J'ai conçu une architecture où le traitement des entités nommées est divisé en deux grandes applications, reliées par un script central `global_launch.py` qui permet d'automatiser l'exécution complète du pipeline. L'objectif était de structurer le processus en plusieurs étapes bien distinctes : d'abord **l'extraction des tokens d'intérêt**, suivie de **la vectorisation et la construction d'une matrice de similarité**, puis l'application d'un **clustering basé sur AffinityPropagation**, avant de passer à la **visualisation des résultats en 2D**. Cette approche permet une gestion plus modulaire et flexible du projet, avec la possibilité d'affiner chaque composant indépendamment.

La première application s'occupe de **l'extraction et du clustering**. Elle commence par lire les fichiers `.bio` grâce au module `reader.py`, qui extrait les tokens et leurs labels associés. Ensuite, ces tokens passent par `spacy_processor.py`, qui applique une normalisation via **lemmatisation et analyse morphosyntaxique (POS)**, un choix qui permet d'améliorer la cohérence des regroupements. Une fois ces formes traitées, elles sont vectorisées dans `vectorizer.py` via des **n-grammes de caractères** et transformées en une **matrice de similarité cosinus**. Cette matrice est ensuite exploitée par `clusterer.py`, qui applique **AffinityPropagation** pour regrouper les tokens similaires en clusters distincts. Les résultats sont enfin stockés sous forme de **fichiers JSON** via `saver.py`, rendant ces données réutilisables pour la visualisation.

La seconde application se charge de **l'analyse et de la visualisation des résultats**. À partir des fichiers JSON produits par l'application précédente, `json_reader.py` récupère les informations nécessaires sur les clusters et leurs centroïdes. Ensuite, `plotter.py` applique une **réduction de dimension en 2D via MDS** afin de représenter graphiquement les relations entre tokens. `main_visualization.py` orchestre le tout, en **créant des sous-graphes comparatifs et en sauvegardant une image `.png` finale** permettant d'analyser les regroupements.

Le script `global_launch.py` permet de tout exécuter en une seule commande. Depuis le dossier `global_launch`, il suffit de lancer : `bash python main.py` Ce script central automatise la séquence complète : d'abord **le clustering**, où les fichiers `.bio` sont lus, transformés et regroupés en clusters, puis la **visualisation**, qui génère des images des regroupements obtenus. Cela garantit un traitement fluide et permet d'expérimenter différentes configurations en ajustant simplement les paramètres de vectorisation ou de clustering.

Concernant l'analyse des résultats, les visualisations générées dans `compare_texts_2d.png` mettent en évidence plusieurs observations intéressantes. Chaque point correspond à un **token**, coloré selon le **cluster auquel il appartient**, et les centroïdes sont mis en évidence avec des marqueurs plus grands. On observe des **clusters bien formés dans certaines parties des sous-graphes**, montrant que l'algorithme a correctement regroupé des tokens similaires. Cependant, **certaines dispersions plus importantes** suggèrent que certains tokens mal segmentés ou bruités par l'OCR ont été intégrés dans des groupes peu cohérents.

Les différences entre les sous-graphes mettent en lumière **l'impact des différentes sources de texte et des techniques d'OCR utilisées**. Par exemple, les résultats obtenus avec Kraken OCR montrent une **meilleure structuration des clusters**, tandis que ceux issus de Tesseract OCR sont **plus diffus**, ce qui indique une segmentation moins précise des entités. De même, des erreurs OCR peuvent provoquer des **variantes orthographiques** qui faussent les regroupements.

D'un point de vue technique, j'ai testé plusieurs configurations pour évaluer **l'impact des paramètres du clustering**. En ajustant les n-grammes utilisés (2,3 vs. 3,4), j'ai remarqué des **changements dans la densité des clusters** : des n-grammes plus longs permettent d'affiner les similarités, mais risquent d'éliminer trop de variations orthographiques utiles. De plus, la **préférence dans AffinityPropagation** influence directement le nombre de clusters : une valeur plus élevée tend à **produire moins de clusters mais plus homogènes**, tandis qu'une valeur plus basse conduit à une **augmentation du nombre de clusters, mais avec parfois des regroupements trop approximatifs**.

Dans l'ensemble, cette architecture modulaire a permis une exécution **automatisée et adaptable**, et l'intégration d'un **moteur spaCy pour la normalisation linguistique** a amélioré la précision globale. Cependant, certains défis restent à résoudre : il faudrait affiner la robustesse des clusters pour **réduire l'impact du bruit introduit par l'OCR**, et expérimenter **d'autres algorithmes de clustering**, comme **DBSCAN** ou **Spectral Clustering**, pour évaluer si une autre approche donnerait de meilleurs résultats. Enfin, l'exploration de modèles plus avancés comme **FastText ou BERT** pourrait permettre une meilleure représentation des similarités sémantiques, au-delà des simples n-grammes de caractères.

Cette expérimentation a validé une **approche intégrée et évolutive**, où chaque composant peut être optimisé séparément, offrant une grande flexibilité pour l'amélioration des résultats.