Digit-Recognition

Joaquín Cotrina Santos
Juan Miguel Fernández Tejada
Francisco Javier Molina Cuenca
Alberto Ramírez Collado

Diciembre de 2025

Índice

---

1. La estructura general

El proyecto *Digit-Recognition* está organizado como un pipeline completo de *machine learning* para el reconocimiento de dígitos manuscritos. La estructura combina varios scripts en R, ficheros de datos y carpetas auxiliares, de forma que todo el flujo —desde la carga de los datos hasta el guardado del modelo final— pueda ejecutarse de manera automática y reproducible.

Estructura de carpetas y archivos principales

A nivel de proyecto se utilizan, principalmente, las siguientes carpetas lógicas:

- **data/**: contiene los datos en bruto (raw/, por ejemplo train.csv) y los datos procesados (processed/, con los ficheros *.rds generados por los scripts).
- **models/**: almacena los modelos entrenados en formato .rds (modelos base, modelos ajustados, meta-modelos y modelo final reducido).
- **results/**: guarda resultados intermedios como gráficos (pixel_histogram.png) y ficheros de evaluación (evaluation.txt).
- **scripts/**: contiene todos los scripts de R que conforman el pipeline: 1_data_prep.R, 2_eda.R, ..., 8_save_model.R, además de utils.R y el script maestro run_all.R.

Esta organización facilita que cada paso del pipeline lea sus entradas y escriba sus salidas en lugares bien definidos, lo que permite reanudar o repetir partes concretas del proceso sin tener que ejecutar necesariamente todo desde cero.

1.1. El script maestro run_all.R

El archivo run_all.R actúa como **script orquestador** de toda la práctica. Su función principal es preparar el entorno de trabajo y ejecutar, en el orden adecuado, el resto de scripts que implementan las diferentes etapas del pipeline.

En primer lugar, define un vector pkgs con todas las librerías que se consideran necesarias para el proyecto (tidyverse, caret, randomForest, e1071, xgboost, data.table, here) y llama a install.packages(pkgs) para asegurarse de que están disponibles en el entorno. Aunque en un entorno de producción sería preferible comprobar una a una cuáles faltan, esta llamada simplifica la instalación en máquinas nuevas.

A continuación, comprueba la disponibilidad de la librería here. Si no está instalada, la instala y, en cualquier caso, la carga. La librería here es clave porque proporciona rutas relativas robustas al directorio raíz del proyecto, evitando errores derivados del directorio de trabajo actual.

Después, el script imprime por consola información de arranque (incluyendo el directorio base detectado por here()) y crea, si no existen, las carpetas fundamentales del proyecto: data, models, results y scripts. Este paso garantiza que el resto de scripts pueda leer y escribir en dichas carpetas sin errores de directorio inexistente.

1.1.1. Función auxiliar run_script

El núcleo de la orquestación lo constituye la función auxiliar run_script(file). Dado el nombre de un script (por ejemplo, "1_data_prep.R"), esta función:

1. Construye la ruta completa con here("scripts", file).
2. Imprime un encabezado en consola indicando qué script se va a ejecutar.
3. Registra la hora de inicio con Sys.time().
4. Intenta ejecutar el script mediante source(path) dentro de un bloque tryCatch.
5. Si la ejecución es correcta, calcula el tiempo transcurrido y lo imprime.
6. Si se produce un error, muestra el mensaje de error, marca el script como fallido y detiene la ejecución global con un stop, de forma que no se continúe el pipeline en un estado inconsistente.

Este diseño proporciona **robustez** (no se ignoran errores) y **trazabilidad** (se sabe qué script falló y cuánto tardó cada uno en ejecutarse).

1.1.2. Ejecución secuencial de los scripts

El vector `scripts_to_run` define el orden en que se van a ejecutar los scripts del proyecto:

`1_data_prep.R`, `2_eda.R`, `3_feature_engineering.R`, `4_models.R`, `5_tunning.R`, `6_ensemble.R`, `7_evaluate.R`, `8_save_model.R`.

Las entradas correspondientes al análisis exploratorio (`2_eda.R`) y a la ingeniería de características (`3_feature_engineering.R`) aparecen comentadas como "opcionales" en el propio código, de manera que el usuario puede decidir si quiere ejecutarlas en cada corrida simplemente comentando o descomentando las líneas correspondientes del vector.

Finalmente, se recorre dicho vector en un bucle y se llama a `run_script` para cada elemento. Si todos los scripts se ejecutan sin errores, el script maestro imprime un mensaje de éxito indicando que la práctica completa se ha ejecutado correctamente.

2. Digit-Recognition

En esta sección se describe, con más detalle, el papel concreto de cada uno de los scripts que forman el pipeline de reconocimiento de dígitos. El flujo lógico sigue el orden en el que son llamados por `run_all.R`: desde la preparación de datos y análisis exploratorio, pasando por la ingeniería de características y el entrenamiento de modelos base, hasta la construcción del ensemble, la evaluación final y el guardado del modelo reducido. El script `utils.R` proporciona, en todo momento, funciones auxiliares reutilizables que simplifican el resto de código.

2.1. data_prep.R

El script `data_prep.R` se encarga de la preparación y partición de los datos a partir de un archivo CSV con imágenes y etiquetas. Comienza cargando las librerías necesarias y definiendo una semilla de aleatoriedad para asegurar la reproducibilidad de todo el proceso. A continuación, lee el archivo `train.csv` desde una ruta controlada por el proyecto y verifica que tenga exactamente 785 columnas, lo que garantiza que el formato de los datos es el esperado antes de continuar.

Después, la primera columna se renombra como label y se transforma en un factor con niveles no numéricos para evitar problemas en los modelos de aprendizaje automático. Las columnas correspondientes a los píxeles se identifican automáticamente, se escalan dividiendo sus valores entre 255 para normalizarlos al intervalo [0,1], y se recombinan con la etiqueta en un nuevo conjunto de datos ya procesado.

Para evitar un consumo excesivo de memoria, el script incluye un paso de seguridad que selecciona aleatoriamente una muestra de 20 000 observaciones, usando una semilla fija para que esta selección sea reproducible. Esta reducción permite trabajar con mayor eficiencia sin alterar la lógica del pipeline de procesamiento.

Finalmente, el conjunto reducido se divide en un 80 % para entrenamiento y un 20 % para validación mediante una partición estratificada que conserva la proporción de clases. Ambos subconjuntos se guardan en formato .rds en la carpeta de datos procesados, y se muestran sus dimensiones en consola como verificación final del proceso.

2.2. eda.R

El script `2_eda.R` está dedicado al **análisis exploratorio de datos (EDA)** sobre el conjunto de entrenamiento ya procesado. Su finalidad es obtener una visión intuitiva de la distribución de las

clases, el aspecto visual de los dígitos y ciertas propiedades estructurales de los píxeles. Tras cargar las librerías necesarias y el archivo `train_processed.rds`, se imprime el número de observaciones cargadas para verificar que la reducción previa del dataset se ha aplicado correctamente.

En primer lugar, se calcula la **distribución de clases** agrupando las observaciones por etiqueta y contando el número de ejemplos por dígito. Este resumen permite detectar de forma inmediata posibles desequilibrios entre clases. A continuación, el script genera una **visualización representativa** de los diez dígitos, mostrando un ejemplo por clase en una cuadrícula de 2 filas por 5 columnas. Para cada etiqueta se selecciona una imagen, se reconstruye su matriz de 28×28 píxeles y se representa gráficamente, mostrando un título con la clase correspondiente.

El análisis continúa con el estudio de la **sparsity de los píxeles**. Para cada imagen se calcula el porcentaje de píxeles activos (valores mayores que cero) y posteriormente se obtiene la media de este porcentaje por clase. Este indicador proporciona información sobre cuán densas o dispersas son, en promedio, las representaciones de cada dígito.

Finalmente, se construye un **histograma del número de píxeles activos por imagen**, utilizando 50 intervalos, lo que permite analizar la distribución global de la información contenida en las imágenes. Este gráfico se guarda en `results/pixel_histogram.png` para su uso posterior en informes o memorias sin necesidad de volver a ejecutar el script.

2.3. feature_engineering.R

El archivo `3_feature_engineering.R` se ocupa de la **ingeniería de características**. A partir de los datos ya procesados y particionados, realiza dos operaciones principales: selección de píxeles por varianza y reducción de dimensionalidad mediante Análisis de Componentes Principales (PCA).

En primer lugar, se cargan las librerías `tidyverse` y `caret`, junto con las funciones auxiliares de `utils.R`. Con `set_seed(42)` se establece una semilla reproducible. Se leen los ficheros `train_processed.rds` y `valid_processed.rds`, que contienen la etiqueta `label` y las columnas de píxeles escalados.

La primera fase de ingeniería de características es una **selección por varianza**. Se identifican las columnas de píxeles y se calcula la varianza de cada una sobre el conjunto de entrenamiento. A continuación, se seleccionan sólo aquellas columnas cuya varianza supera un umbral mínimo (por ejemplo, 10^{-6}). Los píxeles con varianza prácticamente nula (siempre o casi siempre cero) aportan poca información discriminativa y generan dimensiones redundantes que pueden aumentar el coste computacional sin mejorar el rendimiento del modelo. El resultado es un subconjunto reducido de píxeles *informativos*.

Con estos píxeles seleccionados se construyen dos nuevos data frames: `train_var` y `valid_var`, que contienen únicamente la etiqueta y las columnas de píxeles con varianza suficiente. Ambos se guardan en `data/processed` para permitir su reutilización posterior.

La segunda fase aplica **PCA** a las variables de píxeles previamente seleccionadas. Mediante `preProcess` de `caret`, se construye un objeto de preprocesado que realiza, en este orden, centrado, escalado y PCA sobre las columnas de píxeles. El parámetro `thresh = 0.95` indica que se deben conservar suficientes componentes principales como para explicar el 95 % de la varianza total. Este compromiso permite reducir drásticamente la dimensionalidad (pasar de cientos de píxeles a unas decenas de componentes) preservando la mayor parte de la información relevante.

Una vez ajustado el objeto de preprocesado sobre el conjunto de entrenamiento, se emplea `predict` para transformar tanto `train_var` como `valid_var` al nuevo espacio de componentes principales. Se obtienen así `train_pca` y `valid_pca`, que constan de la etiqueta y de un conjunto compacto de variables continuas (las componentes principales). Por último, se guarda el objeto de control de PCA (`pca_ctrl.rds`) y los datasets transformados (`train_pca.rds` y `valid_pca.rds`), que se utilizarán para entrenar y evaluar los modelos de clasificación.

2.4. models.R

El script `4_models.R` aborda el **entrenamiento de los modelos base** sobre los datos ya transformados mediante PCA. El objetivo es disponer de varios clasificadores con estrategias distintas, que luego podrán combinarse en un ensemble más robusto.

Tras cargar las librerías `tidyverse`, `caret`, `randomForest`, `nnet`, `e1071` y `xgboost`, además de las funciones de `utils.R`, se fija la semilla con `set_seed(42)`. Se leen los ficheros `train_pca.rds` y `valid_pca.rds`.

Se define un objeto de control de entrenamiento `trctrl` mediante `trainControl(method = "none", classProbs = TRUE)`. Con `method = "none"` se indica que no se realizará re-muestreo interno (no hay validación cruzada ni bootstrap en esta fase); el objetivo es entrenar modelos concretos, no ajustar hiperparámetros. La opción `classProbs = TRUE` obliga a que los modelos soporten la generación de probabilidades de clase, lo cual es imprescindible más adelante, en el ensemble.

A continuación se entrenan tres modelos diferentes:

- **Random Forest**: Se utiliza la función `randomForest` directamente, pasando como variables de entrada todas las columnas excepto `label` y como respuesta el factor de etiquetas. El número de árboles se fija inicialmente en 50, suficiente para pruebas rápidas y desarrollo; en un entorno de producción se podría aumentar este valor para mejorar la estabilidad y precisión del modelo.
- **Modelo multinomial**: Mediante `caret::train` con `method = "multinom"`, se ajusta un modelo de regresión logística multinomial. La fórmula `label~.` indica que la respuesta es `label` y que todas las demás columnas se utilizan como predictores. El parámetro `MaxNWts = 10000` evita errores relacionados con un número excesivamente alto de pesos en el modelo, algo habitual cuando se manejan muchas variables.
- **Red neuronal simple**: Se entrena también un modelo `multinom` directamente desde el paquete `nnet`, con `maxit = 100` iteraciones y `trace = FALSE` para no imprimir el proceso de optimización. De nuevo, se controla el número máximo de pesos con `MaxNWts = 10000` para prevenir errores.

Una vez entrenados los tres modelos, se agrupan en una lista (`rf`, `glmnet`, `nnet`) y se guardan en `models/models_base.rds`. Este archivo sirve como repositorio de los modelos básicos, que pueden utilizarse tanto para comparaciones como para su inclusión en combinaciones más sofisticadas.

2.5. tunning.R

El archivo `5_tunning.R` (tuning de hiperparámetros) se encarga de ajustar con mayor finura dos modelos clave del pipeline: un Random Forest y una SVM lineal. A diferencia del script anterior, aquí sí se utiliza **validación cruzada** y búsqueda de hiperparámetros.

Se cargan las librerías `caret`, `rpart`, `randomForest`, `e1071`, `nnet` y el script de utilidades. Se fija la semilla aleatoria con `set.seed(123)` y se lee el dataset `train_pca.rds`. Para reducir el coste computacional del tuning, se crea una muestra estratificada del 10 % de las observaciones mediante `createDataPartition`. Esta submuestra, `train_int`, contiene ejemplos representativos de todas las clases, pero en menor número.

El control de entrenamiento se define ahora con `trainControl(method = "cv", number = 3, classProbs = TRUE)`. Es decir, se emplea validación cruzada de 3 folds y se exige que los modelos proporcionen probabilidades de clase, requisito indispensable para el posterior ensemble.

El tuning del Random Forest se realiza con `method = "rf"` y una rejilla de valores para el hiperparámetro `mtry` (por ejemplo, 2 y 4). Para cada valor de `mtry`, `caret` entrena un modelo y evalúa su rendimiento mediante validación cruzada. El mejor modelo según la métrica por defecto (por lo general, precisión) se guarda como `model_rf_tuned.rds`.

En paralelo, se ajusta una SVM lineal con `method = "svmLinear"` y `tuneLength = 2`, lo que indica a `caret` que explore un pequeño conjunto de valores posibles del parámetro de coste. El modelo resultante se guarda en `model_svm_tuned.rds`. Durante todo este proceso, la opción `classProbs = TRUE` garantiza que las predicciones de ambos modelos incluyan probabilidades de clase, algo esencial para usar sus salidas como meta-variables en el ensemble.

2.6. ensemble.R

El script `6_ensemble.R` implementa un esquema de **ensemble mediante stacking**. La idea central es combinar las predicciones de varios modelos base (en este caso, el Random Forest y la SVM ajustados) a través de un meta-modelo que aprende a ponderar dichas predicciones.

Después de cargar las librerías `caret` y `nnet`, y de fijar la semilla con `set.seed(123)`, se lee el dataset `train_pca.rds` y los modelos ajustados `model_rf_tuned` y `model_svm_tuned`.

Se selecciona, mediante `createDataPartition`, un 20 % del dataset para la fase de entrenamiento del meta-modelo. Este subconjunto `train_meta` se utiliza únicamente para aprender la combinación óptima de las salidas de los modelos base, sin mezclarlo con los datos empleados para su entrenamiento inicial.

Sobre `train_meta`, se generan las **predicciones probabilísticas** de ambos modelos base. Cada predicción es un data frame en el que cada columna corresponde a una clase (`L0`, `L1`, etc.) y cada fila a una observación, con valores entre 0 y 1 que suman 1 por fila. Se combinan ambas predicciones mediante `bind_cols`, dando lugar a un conjunto de meta-características que representan, por así decirlo, "lo que piensan" los modelos base sobre cada observación.

Para evitar conflictos con nombres de columnas duplicados o no válidos, se aplica `make.names` con `unique = TRUE` sobre el vector de nombres, de modo que cada columna reciba un identificador sintácticamente correcto y único. Se añade después la columna `label` con la verdadera clase de cada observación de `train_meta`.

Antes de entrenar el meta-modelo, se aplica `na.omit` para eliminar posibles filas con valores faltantes (por ejemplo, si alguna predicción falló puntualmente). Si, tras esta limpieza, el data frame resultante quedara vacío, el script lanza un error crítico solicitando revisar el script de tuning, ya que el ensemble no podría entrenarse.

El meta-modelo escogido es de nuevo una regresión logística multinomial (`nnet::multinom`), entrenada con `label` como respuesta y todas las probabilidades como predictores. Este modelo aprende a combinar la evidencia de ambos clasificadores para producir una predicción final potencialmente más robusta que la de cualquiera de ellos por separado. Se fija un límite de pesos (`MaxNWts = 5000`) y se desactiva la impresión del proceso de entrenamiento (`trace = FALSE`). Finalmente, el meta-modelo se guarda en `models/meta_model.rds`.

2.7. evaluate.R

El script `7_evaluate.R` se encarga de la **evaluación cuantitativa** del ensemble construido en el paso anterior, utilizando el conjunto de validación PCA.

Tras cargar `caret`, `nnet` y las funciones auxiliares, se leen el dataset `valid_pca.rds` y los tres modelos implicados en el ensemble: el Random Forest ajustado, la SVM ajustada y el meta-modelo (`meta_model.rds`).

Siguiendo el mismo esquema que en `6_ensemble.R`, se generan primero las predicciones probabilísticas de los modelos base sobre el conjunto de validación: `pred_rf` y `pred_svm`. Estas se concatenan para formar `meta_valid`, sobre el que se vuelve a aplicar `make.names` para asegurar nombres de columnas válidos.

A continuación, se usan estas meta-características como entrada del meta-modelo, que produce la **predicción final de clase** para cada observación de validación. En lugar de obtener probabilidades, aquí se solicita directamente el tipo `"class"`, es decir, la etiqueta más probable.

Con estas predicciones y las etiquetas verdaderas `valid$label`, se construye una matriz de confusión mediante `confusionMatrix`. Esta matriz resume el número de aciertos y errores por clase, y permite calcular diversas métricas de rendimiento. El script extrae e imprime específicamente la precisión global (`Accuracy`), ofreciendo una medida compacta del rendimiento del ensemble.

Por último, se redirige la salida completa de la matriz de confusión y otras estadísticas al archivo `results/evaluation.txt` utilizando `sink`. De este modo, los resultados quedan documentados y se pueden consultar o adjuntar a informes sin necesidad de volver a ejecutar el script.

2.8. save_model.R

El archivo `8_save_model.R` responde a un requisito específico de la práctica: **entrenar y guardar un modelo final utilizando únicamente los primeros 100 dígitos** del conjunto de entrenamiento transformado. La estructura del modelo sigue siendo la de un ensemble por stacking, pero construido sobre un conjunto de entrenamiento extremadamente reducido.

El script comienza cargando las librerías `caret`, `nnet`, `tidyverse` y las utilidades, e imprime por consola que está cargando datos y modelos para el entrenamiento final. Se lee el dataset `train_pca.rds` y se cargan los modelos ajustados `model_rf_tuned` y `model_svm_tuned`.

Se seleccionan entonces las primeras 100 observaciones del conjunto de entrenamiento PCA:

```
train_small <- train[1:100, ]
```

Sobre este subconjunto reducido se calculan las probabilidades de ambos modelos base, obteniéndose `pred_rf` y `pred_svm`. Al igual que en el ensemble completo, estas probabilidades se concatenan en un data frame `meta_small`, cuyos nombres de columnas se normalizan con `make.names`, y se añade la columna `label` con las etiquetas verdaderas de estas 100 observaciones.

Con estas meta-características se entrena un **meta-modelo reducido** (`meta_model_small`), de nuevo mediante `nnet::multinom` y con parámetros similares a los del ensemble principal (`trace = FALSE`, `MaxNWts = 5000`). Este modelo aprende a combinar las salidas del Random Forest y de la SVM, pero sólo a partir de los primeros 100 casos disponibles.

Finalmente, el modelo reducido se guarda en `models/final_model_100.rds`. El script imprime un mensaje de éxito indicando la ruta exacta de salida, certificando que se ha cumplido el requisito de la práctica de utilizar únicamente los primeros 100 dígitos para el modelo final entregable.

2.9. utils.R

El archivo `utils.R` agrupa diversas **funciones auxiliares** que se reutilizan en varios scripts, mejorando la modularidad y reduciendo la duplicación de código.

En primer lugar, la función `set_seed(s = 123)` encapsula la llamada a `set.seed`, proporcionando una forma uniforme de establecer la semilla aleatoria en todo el proyecto. Esto favorece la reproducibilidad: basta con llamar a `set_seed(42)`, por ejemplo, para garantizar que todas las operaciones que dependan del generador aleatorio produzcan resultados deterministas.

La función `load_csv(path)` sirve como envoltorio de `readr::read_csv`, utilizando especificación explícita de tipos (`col_types = cols()`) para leer archivos CSV con cabeceras conocidas, como los típicos `train.csv` usados en competiciones tipo Kaggle. Aunque en la práctica de Digit-Recognition el CSV se lee principalmente con `read.csv`, esta función está disponible para usos alternativos.

Una de las funciones más utilizadas es `plot_mnist_row(row_vec, title = NULL)`, que recibe un vector numérico de longitud 784 o una fila de `data.frame` y reconstruye la imagen de 28×28 píxeles correspondiente. Internamente, organiza el vector en una matriz de 28 filas, aplica una rotación para que la imagen se vea con la orientación correcta y llama a `image` sin ejes, añadiendo un título opcional. Esta función es la responsable de las visualizaciones de dígitos en el análisis exploratorio.

Las funciones `save_model(model, filename = "final_model.rds")` y `load_model(filename)` son envoltorios simples para `saveRDS` y `readRDS`, respectivamente. Su objetivo es estandarizar el modo en que se guardan y cargan modelos en disco, evitando tener que recordar los detalles de estas funciones en cada script.

Por último, `pixels_to_matrix(df_pixels)` convierte un `data.frame` con columnas de píxeles (típicamente `pixel0` a `pixel783`) en una matriz numérica y normaliza sus valores dividiendo por 255. Esta representación matricial es especialmente útil si se quiere trabajar directamente con las imágenes como matrices (por ejemplo, para operaciones de filtrado, convoluciones sencillas o visualizaciones distintas a las usadas en el EDA).

En conjunto, `utils.R` actúa como una pequeña biblioteca de funciones de soporte que simplifican la escritura del resto de scripts y favorecen que el código del pipeline principal se centre en la lógica de alto nivel (preparación de datos, entrenamiento de modelos, etc.) más que en detalles repetitivos de implementación.

3. Un último vistazo general

El proyecto *Digit-Recognition* implementa un **pipeline completo y automatizado de clasificación de dígitos manuscritos**, que cubre todas las fases habituales de un problema real de *machine learning*: preparación de datos, análisis exploratorio, ingeniería de características, entrenamiento de modelos, ajuste de hiperparámetros, ensamblado mediante *stacking*, evaluación y guardado del modelo final.

El flujo comienza con la carga del dataset original desde un fichero CSV, donde se normalizan las etiquetas, se escalan los valores de los píxeles y se realiza una partición estratificada en conjuntos de entrenamiento y validación. Para garantizar la viabilidad computacional durante el desarrollo, el número de observaciones se reduce de forma controlada, manteniendo la representatividad de las clases.

A continuación, el análisis exploratorio permite estudiar la distribución de los dígitos, visualizar ejemplos reales de cada clase y analizar la estructura de activación de los píxeles, lo que proporciona una comprensión intuitiva del problema. Sobre estos datos se aplica posteriormente una fase de ingeniería de características que combina selección por varianza y reducción de dimensionalidad mediante PCA, logrando un conjunto compacto de variables que conserva el 95 % de la varianza original.

En la fase de modelado, se entrenan distintos clasificadores base (Random Forest, modelo multinomial y red neuronal simple), que representan enfoques complementarios para la tarea de clasificación. Posteriormente, se realiza un ajuste específico de hiperparámetros para los modelos más relevantes (Random Forest y SVM) mediante validación cruzada, mejorando su capacidad de generalización.

El núcleo metodológico del proyecto se encuentra en la construcción del **ensemble por stacking**: las probabilidades de los modelos base se utilizan como meta-características para entrenar un meta-modelo multinomial, capaz de combinar de forma óptima las predicciones individuales. Este enfoque permite obtener un sistema de clasificación más robusto que cualquiera de los modelos por separado.

La evaluación del sistema se lleva a cabo sobre un conjunto de validación independiente, obteniendo la matriz de confusión y la precisión global como medida principal de rendimiento. Finalmente, por exigencias de la práctica, se entrena y guarda un modelo final reducido empleando únicamente las primeras 100 observaciones del conjunto de entrenamiento, manteniendo la misma filosofía de ensemble.

En conjunto, este proyecto no sólo resuelve un problema concreto de reconocimiento de dígitos, sino que también constituye un ejemplo completo y estructurado de cómo diseñar, implementar, evaluar y documentar un sistema de *machine learning* de principio a fin, siguiendo buenas prácticas de reproducibilidad, modularidad y validación experimental.