Digit Recognizer
Aprendizaje Computacional

Emilio Gómez Esteban
Fernando Javier Lopez Cerezo
Maria Peinado Toledo
Alberto Trigueros Postigo

2 de diciembre de 2024

Índice

1. Introducción

El reconocimiento de dígitos escritos a mano es un problema clásico en el ámbito de la visión artificial y el aprendizaje automático. En este contexto, el conjunto de datos MNIST (*Modified National Institute of Standards and Technology*) ha servido como un recurso estándar desde su lanzamiento en 1999, siendo ampliamente utilizado por investigadores y estudiantes para desarrollar y evaluar algoritmos de clasificación. Este conjunto de datos, considerado como el "hola mundo"de la visión artificial, contiene imágenes en escala de grises de dígitos del 0 al 9, cada una con un tamaño de 28×28 píxeles.

El objetivo principal de esta práctica es identificar correctamente los dígitos de un conjunto de datos de imágenes escritas a mano, utilizando diferentes técnicas de aprendizaje automático. Además, se busca determinar el modelo con mayor precisión (*accuracy*) y menor consumo de recursos computacionales. Para lograrlo, se pueden emplear diversos enfoques, como el ajuste de hiperparámetros (*tuning*), la reducción de dimensionalidad (PCA o *encoders*) o técnicas de manipulación de imágenes.

En este trabajo, evaluamos varios modelos de aprendizaje supervisado estudiados en la asignatura, así como combinaciones de ellos (*ensemble methods*). La comparación de los modelos se realiza considerando tanto su desempeño predictivo como su eficiencia computacional. Esto nos permitirá identificar el modelo más adecuado para resolver el problema propuesto, optimizando el balance entre precisión y complejidad.

El informe incluye los pasos seguidos en la práctica, justificaciones para la selección de técnicas, la planificación de las tareas, y los programas desarrollados en R. Además, se adjunta el modelo final entrenado y optimizado para los 100 primeros dígitos del conjunto de datos.

Integrante	**Contribución**
Emilio Gómez Esteban	Rpart
Fernando Javier López Cerezo	Random Forest
Alberto Trigueros Postigo	SVM
María Peinado Toledo	Bagging

Cuadro 1: Contribuciones de los integrantes del grupo.

2. Preprocesamiento de Datos

El preprocesamiento de datos se realizó en tres etapas principales: carga, inspección y división del conjunto de datos.

2.1. Carga de Datos

Se cargaron dos conjuntos de datos utilizando el archivo `train.csv` para el entrenamiento y el archivo `test.csv` para la evaluación. El archivo de entrenamiento incluye tanto las imágenes de dígitos escritos a mano como sus respectivas etiquetas, mientras que el archivo de prueba contiene únicamente las imágenes. El código utilizado para esta etapa es el siguiente:

```
train <- read.csv("train.csv")
test <- read.csv("test.csv")
```

2.2. Inspección de los Datos

Para garantizar la calidad de los datos y evaluar posibles desequilibrios en las clases, se calculó la proporción de cada dígito presente en el conjunto de entrenamiento. Además, las etiquetas (`label`) se convirtieron en factores categóricos, ya que los modelos de clasificación requieren esta estructura para manejar correctamente las clases. El código utilizado es el siguiente:

```
prop.table(table(train$label)) * 100
train$label <- factor(train$label)
```

2.3. División de los Datos

El conjunto de entrenamiento se dividió en dos subconjuntos:

- **Conjunto de entrenamiento**: Contiene el 80 % de los datos originales, utilizado para entrenar los modelos.
- **Conjunto de validación**: Consiste en el 20 % restante, reservado para evaluar el rendimiento de los modelos.

La división se realizó mediante una partición aleatoria. Primero, se generaron índices aleatorios que abarcan todos los ejemplos del conjunto de entrenamiento. A partir de estos índices, se seleccionaron los primeros para el conjunto de validación y los restantes para el entrenamiento. El código utilizado es el siguiente:

```
d_size <- nrow(train)
dtest_size <- ceiling(0.2 * d_size)
samples <- sample(1:d_size, d_size, replace = FALSE)
indexes <- samples[1:dtest_size]

dtrain <- train[-indexes,]
dtest <- train[indexes,]
```

3. Árbol de Decisión (Rpart)

Para abordar el problema de clasificación de dígitos manuscritos, se utilizó una implementación del método de árboles de decisión, **Rpart**, específicamente basado en el algoritmo CART (Classification and Regression Trees). Este algoritmo se utiliza para construir modelos predictivos que dividen los datos en subconjuntos basados en reglas de decisión derivadas de los predictores. Después de realizar el prepocesamiento de los datos, el modelo se ha entrenado de la siguiente forma:

```
start_time <- Sys.time()
tree = rpart(label~., data = dtrain, method = "class")
end_time <- Sys.time()
```

En este bloque de código encontramos:

- `Sys.time()` se utiliza para medir el tiempo de inicio y final del entrenamiento del modelo, lo cual es útil para evaluar el coste computacional de entrenar el modelo.
- `rpart()` es la función utilizada para entrenar el modelo Rpart. La fórmula `label ~ .` indica que se utiliza la variable `label` (el dígito a predecir) como la variable dependiente y todas las demás variables del conjunto de datos como predictores. El parámetro `method = 'class'` indica que el modelo está configurado para resolver un problema de clasificación.

Luego, para evaluar el rendimiento del modelo, se realiza lo siguiente:

```
matrizconfusiontree <- table(predict(tree, newdata = dtest, type = "
    class"), dtest$label)
accuracytree <- sum(diag(matrizconfusiontree)) / sum(
    matrizconfusiontree)
accuracytree

fancyRpartPlot(tree)
barplot(tree$variable.importance)
```

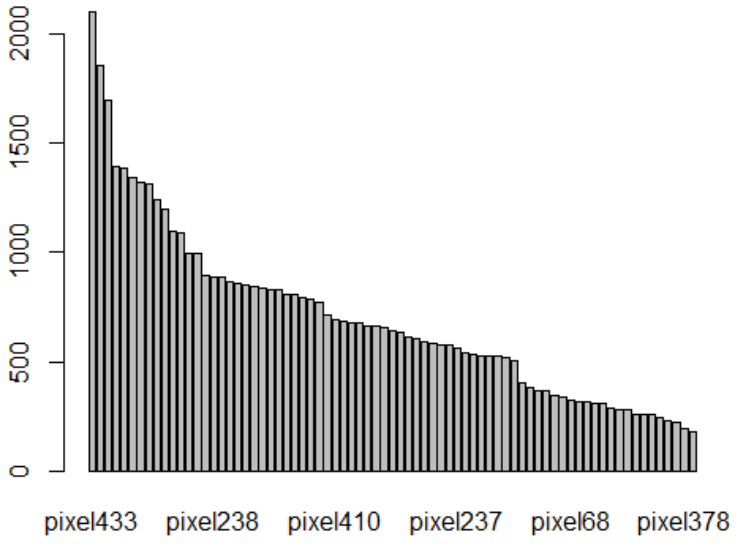
Estas líneas evalúan el modelo utilizando los datos de prueba (`dtest`):

- `predict(tree, newdata = dtest, type = çlass")` genera las predicciones del modelo para los datos de prueba del conjunto `dtest`.
- Usamos la función `table()` sobre el `predict()` anterior con parámetro `dtest$label` para crea la matriz de confusión comparando las predicciones con las etiquetas reales.
- `sum(diag(matrizconfusiontree)) / sum(matrizconfusiontree)` calcula la precisión del modelo como la proporción de predicciones correctas en comparación con el total de predicciones realizadas.
- Usando `fancyRpartPlot(tree)` se interpretan las decisiones del modelo y se puede visualizar el árbol.
- Usando `barplot(tree$variable.importance)` se genera un gráfico de barras para mostrar las variables más relevantes según el modelo.

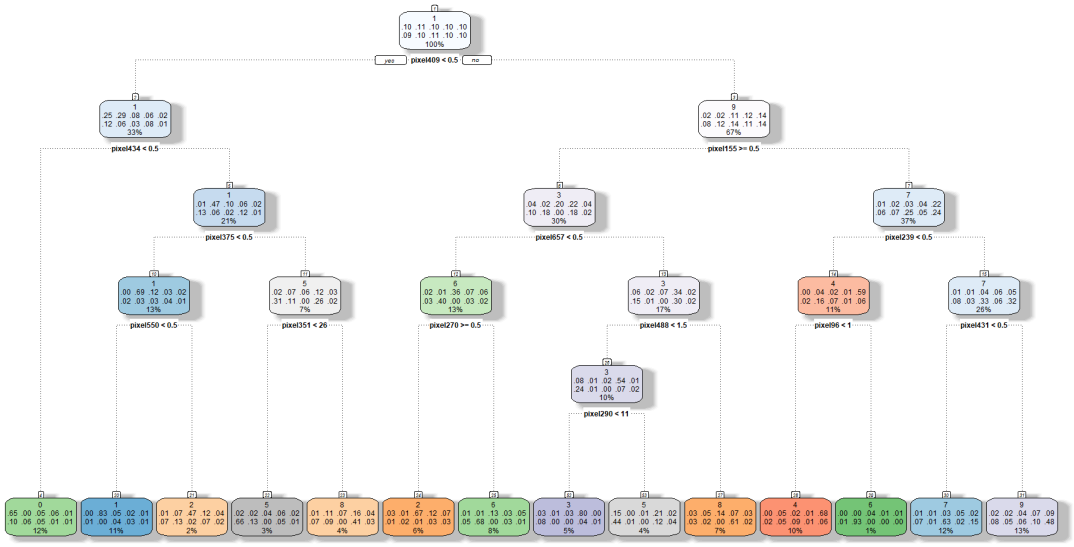
Los resultados obtenidos para este modelo son los siguientes:

- **Precisión**: 0.6333
- **Tiempo de entrenamiento**: 53.2402 segundos

Además, podemos observar la importancia que da este modelo de predicción a cada variable, siendo la más predominante el Pixel 433:



También hemos representado un gráfico del árbol obtenido:



Rattle 2024-Dec-01 19:03:00 lopez

4. Random Forest

Para abordar el problema de clasificación de dígitos manuscritos, se utilizó el algoritmo **Random Forest**, que es un modelo de aprendizaje automático basado en árboles de decisión. A continuación, se describen los resultados obtenidos al entrenar y evaluar el modelo con diferentes configuraciones de número de árboles (`ntree`).

4.1. Entrenamiento y Evaluación del Modelo con 5 Árboles

El primer modelo se entrenó utilizando 5 árboles. El proceso consiste en entrenar un modelo de **Random Forest** con 5 árboles de decisión, guardar el modelo entrenado y evaluar su rendimiento utilizando el conjunto de prueba. El código utilizado para entrenar y evaluar el modelo es el siguiente:

```
start_time <- Sys.time()
rf.model <- randomForest(label ~ ., data = dtrain, ntree = 5)
end_time <- Sys.time()
save(rf.model, file="rf5.rda")
end_time - start_time
```

En este bloque de código:

- `Sys.time()` se utiliza para medir el tiempo de inicio y final del entrenamiento del modelo, lo cual es útil para evaluar el costo computacional de entrenar el modelo.
- `randomForest()` es la función utilizada para entrenar el modelo **Random Forest**. La fórmula `label ~ .` indica que se utiliza la variable `label` (el dígito a predecir) como la variable dependiente y todas las demás variables del conjunto de datos como predictores. El parámetro `ntree = 5` especifica que se utilizarán 5 árboles en el modelo.
- `save(rf.model, file="rf5.rda")` guarda el modelo entrenado en un archivo `.rda`, lo que permite cargar el modelo más tarde sin necesidad de volver a entrenarlo.

Luego, para evaluar el rendimiento del modelo, se realiza lo siguiente:

```
rf.predict <- predict(rf.model, dtest)
matrizconfusionRF <- table(rf.predict, dtest$label)
accuracyRF <- sum(diag(matrizconfusionRF)) / sum(matrizconfusionRF)
accuracyRF
```

- `predict(rf.model, dtest)` genera las predicciones del modelo utilizando el conjunto de prueba `dtest`.
- `table(rf.predict, dtest$label)` construye una matriz de confusión, que muestra cómo se comparan las predicciones con las etiquetas reales del conjunto de prueba.

- `sum(diag(matrizconfusionRF)) / sum(matrizconfusionRF)` calcula la precisión del modelo como la proporción de predicciones correctas en comparación con el total de predicciones realizadas.

Los resultados obtenidos para este modelo son los siguientes:

- **Precisión**: 0.9104
- **Tiempo de entrenamiento**: 24.9980 segundos

4.2. Entrenamiento y Evaluación del Modelo con 50 Árboles

En el segundo modelo, se entrenó un **Random Forest** con 50 árboles. El proceso es similar al anterior, pero con un mayor número de árboles para ver cómo mejora el rendimiento del modelo. El código correspondiente es el siguiente:

```
start_time <- Sys.time()
rf.model <- randomForest(label ~ ., data = dtrain, ntree = 50)
end_time <- Sys.time()
save(rf.model, file="rf50.rda")
end_time - start_time
```

El procedimiento es el mismo que para el modelo con 5 árboles, pero con el parámetro `ntree = 50` para usar 50 árboles. `end_time - start_time` permite calcular el tiempo de entrenamiento, y el modelo entrenado se guarda con `save()` en un archivo `.rda`.

El código para realizar la predicción y calcular la precisión es igual al utilizado en el primer modelo:

```
rf.predict <- predict(rf.model, dtest)
matrizconfusionRF <- table(rf.predict, dtest$label)
accuracyRF <- sum(diag(matrizconfusionRF)) / sum(matrizconfusionRF)
accuracyRF
```

Los resultados obtenidos para este modelo son los siguientes:

- **Precisión**: 0.9632
- **Tiempo de entrenamiento**: 4.4706 minutos

4.3. Entrenamiento y Evaluación del Modelo con 100 Árboles

El tercer modelo se entrenó utilizando 100 árboles. El proceso de entrenamiento es el mismo que en los modelos anteriores, pero con una mayor cantidad de árboles. El código utilizado es el siguiente:

```
start_time <- Sys.time()
rf.model <- randomForest(label ~ ., data = dtrain, ntree = 100)
end_time <- Sys.time()
save(rf.model, file="rf100.rda")
end_time - start_time
```

El bloque de código para realizar las predicciones y calcular la precisión es el mismo que para los modelos anteriores.

Los resultados obtenidos para este modelo son los siguientes:

- **Precisión**: 0.9663
- **Tiempo de entrenamiento**: 16.855 minutos

4.4. Resultados y Comparación

Los resultados obtenidos de los tres modelos entrenados con diferentes configuraciones de árboles son los siguientes:

- **Modelo con 5 árboles**: Precisión = 0.9104, Tiempo de entrenamiento = 24.9980 segundos
- **Modelo con 50 árboles**: Precisión = 0.9632, Tiempo de entrenamiento = 4.4706 minutos
- **Modelo con 100 árboles**: Precisión = 0.9663, Tiempo de entrenamiento = 16.855 minutos

Como se puede observar, al aumentar el número de árboles, la precisión del modelo mejora. Sin embargo, el tiempo de entrenamiento también aumenta considerablemente. Aunque el modelo con 100 árboles tiene una precisión ligeramente mayor (0.9632) en comparación con el modelo de 50 árboles (0.9627), el incremento en precisión es marginal en relación con el aumento significativo en el tiempo de entrenamiento. Esto sugiere que un modelo con 50 árboles podría ser una opción más eficiente en cuanto a tiempo sin perder una precisión significativa.

5. Support Vector Machine (SVM)

Continuamos acercándonos a la clasificación de dígitos manuscritos, esta vez aplicando el algoritmo de clasificación **Support Vector Machine** o **SVM**. Concretamente se aplicó dicho algoritmo con un kernel polinómico.

Este algoritmo consiste en hallar un hiperplano que separe de la mejor forma posible dos clases diferentes de puntos de datos. "De la mejor forma posible" implica el hiperplano con el margen más amplio entre las dos clases. El margen se define como la anchura máxima de la región paralela al hiperplano que no tiene puntos de datos interiores. El algoritmo solo puede encontrar este hiperplano en problemas que permiten separación lineal; en la mayoría de los problemas prácticos, el algoritmo maximiza el margen flexible permitiendo un pequeño número de clasificaciones erróneas.

Se ha entrenado el modelo con el siguiente código:

```
start_time <- Sys.time()
filter <- ksvm(label ~ ., data = dtrain, kernel = "polydot", kpar =
    list(degree = 3), cross = 3)
end_time <- Sys.time()
```

Encontramos los siguientes conceptos:

- `Sys.time()` mide el tiempo de inicio y fin del entrenamiento del modelo, usado para ver el coste computacional de entrenamiento del modelo.
- `ksvm()` función que usamos para entrenar los datos mediante SVM. Asignamos el modelo a la variable filter. Presenta los siguientes parámetros:
  - `label .`: La variable label (la clase a predecir) depende de todas las demás variables del conjunto de datos dtrain.
  - `data = dtrain`: Se entrena con el conjunto de datos dtrain.
  - `kernel = "polydot"`: Se aplica un kernel polinómico.
  - `kpar = list(degree = 3)`: Define el parámetro del kernel polinómico, en este caso un grado de 3 (es decir, un polinomio cúbico).
  - `cross = 3`: Realiza validación cruzada de 3 particiones (3-fold cross-validation).

Se evalúa el modelo con el siguiente código:

```
matrizconfusionKSVMpol<- table(predict(filter, dtest), dtest$label)
accuracyksvmpol <- sum(diag(matrizconfusionKSVMpol)) / sum(
    matrizconfusionKSVMpol)

# Mostrar resultados
print(matrizconfusionKSVMpol)
cat("Accuracy del modelo:", accuracyksvmpol, "\n")
cat("Tiempo de ejecucion SVM:", end_time-start_time, "\n")
```

Encontramos los siguientes conceptos:

- `table()`: Crea una matriz de confusión comparando las predicciones con las etiquetas reales. Presenta los siguientes parámetros:
  - `predict(filter, dtest)`: Genera predicciones del modelo filter (entrenado previamente) sobre el conjunto de prueba dtest.
  - `dtest$label`: Obtiene las etiquetas reales del conjunto de prueba.
- `sum(diag(matrizconfusionKSVMpol)) / sum(matrizconfusionKSVMpol)`: Calcula la precisión (accuracy) dividiendo el número de predicciones correctas entre el total de predicciones realizadas. Sabemos que las predicciones correctas son valores de 1 en la diagonal de la matriz de confusión. Entonces para hallarla sumamos los valores de la diagonal y dividimos entre la suma de todos en la matriz de confusión.
- Por último mostramos los resultados por pantalla.

Los resultados obtenidos son los siguientes:

- **Precisión**: 0.9748
- **Tiempo de entrenamiento**: 35.1554 min

6. Bagging

Para abordar el problema de clasificación de dígitos manuscritos, se utilizó el algoritmo **Bagging**. En este modelo, se selecciona una muestra aleatoria de datos en un conjunto de entrenamiento con reemplazo, lo que significa que los puntos de datos individuales se pueden elegir más de una vez. Después de generar varias muestras de datos, estos modelos débiles se entrenan de manera independiente. Dependiendo del tipo de tarea (regresión o clasificación, por ejemplo), el promedio o la mayoría de esas predicciones arrojan una estimación más precisa.

Después de realizar el prepocesamiento de los datos, el modelo se ha entrenado de la siguiente forma:

```
start_time <- Sys.time()
Modelo_AdaBag  <- bagging(label~.,
                          data=dtrain,
                          na.action = na.omit,
                          mfinal=9,
                          control=rpart.control(cp = 0.001, minsplit=7)
                             )
end_time <- Sys.time()
```

En este bloque de código encontramos:

- `Sys.time()` se utiliza para medir el tiempo de inicio y final del entrenamiento del modelo, lo cual es útil para evaluar el coste computacional de entrenar el modelo.
- `bagging()` es la función utilizada para entrenar el modelo Bagging.
  La fórmula `label ~ .` indica que se utiliza la variable `label` (el dígito a predecir) como la variable dependiente y todas las demás variables del conjunto de datos como predictores.
  El parámetro `data=dtrain` especifica el conjunto de datos de entrenamiento. En este caso, la variable dtrain debe ser un data frame que contenga las variables predictoras y la variable objetivo label.
  El parámetro `na.action = na.omit` indica cómo manejar los valores faltantes. La opción na.omit elimina cualquier fila del conjunto de datos que contenga valores faltantes.
  El parámetro `mfinal = 9` define el número de modelos o iteraciones que se crearán en el proceso de Bagging. Aquí se construirán 9 árboles de decisión.
  El parámetro `control = rpart.control(cp = 0.001, minsplit = 7)` personaliza la construcción del árbol. `cp = 0.001` define el parámetro de complejidad de poda (complexity parameter). Indica el umbral mínimo para la mejora relativa en la reducción del error. Los nodos que no cumplan con este umbral serán podados. `minsplit = 7` establece el número mínimo de observaciones requeridas en un nodo para que pueda dividirse. Aquí se requiere al menos 7 observaciones para intentar una división.

Luego, para evaluar el rendimiento del modelo, se realiza lo siguiente:

```
matrizconfusionBag <- table(dtest[, "label"],predict(Modelo_AdaBag,
    newdata = dtest, type = "class")$class)
```

```
accuracyBag <- sum(diag(matrizconfusionBag)) / sum(matrizconfusionBag)

print(matrizconfusionBag)
cat("Accuracy del modelo:", accuracyBag, "\n")
cat("Tiempo de ejecucion Bagging:", end_time-start_time, "\n")
```

Estas líneas evalúan el modelo utilizando los datos de prueba (`dtest`):

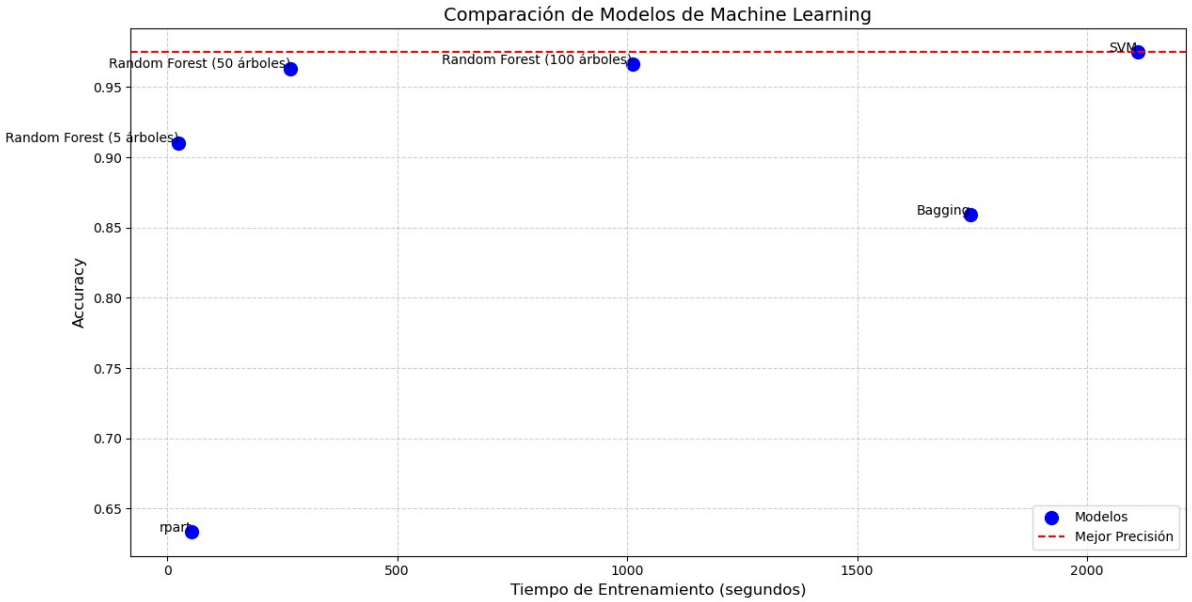
- `predict(ModeloAdaBag, newdata = dtest, type = çlass")` genera las predicciones del modelo para los datos de prueba del conjunto `dtest`.
- Usamos la función `table()` sobre el `predict()` anterior con parámetro `dtest$label` para crea la matriz de confusión comparando las predicciones con las etiquetas reales.
- `sum(diag(matrizconfusionBag)) / sum(matrizconfusionBag)` calcula la precisión del modelo como la proporción de predicciones correctas en comparación con el total de predicciones realizadas.

Los resultados obtenidos para este modelo son los siguientes:

- **Precisión**: 0.8594
- **Tiempo de entrenamiento**: 29.1157 min

7. Conclusión

Hemos decidido comparar los modelos utilizados mediante una gráfica de tipo *scatter* (dispersión), en la cual representamos dos métricas clave de rendimiento: la **precisión** (*accuracy*) y el **tiempo de entrenamiento**. Este enfoque permite visualizar de manera clara el equilibrio entre el desempeño predictivo y la eficiencia computacional de cada modelo. El resultado de esta comparación se refleja en la siguiente gráfica.



En este estudio se compararon diversos modelos de Machine Learning en términos de precisión (*Accuracy*) y tiempo de entrenamiento. A continuación, se comentan los resultados principales:

- **SVM (Support Vector Machine):** Este modelo obtuvo la mayor precisión entre todos los evaluados, alcanzando un valor cercano al límite superior de la métrica en el gráfico. Sin embargo, su tiempo de entrenamiento fue el más alto, lo que podría ser una desventaja en escenarios donde la eficiencia computacional es crítica.
- **Random Forest:**
  - **Con 50 árboles:** Este modelo mostró una precisión cercana a la del SVM, destacándose como una alternativa viable cuando se busca un buen equilibrio entre precisión y tiempo de entrenamiento.
  - **Con 100 árboles:** Aunque mejoró ligeramente la precisión respecto a la configuración de 50 árboles, el incremento en el tiempo de entrenamiento no es proporcional, lo que puede no justificar su uso en comparación con la configuración de 50 árboles.
  - **Con 5 árboles:** Este modelo tiene un tiempo de entrenamiento significativamente menor, pero a costa de una reducción en la precisión. Es adecuado si se busca rapidez y se pueden tolerar sacrificios en el rendimiento predictivo.

- **Bagging:** Aunque obtuvo una precisión aceptable, su tiempo de entrenamiento es considerablemente mayor que el de otros modelos con precisión similar, como el Random Forest (50 árboles). Esto lo hace menos competitivo en este caso.
- **rpart (Árbol de Decisión):** Este modelo fue el más rápido de todos, pero su precisión fue significativamente más baja en comparación con el resto. Es útil únicamente en escenarios donde el tiempo de entrenamiento sea la prioridad absoluta y la precisión no sea crítica.

El modelo más indicado depende del balance entre precisión y tiempo de entrenamiento según los requerimientos del problema:

- Si el objetivo principal es maximizar la precisión y el tiempo de entrenamiento no es un limitante, el modelo **SVM** es la mejor opción.
- Si se busca un compromiso entre alta precisión y eficiencia computacional, el **Random Forest con 50 árboles** se presenta como una alternativa más balanceada.
- En casos donde la rapidez del entrenamiento es crítica, el modelo **rpart** puede ser utilizado, aunque con sacrificios importantes en precisión.

En resumen, para aplicaciones prácticas, recomendamos el uso del **Random Forest con 50 árboles**, ya que combina un rendimiento predictivo competitivo con un tiempo de entrenamiento razonable, siendo una opción versátil para diversos escenarios.