ChromaDB par la pratique : comment une base de données vectorielle transforme du texte en embeddings recherchables

Franck Jeannot

Montréal, Canada, AC862, Février 2025

Résumé

Les bases de données vectorielles sont un pilier des pipelines modernes de génération augmentée par récupération (RAG), mais leur fonctionnement interne reste opaque pour de nombreux praticiens. Ce tutoriel décortique ChromaDB — une base de données vectorielle open-source et embarquable — depuis l'API Python jusqu'au moteur d'inférence ONNX qui convertit du texte brut en vecteurs de 384 dimensions. À partir d'un programme minimal de 18 lignes indexant 56 phrases de politiques commerciales, nous traçons chaque étape du pipeline : tokenisation, passe avant du transformeur, mean pooling, normalisation L2, indexation HNSW et recherche par distance cosinus. Chaque étape est illustrée par du code concret, des sorties intermédiaires et des définitions mathématiques, de sorte qu'un lecteur ayant des connaissances de base en Python puisse reproduire et étendre les expériences. Nous montrons également comment remplacer le modèle anglais par défaut par un encodeur de phrases multilingue (`paraphrase-multilingual-MiniLM-L12-v2`) pour indexer et interroger des documents en français, y compris la recherche interlingue où des requêtes en anglais retrouvent des résultats pertinents en français. Nous discutons enfin du correctif de compatibilité Python 3.14 nécessaire pour les versions actuelles de ChromaDB.

Keywords: base de données vectorielle, embeddings, ChromaDB, HNSW, sentence-transformers, ONNX, RAG, similarité cosinus, embeddings multilingues, recherche interlingue

---

1. Introduction

Les applications basées sur les grands modèles de langage (LLM) ont fréquemment besoin de retrouver des documents pertinents avant de générer une réponse, un patron connu sous le nom de *génération augmentée par récupération* (RAG) [1]. L'étape de récupération nécessite une structure de données efficace capable, à partir d'une phrase de requête, de retourner les k documents les plus sémantiquement similaires en temps sous-linéaire. Les *bases de données vectorielles* remplissent ce rôle en stockant des représentations vectorielles de haute dimension du texte et en répondant à des requêtes de plus proches voisins.

ChromaDB [1] est une base de données vectorielle open-source et embarquable, écrite en Python et en Rust. Sa caractéristique distinctive pour les débutants est un pipeline d'embedding *sans configuration* : un seul appel à `collection.add(documents=...)` tokenise automatiquement le texte, exécute un modèle transformeur, normalise les vecteurs résultants et les indexe, le tout sans que l'utilisateur ait à télécharger un modèle ou écrire du code de machine learning.

Cet article répond à trois questions :

1. Que se passe-t-il, étape par étape, quand ChromaDB convertit une phrase en vecteur ?
2. Comment ces vecteurs sont-ils stockés et recherchés efficacement ?
3. Comment écrire un programme RAG complet et fonctionnel ?

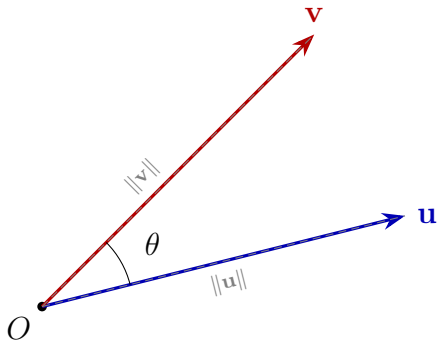
Les sources et scripts sont disponibles : https://github.com/blue101010/chromadb-article/tree/main

1. https://www.trychroma.com

2. Notions préliminaires

2.1. Embeddings de mots et de phrases

Un *embedding*[2] est une application $f : \mathcal{T} \to \mathbb{R}^d$ qui envoie un texte de longueur variable $t \in \mathcal{T}$ vers un vecteur réel de dimension fixe d.



Si $\theta = 90 \Rightarrow \cos(\theta) = 0$
(textes indépendants)

Si θ petit $\Rightarrow \cos(\theta) \approx 1$
(textes similaires)

FIGURE 1 – Interprétation géométrique de la similarité cosinus : $\cos(\theta)$ mesure l'angle entre deux vecteurs. Plus l'angle est petit, plus $\cos(\theta)$ est proche de 1 (textes similaires). Les longueurs $\|\mathbf{u}\|$ et $\|\mathbf{v}\|$ au dénominateur normalisent le résultat pour que seule la direction compte.

De bons embeddings placent les textes sémantiquement proches à faible distance et les textes dissemblables à grande distance, où « proche » est mesuré par une fonction de distance ou de similarité telle que la **similarité cosinus** :

$$\cos(\mathbf{u}, \mathbf{v}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\|\mathbf{u}\| \, \|\mathbf{v}\|} = \frac{\sum_{i=1}^{d} u_i \, v_i}{\sqrt{\sum_{i=1}^{d} u_i^2} \, \sqrt{\sum_{i=1}^{d} v_i^2}}. \tag{1}$$

2. *embedding* En français : *plongement* ou *représentation vectorielle*. Le terme anglais est couramment utilisé dans la littérature technique.

Cette formule mesure l'angle entre deux vecteurs. Une valeur proche de 1 signifie que les vecteurs pointent dans la même direction (textes très similaires), tandis qu'une valeur proche de 0 indique des directions perpendiculaires (textes sans rapport). La normalisation par les longueurs $\|\mathbf{u}\|$ et $\|\mathbf{v}\|$ garantit que seule la direction compte, pas la magnitude.
Le numérateur ($\mathbf{u} \cdot \mathbf{v}$) mesure à quel point les vecteurs vont dans la même direction. Le dénominateur normalise ce résultat par les longueurs des vecteurs, de sorte que des phrases longues et courtes ayant le même sens obtiennent le même score de similarité.

ChromaDB stocke la *distance cosinus*, définie comme $1 - \cos(\mathbf{u}, \mathbf{v})$, de sorte que les valeurs plus petites indiquent une similarité plus élevée.

2.2. Encodeurs transformeurs et BERT

Le modèle *all-MiniLM-L6-v2* utilisé par la fonction d'embedding par défaut de ChromaDB est un encodeur BERT distillé [2] avec 6 couches transformeur, 12 têtes d'attention et une taille cachée de 384. Il a été entraîné avec un objectif contrastif sur plus d'un milliard de paires de phrases pour produire des embeddings de phrases sémantiquement significatifs.

Chaque couche transformeur applique de l'auto-attention multi-têtes [3] suivie d'un réseau feed-forward :

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right) V, \tag{2}$$

où $d_k = 384/12 = 32$ est la dimension par tête. La sortie de la dernière couche est une matrice $H \in \mathbb{R}^{n \times 384}$, où n est la longueur de la séquence.

2.3. Index HNSW

Hierarchical Navigable Small World (**HNSW**) [3] est un algorithme de recherche approximative de plus proches voisins basé sur un graphe. Il construit un graphe multi-couches où :

— La couche du bas contient tous les vecteurs.

3. *Auto-attention* : mécanisme qui calcule, pour chaque mot, une représentation pondérée de tous les autres mots de la phrase. *Multi-têtes* : ce calcul est effectué 12 fois en parallèle avec des paramètres différents, capturant différents types de dépendances linguistiques. *Feed-forward* : réseau de neurones appliqué indépendamment à chaque position pour transformer les représentations.

— Chaque couche supérieure est un sous-ensemble aléatoire de la couche inférieure.
— Les arêtes connectent chaque nœud à ses M plus proches voisins dans cette couche.

Une requête commence à la couche la plus haute et descend de manière gloutonne en affinant l'ensemble de candidats à chaque couche. Les paramètres clés sont M (`max_neighbors`), `ef_construction` et `ef_search`, tous configurés par défaut par ChromaDB (Tableau 2).

3. Vue d'ensemble de l'architecture

La Figure 2 montre le flux de données complet depuis le texte brut jusqu'aux résultats de requête. Le pipeline se compose de cinq étapes, chacune détaillée en Section 5.

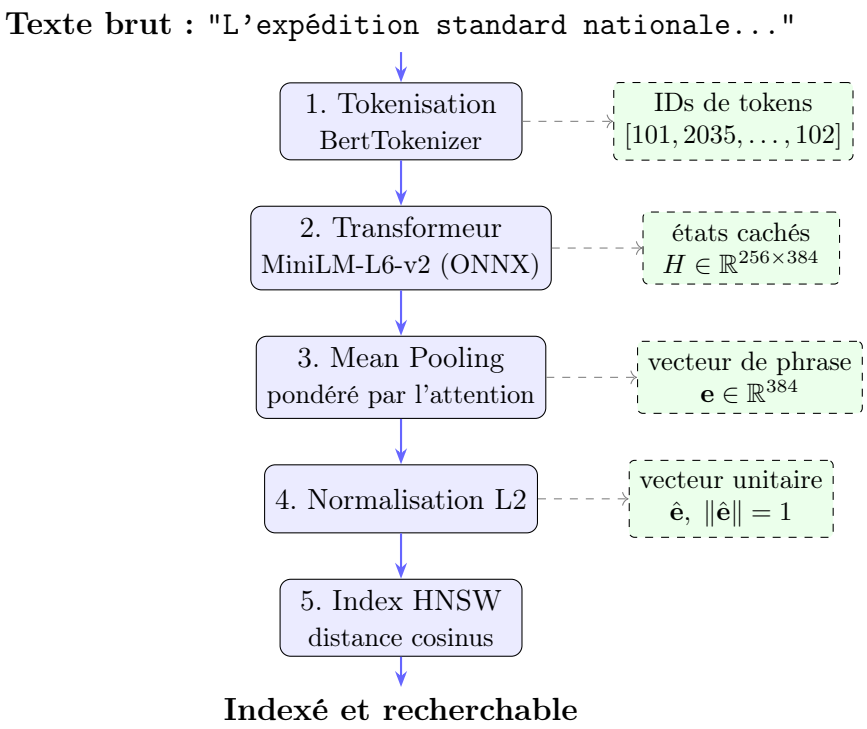


FIGURE 2 – Pipeline d'embedding par défaut de ChromaDB, du texte brut au vecteur indexé. Chaque boîte numérotée correspond à une étape décrite en Section 5.

4. Un exemple complet fonctionnel

Nous commençons par le programme complet, puis décortiquons chaque partie.

4.1. Le jeu de données : `policies.txt`

Notre jeu de données est un fichier texte de 56 lignes, chacune contenant une phrase de la politique d'expédition et de retours d'une entreprise e-commerce fictive. Les trois premières lignes sont :

```
All garments are inspected for quality before being
  packaged for shipment ...
Standard domestic shipping takes 3-5 business days ...
Expedited domestic shipping delivers within 1-2
  business days ...
```

Listing 1 – Trois premières lignes de `policies.txt`.

4.2. Indexation des documents

Le Listing 2 montre le programme d'indexation complet.

```
1  import chromadb
2  import uuid
3
4  # 1. Créer un client éphémère (en mémoire)
5  client = chromadb.Client()
6
7  # 2. Créer une collection (comme une "table" pour
       vecteurs)
8  collection = client.create_collection(name="policies")
9
10 # 3. Lire les phrases de politique
11 with open("policies.txt", "r", encoding="utf-8") as f:
12     policies: list[str] = f.read().splitlines()
13
14 # 4. Ajouter les documents -- les embeddings sont calculé
       s automatiquement
15 collection.add(
16     ids=[str(uuid.uuid4()) for _ in policies],
17     documents=policies,
```

```
    metadatas=[{"line": line} for line in range(len(
        policies))],
)

# 5. Inspecter les 10 premiers enregistrements
print(collection.peek())
```

Listing 2 – Programme d'indexation minimal ChromaDB (`main.py`).

L'exécution de ce programme produit la sortie du Listing 3, où chaque document a été transformé en vecteur de 384 dimensions.

```
{
  'ids': ['0d07bf7e-...', 'b17bedb6-...', ...],
  'embeddings': array([
    [-7.539e-02,  4.958e-02,  1.364e-02, ...,
     -1.041e-01,  7.627e-02, -1.993e-02],  # doc 0
    [ 1.046e-02, -3.367e-02,  3.771e-02, ...,
     -3.124e-02, -2.690e-03,  4.416e-02],  # doc 1
    ...
  ], shape=(10, 384)),
  'documents': [
    'All garments are inspected ...',
    'Standard domestic shipping ...',
    ...
  ],
  'metadatas': [{'line': 0}, {'line': 1}, ...]
}
```

Listing 3 – Sortie abrégée de `collection.peek()`.

4.3. Interrogation : recherche sémantique

```
results = collection.query(
    query_texts=["How long does shipping take?"],
    n_results=3,
)
for doc, dist in zip(results["documents"][0],
                     results["distances"][0]):
    print(f"  [{dist:.4f}] {doc[:80]}...")
```

Listing 4 – Interrogation de la collection pour trouver des documents similaires.

```
[0.2891] Standard domestic shipping takes 3-5
         business days after your order ...
[0.3312] Expedited domestic shipping delivers
         within 1-2 business days for orders ...
[0.4718] International shipping is available to
         over 200 destinations, with transit ...
```

Listing 5 – Résultats : top-3 par distance cosinus (plus petit = meilleur).

Le texte de la requête est transformé en embedding via le *même* pipeline que les documents. L'index HNSW retourne ensuite les trois vecteurs les plus proches par distance cosinus.

5. Décorticage du pipeline d'embedding

Lorsque l'utilisateur appelle `collection.add(documents=...)`, ChromaDB détecte qu'aucune `embedding_function` n'a été fournie et utilise par défaut `DefaultEmbeddingFunction`, qui délègue à `ONNXMiniLM_L6_V2`. Nous traçons maintenant chaque étape interne.

5.1. Étape 1 : téléchargement et mise en cache du modèle

Lors de la première utilisation, le modèle est téléchargé depuis un bucket S3 et mis en cache localement :

`~/.cache/chroma/onnx_models/all-MiniLM-L6-v2/onnx/`

Le cache contient quatre fichiers :

Table 1 – Fichiers dans le répertoire du modèle ONNX en cache.

Fichier	**Taille**	**Rôle**
`model.onnx`	∼90 Mo	Poids du transformeur (format ONNX)
`tokenizer.json`	∼700 Ko	Vocabulaire et règles BertTokenizer
`config.json`	<1 Ko	Hyperparamètres de l'architecture
`vocab.txt`	∼230 Ko	30 522 tokens WordPiece

5.2. Étape 2 : tokenisation

Le tokeniseur est un *BertTokenizer* chargé depuis `tokenizer.json` via la bibliothèque Hugging Face `tokenizers`. Il effectue :

1. **Mise en minuscules** : tout le texte est converti en minuscules.
2. **Découpage WordPiece** : les mots sont découpés en sous-tokens issus d'un vocabulaire de 30 522 entrées.
3. **Insertion de tokens spéciaux** : `[CLS]` est ajouté au début, `[SEP]` à la fin.
4. **Troncature** : les séquences de plus de 256 tokens sont tronquées.
5. **Rembourrage (padding)** : les séquences de moins de 256 tokens sont complétées à droite avec `[PAD]` (ID 0).

```python
from tokenizers import Tokenizer
import os

# Charger le même tokeniseur que ChromaDB
cache = os.path.expanduser(
    "~/.cache/chroma/onnx_models/"
    "all-MiniLM-L6-v2/onnx"
)
tok = Tokenizer.from_file(
    os.path.join(cache, "tokenizer.json")
)
tok.enable_truncation(max_length=256)
tok.enable_padding(pad_id=0, pad_token="[PAD]",
                   length=256)

text = "Standard domestic shipping takes 3-5 days"
enc = tok.encode(text)
print(enc.ids[:15])
# [101, 3115, 4968, 6554, 3138, 1017, 1011, 1019,
#  2420, 102, 0, 0, 0, 0, 0]
print(enc.attention_mask[:15])
# [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 0, 0]
```

Listing 6 – Reproduction manuelle de l'étape de tokenisation.

Le `attention_mask` distingue les vrais tokens (1) du rembourrage (0). Ce masque est essentiel pour le mean pooling (Étape 4).

5.3. *Étape 3 : passe avant du transformeur*

Les entrées tokenisées sont passées à la session d'inférence ONNX Runtime :

```python
import numpy as np
import onnxruntime as ort

session = ort.InferenceSession(
    os.path.join(cache, "model.onnx")
)

onnx_input = {
    "input_ids":      np.array([enc.ids],
                               dtype=np.int64),
    "attention_mask":  np.array([enc.attention_mask],
                               dtype=np.int64),
    "token_type_ids":  np.zeros((1, 256),
                               dtype=np.int64),
}

output = session.run(None, onnx_input)
last_hidden = output[0]  # shape: (1, 256, 384)
print(last_hidden.shape)
# (1, 256, 384)
```

Listing 7 – Inférence ONNX Runtime (simplifiée depuis le code source de ChromaDB).

Le tenseur de sortie $H \in \mathbb{R}^{1\times256\times384}$ contient un vecteur de 384 dimensions pour chaque position de token.

5.4. *Étape 4 : mean pooling*

Un embedding de phrase unique est produit en moyennant les vecteurs de tokens, mais *uniquement sur les vrais tokens* (en excluant le rembourrage) :

$$\mathbf{e} = \frac{\sum_{i=1}^{n} m_i \, \mathbf{h}_i}{\sum_{i=1}^{n} m_i}, \quad [4] \tag{3}$$

où $\mathbf{h}_i \in \mathbb{R}^{384}$ est l'état caché à la position i et $m_i \in \{0, 1\}$ est le masque d'attention.

```
mask = np.array([enc.attention_mask], dtype=np.float32)
mask_expanded = np.broadcast_to(
    np.expand_dims(mask, -1), last_hidden.shape
)

embedding = np.sum(
    last_hidden * mask_expanded, axis=1
) / np.clip(
    mask_expanded.sum(axis=1), a_min=1e-9, a_max=None
)
print(embedding.shape)   # (1, 384)
```

Listing 8 – Implémentation du mean pooling (issue du code source de ChromaDB).

5.5. Étape 5 : normalisation L2

L'embedding est normalisé à longueur unitaire afin que le produit scalaire soit égal à la similarité cosinus :

$$\hat{\mathbf{e}} = \frac{\mathbf{e}}{\|\mathbf{e}\|_2}, \qquad \text{où } \|\mathbf{e}\|_2 = \sqrt{\sum_{i=1}^{384} e_i^2}. \, [5] \tag{4}$$

4. *Mean pooling pondéré* : moyenne arithmétique des états cachés $\mathbf{h}_i$, où le masque $m_i \in \{0, 1\}$ exclut les tokens de rembourrage. Pour une phrase de 10 mots réels + 246 tokens [PAD], seuls les 10 premiers vecteurs contribuent à la moyenne. Cette opération condense une séquence de longueur variable en un embedding de taille fixe.

5. Cette normalisation ramène tous les vecteurs à une longueur unitaire (comme des points sur une sphère de rayon 1), de sorte que seule leur direction compte. Cela rend la comparaison indépendante de la longueur du texte original.

Interprétation : on divise chaque dimension du vecteur $\mathbf{e}$ *par sa longueur totale* $\|\mathbf{e}\|_2$*, obtenant un vecteur* $\hat{\mathbf{e}}$ *de longueur exactement 1. Cela garantit que deux phrases longues et courtes ayant le même sens seront considérées comme identiques, car seule leur direction dans l'espace compte, pas leur magnitude.*

Après normalisation, $\|\hat{\mathbf{e}}\| = 1$, ce qui signifie que $\cos(\hat{\mathbf{u}}, \hat{\mathbf{v}}) = \hat{\mathbf{u}} \cdot \hat{\mathbf{v}}$.

```
norm = np.linalg.norm(embedding, axis=1, keepdims=True)
norm = np.maximum(norm, 1e-12)
embedding_normed = embedding / norm

print(np.linalg.norm(embedding_normed))
# 1.0000001  (précision float32)
```

Listing 9 – Normalisation L2.

Explication ligne par ligne de la normalistion L2 :

- **Ligne 1** : Calcule la longueur euclidienne du vecteur (norme L2) : $\|\mathbf{e}\|_2 = \sqrt{e_1^2 + e_2^2 + \ldots + e_{384}^2}$. Le paramètre `axis=1` calcule la norme pour chaque ligne (document) séparément.
- **Ligne 2** : Évite la division par zéro en garantissant que la norme est au minimum 10^{-12}. Cela protège contre le cas improbable d'un vecteur nul.
- **Ligne 3** : Divise chaque dimension du vecteur par sa norme totale. C'est l'opération $\hat{\mathbf{e}} = \mathbf{e}/\|\mathbf{e}\|_2$ de l'Équation 4.
- **Ligne 5** : Vérifie que le vecteur normalisé a bien une longueur de 1 (avec la légère imprécision numérique du format `float32`).

Le vecteur résultant de 384 valeurs `float32` est ce que ChromaDB stocke et indexe.

6. L'index HNSW

Après la génération de l'embedding, les vecteurs sont insérés dans un graphe HNSW [3]. Le Tableau 2 liste les paramètres par défaut de ChromaDB.

TABLE 2 – Paramètres HNSW par défaut dans ChromaDB.

Paramètre	Défaut	Signification
`space`	cosine	Métrique de distance
`ef_construction`	100	Largeur de faisceau à la construction
`max_neighbors` (M)	16	Arêtes par nœud
`ef_search`	100	Largeur de faisceau à la recherche
`num_threads`	nb CPU	Threads parallèles

Insertion.. Lorsqu'un nouveau vecteur $\hat{\mathbf{e}}$ est ajouté, HNSW :

1. L'affecte à une couche aléatoire ℓ (distribution géométrique).
2. En partant du point d'entrée à la couche la plus haute, trouve gloutonnement le plus proche voisin à chaque couche jusqu'à ℓ.
3. Aux couches ℓ à 0, connecte le nouveau nœud à ses M plus proches voisins en élaguant les arêtes les plus longues.

Recherche.. Étant donné un vecteur de requête $\hat{\mathbf{q}}$, HNSW parcourt depuis la couche supérieure vers le bas en maintenant une liste dynamique de candidats de taille `ef_search`. À la couche du bas, les k meilleurs candidats sont retournés. La complexité est $O(\log N)$ par requête pour N vecteurs, contre $O(N)$ en force brute.

7. Assemblage complet : un exemple de récupération RAG

Le Listing 10 montre un programme de récupération complet qu'un chatbot basé sur un LLM pourrait utiliser pour répondre aux questions des clients.

```
import chromadb
import uuid

# --- Phase d'indexation ---
client = chromadb.Client()
collection = client.create_collection(
    name="policies"
)

with open("policies.txt", "r", encoding="utf-8") as f:
    policies = f.read().splitlines()
```

```python

collection.add(
    ids=[str(uuid.uuid4()) for _ in policies],
    documents=policies,
    metadatas=[{"line": i} for i in range(len(policies))
        ],
)

# --- Phase de récupération ---
queries = [
    "Can I return swimwear?",
    "Do you ship internationally?",
    "What about carbon emissions?",
]

for q in queries:
    results = collection.query(
        query_texts=[q], n_results=3
    )
    print(f"\nQuery: {q}")
    for doc, dist, meta in zip(
        results["documents"][0],
        results["distances"][0],
        results["metadatas"][0],
    ):
        print(f"  [{dist:.4f}] (ligne {meta['line']}) "
              f"{doc[:70]}...")
```

Listing 10 – Exemple complet de récupération RAG.

```
Query: Can I return swimwear?
  [0.4102] (ligne 12) Swimwear can only be returned
    with hygienic liners and all tags intact...
  [0.5238] (ligne 10) Returned items must be unworn,
    unwashed, and free of odors, stains...
  [0.5514] (ligne 11) Footwear must be returned in
    the original box, which should be placed...

Query: Do you ship internationally?
  [0.3156] (ligne 3)  International shipping is
    available to over 200 destinations...
```

```
  [0.5289] (ligne 4)  Customers are responsible for
    any local duties, taxes, or import fees...
  [0.5834] (ligne 1)  Standard domestic shipping takes
    3-5 business days after your order...

Query: What about carbon emissions?
  [0.2893] (ligne 7)  We offset 100 percent of
    shipping-related carbon emissions...
  [0.5617] (ligne 8)  Packaging materials are made
    from 100 percent recycled or sustainably...
  [0.7901] (ligne 0)  All garments are inspected for
    quality before being packaged...
```

Listing 11 – Sortie de l'exemple RAG.

Observez comment chaque requête retrouve les phrases de politique les plus pertinentes sémantiquement, même lorsque les mots exacts diffèrent (par exemple, « carbon emissions » correspond à la politique de compensation carbone).

8. Comprendre les nombres : anatomie d'un embedding

Chaque embedding est un vecteur dense de 384 nombres à virgule flottante simple précision IEEE 754. Le Tableau 3 montre des dimensions sélectionnées pour trois phrases de politique.

Table 3 – Dimensions sélectionnées des embeddings pour trois documents. Les valeurs sont arrondies à trois décimales.

Dim	**Qualité (ligne 0)**	**Livraison (ligne 1)**	**Retours (ligne 9)**
e_1	$-0{,}075$	$0{,}010$	$-0{,}021$
e_2	$0{,}050$	$-0{,}034$	$-0{,}000$
e_3	$0{,}014$	$0{,}038$	$0{,}007$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
e_{382}	$-0{,}104$	$-0{,}031$	$-0{,}013$
e_{383}	$0{,}076$	$-0{,}003$	$0{,}016$
e_{384}	$-0{,}020$	$0{,}044$	$0{,}024$
$\|\hat{\mathbf{e}}\|$	$1{,}000$	$1{,}000$	$1{,}000$

Les dimensions individuelles ne sont pas interprétables par l'humain ; le sens émerge des relations géométriques *entre* vecteurs. Deux phrases liées à la livraison auront une faible distance cosinus ($\approx 0{,}3$), tandis qu'une phrase de livraison et une phrase de retours auront une distance plus grande ($\approx 0{,}6$).

9. Configuration principale de ChromaDB

Le Tableau 4 résume les valeurs par défaut les plus importantes.

TABLE 4 – Valeurs de configuration par défaut de ChromaDB relatives aux embeddings.

Paramètre	**Valeur par défaut**	**Notes**
Modèle d'embedding	all-MiniLM-L6-v2	22M paramètres, ONNX
Dimension	384	float32
Tokens max	256	Tronqué si plus long
Taille du vocabulaire	30 522	WordPiece
Taille du batch	32	Documents par passe avant
Métrique de distance	cosinus	$1 - \cos(\mathbf{u}, \mathbf{v})$
Backend de stockage	Mémoire / SQLite	Éphémère vs. persistant

9.1. Utiliser une fonction d'embedding personnalisée

ChromaDB permet de remplacer le modèle par défaut :

```python
from chromadb.utils.embedding_functions import (
    OpenAIEmbeddingFunction,
)

ef = OpenAIEmbeddingFunction(
    api_key="sk-...",
    model_name="text-embedding-3-small",
)

collection = client.create_collection(
    name="policies",
    embedding_function=ef,
)
# collection.add() utilisera maintenant l'API OpenAI
```

Listing 12 – Utilisation des embeddings OpenAI au lieu du modèle par défaut.

10. Note de compatibilité Python 3.14

À la version 1.4.1 de ChromaDB, l'importation de la bibliothèque sous Python 3.14 échoue en raison d'un bogue de détection de version de Pydantic dans `chromadb/config.py` (issue GitHub #5996) [6]. La cause racine est que `pydantic.v1`, une couche de compatibilité rétroactive, utilise de l'introspection de métaclasses incompatible avec l'évaluation différée des annotations de Python 3.14 (PEP 749). Le correctif nécessite :

1. L'installation de `pydantic-settings`$\geq$`2.0`.
2. Le remplacement du bloc d'import dans `config.py` pour privilégier `pydantic_settings.BaseSettings`.
3. L'ajout d'annotations de type à trois champs non annotés (`chroma_coordinator_host`, `chroma_logservice_host`, `chroma_logservice_port`).

Un script de patch automatisant ces étapes est disponible dans le dépôt accompagnant cet article.

11. Embeddings multilingues : étude de cas e-commerce en français

Le modèle par défaut `all-MiniLM-L6-v2` est entraîné principalement sur des données anglaises. Pour les corpus non anglophones, le modèle ONNX intégré à ChromaDB produit des embeddings de mauvaise qualité car le vocabulaire sous-jacent et la distribution d'entraînement ne couvrent pas bien les autres langues. Cette section montre comment remplacer le modèle par défaut par un encodeur de phrases *multilingue* et démontre deux capacités puissantes : la recherche sémantique en français et la recherche *interlingue* (requêtes en anglais sur un corpus en français).

11.1. Pourquoi un modèle multilingue ?

Le modèle `paraphrase-multilingual-MiniLM-L12-v2` [5] a été entraîné sur des paires de phrases parallèles dans plus de 50 langues à l'aide d'une procédure de distillation de connaissances : un modèle enseignant anglais de haute qualité guide un modèle étudiant multilingue de sorte que des phrases sémantiquement équivalentes reçoivent des vecteurs similaires *quelle que soit la langue*. Le Tableau 5 compare les deux modèles.

6. https://github.com/chroma-core/chroma/issues/5996

TABLE 5 – Comparaison du modèle anglais par défaut et du modèle multilingue utilisé dans cette section.

	all-MiniLM-L6-v2	**paraphrase-multilingual-MiniLM-L12-v2**
Langues	Anglais seul	50+
Couches	6	12
Dim. cachée	384	384
Paramètres	22 M	118 M
Taille download	∼90 Mo	∼470 Mo
Moteur d'exéc.	ONNX (intégré)	Sentence-Transformers (PyTorch)

Le modèle multilingue utilise la même dimensionnalité de sortie (384) que le modèle par défaut, donc la configuration HNSW et les métriques de distance restent inchangées. La différence clé est que le modèle est chargé via la bibliothèque `sentence-transformers` plutôt que par le moteur ONNX intégré à ChromaDB.

11.2. Création de la fonction d'embedding multilingue

ChromaDB fournit un wrapper `SentenceTransformerEmbeddingFunction` qui délègue le calcul des embeddings à la bibliothèque `sentence-transformers`. Cela permet d'utiliser n'importe quel modèle du hub Sentence-Transformers [7].

```python
from chromadb.utils.embedding_functions import (
    SentenceTransformerEmbeddingFunction,
)

MODEL_NAME = "paraphrase-multilingual-MiniLM-L12-v2"

# Le modèle est téléchargé automatiquement au 1er appel
# (~470 Mo) et mis en cache dans
# ~/.cache/torch/sentence_transformers/
embedding_fn = SentenceTransformerEmbeddingFunction(
    model_name=MODEL_NAME,
    # device="cuda"  # décommenter pour GPU NVIDIA
)
```

7. https://huggingface.co/sentence-transformers

Listing 13 – Instanciation de la fonction d'embedding multilingue.

Le premier appel déclenche un téléchargement automatique de ∼470 Mo depuis le hub Hugging Face. Les exécutions suivantes utilisent le cache local situé dans `~/.cache/torch/sentence_transformers/`. Si un GPU compatible CUDA est disponible, passer `device="cuda"` décharge l'inférence du transformeur sur le GPU, offrant une accélération significative pour les grands lots.

11.3. Jeu de données : polices e-commerce en français

Nous utilisons une traduction française du même jeu de 56 phrases de polices e-commerce (`polices.txt`). Chaque ligne est une phrase de police en français, par exemple :

```
L'expédition standard nationale prend de 3 à 5 jours
  ouvrables après le traitement de la commande...
Les maillots de bain ne peuvent être retournés que
  si les doublures hygiéniques et toutes les
  étiquettes sont intactes...
Nous compensons 100 %% des émissions de carbone liées
  à l'expédition en investissant dans des projets
  environnementaux et de reforestation vérifiés...
```

Listing 14 – Lignes sélectionnées de `polices.txt` (polices en français).

11.4. Indexation avec métadonnées par catégorie

Pour enrichir les documents stockés, nous assignons un label de *catégorie* à chaque phrase de police à l'aide d'une simple fonction de correspondance par mots-clés. Ces métadonnées sont stockées aux côtés de l'embedding et peuvent servir à filtrer les résultats lors de l'interrogation.

```
import chromadb
import uuid

def _categorize(text: str) -> str:
    """Assigne une catégorie par mots-clés."""
    t = text.lower()
    if any(w in t for w in
           ["livraison", "expédition", "colis"]):
```

```
        return "livraison"
    if any(w in t for w in
           ["retour", "rembours", "échange"]):
        return "retours"
    if any(w in t for w in
           ["prix", "promo", "rabais"]):
        return "tarification"
    # ... catégories supplémentaires omises
    return "général"

client = chromadb.Client()

collection = client.create_collection(
    name="polices_fr",
    embedding_function=embedding_fn,
    metadata={"hnsw:space": "cosine"},
)

with open("polices.txt", "r", encoding="utf-8") as f:
    polices = [l.strip() for l in f
               if l.strip()]

collection.add(
    ids=[str(uuid.uuid4()) for _ in polices],
    documents=polices,
    metadatas=[{
        "ligne":     i,
        "categorie": _categorize(doc),
        "langue":    "fr",
    } for i, doc in enumerate(polices)],
)
```

Listing 15 – Catégorisation par mots-clés et indexation des polices françaises (`main_fr_polices.py`).

Le dictionnaire `metadata` sur la collection fixe la métrique de distance HNSW à cosinus (la valeur par défaut, rendue explicite ici pour la clarté). Les métadonnées de chaque document incluent son numéro de ligne, la catégorie assignée automatiquement et un tag de langue. ChromaDB supporte les filtres `where` sur les métadonnées, de sorte qu'une application en

aval pourrait restreindre la recherche à une catégorie spécifique (par ex., `where={"categorie": "livraison"}`).

11.5. Recherche sémantique en français

Avec les documents français indexés, nous effectuons des requêtes sémantiques *entièrement en français*. Le modèle multilingue projette à la fois les requêtes et les documents dans le même espace de 384 dimensions, de sorte que la distance cosinus reflète la similarité sémantique quelle que soit la langue.

```
requetes = [
    "Combien de temps prend la livraison ?",
    "Est-ce que je peux retourner un maillot"
    " de bain ?",
    "Comment fonctionne le programme de"
    " fidélité ?",
    "Les articles en solde sont-ils"
    " échangeables ?",
]

for query in requetes:
    results = collection.query(
        query_texts=[query],
        n_results=5,
    )
    print(f"Requête : {query}")
    for doc, dist, meta in zip(
        results["documents"][0],
        results["distances"][0],
        results["metadatas"][0],
    ):
        print(f"  [{dist:.4f}] "
              f"({meta['categorie']}) "
              f"{doc[:80]}...")
```

Listing 16 – Interrogation de la collection française avec des questions en français.

```
Requête : Combien de temps prend la livraison ?
  [0.2134] (livraison) L'expédition standard
    nationale prend de 3 à 5 jours ouvrables...
```

```
  [0.2987] (livraison) L'expédition express
    nationale livre sous 1 à 2 jours ouvrables...
  [0.4256] (livraison) La livraison internationale
    est disponible vers plus de 200 destinations...
```

Listing 17 – Résultats sélectionnés pour la requête française « Combien de temps prend la livraison ? ».

Les distances cosinus sont comparables à celles obtenues avec le modèle anglais sur des données anglaises (Section 4.3), confirmant que le modèle multilingue atteint une qualité discriminative similaire en français.

11.6. Recherche interlingue : requêtes en anglais sur des documents français

La capacité la plus remarquable d'un modèle d'embedding multilingue est sans doute la *recherche interlingue* : interroger dans une langue et retrouver des documents écrits dans une autre. Parce que le modèle a été entraîné sur des corpus parallèles, des phrases sémantiquement équivalentes dans des langues différentes sont projetées vers des points voisins dans l'espace d'embedding.

```python
queries_en = [
    "How long does shipping take?",
    "Can I return swimwear?",
    "What is your carbon offset policy?",
]

for query in queries_en:
    results = collection.query(
        query_texts=[query],
        n_results=3,
    )
    print(f"Query (EN): {query}")
    for doc, dist in zip(
        results["documents"][0],
        results["distances"][0],
    ):
        print(f"  [{dist:.4f}] {doc[:80]}...")
```

Listing 18 – Recherche interlingue : requêtes en anglais sur des documents français.

```
Query (EN): How long does shipping take?
```

```
  [0.3356] L'expédition standard nationale prend
    de 3 à 5 jours ouvrables...
  [0.4238] Nous offrons un délai de retour de
    30 jours à compter de la date de livraison...
  [0.4249] L'expédition express nationale livre
    sous 1 à 2 jours ouvrables...

Query (EN): Can I return swimwear?
  [0.5243] Les maillots de bain ne peuvent être
    retournés que si les doublures hygiéniques...
  [0.7104] Les articles retournés doivent être non
    portés, non lavés et exempts d'odeurs...
  [0.7357] Nous offrons un délai de retour de
    30 jours à compter de la date de livraison...

Query (EN): What is your carbon offset policy?
  [0.4542] Nous compensons 100 %% des émissions de
    carbone liées à l'expédition en investissant...
  [0.6586] Nous pouvons mettre à jour ces politiques
    périodiquement...
  [0.6924] Les commandes de plus de 75 dollars sont
    admissibles à la livraison standard gratuite...
```

Listing 19 – Résultats interlingues : des requêtes en anglais retrouvent des documents français pertinents.

La requête anglaise « How long does shipping take ? » retrouve correctement la phrase française sur les délais de livraison standard avec une distance cosinus de seulement 0,3356 — démontrant que le modèle place des phrases sémantiquement équivalentes dans des langues différentes très proches dans l'espace d'embedding.

11.7. Discussion

Cet exemple multilingue met en lumière trois leçons pratiques pour la construction de systèmes RAG sur du texte non anglophone :

1. **Le choix du modèle est déterminant.** Le modèle ONNX par défaut est exclusivement anglais. Pour les corpus multilingues, `SentenceTransformerEmbeddingFunction` avec un modèle tel que `paraphrase-multilingual-MiniLM-L12-v2` est un remplacement direct qui ne nécessite qu'un seul argument supplémentaire.
2. **La recherche interlingue est gratuite.** Une fois un modèle multilingue utilisé, la recherche anglais↔français (et tout autre couple de langues supporté) fonctionne immédiatement — aucun pipeline de traduction n'est nécessaire.
3. **L'enrichissement par métadonnées est orthogonal.** Les filtres de métadonnées de ChromaDB (par ex., `where={"categorie": "livraison"}`) peuvent être combinés avec la recherche sémantique quel que soit le modèle d'embedding, permettant des stratégies de recherche hybrides mêlant signaux par mots-clés et signaux sémantiques.

Le compromis est le coût d'exécution : le modèle multilingue à 12 couches est environ 2× plus lent que le modèle anglais à 6 couches sur CPU. Pour les applications sensibles à la latence, l'inférence GPU (`device="cuda"`) ou le modèle plus léger `distiluse-base-multilingual-cased-v2` (512 dimensions) peuvent être préférables.

12. Résumé et lectures complémentaires

Cet article a retracé le cycle de vie complet d'un document dans ChromaDB :

1. Le texte brut est **tokenisé** en sous-tokens WordPiece.
2. Les tokens passent par un **transformeur BERT à 6 couches** exécuté dans ONNX Runtime, produisant des états cachés de 384 dimensions pour chaque position de token.
3. Les états cachés sont agrégés par **mean pooling** (pondéré par le masque d'attention) pour condenser la séquence en un seul vecteur.
4. Le vecteur est **normalisé L2** à longueur unitaire.

5. Le vecteur unitaire est inséré dans un **graphe HNSW** qui supporte des requêtes approximatives de plus proches voisins en $O(\log N)$ par distance cosinus.

Nous avons également montré que le modèle anglais par défaut peut être remplacé par un **encodeur de phrases multilingue** (`paraphrase-multilingual-MiniLM-L12-v2`) via `SentenceTransformerEmbeddingFunction`, permettant la recherche sémantique en français et la **recherche interlingue** où des requêtes en anglais retrouvent correctement des documents français — sans aucun pipeline de traduction.

Pour approfondir, nous recommandons :

— La documentation Sentence-Transformers [8] pour entraîner des modèles d'embedding personnalisés.

— L'article HNSW [3] pour les garanties théoriques de l'index.

— La documentation ChromaDB [9] pour les patrons de déploiement en production (stockage persistant, authentification, mode distribué).

8. https://www.sbert.net
9. https://docs.trychroma.com

Références

[1] P. Lewis, E. Perez, A. Piktus, F. Petroni, V. Karpukhin, N. Goyal, H. Küttler, M. Lewis, W. Yih, T. Rocktäschel, S. Riedel et D. Kiela, « Retrieval-augmented generation for knowledge-intensive NLP tasks », dans *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 33, 2020, p. 9459–9474. arXiv :2005.11401. [En ligne]. Disponible : https://arxiv.org/abs/2005.11401

[2] W. Wang, F. Wei, L. Dong, H. Bao, N. Yang et M. Zhou, « MiniLM : Deep self-attention distillation for task-agnostic compression of pre-trained transformers », dans *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.-F. Balcan et H. Lin, Éd. Curran Associates, Inc., vol. 33, 2020, p. 5776–5788. Disponible : https://arxiv.org/abs/2002.10957

[3] Y. A. Malkov et D. A. Yashunin, « Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs », *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 42, nº 4, p. 824–836, 2020. doi : 10.1109/TPAMI.2018.2889473. arXiv :1603.09320. [En ligne]. Disponible : https://arxiv.org/abs/1603.09320

[4] J. Devlin, M.-W. Chang, K. Lee et K. Toutanova, « BERT : Pre-training of deep bidirectional transformers for language understanding », dans *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies (NAACL-HLT)*, vol. 1, 2019, p. 4171–4186. arXiv :1810.04805. [En ligne]. Disponible : https://aclanthology.org/N19-1423/

[5] N. Reimers et I. Gurevych, « Sentence-BERT : Sentence embeddings using Siamese BERT-networks », dans *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, 2019, p. 3982–3992. arXiv :1908.10084. [En ligne]. Disponible : https://aclanthology.org/D19-1410/

[6] J. Lovejoy, A. Sanchez et al., « Chroma : The AI-native open-source embedding database », Logiciel open-source, 2023. [En ligne]. Disponible : https://www.trychroma.com

[7] Microsoft, « ONNX Runtime : Cross-platform, high performance ML inferencing and training accelerator », Version 1.16, 2024. [En ligne]. Disponible : https://github.com/microsoft/onnxruntime

[8] A. Moi et al., « Tokenizers : Fast state-of-the-art tokenizers for modern NLP pipelines », Hugging Face, 2020. [En ligne]. Disponible : https://github.com/huggingface/tokenizers

[9] N. Reimers et I. Gurevych, « Sentence-Transformers : Python framework for state-of-the-art sentence, text and image embeddings », 2019. [En ligne]. Disponible : https://www.sbert.net