilidad específica en el contexto de proyectos de Machine Learning y ciencia de datos. Metodología Waterfall (Cascada)

La metodología Waterfall representa el enfoque más tradicional y estructurado para la gestión de proyectos de software. Desarrollada originalmente por Winston Royce en 1970, esta metodología sigue un proceso secuencial y lineal donde cada fase debe completarse antes de proceder a la siguiente, sin posibilidad de retroceso. Su característica fundamental es la necesidad de documentación exhaustiva en cada fase, junto con una planificación detallada que define completamente los requisitos y el alcance al inicio del proyecto.

En el contexto de este proyecto de predicción del avance del Parkinson, el modelo Waterfall presenta limitaciones significativas que lo hacen inadecuado para proyectos de Machine Learning. La naturaleza exploratoria de los datos requiere una exploración iterativa para comprender su estructura, calidad y potencial predictivo, algo que el modelo Waterfall no permite ya que exige definir completamente los requisitos antes de proceder. Además, la incertidumbre inherente en los proyectos de ML significa que los resultados de una fase pueden invalidar completamente las asunciones de fases anteriores. Por ejemplo, el análisis exploratorio puede revelar que los datos disponibles no son suficientes para el objetivo planteado, requiriendo redefinir el problema inicial.

El desarrollo de modelos ML implica experimentación constante con diferentes algoritmos, hiperparámetros y arquitecturas, una necesidad de iteración y refinamiento continuo que el modelo Waterfall no contempla. A diferencia del desarrollo de software tradicional, donde los requisitos son relativamente estables, en ML la calidad y disponibilidad de los datos puede cambiar dramáticamente la viabilidad del proyecto, haciendo que el enfoque rígido de Waterfall sea contraproducente.

2.1.1.1. Metodología Agile

La metodología Agile surgió como respuesta a las limitaciones del modelo Waterfall, promoviendo un enfoque iterativo e incremental para el desarrollo de software. Formalizada en el Manifiesto Agile de 2001, esta metodología prioriza la adaptabilidad, la colaboración y la entrega temprana de valor. Su filosofía se centra en el desarrollo iterativo mediante sprints cortos, la flexibilidad ante cambios en requisitos, la colaboración continua entre stakeholders y la entrega temprana de valor al usuario.

Agile ofrece ciertas ventajas para proyectos de ML que lo hacen superior a Waterfall, particularmente en su capacidad de adaptación basándose en los hallazgos obtenidos durante el análisis de datos. Su enfoque en la iteración rápida facilita la experimentación con diferentes modelos y enfoques, mientras que el feedback continuo permite validar hipótesis y ajustar el rumbo del proyecto frecuentemente.

Sin embargo, Agile también presenta limitaciones importantes cuando se aplica directamente a proyectos de ML. La metodología no contempla explícitamente las fases críticas de comprensión y preparación de datos, que pueden consumir gran parte del tiempo en un proyecto de ML. Por otra parte, el concepto de «entregable funcional» se vuelve ambiguo en ML, donde un modelo puede funcionar técnicamente pero no tener valor predictivo real. Las métricas tradicionales de Agile, como velocidad y burndown charts, no reflejan adecuadamente el progreso en proyectos de investigación y experimentación.

Aunque Agile maneja bien los cambios en requisitos, no está diseñado para la incertidumbre fundamental sobre la viabilidad técnica que caracteriza a los proyectos de ML.

En conclusión, para este proyecto específico de predicción del avance del Parkinson, CRISP-DM resulta la metodología más apropiada por incluir fases dedicadas específicamente a la comprensión y preparación de datos, aspectos críticos dada la complejidad de los datos clínicos y proteómicos; y por permitir iteraciones entre fases pero de manera estructurada, evitando el riesgo de «deriva» en la experimentación que puede ocurrir con enfoques menos estructurados.

2.2. Planificación temporal

La planificación temporal de este proyecto se ha estructurado considerando que el TFG tiene una asignación de 12 créditos ECTS, lo que equivale a 300 horas de trabajo según el Sistema Europeo de Transferencia y Acumulación de Créditos (25 horas por crédito ECTS). Esta carga de trabajo se ha distribuido siguiendo las seis fases de la metodología CRISP-DM, adaptándose a la disponibilidad variable durante el período de trabajo, de la siguiente manera:

- ***Fase 1: Business Understanding*** (Comprensión del negocio) - **30 horas (10%)**
  - Análisis del contexto clínico del Parkinson: 10 horas
  - Revisión bibliográfica sobre ML aplicado a enfermedades neurodegenerativas: 12 horas
  - Estudio del desafío de Kaggle y análisis de soluciones previas: 8 horas
- ***Fase 2: Data Understanding*** (Comprensión de los datos) - **45 horas (15%)**
  - Exploración inicial del conjunto de datos: 20 horas
  - Análisis estadístico descriptivo: 15 horas
  - Evaluación de calidad de datos y valores faltantes: 10 horas
- ***Fase 3: Data Preparation*** (Preparación de los datos) - **60 horas (20%)**
  - Limpieza y transformación de datos: 15 horas
  - Implementación de técnicas de imputación: 20 horas
  - Construcción de secuencias temporales para modelos Seq2Seq: 25 horas
- ***Fase 4: Modeling*** (Modelado) - **90 horas (30%)**
  - Implementación de modelo baseline: 20 horas
  - Desarrollo de arquitecturas Seq2Seq (LSTM, GRU): 35 horas
  - Experimentación con hiperparámetros y regularización: 25 horas
  - Entrenamiento y ajuste de modelos: 10 horas
- ***Fase 5: Evaluation*** (Evaluación) - **30 horas (10%)**
  - Evaluación sistemática con métricas SMAPE: 15 horas
  - Análisis comparativo y validación de resultados: 15 horas
- ***Fase 6: Deployment*** (Documentación y comunicación) - **30 horas (10%)**
  - Redacción de la memoria técnica: 20 horas
  - Preparación de presentación y revisión final: 10 horas

Por otra parte se plantea una ***reserva para contigencias*** de **15 horas (5%)**, reservadas para imprevistos y revisiones adicionales. Tomando esta distribución como punto de partida, se tiene siempre en cuenta la naturaleza emergente de los problemas en el dominio de la ciencia de datos reflejada por CRISP-DM, manteniendo la posibilidad de redistribuir horas entre fases adyacentes en caso de necesidad.

Fase	Horas	%
Fase 1	30h	10%
Fase 2	45h	15%
Fase 3	60h	20%
Fase 4	90h	30%
Fase 5	30h	10%
Fase 6	30h	10%
Reserva para contingencias	15h	10%

Tabla 1: Resumen de la distribución del tiempo total estimado para cada fase

Para el control del progreso se lleva un registro de horas trabajadas en cada tarea identificada para mantener el control del presupuesto temporal.

2.2.1. Factores de riesgo:

Se han identificado ciertos factores de riesgo que pueden afectar al éxito de la planificación temporal en este proyecto.

- La preparación de datos puede requerir más tiempo del estimado debido a la complejidad de datos clínicos
- Los tiempos de entrenamiento de modelos pueden variar según la convergencia
- La documentación técnica puede extenderse si se requiere mayor profundidad en el análisis

2.3. Presupuesto económico

El desarrollo de este proyecto de investigación en Machine Learning requiere una evaluación económica que considere tanto los recursos humanos como los recursos de infraestructura necesarios para su ejecución.

2.3.1. Recursos humanos

El coste principal del proyecto corresponde al tiempo dedicado por el investigador principal. La valoración se realiza considerando datos salariales actuales del sector tecnológico.

El rol que más se ajusta al perfil necesario para el desarrollo de este proyecto es el de un científico de datos, particularmente calcularemos los costes salariales tomando en consideración un puesto de cientifico de datos sin experiencia (Junior). Según los datos publicados en *Glassdoor* [6] recientemente los salarios brutos oscilan entre los 22.000€ y 30.000€. Tomando la estimación más alta aproximamos unos 15€ por hora trabajada:

Horas totales	300h (12 créditos ECTS × 25 horas/crédito)
Tarifa horaria	15€/h (Científico de datos Junior)
Coste Total	**4.500€**

Tabla 2: Prespuesto económico dedicado a recursos humanos.

Se considera que todas las tareas a realizar pueden englobarse en este mismo rol, con lo que supondría el total de las horas estimadas para la finalización del proyecto.

2.3.2. Recursos de infraestructura y de software

El presupuesto dedicado a infraestructura y software en este proyecto es pequeño en comparación con el presupuesto dedicado a recursos humanos pero es importante detallarlo ya que en proyectos del mismo ámbito el coste de infraestructura (normalmente en la nube) y las licencias de software anuales son muy significativas.

En el contexto de este proyecto no se contemplan gastos debidos a licencias de software ya que la totalidad de las herramientas elegidas para el desarrollo son de código abierto.

Por parte del hardware (infraestructura), el desarrollo se ha llevado a cabo en mi portatil personal. El coste del portatil fue de 800€. Tipicamente se considera amortizado un equipo informático en 4 años (25% anual). Esta amortización considera un uso continuado del equipo, en mi caso al tratarse de mi equipo personal resulta más complicado disociar el uso dedicado al proyecto del uso por otros asuntos propios, por ese mótivo en vez de dar la estimación del coste amortizado usando años (o meses) se utilizarán las 300h planificadas. Para ello se tiene en cuenta el dato de que, a jornada completa, se trabajan 1.826h anuales.

Coste	800€
Horas de uso	300h
Horas amortización	7304h
Porcentaje imputable	4.1%
Coste imputable	**32,8€**

Tabla 3: Prespuesto económico dedicado a recursos humanos.

Otros gastos como los de internet y electricidad son especialmente dificiles de separar del consumo personal y no entrarán en este presupuesto.

2.3.3. Otros gastos

No se han identificado otras posibles fuentes de gastos ya que se optará por opciones disponibles abiertamente, sin embargo se detallan aquí puntos que en otros proyectos similiares o en caso de extender el alcance de este, podrían incurrir en gastos adicionales.

- Acceso a literatura científica: 0 € (acceso a través de la universidad)
- Acceso a los datos: 0€ (Disponibles abiertamente en el desafío de Kaggle)

Categoria	Coste	%
Recursos humanos	4.500€	99.3%
Infraestructura	32.8€	0.7%
Software	0€	0%
Otros	0€	0%
Total	4.532,8€	100%

Tabla 4: Resumen del presupuesto

2.4. Ajuste a la realidad

Una vez finalizado el desarrollo del proyecto, es fundamental analizar el grado de ajuste entre la planificación inicial y la ejecución real, tanto en términos de tiempo como de recursos económicos.

2.4.1. Ajuste temporal

La planificación inicial contemplaba una dedicación total de **300 horas**. En la práctica, la dedicación real fue:

- Horas efectivamente dedicadas: **310 horas**

Este ligero desfase (3,3%) se explica principalmente por tareas no previstas inicialmente, como la documentación adicional e iteraciones validando y refinando el modelo. A pesar de superar el tiempo reservado para contingencias, el proyecto se ha completado con un desfase mínimo, por lo que el ajuste temporal puede considerarse satisfactorio.

2.4.2. Ajuste económico

En cuanto al presupuesto económico, las desviaciones fueron mínimas y debidas también al tiempo adicional.

Categoría	Presupuesto estimado	Coste real	Desviación
Recursos humanos	4.500€	4.650€	+150€
Infraestructura	32,8€	33,9€	0€
Software	0€	0€	0€
Otros	0€	0€	0€
Total	4.532,8€	4.683,9€	**+151.1€**

Tabla 5: Comparativa entre presupuesto estimado y real.

2.4.3. Valoración general

El proyecto se ha desarrollado de forma eficiente, con desviaciones mínimas en tiempo y por lo tanto en costes. El uso de herramientas **open source** ha sido clave para mantener el

presupuesto bajo control, y la amortización del equipo personal ha permitido limitar los costes de infraestructura.

Este ajuste a la realidad confirma la viabilidad económica del enfoque adoptado y sugiere que metodologías similares pueden aplicarse en otros contextos académicos o incluso profesionales con recursos limitados.

Capítulo 3. Análisis exploratorio de datos

En esta sección se documenta el análisis exploratorio inicial realizado sobre los datos provistos por el desafío de predicción de progresión del Parkinson. Primero, se explica la estructura y el contenido de los datos según las descripciones aportadas por la fuente y se continúa por observaciones obtenidas en el cuaderno de Jupyter incluido `eda.ipynb`.

3.1. Terminología y contexto clínico

Un conocimiento, aunque superficial, de la terminología presente en la descripción del reto y el conjunto de datos es importante ya que puede informar decisiones sobre el modelo a implementar. Es por esto que dedicamos una breve sección a conceptos que se mencionan y que requieren una explicación adicional.

3.1.1. UPDRS

La **Unified Parkinson's Disease Rating Scale** (UPDRS) es un instrumento clínico estandarizado que se utiliza para cuantificar la progresión de los síntomas del Parkinson. Está compuesta por una serie de ítems o preguntas que evalúan distintos aspectos de la enfermedad, agrupados en cuatro secciones. Cada ítem se valora mediante una escala ordinal, y la suma de estos valores proporciona una medida de severidad dentro de cada sección. Fue introducido por MDS (*Movement Disorder Society*) [7], por lo que en ocasiones también es referido por MDS-UPDRS.

Las puntuaciones posibles, así como los umbrales de severidad típicamente aceptados, se detallan a continuación:

- Parte I: Experiencias no motoras de la vida diaria (estado mental, humor, comportamiento).
- Parte II: Experiencias motoras de la vida diaria (actividades cotidianas como vestirse, alimentarse, etc.).
- Parte III: Examen motor realizado por personal médico (rigidez, temblores, reflejos posturales).
- Parte IV: Complicaciones motoras asociadas al tratamiento farmacológico (disquinesias, fluctuaciones).

Es importante destacar que cada parte tiene un rango de puntuaciones distinto ya que puede afectar al uso del dato en este proyecto. Las puntuaciones más altas en cualquier parte indican una mayor severidad de los síntomas. Esta escala permite capturar de forma sistemática la evolución de la enfermedad a lo largo del tiempo y evaluar la eficacia de los tratamientos aplicados.

Sección	Rango total	Leve	Severa
Parte I	0–52	0–10	≥22
Parte II	0–52	0–12	≥30
Parte III	0–132	0–32	≥59
Parte IV	0–24	0–4	≥13

Tabla 6: Clasificación de la severidad según las puntuaciones UPDRS.

En el presente proyecto, las puntuaciones UPDRS constituyen las variables objetivo que el modelo de aprendizaje automático debe predecir.

3.1.2. NPX

NPX (**Normalized Protein eXpression**) es una medida relativa de la abundancia de proteínas en una muestra obtenida por técnicas de espectrometría de masas. Es una escala logarítmica donde una diferencia de 1 NPX representa una duplicación en la abundancia relativa (escala logarítmica en base 2). Este tipo de normalización permite comparar muestras entre pacientes y visitas de manera robusta. No tiene unidades absolutas, lo que limita la interpretación clínica directa, pero es útil en modelos comparativos y predictivos.

3.1.3. UniProt

UniProt es una base de datos biológica que proporciona información sobre secuencias y funciones de proteínas En el conjunto de datos, cada proteína está identificada por un código único de UniProt (por ejemplo, `P12345`), que permite consultar fácilmente su función biológica, localización celular y asociaciones conocidas con enfermedades. Este identificador es clave para enriquecer el análisis con conocimiento biológico previo o realizar anotaciones externas.

3.1.4. Proteínas y péptidos

Las **proteínas** son macromoléculas formadas por cadenas largas de aminoácidos. Están involucradas en prácticamente todos los procesos biológicos. Las **péptidos** son fragmentos más cortos de proteínas; en muchos casos, una proteína puede contener múltiples péptidos distintos.

3.2. Descripción del conjunto de datos

El conjunto de datos proporcionado para el presente desafío tiene como objetivo predecir la progresión de la enfermedad de Parkinson (EP) a partir de datos de abundancia proteica. Esta predicción se realiza a partir de muestras de líquido cefalorraquídeo (LCR) analizadas mediante espectrometría de masas. Se trata de un conjunto de datos longitudinal que incluye varias visitas por paciente a lo largo de varios años, acompañadas de evaluaciones clínicas del estado de la enfermedad.

Los datos están organizados en cuatro archivos principales:

3.2.1. `train_peptides.csv`

Contiene mediciones de espectrometría de masas a nivel de péptido, es decir, subcomponentes de proteína . Cada fila representa la abundancia de un péptido específico en una muestra determinada.

Campo	Descripción
visit_id	Identificador único de la visita.
visit_month	Mes relativo desde la primera visita del paciente.
patient_id	Identificador del paciente.
UniProt	Código UniProt de la proteína asociada.
Peptide	Secuencia de aminoácidos del péptido.
PeptideAbundance	Frecuencia relativa del péptido en la muestra.

Tabla 7: Descripción de los campos de los datos de péptidos.

3.2.2. `train_proteins.csv`

Agrega la información del archivo anterior a nivel proteico. Es decir, contiene la abundancia de proteínas obtenida a partir de los péptidos componentes.

Campo	Descripción
visit_id	Identificador único de la visita.
visit_month	Mes relativo desde la primera visita del paciente.
patient_id	Identificador del paciente.
UniProt	Código UniProt de la proteína.
NPX	Abundancia normalizada de la proteína.

Tabla 8: Descripción de los campos de los datos de proteínas.

3.2.3. `train_clinical_data.csv`

Contiene las evaluaciones clínicas asociadas a cada visita en la que se recogió una muestra de LCR. Los campos incluyen información sobre la puntuación del paciente en diferentes partes de la escala UPDRS (**Unified Parkinson's Disease Rating Scale**).

Campo	Descripción
`visit_id`	Identificador único de la visita.
`visit_month`	Mes relativo desde la primera visita del paciente.
`patient_id`	Identificador del paciente.
`updrs_1` - `updrs_4`	Puntuaciones en las partes 1 a 4 de la escala UPDRS.
`upd23b_clinical_state_on_medication`	Indica si el paciente estaba bajo medicación (como Levodopa) durante la evaluación.

Tabla 9: Descripción de los campos de los datos de pacientes.

3.2.4. `supplemental_clinical_data.csv`

Este archivo contiene registros clínicos adicionales de pacientes que no tienen muestras asociadas de LCR. Se utiliza para proporcionar contexto adicional sobre la progresión típica de la enfermedad. Su estructura es exactamente igual que la del fichero `train_clinical_data.csv`.

3.2.5. Consideraciones generales

A partir de la descripción provista de los datos destacamos las siguientes consideraciones que han influido en el análisis.

- La naturaleza longitudinal del conjunto de datos y la estructura jerárquica (paciente,visita,medición) exigen un tratamiento cuidadoso del tiempo como variable.
- Existe una alta dimensionalidad en las mediciones proteicas, tanto a nivel de péptido como de proteína. Es decir para una única visita tenemos muchas proteínas y peptidos distintos que nos proporcionan un dato para esa visita.
- Es necesario tratar cuidadosamente en particular los datos de UPDRS y NPX ya que tienen una escalas particulares, asegurando que los valores están normalizados de manera interpretable por el modelo.

3.3. Observaciones realizadas

En este apartado se comentan las observaciones obtenidas en el cuaderno incluido `eda.ipynb`, comentando además el código utilizado.

3.3.1. Carga

```python
import os
import pandas as pd
import numpy as np
import seaborn as sns
data_path = "amp-parkinsons-disease-progression-prediction"
supplemental = pd.read_csv(os.path.join(data_path,
  "supplemental_clinical_data.csv"))
patient =pd.read_csv(os.path.join(data_path, "train_clinical_data.csv"))
peptides = pd.read_csv(os.path.join(data_path, "train_peptides.csv"))
proteins = pd.read_csv(os.path.join(data_path, "train_proteins.csv"))
```

Listado 1: Carga de los datos usando pandas.

La carga se ha realizado en *DataFrames* de `pandas`, lo tomaremos como punto de partida tanto para este análisis como para el modelo final.

3.3.2. Cardinalidad de los datos

```python
print(f"""Cardinalidad de los datos:
Hay {len(proteins.UniProt.unique())} proteínas únicas
Hay {len(peptides.Peptide.unique())} péptidos únicos
Hay {len(peptides[["Peptide", "UniProt"]].drop_duplicates())} pares de proteina-peptido únicos.
Hay {len(patient.patient_id.unique())} pacientes
Hay {len(patient)} visitas
Hay {len(supplemental.patient_id.unique())} pacientes (suplementario)
Hay {len(supplemental)} visitas (suplementario)
""")
```

Listado 2: Obtención de datos de cardinalidad

Se realiza un recuento de los datos disponibles a distintos niveles, el dato central para el modelo es el de «visita».

Métrica	Valor
Proteínas únicas	227
Péptidos únicos	968
Péptido-proteína únicos	968
Pacientes (principal)	248
Visitas (principal)	2615
Pacientes (sulementario)	771
Visitas (suplementario)	2223

Tabla 10: Cardinalidad del conjunto de datos.

Como podemos observar el modelo podrá disponer para su entrenamiento del orden de 2600 datos de entrada completos, con mediciones de proteínas o hasta 5800 con mediciones de UPDRS. Son números no muy grandes y esto representa uno de los principales riesgos del proyecto ya que la cantidad de los datos de entrado disponibles inlfuyen dramáticamente en la calidad y utilidad predictiva del modelo final. Por otra parte se puede observar que no hay peptidos en común para distintas proteínas.

3.3.3. Validación de las relaciones

```python
# Comprobamos que efectivamente es una clave primaria
print("¿Hay algún visit_id duplicado en clinical?")
print(patient["visit_id"].duplicated().any())
print("¿Hay algún visit_id duplicado en supplemental?")
print(supplemental["visit_id"].duplicated().any())

# Comprobaciones de relación entre tablas

# Comprobación de la relación 1 a 1
print("¿Están todos los visit_id de proteínas en 'clinical'?")
difference = set(proteins["visit_id"]).difference(set(patient["visit_id"]))
print(not difference)
if difference:
    print(len(difference))

print("¿Están todos los visit_id de 'clinical' en proteínas?")
difference = set(patient["visit_id"]).difference(set(proteins["visit_id"]))
print(not difference)
if difference:
    print(len(difference))

print(
    "¿Es cierto que los datos de 'supplemental' no tienen datos de proteínas asociados?"
)
intersection = set(supplemental["visit_id"]).intersection(proteins["visit_id"])
print(not intersection)
if intersection:
    print(len(intersection))

# Comprobación de la relación 1 a n
print("¿Estan todas las mediciones de peptidos asociadas a una medicion de proteínas?")
difference = set(peptides[["visit_id", "UniProt"]].apply(tuple, axis=1)).difference(
    proteins[["visit_id", "UniProt"]].apply(tuple, axis=1)
)
print(not difference)
if difference:
    print(len(difference))
```

Listado 3: Validaciones de las relaciones.

Es importante validar las suposiciones que hacemos sobre los datos, evitando sostener ninguna suposición implícita que pueda producir errores más adelante difíciles de depurar. Unas de estas suposiciones, que se forman de manera inmediata a partir de la descripción de

los datos, es la de las relaciones entre las tablas y qué campos forman claves primarias y claves foráneas.

- ¿Hay algún `visit_id` duplicado en clinical? **Correcto**

  No lo hay, `visit_id`, formado por `patient_id` y `visit_month` es una clave primaria.

- ¿Hay algún `visit_id` duplicado en supplemental? **Correcto**

  No lo hay. Lo mismo aplica para el conjunto de datos suplementario.

- ¿Están todos los visit_id de proteínas en “clinical”? **Incorrecto**

  Existen 45 visitas distintas cuyas mediciones en la tabla de proteínas no se corresponden con ninguna visita que figure en la tabla principal. Estos datos tendrán que ser descartados para el modelo final.

- ¿Están todos los visit_id de “clinical” en proteínas? **Incorrecto**

  Existen 1547, es decir cerca de la mitad de los datos no dispone de mediciones de proteínas. Se deberá decir como tratar tantos registros con valores faltantes, ya que eliminarlos por completo reduciría mucho el conjunto de datos disponible.

- ¿Los datos de “supplemental” no tienen datos de proteínas asociados? **Correcto**

- ¿Estan todas las mediciones de peptidos asociadas a una medicion de proteínas? **Correcto**

3.3.4. Análisis de la distribución de los datos de proteínas

python

```python
sns.set_theme(rc={'figure.figsize':(11.7,8.27)})
proteins["logNPX"] = np.log2(proteins["NPX"])
unique_proteins = proteins["UniProt"].unique()
protein = unique_proteins[1]
some_proteins = proteins.loc[proteins["UniProt"] == protein]
sns_plot = sns.violinplot(some_proteins, x="logNPX", y="UniProt")
sns_plot.get_figure().savefig("log_npx_violin_plot.png")
plt.close()
sns_plot = sns.violinplot(some_proteins, x="NPX", y="UniProt")
sns_plot.get_figure().savefig("npx_violin_plot.png")
```

Listado 4: Creación de gráficos de violin para los valores NPX usando *seaborn*.

Para examinar las distribuciones de las distintas proteínas se utilizan *violin plots*, un *violin plot* es una representación gráfica que combina un diagrama de caja (boxplot) con una estimación de densidad de núcleo (KDE), con el objetivo de visualizar simultáneamente estadísticas resumidas y la distribución completa de una variable continua. En su forma, el gráfico se asemeja a un violín: la anchura de cada sección representa la densidad de probabilidad estimada en ese rango de valores, permitiendo identificar características como asimetrías, modas múltiples o colas largas en la distribución. Superpuesto a esta densidad, el gráfico incluye elementos tradicionales del boxplot como la mediana, marcada dentro de la caja; los cuartiles, que determinan el tamaño de la caja; y los bigotes, la linea que se extiende desde la caja marcando los límites a partir de los cuales un valor pasa a considerarse extremo.

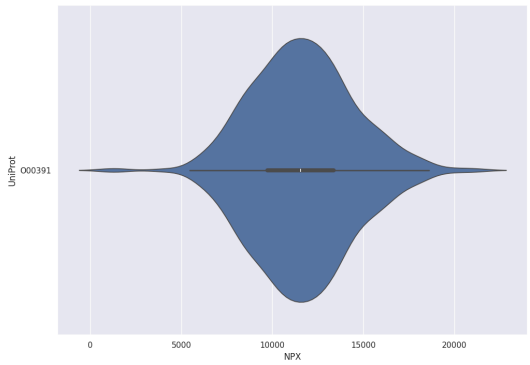

Figura 3: Diagrama de violín de la distribución de la medida NPX para una proteina concreta

En primer lugar podemos observar la escala (en el orden de 10^4 en este caso pero con gran variación entre proteínas), esto nos indica que a pesar de que NPX normalmente se utiliza en escala logarítmica (normalmente entre 10 y 20), en este caso tenemos los valores «linearizados».

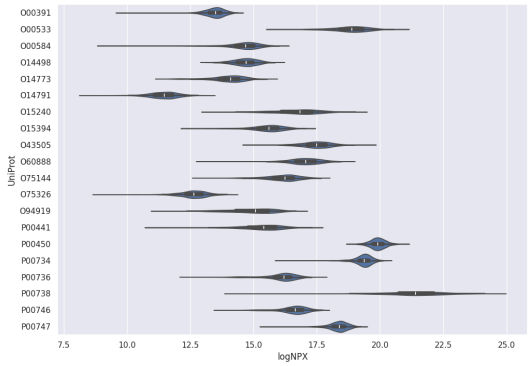

Figura 4: Diagrama de violín utilizando

$$\log_2 \text{NPX}$$

Observando la distribución de esta y otras proteínas vemos que se tratan en general de distribuciones unimodales y simétricas: similares a la distribución normal.

3.3.5. Análisis de la distribución de los datos UPDRS

python

```python
updrs_cols = [f"updrs_{i}" for i in range(1,5)]
updrs = pd.concat([patient[updrs_cols], supplemental[updrs_cols]])
sns_plot = sns.violinplot(updrs)
sns_plot.get_figure().savefig("updrs_violins.png")
updrs_ranges = [52, 52, 132, 24]

for i in range(1,5):
    updrs[f"updrs_{i}"] = updrs[f"updrs_{i}"] / updrs_ranges[i-1]
sns_plot = sns.violinplot(updrs)
sns_plot.get_figure().savefig("norm_updrs_violins.png")
```

Listado 5: Creación de gráficos de violin para los datos de UPDRS usando *seaborn*.

Se observa en general distribuciones con peso cerca del 0, siendo los valores mayores progresivamente más raros. Esto es especialmente notable en la categoríá IV, donde la mayoría de los valores son 0 o cercanos al 0. Cabe señalar también la bimodalidad de la distribución del los datos de la categoría II .

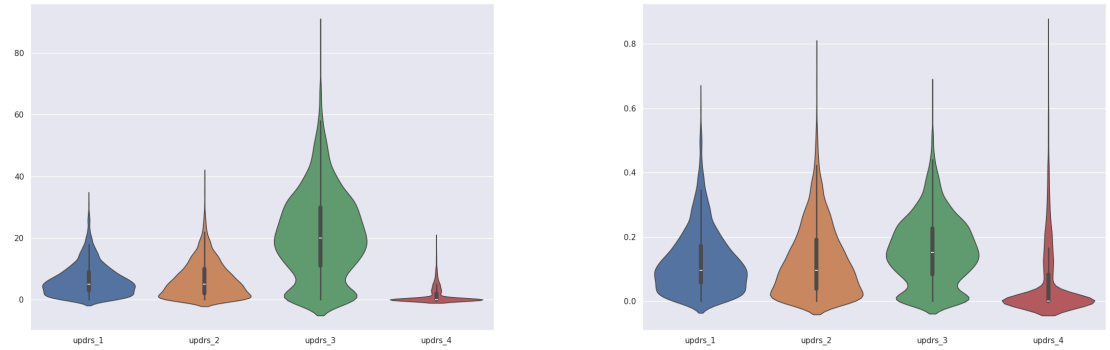

Figura 5: Distribución de los valores UPDRS, incluyendo datos del dataset «clínico» y «suplementario». A la izquierda distribución de los datos sin normalizar, a la derecha distribución de los datos normalizados según los rangos especificados por categoría (Tabla 6).

python

```
for col in updrs_cols:
    print(f"{col}: {len(updrs[updrs[col].isna()])} / {len(updrs)}
({len(updrs[updrs[col].isna()]) * 100/ len(updrs):.2f}%)")
```

Listado 6: Obtención de información sobre los valores nulos.

Se a realizado también un estudio de los valores faltantes en los datos de UPDRS, destacando un altísimo porcentaje de valores faltantes en la categoría IV. En vista de este análisis se juzga sensato imputar a estos casos el valor 0 para su uso en el modelo.

Categoría	Número de valores nulos	Número de valores nulos
I	214	4.42%
II	216	4.46%
III	30	0.62%
IV	1966	40.64%

Tabla 11: Número de valores faltantes para los datos de UPDRS

3.3.6. Distribución de mes de visita

python

```
visit_months = pd.concat([patient["visit_month"],
supplemental["visit_month"]])
sns_plot = sns.histplot(visit_months, binwidth=1)
sns_plot.set_xticks(visit_months.unique())
sns_plot.get_figure().savefig("visit_month_hist.png")
```

Listado 7: Creación de histograma de los meses de visita.

En el siguiente histograma se aprecia la distribución de los meses de visita. Podemos realizar varias observaciones: en primer lugar la primera visita (mes 0) es la más frecuente, de la que más datos disponemos. Esto puede haber dado lugar a que los valores bajos de UPDRS sean los más frecuentes, entendiendo que empeoran con el tiempo. Por otra parte, la frecuencia de visita es tipicamente de 6 meses. Sin embargo, tenemos valores para meses 3, 5 y 9, y las visitas más tardías parecen tener frecuencia anual. Esto tendrá consecuencias en el tratamiento de datos para su uso en un modelo ya que habrá que decidir entre aceptar una frecuencia variable o ajustar o descartar los valores que no encajen en la frecuencia semestral.

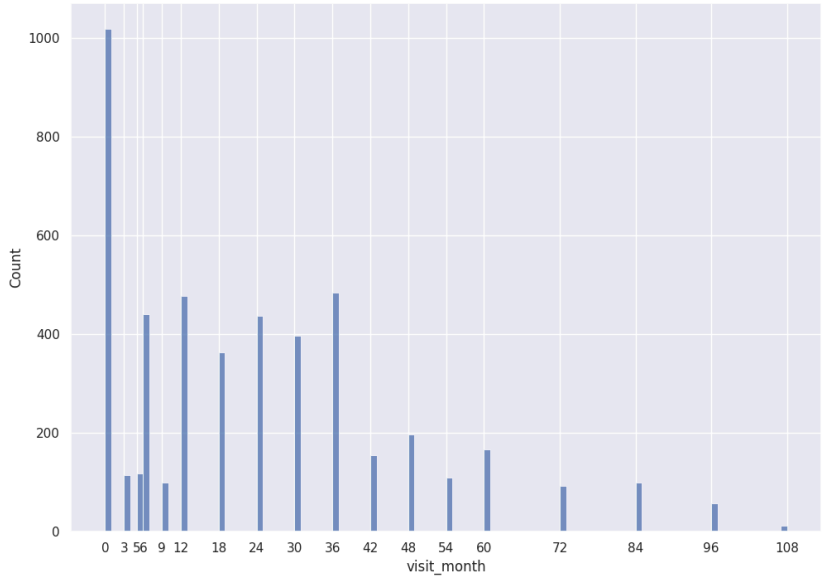

Figura 6: Histograma de los meses de visita.

3.3.7. Evolución temporal

python

```python
for col in updrs_cols:
    sns_plot = sns.lineplot(
        pd.concat([patient, supplemental], axis=0, ignore_index=True).fillna(0),
        x="visit_month",
        y=col,
        estimator="mean",
        errorbar=("ci", 95),
    )
    sns_plot.get_figure().savefig(f"evolution_{col}.png")
    plt.close()
```

Listado 8: Creación de gráficos de evolución temporal

Para estudiar la evolución en las distintas categorías de UPDRS a lo largo del tiempo se ha graficado la media por mes de visita incluyendo un área de color alrededor que representa el intervalo de confianza para la media con confianza 0.95. De esta manera el área se ensancha si hay mayor variación o menos datos.

Se puede observar en todo caso una tendencia a aumentar en el tiempo, aunque el incremento no es dramático. En la categoría IV se observa un incremento pero en todo caso esta categoría se mantiene con valores muy bajos (recordamos que es la correspondiente a los efectos adversos de la medicación).

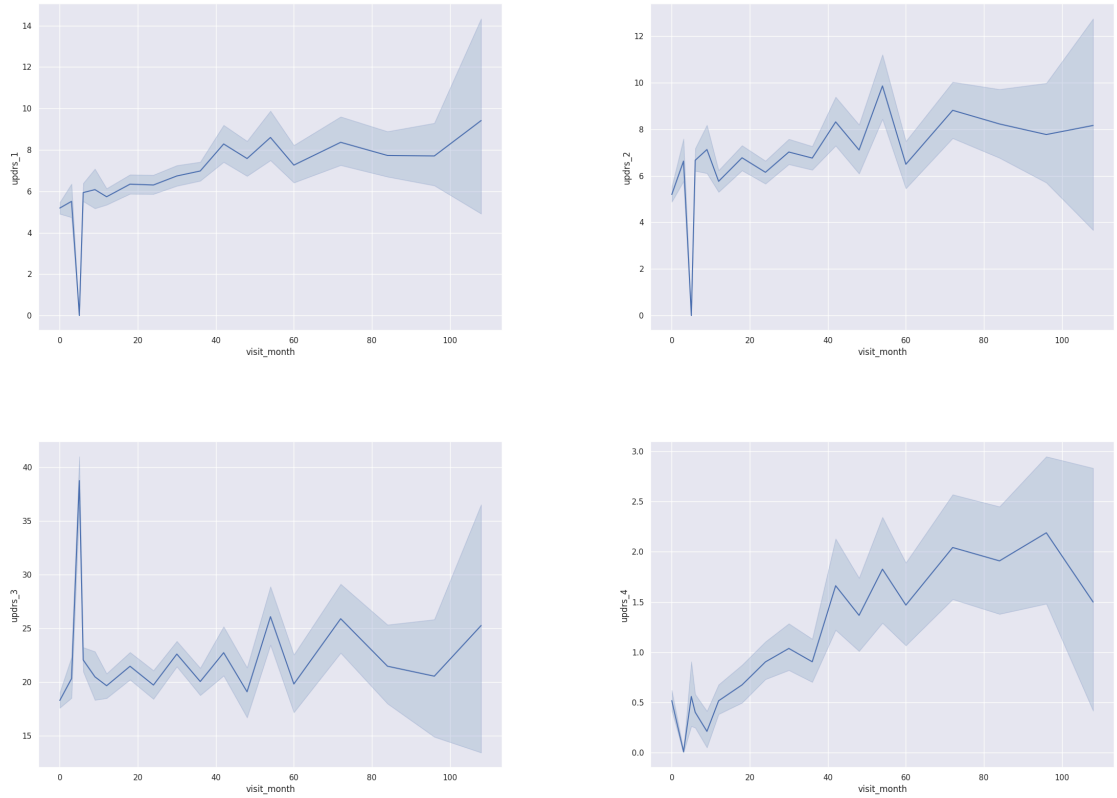

Figura 7: Evolución media a lo largo del tiempo.

En la próxima sección validaremos esta observación encontrando una correlación positiva entre los valores UPDRS y el tiempo.

3.3.8. Correlación cruzada

```python
df = pd.concat([patient, supplemental], axis=0, ignore_index=True).fillna(0)
df = df.rename(columns={"upd23b_clinical_state_on_medication": "on_medication"})

df["on_medication"] = (
    df["on_medication"]
    .case_when(
        [
            (df.on_medication.eq("On"), 1),
            (df.on_medication.eq("Off"), -1),
        ]
    )
    .fillna("0")
)
df = df.drop(columns=["patient_id", "visit_id"])
cross_corr_matrix = df.corr()

plt.figure(figsize=(20, 16))
sns.heatmap(
    cross_corr_matrix,
    annot=False,
    cmap="coolwarm",
    cbar=True,
    square=True,
    xticklabels=True,
    yticklabels=True,
    linewidths=0.1,
)

plt.title("Matriz de correlación cruzada", fontsize=16)
plt.xticks(fontsize=20, rotation=90)
plt.yticks(fontsize=20)
plt.tight_layout()
plt.savefig("correlation.png")
```

Listado 9: Creación de una representación de mapa de calor de la matriz de correlación.

Se utiliza una representación de mapa de calor de la matriz de correlación, obtenida calculando el coeficiente de correlación de Pearson dado por

$$\rho_{X,Y} = \frac{\operatorname{cov}(X,Y)}{\sigma_X \sigma_Y}$$

El coeficiente de correlación de Pearson es la covarianza normalizada a un intervalo de $[-1, 1]$. Captura **exclusivamente** la relación lineal entre 2 variables, esto significa que el coeficiente de Pearson puede ser bajo para dos variables que sí están relacionadas si su relación es no lineal.

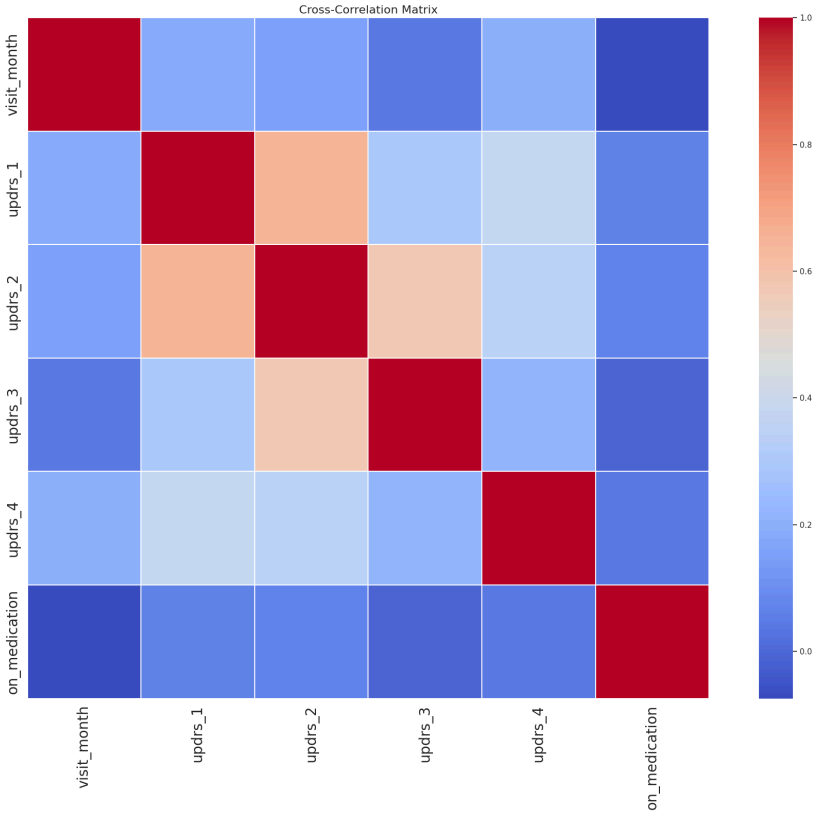

Figura 8: Mapa de calor de la matriz de correlación.

Se observa que las correlaciones ms importantes son entre los valores en las distintas categorías de UPDRS, que tienden a crecer a la vez. No obstante, la relación con el mes de visita no es aparente utilizando esta métrica. Con el objetivo de ilustrar más estas correlaciones encontradas, observamos también los siguientes *scatter plots*, donde, por pares, se tiene en cada eje una de las variables UPDRS. La relación lineal es evidente de esta manera.

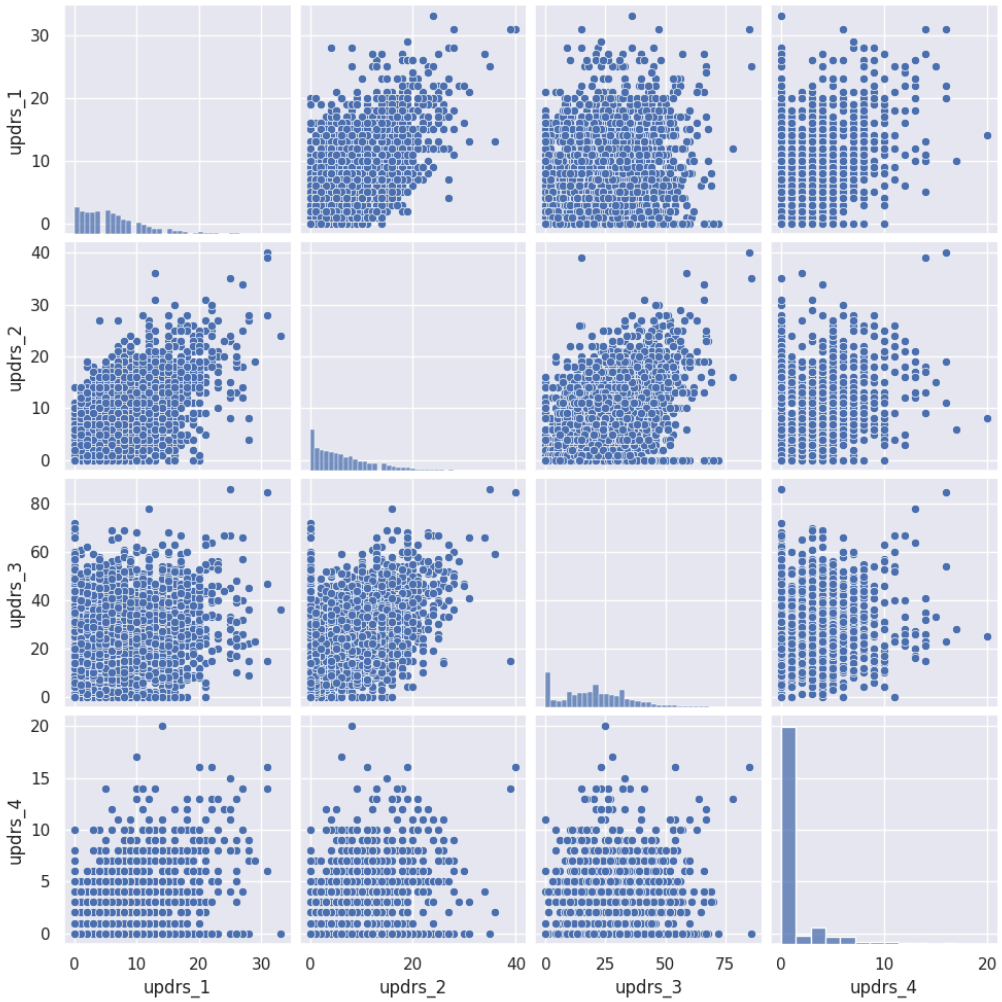

Figura 9: *Pair plot* para las variable UPDRS. La diagonal es un histograma de cada una de las variables, mientras que en el resto de celdas se tiene un *scatter plot* con las variables correspondientes en los ejes.

Capítulo 4. Modelo basado en una red neuronal simple

En este capítulo desarrollaremos un modelo sencillo basado en una red neuronal simple, que tomaremos como base con la que comparar nuestra solución. Gran parte del preprocesado de los datos para preparar las características que tomará el modelo como entrada (*feature engineering*) será común al utilizado finalmente para el modelo Seq2Seq, con lo que se entrará en detalle en esa parte.

Este modelo esta inspirado en las mejores soluciones para el reto original, varias de entre ellas basadas en redes neuronales. Estas soluciones tienen también en común que ignoran por completo los datos de proteínas y péptidos, ya que consistentemente encontraron que introducía más ruido y que la mayor cantidad de información se encontraba en columnas como el mes de visita o si para una visita se tiene información de proteínas. En este capítulo intentamos también replicar estos hallazgos.

Recordamos que el objetivo es dar una predicción a 6, 12 y 24 meses en cada una de las cuatro categorías UPDRS.

4.1. Preprocesado de los datos

En esta sección se describe como se ha llevado a cabo el tratamiento de los datos para finalmente entrenar el modelo base. Este es un paso esencial en el proceso del que depende gran parte del éxito o fracaso de un modelo. Se ha dividido en dos partes:

- **Limpieza de datos**: Tratamiento de valores nulos, normalización, …
- **Creación de características objetivo**: En este caso las características objetivo son los valores UPDRS pasado cierto número de meses, estos datos no están presentes directamente en los datos de entrada y se deben obtener en esta etapa de preprocesado.
- **Creación de características de entrada auxiliares**: Este paso está especialmente inspirado en las soluciones más exitosas del reto que señalan el uso de características como mes de la última visita o la presencia de datos proteómicos.

Partimos de haber cargado los datos de igual manera que en el capítulo anterior en *DataFrames* de *pandas*.

4.1.1. Limpieza de datos

El primer paso es normalizar los valores a utilizar. En el caso de los valores de péptidos y proteínas primero utilizamos los valores en escala logaritmica para la métrica NPX y posteriormente los normalizamos utilizando el rango (*min max scaling*).

En el caso de los datos referentes a UPDRS, trataremos de igual modo a los procedentes de ambos conjuntos de datos teniendo en cuenta en cualquier caso que pueden no tener una medición de proteínas y péptidos asociada. La normalización de los valores UPDRS se realiza de nuevo utilizando los valores máximos especificados en la definición de la escala.

```python
scaled_patient = pd.concat([patient,supplemental])
updrs_ranges = [52,52,132,24]
updrs_cols = [f"updrs_{i}" for i in range(1,5)]
for updrs_range, col in zip(updrs_ranges, updrs_cols):
    scaled_patient[col] /= updrs_range

scaled_protein = proteins.copy()
scaled_protein["NPX"] = np.log2(proteins["NPX"])
scaled_protein = (
    scaled_protein[["UniProt", "NPX"]]
    .groupby("UniProt")
    .agg(["min", "max"])
    .droplevel(0, axis=1)
    .join(proteins.set_index("UniProt"))
)
scaled_protein["NPX"] = (
  (scaled_protein["NPX"] - scaled_protein["min"]) /
  (scaled_protein["max"] - scaled_protein["min"])
).drop(columns=["min", "max"])

scaled_peptide = peptides.copy()
scaled_peptide["PeptideAbundance"]= np.log2(peptides["PeptideAbundance"])
scaled_peptide = (
    scaled_peptide[["UniProt", "PeptideAbundance", "Peptide"]]
    .groupby(["UniProt", "Peptide"])
    .agg(["min", "max"])
    .droplevel(0, axis=1)
    .join(peptides.set_index(["UniProt", "Peptide"]))
)
scaled_peptide["PeptideAbundance"] = (
  (scaled_peptide["PeptideAbundance"] - scaled_peptide["min"]) /
  (scaled_peptide["max"] - scaled_peptide["min"])
).drop(columns=["min", "max"])
```

Listado 10: Normalización de datos de proteínas, péptidos y UPDRS.

Además como parte de la limpieza de los datos se transforma la columna que indica si un paciente esta o no usando medicación en una columna númerica, ya que es provista como cadena y por lo tanto no sería utilizable por el modelo. Dado que realmente se tienen 3 valores posibles en esta columna, «On», «Off» y valor nulo, se ha decidido hacer corresponderles respectivamente 1, −1 y

0. De esta manera a la falta de información sobre el estado del paciente con respecto a la medicación

se le asigna un valor númerico intermedio. El objetivo es poder representar de manera interpretable esta falta de información al modelo ya que en ningún caso puede tratar con valores nulos directamente.

```python
scaled_patient = scaled_patient.rename(
    columns={"upd23b_clinical_state_on_medication": "on_medication"}
)
scaled_patient["on_medication"] = (
    scaled_patient["on_medication"]
    .case_when(
        [
            (scaled_patient.on_medication.eq("On"), 1),
            (scaled_patient.on_medication.eq("Off"), -1),
        ]
    )
    .fillna("0")
)
```

Listado 11: Tratamiento de la variable que representa si el paciente está, en el momento de la visita, tomando alguna medicación.

4.1.2. Creación de características objetivo

Se identifican 12 característias objetivo, correspondientes del producto entre los 3 instantes de tiempo requeridos (dentro de 6, 12 y 24 meses) y las 4 categorías de la escala UPDRS consideradas.

```python
from itertools import product

def safe_get(patient_id, visit_month, target_col):
    try:
        return indexed_scaled_patient.loc[(patient_id, visit_month), [target_col]].iloc[
            0
        ]
    except KeyError:
        return np.nan

with_leads = scaled_patient
indexed_scaled_patient = scaled_patient.set_index(["patient_id", "visit_month"])

for plus_months, target_col in product(
    [6, 12, 24],
    [
        "updrs_1",
        "updrs_2",
        "updrs_3",
        "updrs_4"
    ],
):
    with_leads[f"{target_col}_plus_{plus_months}"] = with_leads.apply(
        lambda row: safe_get(
            row["patient_id"],
            row["visit_month"] + plus_months,
            target_col
        ),
        axis=1,
    )
with_leads = with_leads[~with_leads.updrs_1_plus_6.isna()]
```

Listado 12: Creación de características objetivo

En este fragmento de código se limpian también aquellas visitas que NO tienen datos dentro de 6 meses. Esta decisión elimina una cantidad notable de datos, ya que como vimos en el análisis exploratorio hay visitas que no se ajustan a la cadencia semestral. Por otra parte simplifica significativamente el modelo.

4.1.3. Creación de características de entrada auxiliares

En las soluciones más exitosas dentro del contexto del desafío original se encontró mucha utilidad a características asociadas a las decisiones médicas tomadas en cada caso, que como ya hemos señalado mina la utilidad práctica real del modelo que debería informar estas mismas decisiones. Sin embargo, al formar parte de muchas de las mejores soluciones, se ha decidido incluirlas en este modelo base.

Las características incluidas son:

- Si existen datos proteómicos para una visita, indicando que el médico responsable consideró adecuado que se realizara un análisis.
- Mes de ultima visita.
- Total de visitas.
- Meses desde la última visita.

```python
with_leads = with_leads.set_index(["patient_id", "visit_month"]).join(
    proteins[["patient_id", "visit_month", "NPX"]]
    .groupby(["patient_id", "visit_month"])
    .count(),
    how="left",
).reset_index()
with_leads["did_test"] = with_leads["NPX"].case_when([(with_leads["NPX"] > 0, 1)]).fillna(0)
with_leads = with_leads.drop(columns=["NPX"])
```

Listado 13: Creación de una característica de entrada auxiliar representando si existen datos proteómicos para la visita

```python
with_leads["last_visit"] = with_leads.sort_values(by=['patient_id', 'visit_month']).groupby("patient_id")["visit_month"].shift(1).fillna(0)
with_leads["visit_diff"] = with_leads["visit_month"] - with_leads["last_visit"]
with_leads["visit_count"] = with_leads.groupby('patient_id').cumcount()
```

Listado 14: Creación de características de entrada auxiliares representando el número de visitas hasta la visita dada y la diferencia en meses entre la visita actual y la anterior.

4.2. Entrenamiento del modelo

El primer paso para pasar a entrenar el modelo será crear el conjunto de datos de entrada y el conjunto de datos de salida seleccionando las características adecuadas que hemos estado creando. Además pasamos de un `DataFrame` en `pandas` a arrays de `numpy`.

```python
import numpy as np
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from sklearn.model_selection import KFold

no_na = with_leads.fillna(0)
feature_cols = [
    "visit_month",
    "did_test",
    "on_medication",
    "updrs_1",
    "updrs_2",
    "updrs_3",
    "visit_count",
    "visit_diff",
    "updrs_4"
]
target_cols = [
    f"{target_col}_plus_{plus_months}"
    for plus_months, target_col in product(
        [6, 12, 24],
        [
            "updrs_1",
            "updrs_2",
            "updrs_3",
            "updrs_4"
        ],
    )
]
X = no_na[feature_cols].to_numpy(dtype="float")
y = no_na[target_cols].to_numpy(dtype="float")

```

Listado 15: Creación de los conjuntos de entrada y de salida

Posteriormente definiremos la métrica SMAPE que intentaremos minimizar durante el entrenamiento:

$$\text{SMAPE} = \left(\frac{100}{n}\right) \sum_{t=1}^{n} \frac{|\hat{y}_t - y_t|}{(|y_t| + |\hat{y}_t|)/2}$$

Se pueden definir métricas personalizadas mediante funciones. Destacamos de la definición dos puntos:

- No se calcula la métrica como un porcentaje. Como se va a optimizar el módelo buscando un mínimo para esta función, el resultado es el mismo si no multiplicamos por 100.
- Se añade un sumando muy pequeño al denominador para evitar la división por 0 en el caso en que tanto el valor real como el valor que se ha obtenido son 0.

```python
def smape(y_true, y_pred):
    epsilon = 1e-10
    numerator = tf.abs(y_true - y_pred)
    denominator = tf.abs(y_true) + tf.abs(y_pred) + epsilon
    smape = 2 * numerator / denominator
    return tf.reduce_mean(smape)
```

Listado 16: Definición de la función de pérdida: SMAPE

Definimos ahora el modelo. Se han probado distintas definiciones de tamaños y número de capas intermedias, coeficiente de *dropout* y funciones de activación con resultados similares

```python
def create_model():
    model = Sequential(
        [
          Dense(64, activation="relu"),
          Dropout(0.1),
          Dense(32, activation="relu"),
          Dropout(0.1),
          Dense(16, activation="relu"),
          Dropout(0.1),
          Dense(12, activation="linear"),
        ]
    )
    model.compile(optimizer="adam", loss=smape, metrics=["mae"])
    return model
```

Listado 17: Definición del modelo

Este modelo está diseñado para un problema de regresión con 9 características de entrada y 12 de salida. La capa de entrada se omite en la definición.La red tiene tres capas ocultas (las intermedias entre entrada y salida) con 64, 32 y 16 neuronas respectivamente. Esta es una forma típica de red neuronal, en embudo (*funnel*). cada una con activación ReLU para capturar no linealidades.

Se aplica un *dropout* del 10% después de cada capa oculta para combatir el **sobreajuste** (*overfitting*). El dropout funciona “apagando” aleatoriamente un porcentaje de neuronas durante el entrenamiento, forzando a la red a no depender demasiado de ninguna unidad en particular. Esto promueve una representación más robusta y dispersa del conocimiento, mejorando la capacidad del modelo para generalizar a datos nuevos, especialmente cuando la cantidad de datos es limitada.

Por último, el modelo se compila con el optimizador Adam Adam es una variante avanzada del descenso de gradiente que adapta la tasa de aprendizaje (cuanto pueden variar los coeficientes en cada paso de entrenamiento) para cada parámetro de forma automática. Esto acelera la convergencia y reduce la necesidad de ajustes manuales.

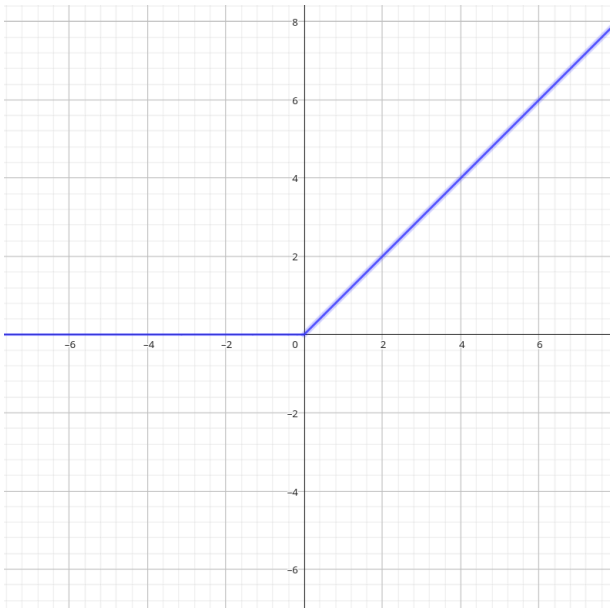

Figura 10: La función $\mathrm{ReLu}(x) = \max(x, 0)$

Pasamos al entrenamiento como tal del modelo, una vez hemos definido el modelo usando `pytorch` resulta tan sencillo como llamar al método fit con los datos y parámetros adecuados.

Se utiliza *K-Fold cross-validation* para evaluar de forma más robusta el desempeño del modelo, especialmente cuando se cuenta con un conjunto de datos pequeño. En lugar de entrenar y validar una sola vez con una partición fija, K-Fold divide los datos en varias "folds" y entrena el modelo en diferentes combinaciones de entrenamiento y validación. Esto ayuda a reducir la varianza en la estimación del error y asegura que el modelo generalice bien a distintos subconjuntos del conjunto de datos, minimizando el riesgo de sobreajuste a una partición particular. Además, proporciona una métrica más confiable y estable del rendimiento real, lo cual es fundamental para tomar decisiones informadas sobre ajustes o mejoras del modelo.

Las épocas (*epochs*) representan cuántas veces el modelo va a recorrer todo el conjunto de entrenamiento durante el aprendizaje. Más epochs permiten al modelo ajustar mejor sus pesos, pero demasiados pueden causar sobreajuste, especialmente con datos pequeños. Por otro lado, el tamaño de lote (*batch size*) indica cuántas muestras procesa el modelo antes de actualizar sus parámetros en cada paso del entrenamiento. Un tamaño de lote pequeño, como 16, hace que las actualizaciones sean más frecuentes y ruidosas, lo que puede ayudar a escapar de mínimos locales y mejorar la generalización, aunque aumenta el tiempo de entrenamiento.

```python
kf = KFold(n_splits=5, shuffle=True, random_state=42)
cv_results = []
for train_idx, val_idx in kf.split(X):
    X_train_fold, X_val_fold = X[train_idx], X[val_idx]
    y_train_fold, y_val_fold = y[train_idx], y[val_idx]

    model = create_model()
    history = model.fit(
        X_train_fold,
        y_train_fold,
        validation_data=(X_val_fold, y_val_fold),
        epochs=50,
        batch_size=16,
        verbose=1,
    )

    val_loss = model.evaluate(X_val_fold, y_val_fold, verbose=0)[0]
    cv_results.append(val_loss)

print(
    f"Cross-Validation SMAPE Loss: {np.mean(cv_results):.4f} ± {np.std(cv_results):.4f}"
)
```

Listado 18: Entrenamiento del modelo, incluyendo los resultados de la validación cruzada mediante K-folds

El resultado obtenido es es siguiente:

```
Cross-Validation SMAPE Loss: 1.1076 ± 0.0171
```

Esto indica que, tras aplicar validación cruzada, el modelo obtuvo un SMAPE medio de 1.1076 (equivalente a un 110.76 %), con una variabilidad entre particiones de ±0.0171 (≈ 1.71 %). Este valor alto muestra que las predicciones, en promedio, difieren de los valores reales en más de su magnitud, y la baja desviación indica que el modelo mantiene un comportamiento similar entre conjuntos de validación. En conclusión, el error es muy significativo.

4.3. Visualización de los resultados del modelo

Para visualizar las predicciones necesitamos evaluar el modelo para uno de los casos en concreto y se compararán con los datos reales. Se selecciona un caso del conjunto de validación.

```python
i = 0
real = no_na.iloc[val_idx].to_dict(orient="records")[i]
def predict(real):
    return dict(
        zip(
            target_cols,
            map(
                float,
                model.call(
                    inputs=np.array(
                      [
                        [real[col] for col in feature_cols]
                      ]
                    )
                ).numpy()[0],
            ),
        )
    ) | {
      k: v for k, v in real.items()
        if k in {f"updrs_{i}" for i in range(1, 5)}
    }

def to_ys(data):
    return [
        [
            data[
              f"updrs_{i}{'_plus_' + str(month) if month > 0 else ''}"
            ]
            for month in [0, 6, 12, 24]
        ]
        for i in range(1, 5)
    ]
```

Listado 19: Evaluación del modelo para un caso del conjunto de validación y se le da un formato más práctico para graficarlo.

```python
def plot(real, colors = ["#61bbb6", "#c3f2f0","#ad56cd", "#4a3b85"]):
    predicted = predict(real)
    x = [0, 6, 12, 24]
    real_ys = to_ys(real)
    predicted_ys = to_ys(predicted)
    for real_y, predicted_y, color, i in zip(
              real_ys,
              predicted_ys,
              colors,
              range(1,5)
        ):
        plt.plot(x, real_y, color = color, label = f"Real updrs_{i}")
        plt.plot(
          x, predicted_y, '-.', color = color,
          label = f"Predicted updrs_{i}"
        )
    plt.legend()
    plt.gcf().set_size_inches((18,6))
    plt.show()
```

Listado 20: Graficar datos reales y prediciones para su comparación.

Estos son algunos de los resultados. Cada color representa una de las categorías UPDRS. Las líneas continuas representan los datos reales y las discontinuas las predicciones.

En los ejemplos seleccionados al azar todos los datos de UPDRS-4 figuran como 0. Es muy probable que estos ceros provengan de los valores faltantes que fueron rellenados. El modelo no predice 0 en ninguno de los ejemplos para UPDRS-4 lo cual puede que haya contribudo al error.

Para cuantificar el efecto de la categoría 4 de UPDRS, (un caso especial por su grna número de valores faltantes) la eliminamos tanto de las características de entrada como de las variables objetivos y obtemos un error reducido aunque aún muy significativo: 0.8993 ± 0.0294.

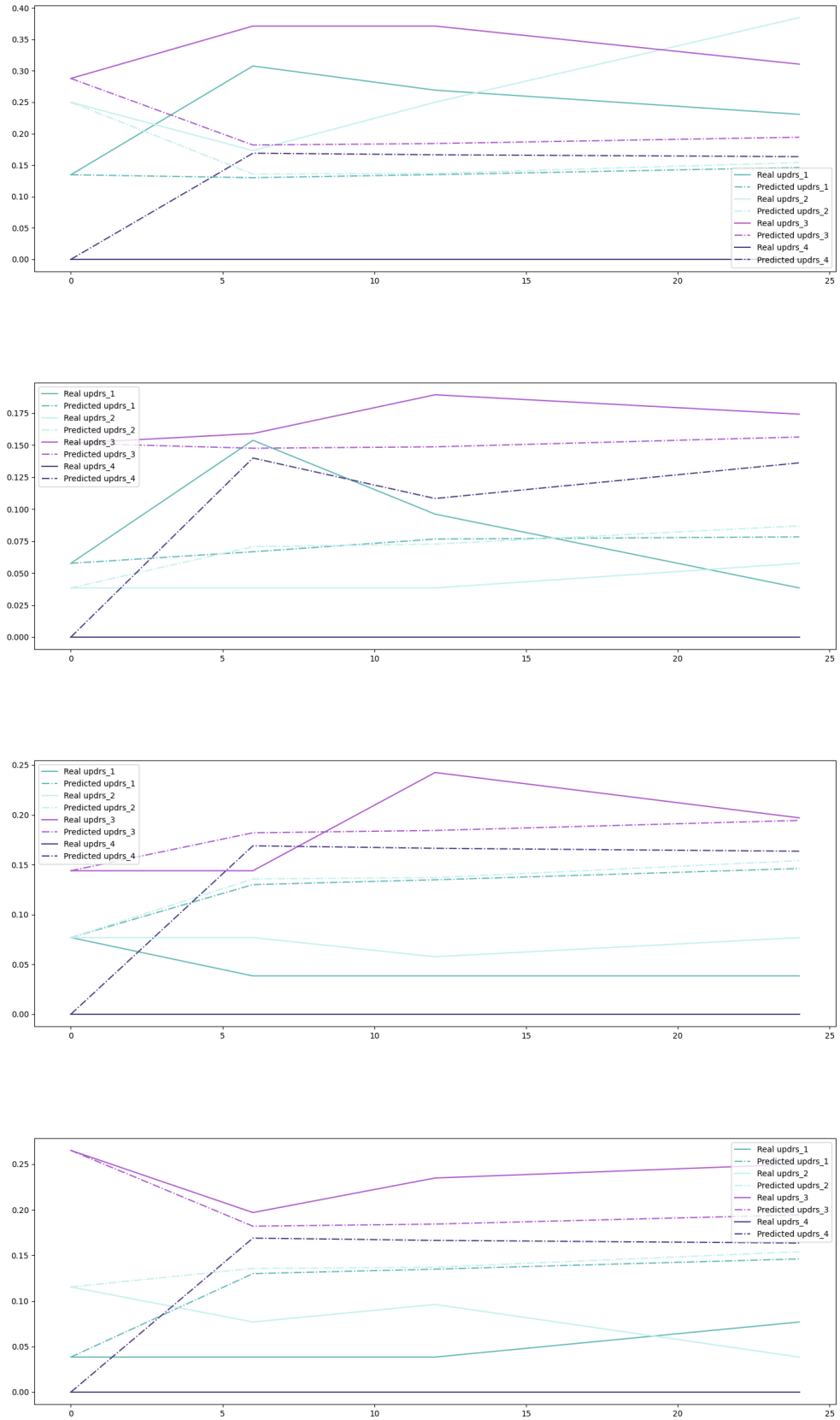

Figura 11: Gráficos comparativos de los datos reales con las predicciones del modelo.

4.4. Incluyendo datos de péptidos

Probamos también a añadir información sobre péptidos y proteínas. Se muestra el proceso para los péptidos, necesitamos que la información de cada péptido este en una columna distinta, asociada a una visita concreta. Para ello utilizamos una operación llamada `pivot`.

La operación pivot consiste en reorganizar una tabla para cambiarla de un formato “largo” a un formato “ancho”. En el formato largo, cada fila suele representar una observación con una etiqueta que indica de qué categoría es y un valor asociado. El pivot transforma esas etiquetas en columnas separadas, colocando en cada celda el valor que corresponda.

Este cambio de forma facilita ciertos análisis y comparaciones, ya que cada columna pasa a representar una categoría específica y cada fila mantiene la identificación única de la observación. Además, si en la tabla original había varias filas que correspondían a la misma combinación de identificadores y categoría, se pueden agregar sus valores usando una función como la suma o el promedio, evitando duplicados y dejando una estructura limpia y consistente. En este caso sabemos que no hay duplicados, pero es necesario que especifiquemos una función de agregación, especificamos la suma de los valores pero no es relevante.

Por otra parte, recordamos que en la primera sección de este capítulo limpiamos ya los datos de los péptidos y proteínas normalizando usando el rango.

python

```python
with_leads = (
    with_leads.set_index(["patient_id", "visit_month"])
    .join(
        scaled_peptide.pivot_table(
            values="PeptideAbundance",
            index=["patient_id", "visit_month"],
            columns=["Peptide"],
            aggfunc="sum",
        ).fillna(0)
    )
    .reset_index()
)
```

Listado 21: Añadir columnas con los datos de abundancia de péptidos a cada visita.

Finalmente especifiacmos estas nuevas columnas como características de entrada, para compesar este aumento de tamaño en los datos de entrada aumentamos también los tamaños de las capas ocultas de la red. Los resultados son peores que cuando no se incluye esta información: 1.3317 ± 0.0219.

Capítulo 5. Solución propuesta

En esta parte del trabajo vamos a centrarnos ya en concreto en la propuesta de solución que hemos implementado. Como referencias para el capítulo se han utilizado un artículo titulado «Long Time Series Deep Forecasting with Multiscale Feature Extraction and Seq2seq Attention Mechanism» [8], una introducción a las RNN publicada por IBM [9] y otras [10].

5.1. Redes Neuronales Recurrentes (RNN)

Las redes neuronales recurrentes (RNN, por sus siglas en inglés) constituyen una clase de redes neuronales especialmente diseñadas para procesar datos secuenciales. A diferencia de arquitecturas tradicionales, las RNN poseen una estructura que incorpora memoria interna, lo que les permite retener información de entradas anteriores para influir en el procesamiento de entradas actuales. Esta característica las hace particularmente efectivas en tareas donde el orden temporal o contextual de los datos es relevante, como la predicción de series temporales, el análisis de texto, la traducción automática o el procesamiento de audio.

A nivel estructural, una característica distintiva de las RNN es que comparten sus parámetros (pesos) a través del tiempo, es decir, los mismos pesos se utilizan en cada paso de la secuencia. Este mecanismo reduce significativamente el número de parámetros necesarios y facilita el modelado de relaciones temporales en los datos. No obstante, las RNN tradicionales presentan una limitación importante conocida como el problema del desvanecimiento del gradiente (vanishing gradient problem).

Durante el entrenamiento de una red neuronal, los errores entre las predicciones y los valores reales se propagan hacia atrás mediante un algoritmo denominado backpropagation con el fin de actualizar los pesos. En el caso de las RNN, se utiliza una variante de este algoritmo llamada Backpropagation Through Time (BPTT), la cual extiende la retropropagación a lo largo de todos los pasos temporales de la secuencia.

Sin embargo, cuando la secuencia de entrada es extensa, los gradientes calculados en los pasos finales del tiempo tienden a multiplicarse repetidamente por valores pequeños (derivadas de funciones de activación como la sigmoide o la tangente hiperbólica). Como consecuencia, estos gradientes pueden decrecer exponencialmente al propagarse hacia pasos más antiguos, lo que hace que los pesos correspondientes apenas se actualicen. Este fenómeno provoca que la red tenga dificultades para aprender dependencias a largo plazo, lo que limita su eficacia en tareas donde la información relevante se encuentra en puntos temporales distantes de la secuencia.

Con el objetivo de superar esta limitación, se han propuesto diversas variantes de las RNN:

- **Long Short-Term Memory (LSTM):** Introducida por Hochreiter y Schmidhuber (1997), esta arquitectura incorpora un mecanismo de memoria explícita mediante celdas y compuertas (entrada, olvido y salida) que controlan de manera dinámica qué información debe conservarse o descartarse. Esto permite preservar gradientes estables a lo largo del tiempo y facilita el aprendizaje de dependencias a largo plazo.
- **Gated Recurrent Units (GRU):** Variante simplificada de las LSTM que utiliza solo dos compuertas (actualización y reinicio). Las GRU mantienen un rendimiento comparable al

de las LSTM, pero con una estructura más compacta y eficiente en términos computacionales.

- **Redes recurrentes bidireccionales (BRNN):** Permiten a la red considerar tanto el pasado como el futuro de una secuencia al procesar los datos en ambas direcciones. Esto es útil en tareas donde la comprensión del contexto completo es crucial.
- **Modelos codificador-decodificador (Encoder–Decoder):** Utilizados frecuentemente en tareas de secuencia a secuencia (Seq2Seq), estos modelos consisten en un codificador que transforma la secuencia de entrada en un vector de contexto fijo, y un decodificador que genera la secuencia de salida a partir de dicho vector. Dado que el vector de contexto puede ser una limitación en secuencias largas, es común complementar esta arquitectura con mecanismos de atención (attention), que permiten al decodificador enfocarse dinámicamente en partes específicas de la secuencia original.

En todas estas variantes, las RNN aplican funciones de activación no lineales como sigmoide, tanh y ReLU, que afectan directamente al comportamiento de la red y al flujo del gradiente. En particular, tanh es preferida en muchos casos por centrar sus salidas alrededor de cero, lo que mejora la estabilidad numérica y el aprendizaje.

5.2. Redes Seq2Seq

Las redes neuronales Sequence-to-Sequence (Seq2Seq) son una clase de modelos de aprendizaje profundo diseñados para transformar una secuencia de entrada de longitud variable en una secuencia de salida también de longitud variable. Este enfoque fue introducido originalmente por Sutskever et al. (2014) para tareas como la traducción automática, y desde entonces se ha utilizado ampliamente en procesamiento del lenguaje natural (NLP), síntesis de voz, y predicción de series temporales ya que esta arquitectura es especialmente útil en problemas donde la relación entre entrada y salida no es uno a uno y donde hay dependencias temporales complejas.

Un modelo Seq2Seq consta principalmente de dos componentes:

- **Codificador (Encoder):** Una red neuronal recurrente (RNN) como LSTM o GRU, que procesa la secuencia de entrada paso a paso y resume su contenido en un vector de estado oculto (estado final del codificador). Este vector intenta capturar toda la información relevante de la entrada.
- **Decodificador (Decoder):** Otra RNN (o LSTM/GRU), que toma el vector de estado del codificador como entrada inicial y genera la secuencia de salida. En modelos más avanzados, como los que incorporan attention, el decodificador también puede acceder a todos los estados intermedios del codificador, lo cual mejora el rendimiento en secuencias largas.

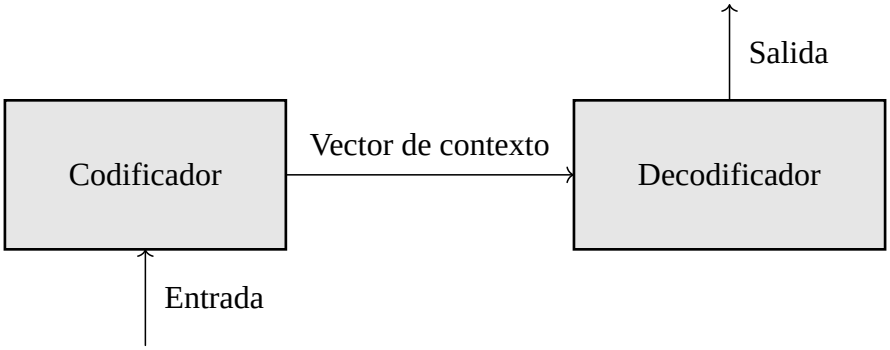

Figura 12: Descripcion

En el contexto de la predicción de series temporales (Forecasting), el codificador toma una secuencia, o ventana temporal y el decodificador intenta predecir los pasos siguientes, lo que se conoce como multi-step / N-step forecasting.

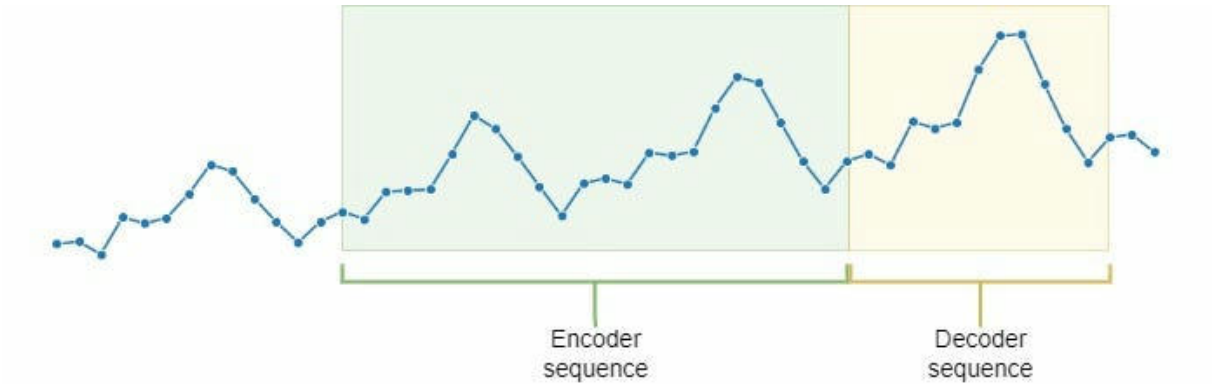

Figura 13: Serie temporal Codificador-Decodificador

Más en detalle,

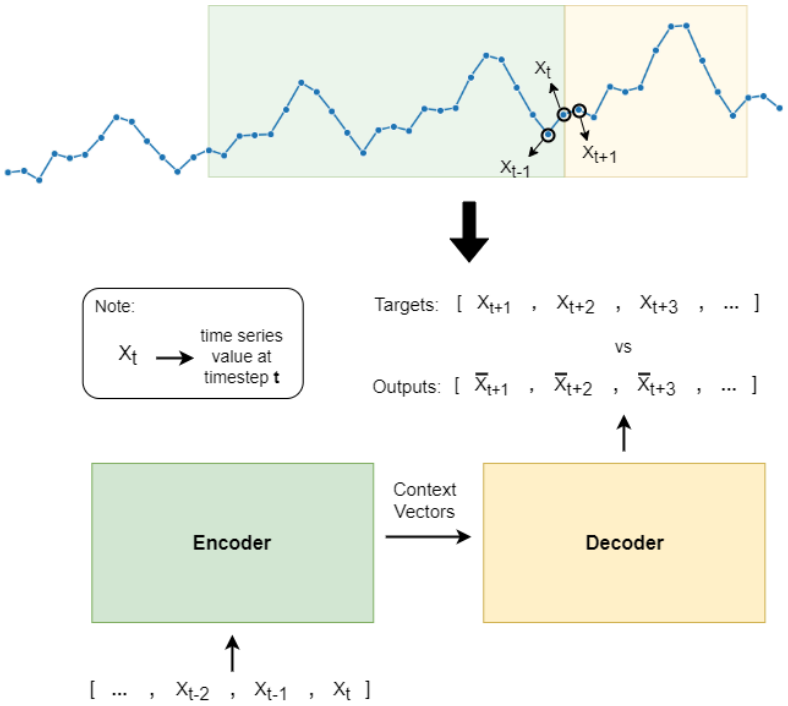

Figura 14: Paso Codificador-Decodificador

5.2.1. Mecanismo de Atención en Series Temporales

En ciencias cognitivas, se ha observado que los seres humanos tienden a enfocar selectivamente su atención en una parte de la información disponible, ignorando el resto. El mecanismo de atención en redes neuronales es conceptualmente similar a este proceso de atención visual humana: se centra en extraer la información más relevante dentro de un conjunto de datos amplio. Este mecanismo ha sido ampliamente adoptado en tareas de aprendizaje profundo basadas en modelos RNN (Redes Neuronales Recurrentes) y CNN (Redes Neuronales Convolucionales).

Bahdanau et al. introdujeron por primera vez el mecanismo de atención en modelos seq2seq, con el objetivo de predecir una secuencia objetivo a partir del aprendizaje de la información más relevante entre todos los datos anteriores. Posteriormente, Vaswani et al. propusieron el modelo Transformer, que elimina la recurrencia y se basa completamente en el mecanismo de atención para establecer dependencias globales entre la entrada y la salida. Por su parte, Lai et al. aplicaron de forma innovadora la atención básica a los estados ocultos de una GRU (Unidad Recurrente con Puertas), aprovechando así la información de cada instante de tiempo. Para mejorar este mecanismo, Shih et al. introdujeron un concepto de atención multivariable, que selecciona las variables más correlacionadas en lugar de los pasos temporales más relevantes.

En esta sección se presentan el mecanismo de atención básica y su aplicación en arquitecturas seq2seq.

⪌ Mecanismo de Atención Básica

El mecanismo de atención básica busca aprender una combinación ponderada de las entradas para determinar qué elementos del pasado deben considerarse más relevantes para una tarea de predicción.

Dado un conjunto de datos de series temporales: $X = [x_1, x_2, ..., x_t]^T$ El cálculo de la atención sobre cada elemento x_i con respecto al elemento actual x_t se realiza con las siguientes fórmulas:

$$\alpha_i = f(x_i, x_t), \ \text{ para } i = 1, 2, ..., t$$

$$A = \text{softmax}\big([\alpha_1, \alpha_2, ..., \alpha_t]^T\big)$$

$$y = A^T \cdot X$$

donde $f(\cdot)$ es una función de medida de correlación y la función softmax se define con respecto al input $\alpha = [\alpha_1, \alpha_2, ..., \alpha_t]^T$ como:

$$A_i = \frac{\exp(\alpha_i)}{\sum_{\{j=1\}}^{\{t\}} \exp\big(\alpha_j\big)}$$

$$A = [A_1, A_2, ..., A_t]^T$$

Es decir, normaliza el vector en una distribución de probabilidad que consta de T probabilidades proporcionales a los exponenciales de los números de entrada.

⋛ Mecanismo de Atención en Modelos Seq2Seq

El mecanismo de atención aplicado a modelos secuencia a secuencia (seq2seq) permite mejorar la calidad de las predicciones en tareas temporales al considerar la relevancia de cada estado oculto codificado.

Dada la serie de entrada: $X = [x_1, x_2, ..., x_t]^T$ Se inicializa el estado oculto de la RNN como h_0. En la fase del codificador, se actualizan los estados ocultos secuencialmente como sigue:

$$h_i = \text{RNN}\Big(x_i, h_{\{i-1\}}\Big) \text{ para } i = 1, 2, ..., t$$

Y calculamos estos estados de manera recurrente hasta llegar a $i = t$ para obtener como output del codificador el vector $h = [h_1, h_2, ..., h_t]^T$.

Posteriormente, En la fase del decodificador, se aplica atención entre el último estado oculto del codificador h_t y todos los estados h para obtener el vector de contexto:

$$s_0 = \text{Attention}(h_t, h)$$

A partir de aquí, se generan predicciones paso a paso. Se inicializa la entrada del decodificador como $y_0 = x_t$ y para cada paso k, se concatenan $s_{\{k-1\}}$ y $y_{\{k-1\}}$ y se procesan mediante la RNN:

$$h_{\{k+t\}} = \text{RNN}\Big(\Big[s_{\{k-1\}}, y_{\{k-1\}}\Big], h_{\{k+t-1\}}\Big)$$

Y después, se aplica nuevamente atención entre $h_{\{k+t\}}$ y la secuencia de estados ocultos:

$$s_k = \text{Attention}\Big(h_{\{k+t\}}, h\Big)$$

Este mecanismo permite que el modelo aprenda de manera efectiva qué partes del historico de de la serie son más relevantes en cada paso del proceso de predicción.

5.2.2. Teacher Forcing y Scheduled Sampling

El teacher forcing es un esquema de entrenamiento clásico en modelos de secuencia basados en redes neuronales recurrentes (RNNs). Su funcionamiento consiste en que, durante el entrenamiento, en cada paso de la secuencia el modelo recibe como entrada el token real previo proveniente de los datos, en lugar de la predicción generada por el propio modelo.

Esta estrategia acelera la convergencia y estabiliza el aprendizaje, ya que evita que los errores se propaguen en etapas tempranas de la generación. Sin embargo, introduce una discrepancia importante entre las fases de entrenamiento e inferencia. En la práctica, durante la inferencia los tokens reales no están disponibles y deben ser reemplazados por las predicciones del modelo. Esta diferencia, conocida como exposure bias, puede dar lugar a que errores iniciales se amplifiquen y acumulen a lo largo de toda la secuencia generada.

Con el objetivo de reducir el sesgo de exposición introducido por el teacher forcing, Bengio et al.[11] propusieron la técnica de scheduled sampling, enmarcada dentro de los métodos de curriculum learning. La idea central es introducir de manera progresiva el uso de los tokens generados por el propio modelo durante el entrenamiento.

En la práctica, para cada paso t de la secuencia, se define una variable binaria $z_t \sim \text{Bernoulli}(\varepsilon_t)$ donde $\varepsilon_t \in [0, 1]$ representa la probabilidad de usar el token real. Así, la entrada al modelo en el instante t se define como:

$$\tilde{y}_{t_1} = \begin{cases} y_{t-1}, \text{ si } z_t = 1 \\ \hat{y}_{t-1}, \text{ si } z_t = 0 \end{cases}$$

El comportamiento de ε_t a lo largo del entrenamiento está gobernado por una *decay function* (función de decaimiento), que controla la transición desde un entrenamiento completamente guiado $(\varepsilon_t \approx 1)$ hacia un entrenamiento menos guiado $(\varepsilon_t \to 0)$.

Entre las funciones de decaimiento más comunes se incluyen:

- **Lineal:** $\varepsilon_t = \max(\varepsilon_0 - kt, 0)$
- **Exponencial:** $\varepsilon_t = \varepsilon_0 \cdot \alpha^t \ , \ \ \alpha < 1$
- **Sigmoide inversa:** $\varepsilon_t = \frac{k}{k + \exp(t/k)}$

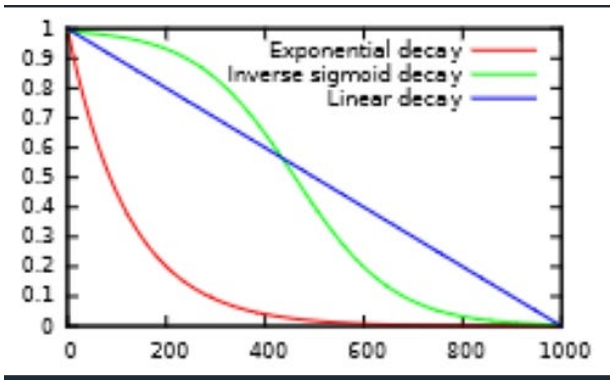

Figura 15: Funciones de decaimiento

Capítulo 6. Implementación del algoritmo Seq2Seq

En el capítulo anterior se ha descrito en detalle la arquitectura seq2seq, el mecanismo de atención y los principios teóricos que sustentan estos modelos. Hemos comprendido cómo un codificador transforma una secuencia de entrada en una representación latente y cómo un decodificador, apoyado en la atención, genera paso a paso una secuencia de salida mientras selecciona dinámicamente la información más relevante del contexto.

En este capítulo pasamos de la teoría a la práctica: presentamos la implementación en PyTorch de un modelo seq2seq con atención. El objetivo no es únicamente mostrar el código, sino desentrañar cada uno de sus componentes para entender cómo los bloques conceptuales estudiados previamente se convierten en operaciones programáticas concretas. A lo largo del recorrido, intercalaremos el código con explicaciones extensas, de modo que se vea claramente la relación entre teoría y práctica.

6.1. Modelo

El código fuente que se comenta en esta sección corresponde al fichero `seq2seq.py`, es un módulo de python que define la arquitectura del modelo apoyándose en `pytorch`.

Durante la explicación abordamos también según salta la necesidad conceptos propios de `pytorch` que es conveniente conocer para entender el código.

python

```python
import random
import torch
from torch import nn
```

Listado 22: Importaciones necesarias para la construcción del modelo seq2seq con atención.

El módulo comienza con las importaciones necesarias: la librería estándar `random` para manejar decisiones estocásticas (como el *teacher forcing*), y `torch` junto con `torch.nn` para construir la red neuronal. Estas son las bases sobre las que se levantará toda la arquitectura seq2seq con atención.

python

```python
def layer_init(layer, w_scale=1.0):
    nn.init.kaiming_uniform_(layer.weight.data)
    layer.weight.data.mul_(w_scale)
    nn.init.constant_(layer.bias.data, 0.0)
    return layer
```

Listado 23: Inicialización de capas lineales con Kaiming uniform y ajuste de pesos y sesgos.

La función layer_init se encarga de inicializar de manera cuidadosa las capas lineales del modelo. Usa la inicialización de Kaiming, que es adecuada para redes profundas con funciones de activación tipo ReLu, y asegura que los sesgos comiencen en cero. Este tipo de detalle evita problemas comunes como gradientes mal escalados desde el inicio del entrenamiento.

La variante Kaiming uniform asigna los pesos a partir de una distribución uniforme dentro de un rango calculado en función del número de unidades de entrada a la capa. El objetivo es

mantener estable la varianza de las activaciones a través de las distintas capas de la red, evitando que se amplifiquen o se reduzcan en exceso.

Este método fue propuesto específicamente para trabajar con activaciones ReLU, que anulan los valores negativos. Si no se aplicara un esquema de inicialización adecuado, la red podría caer en zonas muertas o generar gradientes extremadamente pequeños, ralentizando o incluso bloqueando el aprendizaje. Al aplicar Kaiming uniform se logra que la propagación hacia adelante y hacia atrás se mantenga equilibrada desde el inicio del entrenamiento, favoreciendo una convergencia más rápida y estable.

6.1.1. Codificador

python

```python
class Encoder(nn.Module):
    def __init__(self, enc_feature_size, hidden_size, num_gru_layers, dropout):
        super().__init__()
        self.gru = nn.GRU(
            enc_feature_size,
            hidden_size,
            num_gru_layers,
            batch_first=True,
            dropout=dropout,
        )
```

Listado 24: Definición de la clase Encoder basada en GRU

El codificador (*encoder*) está basado en una red recurrente del tipo GRU. Recibe una secuencia de entrada con múltiples características y produce representaciones ocultas que condensan la información temporal. Como se explicó anteriormente en la teoría, el encoder es la parte encargada de transformar la secuencia original en un espacio latente que el decodificador podrá explotar.

Más adelante se explorará la posibilidad de usar redes neuronales LSTM, sustituyendo a GRU. De momento continuamos con GRU.

python

```python
def forward(self, inputs):
    > Tensor de entrada:
    > inputs: (tamaño de lote, longitud de secuencia de entrada, número de características del codificador)
    output, hidden = self.gru(inputs)
    return output, hidden
    > Tensores de salida:
    > output: (tamaño de lote, longitud de secuencia de entrada, tamaño oculto)
    > hidden: (número de capas GRU, tamaño de lote, tamaño oculto)
```

Figura 16: Método forward del codificador.

El método `forward` del codificador devuelve tanto la secuencia completa de estados ocultos como el último estado de la GRU. El primero servirá para el mecanismo de atención y el segundo inicializará al decodificador.

6.1.2. Decodificador Base

```python
class DecoderBase(nn.Module):
    def __init__(self, device, dec_target_size, target_indices):
        super().__init__()
        self.device = device
        self.target_indices = target_indices
        self.target_size = dec_target_size
```

Listado 25: Definición del decodificador base con parámetros generales

El decodificador base establece la estructura común para todos los decodificadores que se quieran construir encima (Es una clase pensada para ser extendida por subclases). Define variables clave como el tamaño de los objetivos, a qué índices de la salida del decodificador de corresponden las variables objetivo y en qué dispositivo (CPU o GPU) se ejecutará el cálculo.

A lo largo del modulo se incluyen comentarios sobre las dimensiones que tienen los distintos tensores con los que trabaja ya que tener sus dimensiones presentes en todo momento ayuda a razonar sobre el código y a entender también el contenido de los tensores.

```python
def forward(self, inputs, hidden, enc_outputs, teacher_force_prob=None):
    > Dimensiones de los tensores de entrada:
    >> inputs: (tamaño de batch, longitud de secuencia de salida, número de características del decodificador)
    >> hidden: (número de capas GRU, tamaño de batch, dimensión oculta), es decir, el último estado oculto
    >> enc_outputs: (tamaño de batch, longitud de secuencia de entrada, tamaño oculto)
    batch_size, dec_output_seq_length, _ = inputs.shape

    outputs = torch.zeros(
        batch_size,
        dec_output_seq_length,
        self.target_size,
        dtype=torch.float,
    ).to(self.device)
    curr_input = inputs[:, 0:1, :]
    > Dimensiones: (tamaño de batch, 1, número de características del decodificador)
    for t in range(dec_output_seq_length):
        dec_output, hidden = self.run_single_recurrent_step(
            curr_input, hidden, enc_outputs
        )
        > run_single_recurrent_step debe ser implementado por subclases
        outputs[:, t : t + 1, :] = dec_output

        teacher_force = (
            random.random() < teacher_force_prob
            if teacher_force_prob is not None
            else False
        )
        curr_input = inputs[:, t : t + 1, :].clone()
        if not teacher_force:
            curr_input[:, :, self.target_indices] = dec_output
            > Si se aplica teacher forcing, usar el objetivo de este paso como la siguiente entrada; de lo contrario, usar la predicción
    return outputs
```

Figura 17: Método forward del decodificador base para el recorrido paso a paso.

La función forward del decodificador base implementa la lógica de tratar paso a paso la secuencia de salida. Aquí se decide en cada instante si usar teacher forcing o confiar en la predicción anterior. Esto refleja la dificultad práctica en el entrenamiento de modelos secuenciales: encontrar un balance entre guiar al modelo con datos reales y dejarle explorar su propia dinámica.

Como se indica, la lógica propiamente del decodificador tendrá que ser implementada en subclases del decodificador base.

6.1.3. Atención

```python
class Attention(nn.Module):
      def __init__(self, hidden_size, num_gru_layers):
          super().__init__()
          self.attn = nn.Linear(2 * hidden_size, hidden_size)
          self.v = nn.Linear(hidden_size, 1, bias=False)
          > El tamaño oculto para la salida de attn (y entrada de v) puede ser cualquier número. Además, usar dos capas permite tener una función de activación no lineal entre ellas.
```

Figura 18: Implementación del mecanismo de atención

El módulo de atención se implementa aquí como una red pequeña que compara el estado oculto actual del decodificador con cada salida del encoder. La capa lineal `attn` calcula una representación intermedia, y la proyección `v` la condensa en un valor escalar (como se observa en el tamaño de salida de la capa, 1) que representa la "energía" de atención.

```python
def forward(self, decoder_hidden_final_layer, encoder_outputs):
    > Tensores de entrada:
    >> decoder_hidden_final_layer: (tamaño de lote, tamaño oculto)
    >> encoder_outputs: (tamaño de lote, longitud de secuencia de entrada, tamaño oculto)
    hidden = decoder_hidden_final_layer.unsqueeze(1).repeat(
        1, encoder_outputs.shape[1], 1)
        > Repetir el estado oculto del decodificador tantas veces como la longitud de la secuencia de entrada
    energy = torch.tanh(self.attn(torch.cat((hidden, encoder_outputs), dim=2)))
    > Comparar el estado oculto del decodificador con cada salida del codificador usando una capa *tanh* entrenable
    attention = self.v(energy).squeeze(2)
    weightings = torch.nn.functional.softmax(attention, dim=1)
    return weightings
    > weightings: (tamaño de lote, longitud de secuencia de entrada)
```

Figura 19: Cálculo de pesos de atención mediante softmax

En el método `forward` de atención se obtiene el peso asignado a cada posición de la secuencia de entrada. La normalización con *softmax* garantiza que las ponderaciones sumen uno, de forma que pueden interpretarse como una distribución de probabilidad sobre los pasos de la secuencia. No entramos en detalle, ya que es una implementación directa de los fundamentos teóricos explicados en elcapítulo anterior.

6.1.4. Decodificador con atención

```python
class DecoderWithAttention(DecoderBase):
  def __init__(
      self,
      dec_feature_size,
      dec_target_size,
      hidden_size,
      num_gru_layers,
      target_indices,
      dropout,
      device,
  ):
      super().__init__(
          device,
          dec_target_size,
          target_indices,
      )
      self.attention_model = Attention(hidden_size, num_gru_layers)
      self.gru = nn.GRU(
          dec_feature_size + hidden_size,
          > GRU toma el objetivo del paso anterior y la suma ponderada de los estados ocultos del codificador (resultado de la atención)
          hidden_size,
          num_gru_layers,
          batch_first=True,
          dropout=dropout,
      )
      self.out = layer_init(
          nn.Linear(
              hidden_size + hidden_size + dec_feature_size,
              > La capa de salida toma la salida del estado oculto del decodificador, la suma ponderada según la atención y la entrada del decodificador.
              dec_target_size,
          )
      )
```

Figura 20: Definición del decodificador con atención.

El decodificador con atención combina las ideas anteriores: recibe la entrada del paso actual, calcula el contexto mediante la atención y concatena todo para alimentar a la GRU. Posteriormente, una capa lineal genera los parámetros de salida. Aquí primero se definen las capas necesarias para todo el proceso, que se encapsulará en el método que se quedó pendiente de implementar (`run_single_recurrent_step`).

```python
def run_single_recurrent_step(self, inputs, hidden, enc_outputs):
    > Tensores de entrada:
    >> inputs: (tamaño de lote, 1, número de características del decodificador)
    >> hidden: (número de capas GRU, tamaño de lote, tamaño oculto)
    >> enc_outputs: (tamaño de lote, longitud de secuencia de entrada, tamaño
    oculto)
    weightings = self.attention_model(hidden[-1], enc_outputs)
    > Obtener los pesos de atención
    >> weightings: (tamaño de lote, longitud de secuencia de entrada)
    weighted_sum = torch.bmm(weightings.unsqueeze(1), enc_outputs)
    > Luego calcular la suma ponderada
    >> weighted_sum: (tamaño de lote, 1, tamaño oculto)
    output, hidden = self.gru(torch.cat((inputs, weighted_sum), dim=2),
hidden)
    > Después introducir en la GRU
    >> entradas de GRU: (tamaño de lote, 1, número de características del
    decodificador + tamaño oculto)
    >> output: (tamaño de lote, 1, tamaño oculto)
    output = self.out(torch.cat((output, weighted_sum, inputs), dim=2))
    > Obtener predicción
    >> entrada de out: (tamaño de lote, 1, tamaño oculto + tamaño oculto +
    número de objetivos)
    output = output.reshape(output.shape[0], output.shape[1],
self.target_size)
    return output, hidden
    > Tensores de salida:
    >> output: (tamaño de lote, 1, número de objetivos)
    >> hidden: (número de capas GRU, tamaño de lote, tamaño oculto)
```

Figura 21: Ejecución de un paso recurrente del decodificador con atención.

En cada paso, el decodificador con atención calcula primero las ponderaciones, luego el vector de contexto como combinación de las salidas del encoder, y lo introduce en la GRU junto con la entrada actual. La salida se transforma en la predicción final para ese instante. Esta combinación permite que el decodificador se “enganche” dinámicamente a diferentes partes de la secuencia de entrada según lo requiera cada predicción.

6.1.5. Juntándolo todo. Modelo Seq2Seq.

```python
class Seq2Seq(nn.Module):
    def __init__(self, encoder, decoder, lr, grad_clip):
        super().__init__()
        self.encoder = encoder
        self.decoder = decoder
        self.opt = torch.optim.Adam(self.parameters(), lr)
        self.loss_func = Seq2Seq.compute_smape
        self.grad_clip = grad_clip
```

Listado 26: Clase Seq2Seq que unifica encoder, decodificador y entrenamiento.

Finalmente la calse Seq2Seq unifica todo el sistema. Primero como en el resto de módulos se inicializan los componentes y los parámetros que afectarán al modelo. Aparece nuevamente el

optimizador Adam que ya se ha descrito, que se ajusta con el parametro `lr` (*learning rate*). Además se especifica la función de pérdida SMAPE.

El parámetro `grad_clip` merece mención especial: se utiliza para aplicar *gradient clipping*, una técnica que limita la magnitud de los gradientes durante el retropropagado. Esto resulta muy útil en modelos recurrentes, donde los gradientes pueden crecer de forma explosiva y desestabilizar el entrenamiento. Al imponer un tope, `grad_clip` garantiza que las actualizaciones de los parámetros sean más estables y que el entrenamiento progrese de manera controlada.

python

```python
@staticmethod
def compute_smape(prediction, target):
    return (
        torch.mean(
            torch.abs(prediction - target)
            / ((torch.abs(target) + torch.abs(prediction)) / 2.0 + 1e-8)
        )
        * 100.0
    )
```

Listado 27: Método estático que calcula el SMAPE (Symmetric Mean Absolute Percentage Error) entre predicciones y valores reales

Aquí vemos la implementación del cálculo de SMAPE como un método estático dentro de la clase. El hecho de declararlo con `@staticmethod` implica que no depende de la instancia concreta del modelo, sino que se puede invocar directamente desde la clase sin necesidad de crear un objeto. Dado que la fórmula ya fue explicada en detalle en un capítulo previo, basta con señalar que este método proporciona una medida porcentual del error relativo entre la predicción y el valor real.

python

```python
def compute_loss(self, prediction, target):
    > Tensores de entrada:
    > prediction: (tamaño de lote, longitud de secuencia del decodificador, número de objetivos)
    > target: (tamaño de lote, longitud de secuencia del decodificador, número de objetivos)
    loss = self.loss_func(prediction, target)
    return loss if self.training else loss.item()
    > Devuelve el tensor de pérdida si está en modo entrenamiento, sino el valor escalar
```

Figura 22: Cálculo de la función de pérdida en el modelo Seq2Seq

Esta es una función de utilidad que devuelve el resultado del cómputo de la función de pérdida como un tensor cuando el modelo está en modo de entrenamiento, mientras que en modo de evaluación devuelve un valor numérico. Esto es útil porque durante el entrenamiento se necesita mantener el grafo de cómputo para poder hacer *backpropagation*, mientras que en la evaluación basta con obtener un valor (*float*) para inspección o registro.

Más detalladamente, en PyTorch, cada operación que involucra tensores con cierto parámetro activado (`requires_grad=True`) se añade a un grafo dinámico de cómputo. Este grafo registra

las transformaciones realizadas sobre los datos para que, en el momento de llamar a `.backward()`, la librería pueda aplicar automáticamente la regla de la cadena y calcular los gradientes de todos los parámetros implicados. Mientras el modelo está en modo de entrenamiento, es fundamental conservar dicho grafo junto con la pérdida para permitir la retropropagación. Sin embargo, cuando el modelo se encuentra en fase de evaluación o inferencia, no se necesita este mecanismo y resulta más eficiente "desprenderse" del grafo, devolviendo simplemente un valor numérico (`loss.item()`) en lugar de un tensor que siga conectado a la cadena de operaciones.

```python
def forward(self, enc_inputs, dec_inputs, teacher_force_prob=None):
    > Tensores de entrada:
    > enc_inputs: (tamaño de lote, longitud de secuencia de entrada, número de características del codificador)
    > dec_inputs: (tamaño de lote, longitud de secuencia de salida, número de características del decodificador)
    enc_outputs, hidden = self.encoder(enc_inputs)
    > Salidas del codificador:
    > enc_outputs: (tamaño de lote, longitud de secuencia de entrada, tamaño oculto)
    > hidden: (número de capas GRU, tamaño de lote, dimensión oculta), es decir, el último estado oculto
    outputs = self.decoder(dec_inputs, hidden, enc_outputs, teacher_force_prob)
    > Salidas finales:
    > outputs: (tamaño de lote, longitud de secuencia de salida, número de objetivos)
    return outputs
```

Figura 23: Cálculo de la función de pérdida en el modelo Seq2Seq

Como hemos ido viendo, el método `forward` es el núcleo de cualquier clase `nn.Module` en PyTorch, ya que define cómo fluyen los datos a través del modelo. Se define en una dirección, hacia delante, y gracias al grafo de cómputo tenemos el flujo también hacia atrás.

En este caso, se reciben dos secuencias de entrada: `enc_inputs`, correspondiente a los vectores que entran en el codificador, y `dec_inputs`, que se suministran al decodificador. Primero, los datos atraviesan la red recurrente del codificador (`self.encoder`), produciendo como salida tanto las representaciones ocultas de toda la secuencia (`enc_outputs`) como el último estado oculto (`hidden`). Estos elementos constituyen el contexto que se pasará al decodificador.

A continuación, se invoca al decodificador (`self.decoder`), que utiliza tanto el estado oculto como las salidas del codificador para generar la secuencia de salida paso a paso. El parámetro opcional `teacher_force_prob` permite introducir la técnica de *teacher forcing*, que como se explicó en el capítulo teórico, controla hasta qué punto el modelo usa la salida real de entrenamiento en lugar de su propia predicción durante el proceso de decodificación. El valor devuelto, `outputs`, representa finalmente las predicciones del modelo para la secuencia objetivo.

```python
def optimize(self, prediction, target):
    > Tensores de entrada:
    >> prediction & target: (tamaño de lote, longitud de secuencia, dimensión de salida)
    self.opt.zero_grad()
    > Reiniciar gradientes acumulados
    loss = self.compute_loss(prediction, target)
    loss.backward()
    > Calcular gradientes mediante backpropagation
    if self.grad_clip is not None:
        torch.nn.utils.clip_grad_norm_(self.parameters(), self.grad_clip)
        > Recorte de gradientes si está configurado para evitar explosión de gradientes
    self.opt.step()
    > Actualizar parámetros del modelo
    return loss.item()
```

Figura 24: Método de optimización que actualiza los parámetros del modelo mediante retropropagación y *grad clipping*

Este método concentra el ciclo completo de optimización de parámetros en PyTorch. El procedimiento comienza con self.opt.zero_grad(), que limpia los gradientes acumulados en la iteración anterior; esto es esencial, ya que PyTorch por defecto acumula los gradientes. Luego se calcula la pérdida invocando a `compute_loss`, obteniendo un escalar que mide el error actual del modelo (pero, repetimos, dentro de un tensor que guarda el grafo de cómputo).

El paso siguiente es `loss.backward()`, que activa el mecanismo de retropropagación y recorre el grafo de cómputo dinámico para calcular los gradientes de cada parámetro entrenable. Si se ha especificado un valor de `grad_clip`, se aplica `torch.nn.utils.clip_grad_norm_`, que evita que los gradientes crezcan de manera desmesurada y provoquen inestabilidad numérica, un problema común en redes recurrentes. Finalmente, se llama a `self.opt.step()`, que ejecuta la actualización efectiva de los parámetros según la regla del optimizador (en este caso, Adam). El valor retornado es la pérdida convertida a número flotante, útil para su monitorización o registro durante el entrenamiento.

Con esto completamos la parte central de la implementación que es el modelo como un módulo de PyTorch.

6.2. Formato en secuencias

Para que el modelo pueda entrenarse correctamente, es imprescindible preparar los datos en el formato adecuado. Los pasos de limpieza de datos y *feature engineering*, documentados para el modelo basado en una red neuronal simple son de utilidad aquí también, sin embargo el formato de entrada para el modelo Seq2Seq es muy distinto del formato tabular (*dataframe* en `pandas`) del que partimos.

En esta sección nos centraremos en un módulo de Python, `format_seqs.py`, que implementa justamente las funciones necesarias para transformar un *dataframe* de `pandas` en matrices `numpy` con las dimensiones apropiadas para alimentar el modelo. Dicho módulo se encarga de

recorrer los datos, organizarlos por particiones (en nuestro caso, una partición por paciente), mantener el orden temporal y finalmente devolver tensores listos para ser convertidos en `torch.Tensor`.

```python
from copy import copy
from functools import partial
import pandas as pd
import numpy as np
```

Listado 28: Importación de librerías necesarias para la manipulación de datos y funciones auxiliares.

El módulo comienza importando varias librerías estándar. La función `copy` del paquete `copy` se utiliza para clonar estructuras intermedias sin que las modificaciones posteriores alteren los objetos originales. El decorador `partial` de `functools` permite fijar ciertos parámetros de una función para luego aplicarla de manera más cómoda dentro de operaciones como `groupby`. Finalmente, `pandas` y `numpy` son las bibliotecas fundamentales para manipular series temporales en estructuras tabulares y transformarlas en arrays multidimensionales, respectivamente.

```python
def partitioned_series_to_sequences(
    df: pd.DataFrame, seq_len: int, features: list[str], order_key: str
):
    result = []
    buf = []
    for i, row in df.reset_index().sort_values(by=order_key).iterrows():
        if i < seq_len:
            buf.append(list(row[features]))
            continue
        result.append(copy(buf))
        buf.pop(0)
        buf.append(list(row[features]))
    if not result:
        return None
    return np.array(result)
```

Listado 29: Generación de secuencias deslizantes a partir de una partición ordenada de datos.

Esta función constituye el núcleo de la preparación de secuencias. Recibe un `DataFrame` que corresponde a una partición individual (todos los datos de un paciente), junto con una longitud de secuencia `seq_len`, la lista de características relevantes y una clave de orden temporal. El procedimiento recorre cada fila en orden, construyendo un búfer (`buf`) de tamaño fijo que se va desplazando como una ventana móvil sobre los datos. Cuando el búfer alcanza la longitud necesaria, se copia su contenido en la lista de resultados, y a continuación se elimina el primer elemento para dar paso al siguiente. De esta manera, cada fila del DataFrame contribuye a la construcción de múltiples secuencias consecutivas que capturan la dinámica temporal de las variables seleccionadas. Si la partición es demasiado corta y no se obtiene ninguna secuencia completa, se devuelve un valor nulo.

```python
def create_seqs(df: pd.DataFrame, partition_key: str, order_key: str, features : list[str], seq_len: int) -> np.array:
    return np.concat(
        list(
            df.groupby(partition_key)
            .apply(
                partial(
                    partitioned_series_to_sequences,
                    seq_len = seq_len,
                    features=features,
                    order_key=order_key,
                ),
                include_groups=False,
            )
            .dropna()
        )
    )
```

Listado 30: Construcción de secuencias a partir de todas las particiones del DataFrame.

Esta segunda función amplía el enfoque anterior. En lugar de procesar únicamente una partición, toma todo el `DataFrame` y lo divide en grupos definidos por `partition_key` (de manera general podría ser el identificador de un producto, usuario o sensor, en nuestro caso es la columna con el id del paciente). A cada grupo se le aplica la función `partitioned_series_to_sequences` mediante un partial, que ya tiene preconfigurados los argumentos `seq_len`, `features` y `order_key`. El resultado es una colección de secuencias extraídas de cada partición, que luego se concatenan en un único array de `numpy`. En este paso se unifican las distintas series individuales en un formato común, listo para ser usado en el entrenamiento del modelo.

```python
def format_data(
    df: pd.DataFrame, partition_key, order_key,
    encoder_input_features, decoder_input_features,
    output_features, input_seq_length, output_seq_length,
) -> tuple[tuple[np.array, np.array, np.array], list[int]]:
    features = list(set(
            encoder_input_features
            + decoder_input_features
            + output_features
        ))
    feature_index_map = {feature: i for i, feature in
enumerate(features)}
    encoder_input_features_idx = [
        feature_index_map[feature]
        for feature in encoder_input_features]
    decoder_input_features_idx = [
        feature_index_map[feature]
        for feature in decoder_input_features]
    output_features_idx = [
        feature_index_map[feature]
        for feature in output_features]
    seqs = create_seqs(
        df, partition_key, order_key, features=features,
        seq_len=input_seq_length + output_seq_length,
    )
    encoder_inputs = seqs[
        :, :input_seq_length, encoder_input_features_idx]
    decoder_inputs = seqs[
        :,
        input_seq_length - 1 : input_seq_length + output_seq_length - 1,
        decoder_input_features_idx]
    decoder_outputs = seqs[
        :,
        input_seq_length : input_seq_length + output_seq_length,
        output_features_idx]
    feature_index_map = {
        feature: i
        for i, feature in enumerate(decoder_input_features)
    }
    return (
        (encoder_inputs, decoder_inputs, decoder_outputs),
        [feature_index_map[feature] for feature in output_features],
    )
```

Listado 31: Preparación final de entradas y salidas para el modelo Seq2Seq.

La última función, `format_data`, completa la tarea de convertir los datos crudos en tensores bien estructurados. En primer lugar, combina todas las características utilizadas por el codificador, el decodificador y la salida en una única lista, y genera un mapa de índices que asocia cada nombre de característica con su posición en la matriz. Posteriormente, llama a

`create_seqs` para obtener secuencias que incluyen tanto la longitud de entrada como la de salida. Con esas secuencias construye tres subconjuntos:

- `encoder_inputs`: los valores de entrada que alimentan al codificador, extraídos según las características definidas.
- `decoder_inputs`: los valores de entrada para el decoder, que comienzan un paso antes de la salida y se extienden hasta cubrir toda la longitud de predicción.
- `decoder_outputs`: los valores reales que se quieren predecir, alineados en la ventana temporal correspondiente.

Finalmente, devuelve una tupla con estos tres arrays junto con un mapeo de índices para las características de salida. Con esto, los datos ya están listos para ser transformados en tensores de PyTorch y pasar directamente al entrenamiento del modelo Seq2Seq.

6.3. Otras funciones

Finalmente, antes de pasar a la ejecución del modelo y evaluar los resultados comentamos en esta sección un módulo que juega un papel más complementario pero igualmente imprescindible: `aux.py`.

Este módulo contiene funciones auxiliares para entrenar, evaluar y manejar los datos de un modelo Seq2Seq, Incluye generadores de lotes, cálculo de probabilidades de *teacher forcing*, selección del mejor modelo durante el entrenamiento y división de datos en conjunto de entrenamiento y validación. Todas estas utilidades simplifican la interacción con los tensores de Pytorch y los modelos que hemos definido.

python

```python
def batch_generator(data, batch_size):
    encoder_inputs, decoder_inputs, decoder_targets = data
    indices = torch.randperm(encoder_inputs.shape[0])
    for i in range(0, len(indices), batch_size):
        batch_indices = indices[i : i + batch_size]
        yield (
            encoder_inputs[batch_indices],
            decoder_inputs[batch_indices],
            decoder_targets[batch_indices],
            None,
        )
```

Listado 32: Generador de lotes aleatorios

```python
def train(model, data, batch_size, teacher_force_prob):
    model.train()
    epoch_loss = 0.
    num_batches = 0
        for (
          batch_enc_inputs,
          batch_dec_inputs,
          batch_dec_targets,
          _
        ) in batch_generator(data, batch_size):
        output = model(batch_enc_inputs, batch_dec_inputs,
  teacher_force_prob)
        loss = model.optimize(output, batch_dec_targets)
        epoch_loss += loss
        num_batches += 1
    return epoch_loss / num_batches
```

Listado 33: Entrenamiento de un modelo (una *epoch*).

Este método ejecuta una pasada de entrenamiento sobre todos los lotes, es decir sobre todo el conjunto de datos, lo que hemos definido como una *epoch*. Primero activa el modo entrenamiento (`model.train()`) para que PyTorch registre el grafo de cómputo y permita la retropropagación. Luego recorre los batches generados, calcula la salida del modelo, aplica la optimización y acumula la pérdida. Al final, devuelve la pérdida promedio por lote.

```python
def evaluate(model, val_data, batch_size):
    model.eval()
    epoch_loss = 0.
    num_batches = 0
    with torch.no_grad():
        for (
          batch_enc_inputs,
          batch_dec_inputs,
          batch_dec_targets,
          _
        ) in batch_generator(val_data, batch_size):
            output = model(batch_enc_inputs, batch_dec_inputs)
            loss = model.compute_loss(output, batch_dec_targets)
            epoch_loss += loss
            num_batches += 1
    return epoch_loss / num_batches
```

Listado 34: Evaluación de un modelo.

Esta función evalúa el modelo sobre un conjunto de validación. Se activa `model.eval()` para desactivar la acumulación de gradientes y regularizaciones como `Dropout`. `torch.no_grad()` evita construir el grafo de cómputo, reduciendo consumo de memoria. Se calcula la pérdida promedio sobre todos los lotes para monitorear desempeño.

```python
def calc_teacher_force_prob(decay, indx):
    return decay / (decay + math.exp(indx / decay))
```

Listado 35: Cálculo de probabilidad de *teacher forcing*.

Esta función determina la probabilidad de usar *teacher forcing* en la época `indx`. La probabilidad decrece según una función **sigmoide inversa** a medida que avanzan las épocas, controlando el equilibrio entre usar las salidas reales del modelo y los valores correctos como inputs del decoder. Es la función de decaimiento.

El parámetro `decay` controla la velocidad a la que esta probabilidad desciende. De manera quizás un poco contraintuitiva a mayor valor del parámetro `decay` más lentamente cae la probabilidad.

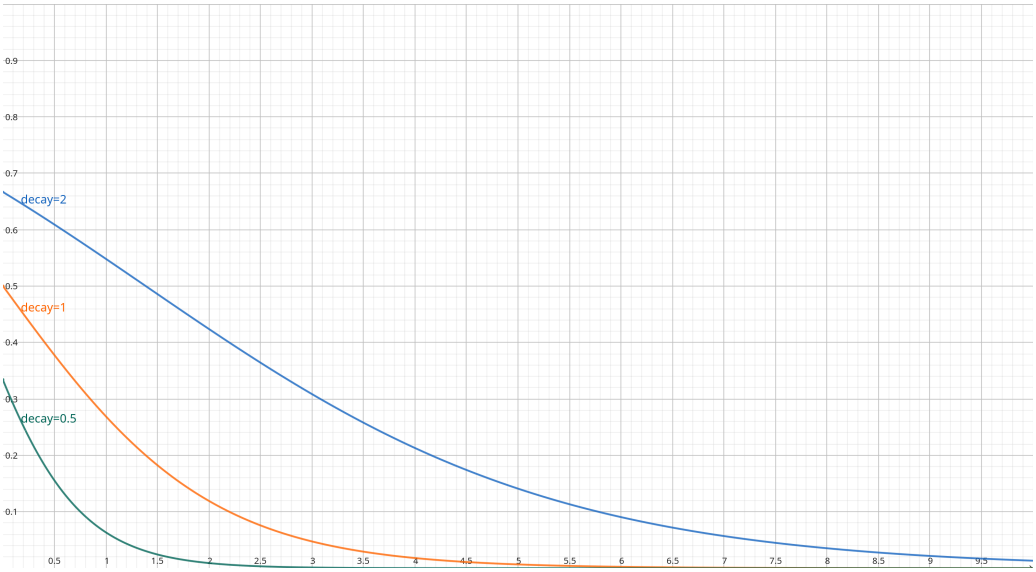

Figura 25: Curvas de la función de decaimiento para valores del parámetro `decay` 0.5, 1 y 2.

```python
def get_best_model(
    model, train_data, val_data,
    batch_size, num_epochs, decay
):
    best_val, best_model = float('inf'), None
    for epoch in range(num_epochs):
        start_t = time()
        teacher_force_prob = calc_teacher_force_prob(decay, epoch)
        train_loss = train(
          model, train_data, batch_size, teacher_force_prob
        )
        val_loss = evaluate(model, val_data, batch_size)
        new_best_val = False
        if val_loss < best_val:
            new_best_val = True
            best_val = val_loss
            best_model = deepcopy(model)
        print(
            f'Epoch {epoch+1} => Train loss: {train_loss:.5f},'
            f' Val: {val_loss:.5f},'
            f' Teach: {teacher_force_prob:.2f},'
            f' Took {(time() - start_t):.1f} s'
            f'{"      (NEW BEST)" if new_best_val else ""}'
        )
    return best_model
```

Listado 36: Selección del mejor modelo durante entrenamiento.

Esta función recorre `num_epochs` pasadas de entrenamiento, calculando pérdida de entrenamiento y validación en cada época. Si la pérdida de **validación** mejora, se hace un `deepcopy` del modelo, conservando la mejor versión. Esto permite **evitar sobreajuste** y seleccionar automáticamente el modelo más generalizable. La impresión muestra métricas por *epoch*, incluida la probabilidad de *teacher forcing*.

```python
def train_val_split(data, p=0.8):
    n = data[0].shape[0]
    indices = random.sample(list(range(n)), k=int(n * p))
    remaining = [i for i in range(n) if i not in indices]
    return (
        tuple(map(lambda arr: torch.Tensor(arr[indices]), data)),
        tuple(map(lambda arr: torch.Tensor(arr[remaining]), data)),
    )
```

Listado 37: División de datos en entrenamiento y validación.

Finalmente tenemos esta función auxiliar que divide un dataset en conjuntos de entrenamiento y validación según un porcentaje dado `p`. Se generan índices aleatorios para el conjunto de entrenamiento y el resto se asigna a validación. Cada parte se transforma en tensores de PyTorch, listos para ser usados por los métodos de entrenamiento y evaluación anteriores.

6.4. Entrenamiento y evaluación

Ya podemos pasar a el entrenamiento y evaluación del modelo. El código restante es sencillo, tratándose principalmente de la definición de los (hiper-)parámetros. La búsqueda de una configuración óptima de estos es un problema grande por si mismo. En esta sección, también compararemos varias configuraciones.

El código que comentaremos a continuación se incluye en un cuaderno de jupyter `model.ipynb`. En este cuaderno importamos los módulos que hemos explicado a lo largo de este capítulo. Como ya hemos mencionado parte de los pasos preliminares pueden ser reutilizados de lo aplcado para el modelo basado en una red neuronal simple, así que obviaremos esa parte.

python

```python
from format_seqs import format_data
from aux import train_val_split
encoder_input_features = [
    "visit_month",
    "updrs_1",
    "updrs_2",
    "updrs_3",
    "updrs_4",
]
decoder_input_features = [
    "visit_month",
    "updrs_1",
    "updrs_2",
    "updrs_3",
    "updrs_4",
]
output_features = decoder_input_features[1:]
> visit_month es la única covariable
data, target_indices = format_data(
    scaled_patient,
    partition_key="patient_id",
    order_key="visit_month",
    encoder_input_features=encoder_input_features,
    decoder_input_features=decoder_input_features,
    output_features=output_features,
    input_seq_length=3,
    output_seq_length=3,
)
train_data, val_data = train_val_split(data, p = 0.8)
> 80% de los datos para entrenar
```

Figura 26: Preparación y configuración de datos para entrenamiento del modelo seq2seq con características UPDRS.

En este fragmento de código se realiza la preparación y configuración de los datos necesarios para entrenar el modelo seq2seq. Primero se definen las características de entrada tanto para el codificador como para el decodificador, que incluyen el mes de visita (`visit_month`) como covariable temporal y las cuatro puntuaciones de la escala UPDRS (`updrs_1` a `updrs_4`) que

evalúan diferentes aspectos de los síntomas del Parkinson. Las características de salida se limitan únicamente a las puntuaciones UPDRS, excluyendo el mes de visita que actúa como variable de control temporal. Posteriormente, se utiliza la función `format_data` para transformar los datos en secuencias apropiadas para el modelo, organizando la información por paciente (`patient_id`) y ordenándola cronológicamente (`visit_month`), con secuencias de entrada y salida de 3 pasos temporales cada una. Finalmente, se divide el conjunto de datos en entrenamiento y validación, asignando el 80% de los datos para el entrenamiento del modelo y reservando el 20% restante para la evaluación de su rendimiento.

Probaremos primero de esta manera, sin incluir las columnas de péptidos y/o proteinas en las características de entrada. Posteriormente las añadiremos y contrastaremos los resultados

```python
from seq2seq import Encoder, Seq2Seq, DecoderWithAttention

enc_feature_size = len(encoder_input_features)
hidden_size = 32
num_gru_layers = 1
dropout = 0.1
dec_feature_size = len(decoder_input_features)
dec_target_size = len(output_features)
device = 'cpu'
lr = 0.0005
grad_clip = 1
batch_size = 100
num_epochs = 50
decay = 3 #Lower means faster decay

encoder = Encoder(enc_feature_size, hidden_size, num_gru_layers, dropout)
decoder_args = (dec_feature_size, dec_target_size, hidden_size,
num_gru_layers, target_indices, dropout, device)
decoder = DecoderWithAttention(*decoder_args)
seq2seq = Seq2Seq(encoder, decoder, lr, grad_clip).to(device)
```

Listado 38: Definición de hiperparámetros y construcción del modelo Seq2Seq

En este fragmento se definen los hiperparámetros principales para el entrenamiento del modelo Seq2Seq con atención, y se construyen las instancias de `Encoder`, `DecoderWithAttention` y `Seq2Seq`. Aquí es donde se fijan tanto las dimensiones de entrada y salida como la capacidad de la red (tamaño de estado oculto, número de capas, *dropout*, etc.), además de parámetros de entrenamiento como la tasa de aprendizaje, `grad_clip` o el número de *epochs*. Muchos de estos parámetros ya han ido aparenciendo pero aprovechamos este bloque, donde están todos juntos, para repasarlos al completo.

El primer paso es calcular el tamaño de entrada del codificador (`enc_feature_size`) y del decoficador (`dec_feature_size` y `dec_target_size`) a partir de las listas de características (*features*).

El hiperparámetro `hidden_size` establece la dimensión del vector oculto interno de la GRU, lo que controla la capacidad de representación del modelo: valores más altos pueden capturar relaciones más complejas, pero también incrementan el riesgo de sobreajuste.

El parámetro `num_gru_layers` indica cuántas capas GRU se utilizarán tanto en codificador como en decodificador. Si se aumentara, el modelo tendría más profundidad. `dropout` ayuda a evitar sobreajuste apagando aleatoriamente un porcentaje de las conexiones durante el entrenamiento.

A nivel de entrenamiento, la tasa de aprendizaje (`lr`) controla la magnitud de las actualizaciones de los parámetros. `grad_clip` se utiliza para evitar explosiones de gradiente, limitando la norma de los gradientes en cada paso de optimización.

Otros hiperparámetros definen aspectos prácticos: `batch_size` regula cuántas secuencias se procesan en paralelo, mientras que `num_epochs` elnumero de pasadas completas durante el entrenamiento al conjunto total de datos. Finalmente, `decay` controla la velocidad con que disminuye la probabilidad de *teacher forcing* (afecta a la funciónde decaimiento).

El codificador se construye pasando el tamaño de entrada, `hidden_size` (tamaño del estado oculto), número de capas GRU y `dropout`. Para el decodificador se prepara una tupla de argumentos (`decoder_args`) que incluyen tanto dimensiones de entrada y salida como la lista de índices de salida (`target_indices`), además de `dropout` y el `device` (dispositivo).

Por último, el modelo Seq2Seq se instancia combinando `encoder` y `decoder`, junto con la tasa de aprendizaje y el valor de gradiente máximo (`grad_clip`). El método `.to(device)` asegura que el modelo se coloque en el dispositivo adecuado (CPU o GPU).

6.4.1. Resultados

Lo único que queda es llamar a la función `get_best_model` con los parámetros adecuados para entrenar al modelo.

6.4.1.1. Modelo sin características de entrada adicionales

Probamos primero con un modelo que haga uso exclusivamente del mes de visita y los valores UPDRS para predecir UPDRS hasta 18 meses en el futuro. La evaluacion de la función de pérdida (SMAPE) con respecto el conjunto de validación es en este caso aproximadamente un **75%**.

Es conveniente mencionar que la selección del conjunto de validación es aleatoria y por lo tanto podemos obtener distintos resultados ejecutando el mismo cuaderno varias veces con los mismos parámetros, sin embargo los resultados rondan este valor.

Se trata de una mejora significativa con respecto el modelo de la red neuronal simple, concretamente se presenta una mejora de $30 - 40\%$ con respecto el modelo de comparación. Hacer uso de una ventana de contexto con valores pasados se demuestra productivo para afinar las predicciones y Seq2Seq nos ha ofrecido una arquitectura del modelo que ha permitido explotar esa información.

Se incluye también una función para crear gráficos similares a los usados para comprobar los resultados del modelo base, la lógica difiere a la hora de evaluar el modelo y por ello es una función a parte incluida en el módulo `plot.py`. Presentamos ahora algunos ejemplos del modelo aplicado a los datos de validación.

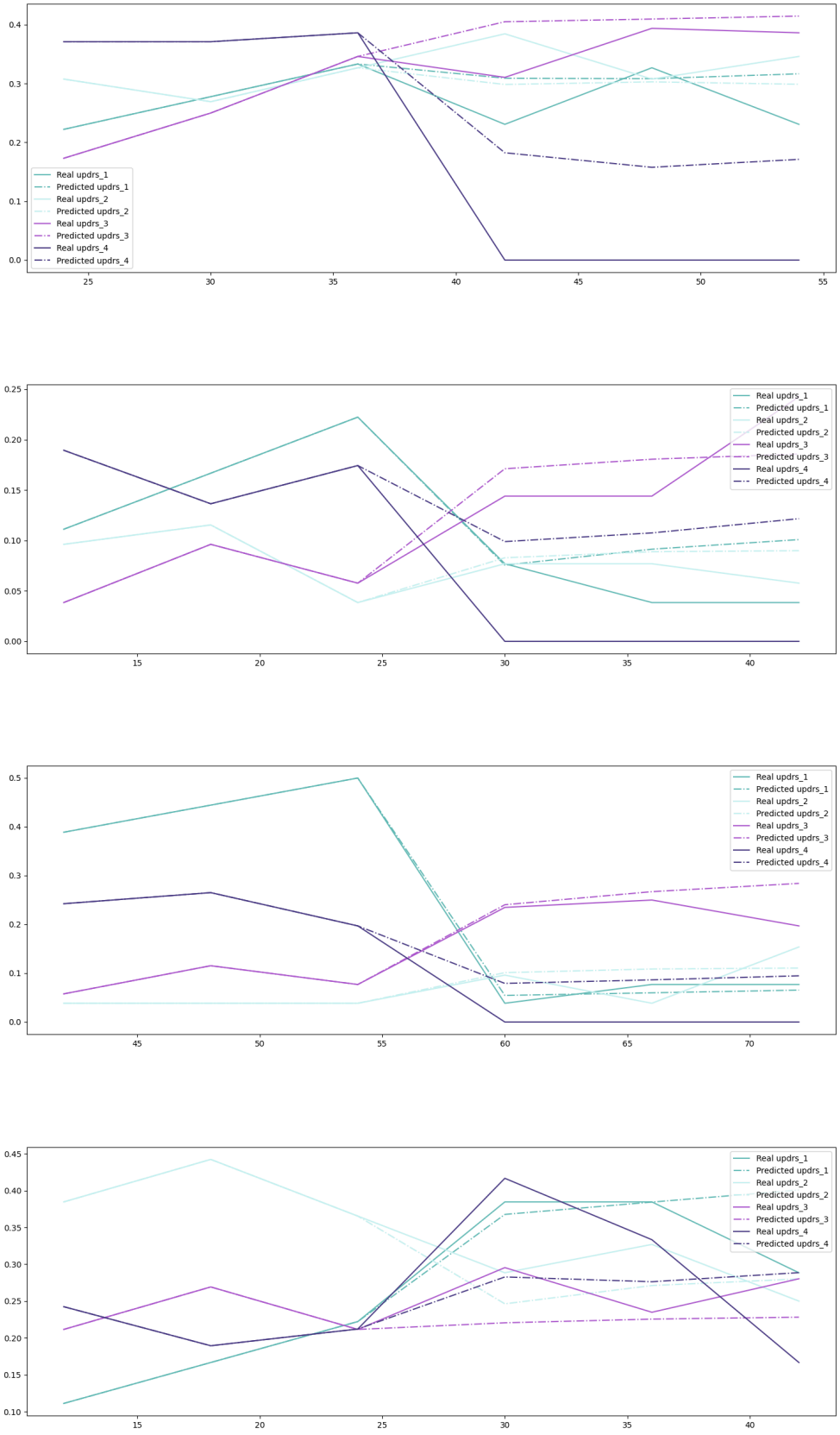

Figura 27: Gráficos comparativos de los datos reales con las predicciones del modelo.

La mejora de la métrica SMAPE supone una mejora tangible en la calidad de las predicciones como se puede ver en los ejemplos. El modelo consigue capturar, si bien de manera imperfecta (todavía estamos hablando de un error de 75%) las tendencias en las distintas categorías.

6.4.1.2. Modelo considerando los datos de proteínas / péptidos

Uno de los objetivos principales a la hora de abordar este proyecto era intentar extraer información significativa de los conjuntos de datos de péptidos y proteinas mediante el uso de Seq2Seq, donde otros enfoques habían fallado.

Como vimos para el modelo base, la inclusión de estos datos reducía la precisión del modelo. En este caso obsevamos que este efecto es mitigado pero concluimos que no es posible, tampoco de esta manera, extraer valor de estos datos que ayuden a mejorar las predicciones. El error para ambos casos, incluyendo datos de proteinas e incluyendo datos de péptidos, vuelve a rondar el **75%**.

6.4.1.3. Modelo con LSTM como red neuronal recurrente alternativa

Se ha implementado también una variante intercambiado las redes neuronales GRU por redes neuronales LSTM. Los cambios respecto a código necesarios para adaptar el modelo que usa GRU han sido mínimos ya que ambas exponen métodos y constructores muy similares en su implementacción en `pytorch`. El único detalle destacable es que las redes LSTM tienen como entrada y salida además del tensor principal y el oculto (*hidden*) un tercero llamado *cell state* que se inicializa a 0, luego hay que encadenar cuidadosamente estos tensores. Para el cálculo de la atención, reutilizamos el mismo módulo haciendo uso sólo de los tensores *hidden* al igual que para GRU.

La implementación finalmente resulta en variantes de los componentes de decodificador y codificador, reutilizando el resto de módulos. Se encuentra en el fichero `lstm.py`.

En cuánto a los resultados aquí otra vez obtenemos un error prácticamente equivalente, al rededor de **75%**.

En la siguiente tabla se resumen los desempeños de distintos intentos, cambiando parámetros, carácterísticas de entrada y red neuronal recurrente.

Entradas	Tipo RNR	*hidden*	Capas	*dropout*	*batch size*	*epochs*	*decay*	Media	Desviación estándar
BASE	GRU	512	2	0.2	32	50	4	72.05	1.41
BASE	GRU	32	1	0.2	32	50	4	71.52	1.90
PROTEIN	GRU	32	1	0.2	32	50	4	74.34	2.26
PROTEIN	GRU	256	1	0.2	32	50	4	79.34	4.35
PROTEIN	GRU	256	1	0.1	64	100	5	75.28	0.89
PROTEIN	LSTM	256	1	0.1	64	100	5	74.81	1.43
PROTEIN	LSTM	512	2	0.1	64	100	5	75.50	1.84
PEPTIDE	LSTM	512	2	0.1	64	100	5	77.11	3.46
PEPTIDE	LSTM	256	1	0.1	64	100	5	78.33	1.97
PEPTIDE	LSTM	32	1	0.1	64	100	5	77.66	2.16
PROTEIN	LSTM	32	1	0.1	64	100	5	74.93	0.83
PROTEIN	LSTM	64	1	0.1	64	100	5	76.63	1.74
BASE	LSTM	64	1	0.1	64	100	5	73.61	0.98
BASE	LSTM	32	4	0.1	64	100	5	75.98	1.94
BASE	LSTM	4	1	0.1	64	100	5	78.95	1.95
BASE	LSTM	8	1	0.1	64	100	5	76.58	0.63
BASE	LSTM	8	4	0.1	64	100	5	79.31	1.12
BASE	LSTM	16	4	0.1	64	100	5	76.70	1.67
BASE	LSTM	16	1	0.1	64	100	5	75.30	1.86

Tabla 12: Resultados de experimentos con distintas configuraciones RNN.

Los datos presentados se han obtenido repitiendo cinco veces cada configuración del modelo, con el fin de reflejar la variabilidad inherente al proceso de entrenamiento. Dicha variación proviene tanto de la forma en que se divide el conjunto en entrenamiento y validación, como de la sensibilidad del ajuste a la tasa de aprendizaje y otros parámetros internos. A partir de estas repeticiones, se calcularon la media y la desviación estándar de las medidas de validación, lo que permite tener una visión más estable del rendimiento sin depender de un único entrenamiento puntual

En los experimentos realizados, las variaciones en el tamaño oculto, número de capas y tasa de dropout no muestran un patrón claro de mejora o empeoramiento sostenido en el rendimiento. Si bien se observan pequeñas fluctuaciones en la media de validación al modificar estos parámetros, las diferencias se mantienen en rangos estrechos y no parecen ser estadísticamente relevantes dadas las desviaciones estándar asociadas.

Al comparar entre los distintos tipos de entrada (BASE, PROTEIN y PEPTIDE), los valores de validación se sitúan en intervalos similares, sin que un conjunto de características se distinga de manera consistente frente a los demás. Esto indica que, al menos con las configuraciones probadas, el tipo de entrada no introduce un efecto determinante en el rendimiento final del modelo.

La comparación entre arquitecturas con GRU y LSTM tampoco refleja una ventaja clara de una sobre otra. En ambos casos los resultados oscilan en márgenes similares y la variabilidad entre repeticiones es comparable. En conjunto, estos resultados sugieren que dentro del rango de hiperparámetros explorado, el modelo no es especialmente sensible a estas variaciones estructurales.

El límite en el rendimiento y la escasa variación frente a los parámetros parecen estar muy ligados al tamaño reducido del conjunto de datos. En primer lugar, al disponer de un volumen limitado de ejemplos, el modelo alcanza rápidamente toda la información disponible, de modo que aumentar el número de capas o el tamaño de las representaciones internas no aporta beneficios adicionales, ya que no hay suficientes patrones nuevos que extraer. En segundo lugar, este mismo tamaño reducido incrementa la proporción de ruido y redundancia en relación con la señal útil: con tan pocos datos, las modificaciones en los hiperparámetros apenas logran explotar información adicional y los resultados permanecen dentro de un rango muy estrecho, sin diferencias significativas.

Capítulo 7. Conclusiones

7.1. Resumen de resultados

El objetivo principal de este trabajo consistía en evaluar el potencial de los modelos Seq2Seq en la predicción de la progresión del Parkinson mediante la escala UPDRS, explorando si esta arquitectura podría superar las limitaciones de enfoques más tradicionales en el análisis de datos temporales biomédicos.

Los resultados obtenidos demuestran que el modelo Seq2Seq logró un rendimiento significativamente superior al modelo de referencia basado en redes neuronales simples. Específicamente, se alcanzó una mejora del 30-40% en la métrica SMAPE, reduciendo el error de validación hasta aproximadamente un 75%. Esta mejora sustancial confirma la capacidad del modelo para capturar dependencias temporales complejas en la progresión de la enfermedad que los modelos más simples no logran explotar.

El análisis de diferentes configuraciones reveló que el modelo base (utilizando únicamente mes de visita y valores UPDRS previos) proporcionó los mejores resultados, manteniéndose consistente tanto con arquitecturas GRU como LSTM. Sin embargo, los intentos de incorporar información adicional proveniente de datos proteómicos y peptídicos no resultaron en mejoras significativas del rendimiento, manteniendo el error en torno al mismo nivel del 75%.

Si bien estos resultados representan un avance conceptual importante y demuestran la validez del enfoque Seq2Seq para este tipo de problemas, el nivel de error obtenido aún no permite considerar el modelo como clínicamente aplicable de forma robusta. No obstante, la capacidad demostrada para capturar tendencias en las distintas categorías UPDRS sugiere que la arquitectura tiene potencial para futuras mejoras con datasets más amplios y completos.

7.2. Limitaciones

La limitación principal que condicionó el desarrollo y los resultados de este trabajo fue la cantidad limitada de datos disponibles. Con aproximadamente 2.600 visitas que incluían datos de proteínas y 5.800 con información UPDRS, el volumen de datos resultó insuficiente para explotar completamente el potencial de los modelos de deep learning, que típicamente requieren grandes cantidades de información para generalizar adecuadamente.

Un problema adicional significativo fue la alta proporción de valores faltantes, especialmente pronunciada en la categoría UPDRS IV, lo que limitó la capacidad del modelo para aprender patrones completos y consistentes. Esta fragmentación en los datos se vio agravada por la heterogeneidad en las mediciones y la escasez de seguimientos longitudinales consistentes para los mismos pacientes a lo largo del tiempo.

Como se detalla en los capítulos previos dedicados al análisis exploratorio y preprocesamiento de datos, estas limitaciones no son inherentes al modelo propuesto, sino que derivan directamente de las características y restricciones de la fuente de datos disponible. El conjunto de datos, pese a ser una iniciativa valios , presenta las limitaciones típicas de los estudios clínicos reales: alta variabilidad en el seguimiento de pacientes, protocolos de medición heterogéneos y dificultades en la recolección sistemática de datos longitudinales.

La tabla de resultados presentada evidencia que las variaciones en hiperparámetros (tamaño oculto, número de capas, tasa de dropout) no produjeron mejoras sustanciales, lo que sugiere que el modelo alcanzó rápidamente el límite de información extraíble del dataset disponible. Esta saturación temprana es característica de escenarios con datos limitados, donde aumentar la complejidad del modelo no aporta beneficios adicionales.

7.3. Futuras líneas de trabajo

7.3.1. Expansión del conjunto de datos

La ampliación significativa del dataset representa la línea de trabajo más crítica para mejorar sustancialmente el rendimiento del modelo. Esta expansión podría abordarse desde múltiples frentes: La integración de datos abiertos anonimizados (Open Data) procedentes de otras iniciativas de investigación similares permitiría aumentar considerablemente el volumen de observaciones longitudinales disponibles

El desarrollo de una interfaz web colaborativa constituye otra vía prometedora para la recolección sistemática de datos clínicos y demográficos. Esta plataforma podría facilitar la contribución de centros médicos especializados, permitiendo la estandarización de protocolos de medición y el seguimiento más consistente de cohortes de pacientes.

La inclusión de variables adicionales como edad, sexo, hábitos de vida, comorbilidades y factores socioeconómicos podría enriquecer significativamente el poder predictivo del modelo. Es evidente que estos factores contextuales tienen una influencia relevante en la progresión de enfermedades neurodegenerativas.

7.3.2. Optimización de hiperparámetros

El enfoque utilizado en este trabajo para la selección de hiperparámetros se basó principalmente en prueba y error guiada por conocimiento del dominio. Futuras iteraciones deberían incorporar métodos automáticos de búsqueda de hiperparámetros que permitan explorar de forma más sistemática y eficiente el espacio de configuraciones posibles. Técnicas como optimización bayesiana, búsqueda en malla (grid search) o búsqueda aleatoria (random search) podrían identificar configuraciones más óptimas que las encontradas manualmente. Adicionalmente, el uso de plataformas de **AutoML** podría automatizar no solo la búsqueda de hiperparámetros sino también la selección de arquitecturas de modelo más apropiadas para este dominio específico.

7.3.3. Extensión del marco experimental

El *transfer learning* con modelos preentrenados representa una oportunidad particularmente relevante. Modelos entrenados en datasets médicos más amplios o en tareas relacionadas de análisis de series temporales podrían transferir conocimientos útiles que compensen la limitación de datos específicos del Parkinson.

La exploración de arquitecturas híbridas que combinen efectivamente datos clínicos, proteómicos y demográficos mediante diferentes modalidades de entrada podría superar las limitaciones observadas en este trabajo. Técnicas de fusión multimodal tardía o temprana podrían permitir que cada tipo de información contribuya de forma más efectiva al modelo final. Con multimodalidad nos referimos aquí, a integrar datos numéricos con, por ejemplo, imágenes de escáneres cerebrales.

7.4. Valor del aprendizaje realizado

Desde la perspectiva académica, este proyecto representa una aplicación integral de técnicas avanzadas de Deep Learning a un problema real de impacto social significativo, cumpliendo plenamente los objetivos formativos de un Trabajo de Fin de Grado. El desarrollo completo del ciclo de un proyecto de Machine Learning, desde el análisis exploratorio inicial hasta la evaluación final de resultados, ha permitido adquirir experiencia práctica en cada una de las etapas críticas: preprocesamiento de datos complejos, diseño de arquitecturas de modelo, implementación técnica y análisis crítico de resultados.

Más allá de los aspectos técnicos específicos, este trabajo posee un impacto conceptual relevante dentro del contexto actual del desarrollo de la inteligencia artificial. El modelo Seq2Seq utilizado constituye una de las arquitecturas precursoras directas de los **Transformers**, que han revolucionado el campo y dado lugar al auge actual de los **Grandes Modelos de Lenguaje** (*LLM*). Haber trabajado en profundidad con mecanismos de atención, codificadores-decodificadores y el procesamiento de secuencias temporales proporciona una base sólida para comprender arquitecturas modernas como BERT, GPT o T5, que dominan el panorama actual de la inteligencia artificial.

La experiencia adquirida trasciende los resultados numéricos específicos obtenidos. El proceso de enfrentarse a las limitaciones reales de los datos, la necesidad de tomar decisiones metodológicas fundamentadas ante la incertidumbre, y la interpretación crítica de resultados en un contexto biomédico, representan aprendizajes fundamentales para el desarrollo profesional en el campo del Machine Learning aplicado.

7.5. Conclusión final

Este trabajo demuestra que los modelos Seq2Seq poseen potencial significativo para la predicción de la progresión del Parkinson, logrando mejoras sustanciales respecto a enfoques más simples. Aunque las limitaciones del dataset impidieron alcanzar niveles de precisión clínicamente aplicables los fundamentos metodológicos desarrollados y las líneas de trabajo futuras identificadas proporcionan una base sólida para investigaciones posteriores que, con mayores volúmenes de datos, podrían contribuir efectivamente al diagnóstico y seguimiento clínico de esta enfermedad neurodegenerativa.

Bibliografía

[1] Accelerating Medicines Partnership Parkinson’s Disease (AMP PD) Consortium and Kaggle, «AMP Parkinson's Disease Progression Prediction». [En línea]. Disponible en: https://www.kaggle.com/c/amp-parkinsons-disease-progression-prediction

[2] S. Raschka, J. Patterson, y C. Nolet, «Machine learning in python: Main developments and technology trends in data science, machine learning, and artificial intelligence», *Information*, vol. 11, n.º 4, p. 193, 2020.

[3] R. Wirth y J. Hipp, «CRISP-DM: Towards a standard process model for data mining», en *Proceedings of the 4th international conference on the practical applications of knowledge discovery and data mining*, 2000, pp. 29-39.

[4] F. Mart\inez-Plumed *et al.*, «CRISP-DM twenty years later: From data mining processes to data science trajectories», *IEEE transactions on knowledge and data engineering*, vol. 33, n.º 8, pp. 3048-3061, 2019.

[5] Wikimedia Commons, «CRISP-DM Process Diagram».

[6] Glassdoor, «Junior Data Scientist Sueldos en España». [En línea]. Disponible en: https://www.glassdoor.es/Sueldos/junior-data-scientist-sueldo-SRCH_KO0,21.htm

[7] C. G. Goetz *et al.*, «Movement Disorder Society-sponsored revision of the Unified Parkinson's Disease Rating Scale (MDS-UPDRS): Process, format, and clinimetric testing plan», *Movement Disorders*, vol. 22, n.º 1, pp. 41-47, 2007, doi: https://doi.org/10.1002/mds.21198.

[8] X. Wang, Z. Cai, Y. Luo, y others, «Long Time Series Deep Forecasting with Multiscale Feature Extraction and Seq2seq Attention Mechanism», *Neural Processing Letters*, vol. 54, pp. 3443-3466, 2022, doi: 10.1007/s11063-022-10774-0.

[9] IBM, «Recurrent Neural Networks (RNNs)». [En línea]. Disponible en: https://www.ibm.com/think/topics/recurrent-neural-networks

[10] M. Brenner, «Implementing Seq2Seq Models for Efficient Time Series Forecasting». 2023.

[11] S. Bengio, O. Vinyals, N. Jaitly, y N. Shazeer, «Scheduled Sampling for Sequence Prediction with Recurrent Neural Networks», en *Advances in Neural Information Processing Systems (NeurIPS)*, 2015, pp. 1171-1179.