UNIVERSIDAD
DE LA RIOJA

Facultad de Ciencia y Tecnología

TRABAJO FIN DE MÁSTER

Máster en Ciencia de Datos y Aprendizaje Automático

Digitalización del archivo musical de la catedral de Santo Domingo de la Calzada

Realizado por:

Gonzalo Santamaría Gómez

Tutelado por:

César Domínguez Pérez

Jónathan Heras Vicente

Resumen

En este trabajo vamos a explicar el proceso de digitalización de un libro de música titulado: "La Música en la Catedral de Santo Domingo de la Calzada". Para ello ha sido necesario realizar un escaneado de cada una de las páginas para, después, aplicarles técnicas de *visión por computador*. La digitalización precisaba de cierta estructura, ya que en los capítulos del libro se pueden encontrar obras musicales de diferentes géneros y autores. En estas obras, a su vez, se puede encontrar tanto texto como pentagramas musicales y es por eso que utilizaremos dos tecnologías que utilizan la *visión por computador*: el *OCR* y el *OMR*. El *OCR* nos servirá para extraer el texto y el *OMR* para extraer la música de las obras. El resultado final será tener las obras almacenadas en un formato que nos permita visualizarlas y escuchar su melodía a través de una pequeña aplicación web.

Abstract

In this work we are going to explain the digitisation process of a music book entitled: "La Música en la Catedral de Santo Domingo de la Calzada". To do this, it was necessary to scan each of the pages and then apply *computer vision* techniques to them. The digitisation required a certain structure, since the chapters of the book contain musical works of different genres and authors. In these works, in turn, both text and musical staves can be found, and that is why we will use two technologies that use *computer vision*: the *OCR* and the *OMR*. The *OCR* will be used to extract the text and the *OMR* to extract the music from the works. The final result will be to have the works stored in a format that allows us to visualise them and listen to their melody through a small web application.

Agradecimientos

Este trabajo ha sido financiado gracias a la *OTRI* suscrita con el *Instituto de Estudios Riojanos* con referencia *OTCA 201104* titulada "Digitalización de los archivos musicales de las Catedrales de Santo Domingo de la Calzada y Calahorra.

Índice general

Capítulo 1

Introducción

En la actualidad, la digitalización se presenta como una de las principales formas de preservación y mejora del acceso a todo tipo de documentos. En los últimos años, gracias al aprendizaje automático, este proceso se ha conseguido ampliar y ya no se limita al escaneo y almacenamiento de archivos en forma de imágenes, si no que ahora también somos capaces de extraer la información que contienen los documentos para poder exportarla.

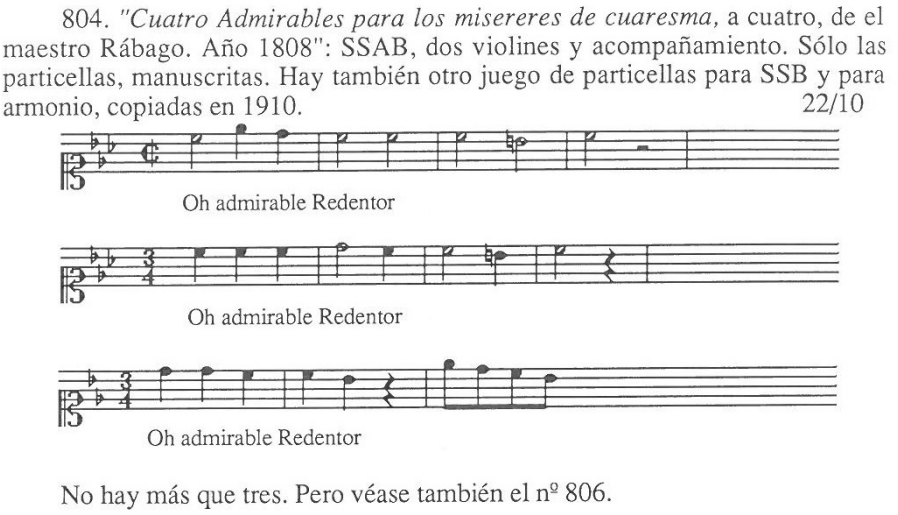
El objetivo de este trabajo es digitalizar un libro de música. Este proyecto fue propuesto por Teresa Cascudo, Profesora Titular del Área de Música de la Universidad de La Rioja y directora del área de investigación de Patrimonio Regional del Instituto de Estudios Riojanos. El libro se titula *La Música en la Catedral de Santo Domingo de la Calzada* [1] y en él se pueden encontrar distintas obras de carácter religioso como misas, villancicos o salmos de diferentes autores. Fue impreso en el año 1988 y no cuenta con una versión digital como sucede ahora con la mayoría de libros.

El proceso de digitalización podría haberse realizado manualmente, pero se ha optado por utilizar las técnicas de visión por computador y aprendizaje automático vistas en el máster. El motivo es que existen más libros con un formato similar y la idea es que el trabajo realizado en este proyecto pueda aplicarse a otros libros de música que se encuentran disponibles. Es decir, pretendemos definir un conjunto de métodos y técnicas que sean aplicables a otros libros similares.

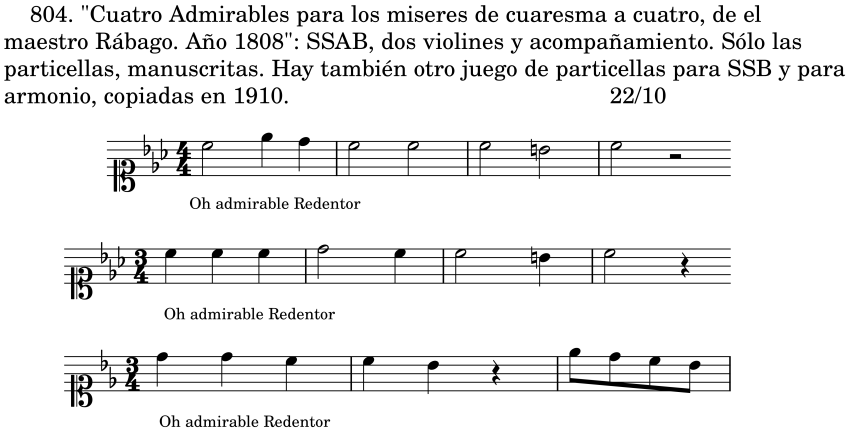
El proyecto se puede dividir en dos fases: la primera es la localización y reconocimiento de texto, y la segunda es la localización y reconocimiento de música. Para ello, utilizaremos principalmente dos tipos de técnicas: el *OCR* (del inglés, *Optical Character Recognition*) para el texto y el *OMR* (del inglés, *Optical Music Recognition*) para la música.

El resultado final será una aplicación web que permita realizar búsquedas de la forma: "Muéstrame todas las obras del autor Manuel de Rábago escritas en Español" o "Muéstrame todos los Villancicos". Además, aplicando las técnicas ya mencionadas, podremos transformar esas obras procedentes de imágenes escaneadas en documentos pdf y así eliminar el ruido y tener una notación general o reproducir el sonido que representa.

Así, por ejemplo, para la obra 804, de la página 179:



nuestro modelo final será capaz de codificar la información y devolver un pdf limpio como el siguiente:



o reproducir[1] la música que representa:

El resto de la memoria se divide en cuatro partes: la primera versará sobre la preparación de datos, la segunda sobre *OCR*, la tercera sobre *OMR* y la última sobre la aplicación web donde se podrá consultar la información del libro.

[1]Solo si abres este documento con Adobe Acrobat Reader DC. También puedes descargar el archivo *Mp3* en: `https://github.com/joheras/MusicaCatedralStoDomingoIER/blob/main/Predicciones/Obra_804/Obra_804.mp3`

Capítulo 2

Preparación de datos

En este capítulo vamos a explicar el proceso que hemos seguido para poder almacenar en formato digital nuestro libro. Principalmente explicaremos la forma de escanear las páginas y mostraremos cómo detectar ciertos bloques de interés mediante técnicas de procesamiento de imágenes. Comentar que el código de este trabajo, tanto de este capítulo como de los siguientes, puede encontrarse en el siguiente enlace: `https://github.com/joheras/MusicaCatedralStoDomingoIER`.

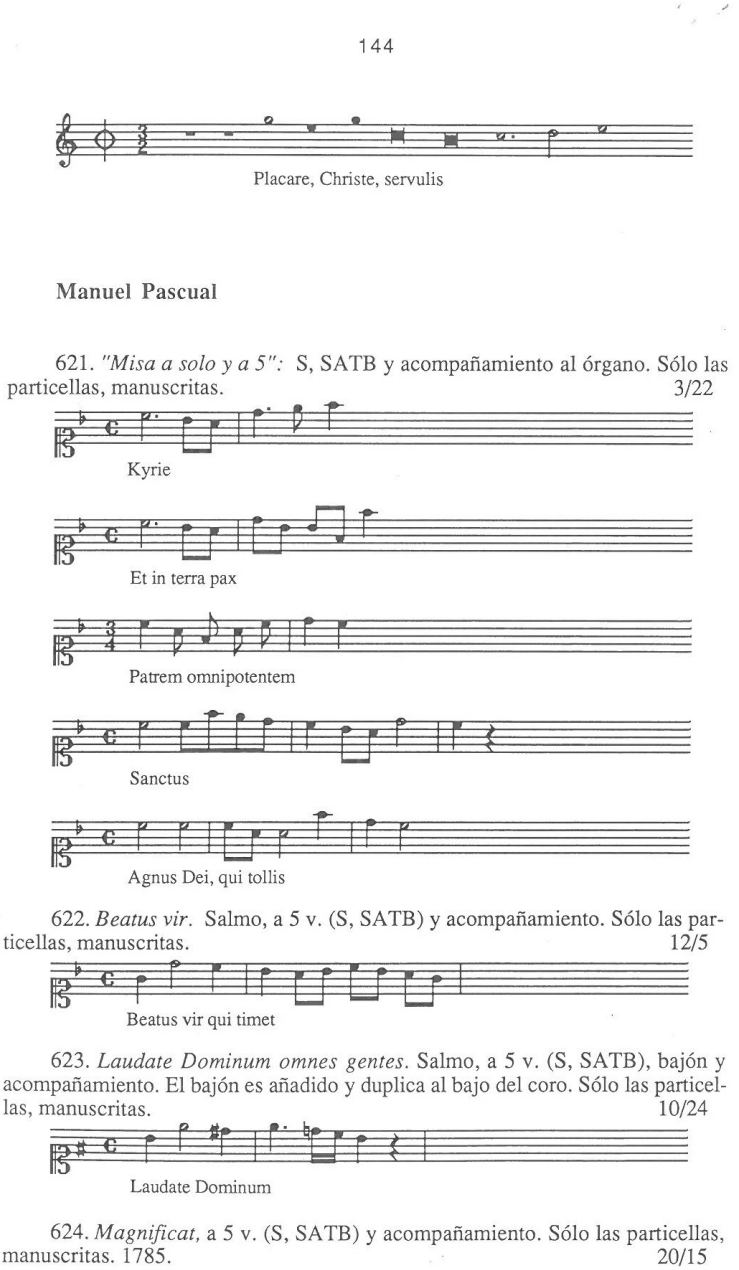
2.1. Escaneado del libro

El libro "*La Música en la Catedral de Santo Domingo de la Calzada*" tiene un total de 390 páginas, de las cuales se han escaneado 383, ya que las restantes forman parte de la introducción. La máquina que se ha utilizado para realizar el escaneado es una *Bizhub c452* situada en la Oficina Técnica 2 del Departamento de Matemáticas y Computación de la Universidad de La Rioja. Esta impresora nos ofrece múltiples opciones de escaneo:

- **Dimensión**: A2, A3, A4, B2, B3, B4...
- **Resolución**: Cuanta más resolución, más píxeles tendrá la imagen y por tanto ocupará más memoria, pero también tendrá mejor calidad.
- **Formato**: pdf, jpg, png...
- **Tipo**: Escala de grises, blanco y negro, a color...
- **Densidad**: Colores más o menos vivos. Por ejemplo, si la imagen está en escala de grises, cuanta más densidad tenga más oscura será.

¿En qué nos vamos a basar para elegir el formato de escaneo? Para tomar esta decisión, vamos a seleccionar las opciones que nos den mejores resultados al aplicar la detección de texto con *OCR*.

¿Por qué solamente tenemos en cuenta el texto y no la música para tomar esta decisión? Porque las técnicas de *OCR* están bastante más consolidadas que las de *OMR*. Existen muchos programas de *OCR* listos para ser usados y que ofrecen muy buenos resultados en lo que a texto se refiere. Por el contrario, para la musica no existen programas generales y será necesario crea nuestros propios modelos y se podrán adaptar al formato de página elegido. Realizando varias pruebas, hemos optado por utilizar un tamaño de escaneo B5, formato de imagen jpg, en color y con una resolución y densidad por defecto. El tamaño de las imágenes escaneadas es de 1200 píxeles de ancho por 1825 de alto. Mostramos una de las páginas en la *figura 2.1.*


144

Placare, Christe, servulis

Manuel Pascual

621. *"Misa a solo y a 5":* S, SATB y acompañamiento al órgano. Sólo las particellas, manuscritas. 3/22

Kyrie

Et in terra pax

Patrem omnipotentem

Sanctus

Agnus Dei, qui tollis

622. *Beatus vir.* Salmo, a 5 v. (S, SATB) y acompañamiento. Sólo las particellas, manuscritas. 12/5

Beatus vir qui timet

623. *Laudate Dominum omnes gentes.* Salmo, a 5 v. (S, SATB), bajón y acompañamiento. El bajón es añadido y duplica al bajo del coro. Sólo las particellas, manuscritas. 10/24

Laudate Dominum

624. *Magnificat,* a 5 v. (S, SATB) y acompañamiento. Sólo las particellas, manuscritas. 1785. 20/15

Figura 2.1: Página 144 del libro.

En la *figura 2.1*, vemos que dentro de una página se puede encontrar diferente tipo de información. Hay *bloques de texto* y *pentagramas* que se pueden combinar para formar *obras*. En este caso, podemos encontrar un pentagrama de la obra 620, las obras 621, 622, 623 completas y el principio de la obra 624, estas cuatro últimas del autor *Manuel Pascual*. También podemos ver, por ejemplo, que la obra 621 es una *Misa*.

En este trabajo se pretende detectar y estructurar la información anterior, teniendo en cuenta características como el número de obra, el tipo de obra y autor, la partitura, etc.

2.2. Nociones previas

Tras el proceso de escaneado, disponemos de un conjunto de ficheros tal que cada uno de ellos contiene una imagen de una de las páginas. En un primer paso trataremos de detectar los distintos bloques que forman dicha imagen, separando entre bloques de texto y bloques de música. Para ello vamos a hacer uso de la librería *OpenCV* [2, 3], desarrollada en el lenguaje de programación C++ y adaptada para ser usada en otros lenguajes. En concreto, nosotros la vamos a utilizar en *Python*. También, nos va a ser útil utilizar la librería *Numpy* para poder tratar las imágenes como vectores [4]. En esencia, detectaremos tres cosas: bloques de texto, pentagramas musicales y obras (esta última es una combinación de las otras dos). Toda esta información la vamos a almacenar para su posterior uso. Dependiendo de lo que queramos detectar, deberemos utilizar unas transformaciones u otras. Comencemos por introducir las definiciones de *kernel* (o núcleo), *umbralizar* y *contorno*, además de lo que son las *operaciones morfológicas* para imágenes binarias.

Kernel. Es un indicador que nos dice cómo tiene que afectar una operación (en nuestro caso, una operación morfológica) a un píxel teniendo en cuenta ciertos píxeles adyacentes. La dimensión se puede denotar por (n, m). Aquí n y m son números impares y denotan respectivamente el número de píxeles adyacentes en las filas y las columnas, generalmente $n = m$. Los vamos a utilizar para transformar, utilizando operaciones morfológicas, imágenes binarias (umbralizadas) de cierta manera.

Umbralizar. Es una operación sobre las imágenes en escala de grises. Bajo una regla determinada cada píxel pasa a ser o blanco o negro. A este tipo de imagen se le llama *imagen binaria*, ya que toman únicamente dos valores. El objetivo de umbralizar nuestras imágenes va a ser para poder aplicarles operaciones morfológicas, las cuales nos permitirán alterar su forma original y así localizar zonas de interés. OpenCV ofrece muchas estrategias diferentes para umbralizar una imagen, la que nosotros hemos usado es el método *Otsu* [5, 6] como se muestra en el siguiente fragmento de código:

```
cv2.threshold(img,0,255,cv2.THRESH_BINARY_INV|cv2.THRESH_OTSU)
```

Operaciones morfológicas. Sirven para todo tipo de imágenes, pero nosotros vamos a aplicarlas solamente a imágenes binarias. El color blanco va a representar el fondo (del inglés, background) y el negro los objetos. A esos objetos les vamos a poder hacer transformaciones para alterar su forma original. En la *figura 2.2.* encontramos algunas de estas operaciones con un pentagrama de nuestro libro.

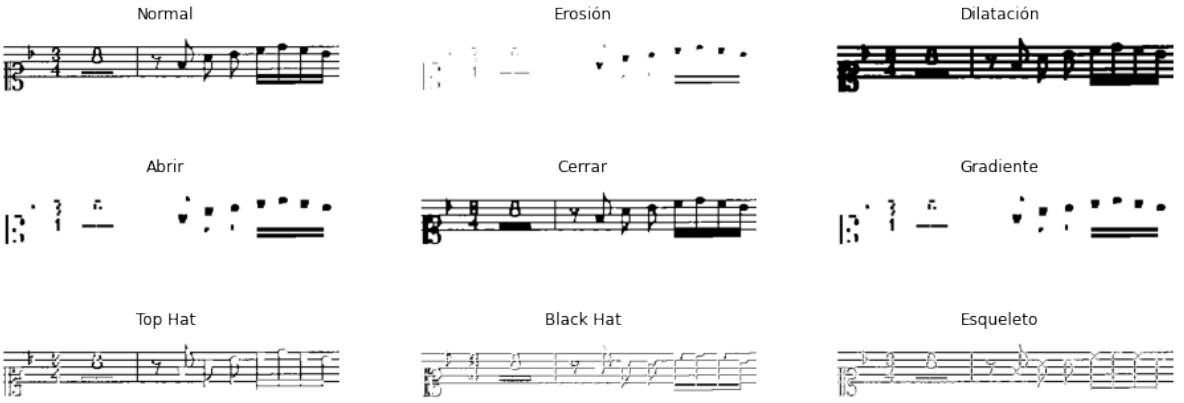


Figura 2.2: Transformaciones morfológicas elementales.

Las transformaciones de este ejemplo están realizadas en todas las direcciones. Podríamos haber optado por realizarlas solo verticalmente o en cualquier otra dirección. Para ello, tendremos que definir un *kernel* que decida si el píxel pasa a ser blanco o negro dependiendo de sus adyacentes. El código para realizar las distintas operaciones morfológicas se encuentra en la *figura 2.3.*

```
kernel = np.ones((3,3)) # operación en todas las direcciones
#kernel = np.ones((3,1)) para que afecte solo verticalmente
erosion = cv2.erode(img,kernel)
dilatacion = cv2.dilate(img,kernel)
abrir = cv2.morphologyEx(img,cv2.MORPH_OPEN,kernel)
cerrar = cv2.morphologyEx(img,cv2.MORPH_CLOSE,kernel)
gradient2 = cv2.morphologyEx(img,cv2.MORPH_GRADIENT,kernel)
tophat = cv2.morphologyEx(img,cv2.MORPH_TOPHAT,kernel)
blackhat = cv2.morphologyEx(img,cv2.MORPH_BLACKHAT,kernel)
```

Figura 2.3: Código Pyhton para realizar las diferentes operaciones morfológicas.

Contornos. Son las zonas de la imagen donde se produce una variación pronunciada en la intensidad de los píxeles. Localizando los contornos podremos ser capaces de detectar ciertas figuras. En las imágenes binarias los contornos son las zonas donde la imagen pasa

de blanco a negro o viceversa. Para nuestro problema hemos utilizado la siguiente función de OpenCV:

```
cnts,_ = cv2.findContours(img,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
```

La idea para detectar zonas de interés, ya sean bloques de texto, pentagramas o cualquier otro objetivo, va a ser siempre la misma: hacer una copia de la imagen original, umbralizarla, realizarle unas transformaciones morfológicas para destacar cierta región dentro de ella y buscar algunos de los contornos de la imagen transformada. En la *figura 2.4.* encontramos un esquema con todo el proceso a seguir.

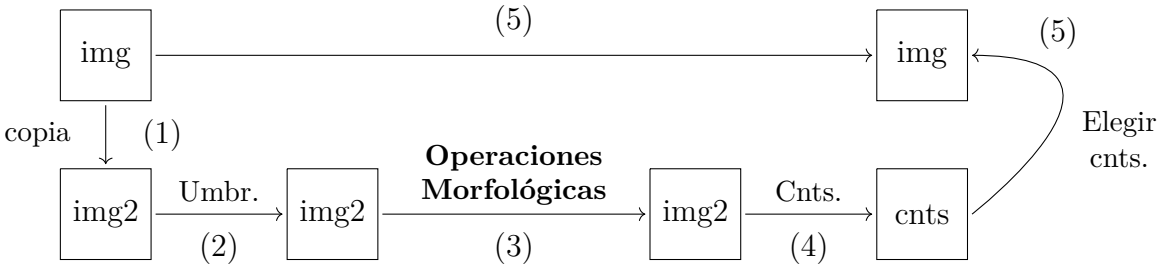


Figura 2.4: Proceso para detectar bloques.

2.3. Extracción de bloques

Veamos cómo podemos usar estas técnicas con dos ejemplos sobre una página de nuestro libro, la página mostrada en la *figura 2.1.* Para detectar los *pentagramas* podemos encontrar el proceso seguido en la *figura 2.5.* y el resultado obtenido en la *figura 2.6.*

```
img2 = img.copy()
img2 = cv2.cvtColor(img,cv2.COLOR_RBGR2GRAY)
_,img2 = cv2.threshold(img2, 0, 255, cv2.THRESH_BINARY_INV | cv2.THRESH_OTSU)
kernel = np.ones((3,1)) #Kernel vertical
img2 = cv2.dilate(img2,kernel,iterations=3) #Dilatamos verticalmente 3 veces
cnts,_ = cv2.findContours(img2, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
for c in cnts:
  area = cv2.contourArea(c)
  if area > 10000: #Escogemos cnts con un área grande
    (x, y, w, h) = cv2.boundingRect(c)
    cv2.rectangle(img, (x, y), (x + w, y + h), (119, 90, 17), 2) #Recuadramos
```

Figura 2.5: Código Python para detectar los pentagramas.

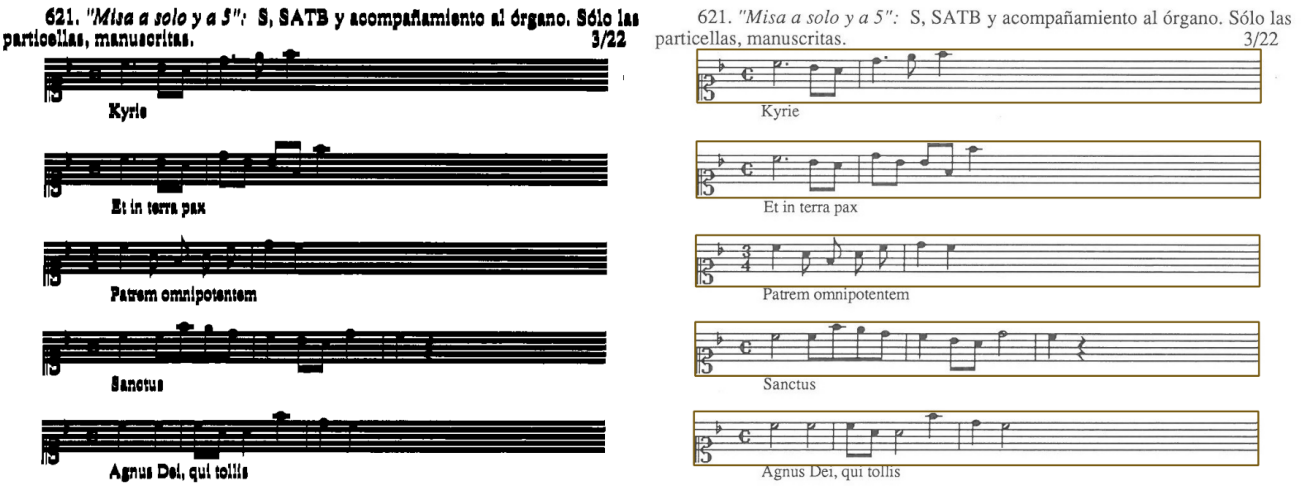


Figura 2.6: Ejemplo de detección de pentagramas. A la izquierda se encuentra la imagen transformada y a la derecha la imagen original con los pentagramas localizados.

Notar que en el código para detectar los pentagramas de la *figura 2.5.* la única operación morfológica que hemos usado es la dilatación, la cual nos permite formar bloques fácilmente reconocibles con los pentagramas (ver *figura 2.6*). Para detectar los *bloques de texto*, podemos ayudarnos de lo anterior. Al tener localizados los pentagramas, podemos poner un fondo blanco sobre ellos y seguir un procedimiento análogo al anterior utilizando otro tipo de operaciones morfológicas. En la *figura 2.7.* tenemos el proceso seguido y en la *figura 2.8.* el resultado obtenido.

```
img2 = img.copy()
img = borrarPentagramas(img2) #Función para borrar los pentagramas
img2 = cv2.cvtColor(img2,cv2.COLOR_BGR2GRAY)
_,img2 = cv2.threshold(img2, 0, 255, cv2.THRESH_BINARY_INV|cv2.THRESH_OTSU)
kernel = np.ones((30,30)) #Kernel en todas las direcciones
img2=cv2.morphologyEx(img2,cv2.MORPH_CLOSE,kernel) #Juntamos las componentes
cnts,_ = cv2.findContours(img2,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
for c in cnts:
  area = cv2.contourArea(c)
  if area > 100: #Escogemos todos los cnts menos los pequeños
    (x, y, w, h) = cv2.boundingRect(c)
    cv2.rectangle(img, (x, y), (x + w, y + h), (119, 90, 17), 2) #Recuadramos
```

Figura 2.7: Código Python para detectar bloques de texto.

La función *borrarPentagramas* de la *figura 2.7.* contiene el código de la *figura 2.5.* a excepción de una modificación en la última línea que añade un fondo blanco dentro de la región delimitada:

```
cv2.rectangle(img, (x, y), (x + w, y + h), (255, 255, 255), -1) #Fondo blanco
```

Para la detección de las obras va a ser más sencillo utilizar *OCR*, ya que todas ellas empiezan con un número (el nº de obra). Podemos sacar la *bounding-box* (cuadro delimitador de un objeto) teniendo en cuenta ese número y el de la siguiente obra o el inicio de una nueva sección. Una cosa que sí hay que tener en cuenta, es que algunas obras empiezan en una página y terminan en otra. En ese caso sí que usaremos operaciones morfológicas para detectar el trozo de obra que se queda en la página siguiente. Por último, comentar que todas las obras van a tener el mismo ancho, así podremos concatenar las obras que empiezan en una página y terminan en otra y mostrarlas como si fuesen una sola imagen.

Tanto la forma de detectar las obras como la forma en la que guardaremos los archivos con la información lo explicaremos con más profundidad en el siguiente capítulo. Ya que todavía nos falta saber cómo funcionan el *OCR*.

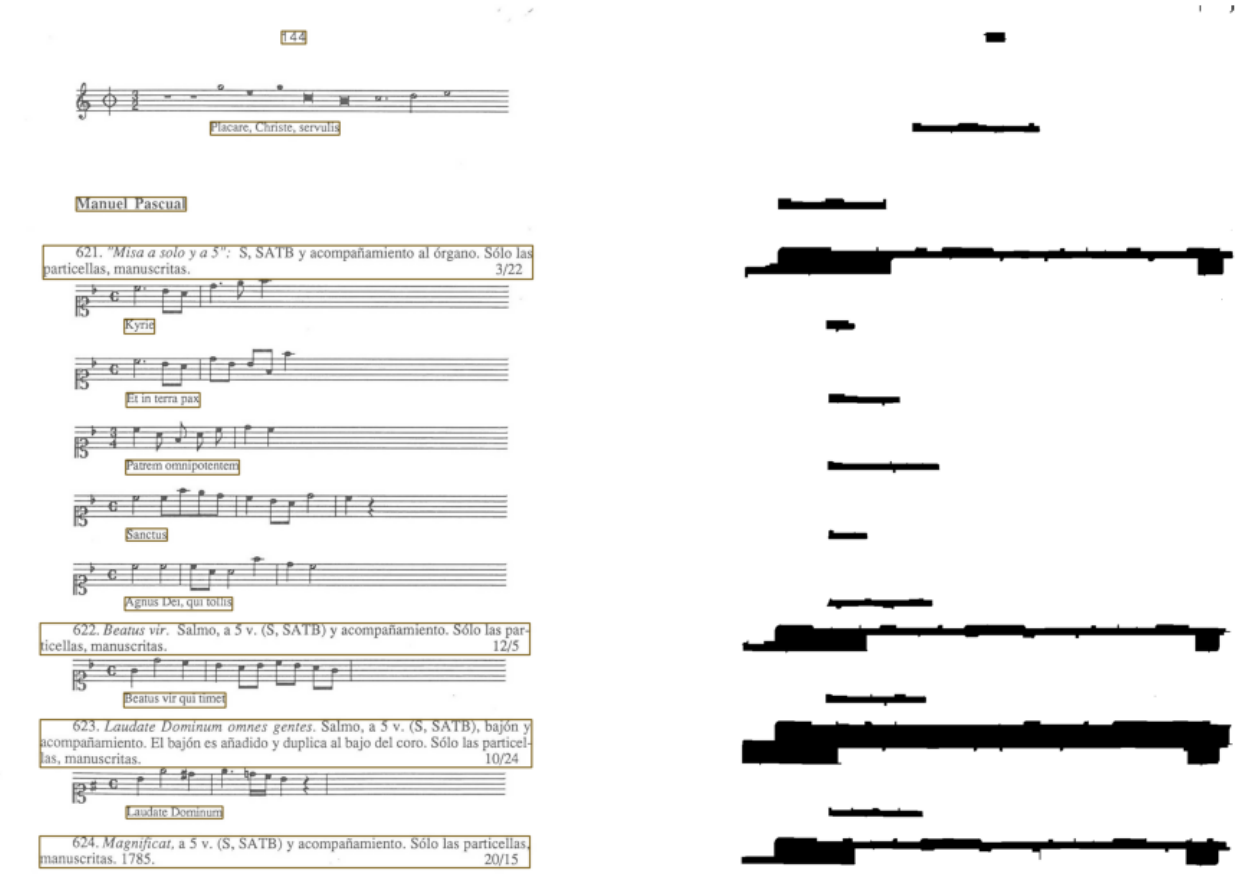


Figura 2.8: Localización del texto del libro utilizando operaciones morfológicas y contornos.

La forma de validar todos los procesos que hemos llevado acabo ha sido de manera manual. Hemos ido realizando pruebas para algunas páginas del libro y posteriormente verificándolas

con el resto de ellas para cerciorarnos de que los resultados son correctos. En el caso de encontrar algún error lo hemos corregido manualmente.

Por último, comentar que en el código de nuestro proyecto podemos encontrar en la mayoría de los cuadernos de *Jupyter* operaciones morfológicas para detectar bloques. Cabe destacar entre ellos los cuadernos con nombre: "*Detectando_pentagramas*", "*Detectando_símbolos*" y "*Descriptores_manual*". Estos dos últimos los comentaremos brevemente en el Capítulo 4 de esta misma memoria, ya que tratan sobre cómo detectar los símbolos musicales de un pentagrama utilizando operaciones morfológicas de *OpenCV* y algunas propiedades de los contornos para clasificarlos sin recurrir al *Deep Learning*.

Capítulo 3

OCR

En este capítulo explicaremos las diferentes alternativas que tenemos para extraer y almacenar el texto del libro. También comentaremos la tecnología que hemos usado para resolver este problema y veremos cómo realizar búsquedas dentro del texto.

3.1. Introducción al OCR

El *OCR*, del inglés *Optical Character Recognition* y traducido al español como *Reconocimiento Óptico de Caracteres*, es una tecnología la cual tiene como finalidad reconocer texto contenido en ficheros con formato de imagen, por lo que se considera un problema de visión por computador [7, 8].

Algunas de las aplicaciones del *OCR* que podemos encontrarnos en la vida real pueden ir desde la detección y reconocimiento de matrículas, o extraer el texto de páginas escaneadas como en nuestro caso. El *OCR* puede combinarse con más tecnologías para obtener un resultado más completo donde se reconozcan todos los atributos importantes de la imagen. En nuestro caso, como comentamos anteriormente, deberemos trabajar conjuntamente para detectar el texto y la música dentro de las páginas del libro.

En nuestro caso, no vamos a necesitar entrenar ningún modelo de *OCR*, ya que utilizaremos modelos publicados en la literatura que presentan un buen rendimiento. En concreto, vamos a utilizar el software de código abierto *Tesseract OCR* [9], al cual da soporte Google. Al igual que con *OpenCV*, este sistema está programado en C++, pero tiene una librería preparada para ser usada directamente en *Python* llamada *pytesseract*. También se ofrecen diferentes idiomas para ser usados, por lo que utilizaremos el *OCR* en español.

3.2. Detección de texto

Para la localización del texto dentro de la imagen podemos utilizar las siguientes funcionalidades de la librería *pytesseract*, la primera de ellas sirve para extraer de forma individual cada una de las palabras contenida en una imagen, mientras que la segunda extrae el texto completo de la imagen.

```
pytesseract.image_to_data(img, lang="spa", output_type=Output.DICT)
pytesseract.image_to_string(img, lang="spa")
```

En la primera línea (la primera funcionalidad) proporcionamos la imagen de la cual queremos extraer el texto, le indicamos el idioma y le decimos que devuelva un diccionario. En dicho diccionario encontraremos 12 claves distintas. Las más importantes son las que hacen referencia a las coordenadas de las palabras respecto a la imagen, el texto de esas palabras y el nivel de confianza. El código de la segunda línea es más simple. Le proporcionas la imagen y devuelve una cadena de caracteres con todo el texto contenido en la imagen. La información que obtenemos es menos elaborada que en el caso anterior, ya que perdemos la posición de cada palabra o la confianza con que esta se ha predicho. En general, perdemos toda la información relacionada con la imagen a la que pertenece el texto. Veamos un ejemplo de como usar las dos funcionalidades utilizando Python.

Empecemos con la más completa, la del diccionario. En la *figura 3.1.* se puede ver el código Python empleado y en la *figura 3.2.* el resultado obtenido para un bloque de texto.

```
img2 = img.copy() # Hacemos una copia para mostrar los resultados
results = pytesseract.image_to_data(img2, lang="spa", output_type=Output.DICT)
n = len(results["text"]) # = len(results["otra clave"])
for i in range(0,n):
  x = results["left"][i] # Esquina superior izquierda, coonrdenada x
  y = results["top"][i] # Esquina superior izquierda, coonrdenada y
  w = results["width"][i] # ancho de la bounding-box
  h = results["height"][i] # alto de la bounding-box
  text = results["text"][i] # texto
  conf = results["conf"][i]/100 # probabilidad

  if conf > 0.7: # Escogemos las predicciones con una probabilidad alta
    cv2.rectangle(img2, (x, y), (x + w, y + h), (0, 255, 0), 2) # Recuadrar
    cv2.putText(img2,text,(x,y-5),cv2.FONT_HERSHEY_SIMPLEX,0.6,(0, 0,255),2)
    # Escribimos el texto en la copia de la imagen
```

Figura 3.1: Código Python para detectar e identificar texto.

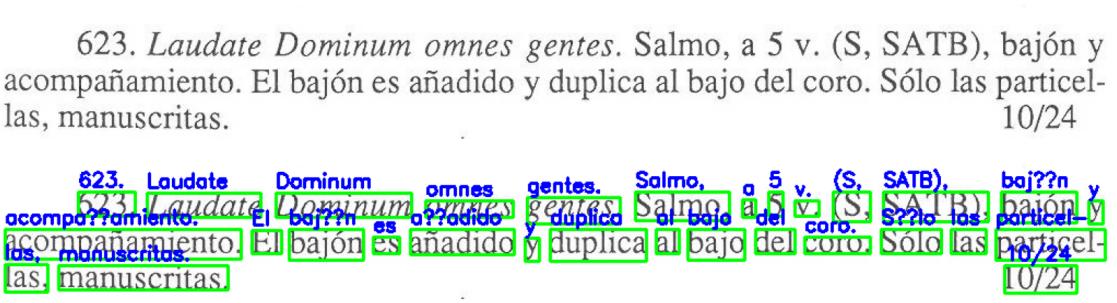


Figura 3.2: Resultado de aplicar *OCR* a un bloque de texto de la página 144 del libro [1].

Vemos que lo hace bastante bien. A partir de las coordenadas que nos devuelve el diccionario, hemos usado *OpenCV* para mostrar los resultados. Comentar que las interrogaciones que salen en la imagen anotada no hacen referencia a un error de identificación del texto, simplemente es que la librería de *OpenCV* no reconoce ni los acentos ni la letra ñ a la hora de escribirlos sobre ella. Para mejorar la detección hemos rechazado aquellas predicciones con una confianza menor que $0,7$.

El resultado de aplicar la segunda funcionalidad de *OCR* al mismo bloque de texto se puede ver en la *figura 3.3*. Se respetan las líneas del texto pero se pierde la noción de posición con respecto a la imagen. Dependiendo de cuáles sean nuestros objetivos nos va a ser más útil una u otra. Para hacer búsquedas dentro de las páginas del libro será preferible tener localizadas las palabras, por lo que será más útil la primera funcionalidad. En cambio, si lo que queremos es almacenar un pequeño párrafo en un archivo *XML*, seguramente sea más sencillo utilizar la segunda forma.

```
cadena = pytesseract.image_to_string(img, lang="spa")
print(cadena)
```

```
623. Laudate Dominum omnes gentes. Salmo, a 5 v. (S, SATB), bajón y
acompañamiento. El bajón es añadido y duplica al bajo del coro. Sólo las particel-
las, manuscritas. 10/24
```

Figura 3.3: Resultado de aplicar *OCR* al bloque de texto mostrado en la *figura 3.2*.

3.3. Almacenamiento de la información

En esta sección vamos a explicar la forma en la que hemos guardado los datos que detectamos en las páginas escaneadas. También, vamos a aprovechar para terminar de explicar cómo hemos detectado las obras. Comentar que no hemos hecho uso de ninguna base de datos para el almacenamiento, esto se debe a que, al tratarse de un único libro, el espacio que ocupa

toda la información es relativamente pequeño, por lo que hemos optado por almacenarla en ficheros *pickle* [10]. Si el tamaño de nuestros datos fuese más grande (tuviésemos muchos más libros) sí que deberíamos valorar la posibilidad de usar un gestor en nuestros datos para poder acceder rápidamente a la información.

Debemos tener en cuenta que el almacenamiento de los registros debe crearse con el fin de que sirva para poder hacer diferentes búsquedas dentro del libro, por lo que cada uno deberá albergar distinta información. Aprovechando que tenemos todas las páginas escaneadas, podemos basar el resto de la información no en imágenes, si no en coordenadas respecto al libro. Vamos a generar cuatro *diccionarios*. El primero contendrá información sobre los pentagramas y tendrá como finalidad acceder a ellos para posteriormente entrenar un modelo que sea capaz de leer la música que contiene, el segundo fichero almacenará el texto y su finalidad será la de realizar la clásica búsqueda de frases, y los dos últimos contendrán información sobre las obras, uno de ellos pensado para hacer búsquedas por autor y el otro para hacer búsquedas por tipo de obra.

Para almacenar los *pentagramas* utilizaremos un *diccionario* donde las claves sean el número de página. Dentro de ellas encontremos una lista con las coordenadas de todos los pentagramas que aparecen. Por ejemplo:

$$\text{pentagramas}[\text{“}pag_1\text{”}] = [(159, 809, 926, 117), (154, 990, 926, 117), (153, 1174, 926, 117)]$$

indicaría que hay tres pentagramas en la página 1. Las coordenadas están representadas en forma de: $(x, y, ancho, alto)$, donde x e y son las coordenadas de la esquina superior izquierda de la *bounding-box* del pentagrama. Para acceder a un pentagrama deberemos seleccionar la página a la que pertenece y utilizar las coordenadas para recortarla. Notar que todos los pentagramas de este libro tienen la misma altura y el mismo ancho. Las imágenes de los mismos vamos a usarlas en el siguiente capítulo para crear un modelo de lectura de símbolos musicales.

El *texto* lo guardaremos utilizando la información que nos devuelve la primera funcionalidad de *pytesseract* explicada en la *figura 3.1*. Recordar que en ella se encontraban todas las palabras detectadas con sus respectivas coordenadas respecto a la imagen. Este diccionario nos va a servir para realizar búsquedas dentro del libro. Por ejemplo, para acceder a la cuarta palabra (empezando desde arriba a la izquierda) detectada de la página 144 deberemos hacer una búsqueda de la siguiente forma:

$$\text{texto}[144][\text{“texto”}][4] = \text{“Manuel”}$$
$$\text{texto}[144][\text{“coordenadas”}][4] = (203,399,103,22)$$

En este caso nuestro diccionario es “*texto*”, el cual tiene como claves las páginas del libro. Al acceder a una clave (página), vemos que obtenemos, a su vez, otro diccionario donde las

claves son "*texto*" y "*coordenadas*". Estas dos claves albergan una lista con las palabras y sus coordenadas respectivamente. De forma indirecta podemos obtener más información como, por ejemplo, la fila a la que pertenece la palabra, ya que podemos agrupar las palabras con coordenadas "y" cercanas.

Para almacenar las *obras* tenemos varias posibilidades, ya que podemos clasificarlas de muchas formas. Una opción podría ser agruparlas por capítulos y almacenar las coordenadas como venimos haciendo hasta ahora, otra opción podría ser almacenarlas directamente por la página en la que se encuentran. No es la opción más eficiente ya que, en este caso, los capítulos o la página a la que pertenece una obra, no es una información demasiado completa. Podemos encontrar variables más ricas en información como el autor o el tipo de obra. Cuando almacenábamos los pentagramas, lo que queríamos era poder acceder a ellos fácilmente para posteriormente entrenar un modelo de lectura de símbolos musicales y cuando almacenábamos el texto el objetivo era poder hacer búsquedas clásicas dentro del libro. En el caso de las obras, tiene más sentido hacer búsquedas por autor o tipo de obra para ser lo más fieles a la redacción del libro. Podemos dividir el problema en tres partes, el primero de ellos será detectar el bloque completo donde se encuentra una obra, el segundo determinar a qué autor pertenece y el último decidir de qué tipo de obra se trata.

Como comentábamos anteriormente, podemos ayudarnos de la estructura del libro para detectar las obras. Aprovecharemos el texto almacenado previamente para hacer un recorrido por todas las páginas e ir anotando "puntos clave". Entendemos por "punto clave" una zona de la página que nos puede ayudar a marcar el inicio y/o el final de una obra. De esta manera, una obra comenzará en un punto clave y terminará en el siguiente. La mayoría de estos puntos claves se obtienen al hacer un recorrido iterativo en el libro en donde se vaya anotando el nº de obra. Algunas veces las obras comienzan en una sección nueva donde hay un pequeño título con el nombre de algún autor, tipo de obra o directamente el inicio de un nuevo capítulo. La implementación de esta funcionalidad se encuentra en el cuaderno "Detectando_Obras" del repositorio.

En la *figura 3.4.* podemos ver el resultado de localizar los puntos clave de las páginas para la detección de las obras. Están recuadrados en verde en la imagen de la izquierda. Recordar que una obra empieza en un punto clave y termina en el siguiente. Comentar que las coordenadas de todas las obras se han almacenado con un mismo ancho. El motivo es que, como ya dijimos, para mostrar algunas de las obras deberemos concatenar dos imágenes distintas, ya que comienzan en una página y terminan en la siguiente. Un claro ejemplo se puede apreciar, también, en la *figura 3.4.* Podemos ver que la obra nº 620 termina en la página 144 y, en este caso, comienza en la página anterior. En la *figura 3.5.* se puede comprobar el resultado que se obtiene al combinar el principio y final de dos páginas para mostrar la obra nº 620 completa.

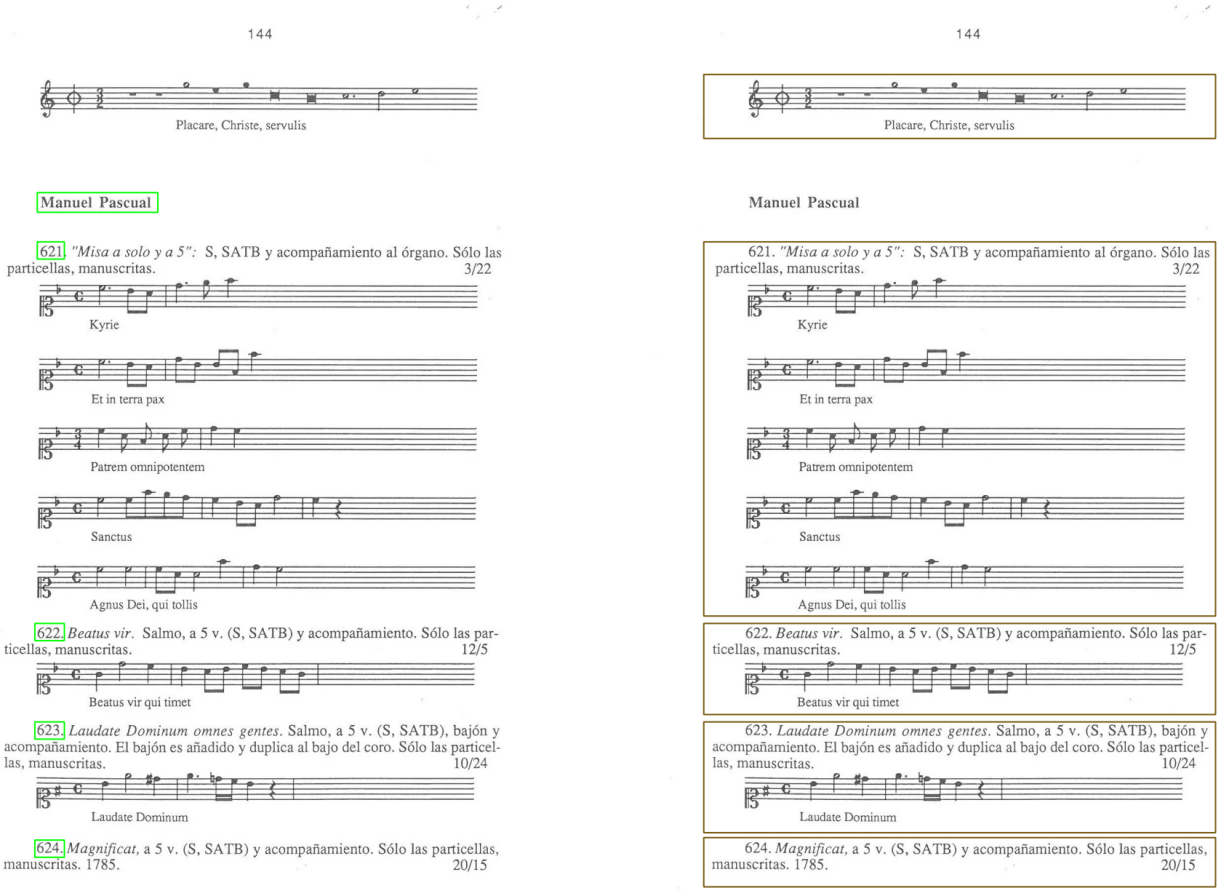


Figura 3.4: Proceso de localización de las obras del libro.

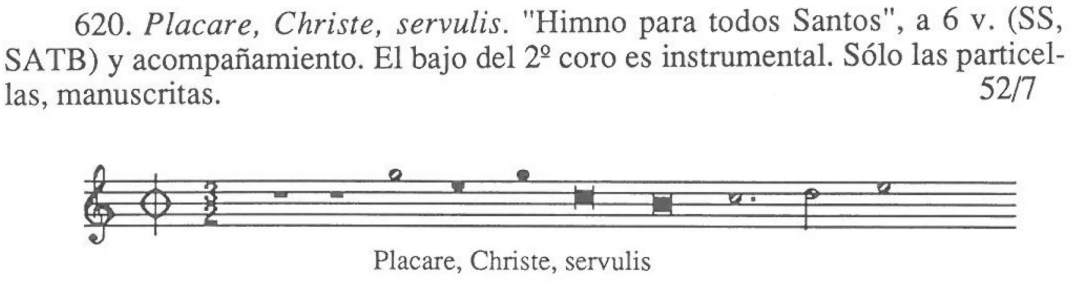


Figura 3.5: Obra construida a partir de concatenar dos páginas.

Para determinar el autor o el tipo de obra nos ayudaremos una vez más de la estructura del libro. Hay capítulos o secciones dedicados completamente a ellos. Para la página 144, la cual llevamos usando como ejemplo hasta ahora y que podemos ver su última aparición en la *figura 3.3*, vemos que dedican una sección a las obras del autor *Manuel Pascual*, por lo que a partir de ese momento todas las obras pertenecerán a ese autor. Lo mismo pasa con los tipos de obra, hay secciones dedicadas enteras a *Misas* o *Villancicos*. En última instancia,

si no conseguimos clasificar las obras de ninguna forma podemos utilizar que, en ocasiones, aparecen nombres de autores y de tipos en la descripción de la obra. Volviendo a la página 144, vemos que podemos clasificar la obra nº 622 como un *Salmo*. En cualquier otro caso diremos que son de autor *anónimo* o con tipo *otro*.

Por último, para guardar toda esta información, hemos optado por construir dos diccionarios donde las claves son los autores y los tipos de obra. Al acceder a un autor obtenemos como resultado otro diccionario donde las claves son sus obras y al acceder a una obra obtenemos finalmente lo siguiente:

Obras[“Manuel Pascual”][622] = {Coordenadas: [(80, 1263, 1080, 193)], Tipo: [“Salmo”], p: 144}

En las coordenadas aparecerán más valores si la obra está en varias páginas. También hemos dado la posibilidad de que una obra tenga varios tipos (lo mismo con los autores). La información del diccionario estructurado por tipos de obra es análogo al anterior:

Obras[“Salmos”][622] = {Coordenadas: [(80, 1263, 1080, 193)], Autor: [“Manuel Pascual”], p: 144}

Si buscamos por autor, es mucho más rápido utilizar el primer diccionario y si buscamos por tipo de obra, el segundo. Podemos combinar estas dos bases de datos para hacer búsquedas del tipo “Mostrar los Salmos de Manuel Pascual”. En este caso nos aparecería un diccionario de obras con la siguiente estructura:

Obras[“Manuel Pascual”][“Salmos”][622] = {Coordenadas: [(80, 1263, 1080, 193)], p: 144}

Por último, comentar que en el cuaderno de *Jupyter* de nuestro proyecto con nombre “*Clasificando_Obras*” se puede encontrar todo el proceso de clasificación.

3.4. Realizando búsquedas

En esta sección explicaremos cómo aprovechar todo lo empleado anteriormente para poder realizar búsquedas. Dependiendo de qué información queramos mostrar, utilizaremos un tipo de diccionario u otro.

Lo primero que podemos hacer es realizar la clásica búsqueda de texto. Construiremos un algoritmo que, dada una frase, nos devuelva su localización en todas las páginas donde aparece. La idea es recorrer las palabras de una misma fila y juntarlas para ver si forman la frase dada. Para mejorar el programa hemos descartado los acentos, los puntos o cualquier otro símbolo de puntuación. También, hemos puesto todas las letras en minúscula. A esta técnica se le suele denominar *normalización* de texto. Hemos dejado que las frases puedan estar inacabadas y los saltos de línea. Así, por ejemplo, la palabra “Manuel” nos aparecerá localizada aunque la busquemos por “mänué”. La función que hemos utilizado para descartar todos estos símbolos la podemos encontrar en la *figura 3.6*.

```
def remove_punctuation(frase): #Fución para "normalizar" frase
  char=re.sub("[%s]"% re.escape(string.punctuation.replace("-","")),"",frase)
  char=char.lower() #Minúsculas
  c = "([^n\ u0300- u036f]|n(?!\ u0303(?![\ u0300-\ u036f])))[\ u0300-\ u036f]+"
  char=re.sub(r c,r"\1",normalize( "NFD", char),0,re.I)
  char=normalize("NFC",char)
  return char

remove_punctuation("PriMër AÑò dÉ MÄsteR")
```

```
"primer año de master"
```

Figura 3.6: Función para normalizar cadenas de caracteres.

Mostramos el resultado de buscar la frase "sólo las particellas" en un trozo de la página 144 en la *figura 3.7.*

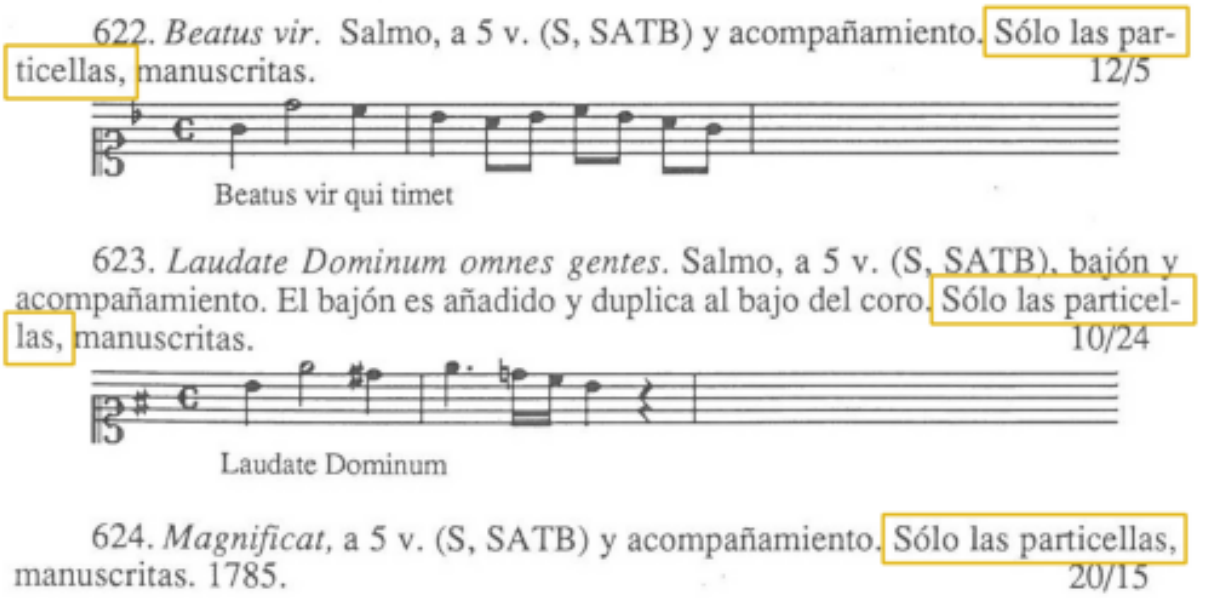


Figura 3.7: Búsqueda clásica dentro del libro.

El código empleado para hacer búsquedas clásicas lo podemos encontrar en el cuaderno de nuestro proyecto con nombre "*Primera_Web*". Para hacer búsquedas por autor o tipo de obra, deberemos tener en cuenta una salvedad más. Los autores a veces aparecen nombrados de distintas formas, en concreto pueden aparecer como: *Nombre Apellido*, *Apellido Nombre*, *Nombre*, *Apellido* o algún mote en concreto. En la *figura 3.8.* se puede ver el código que se ha usado para hacer búsquedas de obras por autor. También sirve para comprobar el uso que le estamos dando a los diccionarios que hemos definido antes.

```
def buscar(autor): #autor es una cadena de caracteres
#Obras = diccionario con la información de las obras explicado antes.
  if autor not in Obras.keys(): print("Ese autor no existe.")
  else:
    print("Aquí están todas las obras de " + autor + ":")
    print("")
    for i in Obras[autor]: #Recorremos todas sus obras
      Foto = [] #En esta lista guardaremos los trozos de la obra
      pag = Obras[autor][i]["pag"] #Página donde empieza la obra
      for coor in Obras[autor][i]["coordendas"]) #Recorremos las coords.
        x,y,w,h = coor                           #Generalmente solo hay una.
        Foto.append(Pagina[pag][y:y+h,x:x+w]) #En Pagina están todas las imgs.
        #Lo que estamos haciendo es quedarnos con el trozo de obra.
        pag=pag+1
      Foto = np.concatenate(Foto,axis=0) #Concatenamos para crear la imagen.
      T="Tipos: "+Obras[autor][i]["Tipo"]+" "+"Pagina: "+Obras[autor][i]["pag"]
      cv2.putText(Foto,T,(15,15),cv2.FONT_HERSHEY_TRIPLEX,0.65,(0,0,0),1)
      cv2.rectangle(Foto,(0,0),(30,30),(119,90,17),2)
      #En T guardamos el resto de la información y luego la escribimos
      fig, ax = plt.subplots(1,1,figsize=(16,16))
      ax.imshow(Foto,cmap=plt.cm.binary) #Mostramos la obra.
  plt.show()
```

Figura 3.8: Función para buscar obras por autor.

En el código de la *figura 3.8.* la cadena de caracteres "autor" ha sido normalizada y puesta en una forma estándar antes de pasar por la función. Recordar que dijimos que los autores pueden aparecer de varias formas en el texto. Lo que hemos hecho es transformar esa cadena a la forma: *apellido nombre*. En la *figura 3.9.* podemos ver el resultado de mostrar todas las obras del autor *Ignacio Cardenal*. Antes de pasar al siguiente capítulo, es importante comentar que para evaluar el funcionamiento de nuestros algoritmos, hemos hecho una revisión manual de los resultados. El primer motivo es que el *Tesseract OCR* es una tecnología muy validada y ofrece, en general, unos resultados muy buenos. Los pequeños fallos que hayamos podido cometer, por ejemplo, al detectar una obra o reconociendo una palabra de forma incorrecta, han sido corregidos a mano, aprovechando que solo contamos con un libro. En resumen, los resultados han sido bastante buenos y la forma de comprobarlo ha sido manualmente. Si tuviésemos el libro anotado, hubiésemos podido calcular *métricas* que nos hubiesen permitido medir matemáticamente lo bien que funcionan las distintas técnicas, pero esto no es factible en este contexto, ya que supondría transcribir de forma manual el libro y es justo lo que

se quiere evitar con este proyecto. Nuestro próximo objetivo será encontrar una forma de codificar la música disponible dentro de las obras.

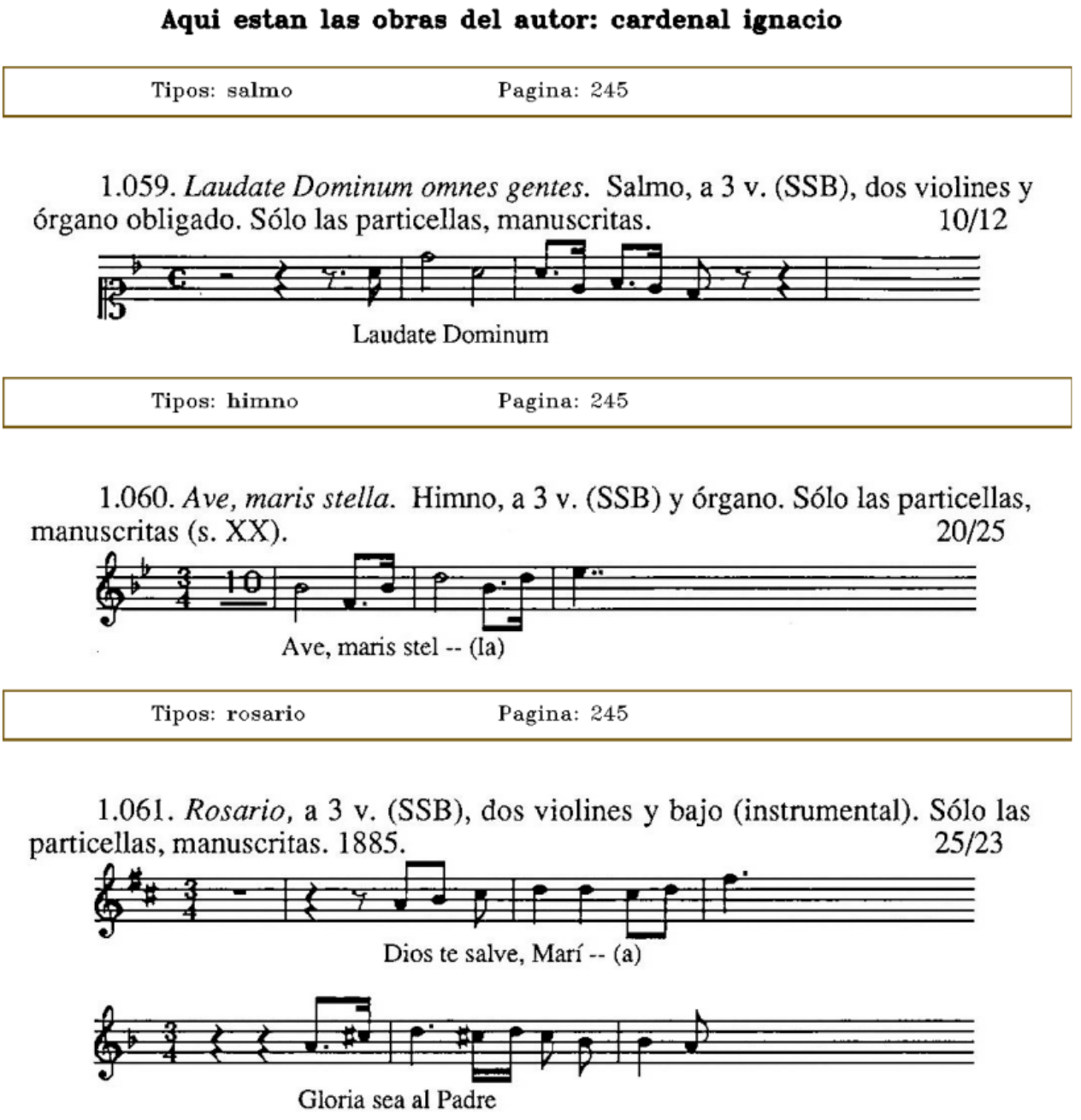


Figura 3.9: Búsqueda de obras por autor.

Capítulo 4

OMR

La labor en la que nos centraremos en este capítulo es la siguiente: dada una obra del libro, leer los símbolos musicales que contiene para ser capaces de representar la música. Para ello no ha sido posible utilizar librerías existentes sino que hemos tenido que crear nuestros propios modelos de *OMR*.

4.1. Introducción al OMR

El *OMR*, del inglés *Optical Music Recognition* y traducido al español como *Reconocimiento Óptico de Música*, es una tecnología la cual tiene como finalidad reconocer la música contenida en ficheros con formato de imagen. Al igual que el *OCR*, vuelve a tratarse de un problema de visión por computador [11]. Cabe destacar que esta tecnología no está tan consolidada como el *OCR* y es por eso que hay muchos problemas abiertos y no es fácil encontrar programas que funcionen en todos los casos [12, 13].

A la hora de almacenar el texto extraído del libro, utilizamos el clásico formato de cadena de caracteres, el cual nos permite recuperar la información con facilidad y mostrar los resultados obtenidos. En el caso de la música, existen muchos formatos que nos van a permitir codificarla [14]. Algunos de los más usados son: una imagen, *MIDI*, *Mp3* o *MusicXML*. Cabe destacar que cada formato sirve para cosas diferentes, ya que la música permite representaciones tanto visuales como sonoras, por lo que deberemos buscar aquel que sea capaz de cubrir mejor nuestras necesidades. Expliquemos brevemente en qué se basan cada uno de ellos, aunque antes, vamos a aclarar que utilizaremos la palabra *conversor* para referirnos a aquel sistema capaz de transformar un tipo de formato en otro, de manera única, sin ambigüedades ni errores. Así, por ejemplo, el *OCR* no lo consideramos un conversor de imagen a texto ya que es un modelo de aprendizaje automático que comete errores. Consideraremos conversor, por ejemplo, un programa que nos transforme las imágenes en formato *BGR* al formato *RGB*, ya que es una transformación matricial *biyectiva* (una imagen, tiene una sola transformación posible y en este caso reversible, aunque esto último no es necesario).

Formato imagen. Tenemos la música representada en una o varias imágenes en formatos como pdf, jpg o png. La forma de representarla es utilizando la notación universal de pentagramas y símbolos musicales. Esta es la forma que disponemos en nuestro libro. No es compatible con los demás formatos, es decir, no existen, por ejemplo, conversores de una imagen a sonido.

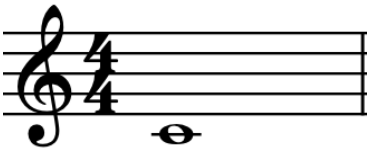
MIDI. Es una secuencia de *eventos* donde se almacenan, entre otras cosas, el tono, la duración o el instrumento que interpreta una nota musical [15]. Una de sus finalidades es la de reproducir sonido. En la *figura 4.1.* se muestra una representación de un pentagrama de nuestro libro en formato *MIDI*. Es bastante intuitivo, solamente hay que indicar cuándo empieza y termina una nota (un tono, columna *Pitch*), con qué fuerza va a sonar (columna *Velocity*) y el instrumento que interpreta la melodía. Vemos que, en este caso, hay 7 *eventos*.

	Start	End	Pitch	Velocity	Instrument
0	0.00	0.50	69	90	Piano
1	0.50	0.75	71	90	Piano
2	0.75	1.00	72	90	Piano
3	1.00	1.25	71	90	Piano
4	1.25	1.50	69	90	Piano
5	1.50	2.25	69	90	Piano
6	2.25	2.75	68	90	Piano

Figura 4.1: Un pentagrama de nuestro libro representado en formato MIDI.

Mp3. Sirve para reproducir sonido y seguramente sea el formato más conocido. La diferencia que tiene con el *MIDI* es que el *Mp3* sí que es audio y no una secuencia de *eventos*. Además, el *Mp3* es un archivo de audio comprimido, que se recupera mediante un algoritmo con pérdida al igual que pasa, por ejemplo, con el formato de imagen jpg.

MusicXML. Es un archivo *XML* [16, 17, 18]. Es más difícil de leer que el formato *MIDI* pero también almacena más información. Los datos se organizan en estructura de árbol y, además de música, se puede añadir texto. Requiere de una escritura bastante más extensa y para el pentagrama de la *figura 4.1.* necesitaríamos muchas líneas de código. No obstante, en la *figura 4.2.* se puede ver un ejemplo de este tipo de formato, de hecho, se conoce conoce como el "Hello World" de *MusicXML*.



```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE score-partwise PUBLIC
    "-//Recordare//DTD MusicXML 3.0 Partwise//EN"
    "http://www.musicxml.org/dtds/partwise.dtd">
<score-partwise version="3.0">
  <part-list>
    <score-part id="P1">
      <part-name>Music</part-name>
    </score-part>
  </part-list>
  <part id="P1">
    <measure number="1">
      <attributes>
        <divisions>1</divisions>
        <key>
          <fifths>0</fifths>
        </key>
        <time>
          <beats>4</beats>
          <beat-type>4</beat-type>
        </time>
        <clef>
          <sign>G</sign>
          <line>2</line>
        </clef>
      </attributes>
      <note>
        <pitch>
          <step>C</step>
          <octave>4</octave>
        </pitch>
        <duration>4</duration>
        <type>whole</type>
      </note>
    </measure>
  </part>
</score-partwise>
```

Figura 4.2: Hello World de MusicXML junto con su representación en forma de imagen.

En [19] se puede acceder a la página oficial de *MusicXML*. En ella encontraremos una variedad de ejemplos y tutoriales de cómo utilizar este tipo de codificación. También es interesante consultar [20] para comprobar los diferentes nombres que se le dan a todas las notas musicales tanto en este formato como en *MIDI*.

Comentar que hemos puesto cuatro formatos que representan la música de formas diferentes. En la primera de ellas contábamos con *imágenes*, en la segunda con una lista de *eventos MIDI*, en la tercera teníamos *sonido* y en la última una secuencia de bloques y figuras dentro de un archivo *XML*. Notar que tanto las imágenes como el sonido son *datos no estructurados*, por lo que no existen métodos de conversión directa a *MIDI* o *MusicXML*, los cuales albergan información en forma de *dato estructurado*. A su vez, tampoco podemos convertir imágenes a sonido ni viceversa. Pensar que tanto una imagen como un archivo de sonido pueden contener otro tipo de información diferente a la musical, entonces ¿qué hacer en esos casos? También, puede darse el caso de que, a pesar de contar con un programa que traduzca, por ejemplo, imágenes a *MIDI*, a una misma imagen a la que solamente hayamos añadido algo de ruido, nos devuelva cosas diferentes. Es por eso que hemos descartado todo este tipo "conversiones" anteriormente, quedándonos únicamente con las que seguro podrán hacerse en todos los casos y devolverán siempre el mismo resultado. El formato *MIDI* puede convertirse a sonido pero no a imágenes. En la *figura 4.3.* podemos encontrar dos imágenes distintas que representan la misma nota y, en consecuencia, tienen el mismo archivo *MIDI*. Un archivo *MIDI* tiene más de una representación posible en forma de imagen.

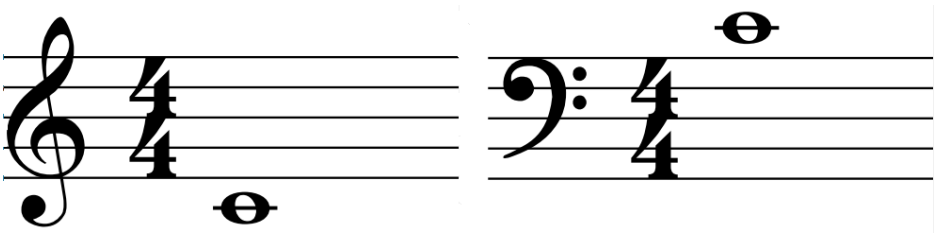


Figura 4.3: Cuarta octava de la nota Do escrita en Clave de Sol y Clave de Fa en cuarta. Ejemplo de cómo una misma nota musical puede ser escrita de dos formas diferentes.

Por el mismo motivo no podemos convertir un archivo MIDI a uno MusicXML, ya que podríamos escribir dos archivos MusicXML distintos que representasen la misma nota musical. Por ejemplo, podríamos escribir dos archivos diferentes que representasen cada una de las imágenes de la *figura 4.3.*

Por último, comentar que MusicXML se puede convertir en imagen, sonido o MIDI. Por lo que es el formato más completo de los cuatro tipos explicados. En la *figura 4.4.* se puede ver un esquema con los distintos tipos de conversiones permitidas.

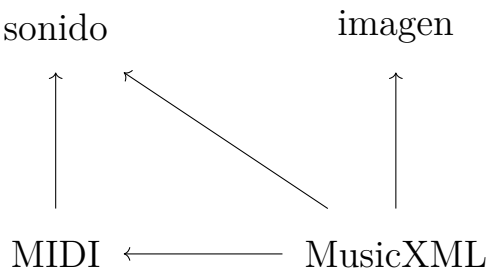


Figura 4.4: Conversiones entre formatos permitidas.

En Internet se pueden encontrar fácilmente páginas donde se hacen estas conversiones. En la *figura 4.5.* se muestran algunas de ellas utilizando las librerías de Pyhton *music21* [21] y *pretty_midi* [22].

```
# Abrimos el archivo MusicXML Transformamos a MIDI y lo guardamos en archivo.mid
c = music21.converter.parse("archivo.xml")
c.write("midi","archivo.mid")

# Mostramos la imagen que genera el archivo MusicXML
c.show()

# Transformamos el archivo MIDI a sonido
midi_data = pretty_midi.PrettyMIDI("archivo.mid")
Fs = 22050 # Frecuencia del sonido
audio_data = midi_data.synthesize(fs=Fs)
ipd.Audio(audio_data, rate=Fs)
```

Figura 4.5: Algunas conversiones utilizando Python.

Una vez hemos hablado de los diferentes formatos para representar la música, deberemos buscar la mejor forma de codificar nuestras obras. A priori, la mejor de todas es la de *MusicXML*, ya que podemos acceder a los demás formatos fácilmente. No obstante, dependerá de los propósitos del problema en cuestión y de otro tipo de factores.

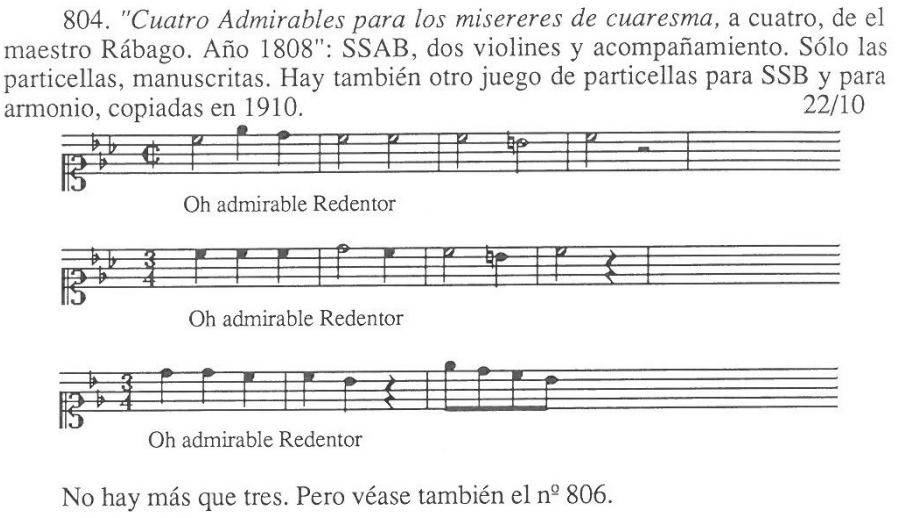
En nuestro caso concreto contamos con las obras en formato imagen, así que tal vez sea suficiente con conseguir sacar el sonido que representan. A pesar de ello, puede ser interesante el formato *MusicXML* para poder reconstruir las imágenes y de esa forma quitarnos el ruido procedente del escaneo. Es importante recordar que no existen conversores de imagen a sonido o *MusicXML*, por lo que si queremos hacer ese salto deberemos utilizar otro camino. La primera posibilidad podría ser anotar a mano todas y cada una de las obras en alguno de los formatos, esta opción vamos a descartarla ya que llevaría demasiado tiempo teniendo en cuenta que hay 1553 obras repartidas en casi 400 páginas y, además, es justo lo que queremos evitar en este proyecto. También es la opción menos interesante de todas y la que menos

conocimientos nos va a aportar. Otra posibilidad es buscar en la literatura qué programas existen y ver como se comportan en nuestro libro. En última instancia, si nada de lo anterior funciona, se creará un modelo propio para solucionar el problema. En la siguiente sección investigaremos sobre las diferentes herramientas que hay disponibles en la literatura.

4.2. Modelos en la literatura

Nuestro siguiente objetivo es buscar las diferentes opciones que hay en Internet para la realización de *OMR*. Un buen punto de partida para comenzar es la bibliografía disponible sobre OMR en [23]. También, hemos buscado en *GitHub* diferentes *topics*. En concreto, hemos buscado: *OMR*, *optical-music-recognition* y *optical music recognition* [24]. Nuestro objetivo es el siguiente: dada una obra de una página escaneada, devolver la música que se representa, a poder ser en formato *MIDI* o *MusicXML*.

Antes de mostrar lo que hemos encontrado, es interesante tener una mínima noción de cómo leer la música (sin entrar en los detalles de cada símbolo) de nuestras obras, eso nos facilitará las cosas a la hora de buscar qué nos puede servir. En la *figura 4.6.* podemos encontrar la obra 804 de nuestro libro. Para leer la música que hay en ella deberemos ir pentagrama por pentagrama, empezando por el de más arriba. Esto es algo recurrente en todo el libro. En otro tipo de melodías puede darse la situación de que más de un pentagrama suene a la vez, esto se conoce como el *número de canales* de la canción y es algo que *MusicXML* o *MIDI* tienen en cuenta. En nuestro caso, siempre va a haber un canal, así que sacando la música de cada pentagrama por separado y uniendo, es suficiente.


804. *"Cuatro Admirables para los misereres de cuaresma,* a cuatro, de el maestro Rábago. Año 1808": SSAB, dos violines y acompañamiento. Sólo las particellas, manuscritas. Hay también otro juego de particellas para SSB y para armonio, copiadas en 1910. 22/10

Oh admirable Redentor

Oh admirable Redentor

Oh admirable Redentor

No hay más que tres. Pero véase también el nº 806.

Figura 4.6: Obra 804 del libro "La Música en la Catedral de Santo Domingo de La Calzada".

Para leer un pentagrama hay que ir de izquierda a derecha. Los primeros símbolos son muy importantes, en ellos podemos encontrar información acerca de la *clave*, que nos dice

qué notas deben aparecer en las líneas horizontales (ver la *figura 4.3*, en ella está la misma nota en dos claves diferentes ocupando posiciones diferentes en el pentagrama), la *armadura*, la cual nos da la *tonalidad* de la melodía, y el *compás*, el cual nos indica el número de *tiempos* que caben en cada línea del pentagrama, por ejemplo, si un tiempo es 1 segundo y el compás es 3×4, entonces en cada línea vertical caben 3 segundos. Los símbolos que van después del *compás* suelen ser *notas* o *silencios*, dependiendo del tipo durarán más o menos, por ejemplo, una negra dura un tiempo y una corchea medio. Estos símbolos se leen de izquierda a derecha hasta llegar al final. En la *figura 4.7.* podemos encontrar todas estas explicaciones de forma visual.

Cabe resaltar que no entra dentro de los objetivos de esta memoria explicar minuciosamente el procedimiento para leer pentagramas musicales. Simplemente quedarse con la idea de que hay una serie de símbolos ordenados de izquierda a derecha y que hay que saber clasificarlos para saber qué nos quiere decir cada uno para así construir la melodía.

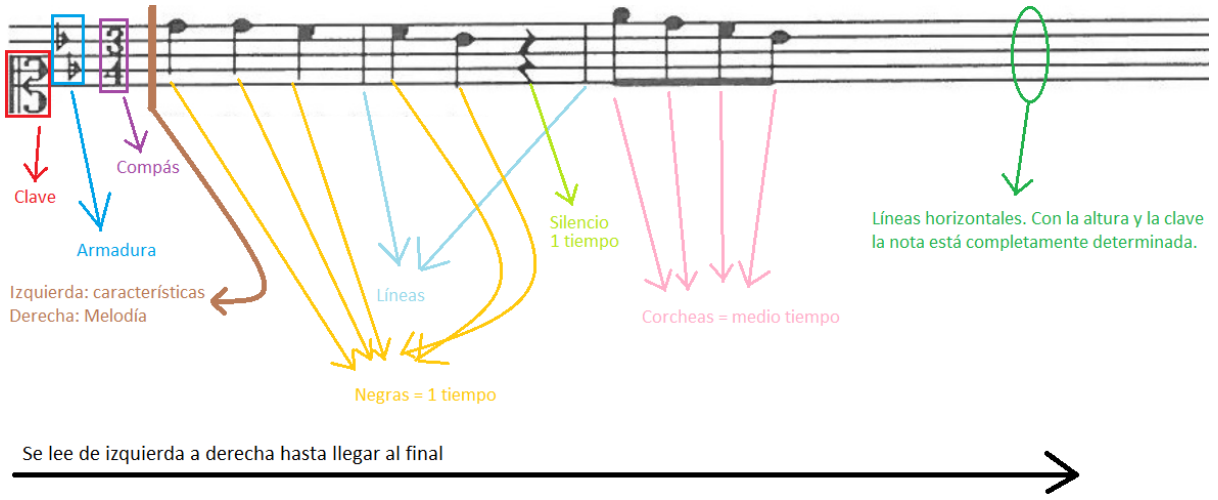


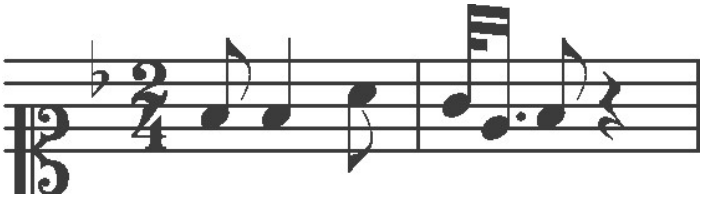
Figura 4.7: Ejemplo de cómo leer y localizar símbolos en un pentagrama de la obra 804 del libro.

Entonces, ¿qué tipo de programas nos pueden servir? para empezar, no nos afecta mucho si es capaz de sacar la música de una obra completa o si solamente sabe leer pentagramas. En el Capítulo 2 vimos cómo detectarlos, por lo que, si es necesario, los detectaremos. Preferiblemente buscaremos programas que nos devuelvan *MIDI* o *MusicXML*, aunque si devuelven cualquier otra notación podemos ajustarla a nuestras necesidades. Lo más importante es que funcione correctamente en nuestro libro. Cabe resaltar que, al igual que nos pasaba en los capítulos anteriores tanto con el *OCR* como con *OpenCV*, no vamos a ser capaces de evaluar los resultados con métricas, ya que no contamos con ningún tipo de anotación en nuestro libro. También es probable que el tipo de anotación cambie bastante dependiendo del programa que utilicemos. En consecuencia, la evaluación de los modelos que probemos de la literatura la volveremos a realizar de manera manual.

Mencionar que, a pesar de haber unos cuantos programas que hacen lo que teníamos en mente, a la hora de la verdad no responden bien ante las imágenes de nuestro libro. La explicación que he encontrado es que los *datasets* (conjunto de datos, ejemplos para "enseñar" a los modelos) utilizados para el entrenamiento, no son del todo buenos y, a la hora de crear un modelo, va a estar muy sesgado por esos datos y no va a generalizar bien. En *GitHub* podemos encontrar muchos programas, tanto gratuitos como de pago, donde podemos realizar pruebas [25]. También hay disponibles *datasets* para poder crear nuestros propios modelos o que han sido utilizados para entrenar algunos modelos ya creados [26].

Los programas de pago y/o de código cerrado los hemos descartado. De entre el resto de modelos que hemos encontrado, la mayoría de ellos devuelven o bien anotaciones propias, o bien formato *MIDI*, *MusicXML* o las tres cosas y funcionan bastante mal en nuestro libro, por lo que no vamos a mostrar ninguna prueba realizada ya que no merece la pena, no son capaces de reconocer nada en nuestras obras. Algunos de ellos cuentan hasta con artículos publicados como es el caso de [27], el cual esta basado en un modelo de detección y devuelve la música que alberga una imagen en formato *MIDI*, también cuenta con un repositorio de *GitHub* [28]. Otro modelo similar al de antes podría ser [29]. Es cuestión de indagar un poco en los distintos programas que hay disponibles. Lo bueno de tener el código abierto, es que puedes entender perfectamente el procedimiento que se ha seguido y puede llegar a servirte de referencia si es que fuera necesario. A pesar de todo lo mencionado hasta ahora, comentar que sí hemos encontrado dos programas que funcionan relativamente bien. Además, son capaces de generalizar bastante ya que, realizando pruebas, ya no solo con nuestro libro, si no con algunas partituras musicales tomadas de internet, ofrecen unos resultados más que correctos. Ambos son de código abierto y utilizan técnicas bastante distintas para llegar al resultado final. Es por eso que merecen una mención antes de pasar a la siguiente sección.

El primero de ellos es un modelo *End-to-End* [30]. Este tipo de modelos se basan en realizar el proceso entero, sin dejar ningún paso final o previo al usuario. El artículo donde se explican las funcionalidades de este modelo de *OMR* lo podemos encontrar en [31]. También cuenta con un repositorio en *GitHub* donde puede verse el código empleado y un resumen del artículo anterior [32]. En el repositorio de nuestro proyecto, hemos realizado algunas pruebas con nuestro libro, para ello, hay que acceder al cuaderno de *Jupyter* con nombre "*tf_deep_omr*". El modelo lee un pentagrama y devuelve una lista con las características de la canción y todos los símbolos detectados. Por último, comentar que para entrenar el modelo *End-to-End* de [31], el autor ha utilizado el *PrIMuS dataset* [33], el cual está anotado en formato *XML* y es una anotación propia que no coincide con la usada en *MusicXML*. También, se ofrece un pequeño programa para transformar esas anotaciones a *MIDI*. En la *figura 4.8.* se puede ver un pentagrama de este *dataset*.



```
#Aquí iría el cuerpo del xml. Lo descartamos ya que lo importante viene después.
<music>
  <body>
    <mdiv>
      <score>
        <scoreDef xml:id="scoredef-1" key.sig="1f" meter.count="2" meter.unit="4">
          <staffGrp xml:id="staffgrp-1">
            <staffDef xml:id="staffdef-1" clef.shape="C" clef.line="1" n="1"/>
          </staffGrp>
        </scoreDef>
        <section xml:id="section-1">
          <measure xml:id="measure-1" right="single">
            <staff xml:id="staff-1" n="1">
              <layer xml:id="layer-1" n="1">
                <note xml:id="note-1" dur="8" oct="4" pname="f"/>
                <note xml:id="note-2" dur="4" oct="4" pname="f"/>
                <note xml:id="note-3" dur="8" oct="4" pname="a"/>
              </layer>
            </staff>
          </measure>
          <measure xml:id="measure-2" right="single">
            <staff xml:id="staff-2" n="1">
              <layer xml:id="layer-2" n="1">
                <beam xml:id="beam-1">
                  <note xml:id="note-4" dur="32" oct="4" pname="g"/>
                  <note xml:id="note-5" dots="1" dur="16" oct="4" pname="e"/>
                </beam>
                <note xml:id="note-6 dur="8" oct="4" pname="f"/>
                <rest xml:id="rest-1" dur="4"/>
              </layer>
            </staff>
          </measure>
        </section>
      </score>
    </mdiv>
  </body>
</music>
```

Figura 4.8: Pentagrama del *PrIMuS dataset* junto con su anotación.

Aunque el código *XML* de la *figura 4.8.* pueda parecer extenso, es bastante más compacto que el de *MusicXML*. En cuanto a los resultados con nuestros pentagramas, son relativamente buenos, pero se deja frecuentemente cosas importantes de localizar, como pueden ser la *armadura* o la *clave*, lo que provoca que la canción cambie totalmente. Aun así, es un modelo a tener en cuenta que se puede combinar con el *OCR* visto en el capítulo anterior para codificar a *MusicXML* nuestras obras (recordar que también se puede añadir texto). En la *figura 4.9.* podemos ver una predicción en un pentagrama de nuestro libro, primero se muestra la imagen y después la lista de anotaciones que devuelve el modelo. La notación esta basada en la de *MusicXML* y reconoce todo bien a excepción del *compás*, que dice que es un 3×2 y es un 3×8, y el silencio de corchea del final, que dice que es una nota.

```
['clef-G2', 'keySignature-BbM', 'timeSignature-3/2', 'note-A4_quarter', 'note-Bb4_eighth', 'barline', 'note-C5_eighth',
'note-Bb4_eighth', 'note-A4_eighth', 'barline', 'note-A4_eighth.', 'barline', 'note-G#4_half', 'note-C5_double_whole.',
'barline']
```

Figura 4.9: pentagrama de libro "La Música en la Catedral de Santo Domingo de la Calzada" junto con la predicción de [31].

Hay que tener en cuenta que con este modelo vamos a tener que programar la transformación de una obra a *MIDI* o *MusicXML*. Tal como devuelve los resultados, nos saldría mejor hacer una traducción a *MusicXML*, ya que nos llevaría el mismo trabajo y obtendríamos una codificación con más información (y compatible con el *MIDI*). El proceso que habría que seguir es el siguiente: detectar los bloques de texto y aplicarles *OCR*, detectar los pentagramas y calcular las predicciones del modelo y, por último, escribir toda esa información en *MusicXML*. Hemos valorado no llevar a cabo todo este trabajo ya que el modelo no es todo lo bueno que nos gustaría y tendríamos que realizar muchas correcciones de forma manual. Si funcionase un poco mejor sería sin duda un candidato a tener en cuenta.

El segundo programa que vamos a mostrar se llama *Audiveris* [34]. Es de código abierto, programado en Java y compatible con los sistemas operativos Windows, Mac y Linux. Nosotros lo hemos usado desde una interfaz descargable desde la página oficial, aunque también puede utilizarse desde la consola de comandos. Aparte de generalizar bastante bien, lo bueno que tiene este modelo es que aplica *OCR* y *OMR* a la vez. Es decir, es un sistema que combina ambas tecnologías para reconocer tanto texto como música dentro de imágenes. La versión que hemos utilizado durante la realización del proyecto es la 5.1 aunque hace poco, el 4 de junio de 2021, fue lanzada su última versión, la 5.2 aunque los cambios con la versión anterior no han sido significativos.

Tras haber procesado la partitura y haber reconocido todos los símbolos, la interfaz nos permite corregir errores a mano antes de guardar el resultado final, el cual puede guardarse

con una extensión propia, en formato *MusicXML* o directamente como una imagen reescrita sin el ruido del escaneo. En la *figura 4.10.* podemos ver la obra 804 de nuestro libro procesada, de hecho, es una captura de pantalla de una parte de la interfaz de *Audiveris*. En ella podemos seleccionar con el ratón las diferentes zonas y cambiar resultados en la localización de simbólos, ya sea para corregir alguna letra mal reconocida, una nota, etc. Vemos que el programa superpone los símbolos predichos, con la parte de la imagen donde cree que están, esto se debe a que parte de este programa (y aunque no lo hayamos dicho anteriormente, el *OCR* también) está basado en modelos de *segmentación*.

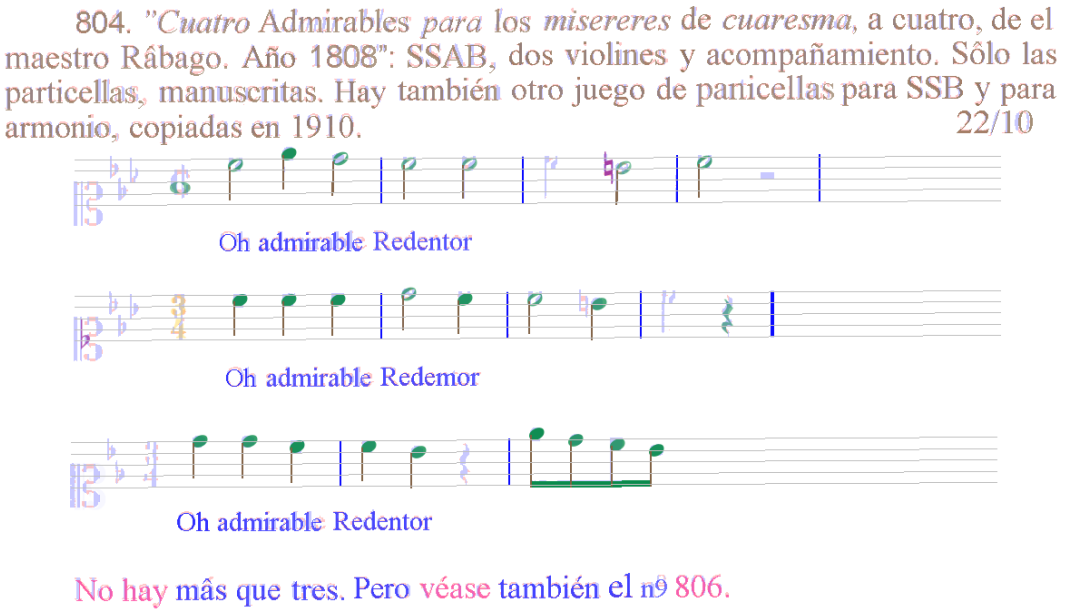


Figura 4.10: Resultado de procesar la obra 804 del libro "La Música en la Catedral de Santo Domingo de la Calzada" desde la interfaz de *Audiveris*.

Comentar que *Audiveris* se suele utilizar con otro programa llamado *MuseScore* [35]. Este programa es un editor de partituras de código abierto y está programado en *C++*. Es capaz de leer archivos *MusicXML* e importarlos a otros formatos, como por ejemplo *MIDI*, *Mp3* o imagen. El procedimiento que suele seguirse es: procesar una imagen en *Audiveris* e importarla al formato *MusicXML*, para después, abrirla con *MuseScore* y guardarla en formato *MIDI* o *Mp3*, dos de los formatos con los que *Audiveris* no trabaja. Esta no es la única forma que tenemos de obtener los distintos formatos de la música, ya que, como vimos en la *figura 4.5*, también podemos utilizar *Python*. En realidad, una vez que tenemos la imagen procesada y transformada a *MusicXML*, el resto de procesos son inmediatos. En la *figura 4.11.* se muestra el resultado de procesar una imagen utilizando *Audiveris*, se puede apreciar que comete algunos errores pero debemos valorar que pueden ser corregidos desde la interfaz y que nos devuelve directamente archivos *MusicXML*. También, se puede escuchar el sonido tanto de la obra original como el de la obra predicha por *Audiveris*.

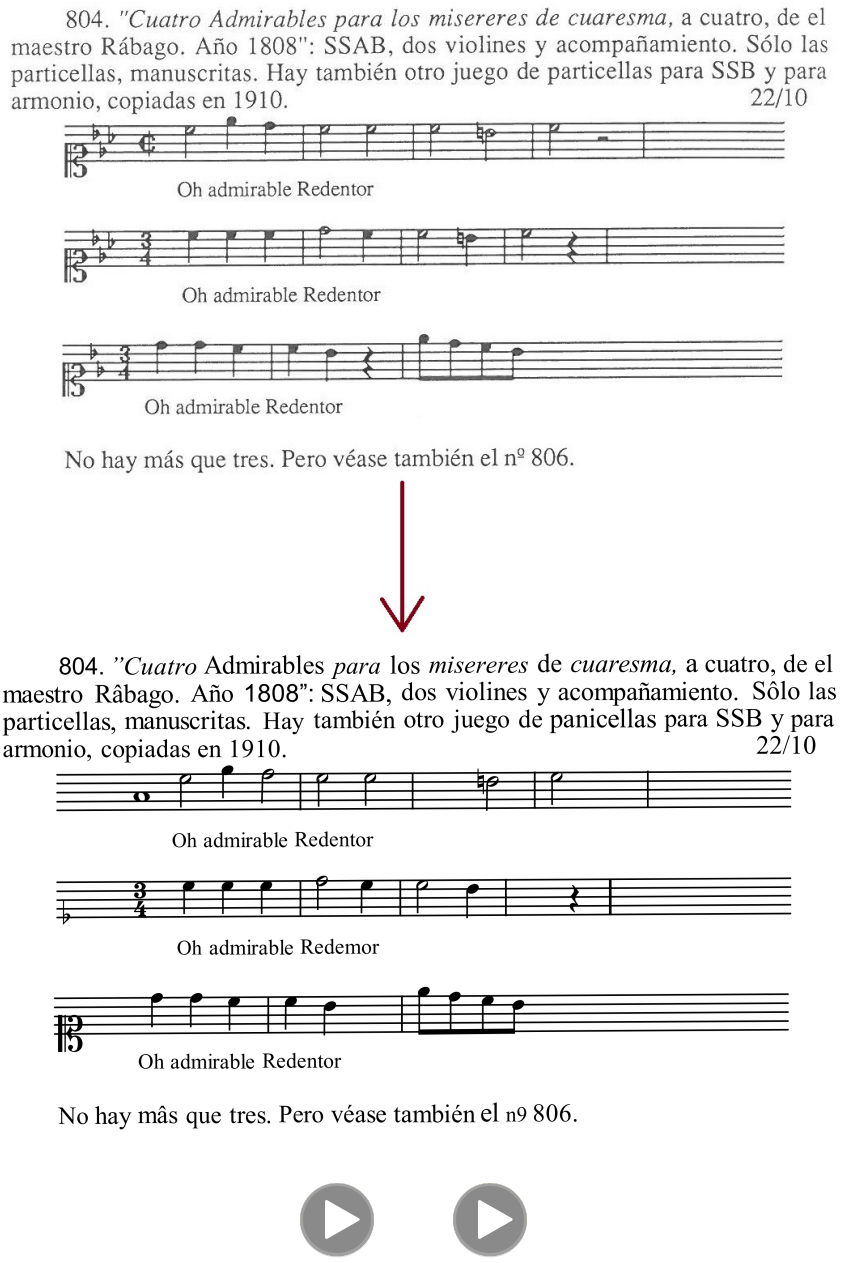


Figura 4.11: Obra 804 del libro "La Música en la Catedral de Santo Domingo de la Calzada" reconstruida con *Audiveris*. El botón de "play" derecho es el audio predicho.

Al reproducir[1] el sonido de la *figura 4.11*, podemos notar que la melodía de la izquierda (la de verdad) y la de la derecha (la predicha) son bastante diferentes. Este es el principal problema del *OMR*, que se necesitan unos modelos con una efectividad muy elevada. A pesar de encontrar cierta similitud entre la imagen de verdad y la predicha, la música de ambas cambia considerablemente. Es por eso que la mayoría de sistemas, como es el caso de *Audiveris*, ofrecen interfaces en las que poder hacer correcciones de forma manual. Además, están

[1]También pueden encontrarse en el repositorio de nuestro proyecto, en la carpeta de "Predicciones". Dentro, a su vez, de las carpteas "*Audiveris*" y "*Obra_804*".

pensados para extraer la música de una o pocas páginas (lo que ocupe una melodía) escaneadas. Pensar que podemos construir modelos con precisiones razonablemente altas, y aun así, obtener melodías diferentes, ya que a poco que se equivoque en uno o más símbolos, la música puede cambiar por completo. Volviendo a la *figura 4.10,* en los dos primeros pentagramas no ha detectado bien la *clave de Do en primera*, esto hace que se tome por defecto la *clave de Sol* y, por tanto, hace que todas las notas cambien por completo. En el caso de nuestro libro tenemos un pequeño problema con este aspecto. Contamos con 1553 obras y tenemos que asumir que vamos a tener fallos que tendremos que corregir de forma manual, lo cual puede llevarnos bastante tiempo.

En la siguiente sección, vamos a construir un modelo de detección, con *dataset* y anotaciones propias, de símbolos musicales para los pentagramas de nuestro libro. Los principales motivos que nos han llevado a tomar esta decisión han sido que, para los *datasets* disponibles en la literatura suele haber un modelo entrenado con ellos, que ya hemos visto que en la mayoría de casos no han funcionado bien en nuestro libro. Otro motivo es que hemos encontrado un modelo basado en la *segmentación* y otro basado en el *End-to-End* y me ha parecido interesante construir desde cero una serie de técnicas respaldadas por un modelo de detección para llegar al resultado final. Probablemente estarán sesgadas para nuestro libro, pero siempre se puede aumentar el número de imágenes de nuestro *dataset* si se obtienen buenos resultados y la estrategia que hemos seguido es buena.

La idea va a ser detectar los pentagramas y bloques de texto, aplicarles *OMR* y *OCR* por separado, para finalmente juntar toda la información y devolver un archivo *MusicXML* con el resultado final. Notar que el paso de aplicar *OMR* a un pentagrama, lo podemos hacer también con [31], ya que no lo hacía del todo mal (ver *figura 4.8*). Además, con un dataset propio, vamos a poder evaluar nuestro modelo con *métricas* que nos indiquen cómo de bien se está realizando el *OMR*. Comentar que el modelo de detección va a necesitar un paso posterior más para terminar de clasificar correctamente los símbolos. Por ejemplo, las notas musicales y las *claves* dependen también de la posición que ocupan en las lineas horizontales del pentagrama, entonces, si detectamos dos *corcheas* (un tipo de nota que dura medio tiempo) deberemos decir su posición en el pentagrama para terminar de clasificarlas.

4.3. Detección de símbolos

En esta sección explicaremos el proceso que hemos seguido para detectar los símbolos musicales, paso previo a la transcripción a formato *MusicXML*. Como siempre, iremos mencionando los cuadernos de *Jupyter* de nuestro proyecto relacionados con lo que estemos explicando.

Nuestra primera tentativa fue probar a utilizar *OpenCV*. Para ello, ha sido necesario encontrar una forma de eliminar las líneas horizontales de los pentagramas, utilizando *opera-*

ciones morfológicas, sin alterar demasiado la forma de los demás símbolos. En la *figura 4.12.* se puede ver el código empleado para eliminar las líneas horizontales y en la *figura 4.13.* el resultado de aplicarlo a un pentagrama de nuestro libro.

```
# Operaciones morfológicas
kernel = np.ones((1,100))
gray = cv2.morphologyEx(gray,cv2.MORPH_TOPHAT,kernel)
kernel = np.ones((2,1))
gray = cv2.erode(gray,kernel)
kernel = np.ones((5,2))
gray = cv2.morphologyEx(gray,cv2.MORPH_CLOSE,kernel)
kernel = np.ones((2,3))
gray = cv2.dilate(gray,kernel)
kernel = np.ones((4,2))
gray[:,50:] = cv2.morphologyEx(gray[:,50:],cv2.MORPH_CLOSE,kernel)

# Eliminación de componentes pequeñas
nb_components,output,stats,_ = cv2.connectedComponentsWithStats(gray,connectivity=8)
sizes = stats[1:,-1]; nb_components = nb_components - 1
min_size = 10
gray = np.zeros((output.shape),dtype="uint8")
for i in range(0,nb_components):
    if sizes[i] >= min_size:
        gray[output == i + 1] = 255
```

Figura 4.12: Código empleado para eliminar las líneas horizontales de un pentagrama umbralizado.

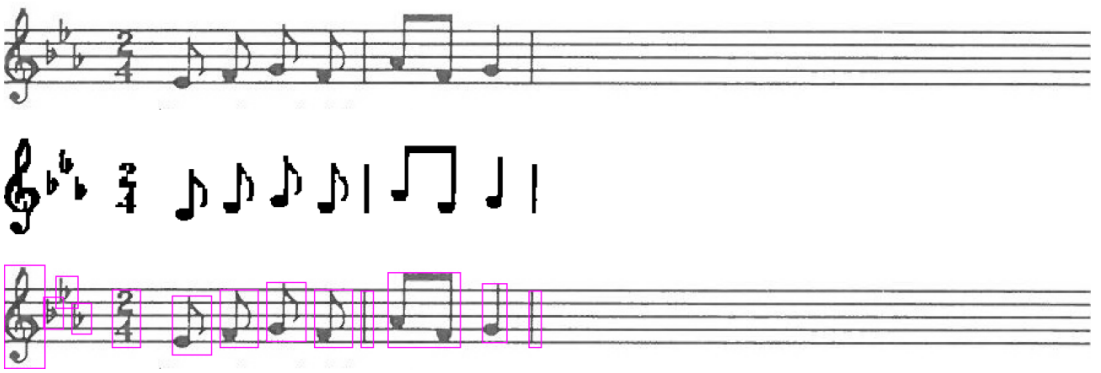


Figura 4.13: Pentagrama umbralizado y con las lineas horizontales eliminadas para la localización de símbolos.

En la *figura 4.13.* se puede comprobar que hemos localizado correctamente todos los objetos, pero todavía nos falta diferenciarlos. Es muy importante tenerlos clasificados correctamente si

queremos transformarlos en música. Lo siguiente que probamos fue sacar algunas propiedades elementales de nuestros contornos para extraer las características [36]. De esta manera, a cada figura le podremos asignar un vector con algunas propiedades: altura, anchura, área, etc. Con esos vectores podemos definir reglas como la siguiente: "Si tiene un altura grande y una anchura intermedia, entonces esa figura es una *Clave de Sol*". Es más, con esos vectores de descriptores construimos un clasificador utilizando el algoritmo *k-Means* [37, 38]. El proceso entero se puede encontrar en el cuaderno del repositorio de nuestro proyecto con nombre "*Descriptores_Manual*". El algoritmo seguido para sacar la música utilizando *OpenCV* puede encontrarse resumido en la *figura 4.14.*

Paso 1 $\longrightarrow$ Borrar líneas horizontales
Paso 2 $\longrightarrow$ Detectar contornos
Paso 3 $\longrightarrow$ Calcular el vector de descriptores a cada contorno
Paso 4 $\longrightarrow$ Clasificar los descriptores definiendo reglas o utilizando *k-Means*

Figura 4.14: Algoritmo de detección de símbolos musicales utilizando *OpenCV* y *k-Means*.

Lo bueno que tiene el procedimiento seguido en la *figura 4.14.* es que es muy rápido, por lo que en poco tiempo tendríamos anotados todos los pentagramas. Lo malo es que lleva mucho tiempo encontrar unos descriptores y unas reglas que generalicen bien. Además, otro punto en contra de esta técnica es que vamos a tener que diferenciar más figuras de las que de verdad harían falta. Por ejemplo, en la *figura 4.13,* el antepenúltimo contorno (el tercero empezando por la derecha) es un bloque de dos *corcheas* que en realidad sería mejor tenerlo separado en dos, uno para cada una de las notas. En el libro son bastante recurrentes estos bloques de dos o más *corcheas*, *semicorcheas* o incluso figuras más inusuales como dos *negras* unidas por una *ligadura*, etc. En todos estos casos es mejor calcular la *bounding-box* por separado y *OpenCV* nos va a devolver esos símbolos musicales unidos como un único objeto. En consecuencia, hemos descartado este procedimiento para localizar y clasificar símbolos musicales por tener que tener en cuenta muchos casos particulares que no son necesarios. A pesar de ello sí que hemos hecho uso de *OpenCV* para que el proceso de anotar sea más rápido, este tipo de pruebas se pueden encontrar en el cuaderno del repositorio con nombre "*Sacar_Musica*".

Como el proceso de anotación "semi-manual" es muy costoso, tras haber etiquetado 20 pentagramas nos hemos detenido. En la *figura 4.15.* se pueden ver algunos de ellos anotados, también podemos visualizar las cinco *claves* que se encuentran en el libro: de arriba hacia abajo son la *Clave de Sol*, las *Claves de Do* en primera, tercera y cuarta, y la *Clave de Fa* en cuarta. En la *figura 4.16.* se encuentra un pentagrama junto con su anotación, la cual hemos elegido almacenar en formato *json* [39].

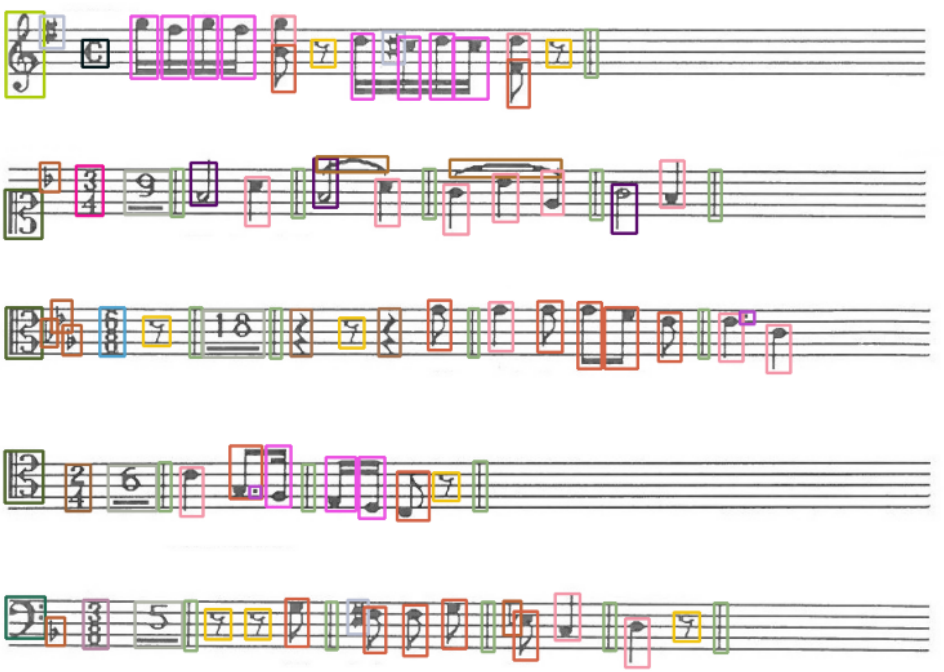
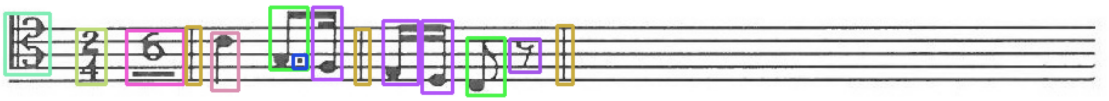


Figura 4.15: Algunos pentagramas del libro "La Música en la Catedral de Santo Domingo de la Calzada" anotados.



```
[#Las box en formato [xmin,ymin,xmax,ymax]
{"box": [14, 20, 51, 70], "label": "claved"},
{"box": [73, 33, 97, 78], "label": "2x4"},
{"box": [114, 34, 161, 78], "label": "sil0"},
{"box": [163, 31, 175, 78], "label": "line"},
{"box": [188, 36, 206, 83], "label": "nota1"},
{"box": [232, 15, 263, 66], "label": "nota1:2"},
{"box": [251, 54, 262, 65], "label": "punt"},
{"box": [267, 15, 291, 74], "label": "nota1:4"},
{"box": [302, 33, 314, 78], "label": "line"},
{"box": [325, 26, 354, 78], "label": "nota1:4"},
{"box": [357, 26, 382, 85], "label": "nota1:4"},
{"box": [394, 40, 425, 87], "label": "nota1:2"},
{"box": [429, 41, 454, 68], "label": "sil1:2"},
{"box": [468, 30, 481, 78], "label": "line"}
]
```

Figura 4.16: Pentagrama del libro "La Música en la Catedral de Santo Domingo de la Calzada" junto con su anotación.

Comentar que la anotación de la *figura 4.16.* tiene como único objetivo entrenar modelos de detección y es independiente del formato *MusicXML* o de cualquier otro. Llegados a este punto, lo que queremos es seguir anotando pentagramas, ya que 20 se quedan algo escasos teniendo en cuenta que el libro cuenta con un total de 2308. El problema es que el proceso de etiquetar imágenes requiere de mucho tiempo y necesitamos una forma de acelerarlo un poco. Para ello, hemos buscado en la literatura algunos modelos de detección para entrenarlos con nuestro pequeño *dataset* (los explicaremos a continuación). De esta forma, tomaremos como anotación de un pentagrama nuevo la predicción del modelo sumada a nuestras correcciones manuales. Seguiremos este proceso hasta tener anotados 300 pentagramas (escogidos de forma aleatoria), los cuales ya suponen una muestra bastante significativa de nuestros datos. A partir de este momento contaremos con un *dataset* al cual le aplicaremos las técnicas rutinarias de detección que se utilizan en el aprendizaje automático. En la *figura 4.17.* se encuentra resumido el proceso de anotación de los pentagramas.

Paso 1	$\longrightarrow$	Entrenar un modelo de detección
Paso 2	$\longrightarrow$	Elegir una muestra aleatoria de pentagramas no anotados
Paso 3	$\longrightarrow$	Calcular las predicciones
Paso 4	$\longrightarrow$	Corregir los errores para completar la anotación
Paso 5	$\longrightarrow$	Volver al paso 1 hasta tener anotados 300 pentagramas

Figura 4.17: Proceso para anotar los pentagramas.

El proceso de la *figura 4.17.* podríamos haberlo seguido hasta haber anotado los 2308 pentagramas, pero nos hubiese llevado bastante tiempo. Es de esperar que si obtenemos unos resultados correctos en nuestro *dataset*, el resultado sea similar en el resto de pentagramas, ya que forman parte del mismo libro y, además, según íbamos anotando más pentagramas el modelo que usábamos como "etiquetador" cada vez obtenía mejores resultados. No obstante, siempre se pueden hacer comprobaciones una vez realizados los experimentos y ver la eficacia de forma manual. A partir de este punto explicaremos los modelos de detección que hemos usado en nuestros experimentos, así como las métricas que hemos utilizado para validarlos. Comentar que hemos utilizado el 75 % de las imágenes para entrenar (conjunto de entrenamiento) y el 25 % restante para validar los resultados (conjunto de test).

Hemos entrenado un total de cuatro modelos repartidos en dos librerías distintas. En concreto, hemos utilizado la librería de *Python* llamada *IceVision* [40, 41], la cual es bastante reciente teniendo en cuenta que su primera versión se lanzó en el año 2020, y la librería *Darknet* [42], programada mayoritariamente en los lenguajes de *C* y *C++*. Para validar los modelos utilizaremos cinco métricas: *Precison*, *Recall*, *F1-score*, *mAP* (del inglés, *mean Average Precision*) y *COCO-metric* [43]. Todas estás métricas dependen de un indicador

conocido como el *IoU* (del inglés, *Intersection over Union*) [44], el cual actúa como umbral para indicar qué predicciones admitimos como correctas y cuáles no. Los modelos de detección nos devuelven un conjunto de predicciones donde se puede encontrar la clase a la que pertenece el objeto, su *bounding-box* y el nivel de confianza con el que se admite que es esa clase y no otra. Lo que se hace es comparar todas las *bounding-box* predichas con las verdaderas y si, dada una predicción, existe una *bounding-box* de las anotaciones de manera que el *IoU* de ambas es mayor que cierto umbral, entonces tomamos esa anotación como correcta. En nuestro caso hemos tomado como umbral el más típico, 0.5. En la *figura 4.18.* podemos ver una ilustración del *IoU*.

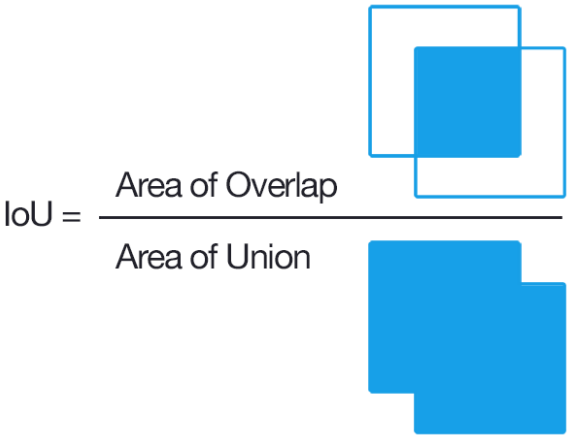


Figura 4.18: Ecuación para medir el *IoU* de dos rectángulos.

Es habitual, también, que dentro de las predicciones haya algunas de ellas que se superpongan. Esto también es un error que deberemos solucionar, ya que puede darse el caso de que nuestro modelo diga que un objeto es dos cosas diferentes. Consideraremos que, aquellas predicciones que tengan un *IoU* mayor que 0.5, están detectando el mismo objeto y, por tanto, debemos elegir solamente una de ellas, la cual será la que tenga el nivel de confianza más alto. Esta técnica se conoce como *Non-Maximum Suppression* [45].

Una vez hemos eliminado todas aquellas predicciones que no nos aportan información estamos preparados para calcular las métricas de la detección. En la *figura 4.19.* podemos encontrar las fórmulas para calcular la *Precision*, el *Recall* y el *F1-score*. La *Precision* puede verse como la proporción de aciertos que se obtienen dentro del conjunto de clases predichas y el *Recall* la proporción de aciertos teniendo en cuenta todas las clases. Puede darse el caso de que haya modelos con precisiones altas pero un *Recall* bajo y la explicación es que el modelo no es capaz de detectar todas las clases, pero las que detecta, las detecta muy bien. El *F1-score* es una medida que se obtiene a partir de estas dos anteriores. La forma de obtener estos indicadores es ir calculándolos para cada clase de forma independiente y luego hacer la media, siempre teniendo en cuenta las condiciones de $IoU \geq 0.5$ y *Non-Maximum Suppression*.

$$Precision = \frac{Predicciones\ correctas}{Total\ predichas}$$

$$Recall = \frac{Predicciones\ correctas}{Total}$$

$$F1 - score = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

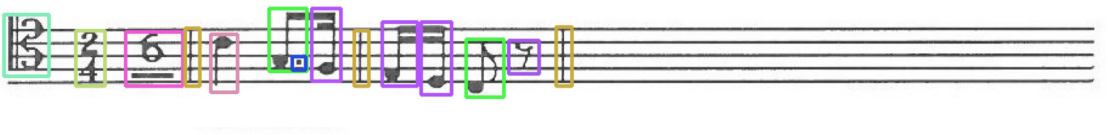
Figura 4.19: Fórmulas para calcular la *Precision*, *Recall* y *F1-score* de cada clase.

Notar que en la fórmula de la *Precision* de la *figura 4.19.* podemos tener un problema si no ha predicho ninguna clase, ya que estaríamos dividiendo 0 entre 0. En esos casos no se tiene en cuenta, como ya hemos dicho anteriormente, esa clase para calcular la *Precision* global. El cálculo del *mAP* depende de la *Precision* y el *Recall* y es algo más complejo de obtener, esta explicado con detalle en [43]. En esencia, se calcula un indicador para cada clase denominado *AP* (del inglés, *Average Precision*) y luego se hace la media de todos ellos para calcular el *mAP*. Reiterar en la importancia que tiene el *IoU* mayor que 0.5 para obtener todos estos valores ya que si diésemos otro umbral, los valores cambiarían. Por último, para calcular el *COCO-metric*, se debe calcular el *mAP* no solamente para el umbral 0.5, si no también para los umbrales 0.55, 0.6,..., 0.95. Con estos diez umbrales distintos obtendremos otros diez valores para el *mAP*, si hacemos la media de todos ellos obtendremos el *COCO-metric*. Notar que, según vayamos aumentando el valor del umbral, el valor del *mAP* será más pequeño.

La versión que hemos utilizado para la librería de *IceVision* es la 0.5.1 y los modelos que hemos utilizado para el entrenamiento han sido: *EfficientDet* [46], *Faster R-CNN* [47] y *RetinaNet* [48], aunque tiene más modelos que se pueden entrenar, como por ejemplo *Mask R-CNN* [49] o, desde la versión 0.5.2, *YOLO v5* [50]. Podíamos detenernos a explicar detalladamente en qué se basan cada uno de estos modelos pero se sale de los objetivos de este trabajo. Existen más librerías en donde se podrían entrenar este tipo de modelos [51, 52], de hecho, podríamos programarlos desde cero si quisiésemos pero la librería de *IceVision* nos ofrece una serie de funcionalidades que nos facilitan mucho todo el proceso de entrenamiento. En este caso es interesante comentar que el modelo usado en el proceso descrito en la *figura 4.17.* ha sido el *Faster R-CNN*. El motivo ha sido que, de los modelos que hemos probado, es el que más rápido conseguía entrenarse con unos resultados aceptables, y como lo único que nos interesaba era anotar imágenes lo más rápido posible, no nos detuvimos a buscar si había alguno mejor. Otra de las cosas de utilidad que nos ofrece *IceVision* es la de aplicar lo que se conoce como *Data Augmentation* [53]. Esta técnica consiste en aumentar el

número de imágenes de nuestro *dataset* por medio de rotaciones, añadiendo ruido, recortando, etc. Algunas de estas transformaciones no tenían sentido en nuestro contexto, por lo que han sido descartadas. Por ejemplo, no tiene sentido rotar un pentagrama 180 grados. No entraremos en más detalles sobre el proceso de entrenamiento, solamente comentar que el modelo que mejores resultados ha obtenido con esta librería ha sido el *Faster R-CNN* y todos los entrenamientos se pueden encontrar en el cuaderno del repositorio de nuestro proyecto con nombre "*Training*", el cual ha sido utilizado desde *Google Colaboratory* para poder utilizar las GPUs que ofrece de forma gratuita.

En la librería *Darknet* se pueden utilizar los modelos *YOLO v2* [54], *YOLO v3* [55] y *YOLO v4* [56]. En nuestro caso hemos optado por entrenar el último de ellos, *YOLO v4*. Debido a que el proceso de entrenamiento de YOLO requiere de más recursos, hemos decidido entrenarlo a través del servidor *Simba* de la Universidad de la Rioja, el cual dispone de cuatro GPUs *Nvidia RTX 2080 Ti*. Una vez terminado el entrenamiento, hemos hecho uso de *Google Colaboratory* para validar en el conjunto de test y ver algunas predicciones, esto se puede encontrar en el cuaderno de nuestro proyecto con nombre "*YOLO*". Comentar que la librería *Darknet* precisa de unas anotaciones específicas distintas a la propuesta en la *figura 4.16*, es por eso que ha sido necesario adaptar nuestras etiquetas a este formato propio, el proceso se encuentra también en el cuaderno antes mencionado. En la *figura 4.20.* podemos ver el pentagrama de la *figura 4.16.* con la anotación en formato *YOLO*.



```
#claseID x y w h
0  0.03 0.38 0.03 0.42
17 0.09 0.47 0.02 0.38
12 0.14 0.47 0.05 0.37
24 0.18 0.46 0.01 0.40
3  0.21 0.50 0.02 0.40
6  0.26 0.34 0.03 0.43        #x = (xmin + xmax)/(2*ancho)
21 0.27 0.50 0.01 0.09        #y = (ymin + ymax)/(2*alto)
7  0.30 0.38 0.02 0.50        #w = (xmax - xmin)/(2*ancho)
24 0.33 0.47 0.01 0.38        #h = (ymax - ymin)/(2*alto)
7  0.36 0.44 0.03 0.44
7  0.39 0.47 0.02 0.50
6  0.44 0.54 0.03 0.40
15 0.47 0.46 0.02 0.23
24 0.51 0.46 0.01 0.41
```

Figura 4.20: Pentagrama del libro "La Música en la Catedral de Santo Domingo de la Calzada" anotado en formato YOLO.

En la *tabla 4.1.* se puede ver un resumen de los resultados obtenidos para cada modelo de detección en el conjunto de *test*. La información ha sido extraida de los cuadernos de *Jupyter* de nuestro proyecto con nombres: "*Metricas*" y *YOLO*.

	Precision	Recall	F1-score	mAP	COCO
EfficientDet	0.28	0.14	0.19	13.86 %	0.16
Faster R-CNN	0.86	0.77	0.82	76.73 %	0.59
RetinaNet	0.73	0.15	0.25	14.73 %	0.19
YOLO v4	0.89	0.90	0.90	68.25 %	0.48

Tabla 4.1: Métricas de los diferentes modelos de detección en el conjunto de test.

En vista de los resultados de la *tabla 4.1*, los modelos *EfficientDet* y *RetinaNet* quedan descartados, vemos que sus resultados son bastante bajos en comparación con los otros dos modelos. También, cabe destacar que a ambos modelos les ha llevado mucho más tiempo entrenarlos que al *Faster R-CNN* dentro de la librería *IceVision*, por lo que obtenemos peores resultados invirtiendo más tiempo. Para los otros dos modelos tenemos unos resultados más que aceptables en la detección. El modelo *YOLO v4* ha sido el que más tiempo de entrenamiento ha necesitado (72 horas), pero también es el que mejores valores de *Precision*, *Recall* y *F1-score* nos ofrece. Las métricas *mAP* y *COCO-metric* son algo más altas en el *Faster R-CNN*, pero las del *YOLO v4* son bastante correctas. También hemos realizado algunas pruebas manuales para ver cómo son las predicciones de cada modelo de forma visual y al final hemos decidido utilizar el modelo *YOLO v4*, ya que es el que menos se confunde de los cuatro modelos estudiados y, en la mayoría de casos, detecta todo y lo detecta de forma correcta. El modelo *Faster R-CNN* también es bastante bueno pero comete algunos errores, como por ejemplo, confundir *corcheas* con *semicorcheas*. No obstante, aunque sepamos que el mejor modelo es el *YOLO v4*, como tenemos cuatro modelos entrenados, podemos usarlos a la hora de digitalizar una obra y comparar los resultados finales. En la *figura 4.21.* podemos ver dos pentagramas anotados por *YOLO v4*.

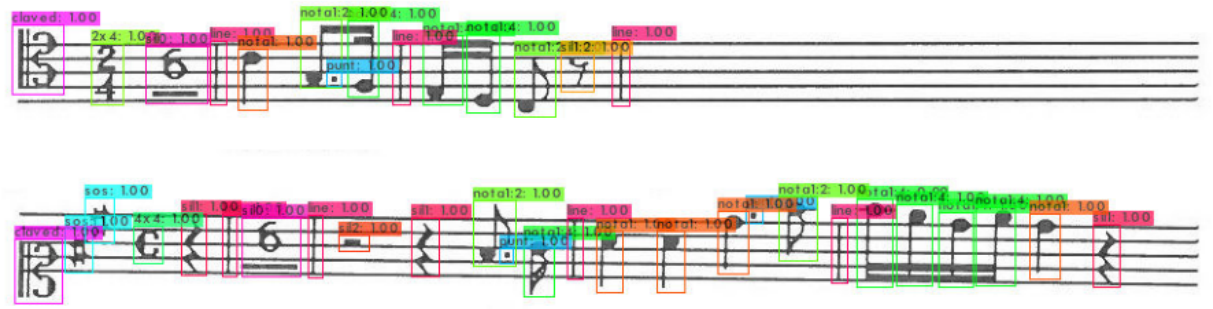


Figura 4.21: Pentagramas del libro "La Música en la Catedral de Santo Domingo de la Calzada" anotados por el modelo *YOLO v4*.

Recordar que, cuando estábamos explicando el programa *Audiveris*, dijimos que los modelos de *OMR* deberían tener una exactitud muy elevada y que, con muy pocos errores, las melodías podrían cambiar completamente. Teniendo en cuenta este aspecto, puede que nuestros modelos se queden algo justos en los valores de nuestras métricas, incluso el *YOLO v4* y el *Faster R-CNN*. No obstante seguiremos con el proceso que estamos proponiendo de digitalizar obras hasta el final y, una vez que tengamos todas las herramientas para ello, sacaremos conclusiones. El siguiente problema que tenemos que resolver es bastante más sencillo que el que acabamos de realizar. Hay algunos símbolos musicales, en concreto las *claves* y sobre todo las *notas* musicales, que dependen también de la posición que ocupan en las líneas del pentagrama. En la siguiente sección daremos una solución para realizar lo más correctamente posible esta tarea.

4.4. Clasificación de notas

Una vez hayamos determinado la altura de aquellos objetos musicales que lo precisen en el pentagrama, habremos terminado de extraer toda la información de la melodía y estaremos preparados para construir un algoritmo propio para digitalizar las obras del libro "La Música en la Catedral de Santo Domingo de la Calzada".

La primera herramienta a la que hemos acudido para resolver esta tarea es, una vez más, *OpenCV*. Para ello, lo primero que hemos necesitado hacer es encontrar una forma de eliminar todos los símbolos musicales sin alterar demasiado las líneas horizontales. Teniendo en cuenta que las líneas del pentagrama son los objetos de mayor longitud horizontal, si hacemos una *erosión* horizontal lo suficientemente grande sobre un pentagrama umbralizado, conseguiremos fácilmente eliminar todos los símbolos musicales. Notar que la operación que queremos hacer es la contraria a la que utilizábamos para conseguir detectar todos los símbolos musicales (ver *figura 4.13*). Antes queríamos eliminar las líneas horizontales para poder detectar fácilmente todos los símbolos musicales, ahora, al contrario, queremos eliminar todos los símbolos para tener localizadas las líneas. En la *figura 4.22.* se encuentra el código *Python* empleado y en la *figura 4.23.* un pentagrama tras haberle aplicado esta transformación.

```
_,gray = cv2.threshold(gray,0,255,cv2.THRESH_BINARY_INV|cv2.THRESH_OTSU) #Umbralizar
kernel = np.ones((1,50))
gray = cv2.erode(gray,kernel) #Erosión horizontal
kernel = np.ones((1,500))
gray = cv2.morphologyEx(gray,cv2.MORPH_CLOSE,kernel) #Cerramos horizontalmente
```

Figura 4.22: Código empleado para eliminar los símbolos musicales de un pentagrama.

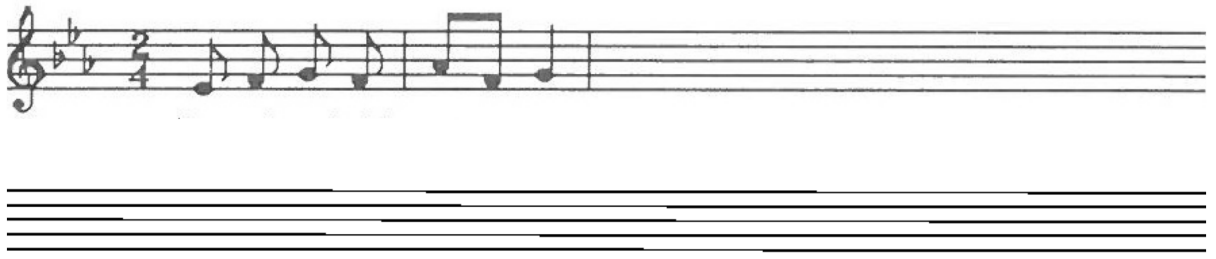


Figura 4.23: Pentagrama del libro "La Música en la Catedral de Santo Domingo de la Calzada" con todos sus símbolos musicales eliminados.

Llegados a este punto y conocedores de la forma de nuestros símbolos musicales, podemos sacar mucha información. Con el pentagrama transformado como en la *figura 4.23.* vamos a poder definir diferentes alturas, que no van a dejar de ser intervalos dentro del eje-y de la fotografía, y, definiendo reglas, terminar de clasificar nuestros símbolos. Por ejemplo, en la *figura 4.15.* podemos ver que hay tres pentagramas con la *clave de Do*, la diferencia que hay entre ellas es que están a diferente altura en el pentagrama, es por eso que la primera de ellas, por ejemplo, se le denomina *calve de Do en primera*, ya que se encuentra en la primera línea del pentagrama. Podemos definir la siguiente regla: "La altura de la *clave de Do* estará determinada por la línea horizontal más cercana a la mitad de su *bounding-box*". Para la *clave de Fa* y *clave de Sol* pasa algo parecido, pero no es el caso de nuestro libro ya que la *clave de Sol* siempre está en segunda y la *clave de Fa* en cuarta.

Sin duda, los símbolos que más varían en relación a la posición en la que se encuentren, son las *notas musicales*. En la *figura 4.24.* podemos ver la *escala de Do mayor* en *clave de Sol*. Vemos que las notas se clasifican de manera distinta según la altura que ocupe en el pentagrama.

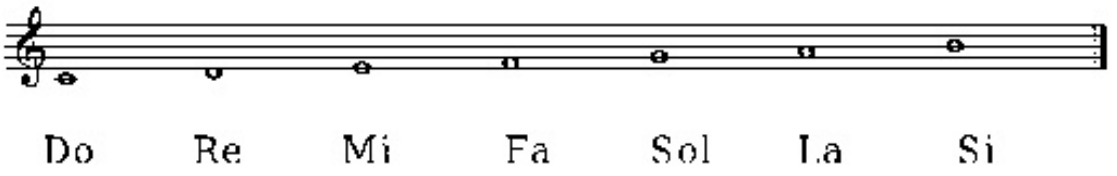


Figura 4.24: Escala de Do mayor escrita en clave de sol.

Al principio del capítulo, en concreto, en la *figura 4.3*, mostrábamos la misma nota escrita en dos *claves* diferentes. Es por eso que una nota musical está completamente determinada por la *clave* y la altura que ocupa en el pentagrama. La forma de determinar la altura de una nota es localizando la posición de su "cabeza" (todas las notas cuentan con una especie de forma circular), en la *figura 4.25.* podemos ver un pentagrama de nuestro libro con todas las notas musicales clasificadas según su altura.

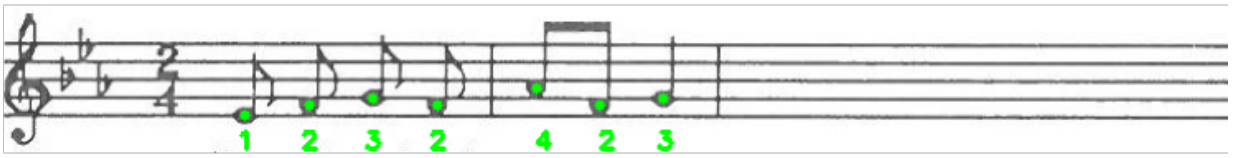


Figura 4.25: Pentagrama del libro "La Música en la Catedral de Santo Domingo de la Calzada" con las notas musicales clasificadas por alturas.

La forma de detectar las diferentes alturas en las notas de la *figura 4.25.* ha sido utilizando *OpenCV* y algunas propiedades de los contornos para detectar las "cabezas" de las notas, para después, buscar la región del pentagrama más cercana a ella. En este caso tenemos que separar en más regiones ya que el espacio que hay entre dos líneas también cuenta. De esta forma, si una nota tiene altura 1 en el pentagrama y además sabemos que está en *clave de Sol*, podemos decir que esa nota musical es un *Mi*, en concreto es la cuarta octava de la nota *Mi*. Con altura 2 y la misma *clave* sería un *Fa*, con altura 3 un *Sol*, etc. Todo el proceso se puede encontrar en el cuaderno de nuestro repositorio con nombre *"Descriptores_Manual"*. El problema de utilizar esta técnica es que nuevamente teníamos que definir demasiadas reglas y casos particulares que podíamos simplificar mucho utilizando otro camino.

Al final, decidimos volver a utilizar un modelo *Deep Learning* para realizar esta tarea. Esta vez el proceso de anotación no fue tan costoso como el anterior, utilizamos los pentagramas que ya teníamos anotados previamente para extraer todas aquellos objetos que fueran notas musicales y anotando su altura de forma manual, en poco tiempo conseguimos construir un *dataset* de 2519 imágenes con 17 clases distintas, reservando un 80 % para entrenar y un 20 % para validar. De esta forma también vamos a poder definir *métricas*, al igual que hacíamos en la detección, para tener unos métodos de validación más sólidos que los que utilizamos visualmente en *OpenCV*. En concreto utilizaremos una métrica denominada *accuracy*, la cual consiste en dividir el nº de aciertos entre el nº total de imágenes (es una proporción de aciertos). En la *figura 4.26.* podemos ver algunas notas musicales de este *dataset*.

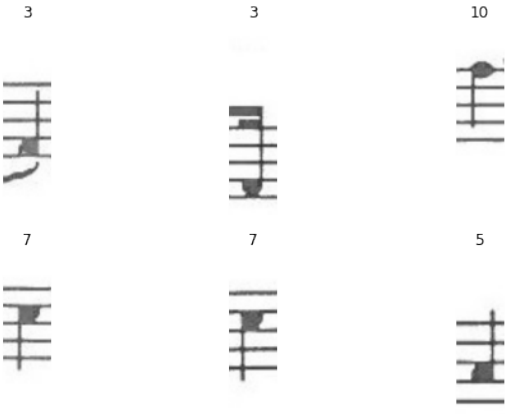


Figura 4.26: Notas musicales clasificadas según su localización en las líneas horizontales del pentagrama.

Lo último que necesitábamos era seleccionar un modelo de clasificación que nos resuelva esta tarea. Son muchas las opciones que teníamos para elegir, pero al final decidimos entrenar una Red Neuronal Convolucional, en concreto elegimos el modelo *ResNet-18* [57], un caso particular de las *arquitecturas ResNet* [58], que aunque no es la red más reciente, tras entrenarla, obtuvimos una *accuracy* del $98,4\,\%$ en el conjunto de validación. Podría haber sido interesante probar más clasificadores pero decidimos no invertir más tiempo en esta tarea ya que el primer modelo que probamos nos proporcionaba unos resultados muy buenos y teníamos que seguir adelante con nuestro principal objetivo. Comentar que a este dataset también le aplicamos algunas técnicas de *Data Augmentation* y un reescalado a tamaño 256 de ancho por 64 de alto. El entrenamiento se realizó con la librería de *Python FastAI* [51] y puede encontrarse en el cuaderno de nuestro proyecto con nombre *"Training"*.

4.5. Digitalizando obras

En esta sección deberemos combinar todos los métodos que hemos ido explicando a lo largo de la memoria para conseguir transformar las obras de nuestro libro a los diferentes formatos. Haciendo una recapitulación, hemos identificado diferentes estructuras dentro del libro como las obras, los pentagramas o los bloques de texto. Hemos utilizado el *OCR* para extraer el texto de las imágenes y hemos construido un modelo de *OMR* propio para detectar los símbolos musicales de los pentagramas. En todos los casos, estamos transformado nuestras imágenes en datos estructurados en forma de cadena de caracteres y de objetos musicales. La manera de aprovechar todos estos métodos para las obras del libro, y en general para cualquier partitura musical, es muy sencilla, ya que solamente tendremos que detectar los bloques de texto y los pentagramas y utilizar *OCR* y *OMR* respectivamente. Luego podemos guardar toda esta información ordenada según su altura para así preservar la estructura original. En la *figura 4.27.* podemos encontrar el proceso para transformar una obra en dato estructurado.

Paso 1	$\longrightarrow$	Detectar bloques de texto
Paso 2	$\longrightarrow$	Detectar pentagramas
Paso 3	$\longrightarrow$	Ordenar los bloques y los pentagramas de menor a mayor en el eje-y
Paso 4	$\longrightarrow$	Aplicar *OCR* a los bloques de texto
Paso 5	$\longrightarrow$	Aplicar *OMR* a los pentagramas
Paso 6	$\longrightarrow$	Guardar la información

Figura 4.27: Algoritmo para extraer la información de una obra.

El método propuesto en la *figura 4.27.* no es la única forma de abordar el problema. *Audiveris*, por ejemplo, sigue otro camino para llegar al resultado final. En nuestro caso, es mejor extraer

la información de esta forma ya que nuestro modelo de *OMR* ha sido entrenado con imágenes de pentagramas individuales.

Llegados a este punto, contamos con un modelo $\mathcal{M}$ capaz de transformar imágenes de obras musicales en dato estructurado. Ahora bien, lo que necesitamos nosotros es tener las obras codificadas en el formato *MusicXML*, por lo que deberemos construir un algoritmo $\mathcal{A}$ que realice esa conversión. No vamos a entrar en los detalles de cómo hemos programado todo el proceso, no obstante, todo el código para esta parte se puede encontrar en el cuaderno de nuestro proyecto con nombre "*image2xml*". Una vez tengamos las obras codificadas en formato *MusicXML*, podremos transformarla a *MIDI*, *Mp3* o en imagen. Con la imagen reconstruida vamos a poder ver de forma visual los errores que ha cometido nuestro modelo al digitalizar la obra y con el *MIDI* o el *Mp3* vamos a poder reproducir el sonido, el cual hemos puesto que sea de piano. En la *figura 4.28.* se encuentra un esquema del flujo que estamos siguiendo para digitalizar una obra musical.

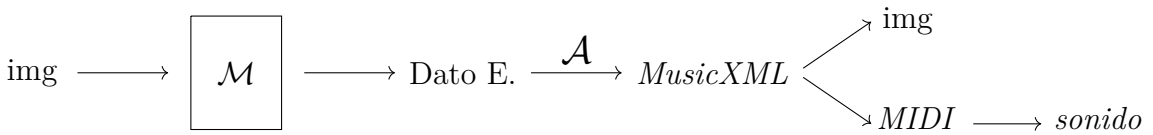


Figura 4.28: Proceso de digitalización de una obra musical.

En la *figura 4.11.* mostrábamos la imagen reconstruida que nos devolvía el programa *Audiveris*. Ahora vamos a mostrar cuatro predicciones, una para cada uno de los cuatro modelos que entrenamos para la detección, de la misma obra, la obra 804.

En la *figura 4.29.* se encuentra la predicción utilizando el modelo *EfficientDet*, en la *figura 4.30.* utilizando el modelo *RetinaNet*, en la *figura 4.31.* utilizando el modelo *Faster R-CNN* y en la *figura 4.32.* utilizando el modelo *YOLO v4*. En todas ellas se encuentra un reproductor de audio. También se pueden encontrar los diferentes archivos *MIDI* o *Mp3* de cada modelo y de la obra original en la carpeta de nuestro repositorio con nombre "*Predicciones*". Antes de las figuras podemos encontrar un reproductor con el audio original.

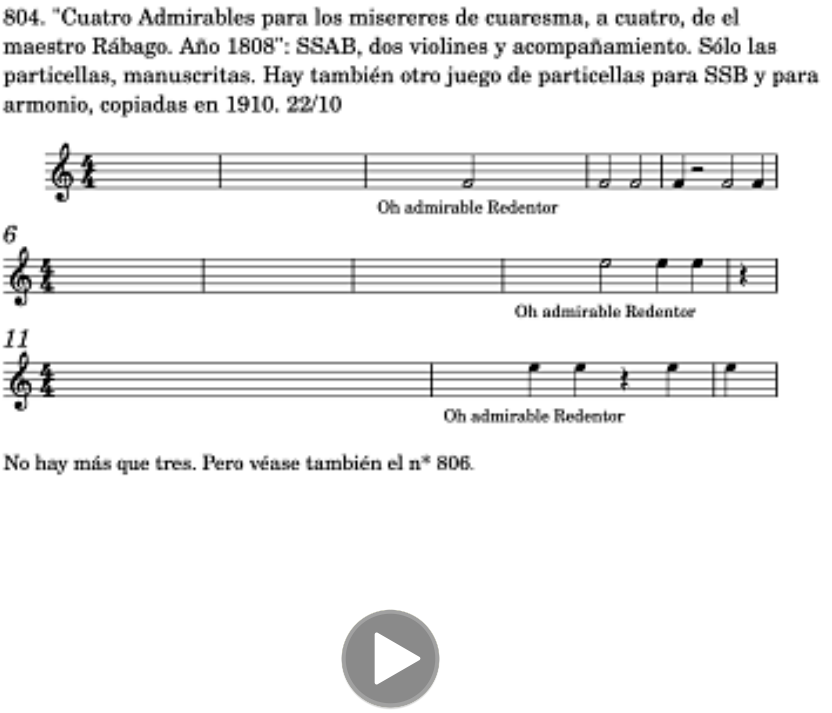


Figura 4.29: Obra 804 del libro reconstruida utilizando *EfficienDet*.

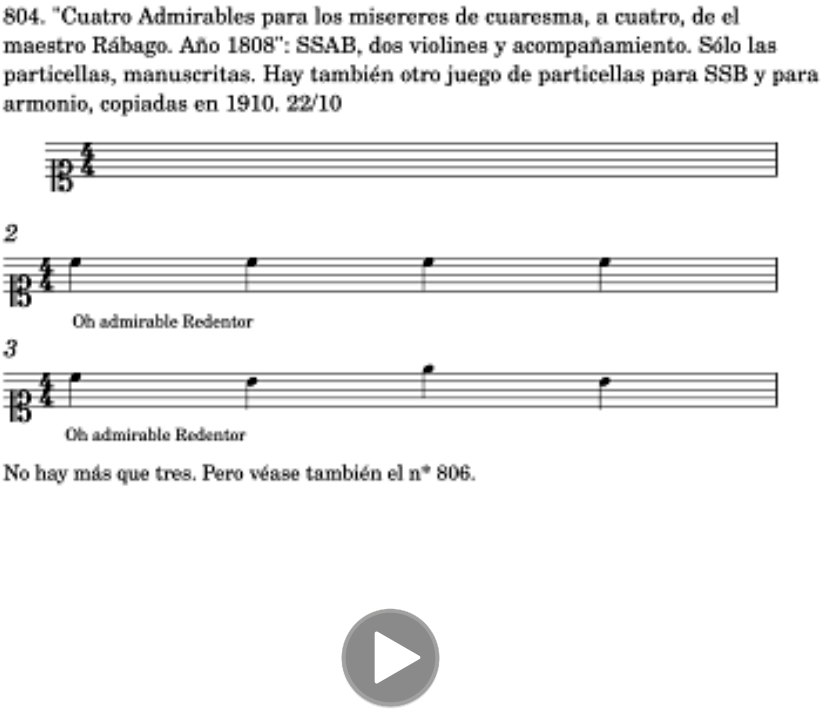


Figura 4.30: Obra 804 del libro reconstruida utilizando *RetinaNet*.

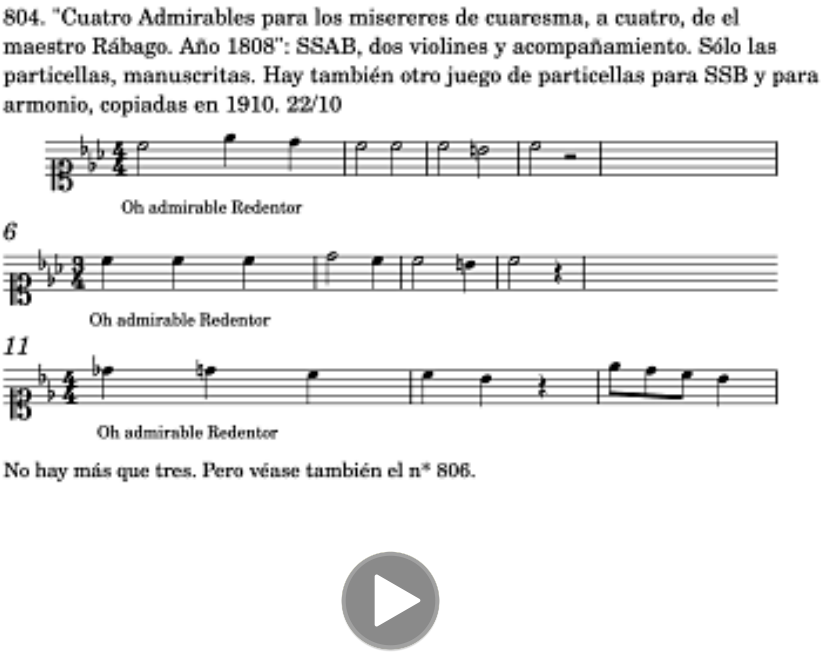


Figura 4.31: Obra 804 del libro reconstruida utilizando *Faster R-CNN*.

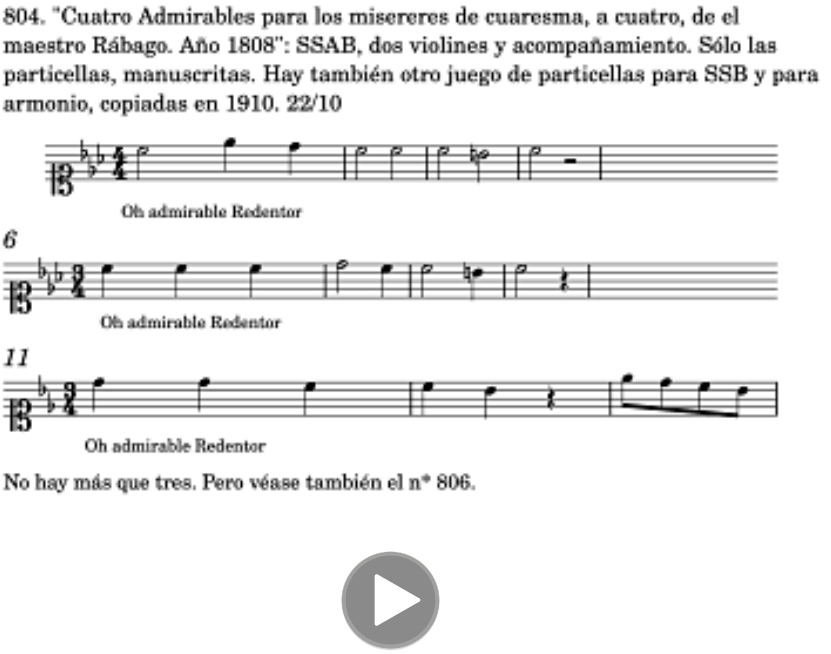


Figura 4.32: Obra 804 del libro reconstruida utilizando *YOLO v4*.

Como conclusiones en este capítulo podemos decir que hay tareas que no se resuelven de forma tan directa y todavía quedan muchos campos de la investigación con muchos problemas abiertos como es el caso del *OMR*. También, podemos asegurar que el planteamiento que hemos seguido, teniendo en cuenta que seguramente se pueda hilar más fino, es eficaz para digitalizar partituras musicales. Nuestros modelos están entrenados con pentagramas de nuestro libro y es probable que no funcionen del todo bien con otras partituras. No obstante siempre podemos aumentar nuestro *dataset* para conseguir que generalicen mejor.

Capítulo 5

Aplicación final

En este capítulo estructuraremos todos los resultados obtenidos a lo largo de la memoria en una pequeña aplicación web. Como venimos realizando todos los experimento en *Python* y como el objetivo de esta aplicación es recopilar todo lo que hemos ido explicando de la manera más sencilla, hemos optado por seguir utilizando *Python* para construir la aplicación. En concreto, hemos utilizado un programa que, dado un cuaderno de *Jupyter*, te crea una página web. El cuaderno de *Jupyter* que hemos utilizado para construir la página web se puede encontrar en el repositorio de nuestro proyecto y se llama *"web"*. El procedimiento a seguir es bastante sencillo, en [59] se pueden encontrar todos los pasos bien explicados.

Para crear la página web vamos a utilizar *Voila* [60, 61]. Su uso es muy sencillo ya que, con poner la instrucción "voila nombre.ipynb" en la consola de comandos, nos muestra el resultado que tendría el cuaderno de *Jupyter* "*nombre.ipynb*" en forma de página web desde una ventana local del navegador. De hecho, con solamente introducir la instrucción "voila" en la consola de comandos, se nos abre una pestaña local en el navegador donde aparecen todos los cuadernos de *Jupyter* que se encuentran en el directorio en el que nos encontremos. Si seleccionamos uno de ellos se generará también la pagina web.

Una vez tenemos creado el cuaderno de *Jupyter* que albergará la página web, deberemos crear un repositorio en *GitHub* con el mismo. Lo próximo que tendremos que hacer es subirla a internet, ya que por el momento solo podemos visualizarla en local. Para ello utilizaremos la plataforma *Heroku* [62, 59], la cual, entre otras, nos va a permitir subir la página web creada por *Voila* a internet para que todo el mundo pueda acceder a ella. El enlace al repositorio de la página web es el siguiente: `https://github.com/gosantam/Web`. Es evidente que esta forma de crear páginas web esta muy limitada, ya que deberá tener estructura de cuaderno de *Jupyter*. Tampoco conseguiremos una página muy optimizada y habrá búsquedas que nos llevarán más tiempo que el que nos llevaría si la hubiésemos programado y subido la web utilizando los procedimientos canónicos. No obstante, es la forma más rápida de crear una pequeña aplicación web y para mostrar los resultados de nuestra memoria es suficiente.

Si accedemos a: `https://afternoon-reaches-92149.herokuapp.com/` encontraremos la aplicación web. En ella se pueden realizar búsquedas de obras por autor y/o tipo de obra ("mostrar las misas de Manuel Pascual"), también se pueden realizar búsquedas clásicas. En la *figura 5.1.* se encuentra el buscador de la página web. Para las búsquedas podemos desplegar las pestañas de "*Autor*" y de "*Tipo de Obra*" para seleccionar las diferentes posibilidades o escribir una cadena de caracteres para comprobar en qué páginas aparece. Si buscamos por obras, podremos visualizar aquellas que cumplan los requisitos que les hemos impuesto además de poder reproducir el sonido que representan. Si buscamos de la manera clásica, nos devolverá las páginas en las que aparece la frase, además esa frase estará recuadrada en las páginas. Detrás de esta sencilla aplicación se encuentran las funciones de búsqueda que explicamos en el Capítulo 2.

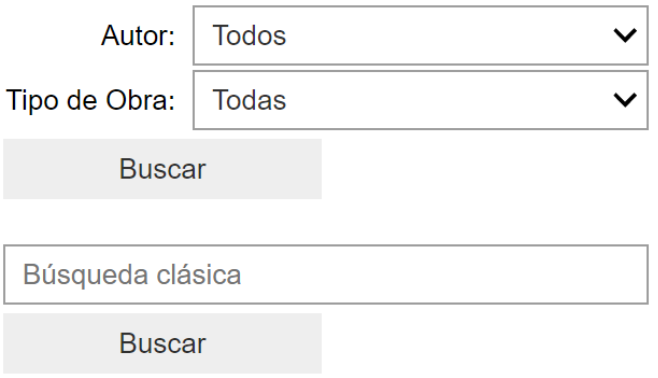


Figura 5.1: Funciones de búsqueda disponibles en la aplicación web del libro "La Música en la Catedral de Santo Domingo de la Calzada".

Por último, comentar que para crear las ventanas de búsqueda dentro de *Python* hemos hecho uso de la librería *ipywidgets* [63, 64].

Un ejemplo de búsqueda que podemos realizar dentro de la aplicación es: "Mostrar todos los salmos del autor Ignacio Cardenal" y el resultado lo podemos encontrar en la *figura 5.2.* Vemos que encima de cada obra se encuentra un reproductor de sonido, así como la página y el número de obra. Si hacemos una búsqueda solo por tipo de obra, nos aparecerá también el autor al que pertenece esa obra y si la hacemos solo por autor, nos aparecerá el tipo de obra dentro de la información.

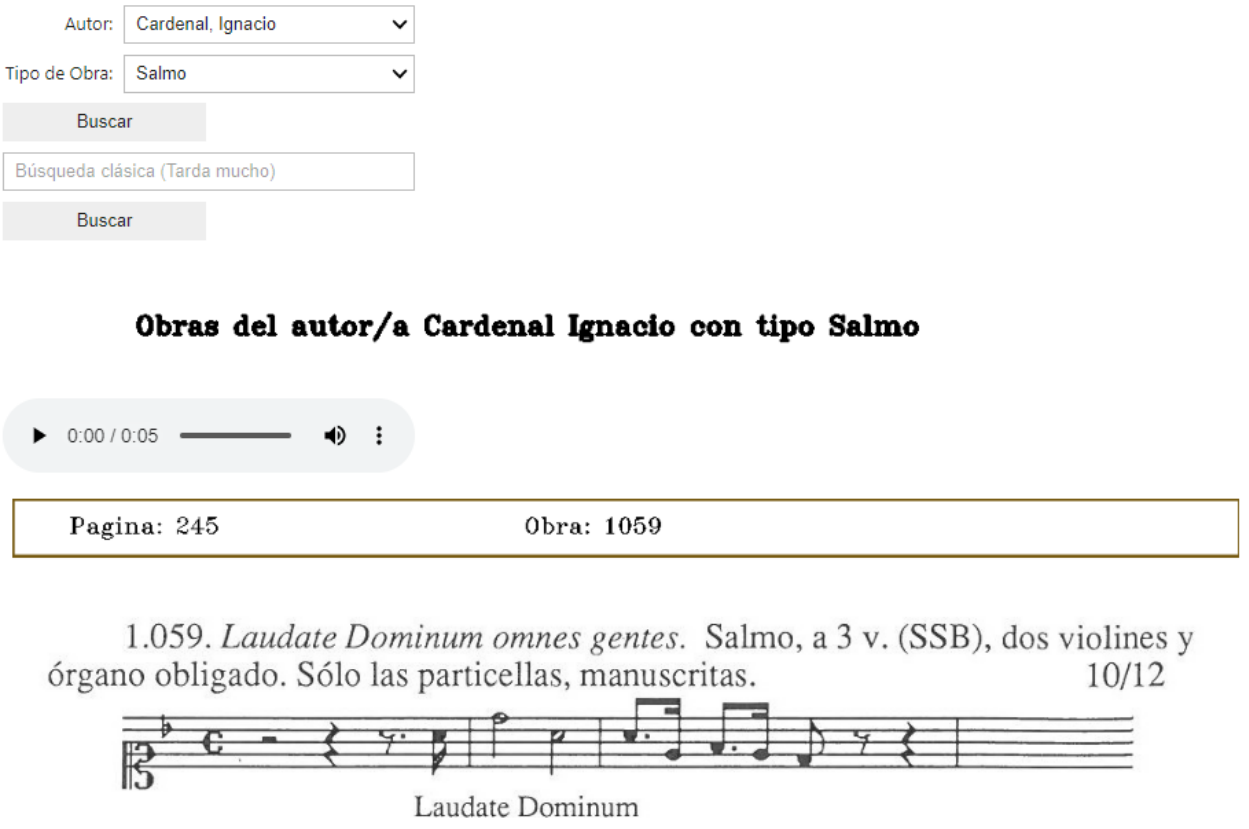


Figura 5.2: Ejemplo de consulta dentro de la aplicación web.

Capítulo 6

Conclusiones

Mientras realizaba el proceso de digitalización del libro, pensaba que no iba a tener problemas y que en poco tiempo lo iba a poder tener preparado. Esto se debe a que, al igual que esta memoria, empecé por los procesos que más rápido y fácil han salido, la localización de bloques y el *OCR*, dejando la extracción de la música para el final.

Cuando llegó el momento de ponerse con el *OMR*, me encontré con que no existía ninguna librería análoga a la de *pytesseract OCR*, por lo que tuve que buscar en la literatura si existía alguna otra opción. Aquí empezó de verdad la labor que se realiza en investigación, ya que tuve que buscar diferentes artículos para poder aprender qué técnicas se usaban y qué cosas se habían conseguido hasta el momento. Me tope con que sí había algunos programas que respondían a mis preguntas como es el caso de *Audiveris*. A pesar de ello, no funcionaban todo lo bien que me habría gustado en el libro. El *OMR* es una tecnología que todavía no ha asentado sus bases y hay muchas personas trabajando en ello.

Estuve bastante tiempo buscando qué soluciones se ofrecían a lo que yo quería hacer, pero al final decidí (una vez informado de las diferentes formas de codificar la música) resolver por mi cuenta el problema de la música. Aunque no fueron pocos los programas que probé, quizá, si hubiera seguido indagando, me hubiese topado con algo que funcionase mejor (y posiblemente de pago), pero no quería invertir más tiempo. Otro motivo que me llevó a tomar esta decisión, fue el hecho de aprender a utilizar las diferentes técnicas que tanto me interesaban de *Deep Learning* y *visión por computador*.

Para la realización de este proyecto, me ha servido de mucha ayuda tener conocimientos musicales básicos, ya que eso me ha servido para plantear el problema de la forma más natural que me ha parecido, que es la de leer pentagrama por pentagrama, símbolo por símbolo, tal como yo lo haría si estuviese tocando un instrumento. El resultado final ha sido bastante bueno, aunque todavía creo que se podría mejorar un poco más. Como posibles trabajos futuros sería interesante conseguir un *dataset* más amplio para construir modelos que sean capaces de generalizar mejor. También queda pendiente construir una página web más elaborada donde se puedan realizar las consultas del libro de forma más eficiente.

Bibliografía

[1] José López-Calo. *La Música en la Catedral de Santo Domingo de la Calzada*. Gobierno de La Rioja. Consejería de Educación, Cultura y Deportes, Enlace: `https://dialnet.unirioja.es/servlet/libro?codigo=593983`.

[2] Adrian Bradski. *Learning OpenCV, [Computer Vision with OpenCV Library ; software that sees]*. O'Reilly Media, 1. ed. edition, 2008. Gary Bradski and Adrian Kaehler.

[3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[4] Charles R. Harris, K. Jarrod Millman, Stéfan J. van der Walt, Ralf Gommers, Pauli Virtanen, David Cournapeau, Eric Wieser, Julian Taylor, Sebastian Berg, Nathaniel J. Smith, Robert Kern, Matti Picus, Stephan Hoyer, Marten H. van Kerkwijk, Matthew Brett, Allan Haldane, Jaime Fernández del Río, Mark Wiebe, Pearu Peterson, Pierre Gérard-Marchant, Kevin Sheppard, Tyler Reddy, Warren Weckesser, Hameer Abbasi, Christoph Gohlke, and Travis E. Oliphant. Array programming with NumPy. *Nature*, 585(7825):357–362, September 2020.

[5] Sunil Bangare, Amruta Dubal, Pallavi Bangare, and Suhas Patil. Reviewing otsu's method for image thresholding. *International Journal of Applied Engineering Research*, 10:21777–21783, 05 2015.

[6] Jamileh Yousefi. Image binarization using otsu thresholding algorithm. 05 2015.

[7] Sayak Paul Adrian Rosebrock, Abhishek Thanki and Jon Haase. *OCR with OpenCV, Tesseract and Python - Intro to OCR*. PyImageSearch, 2020.

[8] Sayak Paul Adrian Rosebrock, Abhishek Thanki and Jon Haase. *OCR with OpenCV, Tesseract and Python - OCR Practitioner Bundle*. PyImageSearch, 2020.

[9] Akhil Nair. Overview of tesseract ocr engine. 12 2016.

[10] *pickle - Python object serialization*. Enlace: `https://docs.python.org/3/library/pickle.html`.

[11] Jorge Calvo-Zaragoza, Jan Hajič Jr., and Alexander Pacha. Understanding optical music recognition. *ACM Comput. Surv.*, 53(4), July 2020.

[12] Ana Rebelo, Ichiro Fujinaga, Filipe Paszkiewicz, Andre R.S. Marcal, Carlos Guedes, and Jamie dos Santos Cardoso. Optical music recognition: state-of-the-art and open issues. *International Journal of Multimedia Information Retrieval*, 1(3):173–190, 2012.

[13] Zhiquing Huang, Xiang Jia, and Yifan Guo. State-of-the-art model for music object recognition with deep learning. *Applied Sciences*, 9(13):2645–2665, 2019.

[14] Andrew Hankinson, Perry Roland, and Ichiro Fujinaga. The music encoding initiative as a document-encoding framework. pages 293–298, 01 2011.

[15] The Editors of Encyclopaedia Britannica. Midi: music technology. *Encyclopedia Britannica*, 2009.

[16] Michael Good. Musicxml for notation and analysis. *Recordare LLC*, 2001.

[17] Michael Good. Musicxml: The first decade. *MakeMusic, Inc., USA*, 2013.

[18] Michael Good. Musicxml: An internet-friendly format for sheet music. *Recordare LLC*, 2001.

[19] Michael Good. Musicxml. `https://www.musicxml.com/`.

[20] Joe Wolfe. Note names, midi numbers and frequencies. `http://newt.phys.unsw.edu.au/jw/notes.html`.

[21] Michael Cuthbert and Christopher Ariza. Music21: A toolkit for computer-aided musicology and symbolic music data. pages 637–642, 01 2010.

[22] Colin Raffel and Daniel P.W. Ellis. Intuitive analysis, creation and manipulation of midi data with pretty_midi. *LabROSA, Columbia University Department of Electrical Engineering*, 2009.

[23] bibliografía omr. `https://omr-research.github.io/omr-research-sorted-by-year.html`.

[24] optical-music-recognition topic en github. `https://github.com/topics/optical-music-recognition`, 2019.

[25] Fred Meister. Awesome sheet music. `https://github.com/ad-si/awesome-sheet-music`, 2019.

[26] Alexander Pacha. Optical music recognition datasets. `https://github.com/apacha/OMR-Datasets`, 2017.

[27] Afika Nyati. cadencv: An optical music recognition system with audible playback. *Massachusetts Institute of Technology*, 2017.

[28] Afika Nyati. cadencv. `https://github.com/afikanyati/cadenCV`, 2017.

[29] Ahmed Ashraf. Mozart. `https://github.com/aashrafh/Mozart`, 2020.

[30] Felipe Roza. End-to-end learning, the (almost) every purpose ml method. 5 2019.

[31] Jorge Calvo-Zaragoza and David Rizo. End-to-end neural optical music recognition of monophonic scores. *Applied Sciences*, 8(4), 2018.

[32] Jorge Calvo Zaragoza. tf-deep-omr. `https://github.com/OMR-Research/tf-end-to-end`, 2018.

[33] The printed images of music staves (primus) dataset. `https://grfia.dlsi.ua.es/primus/`.

[34] Hervé Bitteur. Audiveris. `https://github.com/audiveris`, 2004.

[35] Nicolas Froment. Musescore. `https://github.com/musescore/MuseScore`, 2008.

[36] Opencv: Countour properties. `https://docs.opencv.org/3.4/d1/d32/tutorial_py_contour_properties.html`.

[37] Xin Jin and Jiawei Han. *K-Means Clustering*, pages 563–564. Springer US, Boston, MA, 2010.

[38] Scikit-Learn. Scikit-learn: Kmeans. `https://scikit-learn.org/stable/modules/generated/sklearn.cluster.KMeans.html`.

[39] Felipe Pezoa, Juan L Reutter, Fernando Suarez, Martín Ugarte, and Domagoj Vrgoč. Foundations of json schema. In *Proceedings of the 25th International Conference on World Wide Web*, pages 263–273. International World Wide Web Conferences Steering Committee, 2016.

[40] Lucas Vazquez. Icevision. `https://airctic.com/0.8.0/`.

[41] Lucas Vazquez. Icevision: An agnostic object detection framework. `https://github.com/airctic/icevision`.

[42] Alexey Bochkovskiy. Yolo v4, v3 and v2 for windows and linux. `https://github.com/AlexeyAB/darknet`.

[43] Jonathan Hui. map (mean average precision) for object detection. https://jonathan-hui.medium.com/map-mean-average-precision-for-object-detection-45c121a31173, 2018.

[44] Adrian Rosebrock. Intersection over union (iou) for object detection. https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/, 2016.

[45] Adrian Rosebrock. Non-maximum suppression for object detection in python. https://www.pyimagesearch.com/2014/11/17/non-maximum-suppression-object-detection-python/, 2014.

[46] Mingxing Tan, Ruoming Pang, and Quoc V. Le. Efficientdet: Scalable and efficient object detection. *CoRR*, abs/1911.09070, 2019.

[47] Shaoqing Ren, Kaiming He, Ross B. Girshick, and Jian Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[48] Tsung-Yi Lin, Priya Goyal, Ross Girshick, Kaiming He, and Piotr Dollár. Focal loss for dense object detection, 2018.

[49] Kaiming He, Georgia Gkioxari, Piotr Dollár, and Ross B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[50] Glenn Jocher. ultralytics/yolov5: Yolov5 in pythorch. https://github.com/ultralytics/yolov5.

[51] Jeremy Howard et al. fastai. https://github.com/fastai/fastai, 2018.

[52] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[53] Luis Perez and Jason Wang. The effectiveness of data augmentation in image classification using deep learning. *CoRR*, abs/1712.04621, 2017.

[54] Joseph Redmon and Ali Farhadi. YOLO9000: better, faster, stronger. *CoRR*, abs/1612.08242, 2016.

[55] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[56] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *CoRR*, abs/2004.10934, 2020.

[57] Shaoqing Ren Kaiming He, Xiangyu Zhang. Resnet-18: Deep residual learning for image recognition. `https://www.kaggle.com/pytorch/resnet18`, 2015.

[58] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[59] Peter Kazarinoff. Deploy a jupyter notebook online with voila and heroku. `https://pythonforundergradengineers.com/deploy-jupyter-notebook-voila-heroku.html`, 2020.

[60] Sylvain Corlay y otros. Código de voila. `https://github.com/voila-dashboards/voila`, 2020.

[61] Sylvain Corlay y otros. Voila. `https://voila.readthedocs.io/en/stable/`, 2020.

[62] Orion Henry James Lindenbaum, Adam Wiggins. Heroku: Cloud application platform. `https://id.heroku.com/login`, 2007.

[63] Jason Grout et al. ipywidgets: Interactive html widgets. `https://github.com/jupyter-widgets/ipywidgets`, 2017.

[64] Jason Grout et al. ipywidgets: Jupyter widgets. `https://ipywidgets.readthedocs.io/en/latest/`, 2017.