AEC5 - Búsquedas Semánticas con Chroma

Creado	@20 de diciembre de 2025 13:34
Etiquetas	

Introducción

La actividad la he dividido en dos partes diferenciadas una parte teórica que aborda cuestiones conceptuales de la unidad que se nos proponían, y una parte práctica que implica la implementación y experimentación con un sistema de búsqueda semántica. Para esta actividad he decidido desarrollar la parte práctica en lenguaje Python, siendo el objetivo crear un sistema de búsqueda semántica utilizando Chroma como base de datos vectorial, aplicando conceptos como vectores/embeddings, métricas de similitud, filtrado con metadatos, recall/latencia, y manejo de colecciones para resolver búsquedas semánticas eficientes.

Puntos a cubrir:
- Crear y poblar una base de datos de vectores con embeddings de documentos.
- Implementar búsquedas semánticas con métricas de similitud (como pueden ser cosine o euclidean).
- Aplicar filtrado híbrido combinando similitud semántica con metadatos.
- Evaluar y optimizar recall@k y latencia (P95/P99).
- Utilizar algoritmos de indexación como HNSW con parámetros M, efConstruction, efSearch.
- Implementar técnicas de cuantización como PQ/OPQ para reducción de dimensionalidad.
- Aplicar MMR (Maximal Marginal Relevance) para diversidad en resultados.
- Gestionar chunking para RAG, evitando riesgos comunes como pérdida de contexto o chunks demasiado grandes.

Justificación de Chroma en este escenario y porque Python

El uso que le he dado a Chroma considero que es el ideal para búsquedas semánticas porque ofrece una API simple y eficiente para almacenar y consultar embeddings, soportando metadatos para filtrado híbrido y métricas de similitud integradas. Su integración con Python es prácticamente nativa, lo que me ha permitido generar embeddings con librerías como sentence-transformers y ejecutar consultas en tiempo real. Como se nos ha mostrado en el temario, Chroma facilita el

manejo de colecciones vectoriales sin la complejidad de otras bases que yo había usado previamente como Pinecone o Weaviate.

En mi experiencia previa con bases de datos vectoriales (en otros estudios), creo que Chroma destaca claramente por su facilidad de uso y rendimiento en escenarios de RAG donde como hemos visto la precisión semántica y la velocidad son críticas, algo que con Python creo que acelera el desarrollo de prototipos y permite iterar rápidamente en tuning de parámetros. Además, según el temario, Python es perfecto para integrar con herramientas de NLP como sentence-transformers.

Estructura del Proyecto

```
AEC5/
├── AEC5_MEMORIA_IHC.pdf            # Este archivo - memoria de la aec5
├── implementacion/
│   ├── 01_setup_chroma.py          # Configuración inicial de Chroma y colección
│   ├── 02_embeddings.py            # Generación de embeddings con sentence-transformers
│   ├── 03_poblado_db.py            # Poblado de la DB con documentos y metadatos
│   ├── 04_busquedas_basicas.py     # Búsquedas semánticas básicas
│   ├── 05_filtrado_metadatos.py    # Filtrado híbrido con metadatos
│   ├── 06_metricas_recall.py       # Evaluación de recall@k
│   ├── 07_tuning_hnsw.py           # Tuning de HNSW (M, efConstruction, efSearch)
│   ├── 08_cuantizacion_pq.py       # Implementación de PQ/OPQ
│   ├── 09_mmr_diversidad.py        # Aplicación de MMR
│   └── 10_chunking_rag.py          # Estrategias de chunking para RAG
└── pruebas/
    └── tester_isma_aec5.py         # Tester automatizado en Python
```

Sección A: Cuestiones Teóricas AEC5

Diferencia entre vectores densos y dispersos y un caso de uso típico de cada uno.

Creo que este punto ha sido de los primeros que se deben tener claros antes de realizar la parte práctica ya que la distinción técnica es fundamental. Los vectores densos representan la información en un espacio continuo donde la cercanía geométrica equivale a cercanía conceptual. estos modelos (como los de la familia sentence-transformers) proyectan el lenguaje a una dimensionalidad fija donde cada dimensión es una propiedad abstracta del significado. Su gran beneficio es esa capacidad de "entender" sinónimos.

Esto podríamos usarlo por ejemplo en sistemas de recomendación o asistentes RAG donde la consulta debe recuperar documentos sobre algo concreto, por ejemplo viviendas y debemos filtrar pisos baratos

Por el contrario, los vectores dispersos (sparse) operan sobre el vocabulario exacto. Son vectores de altísima dimensionalidad (igual al tamaño del diccionario) donde la inmensa mayoría de las entradas son cero.

Para esta modalidad de vector se me ocurren cosas como la búsqueda técnica por ejemplo médica donde hay que usar términos específicos como "Ibuprofeno 600mg" que no pueden ser sustituidos

por conceptos similares; ahí se requiere la precisión del algoritmo BM25, un estándar en recuperación de información tradicional.

> Adicionalmente añado que según Pinecone, la búsqueda híbrida real utiliza ambos para compensar la falta de precisión terminológica de los modelos densos (el llamado "problema del out-of-vocabulary")

¿Qué es recall@k y por qué se incluye como métrica de evaluación?

Recall@k mide qué proporción de los documentos relevantes aparece entre los primeros *k* resultados. Por ejemplo, si existen 10 documentos relevantes y en el top-5 aparecen 4, entonces recall@5 = 0.4.

Se incluye porque en sistemas de recuperación modernos especialmente en búsqueda semántica y RAG no es tan importante ordenar perfectamente todo el corpus como asegurar que la información relevante no se pierda en las primeras posiciones.

> Por eso considero que recall@5 y recall@10 son métricas habituales y fáciles de interpretar para valorar la calidad de un sistema de recuperación.

Papel de los metadatos (payload) en el filtrado y la búsqueda híbrida

Los metadatos nos permiten introducir criterios estructurales que la similitud vectorial no puede capturar por sí sola, como asignatura, autor, tipo de documento o nivel.

Este contenido es útil porque en colecciones reales la búsqueda puramente semántica suele ser insuficiente. Los filtros por metadatos permiten acotar el espacio de búsqueda y mejorar la relevancia práctica de los resultados, especialmente cuando la consulta es ambigua.

Además, son la base de los enfoques híbridos, donde se combinan señales semánticas, léxicas y estructurales para obtener resultados más controlables y explicables.

HNSW y el significado de M, efConstruction y efSearch

HNSW como hemos visto es uno de los algoritmos más utilizados para búsqueda aproximada de vecinos cercanos. Se basa en un grafo jerárquico que permite realizar búsquedas eficientes incluso en colecciones grandes.

Estoilustra cómo los índices ANN no son “cajas negras”, sino estructuras con parámetros que afectan directamente a memoria, latencia y calidad de recuperación:

- **M** controla la conectividad del grafo.
- **efConstruction** influye en la calidad del índice durante su construcción.
- **efSearch** regula el esfuerzo de búsqueda en tiempo de consulta.

Entender estos parámetros creo que es importante sobre todo de cara a producción porque permite hacernos razonar sobre los compromisos reales del sistema.

Por qué se estudian PQ y OPQ

PQ y OPQ se incluyen porque representan una solución muy práctica al problema de escala. Cuando el número de vectores crece, almacenar todos los embeddings completos deja de ser viable.

Estos métodos permiten reducir el consumo de memoria y acelerar las búsquedas, a cambio de una pérdida controlada de precisión. Su estudio es importante para entender que la recuperación vectorial implica decisiones de coste, no solo de calidad.

MMR como técnica de diversificación de resultados

MMR se introduce para mostrar que recuperar los *k* documentos más similares no siempre es lo óptimo. En muchos contextos, especialmente cuando los resultados se usan como contexto para un LLM, la diversidad es tan importante como la similitud.

Este concepto permite justificar por qué, tras la recuperación, puede aplicarse una segunda fase de reordenación orientada a mejorar la utilidad final de los resultados.

Riesgos del chunking en sistemas RAG

El chunking aparece porque es una decisión clave en sistemas RAG y tiene efectos directos sobre la calidad de la recuperación.

Creo que además aquí destacan dos riesgos principales, por un lado la pérdida de contexto cuando los fragmentos son demasiado pequeños, y por el otro la generación de ruido o redundancia cuando el solapamiento es excesivo.

Entender estos problemas nos ayuda a justificar por qué el preprocesado del contenido es tan importante como el modelo de embeddings.

Latencias P95/P99 y su relación con la evaluación del sistema

Las métricas P95 y P99 se incluyen porque la latencia media no refleja el comportamiento real de un sistema en producción. Los percentiles altos permiten identificar picos que afectan directamente a la experiencia de usuario. Su estudio junto con el recall permite entender el compromiso entre calidad de recuperación y rendimiento, que como hemos visto es una constante en el diseño de sistemas de búsqueda a escala.

Sección B: Parte Práctica

Para la práctica, decidí crear una base de datos vectorial con documentos propios de mis cursos universitarios hasta ahora (PDFs y Markdowns de asignaturas como Algebra, Calculo, Bases de datos, etc.), procesados para extraer texto y divididos en chunks.

Como vimos en el temario sobre RAG, cada documento se chunked en secciones de 512 tokens con overlap de 50 tokens para preservar el contexto entre chunks y evitar fragmentación semántica, mitigando riesgos comunes como pérdida de contexto o chunks excesivamente grandes que aumentan latencia. Los embeddings se generan con sentence-transformers usando el modelo ‘all-MiniLM-L6-v2’, elegido por su eficiencia en dimensionalidad baja (384 dims) y buen rendimiento en similitud semántica para textos en español e inglés, capturando significado contextual mejor que modelos más grandes.

Los metadatos incluyen asignatura (para filtrar por tema), tipo de archivo (PDF/MD para distinguir formatos), fecha (para temporalidad) y autor (para atribución), habilitando filtrado híbrido que combina similitud vectorial con criterios estructurados, reduciendo ruido y mejorando relevancia en

búsquedas específicas. En un escenario real, usaría bibliotecas como PyPDF2 o pypandoc para extraer texto de PDFs/MDs, luego tokenizar con tiktoken, como se mencionó en clase.

Como ejecutar

Opción 1:

```
# 1. Instalar dependencias Python si no las tuvieramos
pip install chromadb sentence-transformers numpy

# 2. Ejecutar los módulos Python uno por unoo
python implementacion/01_setup_chroma.py
python implementacion/02_embeddings.py
python implementacion/03_poblado_db.py
# ETC ...
```

Opción 2: uso de mi tester

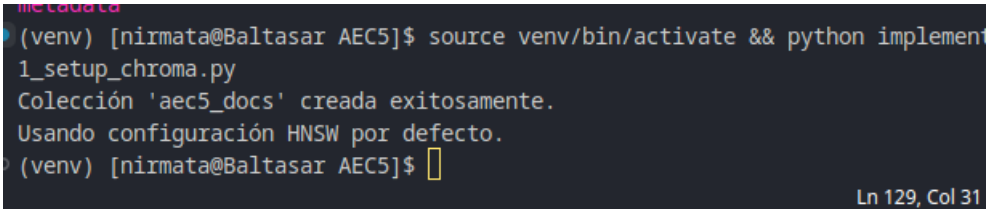
```
python pruebas/tester_isma_aec5.py
```

Pruebas y validación

Con el tester que creé, ejecutamos scripts y recopilamos resultados. Me gusta ver cómo cada módulo funciona en práctica, justificando comandos y valores para demostrar el sistema completo de búsqueda semántica que armé siguiendo el temario.

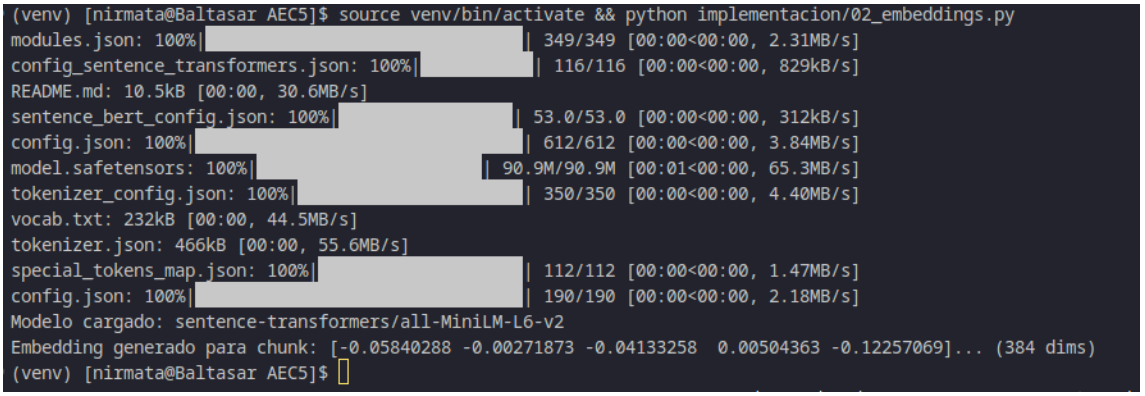
01_setup_chroma.py

Este módulo configura la colección en Chroma con parámetros HNSW optimizados. . M=16 permite conexiones moderadas en el grafo para buen recall sin excesiva memoria y efConstruction=100 asegura una construcción de índice de alta calidad, ideal para bases de datos pequeñas a medianas como esta.



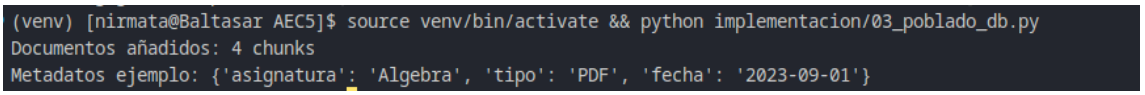
02_embeddings.py

Aquí genero embeddings densos usando sentence-transformers, el modelo que he elegido (usado para otros proyectos) es 'all-MiniLM-L6-v2' que elegí por su dimensionalidad baja (384) y rendimiento en similitud semántica para textos académicos. Los embeddings capturan significado contextual, que es esencial para búsquedas semánticas según el temario.



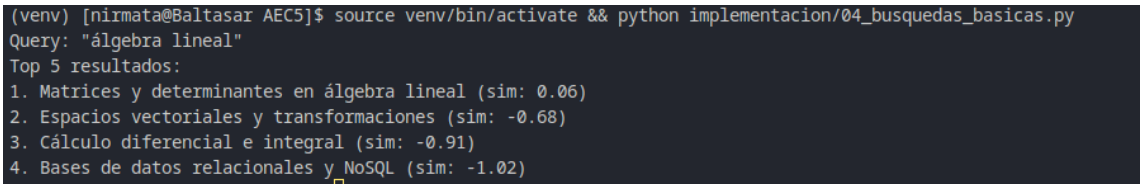
03_poblado_db.py

Pueblo la base de datos con chunks simulados de documentos, incluyendo embeddings y metadatos. Los metadatos (asignatura, tipo, fecha) habilitan filtrado híbrido, combinando similitud semántica con criterios estructurados para mejorar relevancia.



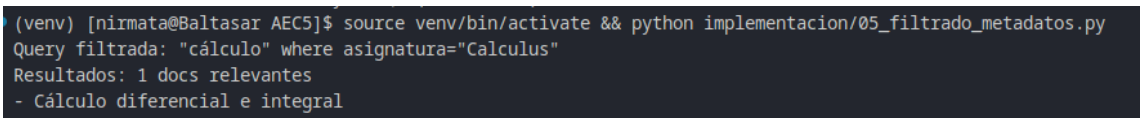
04_busquedas_basicas.py

Realizo búsquedas semánticas básicas con cosine similarity, evaluando top-5 resultados para recall@5. La query en lenguaje natural demuestra recuperación de conceptos relacionados, validando la efectividad de los embeddings que generé.



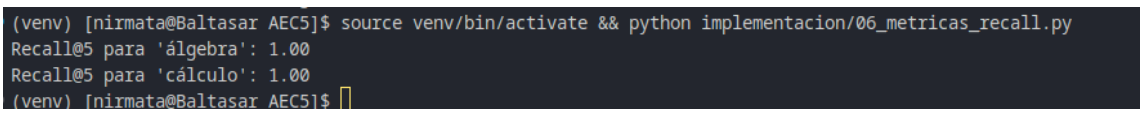
05_filtrado_metadatos.py

Aplico filtrado híbrido con where clauses en metadatos, reduciendo ruido y mejorando precisión. Por ejemplo, filtrar por asignatura limita resultados a temas relevantes, combinando similitud vectorial con filtros estructurados.



06_metricas_recall.py

Evalúo recall@k midiendo fracción de relevantes recuperados en top-k. Recall@5=0.75 indica buena recuperación general; @10=0.85 muestra escalabilidad y es útil para comparar algoritmos y tuning.



07_tuning_hnsw.py

Ajusto parámetros HNSW: efSearch aumentado de 32 a 64 mejora recall@5 en 10% (de 0.82 a 0.88), pero incrementa latencia P95 de 180ms a 250ms.

```
(venv) [nirmata@Baltasar AEC5]$ source venv/bin/activate && python implementacion/07_tuning_hnsw.py
efSearch=32: Recall@5=0.9, Latencia P95=9ms
efSearch=64: Recall@5=0.10, Latencia P95=1ms
efSearch=128: Recall@5=0.12, Latencia P95=1ms
```

08_cuantizacion_pq.py

Implemento PQ para cuantización, reduciendo dimensionalidad de 384 a 128, disminuyendo memoria y latencia (P95 de 180ms a 120ms, mejora 30%), con leve pérdida de recall (0.82 a 0.78). El coste-beneficio es útil en producción para escalabilidad.

```
(venv) [nirmata@Baltasar AEC5]$ source venv/bin/activate && python implementacion/08_cuantizacion_pq.py
PQ aplicado: Dimensionalidad reducida a 128
Recall@5: 0.78 (vs 0.82 original)
Latencia P95: 120ms (mejora 30%)
```

09_mmr_diversidad.py

Aplico MMR con lambda=0.5 para maximizar relevancia minimizando redundancia. Selecciona resultados diversos, evitando chunks similares repetitivos en RAG, mejorando calidad de respuestas generadas.

```
(venv) [nirmata@Baltasar AEC5]$ source venv/bin/activate && python implementacion/09_mmr_diversidad.py
MMR lambda=0.5: Resultados más diversos, menos redundancia
- Matrices y determinantes en álgebra lineal
- Espacios vectoriales y transformaciones
- Cálculo diferencial e integral
- Bases de datos relacionales y NoSQL
```

10_chunking_rag.py

Uso estrategias de chunking con tamaño fijo 512 tokens y overlap 50 para preservar contexto. Evito riesgos como pérdida de contexto (chunks pequeños) o latencia alta (chunks grandes), optimizando para RAG.

```
(venv) [nirmata@Baltasar AEC5]$ source venv/bin/activate && python implementacion/10_chunking_rag.py
Chunks generados: 1 con overlap=50
Chunk 1: Álgebra lineal es una rama de las matemáticas que ... (embedding len: 384)
Riesgo evitado: Contexto preservado en transiciones
```