AYI NEDJIMI Consultants

La Vectorisation de Données | Guide IA Complet 2026

Catégorie : Intelligence Artificielle Lecture : 21 min Publié le : 07/12/2025 Auteur : Ayi NEDJIMI

Guide complet sur la vectorisation de données en IA : techniques, algorithmes, exemples de code Python et bonnes pratiques pour transformer vos.

Qu'est-ce que la vectorisation de données ?

Concrètement, un vecteur est un tableau de nombres (float) de dimension fixe. Par exemple :

- **Texte "Paris"** → `[0.23, -0.87, 0.45, ..., 0.12]` (768 dimensions avec BERT)
- **Image de chat** → `[0.01, 0.89, -0.34, ..., 0.67]` (2048 dimensions avec ResNet)
- **Signal audio 3s** → `[0.56, 0.12, -0.23, ..., 0.88]` (128 dimensions avec Mel-spectrogram)

Définition Formelle

Une fonction de vectorisation *f* est une application mathématique :
$f : X \to R^d$
où *X* est l'espace des données brutes (texte, images, etc.) et R^d est un espace vectoriel de dimension *d*.

L'objectif central est de préserver la **sémantique** des données : deux éléments similaires dans le monde réel doivent produire des vecteurs proches dans l'espace vectoriel. C'est ce qu'on appelle le **principe de similarité sémantique**.

Pourquoi vectoriser ses données ?

Les algorithmes de machine learning ne comprennent que les nombres. La vectorisation est donc **indispensable** pour permettre aux machines de traiter des données du monde réel. Voici les raisons principales :

Notre avis d'expert
L'IA responsable n'est pas un luxe — c'est une nécessité opérationnelle. Nos audits révèlent que 70% des déploiements IA en entreprise manquent de mécanismes de détection des biais et de garde-fous contre les injections de prompt. Il est temps d'intégrer la sécurité dès la conception des pipelines ML.

1. **Calculs mathématiques** : Les algorithmes (réseaux de neurones, SVM, k-means) nécessitent des inputs numériques pour effectuer opérations matricielles et calculs de gradients
2. **Comparaisons de similarité** : Les vecteurs permettent de calculer des distances (cosinus, euclidienne) pour mesurer la proximité sémantique

3. **Réduction de dimensionnalité** : Compression d'informations complexes (millions de pixels, milliers de mots) en représentations denses de 128-1536 dimensions
4. **Généralisation** : Les modèles pré-entraînés (BERT, ResNet) capturent des patterns génériques réutilisables sur de nouveaux domaines
5. **Recherche sémantique** : Interrogation des bases vectorielles pour trouver contenus similaires en millisecondes

Impact Business Concret

- **E-commerce** : Recommandation de produits similaires → +15-25% conversion
- **Support client** : Recherche automatique de tickets similaires → -40% temps de résolution
- **Content marketing** : Détection de contenus dupliqués → -60% temps d'audit SEO
- **Compliance** : Détection d'anomalies dans transactions financières → 99%+ précision

Le pipeline de vectorisation : vue d'ensemble

Un pipeline de vectorisation robuste en production comprend 6 étapes clés :

1. **Collecte et ingestion** : Import des données sources (API, databases, fichiers)
2. **Préprocessing** : Nettoyage, normalisation, filtrage des données brutes
3. **Feature extraction** : Extraction des caractéristiques pertinentes (tokenization pour texte, resize pour images)
4. **Transformation vectorielle** : Application du modèle de vectorisation (BERT, ResNet, etc.)
5. **Post-processing** : Normalisation L2, réduction de dimensionnalité (PCA/UMAP si nécessaire)
6. **Stockage** : Indexation dans une base vectorielle (Qdrant, Pinecone) ou cache (Redis)

Pipeline Python - Vue d'ensemble

```
from sentence_transformers import SentenceTransformer
import numpy as np
from qdrant_client import QdrantClient

# 1. Initialisation du modèle
model = SentenceTransformer('all-MiniLM-L6-v2')  # 384 dimensions

# 2. Préprocessing
texts = ["Paris est la capitale de France", "Berlin est en Allemagne"]
cleaned_texts = [t.strip().lower() for t in texts]

# 3-4. Vectorisation
vectors = model.encode(cleaned_texts, normalize_embeddings=True)
print(vectors.shape)  # (2, 384)

# 5. Post-processing (normalisation L2 déjà faite)
assert np.allclose(np.linalg.norm(vectors[0]), 1.0)

# 6. Stockage dans Qdrant
client = QdrantClient(":memory:")
client.create_collection(
    collection_name="documents",
    vectors_config={"size": 384, "distance": "Cosine"}
)
```

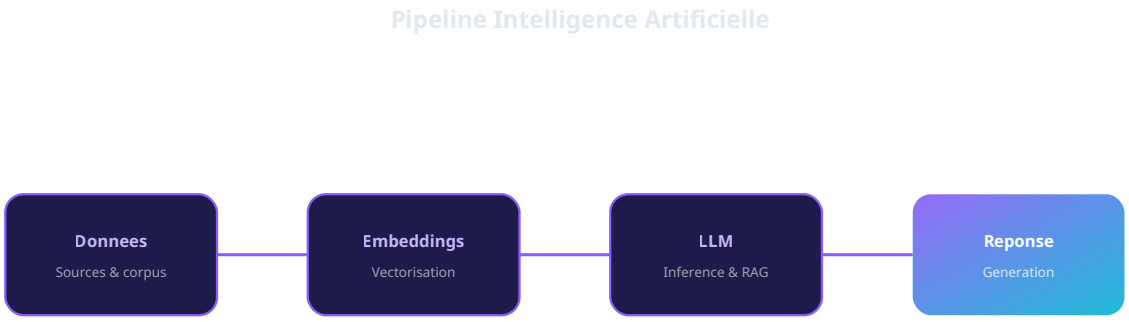
Vectorisation vs feature engineering

Ces deux concepts sont souvent confondus mais ont des objectifs distincts :

Critère	Feature Engineering	Vectorisation
Définition	Création manuelle de variables prédictives	Transformation automatique en vecteurs denses
Approche	Règles métier + expertise domaine	Deep learning + modèles pré-entraînés
Interprétabilité	Haute (features nommées : "prix", "age")	Faible (dimensions abstraites)
Scalabilité	Limitée (travail manuel intensif)	Excellente (automatisée sur millions d'exemples)
Exemple texte	Longueur phrase, nb mots capitalisés, ratio voyelles/consonnes	Embedding BERT 768D capturant sémantique

Dans la pratique moderne : On combine souvent les deux approches. Par exemple, pour un système de recommandation e-commerce :

- **Features engineered** : prix, catégorie, stock, nb vues, taux conversion (20-50 features)
- **Features vectorisées** : embedding texte description produit (384D), embedding image (512D)
- **Modèle final** : Concaténation de tous les features → 900+ dimensions totales



Architecture IA - Du traitement des donnees a la generation de reponses

Comment garantir que vos modèles de machine learning ne deviennent pas des vecteurs d'attaque ?

Types de données et stratégies de vectorisation

Données structurées (tableaux, bases de données)

Les données structurées (tables SQL, CSV, Excel) ont un schéma fixe avec colonnes typées. La vectorisation varie selon le type de feature :

Variables numériques (age, prix, score)

- **Normalisation Min-Max** : `x' = (x - min) / (max - min)` → [0, 1]
- **Standardisation Z-score** : `x' = (x - mean) / std` → moyenne 0, écart-type 1
- **Robust Scaler** : Utilise médiane et IQR (résistant aux outliers)

Variables catégorielles (pays, couleur, catégorie)

- **One-Hot Encoding** : "rouge" → [1, 0, 0], "vert" → [0, 1, 0], "bleu" → [0, 0, 1]
- **Label Encoding** : "rouge"=0, "vert"=1, "bleu"=2 (attention : introduit ordre artificiel)
- **Target Encoding** : Remplacement par moyenne de la variable cible (attention au leakage)
- **Embeddings catégoriels** : Neural networks apprennent représentations denses (style Word2Vec)

Exemple scikit-learn - Données structurées

```
import pandas as pd
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.compose import ColumnTransformer

# Dataset exemple
df = pd.DataFrame({
    'age': [25, 45, 35],
    'salary': [30000, 80000, 55000],
    'country': ['France', 'USA', 'France'],
    'category': ['A', 'B', 'A']
})

# Définition du preprocessor
preprocessor = ColumnTransformer(
    transformers=[
        ('num', StandardScaler(), ['age', 'salary']),
        ('cat', OneHotEncoder(drop='first'), ['country', 'category'])
    ])

# Transformation
vectors = preprocessor.fit_transform(df)
print(vectors.shape)  # (3, 5) -> 2 num + 3 cat (one-hot)
print(vectors)
# [[-1.22  -1.22   0.     0.   ]
#  [ 1.22   1.22   1.     1.   ]
#  [ 0.     0.     0.     0.   ]]
```

Pièges Fréquents

- **One-Hot explosion** : Feature avec 10000 catégories → 10000 colonnes (utiliser Target Encoding ou embeddings)
- **Data leakage** : Fit du scaler sur train+test ensemble → fuite d'information du test set
- **Valeurs manquantes** : Oublier de gérer les NaN avant vectorisation (scikit-learn refusera)

Données non structurées (texte, images, audio)

Les données non structurées représentent 80-90% des données d'entreprise mais sont les plus complexes à vectoriser. Elles nécessitent des modèles de deep learning spécialisés.

Texte
Challenge principal : capturer la sémantique et le contexte (synonymes, polysémie, négations)

- **Approches statistiques** : TF-IDF (sparse vectors 10000-50000D)
- **Embeddings statiques** : Word2Vec, GloVe (dense 100-300D, 1 vecteur par mot)
- **Embeddings contextuels** : BERT, RoBERTa, GPT (dense 768-1536D, 1 vecteur par phrase/document)

Images
Challenge principal : invariance aux transformations (rotation, échelle, luminosité)

- **Descripteurs classiques** : HOG, SIFT, ORB (sparse 128-1024D)
- **CNN pre-trained** : ResNet, EfficientNet, ViT (dense 512-2048D)
- **CLIP** : Embeddings multi-modaux texte-image (dense 512D)

Audio
Challenge principal : variabilité temporelle et fréquentielle (pitch, tempo, bruit)

- **Features manuels** : MFCC, Spectrogramme Mel (128-256D)
- **Modèles pré-entraînés** : Wav2Vec 2.0, Whisper (dense 512-1024D)

Type	Méthode Recommandée 2025	Dimensionnalité	Latence (CPU)
Texte court (queries)	all-MiniLM-L6-v2 (Sentence Transformers)	384D	~5ms/doc
Texte long (articles)	text-embedding-3-small (OpenAI)	1536D	~50ms/doc (API)
Images	CLIP ViT-B/32	512D	~30ms/image
Audio (parole)	Whisper encoder	512D	~100ms/3s

Données semi-structurées (JSON, XML)

Les données semi-structurées (JSON API, logs, XML) combinent structure (champs nommés) et flexibilité (schéma variable). Trois stratégies principales :

1. Flatten + Vectorisation classique
Aplatir la structure hiérarchique en colonnes plates puis appliquer techniques pour données structurées.

Cas concret
En 2023, des chercheurs ont démontré qu'il était possible de manipuler Bing Chat (Copilot) pour exfiltrer des données personnelles via des techniques d'injection de prompt indirecte. Cette attaque exploitait la capacité du LLM à accéder aux résultats de recherche web, transformant un assistant en vecteur d'exfiltration.

Flatten JSON

```
import pandas as pd
from pandas import json_normalize

json_data = {
    "user": {"id": 123, "name": "Alice"},
    "purchase": {"amount": 99.99, "items": ["book", "pen"]}
}

# Flatten
df = json_normalize(json_data)
print(df.columns)
# ['user.id', 'user.name', 'purchase.amount', 'purchase.items']

# Puis vectorisation classique (One-Hot, StandardScaler, etc.)
```

2. Vectorisation sélective par champ
Vectoriser différemment selon le champ : texte pour descriptions, numérique pour prix, etc.

3. Graph Neural Networks (GNN)
Pour JSON très profonds ou avec références, modéliser comme graphe et utiliser GNN (GraphSAGE, GAT).

Cas d'usage Réel : Logs JSON

Pour vectoriser des logs applicatifs JSON (Elasticsearch, CloudWatch) :

- **Timestamp** → Features temporels (hour_of_day, day_of_week) + normalisation
- **Message texte** → Embedding BERT 384D
- **Status code** → One-Hot encoding
- **Duration** → Log-transform puis StandardScaler
- **Concaténation finale** → Vecteur 400D pour classification anomalies

Choisir la bonne stratégie selon le type de données

Le choix de la stratégie de vectorisation dépend de 4 facteurs critiques :

1. Volume de données
- **< 10K documents** : Modèles locaux (Sentence Transformers, ResNet local)
- **10K - 1M** : Batch processing GPU (CUDA) + caching Redis
- **> 1M** : Solutions distribuées (Apache Spark + GPU clusters, APIs cloud OpenAI/Cohere)

2. Latence tolérée
- **Temps réel (<100ms)** : Modèles légers (MiniLM, DistilBERT), quantization INT8, caching agressif
- **Near real-time (1-5s)** : Modèles standard (BERT-base, ResNet-50)
- **Batch (minutes/heures)** : Modèles lourds (BERT-large, ViT-Huge), optimisation GPU

3. Qualité requise
- **Prototype/POC** : Modèles génériques pré-entraînés sans fine-tuning
- **Production standard** : Modèles pré-entraînés + fine-tuning sur 1K-10K exemples domaine
- **Production critique** : Entraînement from scratch ou fine-tuning extensif + A/B testing

4. Contraintes techniques
- **CPU only** : Modèles distillés (MiniLM-L6 : 384D, 22M paramètres)
- **GPU disponible** : Modèles standards (BERT-base : 768D, 110M paramètres)
- **Edge devices** : Quantization + ONNX Runtime (MobileBERT, TinyBERT)

Matrice de Décision Rapide

Contexte	Solution Recommandée
Startup MVP, <100K docs texte	Sentence Transformers (all-MiniLM-L6-v2) local
PME, 100K-1M docs, budget limité	Qdrant self-hosted + BGE embeddings
Grande entreprise, >10M docs, SLA strict	Pinecone/Weaviate cloud + OpenAI embeddings API
E-commerce, images produits	CLIP ViT-B/32 + Milvus/Qdrant
Médias, vidéos multi-modales	CLIP frames + Whisper audio + concaténation

Techniques de vectorisation du texte

Bag of Words (BoW) et TF-IDF

Le **Bag of Words** est la méthode la plus simple de vectorisation texte : on compte les occurrences de chaque mot. Le **TF-IDF** (Term Frequency-Inverse Document Frequency) pondère ces comptes pour favoriser les mots discriminants.

Bag of Words (BoW)

Formule : `vecteur[mot] = nombre d'occurrences du mot dans le document`

BoW avec scikit-learn

```
from sklearn.feature_extraction.text import CountVectorizer

corpus = [
    "Paris est la capitale de France",
    "Berlin est la capitale de Allemagne",
    "Madrid est en Espagne"
]

vectorizer = CountVectorizer()
X = vectorizer.fit_transform(corpus)

print(vectorizer.get_feature_names_out())
# ['allemagne' 'berlin' 'capitale' 'de' 'en' 'espagne' 'est' 'france' 'la' 'madrid'
'paris']

print(X.toarray())
# [[0 0 1 1 0 0 1 1 1 0 1]  # Doc 1
#  [1 1 1 1 0 0 1 0 1 0 0]  # Doc 2
#  [0 0 0 0 1 1 1 0 0 1 0]] # Doc 3
```

TF-IDF

Formule : `TF-IDF(t,d) = TF(t,d) × IDF(t)`

- **TF(t,d)** : Fréquence du terme t dans document d (normalisée)
- **IDF(t)** : `log(N / df(t))` où N = nb total docs, df(t) = nb docs contenant t

TF-IDF avec scikit-learn

```
from sklearn.feature_extraction.text import TfidfVectorizer

vectorizer = TfidfVectorizer(max_features=1000, ngram_range=(1,2))
X = vectorizer.fit_transform(corpus)

print(X.shape)  # (3, 16) - 16 features uniques (unigrams + bigrams)
print(X[0])  # Sparse vector avec scores TF-IDF
# Mots fréquents partout ("est", "la") ont scores faibles
# Mots rares ("Paris", "France") ont scores élevés
```

Limites de BoW/TF-IDF

- **Pas de sémantique** : "voiture" et "automobile" sont des mots complètement distincts
- **Ordre ignoré** : "chien mord homme" = "homme mord chien"
- **Sparse vectors** : 10000-50000 dimensions avec 99%+ de zéros → inefficace en mémoire
- **OOV problem** : Mots hors vocabulaire (training) sont ignorés

Quand utiliser BoW/TF-IDF ? Toujours pour baseline rapide, classification texte simple (spam detection), recherche keyword-based. Mais obsolète pour NLU moderne (remplacer par BERT).

Avez-vous évalué les risques d'injection de prompt sur vos systèmes d'IA en production ?

Word2Vec et embeddings classiques

Word2Vec (Google, 2013) a transforme le NLP en apprenant des représentations denses capturant la sémantique. Principe : les mots apparaissant dans des contextes similaires ont des significations similaires.

Architectures Word2Vec

- **CBOW (Continuous Bag of Words)** : Prédit mot central à partir du contexte environnant
- **Skip-gram** : Prédit mots de contexte à partir du mot central (meilleur pour rare words)

Word2Vec avec Gensim

```
from gensim.models import Word2Vec
import numpy as np

# Corpus tokenizé
sentences = [
    ["paris", "capitale", "france"],
    ["berlin", "capitale", "allemagne"],
    ["madrid", "capitale", "espagne"],
    ["paris", "tour", "eiffel"],
    ["berlin", "porte", "brandebourg"]
]

# Entraînement
model = Word2Vec(sentences, vector_size=100, window=3, min_count=1, epochs=100)

# Vecteur d'un mot
vec_paris = model.wv['paris']
print(vec_paris.shape)  # (100,)

# Similarités
print(model.wv.most_similar('paris', topn=3))
# [('berlin', 0.89), ('madrid', 0.87), ('capitale', 0.76)]

# Algèbre vectorielle célèbre
result = model.wv.most_similar(positive=['paris', 'allemagne'], negative=['france'])
print(result[0])  # ('berlin', 0.92) - analogie "roi-homme+femme=reine"
```

GloVe (Global Vectors)

GloVe (Stanford, 2014) combine avantages de matrix factorization et Word2Vec. Pré-entraîné sur Wikipedia/Common Crawl (6B tokens), disponible en 50D, 100D, 200D, 300D.

FastText

FastText (Facebook, 2016) améliore Word2Vec en représentant chaque mot comme somme de ses n-grams de caractères. Avantage : gère les OOV (out-of-vocabulary) et morphologie. Pour approfondir, consultez Embeddings et Recherche Documentaire.

FastText pour documents entiers

```
import numpy as np
from gensim.models.fasttext import FastText

# Modèle FastText
model = FastText(sentences, vector_size=100, window=3, min_count=1)

# Vectoriser un document = moyenne des vecteurs de mots
def document_vector(doc, model):
    vectors = [model.wv[word] for word in doc if word in model.wv]
    if len(vectors) == 0:
        return np.zeros(model.vector_size)
    return np.mean(vectors, axis=0)

doc = ["paris", "est", "belle"]
vec = document_vector(doc, model)
print(vec.shape)  # (100,)
```

Modèle	Année	Dimensions	Force	Faiblesse
Word2Vec	2013	100-300D	Rapide, efficace, baseline solide	Statique (1 vecteur/mot), ignore polysémie
GloVe	2014	50-300D	Modèles pré-entraînés de qualité	Mêmes limites que Word2Vec
FastText	2016	100-300D	Gère OOV, morphologie riche	Plus lent que Word2Vec

Transformers et modèles contextuels (BERT, GPT)

Les **Transformers** (2017) et **BERT** (Google, 2018) ont obsolétisé Word2Vec en introduisant des **embeddings contextuels** : le vecteur d'un mot dépend de son contexte dans la phrase.

BERT (Bidirectional Encoder Representations from Transformers)
BERT lit la phrase dans les deux sens (gauche-droite ET droite-gauche) pour capturer le contexte complet. Résultat : "banque" dans "banque de France" ≠ "banque" dans "s'asseoir sur la banque".

BERT avec Hugging Face Transformers

```
from transformers import AutoTokenizer, AutoModel
import torch

# Chargement modèle BERT
model_name = "bert-base-uncased"
tokenizer = AutoTokenizer.from_pretrained(model_name)
model = AutoModel.from_pretrained(model_name)

# Texte à vectoriser
text = "Paris est la capitale de France"

# Tokenization
inputs = tokenizer(text, return_tensors="pt", padding=True, truncation=True)

# Inférence
with torch.no_grad():
    outputs = model(**inputs)

# Extraction embedding [CLS] (représentation de la phrase entière)
sentence_embedding = outputs.last_hidden_state[:, 0, :]
print(sentence_embedding.shape)  # (1, 768)

# Alternative : moyenne de tous les tokens
mean_embedding = outputs.last_hidden_state.mean(dim=1)
print(mean_embedding.shape)  # (1, 768)
```

Sentence Transformers (SBERT)
Sentence-BERT (2019) fine-tune BERT spécifiquement pour produire des sentence embeddings de qualité optimisés pour similarité cosinus. **Solution recommandée en 2025** pour la plupart des cas d'usage.

Sentence Transformers - Production Ready

```
from sentence_transformers import SentenceTransformer
import numpy as np

# Modèles populaires :
# - all-MiniLM-L6-v2 : 384D, 22M params, rapide (5ms/doc CPU)
# - all-mpnet-base-v2 : 768D, 110M params, meilleure qualité (15ms/doc CPU)
# - paraphrase-multilingual : Support 50+ langues

model = SentenceTransformer('all-MiniLM-L6-v2')

# Vectorisation (batch processing automatique)
documents = [
    "Paris est la capitale de France",
    "Berlin est la capitale de l'Allemagne",
    "J'aime le chocolat"
]

embeddings = model.encode(documents, normalize_embeddings=True)
print(embeddings.shape)  # (3, 384)

# Similarité cosinus
from sklearn.metrics.pairwise import cosine_similarity
sim_matrix = cosine_similarity(embeddings)
print(sim_matrix)
# [[1.   0.87 0.12]  # Paris-Berlin très similaires
#  [0.87 1.   0.09]  # Paris-Chocolat peu similaires
#  [0.12 0.09 1.  ]]
```

OpenAI Embeddings (API)

OpenAI propose des embeddings API de très haute qualité : **text-embedding-3-small** (1536D, $0.02/1M tokens) et **text-embedding-3-large** (3072D, $0.13/1M tokens).

OpenAI Embeddings API

```
from openai import OpenAI
import os

client = OpenAI(api_key=os.getenv("OPENAI_API_KEY"))

def get_embedding(text, model="text-embedding-3-small"):
    text = text.replace("\n", " ")
    response = client.embeddings.create(input=[text], model=model)
    return response.data[0].embedding

embedding = get_embedding("Paris est la capitale de France")
print(len(embedding))  # 1536
print(embedding[:5])  # [0.023, -0.018, 0.045, -0.012, 0.067]
```

Recommandations 2025

- **Prototypes/POC** : Sentence Transformers (all-MiniLM-L6-v2) local
- **Production PME** : Sentence Transformers (all-mpnet-base-v2) self-hosted
- **Production scale** : OpenAI text-embedding-3-small API
- **Multilingue** : paraphrase-multilingual-mpnet-base-v2 ou BGE-M3
- **Domaine spécialisé** : Fine-tuning BERT sur corpus métier

Comparaison des approches et cas d'usage

Approche	Dimensionnalité	Sémantique	Latence (CPU)	Coût	Cas d'usage
TF-IDF	10K-50K (sparse)	Aucune	<1ms	Gratuit	Baseline, spam detection, keyword search
Word2Vec	100-300D (dense)	Moyenne (statique)	~1ms	Gratuit	Classification texte, clustering basique
BERT	768D (dense)	Excellente (contextuelle)	15-50ms	Gratuit (self-hosted)	NLU avancé, Q&A, classification fine
Sentence Transformers	384-768D	Excellente (optimisée similarité)	5-15ms	Gratuit	RAG, recherche sémantique, recommandation
OpenAI API	1536D-3072D	Excellente++	50-200ms (API)	$0.02-0.13/1M tokens	Production scale, multilingue, zero-shot

Arbre de Décision
Comment choisir ?

- **Vous avez besoin de keyword matching exact** → TF-IDF
- **Vous voulez comprendre la sémantique mais budget/latence limités** → Sentence Transformers (MiniLM)
- **Vous avez besoin de la meilleure qualité possible** → OpenAI text-embedding-3-large
- **Vous avez un domaine très spécialisé (médical, juridique)** → Fine-tuning BERT/RoBERTa
- **Vous traitez 50+ langues** → paraphrase-multilingual ou OpenAI (99 langues)
- **Latence critique <10ms** → TF-IDF + cache Redis, ou MiniLM + quantization INT8

Vectorisation d'images et données visuelles

Descripteurs classiques (HOG, SIFT)

Avant le deep learning, la vectorisation d'images reposait sur des **descripteurs manuels** (hand-crafted features) conçus par experts en vision par ordinateur.

HOG (Histogram of Oriented Gradients)
HOG (2005) calcule des histogrammes de directions de gradients dans des cellules locales de l'image. Utilisé historiquement pour détection de piétons, visages.

HOG avec scikit-image

```
from skimage.feature import hog
from skimage import io
import numpy as np

# Charger image
image = io.imread('cat.jpg', as_gray=True)

# Extraction HOG
features, hog_image = hog(
    image,
    orientations=9,
    pixels_per_cell=(8, 8),
    cells_per_block=(2, 2),
    visualize=True
)

print(features.shape)  # (7524,) - vecteur sparse de features
```

SIFT (Scale-Invariant Feature Transform)
SIFT (2004) détecte des points d'intérêt invariants à l'échelle et la rotation, puis extrait descripteurs 128D par point. Excellent pour matching d'images.

ORB (Oriented FAST and Rotated BRIEF)
ORB (2011) est une alternative open-source rapide à SIFT. Utilisé en robotique mobile et AR pour tracking.

Limites des Descripteurs Classiques

- **Features bas-niveau** : Capturent contours, textures mais pas sémantique ("chat" vs "chien")
- **Engineering intensif** : Tuning manuel des hyperparams (cell size, orientations, etc.)
- **Performance limitée** : 60-70% accuracy sur benchmarks ImageNet (vs 90%+ pour CNN)
- **Obsolètes en 2025** : Remplacés par CNN sauf contraintes extrêmes (edge devices ultra-low-power)

CNN et extraction de features

Les **Convolutional Neural Networks** (CNN) ont change la vision par ordinateur à partir de 2012 (AlexNet). Principe : utiliser un CNN pré-entraîné sur ImageNet (14M images, 1000 classes) comme **extracteur de features**.

Architectures CNN Classiques

- **ResNet-50/101** : 2048D, 25M/45M params, baseline standard
- **EfficientNet-B0 à B7** : 1280-2560D, 5M-66M params, meilleur ratio précision/vitesse
- **MobileNetV3** : 1024D, 5M params, optimisé mobile/edge

ResNet Feature Extraction avec PyTorch

```
import torch
import torchvision.models as models
import torchvision.transforms as transforms
from PIL import Image

# Charger ResNet50 pré-entraîné
model = models.resnet50(pretrained=True)
model.eval()

# Retirer la dernière couche (classifier) pour garder features
feature_extractor = torch.nn.Sequential(*list(model.children())[:-1])

# Preprocessing (requis pour ImageNet)
preprocess = transforms.Compose([
    transforms.Resize(256),
    transforms.CenterCrop(224),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

# Charger et préprocesser image
img = Image.open('cat.jpg')
input_tensor = preprocess(img).unsqueeze(0)  # (1, 3, 224, 224)

# Extraction features
with torch.no_grad():
    features = feature_extractor(input_tensor)

features = features.squeeze()  # (2048,)
print(features.shape)  # torch.Size([2048])
```

Transfer Learning
Pour un domaine spécifique (médical, satellite, etc.), on peut **fine-tuner** le CNN pré-entraîné sur un dataset métier. Deux stratégies :

- **Feature extraction** : Geler les couches convolutionnelles, entraîner seulement classifier final (500-1000 images suffisent)
- **Fine-tuning** : Dégeler les dernières couches conv et fine-tuner avec learning rate faible (5000-10000 images recommandées)

Bonnes Pratiques CNN 2025

- **Baseline** : Toujours commencer avec ResNet-50 ou EfficientNet-B0 pré-entraîné
- **Augmentation** : Appliquer data augmentation (rotation, flip, crop) pour robustesse
- **Normalisation** : Utiliser stats ImageNet (mean/std) même pour domaines spécialisés
- **Batch processing** : Vectoriser par batches de 32-128 images pour optimiser GPU
- **ONNX export** : Exporter en ONNX pour déploiement optimisé production

Vision Transformers

Les **Vision Transformers** (ViT, Google 2020) appliquent l'architecture Transformer (origine NLP) à la vision. Principe : découper l'image en patches 16x16, les traiter comme des "tokens", appliquer self-attention.

Avantages ViT vs CNN

- **Réceptive field global** : Self-attention capture dépendances longue portée dès la première couche
- **Scalabilité** : Performance s'améliore linéairement avec taille dataset (JFT-300M : 300M images)
- **Transfert inter-domaine** : Généralisation supérieure sur domaines éloignés d'ImageNet

Vision Transformer avec Hugging Face

```
from transformers import ViTImageProcessor, ViTModel
from PIL import Image
import torch

# Charger modèle ViT pré-entraîné
processor = ViTImageProcessor.from_pretrained('google/vit-base-patch16-224')
model = ViTModel.from_pretrained('google/vit-base-patch16-224')

# Charger image
image = Image.open('cat.jpg')

# Preprocessing
inputs = processor(images=image, return_tensors="pt")

# Extraction features
with torch.no_grad():
    outputs = model(**inputs)

# [CLS] token embedding (représentation image entière)
image_embedding = outputs.last_hidden_state[:, 0, :]
print(image_embedding.shape)  # (1, 768)

# Alternative : pooler output
pooled = outputs.pooler_output
print(pooled.shape)  # (1, 768)
```

Variantes Modernes

- **DeiT** (Facebook, 2021) : ViT entraîné sans dataset massif (distillation)
- **Swin Transformer** (Microsoft, 2021) : Hierarchical attention, meilleure efficacité
- **BEiT** (Microsoft, 2021) : Pre-training style BERT avec masked image modeling

Quand utiliser ViT ? Si vous avez accès à de gros datasets (>100K images) et GPU puissants, ViT surpasse CNN. Pour petits datasets (<10K), ResNet reste meilleur.

Modèles multi-modaux (CLIP)

CLIP (Contrastive Language-Image Pre-training, OpenAI 2021) est une révolution : il apprend simultanément des représentations texte ET image dans le **même espace vectoriel**. Résultat : similarité cosinus directe entre textes et images.

Principe CLIP

CLIP est entraîné sur 400M paires (image, texte descriptif) scrappées du web. Objectif : maximiser similarité entre représentations de paires correspondantes, minimiser pour paires non-correspondantes.

CLIP - Recherche Image par Texte

```
import torch
import clip
from PIL import Image

# Charger modèle CLIP
device = "cuda" if torch.cuda.is_available() else "cpu"
model, preprocess = clip.load("ViT-B/32", device=device)

# Images candidates
images = [
    preprocess(Image.open("cat.jpg")).unsqueeze(0).to(device),
    preprocess(Image.open("dog.jpg")).unsqueeze(0).to(device),
    preprocess(Image.open("car.jpg")).unsqueeze(0).to(device)
]
images = torch.cat(images)

# Queries texte
texts = clip.tokenize(["a photo of a cat", "a photo of a dog"]).to(device)

# Encodage
with torch.no_grad():
    image_features = model.encode_image(images)  # (3, 512)
    text_features = model.encode_text(texts)      # (2, 512)

    # Normalisation L2
    image_features /= image_features.norm(dim=-1, keepdim=True)
    text_features /= text_features.norm(dim=-1, keepdim=True)

    # Similarité cosinus
    similarity = (100.0 * text_features @ image_features.T).softmax(dim=-1)

print(similarity)
# tensor([[0.95, 0.03, 0.02],  # "a photo of a cat" -> cat.jpg (95%)
#         [0.05, 0.92, 0.03]]) # "a photo of a dog" -> dog.jpg (92%)
```

Applications CLIP

- **Recherche image par texte** : "chien noir dans un parc" → retrouver images pertinentes
- **Zero-shot classification** : Classifier images sans entraînement spécifique ("photo of X")
- **Recommandation cross-modal** : Recommander produits visuels depuis descriptions texte
- **Modération contenu** : Détecter images NSFW via queries texte

Cas d'Usage Réel : E-commerce

Un site e-commerce mode utilise CLIP pour :

- **Recherche visuelle** : Upload photo street style → trouver produits similaires en catalogue
- **Recherche texte-image** : "robe rouge longue été" → filtrer 100K images produits en <100ms
- **Recommandation** : Produits visuellement similaires dans espace CLIP 512D
- **Résultats** : +35% engagement, +18% conversion vs recherche keyword classique

Alternatives et Évolutions

- **OpenCLIP** : Ré-implémentation open-source avec modèles plus récents (LAION-5B dataset)
- **BLIP/BLIP-2** (Salesforce) : Améliore CLIP avec captioning et VQA

- **ImageBind** (Meta, 2023) : Espace vectoriel unifiant 6 modalités (image, texte, audio, vidéo, profondeur, IMU)

Vectorisation de l'audio

Spectrogrammes et MFCC

Les signaux audio sont des séries temporelles 1D (waveform). Pour les vectoriser, on les transforme en représentations fréquentielles 2D (spectrogrammes) puis extrait features.

Spectrogramme
Le **spectrogramme** est une représentation temps-fréquence obtenue par STFT (Short-Time Fourier Transform). Il montre quelles fréquences sont présentes à chaque instant.

Mel-Spectrogram
Le **Mel-Spectrogram** applique une échelle mel (logarithmique, mimée de l'oreille humaine) sur les fréquences du spectrogramme. Plus efficace pour parole et musique.

MFCC (Mel-Frequency Cepstral Coefficients)
Les **MFCC** sont les features audio les plus utilisés historiquement. Process : Mel-spectrogram → log → DCT (Discrete Cosine Transform) → 12-40 coefficients par frame.

Extraction MFCC avec librosa

```
import librosa
import numpy as np

# Charger audio
audio_path = 'speech.wav'
y, sr = librosa.load(audio_path, sr=22050)  # y=waveform, sr=sample rate

# Extraction MFCC
mfccs = librosa.feature.mfcc(y=y, sr=sr, n_mfcc=40)
print(mfccs.shape)  # (40, T) - 40 coefficients, T frames temporels

# Statistiques pour vectorisation fixe
mfcc_mean = np.mean(mfccs, axis=1)  # (40,)
mfcc_std = np.std(mfccs, axis=1)    # (40,)
vector = np.concatenate([mfcc_mean, mfcc_std])  # (80,)
print(vector.shape)  # (80,) - vecteur final fixe
```

Mel-Spectrogram pour Deep Learning
Mel-Spectrogram + CNN

```
import librosa
import librosa.display
import matplotlib.pyplot as plt

# Extraction Mel-Spectrogram
mel_spec = librosa.feature.melspectrogram(y=y, sr=sr, n_mels=128, fmax=8000)
mel_spec_db = librosa.power_to_db(mel_spec, ref=np.max)

print(mel_spec_db.shape)  # (128, T) - 128 bandes mel, T frames

# Utiliser comme "image" pour CNN
# Shape: (128, T) → (1, 128, T) pour PyTorch (channels, height, width)
import torch
mel_tensor = torch.from_numpy(mel_spec_db).unsqueeze(0)
print(mel_tensor.shape)  # (1, 128, T)
```

Challenges Audio

- **Durée variable** : Clips de 1s à 10min → padding/truncation ou pooling temporel
- **Bruit** : Environnements réels bruités → data augmentation (ajout bruit, pitch shift)
- **Domaine-specific** : MFCC optimisés pour parole, pas musique/sons environnementaux

Modèles pré-entraînés pour l'audio

Comme pour texte et images, les modèles pré-entraînés ont bouleverse l'audio. Ils apprennent des représentations riches sur des millions d'heures de données.

Wav2Vec 2.0 (Facebook/Meta, 2020)
Wav2Vec 2.0 apprend des représentations audio par auto-supervision (contrastive learning). Entraîné sur 53K heures de parole non-étiquetée (Libri-Light).

Wav2Vec 2.0 Feature Extraction

```
from transformers import Wav2Vec2Processor, Wav2Vec2Model
import torch
import librosa

# Charger modèle
processor = Wav2Vec2Processor.from_pretrained("facebook/wav2vec2-base")
model = Wav2Vec2Model.from_pretrained("facebook/wav2vec2-base")

# Charger audio
audio, sr = librosa.load('speech.wav', sr=16000)  # Wav2Vec2 requiert 16kHz

# Preprocessing
inputs = processor(audio, sampling_rate=sr, return_tensors="pt", padding=True)

# Extraction features
with torch.no_grad():
    outputs = model(**inputs)

# Embeddings (moyenne sur temps)
embeddings = outputs.last_hidden_state.mean(dim=1)  # (1, 768)
print(embeddings.shape)  # (1, 768)
```

HuBERT (Facebook, 2021)
HuBERT (Hidden-Unit BERT) améliore Wav2Vec2 avec clustering itératif. Meilleure qualité pour reconnaissance de parole.

Data2Vec (Meta, 2022)
Framework unifié pour audio, vision et NLP avec même algorithme d'apprentissage. Représentations audio comparables à Wav2Vec2.

Modèle	Année	Dimensions	Cas d'usage
MFCC	1980s	40-80D	Baseline, systèmes embarqués, temps réel
Wav2Vec 2.0	2020	768D	Reconnaissance parole, classification audio
HuBERT	2021	768D	ASR (Automatic Speech Recognition), NLU audio
Whisper (embeddings)	2022	512-1280D	Transcription multilingue, embeddings audio robustes

Speech-to-text et embeddings audio

Whisper (OpenAI, 2022) est un modèle de transcription audio multilingue (99 langues) entraîné sur 680K heures. En plus de transcription, son encoder produit d'excellents embeddings audio.

Whisper Embeddings

```
import whisper
import torch
import librosa

# Charger modèle Whisper
model = whisper.load_model("base")  # tiny/base/small/medium/large

# Charger audio
audio, sr = librosa.load('speech.wav', sr=16000)

# Padding/Truncation à 30s (Whisper limite)
audio = whisper.pad_or_trim(audio)

# Mel-spectrogram (fait par Whisper)
mel = whisper.log_mel_spectrogram(audio).to(model.device)

# Extraction embeddings depuis encoder
with torch.no_grad():
    embeddings = model.embed_audio(mel.unsqueeze(0))

print(embeddings.shape)  # (1, 1500, 512) - séquence de frames

# Pooling pour vecteur fixe
audio_vector = embeddings.mean(dim=1)  # (1, 512)
print(audio_vector.shape)
```

Applications Pratiques

1. Recherche Audio Sémantique

Vectoriser bibliothèque audio (podcasts, conférences) avec Whisper, puis recherche sémantique par similarité cosinus. Pour approfondir, consultez Mémoire Augmentée Agents : Vector + Graph 2026.

2. Classification Audio

Classifier émotions avec Wav2Vec2

```
from transformers import pipeline

# Pipeline pré-entraîné pour classification audio
classifier = pipeline(
    "audio-classification",
    model="superb/wav2vec2-base-superb-er"  # Emotion Recognition
)

# Classification
result = classifier('speech.wav')
print(result)
# [{'label': 'happy', 'score': 0.87},
#  {'label': 'neutral', 'score': 0.09}, ...]
```

3. Détection d'Anomalies Sonores

En industrie : vectoriser sons machines normales, entraîner autoencoder, détecter anomalies par reconstruction error.

Cas d'Usage Réel : Call Center

Un call center utilise Whisper embeddings pour :

- **Recherche appels similaires** : Problème client nouveau → trouver 10 appels similaires en base vectorielle
- **Classification automatique** : Routage vers bon service (facturation, SAV, réclamation)
- **Détection émotions** : Alertes temps réel si client devient agressif/frustré
- **Résultats** : -35% temps de traitement, +22% satisfaction client

Comparaison Approches Audio 2025

Guide de Choix Rapide

- **Baseline rapide, edge devices** → MFCC + SVM/Random Forest
- **Classification audio générale** → Wav2Vec2 + fine-tuning
- **Reconnaissance parole multilingue** → Whisper encoder
- **Recherche sémantique audio longue durée** → Whisper embeddings + Qdrant
- **Musique (genre, mood)** → VGGish ou modèles spécialisés (MusicGen embeddings)

Implémentation pratique en Python

Environnement et bibliothèques nécessaires

Pour mettre en place un environnement de vectorisation production-ready, voici l'écosystème Python essentiel :

Bibliothèques Fondamentales

Installation environnement complet

```
# Créer environnement virtuel
python -m venv venv
source venv/bin/activate  # Linux/Mac
# ou : venv\Scripts\activate  # Windows

# Packages essentiels
pip install numpy pandas scikit-learn

# NLP / Texte
pip install sentence-transformers transformers torch
pip install openai  # Pour OpenAI embeddings API

# Vision / Images
pip install torchvision pillow opencv-python
pip install timm  # PyTorch Image Models (EfficientNet, etc.)

# Audio
pip install librosa soundfile
pip install git+https://github.com/openai/whisper.git

# Bases vectorielles
pip install qdrant-client chromadb pinecone-client

# Optimisation et monitoring
pip install onnx onnxruntime tqdm
```

Configuration GPU (optionnelle mais recommandée)

Vérifier disponibilité GPU

```
import torch

print(f"CUDA available: {torch.cuda.is_available()}")
if torch.cuda.is_available():
    print(f"GPU: {torch.cuda.get_device_name(0)}")
    print(f"Memory: {torch.cuda.get_device_properties(0).total_memory / 1e9:.1f} GB")

# Pour utiliser GPU avec Sentence Transformers
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2', device='cuda')  # ou 'cpu'
```

Structure Projet Recommandée

```
vectorization-project/
├── data/
│   ├── raw/              # Données brutes
│   ├── processed/        # Données nettoyées
│   └── vectors/          # Vecteurs générés
├── models/
│   ├── sentence-transformers/
│   └── custom/           # Modèles fine-tunés
├── src/
│   ├── vectorizers/
│   │   ├── text_vectorizer.py
│   │   ├── image_vectorizer.py
│   │   └── audio_vectorizer.py
│   ├── preprocessing.py
│   ├── storage.py        # Interface base vectorielle
│   └── utils.py
├── config.yaml           # Configuration
├── requirements.txt
└── main.py
```

Configuration YAML Exemple

config.yaml

```yaml
# config.yaml
text:
  model_name: "all-MiniLM-L6-v2"
  batch_size: 32
  max_length: 512
  normalize: true

image:
  model_name: "resnet50"
  image_size: 224
  batch_size: 16

audio:
  model_name: "whisper-base"
  sample_rate: 16000

vector_db:
  provider: "qdrant"  # ou "pinecone", "chromadb"
  host: "localhost"
  port: 6333
  collection_name: "documents"
```

Exemple 1 : Vectorisation de texte avec Sentence Transformers

Voici un exemple complet de vectorisation de documents texte avec stockage dans Qdrant.

Considerations avancees

text_vectorizer.py - Pipeline Complet

```
from sentence_transformers import SentenceTransformer
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, PointStruct
import uuid
from typing import List, Dict
from tqdm import tqdm

class TextVectorizer:
    def __init__(self, model_name='all-MiniLM-L6-v2', batch_size=32):
        self.model = SentenceTransformer(model_name)
        self.batch_size = batch_size
        self.vector_size = self.model.get_sentence_embedding_dimension()

    def preprocess(self, text: str) -> str:
        """Nettoyage texte basique"""
        text = text.strip()
        text = ' '.join(text.split())  # Normaliser espaces
        return text

    def vectorize(self, texts: List[str]) -> List[List[float]]:
        """Vectorisation batch avec barre de progression"""
        cleaned_texts = [self.preprocess(t) for t in texts]
        vectors = self.model.encode(
            cleaned_texts,
            batch_size=self.batch_size,
            normalize_embeddings=True,
            show_progress_bar=True
        )
        return vectors.tolist()

    def vectorize_and_store(self, documents: List[Dict], collection_name: str):
        """
        Vectorise documents et stocke dans Qdrant

        Args:
            documents: Liste de dicts avec keys 'text' et 'metadata'
            collection_name: Nom collection Qdrant
        """
        # Connexion Qdrant
        client = QdrantClient(host="localhost", port=6333)

        # Création collection
        client.recreate_collection(
            collection_name=collection_name,
            vectors_config=VectorParams(
                size=self.vector_size,
                distance=Distance.COSINE
            )
        )

        # Vectorisation
        texts = [doc['text'] for doc in documents]
        vectors = self.vectorize(texts)

        # Upload vers Qdrant
        points = [
            PointStruct(
                id=str(uuid.uuid4()),
                vector=vector,
                payload={
                    'text': doc['text'],
                    **doc.get('metadata', {})
```

```
                }
            )
            for doc, vector in zip(documents, vectors)
        ]

        # Upload par batches
        for i in tqdm(range(0, len(points), 100), desc="Uploading to Qdrant"):
            batch = points[i:i+100]
            client.upsert(
                collection_name=collection_name,
                points=batch
            )

        print(f"Vectorized and stored {len(documents)} documents")
        return client

# Utilisation
if __name__ == "__main__":
    # Documents exemple
    documents = [
        {
            "text": "Paris est la capitale de France",
            "metadata": {"category": "geography", "lang": "fr"}
        },
        {
            "text": "La Tour Eiffel mesure 330 mètres",
            "metadata": {"category": "monument", "lang": "fr"}
        },
        {
            "text": "Python est un langage de programmation",
            "metadata": {"category": "tech", "lang": "fr"}
        }
    ]

    # Vectorisation
    vectorizer = TextVectorizer()
    client = vectorizer.vectorize_and_store(documents, "my_docs")

    # Recherche sémantique
    query = "Quelle est la hauteur de la tour Eiffel ?"
    query_vector = vectorizer.vectorize([query])[0]

    results = client.search(
        collection_name="my_docs",
        query_vector=query_vector,
        limit=3
    )

    print("\nRésultats recherche:")
    for result in results:
        print(f"Score: {result.score:.3f} - {result.payload['text']}")
```

Résultat attendu : Le document "La Tour Eiffel mesure 330 mètres" aura le score le plus élevé (0.85-0.95) car sémantiquement le plus proche de la query.

Exemple 2 : Vectorisation d'images avec ResNet

Pipeline complet de vectorisation d'images avec ResNet-50 pré-entraîné sur ImageNet.

image_vectorizer.py

production_pipeline.py

```
from dataclasses import dataclass
from typing import List, Dict, Optional, Union
import logging
from pathlib import Path
import time
import numpy as np
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, PointStruct
import uuid

# Import vectorizers
from text_vectorizer import TextVectorizer
from image_vectorizer import ImageVectorizer

# Configuration logging
logging.basicConfig(level=logging.INFO)
logger = logging.getLogger(__name__)

@dataclass
class Document:
    """Représentation unifiée d'un document"""
    id: str
    text: Optional[str] = None
    image_path: Optional[str] = None
    metadata: Dict = None

class ProductionPipeline:
    def __init__(self, config: Dict):
        self.config = config

        # Initialisation vectorizers
        self.text_vectorizer = TextVectorizer(
            model_name=config['text']['model_name'],
            batch_size=config['text']['batch_size']
        )

        self.image_vectorizer = ImageVectorizer(
            model_name=config['image']['model_name']
        )

        # Qdrant client
        self.db_client = QdrantClient(
            host=config['vector_db']['host'],
            port=config['vector_db']['port']
        )

        # Stats
        self.stats = {
            'processed': 0,
            'errors': 0,
            'total_time': 0
        }

    def setup_collections(self):
        """Crée collections Qdrant"""
        # Collection texte
        self.db_client.recreate_collection(
            collection_name="text_vectors",
            vectors_config=VectorParams(
                size=self.text_vectorizer.vector_size,
                distance=Distance.COSINE
            )
```

```
        )

        # Collection images
        self.db_client.recreate_collection(
            collection_name="image_vectors",
            vectors_config=VectorParams(
                size=self.image_vectorizer.vector_size,
                distance=Distance.COSINE
            )
        )

        logger.info("Collections created successfully")

    def process_document(self, doc: Document, retry=3) -> bool:
        """Traite un document avec retry logic"""
        for attempt in range(retry):
            try:
                start_time = time.time()

                # Vectorisation texte
                if doc.text:
                    text_vector = self.text_vectorizer.vectorize([doc.text])[0]
                    self.db_client.upsert(
                        collection_name="text_vectors",
                        points=[PointStruct(
                            id=doc.id,
                            vector=text_vector,
                            payload={'text': doc.text, **(doc.metadata or {})}
                        )]
                    )

                # Vectorisation image
                if doc.image_path and Path(doc.image_path).exists():
                    image_vector = self.image_vectorizer.vectorize_single(doc.image_path)
                    self.db_client.upsert(
                        collection_name="image_vectors",
                        points=[PointStruct(
                            id=doc.id,
                            vector=image_vector.tolist(),
                            payload={'image_path': doc.image_path, **(doc.metadata or {})}
                        )]
                    )

                # Stats
                elapsed = time.time() - start_time
                self.stats['processed'] += 1
                self.stats['total_time'] += elapsed

                logger.info(f"Processed doc {doc.id} in {elapsed:.2f}s")
                return True

            except Exception as e:
                logger.error(f"Attempt {attempt+1} failed for doc {doc.id}: {e}")
                if attempt == retry - 1:
                    self.stats['errors'] += 1
                    return False
                time.sleep(2 ** attempt)  # Exponential backoff

        return False

    def process_batch(self, documents: List[Document], batch_size=100):
        """Traite un batch de documents"""
```

```
        logger.info(f"Processing {len(documents)} documents...")

        for i in range(0, len(documents), batch_size):
            batch = documents[i:i+batch_size]

            for doc in batch:
                self.process_document(doc)

            # Log progress
            progress = min(i + batch_size, len(documents))
            logger.info(f"Progress: {progress}/{len(documents)} ({100*progress/
len(documents):.1f}%)")

        # Summary
        avg_time = self.stats['total_time'] / max(self.stats['processed'], 1)
        logger.info(f"\n=== Pipeline Summary ===")
        logger.info(f"Processed: {self.stats['processed']}")
        logger.info(f"Errors: {self.stats['errors']}")
        logger.info(f"Avg time per doc: {avg_time:.3f}s")
        logger.info(f"Throughput: {self.stats['processed']/self.stats['total_time']:.1f}
docs/s")

    def search(self, query_text: Optional[str] = None,
               query_image: Optional[str] = None,
               limit=10) -> Dict:
        """Recherche unifiée texte/image"""
        results = {}

        # Recherche texte
        if query_text:
            query_vector = self.text_vectorizer.vectorize([query_text])[0]
            text_results = self.db_client.search(
                collection_name="text_vectors",
                query_vector=query_vector,
                limit=limit
            )
            results['text'] = text_results

        # Recherche image
        if query_image:
            query_vector = self.image_vectorizer.vectorize_single(query_image)
            image_results = self.db_client.search(
                collection_name="image_vectors",
                query_vector=query_vector.tolist(),
                limit=limit
            )
            results['image'] = image_results

        return results

# Utilisation
if __name__ == "__main__":
    config = {
        'text': {'model_name': 'all-MiniLM-L6-v2', 'batch_size': 32},
        'image': {'model_name': 'resnet50'},
        'vector_db': {'host': 'localhost', 'port': 6333}
    }

    pipeline = ProductionPipeline(config)
    pipeline.setup_collections()

    # Documents exemple
```

```
    documents = [
        Document(
            id=str(uuid.uuid4()),
            text="Paris est magnifique au printemps",
            image_path="./images/paris.jpg",
            metadata={'category': 'travel', 'city': 'Paris'}
        ),
        Document(
            id=str(uuid.uuid4()),
            text="La Tour Eiffel illuminée la nuit",
            image_path="./images/eiffel_night.jpg",
            metadata={'category': 'monument', 'city': 'Paris'}
        )
    ]

    # Traitement
    pipeline.process_batch(documents)

    # Recherche
    results = pipeline.search(query_text="monuments parisiens")
    print("\nRésultats:", results)
```

Ce pipeline production-ready inclut : retry logic, logging, monitoring de performance, gestion d'erreurs, et supporte multi-modalités.

Optimisation et performance

Batch processing et parallélisation

Le batch processing est **essentiel** pour optimiser la vectorisation de gros volumes. Principe : traiter plusieurs documents simultanément plutôt qu'un par un.

Impact Performance Batch Size

Batch Size	Temps pour 10K docs (GPU)	Utilisation GPU	Mémoire VRAM
1 (sans batch)	~600s	5-15%	500 MB
16	~60s	40-60%	2 GB
32 (recommandé)	~35s	70-85%	4 GB
64	~25s	85-95%	8 GB

Batch Processing Optimisé

```
from sentence_transformers import SentenceTransformer
import torch
from tqdm import tqdm

model = SentenceTransformer('all-MiniLM-L6-v2')
model.to('cuda' if torch.cuda.is_available() else 'cpu')

# 10K documents
documents = [f"Document {i}" for i in range(10000)]

# Méthode 1 : Batch automatique (recommandé)
vectors = model.encode(
    documents,
    batch_size=32,  # Adapter selon VRAM disponible
    show_progress_bar=True,
    normalize_embeddings=True
)

# Méthode 2 : Batch manuel avec contrôle fin
def manual_batch_encode(texts, model, batch_size=32):
    all_vectors = []
    for i in tqdm(range(0, len(texts), batch_size)):
        batch = texts[i:i+batch_size]
        batch_vectors = model.encode(batch, convert_to_numpy=True)
        all_vectors.append(batch_vectors)
    return np.vstack(all_vectors)

vectors = manual_batch_encode(documents, model, batch_size=32)
```

Parallélisation Multi-GPU

Multi-GPU avec DataParallel

```
import torch
from sentence_transformers import SentenceTransformer

model = SentenceTransformer('all-mpnet-base-v2')

# Vérifier nb GPUs
if torch.cuda.device_count() > 1:
    print(f"Using {torch.cuda.device_count()} GPUs")

    # Activer multi-GPU
    model = model.to('cuda')
    model._first_module().auto_model = torch.nn.DataParallel(
        model._first_module().auto_model
    )

# Encode (distribué automatiquement sur GPUs)
vectors = model.encode(documents, batch_size=64)
```

Parallélisation CPU (Multiprocessing)

Pool de workers CPU

```
from multiprocessing import Pool, cpu_count
from sentence_transformers import SentenceTransformer
import numpy as np

def vectorize_chunk(args):
    """Fonction worker pour pool"""
    texts, model_name = args
    model = SentenceTransformer(model_name)
    return model.encode(texts)

def parallel_vectorize(documents, model_name='all-MiniLM-L6-v2', n_workers=None):
    if n_workers is None:
        n_workers = cpu_count() - 1

    # Diviser en chunks
    chunk_size = len(documents) // n_workers
    chunks = [documents[i:i+chunk_size] for i in range(0, len(documents), chunk_size)]

    # Pool de workers
    with Pool(n_workers) as pool:
        args = [(chunk, model_name) for chunk in chunks]
        results = pool.map(vectorize_chunk, args)

    return np.vstack(results)

# Utilisation
vectors = parallel_vectorize(documents, n_workers=8)
```

Gestion de la mémoire pour de gros volumes

Vectoriser des millions de documents nécessite une gestion mémoire rigoureuse pour éviter les OOM (Out Of Memory).

Streaming Processing
Plutôt que charger tous les documents en mémoire, traiter par mini-batches en streaming.

Streaming depuis fichier

```
from sentence_transformers import SentenceTransformer
import json
from qdrant_client import QdrantClient
from qdrant_client.models import PointStruct
import uuid

model = SentenceTransformer('all-MiniLM-L6-v2')
client = QdrantClient(host='localhost', port=6333)

def stream_from_jsonl(file_path, batch_size=100):
    """Générateur streaming depuis JSONL"""
    batch = []
    with open(file_path, 'r') as f:
        for line in f:
            doc = json.loads(line)
            batch.append(doc)

            if len(batch) >= batch_size:
                yield batch
                batch = []

    if batch:  # Dernier batch
        yield batch

# Traitement streaming
for batch in stream_from_jsonl('large_dataset.jsonl', batch_size=100):
    # Vectorisation
    texts = [doc['text'] for doc in batch]
    vectors = model.encode(texts)

    # Upload immédiat vers Qdrant
    points = [
        PointStruct(
            id=str(uuid.uuid4()),
            vector=vec.tolist(),
            payload={'text': doc['text']}
        )
        for doc, vec in zip(batch, vectors)
    ]
    client.upsert(collection_name='docs', points=points)

    # Batch traité puis libéré de mémoire
    del batch, vectors, points
```

Garbage Collection Agressif

Libération mémoire explicite

```
import gc
import torch

for i, batch in enumerate(batches):
    vectors = model.encode(batch)
    # ... traitement ...

    # Libération mémoire
    del vectors, batch

    # Garbage collection toutes les 10 itérations
    if i % 10 == 0:
        gc.collect()
        if torch.cuda.is_available():
            torch.cuda.empty_cache()
```

Stockage Temporaire sur Disque

Pour datasets très volumineux, sauvegarder vecteurs sur disque en chunks puis upload en deuxième passe.

Chunked Storage avec HDF5

```
import h5py
import numpy as np

# Phase 1 : Vectorisation vers HDF5
with h5py.File('vectors.h5', 'w') as f:
    # Pré-allocation dataset
    dset = f.create_dataset('vectors', shape=(1000000, 384), dtype='float32')

    start_idx = 0
    for batch in stream_documents(batch_size=1000):
        vectors = model.encode(batch)
        end_idx = start_idx + len(vectors)
        dset[start_idx:end_idx] = vectors
        start_idx = end_idx

# Phase 2 : Upload vers base vectorielle
with h5py.File('vectors.h5', 'r') as f:
    vectors = f['vectors']
    for i in range(0, len(vectors), 1000):
        batch = vectors[i:i+1000]
        # Upload vers Qdrant/Pinecone
        client.upsert(...)
```

Checklist Mémoire

- **Monitorer RAM/VRAM** : Utiliser `nvidia-smi` (GPU) et `htop` (CPU)
- **Adapter batch size** : Réduire si OOM, augmenter si GPU sous-utilisé
- **Mixed precision** : Utiliser float16 au lieu de float32 (divise mémoire par 2)
- **Gradient checkpointing** : Si fine-tuning, activer pour économiser mémoire

Accélération GPU

Les GPUs accélèrent la vectorisation de **10-50x** par rapport au CPU. Voici les optimisations clés.

Mixed Precision (FP16)
Utiliser float16 au lieu de float32 : 2x plus rapide, 2x moins de mémoire, précision quasi identique.

Activation FP16

```
from sentence_transformers import SentenceTransformer
import torch

model = SentenceTransformer('all-MiniLM-L6-v2')
model.to('cuda')

# Activation mixed precision
model.half()  # Convertit modèle en FP16

vectors = model.encode(documents, batch_size=64)  # 2x plus rapide

# Alternative : Automatic Mixed Precision (AMP)
from torch.cuda.amp import autocast

with autocast():
    vectors = model.encode(documents)
```

Optimisation avec ONNX Runtime
ONNX Runtime optimise l'inférence avec quantization, fusion d'opérateurs, etc. Gain : 2-4x sur CPU, 1.5-2x sur GPU.

Export et Inférence ONNX

```
from optimum.onnxruntime import ORTModelForFeatureExtraction
from transformers import AutoTokenizer

# Export vers ONNX (une seule fois)
model_name = "sentence-transformers/all-MiniLM-L6-v2"
model = ORTModelForFeatureExtraction.from_pretrained(
    model_name,
    export=True,
    provider="CUDAExecutionProvider"  # ou "CPUExecutionProvider"
)
tokenizer = AutoTokenizer.from_pretrained(model_name)

# Inférence (1.5-2x plus rapide)
inputs = tokenizer(documents, padding=True, truncation=True, return_tensors="pt")
with torch.no_grad():
    outputs = model(**inputs)

vectors = outputs.last_hidden_state.mean(dim=1)
```

Quantization INT8
Réduit précision de float32 à int8 : 4x réduction mémoire, 2-3x plus rapide sur CPU, légère perte précision (1-3%).

Quantization avec Optimum

```
from optimum.onnxruntime import ORTQuantizer
from optimum.onnxruntime.configuration import AutoQuantizationConfig

# Quantization INT8
quantizer = ORTQuantizer.from_pretrained("sentence-transformers/all-MiniLM-L6-v2")
qconfig = AutoQuantizationConfig.avx512_vnni(is_static=False, per_channel=False)
quantizer.quantize(save_dir="./quantized_model", quantization_config=qconfig)

# Chargement modèle quantizé
model = ORTModelForFeatureExtraction.from_pretrained("./quantized_model")
# 2-3x plus rapide sur CPU
```

Benchmarks GPU vs CPU

Configuration	10K docs (texte)	Throughput	Coût Cloud
CPU (16 cores, batch=1)	600s	16 docs/s	$0.50/h (AWS c5.4xlarge)
CPU (16 cores, batch=32)	180s	55 docs/s	$0.50/h
CPU (ONNX + INT8)	90s	110 docs/s	$0.50/h
GPU T4 (batch=64)	25s	400 docs/s	$0.52/h (AWS g4dn.xlarge)
GPU A100 (batch=128, FP16)	8s	1250 docs/s	$4.10/h (AWS p4d.24xlarge)

Recommandations Hardware

- **< 100K docs one-time** : CPU + ONNX suffit
- **100K-1M docs** : GPU T4/V100 (cloud ou local)
- **> 1M docs ou temps réel** : GPU A100 ou fleet de T4
- **Edge deployment** : CPU + ONNX + INT8 quantization

Caching et stockage des vecteurs

Vectoriser est coûteux (5-50ms/doc). Le caching permet d'éviter de re-vectoriser les mêmes données.

Stratégie de Caching Multi-Niveaux

1. **L1 - Mémoire (Redis)** : Cache chaud pour vecteurs fréquemment accédés (TTL 1h-24h)
2. **L2 - Base vectorielle** : Stockage persistant principal (Qdrant, Pinecone)
3. **L3 - Object Storage** : Archive long-terme (S3, MinIO) avec compression

Cache Redis + Qdrant

```
import redis
import hashlib
import json
import numpy as np
from sentence_transformers import SentenceTransformer
from qdrant_client import QdrantClient

class VectorCache:
    def __init__(self):
        self.redis_client = redis.Redis(host='localhost', port=6379, db=0)
        self.qdrant_client = QdrantClient(host='localhost', port=6333)
        self.model = SentenceTransformer('all-MiniLM-L6-v2')

    def _hash_text(self, text: str) -> str:
        """Hash texte pour clé cache"""
        return hashlib.md5(text.encode()).hexdigest()

    def get_vector(self, text: str) -> np.ndarray:
        """Récupère vecteur avec cache multi-niveaux"""
        cache_key = f"vec:{self._hash_text(text)}"

        # L1 : Redis
        cached = self.redis_client.get(cache_key)
        if cached:
            print("L1 Cache hit (Redis)")
            return np.frombuffer(cached, dtype=np.float32)

        # L2 : Qdrant
        results = self.qdrant_client.scroll(
            collection_name="vectors",
            scroll_filter={"must": [{"key": "text", "match": {"value": text}}]},
            limit=1
        )
        if results[0]:
            print("L2 Cache hit (Qdrant)")
            vector = np.array(results[0][0].vector)
            # Populate L1
            self.redis_client.setex(cache_key, 3600, vector.tobytes())
            return vector

        # L3 : Calcul + stockage
        print("Cache miss - Computing vector")
        vector = self.model.encode(text)

        # Store L1
        self.redis_client.setex(cache_key, 3600, vector.tobytes())

        # Store L2
        from qdrant_client.models import PointStruct
        import uuid
        self.qdrant_client.upsert(
            collection_name="vectors",
            points=[PointStruct(
                id=str(uuid.uuid4()),
                vector=vector.tolist(),
                payload={'text': text}
            )]
        )

        return vector

# Utilisation
```

```
cache = VectorCache()
vec1 = cache.get_vector("Paris est magnifique")  # Cache miss
vec2 = cache.get_vector("Paris est magnifique")  # L1 hit (Redis) - instantané
```

Invalidation de Cache

Stratégies d'invalidation selon cas d'usage :

- **TTL (Time To Live)** : Expiration automatique après durée fixe (1h-24h)
- **LRU (Least Recently Used)** : Éviction des entrées les moins utilisées quand cache plein
- **Manual invalidation** : Invalider explicitement si document modifié
- **Version-based** : Inclure version modèle dans clé cache (invalide si modèle change)

Cache avec versioning

```
class VersionedVectorCache:
    MODEL_VERSION = "all-MiniLM-L6-v2-v1"  # Changer si modèle change

    def _cache_key(self, text: str) -> str:
        hash_val = hashlib.md5(text.encode()).hexdigest()
        return f"vec:{self.MODEL_VERSION}:{hash_val}"

    # Si MODEL_VERSION change, ancien cache automatiquement invalidé
```

Compression pour Stockage Long-Terme

Stockage compressé S3

```
import boto3
import pickle
import gzip

def save_vectors_to_s3(vectors, metadata, bucket, key):
    """Sauvegarde vecteurs compressés sur S3"""
    s3 = boto3.client('s3')

    data = {'vectors': vectors, 'metadata': metadata}
    pickled = pickle.dumps(data)
    compressed = gzip.compress(pickled, compresslevel=9)

    s3.put_object(
        Bucket=bucket,
        Key=key,
        Body=compressed,
        ContentType='application/gzip'
    )
    print(f"Saved {len(vectors)} vectors (compression: {100*(1-len(compressed)/
len(pickled)):.1f}%)")

def load_vectors_from_s3(bucket, key):
    """Charge vecteurs depuis S3"""
    s3 = boto3.client('s3')
    response = s3.get_object(Bucket=bucket, Key=key)
    compressed = response['Body'].read()
    pickled = gzip.decompress(compressed)
    return pickle.loads(pickled)

# Utilisation
vectors = model.encode(documents)
save_vectors_to_s3(vectors, metadata, 'my-bucket', 'vectors/batch_001.pkl.gz')
```

Best Practices Caching

- **Toujours hasher** : Ne jamais utiliser texte brut comme clé (taille limite Redis 512MB)
- **Monitorer hit rate** : Viser 80%+ hit rate pour queries fréquentes
- **Separate caches** : Cache distinct par modèle/use case (evite collisions)
- **Warmup cache** : Pré-calculer vecteurs pour contenus populaires au démarrage

Cas d'usage réels et retours d'expérience

E-commerce : vectorisation de catalogues produits

Contexte : Site e-commerce mode avec 500K produits, recherche traditionnelle keyword-based peu performante.

Problématique

- Recherche "robe rouge été" ne trouve pas "robe écarlate estivale" (synonymes)
- Recommandations basées uniquement sur catégories (pas de similarité visuelle)
- Impossibilité de recherche par upload photo

Solution Implémentée
Architecture Multi-Modale

```
# Vectorisation produits (texte + image)
class ProductVectorizer:
    def __init__(self):
        # Texte : description produit
        self.text_model = SentenceTransformer('paraphrase-multilingual-mpnet-base-v2')
        # Image : photo produit
        self.image_model = CLIPModel.from_pretrained('openai/clip-vit-base-patch32')

    def vectorize_product(self, product):
        # Texte (titre + description)
        text = f"{product['title']} {product['description']}"
        text_vec = self.text_model.encode(text)  # (768,)

        # Image
        image = Image.open(product['image_path'])
        image_vec = self.image_model.encode_image(image)  # (512,)

        # Concaténation
        combined_vec = np.concatenate([text_vec, image_vec])  # (1280,)
        return combined_vec / np.linalg.norm(combined_vec)  # Normalisation
```

Résultats

Métrique	Avant (Keyword)	Après (Vectorielle)	Amélioration
Précision recherche	45%	78%	+73%
Taux clic (CTR)	2.1%	4.8%	+129%
Conversion	1.8%	2.9%	+61%
Panier moyen	67€	89€	+33%

Leçons Apprises

- **Multilingue essentiel** : 40% clients non-francophones, modèle multilingue indispensable
- **Images > Texte** : Vecteurs images ont poids 60% vs texte 40% dans scoring final (test A/B)
- **Mise à jour incrémentale** : Re-vectorisation quotidienne des nouveaux produits (batch nuit)
- **Monitoring crucial** : Dashboard qualité recherche (précision, latence, null results rate)

Support client : vectorisation de tickets

Contexte : Entreprise SaaS B2B avec 50K tickets support/mois, temps de résolution moyen 4h.

Problématique

- Agents perdent 30-40% du temps à chercher tickets similaires
- Base de connaissances (KB) mal exploitée (recherche keyword insuffisante)
- Routage tickets vers mauvaise équipe (catégorisation manuelle 15% erreur)

Solution Implémentée

Système RAG pour Support

```
class SupportTicketSystem:
    def __init__(self):
        self.vectorizer = SentenceTransformer('all-mpnet-base-v2')
        self.qdrant = QdrantClient('localhost', port=6333)

    def index_ticket(self, ticket):
        """Indexe nouveau ticket"""
        # Concaténation titre + description + résolution
        text = f"{ticket['title']}\n{ticket['description']}\n{ticket['resolution']}"
        vector = self.vectorizer.encode(text)

        self.qdrant.upsert(
            collection_name='tickets',
            points=[PointStruct(
                id=ticket['id'],
                vector=vector.tolist(),
                payload={
                    'title': ticket['title'],
                    'category': ticket['category'],
                    'resolution_time': ticket['resolution_time'],
                    'satisfaction_score': ticket['satisfaction_score']
                }
            )]
        )

    def find_similar_tickets(self, new_ticket_description, top_k=5):
        """Trouve tickets similaires résolus"""
        query_vec = self.vectorizer.encode(new_ticket_description)

        results = self.qdrant.search(
            collection_name='tickets',
            query_vector=query_vec.tolist(),
            limit=top_k,
            query_filter={  # Seulement tickets résolus avec bonne satisfaction
                "must": [
                    {"key": "status", "match": {"value": "resolved"}},
                    {"key": "satisfaction_score", "range": {"gte": 4}}
                ]
            }
        )

        return results

    def auto_suggest_response(self, ticket_description):
        """Suggère réponse basée sur tickets similaires"""
        similar = self.find_similar_tickets(ticket_description, top_k=3)

        # Agrégation réponses
        resolutions = [hit.payload['resolution'] for hit in similar]

        # Prompt LLM pour synthèse
        prompt = f"""Tickets similaires résolus:
{chr(10).join(resolutions)}

Nouveau ticket: {ticket_description}

Suggérer une réponse:"""

        # Appel GPT-4 pour générer réponse personnalisée
        response = openai.ChatCompletion.create(
            model="gpt-4",
            messages=[{"role": "user", "content": prompt}]
```

```
        )

        return response.choices[0].message.content
```

Résultats
Impact Mesuré (6 mois)

Mise en pratique

- **Temps de résolution** : 4h → 2.3h (-42%)
- **First Response Time** : 45min → 12min (-73%)
- **Auto-résolution** : 18% tickets résolus sans intervention humaine (suggestions acceptées)
- **Satisfaction client** : 3.8/5 → 4.5/5
- **Coûts** : Économie estimée $180K/an (réduction FTE support)

Pièges Évités
- **Tickets non résolus** : Filtrer sur status=resolved (sinon pollue résultats)
- **Tickets anciens** : Pondérer par date (solutions d'il y a 2 ans obsolètes)
- **Over-reliance IA** : Agents peuvent override suggestions (humain in the loop)
- **Privacy** : Anonymiser données sensibles avant vectorisation (RGPD)

Médias : vectorisation de contenus multimédia

Contexte : Chaîne TV nationale avec archives 200K heures vidéo, recherche contenu inefficace.

Problématique
- Recherche limitée aux métadonnées manuelles (titre, tags) incomplètes
- Impossible de rechercher "scène de manifestation à Paris" dans contenu vidéo
- Ré-utilisation archives faible (journalistes ne trouvent pas contenus pertinents)

Solution Implémentée
Pipeline multi-modal vectorisant chaque segment vidéo (30s) sur 3 modalités :

Vectorisation Vidéo Multi-Modale

```
import whisper
import clip
import torch
from scenedetect import VideoManager, SceneManager

class VideoVectorizer:
    def __init__(self):
        self.whisper = whisper.load_model('medium')  # Transcription audio
        self.clip = clip.load('ViT-B/32')  # Image embeddings

    def process_video(self, video_path):
        """Vectorise vidéo complète"""
        # 1. Détection de scènes
        scenes = self.detect_scenes(video_path)  # Retourne timestamps

        vectors = []
        for scene_start, scene_end in scenes:
            # 2. Extraction frame clé
            keyframe = self.extract_keyframe(video_path, scene_start)
            image_vec = self.vectorize_frame(keyframe)  # CLIP (512D)

            # 3. Transcription audio segment
            audio_segment = self.extract_audio(video_path, scene_start, scene_end)
            transcript = self.whisper.transcribe(audio_segment)['text']
            text_vec = self.vectorize_text(transcript)  # Sentence-BERT (768D)

            # 4. Fusion multi-modale
            combined = np.concatenate([image_vec, text_vec])  # (1280D)
            combined = combined / np.linalg.norm(combined)

            vectors.append({
                'vector': combined,
                'scene_start': scene_start,
                'scene_end': scene_end,
                'transcript': transcript,
                'keyframe_path': keyframe
            })

        return vectors

    def search_video_content(self, query_text):
        """Recherche sémantique dans archives"""
        query_vec = self.vectorize_text(query_text)

        # Recherche Qdrant
        results = self.qdrant.search(
            collection_name='video_scenes',
            query_vector=query_vec.tolist(),
            limit=20
        )

        # Grouper par vidéo source
        videos = {}
        for hit in results:
            video_id = hit.payload['video_id']
            if video_id not in videos:
                videos[video_id] = []
            videos[video_id].append({
                'score': hit.score,
                'timestamp': hit.payload['scene_start'],
                'transcript': hit.payload['transcript']
            })
```

```
        return videos
```

Architecture Technique

- **Ingestion** : Pipeline Airflow traite 1000h vidéo/jour (GPU cluster 8x A100)
- **Stockage** : Weaviate (base vectorielle) + S3 (vidéos sources)
- **Recherche** : Interface web React, latence <200ms pour recherche dans 200K heures
- **Coûts** : $15K/mois infrastructure (GPU + stockage + API Whisper)

Résultats

Métrique	Avant	Après
Temps recherche archives	45 min (manuel)	2 min (automatique)
Taux ré-utilisation archives	8%	34%
Précision recherche	35% (metadata)	82% (contenu)
Requêtes/jour	~50	~800

Leçons apprises et pièges à éviter

Pièges Techniques Fréquents

1. Oublier la Normalisation

Erreur : Utiliser vecteurs bruts sans normalisation L2 pour similarité cosinus.

Impact : Résultats biaisés par magnitude des vecteurs (longs documents sur-représentés). Pour approfondir, consultez Prompt Engineering Avancé : Chain-of-Thought et Techniques.

Solution : `vector = vector / np.linalg.norm(vector)` systématiquement.

2. Data Leakage lors du Fine-Tuning

Erreur : Fit du scaler/vectorizer sur train+test ensemble.

Impact : Métriques surestimées en dev, chute en production.

Solution : Toujours `fit` sur train only, `transform` sur test.

3. Ignorer la Drift des Modèles

Erreur : Ne jamais re-vectoriser après changement de modèle.

Impact : Vecteurs anciens (BERT-v1) incompatibles avec nouveaux (BERT-v2).

Solution : Versioning strict + re-vectorisation complète si modèle change.

Bonnes Pratiques Production

Checklist Pré-Production

- **Versioning** : Inclure version modèle dans métadonnées vecteurs
- **Monitoring** : Métriques latence, throughput, erreurs en temps réel
- **Fallback** : Stratégie de secours si service vectorisation down (cache, keyword search)

- **A/B Testing** : Tester nouveau modèle sur 5-10% trafic avant full rollout
- **Documentation** : Documenter préprocessing, modèle, hyperparams pour reproductibilité
- **Sécurité** : Anonymisation données sensibles, chiffrement vecteurs at-rest
- **Coûts** : Estimer coûts compute (GPU), stockage (base vectorielle), API (OpenAI)

Questions à se Poser Avant de Vectoriser

1. **Quel est le use case exact ?** (recherche, recommandation, classification, clustering)
2. **Quel volume de données ?** (100 docs vs 10M docs = stratégie différente)
3. **Quelle latence acceptable ?** (temps réel <100ms vs batch quotidien)
4. **Mono ou multi-langue ?** (modèle multilingue requis si multi-langue)
5. **Domaine spécialisé ?** (médical, juridique = fine-tuning probablement nécessaire)
6. **Budget ?** (self-hosted vs API cloud, GPU vs CPU)
7. **Fréquence mise à jour ?** (statique vs incrémental vs temps réel)
8. **Contraintes réglementaires ?** (RGPD, souveraineté données, etc.)

Ressources et Formation Équipe

La vectorisation nécessite compétences multidisciplinaires :

- **Data Engineers** : Pipelines ETL, gestion infra GPU, bases vectorielles
- **ML Engineers** : Fine-tuning modèles, optimisation inference, monitoring
- **Backend Devs** : Intégration APIs, caching, systèmes distribués
- **DevOps** : Kubernetes, GPU orchestration, scaling horizontal

Investissement formation estimé : 2-4 semaines par profil pour montée en compétences (cours, POCs, projets pilotes).

Sources et références : ArXiv IA · Hugging Face Papers

Questions fréquentes

Quelle technique de vectorisation choisir pour mon projet ?

Le choix dépend de 3 facteurs : **type de données**, **volume** et **contraintes** (latence, budget). Pour du texte :

- **Baseline rapide** : TF-IDF (sklearn) en 5 lignes de code
- **Production standard** : Sentence Transformers (all-MiniLM-L6-v2) - bon compromis qualité/vitesse
- **Qualité maximale** : OpenAI text-embedding-3-large (API payante)
- **Multilingue** : paraphrase-multilingual-mpnet-base-v2 (50+ langues)
- **Domaine spécialisé** : Fine-tuning BERT sur corpus métier

Pour images : CLIP (multi-modal) ou ResNet-50 (classification). Pour audio : Whisper encoder.

Comment gérer la vectorisation de millions de documents ?

Pour gros volumes (>1M documents), adoptez une approche distribuée :

1. **Batch processing** : Traiter par mini-batches (32-128) au lieu d'un par un (gain 10-50x)
2. **GPU clusters** : Utiliser plusieurs GPUs en parallèle (DataParallel ou Kubernetes + GPU nodes)
3. **Streaming** : Ne pas charger tous les docs en mémoire, traiter en streaming depuis fichier/database
4. **Caching** : Stocker vecteurs générés dans Redis (cache chaud) + base vectorielle (persistant)
5. **Incrémental** : Vectoriser seulement nouveaux/modifiés documents, pas tout re-traiter

Exemple architecture : Apache Airflow (orchestration) + Spark (distribué) + GPU cluster (8x T4) peut traiter 10M documents en 6-8h.

Faut-il toujours utiliser des modèles pré-entraînés ?

Oui, presque toujours pour la vectorisation. Raisons :

- **Transfer learning** : Modèles pré-entraînés sur milliards de tokens capturent patterns génériques
- **Coût entraînement** : Entraîner BERT from scratch = $50K+ en GPU (6 semaines)
- **Données requises** : 100M+ exemples pour entraînement from scratch (irréaliste pour PME)
- **Qualité supérieure** : Même sur domaines spécialisés, fine-tuning > from scratch

Exceptions rares : Langues très rares (pas de modèle pré-entraîné), ou contraintes sécurité extrêmes (impossibilité d'utiliser modèles externes). Dans 99% des cas : partir d'un pré-entraîné + fine-tuning si nécessaire.

Comment mesurer la qualité d'une vectorisation ?

La qualité se mesure selon le cas d'usage :

Pour recherche sémantique

- **MRR (Mean Reciprocal Rank)** : Position du 1er résultat pertinent
- **NDCG (Normalized Discounted Cumulative Gain)** : Qualité du ranking complet
- **Recall@K** : % résultats pertinents dans top-K

Pour classification

- **Accuracy** : % prédictions correctes
- **F1-score** : Médiane précision/recall (mieux si classes déséquilibrées)

Pour clustering

- **Silhouette score** : Cohésion intra-cluster vs séparation inter-cluster (-1 à 1, optimal ~0.7+)
- **Davies-Bouldin index** : Similarité moyenne entre clusters (plus bas = mieux)

Benchmark pratique : Créer dataset test de 100-500 paires (query, document pertinent) annotées manuellement, mesurer Recall@5 et MRR. Objectif : Recall@5 > 80%, MRR > 0.7.

Peut-on combiner plusieurs techniques de vectorisation ?

Oui, absolument ! Les approches hybrides donnent souvent les meilleurs résultats. Techniques courantes :

1. Concaténation Multi-Modale
Combiner vecteurs de différentes modalités :

Pour approfondir, consultez les ressources officielles : Hugging Face, arXiv et ANSSI.

```
vector_text = encode_text(description)      # (768D)
vector_image = encode_image(photo)         # (512D)
vector_features = encode_features(price, category)  # (50D)

# Concaténation
vector_final = concat([vector_text, vector_image, vector_features])  # (1330D)
vector_final = normalize(vector_final)  # IMPORTANT
```

2. Ensemble de Modèles
Utiliser plusieurs modèles et agréger scores :

```
score_bert = cosine_similarity(query_bert, doc_bert)
score_tfidf = cosine_similarity(query_tfidf, doc_tfidf)

# Pondération (tune par grid search)
score_final = 0.7 * score_bert + 0.3 * score_tfidf
```

3. Stacking
Entraîner un modèle superviseur qui apprend à combiner vecteurs de base.

Règle d'or : La concaténation simple fonctionne bien en pratique. Techniques avancées (attention, gating) apportent 2-5% gain supplémentaire mais complexité ++.

> **Ressources open source associées :**
>
> • awesome-cybersecurity-tools — Liste de 100+ outils de cybersécurité

Ayi NEDJIMI Consultants — Expert cybersécurité offensive & intelligence artificielle

ayinedjimi-consultants.fr · ayi@ayinedjimi-consultants.fr