

Análisis de conveniencia de Pixeltable para la producción de RAG's multimodales

Lucas Kieffer, Mateo Centis

Resumen

Este informe tiene como objetivo documentar y analizar el proceso exploratorio en torno a la herramienta Pixeltable para la realización de RAG's multimodales para la segmentación y recuperación de datos judiciales. Se busca que el prototipo realizado sea capaz de identificar información de diferentes fuentes multimodales (haciendo especial énfasis en el procesamiento de texto) y pueda responder preguntas sobre el contenido.

Índice

1	Introducción	1
1.1	RAG	1
1.2	Pixeltable	1
	Instalación • Funcionamiento • Otras librerías	
2	Desarrollo	3
2.1	Texto	3
2.2	Multimedia	4
	Imagen • Video	
3	Pruebas y resultados	5
3.1	Texto	5
3.2	Multimedia	7
	Imagen • Video	
4	Conclusión	9
4.1	Otras alternativas	9
4.2	Evaluación final	9
	Referencias	10

1. Introducción

Previo a la explicación de la exploración y análisis realizados se debe dejar en claro cuales fueron los objetivos establecidos para este trabajo. El presente informe se separa explícitamente en dos partes: análisis de conceptos relacionados al tema de interés y estudio del qué y el cómo de la herramienta Pixeltable, es decir, que puede realizarse con ella y de qué forma. Por otro lado, la segunda instancia consta del desarrollo de un RAG multimodal utilizando la herramienta previamente analizada, capaz de recuperar información de documentos, imágenes y videos con la finalidad de determinar su alcance y factibilidad a través de las pruebas realizadas.

1.1 RAG

La generación aumentada por recuperación o *Retrieval Augmented Generation* (RAG) [1] es una arquitectura que tiene como objetivo optimizar el funcionamiento de los *Large Language Models* (LLM). Esto se debe a que estos modelos son

extremadamente generales, ya que se encuentran entrenados con información pública de todo internet y están diseñados para ejecutar todo tipo de tareas, desde traducir idiomas hasta programar aplicaciones. Por lo tanto, el objetivo de un RAG es llevar estos modelos a un dominio más específico, en el cual desarrollen la función de producir conocimiento útil especializado en nuestro campo de aplicación.

De esta forma es posible acotar el campo de información disponible a solo nuestra base de conocimiento interna, reduciendo así la posibilidad de alucinaciones, ya que solo puede responder en base a lo que nosotros le suministremos. Esto tiene como ventajas principales:

1. No se presenta falsa información al no tener la respuesta.
2. No se responde citando fuentes no autorizadas.
3. No se responde con información genérica u obsoleta.

1.2 Pixeltable

Pixeltable [2] es una infraestructura de datos declarativa para inteligencia artificial (IA), desarrollada en el lenguaje python. Se creó con el propósito de facilitar los procesos de almacenamiento, transformación, indexación y orquestación de datos de todo tipo (multimodal) de forma unificada y escalable. La clave se encuentra en que no requiere que nosotros escribamos el código de infraestructura, enfocándonos principalmente en el desarrollo del RAG. Para ello, expresa operaciones complejas a través de operaciones simples entre tablas y columnas computadas, permitiendo así almacenar información multimodal como texto, imagen, audio y video en tablas. A continuación, se procederá a explicar su funcionamiento.

1.2.1 Instalación

El proceso de instalación es muy simple y solo requiere, inicialmente, de una versión de python 3.9 o superior¹ para instalar las librerías necesarias:

¹ Se recomienda la instalación dentro de un entorno virtual para evitar conflictos de dependencias.

```
pip install Pixeltable jupyter
```

Esto permite, a priori, ejecutar código de Pixeltable dentro de python y dentro de jupyter notebooks.

1.2.2 Funcionamiento

El desarrollo en Pixeltable se basa, esencialmente, en la construcción y manipulación de tablas como en una base de datos SQL. Estas tablas son las unidades fundamentales de almacenamiento, donde cada una se componen de columnas con tipos definidos capaces de almacenar archivos de forma estructurada como no. Para crear una tabla:

```
libros : pxt.Table =
  → pxt.create_table('directorio.libros', {
    'titulo': pxt.String,
    'autor': pxt.String,
    'portada': pxt.Image
  })
```

A partir de la tabla creada ya podemos comenzar a realizar operaciones al estilo SQL para insertar, modificar y eliminar datos. En este caso se muestra un ejemplo para insertar filas a la tabla:

```
libros.insert([
  {'titulo': '1984', 'autor': 'George
  → Orwell', 'portada':
  → 'Images/1984.jpg'},
  {'titulo': 'Constitución de la Nación
  → Argentina', 'autor': 'Juan Bautista
  → Alberdi', 'portada':
  → 'Imagenes/Constitucion.jpg'}
])
```

Los datos insertados luego pueden ser accedidos a través de la combinación de los métodos `collect()`, `where()` y `select()`:

```
libros.where(libros.autor == 'George
  → Orwell').collect()

films_t.select(libros.titulo,
  → libros.autor).collect()
```

Donde en el primer ejemplo estamos obteniendo la lista de libros para el autor 'George Orwell', mientras que para el segundo ejemplo se está obteniendo la lista completa bajo título y autor. De la misma manera, como se demostró con la consigna `where()` se utiliza `update()` y `delete()` para modificar y eliminar filas de la tabla.

Una característica importante es que las tablas son **persistentes**, es decir, cuando se reinicia el kernel de python o se arranca una nueva sesión, las variables se reiniciarán, por ejemplo `libros`, pero la información se recupera desde donde se guardó, actuando así como una base de datos `libros = pxt.get_table('directorio.libros')`.

Así como se explicaron las tablas que son las unidades básicas de Pixeltable, también existen otros tipos de datos complementarios que serán utilizados durante el desarrollo (véase sección 2). Entre estos se encuentran las *views* que permiten utilizar los datos sin duplicarlos, de forma tal que se puede utilizar un subconjunto de una tabla para realizar ciertas operaciones, gestionando los datos utilizando *iterators* proporcionados por Pixeltable.

Asimismo, vale la pena mencionar la capacidad de definir columnas computadas para las tablas, es decir, columnas que se actualizan automáticamente como resultado de operaciones entre otras columnas, permitiendo además definir funciones personalizadas o *User-Defined Functions* (UDF) que pueden *linkarse* directamente con estas columnas.

1.2.3 Otras librerías

En la tarea de procesamiento de texto se requiere de un LLM (*Large Language Model*) capaz de interpretar y redactar la respuesta a la pregunta inicial en base al contexto otorgado. Este modelo de lenguaje puede ser elegido de forma arbitraria; en este trabajo se decidió avanzar con el modelo **Llama 3.2** [3] de 3 billones de parámetros, este se caracteriza por ser de código abierto y no necesita de muchos recursos para su ejecución, ya que puede ser ejecutado incluso en dispositivos móviles. Para correr el modelo, se utilizó la plataforma **Ollama** [4] que permite instalar y ejecutar LLM's en forma local. Por otro lado, para "conectar" el RAG al LLM Pixeltable proporciona una librería llamada **llama.cpp**, la cual permite integrar modelos de Ollama, en este caso, Llama 3.2. De esta manera, se pueden ejecutar *queries* al LLM desde el código de python, como si de una API se tratase.

Además del LLM, el *pipeline* de procesamiento de texto emplea un modelo específico para generar los *embeddings* de los datos. La función principal de este modelo es transformar el texto (originalmente en el formato alfabético del idioma a procesar) en un vector numérico de alta dimensionalidad. Esto permite realizar cálculos de similitud entre fragmentos de texto, ya que textos con significados similares tienden a ubicarse cerca unos de otros dentro del espacio vectorial definido por el modelo. Este paso es fundamental para el funcionamiento del sistema, ya que permite comparar la consulta del usuario con los distintos fragmentos del documento y seleccionar aquellos más relevantes en función de su similaridad (o aproximación espacial), no necesariamente por coincidencia literal de palabras.

Pixeltable utiliza en su documentación base el modelo *intfloat/e5-large-v2*. Sin embargo, esta elección plantea un desafío importante en sistemas multimodales, ya que este modelo solo toma texto como entrada para generar *tokens* (no es multimodal). En la documentación de Pixeltable sobre RAG,

para procesar cada imagen produce una consulta intermedia a un LLM (ej: GPT-4) para obtener su descripción y, luego, generar los embeddings en base a esta. Esto requiere el envío de cada imagen a un modelo externo para obtener descripciones textuales y poder ser tratadas por el modelo *intfloat/e5-large-v2*, introduciendo latencia adicional y costos extras al proceso. Además, se estaría posiblemente exponiendo información con datos sensibles. Es por ello que se tuvo en cuenta la implementación de modelos de embedding multimodales como **CLIP** [5] (OpenAI's Contrastive Language-Image Pretraining), que procesa imágenes y texto en el mismo espacio vectorial sin pasos intermedios y reduce la complejidad del pipeline al eliminar la dependencia de LLMs externos.

De la misma manera, se debe aclarar que para la detección de objetos en imágenes se realizó la integración del modelo YOLOX dentro de Pixeltable. Para ello, fue necesario utilizar *Pixeltable YOLOX*, la cual es una librería de detección de objetos *fork* de YOLOX, es decir, es una librería mantenida exclusivamente por Pixeltable y ajena a YOLOX. Su uso puede llegar a tornarse problemático, ya que no se encuentra actualizada y genera conflicto de dependencias con las últimas versiones de python. Por lo tanto, para garantizar mayor estabilidad, se necesitó de un cambio de versión de python a su *release* 3.10.10.

2. Desarrollo

Una vez cubierto el funcionamiento básico de la herramienta y configuradas las herramientas necesarias para el espacio de trabajo, se comenzó el desarrollo.

2.1 Texto

Se desarrolló un pipeline completo para la recuperación aumentada por generación (RAG) sobre documentos legales en formato PDF. Este flujo se diseñó para automatizar el proceso de segmentación, indexación y generación de respuestas a partir de preguntas en lenguaje natural. El desarrollo se organizó en torno a tres tablas principales:

- **documents:** almacena los documentos PDF originales.
- **chunks:** contiene los fragmentos textuales extraídos de los documentos, junto con sus embeddings.
- **queries:** registra las preguntas realizadas por el usuario, junto con el contexto relevante y la respuesta generada.

Los documentos fueron particionados en fragmentos mediante la clase `DocumentSplitter`, utilizando como separadores los párrafos y un límite de tokens. A cada fragmento se le calculó un embedding utilizando el modelo *CLIP*, lo que permitió construir un índice sobre la columna de texto. Una vez creado, el índice se actualizará automáticamente a medida que se inserten nuevas filas en la tabla.

```
self.chunks_view = pxt.create_view(
    f'{self.directory}.chunks',
```

```
self.documents_table,
iterator=DocumentSplitter.create(
    document=self.documents_table.docu
    → ment,
    separators='paragraph',
    → token_limit',
    limit=chunkLimit,
    metadata = 'page'
)
)
self.chunks_view.add_embedding_index(
    'text',
    embedding=clip.using(model_id='openai/
    → clip-vit-base-patch32')
)
```

Para cada pregunta insertada en la tabla queries, se definió una función computada que recupera los fragmentos más relevantes en base a la similitud semántica. Esta función devuelve los k fragmentos más similares, ordenados por relevancia.

```
@pxt.query
def _get_top_chunks(query_text : str,
    → kSize: int = 10):
    sim = self.chunks_view.text.similarity
    → (query_text)
    return (
        self.chunks_view.order_by(sim,
            → asc=False)
        .select(self.chunks_view.adjus
            → ted_page,
            → self.chunks_view.text,
            → sim=sim)
        .limit(kSize)
    )
```

A partir de los fragmentos recuperados y la pregunta original, se construyó un prompt estructurado que alimenta al modelo de lenguaje. Este prompt se transforma en una lista de mensajes estilo chat, compatible con el modelo llama3.2:3b de Ollama. Finalmente, se genera la respuesta utilizando la función chat, y se almacena el contenido generado en una columna computada adicional.

```
self.queries_table.add_computed_column(
    prompt=self._create_prompt(self.querie
    → s_table.question_context,
    → self.queries_table.question)
)
self.queries_table.add_computed_column(raw
    → _output=chat(
        messages=self._create_messages(self.qu
        → eries_table.prompt),
```

```

model=self.inference_model,
options={'max_tokens': max_tokens,
→ 'top_p': top_p, 'temperature':
→ temperature},
))

self.queries_table.add_computed_column(out_j
→ put_content=self.queries_table.raw_out_j
→ put.message.content)

```

Todo el flujo está encapsulado en funciones y columnas computadas, lo que permite que, al insertar una nueva pregunta, se dispare automáticamente el proceso de recuperación, generación de prompt y respuesta. Pixeltable almacena las respuestas y su contexto, que quedan para futuras consultas.

Este pipeline es explicado a través de las funciones más esenciales que lo definen. El proceso se completa con funciones auxiliares como la creación del prompt, creación de mensaje, ajuste de páginas, inserción de nuevos documentos y la obtención de las respuestas. Es importante aclarar que en este caso se construye el RAG definiendo un documento inicial para cargar. Luego, si se lo requiere, se puede realizar la subida de otros documentos sin perder la información anteriormente generada.

2.2 Multimedia

A fin de procesar la data de forma multimodal se realizaron dos clases capaces de gestionar el *pipeline* de procesamiento para fotos y videos. Las clases fueron diseñadas para utilizarse a través de tres métodos (además del constructor): `setup()`, `process()` y `search()`.

Para ambas clases se utilizó la librería Pixeltable YOLOX para la detección de imágenes, como fue explicado previamente (véase sección 1.2.3). Dentro de YOLOX existen varios modelos a utilizar (en orden de precisión ascendente):

- `yolox_nano`
- `yolox_tiny`
- `yolox_s`
- `yolox_m`
- `yolox_l`
- `yolox_x`

De los cuales se decidió utilizar, el modelo `yolox_m` ya que en líneas generales realiza detecciones precisas para el costo computacional que requiere. No obstante, se podría realizar un análisis en mayor profundidad según la escala de imágenes necesarias a procesar, priorizando mayor precisión o menor costo computacional.

2.2.1 Imagen

Se desarrolló la clase `ImageProcessor` que recibe en su constructor como parámetro el nombre del directorio, el modelo de YOLOX, el umbral de confianza y el modelo de *embedding* a utilizar.

Por otro lado, la función `setup()` crea la tabla para almacenar las imágenes, agrega las columnas computadas para las detecciones y el índice para los embeddings.

```

self.images_table.add_embedding_index(
    "image",
    string_embed=clip.using(model_id=self.j
→ embed_model),
    image_embed=clip.using(model_id=self.ej
→ mbed_model),
)

```

Como puede verse en el fragmento de código, se muestra la configuración para la creación de los *embeddings*, donde se utiliza el modelo CLIP explicado (véase 1.2.3) para imágenes y texto.

Continuando con el funcionamiento del pipeline, la función `process_images()` se encarga principalmente de generar una lista de las imágenes a procesar e insertarlas en la base de datos. Pixeltable requiere que las imágenes sean pasadas como objetos del tipo `{"image": image_path}` y, aconseja firmemente, que las inserciones a base de datos sean realizadas como un único bloque (no iterando cada una y realizando una inserción por imagen).

Una vez que se definió la creación de la tabla y la inserción de datos en ella, se desarrolló la consulta a través del método `search_image()` que recibe como parámetros el tipo de búsqueda (imagen o texto) y la query en sí (*path* de la imagen o texto). Se generan los tokens de la query utilizando el modelo de embedding y se hace la búsqueda por semejanza como puede verse en el siguiente fragmento de código:

```

results: pxt.dataframe.DataFrameResultSet =
→ (
    self.images_table.order_by(sim,
→ asc=False)
    .limit(limit)
    .select(self.images_table.image,
→ self.images_table.raw_detections)
    .collect()
)

```

2.2.2 Video

Análogamente a como fue explicado para el manejo de imágenes, para los videos también se realizó una clase llamada `VideoProcessor` que cuenta con métodos similares a los explicados previamente. Para no ser redundantes se hará

especial énfasis en los puntos claves que diferencian ambas clases.

Dentro de la función `setup()` se agregó una *view* para los *frames* del video, esto permite llevar a cabo una gestión del video en función de la precisión deseada, donde esta precisión se mide como la cantidad de frames o cuadros por segundo del video que se deseen almacenar.

```
self.frames_view = pxt.create_view(
    "frames",
    self.videos_table,
    iterator=FrameIterator.create(video=self.
    ↪ lf.videos_table.video,
    ↪ fps=self.fps),
    if_exists="replace_force",
)
```

De esta manera, se trata cada video como una secuencia de imágenes y se puede aplicar un procesamiento similar al explicado en la clase anterior. Asimismo, se añadió una columna computada para el audio de cada video, pudiendo almacenar el mismo en formato .mp3 utilizando la función provista por Pixeltable `extract_audio()`.

Por otro lado, las demás funciones mantienen un comportamiento similar a la clase imagen, tanto la forma de procesar los videos como a la hora de buscarlos, realizando una búsqueda de similitud en función del texto o frame buscado.

3. Pruebas y resultados

3.1 Texto

A partir del pipeline explicado en la sección anterior, se realizaron análisis de depuración con el objetivo de identificar los componentes clave que resultan determinantes para el correcto funcionamiento del RAG en las etapas de análisis, particionado y respuesta. Para ello se utilizó como primera prueba el PDF "*Leyes Orgánicas y otras (abril 2024)*" con alrededor de 304 páginas.

Dentro del flujo del pipeline, el proceso de indexación y embedding es el de mayor importancia en el desarrollo completo del RAG. Al cargar el documento, Pixeltable permite observar la división del documento en *batches* a la vez que lo indexa, convirtiendo el texto en vectores numéricos (*embedding*) para poder ser comparados. Esta tarea también depende del tipo de separador que utilizaremos. Pixeltable ofrece separadores por 'heading', 'paragraph', 'sentence', 'token_limit', 'char_limit' y 'page', pero en este caso, como indica la documentación oficial de Pixeltable, se utiliza 'token_limit', por lo tanto fue la decisión a implementar; sin embargo, también se realizaron pruebas con el tipo de separador 'paragraph' corriendo los procesos en dispositivos CPU y GPU (véase Cuadro 1).

Definir un *token_limit* apropiado es, entre muchas cosas, lo más importante para garantizar respuestas precisas a las preguntas que se ingresan como prompt. Inicializar este paráme-

Cuadro 1. Tiempos de carga

Tiempos			
Nº páginas	Dispositivo	Separador	Tiempo (seg)
304	CPU	token_limit	498
304	GPU	token_limit	78
304	CPU	paragraph	501
304	GPU	paragraph	70

tro en un número alto resulta de divisiones de texto donde se mezclan más de una idea a la vez, es decir, los fragmentos generados durante la segmentación del documento tienden a incluir múltiples párrafos o ideas combinadas. Dando como consecuencia una poca efectividad al momento de comparar una pregunta con un fragmento específico, ya que la presencia de conceptos distintos dentro del mismo texto reduce la precisión de la similitud semántica.

Por otro lado, si el separador se define como *paragraph*, entonces las divisiones (chunks o fragmentos) se tomarán a partir de cada párrafo por separado. Es decir, cada vez que se detecta el carácter de salto de línea "\n" en el texto original, el sistema interpreta que comienza un nuevo párrafo, y por lo tanto genera un nuevo fragmento. Esta estrategia es clave si se busca una estructura homogénea y clara, en la que cada partición aborda temas distintos y bien delimitados por los mismos párrafos, aunque en la mayoría de las ocasiones, la fragmentación se procesa de manera errónea en lugares particulares del documento, produciendo particiones en exceso. Esto ocurre en secciones con ítems, listados de ejemplos, lista enumerada, entre otros esquemas de escritura, donde el algoritmo define como párrafos a cada uno de estos ítems de la estructura. En el Cuadro 2, se puede apreciar cómo en un apartado de ejercicios de un documento en particular, el algoritmo define cada pregunta en fragmentos diferentes. Esto es interesante si se busca que el sistema trabaje particularmente sobre cada pregunta, de lo contrario resulta de particiones en exceso e inútiles.

Nº chunk	Texto
0	Guía práctica de comandos 1/2 \n
1	Comandos Windows/DOS \n
2	1. Todos los comandos de Windows tienen una pequeña ayuda que se invoca escribiendo /? a \n
3	continuación del comando. Obtenga la ayuda para el comando dir. \n
4	2. ¿El tipo de archivo en Windows y en Linux, se identifica de la misma manera?, ¿cómo se \n
5	3. Ejecute el comando dir y luego el comando DIR, ¿qué sucede?. \n
6	4. Muestre el contenido del directorio raíz del disco C. \n
7	5. Cuáles son los caracteres comodines en DOS?, ¿trabajan exactamente igual que en Linux?. \n

8	6. Realice las siguientes acciones:\n
9	\$p, \$g, \$l, \$d, \$t, etc. \n
10	a. Mostrar los archivos que empiecen con "W". \n
11	b. Mostrar los archivos cuya extensión sea ".EXE". \n
12	c. Mostrar todos los archivos incluyendo los ocultos. \n
13	d. Mostrar solo los archivos ocultos. \n
14	Comandos Windows/DOS \n

Cuadro 2. Chunks generados por separador paragraph

Finalmente, las pruebas realizadas sobre estos fragmentos definidos por el separador *paragraph*, no concluyeron con buenos resultados. La similitud calculada entre los fragmentos y la pregunta (que define el contexto) resultó imprecisa, ya sea por errores en la fragmentación o falta de contenido. Esto enfatiza más la idea de que las secciones con múltiples ítems deben estar idealmente contenidas en un solo fragmento, en lugar de uno por fragmento.

Otra de las pruebas realizadas tuvo como objetivo determinar si el sistema puede ubicar con precisión cierta información dentro del documento, especificando la página, capítulo, sección o título correspondiente. De esta forma, el usuario realiza las búsquedas de manera más ágil y rápida sin necesidad de leer el índice completo del documento. A partir de la investigación realizada, Pixeltable ofrece un campo opcional dentro de la función de creación de chunks, llamado *metadata*. Aquí se puede definir qué metadatos del documento se extraen por chunks; los campos que ofrece son 'título', 'encabezado', 'línea de origen', 'página' y 'cuadro delimitador'.

Se comenzaron realizando pruebas a partir de la página en la que se encuentra el fragmento. Cada fragmento ahora tiene una columna llamada "texto" y otra llamada "página ajustada", ya que la enumeración de las páginas que realiza Pixeltable comienza en cero; por lo tanto, se le debe sumar uno para obtener el número correcto de página para cada fragmento. Además, es importante aclarar que ahora el contexto que se le pasa al LLM (los k fragmentos con más similitud a la pregunta) debe estar compuesto por la página y el texto de cada fragmento. Esto se logra modificando la función de creación de prompt de la siguiente forma:

```
@staticmethod
@pxt.udf
def _create_prompt(top_k_list: list[dict],
    → question: str) -> str:
    concat_top_k = '\n\n'.join(
        f"Page: {elt['adjusted_page']}\n{e}
        → lt['text']}" for elt in
        → reversed(top_k_list)
    )
    return f'''
    PASSAGES:
```

```
{concat_top_k}
```

```
QUESTION:
```

```
{question}'''
```

Es importante mencionar, que en conjunto a esta función, además existe un método `create_messages(prompt)` que permite estructurar el mensaje de entrada al modelo de lenguaje (LLM). Esta función define el rol que debe tomar el modelo para generar una respuesta precisa. En concreto, se estableció el mensaje inicial *"You are a helpful assistant."* que actúa como contexto general para la respuesta, seguido por el contenido del `prompt` generado por la función mostrada anteriormente. De este modo, se puede controlar el comportamiento del modelo, orientándolo hacia una actitud que el desarrollador quiere. En este caso, se podría modificar este parámetro con el mensaje *"Actúas como un asistente legal en español especializado en derecho jurídico. Debes responder con precisión usando solo los documentos proporcionados"*.

Los resultados obtenidos fueron, por un lado, favorables: ahora el modelo LLM cuenta con el contexto de la página en la que se encuentra cada fragmento, lo que le permite generar respuestas más precisas respecto a la ubicación de la información dentro del documento.

Pregunta	Respuesta
¿Cuál es el artículo y en qué página se encuentra la resistencia a la autoridad penitenciaria?	El artículo que corresponde a la resistencia a la autoridad penitenciaria se encuentra en la página 178 del documento. Específicamente, es el ARTÍCULO 28 .

Cuadro 3. Ubicación del artículo sobre resistencia a la autoridad penitenciaria

Por otro lado, también se observaron limitaciones. Al analizar los k fragmentos seleccionados, se detectó que en ocasiones algunos no tenían asignado un número de página. En estos casos, se pierde la referencia dentro del documento, y el LLM intenta estimar la ubicación de la información consultada a partir de este contexto. Sin embargo, esta estimación no es precisa y puede ser errónea.

Una teoría que se mantuvo sobre la causa de esta pérdida de metadatos puede deberse a que el fragmento está compuesto por información de dos páginas a la vez, dando lugar a incoherencia por parte de Pixeltable. Sin embargo, se cree que esto no sea cierto del todo y Pixeltable está encontrando inconsistencia por otros motivos que no pudieron ser analizados con detenimiento.

Luego, se realizaron pruebas agregando como metadato de título, encabezado y línea de origen. Pero al momento de analizar cada fragmento seleccionado para conformar el

contexto del prompt, se observó que estos campos no podían ser rellenados. Esto se debe a que estos metadatos pueden no estar disponibles en el PDF o no estar siendo extraídos correctamente. Se procedió con la incorporación de otro PDF bien definido (título, subtítulos, índices, etc) de mecánica computacional y nuevamente no se encontraron los metadatos solicitados, exceptuando el número de página. La siguiente tabla muestra la pregunta y la respuesta obtenida:

Pregunta	Respuesta
¿En qué página, capítulo o título está el Ejercicio 2D sobre malla estructurada cartesiana? Respuesta Correcta: Página: 117 - Capítulo: 3. Método de volúmenes finitos (FVM)	El ejercicio "Ejercicio 2D sobre malla estructurada cartesiana" se encuentra en la página 117.

Cuadro 4. Ubicación del ejercicio sobre malla estructurada cartesiana

Por último, se hicieron pruebas en relación a como el sistema puede relacionar diferentes ideas y concluir en una respuesta relacionando ambas cosas. Para llevarlo a cabo se preguntó sobre la siguiente relación:

Pregunta: *¿Cómo se relacionan los derechos de la víctima (Ley 14.181) con los criterios de oportunidad establecidos en el Código Procesal Penal (Ley 12.734)?*

Respuesta: *Según lo indicado en las páginas del texto, los derechos de la víctima establecidos en la Ley 14.181 se relacionan con la asistencia genérica y la asistencia técnica que se le proporciona a quienes invocan su condición de víctimas, sin necesidad de patrocinio letrado. En cuanto a los criterios de oportunidad establecidos en el Código Procesal Penal (Ley 12.734), es importante destacar que la Ley 14.181 establece que la acción pública se tramitará conforme lo previsto para el procedimiento de derechos y promover su justicia. En resumen, los derechos de la víctima establecidos en la Ley 14.181 se relacionan con la asistencia genérica y técnica que se le proporciona a quienes invocan su condición de víctimas, mientras que los criterios de oportunidad establecidos en el Código Procesal Penal (Ley 12.734) se relacionan con la aplicación del procedimiento penal y la asistencia a las víctimas. Ambos están conectados para garantizar los derechos de las víctimas y promover su justicia.*

Según nuestro criterio de evaluadores como programadores (personas no especializadas en temas legales). La respuesta fue parcialmente correcta e incompleta. Se menciona adecuadamente la asistencia genérica y técnica (arts. 81 y 82, Ley 14.181), pero no se menciona la relación directa entre los derechos de la víctima y su participación en el procedimiento. Debería explicarse que la víctima tiene un rol activo en el control del criterio de oportunidad, no sólo recibir asistencia, sino poder oponerse y transformar el curso del proceso. Con

ayuda de herramientas externas para analizar esta respuesta, se propuso un puntaje de 5/10.

Pregunta: *¿Qué rol tiene la víctima en la aplicación del criterio de oportunidad según la Ley 14.181 y el Código Procesal Penal (Ley 12.734)? ¿Puede oponerse o intervenir?*

Respuesta: *Según las disposiciones legales proporcionadas, la víctima tiene un rol importante en la aplicación del criterio de oportunidad, aunque no esté constituida como querellante. La Ley 14.181 establece que la solicitud de aplicación de un criterio de oportunidad debe ser comunicada por el Tribunal a la víctima, y aunque no esté constituida como querellante, quien deberá ser oída, pudiendo formular oposición (artículo 2). Además, el Código Procesal Penal (Ley 12.734) define las diferentes categorías de oponerse o intervenir en la aplicación del criterio de oportunidad, ya sea como querellante o como persona afectada por el delito. Su intervención debe ser solicitada por el Tribunal y deberá ser escuchada en los procedimientos penales. Es importante destacar que las disposiciones legales pueden variar dependiendo de la jurisdicción y del contexto específico del caso, por lo que es recomendable consultar fuentes más actualizadas y específicas para obtener información precisa y detallada.*

La respuesta a la pregunta reformulada fue más correcta y bastante completa, con un puntaje de 8/10. Se identifica que la víctima tiene derecho a ser notificada y oída, incluso si no es querellante. Además, menciona que puede formular oposición, ya que su oposición puede implicar que la acción penal continúe como querella. Esto último demuestra que la calidad de la respuesta mejora notablemente cuando la pregunta es más específica y detallada. La segunda pregunta, al ser más concreta, permitió al sistema ofrecer una respuesta más precisa y completa.

3.2 Multimedia

Se realizó una comprobación del funcionamiento y rendimiento de las clases desarrolladas para la gestión del contenido multimedia del RAG. Para ello, se realizó un *jupyter notebook* para cada clase, en el cual se puede *testear* cada una de sus funcionalidades.

En cada uno de estos cuadernos, se definió un bloque para cada funcionalidad: creación del directorio, tablas de la base de datos, columnas computadas, inserciones de datos y recuperación vía las funciones de búsquedas definidas.

3.2.1 Imagen

Se descargó un *dataset* de miles de imágenes de cámaras de vehículos con una resolución de 480 × 300 píxeles. Se tomó una muestra aleatoria de 115 imágenes y se descargó una imagen de ejemplo de un auto rojo para comprobar la búsqueda por similitud de imagen.

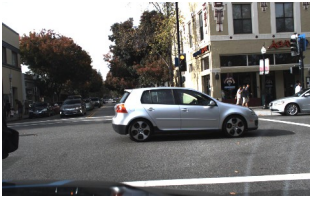


Figura 1. Imagen aleatoria del dataset



Figura 2. Imagen utilizada como *query*

Luego de realizar el procedimiento de configuración, carga y búsqueda se obtuvieron los siguientes tiempos:

Cuadro 5. Tiempos por etapa del proceso para imagen

Etapa	Tiempo (seg)
Inicialización y <i>setup</i>	20.91
Procesamiento e inserción	74.84
Búsqueda	1.34

Nótese como se puede obtener la tasa de imágenes procesadas por segundo como $t_i = \frac{115 \text{ img}}{74.84 \text{ s}} \approx 1.54 \frac{\text{img}}{\text{s}}$. Este dato es importante, ya que es el que denota la escalabilidad del procesamiento². Finalmente, a partir de la similitud de la imagen del vehículo rojo con el resto del dataset, se obtuvo como mejor resultado la siguiente imagen:



Figura 3. Resultado de la *query*

Si bien el modelo del vehículo no es exactamente el mismo, es el que más se asemeja a la imagen buscada dentro del conjunto de datos del dataset.

3.2.2 Video

En cuanto a la prueba en video, se realizó un procedimiento similar, se descargaron videos de prueba de resolución 1920×1080 y 1280×720 .

Por ejemplo, se parte de un video de resolución 1280×720 de aproximadamente 10 segundos de duración. En este video, se observan personas caminando por un parque y, durante un breve lapso de tiempo, pasa una persona en monopatín.



Figura 4. Frame del video

Se descargó una imagen de un monopatín y se realizó la búsqueda en el video por imagen, se obtuvo como resultado:

Frame #5
Time: 0.50s

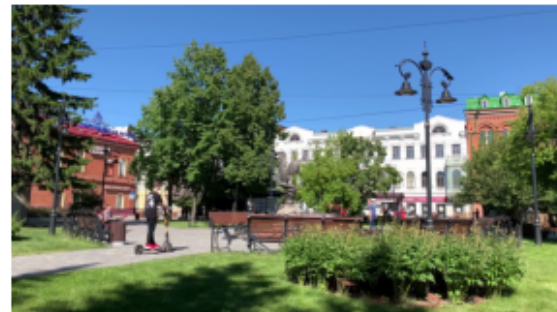


Figura 5. Resultado de la *query*

Para ello, los tiempos empleados fueron:

Cuadro 6. Tiempos por etapa del proceso para video

Etapa	Tiempo (seg)
Inicialización y <i>setup</i>	0.15
Procesamiento e inserción	7.26
Búsqueda	0.88

Observando los tiempos se puede notar como la velocidad es ampliamente superior a la registrada en el procesamiento de imágenes, obteniendo una tasa de frames de video procesados por segundo $t_f = \frac{100 \text{ img}}{7.26 \text{ s}} \approx 13,77 \frac{\text{img}}{\text{s}}$. Este resultado es sorprendente, ya que no solo es un incremento de aproximadamente 794 %, sino que además los frames del video son de una mayor resolución que las imágenes procesadas en las pruebas de la clase imagen.

No se encontró una respuesta segura ante la diferencia de tiempos obtenidos entre ambas clases, nuestras búsquedas no arrojaron resultados concretos así como tampoco la documentación menciona mejoras de rendimiento adicionales respecto del tema.

²El proceso de embedding para imágenes y video presentó fallas al utilizar la bandera *use_fast* para acelerar el procesamiento a través del procesamiento con GPU.

4. Conclusión

A lo largo de este informe hemos explorado la herramienta Pixeltable con el objetivo de analizar su funcionamiento en la construcción de RAG's multimodales.

De la misma manera, se realizó una búsqueda bibliográfica así como algunas pruebas sobre otras alternativas. Este proceso fue fundamental para entender el propósito de Pixeltable, así como para destacar sus ventajas y desventajas.

4.1 Otras alternativas

La opción más famosa de código abierto es **Langchain** [6] (licencia MIT [7]), la cual cuenta al momento de realizar este escrito con más de 112 000 estrellas en su repositorio de Github. Esta alternativa es de las más confiables y poderosas, aunque requiere de un desarrollo mucho más complejo: cada parte del proceso RAG debe implementarse cuidadosamente. Por defecto, no permite la gestión de datos multimodal de forma nativa, ya que se encuentra enfocada en texto. Tampoco ofrece un mecanismo unificado para realizar el *chunking* de documentos, no incluye una base de datos integrada (requiere de una externa como **Chroma** [8]), ni permite realizar consultas tipo SQL para insertar, modificar o consultar datos. Por ello, es necesario implementar APIs personalizadas para acceder a cada "tabla". Cada uno de estos factores, junto con otros no mencionados, hace que una implementación con esta plataforma sea costosa de desarrollar y mantener. No obstante, puede ser una gran solución para empresas con recursos suficientes que buscan un producto robusto y escalable.

Entre otras alternativas de código abierto que vale la pena mencionar se encuentran **LlamaIndex** [9] y **Haystack** [10] (ambas con licencia apache 2.0 [11]).

En cuanto a Haystack, es una alternativa probada (más de 21 mil estrellas en su repositorio) un poco más simple que Langchain y cuenta con similitudes a Pixeltable en cuanto a la base de datos utilizando el concepto de *Document Stores* (no permite gestión multimodal de forma nativa) con una programación más compleja.

Por otro lado, la más famosa de ambas alternativas es LlamaIndex (más de 43 mil estrellas en su repositorio) que mantiene un formato de programación similar a las demás alternativas vistas, con un uso de herramientas externas mayor que Haystack, con mayor similitud a lo explicado para Langchain: uso de bases de datos externa como Chroma, librerías como pydantic para la definición de modelos de datos y gestión multimodal parcial.

Una mención especial merece **RAGFlow** [12], ya que también es una de las alternativas de código abierto más famosas (más de 61 mil estrellas en Github). Esta alternativa se encuentra entre las más destacadas, sobre todo en el contexto de despliegue *on-premise*. Cuenta con muchas funcionalidades ya desarrolladas y es la solución que menos codificación requiere. Además, permite realizar RAG multimodal aunque no tiene opciones nativas para videos. Puede extenderse a través

del desarrollo de métodos propios, convirtiéndola así en una opción poderosa pero, de todas maneras, no tan flexible.

Finalmente, existen otras alternativas que no se han tenido en cuenta por ser estrictamente de pago o no permitir el *self hosting*. Este tipo de soluciones suelen ser ofrecidas como parte de un ecosistema e incluyen todo un abanico de diferentes funcionalidades adicionales como puede ser el caso de **Amazon Kendra** o **Azure AI**.

Estas soluciones implican costos significativos, aunque pueden resultar muy atractivas para grandes organizaciones que priorizan la rapidez de implementación, la robustez y la integración nativa con sus plataformas en la nube.

4.2 Evaluación final

A partir de toda la información recopilada durante el estudio exhaustivo de la herramienta **Pixeltable**, así como la revisión bibliográfica, el proceso de desarrollo y las pruebas de funcionamiento, es posible establecer una conclusión de la herramienta en general, así como respecto a las demás alternativas.

Pixeltable destaca por ofrecer equilibrio entre capacidades, flexibilidad y simplicidad de desarrollo. Tal como se detalló en la sección 1.2, ofrece herramientas de almacenamiento, transformación, indexación y orquestación de datos de todo tipo de forma **unificada** a través de una interfaz **declarativa**. Esto permite, a nivel de infraestructura, tener un pipeline de RAG multimodal en local completamente en funcionamiento teniendo solo instalado en el sistema el intérprete de python, Ollama y un modelo LLM de preferencia.

En cuanto al código, a través de su interfaz declarativa requiere de poca escritura de código (en su página establece que hasta un 80 % menos que otras alternativas), dado que como se fue enseñando a lo largo de este informe, en una misma clase en unas pocas líneas se tiene el código necesario para crear una base de datos PostgreSQL, crear sus tablas, insertar datos a las mismas y realizar las consultas y cálculos necesarios. Esta base de datos es gestionada automáticamente por Pixeltable y se encarga de almacenar embeddings y realizar cálculos vectoriales o semánticos sin tener que definir ningún método específico para esto.

De la misma manera, como ya fue mencionado, cuenta con integración nativa con modelos de lenguaje de Ollama, gestión de modelos de visión (como su propia versión de YOLOX), extracción de audio nativa y gestión nativa de imágenes y videos.

Por el contrario, otras soluciones más modulares y flexibles ya mencionadas requieren orquestar y mantener toda la infraestructura. Así como integrar manualmente múltiples componentes: bases de datos vectoriales, mecanismos de chunking, APIs externas, LLM's, entre otros componentes mencionados.

Contemplando los objetivos del informe y el contexto del mismo, donde se tiene en cuenta el manejo de datos de carácter sensible de carácter público, es requerido gestionar

y mantener una infraestructura propia y no se cuenta con equipos grandes de desarrollo, así como tampoco con grandes recursos, una alternativa como Pixeltable resulta de carácter atractivo.

Ante todo lo explicado durante esta sección, Pixeltable parecería la opción ideal. Sin embargo, durante todo el desarrollo de este proyecto nos hemos topado con limitaciones y problemáticas que, en ocasiones, perturbaban el flujo normal de trabajo. Tanto es así, que la poca información sobre estos problemas por parte de PixelTable, sumada a las diversas actualizaciones en su documentación, nos lleva a pensar que aún se encuentra en un estado "verde" o poco maduro. Aunque, a partir de la información y los resultados obtenidos, se puede afirmar que Pixeltable cuenta, a priori, con un gran potencial a futuro. Algunas de las problemáticas fueron:

■ Error en la conexión a la base de datos de Pixeltable.

Durante todo el desarrollo y prueba de esta tecnología, la conexión a la base de datos fallaba al ejecutar el código que crea o conecta las tablas, lo que hacía tedioso el proceso de prueba. Esta situación se veía agravada al ejecutar un entorno de jupyter notebook dentro del editor visual studio code donde la tasa de errores aumentaba considerablemente. Para minimizar la ocurrencia de este fallo se ejecutaba el servidor de Jupyter Notebook desde la terminal, accediendo al entorno a través del navegador web de forma local. En la documentación existe información sobre este error, no obstante, la recomendación es volver a ejecutar el código hasta que funcione, lo que no resultó de gran ayuda. Sin embargo, creemos que es muy posible que este problema se solucione en futuras actualizaciones.

■ Errores en el código

Reiteradas veces durante el desarrollo surgieron errores de sintaxis en el código, de los cuales no había documentación o información en internet disponible para resolverlos. Esta situación ralentizó considerablemente el proceso de desarrollo y pruebas.

■ Problemas internos en la librería

En algunas ocasiones algunos errores no tenían razón aparente, es decir, el código se suponía correcto pero fallaba código interno de la librería. Este tipo de errores se volvían muy difíciles de *debuggear* ya que la información disponible es escasa y herramientas como la inteligencia artificial tienen una precisión muy baja debido a la poca información pública existente.

■ Fallos al intentar reconstruir la base de datos

Al querer "destruir" directorios o tablas con información podía ocurrir que la librería fallaba y no permitía su eliminación. Para solucionar estos problemas se recurría a reiniciar el kernel de Python y, en algunas ocasiones, se creaban directorios completamente nuevos.

A partir de estos errores que detectamos, queremos realizar algunas recomendaciones con respecto a la herramienta:

■ Verificar periódicamente las actualizaciones en su repositorio oficial

Dado que Pixeltable es una herramienta que se encuentra en una etapa de importante evolución, su documentación experimenta actualizaciones frecuentes que pueden traer nuevas implementaciones que promuevan una mejor eficiencia y rendimiento para el sistema. A su vez, esto puede traer modificaciones en funciones que ya fueron desarrolladas con anterioridad.

■ Implementar pruebas para validar el flujo de datos.

Durante el desarrollo de sistemas RAG con Pixeltable, nos encontramos con la importancia de asegurar que cada etapa del pipeline funcione correctamente ante cambios en los datos o en la configuración. Recomendamos realizar pruebas que verifiquen la integridad de las columnas computadas, la correcta actualización de los embeddings y la generación del prompt. Esto no solo facilita la detección temprana de errores, sino que también permite adaptar el sistema de manera más eficiente a distintos tipos de documentos y casos de uso.

Con esto concluimos que seleccionar Pixeltable a día de hoy como tecnología de desarrollo para un RAG multimodal es una apuesta a futuro, ya que si bien cuenta con un amplio abanico de ventajas que facilitan enormemente el proceso de desarrollo y mantenimiento, aún es una tecnología que no es lo suficientemente robusta y testeada. Presentando fallas durante su desarrollo, errores irresolubles dada su documentación incompleta y cambios frecuentes en todas sus áreas. Es por ello, que para funcionar en entornos productivos exigentes creemos que aún necesita adaptaciones significativas.

Referencias

- [1] Amazon Web Services. What is retrieval-augmented generation? <https://aws.amazon.com/what-is/retrieval-augmented-generation/>.
- [2] PixelTable. Your ai data infrastructure. <https://www.pixeltable.com/>.
- [3] Meta Platforms, Inc. Llama 3.2: Revolucionando la ia y la visión de vanguardia con modelos abiertos y personalizables. <https://about.fb.com/ltam/news/2024/09/llama-3-2-revolucionando-la-ia-y-la-vision-de-vanguardia-con-modelos-abiertos-y-personalizables/>.
- [4] Ollama. Ollama – run large language models locally. <https://ollama.com/>.
- [5] Alec Radford, Jong Wook Kim, Luke Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, Gretchen Krueger, and Ilya Sutskever. Learning transferable visual

models from natural language supervision. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*. PMLR, 2021. OpenAI CLIP.

- [6] LangChain community. Langchain. <https://github.com/langchain/langchain>.
- [7] Open Source Initiative. The mit license. <https://opensource.org/license/mit>.
- [8] Chroma. Chroma: The ai-native open-source vector database. <https://www.trychroma.com/>.
- [9] run-llama / LlamaIndex. Llamaindex: Open-source data orchestration framework for llm applications. <https://www.llamaindex.ai/>.
- [10] deepset GmbH. Haystack: Open-source framework for building production-ready llm applications. <https://haystack.deepset.ai/>.
- [11] Apache Software Foundation. Apache license version 2.0. <https://www.apache.org/licenses/LICENSE-2.0>.
- [12] InfiniFlow Team. Ragflow: An open-source rag engine based on deep document understanding. <https://ragflow.io/>.