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 `1_data_prep.R` se encarga de la **carga, normalización y partición inicial** de los datos. Tras cargar las librerías necesarias y fijar la semilla reproducible, se lee el archivo `train.csv` y se comprueba que contiene exactamente 785 columnas.

La primera columna se renombra como `label` y se transforma en un factor. Para evitar errores posteriores en `caret`, las clases numéricas se renombran como `L0, L1, ..., L9`, asegurando que los nombres son identificadores válidos en R.

Las columnas de píxeles se escalan dividiendo entre 255, y se recombinan con la etiqueta. Para evitar problemas de memoria, se selecciona aleatoriamente una muestra reproducible de 20 000 observaciones.

Por último, el conjunto reducido se divide de forma estratificada en un 80 % para entrenamiento y un 20 % para validación. Ambos subconjuntos se guardan como: `train_processed.rds` y `valid_processed.rds`.

2.2. eda.R

El script `2_eda.R` realiza el **análisis exploratorio** del conjunto de entrenamiento procesado. Se carga `train_processed.rds` y se imprime su tamaño como verificación.

Se calcula la **distribución de clases** por recuento de etiquetas. A continuación, se visualiza un ejemplo de cada dígito en una cuadrícula 2×5, reconstruyendo las imágenes a partir de sus 784 píxeles. Debido al renombrado previo de clases, se utilizan las etiquetas `L0-L9`.

Se analiza también la **sparsity** calculando, para cada clase, el porcentaje medio de píxeles activos (mayores que cero).

Finalmente, se construye un **histograma del número de píxeles activos por imagen**, que se guarda en: `results/pixel_histogram.png`.

2.3. feature_engineering.R

El script `3_feature_engineering.R` implementa la **selección de variables y reducción de dimensionalidad**. Se cargan los conjuntos procesados y se fija una semilla reproducible.

En primer lugar, se aplica **selección por varianza**: se conservan únicamente los píxeles cuya varianza supera el umbral 10^{-6}. Esto elimina dimensiones sin información útil. Los datasets resultantes se guardan como: `train_var.rds` y `valid_var.rds`.

En segundo lugar, se aplica **PCA** sobre los píxeles seleccionados, utilizando centrado, escalado y retención del 95 % de la varianza acumulada. El transformador se ajusta sobre el conjunto de entrenamiento y se aplica también al de validación.

Finalmente, se guardan los resultados: `pca_ctrl.rds`, `train_pca.rds` y `valid_pca.rds`, que se utilizan posteriormente en el entrenamiento de los modelos.

2.4. models.R

El script `4_models.R` realiza el **entrenamiento de los modelos base (baseline)** sobre los datos transformados mediante PCA. Su objetivo es obtener una primera referencia de rendimiento con distintos algoritmos antes de proceder al ajuste fino de hiperparámetros.

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

Se define un control de entrenamiento sin validación cruzada interna: `trainControl(method = "none", classProbs = TRUE)`, ya que en esta fase sólo se pretende entrenar rápidamente los modelos.

Se entrenan cuatro modelos base:

- **Árbol de decisión** mediante `rpart`, con parámetros por defecto.
- **Random Forest** con `ntree = 50` y `mtry = 4`.
- **SVM lineal** con parámetro de coste fijo `C = 1`.
- **Perceptrón multicapa (MLP)** mediante `nnet`, con `size = 5`, `decay = 0.1` y `MaxNWts = 5000`.

De forma opcional, se calcula la **accuracy preliminar** de cada modelo sobre el conjunto de validación para tener una primera referencia comparativa.

Finalmente, los cuatro modelos se guardan conjuntamente en: `models/models_base.rds`.

2.5. tunning.R

El script `5_tunning.R` se encarga del **ajuste de hiperparámetros** de los cuatro modelos entrenados previamente: Random Forest, SVM, Árbol de decisión y Perceptrón multicapa.

Se carga el conjunto `train_pca.rds` y se extrae una **muestra estratificada del 10 %** para reducir el coste computacional del proceso de optimización.

El control de entrenamiento se define mediante validación cruzada de 3 folds con probabilidades activadas: `trainControl(method = "cv", number = 3, classProbs = TRUE)`.

Se realizan los siguientes ajustes:

- **Random Forest**: búsqueda sobre `mtry = {2,4}`.
- **SVM lineal**: ajuste automático del coste con `tuneLength = 2`.
- **Árbol de decisión**: ajuste del parámetro de poda `cp = {0.01, 0.001}`.

- **Perceptrón multicapa**: ajuste de `size = {5,10}` con `decay = 0.1`.

Cada uno de los modelos optimizados se guarda de forma independiente en: `model_rf_tuned.rds`, `model_svm_tuned.rds`, `model_rpart_tuned.rds` y `model_nnet_tuned.rds`.

2.6. ensemble.R

El script `6_ensemble.R` implementa el **ensemble por stacking** utilizando los cuatro modelos ajustados.

Se carga el conjunto completo `train_pca.rds` junto con los cuatro modelos afinados. Se reserva un **20 % del conjunto de entrenamiento** para entrenar el meta-modelo.

Sobre este subconjunto se generan las **predicciones probabilísticas** de:

- Random Forest,
- SVM lineal,
- Árbol de decisión,
- Perceptrón multicapa.

Todas las probabilidades se concatenan en un único conjunto de **meta-características** mediante `bind_cols`. Para evitar conflictos de nombres se aplica: `make.names(..., unique = TRUE)`. Posteriormente se eliminan filas con valores nulos mediante `na.omit`.

El **meta-modelo** se entrena mediante una regresión logística multinomial (`nnet::multinom`) con `MaxNWts = 5000`. El modelo final del ensemble se guarda como:
`models/meta_model.rds`.

2.7. evaluate.R

El script `7_evaluate.R` evalúa el ensemble completo sobre el conjunto de validación PCA.

Se cargan `valid_pca.rds`, los cuatro modelos ajustados y el meta-modelo. Se generan las predicciones probabilísticas de los modelos base, se construyen las meta-características de validación y se introducen en el meta-modelo para obtener la **predicción final de clase**.

Con estas predicciones se construye la **matriz de confusión** mediante `confusionMatrix`, y se imprime por consola la **accuracy final del ensemble**.

El informe completo de evaluación se redirige al archivo: `results/evaluation.txt`.

2.8. save_model.R

El archivo `8_save_model.R` entrena el **modelo final reducido** utilizando exclusivamente los primeros 100 ejemplos del conjunto de entrenamiento, siguiendo el requisito obligatorio de la práctica.

Se cargan los cuatro modelos ajustados y se selecciona: `train_pca[1:100, ]`. Sobre este subconjunto se generan las probabilidades de los cuatro modelos base, que se combinan en un único conjunto de meta-características.

Con estas variables se entrena un **meta-modelo multinomial reducido** mediante `nnet::multinom`. El modelo final se guarda en `models/final_model_100.rds`.

Este modelo conserva la estructura completa del ensemble, pero ajustada únicamente con los 100 primeros ejemplos.

2.9. utils.R

El archivo `utils.R` agrupa las **funciones auxiliares** utilizadas a lo largo de todo el proyecto, con el objetivo de mejorar la modularidad y evitar duplicaciones de código.

La función `set_seed(s = 123)` encapsula la llamada a `set.seed`, proporcionando un mecanismo uniforme para fijar la semilla aleatoria y garantizar la reproducibilidad de todos los procesos.

La función `load_csv(path)` actúa como envoltorio de `readr::read_csv`, permitiendo cargar archivos CSV con tipado explícito, pensados para formatos tipo Kaggle con una columna de etiqueta y 784 píxeles.

Para la visualización de imágenes, se implementa `plot_mnist_row(row_vec, title)`, que reconstruye una imagen de 28×28 píxeles a partir de un vector numérico de longitud 784. La función reorganiza los valores en forma matricial, corrige la orientación mediante rotación y muestra la imagen sin ejes, añadiendo un título opcional. Esta función se utiliza en el análisis exploratorio.

Las funciones `save_model(model, filename)` y `load_model(filename)` son envoltorios directos de `saveRDS` y `readRDS`, respectivamente, y estandarizan el guardado y carga de modelos en el proyecto.

Por último, `pixels_to_matrix(df_pixels)` convierte un `data.frame` de píxeles en una matriz numérica normalizada en el intervalo $[0, 1]$, dividiendo por 255. Esta función resulta útil cuando se requiere trabajar directamente con las imágenes en formato matricial.

En conjunto, `utils.R` actúa como una pequeña biblioteca de soporte que simplifica el resto del pipeline y centraliza las operaciones auxiliares más frecuentes.

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.