Facultad de Ciencias Exactas, Ingeniería y Agrimensura

Tecnicatura Universitaria en Inteligencia Artificial

PROCESAMIENTO DE IMÁGENES

IA 4.4

Informe N°1

Año: 2025

Grupo: 1

Integrantes:

- Calabozo, Nicolas - C-7551/5
- Perez, Sebastián - P-5334/1
- Lapolla, Martín Facundo - L-3372/3

Fecha de presentación: 22/10/2025

Docentes:

- Ing. Gonzalo Sad
- Ing. Juan Manuel Calle
- Ing. Joaquín Allione

Introducción

El presente informe corresponde a la resolución del Trabajo Práctico N°1, cuyo propósito es aplicar los conceptos fundamentales del procesamiento digital a través del desarrollo de dos problemas con enfoques complementarios.

En el Problema 1, se aborda el realce local de contraste mediante ecualización local del histograma, técnica orientada a resaltar detalles ocultos en imágenes con zonas de diferente iluminación. El objetivo es comprender cómo el análisis local permite mejorar el contraste en regiones específicas, frente a la pérdida de detalle que puede producir la ecualización global.

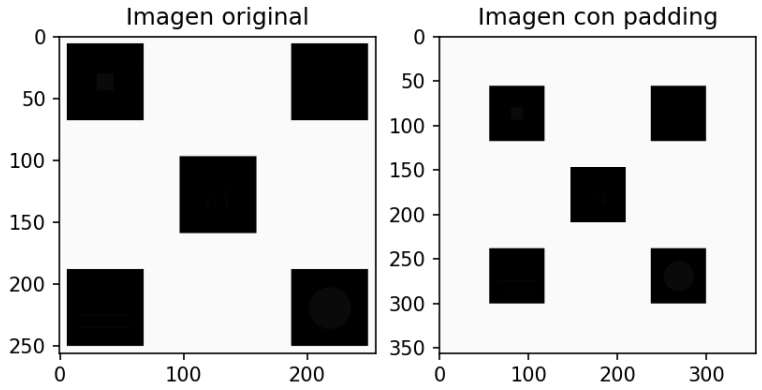
Por otro lado, el Problema 2 se plantea una aplicación de segmentación y análisis de componentes conectados para la validación automática de formularios. En este caso, se combinan técnicas de umbralado, detección de líneas, segmentación y análisis de componentes conectados y de contornos para construir un sistema capaz de verificar la corrección de distintos campos en un conjunto de imágenes con formularios digitalizados.

Ambos desarrollos se implementaron utilizando la biblioteca OpenCV, junto con NumPy y Matplotlib.

Problema 1 - Ecualización local de histograma

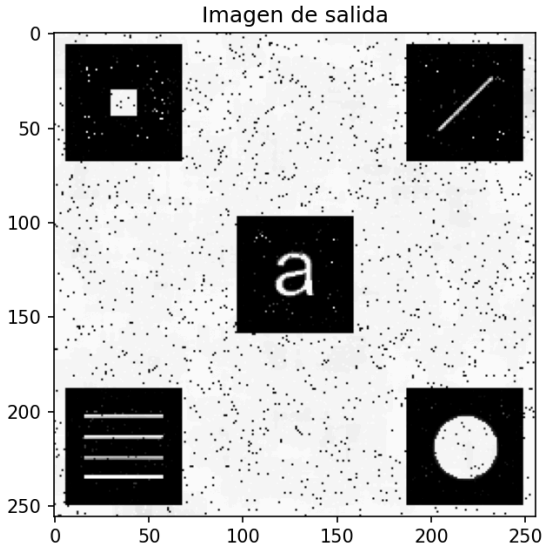
El objetivo de este problema es revelar los objetos ocultos en el archivo **‘*Imagen_con_objetos_ocultos.tiff*’**. Para lograr esto, desarrollamos una función que toma la imagen a procesar y el tamaño de la ventana (puede ser cuadrada o rectangular) como parámetros y devuelve una nueva imagen resultante de aplicar el filtrado sobre la original.

Para poder ver los detalles escondidos, aplicamos una ecualización de histograma de forma local. Elegimos hacerla local y no global porque al dividir la imagen en regiones chicas y aplicar una ecualización de histograma distinto a cada una da un mejor resultado. La transformación se adapta mejor a las características de cada zona, como el contraste , permitiendo realzar detalles que con una ecualización global quedarían invisibles, sobre todo en zonas muy claras u oscuras donde los niveles de intensidad son muy parecidos.

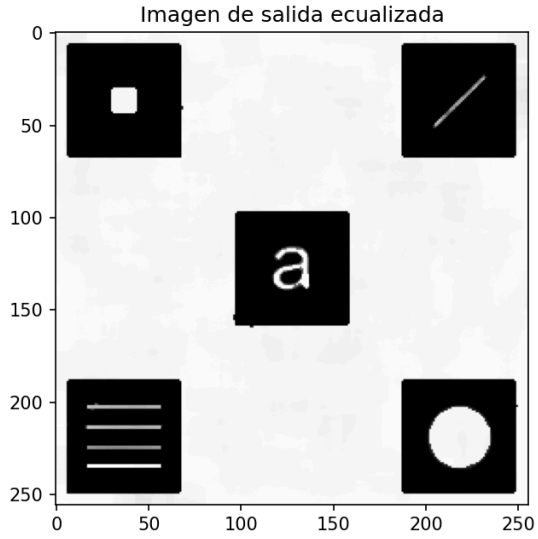


En una primera instancia lo que se hizo dentro de la función **‘ecualizacionLocalHistograma()’** es agregar un padding a la imagen. Esto es necesario porque al usar una ventana deslizante, en los bordes de la imagen habrá píxeles de la ventana que quedan afuera, y no tendríamos valores para realizar la ecualización, dando así un error de overflow. Usamos la función **‘*copyMakeBorder*()’** de OpenCV con el argumento **‘*BORDER_REPLICATE*’** para rellenar, replicando los píxeles del borde. Paralelamente creamos una matriz nueva, del mismo tamaño que la imagen original, y la rellenamos con ceros. En ella vamos a ir guardando el resultado de aplicar el filtrado.

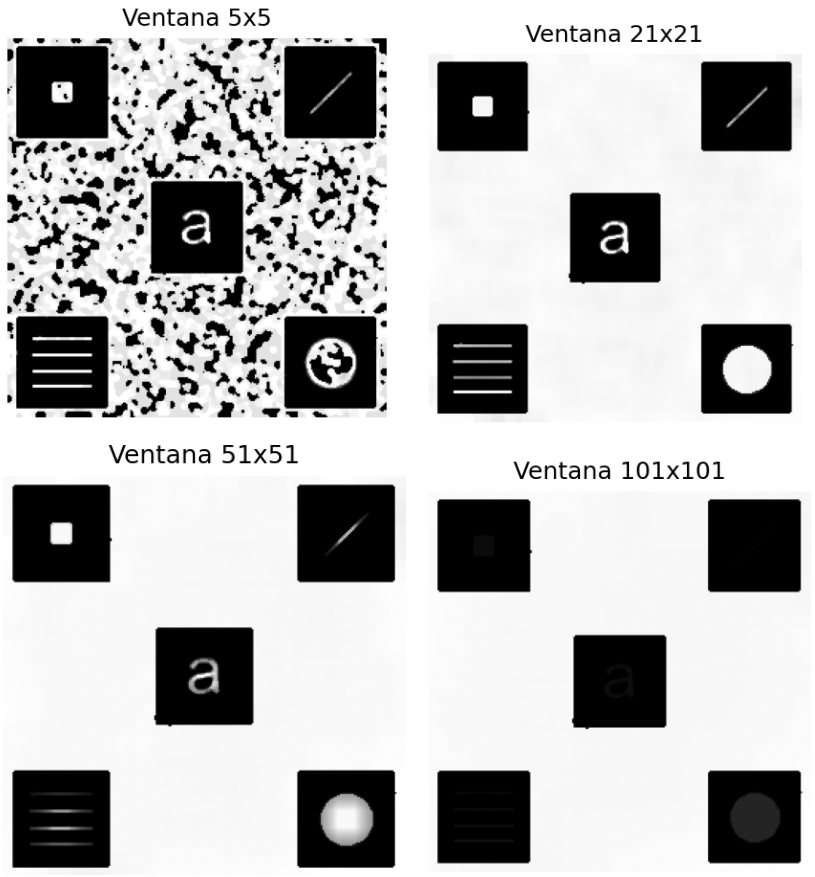
Luego de esto recorrimos cada píxel de la imagen original. En cada píxel, centramos la ventana, extrajimos esa región, le aplicamos la ecualización de histograma **‘cv2.equalizeHist()’**, y el valor actualizado del píxel central de la ventana lo almacenamos en la matriz de salida, en la posición correspondiente. Finalmente se retorna la imagen que obtenemos luego de la aplicación del filtro



Notamos que esta nueva imagen presentaba bastante ruido. Por eso, decidimos aplicar un filtro de mediana (kernel 3x3) al final. Elegimos este filtro porque es muy útil para eliminar ruido del tipo "sal y pimienta" (similar al que teníamos) y preserva bastante bien los bordes, así no perdemos los detalles que habíamos logrado resaltar con la ecualización.



El tamaño de la ventana es de gran importancia porque nos permite definir que tan “local” es nuestro análisis. Dependiendo del tamaño elegido puede suceder que se consiga resaltar detalles finos, pero volviéndose más sensible al ruido. Por otro lado elegir una ventana muy grande evita esa sensibilidad al ruido, pero perdiendo la capacidad de adaptarse a las variaciones locales puesto que el histograma local comienza a parecerse al global.



Problema 2 - Validación de formulario

El segundo problema plantea automatizar la validación de formularios escaneados, siguiendo criterios predefinidos de estructura y contenido. El objetivo principal es poder detectar automáticamente los campos del formulario, segmentarlos, luego validar su contenido y registrar los resultados.

Se nos brinda una serie de imágenes que refieren a diferentes formularios a validar almacenados en los siguientes archivos: **formulario_01.png**, **formulario_02.png, formulario_03.png, formulario_04.png, formulario_05.png**

Cada formulario contiene varios campos que deben ser validados según reglas específicas.

Desarrollo:

Para empezar hay que comprender la estructura del formulario.

Para esto, se tomó la imagen **formulario_vacio.png**, que representa el diseño base sin contenido y tras analizarla se pueden reconocer las siguientes partes clave:

FORMULARIO A		
Nombre y apellido		
Edad		
Mail		
Legajo		
	Si	No
Pregunta 1		
Pregunta 2		
Pregunta 3		
Comentarios		

Encabezado
Fila 1: Celda donde figura el tipo de formulario (A, B o C).
Campos principales
Fila 2: Celda de “Nombre y apellido” y celda con su valor
Fila 3: Celda de “Edad” y celda con su valor
Fila 4: Celda de “Mail” y celda con su valor
Fila 5: Celda de “Legajo” y celda con su valor
Fila 6: Celda vacía, celda con valor “Si” y celda con valor “No”
Fila 7: Celda de “Pregunta 1” y celda con su valor si, celda con su valor no
Fila 8: Celda de “Pregunta 2” y celda con su valor si, celda con su valor no
Fila 9: Celda de “Pregunta 3” y celda con su valor si, celda con su valor no
Fila 10: Celda de “Comentarios” y celda con su valor

Cada uno de estos campos se encuentra en una celda que está delimitado por líneas verticales y horizontales, entonces el primer desafío fue encontrar una forma de detectar automáticamente los límites de cada celda sin recurrir a coordenadas fijas, ya que podrían variar entre formularios.

Nuestro primer acercamiento al problema consistió en detectar las líneas, horizontales y verticales, de manera similar a como se hace en la versión actual, pero con la idea de usar estas líneas para luego determinar sus intersecciones y obtener los vértices de las bounding box de cada campo y luego generar las mismas. Esta idea la dejamos de lado ya que complejizaba mucho el problema y decidimos cambiar el enfoque. Utilizando tres funciones logramos una correcta, y más simple, detección de los campos con los nombres de las celdas y sus valores.

Detección de líneas:

El primer paso es la detección de líneas en la imagen, tanto verticales como horizontales, ya que las mismas representan la separación de los diferentes campos del formulario. Para esto creamos la función **‘encontrar líneas()’** que primero carga la imagen en escala de grises, se genera una máscara binaria donde los píxeles más oscuros toman valor true, luego se calculan las sumas por columnas y por filas, para ver dónde está la mayor concentración de píxeles. Finalmente, se utilizan los valores de longitud mínima para quedarnos con las posiciones donde realmente existen líneas.

Segmentación y obtención de celdas:
Ya con las líneas identificadas, lo siguiente es reconocer sus limitaciones para tener los segmentos que corresponden a cada celda.

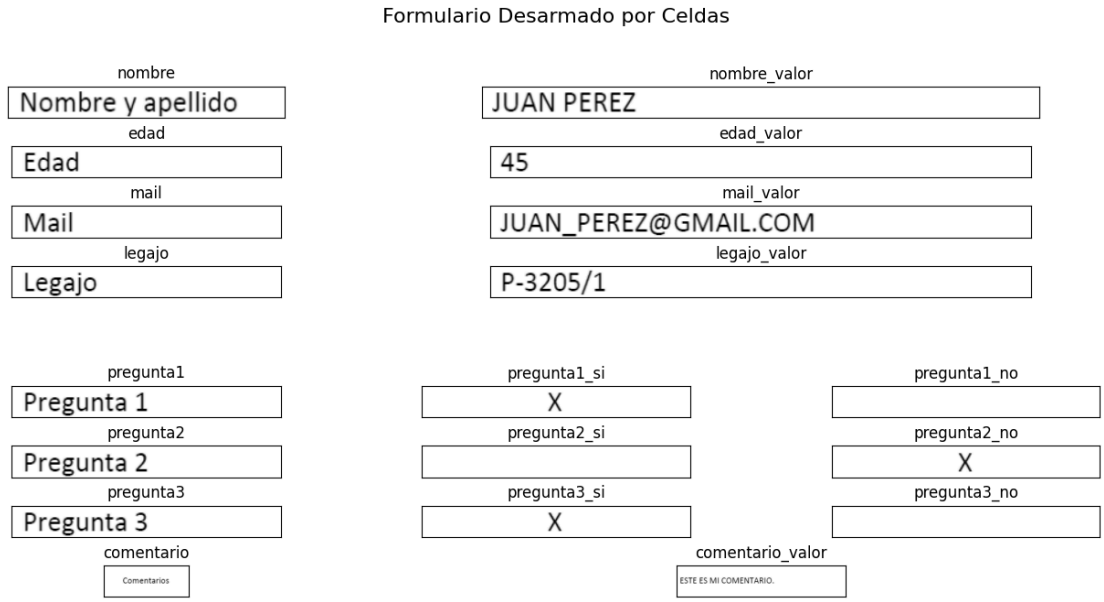
La función **‘encontrar_segmentos()’** continúa el trabajo comenzado por encontrar_linea() en base un umbral, mínimo de longitud del segmento, que eje va a analizar y coordenadas barre las filas o columnas, según corresponda, pixel a pixel buscando secuencias continuas de píxeles negros. Al encontrar el final de una secuencia de píxeles negros mide su longitud y si esta es mayor que el largo mínimo establecido guarda las coordenadas de inicio y fin para el segmento válido. Por último retorna un diccionario donde las claves son las coordenadas X o Y, y los valores son cada uno de los segmentos considerados válidos. Esta función es la que nos permite graficar las líneas delimitadoras que utilizaremos para el despiece en celdas del formulario.

Podemos observar los resultados de aplicar esta función en la siguiente imagen:

Resultado de formulario_01.png

FORMULARIO A		
Nombre y apellido	JUAN PEREZ	
Edad	45	
Mail	JUAN_PEREZ@GMAIL.COM	
Legajo	P-3205/1	
	Si	No
Pregunta 1	X	
Pregunta 2		X
Pregunta 3	X	
Comentarios	ESTE ES MI COMENTARIO.	

Nuestra última función para delimitar los campos es la función **‘encontrar_celda()’**. Esta recibe la imagen del formulario y los segmentos obtenidos en la función anterior para así poder realizar los recortes de los campos. En primera instancia la función ordena la lista de segmentos horizontales, y luego la utiliza para “cortar” la imagen en filas. Valiéndonos de conocer el orden en el que aparecen los campos, definimos explícitamente cada una de las celdas para luego guardar los recortes de cada una en un diccionario que es retornado por la función. Como precaución y para no tener líneas residuales dentro de nuestras celdas, decidimos realizar un cropping de la imagen a partir de un margen, que descartará, por defecto, dos píxeles en todas las direcciones.



Análisis de contenido:
Una vez aisladas las celdas, para interpretar su contenido, se implementó la función **‘contar_espacios_y_ palabras()’** que utiliza el etiquetado de componentes conectados de la función **‘cv2.connectedComponentsWithStats()’** de la librería OpenCV. Nuestra función primero binariza la celda, luego identifica cada carácter que lo compone y calcula las distancias entre los componentes. A partir de la distancia entre componentes y utilizando los stats, podemos definir tanto la ausencia de componentes como la presencia de caracteres, palabras y espacios utilizando un umbral dinámico.
En nuestro primer acercamiento, nos encontramos con el problema principal que no detectaba correctamente los puntos, se realizaron pruebas de diagnóstico para verificar el problema, y se terminó encontrando que los puntos, mediante el primer umbral utilizado, estaban perdiendo la suficiente cantidad de píxeles para no ser detectados como componentes por **cv2** al utilizar componentes 8-conectados. Al utilizar un nuevo umbral que captara los píxeles suficientes, el problema se resolvió automáticamente.
.

Validación:
Con esta información almacenada en un diccionario de campo-valor, utilizamos la función **‘validacion()’** que valida, llamando a la función de validación específica, cada campo según corresponda, para determinar si sus valores son correctos o no. El estado final de un formulario es correcto sólo si todos los campos que los componen son correctos. Para determinar esto último usamos la función **‘estado_formulario()’** que al pasarle los estados de cada formulario nos devuelve un estado general ‘OK’ si todos los campos son correctos o ‘MAL’ si al menos uno es incorrecto.

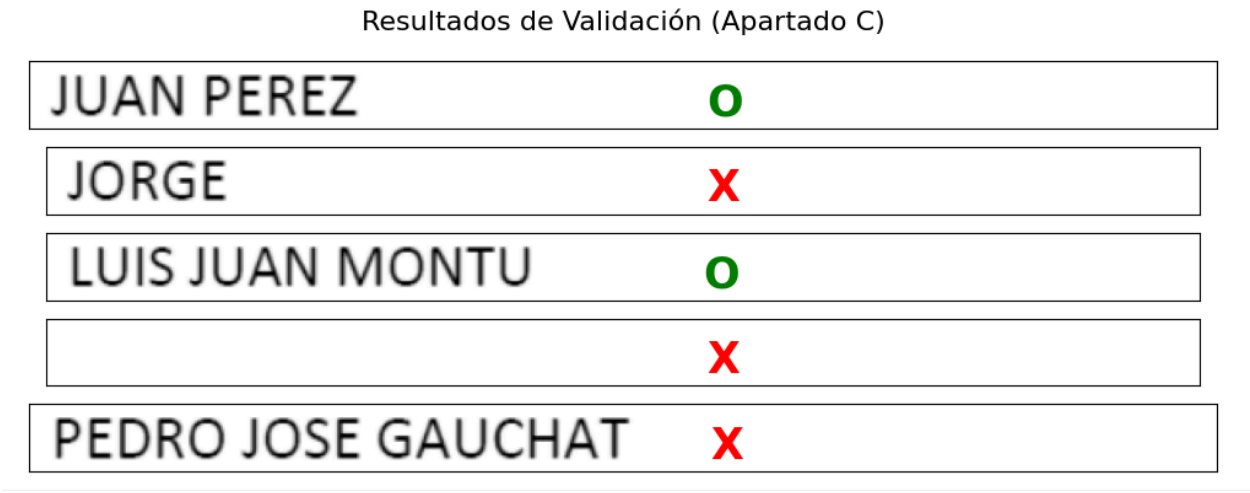
Resultados finales:
El sistema fue probado de forma cíclica para los cinco formularios provistos, arrojando los siguientes resultados por terminal:

```
-----------------------------------------
Estados de todos los formularios
-----------------------------------------
id: 01
tipo_formulario: A
nombre: OK
edad: OK
mail: OK
legajo: OK
pregunta1: OK
pregunta2: OK
pregunta3: OK
comentario: OK
¿Es un formulario válido?: True
-----------------------------------------
id: 02
tipo_formulario: A
nombre: MAL
edad: MAL
mail: MAL
legajo: MAL
pregunta1: MAL
pregunta2: MAL
pregunta3: MAL
comentario: MAL
¿Es un formulario válido?: False
-----------------------------------------
```

---
id: 03
tipo_formulario: A
nombre: OK
edad: OK
mail: OK
legajo: OK
pregunta1: OK
pregunta2: OK
pregunta3: OK
comentario: OK
¿Es un formulario válido?: True
---
id: 04
tipo_formulario: B
nombre: MAL
edad: MAL
mail: MAL
legajo: MAL
pregunta1: OK
pregunta2: MAL
pregunta3: MAL
comentario: MAL
¿Es un formulario válido?: False
---
id: 05
tipo_formulario: B
nombre: OK
edad: MAL
mail: OK
legajo: OK
pregunta1: MAL
pregunta2: MAL
pregunta3: MAL
comentario: OK
¿Es un formulario válido?: False

Con estos resultados de estado general, almacenados en una lista, y además, la lista de todos los valores de nombres y apellido que fuimos almacenado a medida que se procesaba cada formulario, utilizamos la función **‘graficar_estado_formulario()’** para generar una única imagen de salida, a modo de lista también, que en cada fila contiene, el recorte de imagen de campo nombre que identifica a un formulario y a su derecha un símbolo que demuestra su estado general. Teniendo dos casos:
Primero, en caso de estado correcto “OK”, un círculo verde
Segundo, en caso de estado incorrecto “MAL”, una cruz roja

La imagen de salida generada por nuestro sistema de validación es la siguiente:



Una vez obtenidos los resultados de manera gráfica, aprovechamos nuestro diccionario de estados para poder escribir nuestro archivo ‘.csv’. Para ello implementamos una función **‘escribir_csv()’** que toma como parámetro el diccionario antes mencionado, con los estados para cada campo de todos los formularios, y a partir de ellos crea (o sobreescribe) el archivo **‘estados_formularios.csv’**

Conclusión

La principal conclusión es que el procesamiento de imágenes no es un proceso lineal o con resultados obvios. Al contrario, es un proceso iterativo que requiere experimentación, diagnóstico de problemas y refinamiento constante para alcanzar un mejor resultado.

En el Problema 1, pudimos observar que la ecualización de histograma local logró resaltar detalles que no podríamos detectar con la ecualización global, pero a costa de introducir ruido. A partir de esto, y mediante lo hablado en clase, tomamos la decisión de realizar un paso de post-procesamiento al aplicarle un filtro de la mediana lo que logró eliminar el ruido causado por la ecualización. La elección del tamaño de ventana influye directamente en la capacidad del algoritmo para resaltar detalles ocultos, mostrando un equilibrio entre la mejora del contraste y la preservación del ruido.

En el Problema 2, nuestro primer acercamiento fue detectar todas las líneas, encontrar sus intersecciones y construir los vértices para luego construir las bounding box, este acercamiento es de cierta manera el más fácil de pensar pero a la hora de la implementación, nos encontramos con múltiples dificultades que nos llevaron a descartarlo. Decidimos que segmentar en filas y luego columnas, con los datos que ya habíamos obtenido, era un enfoque más sencillo, sacrificando un poco de automatización en el proceso ya que requiere conocimiento previo de la estructura del formulario para poder encontrar las celdas. Una vez las obtuvimos, observamos que teníamos líneas residuales resultantes del indexado, lo que nos llevó a pensar en hacer un recorte del margen de la imagen para eliminarlas. Luego tuvimos que realizar el diagnóstico de los puntos para poder detectarlos como componentes, y finalmente cuando obtuvimos los resultados esperados, nos dimos cuenta que se estaba realizando mal el recorte de márgenes ya que tenía un error en el slicing.

El proceso siempre fue iterativo, de diagnóstico y de mejora hasta obtener los mejores resultados posibles. Hay mucho margen de mejora, pero estamos conformes con el trabajo realizado.