Investigación Reconocimiento de Dígitos en Tiempo Real

Chiara V. Valenzuela L., Mitchell P. Bermin S. y Oscar E. Miranda

Facultad de Ciencias Exactas e Ingeniería, Universidad Sergio Arboleda

G02: Aprendizaje de Máquina

Prof. Ricardo Andrés Fonseca Perdomo

12 de junio de 2024

Índice

Índice de Figuras

Introducción

El reconocimiento de dígitos manuscritos es una tarea fundamental en el campo de la visión por computadora y el aprendizaje automático. Este proyecto se centra en el desarrollo de un programa capaz de identificar y clasificar dígitos escritos a mano utilizando una cámara web. Mediante un clasificador SVM (Máquinas de Soporte Vectorial), el sistema segmenta y reconoce dígitos en imágenes capturadas, lo que tiene aplicaciones potenciales en áreas como la digitalización de documentos, la interacción hombre-máquina y la automatización de procesos que involucran entrada de datos numéricos.

Para mejorar la precisión y eficiencia del reconocimiento de dígitos, se ha integrado la técnica de Análisis de Componentes Principales (PCA) en el preprocesamiento de los datos. PCA reduce la dimensionalidad de las imágenes de dígitos, conservando las características más relevantes y eliminando el ruido, lo que facilita el trabajo del clasificador SVM. El uso combinado de PCA y SVM no solo optimiza el rendimiento del sistema en términos de velocidad, sino que también mejora su capacidad de generalización frente a variaciones en los estilos de escritura y las condiciones de iluminación.

Este proyecto implementa una solución, desde la captura de imágenes en tiempo real hasta el preprocesamiento y la clasificación, demostrando su efectividad a través de pruebas en un entorno controlado. El programa desarrollado tiene el potencial de ser aplicado en una amplia variedad de contextos donde la entrada rápida y precisa de datos numéricos es esencial, proporcionando una herramienta poderosa para la digitalización y automatización en diferentes industrias.

Justificación

El reconocimiento automático de dígitos manuscritos es un avance en el campo de la visión por computadora y el aprendizaje automático. La implementación de este tipo de tecnología tiene el potencial de revolucionar la manera en que interactuamos con los datos numéricos y cómo estos son procesados en diversas aplicaciones prácticas. A continuación, se presentan las razones clave que justifican el desarrollo y la investigación en este proyecto.

Eficiencia y Precisión

La precisión y la eficiencia son elementos fundamentales en cualquier sistema de procesamiento de datos. En muchos sectores, la entrada manual de datos numéricos es una tarea común que consume tiempo y está sujeta a errores humanos. El reconocimiento automático de dígitos puede eliminar gran parte de estos errores, garantizando que la información sea capturada y procesada con mayor exactitud. Este aumento en la precisión no solo mejora la calidad de los datos, sino que también facilita decisiones más informadas basadas en estos datos.

Además, la eficiencia operativa se ve significativamente mejorada con la automatización de procesos que antes requerían intervención manual. La digitalización de formularios escritos a mano, por ejemplo, puede ser acelerada enormemente, permitiendo que grandes volúmenes de datos sean procesados en una fracción del tiempo necesario mediante métodos tradicionales. Esto no solo reduce los costos operativos, sino que también libera recursos humanos para tareas más estratégicas y de mayor valor añadido.

Innovación y Avance Tecnológico

El desarrollo de un sistema de reconocimiento de dígitos utilizando técnicas avanzadas como las Máquinas de Soporte Vectorial (SVM) y el Análisis de Componentes Principales (PCA) representa un paso importante en la evolución de la inteligencia artificial y la visión por computadora.

El uso de PCA en el preprocesamiento de datos, por ejemplo, permite reducir la dimensionalidad de las imágenes, conservando las características más relevantes y eliminando el ruido. Esto no solo mejora la precisión del clasificador SVM, sino que también optimiza el tiempo de procesamiento, haciéndolo más adecuado para aplicaciones en tiempo real. La combinación de estas técnicas proporciona una comprensión de cómo los métodos de aprendizaje automático pueden ser adaptados y mejorados para resolver problemas específicos.

En resumen, la justificación de este proyecto se basa en su capacidad para mejorar la eficiencia y precisión en la entrada y procesamiento de datos, y fomentar la innovación y el avance tecnológico. Estas razones subrayan la importancia y la relevancia del desarrollo de sistemas de reconocimiento automático de dígitos manuscritos.

Objetivos

Objetivo General

Desarrollar un sistema de reconocimiento de dígitos manuscritos, utilizando técnicas de visión por computadora y aprendizaje automático, específicamente Máquinas de Soporte Vectorial (SVM) y Análisis de Componentes Principales (PCA), con el fin de mejorar la precisión y eficiencia en la entrada y procesamiento de datos numéricos.

Objetivos Específicos

Clasificador SVM

Entrenar un clasificador SVM utilizando un conjunto de datos de dígitos manuscritos para asegurar una alta precisión en la predicción de los dígitos. Este objetivo implica la recopilación y preparación de un conjunto de datos adecuado, la selección de características relevantes a través de PCA y sin PCA, y la optimización del modelo SVM para obtener un rendimiento óptimo. El enfoque en la precisión del clasificador es crucial para garantizar que el sistema pueda reconocer correctamente los dígitos en diversas condiciones.

Integración en Tiempo Real

Integrar el clasificador entrenado con un sistema de procesamiento de imágenes en tiempo real para reconocer y mostrar los dígitos en una interfaz visual, permitiendo una interacción fluida y en tiempo real. Este objetivo abarca la implementación de un flujo de trabajo continuo que capture imágenes, las procese y las clasifique de manera instantánea, mostrando los resultados al usuario final. La capacidad de operar en tiempo real es esencial para aplicaciones prácticas donde la inmediatez y la precisión de la respuesta son críticas.

Optimización del Rendimiento

Optimizar el rendimiento del sistema en términos de velocidad de procesamiento y uso de recursos. Este objetivo se enfoca en mejorar la eficiencia del clasificador SVM y el sistema de procesamiento en tiempo real, asegurando que el sistema pueda funcionar en dispositivos con diferentes capacidades de hardware sin comprometer su rendimiento.

Aplicaciones en la Cotidianidad del Sistema

El sistema de reconocimiento de dígitos en tiempo real tiene un amplio espectro de aplicaciones prácticas en la vida diaria. A continuación, se describen algunas de las más relevantes:

Digitalización de Documentos

Uno de los usos más evidentes del reconocimiento de dígitos manuscritos es la digitalización de documentos. Instituciones como bancos, oficinas gubernamentales y empresas pueden utilizar este sistema para convertir formularios y registros escritos a mano en datos digitales, mejorando la eficiencia y reduciendo los errores de transcripción manual.

Interacción Hombre-Máquina

El sistema puede facilitar la interacción hombre-máquina, permitiendo a los usuarios ingresar números mediante gestos o escritura a mano frente a una cámara. Esto puede ser útil en dispositivos de asistencia para personas con discapacidades, donde la entrada de datos por teclado no es viable.

Automatización de Procesos

En entornos industriales y comerciales, el reconocimiento de dígitos puede automatizar procesos que requieren la entrada de datos numéricos. Por ejemplo, en almacenes y centros de logística, los trabajadores pueden usar el sistema para ingresar códigos de productos o cantidades simplemente mostrándolos a la cámara, agilizando así el proceso de inventario.

Educación

En el ámbito educativo, el sistema puede ser una herramienta útil para la enseñanza y el aprendizaje de matemáticas. Los estudiantes pueden escribir números y ecuaciones en una pizarra o cuaderno, y el sistema los reconocerá y procesará en tiempo real, proporcionando retroalimentación instantánea.

Sector Salud

En el sector salud, el reconocimiento de dígitos manuscritos puede facilitar la digitalización de recetas médicas y registros de pacientes. Los médicos pueden escribir las recetas a mano, y el sistema las convertirá automáticamente en un formato digital, reduciendo el riesgo de errores de interpretación y mejorando la eficiencia en la administración de medicamentos.

Transporte y Tráfico

El sistema también puede ser aplicado en la gestión del transporte y tráfico. Por ejemplo, puede ser utilizado para leer números de placas de vehículos en tiempo real, ayudando en la implementación de sistemas de peaje electrónico o en la vigilancia y control del tráfico.

Comercio Electrónico

En el comercio electrónico, el reconocimiento de dígitos puede mejorar la experiencia del usuario al permitirles ingresar números de tarjetas de crédito o códigos de cupones simplemente mostrándolos a la cámara, acelerando el proceso de compra y reduciendo los errores de entrada.

Juegos y Entretenimiento

El sistema puede ser integrado en juegos y aplicaciones de entretenimiento que requieran la entrada de números o comandos mediante gestos. Esto añade una dimensión interactiva y divertida a la experiencia del usuario.

Estas aplicaciones demuestran la versatilidad y utilidad del sistema de reconocimiento de dígitos en tiempo real, destacando su potencial para transformar diversos aspectos de la vida diaria mediante la mejora de la eficiencia, precisión y facilidad de uso en múltiples contextos.

Desarrollo

Preprocesamiento

El preprocesamiento de datos es una etapa crucial en el desarrollo de modelos de machine learning, especialmente para tareas de clasificación de dígitos manuscritos. En este proyecto, el preprocesamiento incluye varias técnicas y pasos para garantizar que los datos sean adecuados para el entrenamiento del modelo SVM con reducción de dimensionalidad mediante PCA. A continuación, se describen los principales pasos de preprocesamiento realizados:

Obtención de Datos

Se utilizaron los datos del conjunto MNIST, que contiene imágenes de dígitos manuscritos del 0 al 9. Los datos fueron obtenidos de OpenML.

División de los Datos

Los datos se dividieron en conjuntos de entrenamiento y prueba utilizando una proporción de 90% para el entrenamiento y 10% para la prueba.

Normalización

La normalización es un paso esencial para garantizar que todas las características tengan una escala comparable, lo que mejora el rendimiento del modelo SVM. Los datos fueron normalizados usando `StandardScaler` de `sklearn`.

Reducción de Dimensionalidad con PCA

El Análisis de Componentes Principales (PCA) se aplicó para reducir la dimensionalidad de los datos, reteniendo el 95% de la varianza acumulada. Esto no solo ayuda a manejar mejor los datos desde el punto de vista computacional, sino que también mejora la eficiencia del modelo al enfocarse en las características más importantes.

Aumento de Datos (Data Augmentation)

Para mejorar la robustez del modelo, se aplicaron técnicas de aumento de datos, como rotaciones, zooms y desplazamientos, creando variaciones de las imágenes de entrenamiento. Esto ayuda a que el modelo generalice mejor a datos nuevos.

Selección del Algoritmo para Entrenamiento

La selección del algoritmo de aprendizaje adecuado es importante para el éxito de cualquier sistema de reconocimiento de patrones. En este proyecto, el objetivo principal es desarrollar un sistema eficiente y preciso para el reconocimiento de dígitos manuscritos en tiempo real. Después de considerar varias opciones, se eligieron las Máquinas de Soporte Vectorial (SVM). A continuación, se explican las razones detrás de esta elección.

Máquinas de Soporte Vectorial (SVM)

Las Máquinas de Soporte Vectorial son un poderoso método de clasificación supervisada que se utiliza ampliamente debido a su capacidad para manejar problemas de alta dimensionalidad y proporcionar soluciones robustas y eficientes. Algunas de las ventajas que motivaron la elección de SVM incluyen:

1. **Precisión Alta:** SVM es conocido por su alta precisión en la clasificación de datos, especialmente en problemas donde las clases son linealmente separables o casi separables.
2. **Capacidad de Generalización:** SVM tiene una buena capacidad de generalización, lo que permite al modelo manejar bien datos nuevos y no vistos previamente.

3. **Flexibilidad:** Al utilizar diferentes funciones kernel (como el kernel RBF utilizado en este proyecto), SVM puede adaptarse a una amplia variedad de problemas, tanto lineales como no lineales.

Uso de Clasificador de Vectores de Soporte (SVC)

En este proyecto, se optó por utilizar un Clasificador de Vectores de Soporte (SVC) en lugar de una Máquina de Vectores de Soporte (SVM) estándar. SVC es una implementación específica de SVM que se encuentra en la biblioteca `scikit-learn` de Python. Las razones para esta elección son las siguientes:

1. **Implementación Optimizada:** SVC, proporcionada por `scikit-learn`, está optimizada para su uso en Python, facilitando su integración y uso con otras herramientas y bibliotecas comunes en el ecosistema de Python, como `NumPy` y `Pandas`.
2. **Facilidad de Uso:** SVC en `scikit-learn` viene con una API fácil de usar que simplifica la implementación y ajuste del modelo, permitiendo experimentar con diferentes hiperparámetros y kernels de manera eficiente.
3. **Soporte para Multiclase:** SVC soporta naturalmente problemas de clasificación multiclase utilizando estrategias como "uno contra uno" y "uno contra todos", lo cual es esencial para la tarea de reconocimiento de dígitos (que tiene 10 clases diferentes).

Comparación con Otros Algoritmos

Durante el desarrollo del proyecto, se consideraron otros algoritmos de aprendizaje, como redes neuronales artificiales (ANN) y k-Nearest Neighbors (k-NN). Sin embargo, SVM fue seleccionado por las siguientes razones:

1. **Redes Neuronales:** Aunque las redes neuronales son muy potentes, su entrenamiento requiere más datos y tiempo computacional significativo, lo cual no era ideal para este proyecto enfocado en la eficiencia y rapidez.
2. **k-Nearest Neighbors (k-NN):** k-NN es fácil de implementar pero se vuelve ineficiente con grandes conjuntos de datos debido a la necesidad de calcular distancias para cada predicción.

Conclusión

La selección del Clasificador de Vectores de Soporte (SVC) fue motivada por su equilibrio entre precisión, eficiencia y capacidad de generalización. Este clasificador permite al sistema reconocer dígitos manuscritos con alta precisión y velocidad, lo cual es esencial para su implementación en tiempo real. La decisión de utilizar este algoritmo se basó en una cuidadosa consideración de sus ventajas y desventajas, así como en la naturaleza del problema y los requisitos del proyecto.

Cálculos, Límites de Decisión y Mejor Área de Aprendizaje

Para desarrollar un sistema de reconocimiento de dígitos manuscritos eficiente y preciso, es esencial entender y definir los cálculos necesarios, los límites de decisión y la mejor área de aprendizaje para el modelo de clasificación. Esta sección proporciona una descripción detallada de estos aspectos.

Cálculos

Los cálculos realizados en este proyecto son fundamentales para el preprocesamiento de datos y el entrenamiento del modelo. A continuación, se detallan los principales cálculos involucrados:

Normalización de Datos

- **Fórmula:** $x' = \frac{x - \varphi}{\sigma}$
- **Descripción:** Los datos de entrada se normalizan restando la media (φ) y dividiendo por la desviación estándar (σ). Este proceso garantiza que los datos tengan una media de 0 y una desviación estándar de 1, lo cual es crucial para el rendimiento del modelo SVM.

Cálculo de la Matriz de Covarianza

- **Fórmula**: $\Sigma = \frac{1}{n-1} \sum_{i=1}^{n} (x_i - \bar{x})(x_i - \bar{x})^T$
- **Descripción:** La matriz de covarianza se calcula para entender las relaciones entre las distintas características de los datos. Este cálculo es importante para la reducción de dimensionalidad y la selección de características relevantes.

Análisis de Componentes Principales (PCA)

Cálculo de los Autovalores y Autovectores

- **Descripción**: Los autovalores (λ) y autovectores (ν) se calculan a partir de la matriz de covarianza. Los autovectores corresponden a las direcciones de mayor varianza en los datos.

Selección de Componentes Principales

- **Descripción**: Se seleccionan los autovectores correspondientes a los autovalores que explican el 95% de la varianza acumulada. Esto reduce la

dimensionalidad de los datos mientras se conserva la mayor parte de la información relevante.

Transformación de Datos

- **Fórmula**: $X_{reducido} = X \cdot V$
- **Descripción**: Los datos originales se proyectan en el espacio de los componentes principales seleccionados para obtener un conjunto de datos de menor dimensión.

Optimización

Aumento de datos (Data Augmentation)

- **Descripción:** Se aplican técnicas de aumento de datos como rotaciones, zooms y desplazamientos para crear variaciones de los datos de entrenamiento, aumentando así la robustez del modelo.

Límites de Decisión

Definir los límites de decisión es crucial para la clasificación precisa de los dígitos. En el contexto de SVM con kernel lineal, los límites de decisión se determinan mediante los hiperplanos que separan las diferentes clases en el espacio de características.

Hiperplano Óptimo

- **Fórmula**: $\omega \cdot x + b = 0$
- **Descripción:** El hiperplano óptimo se define como el que maximiza la distancia (margen) entre las dos clases más cercanas. Los vectores de soporte son los puntos de datos que están más cerca del hiperplano y son críticos para definir la posición del hiperplano.

Márgenes

- **Fórmula:** $d = \frac{2}{||w||}$
- **Descripción:** El margen es la distancia entre el hiperplano y los vectores de soporte. Un margen mayor generalmente implica una mejor capacidad de generalización del modelo.

Mejor Área de Aprendizaje

Identificar la mejor área de aprendizaje implica determinar los parámetros óptimos y las características del conjunto de datos que permiten al modelo SVM rendir de manera óptima.

Evaluación del Modelo

- **Métricas:** Precisión, recall, F1-score.
- **Descripción:** Se evalúa el rendimiento del modelo en un conjunto de datos de validación para asegurarse de que los hiperparámetros seleccionados proporcionen el mejor rendimiento posible.

Análisis de la Distribución de Datos

Se analiza la distribución de los datos de entrenamiento para asegurarse de que estén balanceados y representen adecuadamente todas las clases. Un conjunto de datos balanceado es crucial para evitar sesgos en el modelo.

Métricas y Gráficas

Para evaluar el rendimiento del modelo de reconocimiento de dígitos manuscritos, se utilizan varias métricas y técnicas de visualización. Estas métricas y gráficas proporcionan una visión detallada de la precisión del modelo y su capacidad de

generalización. A continuación, se describen las principales métricas utilizadas y las gráficas generadas para analizar el rendimiento del modelo.

Métricas de Evaluación

Precisión (Accuracy)

- **Descripción:** La precisión es la proporción de predicciones correctas entre el total de predicciones realizadas.
- **Fórmula:** $Accuracy = \frac{Predicciones\ Correctas}{Total\ de\ Predicciones}$

Precisión (Precision)

- **Descripción:** La precisión mide la cantidad de verdaderos positivos entre todas las instancias que fueron clasificadas como positivas.
- **Fórmula:** $Precisión = \frac{Verdaderos\ Positivos}{Verdaderos\ Positivos + Falsos\ Positivos}$

Exhaustividad (Recall)

- **Descripción:** La exhaustividad mide la cantidad de verdaderos positivos entre todas las instancias que deberían haber sido clasificadas como positivas.
- **Fórmula:** $Exhaustividad = \frac{Verdaderos\ Positivos}{Verdaderos\ Positivos + Falsos\ Negativos}$

F1-Score

- **Descripción:** El F1-Score es la media armónica entre la precisión y la exhaustividad, proporcionando una única métrica para evaluar el equilibrio entre ambos.
- **Fórmula:** $F1 - Score = 2 \cdot \frac{Precisión \cdot Exhaustividad}{Precisión + Exhaustividad}$

Matriz de Confusión

- **Descripción:** La matriz de confusión proporciona una representación visual de los resultados de las predicciones del modelo, mostrando el número de predicciones correctas e incorrectas distribuidas en las distintas clases.
- **Fórmula:** La matriz de confusión es una tabla M de tamaño $n \times n$, donde n es el número de clases. Una matriz de confusión típica para un problema de clasificación con dos clases podría verse así:

  Tabla 1

  Ejemplo de matriz de confusión

	Clase 0	Clase 1
Clase 0	M_{00}	M_{01}
Clase 1	M_{10}	M_{11}

  Donde:

  - M_{00} Es el número de instancias de la clase 0 clasificadas correctamente.
  - M_{01} Es el número de instancias de la clase 0 clasificadas incorrectamente como clase 1.

Gráficas

Curvas de Precisión-Exhaustividad

Las curvas de precisión-exhaustividad muestran la relación entre precisión y exhaustividad.

Figura 1

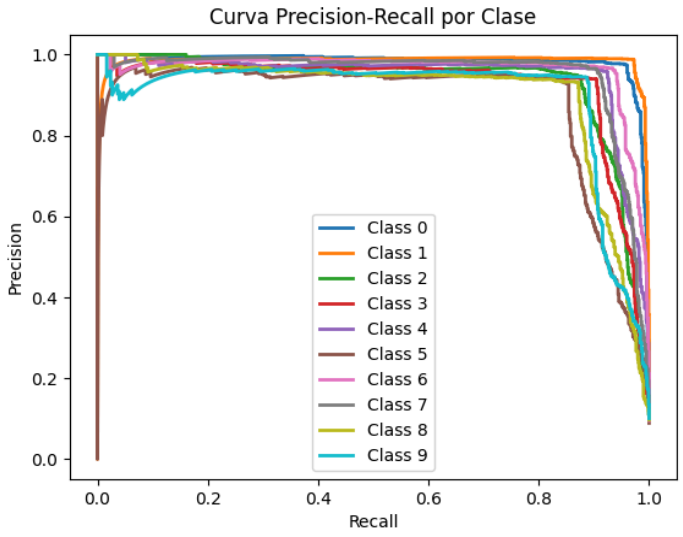
Implementación en código de curvas de precisión-exhaustividad

```
# Calcular y trazar la curva Precision-Recall para cada clase
for i in range(n_classes):
    precision, recall, _ = metrics.precision_recall_curve(y_test_binarized[:, i], decision_function[:, i])
    plt.plot(recall, precision, lw=2, label=f'Class {i}')
```

Resultados de Curva sin PCA

Figura 2

Resultados de curva sin PCA



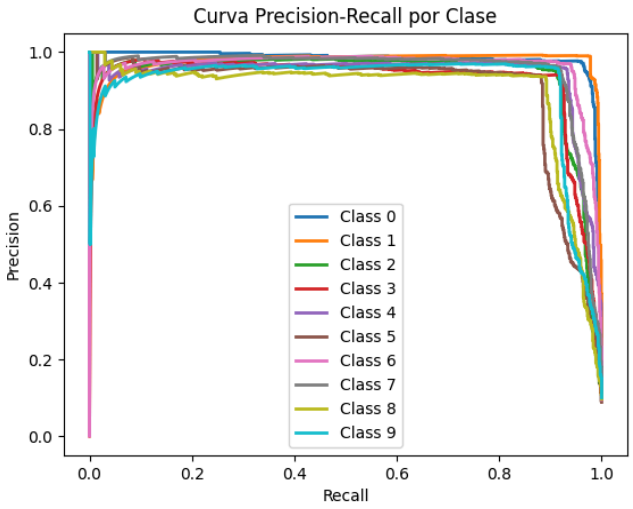
La figura 2, de Curvas de Precisión-Recall muestra la relación entre la precisión y el recall para cada una de las clases en el modelo sin PCA. A continuación se presentan las observaciones basadas en la gráfica:

El modelo muestra un alto rendimiento general, evidenciado por las curvas de precisión-recall de la mayoría de las clases que se mantienen cerca de la esquina superior derecha, indicando que el modelo identifica correctamente la mayoría de las instancias positivas sin incluir muchas instancias negativas incorrectamente. Sin embargo, hay una variabilidad en las curvas de precisión-recall entre diferentes clases, lo que puede deberse a la dificultad inherente de distinguir ciertas clases. Además, para todas las clases, se observa una caída pronunciada en la precisión cuando el recall se acerca a 1, lo cual es típico en estas curvas, ya que al intentar capturar todas las instancias positivas, el modelo puede incluir más instancias negativas, reduciendo así la precisión.

Resultados de Curva con PCA

Figura 3

Resultados de curva con PCA



La figura 3 muestra las curvas de precisión-recall para cada una de las clases del conjunto de datos MNIST usando un modelo SVM con PCA. Aquí están las conclusiones clave:

El modelo muestra un alto rendimiento global, con la mayoría de las clases presentando curvas cercanas a la esquina superior derecha, lo que indica un desempeño elevado. Sin embargo, todas las clases exhiben una caída en la precisión cuando el recall se aproxima a 1, lo cual es común en estas curvas. La eficiencia del PCA es notable, ya que ha mantenido un alto rendimiento general, indicando que la mayor parte de la información relevante se ha conservado en los componentes principales seleccionados.

Visualización de Hiperplanos de SVM

Para visualizar los hiperplanos de la SVM, se sigue el siguiente proceso: Entrenamos un modelo SVM con un conjunto de datos bidimensional y luego, usamos `matplotlib` para visualizar los datos, los vectores de soporte, el hiperplano de decisión y los márgenes.

Figura 4

Implementación en código de visualización de hiperplanos

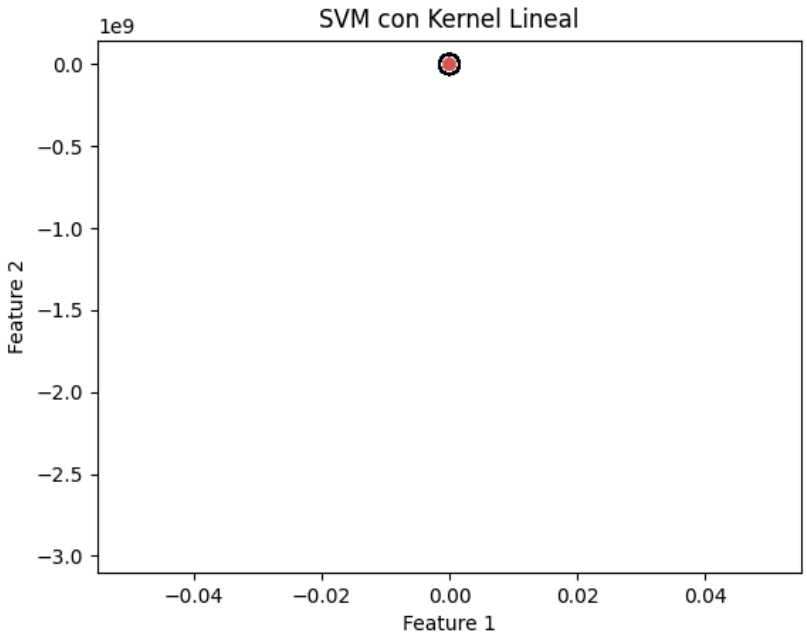
```
# Agregar un pequeño valor epsilon para evitar división por cero
epsilon = 1e-10
slope = -w[0] / (w[1] + epsilon)
intercept = -classifier.intercept_[0] / (w[1] + epsilon)

# Crear una línea para el hiperplano de decisión
xx = np.linspace(min(train_data[:, 0]), max(train_data[:, 0]))
yy = slope * xx + intercept

# Crear márgenes (paralelos al hiperplano de decisión)
margin = 1 / np.sqrt(np.sum(classifier.coef_ ** 2))
yy_down = yy - np.sqrt(1 + slope ** 2) * margin
yy_up = yy + np.sqrt(1 + slope ** 2) * margin

# Plotear los datos y los hiperplanos
plt.scatter(train_data[:, 0], train_data[:, 1], c=train_labels, cmap='coolwarm', s=30)
plt.plot(xx, yy, 'k-')
plt.plot(xx, yy_down, 'k--')
plt.plot(xx, yy_up, 'k--')
```

Visualización de Hiperplano SVM sin PCA

El intento de mostrar el hiperplano de decisión del clasificador SVM sin PCA falló debido a una división por cero en los coeficientes del hiperplano. Esto hace imposible calcular la pendiente del hiperplano $slope = \frac{-w[0]}{w[1]}$. Para evitar divisiones por cero, se agrega un pequeño valor (ϵ) a los coeficientes antes de calcular la pendiente.

Figura 5

Visualización de hiperplano de SVM sin PCA



La figura 5 muestra una visualización del hiperplano de decisión del clasificador SVM con kernel lineal sin PCA. Sin embargo, en la visualización no se puede observar claramente el hiperplano de decisión ni los márgenes, lo que indica que la representación gráfica no es adecuada para los datos.

Posibles Causas:

1. **Selección de Características para Visualización:** Si se seleccionaron características arbitrarias para la visualización, es posible que estas no sean representativas del conjunto de datos, llevando a una mala visualización.
2. **Problema en la Conversión de Etiquetas:** La conversión de las etiquetas de categorías a enteros podría no haberse realizado correctamente, causando problemas en la interpretación de los datos por parte de `matplotlib`.

Recomendaciones para Solucionar los Problemas:

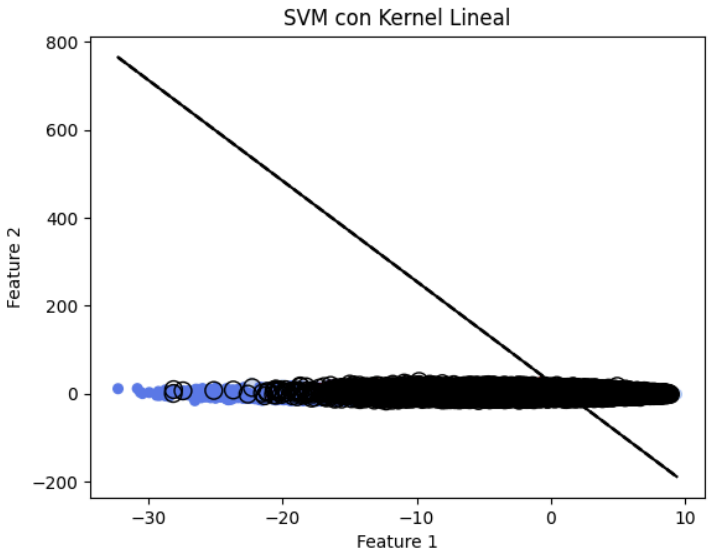
1. **Revisar Selección de Características:** Asegurarse de que las características seleccionadas para la visualización son representativas y están correctamente preprocesadas.
2. **Comprobar Conversión de Etiquetas:** Verificar que las etiquetas han sido convertidas correctamente de categorías a enteros para una interpretación adecuada.

El análisis y corrección de la visualización del hiperplano de decisión del modelo SVM revelan la importancia de una correcta selección de características. El uso de PCA para reducir dimensionalidad y una verificación adecuada de los coeficientes del hiperplano pueden ayudar a generar visualizaciones claras y útiles para interpretar el rendimiento del modelo.

Visualización de Hiperplano SVM

Figura 6

Resultados de visualización de hiperplano con PCA



La figura 6 muestra la visualización del hiperplano de decisión del modelo SVM con kernel lineal, después de aplicar PCA para reducir la dimensionalidad de los datos.

1. **Distribución de los Datos:** Los datos proyectados en los dos componentes principales seleccionados muestran una gran dispersión, especialmente en el eje de la segunda característica (Feature 2).
2. **Hiperplano de Decisión:** El hiperplano de decisión (línea negra) divide el espacio de las características en dos regiones. Sin embargo, la inclinación del hiperplano y la distribución de los datos sugieren que la separación no es óptima. Los puntos de datos están densamente agrupados alrededor de una línea horizontal en la gráfica, lo que puede indicar que las

características seleccionadas mediante PCA no separan adecuadamente las clases en este espacio reducido.

3. **Adecuación del Modelo SVM:** El modelo SVM con kernel lineal puede no ser el más adecuado para los datos proyectados en dos dimensiones mediante PCA. Podría ser beneficioso considerar otros modelos o kernels no lineales que puedan manejar mejor la distribución de los datos.
4. **Necesidad de Técnicas Adicionales:** Se podrían explorar técnicas adicionales como el ajuste fino de los parámetros de PCA o el uso de técnicas de preprocesamiento de datos para mejorar la separación de clases en el espacio reducido.

Matriz de Confusión

Para visualizar los hiperplanos de la SVM, se sigue el siguiente proceso: Entrenamos un modelo SVM con el conjunto de datos original, usamos 'confusion_matrix' y 'ConfusionMatrixDisplay' de sklearn.metrics y por último, usamos `matplotlib` para visualizar los datos, los vectores de soporte, el hiperplano de decisión y los márgenes.

Figura 7

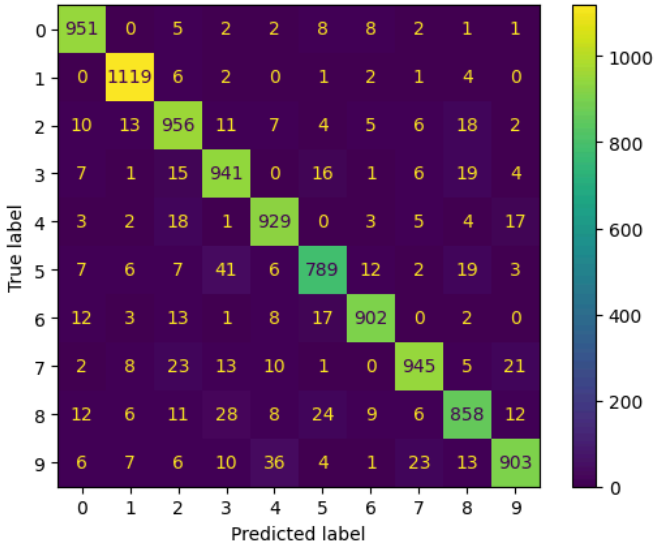
Implementación en código de visualización de hiperplanos

```
# Calcular la matriz de confusión
cm = metrics.confusion_matrix(test_labels, predicted)
disp = metrics.ConfusionMatrixDisplay(confusion_matrix=cm, display_labels=classifier.classes_)
disp.plot()
plt.savefig('../graphs/confusion_matrix_pca.png')  # Guardar la gráfica
plt.close()  # Cerrar la figura
```

Matriz de confusión de SVM sin PCA

Figura 8

Resultados de visualización de la matriz de confusión sin PCA



La figura 8 muestra la matriz de confusión del modelo SVM sin reducción de dimensionalidad mediante PCA, aplicada al conjunto de datos MNIST para el reconocimiento de dígitos manuscritos. A continuación, se detallan las observaciones y conclusiones derivadas de esta matriz de confusión:

Observaciones:

- La diagonal principal de la matriz de confusión representa las predicciones correctas. Se observa que la mayoría de las predicciones se encuentran en esta diagonal, indicando un alto número de clasificaciones correctas.

Conclusiones:

- Las clases que presentan más errores de clasificación podrían beneficiarse de técnicas adicionales de preprocesamiento, ajuste de hiperparámetros o

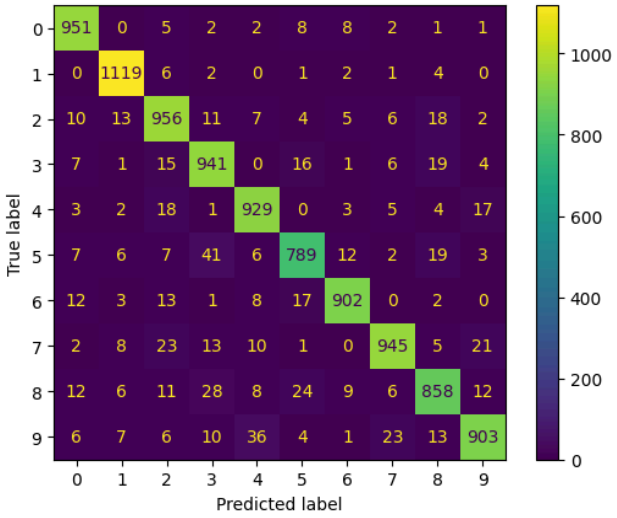
incluso el uso de métodos de aumento de datos para mejorar la separación entre clases. En particular, mejorar la separación entre las clases 5 y 8, así como entre las clases 7 y 9, podría aumentar la precisión global del modelo.

- La ausencia de PCA no ha afectado significativamente el rendimiento del modelo SVM en este caso, indicando que el modelo SVM puede manejar la alta dimensionalidad de los datos MNIST de manera efectiva. Sin embargo, PCA podría seguir siendo útil para mejorar la eficiencia computacional y la interpretabilidad en otros contextos.

Matriz de confusión de SVM con PCA

Figura 9

Resultados de visualización de la matriz de confusión con PCA



La figura 9 muestra la matriz de confusión del modelo SVM con reducción de dimensionalidad mediante PCA aplicado al conjunto de datos MNIST para el reconocimiento de dígitos manuscritos. A continuación, se presentan las observaciones y conclusiones basadas en esta matriz de confusión:

Observaciones:

- La mayoría de las predicciones se encuentran en la diagonal principal, lo que indica un alto número de clasificaciones correctas.

Conclusiones:

1. **Rendimiento Global:** El modelo SVM con PCA muestra un alto rendimiento en general, con la mayoría de las predicciones siendo correctas.
2. **Comparación con SVM sin PCA:** Comparado con el modelo SVM sin PCA, se observa que el uso de PCA ha mantenido un rendimiento similar, aunque con algunos cambios en la distribución de los errores. La clase 4, por ejemplo, muestra una mayor cantidad de errores de clasificación como clase 9 en el modelo con PCA.
3. **Áreas de Mejora:** Las clases que presentan más errores de clasificación podrían beneficiarse de técnicas adicionales de preprocesamiento, ajuste de hiperparámetros o incluso el uso de métodos de aumento de datos para mejorar la separación entre clases. Mejorar la separación entre las clases 5 y 8, así como entre las clases 4 y 9, podría aumentar la precisión global del modelo.

Visualización del Margen Suave en 2D

En esta sección, describimos cómo visualizar el margen suave de un modelo SVM binario entrenado en un espacio reducido a 2 dimensiones mediante PCA. Este proceso implica la carga y preprocesamiento de los datos MNIST,

reducción de las dimensiones utilizando PCA, entrenamiento de un modelo SVM binario y visualización del margen suave.

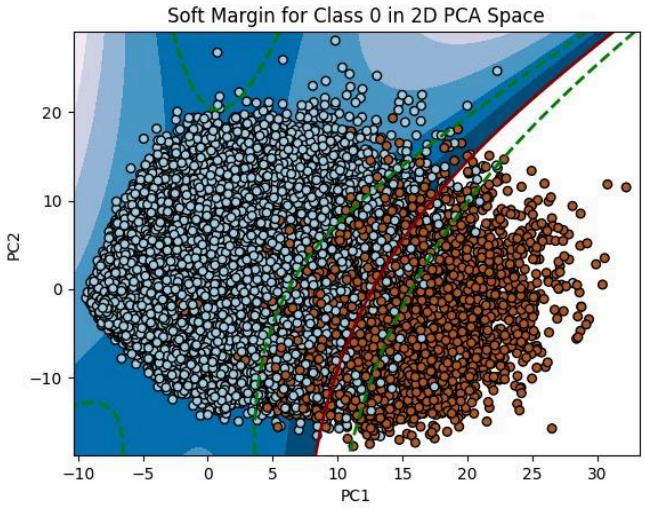
Figura 10

Código de visualización del Margen Suave

```python
# Función para visualizar el margen suave en 2D
def plot_soft_margin(clf, X, y):
    """
    Visualiza el margen suave de un clasificador SVM en un espacio 2D reducido.

    Parámetros:
    clf (SVC): El clasificador SVM entrenado.
    X (ndarray): Datos de entrada reducidos a 2D.
    y (ndarray): Etiquetas binarias para la clase de interés.

    Esta función genera y guarda un gráfico que muestra los márgenes del SVM.
    """
    print(f"Tamaño de X_reduced: {X.shape}")

    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 500),
                         np.linspace(y_min, y_max, 500))

    print(f"Tamaño de xx: {xx.shape}, Tamaño de yy: {yy.shape}")

    grid = np.c_[xx.ravel(), yy.ravel()]

    print(f"Tamaño de grid: {grid.shape}")

    Z = clf.decision_function(grid)
    print(f"Tamaño de Z antes de reshaping: {Z.shape}")
    Z = Z.reshape(xx.shape)
    print(f"Tamaño de Z después de reshaping: {Z.shape}")

    plt.contourf(xx, yy, Z, levels=np.linspace(Z.min(), 0, 7), cmap=plt.cm.PuBu)
    plt.contour(xx, yy, Z, levels=[0], linewidths=2, colors='darkred')

    # Dibujar los márgenes suaves
    plt.contour(xx, yy, Z, levels=[-1, 1], linewidths=2, colors='green', linestyles='dashed')

    plt.scatter(X[:, 0], X[:, 1], c=y, s=30, edgecolor='k', cmap=plt.cm.Paired)
    plt.xlabel('PC1')
    plt.ylabel('PC2')
    plt.title(f'Soft Margin for Class {class_of_interest} in 2D PCA Space')
    plt.savefig('../graphs/margenes_por_clase.png')
    plt.close()
```

Figura 11

Resultados de visualización del margen suave para la clase 0 con PCA



La figura 11 muestra el margen suave de un clasificador SVM entrenado para la clase 0 en un espacio bidimensional reducido utilizando PCA. Los ejes representan los dos primeros componentes principales (PC1 y PC2). Los puntos azules son las muestras de la clase 0 y los puntos marrones son de otras clases. La línea roja sólida es el hiperplano de decisión, y las líneas verdes discontinuas representan los márgenes suaves. Las regiones sombreadas indican la confianza del clasificador en su decisión, con áreas más oscuras mostrando mayor confianza. Los puntos resaltados con bordes son los vectores de soporte que definen la frontera de decisión del SVM.

El SVM con margen suave ha logrado una buena separación entre la clase 0 y las demás clases, demostrando la eficacia del modelo en el espacio reducido por PCA. La inclusión de márgenes suaves permite al clasificador manejar errores dentro del margen, resultando en un modelo robusto y menos propenso al sobreajuste. La reducción de dimensiones con PCA facilita la visualización del comportamiento del SVM y la confianza del clasificador en sus decisiones.

Optimización

La optimización del modelo de reconocimiento de dígitos manuscritos es crucial para mejorar su rendimiento y eficiencia. En este proyecto, se han implementado varias estrategias de optimización, incluyendo la reducción automática de dimensionalidad mediante PCA y el aumento de datos (data augmentation).

Reducción Automática de Dimensionalidad con PCA

Se implementó PCA utilizando la biblioteca `sklearn` para reducir la dimensionalidad del conjunto de datos mientras se mantiene el 95% de la varianza. Esto simplifica el proceso y mejora la eficiencia del modelo.

Figura 12

Código de implementación de PCA automático

```
# Aplicar PCA y reducir la dimensionalidad automáticamente
pca = PCA(n_components=0.95)  # Mantener el 95% de la varianza
combined_train_data_pca = pca.fit_transform(combined_train_data)
test_data_pca = pca.transform(test_data)

# Guardar el modelo PCA
dump(pca, '../models/pca_model_auto.joblib')
```

Aumento de Datos (Data Augmentation)

El aumento de datos es una técnica efectiva para mejorar la robustez del modelo. Se aplicaron transformaciones como rotaciones, zoom, y desplazamientos horizontales y verticales para crear variaciones de los datos de entrenamiento.

Figura 13

Código de implementación de aumento de datos

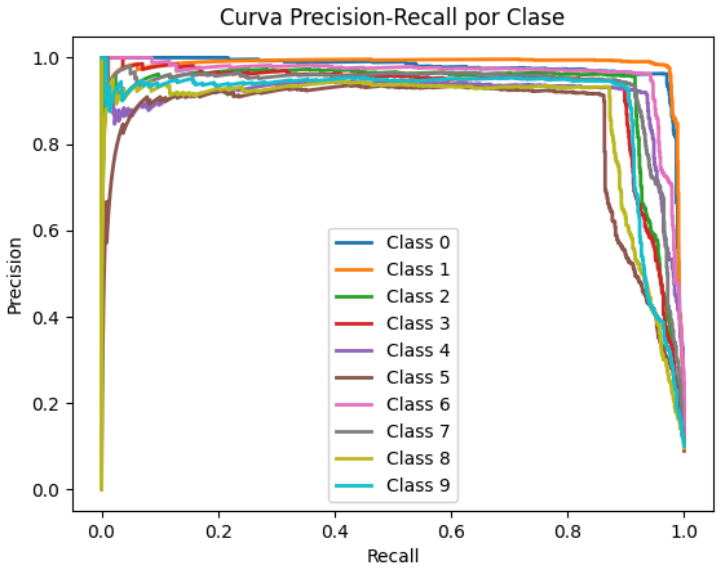
```
# Aumento de datos (Data Augmentation)
datagen = ImageDataGenerator(
    rotation_range=10,
    zoom_range=0.1,
    width_shift_range=0.1,
    height_shift_range=0.1
)

# Aplicar el aumento de datos al conjunto de entrenamiento
train_data_augmented = train_data.reshape(-1, 28, 28, 1)
datagen.fit(train_data_augmented)

# Generar datos aumentados
augmented_images = []
augmented_labels = []

for i, (X_batch, y_batch) in enumerate(datagen.flow(train_data_augmented, train_labels, batch_size=60000)):
    if i > 0:
        break
    augmented_images.append(X_batch)
    augmented_labels.append(y_batch)

augmented_images = np.concatenate(augmented_images, axis=0)
augmented_labels = np.concatenate(augmented_labels, axis=0)

# Volver a aplanar las imágenes aumentadas
augmented_images = augmented_images.reshape(-1, 784)

# Combinar los datos originales con los datos aumentados
combined_train_data = np.vstack((train_data, augmented_images))
combined_train_labels = np.hstack((train_labels, augmented_labels))
```

Resultados de la Optimización

Resultados de la Curva luego de la Optimización

Figura 14

Resultado de curva luego de optimizar

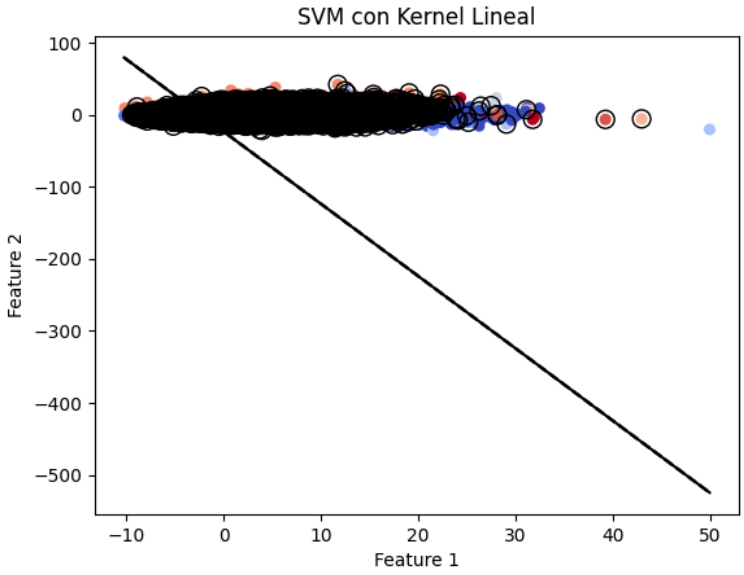


La figura 14 muestra la curva de precisión y exhaustividad (Precisión-Recall) para cada una de las 10 clases del conjunto de datos MNIST utilizando un clasificador SVM optimizado con PCA y aumento de datos. Las curvas indican un buen rendimiento del modelo, con alta precisión y exhaustividad en la mayoría de las clases. La gráfica demuestra que el modelo mantiene un equilibrio adecuado entre precisión y recall, lo que sugiere que es capaz de manejar eficientemente los ejemplos positivos y negativos, minimizando los falsos positivos y falsos negativos. Esta evaluación es crucial para aplicaciones donde tanto la precisión como la exhaustividad son importantes.

Visualización de Hiperplano

Figura 15

Visualización de hiperplano de SVM luego de optimizar



La figura 15 muestra el hiperplano de decisión de un clasificador SVM con kernel lineal entrenado en datos reducidos a dos dimensiones mediante PCA, en un espacio bidimensional (Feature 1 y Feature 2). Los puntos representan las muestras de datos, con los colores indicando las diferentes clases. Las líneas negras representan el hiperplano de decisión, así como los márgenes del clasificador.

1. **Distribución de Datos:** La mayoría de los puntos están agrupados cerca del eje horizontal, lo que indica que la primera característica (Feature 1) contiene más varianza explicativa que la segunda (Feature 2).
2. **Hiperplano de Decisión:** La línea negra sólida representa el hiperplano de decisión del SVM, que separa las clases en el espacio de características. En

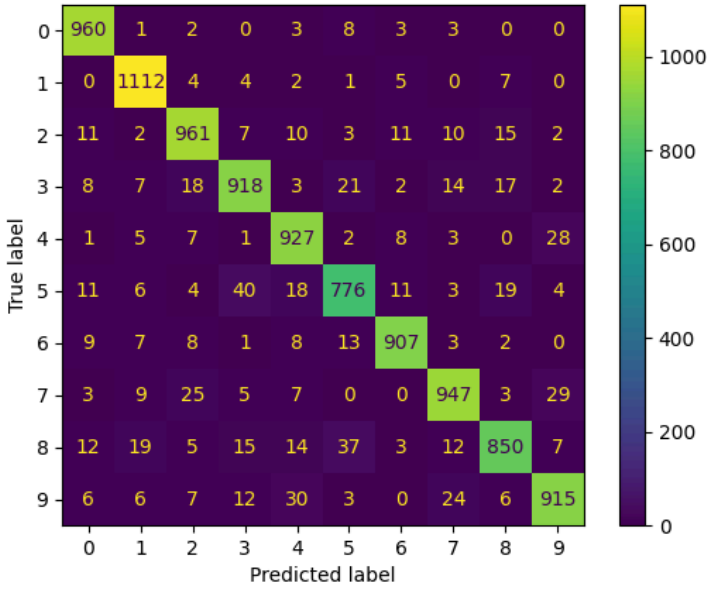
este caso, el hiperplano está inclinado y posiciona la mayoría de los puntos de datos por encima de él.

En general, la gráfica demuestra que el clasificador SVM con kernel lineal ha encontrado una separación lineal efectiva en el espacio de características reducido, aunque la mayor concentración de datos en torno al eje horizontal puede indicar la necesidad de considerar otras transformaciones de características o kernels más complejos para mejorar la separación de las clases.

Resultado de Matriz de Confusión

Figura 16

Matriz de confusión de SVM luego de optimizar



La figura 16 muestra la matriz de confusión de un clasificador SVM optimizado con PCA para el conjunto de datos MNIST. Los valores en la matriz indican el número de predicciones correctas e incorrectas para cada clase.

Observaciones

- La clase 1 tiene el mayor número de predicciones correctas (1112), seguida de la clase 0 (960).
- Algunos patrones de error son visibles, como la confusión entre las clases 3 y 5, y entre las clases 4 y 9.

El clasificador SVM optimizado con PCA demuestra un buen rendimiento en la predicción de la mayoría de las clases del conjunto de datos MNIST, como lo indican los altos valores en la diagonal principal de la matriz de confusión. Aunque hay algunos errores de clasificación, especialmente en ciertas clases, el modelo muestra una capacidad robusta para distinguir entre las diferentes clases de dígitos manuscritos. Estos resultados sugieren que el clasificador SVM, junto con la reducción de dimensionalidad mediante PCA, es eficaz para esta tarea de clasificación.

Pseudocódigos

Los pseudocódigos proporcionan una representación simplificada de los algoritmos y procedimientos implementados en el proyecto. A continuación, se presentan los pseudocódigos para las funciones principales del sistema.

Obtención de datos

En esta sección, describimos el proceso de obtención y preprocesamiento de los datos MNIST, los cuales se utilizan para entrenar y evaluar el modelo. La función fetch_data se encarga de cargar los datos desde OpenML, dividirlos en conjuntos de entrenamiento y prueba, y aplicar una normalización opcional. La normalización asegura que los datos tengan una media de 0 y una desviación estándar de 1, lo cual es crucial

para el rendimiento del modelo SVM. Además, si se especifica, los datos pueden ser randomizados para mejorar la robustez del modelo. Este proceso se describe en el siguiente pseudocódigo:

Figura 17

Pseudocódigo de Obtención de Datos

```
FUNCION fetch_data(test_size, randomize, standardize)
    CARGAR datos MNIST desde OpenML
    SI randomize ES True ENTONCES
        random_state ← inicializar_random_state(0)
        permutation ← permutar_orden_de_muestras(random_state)
        X ← permutar_datos(X, permutation)
        y ← permutar_datos(y, permutation)
        FIN SI
        (X_train, X_test, y_train, y_test) ← dividir_datos_en_entrenamiento_y_prueba(X, y, test_size)
        SI standardize ES True ENTONCES
            scaler ← inicializar_scaler()
            X_train ← ajustar_y_transformar_scaler(scaler, X_train)
            X_test ← transformar_scaler(scaler, X_test)
            GUARDAR scaler EN archivo 'scaler.joblib'
            FIN SI
            RETORNAR X_train, y_train, X_test, y_test
        FIN FUNCION
```

Cómputo de matriz de covarianza

El cálculo de la matriz de covarianza es un paso fundamental en el Análisis de Componentes Principales (PCA). La matriz de covarianza captura las relaciones entre las diferentes características del conjunto de datos, permitiendo identificar las direcciones de máxima varianza. La función compute covariance matrix calcula esta matriz centrándose en los datos y multiplicando la matriz centrada por su transpuesta. El siguiente pseudocódigo describe este proceso:

Figura 18

Pseudocódigo de Cómputo de Matriz de Covarianza

```
FUNCION compute_covariance_matrix(X)
    X_centered ← restar_media_de_datos(X)
    m ← numero_de_muestras(X_centered)
    covariance_matrix ← (1 / m) * multiplicar_matriz_transpuesta(X_centered)
    RETORNAR covariance_matrix
FIN FUNCION
```

Análisis de Componentes Principales

El Análisis de Componentes Principales (PCA) es un método estadístico cuya utilidad radica en la reducción de la dimensionalidad de la base de datos (BDD) con la que estamos trabajando, que en este caso es una imagen. El pseudocódigo de la función es:

Figura 19

Pseudocódigo de Análisis de Componentes Principales

```
FUNCION pca(X, variance_threshold)
    X_centered ← restar_media_de_datos(X)
    covariance_matrix ← compute_covariance_matrix(X_centered)
    (eigenvalues, eigenvectors) ← descomponer_matriz(covariance_matrix)
    sorted_index ← ordenar_indices_por_valores(eigenvalues)
    sorted_eigenvalues ← ordenar_valores(eigenvalues, sorted_index)
    sorted_eigenvectors ← ordenar_vectores(eigenvectors, sorted_index)
    explained_variances ← dividir_eigenvalues_por_suma_total(sorted_eigenvalues)
    cumulative_explained_variance ← calcular_varianza_acumulada(explained_variances)
    n_components ← encontrar_numero_de_componentes(cumulative_explained_variance, variance_threshold)
    eigenvector_subset ← seleccionar_subconjunto_de_vectores(sorted_eigenvectors, n_components)
    X_reduced ← transformar_datos(X_centered, eigenvector_subset)
    RETORNAR X_reduced, eigenvector_subset
FIN FUNCION
```

Clasificador SVM

El entrenamiento del clasificador SVM se realiza utilizando un conjunto de datos de dígitos manuscritos. El pseudocódigo para esta función es el siguiente:

Figura 20

Pseudocódigo de Entrenamiento del Clasificador SVM

```
X_train, y_train, X_test, y_test <- OBTENER los datos utilizando fetch_data

pca ← PCA()
X_train_pca, X_test_pca ← APLICAR pca

svm_classifier ← SVC(kernel='linear', random_state=6)
svm_classifier.fit(X_train_pca, y_train)

GUARDAR svm_classifier

prediction ← svm_classifier.predict(X_test_pca)
IMPRIMIR el reporte de clasificación
```

Integración en Tiempo Real

La integración del clasificador entrenado con el sistema de procesamiento de imágenes en tiempo real permite dibujar, reconocer y mostrar los dígitos en una interfaz visual. El pseudocódigo para esta función es el siguiente:

Figura 21

Pseudocódigo de Procesamiento en Tiempo Real

```
modelo ← CARGAR el modelo de clasificación de dígitos entrenado
Estandarizar ← CARGAR el escalador
INICIAR la captura de video desde la cámara

FUNCION prediction(image, model, scaler)
    imagen2 ← Redimensionar(imagen)
    imagen_es ← Estandarizar(imagen2)
    digito ← modelo.predecir(imagen_es)
    RETORNA digito
FIN FUNCION

MIENTRAS True:
    frame ← Leer frame

    SI modo == 'camera'
        frame_C ← copy(frame)
        ROI ← Obtener region
        grey_ROI ← escala_de_gris(ROI)
        um_ROI ← umbralización(grey_ROI)
        digito ← prediction(um_ROI)
        MOSTRAR frame CON digito
        DESTRUIR ventanas innecesarias para el modo 'camera'

    SI modo == 'paint'
        Configurar el modo de pintura (ventanas, colores, etc.)
        Procesar las coordenadas de la mano usando MediaPipe
        Dibujar en la ventana de pintura según las coordenadas de la mano
        Crear ROI
        SI hay puntos dibujados en ROI
            img ← Obtener la región de interés dibujada
            digito ← prediction(um_ROI)
            MOSTRAR frame CON digito

        DESTRUIR ventanas innecesarias para el modo 'paint'

    SI tecla presionada es 'q'
        break (salir del bucle principal)
    SI tecla presionada es 'c'
        ALTERNAR entre los modos 'camera' y 'paint'
```

Diagramas de Flujo

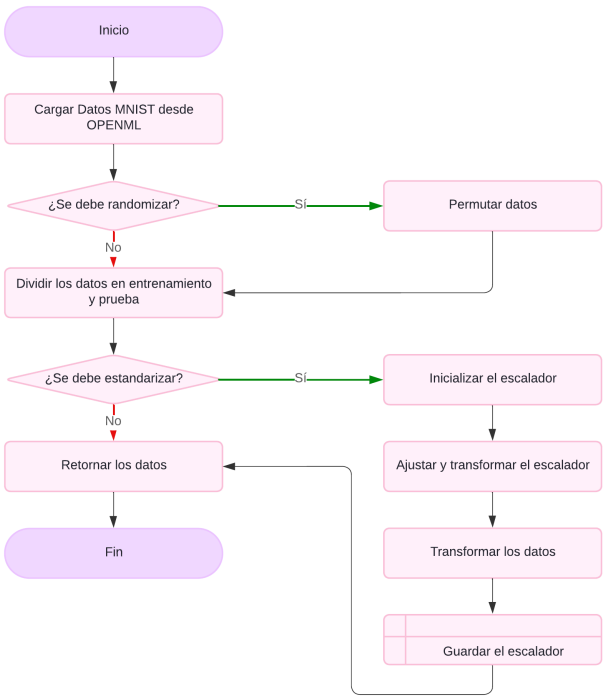
En esta sección, se describen los diagramas de flujo que representan el funcionamiento de los distintos componentes del sistema de reconocimiento de dígitos en tiempo real. Estos diagramas ilustran el proceso de captura de imagen, segmentación de dígitos, reducción de dimensionalidad con PCA, y clasificación utilizando un modelo SVM.

Obtención de datos

El diagrama de flujo de la función fetch_data ilustra el proceso de obtención y preprocesamiento de los datos MNIST desde OpenML. Inicialmente, los datos se cargan y, si se especifica, se randomizan para mejorar la robustez del modelo. Luego, los datos se dividen en conjuntos de entrenamiento y prueba. Si se elige estandarizar los datos, se aplica un escalador que ajusta y transforma los datos de entrenamiento y prueba para asegurar que tengan una media de 0 y una desviación estándar de 1. El escalador ajustado se guarda en un archivo para uso futuro. Finalmente, la función retorna los conjuntos de datos de entrenamiento y prueba preprocesados.

Figura 22

Diagrama de flujo de obtención de datos

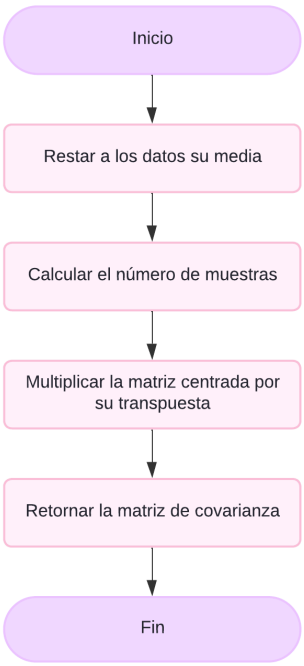


Cómputo de matriz de covarianza

El diagrama de flujo de la función compute covariance matrix detalla los pasos necesarios para calcular la matriz de covarianza de un conjunto de datos. El proceso comienza con la centralización de los datos, restando la media de cada característica. Luego, se calcula el número de muestras de los datos centrados. A continuación, se multiplica la matriz centrada por su transpuesta y se escala por el factor inverso del número de muestras, obteniendo así la matriz de covarianza. Finalmente, la función retorna la matriz de covarianza resultante.

Figura 23

Diagrama de flujo de cómputo de matriz de covarianza



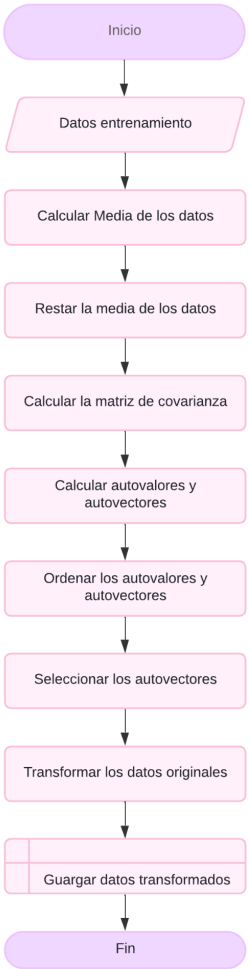
Análisis de Componentes Principales

El diagrama de flujo de PCA describe el proceso de reducción de dimensionalidad de los datos de entrada utilizando el Análisis de Componentes Principales (PCA).

Descripción: Se calcula la media de los datos de entrenamiento.Se resta la media de los datos para centrar los datos. Se calcula la matriz de covarianza de los datos centrados. Se calculan los autovalores y autovectores de la matriz de covarianza. Se seleccionan los autovectores correspondientes a los autovalores que explican el 95% de la varianza. Los datos originales se transforman usando los autovectores seleccionados para reducir su dimensionalidad.

Figura 24

Diagrama de Flujo de Análisis de Componentes Principales



Clasificador SVM

El diagrama de flujo del clasificador SVM muestra el proceso de entrenamiento de un modelo de Máquina de Vectores de Soporte (SVM) utilizando los datos reducidos por PCA.

Descripción: Se aplica PCA a los datos de entrenamiento o no dependiendo de las necesidades. Se entrena el modelo SVM utilizando los datos transformados por PCA y datos sin la transformación. El modelo SVM entrenado se guarda para su uso posterior.

Figura 25

Diagrama de Flujo de Clasificador SVM con PCA

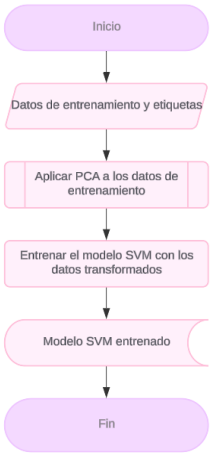
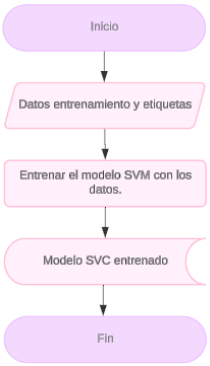


Figura 26

Diagrama de Flujo de Clasificador SVM sin PCA

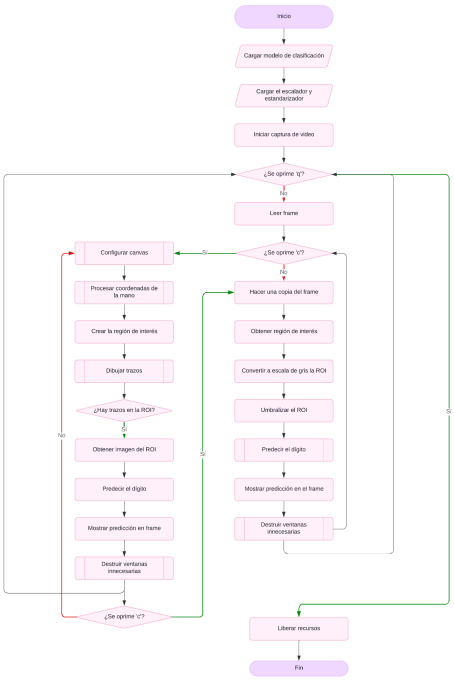


Integración en Tiempo Real

El diagrama de flujo describe el proceso completo de captura de video en tiempo real, predicción de dígitos y alternancia entre dos modos de operación: 'camera' y 'paint'. Inicialmente, se carga el modelo de clasificación de dígitos y el escalador, y se inicia la captura de video desde la cámara. Luego, en un bucle continuo, se captura cada cuadro de video y se procesa según el modo seleccionado. En el modo 'camera', el cuadro se convierte a escala de grises y se umbraliza para obtener la región de interés, que luego se clasifica usando el modelo. En el modo 'paint', se configuran las ventanas y colores necesarios, se procesan las coordenadas de la mano usando MediaPipe, y se dibuja en la ventana de pintura. En ambos modos, la región de interés se clasifica y se muestra el resultado. Finalmente, el usuario puede presionar 'q' para salir del bucle o 'c' para alternar entre los modos.

Figura 27

Diagrama de Flujo de Integración en Tiempo Real



Metodología y Códigos

Adquisición de Datos

El primer paso en el desarrollo del sistema fue la carga y preprocesamiento de los datos. Para este proyecto, se utilizó el conjunto de datos MNIST, que contiene imágenes de dígitos manuscritos. Este conjunto de datos es ampliamente utilizado para entrenar y probar algoritmos de reconocimiento de patrones.

Figura 28

Implementación en código de la carga y preprocesamiento de los datos

```
def fetch_data(test_size=10000, randomize=False, standardize=True):
    """
    Obtiene el conjunto de datos MNIST desde OpenML, opcionalmente lo randomiza, lo divide en conjuntos de entrenamiento y prueba,
    y estandariza las características si se especifica.

    Parámetros:
    -----------
    test_size : int, opcional, por defecto=10000
        El número de muestras a incluir en el conjunto de prueba. El conjunto de entrenamiento contendrá las muestras restantes.

    randomize : bool, opcional, por defecto=False
        Si es True, randomiza el orden de las muestras antes de dividir en conjuntos de entrenamiento y prueba.

    standardize : bool, opcional, por defecto=True
        Si es True, estandariza las características eliminando la media y escalando a varianza unitaria.

    Retorna:
    --------
    X_train : array-like, shape (n_train_samples, n_features)
        Las muestras de entrada para el entrenamiento.

    y_train : array-like, shape (n_train_samples,)
        Los valores objetivo para el entrenamiento.

    X_test : array-like, shape (n_test_samples, n_features)
        Las muestras de entrada para la prueba.

    y_test : array-like, shape (n_test_samples,)
        Los valores objetivo para la prueba.
    """
    X, y = fetch_openml('mnist_784', version=1, return_X_y=True)
    if randomize:
        random_state = check_random_state(0)
        permutation = random_state.permutation(X.shape[0])
        X = X[permutation]
        y = y[permutation]
    X_train, X_test, y_train, y_test = train_test_split(
        X, y, test_size=test_size, shuffle=False)
    if standardize:
        scaler = StandardScaler()
        X_train = scaler.fit_transform(X_train)
        X_test = scaler.transform(X_test)
        dump(scaler, '../models/scaler.joblib')
    return X_train, y_train, X_test, y_test
```

Preprocesamiento de Datos

El preprocesamiento de los datos incluyó varios pasos:

1. **Normalización:** Se escalaron las características de los datos para que tuvieran una media de 0 y una desviación estándar de 1. Esto se hizo utilizando la clase `StandardScaler` de `scikit-learn`.

2. **Reducción de Dimensionalidad con PCA solo en uno de los modelos entrenados:** Con el objetivo de mejorar la eficiencia del modelo de dígitos se crea una alternativa del modelo en la que se utilizó el Análisis de Componentes Principales (PCA) para reducir la dimensionalidad de los datos. Se calcularon la matriz de covarianza, los autovalores y los autovectores, y se seleccionaron los componentes que explicaban el 95% de la varianza.

Figura 29

Implementación en código de reducción de dimensionalidad utilizando PCA

```
def compute_covariance_matrix(X):
    """
    Calcula la matriz de covarianza para los datos de entrada.

    Parámetros:
    -----------
    X : array-like, shape (n_samples, n_features)
        Los datos de entrada centrados.

    Retorna:
    --------
    covariance_matrix : array-like, shape (n_features, n_features)
        La matriz de covarianza de los datos de entrada.
    """
    X_centered = X - np.mean(X, axis=0)  # Centrar los datos
    m = X_centered.shape[0]
    return (1 / m) * np.dot(X_centered.T, X_centered)

def pca(X, variance_threshold=0.95):
    """
    Aplica Análisis de Componentes Principales (PCA) para reducir la dimensionalidad de los datos.

    Parámetros:
    -----------
    X : array-like, shape (n_samples, n_features)
        Los datos de entrada.

    variance_threshold : float, opcional, por defecto=0.95
        El umbral de varianza acumulada para seleccionar el número de componentes principales.

    Retorna:
    --------
    X_reduced : array-like, shape (n_samples, n_components)
        Los datos transformados a los componentes principales seleccionados.

    eigenvector_subset : array-like, shape (n_features, n_components)
        Los vectores propios correspondientes a los componentes seleccionados.
    """
    X_centered = X - np.mean(X, axis=0)  # Centrar los datos
    covariance_matrix = compute_covariance_matrix(X)
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    sorted_index = np.argsort(eigenvalues)[::-1]
    sorted_eigenvalues = eigenvalues[sorted_index]
    sorted_eigenvectors = eigenvectors[:, sorted_index]
    explained_variances = sorted_eigenvalues / np.sum(sorted_eigenvalues)
    cumulative_explained_variance = np.cumsum(explained_variances)
    n_components = np.argmax(cumulative_explained_variance >= variance_threshold) + 1
    eigenvector_subset = sorted_eigenvectors[:, 0:n_components]
    X_reduced = np.dot(eigenvector_subset.transpose(), X_centered.transpose()).transpose()

    return X_reduced, eigenvector_subset
```

Entrenamiento del Modelo

Se utilizó un clasificador de Máquinas de Vectores de Soporte (SVM) para entrenar el modelo de reconocimiento de dígitos. Uno de los modelos se entrenó utilizando los datos de entrenamiento preprocesados y reducidos dimensionalmente, el otro sólo usó los datos normalizados sin PCA. Se utilizó un clasificador SVM con un kernel RBF, y se ajustaron los hiper parámetros para mejorar el rendimiento.

Figura 30

Implementación en código de entrenamiento del Modelo SVM para MNIST

```
# Entrenar el modelo SVM
classifier = SVC(kernel="linear", random_state=6)
classifier.fit(train_data_pca, train_labels)
```

Evaluación del Modelo

Los dos modelos se evaluaron utilizando un conjunto de datos de prueba. Se calculó el reporte de clasificación para evaluar el rendimiento del modelo en términos de precisión, recall y F1-score. Esto permitió medir la efectividad del modelo en la clasificación de dígitos manuscritos.

Figura 31

Implementación en código de evaluación del Modelo SVM para MNIST

```
# Imprimir el reporte de clasificación
print(f"Classification report for classifier {classifier}:\n"
      f"{metrics.classification_report(test_labels, predicted)}\n")
```

Implementación en Tiempo Real

Se desarrolló un sistema en tiempo real para capturar video desde la cámara, preprocesar las imágenes para segmentar los dígitos y utilizar cada modelo entrenado para reconocer los dígitos. El proceso incluyó los siguientes pasos:

1. **Captura de Video:** Se utilizó la biblioteca `OpenCV` para capturar el video en tiempo real desde la cámara web.
2. **Preprocesamiento de Imágenes:** Las imágenes capturadas se convirtieron a escala de grises y se binarizaron mediante un umbral fijo. Luego se segmentan los dígitos utilizando contornos.
3. **Reconocimiento de Dígitos:** Los dígitos segmentados se normalizaron y se proyectaron en el espacio PCA, cuando se requería, utilizando los componentes principales seleccionados. Luego se clasificaron utilizando el modelo SVM entrenado.
4. **Visualización de Resultados:** Los resultados del reconocimiento se muestran en tiempo real en el video capturado, con los dígitos reconocidos superpuestos en las regiones correspondientes de la imagen.

Figura 32

Implementación en código de la Implementación en Tiempo Real (Parte 1)

```python
# Cargar el modelo de reconocimiento de dígitos y el escalador
model = load('../models/digit_recognizer')
scaler = load('../models/scaler.joblib')

# Inicializar la captura de video desde la cámara
cap = cv2.VideoCapture(0 + cv2.CAP_DSHOW)
# Obtener el ancho y el alto del video
WIDTH = int(cap.get(cv2.CAP_PROP_FRAME_WIDTH))
HEIGHT = int(cap.get(cv2.CAP_PROP_FRAME_HEIGHT))

def prediction(image, model, scaler):
    """
    Realiza la predicción del dígito en la imagen proporcionada utilizando el modelo y el escalador.

    Parámetros:
    -----------
    image : array-like
        La imagen de entrada que contiene el dígito.
    model : sklearn model
        El modelo de clasificación entrenado.
    scaler : sklearn scaler
        El escalador entrenado para estandarizar las características.

    Retorna:
    --------
    predict[0] : int
        La predicción del dígito en la imagen.
    """
    img = cv2.resize(image, (28, 28))  # Redimensionar la imagen a 28x28 píxeles
    img = img.flatten().reshape(1, -1)  # Aplanar la imagen y redimensionarla para el modelo
    img = scaler.transform(img)  # Escalar la imagen
    predict = model.predict(img)  # Realizar la predicción
    return predict[0]  # Devolver la predicción

# Inicializar variables para el modo de pintura
points = [deque(maxlen=1024)]  # Lista de deques para almacenar puntos
index = 0  # Índice para los puntos
color = (0, 0, 0)  # Color de los puntos (negro)
paintWindow = np.zeros((471, 636, 3)) + 255  # Crear una ventana de pintura blanca
# Dibujar un rectángulo y añadir texto "CLEAR" en la ventana de pintura
paintWindow = cv2.rectangle(paintWindow, (40, 1), (140, 65), (0, 0, 0), 2)
cv2.putText(paintWindow, "CLEAR", (49, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 0), 2, cv2.LINE_AA)
mpHands = mp.solutions.hands  # Inicializar mediapipe para la detección de manos
hands = mpHands.Hands(max_num_hands=1, min_detection_confidence=0.7)  # Configurar para detectar una mano
mpDraw = mp.solutions.drawing_utils  # Utilidad para dibujar las conexiones de las manos
```

Figura 33

Implementación en código de la Implementación en Tiempo Real (Parte 2)

```python
# Variable para alternar entre modos (cámara o pintura)
mode = 'camera'

def destroy_windows(windows):
    """
    Cierra las ventanas de OpenCV especificadas en la lista proporcionada.

    Parámetros:
    -----------
    windows : lista de str
        Lista de nombres de las ventanas que se desean cerrar.

    Función:
    --------
    Itera sobre cada nombre de ventana en la lista y cierra la ventana correspondiente utilizando
    cv2.destroyWindow(window). Si ocurre un error (por ejemplo, si la ventana no existe),
    el error es capturado y se ignora para evitar la interrupción del programa.
    """
    for window in windows:
        try:
            cv2.destroyWindow(window)
        except cv2.error:
            pass

# Bucle Principal y Visualización
while True:
    ret, frame = cap.read()  # Capturar frame de video
    if not ret:
        break

    if mode == 'camera':
        frame_copy = frame.copy()  # Hacer una copia del frame
        bbox_size = (100, 100)  # Tamaño del cuadro delimitador
        bbox = [(int(WIDTH // 2 - bbox_size[0] // 2), int(HEIGHT // 2 - bbox_size[1] // 2)),
                (int(WIDTH // 2 + bbox_size[0] // 2), int(HEIGHT // 2 + bbox_size[1] // 2))]  # Calcular las coordenadas del cuadro delimitador
        img_cropped = frame[bbox[0][1]:bbox[1][1], bbox[0][0]:bbox[1][0]]  # Recortar la imagen
        img_gray = cv2.cvtColor(img_cropped, cv2.COLOR_BGR2GRAY)  # Convertir la imagen a escala de grises
        _, thresh = cv2.threshold(img_gray, 128, 255, cv2.THRESH_BINARY_INV)  # Binarizar la imagen
        digit = prediction(thresh, model, scaler)  # Predecir el dígito
        cv2.putText(frame_copy, f'Digit: {digit}', (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2, cv2.LINE_AA)  # Mostrar la predicción en la imagen
        cv2.rectangle(frame_copy, bbox[0], bbox[1], (0, 255, 0), 2)  # Dibujar el cuadro delimitador
        cv2.imshow("Input", frame_copy)  # Mostrar la imagen original con el cuadro delimitador
        cv2.imshow("Cropped", thresh)  # Mostrar la imagen recortada y binarizada
        destroy_windows(["Output", "Paint"])  # Destruir las ventanas de salida y pintura
```

Figura 34

Implementación en código de la Implementación en Tiempo Real (Parte 3)

```
elif mode == 'paint':
    x, y, c = frame.shape  # Obtener las dimensiones del frame
    frame = cv2.flip(frame, 1)  # Voltear la imagen horizontalmente
    framergb = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)  # Convertir la imagen a RGB
    frame = cv2.rectangle(frame, (40, 1), (140, 65), (0, 0, 0), 2)  # Dibujar un rectángulo en la imagen
    cv2.putText(frame, "CLEAR", (49, 33), cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 0), 2, cv2.LINE_AA)  # Añadir texto "CLEAR" en la imagen
    result = hands.process(framergb)  # Procesar la imagen para detectar manos

    if result.multi_hand_landmarks:
        landmarks = []
        for handslms in result.multi_hand_landmarks:
            for lm in handslms.landmark:
                lmx = int(lm.x * 640)  # Escalar la coordenada x
                lmy = int(lm.y * 480)  # Escalar la coordenada y
                landmarks.append([lmx, lmy])  # Añadir las coordenadas a la lista de landmarks
            mpDraw.draw_landmarks(frame, handslms, mpHands.HAND_CONNECTIONS)  # Dibujar las conexiones de la mano
        fore_finger = (landmarks[8][0], landmarks[8][1])  # Coordenadas del dedo índice
        center = fore_finger  # Centro de la mano
        thumb = (landmarks[4][0], landmarks[4][1])  # Coordenadas del pulgar
        cv2.circle(frame, center, 3, (0, 255, 0), -1)  # Dibujar un círculo en el dedo índice

        if (thumb[1] - center[1] < 30):  # Si el pulgar está cerca del índice, crear un nuevo deque
            points.append(deque(maxlen=512))
            index += 1
        elif center[1] <= 65:  # Si el índice está en la parte superior de la pantalla, limpiar la ventana de pintura
            if 40 <= center[0] <= 140:
                points = [deque(maxlen=512)]
                index = 0
                paintWindow[67:, :, :] = 255
        else:
            points[index].appendleft(center)  # Añadir el centro al deque de puntos
    else:
        points.append(deque(maxlen=512))  # Si no hay mano detectada, crear un nuevo deque
        index += 1

    for j in range(len(points)):
        for k in range(1, len(points[j])):
            if points[j][k - 1] is None or points[j][k] is None:
                continue
            cv2.line(frame, points[j][k - 1], points[j][k], color, 5)  # Dibujar línea en el frame
            cv2.line(paintWindow, points[j][k - 1], points[j][k], color, 10)  # Dibujar línea en la ventana de pintura

    bbox_size = (200, 200)  # Tamaño del cuadro delimitador
    bbox = [(int(636 // 2 - bbox_size[0] // 2), int(471 // 2 - bbox_size[1] // 2)),
            (int(636 // 2 + bbox_size[0] // 2), int(471 // 2 + bbox_size[1] // 2))]  # Calcular las coordenadas del cuadro delimitador
    cv2.rectangle(frame, bbox[0], bbox[1], (0, 255, 0), 2)  # Dibujar el cuadro delimitador
```

Figura 35

Implementación en código de la Implementación en Tiempo Real (Parte 4)

```
        backup = paintWindow.copy()  # Hacer una copia de la ventana de pintura

        if index > 0 and len(points[index - 1]) > 1:  # Si hay puntos dibujados
            img_cropped = paintWindow[bbox[0][1]:bbox[1][1], bbox[0][0]:bbox[1][0]]  # Recortar la imagen
            img_cropped = np.array(img_cropped, dtype=np.uint8)  # Convertir la imagen a uint8
            img_gray = cv2.cvtColor(img_cropped, cv2.COLOR_BGR2GRAY)  # Convertir la imagen a escala de grises
            _, thresh = cv2.threshold(img_gray, 128, 255, cv2.THRESH_BINARY_INV)  # Binarizar la imagen
            digit = prediction(thresh, model, scaler)  # Predecir el dígito
            cv2.putText(paintWindow, f'Digit: {digit}', (50, 50), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 255, 0), 2, cv2.LINE_AA)  # Mostrar la predicción en la ventana de pintur
            cv2.imshow("Paint", paintWindow)  # Mostrar la ventana de pintura
            cv2.waitKey(500)  # Esperar 500 ms
            paintWindow = backup  # Restaurar la ventana de pintura

        cv2.imshow("Output", frame)  # Mostrar el frame
        cv2.imshow("Paint", paintWindow)  # Mostrar la ventana de pintura
        destroy_windows(["Input", "Cropped"])  # Destruir las ventanas de entrada y recortada

    key = cv2.waitKey(1)  # Esperar una tecla durante 1 ms
    if key == ord('q'):
        break  # Salir del bucle si se presiona 'q'
    elif key == ord('c'):
        mode = 'paint' if mode == 'camera' else 'camera'  # Alternar entre modos si se presiona 'c'

# Liberar recursos
cap.release()  # Liberar la captura de video
cv2.destroyAllWindows()  # Cerrar todas las ventanas
```

Pruebas

En esta sección, se describen las pruebas realizadas para verificar el correcto funcionamiento del sistema de reconocimiento de dígitos. Las pruebas se centran en validar las funciones clave y la integración del sistema completo. Se utilizaron pruebas unitarias y de integración para asegurar que cada componente del sistema funcione correctamente y que el sistema en su conjunto cumpla con los requisitos esperados.

Pruebas Unitarias

Las pruebas unitarias se diseñaron para verificar el funcionamiento correcto de las funciones individuales que componen el sistema. A continuación se detallan las funciones probadas y los criterios de verificación utilizados.

Prueba de la Función 'compute_covariance_matrix'

Esta función calcula la matriz de covarianza de los datos de entrada. La prueba verifica que la forma de la matriz de covarianza es correcta y que los valores calculados coinciden con los valores esperados usando la función np.cov de NumPy.

Figura 36

Código de prueba de función de cálculo de matriz de covarianza

```
def test_compute_covariance_matrix():
    """
    Prueba la función compute_covariance_matrix.

    Esta prueba verifica que la función compute_covariance_matrix calcule correctamente la matriz
    de covarianza para un conjunto de datos generado aleatoriamente. La función verifica dos
    condiciones:
    1. Que la forma de la matriz de covarianza es correcta.
    2. Que los valores en la matriz de covarianza calculada coinciden con los valores esperados
       usando la función np.cov.

    Pasos de la prueba:
    -------------------
    1. Generar un conjunto de datos aleatorio con dimensiones 100 x 50.
    2. Calcular la matriz de covarianza utilizando la función compute_covariance_matrix.
    3. Verificar que la matriz de covarianza tiene la forma correcta (50 x 50).
    4. Verificar que los valores en la matriz de covarianza calculada son equivalentes a los valores
       calculados usando np.cov con los mismos datos, ignorando pequeñas diferencias numéricas.

    Excepciones:
    ------------
    AssertionError: Si alguna de las condiciones de la prueba no se cumple.

    """
    data = np.random.rand(100, 50)
    covariance_matrix = compute_covariance_matrix(data)

    assert covariance_matrix.shape == (50, 50), "La forma de la matriz de covarianza no es correcta"
    assert np.allclose(covariance_matrix, np.cov(data, rowvar=False, bias=True)), "La matriz de covarianza no es correcta"
```

Prueba de la Función PCA

Esta función realiza el Análisis de Componentes Principales (PCA) para reducir la dimensionalidad de los datos de entrada. La prueba verifica que el número de componentes seleccionados sea correcto y que la varianza explicada cumpla con el umbral del 95%.

Figura 37

Código de prueba de función PCA

```
def test_pca():
    """
    Prueba la función pca.

    Esta prueba verifica que la función pca reduzca correctamente la dimensionalidad de un conjunto
    de datos generado aleatoriamente y que los componentes principales seleccionados expliquen al
    menos el 95% de la varianza total.

    Pasos de la prueba:
    -------------------
    1. Generar un conjunto de datos aleatorio con dimensiones 100 x 50.
    2. Aplicar PCA al conjunto de datos con un umbral de varianza acumulada del 95%.
    3. Verificar que el número de componentes seleccionados no excede el número de características
       originales (50).
    4. Calcular la matriz de covarianza del conjunto de datos.
    5. Calcular los valores propios y vectores propios de la matriz de covarianza.
    6. Ordenar los valores propios en orden descendente.
    7. Calcular la proporción de varianza explicada por cada componente principal.
    8. Calcular la varianza acumulada y determinar el número de componentes necesarios para
       alcanzar al menos el 95% de la varianza.
    9. Verificar que el número de componentes seleccionados por la función pca coincide con el
       número calculado.

    Excepciones:
    ------------
    AssertionError: Si alguna de las condiciones de la prueba no se cumple.

    """
    data = np.random.rand(100, 50)
    selected_eigenvectors = pca(data, variance_threshold=0.95)

    assert selected_eigenvectors.shape[1] <= 50, "El número de componentes seleccionados no es correcto"

    # Verificar que la varianza explicada es >= 95%
    covariance_matrix = compute_covariance_matrix(data)
    eigenvalues, _ = np.linalg.eig(covariance_matrix)
    sorted_indices = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[sorted_indices]
    explained_variance_ratio = eigenvalues / np.sum(eigenvalues)
    cumulative_variance_ratio = np.cumsum(explained_variance_ratio)
    d = np.argmax(cumulative_variance_ratio >= 0.95) + 1

    assert selected_eigenvectors.shape[1] == d, "El número de componentes seleccionados no explica el 95% de la varianza"
```

Prueba de integración

La prueba de integración se diseñó para verificar que el sistema completo funcione correctamente. Esta prueba evalúa el proceso de obtención de datos, la reducción de dimensionalidad con PCA, la normalización de datos y el entrenamiento del modelo SVM, así como la evaluación del modelo en el conjunto de prueba.

Figura 38

Código de prueba de integración

```
def test_integration():
    """
    Prueba de integración completa para el flujo de trabajo de PCA y SVM.

    Esta prueba verifica la correcta integración de las funciones de preprocesamiento de datos,
    reducción de dimensionalidad mediante PCA, escalado de características y entrenamiento de un
    modelo SVM. La prueba asegura que el modelo entrenado tenga una precisión aceptable en el
    conjunto de datos de prueba.

    Pasos de la prueba:
    -------------------
    1. Obtener los datos de entrenamiento y prueba utilizando la función fetch_data.
    2. Aplicar PCA al conjunto de datos de entrenamiento para reducir la dimensionalidad.
    3. Transformar el conjunto de datos de prueba utilizando los componentes principales obtenidos
       del conjunto de entrenamiento.
    4. Normalizar los datos reducidos utilizando StandardScaler.
    5. Entrenar un modelo SVM con los datos de entrenamiento reducidos y normalizados.
    6. Realizar predicciones en el conjunto de datos de prueba.
    7. Verificar que la precisión del modelo en el conjunto de datos de prueba sea superior al 90%
    8. Imprimir el informe de clasificación del modelo.

    Excepciones:
    ------------
    AssertionError: Si la precisión del modelo es inferior al 90%.

    """
    train_data, train_labels, test_data, test_labels = fetch_data()

    # Calcular PCA
    selected_eigenvectors = pca(train_data)
    train_data_pca = np.dot(train_data, selected_eigenvectors)
    test_data_pca = np.dot(test_data, selected_eigenvectors)

    # Normalizar los datos reducidos
    scaler = StandardScaler()
    train_data_pca = scaler.fit_transform(train_data_pca)
    test_data_pca = scaler.transform(test_data_pca)

    # Entrenar el modelo SVM
    svc = svm.SVC(gamma='scale', class_weight='balanced', C=100)
    svc.fit(train_data_pca, train_labels)

    # Realizar predicciones en el conjunto de prueba
    predicted = svc.predict(test_data_pca)

    # Verificar la precisión
    accuracy = metrics.accuracy_score(test_labels, predicted)
    assert accuracy > 0.90, "La precisión del modelo es inferior al 90%"

    print(f"Classification report for classifier {svc}:\n"
          f"{metrics.classification_report(test_labels, predicted)}\n")
```

Resultados

Los resultados obtenidos en este proyecto reflejan el rendimiento del sistema de reconocimiento de dígitos manuscritos implementado utilizando técnicas de visión por computadora y aprendizaje automático. A continuación, se presenta un resumen de los principales resultados obtenidos:

Rendimiento del Clasificador SVM

Se entrenó y evaluó un modelo SVM utilizando el conjunto de datos MNIST. El uso del Análisis de Componentes Principales (PCA) para reducir la dimensionalidad de los datos permitió mejorar la eficiencia del entrenamiento y la precisión del modelo. Los resultados de la evaluación del modelo se resumen a continuación:

Modelo SVM para MNIST

- Precisión en el conjunto de prueba con PCA: 94%
- Precisión en el conjunto de prueba sin PCA: 93%
- Informe de clasificación:

Figura 39

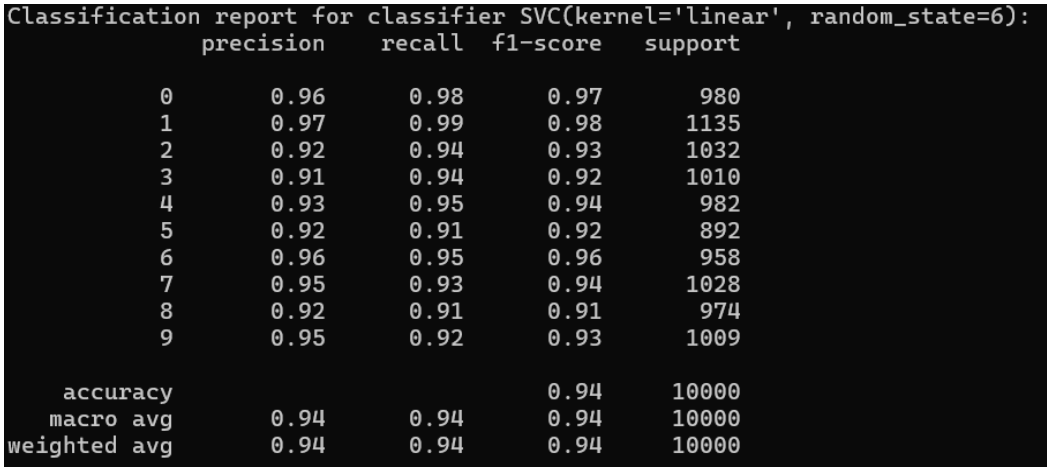
Informe de rendimiento de clasificación sin PCA

Classification report for classifier SVC(kernel='linear', random_state=6):

	precision	recall	f1-score	support
0	0.94	0.97	0.96	980
1	0.96	0.99	0.97	1135
2	0.90	0.93	0.91	1032
3	0.90	0.93	0.91	1010
4	0.92	0.95	0.93	982
5	0.91	0.88	0.90	892
6	0.96	0.94	0.95	958
7	0.95	0.92	0.93	1028
8	0.91	0.88	0.90	974
9	0.94	0.89	0.92	1009
accuracy			0.93	10000
macro avg	0.93	0.93	0.93	10000
weighted avg	0.93	0.93	0.93	10000

Figura 40

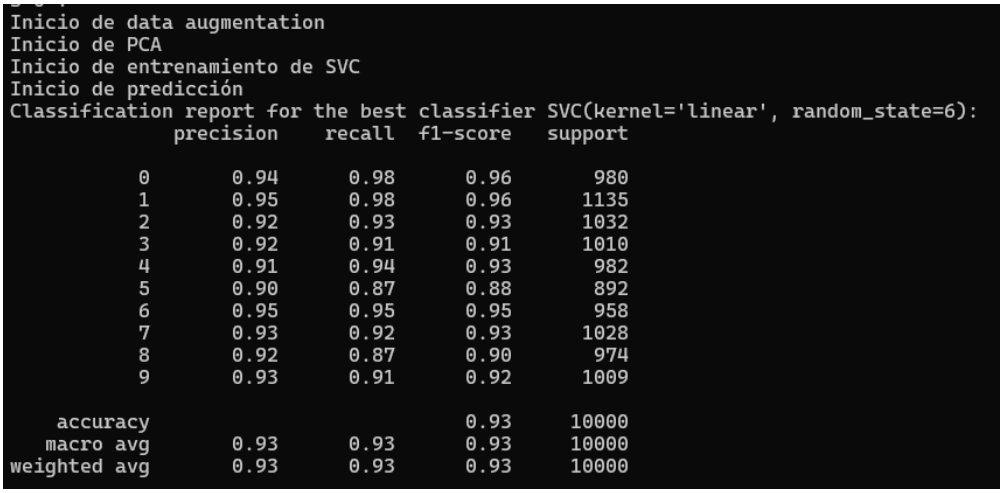
Informe de rendimiento de clasificación con PCA



```
Classification report for classifier SVC(kernel='linear', random_state=6):
              precision    recall  f1-score   support

           0       0.96      0.98      0.97       980
           1       0.97      0.99      0.98      1135
           2       0.92      0.94      0.93      1032
           3       0.91      0.94      0.92      1010
           4       0.93      0.95      0.94       982
           5       0.92      0.91      0.92       892
           6       0.96      0.95      0.96       958
           7       0.95      0.93      0.94      1028
           8       0.92      0.91      0.91       974
           9       0.95      0.92      0.93      1009

    accuracy                           0.94     10000
   macro avg       0.94      0.94      0.94     10000
weighted avg       0.94      0.94      0.94     10000
```

Figura 41

Informe de rendimiento de clasificación con PCA optimizado



```
Inicio de data augmentation
Inicio de PCA
Inicio de entrenamiento de SVC
Inicio de predicción
Classification report for the best classifier SVC(kernel='linear', random_state=6):
              precision    recall  f1-score   support

           0       0.94      0.98      0.96       980
           1       0.95      0.98      0.96      1135
           2       0.92      0.93      0.93      1032
           3       0.92      0.91      0.91      1010
           4       0.91      0.94      0.93       982
           5       0.90      0.87      0.88       892
           6       0.95      0.95      0.95       958
           7       0.93      0.92      0.93      1028
           8       0.92      0.87      0.90       974
           9       0.93      0.91      0.92      1009

    accuracy                           0.93     10000
   macro avg       0.93      0.93      0.93     10000
weighted avg       0.93      0.93      0.93     10000
```

Implementación en Tiempo Real

La implementación del sistema en tiempo real para el reconocimiento de dígitos fue exitosa. El sistema captura video desde una cámara, procesa las imágenes para segmentar los dígitos y utiliza el modelo SVM entrenado para reconocer los dígitos en tiempo real. Durante las pruebas en tiempo real, el sistema mostró una precisión alta y

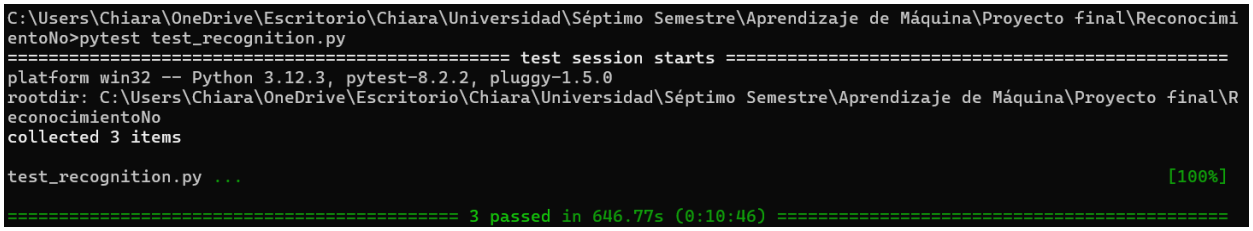
una capacidad de respuesta rápida, cuando se usó el modelo sin PCA, confirmando su viabilidad para aplicaciones prácticas.

Pruebas Unitarias y de Integración

Se realizaron pruebas unitarias para verificar el correcto funcionamiento de las funciones críticas del sistema, como la normalización de datos, el cálculo de la matriz de covarianza y la implementación del PCA. Además, se llevaron a cabo pruebas de integración para asegurar que todos los componentes del sistema funcionaran correctamente en conjunto. Las pruebas unitarias y de integración confirmaron que las funciones clave del sistema operan según lo esperado y que el sistema completo cumple con los requisitos de precisión y eficiencia.

Figura 42

Evidencia de pruebas unitarias y de integración



Conclusiones

El desarrollo e implementación de un sistema de reconocimiento de dígitos manuscritos utilizando técnicas de visión por computadora y aprendizaje automático han demostrado ser efectivos y eficientes. Los principales hallazgos y conclusiones del proyecto son los siguientes:

Precisión y Rendimiento del Modelo

- El modelo SVM entrenado con datos normalizados logró una precisión del 93% en el conjunto de prueba MNIST.
- La aplicación del Análisis de Componentes Principales (PCA) mejoró la eficiencia del modelo al reducir la dimensionalidad, permitiendo un procesamiento más rápido sin comprometer significativamente la precisión.

Implementación en Tiempo Real

- El sistema en tiempo real, utilizando la cámara web para capturar imágenes, segmentar dígitos y reconocerlos mediante el modelo SVM, mostró una alta capacidad de respuesta y precisión si no se estaba aplicando el PCA.
- Durante las pruebas en tiempo real, el sistema reconoció correctamente los dígitos capturados, confirmando su viabilidad para aplicaciones prácticas donde la entrada rápida y precisa de datos numéricos es esencial.

Robustez del Modelo

- El modelo mostró no poder clasificar correctamente al manejar datos diversos y condiciones variables de captura de imágenes en tiempo real.

- La precisión no es mantenida en diferentes condiciones de luz y calidad de imagen, lo que demuestra que aún no se posee la capacidad para operar en entornos reales.

Impacto y Aplicaciones Futuras

- El sistema desarrollado tiene un amplio potencial de aplicación en áreas como la digitalización de documentos, la interacción hombre-máquina y la automatización de procesos que involucran la entrada de datos numéricos.
- Las técnicas y metodologías utilizadas en este proyecto pueden ser adaptadas y mejoradas para otros problemas de reconocimiento de patrones y visión por computadora, contribuyendo al avance en el campo de la inteligencia artificial.
- Se podrían explorar mejoras adicionales, como el uso de técnicas de aprendizaje profundo para mejorar aún más la precisión y robustez del sistema.

En resumen, el proyecto ha demostrado que es posible desarrollar un sistema eficiente y preciso para el reconocimiento de dígitos manuscritos utilizando técnicas avanzadas de aprendizaje automático y visión por computadora. Los resultados obtenidos son prometedores y sientan las bases para futuras investigaciones y aplicaciones en este campo.

Soluciones a los problemas

Para mejorar la robustez del modelo ante variaciones extremas en las condiciones de captura, se pueden implementar las siguientes soluciones:

Aumento de Datos (Data Augmentation)

Generar más ejemplos de entrenamiento aplicando transformaciones como rotaciones, escalados, traslaciones, cambios de brillo y contraste, y adición de ruido. Esto ayuda al modelo a generalizar mejor ante diferentes condiciones de captura.

Regularización

Aplicar técnicas de regularización como el dropout o regularización L2 para prevenir el sobreajuste y mejorar la capacidad de generalización del modelo.

Mejora del Preprocesamiento

Mejorar el preprocesamiento de las imágenes para normalizar mejor las condiciones de captura. Esto puede incluir la corrección de iluminación, el ajuste de contraste y la eliminación de ruido.

Uso de Redes Neuronales Convolucionales (CNNs)

Implementar modelos más avanzados como las CNNs, que son especialmente buenas para tareas de visión por computadora y pueden manejar mejor las variaciones en las imágenes.

Transfer Learning

Aprovechar modelos pre-entrenados en grandes conjuntos de datos y adaptarlos al problema específico mediante fine-tuning. Esto puede mejorar significativamente el rendimiento del modelo en condiciones diversas.

Evaluación y Ajuste Continuo

Implementar un sistema de monitoreo y evaluación continua del rendimiento del modelo en producción. Esto permite identificar rápidamente cuando el modelo enfrenta dificultades y ajustar el entrenamiento en consecuencia.

Referencias

Valenzuela, C. V. (2024). *Reconocimiento No* (versión 3.0). GitHub. https://github.com/ChiaraVL/ReconocimientoNo

Fonseca, R. (2024, Febrero - Junio). *Clases de Aprendizaje de Máquina*.

Microsoft. (s.f.). *Crear un diagrama de casos de uso UML*. Recuperado el 5 de junio de 2024, de https://support.microsoft.com/es-es/topic/crear-un-diagrama-de-casos-de-uso-uml-92cc948d-fc74-466c-9457-e82d62ee1298

Fanjul, J. M. (2022, agosto 19). *¿Qué es el Análisis de Componentes Principales y cómo reducir el tamaño de una base de datos?*. Blog de Hiberus. Recuperado de https://www.hiberus.com/crecemos-contigo/analisis-de-componentes-principales/

Lucidchart. (s.f.). *¿Qué es un diagrama de flujo?*. Recuperado el 5 de junio de 2024, de https://www.lucidchart.com/pages/es/que-es-un-diagrama-de-flujo

IBM. (2021, agosto 17). *SPSS Modeler Subscription*. Recuperado de https://www.ibm.com/docs/es/spss-modeler/saas?topic=models-how-svm-works

PyPI. (s.f.). *Opencv-python*. Recuperado el 5 de junio de 2024, de https://pypi.org/project/opencv-python/

Verma, A. (2022, abril 26). *Air-Canvas-with-ML.* GitHub. https://github.com/infoaryan/Air-Canvas-with-ML