**Description**

VectorDB is a lightweight Python package for storing and retrieving text using chunking, embedding, and vector search techniques. It provides an easy-to-use interface for saving, searching, and managing textual data with associated metadata and is designed for use cases where low latency is essential.

**Why use vector search and embeddings with large language models?**

Vector search and embeddings are essential when working with large language models because they enable efficient and accurate retrieval of relevant information from massive datasets. By converting text into high-dimensional vectors, these techniques allow for quick comparisons and searches, even when dealing with millions of documents. This makes it possible to find the most relevant results in a fraction of the time it would take using traditional text-based search methods. Additionally, embeddings capture the semantic meaning of the text, which helps improve the quality of the search results and enables more advanced natural language processing tasks.

**Usage**

Quick example that loads data into memory, and runs retrieval. All data will be handled locally, including embeddings and vector search, completely trasparent for the user with maximum possible performance.

‘’’

from vectordb import Memory

# Memory is where all content you want to store/search goes.

memory = Memory()

memory.save(

["apples are green", "oranges are orange"], # save your text content. for long text we will automatically chunk it

[{"url": "https://apples.com"}, {"url": "https://oranges.com"}], # associate any kind of metadata with it (optional))

# Search for top n relevant results, automatically using embeddings

query = "green"

results = memory.search(query, top\_n = 1)

print(results)

‘’’

This returns the chunks with the added metadata and the vector distance (where 0 is the exact match and higher means further apart)

**Options**

**Memory(memory\_file=None, chunking\_strategy={"mode":"sliding\_window"}, embeddings="normal")**

* memory\_file: *Optional.* Path to the memory file. If provided, memory will persist to disk and loaded/saved to this file.
* chunking\_strategy: *Optional.* Dictionary containing the chunking mode.

Options:  
{'mode':'sliding\_window', 'window\_size': 240, 'overlap': 8} (default)  
{'mode':'paragraph'}

* embeddings: *Optional.*

Options:  
fast - Uses Universal Sentence Encoder 4  
normal - Uses "BAAI/bge-small-en-v1.5" (default)  
best - Uses "BAAI/bge-base-en-v1.5"  
multilingual - Uses Universal Sentence Encoder Multilingual Large 3

You can also specify a custom HuggingFace model by name eg. TaylorAI/bge-micro-v2. See also [Pretrained models](https://www.sbert.net/docs/pretrained_models.html) and [MTEB](https://huggingface.co/spaces/mteb/leaderboard).

**Memory.save(texts, metadata, memory\_file=None) :** Save content to memory. Metadata will be automatically optimized to use less resources.

* texts: *Required.* Text or list of texts to be saved.
* metdata: *Optional.* Metadata or list of metadata associated with the texts.
* memory\_file: *Optional.* Path to persist the memory file. By default

**Memory.search(query, top\_n=5, unique=False, batch\_results="flatten")** : Search inside memory.

* query: *Required.* Query text or list of queries (see batch\_results option below for handling results for a list).
* top\_n: *Optional.* Number of most similar chunks to return (default: 5).
* unique: *Optional.* Return only items chunks from unique original texts (additional chunks coming from the same text will be ignored). Note this may return less chhunks than requested (default: False).
* batch\_results: *Optional.* When input is a list of queries, output algorithm can be "flatten" or "diverse". Flatten returns true nearest neighbours across all input queries, meaning all results could come from just one query. "diverse" attempts to spread out the results, so that each query's nearest neighbours are equally added (neareast first across all queries, than 2nd nearest and so on). (default: "flatten")

**Memory.clear()** : Clears the memory.

**Memory.dump()** : Prints the contents of the memory.

**Embeddings performance analysis**

We constantly evaluate embedding models using standardized benchmarks (higher is better). Average latency is measured locally on CPU (lower is better). Benchmark data pulled from [MTEB](https://huggingface.co/spaces/mteb/leaderboard).

| **Model** | **Latency** | **Benchmark 1** | **Benchmark 2** | **Benchmark 3** | **Benchmark 4** |
| --- | --- | --- | --- | --- | --- |
| all-mpnet-base-v2 | 6.12 s | 80.28 | 65.07 | 43.69 | 83.04 |
| all-MiniLM-L6-v2 | 1.14 s | 78.9 | 63.05 | 42.35 | 82.37 |
| BAAI/bge-large-en-v1.5 | 20.8 s | 83.11 | 75.97 | 46.08 | 87.12 |
| BAAI/bge-base-en-v1.5 | 6.48 s | 82.4 | 75.53 | 45.77 | 86.55 |
| BAAI/bge-small-en-v1.5 | 1.85 s | 81.59 | 74.14 | 43.82 | 84.92 |

**Vector search performance analysis**

VectorDB is also optimized for speed of retrieval. We automatically uses [Faiss](https://github.com/facebookresearch/faiss) for low number of chunks (<4000) and [mrpt](https://github.com/vioshyvo/mrpt) for high number of chunks to ensure maximum performance across the spectrum of use cases.