

---

**title: “Optimización Numérica y sistemas lineales en R”**

**subtitle: “Aplicaciones con paralelización”**

**format:**

**html:**

**toc: true**

**code-fold: false**

**self-contained: true**

**editor: visual**

---

## ¿Qué es un óptimo y un algoritmo de optimización?

---

- El óptimo global es la mejor solución posible entre todas las opciones existentes, sin excepción. Y un algoritmo para encontrar un óptimo es, simplemente, una **receta** o una serie de **pasos metódicos** que sigue un ordenador para buscar y comparar sistemáticamente esas opciones hasta encontrar esa solución ganadora.

Veamos un ejemplo didáctico: <https://clementmihailescu.github.io/Pathfinding-Visualizer/#>

La **Optimización Numérica**, abordada en el Capítulo 8 de “A First Course in Statistical Programming with R”, es una de las áreas más importantes y aplicadas en Data Science. Este documento sirve como una guía práctica que explora dos de sus aplicaciones fundamentales. Partiremos de los conceptos teóricos del libro para construir soluciones ejecutables en R, y finalizaremos demostrando cómo acelerar estas tareas mediante computación en paralelo.

### a) Optimización de Funciones Continuas: Entrenamiento de Redes Neuronales

---

Comenzaremos por el problema general de optimización sobre funciones continuas, que es el núcleo del machine learning moderno.

## Conceptos del Capítulo 8: Optimización General

**Ejemplo:** En el modelado estadístico, a menudo queremos encontrar un conjunto de parámetros para un modelo que minimice los errores de predicción. Aquí, los parámetros del modelo son las variables  $x$  y la medida del error de predicción es la función  $f(x)$  que queremos minimizar.

**Definición:** La optimización numérica es el área de la computación que se enfoca en resolver el siguiente problema: dada una función  $f(\cdot)$ , ¿qué valor de  $x$  hace que  $f(x)$  sea lo más grande o lo más pequeño posible? El entrenamiento de redes neuronales es un problema de minimización a gran escala sobre una función continua.

El proceso que seguiremos para entrenar nuestra red neuronal se ilustra en el siguiente esquema:

```
%%| label: fig-nn-methodology
%%| fig-cap: "Esquema metodológico para el entrenamiento de una red neuronal."

graph TD
    A[1. Modelo y clasificación] --> B[2. Definir Problema de Optimización<br><i>Fu<
    B --> C{3. Iniciar Entrenamiento (fit)};
    C --> D[Cálculo Iterativo];
    subgraph "Ciclo de Convergencia"
        D -- En cada paso --> E[a. Calcular Gradiente];
        E --> F[b. Ajustar Parámetros];
    end
    F --> C;
    C -- Convergencia Alcanzada --> G[4. Modelo Óptimo (Mínimo Local)];
```

## 1. Clasificación de Dígitos MNIST con un Modelo Random Forest

El Random Forest no se optimiza minimizando una función de pérdida continua, a diferencia de otros modelos. Su entrenamiento es una partición recursiva de nodos, donde la “optimización” es avara y local. El criterio clave para cada división es la **reducción de impureza**, medida por

funciones como la **Impureza de Gini**, buscando en cada paso la característica y el valor que maximizan dicha reducción.

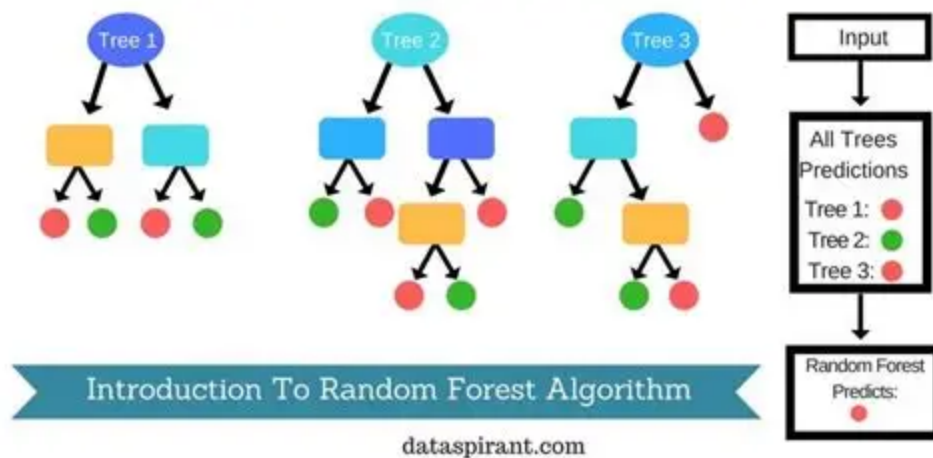
Este proceso se aplica a cada árbol individual, que se entrena con una muestra aleatoria de datos y un subconjunto de características. El modelo final combina las predicciones de todos los árboles por votación, lo que lo hace robusto y menos propenso al sobreajuste.

## Fórmula para la Impureza de Gini

La medida de impureza de Gini para un nodo con  $K$  clases es:

$$G(p) = 1 - \sum_{k=1}^K p_k^2$$

Donde  $p_k$  es la proporción de instancias de la clase  $k$  en ese nodo. El objetivo del algoritmo es encontrar la división que maximiza la reducción de esta impureza.



## Pseudo-fórmula para la Optimización de un Nodo

Para cada nodo en un árbol:

1. Iterar a través de cada característica y umbral de división.
2. Calcular la impureza de Gini del nodo padre:  $G_{\text{padre}}$ .
3. Calcular la ganancia de la división:  $\text{Ganancia} = G_{\text{padre}} - G_{\text{hijos\_ponderada}}$ .
4. Seleccionar la división (característica + umbral) con la Ganancia más alta.
5. Dividir el nodo y repetir recursivamente hasta que el árbol esté completo.

A seguir detallamos el procedimiento para entrenar y evaluar un modelo de Machine Learning con el objetivo de clasificar dígitos manuscritos del conjunto de datos MNIST. Se emplea un modelo Random Forest, una técnica robusta implementada íntegramente en el entorno de R.

---


## Paso 1: Carga de Librerías

Se inicia el proceso cargando las librerías de R necesarias para el análisis. Cada una cumple una función específica:

- **randomForest:** Provee el algoritmo para la construcción del modelo.
- **dslabs:** Facilita el acceso y la carga del dataset MNIST.
- **caret:** Ofrece herramientas para una evaluación detallada del rendimiento del modelo.

```
#| label: cargar-librerias
#| message: false
#| warning: false

# Es necesario tener estos paquetes instalados previamente (ej: install.packages("r
library(randomForest)
library(dslabs)
library(caret)
```



---

## Paso 2: Carga y Segmentación de los Datos

El dataset MNIST se carga utilizando la función `read_mnist` del paquete `dslabs`. Inmediatamente, los datos son segmentados en dos conjuntos independientes:

1. **Conjunto de Entrenamiento:** Datos utilizados para que el modelo aprenda los patrones.
2. **Conjunto de Prueba:** Datos reservados para evaluar el modelo sobre información nueva.

```
#| label: cargar-datos

# Se carga el dataset completo
mnist <- read_mnist()
```

```
# Se asignan los datos de entrenamiento
x_train <- mnist$train$images
y_train <- mnist$train$labels

# Se asignan los datos de prueba
x_test <- mnist$test$images
y_test <- mnist$test$labels

# Se muestra un resumen de las dimensiones de los conjuntos de datos
cat("Dimensiones del conjunto de entrenamiento:", dim(x_train)[1], "muestras\n")
cat("Dimensiones del conjunto de prueba:", dim(x_test)[1], "muestras\n")
```

---

### Paso 3: Preparación de los Datos

Los modelos basados en árboles, como Random Forest, requieren una preparación mínima. El único paso esencial es la conversión de la variable objetivo a un tipo `factor`. Esto asegura que el algoritmo trate el problema como uno de clasificación categórica en lugar de regresión numérica.

```
#| label: preparar-datos

# Se convierte la variable respuesta del conjunto de entrenamiento a factor
y_train <- as.factor(y_train)
```

---

### Paso 4: Entrenamiento del Modelo

En esta fase, se construye el modelo Random Forest. El algoritmo es alimentado con las imágenes de entrenamiento (`x_train`) y sus correspondientes etiquetas (`y_train`). Se establece una semilla de aleatoriedad (`set.seed`) para garantizar que los resultados del entrenamiento sean reproducibles.

```
#| eval: false
#| include: false

#| label: entrenar-modelo
#| cache: true
```

```

tiempo_entrenamiento <- system.time({

# Se fija la semilla para la reproducibilidad
set.seed(123)
cat("Iniciando el proceso de entrenamiento del modelo...\n")
# Se entrena el modelo Random Forest.
modelo_rf <- randomForest(x = x_train, y = y_train, ntree = 100)

})

cat("\n--- Tiempo de ejecución del entrenamiento ---\n")
print(tiempo_entrenamiento)

```

Para 100 arboles entrenados

- El tiempo elapsed fue de **861.23** segundos.

## Paso 5: Evaluación del Rendimiento

Una vez entrenado, el rendimiento del modelo es puesto a prueba. Se utilizan las imágenes del conjunto de prueba ( `x_test` ) para generar predicciones. Estas predicciones son luego comparadas con las etiquetas reales ( `y_test` ) para cuantificar la precisión del modelo.

```

#| label: evaluar-modelo

# Se generan predicciones sobre el conjunto de prueba
predicciones <- predict(modelo_rf, newdata = x_test)

# Se crea la matriz de confusión para comparar predicciones con valores reales
# La variable y_test también se convierte a factor para una comparación consistente
matriz_confusion <- confusionMatrix(data = predicciones, reference = as.factor(y_te

```



## Paso 6: Resultados Finales

Los resultados se presentan a través de la matriz de confusión. La métrica clave es la **Precisión (Accuracy)**, que representa el porcentaje total de clasificaciones correctas realizadas por el modelo sobre el conjunto de prueba.

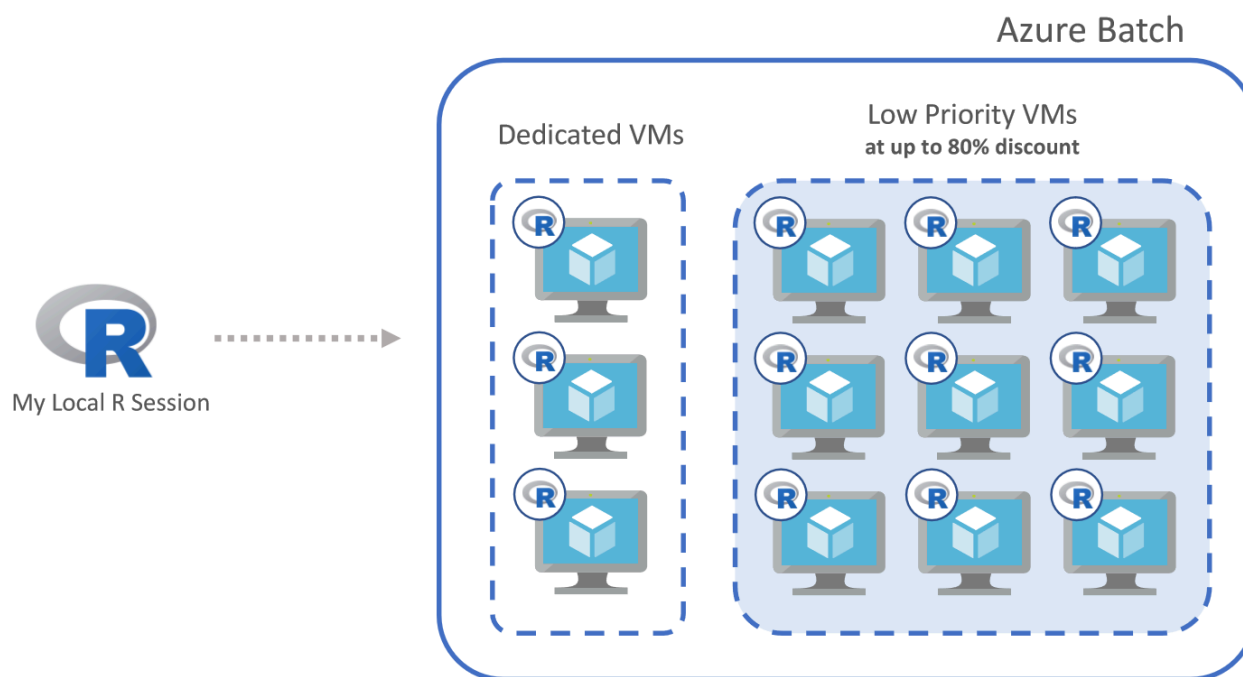
```
#| label: resultados

# Se imprime el informe completo de la matriz de confusión
print(matriz_confusion)

# Se extrae y muestra la precisión general de manera destacada
precision <- matriz_confusion$overall['Accuracy'] * 100
cat(paste("\nPrecisión General del Modelo:", round(precision, 2), "%\n"))
```

La alta precisión obtenida confirma la efectividad del modelo Random Forest para esta tarea de clasificación de imágenes, constituyendo una alternativa robusta y nativa en el ecosistema R.

## Paso 7: Aceleración con Entrenamiento Paralelo



Para reducir el tiempo de cómputo, se implementa el entrenamiento en paralelo. El trabajo de construir los árboles se distribuye entre los núcleos disponibles de la CPU. El código detecta el sistema operativo para aplicar el método de paralelización más adecuado.

```
#| label: cargar-librerias
#| message: false
#| warning: false
```

```

#library(randomForest)
#library(dslabs)
#library(caret)
library(parallel)

#| label: entrenar-paralelo
#| cache: true

# 1. Definir parámetros
total_arboles <- 100 # Se usa un número mayor para que la paralelización sea más e
num_nucleos <- detectCores() - 1 # Dejar un núcleo libre para el sistema
arboles_por_nucleo <- ceiling(total_arboles / num_nucleos)

cat(paste("Iniciando entrenamiento en paralelo sobre", num_nucleos, "núcleos.\n"))
cat(paste("Cada núcleo construirá aproximadamente", arboles_por_nucleo, "árboles.\n")

# 2. Medir el tiempo de ejecución del entrenamiento en paralelo

# Crear un clúster de procesos trabajadores (workers)
# num_nucleos debe estar definido previamente, ej: num_nucleos <- detectCores() - 1
cl <- makeCluster(num_nucleos)

# Exportar las variables necesarias del entorno principal a cada worker del clúster
# Las variables x_train, y_train y arboles_por_nucleo deben existir en tu entorno
clusterExport(cl, varlist = c("x_train", "y_train", "arboles_por_nucleo"), envir =

# Cargar la librería 'randomForest' en cada worker del clúster
clusterEvalQ(cl, library(randomForest))

tiempo_paralelo <- system.time({

# Ejecutar el entrenamiento en paralelo. parLapply distribuye la tarea entre los wo
lista_de_bosques <- parLapply(cl, 1:num_nucleos, function(i) {
  randomForest(x = x_train, y = y_train, ntree = arboles_por_nucleo)
})

# Detener el clúster para liberar los recursos. ¡Paso muy importante!
stopCluster(cl)

```



```
# 3. Combinar los resultados en un único modelo
modelo_rf_paralelo <- do.call(combine, lista_de_bosques)

}))

cat("\nEntrenamiento en paralelo finalizado!\n")
cat("Tiempo total de ejecución:\n")
nprint(tiempo_paralelo)
```

Para 100 árboles entrenados

- Tiempo Real (elapsed): La tarea completa tardó **520.39** segundos en finalizar desde el inicio hasta el fin.
- Trabajo del Proceso Principal (user): El proceso principal de R solo trabajó activamente segundos.

## Paso 7: Evaluación del Modelo Paralelo

Finalmente, se evalúa el modelo resultante del entrenamiento en paralelo. Se espera que la precisión sea muy similar a la del modelo secuencial, pero con un tiempo de entrenamiento significativamente menor.

```
#| label: evaluar-paralelo

# Se evalúa el nuevo modelo con los datos de prueba
predicciones_paralelo <- predict(modelo_rf_paralelo, newdata = x_test)
matriz_confusion_paralelo <- confusionMatrix(data = predicciones_paralelo, referenc
precision_paralelo <- matriz_confusion_paralelo$overall['Accuracy'] * 100

cat(paste("\nPrecisión del Modelo entrenado en Paralelo:", round(precision_paralelo
```

## b) Resolución de Sistemas Lineales: Programación Lineal

Ahora nos enfocamos en un tipo específico de optimización donde tanto el objetivo como las restricciones son lineales.

## Conceptos del Capítulo 8: Programación Lineal (Sección 8.5)

**Ejemplo Motivador (del libro):** Una compañía tiene dos procedimientos para reducir la contaminación. Cada uno tiene un costo y una efectividad diferente para reducir dos tipos de gases. El objetivo es cumplir con las regulaciones de reducción de ambos gases al **mínimo costo** posible.

**Definición Simple (del libro):** Cuando la función objetivo (ej. el costo) y las restricciones (ej. los límites de reducción) pueden expresarse como ecuaciones o desigualdades **lineales**, el problema se denomina **Programación Lineal**.

El método para resolver este tipo de sistemas se basa en las propiedades geométricas del problema, como se ilustra a continuación:

```
%%| label: fig-lp-methodology
%%| fig-cap: "Esquema metodológico para la resolución de un problema de Programación Lineal"
graph TD
    A[1. Definir Problema de Negocio<br><i>Objetivo y Restricciones</i>] --> B[2. Formular el Problema de Optimización<br><i>Función Objetivo y Restricciones</i>]
    B --> C[3. Aplicar Algoritmo Simplex]
    subgraph "Búsqueda Eficiente"
        C --> D[a. Iniciar en un vértice]
        D --> E{b. ¿Hay un vértice vecino mejor?}
        E -- Sí --> F[c. Moverse al mejor vértice]
        F --> E
    end
    end
    E -- No --> G[4. Solución Óptima Encontrada<br><i>(El mejor vértice)</i>]
```

## 1. Definición del Problema y Solución

Vamos a resolver un problema clásico de mezcla de productos para maximizar el beneficio.

- **Objetivo:** Maximizar el beneficio total.
  - Beneficio por unidad de A: \$40
  - Beneficio por unidad de B: \$30
  - Función objetivo:  $C(A,B)=40*A + 30B$

- **Restricciones (Sistema Lineal):**

1. **Mano de Obra:**  $1 \cdot A + 2 \cdot B \leq 240$

2. **Materia Prima:**  $3 \cdot A + 1 \cdot B \leq 300$

```
#| label: setup-lp
library(lpSolve)

# Definir los componentes del problema
objective.in <- c(40, 30)
const.mat <- matrix(c(1, 2, 3, 1), nrow = 2)
const.dir <- c("<=", "<=")
const.rhs <- c(240, 300)

# Resolver el sistema
solucion_optima <- lp("max", objective.in, const.mat, const.dir, const.rhs)

# Interpretar la solución
cat("Beneficio máximo: $", solucion_optima$objval, "\n")
cat("Unidades de A:", solucion_optima$solution[1], "\n")
cat("Unidades de B:", solucion_optima$solution[2], "\n")
```

## 2. Implementación en Paralelo: Simulación de Escenarios

Un solo problema de programación lineal se resuelve muy rápido. La paralelización se vuelve indispensable cuando necesitamos resolver **miles de problemas similares**, por ejemplo, al simular el efecto de la variabilidad en los recursos disponibles.

**Tarea:** Resolveremos el problema 5,000 veces, pero cada vez con una ligera variación aleatoria en los recursos de mano de obra y materia prima.

```
#| label: paralelo-pl-espanol
# Cargar las librerías necesarias en la sesión principal
library(lpSolve)
library(parallel)

# Función que resuelve un escenario de Programación Lineal (PL)
resolver_escenario_pl <- function(vector_rhs) {
  objetivo <- c(40, 30)
  matriz_restr <- matrix(c(1, 3, 2, 1), nrow = 2)
  dir_restr <- c("<=", "<=")
```

```

solucion_lp <- lp("max", objetivo, matriz_restr, dir_restr, vector_rhs)

if (solucion_lp$status == 0) return(solucion_lp$solution) else return(c(NA, NA))
}

# Generar 5000 escenarios de restricciones aleatorias
num_simulaciones <- 5000
set.seed(42)
lista_escenarios <- lapply(1:num_simulaciones, function(i) {
  c(240 + rnorm(1, 0, 10), 300 + rnorm(1, 0, 15))
})

# --- Ejecución Secuencial ---
cat("Iniciando ejecución secuencial para", num_simulaciones, "escenarios...\n")
tiempo_secuencial <- system.time({
  resultados_secuencial <- lapply(lista_escenarios, resolver_escenario_pl)
})
cat("Tiempo secuencial:\n")
print(tiempo_secuencial)

# --- Ejecución en Paralelo ---
num_nucleos <- detectCores() - 1

# Medir el tiempo de toda la operación en paralelo
tiempo_paralelo <- system.time({

  # 1. Crear el clúster de procesos trabajadores
  clúster <- makeCluster(num_nucleos)

  # 2. Exportar los objetos necesarios (la función y los datos)
  clusterExport(clúster, varlist = c("resolver_escenario_pl", "lista_escenarios"))

  # 3. Cargar la librería 'lpSolve' en cada núcleo del clúster
  clusterEvalQ(clúster, library(lpSolve))

  # 4. Ejecutar la tarea en paralelo
  resultados_paralelo <- parLapply(clúster, lista_escenarios, resolver_escenario_pl)

  # 5. Detener el clúster (paso fundamental)
  stopCluster(clúster)
})

```

```
) # Fin del bloque medido por system.time

cat("\nTiempo paralelo:\n")
print(tiempo_paralelo)

# Calcular y mostrar la mejora de velocidad
ganancia_velocidad <- round(tiempo_secuencial['elapsed'] / tiempo_paralelo['elapsed'])
cat("\nGanancia de velocidad (Tiempo Secuencial / Tiempo Paralelo):", ganancia_velocidad)
```

**Observación:** Cuando una tarea consiste en muchas operaciones pequeñas e independientes, `mclapply` puede distribuirlas eficientemente entre los núcleos del procesador. Esto permite un rendimiento muy superior al de un bucle secuencial, siendo una técnica clave para análisis y simulaciones a gran escala.