



# Recherche d'information

## Méthodes

**Enzo Doyen**

2025 - LGC6KM43 - M2

# **Plan**

**I.** Prétraitements

**II.** Modèles de récupération de documents

**III.** Modèles de récupération de documents : recherche par mots-clés

**IV.** Modèles de récupération de documents : recherche dense

**V.** Modèles de récupération de documents : recherche hybride



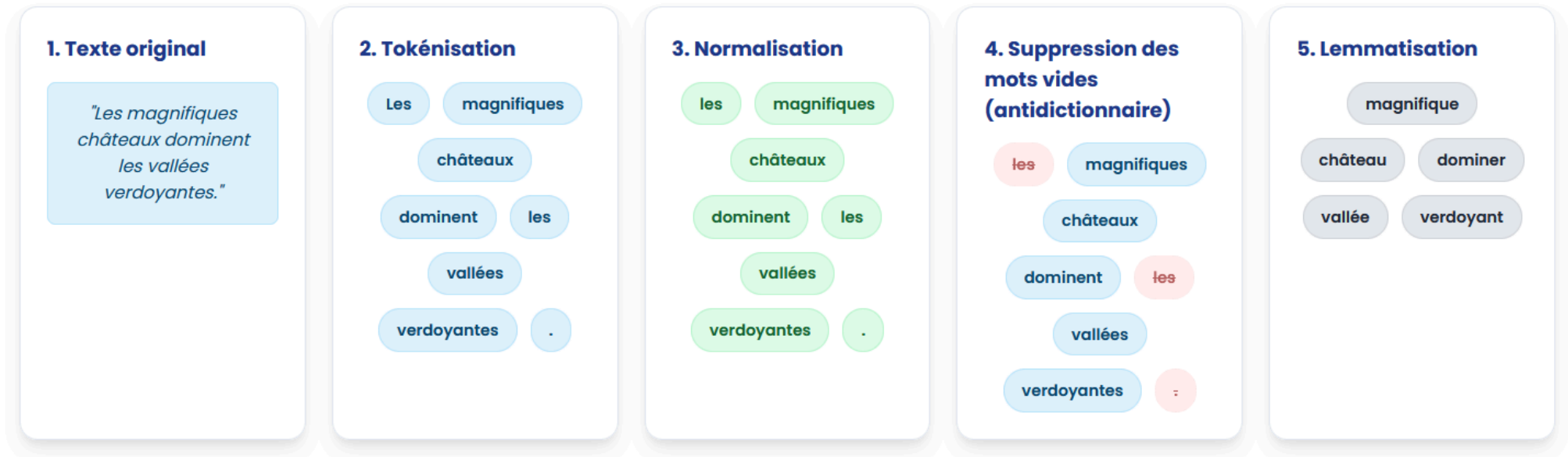
# I. Prétraitements

# Prétraitement des données textuelles

Avant d'indexer des documents pour de la recherche d'information, et pour améliorer l'efficacité de la recherche, il est souvent nécessaire de les traiter au préalable (en d'autres termes, opérer des « prétraitements »).

L'objectif est de passer d'un **texte brut** à un **texte lexicalement et sémantiquement riche**.

# Prétraitement des données textuelles




Étapes de prétraitement des données textuelles : tokénisation, normalisation, suppression des mots vides (*stopwords*) et lemmatisation

# Tokénisation

Original	Tokénisation simple	Tokenisation avancée
l'école	["l", " ' ", "école"]	["l'", "école"]
n'a	["n", " ' ", "a"]	["n'", "a"]
aujourd'hui	["aujourd", " ' ", "hui"]	["aujourd'hui"]
arrière-grand-père	["arrière", "- ", "grand", "- ", "père"]	["arrière-grand-père"]
Bourg-en-Bresse	["Bourg", "- ", "en", "- ", "Bresse"]	["Bourg-en-Bresse"]
jean.d@email.fr	["jean", ".", "d", "@", "email", ".", "fr"]	["jean.d@email.fr"]
12€50	["12", "€", "50"]	["12€50"]

# Tokénisation avec *NLTK*

```
1  import nltk
2  from nltk.tokenize import word_tokenize
3  nltk.download("punkt")
4  text = "Les chiens ont l'habitude d'aboyer tous les matins."
5  tokens = word_tokenize(text, language="french")
```

 Python

Sortie: ['Les', 'chiens', 'ont', "l'habitude", 'd', 'aboyer', 'tous', 'les', 'matins', '.']

# Tokénisation avec *spaCy*

```
1 import spacy
```

 Python

```
2
```

```
3 nlp = spacy.load("fr_core_news_sm")
```

```
4 text = "Les chiens ont l'habitude d'aboyer tous les  
matins."
```

```
5 doc = nlp(text)
```

```
6 tokens = [token.text for token in doc]
```



# ***spaCy* et modèles pour le français**

Dans *spaCy*, modèles de différentes tailles disponibles pour le français, avec différents corpus d'entraînement et différentes architectures.

Un modèle plus grand donne généralement de meilleures performances, mais est aussi plus lourd et plus lent à charger/utiliser.

---

**<https://spacy.io/models/fr>**

Modèles disponibles : fr\_core\_news\_sm (petit), fr\_core\_news\_md (moyen), fr\_core\_news\_lg (grand), fr\_core\_news\_trf (Transformer)

# Normalisation

La **normalisation** consiste à standardiser le texte en éliminant les variations, pour ainsi faciliter le traitement ultérieur.

# Normalisation

La **normalisation** consiste à standardiser le texte en éliminant les variations, pour ainsi faciliter le traitement ultérieur.

```
~ ➔ cat zoe.js
console.log('Is Zoë, Zoë?')
console.log('Zoë' === 'Zoë')
~ ➔ node zoe.js
Is Zoë, Zoë?
false
~ ➔
```

Source : <https://withblue.ink/2019/03/11/why-you-need-to-normalize-unicode-strings.html> 

# Normalisation

La **normalisation** consiste à standardiser le texte en éliminant les variations, pour ainsi faciliter le traitement ultérieur.

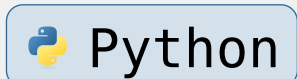
Par exemple, une normalisation voulue en français pour le mot « œuf » :

œuf = oeuf = Œuf = OEUF = ŒUF

# Normalisation : casse

La normalisation au niveau de la casse se fait généralement en convertissant tout le texte en minuscules :

```
1 _ = "OEUF".lower()
```



# Normalisation : espaces

```
1 _ = " oeuf ".strip()
```

 Python

```
2 _ = " oeuf".lstrip()
```

```
3 _ = "oeuf ".rstrip()
```

```
4 _ = " ".join(" un oeuf sur la table".split())
```

# Normalisation : diacritiques et ligatures

Source		NFD		NFC		NFKD		NFKC
fi FB01	:	fi FB01		fi FB01		f i 0066 0069		f i 0066 0069
2 <sup>5</sup> 0032 2075	:	2 5 0032 2075		2 5 0032 2075		2 5 0032 0035		2 5 0032 0035
fi 1E9B 0323	:	f ̇ ̇ 017F 0323 0307		fi 1E9B 0323		s ̇ ̇ 0073 0323 0307		š 1E69

Source : <https://www.unicode.org/reports/tr15/>

# Normalisation : diacritiques et ligatures

```
1 import unicodedata
2 unicodedata.normalize('NFKC', "IJ") # 'IJ'
3 unicodedata.normalize('NFKD', "½") # '1/2'
```

 Python

```
1 from unicode import unicode
2 unicode("épée", "utf-8") # 'epee'
```

 Python

## Attention

Certaines ligatures, considérées comme des lettres à part entières, ne sont pas décomposables directement (p. ex. « œ », « æ »).



# Suppression des mots vides (*stopwords*)

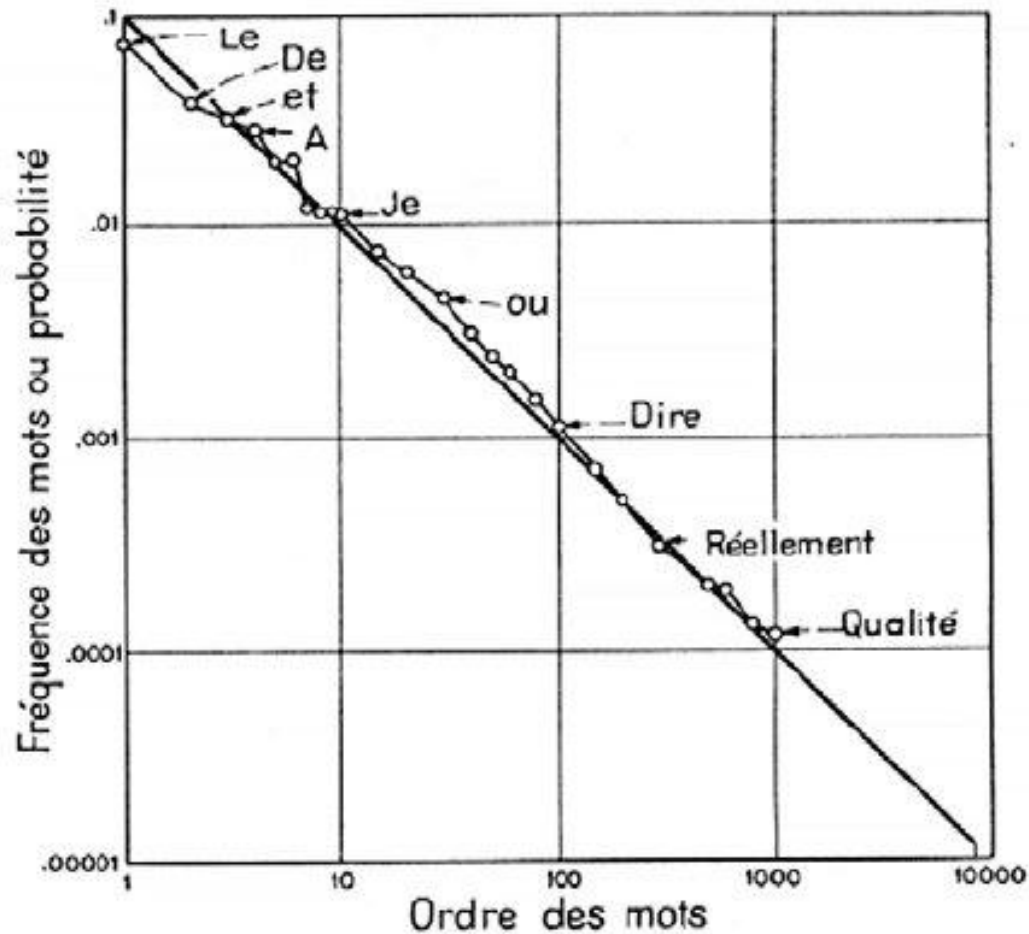
Les mots vides sont des mots qui n'apportent pas de valeur sémantique à un document, et qui peuvent être supprimés pour alléger l'indexation.

Ils incluent bien souvent les mots grammaticaux, par exemple en français :  
« le », « la », « et », « à », « de », etc.

# Suppression des mots vides (*stopwords*)

Lien avec la **loi de Zipf** (Zipf, 1932) : observation empirique selon laquelle, dans un corpus de texte, la fréquence d'un mot est inversement proportionnelle à son rang dans la liste des mots par fréquence.

Un mot de rang 1 (le plus fréquent) apparaîtra deux fois plus souvent qu'un mot de rang 2, trois fois plus souvent qu'un mot de rang 3, etc.



Représentation graphique de la loi de Zipf. Source : Mandelbrot (1965)

# Suppression des mots vides (*stopwords*) avec *NLTK*

```
1 import nltk
2 from nltk.corpus import stopwords
3
4 nltk.download("stopwords")
5 english_sw = stopwords.words('english')
6 french_sw = stopwords.words('french')
```



# Suppression des mots vides (*stopwords*) avec *spaCy*

```
1 import spacy
2 from spacy.lang.en import stop_words
3
4 nlp = spacy.load('en_core_web_sm')
5
6 stop_words = stop_words.STOP_WORDS
```



# Suppression des mots vides (*stopwords*)

Liste en français pour spaCy : [https://github.com/explosion/spaCy/blob/master/spacy/lang/fr/stop\\_words.py](https://github.com/explosion/spaCy/blob/master/spacy/lang/fr/stop_words.py) 

```
STOP_WORDS = set(
    """
a à â abord afin ah ai aie ainsi ait allaient allons
alors anterieur anterieure anterieures antérieur antérieure antérieures
apres après as assez attendu au
aupres auquel aura auraient aurait auront
aussi autre autrement autres autrui aux auxquelles auxquels avaient
avais avait avant avec avoir avons ayant

bas basee bat
```

Google

site:github.com "stopwords" "french"

All

Images

Videos

Short videos

Web

News

Books

More ▾

Words

List

Dictionary



GitHub

<https://github.com> › [stopwords-iso](#) › [stopwords-fr](#) ⋮

### stopwords-iso/stopwords-fr: French stopwords collection

The most comprehensive collection of **stopwords** for the **french** language. A multiple language collection is also available.



GitHub

<https://github.com> › [gillesbastin](#) › [french\\_stopwords](#) ⋮

### gillesbastin/french\_stopwords: A list of stopwords to be ...

This repository contains a carefully curated list of **stopwords** for preprocessing **French** texts, specifically designed for text mining tasks ( `FRENCH_STOPWORDS` ).



GitHub

<https://github.com> › [stopwords-json](#) ⋮

### Stopwords for 50 languages in JSON format

# Lemmatisation

La lemmatisation est le processus de réduction des mots à leur forme canonique (leur « lemme »), ce qui permet de regrouper les différentes formes d'un mot.



# Lemmatisation avec *spaCy*

```
1 import spacy
2
3 nlp = spacy.load("fr_core_news_sm")
4 text = "Les chiens ont l'habitude d'aboyer tous les matins."
5 lemmas = [token.lemma_ for token in nlp(text)]
```



Sortie: ['le', 'chien', 'avoir', 'le', 'habitude', 'de', 'aboyer', 'tout', 'le', 'matin', '.']

# Lemmatisation avec *treetaggerwrapper*

```
1 import treetaggerwrapper
```

 Python

```
2
```

```
3 tagger = treetaggerwrapper.TreeTagger(TAGLANG='fr')
```

```
4 tags =  
treetaggerwrapper.make_tags(tagger.tag_text(text))
```

```
5
```

```
6 lemmas = [t.lemma for t in tags if t.lemma is not  
None]
```

# Autres types de prétraitements

- ♦ corrections de fautes d'orthographe ;
- ♦ correction phonétique ;
- ♦ *chunking* (découpage en phrases ou en segments, notamment pour les systèmes de *RAG*) ;
- ♦ suppression d'informations sensibles (par exemple, pour des raisons de confidentialité) ;
- ♦ ...



## **II. Modèles de récupération de documents**

# Méthodes de récupération de documents

Plusieurs méthodes peuvent être utilisées pour récupérer des documents pertinents à partir d'une requête formulée en langage naturel :

- ♦ **recherche par mots-clés** : correspondance exacte (ou quasi exacte) entre les termes de la requête et ceux présents dans les documents. La pertinence peut être estimée grâce à des **modèles épars** comme TF-IDF ou BM25, qui pondèrent l'importance des mots en fonction de leur fréquence dans le document et dans la collection.
- ♦ **recherche sémantique/dense** : elle vise à interpréter le sens de la requête afin d'identifier des documents pertinents, même si les termes employés diffèrent. Cette approche s'appuie sur des **modèles denses** qui exploitent des plongements vectoriels (au niveau des mots ou de la phrase) pour mesurer la similarité sémantique entre la requête et les documents.



# **III. Modèles de récupération de documents : recherche par mots-clés**

# Recherche par mots-clés : TF-IDF

**TF-IDF** : *term frequency-inverse document frequency* (**Sparck Jones, 1972**)

Méthode de pondération des termes dans un document, qui permet de mesurer l'importance d'un mot dans un document par rapport à un corpus.

Repose sur deux mesures : TF (*term frequency*) et IDF (*inverse document frequency*).

# Recherche par mots-clés : TF-IDF

**TF-IDF** : *term frequency-inverse document frequency* (**Sparck Jones, 1972**)

Méthode de pondération des termes dans un document, qui permet de mesurer l'importance d'un mot dans un document par rapport à un corpus.

Repose sur deux mesures : TF (*term frequency*) et IDF (*inverse document frequency*).

**TF** correspond au nombre d'occurrences d'un mot dans **un seul document**, tandis que **IDF** mesure l'importance d'un mot dans l'**ensemble du corpus**.



# Recherche par mots-clés : TF-IDF

**TF** correspond au nombre d'occurrences d'un mot dans **un seul document**, tandis que **IDF** mesure l'importance d'un mot dans l'**ensemble du corpus**.

Intuition : si un mot apparaît beaucoup dans un document, il est probablement important et pertinent (capturé par **TF**). Cependant, si le même mot apparaît dans de nombreux documents du corpus, il est surement moins pertinent pour la recherche (capturé par **IDF**).

# Recherche par mots-clés : TF-IDF

Le score TF-IDF d'un terme  $t$  dans un document  $d$  par rapport à un corpus  $D$  se calcule alors comme suit :

$$\text{TF-IDF}(t, d, D) = \text{TF}(t, d) \times \text{IDF}(t, D)$$

# Recherche par mots-clés : TF-IDF | Calcul de TF

$$\text{TF}(t, d) = \frac{\text{occurrences du terme } t \text{ dans le document } d}{\text{nombre total de termes dans le document } d}$$

Ou plus formellement :

$$\text{TF}(t, d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

# Recherche par mots-clés : TF-IDF | Calcul d'IDF

Pour rappel, IDF mesure l'importance d'un mot dans l'ensemble du corpus.

$$\text{IDF}(t, D) = \log \left( \frac{\text{nombre total de documents dans le corpus } D}{\text{nombre de documents contenant le terme } t} \right)$$

Le logarithme est utilisé pour que les valeurs ne soient pas trop grandes (ce qui serait le cas avec un très grand corpus), et ainsi ne pas donner trop de poids aux mots très rares.

# Recherche par mots-clés : TF-IDF

Plus le score TF-IDF d'un terme est élevé, plus ce terme est jugé important et représentatif du document dans lequel il apparaît, en comparaison avec les autres documents du corpus.

Les termes avec un TF-IDF élevé peuvent être utilisés comme « mots-clés » du document et servent au référencement et à la recherche de documents par terme.

# Recherche par mots-clés : TF-IDF

```
1 from sklearn.feature_extraction.text import  
  TfidfVectorizer  
2 from typing import List  
3  
4 corpus: List[str] = [] # liste de documents  
5 vectorizer = TfidfVectorizer()  
6 X = vectorizer.fit_transform(corpus) # shape:  
  (n_docs, n_features)
```



# Recherche par mots-clés : TF-IDF

Avec `TfidfVectorizer()`, possibilité de définir plusieurs options :

- ♦ `stop_words` : langue de la liste de mots à ignorer
- ♦ `ngram_range` : pour utiliser des n-grammes ; prend un tuple (`min_x`, `max_x`) où `min_x` et `max_x` sont des entiers définissant la taille minimale et maximale des n-grammes à considérer. Par exemple : (1, 2) = unigrammes et bigrammes ; (1, 1) = unigrammes uniquement. Par défaut : (1, 1).

Documentation : [https://scikit-learn.org/stable/modules/generated/sklearn.feature\\_extraction.text.TfidfVectorizer.html](https://scikit-learn.org/stable/modules/generated/sklearn.feature_extraction.text.TfidfVectorizer.html) 

# Récupération des résultats par requête et classement

Rappelons le problème de base de la recherche d'information : étant donné une requête de recherche  $q$ , on veut trouver les documents les plus pertinents dans un corpus. Ces documents doivent ensuite être **récupérés**, et **classés** par ordre de pertinence.



# Similarité cosinus

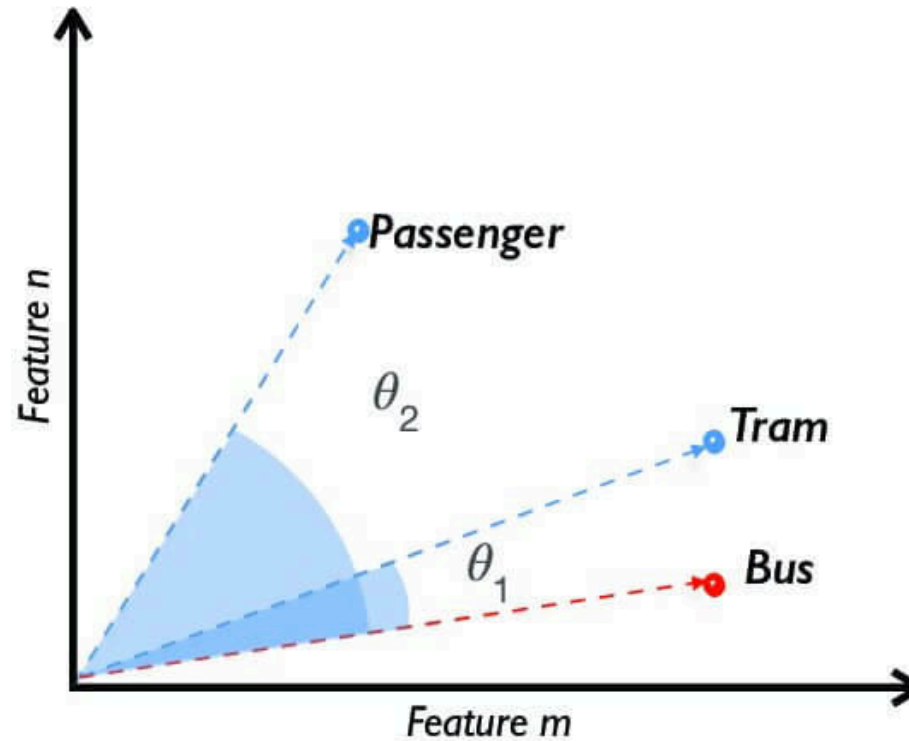
On peut transformer notre requête  $q$  en vecteur TF-IDF de la même manière que pour les documents du corpus, puis calculer la similarité entre ce vecteur de requête et les vecteurs des documents pour obtenir un classement des résultats (documents les plus similaires à  $q$  en premier).

Ce calcul de similarité peut être effectué à l'aide de la similarité cosinus, qui mesure l'angle entre deux vecteurs dans un espace vectoriel.

$$\text{sim}(a, b) = \cos(\theta) = \frac{A \cdot B}{\|A\| \|B\|}$$

où  $A \cdot B$  est le produit scalaire des vecteurs  $A$  et  $B$ , et  $\|A\|$  et  $\|B\|$  sont les normes (ou longueurs) des vecteurs.

# Similarité cosinus



Source : Kalwar et al. (2023)

# Similarité cosinus

Implémentation simple avec *scikit-learn* :

```
1 from sklearn.metrics.pairwise import  
  cosine_similarity  
2  
3 query_vec = vectorizer.transform([query])  
4 similarities = cosine_similarity(query_vec,  
  X).flatten()
```



# Récupération des documents

Ensuite, on peut récupérer les meilleurs  $n$  documents en triant la liste par ordre croissant et en récupérant les indices correspondants avec `np.argsort()` :

```
1 import numpy as np
2 top_n = np.argsort(similarities[::-1][:n])
```



# Exemple : np.argsort ( )

```
import numpy as np
arr = np.array([7, 2, 9, 1, 5])
indices = np.argsort(arr)
print(indices) # Output: [3 1 4 0 2]
```



# Recherche par mots-clés : BM25

Le score **Okapi BM25** améliore TF-IDF sur plusieurs points :

- ♦ **requête à plusieurs termes** : BM25 gère mieux les requêtes composées de plusieurs termes, en tenant compte de la pertinence de chaque terme dans le document.
- ♦ **saturation** : BM25 modélise un effet de saturation de la fréquence des termes. En d'autres mots : à quel point la répétition d'un même terme dans un document augmente-t-il la pertinence de ce dernier ?
- ♦ **normalisation de la longueur des documents** : BM25 permet de pénaliser les documents plus longs pour éviter qu'ils ne dominent les résultats simplement en raison de leur taille.

# Recherche par mots-clés : BM25

Pour un document  $d$  et une requête  $q$  :

$$\text{BM25}(d, q) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{\text{tf}(t, d)(k_1 + 1)}{\text{tf}(t, d) + k_1 \left(1 - b + b \frac{|d|}{\text{avgdl}}\right)}$$

# Recherche par mots-clés : BM25

Pour un document  $d$  et une requête  $q$  :

$$\text{BM25}(d, q) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{\text{tf}(t, d)(k_1 + 1)}{\text{tf}(t, d) + k_1 \left(1 - b + b \frac{|d|}{\text{avgdl}}\right)}$$



# Recherche par mots-clés : BM25

Pour un document  $d$  et une requête  $q$  :

$$\text{BM25}(d, q) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{\text{tf}(t, d)(k_1 + 1)}{\text{tf}(t, d) + k_1 \left(1 - b + b \frac{|d|}{\text{avgdl}}\right)}$$

# Recherche par mots-clés : BM25

La formule de l'**IDF** est légèrement modifiée par rapport à celle de TF-IDF :

$$\text{IDF}(t) = \log \left( \frac{N - n(t) + 0.5}{n(t) + 0.5} \right)$$

où  $N$  est le nombre de documents dans le corpus, et  $n(t)$  est le nombre de documents contenant le terme  $t$ .

**BM25** ajoute +0,5 pour empêcher la division par zéro (si le terme  $t$  n'apparaît jamais dans le corpus) et pour éviter que les termes très rares n'aient un poids trop élevé.

# Recherche par mots-clés : BM25

Pour un document  $d$  et une requête  $q$  :

$$\text{BM25}(d, q) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{\text{tf}(t, d)(k_1 + 1)}{\text{tf}(t, d) + k_1 \left(1 - b + b \frac{|d|}{\text{avgdl}}\right)}$$

somme des scores pour chaque terme

# Recherche par mots-clés : BM25

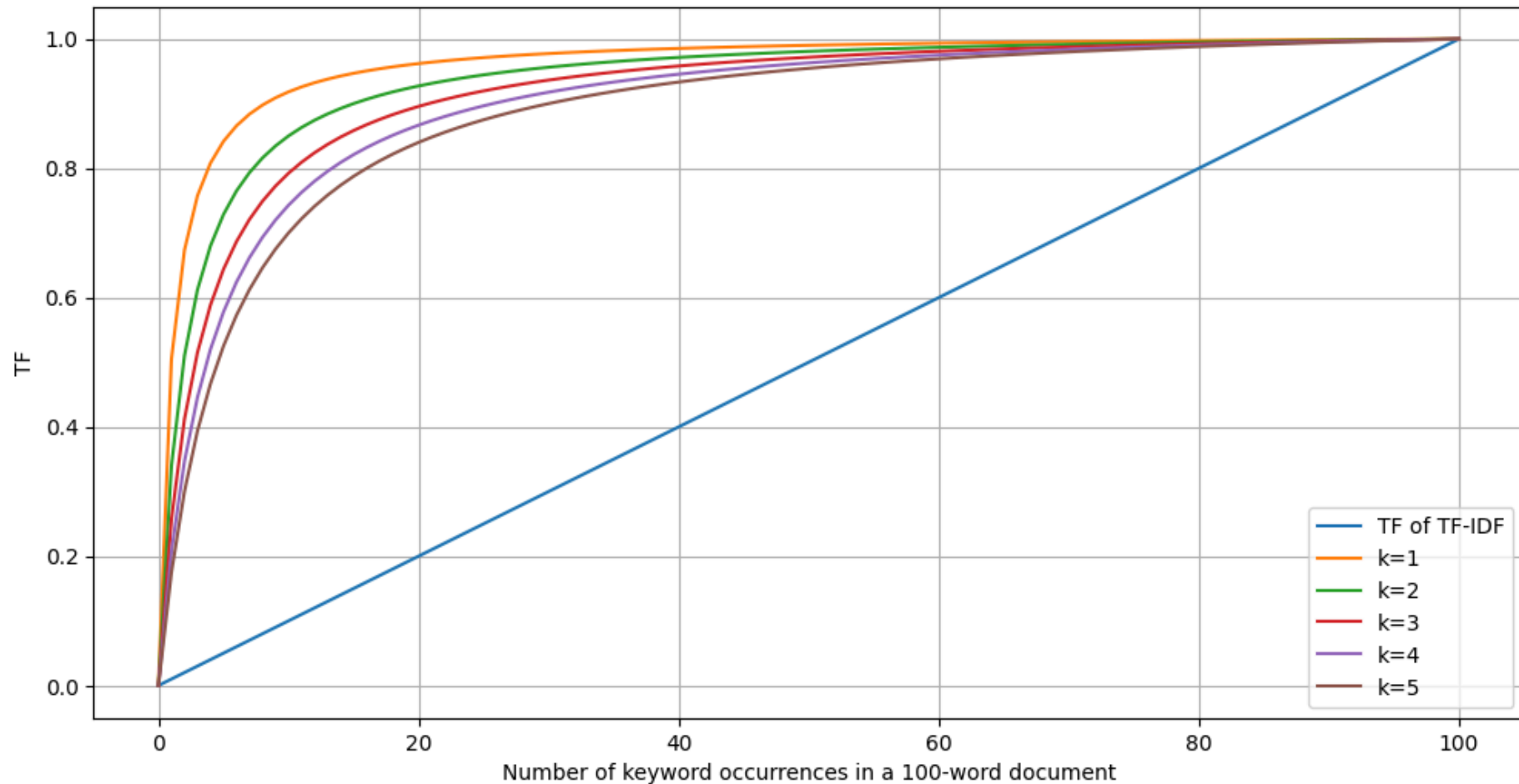
Pour un document  $d$  et une requête  $q$  :

$$\text{BM25}(d, q) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{\text{tf}(t, d)(k_1 + 1)}{\text{tf}(t, d) + k_1 \left(1 - b + b \frac{|d|}{\text{avgdl}}\right)}$$

somme des scores pour chaque terme ↑

↑ saturation tf

$k_1$  contrôle la saturation de la formule TF (*term frequency*) ; limite combien un terme dans la requête change le score d'un document. Un  $k_1$  plus élevé signifie que le score augmente moins rapidement avec la fréquence du terme. Généralement compris entre 1,2 et 2.



Source : <https://zilliz.com/learn/mastering-bm25-a-deep-dive-into-the-algorithm-and-application-in-milvus>

# Recherche par mots-clés : BM25

Pour un document  $d$  et une requête  $q$  :

$$\text{BM25}(d, q) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{\text{tf}(t, d)(k_1 + 1)}{\text{tf}(t, d) + k_1 \left(1 - b + b \frac{|d|}{\text{avgdl}}\right)}$$

somme des scores pour chaque terme (pointing to the sum)

saturation tf (pointing to  $k_1$ )

long. doc. (pointing to  $|d|$ )

long. moy. tous docs (pointing to  $\text{avgdl}$ )

# Recherche par mots-clés : BM25

Pour un document  $d$  et une requête  $q$  :

$$\text{BM25}(d, q) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{\text{tf}(t, d)(k_1 + 1)}{\text{tf}(t, d) + k_1 \left(1 - b + b \frac{|d|}{\text{avgdl}}\right)}$$

Diagram annotations:

- $\sum_{t \in q}$ : somme des scores pour chaque terme
- $k_1$ : saturation tf
- $b$ : norm. longueur
- $|d|$ : long. doc.
- $\text{avgdl}$ : long. moy. tous docs

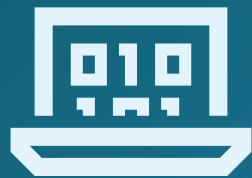
$b \in [0, 1]$  contrôle la pénalité appliquée aux documents longs.

# Recherche par mots-clés : BM25

Plusieurs implémentations Python existent pour BM25 :

- ♦ **rank-bm25** : implémente Okapi BM25 (la version de base de BM25), ainsi que des variantes comme BM25+ et BM25L. <https://pypi.org/project/rank-bm25/>
- ♦ **bm25s** : une autre implémentation plus rapide comparativement à rank-bm25. <https://bm25s.github.io/>





## **IV. Modèles de récupération de documents : recherche dense**

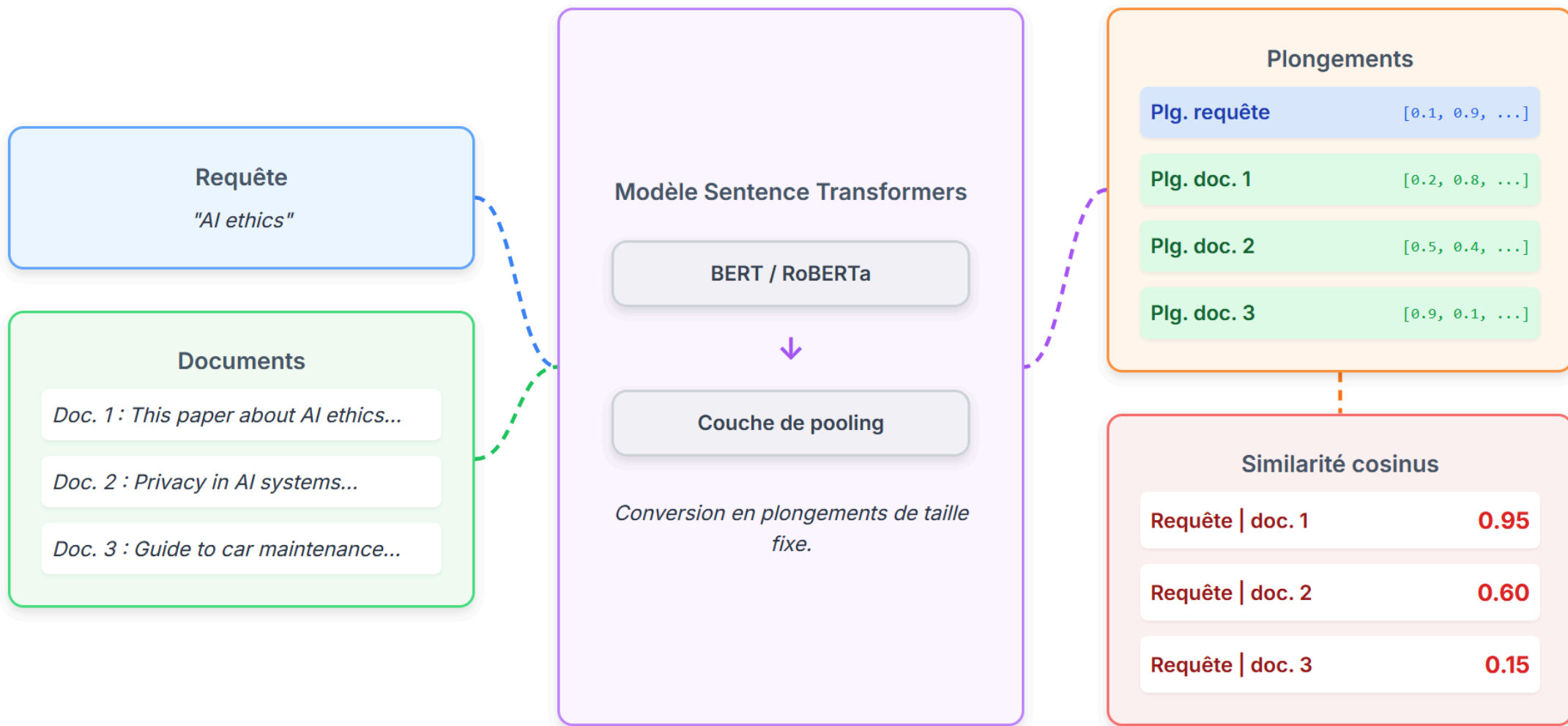
# Recherche dense

Les modèles que nous avons vus jusqu'à présent (TF-IDF et BM25, dits « **modèles épars** ») ne permettent pas de capturer l'aspect sémantique des documents et des requêtes. Cela signifie que le sens des mots est ignoré, et que les synonymes ne sont pas pris en compte pour la recherche.

Les **modèles denses** permettent de surmonter ces limitations en capturant les représentations vectorielles des mots et des documents.

# Recherche dense

Des bibliothèques Python telles que sentence-transformers [?] permettent de créer facilement ces représentations vectorielles.



# Recherche dense

Des bibliothèques Python telles que sentence-transformers [?] permettent de créer facilement ces représentations vectorielles :

```
1 model = SentenceTransformer(MODEL)
2 doc_embeddings = model.encode_document(documents,
   convert_to_tensor=True)
3 query_embedding = model.encode_query(query,
   convert_to_tensor=True)
4 similarity_scores = model.similarity(query_embedding,
   doc_embeddings)[0]
5 scores, indices = torch.topk(similarity_scores,
   k=TOP_K)
```



# Recherche dense : recherche symétrique/ asymétrique

On dit qu'une recherche est **symétrique** quand la requête et les documents ont à peu près la même longueur.

On dit qu'une recherche est **asymétrique** quand la requête est bien plus courte que les documents (p. ex., question ou recherche par mots-clés).

# Recherche dense : recherche symétrique/asymétrique

En fonction du type de recherche, les fonctions de sentence-transformers à utiliser diffèrent :

- ♦ en cas de recherche symétrique, on utilise `model.encode()` ;
- ♦ en cas de recherche asymétrique, on utilise `model.encode_query()` pour la requête et `model.encode_document()` pour les documents.

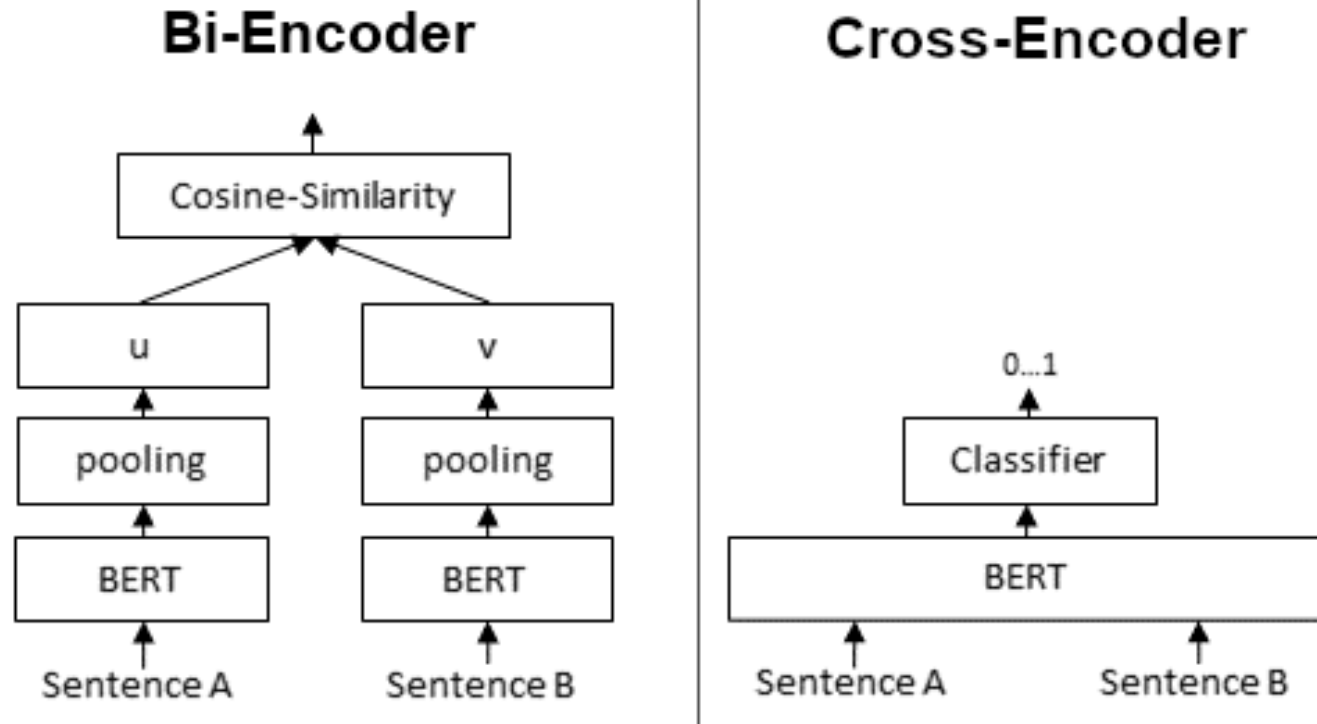
Plus d'informations : [https://sbert.net/examples/sentence\\_transformer/applications/semantic-search/README.html#symmetric-vs-asymmetric-semantic-search](https://sbert.net/examples/sentence_transformer/applications/semantic-search/README.html#symmetric-vs-asymmetric-semantic-search)

# Recherche dense : types d'encodeurs/modèles

- ♦ **bi-encodeur** : un modèle qui encode la requête et les documents indépendamment, avec des plongements distincts. La similarité entre la requête et les documents est mesurée selon la distance des plongements.
- ♦ **cross-encodeur** : un modèle qui encode la requête et les documents ensemble et qui donne un score de similarité.



# Recherche dense : types d'encodeurs/modèles



Source : [https://sbert.net/examples/cross\\_encoder/applications/README.html](https://sbert.net/examples/cross_encoder/applications/README.html) 

# Recherche dense : types d'encodeurs/modèles

## Bi-encodeurs

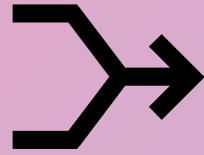
- ♦ <https://huggingface.co/google/embeddinggemma-300m> (multilingue)
- ♦ <https://huggingface.co/intfloat/multilingual-e5-large> (multilingue ; (Wang et al., 2024))
- ♦ <https://huggingface.co/antoinelouis/biencoder-mMiniLMv2-L6-mmarcoFR> (français)

## Cross-encodeurs

En français (basés sur le modèle CamemBERT (Martin et al., 2020)) :

- ♦ <https://huggingface.co/antoinelouis/crossencoder-camembert-base-mmarcoFR>

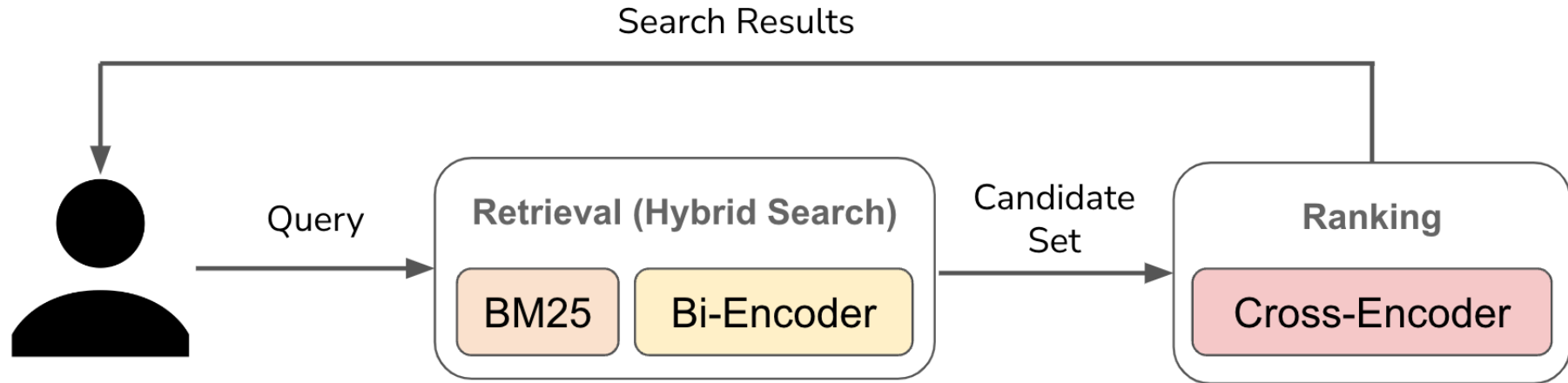
- ♦ <https://huggingface.co/dangvantuan/CrossEncoder-camembert-large>



## **V. Modèles de récupération de documents : recherche hybride**

# Recherche hybride

La recherche hybride combine les scores des modèles épars (comme BM25) et des modèles denses (comme ceux basés sur des plongements).



Source : <https://cameronrwolfe.substack.com/p/the-basics-of-ai-powered-vector-search>

# Recherche hybride

La recherche hybride combine les scores des modèles épars (comme BM25) et des modèles denses (comme ceux basés sur des plongements).

On ajoute une étape supplémentaire, celle du **reclassement** (*reranking*), qui sert à obtenir les documents les plus pertinents d'après les meilleurs résultats des deux modèles.

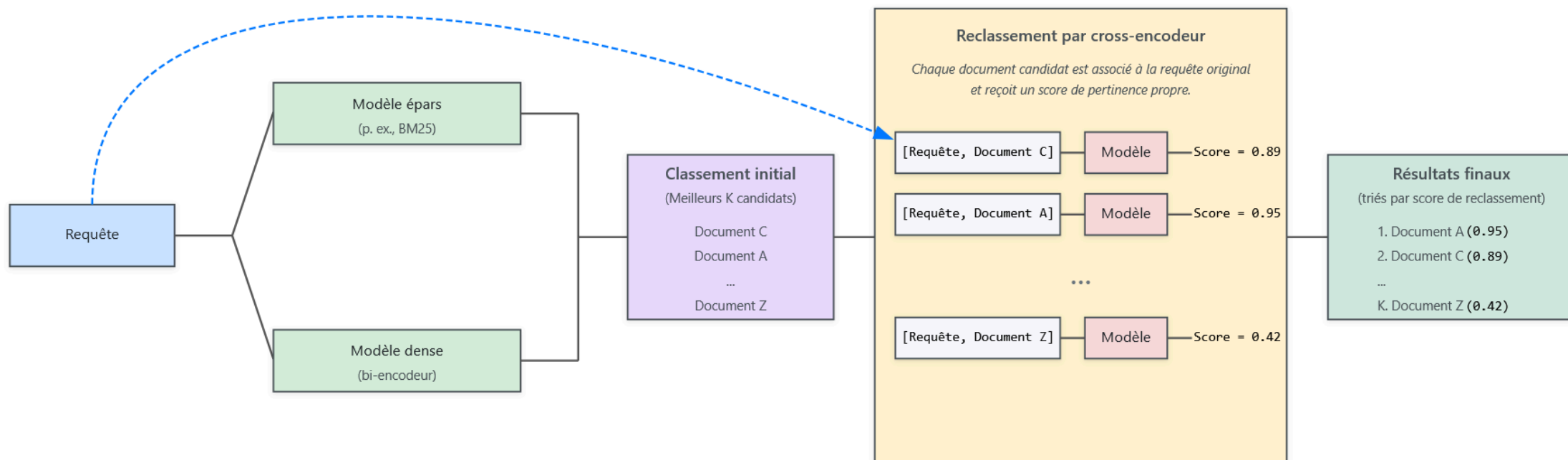
# Recherche hybride : reclassement

L'étape de reclassement peut se faire de plusieurs manières :

- ♦ **reclassement par cross-encodeur** : on utilise un modèle de type cross-encodeur pour calculer un score de pertinence entre la requête et les documents récupérés par les modèles épars et denses.
- ♦ **reclassement par RRF (Ranked Retrieval Fusion)** : algorithme qui unifie les classements des différents modèles.
- ♦ **reclassement par modèle de langue à base d'instructions** (p. ex., GPT-5) : méthode explorée plus récemment (**Déjean et al., 2024 ; Sun et al., 2024**). **[+]**

Ces méthodes ne sont pas exclusives et peuvent être combinées (RRF pour unifier, puis cross-encodeur pour obtenir un classement plus précis).

# Recherche hybride : reclassement par cross-encodeur





# Recherche hybride : reclassement par RRF

L'algorithme RRF (Ranked Retrieval Fusion) est défini comme suit :

$$\text{RRF}(d) = \sum_{s \in S} \frac{1}{k + \text{rank}_s(d)}$$

Où  $d$  est un document,  $S$  un ensemble de systèmes de recherche,  $\text{rank}_s(d)$  le rang du document  $d$  dans la liste des résultats du système  $s$ , et  $k$  une constante utilisée pour atténuer l'influence des rangs élevés (généralement 60).

# Recherche hybride : reclassement par RRF

Épars	Dense	Score RRF (k=60)	Final
1	2	$\frac{1}{k+1} + \frac{1}{k+2} = \frac{1}{61} + \frac{1}{62} \approx 0.03252$	1
2	1	$\frac{1}{k+2} + \frac{1}{k+1} = \frac{1}{62} + \frac{1}{61} \approx 0.03252$	1
3	4	$\frac{1}{k+3} + \frac{1}{k+4} = \frac{1}{63} + \frac{1}{64} \approx 0.03150$	4
5	2	$\frac{1}{k+5} + \frac{1}{k+2} = \frac{1}{65} + \frac{1}{62} \approx 0.03151$	3

## ⊕ Ressources complémentaires

- ♦ *Sentence Embeddings. Cross-encoders and Re-ranking* : [https://osanseviero.github.io/hackerllama/blog/posts/sentence\\_embeddings2/](https://osanseviero.github.io/hackerllama/blog/posts/sentence_embeddings2/)
- ♦ *Awesome Information Retrieval* : <https://github.com/harpribo/awesome-information-retrieval>

# Bibliographie

- Déjean, H., Clinchant, S., et Formal, T. (mars 2024). *A Thorough Comparison of Cross-Encoders and LLMs for Reranking SPLADE* (Numéro arXiv:2403.10407). arXiv. [10.48550/arXiv.2403.10407](#)
- Kalwar, S., Rossi, M., et Sadeghi, M. (2023). Automated Creation of Mappings Between Data Specifications Through Linguistic and Structural Techniques. *IEEE Access*, 11, 30324–30339. [10.1109/ACCESS.2023.3259904](#)
- Mandelbrot, B. B. (1965). Information Theory and Psycholinguistics. In B. B. Wolman et E. Nagel (éds.), *Scientific Psychology: Scientific Psychology*. Basic Books.
- Manning, C. D., Raghavan, P., et Schütze, H. (2008). *Introduction to Information Retrieval*. Cambridge University Press.
- Martin, L., Muller, B., Suárez, P. J. O., Dupont, Y., Romary, L., Clergerie, É. V. de la, Seddah, D., et Sagot, B. (2020). CamemBERT: A Tasty French Language Model. *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, 7203–7219. [10.18653/v1/2020.acl-main.645](#)
- Mitra, B. (2018). *An Introduction to Neural Information Retrieval* (Numéro v.41). Now Publishers.
- Sparck Jones, K. (janvier 1972). A Statistical Interpretation of Term Specificity and Its Application in Retrieval. *Journal of Documentation*, 28(1), 11–21. [10.1108/eb026526](#)
- Sun, W., Yan, L., Ma, X., Wang, S., Ren, P., Chen, Z., Yin, D., et Ren, Z. (décembre 2024). *Is ChatGPT Good at Search? Investigating Large Language Models as Re-Ranking Agents* (Numéro arXiv:2304.09542). arXiv. [10.48550/arXiv.2304.09542](#)
- Wang, L., Yang, N., Huang, X., Yang, L., Majumder, R., et Wei, F. (février 2024). *Multilingual E5 Text Embeddings: A Technical Report* (Numéro arXiv:2402.05672). arXiv. [10.48550/arXiv.2402.05672](#)
- Zhai, C., et Massung, S. (juin 2016). *Text Data Management and Analysis: A Practical Introduction to Information Retrieval and Text Mining*. Association for Computing Machinery and Morgan & Claypool. [10.1145/2915031](#)

Zipf, G. K. (1932). *Selected Studies of the Principle of Relative Frequency in Language* (p. 57). Harvard Univ. Press. [10.4159/harvard.9780674434929](#)