UNIVERSIDAD
SERGIO ARBOLEDA

Segundo Parcial

"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"



Chiara Vivian Valenzuela Losada

Oscar Eduardo Miranda Puentes

Mitchell Phillip Bermin Suarez

Aprendizaje de Máquina

Profesor Ricardo Andrés Fonseca Perdomo

Universidad Sergio Arboleda

Bogotá D.C, Colombia

Abril 30 de 2024

Índice

Desarrollo

El conjunto de datos recoge una gran cantidad de imágenes de números dígitos realizados a mano. Estas imágenes se corresponden con una categoría objetivo que indica qué número está escrito en la imagen. El objetivo será observar el comportamiento de un modelo de máquina de soporte vectorial para realizar la clasificación de un conjunto de datos no etiquetado. A continuación, se explicarán los procesos que se realizaron para cumplir el objetivo anteriormente mencionado.

Cargue y exploración del dataset

Primero se carga el archivo del conjunto de datos para leerlo como un arreglo numpy. Esto mediante la función **load_mnist_data()** en la cual se recibe la ruta del archivo y se separa en los conjuntos de datos para su posterior procesamiento. [1]

```
def load_mnist_data(file_path):
    if not Path(file_path).is_file():
        raise FileNotFoundError("El archivo MNIST no se encontró en la ruta especificada.")
    # Cargar los datos
    data = np.load(file_path)
    return data['training_data'].reshape(60000, 28, 28), data['training_labels'].reshape(60000,), data['test_data'].reshape(10000, 28, 28)
```

Se accede a la estructura del archivo mediante el método **keys()**, para visualizar las divisiones dentro del mismo. Se observa que, en efecto, contiene tres campos: **training_data, test_data y training_labels.**

```
# Accede a los datos dentro del archivo .npz
for clave in datos_npz.keys():
    print(clave)  # Imprime las claves
```

```
training_data
test_data
training_labels
```

Luego, se imprimen los tamaños de cada campo del dataset. El set de entrenamiento posee 60000 muestras y el de prueba cuenta con 10000. Además, estos dos tienen una dimensión de 28 x 28 píxeles de resolución en cada una de las muestras. Por último, en cuanto a los data

sets cargados, se tiene separado la cantidad de etiquetas de entrenamiento siendo el mismo número de muestras, correspondiendo a cada una de estas imágenes.

Luego se realiza un análisis de características de los datos y se obtienen las siguientes conclusiones.

```
# Comprobando el balance de las etiquetas y la normalización
print("Distribución de etiquetas de entrenamiento:", np.unique(training_labels, return_counts=True))
print("Rango de valores de píxeles en los datos de entrenamiento:", training_data.min(), "a", training_data.max())
print("Promedio de los valores de píxeles:", np.mean(training_data))
print("Desviación estándar de los valores de píxeles:", np.std(training_data))
```

- **Distribución de etiquetas de entrenamiento:** La salida (array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9]), array([5903, 6770, 5991, 6085, 5860, 5396, 5870, 6265, 5878, 5982])) se refiere a la distribución de las etiquetas en el conjunto de entrenamiento. El primer array representa las etiquetas únicas, que en el caso de MNIST son los dígitos del 0 al 9. El segundo array muestra el número de muestras que corresponden a cada etiqueta. Por ejemplo, hay 5903 imágenes etiquetadas como 0, 6770 imágenes etiquetadas como 1, y así sucesivamente hasta el 9.
- **Rango de valores de píxeles:** La parte que dice Rango de valores de píxeles: 0.0 a 1.0 indica que todos los valores de píxeles en el conjunto de datos están normalizados para estar dentro del rango de 0 a 1. Este es un paso común en el preprocesamiento de imágenes para asegurar que el modelo de machine learning no se vea afectado por la variación en la escala de los valores de entrada, lo que puede mejorar la estabilidad y rendimiento del entrenamiento.
- **Promedio y Desviación Estándar de los valores de píxeles:**
    - ***Promedio (Media):*** Un promedio de aproximadamente 0.131 sugiere que la mayoría de los píxeles en las imágenes son bastante oscuros, ya que este valor es cercano a 0 (donde 0 es negro puro y 1 es blanco puro en imágenes normalizadas).

- ***Desviación estándar:*** Una desviación estándar de aproximadamente 0.308 indica cuánto varían los valores de los píxeles respecto al promedio. Un valor más alto indicaría una mayor dispersión de los valores de los píxeles, mientras que un valor más bajo indica que los píxeles tienden a estar más cerca del valor medio.

Para la visualización de los datos, se define una figura con diez ejes para poder visualizar las primeras diez imágenes contenidas en el conjunto de entrenamiento. Itera sobre estos ejes para asociar cada imagen con su correspondiente etiqueta y así comprobar la naturaleza del conjunto de datos, esto se realiza dos veces uno para los datos de entrenamiento y la otra para los datos de prueba.

```
# Visualizar las primeras 10 imágenes de entrenamiento
fig, axes = plt.subplots(1, 10, figsize=(10, 1))
for i, ax in enumerate(axes):
    ax.imshow(training_data[i], cmap='gray')
    ax.set_title(f'Label: {training_labels[i]}')
    ax.axis('off')
plt.show()
```

[3]

Las visualizaciones del set de entrenamiento tienen el objetivo de verificar cada etiqueta para comprobar que la relación con cada dígito sea correcta antes de usar los datos para el entrenamiento de la SVM y así evitar errores. Ahora se mostrarán las de prueba, que carecen de etiquetas.

```
# Visualizar las primeras 10 imágenes de test
fig, axes = plt.subplots(1, 10, figsize=(10, 1))
for i, ax in enumerate(axes):
    ax.imshow(test_data[i], cmap='gray')
    ax.axis('off')
plt.show()
```

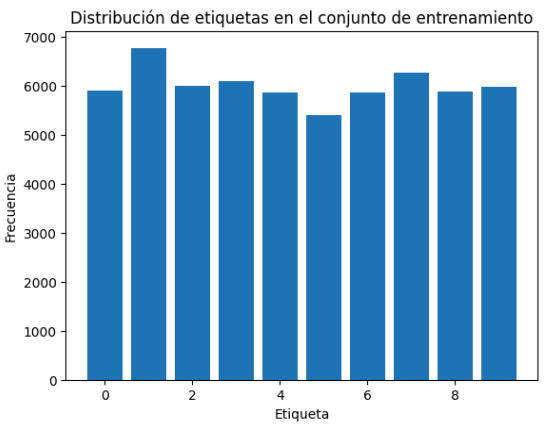
[3]

Se observan las primeras imágenes de los números dígitos del conjunto de prueba, como se extraía anteriormente de la estructura del dataset, estas imágenes no están asociadas a una etiqueta al no existir dentro del conjunto completo original.

Se realiza un análisis preliminar de los datos de entrenamiento, con el fin de identificar la necesidad de implementar técnicas de preprocesamiento en el conjunto de datos. Adicionalmente se crea un gráfico que muestra la frecuencia de los valores únicos de las etiquetas del conjunto de entrenamiento. Se traza la distribución de frecuencia de cada etiqueta mediante un histograma.

```
# Análisis de la necesidad de preprocesamiento
# Comprobando si los datos están normalizados y si las etiquetas están balanceadas
print("Distribución de etiquetas de entrenamiento:")
unique, counts = np.unique(training_labels, return_counts=True)
plt.bar(unique, counts)
plt.title("Distribución de etiquetas en el conjunto de entrenamiento")
plt.xlabel("Etiqueta")
plt.ylabel("Frecuencia")
plt.show()
```

La gráfica muestra la distribución de las etiquetas en el conjunto de entrenamiento, y parece bastante uniforme, lo que es una excelente señal. Esto significa que cada clase de dígito tiene aproximadamente el mismo número de muestras, lo cual es deseable en un conjunto de datos para evitar sesgos hacia clases más frecuentes durante el entrenamiento del modelo.



Se verifica la existencia de valores faltantes dentro del dataset. Con el uso de la función **check missing values,** se recibe el conjunto de datos para comprobar con el método **isnan**, de forma booleana, si dentro de los datos existen valores NaN. El llamado de la función se realiza tanto en los datos como en las etiquetas de estos.

```python
# Verificar si hay valores faltantes
def check_missing_values(data):
    if np.isnan(data).any():
        return True
    else:
        return False

missing_values_training_data = check_missing_values(training_data)
missing_values_training_labels = check_missing_values(training_labels)

print(f'Valores faltantes en los datos de entrenamiento: {missing_values_training_data}')
print(f'Valores faltantes en las etiquetas de entrenamiento: {missing_values_training_labels}')
```

```
Valores faltantes en los datos de entrenamiento: False
Valores faltantes en las etiquetas de entrenamiento: False
```

Ahora se verifica la existencia de datos atípicos mediante la función **check outliers**[2]. Esta función recibe el conjunto de datos y dos valores, un mínimo y un máximo para poder establecer los límites de clasificación para considerar a un dato como outlier. La función retorna un resultado booleano.

```python
# Verificar si hay valores atípicos
def check_outliers(data, min_val=0.0, max_val=1.0):
    if ((data < min_val) | (data > max_val)).any():
        return True
    else:
        return False

outliers_training_data = check_outliers(training_data)

print(f'Valores atípicos en los datos de entrenamiento: {outliers_training_data}')
```

```
Valores atípicos en los datos de entrenamiento: False
```

Se revisaron los conjuntos de datos para determinar si hay valores faltantes o atípicos. La verificación de valores faltantes es una práctica común en el preprocesamiento de datos para garantizar que el modelo de machine learning no se encuentre con entradas inesperadas que podrían causar errores o comportamientos impredecibles. Los valores faltantes pueden necesitar ser imputados o las muestras correspondientes podrían ser excluidas del entrenamiento. La detección de valores atípicos es igualmente importante, ya que los valores

extremos pueden influir desproporcionadamente en el entrenamiento del modelo, especialmente en algoritmos sensibles a la escala de los datos como las SVM o las redes neuronales. En el caso del MNIST, los valores atípicos podrían indicar problemas con el conjunto de datos o errores en el proceso de carga o normalización. En este caso, no se han encontrado valores faltantes o valores atípicos, por lo que se proseguirá y se dará por finalizada la carga y exploración del dataset.

Partición de los datos

Se define una función para mezclar y dividir el dataset. Cumpliendo con la instrucción, el código reserva 10000 imágenes del conjunto establecido por el parámetro **validation size.** Se realiza una mezcla aleatoria utilizando el método **np.random.permutation.** Posteriormente se realiza la correspondiente separación en entrenamiento y validación. Se separan además los índices de los conjuntos para poder atribuir las etiquetas a cada dato en los conjuntos de datos separados. Al final, se retornan los dos conjuntos de datos junto con los conjuntos de etiquetas correspondientes a cada uno de estos. [4]

```python
# Función para mezclar y dividir el conjunto de datos
def shuffle_and_split_data(training_data, training_labels, validation_size=10000):
    # Generar índices aleatorios
    shuffled_indices = np.random.permutation(len(training_data))

    # Dividir los índices para los conjuntos de entrenamiento y validación
    validation_indices = shuffled_indices[:validation_size]
    train_indices = shuffled_indices[validation_size:]

    # Separar los datos y las etiquetas en conjuntos de entrenamiento y validación
    validation_data = training_data[validation_indices]
    validation_labels = training_labels[validation_indices]
    train_data = training_data[train_indices]
    train_labels = training_labels[train_indices]

    return train_data, train_labels, validation_data, validation_labels
```

Se realiza el llamado de la función de división y se comprueba que los tamaños de los conjuntos sean los esperados. Se puede observar que efectivamente el conjunto de validación reservó las 10000 imágenes del conjunto original.

```
# Llamar a la función con los datos cargados previamente
train_data, train_labels, validation_data, validation_labels = shuffle_and_split_data(training_data, training_labels)

# Verificación de las longitudes de los conjuntos resultantes
print("Conjunto de Entrenamiento:", train_data.shape, train_labels.shape)
print("Conjunto de Validación:", validation_data.shape, validation_labels.shape)
```

```
Conjunto de Entrenamiento: (50000, 28, 28) (50000,)
Conjunto de Validación: (10000, 28, 28) (10000,)
```

La forma anterior de manejar los datos garantiza que la mezcla y la división sean consistentes entre los datos y las etiquetas, manteniendo la correspondencia entre las imágenes y sus etiquetas correspondientes. Ya que es crucial mantener la distribución de las clases en ambos conjuntos.

Implementación del Algoritmo de entrenamiento

Lo primero que se hace para la implementación correcta del algoritmo, es mediante el método **reshape** aplanar la dimensión de la matriz de entrenamiento y validación a una. Asimismo, las imágenes se aplanan a un vector de 784 elementos al multiplicar sus píxeles. Se declara una lista con los valores para variar el comportamiento a medida que cambia el tamaño del conjunto de datos. Por último, se definen listas para guardar los valores de precisión que se obtuvo con cada valor de ejemplo.

```
train_data_flat = train_data.reshape(-1, 28*28)
validation_data_flat = validation_data.reshape(-1, 28*28)

# Números de ejemplos de entrenamiento para iterar
training_sizes = [100, 200, 500, 1000, 2000, 5000, 10000]

# Listas para guardar las precisiones
train_accuracies = []
validation_accuracies = []
```

Se crea un bucle **for** que se va a encargar de iterar sobre la lista de valores de ejemplo previamente establecida, y a su vez entrenar el modelo de la máquina de soporte con cada uno de dichos valores. Para cada uno de los valores se instancia el modelo de SVM, este se

establece que su kernel para el límite de decisión es del tipo linear. Se entrena el modelo con los datos divididos para el entrenamiento con sus respectivas etiquetas.

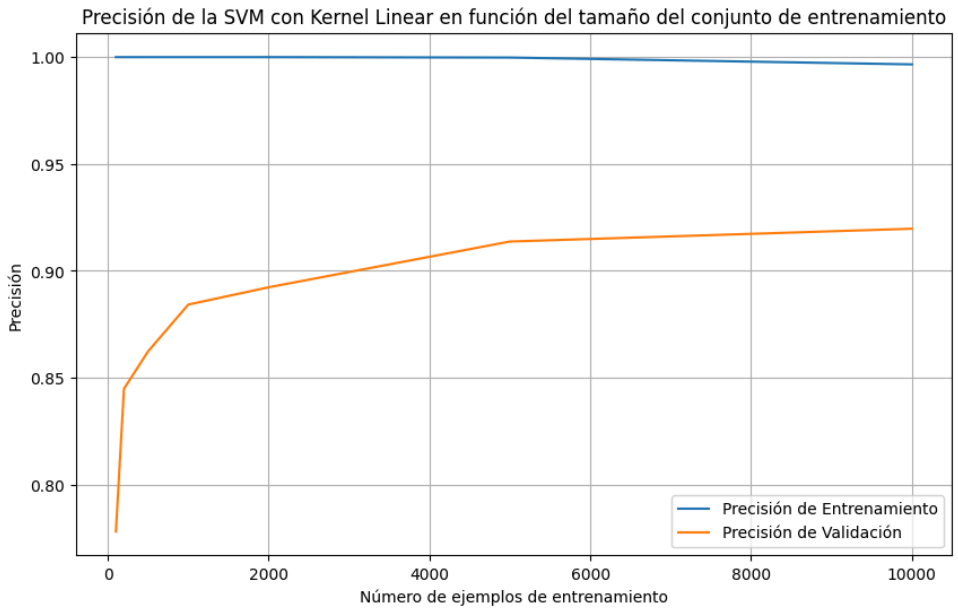
```
# Entrenar con diferentes tamaños del conjunto de entrenamiento
for size in training_sizes:
    # Entrenar el modelo SVM
    svm_l = SVC(kernel='linear')
    svm_l.fit(train_data_flat[:size], train_labels[:size])

    # Evaluar y almacenar la precisión en el conjunto de entrenamiento
    train_accuracy = accuracy_score(train_labels[:size],
                                    svm_l.predict(train_data_flat[:size]))
    train_accuracies.append(train_accuracy)

    # Evaluar y almacenar la precisión en el conjunto de validación
    validation_accuracy = accuracy_score(validation_labels,
                                         svm_l.predict(validation_data_flat))
    validation_accuracies.append(validation_accuracy)
```

Posteriormente se realizan las predicciones tanto con el conjunto de entrenamiento y el de validación. Además, se almacenan los resultados de precisión dentro de las listas declaradas en el paso anterior. Con esto se puede observar cómo se comporta el modelo con datos conocidos y desconocidos, verificando qué tanto varía el **accuracy**. Después de realizar el bucle se imprimen los valores de cada iteración y adicionalmente se muestran los resultados de precisión tanto en entrenamiento como en validación para cada una de estas.

```
Tamaño del conjunto de entrenamiento: 100, Precisión de entrenamiento: 1.0, Precisión de validación: 0.778
Tamaño del conjunto de entrenamiento: 200, Precisión de entrenamiento: 1.0, Precisión de validación: 0.8448
Tamaño del conjunto de entrenamiento: 500, Precisión de entrenamiento: 1.0, Precisión de validación: 0.8623
Tamaño del conjunto de entrenamiento: 1000, Precisión de entrenamiento: 1.0, Precisión de validación: 0.8842
Tamaño del conjunto de entrenamiento: 2000, Precisión de entrenamiento: 1.0, Precisión de validación: 0.8923
Tamaño del conjunto de entrenamiento: 5000, Precisión de entrenamiento: 0.9998, Precisión de validación: 0.9137
Tamaño del conjunto de entrenamiento: 10000, Precisión de entrenamiento: 0.9966, Precisión de validación: 0.9197
```

Finalmente se procede a realizar una gráfica para comparar visualmente la precisión del modelo en las iteraciones. Se establece el eje horizontal como el valor de la iteración y el vertical como el resultado en precisión.

```
1  plt.figure(figsize=(10, 6))
2  plt.plot(training_sizes, train_accuracies, label='Precisión de Entrenamiento')
3  plt.plot(training_sizes, validation_accuracies, label='Precisión de Validación')
4  plt.xlabel('Número de ejemplos de entrenamiento')
5  plt.ylabel('Precisión')
6  plt.title('Precisión de la SVM con Kernel Linear en función del tamaño del conjunto de entrenamiento')
7  plt.legend()
8  plt.grid(True)
9  plt.show()
```



Los resultados indican que el modelo SVM lineal tiene un buen rendimiento en cuanto a precisión de entrenamiento, alcanzando la perfección o cerca de ella en los conjuntos más pequeños, y un rendimiento algo menor pero alto en los conjuntos más grandes. En cuanto a la precisión de validación, los resultados también muestran una mejora gradual conforme aumenta el tamaño del conjunto de entrenamiento.

Análisis de los Resultados

- ***Precisión de entrenamiento:*** El hecho de que la precisión de entrenamiento sea tan alta (incluso perfecta en los tamaños menores) puede indicar un sobreajuste en esos tamaños de entrenamiento más pequeños, especialmente porque el modelo está aprendiendo a clasificar perfectamente el conjunto de entrenamiento, pero no generaliza igual de bien en el conjunto de validación.
- ***Precisión de validación:*** La precisión de validación mejora consistentemente a medida que se incrementa el número de ejemplos de entrenamiento. Esto es esperado, ya que un mayor número de ejemplos proporciona una representación más rica y diversa del espacio de características, permitiendo al modelo generalizar mejor. La precisión de validación llega hasta un 91.97% para 10,000 ejemplos de entrenamiento,

lo cual está dentro del rango esperado de 70% a 90% mencionado en las instrucciones del examen.

Conclusiones Generales

Las conclusiones generales de este estudio sobre el uso de Máquinas de Soporte Vectorial (SVM) para clasificar dígitos manuscritos del dataset MNIST son las siguientes:

Calidad y Preprocesamiento de Datos

- Los datos están bien balanceados entre las diversas clases (dígitos del 0 al 9), lo cual es esencial para evitar sesgos en el entrenamiento del modelo hacia las clases más frecuentes.
- Los datos están normalizados en un rango de 0 a 1, lo cual es crucial para el rendimiento y estabilidad del modelo SVM, especialmente dado que utiliza cálculos de distancia en su funcionamiento.
- La verificación de la distribución de los valores de píxeles y la ausencia de valores atípicos y faltantes sugieren que el dataset MNIST está bien curado y es adecuado para entrenar modelos de aprendizaje automático sin necesidad de preprocesamiento adicional extenso.

Visualización y Análisis Preliminar

- La visualización de los datos proporcionó una verificación útil de la correspondencia entre las imágenes y sus etiquetas, lo cual es vital para garantizar la integridad del entrenamiento.

Sugerencias para Mejorar

- Más datos o técnicas de aumento: Considerar el uso de más datos de entrenamiento o aplicar técnicas de aumento de datos podría ayudar a mejorar la generalización del modelo, especialmente si los recursos lo permiten.

- Exploración de características adicionales: Aunque el ejercicio especifica el uso de píxeles en bruto como características, en la práctica, la extracción de características más sofisticadas podría mejorar la capacidad del modelo para distinguir entre clases.

En resumen, los resultados son prometedores y muestran un buen ajuste inicial del modelo a los datos. La mejora continua de la precisión de validación con el aumento del tamaño del conjunto de entrenamiento es un indicativo positivo de que el modelo se está beneficiando de más datos para aprender las características del problema.

Referencias

1. GfG, A. (2020, julio 1). Ways to import CSV files in Google Colab. GeeksforGeeks. https://www.geeksforgeeks.org/ways-to-import-csv-files-in-google-colab/
2. Khadija. (2021, septiembre 8). What if there are a lot of outliers in your dataset. Stack Overflow. https://stackoverflow.com/questions/69104651/what-if-there-is-a-lot-of-outliers-in-your-dataset
3. LeCun, Yann and Cortes, Corinna and Burges, CJ. (2010). MNIST handwritten digit database. Tensor Flow. http://yann.lecun.com/exdb/mnist
4. Ricardo Fonseca. (2024). Función de shuffle y split. Clase de Aprendizaje de Máquina