

Segundo Parcial

Estudiantes:

Daniela Pinzón Callejas

Daniel Felipe Oviedo Trujillo

Miguel Ángel Thomas González

Materia:

Aprendizaje de Máquina:PCIA5011-G02

Universidad Sergio Arboleda

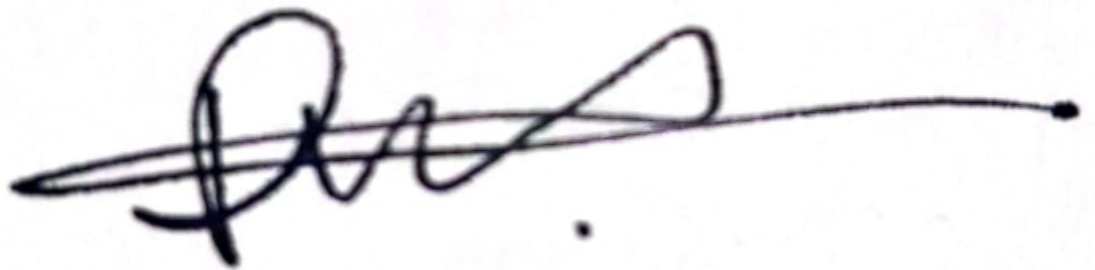


Escuela de Ciencias Exactas e Ingeniería
Ciencias de la Computación e Inteligencia Artificial

2024

Martes 30 de abril

"Certifico que todas las soluciones son enteramente de mi autoría y que no he consultado las soluciones de otro estudiante. He dado crédito a todas las fuentes externas que consulté para la solución del ejercicio".

A stylized handwritten signature in dark ink, featuring a large initial 'D' and a long horizontal stroke extending to the right.

Daniela Pinzón Callejas

A handwritten signature in black ink, with a large 'M' and 'G' and a horizontal line at the bottom.

Miguel Ángel Thomas González

A handwritten signature in dark ink, consisting of a stylized 'D' and 'F' followed by the text 'aniel O.' written in a cursive script.

Daniel Felipe Oviedo Trujillo

Introducción:

El presente informe es un análisis detallado del procedimiento seguido por nuestro equipo de trabajo para abordar el parcial propuesto. A lo largo de este documento, se describirán minuciosamente todas las funciones implementadas y las metodologías empleadas en la resolución del ejercicio. Este análisis tiene como objetivo no solo documentar los pasos y decisiones tomados durante el proceso, sino también justificar las elecciones metodológicas y técnicas realizadas, proporcionando un marco claro y estructurado para entender cómo se llegó a las soluciones finales. Además, se discutirán los desafíos enfrentados y cómo estos fueron superados por el equipo, evidenciando la aplicación práctica de los conocimientos teóricos adquiridos en el curso.

Procedimiento:

1. Importación de librerías:

```
import os
import rarfile
import requests
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.svm import SVC
from sklearn.metrics import accuracy_score
```

- **os**: es una librería que nos permitió el manejo, navegación y gestión de directorios para guardar los datos descargados en el almacenamiento local.
- **rarfile**: es una librería que nos permite extraer y descomprimir información de archivos '.rar'.
- **request**: esta librería nos permite gestionar peticiones HTTP en python y se usó para acceder a la URL de descarga del dataset.
- **numpy**: es una herramienta que nos facilita el uso y manipulación sencilla de matrices, así como también nos permite usar funciones matemáticas necesarias para la computación de alto rendimiento.
- **pandas**: esta herramienta nos permite el manejo y análisis de dataframes.
- **matplotlib**: es una herramienta gráfica que nos ofrece funciones para la visualización de gráficas.
- **sklearn**: esta es una librería usada para aprendizaje automático ya que nos ofrece modelos de aprendizaje como SVC (Support Vector Classifier).

2. Carga de datos y exploración del dataset

2.1. Carga de datos:

- **Función - 'load_mnist_data'**: La función load_mnist_data automatiza la descarga y extracción de estos datos desde un archivo comprimido y los convierte en DataFrames de Pandas. Estos DataFrames organizan los datos de entrenamiento, las etiquetas y los datos de prueba en formatos tabulares, facilitando su manipulación y análisis posterior.

```
def load_mnist_data(url, temp_dir='temp_data'):
```

Descarga un archivo RAR desde la URL dada, descomprime para extraer un archivo NPZ,

y carga los datos del conjunto de MNIST en DataFrames de Pandas.

Parámetros:

- url: URL del archivo .rar que contiene el archivo .npz.
- temp_dir: Directorio temporal para almacenar los archivos descargados y descomprimidos.

Retorna:

- Un diccionario con los DataFrames 'df_training_data', 'df_training_labels', y 'df_test_data',
o None si el archivo no se puede cargar.

"""

Asegurarse de que el directorio temporal exista

```
if not os.path.exists(temp_dir):  
    os.makedirs(temp_dir)
```

Ruta del archivo RAR descargado

```
rar_path = os.path.join(temp_dir, 'mnist-data.rar')
```

Descargar el archivo RAR

```
print("Descargando el archivo...")  
response = requests.get(url)  
with open(rar_path, 'wb') as file:  
    file.write(response.content)
```

Extraer y verificar el contenido del archivo RAR

try:

```
    with rarfile.RarFile(rar_path) as rf:  
        rf.extractall(path=temp_dir)  
        print("Archivos contenidos en el RAR:",
```

```
rf.namelist())
```

except rarfile.Error as e:

```
    print("Error al descomprimir el archivo RAR:", e)  
    return None
```

Suponiendo que el archivo se llama 'mnist_data.npz'

```
npz_path = os.path.join(temp_dir, 'mnist-data.npz')
```

Cargar los datos si el archivo NPZ existe

```

if os.path.exists(npz_path):
    with np.load(npz_path) as data:
        # Aplanar las imágenes y crear DataFrames
        df_training_data =
pd.DataFrame(data['training_data'].reshape(data['training_data'].
shape[0], -1))
        df_training_labels =
pd.DataFrame(data['training_labels'], columns=['Label'])
        df_test_data =
pd.DataFrame(data['test_data'].reshape(data['test_data'].shape[0]
, -1))

    print("Datos cargados exitosamente en DataFrames.")
    return {
        'df_training_data': df_training_data,
        'df_training_labels': df_training_labels,
        'df_test_data': df_test_data
    }
else:
    print(f"El archivo {npz_path} no se encontró después de
descomprimir.")
    return None

```

- INPUT:

```

url =
'https://github.com/danielal612022/Parcial_Machine_Learning/raw/m
ain/mnist-data.rar'
data = load_mnist_data(url)

if data:
    df_training_data = data['df_training_data']
    df_training_labels = data['df_training_labels']
    df_test_data = data['df_test_data']

    # Mostramos las dimensiones del conjunto de datos
    print("Training Data Shape:", df_training_data.shape)
    print("Training Labels Shape:", df_training_labels.shape)
    print("Test Data Shape:", df_test_data.shape)
else:
    print("Los datos no se pudieron cargar o procesar
correctamente.")

```

OUTPUT:

```
Descargando el archivo...
Archivos contenidos en el RAR: ['mnist-data.npz']
Datos cargados exitosamente en DataFrames.
Training Data Shape: (60000, 784)
Training Labels Shape: (60000, 1)
Test Data Shape: (10000, 784)
```

- **Revisión valores nulos:** Además de cargar y organizar los datos, realizamos una verificación exhaustiva para detectar valores nulos en cada DataFrame, confirmando que no existen valores nulos en ninguno de los conjuntos de datos (entrenamiento, etiquetas y prueba). Esta verificación asegura que los datos están completos y no requerirán tratamientos adicionales para manejar ausencias, lo cual es crucial para mantener la integridad del análisis y la eficacia del entrenamiento de modelos.

INPUT:

```
df_training_data.info()
OUTPUT:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Columns: 784 entries, 0 to 783
dtypes: float32(784)
memory usage: 179.4 MB

df_training_labels.info()
OUTPUT:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Data columns (total 1 columns):
#   Column  Non-Null Count  Dtype
---  -
0   Label   60000 non-null  int64
dtypes: int64(1)
memory usage: 468.9 KB

df_test_data.info()
OUTPUT:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 10000 entries, 0 to 9999
Columns: 784 entries, 0 to 783
dtypes: float32(784)
memory usage: 29.9 MB
```

2.2. Exploración del conjunto de datos:

- **Objetivo:** Entender la estructura, tipo y rango de los datos contenidos en los DataFrames `training_data`, `training_labels`, y `test_data`. Características de los Puntos de Muestra:
- **Datos de Entrenamiento y Prueba (`training_data`, `test_data`):** Cada imagen de MNIST está representada como un array de 784 píxeles (después de aplanar la imagen original de 28x28 píxeles). Cada valor en el array representa la intensidad de un píxel en escala de grises, donde 0 es negro y 255 es blanco.
- **Etiquetas de Entrenamiento (`training_labels`):** Cada etiqueta es un entero de 0 a 9, correspondiendo al dígito representado por la imagen asociada.
- **Revisión Inicial:** Verificamos si hay valores null en los datasets

INPUT:

```
dataframes = {
    'Training Data': df_training_data,
    'Training Labels': df_training_labels,
    'Test Data': df_test_data
}

# Recorrer cada DataFrame y contar los valores nulos
for name, df in dataframes.items():
    null_count = df.isnull().sum().sum() # Suma total de valores nulos
    en el DataFrame
    if null_count == 0:
        print(f"No hay valores nulos en {name}.")
    else:
        print(f"Hay {null_count} valores nulos en {name}.")
```

OUTPUT:

```
No hay valores nulos en Training Data.
No hay valores nulos en Training Labels.
No hay valores nulos en Test Data.
```

- **Escalada de los datos:** El análisis de los valores únicos en el conjunto de datos revela que las imágenes ya están escaladas en un rango de 0 a 1, lo que indica que los datos están normalizados. Esta normalización previa elimina la necesidad de realizar un escalado adicional, ya que hacerlo podría interferir negativamente con el entrenamiento del modelo. Este estado de los datos facilita el proceso de modelado, ya que trabajar con datos normalizados es esencial para la mayoría de los algoritmos de aprendizaje automático, permitiendo un entrenamiento más eficiente y evitando problemas comunes como el de los gradientes que desaparecen.

INPUT:

```
valores_unicos = np.unique(df_training_data)
```

```
print("Valores unicos del conjunto de datos: \n",valores_unicos)
OUTPUT (Pequeño muestreo del resultado original):
Valores unicos del conjunto de datos:
[0.          0.00392157 0.00784314 0.01176471 0.01568628 0.01960784
 0.02352941 0.02745098 0.03137255 0.03529412 0.03921569 0.04313726
 0.04705882 0.05098039 0.05490196 0.05882353 0.0627451  0.06666667
 0.07058824 0.07450981 0.07843138 0.08235294 0.08627451 0.09019608
 0.09411765 0.09803922 0.10196079 0.10588235 0.10980392 0.11372549
 0.11764706 0.12156863 0.1254902  0.12941177 0.13333334 0.13725491
 0.14117648 0.14509805 0.14901961 0.15294118 0.15686275 0.16078432
 0.16470589 0.16862746 0.17254902 0.1764706  0.18039216 0.18431373
 0.1882353  0.19215687 0.19607843 0.2          0.20392157 0.20784314
 0.9882353  0.99215686 0.99607843 1.          ]
```

2.3. Visualización de datos:

- **Verificación de la Integralidad de los Datos:** Utilizar np.unique para identificar todas las etiquetas únicas en el conjunto de entrenamiento asegura que todas las clases estén presentes y correctamente representadas. Este paso es fundamental para prevenir sesgos en el modelo debido a una distribución desequilibrada de las clases, lo cual es crucial para mantener la integridad y la fiabilidad del proceso de entrenamiento del clasificador.
- **Validación de Correspondencia de Datos y Etiquetas:** Al seleccionar una imagen de cada clase basada en las etiquetas y verificar su correspondencia, el código garantiza que las etiquetas sean precisas y estén bien asociadas con sus imágenes correspondientes. Este paso es vital para la verificación manual y asegura que no haya errores en la etiquetación, lo cual podría llevar a un entrenamiento incorrecto del modelo.
- **Facilitación de la Inspección Visual:** La función plot_images proporciona una herramienta visual para inspeccionar y verificar la calidad y características de las imágenes que se utilizarán para entrenar el modelo. La visualización de las imágenes con sus etiquetas correspondientes no solo ayuda en la verificación manual de los datos, sino que también facilita la comprensión y la demostración de la naturaleza de los datos a otros interesados o durante el análisis de datos.

INPUT:

```
unique_labels = np.unique(df_training_labels['Label'])
# Seleccionar la primera imagen para cada etiqueta única
selected_images = []
selected_labels = []
for label in unique_labels:
    # Encuentra el índice de la primera ocurrencia de cada etiqueta
    index = df_training_labels[df_training_labels['Label'] ==
label].index[0]
```

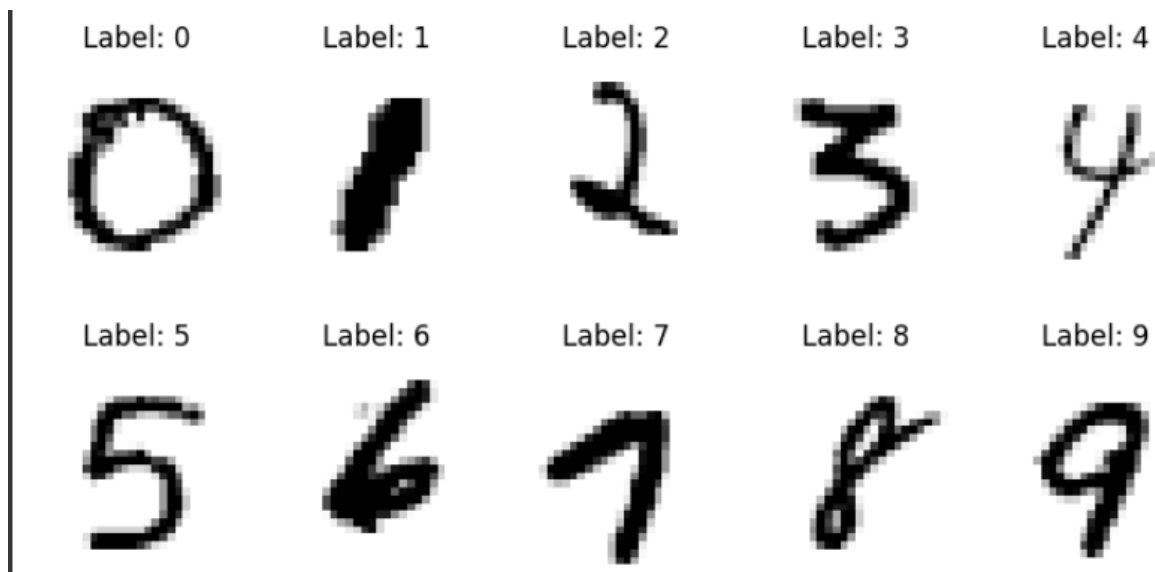


```

selected_images.append(df_training_data.iloc[index].values)
selected_labels.append(label)
def plot_images(images, labels, num_cols=5):
    """Función para mostrar una cuadrícula de imágenes MNIST con sus
    etiquetas."""
    num_rows = np.ceil(len(images) / num_cols).astype(int)
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(1.5*num_cols,
    2*num_rows))
    for i, ax in enumerate(axes.flat):
        if i < len(images):
            # Las imágenes están aplanadas y necesitan ser
            redimensionadas a 28x28 para visualización
            img_shape = (28, 28)
            ax.imshow(images[i].reshape(img_shape), cmap='binary')
            ax.set_title(f"Label: {labels[i]}")
            ax.axis('off')
    plt.tight_layout()
    plt.show()
# Llama a la función de trazado
plot_images(selected_images, selected_labels, num_cols=5)

```

OUTPUT:



3. División de los datos:

3.1. Mezclar los datos de entrenamiento: El barajado de los datos es esencial por varias razones:

- **Varianza de los Datos:** Al barajar los datos, aseguramos que cada partición que creamos (en este caso, el conjunto de entrenamiento y de validación) sea una representación estadísticamente válida del conjunto de datos

completo. Esto evita cualquier sesgo que pueda surgir debido al orden en que los datos fueron originalmente almacenados o recogidos.

- **Distribución Uniforme de Clases:** Es crucial especialmente en conjuntos de datos clasificados como MNIST, donde las clases (dígitos del 0 al 9) deben estar igualmente representadas en cada subconjunto de datos para evitar sesgos en el entrenamiento del modelo. El barajado asegura que todas las clases estén mezcladas de manera aleatoria y distribuidas uniformemente entre los conjuntos de entrenamiento y validación.
- **Prevención del Sobreajuste:** Barajar los datos antes de la partición también ayuda a prevenir el sobreajuste. Sin barajar, podrías terminar con particiones que no contienen variaciones representativas de los datos, haciendo que el modelo aprenda patrones específicos del conjunto de entrenamiento que no generalizan bien a datos nuevos.

INPUT:

```
# Convertimos DataFrames a numpy arrays para facilitar el manejo
training_data = df_training_data.to_numpy()
training_labels = df_training_labels.to_numpy()
# Crear un array de índices y barajarlos
indices = np.arange(df_training_data.shape[0])
np.random.shuffle(indices)
# Aplicar el orden barajado a los datos y etiquetas
shuffled_training_data =
df_training_data.iloc[indices].reset_index(drop=True)
shuffled_training_labels =
df_training_labels.iloc[indices].reset_index(drop=True)
```

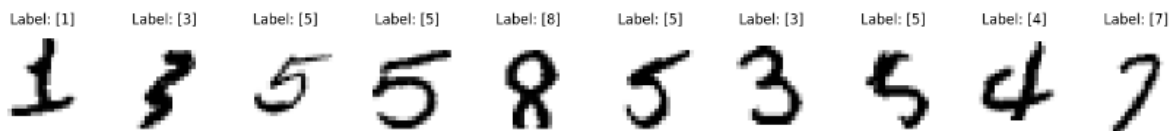
3.2. División de los datos: La partición de los datos en conjuntos de entrenamiento y validación tiene su propia importancia:

- **Evaluación Independiente:** Al separar un conjunto de validación del conjunto de entrenamiento, podemos evaluar el rendimiento del modelo de manera independiente durante el proceso de entrenamiento. Esto proporciona una estimación honesta de cómo el modelo podría comportarse en el conjunto de prueba y, eventualmente, en datos reales no vistos.
- **Afinamiento del Modelo:** El conjunto de validación se puede utilizar para ajustar los parámetros del modelo, seleccionar características, y tomar decisiones de diseño sin comprometer la integridad del conjunto de prueba. Esto asegura que las mejoras en el modelo se deben a verdaderas mejoras en la capacidad predictiva y no a un ajuste a las peculiaridades específicas del conjunto de prueba.
- **Control del Sobreajuste:** La partición permite monitorear si el modelo está empezando a sobreajustarse al conjunto de entrenamiento. Observando el rendimiento tanto en el conjunto de entrenamiento como en el de validación, podemos tomar medidas si vemos que el rendimiento en el conjunto de validación comienza a decaer mientras que el rendimiento en el conjunto de entrenamiento sigue mejorando.

INPUT:

```
sample_images = train_data.head(10).to_numpy()
sample_labels = train_labels.head(10).to_numpy()
def plot_sample_images(images, labels, num_rows=1, num_cols=10):
    fig, axes = plt.subplots(num_rows, num_cols, figsize=(1.5 *
num_cols, 2 * num_rows))
    for i, ax in enumerate(axes.flat):
        ax.imshow(images[i].reshape(28, 28), cmap='binary')
        ax.set_title(f"Label: {labels[i]}")
        ax.axis('off')
    plt.tight_layout()
    plt.show()
# Llamar a la función para visualizar las imágenes
plot_sample_images(sample_images, sample_labels)
```

OUTPUT:



Muestras para verificación Rápida

Selección de Muestras para Confirmación Post-Barajado:

- Extraer las primeras 10 imágenes y sus etiquetas correspondientes después de barajar asegura una verificación rápida para confirmar que el proceso de barajado se ha realizado correctamente. Al convertir estas muestras en arrays de NumPy, se facilita su manipulación para procesos de visualización.

Visualización para Verificación de Correspondencia:

- La función `plot_sample_images` está diseñada para mostrar claramente las imágenes junto con sus etiquetas, lo que es esencial para verificar que cada imagen todavía corresponde a su etiqueta correcta después del barajado. Esto es crucial para asegurar que el proceso de barajado no haya desalineado las imágenes de sus etiquetas, lo cual podría afectar negativamente el rendimiento del modelo de aprendizaje automático.

4. Implementación del algoritmo de Entrenamiento

- 4.1. Se entrena el modelo, la función encargada entrena el clasificador SVM lineal utilizando tamaños de muestra progresivamente mayores del conjunto de datos de entrenamiento. Se convierten los DataFrames a arrays de NumPy

para compatibilidad con scikit-learn y para facilitar operaciones de índice y cálculos.

El entrenamiento progresivo permite evaluar cómo afecta la cantidad de datos al rendimiento del modelo, proporcionando insights sobre el aprendizaje del modelo y la generalización. La conversión a NumPy es crucial para evitar problemas de compatibilidad y mejorar el rendimiento computacional durante el entrenamiento del modelo.

Input:

```
def train_svm_classifier(train_data, train_labels, valid_data,
valid_labels, training_sizes):
    train_accuracies = []
    valid_accuracies = []

    # Convertir DataFrames a arrays de NumPy si aún no están
convertidos
    train_data_np = train_data.to_numpy()
    train_labels_np = train_labels.values.ravel()
    valid_data_np = valid_data.to_numpy()
    valid_labels_np = valid_labels.values.ravel()

    for size in training_sizes:
        svm_classifier = SVC(kernel='linear')
        svm_classifier.fit(train_data_np[:size],
train_labels_np[:size])

        # Precisión en el conjunto de entrenamiento
        train_accuracy = accuracy_score(train_labels_np[:size],
svm_classifier.predict(train_data_np[:size]))
        train_accuracies.append(train_accuracy)

        # Precisión en el conjunto de validación
        valid_accuracy = accuracy_score(valid_labels_np,
svm_classifier.predict(valid_data_np))
        valid_accuracies.append(valid_accuracy)

    return training_sizes, train_accuracies, valid_accuracies
```

- 4.2. Se calcula la precisión en los conjuntos de Entrenamiento y Validación, dentro de la función se calcula y almacena la precisión del modelo tanto en el conjunto de entrenamiento como en el de validación para cada tamaño de muestra especificado.

La idea es poder medir la precisión en ambos conjuntos permite identificar si el modelo está sobreajustando (si la precisión en el conjunto de

entrenamiento es mucho más alta que en el de validación). Este seguimiento es esencial para optimizar el modelo y asegurar que sea robusto y fiable.

La función *plot_accuracies* gráfica las precisiones de entrenamiento y validación en función del tamaño de la muestra de entrenamiento. Las etiquetas del eje x son rotadas para mejorar la legibilidad, especialmente con muchos puntos de datos.

La visualización efectiva de los resultados es crucial para interpretar el comportamiento del modelo a medida que cambia el tamaño del conjunto de entrenamiento. Rotar las etiquetas mejora la claridad y facilita la interpretación rápida de los datos, lo cual es especialmente útil en presentaciones o análisis detallados

Input:

```
def plot_accuracies(training_sizes, train_accuracies,
valid_accuracies):
    plt.figure(figsize=(18, 4))
    plt.plot(training_sizes, train_accuracies, label='Precisión de
Entrenamiento', marker='o')
    plt.plot(training_sizes, valid_accuracies, label='Precisión de
Validación', marker='x')
    plt.title('Precisión del SVM Lineal en función del Tamaño de
Muestra de Entrenamiento')
    plt.xlabel('Número de Ejemplos de Entrenamiento')
    plt.ylabel('Precisión')
    plt.legend()
    plt.grid(True)
    plt.xticks(training_sizes, rotation=50) # Rotar las etiquetas del
eje x 45 grados
    plt.yticks(np.arange(0.5, 1.05, 0.05))
    plt.show()

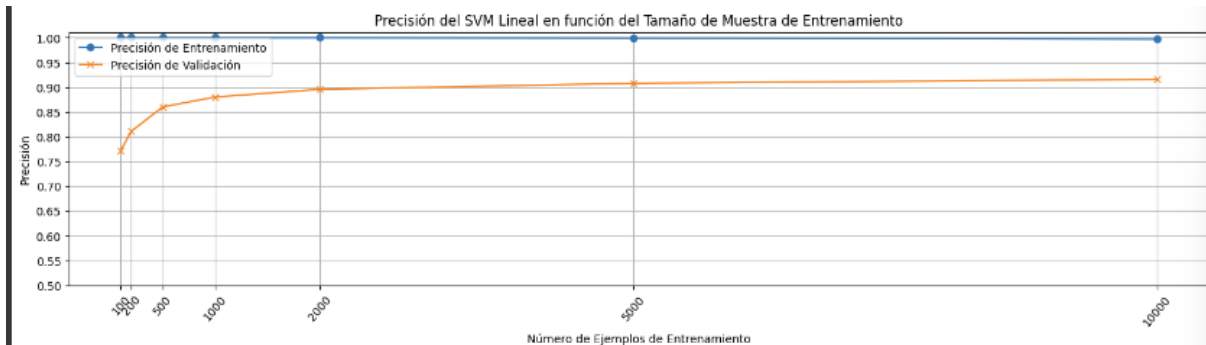
# Asumiendo que las variables de datos ya están definidas:
training_sizes = [100, 200, 500, 1000, 2000, 5000, 10000]

# Entrenar el clasificador SVM y obtener precisiones
training_sizes, train_accuracies, valid_accuracies =
train_svm_classifier(
    train_data, train_labels, valid_data, valid_labels, training_sizes)

# Imprimir las precisiones obtenidas
for size, train_acc, valid_acc in zip(training_sizes, train_accuracies,
valid_accuracies):
    print(f"Tamaño de muestra: {size}, Precisión de Entrenamiento:
{train_acc:.4f}, Precisión de Validación: {valid_acc:.4f}")
plot_accuracies(training_sizes, train_accuracies, valid_accuracies)
```

Output:

```
Tamaño de muestra: 100, Precisión de Entrenamiento: 1.0000, Precisión de Validación: 0.7718
Tamaño de muestra: 200, Precisión de Entrenamiento: 1.0000, Precisión de Validación: 0.8108
Tamaño de muestra: 500, Precisión de Entrenamiento: 1.0000, Precisión de Validación: 0.8600
Tamaño de muestra: 1000, Precisión de Entrenamiento: 1.0000, Precisión de Validación: 0.8800
Tamaño de muestra: 2000, Precisión de Entrenamiento: 1.0000, Precisión de Validación: 0.8955
Tamaño de muestra: 5000, Precisión de Entrenamiento: 0.9992, Precisión de Validación: 0.9078
Tamaño de muestra: 10000, Precisión de Entrenamiento: 0.9968, Precisión de Validación: 0.9158
```



Finalmente, el código ejecuta la función de entrenamiento y luego usa la función de visualización para mostrar los resultados, además de imprimir cada precisión obtenida.

Ejecutar estas funciones y visualizar los resultados proporciona una comprensión completa del impacto del tamaño del conjunto de entrenamiento en la precisión del modelo. Imprimir las precisiones da una referencia inmediata sobre el rendimiento y es útil para documentación o análisis posterior.

5. Conclusiones (Adicional)

El uso del conjunto de datos de prueba en proyectos de clasificación del MNIST abarca varias etapas críticas del proceso de aprendizaje automático, comenzando por la carga inicial de datos y culminando en la evaluación final del modelo. Las etapas clave incluyen:

Carga de Datos y Preparación Inicial:

- La carga efectiva del conjunto de datos MNIST marca el comienzo del proceso de análisis de datos. Es crucial garantizar que la carga de datos se realice de manera que preserve la integridad y la estructura del conjunto de datos, lo que permite un manejo eficiente y un preprocesamiento adecuado. La revisión de la correcta carga y formato de los datos es fundamental para evitar complicaciones en las etapas de procesamiento y análisis subsiguientes

Exploración y Preprocesamiento de Datos:

- Tras la carga, se realiza una exploración de los datos para identificar características clave, distribuciones y posibles anomalías. Este paso es esencial para comprender la naturaleza de los datos y para determinar las necesidades de preprocesamiento como la normalización, escalado o manipulaciones específicas que puedan ser necesarias para optimizar el rendimiento del modelo.

División de Datos en Conjuntos de Entrenamiento, Validación y Prueba:

- La división adecuada de los datos en conjuntos de entrenamiento, validación y prueba es crucial para un desarrollo de modelo equilibrado. El conjunto de entrenamiento permite al modelo aprender las características de los datos, mientras que el conjunto de validación ayuda a afinar los hiperparámetros y a realizar ajustes iterativos. El conjunto de prueba, siendo independiente, es vital para evaluar cómo el modelo entrenado generaliza a nuevos datos.

Entrenamiento y Ajuste del Modelo:

- El proceso de entrenamiento ajusta el modelo a los datos, mientras que la fase de ajuste de hiperparámetros optimiza su configuración. Estas etapas son cruciales para el desarrollo de un modelo robusto y eficiente. Durante estas fases, es esencial monitorizar el rendimiento del modelo para asegurar que no se produzca sobreajuste y para garantizar que el modelo mantiene una buena capacidad de generalización.

Evaluación Final Usando el Conjunto de Datos de Prueba:

- La evaluación final del modelo se realiza utilizando el conjunto de datos de prueba. Esta evaluación es decisiva, ya que proporciona una métrica objetiva del rendimiento del modelo en datos no vistos. La precisión, junto con otras métricas evaluadas en esta fase, es fundamental para determinar la viabilidad y eficacia del modelo en tareas de clasificación realistas.

En general del proyecto se puede concluir que el flujo completo desde la carga de datos hasta la evaluación final utilizando el conjunto de datos de prueba subraya la importancia de cada paso en el proceso de desarrollo del modelo. Este enfoque estructurado asegura que los modelos de aprendizaje automático sean desarrollados de manera que no solo sean precisos, sino también robustos y capaces de generalizar bien más allá de los datos de entrenamiento. La correcta implementación de estos pasos es esencial para el éxito en la clasificación automatizada de dígitos y otras tareas de clasificación similares, garantizando que los sistemas desarrollados sean confiables y efectivos en aplicaciones prácticas.

Recursos

rarfile - RAR archive reader for Python — RarFile documentation. (s/f). Readthedocs.io.

Recuperado el 29 de abril de 2024, de <https://rarfile.readthedocs.io/>

Python Packaging User Guide. (s/f). Python.org. Recuperado el 29 de abril de 2024, de

<https://packaging.python.org/en/latest/>

Version handling - packaging. (s/f). Pypa.io. Recuperado el 29 de abril de 2024, de

<https://packaging.pypa.io/en/latest/version.html>

sys — Parámetros y funciones específicos del sistema. (s/f). Python documentation.

Recuperado el 29 de abril de 2024, de <https://docs.python.org/es/3/library/sys.html>

os — Miscellaneous operating system interfaces. (s/f). Python Documentation. Recuperado

el 29 de abril de 2024, de <https://docs.python.org/3/library/os.html>

Requests: HTTP for humans™ — requests 2.31.0 documentation. (s/f). Readthedocs.io.

Recuperado el 29 de abril de 2024, de <https://requests.readthedocs.io/en/latest/>

NumPy Documentation. (s/f). Numpy.org. Recuperado el 29 de abril de 2024, de

<https://numpy.org/doc/>

Pandas documentation — pandas 2.2.2 documentation. (s/f). Pydata.org. Recuperado el 29

de abril de 2024, de <https://pandas.pydata.org/docs/>

matplotlib.pyplot — Matplotlib 3.5.3 documentation. (s/f). Matplotlib.org. Recuperado el 29

de abril de 2024, de https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html

Sklearn.Svm.SVC. (s/f). Scikit-Learn. Recuperado el 29 de abril de 2024, de

<https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html>

Sklearn.Metrics.Accuracy_score. (s/f). Scikit-Learn. Recuperado el 29 de abril de 2024, de

https://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

Pinzón, D. & Oviedo, D. & Thomas, M. (2024). *Partial Machine Learning*. Google Colab.

https://colab.research.google.com/drive/1e-VFWylaZe_alhtKaSpAFdw180PqpMfK?usp=sharing