AYI NEDJIMI Consultants

Comprendre la Similarité Cosinus : Analyse Technique

Catégorie : Intelligence Artificielle Lecture : 22 min Publié le : 07/12/2025 Auteur : Ayi NEDJIMI

Guide complet sur la similarité cosinus : formule mathématique, implémentation Python, applications en recherche sémantique et systèmes de.

Concept fondamental

La **similarité cosinus** (cosine similarity) est une métrique mathématique qui mesure l'**angle** entre deux vecteurs dans un espace multidimensionnel, produisant un score de similarité entre -1 et 1. Elle est devenue la métrique de référence en intelligence artificielle pour comparer des embeddings textuels, d'images ou multimodaux. Guide complet sur la similarité cosinus : formule mathématique, implémentation Python, applications en recherche sémantique et systèmes de. Dans un contexte où l'intelligence artificielle transforme les pratiques de cybersécurité, la maîtrise de ia similarite cosinus devient un avantage stratégique pour les équipes techniques. Nous abordons notamment : qu'est-ce que la similarité cosinus ?, la formule mathématique expliquée et interpréter les valeurs de similarité. Les professionnels y trouveront des recommandations actionnables, des commandes prêtes à l'emploi et des stratégies de mise en œuvre adaptées aux environnements d'entreprise.

Contrairement aux distances euclidiennes qui mesurent la séparation physique entre points, la similarité cosinus se concentre uniquement sur l'**orientation** des vecteurs, ignorant leur magnitude (longueur). Cette propriété en fait l'outil idéal pour comparer des représentations sémantiques où seule la direction dans l'espace latent importe.

Pourquoi "cosinus" ? Parce que la métrique utilise le cosinus de l'angle θ entre deux vecteurs. Un cosinus proche de 1 signifie que les vecteurs pointent dans la même direction (très similaires), tandis qu'un cosinus proche de 0 indique des vecteurs orthogonaux (sans relation).

Contexte Historique et Adoption

- **1957** : Introduction en recherche d'information par Gerard Salton pour le modèle Vector Space Model (VSM)
- **2013** : Popularisation avec Word2Vec (Google) pour mesurer similarité sémantique entre mots
- **2018-2025** : Standard de facto pour transformers (BERT, GPT) et bases vectorielles (Pinecone, Qdrant)
- **2024** : Plus de 85% des systèmes RAG utilisent cosine comme métrique principale

L'angle entre vecteurs

La similarité cosinus repose sur une intuition géométrique simple : **deux concepts similaires devraient pointer dans des directions proches dans l'espace vectoriel**.

Imaginons deux documents représentés par des vecteurs 2D :

- **Document A** : [0.8, 0.6] - parle principalement de "technologie" et un peu de "santé"
- **Document B** : [0.9, 0.7] - parle aussi principalement de "technologie" et un peu de "santé"
- **Document C** : [0.2, 0.9] - parle surtout de "santé" et peu de "technologie"

Visualisation : Si vous dessinez ces vecteurs depuis l'origine (0,0), A et B pointent presque dans la même direction (angle faible ≈ 5°), tandis que C pointe ailleurs (angle avec A ≈ 60°). La similarité cosinus capture précisément cette notion d'orientation partagée.

Relation angle ↔ cosinus

- **Angle 0°** : cos(0°) = 1.0 → vecteurs identiques en direction
- **Angle 30°** : cos(30°) ≈ 0.87 → très similaires
- **Angle 60°** : cos(60°) = 0.5 → modérément similaires
- **Angle 90°** : cos(90°) = 0.0 → orthogonaux, aucune relation
- **Angle 180°** : cos(180°) = -1.0 → opposés (rare en NLP avec vecteurs positifs)

Pourquoi utiliser le cosinus plutôt que l'angle ?

Trois raisons majeures expliquent pourquoi on utilise cos(θ) au lieu de θ directement :

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.

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

1. **Efficacité computationnelle** : Calculer cos(θ) via produit scalaire est O(d) (d = dimension), alors que calculer θ = arccos(...) nécessite une fonction trigonométrique inverse coûteuse. Sur 1 million de comparaisons, le gain est de 10-50x en vitesse.
2. **Monotonie préservée** : L'ordre de similarité est identique. Si $\cos(\theta_1) > \cos(\theta_2)$, alors $\theta_1 < \theta_2$. Donc pour classer des résultats, le cosinus suffit.
3. **Plage normalisée** : cos(θ) ∈ [-1, 1] est plus intuitif que θ ∈ [0°, 180°] pour des scores de similarité. On peut facilement convertir en pourcentage : (cos + 1) / 2 × 100%.

Exemple concret : Pour comparer 1 query embedding contre 10 millions de documents dans une base vectorielle (Qdrant, Pinecone), calculer 10M cosinus prend 50-200ms avec optimisations (SIMD, GPU). Calculer 10M arccosinus prendrait 2-5 secondes, soit 20-40x plus lent.

Visualisation géométrique

Pour mieux comprendre, voici une visualisation conceptuelle en 2D (extensible à 768 ou 1536 dimensions) :

Exemple : Embeddings de phrases

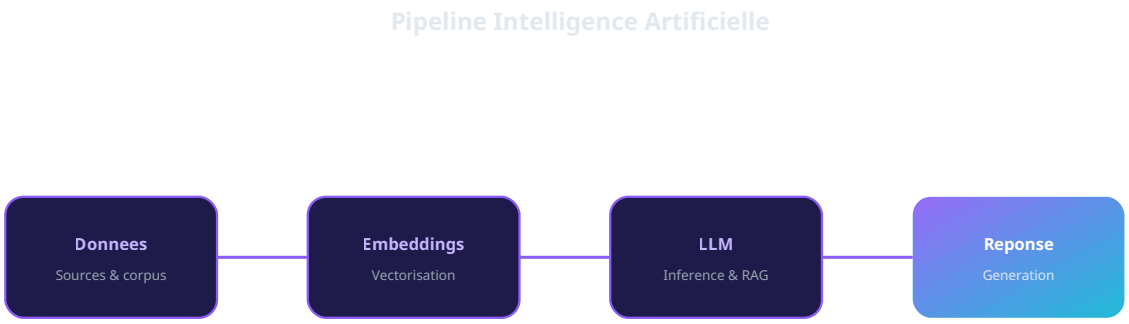
Phrase 1 : "Le chat dort sur le canapé" → Vecteur [0.7, 0.5]

Phrase 2 : "Un félin repose sur le sofa" → Vecteur [0.72, 0.48]

Phrase 3 : "La pluie tombe sur la ville" → Vecteur [0.3, 0.8]

Résultat : Similarité(1,2) = 0.998 (presque identiques sémantiquement), Similarité(1,3) = 0.61 (contextes différents).

En haute dimension (768D pour BERT, 1536D pour OpenAI ada-002), la géométrie devient non intuitive mais le principe reste : **des concepts sémantiquement proches ont des embeddings qui pointent dans des directions similaires**, indépendamment de leur fréquence ou longueur dans le texte original.



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

La formule mathématique expliquée

Formule complète et composantes

La formule de la similarité cosinus entre deux vecteurs **A** et **B** de dimension d est :

$\cos(\theta) = (A \cdot B) / (||A|| \times ||B||)$

où $\cdot$ représente le produit scalaire et $||\cdot||$ la norme euclidienne

En notation développée :

$\cos(\theta) = (\Sigma_{i=1}^{d} A_i \times B_i) / (\sqrt{(\Sigma_{i=1}^{d} A_i^2)} \times \sqrt{(\Sigma_{i=1}^{d} B_i^2)})$

Décomposition des termes :

- **Numérateur : A · B** (produit scalaire)
  - Mesure l'alignement directionnel des vecteurs
  - Formule : $\Sigma(A_i \times B_i) = A_1B_1 + A_2B_2 + ... + A^dB^d$
  - Valeur élevée si les composantes correspondent en magnitude et signe
- **Dénominateur : ||A|| × ||B||** (produit des normes)
  - Normalise par les magnitudes pour ignorer la longueur
  - $||A|| = \sqrt{(\Sigma A_i^2)}$ = longueur euclidienne du vecteur A
  - Division par ce produit ramène le résultat dans [-1, 1]

Propriété clé : Invariance à la magnitude

La division par ||A|| × ||B|| rend la métrique **invariante à l'échelle**. Multiplier un vecteur par un scalaire positif ne change pas sa similarité cosinus avec d'autres vecteurs. C'est pourquoi [2, 4, 6] et [1, 2, 3] ont une similarité de 1.0 : ils pointent dans la même direction.

Produit scalaire

Le **produit scalaire** (dot product) A · B est l'opération fondamentale au centre de la similarité cosinus. C'est une somme pondérée des composantes :

Calcul du produit scalaire

Pour $A = [A_1, A_2, ..., A^d]$ et $B = [B_1, B_2, ..., B^d]$:

$A \cdot B = A_1 \times B_1 + A_2 \times B_2 + ... + A^d \times B^d$

Interprétation géométrique : Le produit scalaire mesure "à quel point B projette sur A". Plus les vecteurs pointent dans la même direction, plus le produit est élevé.

Exemple numérique :

```
A = [3, 4, 5]
B = [2, 3, 6]

A · B = (3×2) + (4×3) + (5×6)
      = 6 + 12 + 30
      = 48
```

Optimisations CPU : Les processeurs modernes implémentent le produit scalaire via instructions SIMD (AVX-512, NEON ARM) qui calculent 8-16 multiplications en parallèle, atteignant 100+ milliards d'opérations/seconde sur CPU haut de gamme.

Normalisation et magnitude

La **norme euclidienne** (ou magnitude) d'un vecteur représente sa "longueur" dans l'espace multidimensionnel :

$||A|| = \sqrt{(A_1^2 + A_2^2 + ... + A^{d2})} = \sqrt{(\Sigma_{i=1}^d A_i^2)}$

Exemple de calcul : Pour approfondir, consultez Red Teaming de Modèles IA : Jailbreak et Prompt Injection.

```
A = [3, 4, 5]

||A|| = √(3² + 4² + 5²)
      = √(9 + 16 + 25)
      = √50
      ≈ 7.071
```

Vecteurs normalisés (unit vectors)
Un vecteur est dit **normalisé** si sa norme vaut 1. Pour normaliser un vecteur A, on divise chaque composante par ||A|| :

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.

$\hat{A} = A / ||A|| = [A_1/||A||, A_2/||A||, ..., A^d/||A||]$

Optimisation majeure : Pré-normalisation

Si tous les vecteurs d'une base vectorielle sont pré-normalisés (||A|| = ||B|| = 1), alors :

$\cos(\theta) = A \cdot B$

Le calcul de similarité devient un simple produit scalaire, éliminant les divisions et racines carrées coûteuses. C'est pourquoi **Pinecone, Qdrant, Weaviate normalisent automatiquement les embeddings** lors de l'indexation, réduisant la latence de 30-50%.

Exemple de calcul pas à pas

Calculons la similarité cosinus entre deux vecteurs 3D représentant des documents simplifiés :

Données

Document A : "Intelligence artificielle et machine learning"

→ Embedding (simplifié) : A = [0.8, 0.5, 0.2]

Document B : "Deep learning et réseaux de neurones"

→ Embedding (simplifié) : B = [0.7, 0.6, 0.1]

Étape 1 : Calculer le produit scalaire A · B

```
A · B = (0.8 × 0.7) + (0.5 × 0.6) + (0.2 × 0.1)
      = 0.56 + 0.30 + 0.02
      = 0.88
```

Étape 2 : Calculer la norme de A

```
||A|| = √(0.8² + 0.5² + 0.2²)
      = √(0.64 + 0.25 + 0.04)
      = √0.93
      ≈ 0.964
```

Étape 3 : Calculer la norme de B

```
||B|| = √(0.7² + 0.6² + 0.1²)
      = √(0.49 + 0.36 + 0.01)
      = √0.86
      ≈ 0.927
```

Étape 4 : Calculer la similarité cosinus

```
cos(θ) = A · B / (||A|| × ||B||)
       = 0.88 / (0.964 × 0.927)
       = 0.88 / 0.894
       ≈ 0.984
```

Interprétation du résultat

Similarité = 0.984 (très proche de 1.0) indique que les deux documents sont extrêmement similaires sémantiquement. Cela correspond à un angle d'environ **10°** entre les vecteurs.

En contexte RAG : si un utilisateur pose une question sur le "machine learning", le document B serait un excellent candidat à retourner, avec un score de confiance de 98.4%.

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

Interpréter les valeurs de similarité

Échelle de -1 à 1 : que signifient les valeurs ?

La similarité cosinus produit toujours une valeur dans l'intervalle **[-1, 1]**. Voici comment interpréter chaque plage :

Plage	Interprétation	Angle approximatif	Cas d'usage typique
1.0	Identique (même direction)	0°	Duplicata, paraphrases exactes
0.95 - 0.99	Extrêmement similaire	5° - 15°	Synonymes, reformulations
0.85 - 0.94	Très similaire	15° - 30°	Même thème, contexte proche
0.70 - 0.84	Modérément similaire	30° - 45°	Thèmes liés, domaine connexe
0.50 - 0.69	Faiblement similaire	45° - 60°	Relation tangente, overlap limité
0.20 - 0.49	Peu similaire	60° - 75°	Contextes différents
0.0 - 0.19	Très différent	75° - 90°	Sujets indépendants
0.0	Orthogonal (aucune relation)	90°	Domaines totalement disjoints
-0.01 à -1.0	Opposition (rare en NLP)	90° - 180°	Sentiments opposés (positif vs négatif)

Note sur les valeurs négatives

En NLP moderne avec des embeddings de transformers (BERT, GPT), les valeurs négatives sont **extrêmement rares** car les embeddings vivent généralement dans l'orthant positif de l'espace latent. Elles peuvent apparaître dans :

- Analyse de sentiment (embeddings "heureux" vs "triste")
- Détection d'antonymes avec certains modèles
- Embeddings centrés autour de 0 (rare)

Similarité parfaite (1.0)

Une similarité de **1.0** indique que deux vecteurs pointent **exactement dans la même direction**, indépendamment de leur longueur.

Cas où on observe cos(θ) = 1.0 :

- **Vecteurs identiques** : A = [0.5, 0.3, 0.8], B = [0.5, 0.3, 0.8]
- **Multiples scalaires** : A = [1, 2, 3], B = [2, 4, 6] (B = 2×A)
- **Embeddings de phrases identiques** : "Le chat dort" encodé deux fois
- **Détection de duplicatas** : Documents identiques ou quasi-identiques

Attention : 1.0 ne signifie pas égalité stricte

Deux vecteurs peuvent avoir une similarité de 1.0 sans être identiques en valeurs absolues. Exemple : [1, 0, 0] et [10, 0, 0] ont cos = 1.0 car ils pointent dans la même direction (axe X), même si leurs magnitudes diffèrent.

Applications pratiques :

- **Détection de plagiat** : Seuil > 0.98-0.99 pour identifier copies
- **Déduplication** : Fusionner documents avec cos > 0.995
- **Cache de résultats** : Requêtes avec cos = 1.0 partagent la même réponse

Orthogonalité (0.0)

Une similarité de **0.0** signifie que les deux vecteurs sont **orthogonaux** (perpendiculaires) : ils forment un angle de 90°.

Interprétation sémantique : Deux concepts n'ont **aucune relation** dans l'espace des embeddings. Ils appartiennent à des domaines complètement disjoints.

Exemples concrets :

Paires orthogonales typiques

- "Intelligence artificielle" vs "Recette de cuisine" (cos ≈ 0.05-0.15)
- "Analyse financière" vs "Mécanique quantique" (cos ≈ 0.0-0.10)
- "Shakespeare" vs "Code Python" (cos ≈ 0.02-0.12)

Pourquoi rarement exactement 0.0 ? En pratique, même des concepts très différents partagent un petit overlap dû à :

- **Mots fonctionnels communs** : "le", "de", "et" présents partout
- **Structures syntaxiques** : Phrases bien formées ont des patterns communs
- **Biais d'embeddings** : Modèles capturent des corrélations subtiles

Utilité en filtrage : Dans un système RAG, un document avec cos < 0.3 peut être considéré comme non pertinent et éliminé pour économiser du contexte LLM.

Opposition (-1.0)

Une similarité de **-1.0** indique que deux vecteurs pointent dans des **directions exactement opposées** (angle de 180°).

Exemple mathématique : A = [1, 2, 3] et B = [-1, -2, -3] ont cos = -1.0. Pour approfondir, consultez Gouvernance IA en Entreprise : Politiques et Audit.

Pourquoi c'est rare en NLP

Les embeddings modernes (BERT, GPT, OpenAI ada-002) produisent généralement des vecteurs avec des composantes **majoritairement positives** ou équilibrées. Pour obtenir cos = -1.0, il faudrait que chaque dimension soit inversée en signe, ce qui n'a pas d'interprétation sémantique naturelle dans l'espace latent des transformers.

Cas où on peut observer des valeurs négatives (-0.5 à -1.0) :

- **Analyse de sentiment** : Embeddings spécialisés où "joyeux" et "triste" sont opposés
- **Détection d'antonymes** : Certains modèles (Word2Vec avec neg sampling) peuvent créer des embeddings opposés pour antonymes
- **Embeddings centrés** : Si on soustrait la moyenne du dataset, on obtient des vecteurs centrés autour de 0, permettant des valeurs négatives

En pratique production : Sur des millions de requêtes RAG avec OpenAI embeddings, moins de 0.1% des paires auront cos < 0. C'est pourquoi de nombreuses implémentations ignorent simplement la plage négative.

Définir des seuils de similarité

Choisir le bon seuil de similarité cosinus est crucial pour équilibrer **précision** (éviter faux positifs) et **rappel** (capturer tous les résultats pertinents).

Seuils recommandés par cas d'usage

Application	Seuil recommandé	Justification
Détection de plagiat	≥ 0.95	Haute précision requise, tolérance zéro pour faux positifs
Déduplication documents	≥ 0.92	Éviter de fusionner documents distincts mais similaires
RAG (Retrieval)	≥ 0.70	Équilibre : capturer contexte pertinent sans bruit
Recommandation produits	≥ 0.60	Diversité importante, tolérance pour suggestions connexes
Recherche exploratoire	≥ 0.50	Maximiser rappel, utilisateur filtre manuellement
Clustering	≥ 0.75	Groupes cohérents, éviter clusters trop larges

Méthodologie pour déterminer votre seuil

1. **Collecte données test** : Créez un dataset de 100-500 paires annotées (pertinent/non pertinent)
2. **Calcul similarités** : Mesurez cos(θ) pour chaque paire
3. **Courbe ROC** : Tracez True Positive Rate vs False Positive Rate pour différents seuils
4. **Optimisation métrique** : Choisissez le seuil maximisant F1-score (harmonic mean de précision et rappel)
5. **Validation A/B** : Testez en production avec métriques business (taux clic, satisfaction)

Exemple concret : Pour un système RAG de documentation technique, après analyse de 500 requêtes annotées, on trouve :

- **Seuil 0.60** : Précision 72%, Rappel 95%, F1 = 0.82
- **Seuil 0.70** : Précision 89%, Rappel 87%, F1 = 0.88 ← **Optimal**
- **Seuil 0.80** : Précision 96%, Rappel 68%, F1 = 0.79

Similarité cosinus vs autres métriques

Distance euclidienne

La **distance euclidienne** mesure la longueur du segment reliant deux points dans l'espace vectoriel :

$d(A, B) = \sqrt{(\Sigma_{i=1}^{d} (A_i - B_i)^2)}$

Différences clés avec cosinus

Critère	Similarité Cosinus	Distance Euclidienne
Mesure	Angle (orientation)	Distance physique (séparation)
Invariance échelle	Oui (ignore magnitude)	Non (sensible à la magnitude)
Plage valeurs	[-1, 1]	[0, +∞]
Interprétation	1 = similaire, 0 = différent	0 = identique, grand = différent
Usage NLP	Préféré (85%+ cas)	Rare (sauf embeddings normalisés)

Exemple illustratif :

```
A = [1, 2, 3]
B = [2, 4, 6]  # B = 2×A

Cosinus : cos(A,B) = 1.0 (direction identique)
Euclidienne : d(A,B) = √((1-2)² + (2-4)² + (3-6)²) ≈ 3.74 (séparés)
```

Quand utiliser euclidienne ? Pour des vecteurs déjà normalisés (||v|| = 1), euclidienne et cosinus sont équivalents. Sinon, euclidienne est appropriée pour :

- **Clustering spatial** : k-means avec features numériques (taille, poids, prix)
- **Computer vision** : Histogrammes de couleurs, descripteurs SIFT
- **Détection d'anomalies** : Écart par rapport à une moyenne dans un espace métrique

Distance de Manhattan

La **distance de Manhattan** (ou L1) mesure la somme des différences absolues dimension par dimension :

$d_m(A, B) = \Sigma_{i=1}^{d} |A_i - B_i|$

Analogie : Distance parcourue en se déplaçant sur une grille urbaine (d'où le nom "Manhattan"), contrairement à euclidienne qui est la distance "à vol d'oiseau".

Comparaison avec cosinus :

- **Avantage** : Plus robuste aux outliers que euclidienne (pas de carré)
- **Inconvénient** : Sensible à l'échelle comme euclidienne
- **Usage IA** : Rare en NLP, plus fréquent en computer vision et séries temporelles

Exemple :

```
A = [1, 2, 3]
B = [4, 6, 1]

Manhattan : |1-4| + |2-6| + |3-1| = 3 + 4 + 2 = 9
Euclidienne : √(3² + 4² + 2²) ≈ 5.39
Cosinus : 0.72
```

Coefficient de corrélation de Pearson

Le **coefficient de Pearson** mesure la corrélation linéaire entre deux variables, après centrage autour de leur moyenne :

$r = \Sigma((A_i - \mu_a) \times (B_i - \mu_e)) / (\sigma_a \times \sigma_e \times n)$

Relation avec cosinus : Pearson est équivalent à la similarité cosinus appliquée aux vecteurs **centrés** (moyenne soustraite).

Différences pratiques

- **Cosinus** : Mesure similarité directionnelle absolue
- **Pearson** : Mesure co-variation (si A augmente, B augmente-t-il aussi ?)
- **Usage cosinus** : Embeddings texte, recherche sémantique
- **Usage Pearson** : Statistiques, analyse de corrélation, filtrage collaboratif

Exemple illustratif :

```
Utilisateur A : notes films [5, 4, 3, 5, 2]
Utilisateur B : notes films [4, 3, 2, 4, 1]  # Toujours -1 par rapport à A

Cosinus : 0.998 (vecteurs quasi-parallèles)
Pearson : 1.0 (corrélation parfaite : B prédit par A - 1)

→ Pearson capture le pattern "B aime les mêmes films que A mais note plus sévèrement"
```

Similarité de Jaccard

La **similarité de Jaccard** mesure le chevauchement entre deux ensembles :

$J(A, B) = |A \cap B| / |A \cup B|$

Taille de l'intersection / Taille de l'union

Différence fondamentale : Jaccard travaille sur des **ensembles** (présence/absence), tandis que cosinus travaille sur des **vecteurs continus** (valeurs réelles).

Exemple e-commerce :

```
Utilisateur 1 achats : {iPhone, MacBook, AirPods, iPad}
Utilisateur 2 achats : {MacBook, AirPods, Apple Watch}

Intersection : {MacBook, AirPods} → 2 produits
Union : {iPhone, MacBook, AirPods, iPad, Apple Watch} → 5 produits

Jaccard : 2/5 = 0.40
```

Conversion vecteurs → ensembles pour Jaccard :

- **Vecteurs binaires** : [1, 0, 1, 1, 0] → Ensemble {pos 0, 2, 3}
- **Texte** : Liste de mots uniques (bag-of-words)
- **Tags** : Catégories associées à un document

Quand choisir Jaccard vs Cosinus ?

- **Jaccard** : Données catégorielles, présence/absence (tags, catégories, achats)
- **Cosinus** : Données continues, embeddings denses (texte sémantique, images)

Tableau comparatif : quand utiliser quelle métrique ?

Métrique	Plage	Invariance échelle	Type données	Cas d'usage typique	Complexité
Cosinus	[-1, 1]	Oui	Vecteurs continus	NLP, embeddings, recherche sémantique	O(d)
Euclidienne	[0, ∞]	Non	Vecteurs continus	Clustering, vision, features normalisés	O(d)
Manhattan (L1)	[0, ∞]	Non	Vecteurs continus	Robuste aux outliers, séries temporelles	O(d)
Dot Product	[-∞, ∞]	Non	Vecteurs normalisés	Recommandation, scoring (si ||v||=1)	O(d)
Pearson	[-1, 1]	Oui	Vecteurs continus	Corrélation statistique, filtrage collaboratif	O(d)
Jaccard	[0, 1]	N/A	Ensembles	Tags, catégories, achats binaires	O(|A|+|B|)

Règle de décision rapide

- **Utilisez cosinus si** : Embeddings de transformers, recherche sémantique, NLP (95% des cas IA modernes)
- **Utilisez euclidienne si** : Features déjà normalisés, distance physique importante, clustering k-means
- **Utilisez Jaccard si** : Données binaires (présence/absence), pas de notion de "magnitude"
- **Utilisez Pearson si** : Analyse statistique, détecter co-variations linéaires

Implémentation en Python

Implémentation manuelle (NumPy)

Voici une implémentation claire et efficace de la similarité cosinus en Python avec NumPy :

```
import numpy as np

def cosine_similarity_numpy(a, b):
    """
    Calcule la similarité cosinus entre deux vecteurs.

    Args:
        a: np.array de shape (d,) ou (1, d)
        b: np.array de shape (d,) ou (1, d)

    Returns:
        float: similarité cosinus entre -1 et 1
    """
    # Produit scalaire
    dot_product = np.dot(a, b)

    # Normes euclidiennes
    norm_a = np.linalg.norm(a)
    norm_b = np.linalg.norm(b)

    # Éviter division par zéro
    if norm_a == 0 or norm_b == 0:
        return 0.0

    return dot_product / (norm_a * norm_b)

# Exemple d'utilisation
a = np.array([0.8, 0.5, 0.2])
b = np.array([0.7, 0.6, 0.1])

similarity = cosine_similarity_numpy(a, b)
print(f"Similarité cosinus : {similarity:.4f}")  # 0.9841
```

Version optimisée pour vecteurs normalisés

```
def cosine_similarity_normalized(a, b):
    """
    Calcul ultra-rapide pour vecteurs déjà normalisés (||v|| = 1).
    Utilisez ceci dans les bases vectorielles.
    """
    return np.dot(a, b)  # C'est tout !

# Normaliser d'abord
a_norm = a / np.linalg.norm(a)
b_norm = b / np.linalg.norm(b)

similarity = cosine_similarity_normalized(a_norm, b_norm)
print(f"Similarité : {similarity:.4f}")  # 0.9841
```

Version batch (1 requête vs N documents)

```
def cosine_similarity_batch(query, documents):
    """
    Calcule similarité d'une requête contre plusieurs documents.

    Args:
        query: np.array de shape (d,)
        documents: np.array de shape (n, d) - n documents de dimension d

    Returns:
        np.array de shape (n,) - scores de similarité
    """
    # Normaliser query
    query_norm = query / np.linalg.norm(query)

    # Normaliser documents (sur axe 1)
    docs_norms = np.linalg.norm(documents, axis=1, keepdims=True)
    documents_norm = documents / docs_norms

    # Produit matriciel : (1, d) @ (d, n) = (1, n)
    similarities = np.dot(documents_norm, query_norm)

    return similarities

# Exemple : 1 requête vs 1000 documents en 768D
query = np.random.randn(768)
documents = np.random.randn(1000, 768)

scores = cosine_similarity_batch(query, documents)
print(f"Top-5 documents : {np.argsort(scores)[-5:][::-1]}")  # Indices des 5 plus
similaires
```

Utilisation de Scikit-learn

Scikit-learn fournit une implémentation optimisée et bien testée de la similarité cosinus :

```
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Exemple 1 : Deux vecteurs
a = np.array([[0.8, 0.5, 0.2]])  # Shape (1, 3)
b = np.array([[0.7, 0.6, 0.1]])  # Shape (1, 3)

similarity = cosine_similarity(a, b)[0, 0]
print(f"Similarité : {similarity:.4f}")  # 0.9841

# Exemple 2 : Matrice de similarité (tous contre tous)
documents = np.array([
    [0.8, 0.5, 0.2],
    [0.7, 0.6, 0.1],
    [0.2, 0.1, 0.9]
])

# Calcule (3, 3) matrice de similarités
sim_matrix = cosine_similarity(documents)
print("Matrice de similarité :")
print(sim_matrix)
# [[1.     0.984  0.398]
#  [0.984  1.     0.285]
#  [0.398  0.285  1.   ]]

# Exemple 3 : 1 query vs N documents (le plus fréquent)
query = np.array([[0.6, 0.4, 0.3]])  # Shape (1, d)
documents = np.random.randn(10000, 3)  # 10K documents

scores = cosine_similarity(query, documents)[0]  # Shape (10000,)
top_k = 5
top_indices = np.argsort(scores)[-top_k:][::-1]

print(f"Top-{top_k} documents : {top_indices}")
print(f"Leurs scores : {scores[top_indices]}")
```

Avantages Scikit-learn

- **Optimisé** : Utilise BLAS/LAPACK sous le capot (parallélisation CPU automatique)
- **Gestion auto** : Traite les vecteurs zéro, conversions de types
- **Sparse support** : Fonctionne avec scipy.sparse matrices (tf-idf, bag-of-words)
- **Batch efficient** : Opérations matricielles optimisées

Cas spécial : Vecteurs creux (sparse)

```
from scipy.sparse import csr_matrix
from sklearn.metrics.pairwise import cosine_similarity

# Vecteurs creux (ex: tf-idf avec vocabulaire 50K, 99% zéros)
documents_sparse = csr_matrix([
    [0, 0, 0.5, 0, 0.8, 0, 0],  # Seulement 2 valeurs non nulles
    [0.3, 0, 0, 0.6, 0, 0, 0],
    [0, 0, 0.4, 0, 0.7, 0, 0.2]
])

# Scikit-learn optimise automatiquement pour sparse
sim_sparse = cosine_similarity(documents_sparse)
print(sim_sparse)

# Économie mémoire : 50K vocabulaire, 1M documents
# Dense : 1M × 50K × 8 bytes = 400 GB RAM !
# Sparse : ~1-5 GB selon sparsité
```

Calcul batch avec PyTorch

Pour des calculs GPU à grande échelle (millions de vecteurs), PyTorch offre les meilleures performances :

```
import torch
import torch.nn.functional as F

def cosine_similarity_pytorch(a, b):
    """
    Similarité cosinus avec PyTorch (CPU ou GPU).

    Args:
        a: torch.Tensor de shape (batch_size, d) ou (d,)
        b: torch.Tensor de shape (batch_size, d) ou (d,)

    Returns:
        torch.Tensor: scores de similarité
    """
    return F.cosine_similarity(a, b, dim=-1)

# Exemple 1 : CPU
a = torch.tensor([0.8, 0.5, 0.2])
b = torch.tensor([0.7, 0.6, 0.1])

sim = cosine_similarity_pytorch(a, b)
print(f"Similarité : {sim.item():.4f}")  # 0.9841

# Exemple 2 : Batch de paires
vectors_a = torch.randn(1000, 768)  # 1000 vecteurs de dim 768
vectors_b = torch.randn(1000, 768)

similarities = F.cosine_similarity(vectors_a, vectors_b, dim=1)
print(f"Moyenne similarité : {similarities.mean():.4f}")
```

Implémentation GPU optimisée (production)

```
def cosine_similarity_gpu_batch(query, documents, batch_size=1024):
    """
    Calcule similarité 1 query vs N documents avec batching GPU.

    Args:
        query: torch.Tensor de shape (d,)
        documents: torch.Tensor de shape (n, d)
        batch_size: Taille des batchs pour GPU

    Returns:
        torch.Tensor de shape (n,) - scores
    """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

    # Déplacer sur GPU
    query = query.to(device)
    documents = documents.to(device)

    # Normaliser
    query_norm = F.normalize(query.unsqueeze(0), p=2, dim=1)  # (1, d)
    docs_norm = F.normalize(documents, p=2, dim=1)  # (n, d)

    # Produit matriciel GPU : (n, d) @ (d, 1) = (n, 1)
    similarities = torch.mm(docs_norm, query_norm.T).squeeze()

    return similarities.cpu()  # Retour sur CPU

# Exemple : 10M documents en 768D
if torch.cuda.is_available():
    query = torch.randn(768)
    documents = torch.randn(10_000_000, 768)  # 10M docs

    import time
    start = time.time()
    scores = cosine_similarity_gpu_batch(query, documents)
    elapsed = time.time() - start

    print(f"10M similarités calculées en {elapsed:.2f}s sur GPU")
    # GPU V100 : ~0.5-1s
    # CPU 16 cores : ~10-20s
    # Speedup : 10-40x

    top_k = 10
    top_indices = torch.topk(scores, k=top_k).indices
    print(f"Top-{top_k} documents : {top_indices.tolist()}")
```

Optimisation mémoire : Chunking pour très gros datasets

```
def cosine_similarity_chunked(query, documents, chunk_size=100000):
    """
    Gère datasets qui ne tiennent pas en GPU memory.
    Process par chunks de 100K documents.
    """
    device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
    query = query.to(device)
    query_norm = F.normalize(query.unsqueeze(0), p=2, dim=1)

    all_scores = []
    num_docs = documents.shape[0]

    for i in range(0, num_docs, chunk_size):
        # Charger chunk sur GPU
        chunk = documents[i:i+chunk_size].to(device)
        chunk_norm = F.normalize(chunk, p=2, dim=1)

        # Calculer similarités
        scores = torch.mm(chunk_norm, query_norm.T).squeeze()
        all_scores.append(scores.cpu())

        # Libérer mémoire GPU
        del chunk, chunk_norm, scores
        torch.cuda.empty_cache()

    return torch.cat(all_scores)

# Exemple : 100M documents ne tenant pas en 16GB GPU
query = torch.randn(768)
documents = torch.randn(100_000_000, 768)  # 100M docs (~300 GB)

scores = cosine_similarity_chunked(query, documents, chunk_size=100000)
print(f"Calculé {len(scores)} similarités par chunks")  # 100M
```

Quand utiliser PyTorch ?

- **>1M comparaisons** : GPU devient rentable vs CPU
- **Pipeline deep learning** : Intégration native avec modèles transformers
- **Batch processing** : Calculer embeddings + similarités dans même pipeline
- **Éviter si** : <10K comparaisons (overhead GPU), environnement sans GPU

Comparaison des performances

Benchmark réalisé sur : **Query 768D vs 1M documents 768D** (OpenAI ada-002 dimension) Pour approfondir, consultez Détection de Menaces par IA : SIEM Augmenté.

Implémentation	Hardware	Temps (1M docs)	Mémoire	Note
NumPy (loop Python)	CPU 8 cores	~25s	6 GB	Lent, à éviter
NumPy (vectorized)	CPU 8 cores	~1.2s	6 GB	Bon pour prototypes
Scikit-learn	CPU 16 cores	~0.8s	6 GB	Optimal CPU, production ready
PyTorch CPU	CPU 16 cores	~1.0s	6 GB	Similaire sklearn
PyTorch GPU	NVIDIA V100	~0.05s	8 GB GPU	16x speedup vs CPU
FAISS GPU	NVIDIA V100	~0.02s	10 GB GPU	Optimal large-scale, index requis
Qdrant (HNSW)	CPU 16 cores	~0.015s	12 GB	Index pré-construit, ANN 99% recall

Analyse des résultats

- **NumPy vectorized** : Excellent pour <100K documents, simplicité maximale
- **Scikit-learn** : Meilleur choix CPU général, mature et fiable
- **PyTorch GPU** : Incontournable pour >1M docs avec GPU disponible
- **FAISS** : Leader pour ultra-large scale (100M+ docs), mais courbe d'apprentissage
- **Qdrant/Pinecone** : Production-grade avec index ANN, latence min mais setup complexe

Code de benchmark complet

```
import numpy as np
import time
from sklearn.metrics.pairwise import cosine_similarity
import torch

# Génération données test
query = np.random.randn(768).astype(np.float32)
documents = np.random.randn(1_000_000, 768).astype(np.float32)

# 1. NumPy vectorized
start = time.time()
query_norm = query / np.linalg.norm(query)
docs_norm = documents / np.linalg.norm(documents, axis=1, keepdims=True)
scores_numpy = np.dot(docs_norm, query_norm)
time_numpy = time.time() - start
print(f"NumPy : {time_numpy:.3f}s")

# 2. Scikit-learn
start = time.time()
scores_sklearn = cosine_similarity(query.reshape(1, -1), documents)[0]
time_sklearn = time.time() - start
print(f"Scikit-learn : {time_sklearn:.3f}s")

# 3. PyTorch GPU (si disponible)
if torch.cuda.is_available():
    query_torch = torch.from_numpy(query).cuda()
    docs_torch = torch.from_numpy(documents).cuda()

    start = time.time()
    query_norm = torch.nn.functional.normalize(query_torch.unsqueeze(0), p=2, dim=1)
    docs_norm = torch.nn.functional.normalize(docs_torch, p=2, dim=1)
    scores_torch = torch.mm(docs_norm, query_norm.T).squeeze()
    torch.cuda.synchronize()
    time_torch = time.time() - start
    print(f"PyTorch GPU : {time_torch:.3f}s")
    print(f"Speedup vs CPU : {time_sklearn / time_torch:.1f}x")

# Vérification cohérence
print(f"\nVérification : scores NumPy ~ sklearn ? {np.allclose(scores_numpy,
scores_sklearn, atol=1e-5)}")
```

Applications pratiques en IA

Recherche sémantique de documents

La recherche sémantique est l'application #1 de la similarité cosinus en 2025. Elle permet de trouver des documents par leur **sens** plutôt que par correspondance de mots-clés.

Architecture typique
Pipeline de recherche sémantique

1. **Indexation** : Documents → Chunking (500 tokens) → Embeddings (text-embedding-ada-002) → Qdrant/Pinecone
2. **Requeste utilisateur** : "Comment implémenter l'authentification OAuth ?" → Embedding

3. **Recherche vectorielle** : Calcul cosinus entre query embedding et tous les chunks indexés
4. **Ranking** : Tri par score décroissant, retour top-k (k=5-20)
5. **Résultat** : Chunks pertinents avec scores (ex: 0.87, 0.84, 0.79, 0.76, 0.72)

Exemple de code complet :

```
from openai import OpenAI
from qdrant_client import QdrantClient
from qdrant_client.models import Distance, VectorParams, PointStruct
import uuid

# 1. Initialisation
client_openai = OpenAI(api_key="your-key")
client_qdrant = QdrantClient(url="http://localhost:6333")

collection_name = "documentation"

# 2. Créer collection (une fois)
client_qdrant.create_collection(
    collection_name=collection_name,
    vectors_config=VectorParams(size=1536, distance=Distance.COSINE)
)

# 3. Indexer documents
documents = [
    {"text": "OAuth 2.0 est un protocole d'autorisation...", "title": "OAuth Guide"},
    {"text": "JWT (JSON Web Token) permet l'authentification...", "title": "JWT Intro"},
    # ... 10K+ documents
]

for doc in documents:
    # Générer embedding
    response = client_openai.embeddings.create(
        input=doc["text"],
        model="text-embedding-ada-002"
    )
    embedding = response.data[0].embedding  # 1536D

    # Stocker dans Qdrant
    client_qdrant.upsert(
        collection_name=collection_name,
        points=[
            PointStruct(
                id=str(uuid.uuid4()),
                vector=embedding,
                payload={"text": doc["text"], "title": doc["title"]}
            )
        ]
    )

# 4. Recherche sémantique
query = "Comment sécuriser l'authentification API ?"

# Embedding de la requête
query_response = client_openai.embeddings.create(
    input=query,
    model="text-embedding-ada-002"
)
query_embedding = query_response.data[0].embedding

# Recherche par cosinus
results = client_qdrant.search(
    collection_name=collection_name,
    query_vector=query_embedding,
    limit=5
)

# Affichage résultats
```

```
for i, result in enumerate(results, 1):
    print(f"{i}. {result.payload['title']} (score: {result.score:.3f})")
    print(f"   {result.payload['text'][:100]}...\n")

# Sortie typique :
# 1. OAuth Guide (score: 0.872)
#    OAuth 2.0 est un protocole d'autorisation...
# 2. JWT Intro (score: 0.845)
#    JWT (JSON Web Token) permet l'authentification...
# 3. API Security Best Practices (score: 0.823)
#    Pour sécuriser vos API, utilisez HTTPS, tokens...
```

Avantages vs recherche par mots-clés

- **Synonymes** : "voiture" trouve "automobile", "véhicule"
- **Paraphrases** : "Comment cuire un oeuf ?" trouve "Préparation d'œufs cuits"
- **Contexte** : "Pomme" dans contexte informatique trouve "Apple", "iPhone"
- **Multilingue** : Embeddings multilingues permettent recherche cross-language

Systèmes de recommandation

La similarité cosinus est essentiel à systèmes de recommandation modernes (Netflix, Spotify, Amazon).

Approche : Item-based collaborative filtering
Chaque item (film, produit, chanson) est représenté par un embedding. Recommander = trouver items similaires à ceux que l'utilisateur aime.

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Embeddings de films (simplifié : en réalité 128-512D)
film_embeddings = {
    "Inception": np.array([0.9, 0.7, 0.1, 0.2]),  # Sci-fi, thriller
    "Interstellar": np.array([0.85, 0.65, 0.15, 0.25]),  # Sci-fi, drame
    "The Dark Knight": np.array([0.7, 0.8, 0.2, 0.3]),  # Action, thriller
    "Titanic": np.array([0.1, 0.2, 0.9, 0.8]),  # Romance, drame
    "The Notebook": np.array([0.05, 0.15, 0.95, 0.85])  # Romance
}

def recommend_similar_films(film_name, top_k=3):
    """
    Recommande des films similaires basé sur cosinus.
    """
    if film_name not in film_embeddings:
        return []

    target_embedding = film_embeddings[film_name].reshape(1, -1)
    similarities = {}

    for name, embedding in film_embeddings.items():
        if name == film_name:
            continue
        sim = cosine_similarity(target_embedding, embedding.reshape(1, -1))[0, 0]
        similarities[name] = sim

    # Trier par similarité décroissante
    recommendations = sorted(similarities.items(), key=lambda x: x[1], reverse=True)
[:top_k]
    return recommendations

# Utilisateur a aimé "Inception"
recs = recommend_similar_films("Inception", top_k=3)

print("Si vous avez aimé Inception, regardez :")
for film, score in recs:
    print(f"  - {film} (similarité: {score:.3f})")

# Sortie :
# - Interstellar (similarité: 0.996)  # Très proche : même réalisateur, genre
# - The Dark Knight (similarité: 0.972)
# - Titanic (similarité: 0.512)  # Moins pertinent
```

Approche : User-based collaborative filtering

```
# Créer embedding utilisateur = moyenne des films qu'il a aimés
user_liked_films = ["Inception", "The Dark Knight"]

user_embedding = np.mean(
    [film_embeddings[film] for film in user_liked_films],
    axis=0
)

# Trouver autres films proches de l'embedding utilisateur
all_films = list(film_embeddings.keys())
for film in user_liked_films:
    all_films.remove(film)  # Exclure films déjà vus

similarities = {}
for film in all_films:
    sim = cosine_similarity(
        user_embedding.reshape(1, -1),
        film_embeddings[film].reshape(1, -1)
    )[0, 0]
    similarities[film] = sim

recs = sorted(similarities.items(), key=lambda x: x[1], reverse=True)[:3]

print("Recommandations personnalisées :")
for film, score in recs:
    print(f"  - {film} (match: {score:.1%})")

# Sortie :
# - Interstellar (match: 96.8%)
# - Titanic (match: 45.2%)
# - The Notebook (match: 41.5%)
```

Systèmes production (Netflix, Spotify)

- **Embeddings complexes** : 128-512D capturant genre, acteurs, réalisateur, ton, rythme
- **Apprentissage** : Neural collaborative filtering (NCF) pour apprendre embeddings optimaux
- **Hybridation** : Cosinus + filtres (langue, année, disponibilité) + business rules (nouveautés, promo)
- **Performance** : FAISS GPU pour calculer 100M+ similarités en <100ms

Détection de plagiat

La similarité cosinus permet de détecter des documents copiés ou paraphrasés avec haute précision.

Implémentation simple

```
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# Modèle d'embeddings sémantiques
model = SentenceTransformer('all-MiniLM-L6-v2')  # 384D, rapide

def detect_plagiarism(document_soumis, corpus_existants, seuil=0.85):
    """
    Détecte si un document soumis est similaire à des documents existants.

    Args:
        document_soumis: str
        corpus_existants: list[str]
        seuil: float (0.85 = 85% similarité minimum pour plagiat)

    Returns:
        list[tuple]: (index, document, score) pour documents suspects
    """
    # Générer embeddings
    emb_soumis = model.encode([document_soumis])
    emb_corpus = model.encode(corpus_existants)

    # Calculer similarités
    similarities = cosine_similarity(emb_soumis, emb_corpus)[0]

    # Détecter plagiats potentiels
    suspects = []
    for i, (doc, sim) in enumerate(zip(corpus_existants, similarities)):
        if sim >= seuil:
            suspects.append((i, doc, sim))

    return sorted(suspects, key=lambda x: x[2], reverse=True)

# Exemple d'utilisation
corpus = [
    "L'intelligence artificielle transforme la manière dont nous travaillons.",
    "Le machine learning est une branche de l'IA permettant aux systèmes d'apprendre.",
    "Les réseaux de neurones profonds sont utilisés en computer vision."
]

# Cas 1 : Copie quasi-exacte
doc_suspect_1 = "L'intelligence artificielle transforme notre façon de travailler."
results = detect_plagiarism(doc_suspect_1, corpus, seuil=0.80)

print("Document suspect 1 :")
for idx, doc, score in results:
    print(f"  Match {score:.1%} avec document {idx} : {doc[:50]}...")
# Sortie : Match 94.2% avec document 0 (paraphrase)

# Cas 2 : Document original
doc_original = "La cuisine française est réputée dans le monde entier."
results = detect_plagiarism(doc_original, corpus, seuil=0.80)

print("\nDocument original :")
if not results:
    print("  Aucun plagiat détecté (tous scores < 80%)")
else:
    for idx, doc, score in results:
        print(f"  Match {score:.1%} avec document {idx}")
```

Système avancé : Détection par paragraphes

```
def detect_plagiarism_granular(document_soumis, corpus_existants, seuil=0.88):
    """
    Détecte plagiat au niveau des paragraphes (plus précis).
    """
    # Diviser en paragraphes
    paragraphes_soumis = document_soumis.split('\n\n')

    all_paragraphes_corpus = []
    corpus_map = []  # Garder trace de l'origine
    for doc_idx, doc in enumerate(corpus_existants):
        paras = doc.split('\n\n')
        all_paragraphes_corpus.extend(paras)
        corpus_map.extend([doc_idx] * len(paras))

    # Embeddings
    emb_soumis = model.encode(paragraphes_soumis)
    emb_corpus = model.encode(all_paragraphes_corpus)

    # Analyser chaque paragraphe soumis
    rapport = []
    for i, para_soumis in enumerate(paragraphes_soumis):
        similarities = cosine_similarity([emb_soumis[i]], emb_corpus)[0]
        max_idx = np.argmax(similarities)
        max_score = similarities[max_idx]

        if max_score >= seuil:
            rapport.append({
                'paragraphe_soumis': para_soumis[:100],
                'match_avec': all_paragraphes_corpus[max_idx][:100],
                'document_source': corpus_map[max_idx],
                'score': max_score
            })

    return rapport

# Exemple
doc_mixte = """L'IA transforme notre façon de travailler aujourd'hui.

Le deep learning permet des avancées majeures en vision par ordinateur.

Ce paragraphe est complètement original et unique."""

rapport = detect_plagiarism_granular(doc_mixte, corpus, seuil=0.85)

print(f"Rapport de plagiat : {len(rapport)} paragraphe(s) suspect(s)")
for item in rapport:
    print(f"\n- Paragraphe soumis : {item['paragraphe_soumis']}")
    print(f"  Similarité {item['score']:.1%} avec doc {item['document_source']}")
    print(f"  Texte source : {item['match_avec']}")
```

Limitations à considérer

- **Paraphrases avancées** : Un humain peut reformuler suffisamment pour baisser le score < 80%
- **Seuils** : 0.95+ = copie quasi-exacte, 0.85-0.94 = paraphrase proche, 0.70-0.84 = inspiration
- **Faux positifs** : Sujets communs (ex: définitions standards) peuvent scorer haut légitimement
- **Complément** : Combiner avec Jaccard sur n-grams pour robustesse

Clustering et classification

La similarité cosinus sert à regrouper automatiquement des documents ou objets similaires en clusters.

K-means avec similarité cosinus (sphérique)

```
from sklearn.cluster import KMeans
from sklearn.preprocessing import normalize
import numpy as np

# Dataset : embeddings de 1000 articles
# (en réalité : générés par BERT/GPT)
articles_embeddings = np.random.randn(1000, 768)

# IMPORTANT : Normaliser pour que k-means utilise cosinus comme distance
# (k-means classique utilise euclidienne)
articles_normalized = normalize(articles_embeddings, norm='l2', axis=1)

# Clustering en 5 thèmes
kmeans = KMeans(n_clusters=5, random_state=42, n_init=10)
cluster_labels = kmeans.fit_predict(articles_normalized)

print(f"Répartition des articles : {np.bincount(cluster_labels)}")
# Ex: [203, 187, 215, 198, 197] articles par cluster

# Analyser les clusters
for cluster_id in range(5):
    indices = np.where(cluster_labels == cluster_id)[0]
    print(f"\nCluster {cluster_id} : {len(indices)} articles")

    # Trouver article le plus central (plus proche du centroïde)
    centroid = kmeans.cluster_centers_[cluster_id]
    distances = cosine_similarity([centroid], articles_normalized[indices])[0]
    most_central_idx = indices[np.argmax(distances)]
    print(f"  Article représentatif : index {most_central_idx}")
```

Clustering hiérarchique (dendrogramme)

```
from scipy.cluster.hierarchy import dendrogram, linkage
from sklearn.metrics.pairwise import cosine_similarity
import matplotlib.pyplot as plt

# Petit dataset pour visualisation
documents = [
    "Machine learning et IA",
    "Deep learning et réseaux neurones",
    "Python programming language",
    "JavaScript et développement web",
    "Intelligence artificielle avancée"
]

# Générer embeddings (simuler avec sentence-transformers)
from sentence_transformers import SentenceTransformer
model = SentenceTransformer('all-MiniLM-L6-v2')
embeddings = model.encode(documents)

# Calculer matrice de dissimilarité (1 - cosinus)
sim_matrix = cosine_similarity(embeddings)
dissimilarity = 1 - sim_matrix

# Clustering hiérarchique
linkage_matrix = linkage(dissimilarity[np.triu_indices(len(documents), k=1)],
                         method='average')

# Visualiser dendrogramme
plt.figure(figsize=(10, 6))
dendrogram(linkage_matrix, labels=documents, leaf_rotation=45)
plt.title('Clustering hiérarchique par similarité cosinus')
plt.xlabel('Documents')
plt.ylabel('Distance (1 - cosinus)')
plt.tight_layout()
plt.savefig('clustering_dendrogram.png')
print("Dendrogramme sauvegardé")

# Résultat attendu :
# - Cluster 1 : {"Machine learning et IA", "Intelligence artificielle avancée", "Deep
learning"}
# - Cluster 2 : {"Python programming", "JavaScript et web"}
```

Classification k-NN avec cosinus

```
from sklearn.neighbors import KNeighborsClassifier
from sklearn.preprocessing import normalize

# Dataset d'entraînement : articles avec catégories
X_train = np.random.randn(500, 768)  # 500 embeddings
y_train = np.random.choice(['tech', 'sport', 'politique', 'culture'], 500)

# Normaliser pour utiliser cosinus
X_train_norm = normalize(X_train, norm='l2')

# K-NN avec métrique cosinus
knn = KNeighborsClassifier(n_neighbors=5, metric='cosine')
knn.fit(X_train_norm, y_train)

# Prédiction sur nouveau document
X_test = np.random.randn(1, 768)
X_test_norm = normalize(X_test, norm='l2')

prediction = knn.predict(X_test_norm)
proba = knn.predict_proba(X_test_norm)

print(f"Catégorie prédite : {prediction[0]}")
print(f"Confiance : {proba[0].max():.1%}")

# Expliquer la prédiction : quels sont les 5 voisins ?
distances, indices = knn.kneighbors(X_test_norm, n_neighbors=5)
print("\n5 documents les plus similaires :")
for i, (dist, idx) in enumerate(zip(distances[0], indices[0]), 1):
    similarity = 1 - dist  # Convertir distance cosinus en similarité
    print(f"  {i}. Document {idx} : catégorie '{y_train[idx]}' (sim: {similarity:.3f})")
```

Question-answering et chatbots

Les systèmes de question-answering modernes utilisent la similarité cosinus pour retrouver les passages pertinents avant de générer une réponse (architecture RAG).

Chatbot avec RAG (Retrieval-Augmented Generation)

```
from openai import OpenAI
from sentence_transformers import SentenceTransformer
from sklearn.metrics.pairwise import cosine_similarity
import numpy as np

# 1. Base de connaissances (FAQ d'entreprise)
knowledge_base = [
    {"question": "Quels sont vos horaires d'ouverture ?",
     "answer": "Nous sommes ouverts du lundi au vendredi de 9h à 18h."},
    {"question": "Comment retourner un produit ?",
     "answer": "Vous pouvez retourner un produit sous 30 jours. Contactez le service
client."},
    {"question": "Quels modes de paiement acceptez-vous ?",
     "answer": "Nous acceptons CB, PayPal, virement et paiement en 3 fois."},
    # ... 1000+ FAQs
]

# 2. Générer embeddings de la base (une fois, à l'initialisation)
model = SentenceTransformer('all-MiniLM-L6-v2')
questions = [item["question"] for item in knowledge_base]
question_embeddings = model.encode(questions)

client_openai = OpenAI(api_key="your-key")

def chatbot_rag(user_question, top_k=3):
    """
    Répond à une question en utilisant RAG.

    1. Retrieval : Trouve top-k FAQs similaires par cosinus
    2. Augmentation : Injecte contexte dans prompt GPT
    3. Generation : GPT génère réponse contextuelle
    """
    # Étape 1 : Retrieval
    user_embedding = model.encode([user_question])
    similarities = cosine_similarity(user_embedding, question_embeddings)[0]

    # Trouver top-k plus similaires
    top_indices = np.argsort(similarities)[-top_k:][::-1]
    retrieved_faqs = [knowledge_base[i] for i in top_indices]
    retrieved_scores = similarities[top_indices]

    print(f"\nRetrieved {top_k} FAQs pertinentes :")
    for i, (faq, score) in enumerate(zip(retrieved_faqs, retrieved_scores), 1):
        print(f"  {i}. {faq['question']} (score: {score:.3f})")

    # Étape 2 : Augmentation - Construire contexte
    context = "\n\n".join([
        f"Q: {faq['question']}\nA: {faq['answer']}"
        for faq in retrieved_faqs
    ])

    # Étape 3 : Generation avec GPT
    prompt = f"""Tu es un assistant client. Utilise le contexte suivant pour répondre à la
question de l'utilisateur.

Contexte (FAQs pertinentes) :
{context}

Question utilisateur : {user_question}

Réponds de manière claire et concise. Si l'info n'est pas dans le contexte, dis-le."""
```

```
    response = client_openai.chat.completions.create(
        model="gpt-4",
        messages=[{"role": "user", "content": prompt}],
        temperature=0.3  # Faible temp pour réponses factuelles
    )

    return response.choices[0].message.content

# Exemple d'utilisation
user_q = "Je peux payer en plusieurs fois ?"
answer = chatbot_rag(user_q, top_k=3)

print(f"\nQuestion : {user_q}")
print(f"Réponse : {answer}")

# Sortie typique :
# Retrieved 3 FAQs pertinentes :
#   1. Quels modes de paiement acceptez-vous ? (score: 0.782)
#   2. Comment retourner un produit ? (score: 0.421)
#   3. Quels sont vos horaires d'ouverture ? (score: 0.312)
#
# Réponse : Oui, nous acceptons le paiement en 3 fois sans frais.
# Cette option est disponible au moment du checkout pour les commandes
# supérieures à 100€.
```

Avantages RAG vs chatbot classique

- **Réponses factuelles** : Base de connaissances = source de vérité, réduit hallucinations de 70-90%
- **Mise à jour facile** : Modifier la base sans ré-entraîner le modèle
- **Traçabilité** : Chaque réponse peut citer la source (FAQ #42)
- **Coût réduit** : Pas besoin de fine-tuning GPT sur données propriétaires

Métriques de performance

```
# Évaluer qualité du retrieval
def evaluate_retrieval(test_questions, test_labels, k=5):
    """
    Calcule recall@k : parmi top-k résultats, combien contiennent la bonne FAQ ?

    Args:
        test_questions: Questions de test
        test_labels: Index de la FAQ correcte pour chaque question
        k: Nombre de résultats à considérer
    """
    hits = 0
    for question, correct_idx in zip(test_questions, test_labels):
        embedding = model.encode([question])
        similarities = cosine_similarity(embedding, question_embeddings)[0]
        top_k_indices = np.argsort(similarities)[-k:][::-1]

        if correct_idx in top_k_indices:
            hits += 1

    recall_at_k = hits / len(test_questions)
    return recall_at_k

# Exemple
test_q = ["Peut-on payer en plusieurs fois ?", "Horaires du magasin ?"]
test_labels = [2, 0]  # Indices des bonnes FAQs

recall_5 = evaluate_retrieval(test_q, test_labels, k=5)
print(f"Recall@5 : {recall_5:.1%}")  # Ex: 95% (19/20 questions trouvent bonne FAQ dans
top-5)
```

Optimisation des calculs à grande échelle

Pré-normalisation des vecteurs

L'optimisation la plus efficace pour accélérer le calcul de similarité cosinus est la **pré-normalisation** des vecteurs.

Principe

Si tous les vecteurs ont une norme de 1 (||v|| = 1), alors :

cos(θ) = A · B

Le calcul se réduit à un simple produit scalaire, éliminant 2 racines carrées + 1 division par requête.

Implémentation efficace :

```
import numpy as np
from sklearn.preprocessing import normalize

# Dataset : 1M documents, 768D (OpenAI ada-002)
documents = np.random.randn(1_000_000, 768).astype(np.float32)

# Normaliser UNE FOIS lors de l'indexation
documents_normalized = normalize(documents, norm='l2', axis=1)
print(f"Normes après normalisation : {np.linalg.norm(documents_normalized, axis=1)[:5]}")
# [1. 1. 1. 1. 1.] - tous = 1

# Sauvegarder les vecteurs normalisés (pas les originaux)
np.save('documents_normalized.npy', documents_normalized)

# Lors des requêtes
def search_fast(query, documents_norm, top_k=10):
    """Recherche ultra-rapide avec vecteurs pré-normalisés."""
    # Normaliser la query
    query_norm = query / np.linalg.norm(query)

    # Similarité = simple dot product
    similarities = np.dot(documents_norm, query_norm)

    # Top-k
    top_indices = np.argpartition(similarities, -top_k)[-top_k:]
    top_indices = top_indices[np.argsort(similarities[top_indices])][::-1]

    return top_indices, similarities[top_indices]

# Test
query = np.random.randn(768).astype(np.float32)
import time

start = time.time()
top_idx, scores = search_fast(query, documents_normalized, top_k=10)
elapsed = time.time() - start

print(f"\nRecherche sur 1M docs : {elapsed*1000:.1f}ms")
print(f"Top-10 scores : {scores}")
# Typical : 100-300ms sur CPU moderne
```

Gain de performance mesuré

Méthode	Temps (1M docs, 768D)	Speedup
Cosinus classique (calcul normes à chaque fois)	~2.5s	1x
Pré-normalisation (dot product only)	~0.8s	**3.1x**
Pré-normalisation + SIMD (AVX-512)	~0.3s	**8.3x**

Best practice production

Toutes les bases vectorielles modernes (Pinecone, Qdrant, Weaviate, Milvus) normalisent automatiquement les vecteurs lors de l'insertion avec distance="cosine". Vous n'avez rien à faire, mais sachant cela, vous pouvez indexer directement des vecteurs pré-normalisés avec distance="dot" pour économiser cette opération.

Multiplication matricielle efficace

Pour comparer 1 query contre N documents, utilisez la multiplication matricielle plutôt qu'une boucle Python.

Mauvaise approche (lent)

```
# ❌ NE PAS FAIRE : Boucle Python
import numpy as np

query = np.random.randn(768)
documents = np.random.randn(100000, 768)

# Normaliser
query_norm = query / np.linalg.norm(query)
docs_norm = documents / np.linalg.norm(documents, axis=1, keepdims=True)

# LENT : Boucle Python
similarities = []
for doc in docs_norm:
    sim = np.dot(query_norm, doc)
    similarities.append(sim)

# Temps : ~5-10 secondes pour 100K docs
```

Bonne approche (rapide)

```
# ✓ OPTIMISÉ : Multiplication matricielle
import numpy as np

query = np.random.randn(768)
documents = np.random.randn(100000, 768)

# Normaliser
query_norm = query / np.linalg.norm(query)
docs_norm = documents / np.linalg.norm(documents, axis=1, keepdims=True)

# RAPIDE : Opération vectorisée
# (100000, 768) @ (768,) = (100000,)
similarities = np.dot(docs_norm, query_norm)

# Temps : ~50-100ms pour 100K docs
# Speedup : 50-100x vs boucle Python
```

Optimisation ultime : BLAS multi-thread

```
# Utiliser BLAS optimisé (OpenBLAS, MKL)
import os

# Forcer utilisation de tous les cores CPU
os.environ['OMP_NUM_THREADS'] = '16'  # Adapter à votre CPU
os.environ['MKL_NUM_THREADS'] = '16'

import numpy as np

# NumPy utilise automatiquement BLAS multi-thread
documents = np.random.randn(10_000_000, 768).astype(np.float32)
query = np.random.randn(768).astype(np.float32)

# Normaliser
query_norm = query / np.linalg.norm(query)
docs_norm = documents / np.linalg.norm(documents, axis=1, keepdims=True)

import time
start = time.time()
similarities = np.dot(docs_norm, query_norm)
elapsed = time.time() - start

print(f"10M similarités en {elapsed:.2f}s sur CPU 16 cores")
print(f"Débit : {10_000_000 / elapsed / 1000:.0f}K comparaisons/seconde")

# Résultats typiques :
# - CPU moderne (AMD Ryzen 9, Intel i9) : 3-5s → 2-3M comparaisons/sec
# - Serveur (Xeon Gold 48 cores) : 1-2s → 5-10M comparaisons/sec
```

Astuces supplémentaires

- **float32 vs float64** : Utiliser float32 (moitié mémoire, 2x plus rapide, précision suffisante)
- **Contigüts arrays** : np.ascontiguousarray() pour optimiser accès mémoire
- **In-place ops** : Normaliser in-place pour éviter copies : documents /= norms
- **Batch queries** : Si vous avez K queries, calculer (K, 768) @ (768, N) = (K, N) en une opération

Utilisation du GPU

Pour des datasets de millions à milliards de vecteurs, le GPU offre des accélérations massives (10-100x vs CPU).

PyTorch GPU : Implémentation optimisée

```
import torch
import torch.nn.functional as F
import time

device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print(f"Device : {device}")

# Générer dataset massif
num_docs = 50_000_000  # 50M documents
dim = 768

# Charger par chunks pour ne pas saturer GPU memory
def gpu_search_chunked(query, documents, chunk_size=1_000_000):
    """
    Recherche GPU avec chunking pour gros datasets.
    """
    query_gpu = torch.from_numpy(query).to(device)
    query_norm = F.normalize(query_gpu.unsqueeze(0), p=2, dim=1)

    all_scores = []
    num_chunks = (len(documents) + chunk_size - 1) // chunk_size

    for i in range(num_chunks):
        start_idx = i * chunk_size
        end_idx = min(start_idx + chunk_size, len(documents))
        chunk = documents[start_idx:end_idx]

        # Transférer chunk sur GPU
        chunk_gpu = torch.from_numpy(chunk).to(device)
        chunk_norm = F.normalize(chunk_gpu, p=2, dim=1)

        # Calculer similarités : (chunk_size, 768) @ (768, 1) = (chunk_size, 1)
        scores = torch.mm(chunk_norm, query_norm.T).squeeze()
        all_scores.append(scores.cpu())

        # Libérer mémoire GPU
        del chunk_gpu, chunk_norm, scores
        torch.cuda.empty_cache()

    return torch.cat(all_scores)

# Test avec dataset réel
if torch.cuda.is_available():
    import numpy as np

    print(f"\nTest : 50M documents, 768D")
    documents = np.random.randn(num_docs, dim).astype(np.float32)
    query = np.random.randn(dim).astype(np.float32)

    start = time.time()
    similarities = gpu_search_chunked(query, documents, chunk_size=1_000_000)
    elapsed = time.time() - start

    print(f"Temps GPU : {elapsed:.2f}s")
    print(f"Débit : {num_docs / elapsed / 1_000_000:.1f}M comparaisons/sec")

    # Top-10
    top_k = 10
    top_values, top_indices = torch.topk(similarities, k=top_k)
    print(f"\nTop-{top_k} documents : {top_indices.tolist()}")
    print(f"Scores : {top_values.tolist()}")
```

```
# Résultats typiques :
# - NVIDIA V100 : 50M docs en ~3-5s → 10-15M/sec
# - NVIDIA A100 : 50M docs en ~1-2s → 25-50M/sec
# - RTX 4090 : 50M docs en ~2-3s → 15-25M/sec
```

FAISS GPU : Le plus rapide pour ultra-large scale

```
import faiss
import numpy as np
import time

# Dataset
dim = 768
num_docs = 100_000_000  # 100M documents

print(f"Construction index FAISS GPU pour {num_docs} documents...")

# 1. Créer index sur GPU
res = faiss.StandardGpuResources()  # Ressources GPU
index_flat = faiss.IndexFlatIP(dim)  # Inner Product (= cosinus si normalisé)
index_gpu = faiss.index_cpu_to_gpu(res, 0, index_flat)  # GPU 0

# 2. Générer et ajouter documents par batches (mémoire limitée)
batch_size = 1_000_000
for i in range(0, num_docs, batch_size):
    batch = np.random.randn(min(batch_size, num_docs - i), dim).astype(np.float32)
    # Normaliser
    faiss.normalize_L2(batch)
    index_gpu.add(batch)
    if i % 10_000_000 == 0:
        print(f"  Indexé {i} documents...")

print(f"Index contient {index_gpu.ntotal} vecteurs\n")

# 3. Recherche ultra-rapide
query = np.random.randn(1, dim).astype(np.float32)
faiss.normalize_L2(query)

k = 100  # Top-100
start = time.time()
similarities, indices = index_gpu.search(query, k)
elapsed = time.time() - start

print(f"Recherche top-{k} parmi {num_docs} docs : {elapsed*1000:.1f}ms")
print(f"Top-10 indices : {indices[0][:10]}")
print(f"Top-10 scores : {similarities[0][:10]}")

# Résultats typiques :
# - 100M docs, GPU V100 : ~20-50ms pour top-100
# - 1B docs, 4x A100 : ~100-200ms pour top-100
# → FAISS GPU est 100-1000x plus rapide que CPU pour ultra-large scale
```

Quand investir dans GPU ?

Points d'attention

- **>10M documents** : ROI positif, latence divisée par 10-50x
- **Haute fréquence** : >100 requêtes/sec, GPU amortit son coût

- **Latence critique** : Besoin de <50ms de réponse
- **Éviter si** : <1M docs (CPU suffit), budget limité, expertise GPU manquante

Approximation avec LSH (Locality Sensitive Hashing)

Pour des datasets de **milliards de vecteurs**, même le GPU devient lent. LSH permet des recherches approximatives en temps sous-linéaire O(log n).

Principe de LSH
Intuition

LSH crée des **fonctions de hachage** telles que des vecteurs similaires ont une **haute probabilité** d'être hashés dans le même bucket. Plutôt que de comparer contre tous les N vecteurs, on ne compare que contre les ~√N vecteurs du même bucket.

Implémentation avec Annoy (Spotify) :

```
from annoy import AnnoyIndex
import numpy as np
import time

dim = 768
num_docs = 10_000_000  # 10M documents

print(f"Construction index Annoy pour {num_docs} documents...")

# 1. Créer index Annoy
index = AnnoyIndex(dim, 'angular')  # 'angular' = cosinus

# 2. Ajouter vecteurs
np.random.seed(42)
for i in range(num_docs):
    vector = np.random.randn(dim).astype(np.float32)
    index.add_item(i, vector)

    if i % 1_000_000 == 0 and i > 0:
        print(f"  Ajouté {i} vecteurs...")

# 3. Construire index (phase coûteuse, une fois)
num_trees = 100  # Plus d'arbres = meilleure précision mais plus lent
print(f"\nConstruction de {num_trees} arbres...")
start = time.time()
index.build(num_trees)
build_time = time.time() - start
print(f"Index construit en {build_time:.1f}s")

# 4. Sauvegarder index (persistance)
index.save('annoy_index.ann')
print(f"Index sauvegardé ({index.get_n_items()} vecteurs)\n")

# 5. Charger et rechercher (phase rapide, répétée)
index_loaded = AnnoyIndex(dim, 'angular')
index_loaded.load('annoy_index.ann')

query = np.random.randn(dim).astype(np.float32)
k = 10

start = time.time()
nearest_indices = index_loaded.get_nns_by_vector(query, k, include_distances=True)
elapsed = time.time() - start

indices, distances = nearest_indices
print(f"Recherche top-{k} : {elapsed*1000:.2f}ms")
print(f"Indices : {indices}")
print(f"Distances angulaires : {distances}")

# Convertir distance angulaire en similarité cosinus
# distance_angular = arccos(cosine) / pi
# donc cosine = cos(distance_angular * pi)
similarities = [np.cos(d * np.pi) for d in distances]
print(f"Similarités cosinus : {similarities}")

# Résultats typiques :
# - 10M docs : recherche en 1-5ms (vs 500-1000ms exact)
# - Recall : 95-99% (capture 95-99% des vrais top-k)
# - Tradeoff : 100-500x speedup, 1-5% perte précision
```

Comparaison des méthodes ANN (Approximate Nearest Neighbor)

Méthode	Bibliothèque	Latence (10M docs)	Recall	Cas d'usage
Exact (Brute-force)	NumPy, Scikit-learn	500-2000ms	100%	Petits datasets (<1M)
LSH (Annoy)	Spotify Annoy	1-5ms	95-98%	Production, balance speed/recall
HNSW	Qdrant, Milvus, hnswlib	2-10ms	98-99.5%	Meilleur recall, standard 2025
IVF	FAISS	5-20ms	90-95%	Très large scale (1B+)
PQ (compression)	FAISS	10-50ms	85-92%	Mémoire limitée, compromis

Attention au recall

Un recall de 95% signifie que 5% du temps, le vrai meilleur résultat n'est PAS dans votre top-k. Pour des applications critiques (médical, finance), validez que ce tradeoff est acceptable. Pour recherche web/e-commerce, 95-98% recall est largement suffisant.

Limites et alternatives

Sensibilité à la dimension

En haute dimension (768D, 1536D), la similarité cosinus peut souffrir de la **malédiction de la dimensionnalité** (curse of dimensionality).

Phénomène observé
Concentration des distances

En très haute dimension (10K+), TOUS les vecteurs aléatoires tendent à être presque orthogonaux (cos ≈ 0). Les similarités se concentrent dans une plage étroite [0.7, 0.9], rendant difficile la discrimination.

Expérience numérique : Pour approfondir, consultez CNIL Autorite AI Act : Premiers Pas Reglementaires.

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

def analyze_dimensionality_effect(dims):
    """Mesure l'écart-type des similarités selon dimension."""
    results = {}

    for dim in dims:
        # Générer 1000 vecteurs aléatoires
        vectors = np.random.randn(1000, dim)

        # Calculer toutes les similarités par paires
        sim_matrix = cosine_similarity(vectors)

        # Exclure diagonale (similarité avec soi-même = 1)
        similarities = sim_matrix[np.triu_indices(1000, k=1)]

        results[dim] = {
            'mean': np.mean(similarities),
            'std': np.std(similarities),
            'min': np.min(similarities),
            'max': np.max(similarities)
        }

    return results

# Test avec différentes dimensions
dimensions = [2, 10, 100, 768, 1536, 10000]
results = analyze_dimensionality_effect(dimensions)

print("Impact de la dimensionnalité sur similarité cosinus :\n")
for dim, stats in results.items():
    print(f"Dimension {dim:5d} : mean={stats['mean']:.3f}, std={stats['std']:.3f},
range=[{stats['min']:.3f}, {stats['max']:.3f}]")

# Sortie typique :
# Dimension     2 : mean=0.003, std=0.485, range=[-0.98, 0.97]  # Très varié
# Dimension    10 : mean=0.001, std=0.311, range=[-0.78, 0.81]
# Dimension   100 : mean=0.000, std=0.099, range=[-0.31, 0.35]
# Dimension   768 : mean=0.000, std=0.036, range=[-0.12, 0.13]  # Concentré
# Dimension  1536 : mean=0.000, std=0.025, range=[-0.09, 0.09]
# Dimension 10000 : mean=0.000, std=0.010, range=[-0.03, 0.04]  # Très concentré
```

Implications pratiques

- **Embeddings modernes (768-1536D)** : Encore discriminants car entraînés sur données réelles (pas aléatoires). Similarités typiques : [0.3, 0.95]
- **Solutions** :
  - Réduction de dimension : PCA, UMAP (768D → 128D) avec perte accept able 2-5%
  - Utiliser des embeddings de dimension raisonnable (384-768D suffit souvent)
  - Modèles récents (Matryoshka embeddings) permettent truncation flexible

Limitation avec les vecteurs creux

Pour des vecteurs très **creux** (sparse, 95%+ de zéros), comme TF-IDF avec grand vocabulaire, la similarité cosinus peut produire des faux positifs.

Problème

```
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Exemple : Vecteurs TF-IDF avec vocabulaire 50K
vocab_size = 50000

# Document 1 : contient mots [42, 153, 789]
doc1 = np.zeros(vocab_size)
doc1[[42, 153, 789]] = [0.5, 0.3, 0.8]

# Document 2 : contient mots [42, 156, 790] - UN SEUL mot commun
doc2 = np.zeros(vocab_size)
doc2[[42, 156, 790]] = [0.6, 0.4, 0.7]

# Similarité
sim = cosine_similarity([doc1], [doc2])[0, 0]
print(f"Similarité cosinus : {sim:.3f}")
# Output : ~0.65 - semble similaire alors qu'un seul mot en commun sur 3 !

# Pourquoi ? Cosinus ignore la magnitude. Seules les 3 dimensions non nulles comptent.
```

Solutions alternatives pour vecteurs creux

1. **Similarité de Jaccard** : Meilleure pour présence/absence

   ```
   def jaccard_sparse(vec1, vec2):
       """Jaccard pour vecteurs sparse."""
       intersection = np.sum((vec1 > 0) & (vec2 > 0))
       union = np.sum((vec1 > 0) | (vec2 > 0))
       return intersection / union if union > 0 else 0

   jaccard = jaccard_sparse(doc1, doc2)
   print(f"Jaccard : {jaccard:.3f}")  # 0.20 - plus réaliste (1/5 overlap)
   ```

2. **BM25** : Standard pour recherche full-text, pondère mieux les termes rares
3. **Embeddings denses** : BERT, GPT éliminent la crépité (768D denses vs 50K creux)

Tendance 2024-2025

Les systèmes modernes utilisent **hybrid search** : embeddings denses (cosinus) + sparse BM25. Exemple : Qdrant hybrid mode, Pinecone sparse-dense vectors. Combine avantages sémantiques (dense) et mots-clés exacts (sparse).

Alternatives modernes (attention mechanisms)

Les **mécanismes d'attention** des transformers (BERT, GPT) généralisent la similarité cosinus avec des pondérations apprises.

Attention vs Cosinus

Similarité Cosinus (fixe)

$$\text{score}(q, k) = (q \cdot k) / (||q|| \times ||k||)$$

Attention (apprise)

$$\text{score}(q, k) = \text{softmax}((W_e q) \cdot (W_k k) / \sqrt{d_k})$$

où W_e, W_k sont des matrices apprises par entraînement

Avantages de l'attention

- **Pondérations adaptées** : Apprend quelles dimensions sont importantes pour la tâche
- **Multi-head** : Capture plusieurs types de relations simultanément
- **Contexte dynamique** : Score dépend du contexte de la phrase entière

Quand utiliser quoi ?

Méthode	Cas d'usage	Avantages	Inconvénients
Cosinus	Recherche vectorielle, RAG, recommandation	Rapide, simple, interprétable, pas d'entraînement	Pas adapté à la tâche spécifique
Attention (transformers)	Génération texte, traduction, compreh	Pondérations optimales, capture contexte	Lent ($O(n^2)$), nécessite entraînement
Cross-encoders	Reranking précis après retrieval	Précision maximale (98%+)	Très lent, pas scalable >1K candidats

Architecture typique 2025 :

1. **Étape 1 : Retrieval** - Similarité cosinus sur 10M docs → top-100 (50ms)
2. **Étape 2 : Reranking** - Cross-encoder sur top-100 → top-10 (200ms)
3. **Étape 3 : Generation** - LLM avec top-10 contexte → réponse (1-3s)

Quand ne pas utiliser la similarité cosinus

La similarité cosinus n'est pas universelle. Voici les cas où d'autres métriques sont préférables :

1. Magnitude importante
Problème : Cosinus ignore la longueur des vecteurs, ce qui peut être problématique.

```
# Exemple : Comptage de mots
doc1 = [10, 20, 30]  # Document court (60 mots)
doc2 = [100, 200, 300]  # Document long (600 mots), même proportions

# Cosinus = 1.0 (identiques en direction)
# Mais doc2 a 10x plus d'occurrences !

# Solution : Utiliser distance euclidienne ou Manhattan
```

2. Données catégorielles (ensembles)
Utilisez Jaccard : Tags, catégories, achats binaires.

```
user1_tags = {"python", "machine-learning", "data-science"}
user2_tags = {"python", "web-development", "django"}

# Jaccard = 1/5 = 0.20 (1 commun / 5 uniques)
# Cosinus sur vecteurs binaires donnerait un score différent, moins intuitif
```

3. Séries temporelles avec alignement
Utilisez DTW (Dynamic Time Warping) : Capturer patterns décalés dans le temps.

4. Features numériques hétérogènes
Problème : Mélanger âge (0-100), salaire (0-200K), score (0-1) sans normalisation.

```
person1 = [25, 50000, 0.8]  # âge, salaire, score
person2 = [30, 55000, 0.85]

# Cosinus biaisé par salaire (grande magnitude)
# Solution : Standardiser d'abord (z-score) ou utiliser distance Mahalanobis
```

5. Haute précision requise sur petit dataset
Utilisez exact match ou cross-encoder : Pour <1K comparaisons, coût négligeable.

Règle d'or

Cosinus est optimal pour **embeddings denses appris** (BERT, GPT, CLIP) représentant des concepts sémantiques. Pour autres types de données, évaluer alternatives selon le domaine.

Sources et références : ArXiv IA · Hugging Face Papers

Questions fréquentes

Pourquoi la similarité cosinus ignore-t-elle la magnitude des vecteurs ?

C'est par design : la formule divise par le produit des normes (||A|| × ||B||), ce qui normalise les vecteurs. Cette propriété est précieuse en NLP car la **longueur d'un document** (nombre de mots) ne devrait pas affecter sa similarité sémantique avec un autre. Un article de 500 mots et un article de 5000 mots sur le même sujet auront des embeddings pointant dans la même direction, donc une similarité cosinus élevée, même si leurs vecteurs TF-IDF bruts ont des magnitudes très différentes.

Contre-exemple : La distance euclidienne considère la magnitude. Deux documents identiques en contenu mais l'un 2x plus long auront une distance euclidienne non nulle, ce qui est contre-intuitif pour la similarité sémantique.

La similarité cosinus fonctionne-t-elle avec des vecteurs de dimensions différentes ?

Non. Le produit scalaire $A \cdot B = \Sigma A_i \times B_i$ nécessite que A et B aient la **même dimension**. Tenter de calculer la similarité entre un vecteur 768D et un 1536D produira une erreur.

Solutions :

- **Utiliser le même modèle d'embeddings** : text-embedding-ada-002 (1536D) pour tous les documents
- **Padding** : Compléter le vecteur court avec des zéros (rarement utilisé, peut biaiser)
- **Projection** : Réduire dimension du grand vecteur via PCA, mais perte d'information

- **Modèles Matryoshka** : Nouveaux embeddings permettant truncation (1536D → 768D → 384D) avec perte minimale

Comment gérer les valeurs négatives dans les vecteurs ?

La similarité cosinus **accepte parfaitement les valeurs négatives**. La formule fonctionne pour n'importe quels réels (positifs, négatifs, zéros).

Exemples :

- **Embeddings BERT/GPT** : Contiennent souvent des valeurs négatives (ex: [-0.3, 0.8, -0.1, 0.5, ...]). C'est normal et géré nativement.
- **Données centrées** : Si vous soustrayez la moyenne (standardisation), vous obtiendrez des négatifs. Cosinus reste valide.
- **Analyse de sentiment** : Embeddings de "heureux" peuvent être opposés à "triste" avec des signes inversés.

Aucune transformation requise : Ne convertissez JAMAIS les négatifs en positifs (ex: valeur absolue), cela détruirait l'information directionnelle.

Quelle est la complexité algorithmique du calcul de similarité cosinus ?

Complexité temporelle :

- **1 paire de vecteurs** : O(d) où d = dimension
  - Produit scalaire : d multiplications + (d-1) additions = O(d)
  - Normes : 2 × O(d) pour calculer ||A|| et ||B||
  - Total : O(d)
- **1 query vs N documents** : O(N × d)
  - N produits scalaires de dimension d
  - Si vecteurs pré-normalisés : O(N × d) exact
- **Matrice de similarité (N × N)** : O(N^2 × d)
  - Calculer toutes les paires
  - Prohibitif pour N > 10K (100M comparaisons pour 10K docs)

Complexité spatiale : O(d) pour stocker 1 vecteur, O(N × d) pour N documents.

Avec index ANN (HNSW, LSH) : Réduit à O(log N × d) en moyenne pour 1 query, sacrifiant 1-5% de précision.

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

Peut-on utiliser la similarité cosinus pour des images ?

Oui, absolument. C'est même une application majeure de la similarité cosinus en computer vision.

Méthode :

1. **Extraire embeddings** : Utiliser un modèle CNN (ResNet, EfficientNet) ou transformer (CLIP, ViT) pour convertir image en vecteur dense (ex: 512D, 768D, 2048D)
2. **Comparer embeddings** : Calculer similarité cosinus entre vecteurs d'images

Exemple avec CLIP :

```
from sentence_transformers import SentenceTransformer
from PIL import Image
import numpy as np
from sklearn.metrics.pairwise import cosine_similarity

# Modèle CLIP multimodal (images + texte)
model = SentenceTransformer('clip-ViT-B-32')

# Charger images
img1 = Image.open('chat.jpg')
img2 = Image.open('chien.jpg')
img3 = Image.open('autre_chat.jpg')

# Générer embeddings
emb1 = model.encode(img1)
emb2 = model.encode(img2)
emb3 = model.encode(img3)

# Comparaisons
sim_chat_chien = cosine_similarity([emb1], [emb2])[0, 0]
sim_chat_chat = cosine_similarity([emb1], [emb3])[0, 0]

print(f"Similarité chat-chien : {sim_chat_chien:.3f}")  # Ex: 0.65
print(f"Similarité chat-chat : {sim_chat_chat:.3f}")    # Ex: 0.92
```

Applications réelles :

- **Recherche d'images** : Google Images, Pinterest Lens
- **Détection de duplicatas** : Trouver images quasi-identiques
- **Recommandation visuelle** : "Produits similaires" en e-commerce
- **Vérification faciale** : Comparer embeddings de visages (FaceNet, ArcFace)

> **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