Trabajo de Fin de Grado
Ingeniería de Tecnologías Industriales

Reconocimiento de Sellos en Legajos de Archivo Histórico

Autor: Miguel Ángel Rodrigo de Lisbona
Tutor: Daniel Limón Marruedo

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Trabajo de Fin de Grado
Ingeniería de Tecnologías Industriales

Reconocimiento de Sellos en Legajos de Archivo Histórico

Autor:
Miguel Ángel Rodrigo de Lisbona

Tutor:
Daniel Limón Marruedo
Profesor titular

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Proyecto Fin de Carrera: Reconocimiento de Sellos en Legajos de Archivo Histórico

Autor: Miguel Ángel Rodrigo de Lisbona

Tutor: Daniel Limón Marruedo

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mis maestros

Resumen

La Fundación Osborne posee un archivo de documentos relacionados con la empresa que datan desde principios de 1800. Previas investigaciones han permitido descubrir información sobre la vida de Tolkien en el archivo. Se sabe además, que existe correspondencia con casas reales o el Vaticano. También existe correspondencia de personajes públicos como Washington Irving. Los inicios de la empresa están estrechamente relacionados con otros personajes históricos como Fernán Caballero, cuñada del fundador, o "Frasquita" Larrea, su suegra. Por ello, podría haber información importante por descubrir sobre dichos personajes que podría ser de valor histórico.

Para extraer dicha información, se ha optado por desarrollar una web donde se suben los archivos escaneados junto con su transcripción. Además de ello, se quiere desarrollar un sistema que realice dichas transcripciones de manera autónoma mediante detección de los caracteres manuscritos.

En este trabajo se describe el desarrollo de un sistema que reconozca los escudos en las cartas para clasificar su autor de manera automática. Además, el sistema debe calcular la posición de dichos escudos con el objetivo de eliminarlos. Esto es necesario ya que los sistemas de detección de caracteres necesitan que los elementos que no son texto sean eliminados del documento antes de empezar cualquier análisis.

Abstract

Fundación Osborne owns an archive of documents related to the company dated up to early 1800. Previous investigations on the archive have revealed information about the life of Tolkien. Moreover, it is known that the archive has correspondence with royal houses as well as The Vatican. There are also letters of historical characters such as Fernán Caballero – founder's sister-in-law – or "Frasquita" Larrea – his mother-in-law. For those reasons, it is reasonable to think that there might be some historical value on the documents that remain not investigated.

In order to retrieve that information, there is a website in development where scanned documents will be uploaded with their respective transcription and metadata. Those transcriptions are currently being typed manually. A system to make those transcriptions automated is being developed too by recognizing handwritten characters on the documents.

The purpose of this work is to describe the development of a system which recognizes seals on letters to automatically classify them by author. Moreover, the system calculates the position of such seals to delete them from the document. This is a need because character recognition system needs every non-text element removed from the image before any analysis is performed.

Índice

Índice de Figuras

1 LA IMPORTANCIA DE PRESERVAR NUESTRA HISTORIA

Un pueblo sin conocimientos de su historia, origen y cultura es como un árbol sin raíces.

- Marcus Garvey -

La Historia. Alrededor de 6000 años de documentos escritos, miles de generaciones plasmando pensamientos y vivencias para aquellos que aún están por venir, haciendo posible el ciclo motor del progreso de todas las sociedades: intentar, fracasar, aprender y repetir. Parafraseando a Warren Buffet, si hay algo más importante que aprender de los errores de uno mismo, es aprender de los errores de los demás. Vivimos en un mundo en el que todos aparentan (o incluso creen) saber qué hacer para resolver los problemas. Pero cuando dos personas tienen opiniones incompatibles, al menos uno debe estar equivocado. La realidad es que ni políticos, ni científicos, ni guías espirituales de ningún tipo saben con absoluta certeza la mejor solución a todo. Es aquí donde la historia nos proporciona la más valiosa de las fuentes de conocimiento para permitir tanto a sociedades como a individuos tomar decisiones un poco menos a ciegas. Nadie duda de la importancia de la investigación, preservación y difusión de la historia además de garantizar la no distorsión de la misma. Por todo ello, la digitalización y catalogación de la totalidad de textos escritos por la humanidad a lo largo de su paso por este mundo tendría un incuestionable gran valor. Realizar esta tarea de manera manual es evidentemente inviable. Es por ello que existen multitud de líneas de investigación a lo largo y ancho del globo con el objetivo de conseguir automatizar esta tarea.

Este trabajo describe parte del proyecto llevado a cabo por Talemtum Lab Osborne mediante la colaboración de la Fundación Osborne[1], Telefónica Talemtum Labs[2] y la Fundación SEPI[3]. El objetivo del proyecto es la digitalización del archivo histórico que posee el Grupo Osborne almacenado en sus primeras bodegas en El Puerto de Santa María. Dicho archivo contiene, por un lado, facturas, notas de pago y demás documentación de carácter administrativo y, por otro, correspondencia entre miembros de la familia y con otros personajes públicos de la época de la fundación de la empresa en 1772. Entre estos personajes se encuentran Washington Irving, J.R.R. Tolkien o Isaac Peral, quien ofrece sus servicios a la empresa en una de sus cartas, así como miembros de distintas casas reales o El Vaticano.

El archivo ya ha sido utilizado en una reciente investigación sobre la biografía de Tolkien, muy relacionado con la familia Osborne al ser criado y educado por uno de sus miembros, y existen múltiples indicios de que podrían existir en él más documentos de importante valor histórico. La suegra de Thomas Osborne Mann, fundador de la empresa, fue Francisca Javiera Ruiz de Larrea y Aherán, **Frasquita Larrea**. Esta señora fue una escrtora gaditana impulsora de la tertulia literaria que se cree que tuvo una gran influencia en la Constitución de 1812. La cuñada de Thomas fue la escritora Cecilia Böhl de Faber y Larrea, **Fernán Caballero**.

Gran parte de la documentación que se posee tiene un contenido desconocido, de ahí nace la motivación por

[1] Fundación Osborne es el vehículo canalizador de la Responsabilidad Social Corporativa del Grupo Osborne. Posee dos fines fundamentales: la preservación y difusión del patrimonio histórico de la compañía y la promoción de la formación y el emprendimiento en las áreas de influencia del Grupo Osborne [1].

[2] La Fundación Telefónica tiene por misión mejorar las oportunidades de desarrollo de las personas a través de proyectos educativos, sociales y culturales, adaptados a los retos del mundo digital [2].

[3] La Fundación SEPI, entre otras actividades, promueve y gestiona un extenso Programa de Becas de Inserción Laboral que cada año facilita la formación práctica en las empresas a más de 300 jóvenes titulados [3].

digitalizarlo para permitir agilizar su investigación. Para ello, se desarrolla una página web donde se muestran los documentos escaneados junto con su transcripción. Esta transcripción es necesaria por un doble motivo: por un lado, los documentos tienen en su mayoría una caligrafía de muy difícil reconocimiento lo cual ralentiza enormemente la lectura e interpretación. Por otro lado, si se quiere realizar algún tipo de búsqueda en función de algún término que forme parte del texto, sería imposible sin transcripción previa.

2 DESCRIPCIÓN DEL PROBLEMA

Somos como enanos a los hombros de gigantes. Podemos ver más, y más lejos que ellos, no porque la agudeza de nuestra vista ni por la altura de nuestro cuerpo, sino porque somos levantados por su gran altura.

- Bernard de Chartres -

Este trabajo forma parte de un proyecto mayor que pretende extraer información de los documentos del archivo descrito mediante técnicas de visión artificial. El problema al que se dedica esta memoria consiste en localizar la posición de posibles escudos o sellos en correspondencia con un doble objetivo:

- Localizar la posición de dicho sello con el objetivo de eliminarlo de la imagen y que no suponga un problema para otros sistemas de reconocimiento, como por ejemplo la transcripción automatizada del texto.
- Reconocer de qué sello se trata y clasificar el documento en cuestión de acuerdo con la información disponible de dicho sello.

Algunos ejemplos de documentos se muestran a continuación.

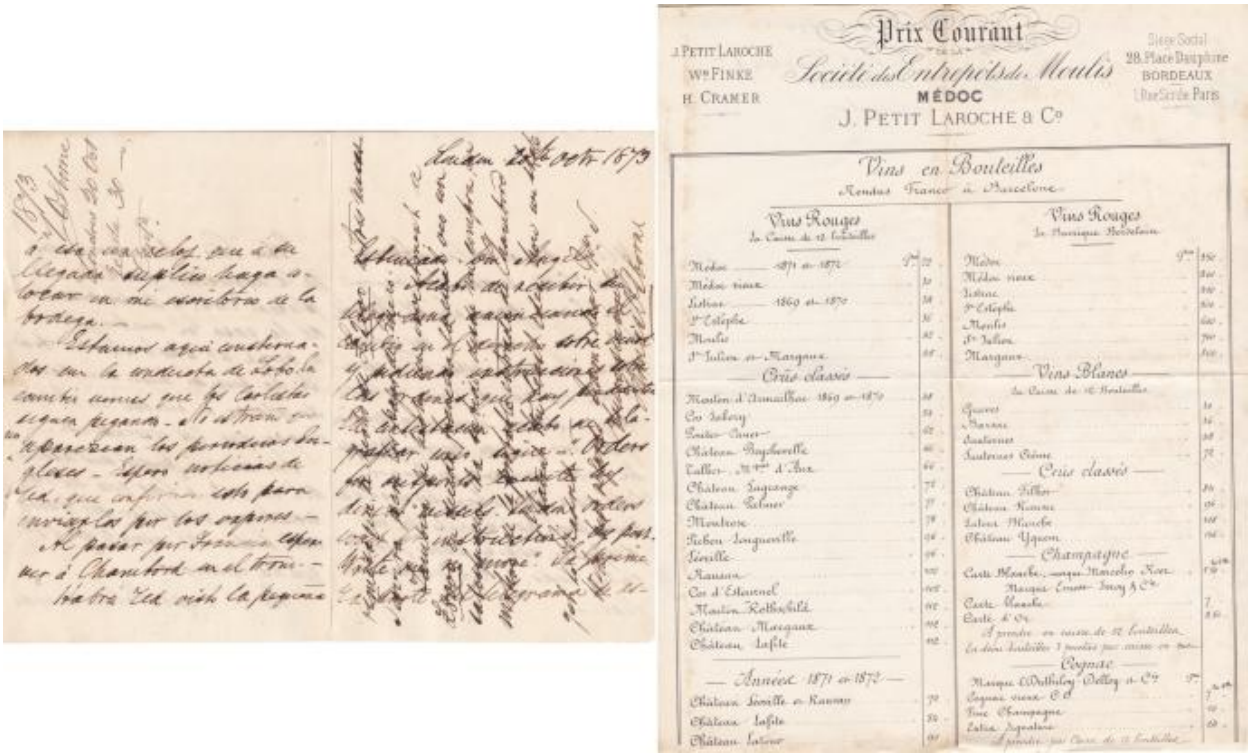


Figura 2.1. Ejemplo de tipos de documentos que pueden presentarse

En la figura anterior pueden apreciarse dos tipos de documentos distintos. Uno presenta texto en vertical además del texto horizontal. Esto es una gran dificultad añadida como se verá más adelante, ya que el texto en diferentes direcciones que intersecta se detecta como un único objeto en la imagen.

Figura 2.2. Dos ejemplos más de documentos

Puede verse en el documento de la izquierda como existen varias manchas además de las marcas de pliegues oscurecidas por el envejecimiento. Todos estos elementos dificultan la detección. En concreto, la mancha roja es particularmente difícil de diferenciar de un sello.

El caso de la derecha representa la situación ideal, donde el documento presenta un sello claramente separado del texto, mientras que el resto del papel apenas presenta envejecimiento o imperfección alguna.

El problema del reconocimiento automático de elementos en documentos escritos, ya sean caracteres, firmas, logos o cualquier otro elemento, es un tema de gran interés, y por lo tanto, existe una extensa bibliografía con muy diversos métodos para atajar el problema. En [4] se estudia cómo extraer y reconocer sellos estampados con el objetivo de clasificar documentos. En un primer paso se realiza una segmentación mediante una Transformada de Hough para círculos para hallar regiones de sellos. Después se detectan los caracteres aislados hallados en las regiones se aplica un sistema de clasificación basado en características invariantes a la rotación y la escala. Después se utiliza la información de la posición relativa de cada pareja de caracteres para clasificar el sello. En nuestro caso, los sellos carecen de texto alguno, por lo tanto no pueden clasificarse así. Sin embargo, el método de segmentación mediante TdH sí que podría resultar de interés.

En [5], se estudia un método para la segmentación y clasificación de firmas. El algoritmo propuesto extrae el contorno de las regiones de las firmas y utiliza información de la posición relativa de los puntos de ese contorno para la clasificación. En nuestro caso, el deterioro del papel con el paso de los años, además de que los sellos tienen forma mucho más compleja que tan solo trazos hacen que la detección de contornos no siempre devuelva el mismo resultado para el mismo sello.

Nandedkar, Mukhopadhyay y Sural [6] aplican un método mucho más sencillo en el cual utilizan una convolución gaussiana a la imagen como filtro de paso bajo para separar texto de logotipos. Esto se basa en el hecho de que el texto añade componentes de alta frecuencia al espectro de una imagen, mientras que un logotipo tan solo posee componentes de baja frecuencia. Como se verá más adelante, este método tan sólo funciona para sellos con grandes regiones homogéneas, mientras que falla para aquellos con trazos más delgados.

Antes de desarrollar ningún método, necesitamos alguna forma de medir cómo de efectivo resulta. Para medir la calidad de los resultados obtenidos con cada experimento se ha construido un *ground truth*.

Un detalle muy importante es que **el conjunto total de documentos se considera cerrado**. Es decir, se supone que la totalidad de documentos del archivo ha sido escaneado y que no van a descubrirse documentos nuevos. Esto es de gran relevancia en el algoritmo final desarrollado. En caso de que se contemplase la posibilidad de continuar añadiendo nuevos documentos, deberían agregarse pasos adicionales (además de triviales) al algoritmo que definieran cómo clasificar los nuevos documentos que se incorporasen a la colección. Esto no significa en cambio que este algoritmo sólo sea funcional para este conjunto cerrado de documentos, sino que se diseña con el objetivo de que sea útil para cualquier conjunto cerrado de documentos manuscritos, siempre y cuando presenten una estructura suficientemente similar. No se considera que deba

funcionar por ejemplo para imágenes que presenten más de dos páginas de un documento.

Las tecnologías que se han utilizado son:

- Python y en menor medida C++ para codificar los algoritmos.
- OpenCV, librería de visión artificial de código abierto por excelencia.
- Pygame junto con Tkinter para desarrollar una aplicación que nos permita agilizar el proceso de crear un ground truth.
- MySQL para todas las necesidades de base de datos del proyecto, tanto almacenar resultados, como la información del ground truth o la ruta de las imágenes de cada uno de los sellos.

3 CONSTRUCCIÓN DE UN GROUND TRUTH

Saber qué conoces y qué no conoces. Ese es el verdadero conocimiento.

- Confucio -

En *machine learning* se denomina ground truth a la colección de datos supervisados que se realiza con el objetivo de someter a prueba una hipótesis. Este ground truth consiste en una base de datos que relaciona cada documento con el sello que contiene, si contuviese alguno. En tal caso, se almacenan además las coordenadas de dicho sello en la imagen.

Para llevar a cabo este procedimiento se ha desarrollado un software que permite agilizar la tarea. En él puede visualizarse el documento en cuestión, acercar o alejar el zoom además de desplazar la imagen. Una vez localizado el sello, se seleccionan dos esquinas opuestas de este, eligiendo cada una con el click de cada uno de los dos botones del ratón. Esta interfaz se ha creado utilizando la librería Pygame, que permite dibujar gráficos sencillos en una ventana mediante un script de Python.

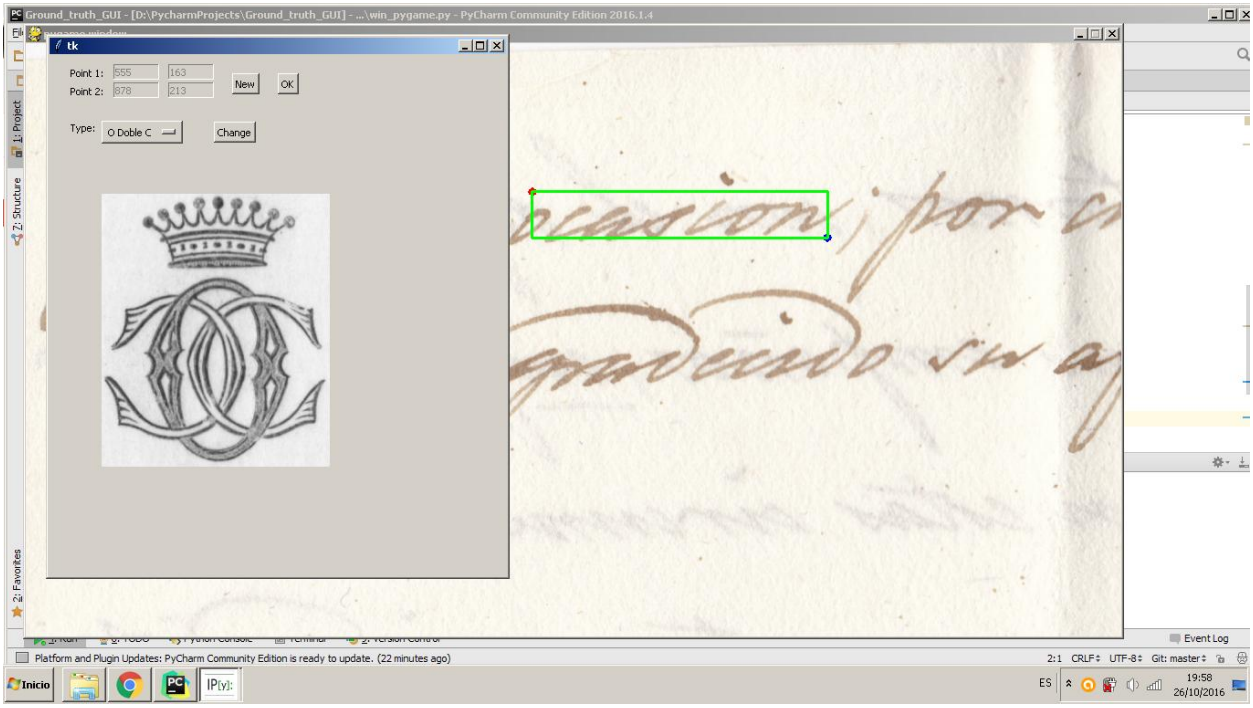


Figura 3.1. Software para la elaboración de un ground truth

Una vez que el sello ha sido encuadrado, se utiliza el panel de control que se muestra en la imagen para seleccionar cuál es el tipo de sello que se ha encuadrado. Para esto, el panel de control se comunica con una base de datos MySQL donde se almacenan tanto la ruta de la imagen como un nombre arbitrario que se ha escogido para describir cada sello. El usuario puede seleccionar el sello deseado de la lista desplegable y pulsar "Change" para que cambie la vista en el panel de control a una imagen del sello que se ha seleccionado y así asegurarse de que ha elegido el nombre de sello correcto. Puede darse el caso de que el sello encontrado en el documento sea la primera vez que se locacaliza y por tanto no exista ningún registro de él en la base de datos. Si esto ocurriera, se utiliza el botón "New" y se abrería una ventana emergente donde se le da un nombre al

sello y se le añaden de manera opcional metadatos como el autor al que corresponde el sello o una fecha aproximada a la que pertenece. Además, se copia en la carpeta de imágenes de sello un recorte de la imagen con las coordenadas el rectángulo que se haya seleccionado, que debería estar alrededor del sello que se desea almacenar. Este nuevo sello se añade al menú desplegable de sellos.

Si el usuario pulsase "Ok", se añadiría una nueva fila a otra tabla de la base de datos. Esta nueva tabla contiene la ruta a cada imagen de cada documento junto con el nombre del sello que contiene (o "no_seal" en caso de que no tenga ninguno), además de las coordenadas del rectángulo seleccionado. Al hacer esto, el programa abre automáticamente el siguiente documento salvo que ya no exista ninguno más. Un detalle importante es que el programa almacena en un archivo de texto cuántos documentos se han clasificado ya. De este modo, si se interrumpe el proceso sin haber terminado no es necesario comenzar de nuevo.

El software no posee la interfaz más atractiva ni atiende a bonitos patrones de diseño, pero cumple con toda la funcionalidad que se precisa para su tarea. El resultado final es una tabla en la base de datos con los 155 documentos clasificados más otra que registra cada uno de los distintos tipos de sellos que aparecen en los documentos.

El código del programa desarrollado puede consultarse en el **anexo A**.

4 ESQUEMA GENERAL DE UN OCR

Conoce bien las reglas, para que puedas romperlas de manera efectiva.

- Dalai Lama XIV -

En este capítulo se explicará cuáles son los pasos que suelen incorporarse en un algoritmo destinado a realizar un OCR. Después se explicarán varias maneras mediante las cuales se ha intentado abordar cada uno de esos pasos en este trabajo y por qué se han descartado o escogido cada uno de esos métodos.

El proceso de reconocer el contenido de un documento escrito de manera automática se denomina *reconocimiento óptico de caracteres* u *OCR*. Debe recordarse que el problema que se aborda en este trabajo no es estrictamente un OCR, ya que no se pretende obtener el contenido del texto escrito. Sin embargo existen similitudes evidentes entre este tipo de problema y el que aquí se presenta. Nuestro objetivo se centrará en intentar descartar todo el texto posible, además de manchas y otros elementos para quedarnos solamente con píxeles de sello. De manera general todos los métodos de OCR tienden a seguir un esquema, aunque con frecuencia suelen omitirse alguno o varios de sus pasos en función del problema concreto y la solución que se ofrezca.

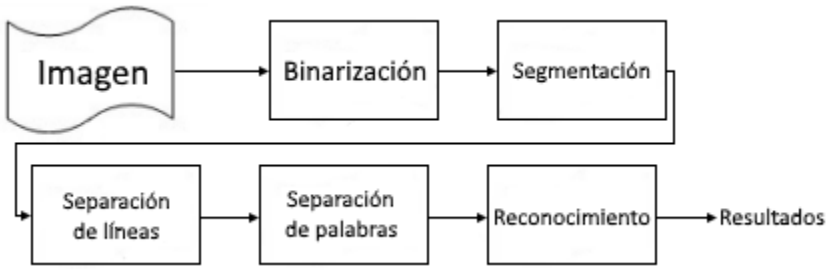


Figura 4.1. Esquema general de un OCR

4.1 Binarización

En un primer paso se busca obtener a partir de una imagen en color una matriz binaria. Este paso no busca eliminar texto aún, sino simplemente una matriz de dimensiones iguales a las que tendría la imagen en escala de grises con valor falso booleano en lo que corresponda al papel y verdadero en donde hubiese algún elemento más oscuro, ya sea texto, sello o cualquier otro elemento gráfico del documento. Sí que se encarga este paso del proceso de filtrar aquellos elementos indeseables que erróneamente se cuelen como información escrita del documento, como tinta que se transparente del otro lado del papel, suciedad, marcas de envejecimiento o pliegues…etc. Primero se discutirán los métodos planteados en este trabajo para la binarización propiamente dicha. A continuación, el filtrado complementario al proceso de binarización que eliminaría el ruido mencionado.

4.1.1 Elección manual de umbral fijo

Consiste en elegir un umbral de intensidad de gris a partir del cual se considera si un píxel corresponde a tinta o papel. Este umbral se elegirá de manera experimental probando hasta encontrar un número que parezca

acertado. Para encontrarlo, se crea un script que permite modificar este umbral además de visualizar los resultados en tiempo real. Para mejorar aún más el método, se aplica un filtrado gaussiano con el objetivo de eliminar pequeños ruidos puntuales. Los parámetros de este filtro también son modificables en la herramienta. Tras múltiples ensayos puede verse que un umbral fijo no es un parámetro suficientemente robusto para todos los documentos. El código de esta herramienta se encuentra adjunto en el **anexo B.**

4.1.2 Método de Otsu

Este método recibe nombre en honor Noboyuki Otsu, quien creó el método en 1979 [7]. Su objetivo es automatizar la tarea de encontrar el valor umbral para imágenes en escalas de grises de manera que todo píxel con una intensidad superior a dicho valor se considera objeto de interés, y todo aquel valor inferior, entorno (o, en nuestro caso, justo al contrario). Para hallar este valor umbral el método se vale del histograma de la imagen para hallar la varianza en las frecuencias de las intensidades que pertenecen a cada uno de los dos segmentos resultantes de cada umbral seleccionado. Aquel umbral que minimice esta varianza a la vez que maximice la varianza entre frecuencias de segmentos distintos será el umbral óptimo.

Descripción matemática

Partamos de dos segmentos de puntos $K_0(t)$ y $K_1(t)$ obtenidos a partir del umbral **t**. Sea $p(g)$ la probabilidad de ocurrencia de del valor de gris $0 < g < G$ (con G=255 en nuestra imagen gris de 8 bits).

Supongamos la siguiente imagen en escala de grises. Se ha creado una imagen de pequeñas dimensiones para así obtener un histograma sencillo.

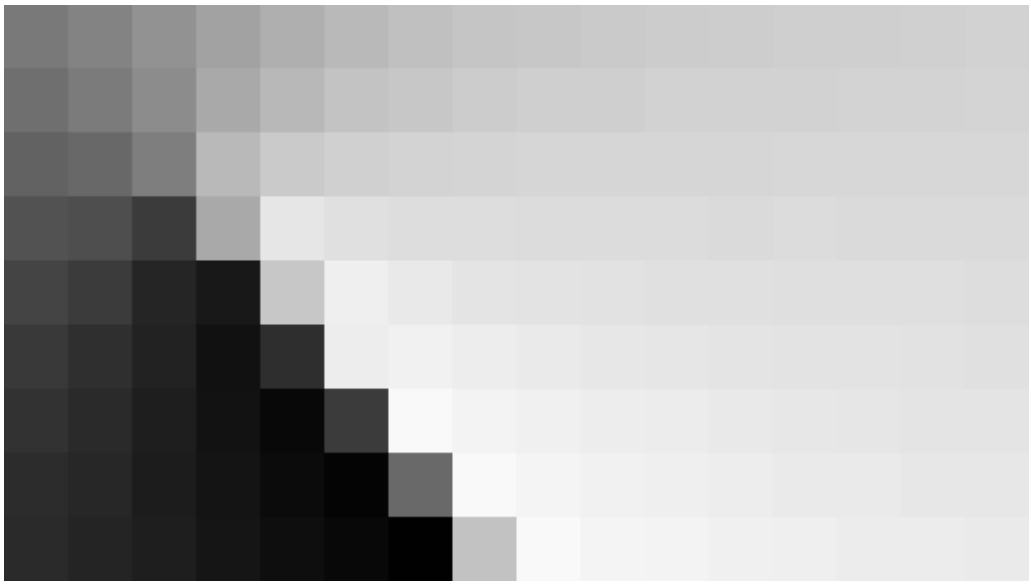


Figura 4.2. Imagen en escala de grises para aplicar Otsu. Dimensiones: 16x9px.

Para esta imagen se ha obtenido el histograma siguiente.

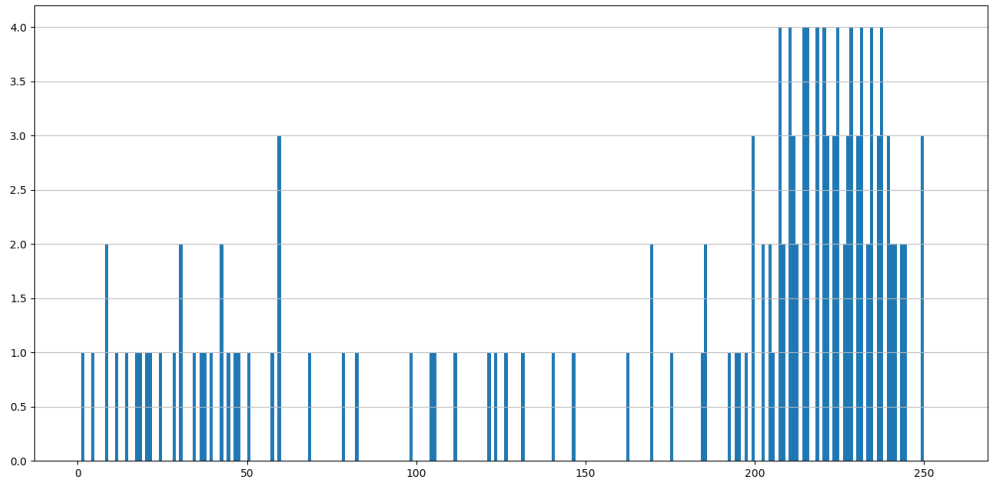


Figura 4.3. Histograma de la imagen en escala de grises.

En el cual, los elementos definidos serían:

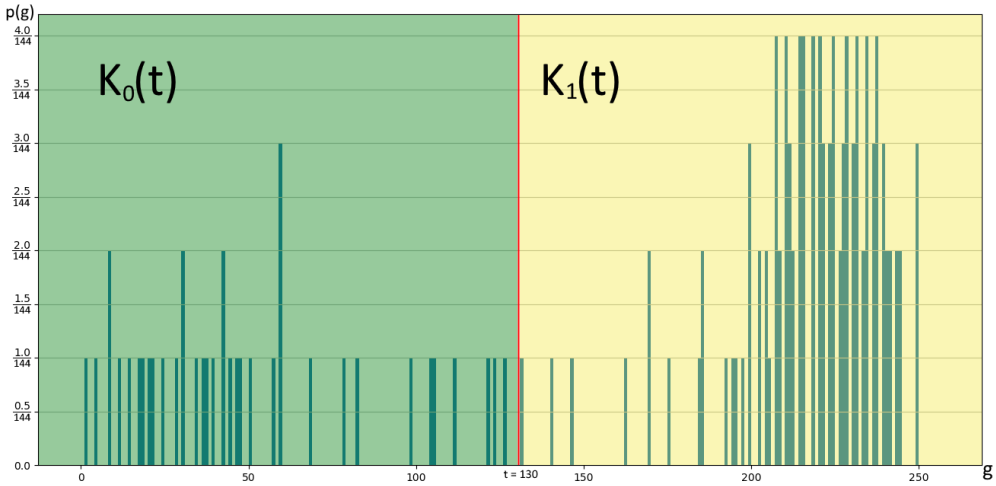


Figura 4.4. Descripción gráfica de elementos definidos para la explicación del algoritmo de Otsu.

Nótese que, para ser rigurosos dividimos las frecuencias de cada intensidad de gris por el área total de la imagen (144px) para así obtener probabilidad. El cálculo con frecuencias es equivalente, sólo se ha realizado la conversión por una cuestión de rigor matemático.

Entonces la probabilidad de que cada píxel pertenezca a uno u otro segmento sería:

$$K_0(t): P_0(t) = \sum_{g=0}^{t} p(g)$$

$$K_1(t): P_1(t) = \sum_{g=t+1}^{G} p(g) = 1 - P_0(t)$$

Por tanto, si definimos $\bar{g}$ como la media aritmética entre las intensidades de la imagen completa y $\bar{g}_0$ y $\bar{g}_1$ como las medias de intensidades dentro de cada segmento, entonces las varianzas entre intensidades dentro de cada segmento pueden calcularse como:

$$\sigma_0^2(t) = \sum_{g=0}^{t} (g - \overline{g_0})^2 p(g)$$

$$\sigma_1^2(t) = \sum_{g=t+1}^{G} (g - \overline{g_1})^2 p(g)$$

El objetivo es minimizar la varianza dentro de cada segmento y a su vez maximizar la varianza entre distintos segmentos. Para ello creamos una variable que englobe el cociente entre ambas medidas y se optimiza dicha variable:

$$Q(t) = \frac{\sigma_{01}^2(t)}{\sigma_{in}^2(t)}$$

Donde $\sigma_{01}^2(t)$ es la varianza entre ambos segmentos y se define:

$$\sigma_{01}^2(t) = P_0(t) \cdot (\overline{g_0} - \bar{g})^2 + P_1(t) \cdot (\overline{g_1} - \bar{g})^2,$$

y $\sigma_{in}^2(t)$, la varianza dentro de los segmentos se define por:

$$\sigma_{in}^2(t) = P_0(t) \cdot {\sigma_0}^2(t) + P_1(t) \cdot {\sigma_1}^2(t)$$

La implementación más simple prueba todos los valores de ***t*** y escoge aquel para el cual ***Q(t)*** es máximo. Sin embargo esta implementación es muy lenta. Las implementaciones más utilizadas subdividen el rango de posibles valores de ***t*** de manera que es posible hallar el óptimo con tan solo unos cuantos intentos.

Resultados obtenidos

A menudo suele destacarse el tiempo de ejecución como desventaja de este método pero en la práctica no se han notado diferencias destacables entre usarlo o elegir un umbral de manera manual. En cuanto a la calidad de la imagen binarizada obtenida, los resultados son bastante buenos en la mayoría de los casos. Como cabe esperar, en los casos en los que apenas hay nada escrito en el papel y este presenta un evidente envejecimiento, el método cataloga este ruido como falso positivo. Es por esto que, aunque este método aporta generalmente resultados bastante buenos, no es suficiente en sí mismo para segmentar tinta y papel cuando el ratio cantidad de tinta frente a cantidad de ruido es demasiado bajo.

4.1.3 Método adaptativo

Este método pretende aplicar un umbral distinto a cada píxel de la imagen en función de la intensidad que tengan sus píxeles vecinos. De este modo, en imágenes iluminadas de manera heterogénea, el umbral es capaz de adaptarse a estos cambios de iluminación sin que las zonas muy oscuras contaminen a las claras con un umbral demasiado restrictivo y viceversa.

Para nosotros esto puede ser interesante ya que los defectos localizados en el papel son un problema frecuente. En nuestras pruebas el umbral se escoge igual a la media ponderada con un kernel gaussiano de los píxeles vecinos al píxel a umbralizar en cuestión.

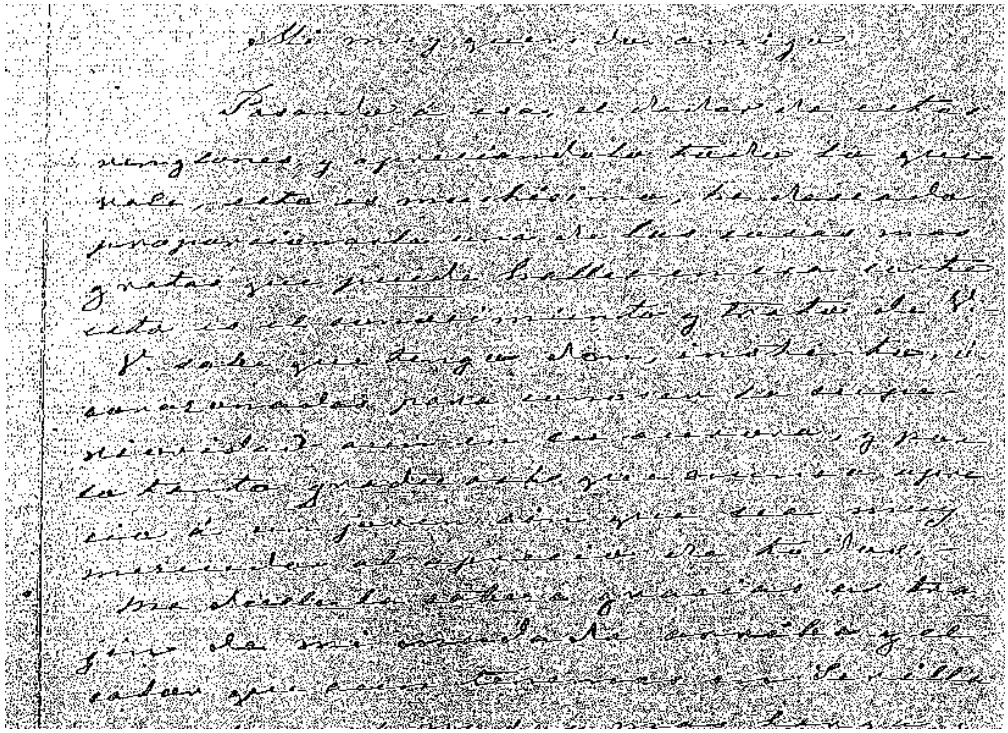


Figura 4.5. Resultado de umbral adaptativo.

Nótese como los detalles a menor escala de los caracteres aparecen mejor definidos que en los otros métodos.

Figura 4.6. Detalle de trazos con un umbral adaptativo con respecto a un umbral de Otsu.

Se puede ver que este método devuelve demasiados falsos positivos en regiones donde no hay tinta, tan sólo papel ya que el umbral adaptativo en dichas regiones es demasiado generoso al no encontrar nada suficientemente oscuro.

Debido a que la dificultad añadida de eliminar los falsos positivos del método adaptativo supone un precio a pagar muy superior a la mejor definición obtenida en los trazos de tinta, se emplea el método de Otsu para solventar el problema de la binarización.

En [8] se comparan múltiples métodos de binarización y se destaca el hecho conocido de que para cada tipo de imagen suele ser conveniente un algoritmo de binarización distinto. Además, se propone un método para seleccionar el método más oportuno de manera automática, mediante un enfoque de machine learning. En nuestro caso, las imágenes tienen unas características relativamente homogéneas en cuanto a ruido de fondo, iluminación y contraste. Por tanto, tras probar el método de Otsu en varios casos y ver que funciona de manera correcta no se considera necesario recurrir a métodos más complejos.

4.2 Segmentación

Este paso tiene la finalidad de separar los distintos elementos gráficos del documento como pueden ser firmas, destinatarios, fechas, el contenido principal de la carta, etc. Para entender la importancia de este paso podemos imaginar qué ocurriría si se intentase reconocer caracteres en un logotipo o clasificar un sello a partir de una palabra. Es evidente que cualquier información que el sistema devolviese en estos casos no sólo sería errónea sino que podría empeorar los resultados arrojados a partir de otras porciones vecinas del documento. Por ejemplo, es frecuente en los sistemas más avanzados realimentar una detección de texto de manera que se corrija la estimación de una palabra en función de cuáles sean las palabras cercanas que se detecten. Es fácil entender que una mala segmentación haría que esta realimentación empeorase los resultados obtenidos. En nuestro caso, debemos separar en la mayoría de los casos tan solo sello de texto. Para resolver esta parte del proceso se han probado varios métodos.

4.2.1 Segmentación mediante enfoque frecuencial

Como ya se mencionó, en [6] se desarrolla un método para separar texto de elementos gráficos en documentos escaneados. La idea se basa en el hecho de que el texto introduce componentes armónicas de alta frecuencia a la imagen. Por lo tanto, un filtro de paso bajo eliminaría ese texto. En el algoritmo, se utiliza un filtro de Gauss para eliminar las componentes de alta frecuencia. El resultado es una imagen con los elementos de texto eliminados casi en su totalidad mientras que los elementos gráficos (en el artículo incluye también logos) permanecerían en la imagen, idealmente, en su totalidad. El algoritmo expuesto por Nandedkar, Mukhopadhyay y Sural es el siguiente:

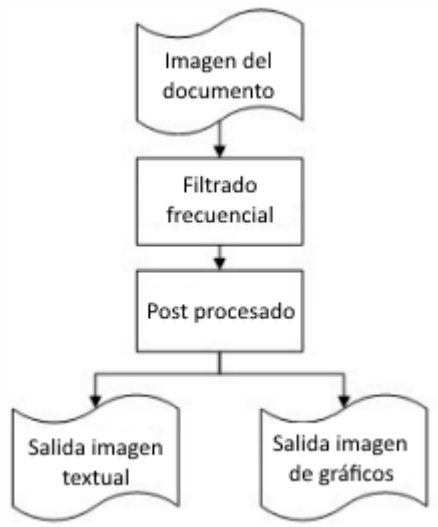


Figura 4.7. Algoritmo espectral de seccionamiento

El algoritmo original presenta algunos pasos adicionales para separar distintos tipos de elementos gráficos, pero para nuestro problema, son pasos innecesarios.

Filtrado frecuencial

El filtrado frecuencial es el paso más importante del método, y es donde se elimina la mayor parte del texto. Es una evidencia fácilmente comprobable que el texto contribuye a componentes espectrales de muy alta frecuencia en imágenes de documentos. Por ello, se convoluciona con un kernel gausiano como filtrado de paso bajo y se utiliza un post procesado para eliminar restos residuales de texto. El algoritmo puede verse en la figura X.

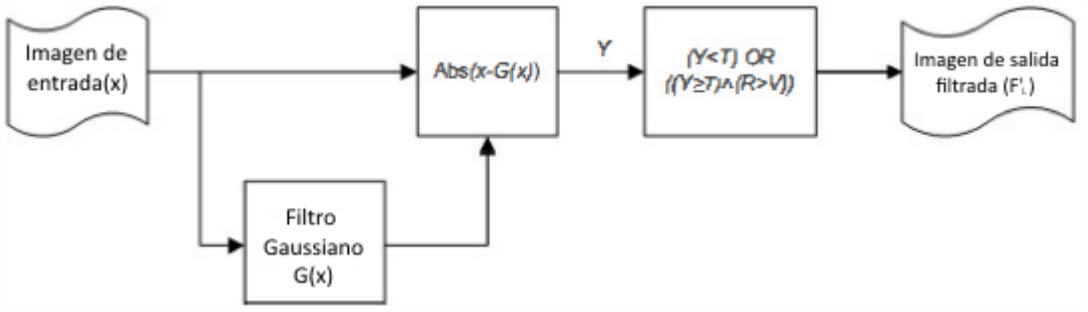


Figura 4.8. Filtrado frecuencial

El proceso consiste en:

1. Se toma la imagen original convertida a escala de grises, X.
2. Se le aplica un filtrado gaussiano, G(X).

3. Se calcula la imagen la imagen de **paso alto** Y, restándole G(X) a la original.

$$Y = |X - G(X)|$$

4. Un filtrado selectivo en frecuencias sencillo sería entonces

$$F_L = \begin{cases} I(i,j), & \text{si } Y(i,j) < T \\ B_k, & \text{En caso contrario} \end{cases}$$

Donde I es la imagen original de dimensiones NxM, $1 \leq i \leq N$ y $1 \leq j \leq M$. T es el umbral de frecuencias. Los valores que superan un cierto valor en la imagen filtrada Y se consideran de alta frecuencia, los que no, de baja frecuencia. Por último, B_k es el valor del color considerado de fondo. En el artículo utilizan una media de los colores de fondo del papel, obtenidos sumando los píxeles de la imagen original I, que han quedado como fondo en una binarización de la imagen en escala de grises X. En este trabajo se ha considerado innecesario ya que no se pretende reconstruir los huecos dejados por los elementos de eliminados del documento, y se ha tomado B_k como blanco directamente.

Hay que destacar que un filtrado solamente en frecuencias no es apto para elementos cromáticos cuyos tonos cambien de manera abrupta, ya que también introducen componentes de alta frecuencia y dicho filtro los eliminaría. Para solventar este problema, se conservan también aquellos elementos que tengan una cantidad de color suficientemente alta, esto se consigue cuantificar mediante la *cromaticidad*.

Definición cromaticidad: Se entiende por cromaticidad de un tono de color a la medida de cómo de alejado se encuentre ese color de tonos grises, independientemente de su luminancia, es decir, de cómo de claro u oscuro sea.

En un espacio de colores YC_rC_b, la cromaticidad R se cuantifica como

$$R = \sqrt{{C_r}^2 + {C_b}^2}$$

Por tanto, dada nuestra imagen original, se define la matriz de cromaticidad aquella que contiene para cada píxel el valor de su cromaticidad.

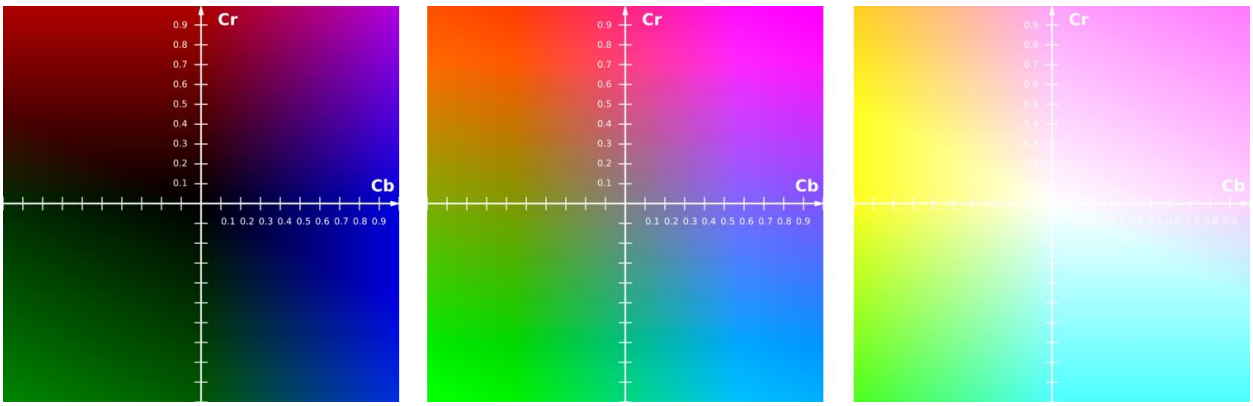


Figura 4.9. Espacios de color YC_rC_b para valores de luma (Y) de 0, 0.5 y 1 respectivamente.

En la figura anterior puede verse como aquellos colores con una menor R, es decir, aquellos más cercanos al origen de coordenadas, poseen tonos más grises.

Volviendo al problema anterior, queremos evitar que al filtrar componentes de alta frecuencia eliminemos elementos gráficos que posean cambios bruscos en los colores. Para ello, imponemos un umbral V. Aquellos píxeles cuya cromaticidad sea superior a dicho umbral serán conservados en la imagen filtrada. Esto no supone ningún problema para la eliminación de texto ya que los caracteres tienen generalmente una cromaticidad muy baja.

Los resultados obtenidos mediante este procedimiento son los siguientes.

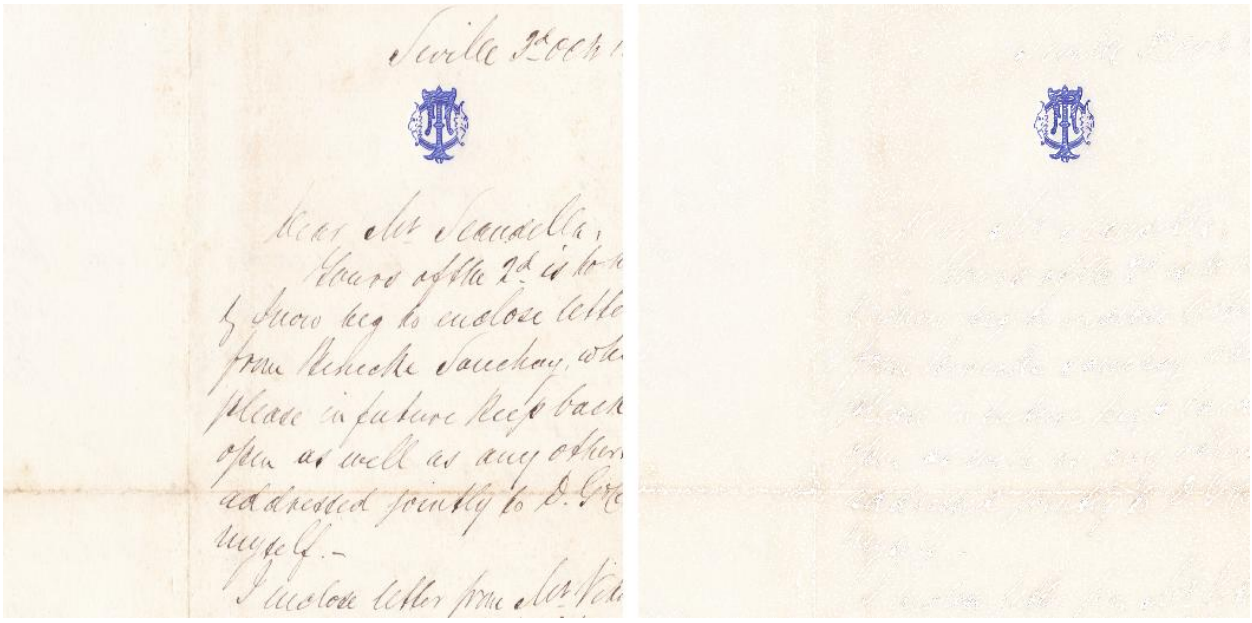


Figura 4.10. Ejemplo de correcto funcionamiento del filtrado frecuencial.

Figura 4.11. Ejemplo en el cual el filtrado frecuencial falla.

Como puede verse, los resultados en el primer caso son verdaderamente alentadores. Sin embargo, en el segundo caso el filtrado frecuencial elimina buena parte del sello además del texto. Esto se debe a que el sello está compuesto de trazos finos y una zona más gruesa pero que en realidad son franjas estrechas con diferencias tonales entre colores de muy baja cromaticidad. Este método es muy bueno cuando los gráficos a separar del texto poseen regiones amplias de colores uniformes, como logotipos, y que, si poseen algún trazo estrecho, éste posee una alta cromaticidad o no supone una porción importante del elemento en cuestión. Para nuestro caso por tanto, no supone una solución robusta.

Por todo eso, se procederá a probar otro método.

4.2.2 Extracción y descripción de características

Para describir en qué consiste este enfoque, trate el lector de localizar la posición de cada uno de los fragmentos en la imagen inferior.

De manera inconsciente habrá tratado de buscar los elementos que pudieran ser más fácilmente reconocibles en cada uno de los fragmentos, como bordes o esquinas, para luego intentar localizar esos elementos en la imagen completa. Puede observarse que, generalmente, el primer fragmento resulta más difícil de localizar, mientras que el último es el más sencillo. La facilidad o dificultad de localizar cada fragmento de manera correcta depende de lo diferente que sea de sus alrededores cercanos el elemento que inconscientemente usamos para comparar. Típicamente, las esquinas de objetos ofrecen la mayor facilidad. Este elemento que inadvertidamente seleccionamos como pivote se denomina, en el ámbito de la visión artificial, *característica*.

Característica: Porción de la imagen que posee alguna propiedad que la hace muy distinta al resto de imagen que la rodea, típicamente alguna medida relacionada con el gradiente.

Realmente no existe una definición global que pueda ser concreta sobre qué es una característica en visión artificial, ya que depende del algoritmo en concreto del que se hable. Pueden ser bordes, esquinas, blobs (o regiones de interés) o lo que se conoce como *ridges*, en español crestas. Estas últimas son regiones alargadas. Por ejemplo, una imagen de un avión visto desde la planta está compuesta por dos *ridges* que se cruzan entre sí. Es importante destacar que no se ha incluido en la definición ninguna referencia a la **dimensión de una característica con respecto a la dimensión de la imagen.** Esto es así ya que en muchos algoritmos (SIFT, SURF, BRIEF…) se busca definir una característica como invariante a la escala, es decir, sigue siendo una característica independientemente del tamaño que tenga dentro de la imagen. Esto no es cierto en la totalidad de algoritmos, como es el caso del detector de esquinas Harris.

Existen multitud de algoritmos para esta tarea, pero en esta sección nos limitaremos a emplear uno de ellos: SURF. Nuestro objetivo de momento es simplemente ver si merece la pena invertir en este enfoque o, por el contrario, sería mejor descartarlo y buscar una nueva solución a nuestro problema.

Para nuestro experimento vamos a tomar dos imágenes de dos sellos extraídas de la base de datos de documentos. Después se comparará un documento con ambos sellos. Uno de ellos es el mismo que aparece en el documento, el otro no.

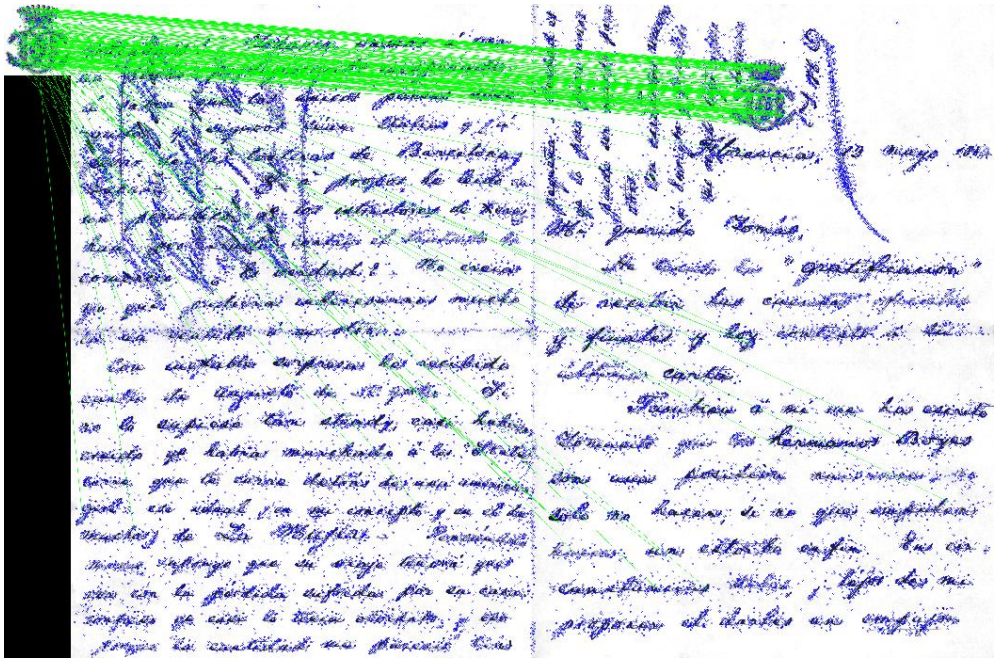


Figura 4.12. Comparación de características entre el documento y su mismo sello.

Figura 4.13. Comparación de características entre el documento y un sello distinto.

El sistema calcula primero qué puntos de la imagen son mucho más distintos del resto (esquinas de objetos en este caso) y se han dibujado en azul. Después compara los puntos de la imagen del sello con los del documento y empareja aquellos que determina que más se parecen. Estos emparejamientos se han dibujado en verde. Más adelante se explicará en detalle todo el proceso.

Este primer experimento parece indicar que este sistema es bastante bueno para resolver el problema. Hay dos consideraciones importantes que hacer antes de continuar.

1. Este sistema no solo puede ser útil para segmentar estimando la posición del sello, sino que además se puede emplear en la fase de reconocimiento, para saber qué sello es.
2. Aunque los resultados parecen muy positivos no hay que olvidar que para el correcto funcionamiento de este sistema es necesario estar en posesión de una imagen de cada sello. Este no es un requisito absolutamente utópico, pero preferiría salvarse si fuera posible. Esto se debe a que en una aplicación real no es posible disponer de esta información ya que, si así fuera, significaría que la totalidad de

documentos han sido clasificados y por lo tanto este proceso carecería de utilidad.

4.2.3 Segmentación mediante componentes conexas

En visión artificial, la búsqueda de componentes conexas se refiere a la familia de algoritmos que pretende localizar grupos de píxeles que cumplan una determinada condición y que se encuentren en contacto directo. La idea es agrupar los píxeles oscuros que aparezcan conexos en la imagen e identificar qué propiedades poseen los que pertenecen a un sello con respecto a los que no.

OpenCV ofrece un detector de blobs que implementa componentes conexas utilizando como condición múltiples binarizados a distinto umbral y agrupando aquellos píxeles conexos en cada umbral. Este sistema es muy útil cuando existen objetos muy distintos en forma y/o color y que poseen mucho contraste con respecto del fondo. En nuestro caso, los objetos que queremos detectar son sellos y palabras. No están claramente diferenciados y, en muchos casos, el contraste con el fondo es escaso. Por todo esto el bajo control que ofrece este algoritmo no es suficiente.

Por ello se ha empleado el detector de componentes conexas sencillo sin funcionalidad adicional. Este algoritmo toma una imagen binarizada y devuelve una imagen de etiquetas. Esto consiste en que cada píxel se sustituye por una etiqueta de manera que aquellos píxeles que se encuentren conexos posean todos la misma etiqueta, única para cada conjunto conexo. Una etiqueta no es más que un valor entero incremental a partir de 0, valor que se reserva para los píxeles del fondo.

Una vez que sabemos qué píxeles pertenecen al mismo objeto de la imagen binaria se pueden hacer todo tipo de operaciones. Cabe destacar que, una vez más, aún no se pretende desarrollar un sistema perfectamente funcional y capaz de resolver el problema, sino un prototipo rápido del algoritmo que nos permita juzgar si merece la pena seguir desarrollándolo o si en cambio debe ser descartado y sustituido.

En esta primera prueba se ha calculado el área de cada región conexa como el número de píxeles que se incluyen dentro de ésta, y se ha descartado como posible candidato a sello aquellas regiones demasiado pequeñas. Además, es frecuente que palabras de diferentes líneas en las que aparecen caracteres con pronunciada verticalidad como podrían ser una "p" y una "t" se unifiquen en una misma región al aparecer solapados dichos caracteres. Para eliminar este común error, se ha calculado el *bounding box* o cuadro delimitador de cada región y se ha calculado el área de éste cuadro como el área geométrica del rectángulo que lo compone. A continuación se ha obtenido el ratio del área de la región (es decir, el conteo de puntos que forman parte de esta región) con respecto al área de su cuadro. A este ratio se le ha denominado **ratio de relleno**. Si el ratio de relleno es demasiado pequeño, la región es descartada también.

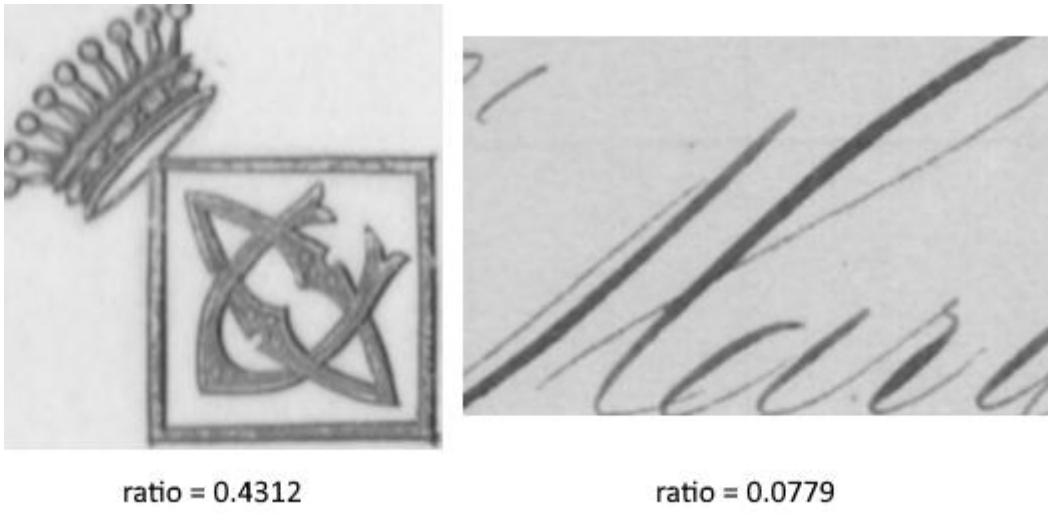


Figura 4.14. Ejemplo de ratio de relleno para dos regiones detectadas

Se ha dejado al sistema correr sin filtrar según el ratio y manualmente se ha seleccionado cuáles de las regiones devueltas corresponden a un sello y cuáles no. Estos son los datos recogidos.

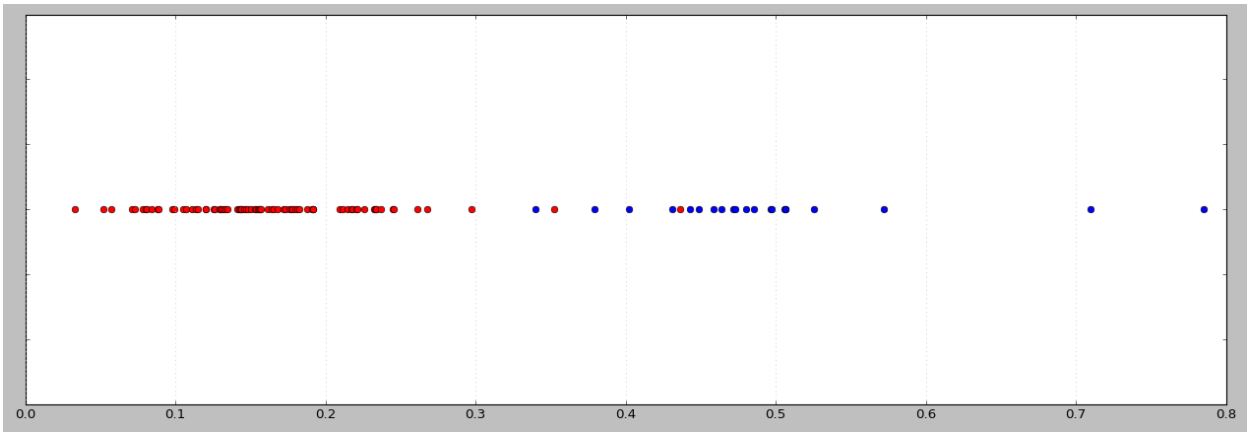


Figura 4.15. Distribución de ratios de relleno. En rojo aquellas regiones que no son un sello y en azul las que sí.

Los resultados son claros: el filtrado de regiones según estos parámetros parece efectivo, mientras los sellos cumplan la condición de estar claramente separados del texto (sin ningún solapamiento) y con una morfología diferenciable a la que típicamente presenta el texto. A continuación se muestra un ejemplo de una correcta identificación de un sello seguido de otro en el cual el sistema prototipo no es capaz de localizarlo.

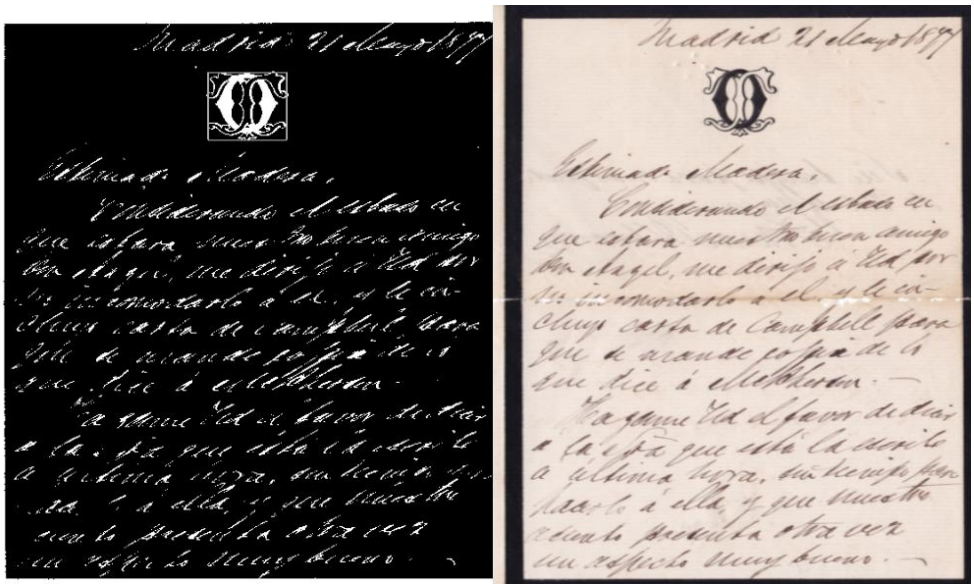


Figura 4.16. Ejemplo del correcto funcionamiento del algoritmo.

Figura 4.17. Ejemplo de sello incapaz de ser detectado por el algoritmo actual

En el segundo ejemplo, la región detectada incluye además del sello todo el texto que le rodea por encontrarse demasiado próximo. Esto invalida todas las pruebas siguientes.

4.2.4 Conclusiones sobre segmentación

Por un lado disponemos de detección de características, un sistema que parece funcionar muy bien pero que requiere una imagen previa de cada sello. Por otro lado tenemos componentes conexas, que no es robusto para la totalidad de los casos. La solución propuesta final consiste en utilizar componentes conexas para extraer una muestra de cada sello y a continuación emplear detección de características a partir de la base de datos de sellos obtenida para localizar la posición del sello a la vez que clasificarlo. El algoritmo se divide de esta manera en dos fases:

1. Fase heurística, en la que se emplea componentes conexas y una serie de filtros heurísticos con el objetivo de conseguir al menos una muestra de cada uno de los sellos
2. Fase de extracción de características, donde se emplean los prototipos de sellos hallados para localizar y reconocer los sellos, si los hubiera, de cada uno de los documentos.

4.2.5 Notas sobre la segmentación mediante transformada de Hough

Con inspiración en la literatura especializada [4] se experimentó con un método basado en la transformada de Hough. Los sellos no presentan, en un caso general ninguna curva de parametrización sencilla. Aun así, pueden estar presentes en algunos casos elementos similares a circunferencias o elipses. Se prueba a buscar estas formas geométricas en los documentos y se obtienen los resultados siguientes.

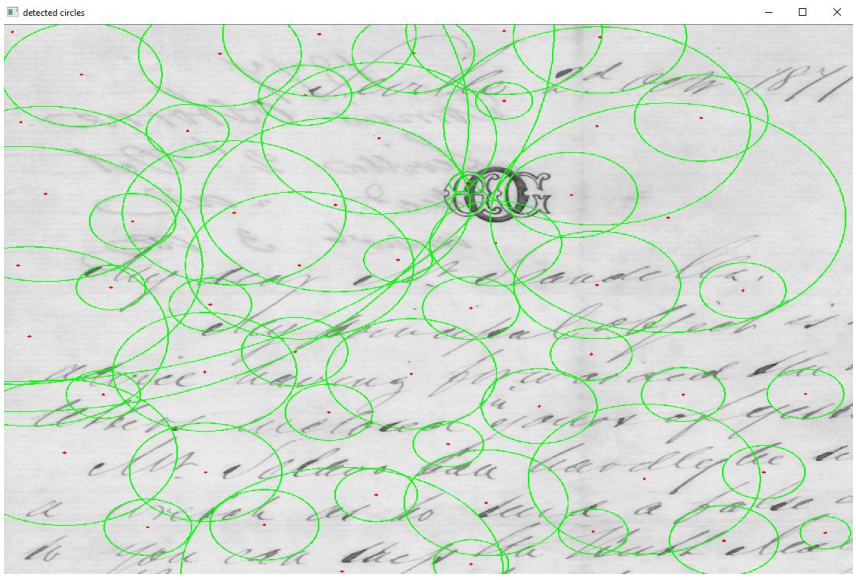


Figura 4.18. Resultados de la transformada de Hough para círculos.

Bajando lo suficiente el umbral, el sistema detecta correctamente algunas de las formas en el sello, sin embargo, esto hace aparecer formas por toda la imagen. Como cabía esperar, la transformada de Hough no supone un método robusto para abordar el problema que se presenta. Esto se debe a que los sellos no presentan, en el caso general, formas geométricas sencillas. Menos aún existe parametrización alguna de alguna curva que sea útil en la generalidad de los casos presentes.

4.3 Pasos restantes de un OCR estándar. Separación de líneas y palabras y sistema de reconocimiento

En el caso del problema expuesto, no estamos tratando de transcribir un texto. Es por ello que, supuesta resuelta la segmentación, no parece necesario estudiar sistemas para separar líneas, y menos palabras. En cuanto al sistema de reconocimiento, en el caso dado debe clasificar de qué sello en cuestión se trata. Como ya se ha mencionado el sistema de extracción de características realizará esta función.

Como se aclaró al exponer el esquema estándar de un sistema de OCR, rara vez se implementa el modelo tal

cual. Es bastante frecuente que se omita alguno de los pasos o incluso que se agreguen pasos adicionales. En general en el campo de la visión artificial es poco frecuente encontrar soluciones multipropósito para un problema y por lo tanto, tampoco es común que todas las soluciones se adapten a un esquema rígido.

5 FASE HEURÍSTICA

Que las matemáticas son una ciencia deductiva es un cliché. En realidad consisten en prueba y error.

- Paul Halmos -

El objetivo de esta fase es encontrar al menos una muestra de todos y cada uno de los sellos que aparecen en el archivo de documentos. Estas muestras van destinadas a alimentar el sistema de extracción de características. La idea es que se cataloguen manualmente antes de la siguiente fase una vez extraídas. Esto no supone un trabajo redundante ya que la primera fase reduce el proceso de catalogación manual a unas pocas imágenes que además se encuentran ya recortadas de los documentos. Nótese que para el correcto funcionamiento del sistema global el hecho de que existan algunos falsos positivos en esta primera fase no es un gran problema mientras que el hecho de que algún sello no sea encontrado supone un fallo insalvable. Los falsos positivos pueden eliminarse durante la catalogación siempre y cuando supongan un número razonablemente pequeño. La descripción de esta etapa se hará describiendo cronológicamente su desarrollo, los resultados que se obtuvieron y cómo evolucionó en complejidad para mejorarlos. El algoritmo comienza con los siguientes pasos.

5.1 Crear e inicializar objeto documento

```
class Documento:
    img = np.array([])
    path = ""
    bin_img = np.array([])
    bin_thresh = 0
    kernel = np.ones((11, 11), np.uint8)
    label_img = np.array([])
    regions = []
    seals = []
    has_2_pages = False
    lines_y = np.array([])

    def __init__(self):
        del self.regions[:]
        del self.seals[:]
        self.lines_y = np.array([[]])
```

Las variables miembro se han creado como variables de clase y no de objeto ya que sólo va a haber una instancia de la clase Documento en todo momento. Esto **no significa que se vaya a definir como singleton**, ya que los mecanismos que el lenguaje provee para construir este patrón lo hacen engorroso sin gran ventaja para el caso expuesto. Para la inicialización es necesario vaciar las listas y arrays mostrados ya que éstos se van a construir añadiendo elementos en forma de cola.

5.2 Cargar imagen del documento

```
def load_img(self, path):
    self.path = path
    self.img = cv2.imread(self.path, cv2.IMREAD_GRAYSCALE)
```

La imagen se carga directamente como escala de grises. En todo el proceso no se va a utilizar la información del color.

5.3 Obtener imagen binaria

```
def get_bin_img(self):
    self.img = cv2.GaussianBlur(self.img, (3, 3), 0)
    self.bin_thresh, self.bin_img = cv2.threshold(self.img, 0, 1,
                                    cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)
```

Se va a emplear el método de Otsu (método cv2.threshold con el parámetro cv2.THRESH_OTSU). Previamente es necesario realizar un filtrado gaussiano con el objetivo de eliminar imperfecciones puntuales en la imagen que pueden aparecer por múltiples motivos. A continuación se muestra un ejemplo en el cual este filtrado mejora notablemente la binarización.

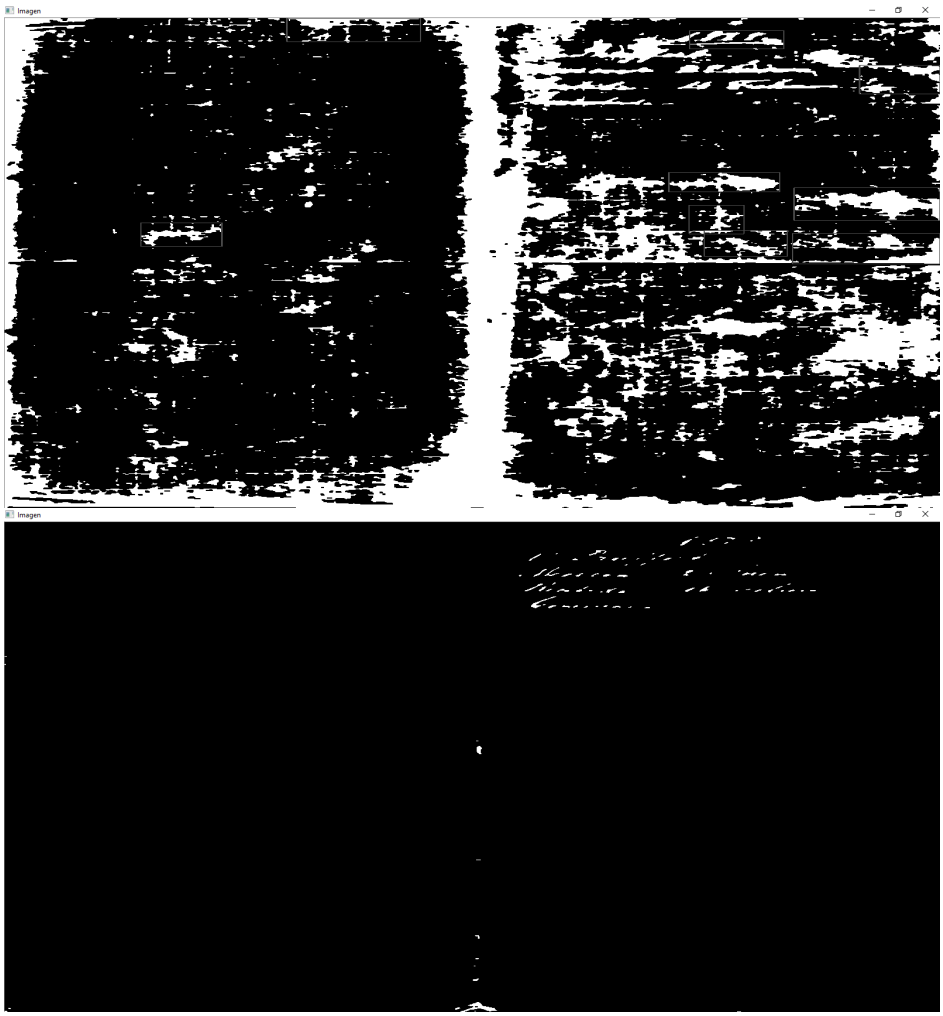


Figura 5.1. La imagen superior muestra el resultado de aplicar un umbral Otsu sobre la imagen original. La de abajo tras un filtrado gaussiano previo. Al ser papel en "blanco" casi en su totalidad, el umbral Otsu sin filtrar recoge demasiado ruido.

Se ha escogido un kernel pequeño (de dimensión 3x3) con el objetivo de no producir gran distorsión, sino

simplemente eliminar ruido.

5.4 Obtener regiones

En teoría de grafos se conoce como componentes conexas a un subconjunto del grafo en el que se cumple la condición de que dos vértices cualesquiera se encuentran conectados por algún camino, sin que este tenga que ser directo, sino que puede haber otros vértices en él. Debe cumplirse además que ningún vértice se encuentra conectado a otros que no pertenezcan a dicho subgrafo (de lo contrario estos nuevos vértices pasarían a formar parte del subgrafo).

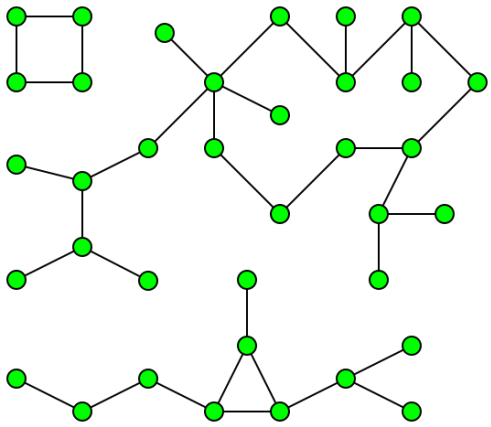


Figura 5.2. Componentes conexas de un grafo

El *etiquetado de componentes conexas* consiste en la aplicación práctica de la definición anterior a una imagen para localizar regiones de píxeles que se encuentran en contacto directo además de separadas del resto de regiones de píxeles conexos.

```
def get_regions(self):
    aux_label_img = measure.label(self.bin_img)
    regs = measure.regionprops(aux_label_img)

    self.label_img = Region.Bbox.reetiquetado(regs, aux_label_img)
    regs = measure.regionprops(self.label_img)
    i = 0
    del self.regions[:]
    for reg in regs:
        self.regions.append(Region(self, reg.bbox, reg.area, i))
        i += 1
```

La función label() toma como argumento una imagen binaria y devuelve la imagen de etiquetas, donde cada píxel contiene, en lugar de un valor binario, un entero que hace referencia a qué región pertenece. De esta forma, todos los píxeles de una misma región (píxeles conexos) poseen la misma etiqueta, mientras que píxeles de regiones diferentes poseen una diferente, es decir, cada etiqueta es única para cada región. Aquellos píxeles que pertenecían al fondo en la imagen binaria original son etiquetados con un cero.

La función regionprops() toma por argumento una imagen de etiquetas y devuelve una lista de regiones. Cada región posee una gran cantidad de propiedades computadas, como las coordenadas del píxel superior izquierdo y el inferior derecho de la caja delimitadora de la región (bounding box), o la cantidad de píxeles que contiene (denominada área de la región).

La función reetiquetado toma por argumento una lista de regiones y hace dos modificaciones:

1. Algunos documentos poseen un cuadrado negro alrededor que forma parte del papel como elemento decorativo. Este cuadrado es detectado como una región de grandes dimensiones cuando en realidad debería ser etiquetado como fondo. Esta discrepancia resulta muy problemática y se elimina convirtiendo en cero la etiqueta a aquellas regiones cuyo área de su bounding box (no el área de la región ya que esta medida solo cuenta los píxeles que forman parte de ésta) sea superior al 40% de la

imagen del documento al completo.

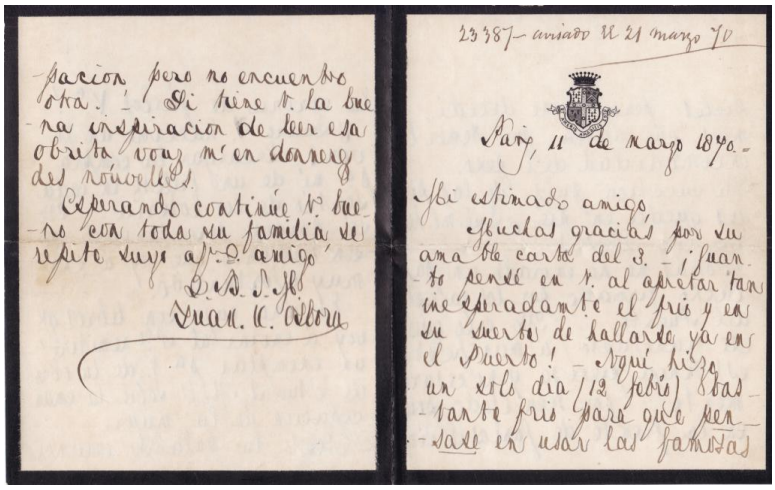


Figura 5.3. Ejemplo de documento con borde negro.

2. Con frecuencia los sellos aparecen separados en varias regiones muy cercanas o que incluso se solapan, debido a que presentan huecos en blanco que hace que partes del mismo sello no aparezcan estrictamente conexas en la imagen. Esto es un problema ya que se quiere analizar las propiedades de las regiones con el objetivo de quedarnos tan solo con aquellas cuyas propiedades supongan un indicio de que se trata de un sello. Sin embargo, si el sello aparece segmentado en varias regiones esto no será posible. Para solventarlo, se detectan aquellas regiones que intersectan o que se encuentran a menos de un número establecido de píxeles (se ha utilizado 4px como umbral). Si dos regiones intersectan, la etiqueta mayor de ambos se sustituye por la menor. De esta forma ambos pasan a ser la misma región con etiqueta igual a la menor de las dos.

```
def reetiquetado(regions, label_image):
    j = 0
    for region in regions:
        # Eliminar borde
        minr, minc, maxr, maxc = region.bbox
        bbox_height = maxr - minr
        bbox_width = maxc - minc
        img_dims = label_image.shape
        if bbox_height * bbox_width > img_dims[0] * img_dims[1] * 0.4:
            for points in region.coords:
                label_image[points[0], points[1]] = 0
        else:
            # Colisiones
            j += 1
            all_other_regions = regions
            for i in range(0, len(all_other_regions)):
                if (Region.Bbox.colision(region.bbox,
                          all_other_regions[i].bbox, 4) and
                          region.label != all_other_regions[i].label):
                    for points in all_other_regions[i].coords:
                        label_image[points[0], points[1]] = min(region.label,
                                                  all_other_regions[i].label)
                    region.label = min(region.label,
                                       all_other_regions[i].label)
                    regions[i].label = min(region.label,
                                           all_other_regions[i].label)

    return label_image
```

Se ha denominado colision() a la función que detecta intersecciones porque su comportamiento es exactamente el mismo que un detector de colisiones entre bounding boxes 2D, algoritmo básico en el mundo de los videojuegos. Una vez que se dispone de la nueva imagen de etiquetas con el posible borde eliminado y con las regiones que colisionan unificadas, se procede a recalcular las regiones y sus propiedades, como puede verse en el primer código mostrado.

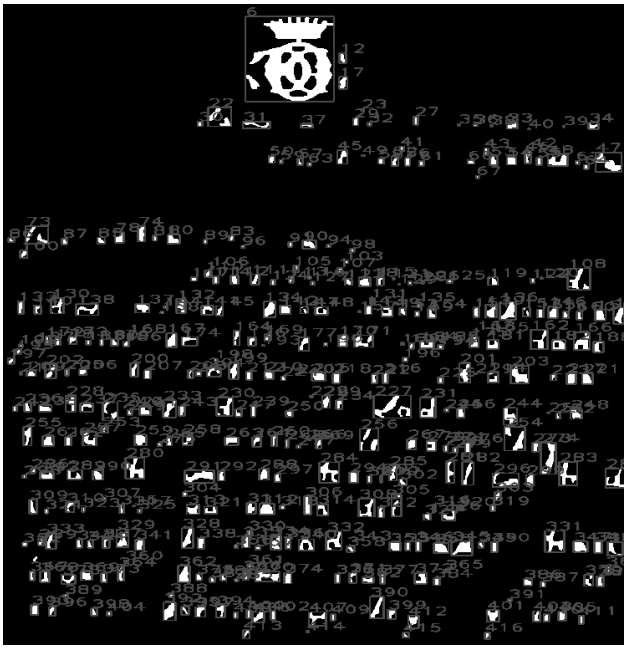


Figura 5.4. Fragmento de documento con todas las regiones obtenidas.

5.5 Filtrar regiones

Ahora que disponemos de una lista de regiones en el documento con sus respectivas propiedades, podemos proceder a eliminar todas aquellas que no son un sello. Para gestionar los test, existen dos diccionarios: uno de ellos indica cuáles de los tests están activos y nos permite fácilmente activar o desactivar alguno de estos filtros para ver cómo cambian los resultados, el otro es único para cada región e indica cuáles de los test se han superado y cuáles no.

```
class Tests:
    active_tests = {
        "area": True,
        "aspect_ratio": True,
        "filled_area": True,
        "simmetry": True,
    }

    def __init__(self, document, region):
        self.passed_tests = {
            "area": False,
            "aspect_ratio": False,
            "filled_area": False,
            "simmetry": False,
        }
```

Además, existe un tercer diccionario que tiene la finalidad de facilitar la edición de los parámetros de filtrado.

```
settings = {
    "max_area": 40000,
    "max_aspect_ratio": 3,
    "min_filled_area_ratio": 0.2,
    "max_filled_area_ratio": 0.9,
    "symmetry_ratio_thresh": 0.25,
```

```
    "symm_recheck_enlargement_px": 10,
    "symm_recheck_thresh": 0.1,
}
```

A continuación se explicarán cada uno de los test, además de los parámetros necesarios para ellos.

El primer y lógico filtrado es eliminar aquellas regiones que sean tan pequeñas que no es posible que sean un sello. Ya que poseemos un registro con todos los sellos que pueden aparecer en un documento elaborado durante la etapa de *ground truth* y que buena parte de ellos poseen dimensiones similares, podemos tomar el más pequeño de ellos, disminuir un poco el área de su imagen y fijar este valor como umbral.

```
def area(self):
    width = self.region.maxr - self.region.minr
    height = self.region.maxc - self.region.minc
    if width * height <= Region.settings.get("max_area"):
        self.passed_tests.update({"area": False})
    else:
        self.passed_tests.update({"area": True})
```

La siguiente comprobación es que el sello no sea demasiado alargado. Con frecuencia las regiones pertenecen a palabras, que tienden a generar cajas bajas y alargadas. Algunas de las regiones son parte del pliegue del papel que por algún motivo, sufre un envejecimiento mayor que el resto del papel y se oscurece en ocasiones notablemente más. Estas regiones les ocurre justo lo contrario, son estrechas y muy altas. Para filtrarlas se calcula el ratio de aspecto, tanto ancho dividido por alto como su inversa, y se establece que se encuentre en un intervalo delimitado por un umbral prefijado.

```
def aspect_ratio(self):
    width = self.region.maxr - self.region.minr
    height = self.region.maxc - self.region.minc
    if (width / height > Region.settings.get("max_aspect_ratio") or
       height / width > Region.settings.get("max_aspect_ratio")):
        self.passed_tests.update({"aspect_ratio": False})
    else:
        self.passed_tests.update({"aspect_ratio": True})
```

Ahora se procederá a filtrar aquellas regiones con un ratio de relleno demasiado pequeño como ya se vio anteriormente en el apartado 4 de este documento.

```
def filled_area_ratio(self):
    width = self.region.maxr - self.region.minr
    height = self.region.maxc - self.region.minc
    filled_area = self.region.filled_area
    ratio = float(filled_area) / (width * height)
    # umbral superior solo para evitar falsos positivos de regiones negras
    if (ratio < Region.settings.get("min_filled_area_ratio") or
       ratio > Region.settings.get("max_filled_area_ratio")):
        self.passed_tests.update({"filled_area": False})
    else:
        self.passed_tests.update({"filled_area": True})
```

Puede sorprender que sea necesario establecer un límite superior. En ocasiones se presentan regiones completamente oscuras. Un sello nunca será completamente oscuro (los ratios de los sellos se encuentran típicamente alrededor de 40-50%).

A continuación se realiza el más complejo de los test. Se ha observado que buena parte de los test presentan una clara simetría mientras que los falsos positivos no. Se va a comprobar si puede utilizarse esta información para eliminar falsos positivos.

Para calcular la simetría, se procede de la siguiente forma:

1. Se corta la región por la mitad por un eje vertical. Se voltea y se substrae una mitad de la otra. Al ser imágenes binarias, tan solo existen dos posibilidades. Para cada píxel que ambas tengan en blanco o ambas en negro obtendremos un píxel negro. En caso contrario, obtendremos uno blanco en la imagen resultado. Como puede verse, el comportamiento que se busca es un XOR. Realizar esta operación

mediante XOR nos permite además no tener que encargarnos del overflow, ya que los operandos no dejan de ser enteros de 8 bits, y no binarios reales (en el caso de operar 0 - 255 el resultado es 1, cuando el valor deseado sería 255; esto no solo ofrece resultados indeseables sino que además altera los resultados en función del orden de los operandos).

2. Se calcula el ratio del área obtenida con respecto al área original de una de las dos mitades. Si este ratio vale 0, significa simetría perfecta. A medida que el valor crece, significa menos y menos simetría.

```
@staticmethod
def symmetry_ratio(bin_seal):
    """
    Medidor de ratio de simetría. Si la imagen tiene un pixel de alto o
ancho (es un entero en lugar de un array) o menos (es tipo None porque no
se ha cargado), devuelve 'inf' para evitar errores en tiempo de ejecución.
Es importante distinguir entre imágenes con ancho par o impar a la hora de
plegarlas sobre sí para descartar o no la columna central. Esto evita que
se operen con matrices de dimensiones incompatibles, lo que también
generaría un error en tiempo de ejecución.
    :param bin_seal: imagen binarizada de la región a testear
    :return: ratio de simetría. NOTA: Cuanto más cerca de 0, más
simétrico.
    """
    fil, col = bin_seal.shape
    if col <= 1 or fil <= 1:
        return float('inf')

    if col % 2 != 0:
        img_right = bin_seal[0:fil, int(col / 2) + 1:col]
    else:
        img_right = bin_seal[0:fil, int(col / 2):col]
    img_left = bin_seal[0:fil, 0:int(col / 2)]

    flip_left_img = cv2.flip(img_left, 1)  # 1 significa eje y
    subtracted_img = flip_left_img ^ img_right  # ^: operador XOR

    sub_area = np.sum(subtracted_img)
    ref_area = (np.sum(img_left) + np.sum(img_right)) / 2.0

    ratio = sub_area / ref_area
    return ratio
```

Nótese que la función anterior es un método estático por lo tanto no tiene acceso a ninguna de las propiedades del objeto (ni tampoco las necesita).

Ahora que ya tenemos cómo calcular el ratio de simetría, vamos a proceder a emplearlo. El procedimiento será el siguiente:

1. Se calcula el ratio de simetría de la región.
2. Si este no estuviera por debajo del límite establecido se agranda la región empleando las coordenadas de ésta y la imagen original. Este agrandamiento se realiza añadiendo el parámetro **"symm_recheck_enlargement_px"** en todas las direcciones.
3. Una vez que la imagen se ha agrandado, se vuelve a binarizar con un umbral ligeramente superior al original. Concretamente uno más el parámetro **"symm_recheck_thresh"** veces.
4. A esta región ligeramente oscurecida se le calcula su nueva caja delimitadora ya que el agrandamiento es arbitrario y rara vez coincidirá. Ya tenemos la región que resultaría si el umbral de binarizado hubiera sido ese. Procedemos a calcular el nuevo ratio de simetría.

5. Si fuera mayor de lo deseado, se procedería a repetir los pasos 2 a 4, con la diferencia de que el nuevo umbral a utilizar sería uno menos **"symm_recheck_thresh"**.

El motivo por el que son necesarias estas comprobaciones adicionales es porque el test de simetría es altamente sensible al ruido. Cualquier mancha adicional añadida a la región hace que la división vertical no coincida con la mitad del sello y por tanto se obtienen ratios muy inferiores a los esperados. Igualmente, cualquier elemento que le falte al sello debido a un umbral demasiado restrictivo produce el mismo efecto.

```
def symmetry(self):
    minr = self.region.minr
    maxr = self.region.maxr
    minc = self.region.minc
    maxc = self.region.maxc

    seal_img = self.document.bin_img[minr:maxr, minc:maxc]
    ratio = self.symmetry_ratio(seal_img)

    if ratio < Region.settings.get("symmetry_ratio_thresh"):
        thickness = Region.settings.get("symm_recheck_enlargement_px")
        enlarged_img = self.document.img[abs(minr - thickness):maxr +
                       thickness, abs(minc - thickness):maxc + thickness]
        enlarged_img_coords = [minr - thickness, maxr + thickness, minc -
                               thickness, maxc + thickness]

        dark_thresh = self.document.bin_thresh * (1 +
                      Region.settings.get("symm_recheck_thresh"))
        darker_seal_img = (enlarged_img < dark_thresh).astype('uint8') * 255
        cropped_img, new_coords = Region.Bbox.detectar_bbox(darker_seal_img,
enlarged_img_coords)
        ratio = self.symmetry_ratio(cropped_img)

        if ratio < Region.settings.get("symmetry_ratio_thresh"):
            light_thresh = self.document.bin_thresh * (1 -
Region.settings.get("symm_recheck_thresh"))
            lighter_seal_img = (enlarged_img < light_thresh).astype('uint8')
* 255
            cropped_img, new_coords =
Region.Bbox.detectar_bbox(lighter_seal_img, enlarged_img_coords)
            ratio = self.symmetry_ratio(cropped_img)

            if ratio < Region.settings.get("symmetry_ratio_thresh"):
                self.passed_tests.update({"filled_area": False})
            else:
                self.passed_tests.update({"filled_area": True})
                self.region.minr = new_coords[0]
                self.region.maxr = new_coords[1]
                self.region.minc = new_coords[2]
                self.region.maxc = new_coords[3]
        else:
            self.passed_tests.update({"filled_area": True})
            self.region.minr = new_coords[0]
            self.region.maxr = new_coords[1]
            self.region.minc = new_coords[2]
            self.region.maxc = new_coords[3]
```

Una vez que tenemos los cuatro filtros, se procede a aplicar los que hubiese activos.

```
def apply_active_tests(self):
    """
    Aplica aquellos test que hubiera activos. En caso de que la región supere
todos, es añadida a la lista de sellos del documento al que pertenece la
```

```
región en cuestión.
    """
    self.region.region_is_seal = True
    if self.region.test.active_tests.get("area") is True:
        self.region.test.area()
        self.region.region_is_seal *=/
            self.region.test.passed_tests.get("area")
    if self.region.test.active_tests.get("aspect_ratio") is True:
        self.region.test.aspect_ratio()
        self.region.region_is_seal *=/
            self.region.test.passed_tests.get("aspect_ratio")
    if self.region.test.active_tests.get("filled_area") is True:
        self.region.test.filled_area_ratio()
        self.region.region_is_seal *=/
            self.region.test.passed_tests.get("filled_area")
    if self.region.test.active_tests.get("symmetry") is True:
        self.region.test.symmetry()
        self.region.region_is_seal *=/
            self.region.test.passed_tests.get("symmetry")

    if self.region.region_is_seal:
        self.document.seals.append(self.region)
```

5.6 Primeros resultados obtenidos

Los primeros resultados ofrecen una evidencia clara: el test de simetría no es útil. No sólo existen múltiples sellos que no tienen una gran simetría sino que es frecuente que palabras o grupos de palabras que poseen una simetría anormalmente elevada.

Para expresar los resultados se han utilizado las siguientes definiciones:

- Precisión: Porcentaje de los elementos detectados que suponen detecciones acertadas.
- Exhaustividad: Porcentaje de la totalidad de casos positivos que han sido detectados.

Por tanto, tras desactivar dicho filtrado se obtienen los siguientes resultados.

Precisión	35.9%
Exhaustividad	84.85%
Protitipos de sellos encontrados[4]	100%

Ya se ha conseguido el objetivo principal, obtener una copia de cada sello. Sin embargo, el número de falsos positivos es demasiado alto lo que podría ser un engorro a la hora de emplear este método. Esto se refleja en una bajísima precisión. El objetivo ahora es disminuir el número de falsos positivos sin eliminar positivos reales.

5.7 Test de posición

Una buena parte de los sellos se encuentra en la parte superior del documento. Por ello, se planteó eliminar falsos positivos mediante un nuevo filtro que eliminase todos los positivos que se tuviesen una coordenada "y" superior a un cierto umbral. Sin embargo, existen algunos pocos sellos que no cumplen esta condición. Ya que se quiere priorizar localizar el mayor número posible de sellos sobre eliminar falsos positivos y que no se puede garantizar que esta condición vaya a cumplirse para otros conjuntos de documentos, se desestima el uso

[4] Porcentaje de los distintos tipos sello existente de los cuales se ha encontrado al menos una copia.

de este filtro.

5.8 Eliminación de sellos contenidos en otros

Existe un caso concreto en el cual dos regiones son detectadas como sello estando una contenida dentro de la otra.

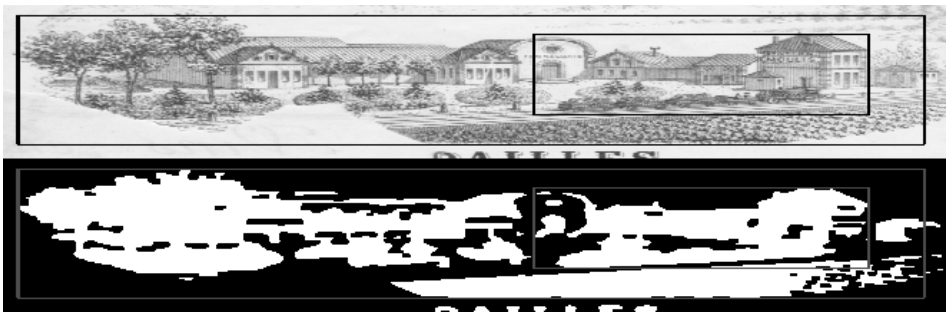


Figura 5.5. Sello detectado contenido dentro de otro sello.

Esto ocurre porque la mayor es lo suficientemente grande como para contener otra región que también lo sea. Existe una pequeña probabilidad de que la detección de colisiones unifique por separado dos conjuntos de regiones, uno contenido dentro de otro. Una segunda pasada del detector eliminaría el problema. Tendría coherencia eliminar estos casos en la propia detección de colisiones, sin embargo, allí aún no se ha eliminado ninguna región y el número total suele ser del orden de 10^3-10^4 antes del filtrado por área. Esto significa que unificar sellos que están contenidos en otros como uno solo supone realizar un número mucho mayor de operaciones en esa sección del código. Por lo tanto, se elimina aquí, después de pasar los test. En este caso se prioriza eficiencia sobre coherencia del código por el gran beneficio obtenido.

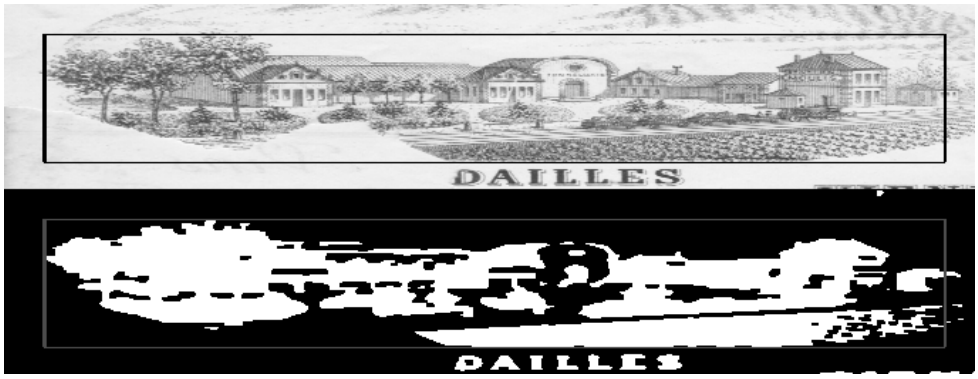


Figura 5.6 Sello detectado contenido eliminado.

5.9 Transformaciones morfológicas

Existen multitud de imágenes que presentar ruido que se corresponden con suciedad o envejecimiento del papel y que no contienen información. También se dan casos en los que la binarización elimina demasiada tinta y quedan regiones separadas que deberían estar juntas. Este último caso se solventa en cierta medida en el detector de colisiones entre regiones. Sin embargo, la capacidad para resolverlo ahí es limitada ya que unificar regiones demasiado separadas es propenso a la unificación de regiones que deberían estar separadas, especialmente cuando ambas regiones en cuestión poseen grandes dimensiones.

Para resolver todos estos problemas se realizan transformaciones morfológicas sobre la imagen binarizada con el objetivo tanto de eliminar ruido como de unificar regiones que no deben estar separadas. Para no eliminar demasiados píxeles o no unificar en exceso, es necesario ajustar los parámetros de manera adecuada. Esto se ha llevado a cabo con la ayuda del siguiente script

```
import cv2
import paths

img_path = paths.path_to_imgs + '/1863-L119.M13/107/IMG_0002.png'

img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
img2 = cv2.GaussianBlur(img, (3, 3), 0)
bin_thresh, bin_img = (cv2.threshold(img, 0, 1, cv2.THRESH_BINARY_INV +
                      cv2.THRESH_OTSU))
bin_thresh, bin_img2 = cv2.threshold(img2, 0, 1, cv2.THRESH_BINARY_INV +
                       cv2.THRESH_OTSU)


se1 = cv2.getStructuringElement(cv2.MORPH_RECT, (25, 25))
se2 = cv2.getStructuringElement(cv2.MORPH_RECT, (10, 10))
mask = cv2.morphologyEx(bin_img, cv2.MORPH_CLOSE, se1)
mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, se2)

out = img * mask
out = (out > 5).astype('uint8')

cv2.namedWindow("img", cv2.WINDOW_NORMAL)
cv2.imshow("img", bin_img*255)
cv2.namedWindow("img2", cv2.WINDOW_NORMAL)
cv2.imshow("img2", bin_img2*255)

cv2.waitKey()
```

El código permite modificar los parámetros de las transformaciones además de la imagen escogida para los experimentos y visualizar los resultados obtenidos.

Copiar y pegar código en distintas partes de un proyecto nunca es buena idea ya que cualquier edición necesaria en dicho código se vuelve muy engorrosa. Se hace necesario ir buscando todos los lugares donde ese código aparece y pegar la nueva solución. La situación es aún peor cuando algunos de esos sitios son pasados por alto, lo que hace que existan distintas versiones de lo que debería ofrecer el mismo comportamiento. El resultado es lo que se conoce como *spaghetti code*, que genera un proyecto muy difícil de hacer evolucionar y con múltiples errores.

Dicho esto, nuestro script para testear parámetros no va a formar parte del código del proyecto y hubiese sido eliminado de no ser por la necesidad de añadirlo aquí. En este caso copiar y pegar es perfectamente aceptable mientras que importar este archivo sería erróneo.

5.10 Separación de líneas

Es muy frecuente que los falsos positivos provengan de conjuntos de palabras de varias líneas. Como se vio en la figura de los ratios de relleno, no existe un umbral para este parámetro que nos permita eliminar la totalidad de falsos positivos y quedarnos con todos los sellos reales. Por ello, se ha implementado un sistema capaz de estimar las coordenadas verticales de los huecos que separan líneas de texto.

Para realizar esta tarea, se realiza un conteo de cuántos píxeles blancos existen en la imagen binaria en cada una de sus filas. Con esta información se construye un histograma de píxeles por fila.

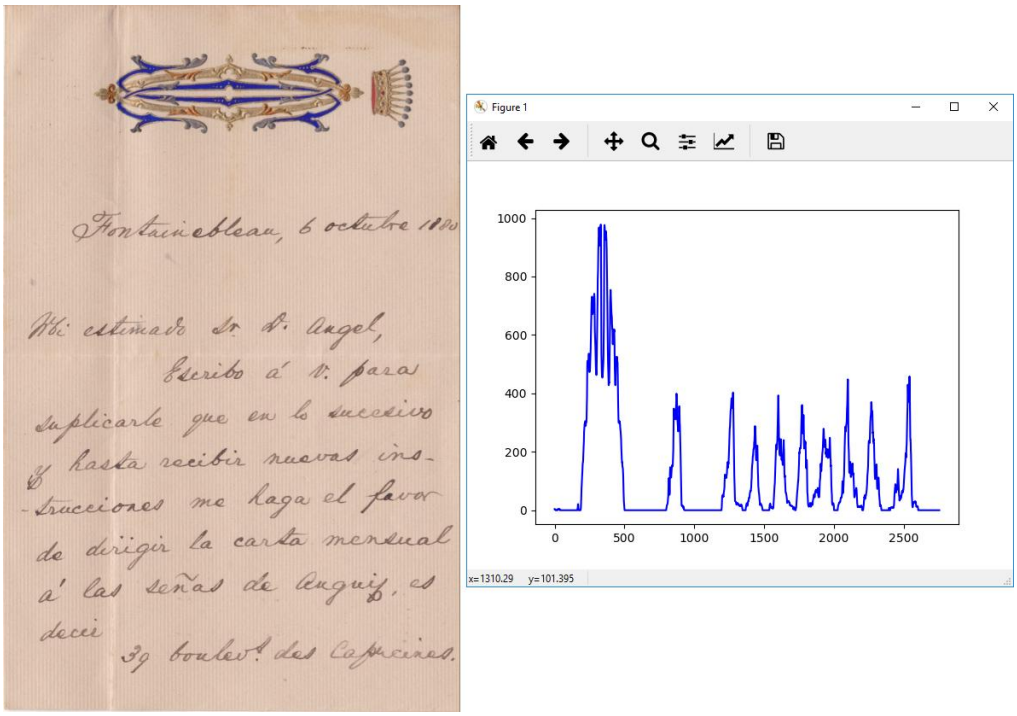


Figure 5.7. Documento junto con su histograma horizontal.

De este histograma nos interesa hallar los mínimos locales. Sin embargo puede verse que existe un gran número de mínimos que no nos resultan interesantes. Antes de realizar ninguna operación necesitamos definir qué necesitamos para poder obtener los mínimos que queremos.

- Que la serie de puntos tuviese una derivada más o menos continua (dentro del hecho de que tanto en dominio como en imagen la función se encuentra muestreada).
- Que los mínimos que localicemos se encuentren al menos separados una cierta distancia, para garantizarnos que no corresponden a la misma línea.

Para conseguir estos objetivos, hacemos pasar los datos por un filtro de paso bajo. Se ha escogido el filtro de Savitzky-Golay ya que posee la ventaja de mantener la forma y características de la señal original mejor que otros tipos de filtrados, como los basados en técnicas relacionadas con medias móviles [9]. La idea detrás del enfoque de este filtro es hacer para cada punto un ajuste por mínimos cuadrados con un polinomio de alto orden sobre una ventana de dimensión impar centrada en el punto [10].

```
@staticmethod
def savitzky_golay(y, window_size, order, deriv =0, rate=1):
    """
    Parameters
    ----------
    y : array_like (numpy, lista...), forma (N,)
        valores de la señal.
    window_size : int
        tamaño de la imagen. Debe ser un entero impar.
    order : int
        orden del polinomio a utilizar.
        Debe de ser menor de `window_size` - 1.
    deriv: int
        orden de la derivada a calcular (por defecto = 0 significa sólo
ajustar, no derivar)
    Returns
    -------
    ys : ndarray, forma (N)
        señal suavizada (o su n-ésima derivada).
    """
    from math import factorial
```

```
    try:
        window_size = np.abs(np.int(window_size))
        order = np.abs(np.int(order))
    except ValueError as msg:
        raise ValueError("window_size and order have to be of type int")
    if window_size % 2 != 1 or window_size < 1:
        raise TypeError("window_size size must be a positive odd number")
    if window_size < order + 2:
        raise TypeError("window_size is too small for the polynomials order")
    order_range = range(order + 1)
    half_window = (window_size - 1) // 2
    # precomputar coeficientes
    b = np.mat([[k ** i for i in order_range] for k in range(-half_window,
half_window + 1)])
    m = np.linalg.pinv(b).A[deriv] * rate ** deriv * factorial(deriv)
    # acomodar la señal en los extremos con valores tomados de la propia
señal
    firstvals = y[0] - np.abs(y[1:half_window + 1][::-1] - y[0])
    lastvals = y[-1] + np.abs(y[-half_window - 1:-1][::-1] - y[-1])
    y = np.concatenate((firstvals, y, lastvals))

    return np.convolve(m[::-1], y, mode='valid')
```

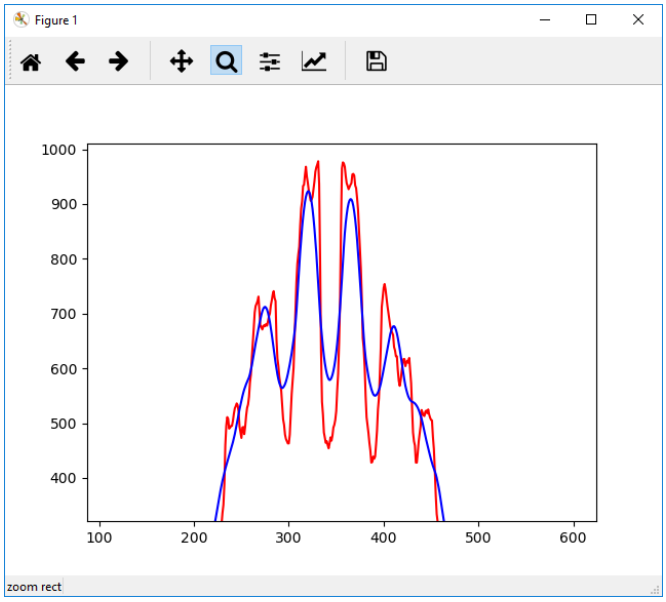


Figure 5.8. Detalle del histograma ajustado. En rojo la señal original y en azul la señal filtrada.

Una vez tenemos la señal filtrada, procedemos a encontrar los mínimos, eliminando aquellos que estén demasiado próximos. Para lograr esto, primero se toman todos los puntos que cumplen la condición de que son menores que sus dos puntos vecinos. Después, se ordenan de menor a mayor. El primero se toma como mínimo garantizado y se añade a una lista de mínimos. A partir de ahí, se recorre la lista y para cada siguiente elemento se comprueba su distancia con todos aquellos que ya han sido añadidos a la lista de mínimos. Si la distancia con todos ellos es mayor que un umbral prefijado se añade este punto a la lista de mínimos. Para calcular este umbral, se ha visto cuáles son las separaciones típicas entre líneas en píxeles, y se ha dividido por la mitad este valor.

```
@staticmethod
def find_min(a):
    """
    Encontrar el mínimos en array 'a'. Si entre mínimo y mínimo hay menos de
min_lin_dist_px píxeles se ignora el más grande de los dos.
    :param a: Array to find local minima in.
    :return: Indexes of local minima in array
```

```
    """
    # take those points which are smaller than their immediate neighbours
    is_min = np.r_[True, a[1:] < a[:-1]] & np.r_[a[:-1] < a[1:], True]
    min_points = []
    true_mins = []
    min_dist = LineSeparator.settings.get("min_lin_dist_px")

    for i, value in enumerate(a):
        if is_min[i]:
            min_points.append((i, value))

    min_points.sort(key=lambda p: p[1])  # lambda function tells sort() which
number of the tuple to use. We want to sort using y value.
    if len(min_points) > 0:
        true_mins.append(min_points[0])

        for point in min_points[1:]:
            too_close = False
            for true_min in true_mins:
                dist = abs(point[0] - true_min[0])
                if dist < min_dist:
                    too_close = True

            if not too_close:
                true_mins.append(point)

    return true_mins
```

Esta búsqueda de mínimos devuelve los siguientes resultados:

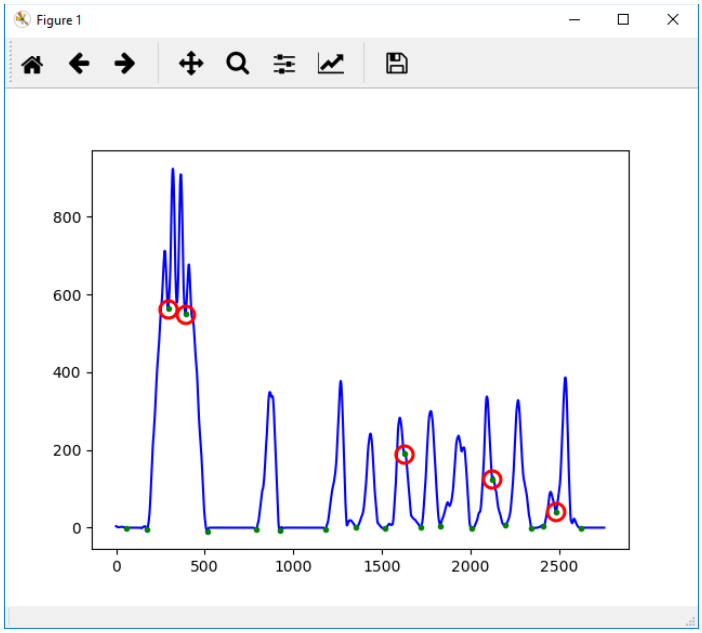


Figura 5.9. En verde, mínimos encontrados en la señal filtrada En rojo, falsos positivos (seleccionados manualmente).

Como puede verse, existen algunos falsos positivos. Éstos tienen valores claramente diferentes y además no son numerosos. Estas cualidades hacen fácil su eliminación. Para conseguirlo se ha empleado un método basado en la desviación de la mediana [11].

```
def is_outlier(points, thresh=2.0035):
    """
    Returns a boolean array with True if points are outliers and False
    otherwise.

    Parameters:
```

```
    ----------
        points : An numobservations by numdimensions array of observations
        thresh : The modified z-score to use as a threshold. Observations
with a modified z-score (based on the median absolute deviation) greater than
this value will be classified as outliers.

NOTE: Thresh estimated from average of min and max thresholds that would
filter properly for test image.

    Returns:
    --------
        mask : A numobservations-length boolean array.

    if len(points.shape) == 1:
        points = points[:, None]
    median = np.median(points, axis=0)
    diff = np.sum((points - median) ** 2, axis=-1)
    diff = np.sqrt(diff)
    med_abs_deviation = np.median(diff)

    if med_abs_deviation == 0:
        return np.ones(len(points), dtype=np.bool)
    else:
        modified_z_score = 0.6745 * diff / med_abs_deviation
        return modified_z_score > thresh
```

Como indica la *docstring*, el parámetro thresh se ha obtenido probando distintos umbrales. El umbral empleado resulta de la media entre el mayor y menor umbral que filtraban correctamente en una imagen de prueba. Los resultados de este filtrado se muestran a continuación.

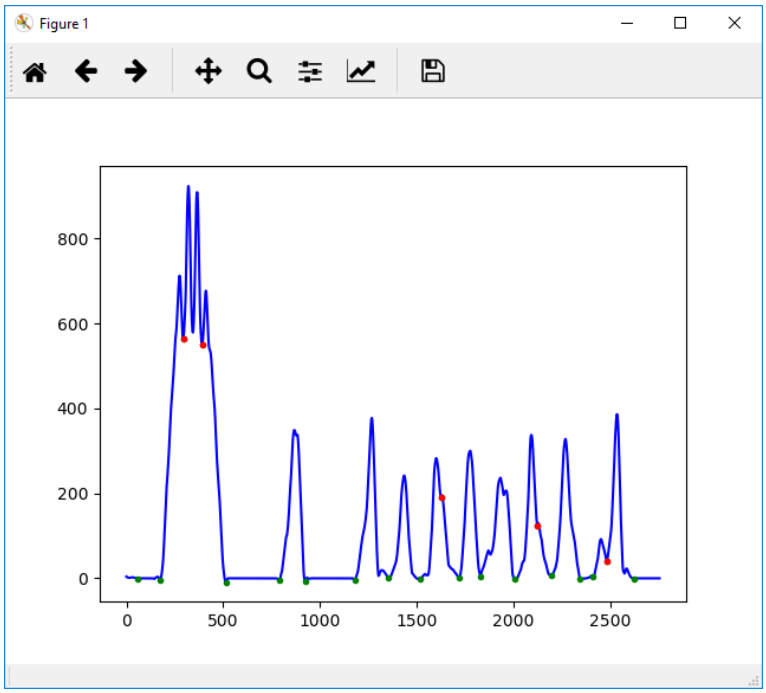


Figura 5.10. En rojo los mínimos que no han pasado el filtro.

Aún no hemos terminado. Muchos de los documentos poseen dos páginas. Las páginas de cada uno de los documentos no tienen porqué tener líneas coincidentes, de hecho rara vez ocurre. Por lo tanto es necesario dividir el documento en dos si este fuera el caso. Esto se hace calculando el histograma de píxeles, esta vez sobre el eje horizontal. Sólo se hace en el tercio central de la imagen, ya que fuera de este intervalo nunca

estará la separación buscada. Se convierte en cero entonces todo valor del histograma menor de 10, para eliminar el efecto del ruido. Entonces se busca la mayor racha consecutiva de valores. Si ésta resulta ser de ceros y además es mayor que 1/7 del ancho de la imagen, se establece que es una separación entre páginas.

```
def longest_streak(a):
    """
    Finds longest streak in a list or numpy array
    :param a: list or numpy array
    :return: longest streak found
    """
    lst = []
    for n, c in groupby(a):
        streak = list(c)
        if len(streak) > len(lst):
            lst = streak

    return lst


def has_two_pages(bin_img):
    """
    Return true if the longest streak in a vertical projection is made on
zeros and is longer than 1/7 of the document width
    :param bin_img: Document image to compute if it has two pages or not, in
binary format
    :return: True if it has 2 pages, False otherwise. It is based on the
assumption of 2 pages max per image.
    """
    first_col = int(bin_img.shape[1] / 3)
    last_col = int(2 * bin_img.shape[1] / 3)
    hist = LineSeparator.proyect(bin_img[:, first_col:last_col],
LineSeparator.axis["vertical"])
    for i, el in enumerate(hist):
        if el <= 10:
            hist[i] = 0

    max_streak = LineSeparator.longest_streak(hist)
    if max_streak[0] == 0 and len(max_streak) > int(bin_img.shape[1] / 7):
        return True
    else:
        return False
```

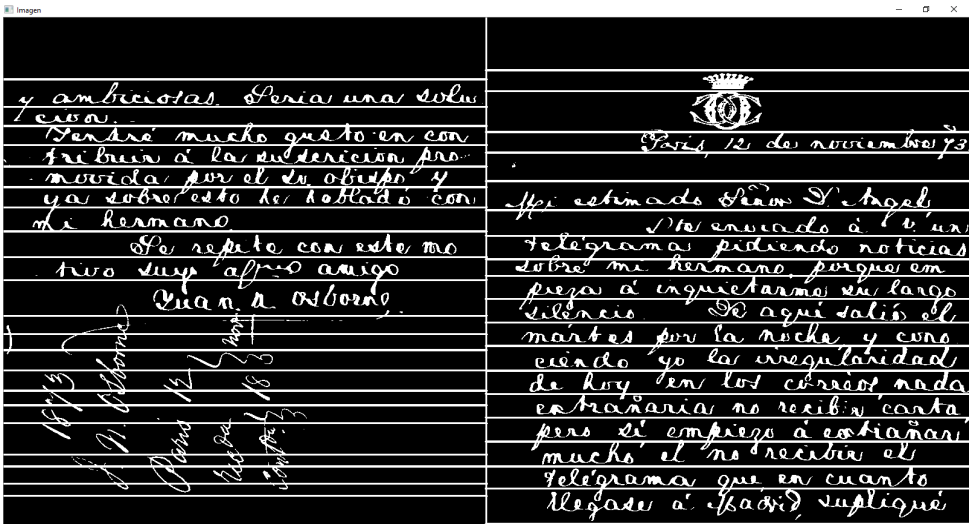
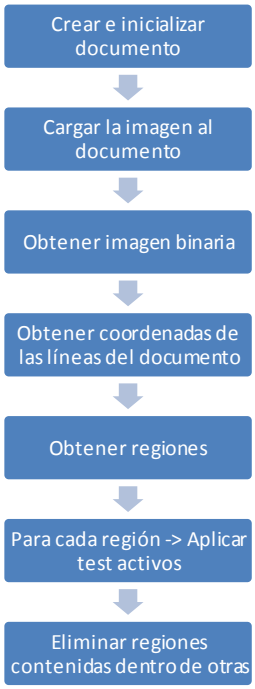


Figura 5.11. Resultado de division en páginas y líneas.

5.11 Algoritmo final para la fase heurística

El algoritmo definitivo quedaría así. Para cada imagen de documento al que se le quiera detectar su/s posible/s sello/s.



El resultado tras este proceso son las coordenadas de los posibles sellos del documento dado. En el código mostrado a continuación, se han utilizado estos datos obtenidos para almacenarlos en la base de datos (y así poder extraer estadísticas sobre la calidad de los resultados). Además, **se almacena una copia de cada fragmento de imagen que se halla determinado que podría ser un sello**. Esto es necesario para poder efectuar la fase dos.

```
import os
import cv2

from Database import DatabaseHeur
import SellosHeuristica as Sh
import paths


path = paths.path_to_imgs
walk = os.walk(path)
db = DatabaseHeur('docs_osborne', 'testuser', 'test123', 'heur_results6')

i = 0
for root, dirs, files in walk:
    for curr_file in files:
        if not curr_file.endswith(".png") or curr_file.startswith("."):
            continue
        img_path = root + '/' + curr_file
        print(img_path)
        documento = Sh.Documento()
        documento.load_img(img_path)
        documento.get_bin_img()
        documento.get_lines()
        documento.get_regions()

        for region in documento.regions:
```

```
            region.test.apply_active_tests()
        documento.elim_self_contain()

        for seal in documento.seals:
            seal_img = documento.img[seal.minr:seal.maxr,seal.minc:seal.maxc]

            width = seal.maxr - seal.minr
            height = seal.maxc - seal.minc
            cv2.imwrite(paths.path_to_heur + '/' + str(i) + curr_file,
                        seal_img)
            db.insert_results({
                "path": root.replace("\\", "/"),
                "reg_id": str(i),
            })
            i += 1
```

Los resultados obtenidos son:

Precisión	46.67%
Exhaustividad	84.85%
Prototipos de sellos encontrados	100%

Ya que solamente se han eliminado falsos positivos, obtenemos iguales datos salvo por la precisión.

6 FASE DE EXTRACCIÓN Y COMPARACIÓN DE CARACTERÍSTICAS

El arte de la simplicidad es un puzle hecho de complejidad.

- Douglas Horton -

El objetivo de la extracción de características es encontrar objetos que aparezcan en distintas imágenes, a pesar de presentar transformaciones homográficas o algún otro tipo de cambio menor, como cubrimiento parcial del objeto, deterioro en una porción minoritaria, etc. Esto se hace partiendo de una imagen que sirve de modelo del objeto a encontrar, se realiza una extracción de características donde se obtienen las tres componentes indispensables de una característica: coordenadas, intensidad (típicamente magnitud del gradiente, aunque el sentido de este valor depende de la definición usada de característica) y descriptor. Luego se hace lo mismo en la imagen completa donde se quiere encontrar el objeto. Más tarde se comparan las características encontradas en la imagen y en el objeto modelo. Aquellas que se parezcan lo suficiente suponen una correspondencia o *match*. Aquella zona de la imagen que dé suficientes matches, contiene al objeto buscado.

El proceso normal en una extracción de características es:

1. Detección de puntos clave o *keypoints*: según el algoritmo utilizado, estos puntos tienen una definición matemática diferente. En general, se busca encontrar puntos que tienen alguna propiedad matemática que los hace suficientemente diferentes al resto de puntos próximos. Este paso puede incluir un proceso de filtrado en el que se desea eliminar puntos que no son exactamente lo que se estaba buscando. Por ejemplo, si se desea buscar esquinas utilizando el gradiente, tendremos que eliminar los bordes, empleando por ejemplo la matriz hessiana.
2. Descripción de dichos puntos. A cada punto se le calcula un valor identificativo único denominado **descriptor**. En función del algoritmo escogido este descriptor puede ser un conjunto de vectores o un valor binario.
3. Matching. Consiste en la búsqueda de correspondencias de estos puntos clave en función de los descriptores. Para cada punto del modelo se le busca el punto de la imagen cuyo descriptor se parece más. Este "parecerse más" puede variar también según el algoritmo: mínima distancia euclídea o mínima distancia Manhattan[5] son los criterios más comunes, aunque no los únicos.
4. Desechado de matches erróneos. Se toma algún tipo de criterio para intentar eliminar matches erróneos en el paso anterior. El criterio más común es el descrito por D. Lowe desarrollado para utilizarlo en su algoritmo SIFT, pero extendido hoy en día casi como criterio estándar de eliminación de falsos matches. Consiste en, dado un keypoint, calcular sus dos mejores matches. Si el primero y el segundo se parecen demasiado (un 70% o más es el valor utilizado por él, aunque este valor es configurable a medida para satisfacer las necesidades de cada problema), entonces es un falso match. Si el primer y el segundo candidato son suficientemente diferentes, se considera válido. Este método elimina un alto porcentaje de falsos positivos, eliminando un número muy bajo de positivos verdaderos [12].

[5] En lugar de encontrar la distancia en línea recta, la distancia Manhattan supone la suma de las distancias en cada una de las coordenadas. Por ejemplo, en el plano: dados dos puntos (x0, y0) y (x1, y1), su distancia Manhattan sería $|x0 - x1| + |y0 - y1|$. Su nombre se debe a que sería la distancia a recorrer para llegar de un punto a otro por las calles de Manhattan ya que buena parte de sus manzanas son perfectamente rectangulares.

6.1 Elección de algoritmo para extracción de características

Existen diversos algoritmos para llevar a cabo esta tarea. Se pondrán a prueba los resultados obtenidos así como los tiempos de ejecución de los siguientes algoritmos:

- SIFT (*Scale-Invariant Feature Transform*)
- SURF (*Speeded-up Robust Features*)
- FAST (*Features from Accelerated Segment Test*)
- BRIEF (*Binary Robust Independent Elementary Features*)
- ORB (*Orientated FAST and Rotated BRIEF*)

6.2 SIFT

Este clásico algoritmo de extracción de características surge en 1999 por David Lowe [12] por la necesidad de desarrollar un algoritmo que no dependiese de la escala, es decir, que las características no desaparezcan en la detección al aparecer el objeto con distintas dimensiones en la imagen, bien por hacer zoom de cualquier forma, bien porque el objeto se ha acercado físicamente al objetivo de la cámara, bien porque el documento se ha escaneado con sensores que poseen distinta densidad de píxeles.

Previamente, el algoritmo más extendido para llevar a cabo esta tarea era el detector de esquinas de Harris [13] o el de Shi-Tomasi [14]. Éstos presenta la enorme ventaja de que una esquina detectada permanecía siendo reconocida como tal a pesar de que el objeto rotase. Es decir, dichos detectores son invariantes a la rotación. Sin embargo, al modificar la escala ocurre el siguiente fenómeno.

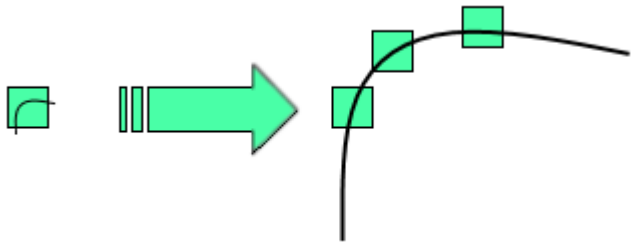


Figura 6.1. Una esquina deja de serlo al aumentar su escala.

Como puede apreciarse en la figura, el objeto sería detectado como una esquina en la situación de la izquierda. Sin embargo, en la situación derecha el objeto ha incrementado su tamaño tanto que un que la porción que queda dentro de la ventana utilizada por el detector (que podría ser en este experimento teórico incluso la imagen completa) ha dejado de ser una esquina y, por lo tanto, ningún detector de esquinas podrá localizar dónde se encuentra dicho objeto ahora.

El detector SIFT no sufre este problema, ya que es invariante en escala. Por tanto elementos que cambien de tamaño seguirán siendo reconocidos por el algoritmo. El método funciona principalmente en cuatro pasos que se describen brevemente a continuación. Las descripciones siguientes además de las figuras han elaborado con el apoyo del material de [15] y [16].

6.2.1 Paso 1: Detección de extremos en el espacio de escalas

De la imagen de arriba se deduce que tomar una ventana de tamaño fijo no es útil para detectar esquinas de tamaños variables. Para detectar esquinas mayores necesitaremos una ventana mayor. Para esto, se utiliza un filtrado en el espacio de escala. En él, el *Laplaciano del Gaussiano* (LdG de ahora en adelante) es calculado para la imagen con varios valores de σ.

Definición Laplaciano del Gaussiano:

Dada una imagen de entrada f(x,y), esa imagen está convuelta con un kernel gaussiano:

$$g(x, y, \sigma) = \frac{1}{2\pi\sigma} e^{-(x^2+y^2)/2\sigma}$$

siendo σ el denominado factor de escala del operador. Dada la imagen convuelta:

$$L(x, y; \sigma) = g(x, y, \sigma) * f(x, y)$$

Entonces se define como Laplaciano de Gaussiano:

$$\nabla^2 L = L_{xx} + L_{yy}$$

Este operador devuelve una fuerte respuesta positiva para regiones oscuras de dimensión $\sqrt{\sigma}$ y respuestas fuertemente negativas para regiones claras de igual tamaño [17].

El LdG actúa como un detector de blobs, que detecta blobs de distintos tamaños debido al cambio en el valor de σ. En resumen, σ actúa como un parámetro de escalado. Por ejemplo, en la imagen superior, LdG con un σ pequeño devuelve valores altos para la esquina pequeña mientras que con un σ mayor, los devuelve para esquinas más grandes. Por tanto, podemos localizar los máximos locales a través tanto para cada escala como comparando escalas distintas con lo cual obtendríamos una lista de valores (x, y, σ) para cada potencial *keypoint* (x, y) en la escala σ.

El problema es que el LdG es algo costoso desde un punto de vista computacional. Para solventar este inconveniente SIFT utiliza en su lugar la diferencia gaussiana o *Diferencia de Gaussianos* (DdG de ahora en adelante), que **es una aproximación** de LdG.

Definición Diferencia de Gaussianos:

Dado que la imagen en el escala-espacio cumple la ecuación de difusión

$$\frac{\partial L}{\partial \sigma} = \frac{1}{2}\nabla^2 L$$

Tenemos la siguiente igualdad

$$\nabla^2 L = \lim_{\Delta\sigma \to 0} \frac{1}{2\Delta\sigma}(L(x, y, \sigma + \Delta\sigma) - L(x, y, \sigma - \Delta\sigma))$$

En la práctica la DdG es obtenida como la diferencia entre dos imágenes convolucionadas con dos kernels gaussianos con distinta σ lo suficientemente parecidas. El proceso se repite para diferentes niveles de una pirámide gaussiana.

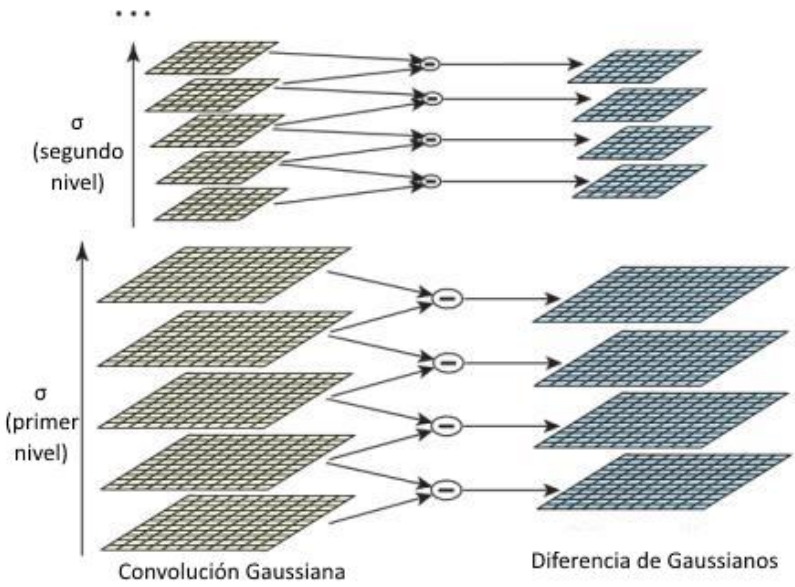


Figura 6.2. Proceso de obtención de la diferencia de gaussianos.

Una vez que las imágenes de diferencias de gaussianos son obtenidas, se buscan extremos locales comparando en cada escala los 8 valores vecinos además de los 9 contiguos en una escala superior y los otros 9 que se encuentran en la inferior. Si es un extremo local, es un punto clave potencial.

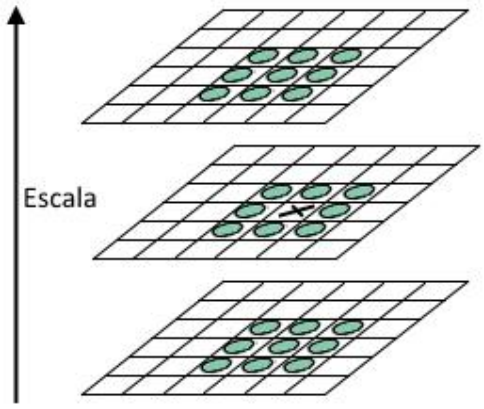


Figura 6.3. Búsqueda de extremos locales en el espacio de escalas.

Con respecto a los parámetros a utilizar, se recomienda en [12], basándose en datos obtenidos de manera empírica utilizar 4 niveles para la pirámide, 5 escalas partiendo de $\sigma = 1.6$ e incrementando dicho valor en $\sqrt{2}$.

6.2.2 Paso 2: Afinamiento en la localización de puntos clave.

Una vez que los puntos clave han sido localizados, es necesario un proceso de afinamiento de los valores obtenidos para mejorar la precisión de los resultados. Para ello, se utilizan series de expansión de Taylor para mejorar la localización de extremos locales en el espacio de escalas. Además de esto, si este extremo no supera un cierto umbral (0.03 es el valor utilizado por Lowe), es rechazado.

Cabe destacar también, que DdG ofrece una respuesta notable para bordes, así que estos deben ser eliminados. Para esto se utiliza una idea similar a la que se usa para el detector de esquinas Harris. Se toma una matriz Hessiana de dimensión 2x2 para calcular la curvatura. En el caso de bordes en lugar de esquinas un autovalor de dicha matriz será significativamente mayor que el otro. Por lo tanto, si el ratio de un autovalor con respecto

del otro es superior a un cierto umbral (10 es el valor propuesto por Lowe), es descartado.

Este paso elimina cualquier punto clave con bajo contraste, ya que es muy probable que se deban a ruido. Se eliminan además puntos situados en bordes para quedarnos sólo con las esquinas. De esta manera quedan tan sólo puntos de gran interés y utilidad.

6.2.3 Paso 3: Asignación de orientación

Ahora, asignamos una orientación a cada punto clave obtenido para conseguir la invariancia a la rotación. Se toma una vecindad alrededor del punto dependiendo de la escala, y se calculan la magnitud y dirección del gradiente en dicha región para cada píxel. Se crea un histograma de orientaciones con 36 intervalos cubriendo los 360 grados, en el cual cada orientación es ponderada en función de la magnitud de su gradiente además de un kernel gaussiano centrado en el punto y de σ igual a 1.5 veces la escala del punto. Una vez calculado el histograma, se toma el pico más elevado además de los picos que midan al menos un 80% el valor de éste. Finalmente, se toman todas estas orientaciones y, en caso de que fueran más de una, se crean tantos puntos clave como fueran necesarios con igual posición y escala pero asignándole cada una de las orientaciones obtenidas.

6.2.4 Paso 4: Creación de un descriptor para el punto

Se toma una vecindad de 16x16 alrededor del punto. Ésta se subdivide a su vez en 16 bloques de dimensión 4x4. Para cada sub-bloque, se toma un histograma de orientación de 8 divisiones. En total, 128 valores son calculados. Este vector asociado a cada punto clave se denomina descriptor. Estos valores son refinados posteriormente para mejorar la robustez frente a cambios de iluminación.

6.2.5 Paso 5: Emparejamiento de puntos

El objetivo del emparejamiento de puntos consiste en localizar qué puntos en una imagen se corresponden con el mismo objeto de otra. En este paso no se describe un método para llevar esto a cabo ya que existe toda una familia de algoritmos destinados a resolver esta tarea. Se trata de proponer un método para eliminar los falsos positivos que puedan surgir en el emparejamiento de puntos con descriptores SIFT.

Se propone tomar un punto clave de una imagen y después tomar de la otra imagen otros dos puntos cuyos descriptores sean los más parecidos a los del punto original. Dada las distancias de los descriptores de estos dos puntos al punto de la imagen, si ambas difieren menos de un 80% ambos puntos son descartados y no se empareja ningún punto de la segunda imagen con el punto en cuestión de la primera.

6.3 SURF

Durante algún tiempo, SIFT fue el estándar a la hora de resolver esta tarea, sin embargo es computacionalmente muy costoso. No sólo los procedimientos de obtención de puntos clave y descriptores sino que, además, dichos descriptores contienen un gran número de valores. Esto hace que el posterior proceso de emparejamiento de puntos se ralentice enormemente.

En 2006, Bay, H., Tuytelaars, T. and Van Gool, L publicaron un artículo llamado “SURF: Speeded Up Robust Features” [18], el cual, como indica su nombre, introdujo una versión acelerada de SIFT.

6.3.1 Búsqueda de puntos clave

En SIFT, Lowe aproxima el laplaciano del gaussiano mediante una diferencia de gaussianos con el objetivo de trasladar la búsqueda de puntos clave al espacio de escalas. SURF va un paso más allá aproximando el LdG mediante lo que denominan ellos un *filtro de cajas*. La imagen inferior ilustra una demostración de dicha aproximación. En la imagen derecha se muestra el resultado de cada una de las dos derivadas segundas parciales de la imagen original y a la izquierda la matriz utilizada como kernel de convolución para aproximar dicha derivada. Una gran ventaja de esta aproximación es que la convolución con estas "cajas" puede calcularse de manera muy rápida con ayuda de imágenes integrales. También puede realizarse en paralelo de manera independiente para distintas escalas. Por otra parte, SURF utiliza el determinante de la matriz Hessiana tanto para la escala como para la localización, disminuyendo por tanto el número de cálculos necesarios.

6.3.2 Cálculo de la orientación

Para asignar una **orientación**, SURF utiliza las respuestas a *wavelets* (en concreto la *Transformada de Wavelet de Haar* o HWT por sus siglas en inglés) en las direcciones tanto en horizontal como vertical para una vecindad de dimensión 6*s, donde *s* es la dimensión del punto calculado (no necesariamente tiene que ser un único píxel). A cada una de esas respuestas se las pondera con una adecuada función gaussiana. A continuación, se las sitúa en el espacio tal y como se aprecia en la imagen inferior. La orientación dominante se calcula sumando las respuestas que se encuentran en una abertura de 60º. Es interesante destacar que las respuestas a wavelets también se pueden calcular fácilmente mediante el uso de imágenes integrales para cualquier escala. También cabe destacar que los descriptores son independientes de esta orientación, por lo tanto, se puede prescindir de este cálculo. En nuestro caso, la invariancia a la rotación no es necesaria ya que se trata de documentos que siempre van a ser escaneados con la misma orientación. Este método es denominado Upright-SURF, o U-SURF. Mejora la velocidad de manera notable y proporciona robustez con una precisión de ±15º. Ese será el método que utilizaremos nosotros ya que los sellos pueden estar ligeramente rotados pero no nos importa calcular cuál es el valor de dicha rotación.

6.3.3 Asignación de descriptores

Para la descripción de características, SURF utiliza respuestas a wavelet de Haar en las direcciones horizontal y vertical, nuevamente con ayuda de las imágenes integrales. Se toma una vecindad de 20sx20s alrededor del punto (donde *s* es nuevamente la dimensión del punto calculado). Esta región se divide en subregiones de 4x4. Para cada una de ellas, se calcula la respuesta a wavelet nuevamente en ambas direcciones y se crea un vector de la siguiente manera:

$$v = (\sum d_x, \sum d_y, \sum |d_x|, \sum |d_y|)$$

Todos estos vectores colocados de manera consecutiva como un único vector suponen un descriptor del punto con una dimensión de 64 valores. Menor dimensión significa mayor velocidad en computación además de en el emparejamiento posterior.

Si mayor distinción para los descriptores de cada punto fuera necesaria, SURF proporciona la posibilidad de aumentar la dimensión de dichos descriptores a 128 calculando por separado las sumas de las distintas respuestas: tanto d_x como $|d_x|$ se suman por separado en función de si su respectivo d_y es positivo o negativo. Para d_y se hace lo mismo.

6.3.4 Afinamiento de emparejamiento de puntos

Para mejorar el futuro emparejamiento SURF ofrece además otra mejora. Utilizando el signo del laplaciano (que es el mismo que la traza de la matriz hessiana) para cada punto de interés. No supone ningún aumento del coste computacional ya que dicha matriz ha sido ya calculada durante la fase de detección. El signo del

laplaciano sirve para distinguir si el punto de interés supone una región clara sobre fondo oscuro o al contrario. Durante el emparejamiento sólo tenemos que comparar puntos en los cuales este valor coincida, mejorando tanto la precisión del emparejamiento como la velocidad con la que se ejecuta.

6.3.5 Comparativa con SIFT

Las pruebas demuestran que SURF supone un incremento de velocidad considerable con una precisión comparable con SIFT. SURF demuestra ser robusto ante rotación y desenfoque, sin embargo, no es muy bueno ante cambios de iluminación y de cambios en puntos de vista [18]. Esto no supone ningún problema para nuestro caso.

6.4 ORB

Como alternativa a ambos expuestos anteriormente, se desarrolló un algoritmo que fuera libre. En 2011 Ethan Rublee, Vincent Rabaud, Kurt Konolige y Gary R. Bradski publicaron un artículo titulado "ORB: An efficient alternative to SIFTor SURF" [19].

Básicamente ORB supone una fusión entre el detector de puntos clave FAST y el generador de descriptores BRIEF con modificaciones múltiples con el objetivo de mejorar el rendimiento. En un primer paso, utiliza FAST para encontrar los puntos. Después, aplica un detector de esquinas Harris con el objetivo de localizar los máximos N valores entre los puntos obtenidos previamente. Esto elimina los puntos encontrados con FAST que no posean suficiente contraste. Se utiliza además una pirámide para calcular propiedades multiescala. Un problema importante es que FAST no computa ningún valor que nos proporcione información sobre la orientación. Para conseguir invarianza ante la rotación los autores añadieron la siguiente modificación.

Se computa el centroide ponderado de la región en cuyo centro se encuentra el punto clave en cuestión. El vector que va desde el centro hasta este centroide nos proporciona la orientación. Para mejorar este valor, se computan además los momentos dentro de una región circular centrada en el punto.

Ahora, para los descriptores se hace uso de BRIEF. Nuevamente, BRIEF responde de manera pobre con respecto a la rotación, esto para el caso expuesto no supone un problema. Aún así, ORB soluciona este problema "orientando" BRIEF en la dirección calculada previamente. Para cada punto, se toma la vecindad necesaria y se rota multiplicando las coordenadas por la matriz de rotación correspondiente.

ORB discretiza en incrementos de 12º y construye una tabla de búsqueda de patrones BRIEF precomputados. Siempre y cuando la orientación utilizada inicialmente sea correcta, los valores del descriptor serán robustos con respecto a la rotación.

Una propiedad importante de BRIEF es que cada propiedad binaria obtenida tiene una alta varianza pero la media permanece cerca de 0.5. Pero una vez orientado con respecto a la dirección del punto clave esta propiedad se pierde y se vuelve más distribuido. Alta varianza significa que es el descriptor mucho más distintivo ya que responde de manera distinta ante la entrada de distintos puntos. Otra propiedad deseable sería que los test no estuvieran correlados ya que entonces cada test realizado contribuiría al resultado final. Para resolver todo esto ORB realiza una búsqueda voraz entre todos los posibles test binarios para encontrar aquellos que tienen gran varianza y media cercana a 0.5, además de no tener correlación entre sí. El resultado se denomina rBRIEF.

Para el emparejamiento de descriptores, se utiliza un LSH (del inglés *Locality-Sensitive Hashing*) el cual mejora los resultados con respecto al LSH original.

6.5 Elección del método

Para determinar qué método emplear, se han realizado dos pruebas. La primera escogiendo sellos modelo y documentos que coinciden y contabilizando la cantidad de matches de puntos clave con cada uno de los métodos, además del porcentaje de puntos emparejados erróneamente. Es decir, se han tenido en cuenta tanto la habilidad para detectar puntos clave como la calidad de los descriptores asignados a cada uno de esos puntos. Los falsos emparejamientos se han medido contando los matches que se realizan fuera del sello. Esta medida no es completamente precisa ya que puede haber matches erróneos dentro del sello, sin embargo, supone una aproximación bastante práctica y ahorra la necesidad de revisar los cientos de matches uno por uno. La segunda prueba consiste en hacer pasar a cada uno de los métodos de extracción de características por la totalidad de imágenes de documentos escaneadas ignorando la calidad de los resultados sin implementar nigún sistema para eliminar los emparejamientos erróneos, con el único objetivo de medir el tiempo de ejecución. Las conclusiones a las que se han llegado son las siguientes:

SIFT	
Puntos clave detectados	372
Porcentaje de matches erróneos	5.38%
Tiempo total de ejecución	7.41h
SURF	
Puntos clave detectados	309
Porcentaje de matches erróneos	4.85%
Tiempo total de ejecución	2.65h
ORB	
Puntos clave detectados	110
Porcentaje de matches erróneos	7.27%
Tiempo total de ejecución	$8.21*10^{-2}$h

Puede verse como ORB se ejecuta varios órdenes de magnitud más rápido que sus dos competidores. Sin embargo, la calidad de los resultados obtenidos no es tan buena. Cabe destacar que se ha empleado un *matcher* de fuerza bruta para hallar los descriptores más parecidos en el caso de ORB, que es mucho más lento. En cuanto a SURF, se ejecuta varias veces más rápido sin que eso repercuta en la calidad de los datos que devuelve con respecto a SIFT. Es el resultado que cabía esperar de acuerdo a la literatura, ya que no se esperan grandes cambios en la perspectiva o iluminación de los sellos [18]. Por todo esto, se ha optado por elegir SURF como método para abordar el problema.

Para eliminar los falsos positivos, en el algoritmo final se ha empleado información sobre la posición en la que se han encontrado los puntos clave y se ha decidido que aquellos que se alejen de la nube de puntos principal, son outliers. En caso de que esa nube principal no contenga suficientes puntos con respecto al total extraído de cada sello modelo original, se considera que el documento no posee ningún sello.

A continuación se define en profundidad el proceso final diseñado.

6.6 Descripción del algoritmo

Los objetivos son: detectar de qué sello se trata, localizar su posición y devolver una imagen sin sello. Esto es así porque para otras tareas de reconocimiento es deseable disponer de una imagen sin elementos gráficos.

Una vez que disponibles los prototipos de sellos, se necesita tener una base de datos de los keypoints y descriptores de cada sello para después emplear esta información para comparar con cada uno de los documentos. De esta manera, esta fase se divide a su vez en dos pasos: la obtención de la base de datos de características y el uso de esta base para resolver el problema.

Para obtener la base de datos se extraen los datos de cada una de las imágenes de sellos guardadas y se serializan y almacenan en un archivo utilizando Numpy. La siguiente clase permite la escritura y lectura de estos datos.

```
import numpy as np
import os
import cv2


class FeaturesIO:
    @staticmethod
    def write_features_to_file(filename, locs, desc, shape):
        np.savez(filename, np.hstack((locs, desc)), shape)

    @staticmethod
    def pack_keypoint(keypoints, descriptors):
        kpts = np.array([[kp.pt[0], kp.pt[1], kp.size, kp.angle,
                          kp.response, kp.octave,
                          kp.class_id]
                         for kp in keypoints])
        desc = np.array(descriptors)
        return kpts, desc

    @staticmethod
    def unpack_keypoint(file):
        seals_kps = []
        seals_des = []
        array = file['arr_0']
        num_elements = len(array)
        for i in range(0, int(num_elements/2)):
            kpts = array[i]

            keypoints = [cv2.KeyPoint(x, y, _size, _angle, _response,
                int(_octave), int(_class_id)) for x, y, _size,
                _angle,_response, _octave, _class_id in list(kpts)]
            seals_kps.append(keypoints)

        for i in range(int(num_elements/2), num_elements):
            desc = np.array(array[i]).astype(np.float32)  # float32 for surf;
uint8 for ORB
            seals_des.append(desc)

        seal_shps = file['arr_1']

        return seals_kps, seals_des, seal_shps

    @staticmethod
    def process_and_save(path, resultname, detector):
        walk = os.walk(path)
        all_k = []
        all_d = []
        all_s = []
```

```
        for root, dirs, files in walk:
            for curr_file in files:
                if not curr_file.endswith(".png") or curr_file ==
"no_seal.png":
                    continue

                # print(curr_file)
                img = cv2.imread(path + '/' + curr_file, 0)

                k = detector.detect(img, None)
                if len(k) > 0:
                    k, des = detector.compute(img, k)
                else:
                    des = []
                k, des = FeaturesIO.pack_keypoint(k, des)
                all_k.append(k.tolist())
                all_d.append(des.tolist())
                all_s.append(img.shape)

        FeaturesIO.write_features_to_file(resultname, all_k, all_d, all_s)

    @staticmethod
    def load_features(filename):
        file = np.load(filename)
        kp, des, shps = FeaturesIO.unpack_keypoint(file)
        return kp, des, shps
```

Su uso para almacenar puntos se muestra a continuación:

```
from FeaturesIO import FeaturesIO as FtIO
import FeaturesDetector
import paths

path = paths.path_to_prototypes

surf = FeaturesDetector.create_detector()
FtIO.process_and_save(path, "car_sellos", surf)
```

Para empleado una clase que se ha denominado EliminacionSellos y se ha creado una instancia de dicha clase para cada sello. Esta clase se describe un poco más adelante. Se recuerda que las variables de clase poseen un valor común para todos los objetos de la clase y editarlas en uno de ellos afecta a todas las instancias de la clase, mientras que las variables de objeto son únicas para cada instancia.

Se ha creado una estructura a la que se ha denominado *matriz de acumulación de evidencias* o matriz de evidencias por simplicidad. Para crear esta matriz, la imagen original se subdivide por una constante N prefijada. De esta forma, cada una de sus celdas de la nueva matriz corresponderá a un grupo de píxeles (H/N)x(W/N) del documento (salvo el borde final, que podrá ser menor), donde H y W son el alto y ancho de la imagen respectivamente. Cada celda de esta matriz poseerá un valor entero igual al número de keypoints que han sido emparejados dentro de esta celda.

6.6.1 Descripción de la clase EliminacionSellos

```
class EliminacionSellos:
    doc_img = 0
    doc_kps = []
    doc_des = []
    kps_saved = []
```

```
    desc_saved = []
    seals_dims = []
    surf = FeaturesDetector.create_detector()

    def __init__(self, img, index):
        EliminacionSellos.doc_img = img.copy()
        self.kp_matched = []
        self.evidence_matrix = em.EvidenceMatrix(img.shape)
        self.position = (0, 0)  # position is (rows, cols), therefore, (y, x)
        self.max_occurrences = 0
        self.index = index
        self.total_matches = 0


    def get_keypoints_from_db(self, path):
        EliminacionSellos.kps_saved, EliminacionSellos.desc_saved,
EliminacionSellos.seals_dims\
            = FeaturesIO.load_features(path)

    def get_document_features(self):
        EliminacionSellos.doc_kps, EliminacionSellos.doc_des =\

EliminacionSellos.surf.detectAndCompute(EliminacionSellos.doc_img, None)

    def get_matched_keypoints(self):
        # FLANN parameters
        FLANN_INDEX_KDTREE = 0
        index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
        search_params = dict(checks=50)  # or pass empty dictionary

        flann = cv2.FlannBasedMatcher(index_params, search_params)

        aux_kp = []

        matches = flann.knnMatch(EliminacionSellos.desc_saved[self.index],
EliminacionSellos.doc_des, k=2)
        for j, (m, n) in enumerate(matches):
            if m.distance < 0.7 * n.distance:
                aux_kp.append(EliminacionSellos.doc_kps[m.trainIdx])

        self.kp_matched = aux_kp
        self.total_matches = len(self.kp_matched)

    def compute_evidence_matrix(self):
        self.evidence_matrix.calc_occurrences(self.kp_matched)

    def compute_position_and_max_occurences(self):
        num_cells = int(self.evidence_matrix.SEAL_DIMENSION /
self.evidence_matrix.DIVISION_SIZE)
        kernel = np.ones((num_cells, num_cells))

        occurrences = cv2.filter2D(self.evidence_matrix.evidence_matrix, -1,
kernel)
        self.max_occurrences = np.amax(occurrences)
        max_index = np.where(occurrences == self.max_occurrences)
        avg_index = (np.average(max_index[0]), np.average(max_index[1]))

        final_coords = (int(avg_index[0].item()), int(avg_index[1].item()))

        self.position = (final_coords[0] *
self.evidence_matrix.DIVISION_SIZE,
                         final_coords[1] *
```

```
self.evidence_matrix.DIVISION_SIZE)

    def remove_seal(self):
        cv2.circle(EliminacionSellos.doc_img, (self.position[1],
self.position[0]), 200, (255, 0, 255), -1)
```

Variables de objeto

- kp_matched: lista de keypoints que han sido emparejados
- evidence_matrix: matriz de acumulación de evidencias
- position: coordenadas de las dos esquinas opuestas del sello encontrado
- max_occurrences: número de emparejamientos máximo en la matriz de evidencias

Variables de clase

- doc_img: Imagen del documento
- doc_kps: Keypoints totales encontrados en el documento
- kps_saved: Keypoints almacenados en la base de datos
- desc_saved: Descriptores correspondientes a esos keypoints
- seals_dims: Dimensiones de los sellos guardados
- detector: Detector de características configurado

6.6.2 Descripción de matriz de acumulación evidencias

```
import numpy as np


class EvidenceMatrix:
    DIVISION_SIZE = 10
    SEAL_DIMENSION = 300  # seals are about 300x300px

    def __init__(self, shape):
        fils, cols = shape
        self.evidence_matrix = np.array([])
        self.evidence_matrix.resize((fils / EvidenceMatrix.DIVISION_SIZE,
cols / EvidenceMatrix.DIVISION_SIZE))
        self.evidence_matrix.fill(0)

    def calc_occurrences(self, keypoints):
        for kp in keypoints:
            # points are x, y instead of rows and columns
            normal = (int(kp.pt[1] / EvidenceMatrix.DIVISION_SIZE),
int(kp.pt[0] / EvidenceMatrix.DIVISION_SIZE))
            self.evidence_matrix[normal[0], normal[1]] += 1
```

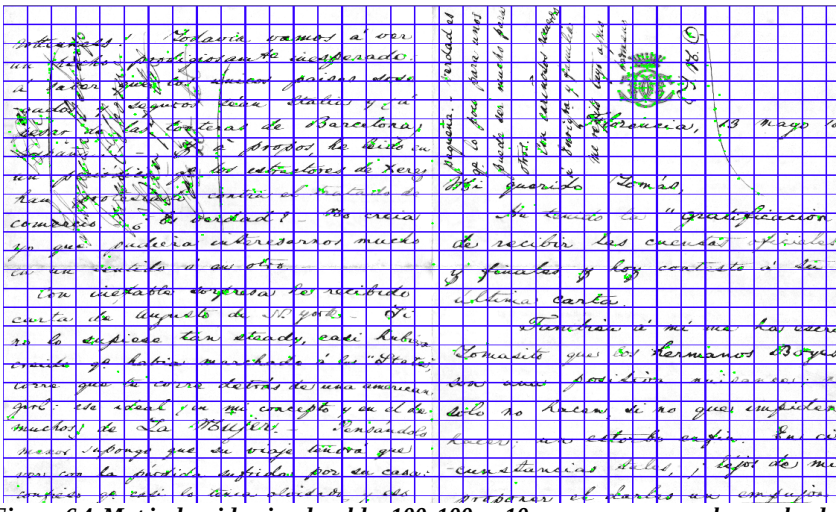


Figura 6.4. Matriz de evidencias de celdas 100x100px, 10 veces mayores que las empleada.

6.6.3 Algoritmo final

Una vez que poseemos un objeto de tipo EliminacionSellos para cada prototipo de sello almacenado, se efectúan las siguientes operaciones.

1. Se cargan los datos de sellos guardados en la base de datos de características.
2. Se calcula la matriz de evidencias para cada uno de ellos y se guarda en cada uno de los objetos creados.
3. Se obtiene el valor máximo de cada matriz de evidencias.
4. Para cada sello almacenado se calcula el ratio de máximo de su correspondiente matriz de evidencias dividido por el número total de correspondencias de puntos obtenidas en el documento.
5. Aquel sello que obtenga el mayor ratio, es el sello que aparece en el documento. Si el mayor ratio es demasiado pequeño, es que no aparece ninguno.

```
import EliminacionSellos as ElimSe


def detectar_sello(img, num_elements):
    elim_sellos = []

    for i in range(0, num_elements):
        elim_sellos.append(ElimSe.EliminacionSellos(img, i))

    elim_sellos[0].get_keypoints_from_db('car_sellos.npz')
    elim_sellos[0].get_document_features()

    real_seal = -1
    max_ratio = 0
    for i in range(0, num_elements):
        elim_sellos[i].get_matched_keypoints()
        elim_sellos[i].compute_evidence_matrix()
        elim_sellos[i].compute_position_and_max_occurences()
        if elim_sellos[i].total_matches > 0:
            if elim_sellos[i].max_occurrences / elim_sellos[i].total_matches
> max_ratio:
                max_ratio = elim_sellos[i].max_occurrences /
len(elim_sellos[i].desc_saved[i])
                real_seal = i
```

```
    elim_sellos[real_seal].remove_seal()

    center_coords = elim_sellos[real_seal].position  # (row,col) = (y,x)
    real_seal_height, real_seal_width =
ElimSe.EliminacionSellos.seals_dims[real_seal]
    div_size = elim_sellos[real_seal].evidence_matrix.DIVISION_SIZE

    pt1 = (int(center_coords[1] * div_size - real_seal_width / 2),
           int(center_coords[0] * div_size - real_seal_height / 2))
    pt2 = (int(center_coords[1] * div_size + real_seal_width / 2),
           int(center_coords[0] * div_size + real_seal_height / 2))

    corner_coords = [pt1, pt2]

    if max_ratio < 0.2:
        real_seal = -1

    return elim_sellos[real_seal].doc_img, corner_coords, real_seal,
max_ratio
```

7 RESULTADOS Y CONCLUSIONES

La conclusión es aquel punto en el que uno se cansa de pensar.

- Martin H. Fischer -

Ya que el algoritmo consta de dos fases, se discutirá primero los resultados obtenidos en cada una de éstas, para así identificar con mayor facilidad posibles mejoras.

7.1 Resultados de la fase heurística

El objetivo principal de esta fase es adquirir de manera automática al menos un prototipo de cada tipo de sello existente en el conjunto de documentos. Sería deseable además, que no se detectase ningún falso positivo. De esta forma el sistema sería totalmente automático. También sería de gran valor que el sistema pudiese distinguir cuáles de los sellos encontrados suponen distintas muestras de el mismo sello, pero para esto es necesario disponer de un sistema eficaz para reconocer qué tipo de sello corresponde a cada uno. Ya que en un primer desarrollo se quiere separar ambar funcionalidades, no se contemplará esta característica a pesar de que en un sistema completamente autónomo sería imprescindible. Se obtienen los siguientes resultados, ya recogidos en el capítulo 5:

Precisión	46.67%
Exhaustividad	84.85%
Prototipos de sellos encontrados	100%

La precisión es muy baja debido al elevado número de falsos positivos. La exhaustividad no es del 100% ya que existen algunos sellos que se escapan de la detección, en su mayoría por tener texto escrito encima. El objetivo principal, que es obtener al menos una imagen de cada tipo de sello, se ha conseguido. De hecho, se obtienen múltiples copias de todos aquellos sellos que aparecen en varios documentos (existen algunos que solo aparecen una sola vez).

Puede verse que esta fase cumple con los objetivos para nuestro caso en concreto. Para un caso general existe una probabilidad de un 15.15% de que un sello sea erróneamente detectado como que no lo es. Teniendo en cuenta que en nuestro caso existe una probabilidad de un 34.48% de que un sello solo aparezca una sola vez, tenemos que **la probabilidad de que un prototipo de sello que solo aparezca una vez sea omitido es de 5.22%**. La probabilidad de omitir los sellos que aparecen múltiples veces es aún mucho menor. Por tanto podemos decir que, aunque la enorme cantidad de falsos positivos impide la absoluta autonomía del algoritmo, esta fase funciona de manera satisfactoria.

7.1.1 Posibles mejoras al método heurístico

Una mejora clara que podría llevarse a cabo es incrementar el número de parámetros que se miden, esto permite hacer más permisivos el resto de filtros eliminando menos falsos negativos, además de obtener un nuevo filtro que elimine alguno de los falsos positivos existentes. No se ha llevado a cabo ninguna medida sobre los los momentos de las regiones. A pesar de que se haya tenido en cuenta la forma de la región, una región muy alargada puede serlo por la presencia de un pequeño trazo que represente una parte muy

minoritaria de la región. Los momentos nos ofrecen información adicional sobre la distribución de los píxeles que podría ser de gran utilidad. Además, la posición de un eje principal de inercia en una determinada orientación podría suplir tal vez las carencias que la medida del ratio de simetría posee, algo que sólo los adecuados experimentos pueden responder.

Otra medida muy relacionada con la anterior es analizar dónde se localiza la mayoría de los píxeles, ya que las regiones de texto aparentan tener una mayor concentración en general de píxeles en la parte inferior. Sería necesario medir este parámetro y establecer si verdaderamente existe alguna correlación, de la misma manera que se hizo con el ratio de relleno.

Sin duda ninguna, la mayor mejora posible que se puede realizar a este método es imponer un método de clasificación estadístico, basado por ejemplo en un *clasificador bayesiano ingenuo*. Actualmente los filtros imponen un umbral a partir del cual se elimina o se acepta la región. Este nuevo sistema permitiría agregar la probabilidad de que la región sea o no sello en función de las probabilidades individuales de cada filtro. Si fuera necesario, se podría dejar un nuevo umbral en alguna de las propiedades.

7.2 Resultados de la fase de extracción de características

Porcentaje de clasificados correctamente	51.72%
Media de ratio en caso de acierto	0.70632
Media de ratio en caso de error	0.006951

Los resultados de esta fase no son excepcionalmente buenos. Tan solo un 51.72% de los casos se han clasificado correctamente. Esto se debe a que el mero hecho de imprimir un sello en otro papel, ya modifica el entorno de una esquina, haciendo que su descriptor se vea alterado. Por ello se ha comprobado que la extracción de características no supone una solución robusta para localizar objetos con la misma forma en entornos diferentes, aunque sí lo sea para localizar el mismo objeto, que para lo que verdaderamente se han diseñado.

Sin embargo, este enfoque no es completamente inútil. Existe una clara diferencia en los ratios entre los casos en los que el sistema acierta y en los que no. Con todo esto, puede afirmarse que el sistema es apto para ser empleado en conjunto con otras medidas que permitan una clasificación más robusta, ya que se tiene una clara medida de la fiabilidad de la clasificación en cada caso.

Un dato a destacar es que, en caso de que el sello reconocido sea el correcto, el sistema detecta su posición correctamente en la totalidad de los casos, haciendo posible su eliminación del documento. Esta eliminación es necesaria para otras tareas de visión artificial que pudieran realizarse sobre la imagen.

Entre los sistemas complementarios que podrían emplearse podría encontrarse el trabajo de Kameshiro y otros [20], donde se desarrolla un sistema para representar la forma del contorno de imágenes de caracteres chinos manuscritos. Sería interesante medir cómo de bien funciona para nuestros sellos ya que, incluso si se hallara alguna propiedad que los hace muy diferentes (en todos o en algún subconjunto de éstos), también podría ser útil utilizar esta información recién descubierta para la clasificación.

Otra forma interesante de abordar el problema sería mediante el uso de la transformada de Fourier. Esta herramienta es ampliamente usada para reconocimiento de voz. Es además, por poner un ejemplo, el motor principal detrás del funcionamiento de Shazam, software capaz de reconocer canciones en tiempo real y a pesar de existir una notable cantidad de ruido de fondo [21]. Ya que una imagen no deja de ser igualmente una función digital (discretizada tanto en los valores que puede tomar como en los valores que pueden tomar las variables independientes) con la salvedad de ser bidimensional, es razonable pensar que este enfoque pueda resultar de utilidad. En [22] Hopkins y Anderser desarrollan un método para reconocer caracteres mediante esta herramienta matemática, llegando a obtener hasta un 94% de aciertos tras un extensivo afinamiento de los

parámetros del clasificador.

Anexo A. Código Ground Truth GUI[6]

Para facilitar la navegación por el código se añade el siguiente árbol de dependencias. En verde, los archivos que forman parte de librerías de python.

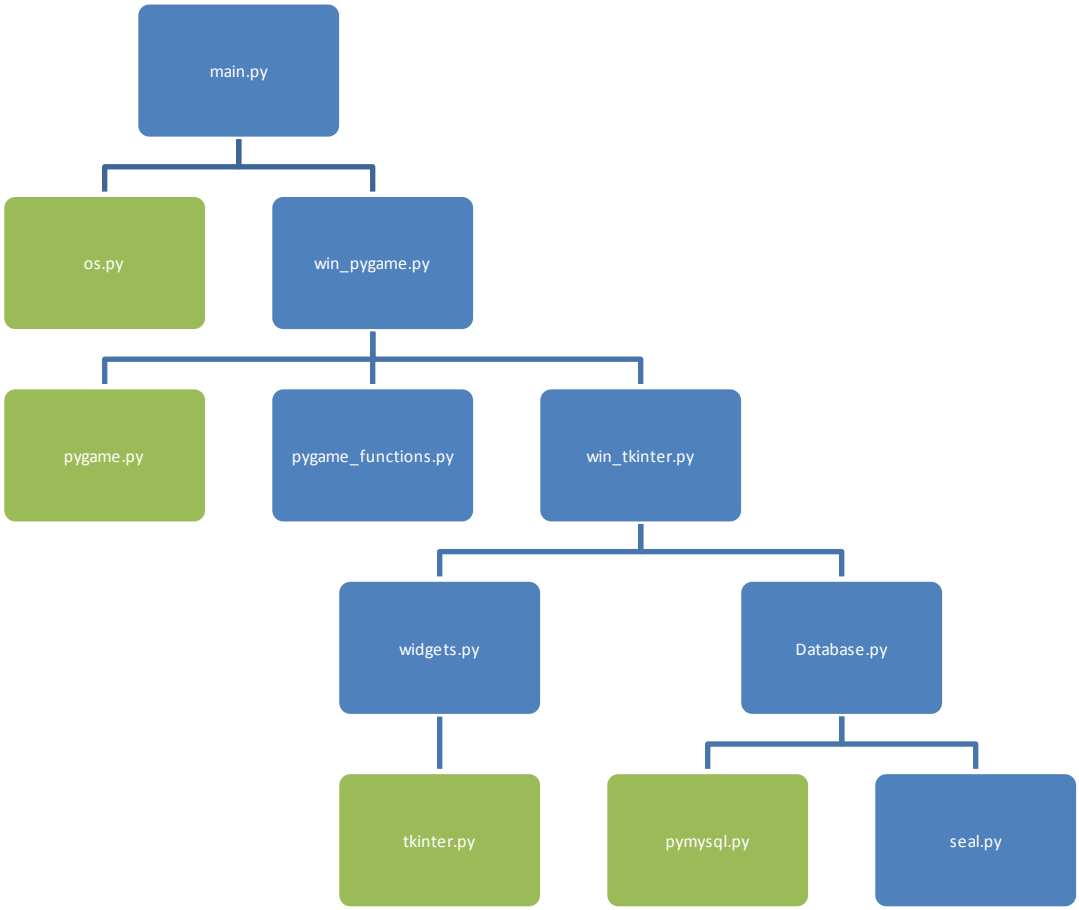


Figura A.1. Árbol de dependencias de Ground Truth GUI

main.py

```
"""
main.py se encarga de proveer a las ventanas de una lista que contenga el
directorio de cada una de las imágenes. Además le pasa el índice para saber por
qué imagen nos quedamos la última vez. También mantiene los bucles infinitos de
las ventanas
"""
import os
import win_pygame

from config import path_to_documents
```

[6] Nota aclarativa: los códigos presentes en este documento pueden formar parte de proyectos en continuo desarrollo. La versión mostrada a continuación no tiene por qué corresponderse con la versión actual en el momento de lectura. En cualquier caso, la versión más actual debería encontrarse disponible en https://github.com/miguel-rodrigo/OsborneSeals o https://github.com/miguel-rodrigo/ground_truth_GUI para el Ground Truth GUI.
Las tabulaciones y saltos de línea han sido ajustados para facilitar la lectura satisfaciendo las restricciones de Word. El código podría necesitar reajustes de estos aspectos para ser utilizado.

```
from config import path_to_index


path = path_to_documents
index_path = path_to_index
walk = os.walk(path)
index_file = open(index_path + '/' + 'index.txt')
index = int(index_file.read())
index_file.close()

doc_paths = []
for root, dirs, files in walk:
    there_is_any_img = False
    for curr_file in files:
        if curr_file.endswith(".png"):
            there_is_any_img = True

    if there_is_any_img:
        root = root.replace("\\", "/")
        doc_paths.append(root)

doc_win = win_pygame.WinPygame(doc_paths, index)

while 1:
    doc_win.main_loop()
    doc_win.control_panel.root.update()
    doc_win.control_panel.root.update_idletasks()
```

win_pygame.py

```
import os
import sys
import pygame
from pygame_functions import PygameFunctions as pf
import win_tkinter

from config import path_to_documents
from config import path_to_seals_db


class WinPygame:
    """
    Crea una instancia de la ventana de visualización de imágenes y maneja los
controles de navegación.
    Crea una instancia del panel de control
    Guarda la imagen del nuevo sello en caso de que uno sea encontrado
    """
    def __init__(self, path=None, index=0):
        self.size = self.width, self.height = 1200, 650
        self.speed = [1, 1]
        self.black = 0, 0, 0

        pygame.init()

        self.screen = pygame.display.set_mode(self.size)

        if path is not None:
            onlyimages =\
              [f for f in os.listdir(path[index])
```

```
                if os.path.isfile(os.path.join(path[index], f)) and
                f.endswith('.png')]
            self.doc_img = pygame.image.load(path[index] + '/' + onlyimages[0])
        else:
            # Random test image just for debugging
            self.doc_img =\
        pygame.image.load(path_to_documents + "/1823-L119.M3/117/IMG_0001.png")

        self.img_c = self.doc_img.copy()

        self.scale = 1
        self.originx, self.originy = (0, 0)
        self.pt1x, self.pt1y = (0, 0)
        self.pt2x, self.pt2y = (0, 0)
        self.movement_speed = 10
        self.clock = pygame.time.Clock()

        self.image =\
pygame.transform.scale(self.img_c, (int(self.doc_img.get_width() * self.scale),
int(self.doc_img.get_height() * self.scale)))

        # Only create control panel if we are not testing
        if __name__ != "__main__":
            self.control_panel =\
                win_tkinter.WinControlPanel(self, (self.pt1x, self.pt1y,
                    self.pt2x, self.pt2y), path, index)

    def main_loop(self):
        for event in pygame.event.get():
            if event.type == pygame.QUIT or (event.type == pygame.KEYUP and
event.key == pygame.K_ESCAPE):
                if __name__ == "__main__":  # If there is a control panel,
closing will be handled there.
                    sys.exit()
            elif event.type == pygame.MOUSEBUTTONDOWN:
                if event.button == 1 or event.button == 3:
                    pf.clear_old_rect(self.img_c, self.doc_img, (self.pt1x,
                                      self.pt1y, self.pt2x, self.pt2y),
                                      event.button)
                    mouse_coords = pygame.mouse.get_pos()
                    self.pt1x, self.pt1y, self.pt2x, self.pt2y = \
                        pf.draw_new_rect((self.pt1x, self.pt1y, self.pt2x,
                                          self.pt2y), self.img_c,
                                         (self.originx, self.originy),
                                          self.scale, mouse_coords, event.button)
                    if __name__ != "__main__":
                        self.control_panel.update_labels((self.pt1x, self.pt1y,
                                                          self.pt2x, self.pt2y))

                if event.button == 4 or event.button == 5:
                    self.scale = pf.scale_img(self.scale, event.button)

        # Transform image acording to translation and scale
        self.originx, self.originy = pf.move_image(self.originx, self.originy)
        self.image = pygame.transform.scale(self.img_c,
                         (int(self.img_c.get_width() * self.scale),
                          int(self.img_c.get_height() * self.scale)))

        self.screen.fill(self.black)
        self.screen.blit(self.image, (self.originx, self.originy))
        pygame.display.flip()
```

```
            self.clock.tick(60)

        def update_img(self, path):
            self.doc_img = pygame.image.load(path)
            self.img_c = self.doc_img.copy()
            self.scale = 1
            self.originx, self.originy = (0, 0)

        def save_seal(self, name):
            width = self.pt2x - self.pt1x
            height = self.pt2y - self.pt1y
            cropped = pygame.Surface((abs(width), abs(height)))
            cropped.blit(self.doc_img, (0, 0), (self.pt1x, self.pt1y, width,
                         height))

            pygame.image.save(cropped, path_to_seals_db + name + '.png')

    if __name__ == "__main__":
        win = WinPygame()
        while 1:
            win.main_loop()
```

pygame_functions.py

```
    import pygame


    class PygameFunctions:
        MOVEMENT_SPEED = 10
        SCALE_INCREMENT = 0.05

        @staticmethod
        def move_image(orx, ory):
            if pygame.key.get_pressed()[pygame.K_LEFT] != 0:
                orx += PygameFunctions.MOVEMENT_SPEED
            elif pygame.key.get_pressed()[pygame.K_RIGHT] != 0:
                orx -= PygameFunctions.MOVEMENT_SPEED
            if pygame.key.get_pressed()[pygame.K_UP] != 0:
                ory += PygameFunctions.MOVEMENT_SPEED
            elif pygame.key.get_pressed()[pygame.K_DOWN] != 0:
                ory -= PygameFunctions.MOVEMENT_SPEED

            return orx, ory

        @staticmethod
        def calc_rect(coords):
            minx = min(coords[0], coords[2])
            maxx = max(coords[0], coords[2])
            miny = min(coords[1], coords[3])
            maxy = max(coords[1], coords[3])

            return minx, miny, maxx, maxy

        @staticmethod
        def clear_old_rect(img, model_img, coords, button):
            rect_coords = PygameFunctions.calc_rect(coords)
            img.blit(model_img, (rect_coords[0] - 5, rect_coords[1] - 5),
                     area=(rect_coords[0] - 5, rect_coords[1] - 5,
                     rect_coords[2] - rect_coords[0] + 10,
```

```
                rect_coords[3] - rect_coords[1] + 10))
        if button == 1:
            pygame.draw.circle(img, (0, 0, 255), (coords[2], coords[3]), 5)
        elif button == 3:
            pygame.draw.circle(img, (255, 0, 0), (coords[0], coords[1]), 5)

    @staticmethod
    def draw_new_rect(coords, img, img_origin, scale, mouse_coords, button):
        newx, newy = (int((mouse_coords[0] - img_origin[0]) / scale),
                      int((mouse_coords[1] - img_origin[1]) / scale))
        new_coords = 0
        if button == 1:
            pygame.draw.circle(img, (255, 0, 0), (newx, newy), 5)
            pygame.draw.rect(img, (0, 255, 0), (newx, newy, coords[2] - newx,
                             coords[3] - newy), 3)
            new_coords = (newx, newy, coords[2], coords[3])
        elif button == 3:
            pygame.draw.circle(img, (0, 0, 255), (newx, newy), 5)
            pygame.draw.rect(img, (0, 255, 0), (newx, newy, coords[0] - newx,
                             coords[1] - newy), 3)
            new_coords = (coords[0], coords[1], newx, newy)

        return new_coords

    @staticmethod
    def scale_img(scale, button):
        if button == 4:
            scale += PygameFunctions.SCALE_INCREMENT
        elif button == 5 and scale > 0:
            scale -= PygameFunctions.SCALE_INCREMENT

        return scale
```

win_tkinter.py

```
import tkinter as tk
from tkinter import messagebox
import database
import widgets
import os

from config import path_to_documents
from config import db_credentials
from config import path_to_index


class WinControlPanel:
    def __init__(self, img_window, coords, paths, index=0):
        self.root = tk.Tk()
        self.root.geometry("500x570+30+30")

        self.db = database.Database(db_credentials['db_name'],
                                    db_credentials['user_name'],
                                    db_credentials['pwd'],
                                    db_credentials['tables'])
        self.db.load_seals()

        self.coords = coords
        self.paths = paths
        self.path_index = index
```

```
            # POINT 1 DISPLAY
            self.pt1_label = tk.Label(self.root, text="Point 1:")
            self.pt1_label.place(x=20, y=10)

            self.pt1x_value = tk.IntVar()
            self.pt1x_info = tk.Entry(self.root, state=tk.DISABLED,
                                      textvariable=self.pt1x_value)
            self.pt1x_info.place(x=70, y=10, width=50)
            self.pt1x_value.set(str(coords[0]))

            self.pt1y_value = tk.IntVar()
            self.pt1y_info = tk.Entry(self.root, state=tk.DISABLED,
                                      textvariable=self.pt1y_value)
            self.pt1y_info.place(x=130, y=10, width=50)
            self.pt1y_value.set(str(coords[1]))

            # POINT 2 DISPLAY
            self.pt2_label = tk.Label(self.root, text="Point 2:")
            self.pt2_label.place(x=20, y=30)

            self.pt2x_value = tk.IntVar()
            self.pt2x_info = tk.Entry(self.root, state=tk.DISABLED,
                                      textvariable=self.pt2x_value)
            self.pt2x_info.place(x=70, y=30, width=50)
            self.pt2x_value.set(str(coords[2]))

            self.pt2y_value = tk.IntVar()
            self.pt2y_info = tk.Entry(self.root, state=tk.DISABLED,
                                      textvariable=self.pt2y_value)
            self.pt2y_info.place(x=130, y=30, width=50)
            self.pt2y_value.set(str(coords[3]))

            # NEW SEAL WINDOW
            def on_new_seal():
                self.new_seal_win = WinNewSeal(self.db, img_window)
            self.new_seal_butt = tk.Button(self.root, text='New',
                                           command=on_new_seal)
            self.new_seal_butt.place(x=200, y=20)

            # SEAL SELECTION ITEMS
            self.seal_type_list = widgets.SealsList(self.root, self.db)
            self.seal_type_list.x = 20
            self.seal_type_list.y = 70
            self.seal_type_list.place_items()

            # OK BUTTON
            def on_ok_button():
                self.db.insert_document(self.paths[self.path_index],
                                        self.seal_type_list.curr_seal_type.get(),
                                       (self.pt1x_value.get(), self.pt1y_value.get(),
                                        self.pt2x_value.get(), self.pt2y_value.get()))

                if self.path_index < len(self.paths):
                    self.path_index += 1
                    onlyimages =\
                      [f for f in os.listdir(self.paths[self.path_index])
                       if os.path.isfile(os.path.join(self.paths[self.path_index],
                                                      f)) and f.endswith('.png')]

                    # Llamar método de ventana de pygame que actualiza a la nueva
imagen
                    if __name__ != "__main__":
```

```
                    img_window.update_img(self.paths[self.path_index] + '/' +
                                          onlyimages[0])
            else:
                messagebox.showinfo("End of classification", "There are no more
documents to classify")

        self.ok_button = tk.Button(self.root, text='OK',command=on_ok_button)
        self.ok_button.place(x=250, y=20)

        def on_closing():
            if messagebox.askokcancel("Quit", "Do you want to quit?"):
                index_file = open(path_to_index + '/index.txt', 'w')
                index_file.write(str(self.path_index))
                index_file.close()
                self.root.destroy()

        self.root.protocol("WM_DELETE_WINDOW", on_closing)

    def update_labels(self, new_coords):
        self.pt1x_value.set(str(new_coords[0]))
        self.pt1y_value.set(str(new_coords[1]))
        self.pt2x_value.set(str(new_coords[2]))
        self.pt2y_value.set(str(new_coords[3]))


class WinNewSeal:
    def __init__(self, db, img_win):
        self.db = db
        self.img_win = img_win

        self.root = tk.Tk()
        self.width = 300
        self.height = 100
        geom_str = "%ix%i+30+30" % (self.width, self.height)
        self.root.geometry(geom_str)

        # SEAL INFO
        self.name_label = tk.Label(self.root, text='Nombre')
        self.name_label.place(x=20, y=10)
        # self.seal_name = tk.StringVar()
        self.name_info = tk.Entry(self.root)  # , textvariable=self.seal_name)
        self.name_info.place(x=70, y=10)

        self.author_label = tk.Label(self.root, text='Autor')
        self.author_label.place(x=20, y=40)
        # self.seal_author = tk.StringVar()
        self.author_info = tk.Entry(self.root)  # ,
textvariable=self.seal_author)
        self.author_info.place(x=70, y=40)


        # OK BUTTON
        self.ok_button = tk.Button(self.root, text='OK',
                                   command=self.on_ok_button)
        self.ok_button.place(x=200, y=20)

    def on_ok_button(self):
        self.db.insert_seal(self.name_info.get(), self.author_info.get())
        self.img_win.save_seal(self.name_info.get())
        self.root.destroy()
```

```
if __name__ == "__main__":
    path = path_to_documents
    walk = os.walk(path)

    doc_paths = []
    for root, dirs, files in walk:
        there_is_any_img = False
        for curr_file in files:
            if curr_file.endswith(".png"):
                there_is_any_img = True

        if there_is_any_img:
            root = root.replace("\\", "/")
            doc_paths.append(root)

    win = WinControlPanel(None, (10, 20, 30, 40), doc_paths)
    while 1:
        win.root.update_idletasks()
        win.root.update()
```

widgets.py

```
"""
Archivo destinado a agrupar elementos de la interfaz que deban ir todos juntos.
Al final tan sólo contiene la lista de sellos incluyendo el menú desplegable,
la lista de éstos, la imagen de muestra correspondiente a dicho sello y el
botón que sirve para cambiar dicha imagen.
"""

import tkinter as tk


class SealsList:
    def __init__(self, master, db):
        self.db = db
        self.x = 0
        self.y = 0
        self.label = tk.Label(master, text='Type:')  # <--LABEL

        self.OPTIONS = [seal.name for seal in self.db.seal_list]
        self.curr_seal_type = tk.StringVar(master)
        self.curr_seal_type.set(self.OPTIONS[0])
        self.option_menu = tk.OptionMenu(master, self.curr_seal_type,
*self.OPTIONS)  # <--OPTION MENU

        self.canvas = tk.Canvas(master, width=600, height=400)  # <--CANVAS
        self.img_route = self.db.seal_list[0].img_route
        self.img_route = self.img_route.replace("\\", "/")
        photo = tk.PhotoImage(file=self.img_route)
        self.seal_img = self.canvas.create_image(0, 0, anchor=tk.NW,
image=photo)
        self.canvas.image = photo

        self.change_button = tk.Button(master, text="Change",
command=self.on_change_item)

    def on_change_item(self):
        index = self.OPTIONS.index(self.curr_seal_type.get())
        # change image
        path = self.db.seal_list[index].img_route
```

```
        path = path.replace("\\", "/")
        photo = tk.PhotoImage(file=path)
        self.canvas.itemconfig(self.seal_img, image=photo)
        self.canvas.image = photo

    def place_items(self):
        self.label.place(x=self.x, y=self.y)
        self.option_menu.place(x=self.x + 35, y=self.y)
        self.canvas.place(x=self.x + 35, y=self.y + 80)
        self.change_button.place(x=self.x + 160, y=self.y+3)
```

database.py

```
import pymysql
import seal

from config import path_to_seals


class Database:
    def __init__(self, db_name, user_name, passwd, tables):
        self.seal_list = []  # list of currently stored seals
        self.db = pymysql.connect("localhost", user_name, passwd, db_name)
        self.cursor = self.db.cursor()
        self.table_names = tables

    def load_seals(self):
        sql = "select * from %s" % self.table_names[0]

        try:
            self.cursor.execute(sql)
            result = self.cursor.fetchall()

            for row in result:
                name = row[0]
                author = row[1]
                route = row[2]
                new_seal = seal.Seal(route, name, author)
                self.seal_list.append(new_seal)
        except RuntimeError:
            print("error: unable to fetch seals data\n")

    def insert_seal(self, name, author):
        path = path_to_seals + name + ".png"
        sql = """insert into %s(nombre, autor, ruta, añadido_manual)
                 values ('%s', '%s', '%s', 1)""" % (self.table_names[0], name,
                                                    author, path)

        try:
            self.cursor.execute(sql)
            self.db.commit()
        except RuntimeError:
            self.db.rollback()
            print("unable to insert data into database\n")

    def insert_document(self, route, seal_name, coords):
        sql = """insert into %s(ruta, sello, coordenadas_x1, coordenadas_y1,
            coordenadas_x2, coordenadas_y2)
            values ('%s', '%s', '%s', '%s', '%s', '%s')"""%
```

```
(self.table_names[1],route, seal_name,
coords[0], coords[1], coords[2], coords[3])

        try:
            self.cursor.execute(sql)
            self.db.commit()
        except RuntimeError:
            self.db.rollback()
            print("unable to insert data into database\n")
```

seal.py

```
class Seal:
    def __init__(self, img_route='', name='', author=''):
        self.img_route = img_route
        self.name = name
        self.author = author
```

Anexo B. Tests de elección de umbral de binarizado

Para calcular el mejor sistema de binarizado se ha diseñado esta aplicación que permite cargar una imagen de prueba y ver en tiempo real el resultado de cada binarizado distinto que se aplique. El sistema permite probar un umbral fijo, el método de Otsu o el método adaptativo mediante media ponderada con kernel gaussiano. Además permite realizar una distorsión gaussiana para eliminar ruidos. El resultado se puede almacenar pulsando la tecla TAB (elegida arbitrariamente) para así tener almacenados resultados y poder comparar los distintos métodos. Pulsar ESC termina la ejecución.

bin_test.py

```
import cv2
import os
from paths import path_to_imgs
from Umbralizacion import Umbralizacion


def do_nothing(x):
    pass

file = path_to_imgs + '1862-L119.M13/109/IMG_0001.png'
img = cv2.imread(file)
nombre = os.path.splitext(os.path.basename(file))[0]

cv2.namedWindow('image_window', cv2.WINDOW_NORMAL)
cv2.namedWindow('control_window', cv2.WINDOW_OPENGL)

METHODLABEL = '1:OTSU-2:FIX-3:ADAP'

cv2.createTrackbar('Vecinity width', 'control_window', 0, 10, do_nothing)
cv2.createTrackbar('Vecinity height', 'control_window', 0, 10, do_nothing)
cv2.createTrackbar('Sigma X', 'control_window', 0, 100, do_nothing)
cv2.createTrackbar('Sigma Y', 'control_window', 0, 100, do_nothing)
cv2.createTrackbar(METHODLABEL, 'control_window', 0, 2, do_nothing)
cv2.createTrackbar('Bin Threshold', 'control_window', 0, 255, do_nothing)
cv2.createTrackbar('SWITCH', 'control_window', 0, 1, do_nothing)

gray_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
filtered_img = gray_img
```

```
bin_img = Umbralizacion.umbralizar_imagen(filtered_img,
Umbralizacion.MetodoUmbralizado.fixed, 180)

while 1:
    cv2.imshow('image_window', bin_img)

    vec_w = cv2.getTrackbarPos('Vecinity width', 'control_window')
    vec_h = cv2.getTrackbarPos('Vecinity height', 'control_window')
    sigX = cv2.getTrackbarPos('Sigma X', 'control_window')
    sigY = cv2.getTrackbarPos('Sigma Y', 'control_window')
    method = cv2.getTrackbarPos(METHODLABEL, 'control_window')
    threshold = cv2.getTrackbarPos('Bin Threshold', 'control_window')
    switch = cv2.getTrackbarPos('SWITCH', 'control_window')

    method_enum = Umbralizacion.MetodoUmbralizado(method)

    kernel = (vec_w*2 + 1, vec_h*2 + 1)
    if switch == 1:
        filtered_img = cv2.GaussianBlur(gray_img, kernel, sigX)
        bin_img = Umbralizacion.umbralizar_imagen(filtered_img, method_enum,
threshold)

    k = cv2.waitKey(1) & 0xFF
    if k == 27:
        break
    elif k == 9:  # TAB key
        filestring = '../%s_met_%d_vec_%d_sig_%d_thr_%d.png' % (nombre, method,
                     vec_w, sigX, threshold)
        # cv2.imwrite(filestring, bin_img)

cv2.destroyAllWindows()
```

Umbralizacion.py

```
import cv2
from enum import Enum


class Umbralizacion:
    class MetodoUmbralizado(Enum):
        otsu = 0
        fixed = 1
        adaptive = 2

    @staticmethod
    def umbralizar_imagen(imagen_grises, metodo_umbralizar, umbral=180):
        imagen_umbralizada = imagen_grises

        if metodo_umbralizar == Umbralizacion.MetodoUmbralizado.otsu:
            ret, imagen_umbralizada = cv2.threshold(imagen_grises, 0, 255,
                                      cv2.THRESH_BINARY + cv2.THRESH_OTSU)
        elif metodo_umbralizar == Umbralizacion.MetodoUmbralizado.fixed:
            ret, imagen_umbralizada = cv2.threshold(imagen_grises, umbral, 255,
                                      cv2.THRESH_BINARY)
        elif metodo_umbralizar == Umbralizacion.MetodoUmbralizado.adaptive:
            imagen_umbralizada = cv2.adaptiveThreshold(imagen_grises, 255,
                cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY, 11, 2)

        return imagen_umbralizada
```

Anexo C. Fase heurística

SellosHeuristica.py

Este archivo incluye todo el código necesario para llevar a cabo esta etapa. En realidad incluye tres clases distintas: la clase *Documento*, que define las variables y comportamientos relativos al documento analizado en cuestión; la clase *Región*, que hace lo propio para cada una de las regiones que se encuentren en el documento y la clase *LineSeparator*, que tiene la finalidad de calcular las coordenadas de la separación entre las líneas de texto en el documento.

La estructura consiste en un objeto de tipo documento que posee una lista de objetos de tipo región. La clase de separación de líneas hace las veces de *wrapper* y por tanto sólo posee métodos estáticos, que son llamados desde la clase documento sin necesidad de crear ningún objeto.

La clase región a su vez posee dos clases: una que se encarga de los test a los que se someten las regiones y otra para los *bounding boxes* de dicha región.

Clase Documento

```
class Documento:
    """
    Clase del documento analizado en cuestión. Sólo va a haber una instancia de
documento en todo momento.
    """
    img = np.array([])
    path = ""
    bin_img = np.array([])
    bin_thresh = 0
    kernel = np.ones((11, 11), np.uint8)
    label_img = np.array([])
    regions = []
    seals = []
    has_2_pages = False
    lines_y = np.array([])

    def __init__(self):
        del self.regions[:]
        del self.seals[:]
        self.lines_y = np.array([[]])

    def load_img(self, path):
        self.path = path
        self.img = cv2.imread(self.path, cv2.IMREAD_GRAYSCALE)

    def get_bin_img(self):
        self.img = cv2.GaussianBlur(self.img, (3, 3), 0)
        self.bin_thresh, self.bin_img = cv2.threshold(self.img, 0, 1,
            cv2.THRESH_BINARY_INV + cv2.THRESH_OTSU)

    def apply_img_corrections(self, img=None):
        if img is None:
            aux_img = self.bin_img
        else:
            aux_img = img
        aux_img = cv2.GaussianBlur(aux_img, (11, 11), 0)
        se1 = cv2.getStructuringElement(cv2.MORPH_RECT, (10, 10))
        se2 = cv2.getStructuringElement(cv2.MORPH_RECT, (4, 4))
        mask = cv2.morphologyEx(aux_img, cv2.MORPH_CLOSE, se1)
        mask = cv2.morphologyEx(mask, cv2.MORPH_OPEN, se2)
```

```
        out = self.img * mask

        if img is None:
            self.bin_img = (out > 5).astype('uint8')
        else:
            return (out > 5).astype('uint8')

    def get_lines(self):
        """
        Calcula la coordenada vertical de la separación en líneas del
        documento.
        :return: Almacena las coordenadas en self.lines_y
        """
        if LineSeparator.has_two_pages(self.bin_img):
            self.has_2_pages = True
            ranges = ((0, self.bin_img.shape[1] // 2),
                      (self.bin_img.shape[1] // 2, self.bin_img.shape[1]))
        else:
            self.has_2_pages = False
            ranges = ((0, self.bin_img.shape[1]),)

        lines_y = []
        for page, rng in enumerate(ranges):
            hist = LineSeparator.proyect(self.bin_img[:, rng[0]:rng[1]],
                axis=LineSeparator.axis["horizontal"])

            if hist.max() == 0:
                continue

            smooth_hist = LineSeparator.savitzky_golay(hist, 51, 3)

            mins = LineSeparator.find_min(smooth_hist)
            min_x = [i[0] for i in mins]
            min_y = [i[1] for i in mins]

            is_out = LineSeparator.is_outlier(np.array(min_y))
            outliers_x = []
            outliers_y = []
            for i, x in enumerate(is_out):
                if x:
                    if hist[min_x[i]] > 0:
                        outliers_x.append(min_x[i])
                        outliers_y.append(min_y[i])
                    else:
                        is_out[i] = False

            for i, xs in enumerate(min_x):
                if not is_out[i]:
                    # cv2.line(self.bin_img, (rng[0], xs), (rng[1], xs), 1, 10)
                    lines_y.append([xs, page])

        self.lines_y = np.array(lines_y)

    def get_regions(self):
        aux_label_img = measure.label(self.bin_img)
        regs = measure.regionprops(aux_label_img)
        aux_label_img = Region.Bbox.eliminar_borde(regions=regs,
            label_image=aux_label_img)
        regs = measure.regionprops(aux_label_img)

        self.label_img = Region.Bbox.reetiquetado(regs, aux_label_img)
        regs = measure.regionprops(self.label_img)
```

```
        i = 0
        del self.regions[:]
        for reg in regs:
            self.regions.append(Region(self, reg.bbox, reg.area, i))
            i += 1

    def elim_self_contain(self):
        """
        Elimina aquellas regiones que se encuentran contenidas dentro de otras
        regiones. Una segunda pasada del detector de colisiones solventaría
        esto, pero allí hay demasiadas regiones aún, y sería muy lento.
        """
        for seal in self.seals:
            seal_bbox = (seal.minr, seal.minc, seal.maxr, seal.maxc)
            j = 0
            while j < len(self.seals):
                bbox2 = (self.seals[j].minr, self.seals[j].minc,
                    self.seals[j].maxr, self.seals[j].maxc)
                if Region.Bbox.colision(seal_bbox, bbox2, 4)
                   and seal != self.seals[j]:
                    self.regions[seal.id].minr =\
                        self.seals[j].minr =\
                        min(seal.minr, self.seals[j].minr)
                    self.regions[seal.id].minc =\
                        self.seals[j].minc =\
                        min(seal.minc, self.seals[j].minc)
                    self.regions[seal.id].maxr =\
                        self.seals[j].maxr =\
                        max(seal.maxr, self.seals[j].maxr)
                    self.regions[seal.id].maxc =\
                        self.seals[j].maxc =\
                        max(seal.maxc, self.seals[j].maxc)
                    self.regions[seal.id].filled_area =\
                        self.seals[j].filled_area =\
                        seal.filled_area + self.seals[j].filled_area
                    del self.seals[j]
                j += 1
```

Clase Region

Esta clase posee dos diccionarios muy interesantes. *settings* permite configurar los parámetros de los test. *active_tests* permite activar o desactivar los distintos tests de manera que se lleven o no a cabo. Puede verse que tanto el test de simetría como el de posición (aquel que rechaza cualquier sello que se encuentre en la mitad inferior del documento), se encuentran desactivados como se explicó con anterioridad.

```
class Region:
    settings = {
        "max_area": 40000,
        "max_aspect_ratio": 3,
        "min_filled_area_ratio": 0.2,
        "max_filled_area_ratio": 0.9,
        "simmetry_ratio_thresh": 0.25,
        "simm_recheck_enlargement_px": 10,
        "simm_recheck_thresh": 0.1,  # cambio porcentual en el umbral de
                                     # binarizacion para probar en el test de
                                     # simetría
    }

    def __init__(self, document, coords, filled_area, reg_id):
        self.minr, self.minc, self.maxr, self.maxc = coords
        self.filled_area = filled_area
```

```
        self.id = reg_id
        self.region_is_seal = False
        self.test = self.Tests(document, self)

    def __eq__(self, other):
        return self.id == other.id

    class Bbox:
        @staticmethod
        def colision(bbox1, bbox2, min_separacion):
            """
            Simple detector de colisiones entre bboxes con una cierta distancia
            umbral
            :param bbox1: -
            :param bbox2: -
            :param min_separacion: distancia umbral dentro de la cual se
                considera colisión igualmente
            :return: True -> Colisionan; False -> No colisionan
            """
            a_minr, a_minc, a_maxr, a_maxc = bbox1
            b_minr, b_minc, b_maxr, b_maxc = bbox2

            a_width = a_maxr - a_minr + min_separacion
            a_height = a_maxc - a_minc + min_separacion
            b_width = b_maxr - b_minr + min_separacion
            b_height = b_maxc - b_minc + min_separacion

            a_x = a_minr + a_width / 2
            a_y = a_minc + a_height / 2
            b_x = b_minr + b_width / 2
            b_y = b_minc + b_height / 2

            return (abs(a_x - b_x) * 2 < a_width + b_width and
                    abs(a_y - b_y) * 2 < a_height + b_height)

        @staticmethod
        def reetiquetado(regions, label_image):
            j = 0
            for region in regions:
                j += 1
                all_other_regions = regions
                for i in range(0, len(all_other_regions)):
                    if (Region.Bbox.colision(region.bbox,
                      all_other_regions[i].bbox, 4) and
                      region.label != all_other_regions[i].label):
                        if region.label < all_other_regions[i].label:
                            for points in all_other_regions[i].coords:
                                label_image[points[0], points[1]] =\
                                    region.label
                                all_other_regions[i].label = region.label
                        else:
                            :
                            for points in region.coords:
                                label_image[points[0], points[1]] =\
                                    all_other_regions[i].label
                                region.label = all_other_regions[i].label

            return label_image


        @staticmethod
        def eliminar_borde(regions, label_image):
```

```
        """
        Toma la imagen de etiquetas original y si encuentra alguna región
        con un área mayor del 40% de la imagen original, la elimina.Esto es
        necesario para ya que en ocasiones las imágenes poseen un borde
        negro que no es deseable considerar como region.Se usa 40% ya que
        algunas imágenes contienen dos págs y dicho borde será un 50%
        aprox. De esta forma se garantiza que el borde cumplirá esta
        condición y se minimiza la posibilidad de eliminar un sello por
        error (es improbable que un sello ocupe más de el 40% de una
        imagen).
        :param regions: Lista de regiones encontradas
        :param label_image: Imagen de etiquetas
        :return: Imagen de etiquetas con el supuesto borde etiquetado como
        fondo (etiqueta = 0).
        """
        for region in regions:
            minr, minc, maxr, maxc = region.bbox
            bbox_height = maxr - minr
            bbox_width = maxc - minc
            img_dims = label_image.shape
            if bbox_height * bbox_width > img_dims[0] * img_dims[1] * 0.3:
                for points in region.coords:
                    label_image[points[0], points[1]] = 0

        return label_image

    @staticmethod
    def detectar_bbox(img_region, coords):
        """
        Readapta los bboxes de las regiones que hayan crecido o encogido
        tras el cambio de threshold (solo útil tras test de simetría)
        :param img_region: recorte de la imagen de la región tras crecer o
         encoger
        :param coords: coordenadas de la img_region en el documento
         original
        :return:tupla(nuevo recorte de la imagen, coords de su nuevo bbox)
        """
        fil, col = img_region.shape
        min_col = 0
        limit_detect = False
        for i in range(0, col):
            for j in range(0, fil):
                if img_region[j, i] != 0:
                    min_col = i
                    limit_detect = True
                    break
            if limit_detect:
                break

        max_col = 0
        limit_detect = False
        for i in range(col-1, -1, -1):
            for j in range(fil-1, -1, -1):
                if img_region[j, i] != 0:
                    max_col = i
                    limit_detect = True
                    break
            if limit_detect:
                break

        min_fil = 0
        limit_detect = False
```

```
            for i in range(0, fil):
                for j in range(0, col):
                    if img_region[i, j] != 0:
                        min_fil = i
                        limit_detect = True
                        break
                if limit_detect:
                    break

            max_fil = 0
            limit_detect = False
            for i in range(fil-1, -1, -1):
                for j in range(col-1, -1, -1):
                    if img_region[i, j] != 0:
                        max_fil = i
                        limit_detect = True
                        break
                if limit_detect:
                    break

            new_coords = np.array([coords[0] + min_fil, coords[0] + max_fil
                coords[2] + min_col, coords[2] + max_col])

            return img_region[min_fil:max_fil, min_col:max_col], new_coords

    class Tests:
        active_tests = {
            "area": True,
            "aspect_ratio": True,
            "filled_area": True,
            "simmetry": False,
            "position": False,
            "between_lines": True,
        }

        def __init__(self, document, region):
            self.passed_tests = {
                "area": False,
                "aspect_ratio": False,
                "filled_area": False,
                "simmetry": False,
                "position": False,
                "between_lines": False,
            }
            self.document = document
            self.region = region

        def area(self):
            """
            Test para eliminar regiones pequeñas que solo son ruido
            """
            width = self.region.maxr - self.region.minr
            height = self.region.maxc - self.region.minc
            if width * height <= Region.settings.get("max_area"):
                self.passed_tests.update({"area": False})
            else:
                self.passed_tests.update({"area": True})

        def aspect_ratio(self):
            """
            Las regiones que solo contienen palabras tienden a ser muy
            alargadas. Muchas regiones compuestas por ruido (pliegues de papel,
```

```
        manchas...) tienden a ser muy altas. Este test tiene por finalidad
        eliminar ese tipo de regiones.
        """
        width = self.region.maxr - self.region.minr
        height = self.region.maxc - self.region.minc
        if (width / height > Region.settings.get("max_aspect_ratio") or
           height / width > Region.settings.get("max_aspect_ratio")):
            self.passed_tests.update({"aspect_ratio": False})
        else:
            self.passed_tests.update({"aspect_ratio": True})

    def filled_area_ratio(self):
        """
        Comprueba si la region tiene un ratio de relleno suficientemente
        alto.
        """
        width = self.region.maxr - self.region.minr
        height = self.region.maxc - self.region.minc
        filled_area = self.region.filled_area
        ratio = float(filled_area) / (width * height)
        # Umbral máximo útil para eliminar regiones completamente negras
        if (ratio < Region.settings.get("min_filled_area_ratio") or
           ratio > Region.settings.get("max_filled_area_ratio")):
            self.passed_tests.update({"filled_area": False})
        else:
            self.passed_tests.update({"filled_area": True})

    @staticmethod
    def symmetry_ratio(bin_seal):
        """
        Medidor de ratio de simetría. Es importante distinguir entre
        imágenes con ancho par o impar a la hora de plegarlas sobre sí.
        :param bin_seal: imagen binarizada de la región a testear
        :return: ratio de simetría. NOTA: Cuanto más cerca de 0, más
        simétrico. Podría redefinirse como 1-(ratio actual) para cambiar
        eso
        """
        fil, col = bin_seal.shape
        if col <= 1 or fil <= 1:
            return float('inf')
        # TODO: Tranformar en raise + un error.

        if col % 2 != 0:
            img_right = bin_seal[0:fil,          else:
            img_right = bin_seal[0:fil, int(col / 2):col]
        img_left = bin_seal[0:fil, 0:int(col / 2)]

        flip_left_img = cv2.flip(img_left, 1)  # 1 means y axis
        subtracted_img = abs((flip_left_img / 255 - img_right / 255)) * 255

        sub_area = np.sum(subtracted_img)
        ref_area = (np.sum(img_left) + np.sum(img_right)) / 2.0

        ratio = sub_area / ref_area
        return ratio

    def symmetry(self):
        minr = self.region.minr
        maxr = self.region.maxr
        minc = self.region.minc
        maxc = self.region.maxc
```

```
            seal_img = self.document.bin_img[minr:maxr, minc:maxc]
            ratio = self.symmetry_ratio(seal_img)

            if ratio < Region.settings.get("simmetry_ratio_thresh"):
                thickness = Region.settings.get("simm_recheck_enlargement_px")
                enlarged_img = self.document.img[abs(minr - thickness):maxr +
                    thickness, abs(minc - thickness):maxc + thickness]
                enlarged_img_coords =\
                    [minr - thickness, maxr + thickness,
                     minc - thickness, maxc + thickness]

                dark_thresh = self.document.bin_thresh *
                    (1 + Region.settings.get("simm_recheck_thresh"))
                darker_seal_img =\
                    (enlarged_img < dark_thresh).astype('uint8') * 255
                darker_seal_img =\
                    self.document.apply_img_corrections(darker_seal_img)
                cropped_img, new_coords =\
                    Region.Bbox.detectar_bbox(darker_seal_img,
                    enlarged_img_coords)
                ratio = self.symmetry_ratio(cropped_img)

                if ratio < Region.settings.get("simmetry_ratio_thresh"):
                    light_thresh = (self.document.bin_thresh *
                        (1 - Region.settings.get("simm_recheck_thresh")))
                    lighter_seal_img =\
                        (enlarged_img < light_thresh).astype('uint8') * 255
                    lighter_seal_img =\
                        self.document.apply_img_corrections(lighter_seal_img)
                    cropped_img, new_coords =\
                        Region.Bbox.detectar_bbox(lighter_seal_img,
                                                  enlarged_img_coords)
                    ratio = self.symmetry_ratio(cropped_img)

                    if ratio < Region.settings.get("simmetry_ratio_thresh"):
                        self.passed_tests.update({"filled_area": False})
                    else:
                        self.passed_tests.update({"filled_area": True})
                        self.region.minr = new_coords[0]
                        self.region.maxr = new_coords[1]
                        self.region.minc = new_coords[2]
                        self.region.maxc = new_coords[3]
                else:
                    self.passed_tests.update({"filled_area": True})
                    self.region.minr = new_coords[0]
                    self.region.maxr = new_coords[1]
                    self.region.minc = new_coords[2]
                    self.region.maxc = new_coords[3]

        def position(self):
            height = (self.region.maxr + self.region.minr) / 2

            docr = self.document.img.shape[0]

            if height > docr / 2:
                self.passed_tests.update({"position": False})
            else:
                self.passed_tests.update({"position": True})

        def is_between_lines(self):
            """
            Tests if region contains more than one line. In that case, region
```

```
        is not a seal.

        NOTE: Page is stored as the second value in lines_y where
        --> 0->1st page; 1->2nd page
        """
        if not self.document.has_2_pages:
            lines_y = self.document.lines_y
        else:
            page =\
               int(self.region.minc >int(self.document.bin_img.shape[1]/2))
            line_list = []
            for ly in self.document.lines_y:
                if ly[1] == page:
                    line_list.append(ly)
            lines_y = np.array(line_list)

        height = self.region.maxr - self.region.minr
        nearest_higher_index =\
            np.searchsorted(lines_y[:, 0],
                            self.region.minr + int(height/3)).item()
        if nearest_higher_index < len(lines_y):
            nearest_higher = lines_y[nearest_higher_index, 0]
        else:
            nearest_higher = 0  # if we reach this condition, the test
                                # should get passed anyway

        if (self.region.minr + int(height/3) < nearest_higher <
            self.region.maxr - int(height/3)):
            self.passed_tests.update({"between_lines": False})
        else:
            self.passed_tests.update({"between_lines": True})

    def apply_active_tests(self):
        """
        Aplica todo test que se encuentre activo. Si la región superase
        todos los test, se añadiría a la lista de sellos
        """
        self.region.region_is_seal = True
        if self.region.test.active_tests.get("area") is True:
            self.region.test.area()
            self.region.region_is_seal *=\
                self.region.test.passed_tests.get("area")
        if self.region.test.active_tests.get("aspect_ratio") is True:
            self.region.test.aspect_ratio()
            self.region.region_is_seal *=\
                self.region.test.passed_tests.get("aspect_ratio")
        if self.region.test.active_tests.get("filled_area") is True:
            self.region.test.filled_area_ratio()
            self.region.region_is_seal *=\
                self.region.test.passed_tests.get("filled_area")
        if self.region.test.active_tests.get("simmetry") is True:
            self.region.test.symmetry()
            self.region.region_is_seal *=\
                self.region.test.passed_tests.get("simmetry")
        if self.region.test.active_tests.get("position") is True:
            self.region.test.position()
            self.region.region_is_seal *=\
                self.region.test.passed_tests.get("position")
        if self.region.test.active_tests.get("between_lines") is True:
            self.region.test.is_between_lines()
            self.region.region_is_seal *=\
                self.region.test.passed_tests.get("between_lines")
```

```
            if self.region.region_is_seal:
                self.document.seals.append(self.region)
```

Clase LineSeparator

```
class LineSeparator:
    settings = {
        "min_lin_dist_px": 60
    }

    axis = {
        "vertical": 0,
        "horizontal": 1,
    }

    @staticmethod
    def proyect(bin_img, axis):
        """
        Cuenta el número de pixels tras proyectar la imagen en la dirección
        especificada
        :param bin_img: -
        :param axis: eje sobre el que proyectar. Utilícese el diccionario axis
        :return:
        """
        if np.array_equal(bin_img, bin_img.astype(bool)):
            hist = np.sum(bin_img, axis)
        else:
            raise NameError("Image passed to LineSeparator is not binary")

        return hist

    @staticmethod
    def longest_streak(a):
        """
        Encuentra la racha más larga en ya sea un array de numpy o una lista
        :param a: lista o array de numpy
        :return: racha más larga encontrada
        """
        lst = []
        for n, c in groupby(a):
            streak = list(c)
            if len(streak) > len(lst):
                lst = streak

        return lst

    @staticmethod
    def has_two_pages(bin_img):
        """
        Comprueba si la racha más larga encontrada contiene solo ceros y además
        es mayor de 1/7 del ancho total de la imagen del documento
        :param bin_img: Imagen binaria del documento para computer si está
        compuesta o no de dos páginas
        :return: True si tiene dos páginas, False en caso contrario. Se basa en
        la hipótesis de que ningún documento poseerá más de dos páginas.
        """
        first_col = int(bin_img.shape[1] / 3)
        last_col = int(2 * bin_img.shape[1] / 3)
        hist = LineSeparator.proyect(bin_img[:, first_col:last_col],
                                     LineSeparator.axis["vertical"])
```

```
        for i, el in enumerate(hist):
            if el <= 10:
                hist[i] = 0

        max_streak = LineSeparator.longest_streak(hist)
        if len(max_streak) > int(bin_img.shape[1] / 40):
            return True
        else:
            return False

    @staticmethod
    def savitzky_golay(y, window_size, order, deriv=0, rate=1):
        """Suaviza (y opcionalmente deriva) datos mediante un filtro de
Savitzky-Golay.
        El filtro Savitzky-Golay ruidos de alta frecuencia.
        Posee la ventajad de preserver la forma y características de la imagen
original major que otros enfoques de filtrado como las técnicas basadas en
medias móviles
        Parámetros
        ----------
        y : array_like, shape (N,)
            Valores de la señal.
        window_size : int
            Dimensión de la ventana. Debe ser un entero impar.
        order : int
            Orden del polinomio utilizado para el filtrado.
            Debe ser menor que `window_size` - 1.
        deriv: int
            Orden de la derivada a computar (por defecto = 0 tan solo suavizar)
        Returns
        -------
        ys : ndarray, shape (N)
            Señal suavizada (o su n-ésima derivada).
        Notas
        -----
        La idea principal tras este enfoque consiste en realizar, para cada
        punto, un ajuste mediante mínimos cuadrados a un polinomio de orden
        elevado sobre una vecindad de dimensión impar centrada en el punto
        Referencias
        ----------
        .. [1] A. Savitzky, M. J. E. Golay, Smoothing and Differentiation of
           Data by Simplified Least Squares Procedures. Analytical
           Chemistry, 1964, 36 (8), pp 1627-1639.
        .. [2] Numerical Recipes 3rd Edition: The Art of Scientific Computing
           W.H. Press, S.A. Teukolsky, W.T. Vetterling, B.P. Flannery
           Cambridge University Press ISBN-13: 9780521880688
        """
        from math import factorial

        try:
            window_size = np.abs(np.int(window_size))
            order = np.abs(np.int(order))
        except ValueError as msg:
            raise ValueError("window_size and order have to be of type int")
        if window_size % 2 != 1 or window_size < 1:
            raise TypeError("window_size size must be a positive odd number")
        if window_size < order + 2:
            raise TypeError("window_size is too small for the polynomials
order")
        order_range = range(order + 1)
        half_window = (window_size - 1) // 2
        # precompute coefficients
```

```
        b = np.mat([[k ** i for i in order_range] for k in range(-half_window,
half_window + 1)])
        m = np.linalg.pinv(b).A[deriv] * rate ** deriv * factorial(deriv)
        # pad the signal at the extremes with
        # values taken from the signal itself
        firstvals = y[0] - np.abs(y[1:half_window + 1][::-1] - y[0])
        lastvals = y[-1] + np.abs(y[-half_window - 1:-1][::-1] - y[-1])
        y = np.concatenate((firstvals, y, lastvals))
        return np.convolve(m[::-1], y, mode='valid')

    @staticmethod
    def find_min(a):
        """
        Encontrar el mínimos en array 'a'. Si entre mínimo y mínimo hay menos
de min_lin_dist_px píxeles se ignora el
        más grande de los dos.
        :param a: Array en el cual encontrar los mínimos locales.
        :return: Índice en el cual se encuentran los mínimos
        """
        # Tomar todos los puntos menores que sus dos puntos contiguos
        is_min = np.r_[True, a[1:] < a[:-1]] & np.r_[a[:-1] < a[1:], True]
        min_points = []
        true_mins = []
        min_dist = LineSeparator.settings.get("min_lin_dist_px")

        for i, value in enumerate(a):
            if is_min[i]:
                min_points.append((i, value))

        min_points.sort(key=lambda p: p[1])  # function lambda empleada para
                                             # informar a sort() sobre qué
                                             # componente de la tupla emplear
                                             # para la ordenación
        if len(min_points) > 0:
            true_mins.append(min_points[0])

            for point in min_points[1:]:
                too_close = False
                for true_min in true_mins:
                    dist = abs(point[0] - true_min[0])
                    if dist < min_dist:
                        too_close = True

                if not too_close:
                    true_mins.append(point)

        return true_mins

    @staticmethod
    def is_outlier(points, thresh=2.0035):
        """
        Devuelve un array booleano con True en aquellas componentes que sean
        outliers y False en caso de que no lo sean.
        Parámetros:
        -----------
            points : Array de tamaño (observaciones x dimension de
                     observaciones)
            thresh : z-puntuación modificada para emplear como umbral. Aquellas
                observaciones con una z-puntuación (basada en la desviación
                mediana absoluta) mayores que este valor, serán clasificadas
como outliers
                NOTA: El valor se ha calculado como la media entre el mayor y
```

```
            menor valor que filtraba a la perfección para una imagen de
prueba.

        Returns:
        --------
            mask : Array de dimensión igual al número de observaciones
        References:
        ----------
            Boris Iglewicz and David Hoaglin (1993), "Volume 16: How to Detect
            and Handle Outliers", The ASQC Basic References in Quality Control:
            Statistical Techniques, Edward F. Mykytka, Ph.D., Editor.
        """
        if len(points.shape) == 1:
            points = points[:, None]
        median = np.median(points, axis=0)
        diff = np.sum((points - median) ** 2, axis=-1)
        diff = np.sqrt(diff)
        med_abs_deviation = np.median(diff)

        if med_abs_deviation == 0:
            return np.ones(len(points), dtype=np.bool)
        else:
            modified_z_score = 0.6745 * diff / med_abs_deviation
            return modified_z_score > thresh
```

sellos_heuristica_secuencial.py

Ahora que disponemos de todas las herramientas necesarias para llevar a cabo esta etapa, tan solo nos falta utilizarlas. A este *script* se le pasa la ruta donde están las imágenes y va cargando una a una. Para cada una de ellas, crea un objeto documento y llama a todas las funciones de esta clase definidas con anterioridad, con el objetivo de hallar si en este documento existiera alguna región con las características de un sello. Si alguna región se determinase que es un sello, se guarda en la carpeta especificada un recorte de la imagen con las coordenadas de la región y se añade una nueva entrada a la base de datos de prototipos de sellos con la ruta de la imagen y su identificador.

```
import os
import numpy as np
import cv2

from Database import DatabaseHeur
import SellosHeuristica as Sh
import paths


path = paths.path_to_imgs
walk = os.walk(path)
db = DatabaseHeur('docs_osborne', 'testuser', 'test123', 'heur_results6')

i = 0
for root, dirs, files in walk:
    for curr_file in files:
        if not curr_file.endswith(".png") or curr_file.startswith("."):
            continue
        img_path = root + '/' + curr_file
        print(img_path)
        documento = Sh.Documento()
        documento.load_img(img_path)
        documento.get_bin_img()
        documento.apply_img_corrections()
        documento.get_lines()
        documento.get_regions()
```

```
        for region in documento.regions:
            region.test.apply_active_tests()
        documento.elim_self_contain()

        for seal in documento.seals:
            seal_img = documento.img[seal.minr:seal.maxr, seal.minc:seal.maxc]

            width = seal.maxr - seal.minr
            height = seal.maxc - seal.minc
            cv2.imwrite(paths.path_to_heur + '/' + str(i) + curr_file,
                        seal_img)
            db.insert_results({
                "path": root.replace("\\", "/"),
                "reg_id": str(i),
            })
            i += 1
```

Paths.py

Para simplificar el poder trabajar en distintos entornos, se ha creado este archivo que simplemente almacena las distintas rutas necesarias para el funcionamiento del resto de scripts. Este archivo es **común para ambas fases del algoritmo**. Un ejemplo de este archivo podría ser:

```
path_to_imgs = 'C:/Users/usuario/Desktop/Documents/'
path_to_prototypes = 'C:/Users/usuario/Desktop/proto_base/'
path_to_heur = 'C:/Users/usuario/Desktop/heur/'
```

Es imprescindible que las tres variables estén definidas. La primera localiza la carpeta raíz de las imágenes de documentos. La segunda, los prototipos de sellos una vez que se han eliminado manualmente los falsos positivos. La tercera, es donde la fase heurística almacena todos los candidatos a sellos encontrados.

Database.py

Este archivo NO es el mismo que el mencionado en el anexo B, ya que pertenecen a proyectos (y entornos virtuales separados). Sí que es común con la segunda fase del algoritmo (anexo D), al igual que el anterior.

```
import pymysql


class Database:
    sql = ""

    def __init__(self, db_name, user_name, passwd, tables):
        self.db = pymysql.connect("localhost", user_name, passwd, db_name)
        self.cursor = self.db.cursor()
        self.table_names = tables

    def run_query(self, query):
        try:
            self.cursor.execute(query)
            self.db.commit()
        except RuntimeError:
            self.db.rollback()
            print("unable to insert data into database\n")

    def insert_results(self, params):
        raise NotImplementedError("Method not implemented")


class DatabaseFeatures(Database):
    sql = """insert into %s(ruta, sello, coordenadas_x1, coordenadas_y1,
coordenadas_x2, coordenadas_y2, ratio) values ('%s', '%s', '%s', '%s', '%s',
```

```
'%s', '%s')"""

    def insert_results(self, params):
        path = params.get("path")
        coords = params.get("coords")
        found_seal_name = params.get("found_seal_name")
        max_ratio = params.get("max_ratio")
        query = self.sql % (self.table_names, path, found_seal_name,
                            coords[0][0], coords[0][1], coords[1][0],
                            coords[1][1], max_ratio)

        super().run_query(query)


class DatabaseHeur(Database):
    sql = "insert into %s(ruta, region_id) values ('%s', '%s')"

    def insert_results(self, params):
        path = params.get("path")
        reg_id = params.get("reg_id")
        query = self.sql % (self.table_names, path, reg_id)

        super().run_query(query)
```

Anexo D. Fase de Extracción de Características.

Primero es necesario crear una base de datos con las características extraídas de los prototipos de sellos encontrados y luego utilizar esta información para compararla con los documentos.

FeaturesIO.py

Se encarga de lidiar con la extracción y almacenamiento de características, además de su posterior lectura.

```
import cv2


class FeaturesIO:
    @staticmethod
    def write_features_to_file(filename, locs, desc, shape):
        np.savez(filename, np.hstack((locs, desc)), shape)

    @staticmethod
    def pack_keypoint(keypoints, descriptors):
        kpts = np.array([[kp.pt[0], kp.pt[1], kp.size, kp.angle,
                          kp.response, kp.octave,
                          kp.class_id]
                         for kp in keypoints])
        desc = np.array(descriptors)
        return kpts, desc

    @staticmethod
    def unpack_keypoint(file):
        seals_kps = []
        seals_des = []
        array = file['arr_0']
        num_elements = len(array)
        for i in range(0, int(num_elements/2)):
            kpts = array[i]
```

```
            keypoints = [cv2.KeyPoint(x, y, _size, _angle, _response,
                int(_octave), int(_class_id)) for x, y, _size, _angle,
                _response, _octave, _class_id in list(kpts)]
            seals_kps.append(keypoints)

        for i in range(int(num_elements/2), num_elements):
            desc = np.array(array[i]).astype(np.float32)  # float32 for surf;
uint8 for ORB
            seals_des.append(desc)

        seal_shps = file['arr_1']

        return seals_kps, seals_des, seal_shps

    @staticmethod
    def process_and_save(path, resultname, detector):
        walk = os.walk(path)
        all_k = []
        all_d = []
        all_s = []
        for root, dirs, files in walk:
            for curr_file in files:
                if (not curr_file.endswith(".png")
                    or curr_file == "no_seal.png"):
                    continue

                # print(curr_file)
                img = cv2.imread(path + '/' + curr_file, 0)

                k = detector.detect(img, None)
                if len(k) > 0:
                    k, des = detector.compute(img, k)
                else:
                    des = []
                k, des = FeaturesIO.pack_keypoint(k, des)  #
                all_k.append(k.tolist())
                all_d.append(des.tolist())
                all_s.append(img.shape)

        FeaturesIO.write_features_to_file(resultname, all_k, all_d, all_s)

    @staticmethod
    def load_features(filename):
        file = np.load(filename)
        kp, des, shps = FeaturesIO.unpack_keypoint(file)
        return kp, des, shps
```

FeaturesDetector.py

Cumple la finalidad de facilitar la configuración del detector de características

```
import cv2.xfeatures2d as xf


def create_detector():
    surf = xf.SURF_create(hessianThreshold=400, upright=True, extended=True)
    return surf
```

creacion_base_datos_sellos.py

Técnicamente no es una base de datos, tan solo una serialización. Extrae y almacena en un archivo las características extraídas de los prototipos de sellos.

```
from FeaturesIO import FeaturesIO as FtIO
import FeaturesDetector
import paths

path = paths.path_to_prototypes

detector = FeaturesDetector.create_detector()
FtIO.process_and_save(path, "car_sellos", detector)
```

EvidenceMatrix.py

Este archivo ha sido explicado en profundidad en el capítulo 6.

```
import numpy as np


class EvidenceMatrix:
    DIVISION_SIZE = 10
    SEAL_DIMENSION = 300  # seals are about 300x300px

    def __init__(self, shape):
        fils, cols = shape
        self.evidence_matrix = np.array([])
        self.evidence_matrix.resize((fils / EvidenceMatrix.DIVISION_SIZE,
                                     cols / EvidenceMatrix.DIVISION_SIZE))
        self.evidence_matrix.fill(0)

    def calc_occurrences(self, keypoints):
        for kp in keypoints:
            # points are x, y instead of rows and columns
            normal = (int(kp.pt[1] / EvidenceMatrix.DIVISION_SIZE),
int(kp.pt[0] / EvidenceMatrix.DIVISION_SIZE))
            self.evidence_matrix[normal[0], normal[1]] += 1
```

EliminacionSellos.py

Detecta el tipo de sello y su posición. Además, permite borrarlo del documento con el objetivo de facilitar otras tareas de visión artificial.

```
import cv2
import numpy as np
from FeaturesIO import FeaturesIO

import EvidenceMatrix as em
import FeaturesDetector


class EliminacionSellos:
    doc_img = 0
    doc_kps = []
```

```
    doc_des = []
    kps_saved = []
    desc_saved = []
    seals_dims = []
    surf = FeaturesDetector.create_detector()

    def __init__(self, img, index):
        EliminacionSellos.doc_img = img.copy()
        self.kp_matched = []
        self.evidence_matrix = em.EvidenceMatrix(img.shape)
        self.position = (0, 0)  # position is (rows, cols), therefore, (y, x)
        self.max_occurrences = 0
        self.index = index
        self.total_matches = 0

    def get_keypoints_from_db(self, path):
        (EliminacionSellos.kps_saved, EliminacionSellos.desc_saved,
            EliminacionSellos.seals_dims)\
            = FeaturesIO.load_features(path)

    def get_document_features(self):
        EliminacionSellos.doc_kps, EliminacionSellos.doc_des =\
            EliminacionSellos.surf.detectAndCompute(EliminacionSellos.doc_img,
                                                    None)

    def get_matched_keypoints(self):
        # FLANN parameters
        FLANN_INDEX_KDTREE = 0
        index_params = dict(algorithm=FLANN_INDEX_KDTREE, trees=5)
        search_params = dict(checks=50)  # or pass empty dictionary

        flann = cv2.FlannBasedMatcher(index_params, search_params)

        aux_kp = []

        matches = flann.knnMatch(EliminacionSellos.desc_saved[self.index],
                                 EliminacionSellos.doc_des, k=2)
        for j, (m, n) in enumerate(matches):
            if m.distance < 0.7 * n.distance:
                aux_kp.append(EliminacionSellos.doc_kps[m.trainIdx])

        self.kp_matched = aux_kp
        self.total_matches = len(self.kp_matched)

    def compute_evidence_matrix(self):
        self.evidence_matrix.calc_occurrences(self.kp_matched)

    def compute_position_and_max_occurences(self):
        num_cells = int(self.evidence_matrix.SEAL_DIMENSION /
self.evidence_matrix.DIVISION_SIZE)
        kernel = np.ones((num_cells, num_cells))

        occurrences =\
            cv2.filter2D(self.evidence_matrix.evidence_matrix, -1, kernel)
        self.max_occurrences = np.amax(occurrences)
        max_index = np.where(occurrences == self.max_occurrences)
        avg_index = (np.average(max_index[0]), np.average(max_index[1]))

        final_coords = (int(avg_index[0].item()), int(avg_index[1].item()))

        self.position = (final_coords[0] * self.evidence_matrix.DIVISION_SIZE,
                         final_coords[1] * self.evidence_matrix.DIVISION_SIZE)
```

```
    def remove_seal(self):
        cv2.circle(EliminacionSellos.doc_img,
                   (self.position[1], self.position[0]), 200, (255, 0, 255), -1)
```

FuncionCaracteristicasCompleta.py

Contiene la función que recopila todas las herramientas definidas con anterioridad. Permite encapsular esta fase del algoritmo para ser utilizada sin necesidad de conocer su funcionamiento.

```
import EliminacionSellos as ElimSe


def detectar_sello(img, num_elements):
    elim_sellos = []

    for i in range(0, num_elements):
        elim_sellos.append(ElimSe.EliminacionSellos(img, i))

    elim_sellos[0].get_keypoints_from_db('car_sellos.npz')
    elim_sellos[0].get_document_features()

    real_seal = -1
    max_ratio = 0
    for i in range(0, num_elements):
        elim_sellos[i].get_matched_keypoints()
        elim_sellos[i].compute_evidence_matrix()
        elim_sellos[i].compute_position_and_max_occurences()
        if elim_sellos[i].total_matches > 0:
            if (elim_sellos[i].max_occurrences / elim_sellos[i].total_matches
                > max_ratio):
                max_ratio =\
                  elim_sellos[i].max_occurrences / elim_sellos[i].total_matches
                real_seal = i


    elim_sellos[real_seal].remove_seal()

    center_coords = elim_sellos[real_seal].position  # (row,col) = (y,x)
    real_seal_height, real_seal_width =\
        ElimSe.EliminacionSellos.seals_dims[real_seal]
    div_size = elim_sellos[real_seal].evidence_matrix.DIVISION_SIZE

    pt1 = (int(center_coords[1] * div_size - real_seal_width / 2),
           int(center_coords[0] * div_size - real_seal_height / 2))
    pt2 = (int(center_coords[1] * div_size + real_seal_width / 2),
           int(center_coords[0] * div_size + real_seal_height / 2))

    corner_coords = [pt1, pt2]


    return elim_sellos[real_seal].doc_img, corner_coords, real_seal, max_ratio
```

detectar_sellos_main.py

```
import cv2
import os
import numpy as np
from FuncionSellosCompleta import detectar_sello
from Database import DatabaseFeatures
import paths

path = paths.path_to_imgs
walk = os.walk(path)
db = DatabaseFeatures('docs_osborne', 'testuser', 'test123', 'results8')

seal_string = ['Corona Horizontal', 'Escudo 4 regiones', 'Jorge Muller',
               'Leon Unicornio Postal', 'OC Corona Diagonal',
               'Oxforshire Infantry', 'O Doble C', 'O con T',
               'Moneda O doble C', 'MSA', 'Grifos', 'Dedal MSE',
               'TO Marrón', 'O T casi I', 'AOC', 'Viceconsulado Imp de Rusia']

file = np.load('car_sellos.npz')
num_elements = len(file['arr_1'])
file.close()

for root, dirs, files in walk:
    max_ratio = 0
    curr_name = ''
    max_coords = ([0, 0], [0, 0])
    there_is_any_image = False
    for curr_file in files:
        if curr_file.endswith('.png'):
            there_is_any_image = True
            img = cv2.imread(root + '/' + curr_file, 0)
            img2, coords, seal_number, ratio = detectar_sello(img,
num_elements)

            if ratio > max_ratio:
                max_ratio = ratio
                curr_name = seal_string[seal_number]
                max_coords = coords

    if there_is_any_image:
        path_to_save = root.replace("\\", "/")
        db.insert_results({
            "path": path_to_save,
            "coords": max_coords,
            "found_seal_name": curr_name,
            "max_ratio": max_ratio,
        })
```

Además de estos archivos, esta fase utiliza paths.py y Database.py, incluídos en el anexo C.

REFERENCIAS

BIBLIOGRAPHY

[1] Fundación Osborne. Fundación Osborne. [Online]. http://www.osborne.es/sobre-nosotros/responsabilidad-social-corporativa/

[2] Fundación Telefónica. Fundación Telefónica. [Online]. http://www.fundaciontelefonica.com/conocenos/la-mision/

[3] Fundación SEPI. Fundación SEPI. [Online]. https://www.fundacionsepi.es/conozcanos/presentacion.asp

[4] Partha Pratim Roy, Umapada Pal, and Josep Lladós, "Seal detection and recognition : An approach for document indexing," *10th International Conference on Document Analysis and Recognition*, pp. 101-105, 2009.

[5] Guangyu Zhu, Yefeng Zheng, David Doermann, and Stefan Jaeger, "Signature Detection and Matching for Document Image Retrieval," *EEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE, VOL. 31, NO. 11*, pp. 2015-2031, 2009.

[6] Amit Vijay Nandedkar, Jayanta Mukhopadhyay, and Shamik Sural, "Text-Graphics Separation to Detect Logo and Stamp from Color Document Images: A Spectral Approach," *13th International Conference on Document Analysis and Recognition (ICDAR)*, pp. 571-575, 2015.

[7] Noboyuki Otsu, "A Threshold Selection Method from Gray-Level Histograms," *IEEE TRANSACTIONS ON SYSTEMS, MAN, AND CYBERNETICS, VOL. SMC-9, NO. 1*, pp. 62-66, 1979.

[8] T. Chattopadhyay, V. Ramu Reddy, and Utpal Garain, "Automatic Selection of Binarization Method for Robust OCR," in *12th International Conference on Document Analysis and Recognition*, 2013, pp. 1170-1174.

[9] José Luis Guiñón, Emma Ortega, José García-Antón, and Valentín Pérez-Herranz, "Moving Average and Savitzki-Golay Smoothing Filters Using Mathcad," in *International Conference on Engineering Education – ICEE*, Coimbra, Portugal, 2007.

[10] A. Saviztky and M. J. E. Golay, *Smoothing and Differentiation of Data by Simplified Least Squares Procedures.*: Analytical Chemistry, 1964.

[11] Boris Iglewics and David Hoaglin, *Volumen 16: How to Detect and Handle Outliers. The ASQC Basic References in Quality Control: Statistical Techniques*, Ph.D. Edward F. Mykytka, Ed., 1993.

[12] David G. Lowe, "Distinctive Image Features from Scale-Invariant Keypoints," *International Journal of Computer Vision*, pp. 91-110, 2004.

[13] C. Harris and M. Stephen, "Acombined corner and edge detector," in *Fourth Alvey Vision Conference*, Manchester, UK, 1988, pp. 147–151.

[14] Jianbo Shi and Carlo Tomasi, "Good Features to Track," in *IEEE Conference on Computer Vision and*

Pattern Recognition (CVPR94), Seattle, 1994.

[15] Alexander Mordvintsev and Abid K. (2013) OpenCV-Python Tutorials. [Online]. http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_table_of_contents_feature2d/py_table_of_contents_feature2d.html

[16] OpenCV Organization. [Online]. http://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_table_of_contents_feature2d/py_table_of_contents_feature2d.html

[17] Editores anónimos de wikipedia. Laplaciano de Gauss - Wikipedia. [Online]. https://en.wikipedia.org/wiki/Blob_detection#The_Laplacian_of_Gaussian

[18] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool, "SURF: Speeded Up Robust Features," 2008.

[19] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski, "ORB: an efficient alternative to SIFT or SURF," *2011 IEEE International Conference on Computer Vision (ICCV)*, 2011.

[20] Taizo KAMESHIRO, Takashi HIRANO, Yasuhiro OKADA, and Fumio YODA, "A Document Retrieval Method from Handwritten Characters Based on OCR and Character Shape Information," in *Sixth International Conference on Document Analysis and Recognition*, 2001.

[21] Avery Li-Chung Wang, "An Industrial-Strength Audio Search Algorithm," 2002.

[22] Jared Hopkins and Tim L. Andersen, "A Fourier-descriptor-based character recognition engine implemented under the Gamera open-source document-processing framework," in *Document Recognition and Retrieval XII*, San Jose, CA, USA, 2005.