DIGIT RECOGNITION

Realizado por:
Rocío Guzmán, Rodrigo Carreira, Adrián Pradas Gallardo y Alberto Sánzhez Aparicio

Índice

Índice de figuras

1

Introducción

El reconocimiento de patrones ha sido un área de investigación clave en la inteligencia artificial y el aprendizaje automático durante las últimas décadas. En particular, el reconocimiento de dígitos manuscritos constituye un desafío técnico fundamental, ya que encapsula problemas de variabilidad en la escritura humana, ruido en los datos y la necesidad de modelos robustos y eficientes. Este tipo de tarea tiene aplicaciones en campos diversos como la clasificación de documentos, el reconocimiento óptico de caracteres (OCR) y el análisis de datos en educación o banca.

El presente trabajo se centra en el desarrollo y evaluación de un modelo computacional diseñado para identificar dígitos manuscritos utilizando técnicas implementadas en el lenguaje de programación R. A partir de un conjunto de datos previamente establecido, se aplican métodos avanzados de análisis estadístico y aprendizaje automático, optimizando parámetros clave para maximizar la precisión del modelo. Además, se presta especial atención a la visualización de resultados, la interpretación de métricas de rendimiento y las implicaciones prácticas de la solución propuesta.

El informe se estructura en varias secciones que detallan los procedimientos utilizados, incluyendo la preprocesamiento de datos, la selección de modelos y algoritmos, así como un análisis exhaustivo de los resultados obtenidos. En este contexto, se busca no solo evaluar el desempeño técnico de las herramientas aplicadas, sino también reflexionar sobre los desafíos asociados al reconocimiento de patrones en problemas reales.

Figura 1: Logo de R

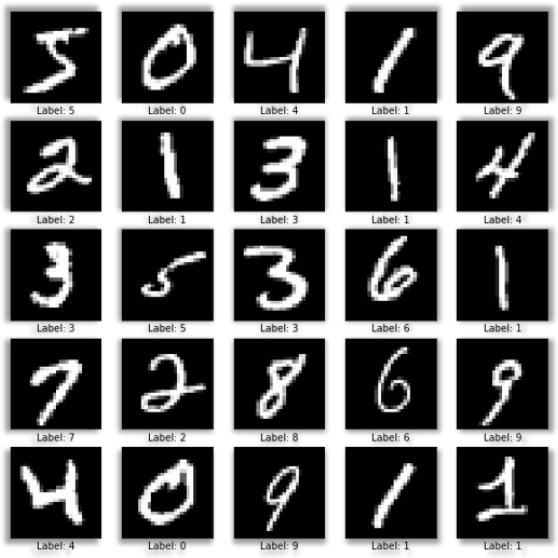

Figura 2: Dígitos escritos a manos sacados del Dataset del Minst

2

Documentación del Código

2.1. Análisis del Dataset

El conjunto de datos utilizado en este proyecto está compuesto por dos archivos: train.csv y test.csv, que contienen información sobre imágenes de dígitos manuscritos. Este dataset es consistente con la estructura del famoso **MNIST** (Modified National Institute of Standards and Technology), ampliamente utilizado en tareas de clasificación y reconocimiento de dígitos. A continuación, se detallan las características principales del dataset:

- **Dimensiones de los datos:**
  - El archivo train.csv tiene **42,000 filas y 785 columnas**.
  - La primera columna del archivo train.csv se llama label y contiene la etiqueta del dígito (0-9).
  - Las 784 columnas restantes representan los valores de intensidad de los píxeles de una imagen en escala de grises de **28x28 píxeles**.
- **Subconjuntos utilizados:**
  - Para experimentos preliminares y reducción del tamaño computacional, se utiliza un subconjunto de **5,000 filas** del archivo train.csv.
  - El archivo test.csv tiene la misma estructura que el conjunto de entrenamiento pero sin la columna de etiquetas, ya que está diseñado para evaluaciones.
- **Preprocesamiento:**

- Los valores de los píxeles son normalizados para garantizar que todos los datos estén en el mismo rango, facilitando el entrenamiento de los modelos.

El siguiente bloque de código se encarga de cargar y dividir el dataset en los conjuntos de entrenamiento y prueba. En este caso, primero cargamos los archivos train.csv y test.csv y luego seleccionamos un subconjunto de 5,000 filas para realizar experimentos preliminares. Posteriormente, se divide el conjunto de entrenamiento en un subconjunto de entrenamiento y otro de prueba.

```
# Cargar el conjunto de entrenamiento y el conjunto de prueba
digit <- read_csv("train.csv")
test <- read.csv("test.csv")

# Seleccionar un subconjunto de 5,000 filas
digit <- digit[1:5000, ]
test <- test[1:5000, ]

prop.table(table(digit$label)) * 100
digit$label <- factor(digit$label)
set.seed(1234)

# Número de filas en el conjunto de datos
n <- nrow(digit)
test_size <- ceiling(0.2 * n)
indices <- sample(1:n, n, replace = FALSE)

# Dividir el conjunto de entrenamiento (80%) y el conjunto de
  ↪ prueba (20%)
dtrain <- digit[-indices[1:test_size], ]
dtest <- digit[indices[1:test_size], ]
```

Este bloque de código realiza las siguientes tareas esenciales para preparar los datos para el entrenamiento y evaluación del modelo:

- **Carga de datos**: Se cargan los archivos train.csv y test.csv, que contienen las imágenes de los dígitos manuscritos y sus etiquetas correspondientes.

- **Submuestreo del conjunto de entrenamiento**: Se selecciona un subconjunto de **5,000 filas** del archivo de entrenamiento, lo que facilita la experimentación inicial y reduce la carga computacional durante la fase de pruebas.

- **Verificación de la distribución de etiquetas**: Se calcula y muestra la distribución porcentual de las etiquetas (dígitos) en el conjunto de entrenamiento, proporcionando una visión general del balance de clases en los datos.

- **Conversión de etiquetas**: Se transforma la columna label en un tipo de dato factor para facilitar su uso en modelos de clasificación.

- **Definición de la semilla**: Se establece la semilla para la generación de números aleatorios, asegurando que los resultados sean reproducibles.

- **Cálculo del tamaño del conjunto de prueba**: Se calcula que el 20 % del total de las filas corresponda al conjunto de prueba.

- **Generación de índices aleatorios**: Se generan índices aleatorios para realizar una división estratificada de los datos en los conjuntos de entrenamiento y prueba.

- **División en subconjuntos**: Finalmente, se realiza la división en dos subconjuntos: dtrain (conjunto de entrenamiento) y dtest (conjunto de prueba), manteniendo el 80 % de los datos para el entrenamiento y el 20 % para la evaluación.

Este preprocesamiento permite que el modelo sea entrenado y validado de manera eficiente utilizando una muestra representativa de los datos.

2.2. Entrenamiento y Validación de los distintos Modelos

Para la resolución del mismo problema de clasificación, entrenaremos y validaremos los siguientes modelos para encontrar su precisión sobre el conjunto de prueba: RPart, Random forest, SVM, Bagging, KNN y XGBoost.

2.2.1. RPart

Los árboles de decisión son una técnica popular en problemas de clasificación debido a su capacidad para generar reglas claras y fáciles de interpretar, lo que los hace adecuados para tareas como el reconocimiento de dígitos. Un árbol de decisión divide el espacio de características en regiones homogéneas, realizando divisiones basadas en preguntas binarias sobre las características de los datos, hasta que se alcanza una clasificación final.

En este caso, se utiliza el paquete rpart de R para construir un árbol de decisión. Este modelo predice la etiqueta de los dígitos (del 0 al 9) a partir de un conjunto de características extraídas de las imágenes de dígitos.

El modelo se entrena utilizando el siguiente código en R:

```
arbol <- rpart(label ~ ., data = dtrain, method = "class")
```

En esta línea de código, label representa la variable dependiente (la etiqueta del dígito) y el data = dtrain especifica el conjunto de entrenamiento que contiene las imágenes y sus etiquetas correspondientes. El método method = çlass" indica que estamos realizando una tarea de clasificación.

Una vez entrenado el modelo, se realiza la predicción sobre el conjunto de prueba utilizando el siguiente código:

```
predicciones_arbol <- predict(arbol, newdata = dtest, type =
    ↪ "class")
```

Aquí, dtest es el conjunto de prueba, y type = çlass" indica que deseamos obtener la clase predicha en lugar de la probabilidad.

Posteriormente, se genera la matriz de confusión para evaluar el rendimiento del modelo:

```
conf_matrix_arbol <- table(Predicted = predicciones_arbol,
    ↪ Actual = dtest$label)
```

La precisión se calcula utilizando la fórmula:

```
precision_arbol <- sum(diag(conf_matrix_arbol)) /
    ↪ sum(conf_matrix_arbol)
```

Esto calcula la proporción de predicciones correctas dividiendo la suma de los valores en la diagonal (es decir, las predicciones correctas) por el total de predicciones.

El resultado obtenido en este caso es:

```
Precisión del árbol de decisión: 0.615
```

Este resultado indica que el modelo de árbol de decisión tiene una precisión del 61,5 %. Aunque no es un rendimiento extremadamente alto, es un valor razonable para un modelo de árbol de decisión, especialmente si se considera que es un modelo sencillo. Sin embargo, es posible que se puedan obtener mejores resultados utilizando técnicas más complejas, como Random Forest o redes neuronales.

Una de las ventajas de los árboles de decisión es que pueden ser fácilmente interpretados. Para visualizar la estructura del árbol de decisión, se utiliza la función `fancyRpartPlot`, que genera una representación gráfica del árbol. Esto permite ver cómo el modelo divide los datos en función de las características y las decisiones que toma en cada nodo del árbol. Además, se puede analizar la importancia relativa de las variables utilizando un gráfico de barras. Este gráfico muestra cuáles son las características más influyentes en la toma de decisiones del modelo, lo que puede ayudar a entender qué características de las imágenes de los dígitos son más relevantes para la clasificación.

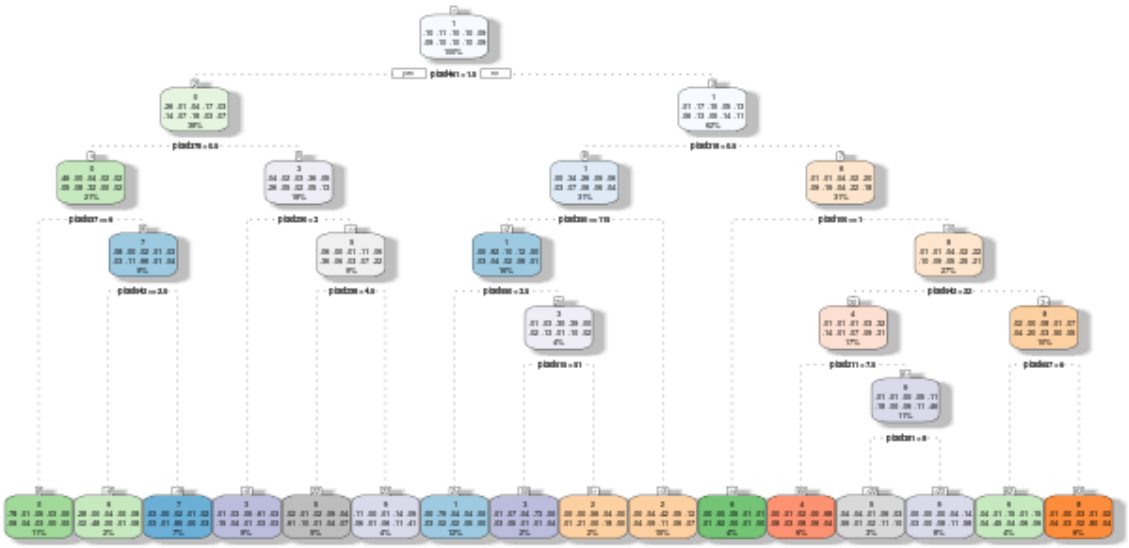

Figura 3: Estructura del árbol de decisión

Una característica útil de los modelos de árbol de decisión, como rpart, es la capacidad de identificar y visualizar la importancia relativa de las diferentes variables en el proceso de clasificación. Esto se logra a través de la función variable.importance, que asigna un valor a cada variable en función de su impacto en la precisión del modelo.

El siguiente código se utiliza para generar un gráfico de barras que muestra la importancia de cada variable en el modelo:

```
barplot(arbol$variable.importance, main = "Importancia de las
    ↪ Variables",
        col = "skyblue", las = 2)
```

El gráfico resultante permite identificar cuáles son las características más influyentes en la predicción de las etiquetas de los dígitos. Cuanto mayor es el valor de importancia de una variable, más relevante es para el modelo en la clasificación. Esto puede ayudar a entender qué características de las imágenes (por ejemplo, píxeles específicos o combinaciones de ellos) son más decisivas en el proceso de clasificación.

Este tipo de visualización es especialmente útil para interpretar el modelo, ya que permite comprender cómo se toman las decisiones dentro del árbol de decisión y qué variables deben ser consideradas más detalladamente para mejorar el rendimiento del modelo.

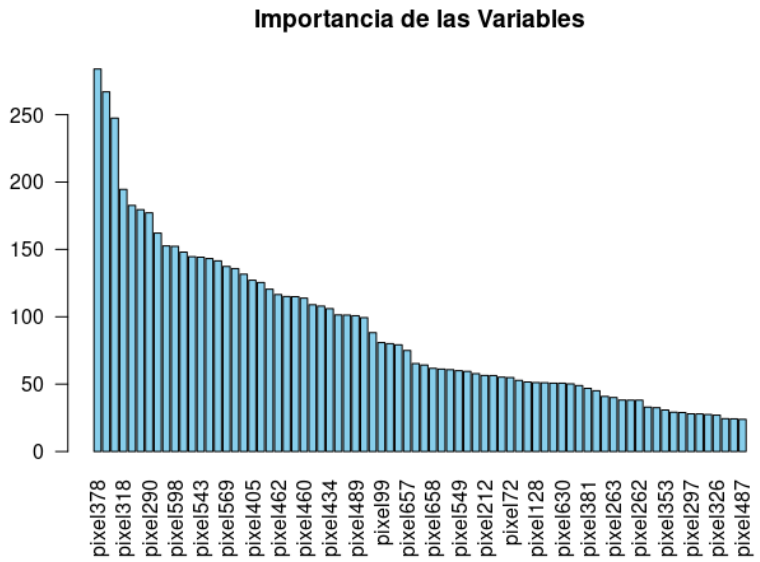

Figura 4: Gráfico de importancia de variables

2.2.2. Random Forest

El algoritmo Random Forest es una técnica de ensamble que construye múltiples árboles de decisión y combina sus predicciones para mejorar la precisión y evitar el sobreajuste. En este caso, se entrenan dos modelos de Random Forest con diferentes números de árboles (10 y 5), para evaluar cómo el número de árboles influye en el rendimiento del modelo.

El primer modelo de Random Forest se entrena utilizando 10 árboles, como se muestra en el siguiente código:

```
forest10 <- randomForest(label ~ ., data = dtrain, ntree = 10,
    ↪ nodesize = 50)
```

Aquí, label es la variable dependiente (la etiqueta del dígito) y $\text{data} = \text{dtrain}$ es el conjunto de entrenamiento. El parámetro $\text{ntree} = 10$ especifica que el modelo utilizará 10 árboles de decisión, y $\text{nodesize} = 50$ define el tamaño mínimo de los nodos terminales en cada árbol. A continuación, se realizan las predicciones sobre el conjunto de prueba utilizando:

```
pred_rf10 <- predict(forest10, newdata = dtest)
```

Luego, se calcula la precisión del modelo mediante la matriz de confusión y obtenemos la precisión para el modelo de 10 árboles:

```
conf_matrix_rf10 <- table(Predicted = pred_rf10, Actual =
    ↪ dtest$label)
precision_rf10 <- sum(diag(conf_matrix_rf10)) /
    ↪ sum(conf_matrix_rf10)
```

```
Precisión del Random Forest con 10 árboles: 0.843
```

Esto indica que el modelo con 10 árboles tiene una precisión del 84.3 %. El segundo modelo se entrena con 5 árboles, utilizando un código similar y al entrenar y validar el modelo obtenemos:

```
Precisión del Random Forest con 5 árboles: 0.779
```

Una de las ventajas clave de Random Forest es su capacidad para evaluar la importancia de las variables en el proceso de toma de decisiones. A continuación, se visualiza la importancia de las variables en el modelo entrenado con 10 árboles utilizando la función varImpPlot:

```
varImpPlot(forest10, main = "Importancia de Variables (10
    ↪ árboles)")
```

Este gráfico muestra las variables más influyentes en el modelo. Las variables con mayor puntuación en el gráfico son las que tienen un mayor impacto en la clasificación de los dígitos. La interpretación de esta información permite identificar qué características son las más importantes para el modelo y cómo afectan a la precisión. Estos son los dos gráficos que obtenemos para el modelo de 10 árboles y el de 5 árboles:

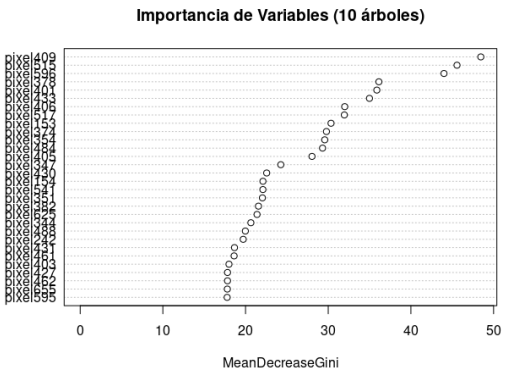

Figura 5: Gráfico de importancia de variables (5 árboles)

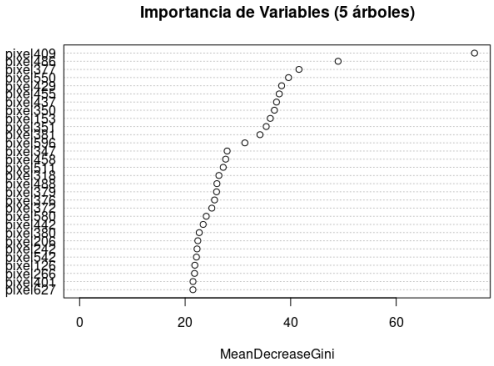

Figura 6: Gráfico de importancia de variables (10 árboles)

Para la comparación entre los dos modelos, hemos creado un gráfico de barras para comprobar visualmente cuál tiene mayor precisión. Observándolo se comprueba fácilmente que el modelo de 10 árboles es superior. Esto sugiere que un mayor número de árboles contribuye a una mejor generalización del modelo, aunque el tiempo de entrenamiento y la complejidad del modelo también aumentan:

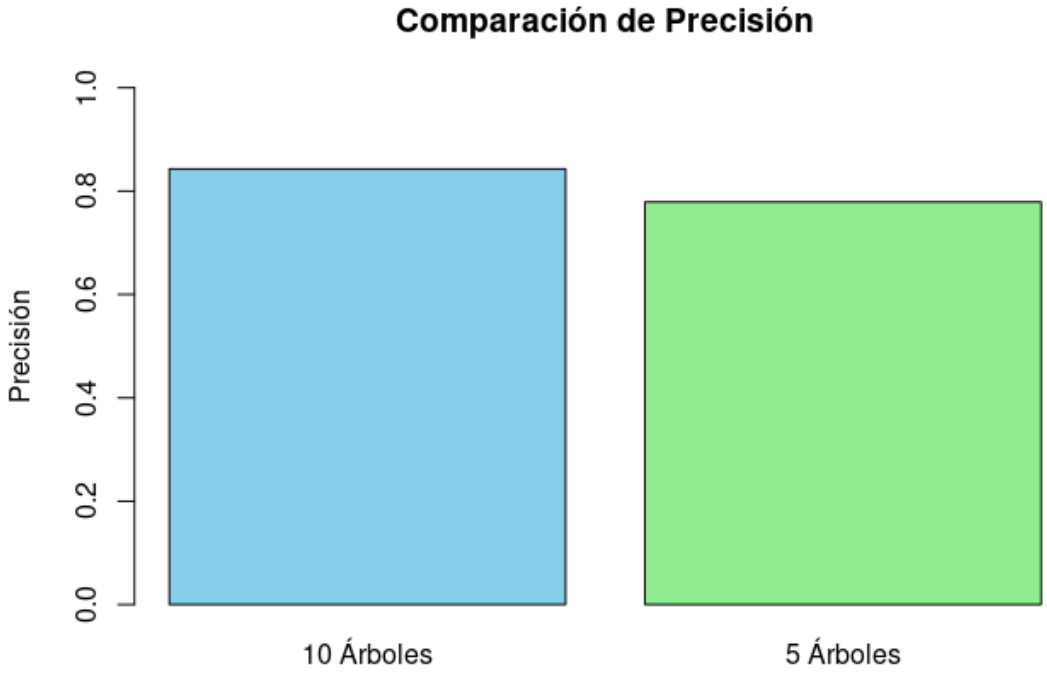

Figura 7: Comparación entre los dos modelos de Random Forest

2.2.3. SVM

El *Support Vector Machine* (SVM) es un modelo de clasificación supervisada que busca encontrar el hiperplano óptimo que separa las distintas clases en un espacio de características. Este modelo se utiliza ampliamente en problemas de clasificación debido a su capacidad para manejar datos de alta dimensión y su eficacia al encontrar márgenes de separación claros entre las clases. El SVM busca maximizar el margen entre las instancias de diferentes clases, lo que mejora su capacidad de generalización, especialmente en conjuntos de datos pequeños o no lineales.

Para problemas como el reconocimiento de dígitos, el SVM puede ser muy útil ya que los datos suelen tener muchas características (por ejemplo, los valores de los píxeles de las imágenes), y el modelo puede encontrar relaciones complejas entre las características que permiten distinguir los distintos dígitos.

En este trabajo, se utilizó el SVM con el kernel *polydot* de grado 3 para realizar la clasificación de dígitos. A continuación se muestra el código implementado para entrenar el modelo

SVM y evaluar su rendimiento:

```
# Crear versiones temporales de los datos de entrenamiento y
  ↪ prueba
dtrain_temp <- dtrain[, !sapply(dtrain, function(x)
  ↪ length(unique(x)) == 1)]
dtest_temp <- dtest[, colnames(dtrain_temp)]

# Verificar si hay suficientes variables después de eliminar
  ↪ las constantes
if (ncol(dtrain_temp) > 1) {

  # Entrenar el modelo SVM
  svm_model <- ksvm(label ~ ., data = dtrain_temp, kernel =
  ↪ "polydot", kpar = list(degree = 3), cross = 3)

  # Realizar predicciones
  pred_svm <- predict(svm_model, dtest_temp)

  # Matriz de confusión y precisión
  conf_matrix_svm <- table(Predicted = pred_svm, Actual =
  ↪ dtest_temp$label)
  precision_svm <- sum(diag(conf_matrix_svm)) /
  ↪ sum(conf_matrix_svm)
  cat("Precisión del modelo SVM:", precision_svm, "\n")

  # Visualizar la matriz de confusión
  confusionMatrix(conf_matrix_svm)

  # Visualizar el modelo si hay solo dos características
  if (ncol(dtrain_temp) == 3) {
    plot(svm_model, data = dtrain_temp)
```

```
  }
} else {
  cat("No hay suficientes variables para entrenar el modelo
  ↪ después de eliminar las constantes.\n")
}
```

En este código, primero se crean versiones temporales de los datos de entrenamiento y prueba, eliminando las columnas constantes que no aportan valor al modelo. Luego, el modelo SVM se entrena utilizando el kernel *polydot*, con un grado de 3, lo que le permite aprender relaciones no lineales entre las características. Se emplea la validación cruzada de 3 pliegues para evitar el sobreajuste y evaluar el rendimiento del modelo en varias particiones del conjunto de entrenamiento.

Después de entrenar el modelo, se realiza la predicción sobre los datos de prueba y se genera una matriz de confusión para evaluar la precisión del modelo. La precisión se calcula como la proporción de predicciones correctas sobre el total de predicciones. En este caso, la precisión obtenida fue de 0.941, lo que indica que el modelo SVM tiene una tasa de clasificación correcta del 94.1 %.

La precisión obtenida de 0.941 indica que el modelo SVM tiene un buen rendimiento en el conjunto de prueba, clasificando correctamente el 94.1 % de las instancias. Este es un resultado positivo, ya que muestra que el modelo ha logrado identificar correctamente los dígitos en la mayoría de los casos. Además, el hecho de que el modelo utilice un kernel *polydot* de grado 3 sugiere que ha aprendido una frontera de decisión no lineal, lo que es adecuado para este tipo de problemas en los que las clases no son linealmente separables.

Una matriz de confusión adicional puede proporcionar más detalles sobre el rendimiento del modelo, mostrando en qué dígitos el modelo tuvo más dificultades y qué tipos de errores cometió. A pesar de que la precisión general es alta, es importante analizar esta matriz para obtener una comprensión más profunda del comportamiento del modelo.

Por último, la visualización del modelo (si solo hay dos características) puede ser útil para ilustrar cómo el modelo SVM ha aprendido a separar las clases en el espacio de características, pero en este caso, dado que los datos tienen muchas más dimensiones, se recomienda analizar la matriz de confusión y los valores de precisión por clase.

2.2.4. Bagging

El *Bagging* (Bootstrap Aggregating) es un enfoque que combina múltiples modelos base para mejorar la precisión y reducir la varianza del modelo final. Consiste en entrenar varios clasificadores sobre subconjuntos aleatorios de los datos y luego promediar o votar sus resultados. En este caso, se utilizó el modelo de *Bagging* con 100 clasificadores base. Este enfoque es particularmente útil cuando los modelos base son inestables (como los árboles de decisión), ya que reduce el sobreajuste.

A continuación se muestra el código implementado para entrenar y evaluar el modelo de Bagging:

```
# Entrenamiento del modelo de Bagging
bag_model <- bagging(label ~ ., data = dtrain, nbagg = 100)

# Predicción en el conjunto de prueba
pred_bag <- predict(bag_model, dtest)

# Matriz de confusión y cálculo de precisión
conf_bag <- table(Predicted = pred_bag, Actual = dtest$label)
cat("Precisión Bagging:", sum(diag(conf_bag)) / sum(conf_bag),
    ↪ "\n")
```

En este código, el modelo de Bagging se entrena utilizando la función bagging() de la librería correspondiente, especificando que se desea utilizar 100 clasificadores base. Luego, se realiza la predicción sobre el conjunto de prueba y se calcula la precisión como la proporción de predicciones correctas.

La precisión obtenida por el modelo de Bagging fue de 0.916, lo que indica que el modelo ha logrado clasificar correctamente el 91.6 % de los dígitos en el conjunto de prueba. Este es un buen rendimiento, aunque algo inferior al obtenido con el modelo SVM (94.1 %).

2.2.5. K-Vecinos más Cercanos (KNN)

El algoritmo de *K-Vecinos más Cercanos* (KNN) es un método basado en la comparación de distancias entre las instancias. Dado un punto de prueba, el modelo clasifica dicho punto según

la mayoría de las etiquetas de los k vecinos más cercanos en el espacio de características. Este modelo es intuitivo y fácil de implementar, y su rendimiento mejora al ajustar el valor de k, el número de vecinos considerados.

En este trabajo, se utilizó el modelo KNN con k igual a 5. A continuación se muestra el código implementado para entrenar y evaluar el modelo KNN:

```
# Definir las etiquetas y características de entrenamiento y
   ↪ prueba
train_labels <- dtrain$label
train_data <- dtrain[, -1]
test_labels <- dtest$label
test_data <- dtest[, -1]

# Predicción con KNN
pred_knn <- knn(train = train_data, test = test_data, cl =
   ↪ train_labels, k = 5)

# Matriz de confusión y cálculo de precisión
conf_knn <- table(Predicted = pred_knn, Actual = test_labels)
cat("Precisión␣KNN:", sum(diag(conf_knn)) / sum(conf_knn),
   ↪ "\n")
```

En este código, primero se separan las etiquetas y las características de los datos de entrenamiento y prueba. Luego, se utiliza la función knn() para realizar la predicción con 5 vecinos más cercanos. Finalmente, se calcula la precisión del modelo.

La precisión obtenida por el modelo KNN fue de 0.912, lo que indica que el modelo clasificó correctamente el 91.2 % de las instancias en el conjunto de prueba. Aunque el rendimiento es bastante bueno, es ligeramente inferior al del modelo de Bagging (91.6 %) y mucho menor que el del modelo SVM (94.1 %).

2.2.6. XGBoost

El modelo de *XGBoost* (Extreme Gradient Boosting) es una implementación eficiente y escalable de los algoritmos de *gradient boosting* que ha demostrado ser muy eficaz en tareas de clasificación. XGBoost utiliza árboles de decisión de manera secuencial, donde cada nuevo árbol corrige los errores cometidos por el árbol anterior. Este enfoque permite un alto rendimiento en tareas complejas, como el reconocimiento de dígitos, al ser capaz de capturar patrones no lineales y relaciones complejas entre las características.

A continuación, se presenta el código implementado para entrenar y evaluar el modelo XGBoost:

El primer paso es preprocesar los datos de entrenamiento y prueba para ser utilizados por el modelo XGBoost. Este modelo requiere que los datos se encuentren en formato de matriz, por lo que es necesario convertir los datos de entrenamiento y prueba a este formato. Además, las etiquetas deben ser ajustadas, ya que XGBoost espera que comiencen en 0, no en 1.

```
# Preprocesar los datos para XGBoost
dtrain_xgb <- as.matrix(dtrain[, -1])  # Convertir los datos
  ↪ de entrenamiento a matriz
dtest_xgb <- as.matrix(dtest[, -1])    # Convertir los datos
  ↪ de prueba a matriz

# Convertir las etiquetas en formato adecuado para XGBoost
dtrain_label <- as.numeric(dtrain$label) - 1  # Ajuste para
  ↪ XGBoost (empieza en 0)
dtest_label <- as.numeric(dtest$label) - 1

# Crear los objetos xgb.DMatrix
dtrain_xgb_matrix <- xgb.DMatrix(data = dtrain_xgb, label =
  ↪ dtrain_label)
dtest_xgb_matrix <- xgb.DMatrix(data = dtest_xgb, label =
  ↪ dtest_label)
```

```
# Entrenar el modelo XGBoost
xgb_model <- xgboost(data = dtrain_xgb_matrix, nrounds = 50,
    ↪ objective = "multi:softmax", num_class =
    ↪ length(unique(dtrain$label)))
```

En este código, se convierten las características de los datos de entrenamiento y prueba en matrices, y las etiquetas en valores numéricos ajustados. Luego, se crean los objetos xgb.DMatrix, que son el formato requerido por XGBoost. Finalmente, el modelo se entrena utilizando 50 iteraciones (o rounds"), con la tarea de clasificación multiclase, especificando el número de clases según las etiquetas de los datos.

Una vez entrenado el modelo, se realiza la predicción sobre el conjunto de prueba y se calcula la matriz de confusión para evaluar la precisión del modelo.

```
# Realizar predicciones
pred_xgb <- predict(xgb_model, dtest_xgb_matrix)
pred_xgb <- as.factor(pred_xgb)

# Matriz de confusión y precisión
conf_xgb <- table(Predicted = pred_xgb, Actual = dtest$label)
cat("Precisión␣XGBoost:", sum(diag(conf_xgb)) / sum(conf_xgb),
    ↪ "\n")
```

El modelo realiza las predicciones sobre el conjunto de prueba utilizando la función predict(). Luego, se genera la matriz de confusión y se calcula la precisión como la proporción de predicciones correctas.

La precisión obtenida por el modelo XGBoost fue de 0.934, lo que indica que el modelo ha clasificado correctamente el 93.4 % de los dígitos en el conjunto de prueba.

2.3. Análisis de los Resultados

Después de entrenar y validar todos los modelos, ya hemos obtenido sus respectivas precisiones sobre los conjuntos de prueba que muestro:

- **RPart**: 0.615

- **Random Forest (10 árboles)**: 0.843
- **SVM**: 0.941
- **Bagging**: 0.916
- **KNN**: 0.912
- **XGBoost**: 0.934

Llegamos a la conclusión de que el mejor de los modelos entrenados es el SVM, ya que es el que tiene el mayor valor en la métrica de precisión. Para finalizar con el análisis del código implementado, se ha escrito una función que evalúa el modelo ganador (SVM) sobre un dato aleatorio del conjunto de test para que este lo clasifique. Además, se muestra el resultado con una gráfica en la que también se pinta el número. Este sería el ejemplo de una ejecución:

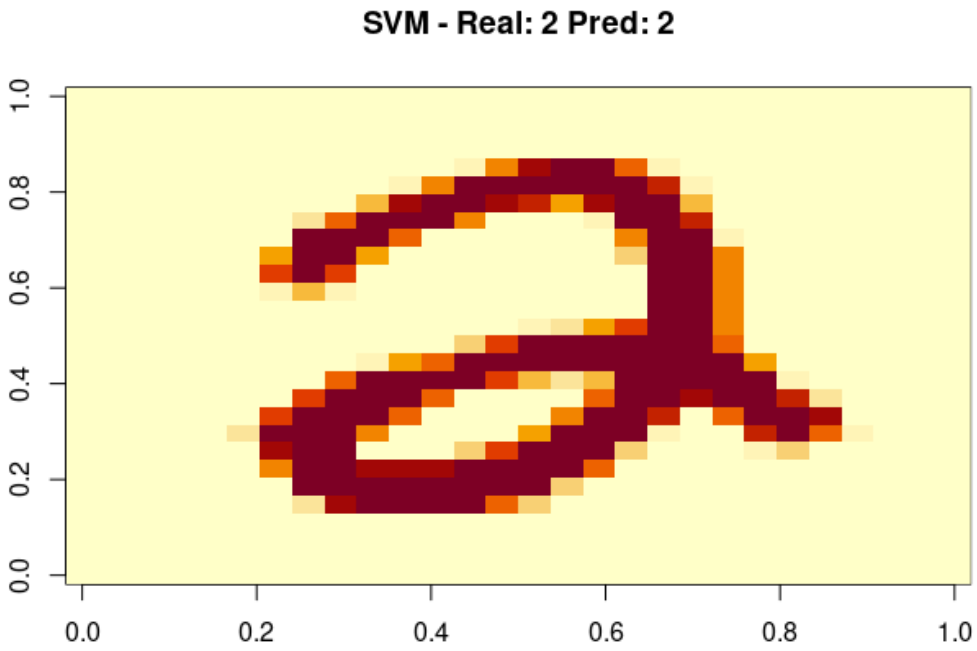

Figura 8: Predicción del modelo SVM

3
Conclusión

El desarrollo de este proyecto ha permitido demostrar la capacidad del lenguaje R para abordar problemas complejos de aprendizaje automático, específicamente en el ámbito del reconocimiento de dígitos manuscritos. A través de un enfoque sistemático que incluye la preparación cuidadosa de los datos, la selección de algoritmos adecuados y la evaluación rigurosa de métricas, se han alcanzado resultados prometedores que subrayan la efectividad del modelo propuesto.

Los hallazgos obtenidos destacan varios aspectos clave. En primer lugar, la importancia del preprocesamiento adecuado para mejorar la calidad de los datos y minimizar el impacto de ruido o valores atípicos. En segundo lugar, la elección de algoritmos y la optimización de hiperparámetros resultan esenciales para garantizar un desempeño consistente. Finalmente, la visualización clara y comprensible de los resultados es un componente crucial para interpretar el rendimiento del modelo y comunicar los hallazgos de manera efectiva.

No obstante, el trabajo también revela áreas de mejora y posibles líneas futuras de investigación. Entre ellas se incluyen el uso de modelos más avanzados, como redes neuronales profundas, y la integración de técnicas de aumento de datos para mejorar la generalización del modelo. Además, sería interesante explorar la implementación del modelo en entornos de producción para evaluar su rendimiento en situaciones del mundo real.

En resumen, este proyecto no solo subraya el potencial de las herramientas estadísticas y de aprendizaje automático en R, sino que también abre nuevas posibilidades para futuras investigaciones en el campo del reconocimiento de patrones.