

# Reconocimiento e identificación de cartas con diferentes técnicas de matching

Guillermo Diz Gil

27 de junio de 2022

## Resumen

El reconocimiento de una baraja de cartas en una imagen digital es de potencial utilidad para juegos en línea, asistentes virtuales, etc. En este artículo, trataremos el reconocimiento e identificación de una o más cartas en una imagen a través de dos enfoques, el *feature matching* y el *template matching*, y estableceremos semejanzas y diferencias entre ellos a través de una implementación práctica.

**Palabras clave:** cartas, reconocimiento, identificación, feature, template

## 1. Introducción

Cuando hablamos de reconocimiento e identificación de cartas, hablamos de dos fases con diferente aplicación: el reconocimiento consiste en la aplicación de una o más técnicas de tratamiento de imágenes digitales que nos permitan reconocer y aislar una carta de una imagen realizando transformaciones en ella, como binarización, detección de contornos, transformaciones afines, etc. La identificación de una carta es la fase que sigue al reconocimiento, y consiste en aplicar técnicas de comparación de imágenes que nos permitan, dadas ciertas características de la carta, establecer una relación entre ella y una carta ya conocida, con la intención de averiguar, por ejemplo, su palo y su número.

Para la explicación de la metodología usaremos una baraja de cartas francesa estándar de 52 cartas, con 13 cartas por cada palo y cada una numerada del 1 al 13, nombrándose el 11 como J, el 12 como Q y el 13 como K. No obstante, la metodología que se describirá a continuación puede ser aplicada, sin pérdida de eficiencia y robustez, a otras barajas de cartas, siempre y cuando cumplan una serie de requisitos.

El objetivo final será establecer una separación entre las responsabilidades de las diferentes fases que intervienen en el reconocimiento e identificación de cartas, estableciendo un marco de interfaces comunes para que diferentes algoritmos puedan operar entre sí, y ser reemplazados por otros que cumplan la misma función aunque realicen su trabajo de forma distinta. La separación de responsabilidades y el principio de única responsabilidad son dos principios muy comúnmente aplicados en la industria de la ingeniería del software ya que mejoran la mantenibilidad y legibilidad de un procedimiento complejo, como es este caso.

## 2. Planteamiento teórico

El planteamiento teórico de este artículo se basa, en líneas generales, en el trabajo previo realizado por Dan Snyder [1]. Sin embargo, Snyder dividió de forma casi completa su trabajo en dos enfoques: el *feature matching* y el *template matching*. En este artículo, vamos a intentar reconciliar ambos enfoques en una metodología común. De esta forma, podremos analizar sus similitudes y diferencias de una forma más sencilla de comprender.

Dividiremos así la metodología en dos fases, la detección y el reconocimiento de cartas, con una interfase, el entrenamiento, que se llevará a cabo exclusivamente para el *feature matching*. Aunque solo uno de estos enfoques puede aplicarse a la vez, la fase de entrenamiento se llevará siempre a cabo en nuestra metodología, con el objetivo de recopilar toda la información necesaria para ejecutar la detección con cualquiera de los métodos propuestos.

## 2.1. Detección

En esta fase nos centraremos en obtener una o más cartas a partir de una imagen. Cabe destacar que a lo largo del proceso nos encontraremos con dos tipos de imágenes: las plantillas, o *template images*, que son aquellas que usaremos para aprender qué tipo de cartas hay y cómo son, y las de consulta, o *query images*, que son las imágenes que introducirá el usuario para detectar e identificar cartas en ellas utilizando la información extraída de las primeras. Es lógico pensar que el primer tipo pueden ser fotografías tomadas con especial cuidado, colocando la carta en una posición favorable y sobre un fondo que sea claramente diferenciable de la carta, de forma que facilitemos al algoritmo la detección de la carta. Es por esto que podemos considerar este primer tipo de imágenes como un subconjunto del segundo tipo. Si obtenemos un algoritmo capaz de detectar cartas en una *query image*, entonces podremos usarlo de igual manera para detectar cartas en una imagen con mejores condiciones.

Por el hecho de simplificar la explicación, utilizaremos como ejemplo una imagen de entrenamiento como la siguiente para ilustrar el procedimiento.

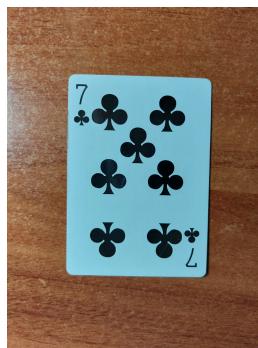


Figura 1: Una imagen plantilla que contiene un siete de tréboles.

### 2.1.1. Detección del contorno principal

Lo primero que necesitaremos llevar a cabo es una detección de contornos. Para ello, usaremos la técnica de binarización de forma que podamos separar claramente las cartas, que son de un color blanco o muy similar, del fondo, que suponemos que será de un color distinto. Para hallar un umbral correcto, utilizaremos el método de Otsu, aunque otros métodos para calcular un umbral podrían utilizarse siempre y cuando puedan separar claramente el contorno de la carta.

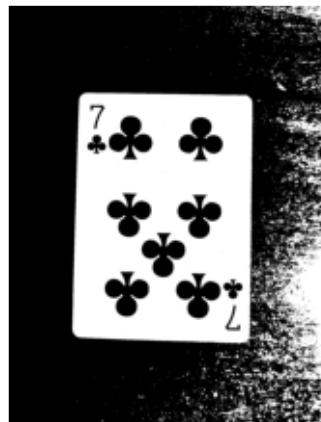


Figura 2: Imagen tras binarizarla utilizando el método de Otsu.

Una vez la imagen está segmentada, es el momento de localizar los contornos que nos puedan ser útiles, es decir, aquellos que puedan contener una carta. No todos los contornos son válidos, por lo que implementaremos una heurística que descarte aquellos que no cumplan todas las condiciones siguientes:

- El polígono resultante tiene cuatro vértices.
- El área del polígono debe ser mayor al 3% del área total de la imagen.

Nótese que, mientras la primera regla es inamovible siempre que consideremos cartas rectangulares, la segunda regla podría variar dependiendo del tamaño que esperamos que la carta represente dentro de una imagen, por lo que se podría considerar un parámetro del algoritmo. De esta forma, eliminaremos la mayoría de los contornos que se hayan podido detectar y que no representen cartas. Además, ordenaremos los que hayan pasado el filtro por el tamaño de su área, esperando que aquellos de mayor tamaño sean los que representen cartas y así evitemos procesar contornos indeseables.

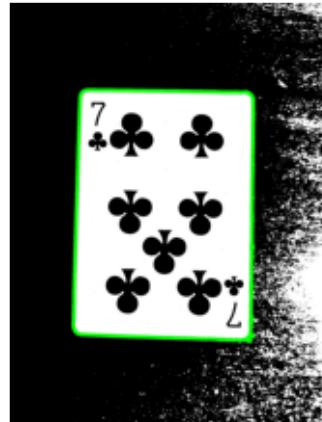


Figura 3: Contorno resultante tras aplicar el filtro.

#### 2.1.2. Determinar la orientación de la carta

Ahora podemos aislar el contorno obtenido calculando el rectángulo que lo contiene y recortándolo de la imagen, subimagen que a partir de ahora llamaremos ROI (*Region Of Interest*, en inglés). Sin embargo, nos encontramos con otro problema: la carta podría estar rotada o sesgada. Para poder compararla con otras cartas y posteriormente realizar el reconocimiento, es sumamente importante que todas las cartas se encuentren en la misma posición y tengan el mismo tamaño, que en nuestro caso será vertical. Para solucionar este problema, usaremos una transformación afín de forma que los cuatro vértices de la carta se correspondan con los cuatro vértices de un rectángulo en posición vertical y con la misma proporción que una carta plana.

Para poder llevar esto a cabo, necesitamos relacionar cada vértice de la carta con cada vértice del rectángulo. Como podemos observar, solo dos de los vértices de la carta sirven como esquina superior izquierda del rectángulo, dada su simetría. Sin embargo, estos dos vértices tienen una peculiaridad: tienen un contorno más cerca (el número) que el resto de vértices. Este es un requisito importante para poder detectar cartas de forma correcta: al menos uno de sus esquinas debe estar marcada, y esa esquina debe funcionar como esquina superior izquierda al transformar la carta en un rectángulo.

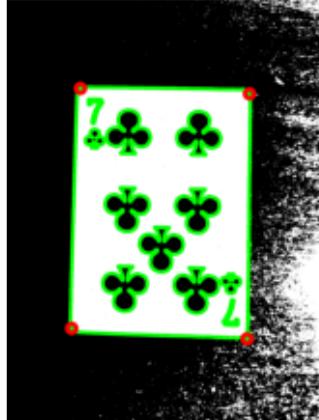


Figura 4: Contornos dentro de la carta (en color verde) y vértices (en color rojo).

Podemos iterar sobre cada uno de los vértices y anotar la mínima distancia entre cada uno de ellos y cualquiera de los contornos que existen dentro de la carta. De esta forma, aquel vértice con la menor distancia podrá ser usado como el vértice principal.

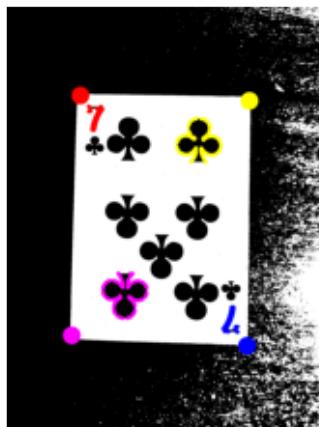


Figura 5: Vértices y contornos más cercanos. Los primeros, segundos, terceros y cuartos se representan con color rojo, azul, amarillo y rosa, respectivamente.

Ahora que conocemos cuál es el vértice que ocupará la esquina superior izquierda, solo es cuestión de saber el vértice que ocupará la esquina superior derecha. Tenemos dos posibles soluciones para esto:

- **Si conocemos la contigüidad de los vértices:** en este caso, podemos calcular la distancia entre el vértice que consideramos la esquina superior izquierda con su anterior y siguiente. El que tenga la menor distancia formará la arista vertical, que es de menor longitud que las aristas laterales. El resto de esquinas se pueden obtener iterando el resto de vértices en el mismo orden que hemos seguido tras hacer esta comprobación. Nótese que conocer que dos esquinas opuestas están marcadas significa conocer la contigüidad de los vértices, ya que una vez obtengamos el vértice con menor distancia al principal (aquel que forma la arista vertical), el segundo vértice con un contorno más cercano será el vértice opuesto al principal.
- **Si no conocemos la contigüidad de los vértices:** podemos hallar el vértice que forma la arista vertical calculando la menor distancia con cualesquiera de los tres vértices restantes, y así sucesivamente con el resto de vértices.

#### 2.1.3. Aplicación de transformación afín

Podemos calcular así la matriz de transformación afín y obtener nuestra carta de forma completamente aislada del resto de la imagen. Es importante volver a aplicar una binarización a la imagen

resultante, ya que tras aplicar la transformación afín es probable que haya píxeles que tengan un valor diferente.

Por otro lado, es conveniente que el tamaño de la imagen resultante sea siempre el mismo, de forma que facilitemos el entrenamiento y la posterior identificación. Trabajar con imágenes de grandes tamaños requerirá más memoria y mayor tiempo de procesamiento, mientras que imágenes más pequeñas podrían reducir la calidad visual y por tanto alterar la capacidad de extraer características o representar rasgos importantes de la imagen. Además, la imagen también debería respetar la ratio entre la altura y la anchura de una carta en la realidad, cuyo valor ronda el 1.416 tras ser medido experimentalmente.

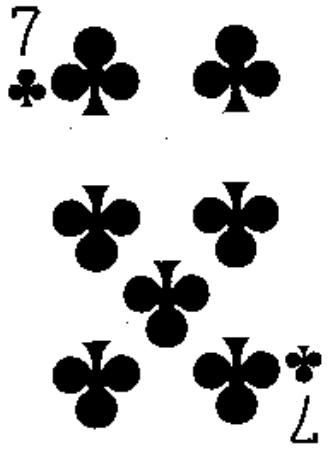


Figura 6: Carta recortada, binarizada y lista para ser usada en la identificación.

## 2.2. Entrenamiento

Como hemos explicado anteriormente, esta fase se lleva a cabo tras terminar con la detección y antes de proceder con la identificación de la carta. Aunque es una fase más que realizamos, solo será de utilidad para llevar a cabo una identificación basada en *feature matching*, por lo que en caso de querer una implementación completamente basada en *template matching* es conveniente omitirla para ahorrar tiempo de cómputo y acelerar el procedimiento.

Las *features* o características de una imagen en este contexto son, resumidamente, puntos de una imagen que ofrecen una gran información sobre la misma [2], y que trataremos de buscar en otras imágenes con el objetivo de saber si ambas son similares o no. Podemos ver las características de una imagen como cajas negras cuyo grado de similitud con otras características está directamente relacionado con el grado de similitud entre dos imágenes.

Para operar con características necesitaremos, simplificando, dos agentes: un extractor de características y un *matcher* o emparejador de características. El primero nos servirá para extraer las características de una imagen. Realizaremos esto sobre la imagen resultante de la identificación, ya que es una forma estandarizada de representar una carta y aumenta la probabilidad de que haya coincidencias entre dos imágenes de cartas idénticas. En el caso de las imágenes plantillas, guardaremos las características que hayamos encontrado para poder usarlas más tarde sin necesitar de volver a extraerlas.

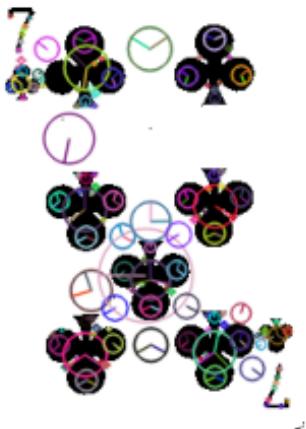


Figura 7: Características SIFT extraídas de una carta.

Como extractores, usaremos SIFT [3] (*Scale Invariant Feature Transform*) y ORB [4] (*Oriented FAST and Rotated BRIEF*). Ambos tienen grandes ventajas, ya que sus características son independientes de la rotación y la escala de las imágenes. Sin embargo, sus características no son compatibles entre sí, por lo que utilizaremos únicamente una de ellas.

### 2.3. Identificación

Es importante recordar de nuevo el significado de identificar una carta. Mientras que la detección tiene como finalidad encontrar una carta en una imagen, la identificación tiene como finalidad identificar la carta dados unos parámetros como pueden ser el palo y el número. Por supuesto, para realizar esta identificación es necesario contar con ciertos recursos de antemano. Llegados a este punto, necesitaremos:

- Por cada carta del mazo que queremos identificar, necesitaremos una imagen de la carta aislada, aplanaada y binarizada que haya pasado el proceso de detección satisfactoriamente, y de la que sepamos su palo y su número.
- En caso de usar *feature matching*, necesitaremos las características de la imagen anterior, extraídas con cualquiera de los extractores propuestos.

Es en este momento donde será importante decantarse por una de las dos opciones que planteamos para identificar la carta.

#### 2.3.1. Feature matching

Primero, extraemos las características de la carta a identificar procedente de la imagen de consulta, una vez pasada por la fase de detección. Con las características de cada una de las cartas plantilla y las de la carta que queremos identificar, podremos realizar el emparejamiento entre las características de la carta en cuestión y cada una de las cartas plantilla. De esta forma, el emparejamiento con mejores aciertos, dicho de otra forma, con menor ambigüedad entre los pares de características que relacionan ambas imágenes, será el que consideremos válido.

Cabe destacar que podemos establecer una calidad mínima en los pares de características, de forma que si el mejor de los emparejamientos no llega a una cota inferior de calidad, podemos dar por hecho que no hemos podido identificar la imagen como una carta. Este enfoque es conveniente en caso de querer identificar varias cartas en una misma imagen. Basta con tratar de identificar todas las potenciales cartas que hayan pasado la fase de detección, hasta encontrar un número  $n$  de cartas que no cumplan la cota inferior de calidad. En ese caso, podremos considerar que hemos dejado de analizar cartas y que ya no existen más cartas en la imagen. Esto también podrá aplicarse al *template matching*, como veremos más tarde.

Podemos usar multitud emparejadores para realizar esta tarea. Entre ellos, destaca BF (*Brute-Force matcher*) y FLANN (*Fast Library for Approximate Nearest Neighbors*). Ambos soportan tanto características SIFT como ORB, pero solo podremos usar uno a la vez. Los resultados de FLANN tienden a ser mejores y más rápidos en la mayoría de las situaciones.

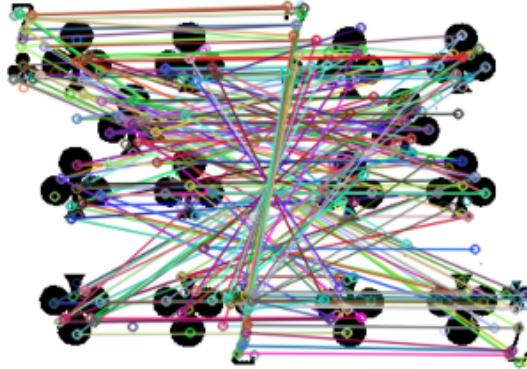


Figura 8: Pares de características SIFT (FLANN) entre dos cartas semejantes.

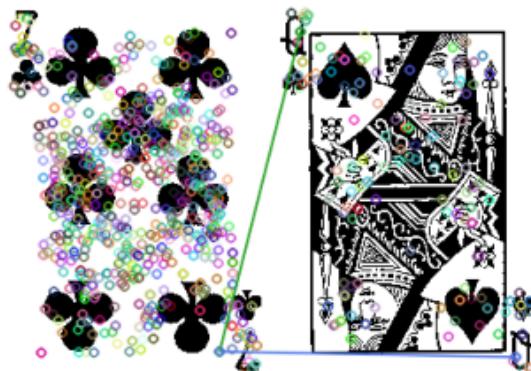


Figura 9: Pares de características SIFT (FLANN) entre dos cartas diferentes.

### 2.3.2. Template matching

Este método tiende a dar mejores resultados que el *feature matching*, e incluso llega a ser considerablemente más rápido, teniendo en cuenta que no es necesario pasar por la fase de entrenamiento ni extraer características de la imagen de consulta.

Su funcionamiento es aún más simple: dadas dos imágenes suficientemente similares, podemos restar los valores de sus píxeles uno a uno y calcular la diferencia al cuadrado. Este método, que puede parecer simple, es suficientemente eficaz como para conseguir buenos resultados. Sin embargo, puede fallar si las imágenes, a pesar de ser idénticas, están ligeramente desplazadas una respecto a la otra.

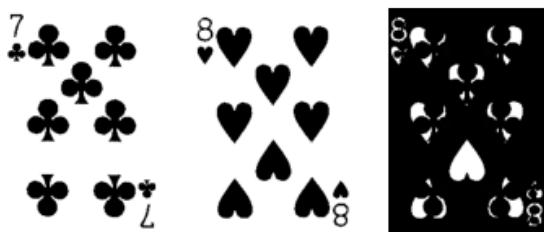


Figura 10: Dos imágenes diferentes a la izquierda y su resta a la derecha. Menor cantidad de píxeles blancos implica mayor coincidencia.

Para evitar, este problema, se ha usado otro método para medir la diferencia entre ambas imágenes llamado coeficiente de correlación [5]. Al tener en cuenta posibles desplazamientos de la imagen, calcula la diferencia entre ambas imágenes en múltiples iteraciones y devuelve el mejor resultado encontrado. Es de esperar que dos imágenes casi idénticas obtengan, tras varias iteraciones,

una puntuación casi perfecta.

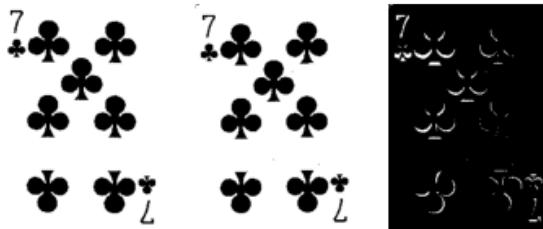


Figura 11: Dos imágenes casi idénticas a la izquierda y su resta a la derecha. Menor cantidad de píxeles blancos implica mayor coincidencia.

### 3. Implementación

Toda la implementación se ha realizado en un notebook de Jupyter autocontenido. Todo el código sigue la misma estructura explicada a lo largo de este artículo, y se ha intercalado con explicaciones teóricas. Además, cada bloque de código se encuentra perfectamente documentado de forma que se facilite su comprensión y seguimiento. Cabe destacar que el idioma utilizado tanto para la escritura del código fuente como para los comentarios ha sido el inglés, por ser el idioma *de facto*.

```
# This method is a generator that will try to identify and isolate card-like rectangles
# from a given image. All the returned rectangles are contained in the image, and they
# are returned as binary images that have been straightened using an affinity matrix
def get_potential_cards(image, dst_width=200, dst_height=283, card_min_image_area_percent=0.03, roi_min_image_area_percent=0.001):
    # First, let's binarize the image using Otsu's thresholding.
    gray_image = cv.cvtColor(image, cv.COLOR_BGR2GRAY)
    _, thresh_image = cv.threshold(gray_image, 0, 255, cv.THRESH_BINARY + cv.THRESH_OTSU)
    # Now, we get all possible contours in the binarized image.
    contours, _ = cv.findContours(
        thresh_image, cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)

    # Not all contours are potential cards: we want them to meet
    # some conditions to avoid unnecessary computations.
    best_card_contours = get_best_card_contours(
        image, contours, card_min_image_area_percent)
    while True:
        best_card_contour = next(best_card_contours)
        if best_card_contour is None:
            break

        # It seems that we've got a potential card in this contour, so
        # let's assume it is so. The card could be rotated, skewed...
        # We'll need to straighten the card so it's top-left corner matches
        # the top-left corner of the result image, as training cards do.

        # We start by isolating the ROI to a new image and getting some
        # information like vertices, bounding rect area, etc.
        roi_x, roi_y, roi_w, roi_h = cv.boundingRect(best_card_contour)
        roi_thresh_image = thresh_image[roi_y:roi_y +
                                         roi_h, roi_x:roi_x + roi_w]
        roi_origin = [roi_x, roi_y]
```

Figura 12: Algunas de las funciones del código fuente.

La gran mayoría del código es de autoría propia, exceptuando algunas funciones auxiliares de pequeño tamaño que, aunque no han sido copiadas, sus ideas han sido extraídas y plasmadas de forma que mantengan la cohesión del resto del código. La mayoría de sitios web de los que se ha extraído código o ideas han sido la documentación oficial de OpenCV y StackOverflow.

Para la realización del trabajo, se ha utilizado Python 3.10.4 junto a las librerías OpenCV 4.6.0.66, Numpy 1.22.4 y Matplotlib 3.5.2, además de los requisitos de cada una de estas librerías. Dado que el algoritmo SIFT ya ha sido liberado de su patente, todos los algoritmos usados en este trabajo provienen del software libre.

## 4. Experimentación

Para la experimentación, se han tomado 17 fotos distintas, cada una conteniendo una o más cartas, desde distintos ángulos, inclinaciones, fondos e iluminaciones. Dada la gran cantidad de configuraciones disponibles dentro de cada método de identificación, se ha optado por escoger los mejores valores encontrados experimentalmente para cada uno de ellos:

- Para el feature matching, se han utilizado características de tipo SIFT con un emparejador FLANN configurado específicamente para SIFT.
- Para el pattern matching, se ha utilizado una comparación de imágenes basada en el coeficiente de correlación.

Los resultados de la detección e identificación de imágenes utilizando cada una de estas configuraciones han resultado ser los dados por la tabla.

	Feature matching		Pattern matching		Total
	Bien	Mal	Bien	Mal	
Q1	1	1	1	0	1
Q2	1	1	1	0	1
Q3	1	0	1	0	1
Q4	1	0	1	0	1
Q5	0	1	0	0	1
Q6	0	0	0	0	1
Q7	0	0	0	0	1
Q8	1	0	1	0	1
Q9	1	0	1	0	1
Q10	2	0	2	0	2
Q11	0	0	0	0	2
Q12	0	0	0	0	2
Q13	3	1	3	0	3
Q14	1	0	1	0	1
Q15	4	1	5	0	5
Q16	0	0	0	0	1
Q17	1	2	2	1	3

Nótese que se considera que una carta ha sido bien tratada si se ha detectado e identificado correctamente. Una carta mal tratada ha podido ser mal detectada (falso positivo o carta incompleta/mal transformada) o mal identificada.

Como podemos observar, ambas configuraciones tienen una tasa de acierto bastante similar. Sin embargo, el pattern matching toma ligeramente la delantera ya que los fallos son casi inexistentes. Feature matching, por otro lado, parece tener problemas especialmente con cartas de números altos, donde el contenido está contenido por un rectángulo dentro de la carta. Esto provoca que se haga un doble matching de la carta, primero por su borde exterior y luego por este borde, generando un falso positivo.

A continuación mostraremos algunas de las imágenes de consulta más relevantes.

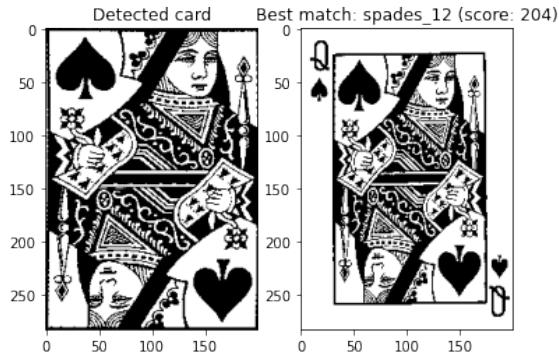


Figura 13: Falso positivo dado por feature matching en una carta de número alto.



Figura 14: Imagen Q5, as de tréboles delante de una pantalla con multitud de elementos en ella. Gran dificultad.



Figura 15: Imagen Q15, cinco cartas diferentes encima de una mesa. Dificultad media.



Figura 16: Imagen Q16, siete de tréboles muy distorsionado por la perspectiva. Gran dificultad.

## 5. Manual de usuario

Es necesario tener instalado Python 3 (se recomienda la última versión) en tu sistema antes de comenzar a ejecutar el proyecto.

Tras descargar y descomprimir el archivo comprimido que contiene el proyecto, entra en la carpeta raíz donde se encuentra el archivo `requirements.txt` y ejecuta el siguiente comando para instalar las dependencias necesarias en tu instalación de Python:

```
pip install -r requirements.txt
```

A continuación, necesitarás un intérprete de notebooks de Jupyter, así como un kernel de IPython. Tienes varias opciones:

- Instala una distribución de Python científica como Anaconda, que ya contendrá un visualizador de Jupyter notebooks y todas las dependencias necesarias para correr el proyecto.
- Usa Visual Studio Code utilizando el plugin [Jupyter](#), tal y como describe el siguiente [tutorial](#).

El principal archivo que contiene todo el código fuente se encuentra en `src/main.ipynb`. En general, todo el código es autoexplicativo. Si no se desea cambiar las imágenes plantilla y solo se quiere identificar cartas en imágenes, basta con introducir nuevas imágenes en la carpeta `query` y ejecutar todos los bloques excepto los dos últimos. Una vez allí, escoge uno de los bloques según la técnica de identificación que deseas aplicar, cambia el nombre de la imagen a analizar (`query_image_name`) y ejecuta el bloque. Deberías ver los resultados rápidamente. No es necesario que vuelvas a ejecutar el resto de bloques para cambiar la imagen a identificar.

En caso de querer cambiar las imágenes plantilla, puedes reemplazarlas por cualquier otra imagen. Es necesario eliminar la imagen correspondiente a la imagen plantilla en la carpeta `training` en caso de haberla modificado, con el objetivo de que la imagen se vuelva a entrenar. Borrar la carpeta `training` hará que se entrenen todas las imágenes plantilla de nuevo.

Algunas variables globales pueden ser cambiadas para cambiar el tipo de baraja que se usa, las carpetas donde se encuentran las imágenes o la nomenclatura que siguen los archivos. En caso de realizar algún cambio, es importante asegurarse de que las carpetas y las imágenes se encuentren sincronizadas con estos cambios.

## 6. Conclusiones

Los resultados del estudio de Snyder fueron desalentadores: mientras el feature matching apenas era capaz de reconocer adecuadamente una carta en un ambiente favorable, el template matching parecía tardar un minuto o más para reconocer una única carta.

Mientras que es cierto que el feature matching se enfrenta a un número mayor de problemas a la hora de reconocer cartas, especialmente por su simetría, en este trabajo se ha conseguido una implementación capaz de identificar cartas con una tasa de acierto muy alta en apenas unos

segundos, equiparándose en tiempo con el template matching. Si bien es cierto que sigue sin ser la opción ideal, es una opción a tener en cuenta. El template matching, por su lado, hace gala de su robustez y es capaz de identificar casi cualquier carta sin ningún problema en el mismo o menos tiempo del que lo hace el feature matching.

Además de cosechar ambos éxitos, es necesario resaltar como se ha conseguido crear un marco común para que muchos algoritmos distintos interoperen con el mismo objetivo. Separar la detección de cartas de la identificación ha sido un gran acierto, ya que ambos métodos se aprovechan de las ventajas que les proporciona las transformaciones que se realizan en esta fase temprana.

A pesar de que el trabajo inicialmente estaba únicamente orientado hacia el feature matching, la adición del template matching ha sido un gran acierto en todos los sentidos.

## 7. Autoevaluación

Apartado	Puntuación
Comprendión y dominio del tema	Excelente (1 pto)
Exposición didáctica	Excelente (1 pto)
Integración del equipo	No aplica
Objetivos	Excelente (1 pto)
Aspectos didácticos	Bien (0.65 ptos)
Trabajo reutilizable	Excelente (1 pto)
Experimentación y conclusiones	Bien (0.65 ptos)
Contenidos	Excelente (1 pto)
Divulgación de los contenidos	Excelente (1 pto)
Organización de documentación	Excelente (1 pto)
Bibliografía. Recursos	Bien (0.65 ptos)

## 8. Tabla de tiempos

Fecha de la actividad	Tiempo (h)	Actividad realizada
10 de junio	3h	Lectura y comprensión del TDA
10 de junio	20min	Toma de fotos plantilla
10 de junio	3h	Inicio del proyecto como Jupyter Notebook
11 de junio	4h	Primeros pasos con OpenCV
12 de junio	6h	Detección de cartas básica
17 de junio	6h	Extracción y guardado de características
18 de junio	3h	Feature matching básico
19 de junio	3h	Mejora del feature matching
23 de junio	2h	Comentar código anterior
24 de junio	6h	Mejora de la detección
25 de junio	8h	Transformación afín según vértices
25 de junio	6h	Template matching
26 de junio	6h	Mejora del código (abstracciones, etc)
26 de junio	8h	Realizar documentación
27 de junio	2h	Realizar documentación
27 de junio	6h	Realizar presentación

## Referencias

- [1] Snyder, D. (n.d.). Playing card detection and identification - Stanford University. Playing Card Detection and Identification. Retrieved June 26, 2022, from [https://web.stanford.edu/class/ee368/Project\\_Winter\\_1819/Reports/snyder.pdf](https://web.stanford.edu/class/ee368/Project_Winter_1819/Reports/snyder.pdf)
- [2] Understanding features. OpenCV. (n.d.). Retrieved June 27, 2022, from [https://docs.opencv.org/4.x/df/d54/tutorial\\_py\\_features\\_meaning.html](https://docs.opencv.org/4.x/df/d54/tutorial_py_features_meaning.html)
- [3] Introduction to SIFT (scale-invariant feature transform). OpenCV. (n.d.). Retrieved June 27, 2022, from [https://docs.opencv.org/4.x/da/df5/tutorial\\_py\\_sift\\_intro.html](https://docs.opencv.org/4.x/da/df5/tutorial_py_sift_intro.html)

- [4] Orb (oriented fast and rotated brief). OpenCV. (n.d.). Retrieved June 27, 2022, from [https://docs.opencv.org/4.x/d1/d89/tutorial\\_py\\_orb.html](https://docs.opencv.org/4.x/d1/d89/tutorial_py_orb.html)
- [5] Template matching. OpenCV. (n.d.). Retrieved June 27, 2022, from [https://docs.opencv.org/4.x/d4/dc6/tutorial\\_py\\_template\\_matching.html](https://docs.opencv.org/4.x/d4/dc6/tutorial_py_template_matching.html)