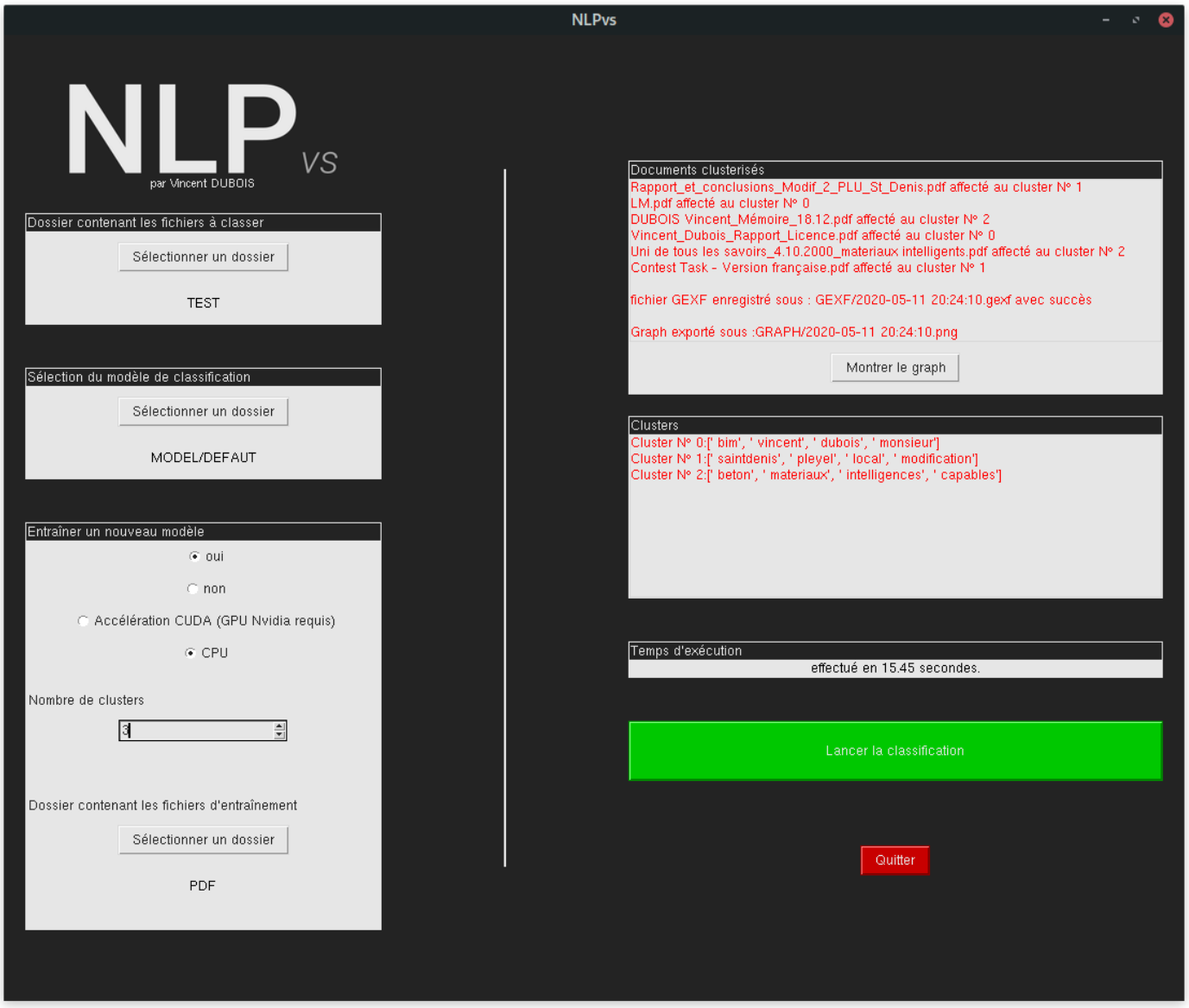
NLPvs

Outil de classification de documents au format PDF



Développé par DUBOIS Vincent dans le cadre d'une mission pour la chaire A.I. : Architecture Intelligence

ABSTRACT

NLPvs est un outil de classification non-supervisé de documents au format PDF.

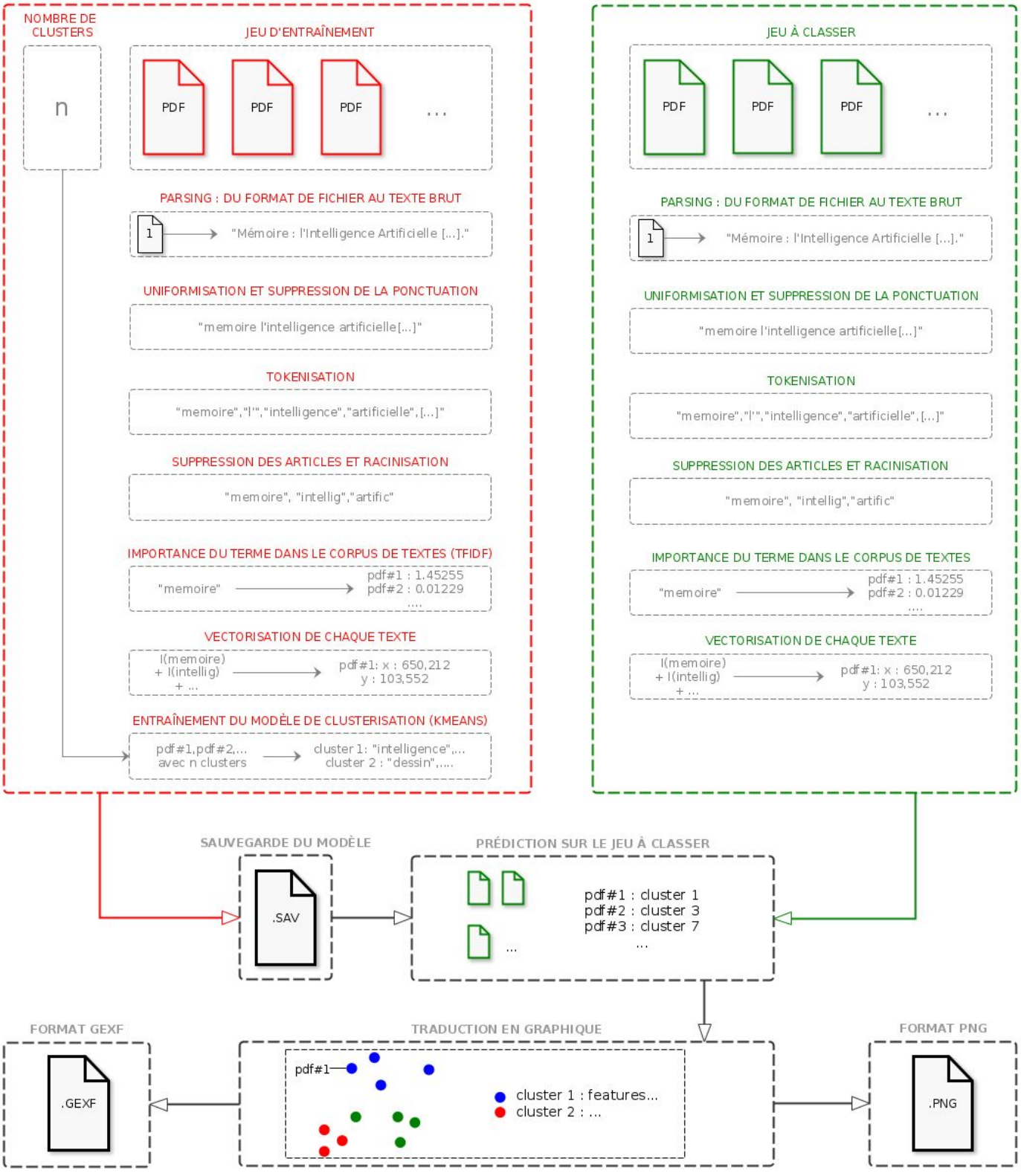
Pour un corpus de documents à classer défini par l'utilisateur, l'outil se charge de les répartir au sein de clusters selon le modèle de Machine Learning pré-entraîné choisi, et de restituer cette classification sous forme graphique montrant la proximité des documents entre eux ainsi que leurs répartition au sein des clusters.
Il est également possible d'entraîner son propre modèle, en spécifiant un corpus de documents d'entraînement ainsi qu'un nombre de clusters (avec une possibilité de l'accélérer grâce aux coeurs CUDA sur un GPU Nvidia). Après l'entraînement, ce modèle est automatiquement sauvegardé (pour permettre une réutilisation pour un autre jeu de documents) et procède à la classification du corpus de documents à classer.

Pour chaque corpus (entraînement ou classification) de PDF, un traitement initial est requis:

- **"Parsing" des fichiers PDF** : il s'agit ici d'extraire le contenu textuel de chaque fichier (sous forme de chaîne de caractères) et d'en supprimer toute mise en page.
- **Uniformisation et suppression de la ponctuation de chaque texte.**
- **"Tokenisation"** : découpage d'un texte en termes (mots) afin de créer une entrée de vocabulaire (un "jeton")
- **Suppression des articles et "stop words"** (verbes être et avoir, etc…) : élimination des termes les plus communs entre chaque texte afin de les alléger pour les étapes suivantes.
- **"Racinisation"** : réduction de chaque terme à sa "racine" en enlevant conjugaison, suffixe ou préfixe.
- **Mesure du "Text Frequency - Inverted Frequency Index" (TF-IDF)** : Estimation de la fréquence d'un terme (ou groupe de termes) à travers un corpus, sur la base de sa rareté au sein du corpus (moins un terme est fréquent, plus il a de chances d'être porteur de sens pour pouvoir différencier les textes entre eux).
- **Vectorisation de chaque texte** : Associe un vecteur à chaque texte sur la base des indices de fréquence.

Un tel corpus peut ensuite servir de base pour entraîner un modèle (basé sur l'algorithme de clusterisation Kmeans) afin d'en tirer des clusters propres à ce modèle puisque lié aux vocabulaire de termes du corpus d'entraînement (les features de ces clusters seront les termes les plus significatifs qui ont servi au modèle pour différencier les textes entre eux).

Le modèle est ensuite chargé de "prédire" auquel de ses clusters appartient chaque texte du corpus à classer. Enfin, la distance cosinus entre chaque texte vectorisé sera calculée, dans le but de générer un graphique légendé qui traduira la clusterisation et cette distance en un seul document au format PNG. Un document au format GEXF sera également produit de manière systématique.


NOMBRE DE CLUSTERS
n
JEU D'ENTRAÎNEMENT
PDF
PDF
PDF
...
PARSING : DU FORMAT DE FICHIER AU TEXTE BRUT
1
"Mémoire : l'Intelligence Artificielle [...]."
UNIFORMISATION ET SUPPRESSION DE LA PONCTUATION
"memoire l'intelligence artificielle[...]"
TOKENISATION
"memoire","l'","intelligence","artificielle",[...]"
SUPPRESSION DES ARTICLES ET RACINISATION
"memoire", "intellig","artific"
IMPORTANCE DU TERME DANS LE CORPUS DE TEXTES (TFIDF)
"memoire"
pdf#1 : 1.45255
pdf#2 : 0.01229
....
VECTORISATION DE CHAQUE TEXTE
l(memoire)
+ l(intellig)
+ ...
pdf#1: x : 650,212
y : 103,552
ENTRAÎNEMENT DU MODÈLE DE CLUSTERISATION (KMEANS)
pdf#1,pdf#2,...
avec n clusters
cluster 1: "intelligence",...
cluster 2 : "dessin",....
JEU À CLASSER
PDF
PDF
PDF
...
PARSING : DU FORMAT DE FICHIER AU TEXTE BRUT
1
"Mémoire : l'Intelligence Artificielle [...]."
UNIFORMISATION ET SUPPRESSION DE LA PONCTUATION
"memoire l'intelligence artificielle[...]"
TOKENISATION
"memoire","l'","intelligence","artificielle",[...]"
SUPPRESSION DES ARTICLES ET RACINISATION
"memoire", "intellig","artific"
IMPORTANCE DU TERME DANS LE CORPUS DE TEXTES
"memoire"
pdf#1 : 1.45255
pdf#2 : 0.01229
....
VECTORISATION DE CHAQUE TEXTE
l(memoire)
+ l(intellig)
+ ...
pdf#1: x : 650,212
y : 103,552
SAUVEGARDE DU MODÈLE
.SAV
PRÉDICTION SUR LE JEU À CLASSER
pdf#1 : cluster 1
pdf#2 : cluster 3
pdf#3 : cluster 7
...
...
FORMAT GEXF
.GEXF
TRADUCTION EN GRAPHIQUE
pdf#1
cluster 1 : features...
cluster 2 : ...
FORMAT PNG
.PNG

ASPECTS TECHNOLOGIQUES

Langage de programmation

NLPvs a été écrit en Python, sous sa version 3.7.6 (prodiguée par défaut dans la version 4.8.3 du framework Anaconda, détaillé dans le chapitre suivant).

Environnement de développement

Ce programme a été développé et testé sous trois systèmes d'exploitations différents :

- Manjaro Linux (19.10) (OS Principal de développement)
- Pop!_os (20.04)
- Windows 10

Le choix d'un framework commun aux trois systèmes d'exploitation s'est porté sur Anaconda (version 4.8.3), et plus particulièrement sur sa version minimale sans interface graphique, Miniconda, rendant possible une procédure d'installation des modules universelle et simplifiée via le terminal système (à l'exception de Windows, qui requiert l'utilisation du terminal d'Anaconda, étant donné que l'invite de commandes de base ne supporte pas les commandes spécifiques à Anaconda).

Visual Studio Code a été choisi comme IDE pour la quasi intégralité du développement (un recours à un autre IDE, PyCharm a été requis sous Windows 10 suite à une incompatibilité avec VS Code pour tester le code source).

Enfin, ce projet a été développé en utilisant le système de contrôle de version Git, associé à un dépôt GitHub. Ces deux solutions ont permis de facilement identifier les différents changements opérés localement sur les fichiers sources, puis les synchroniser avec le dépôt distant sur GitHub afin de communiquer aux enseignants les différents changements. La fonction de clonage du dépôt permet également d'effectuer simplement une copie des fichiers sources de manière ponctuelle sur un système à des fins de test.

Le dépôt GitHub a par ailleurs été alimenté d'une documentation tout au long du développement. Elle comprend entre autres une liste des bibliothèques utilisées, des références de travaux ou sources de documentation, ou encore une liste des fonctionnalités.

Modules employés

Plusieurs modules ont été nécessaires pour l'intégration des fonctionnalités clés du programme :

- **PDFminer**
  Cette bibliothèque permet le "parsing" des fichiers au format PDF, c'est à dire l'extraction de tout objet texte présent au sein du document, pour en sortir du texte brut sous la forme d'une chaîne de caractère.

- **NLTK**
  NLTK est une bibliothèque permettant le traitement des textes sous forme de chaîne de caractères afin de les préparer pour des opérations de NLP (Natural Language Processing). Au sein du programme, elle sert principalement à deux étapes : la "tokenisation" (découpage de la chaîne de caractère en une liste de "tokens" (voir section suivante)) et la suppression des "stopwords"(suppression des mots récurrents avec peu de signification sémantique, comme les articles).
  Un téléchargement des bibliothèques contenant les "stopwords" français est requis via un script à exécuter inclus dans le dépôt du projet.

- **French Lefff Lemmatizer**
  Cette bibliothèque a une fonction bien précise, celle de la "racinisation" des mots, c'est à dire la troncature en supprimant préfixes et suffixes, de manière à empêcher la prise en compte répétée de termes de la même famille sémantiquement assez proches (comme "doublé", "doublée" ou encore "doublées" qui seront réunis sous le même token "doubl").

- **Pandas**
  Pandas est une bibliothèque permettant de créer des "Data Frames", objets sous forme de tableau permettant d'organiser des données diverses en colonnes et lignes, de manière à les visualiser et pouvoir y faire appel plus simplement par la suite. Sous python, la manière la plus simple de créer des "Data Frames" est de partir d'un dictionnaire ("clé1" : valeur1, "clé2" : valeur2, etc…), ou de deux listes (pour chaque colonne/ligne).

- **Scikit Learn**
  Scikit Learn est une bibliothèque orientée "Machine Learning", contenant une multitude d'algorithmes généraux à importer et configurer pour en former un modèle que l'on peut entrainer selon ses propres données afin d'accomplir différentes tâches. Ici, trois algorithmes seront appelés : TF-IDF Vectorizer (Term Frequency - Inverted Frequency Index), servant à la vectorisation des textes traités aux préalable, ainsi que K-Means, un algorithme de clusterisation (classification non-supervisée) permettant le regroupement des textes vectorisés en un nombre n défini de clusters, ainsi que MDS, servant ponctuellement à mapper les vecteurs de sortie sur un plan en deux dimensions.

Une bibliothèque annexe installée en même temps que Scikit-Learn nommée joblib sert à effectuer des sauvegardes des modèles créés dans un format générique .SAV permettant une réutilisation ultérieure.

- **Numpy**
  Cette bibliothèque permet de créer des matrices sous Python (objets de type np.array). Elle est particulièrement utile quand il s'agit d'effectuer des opérations sur des vecteurs (étant donné que les textes sont vectorisés), comme par exemple le calcul de leur distance euclidienne.
  Une version de Numpy tirant parti des coeurs CUDA est également disponible sous le nom Cupy, bibliothèque de base pour de nombreuses librairies offrant une accélération CUDA.

- **CuML**
  CuML est une bibliothèque appartenant à la suite RAPIDS, contenant des versions accélérées par CUDA de certains algorithmes de Scikit-Learn. Au sein du programme, cette bibliothèque sert à construire un modèle basé sur K-Means accéléré par CUDA à la place de sa version traditionnelle tournant sur le CPU. C'est une, fonctionnalité que l'on peut définir via l'interface graphique utilisateur si l'on possède un GPU Nvidia compatible.

- **Matplotlib**
  Matplotlib contient des fonctions permettant de générer des graphiques à partir d'éléments Python. Elle sert dans le programme à montrer sous forme graphique le résultat de la clusterisation des textes grâce à des couleurs, ainsi que la proximité entre ces textes. Elle sert également de manière ponctuelle à traduire les couleurs RGB en Hexadécimal (pour l'attribution des couleurs des clusters).

- **Networkx**
  Networkx est une bibliothèque orientée sur la représentation de réseaux de neurones. Elle permet de définir différentes options pour chacun des noeuds (coordonnées, taille,etc…), ainsi que leurs connexions (définition du "poids" de chacune). Enfin, elle permet d'exporter ce type de représentation au format .GEXF, format dérivé du XML orienté sur les réseaux de neurones.

D'autres bibliothèques ont servi pour mettre en forme l'interface graphique :

- **PIL (Pillow)**
  Pillow est une bibliothèque permettant le traitement d'images dans Python, du simple affichage à la transformation des canaux de couleurs des pixels. Ici, elle sert à l'affichage du logo ainsi que celui du graph exporté dans une fenêtre dédiée (accessible via le bouton "montrer graph").

- **TkInter**
  Tkinter est une des bibliothèques les plus connues de création d'interfaces graphiques sous Python. Cette bibliothèque a été choisie pour sa simplicité de création de widgets et sa documentation abondante. C'est par ailleurs la seule de cette liste qui est intégrée à Python.

DE LA CHAÎNE DE CARACTÈRES AUX TOKENS

Une étape cruciale à tout type de traitement en vue d'effectuer du Natural Language Processing (NLP) sur un corpus de texte est le "pre-processing" des chaînes de caractères brutes obtenues suite au parsing.

```
for entry in noms_fichiers:

textes_fichiers.append(pdfminer.high_level.extract_text(entry,"",None,0,True,'utf-8',None)
```

A l'issue du parsing, pour chaque document d'un corpus, une chaîne de caractères encodée en UTF-8 sera obtenue. Elles seront toutes regroupées dans une seule liste (textes_fichiers) via la boucle ci-dessus.

```
def nettoyage_sans_phrases(txt):

   txt_ss = list()

   for texte in txt:
       texte = "".join(texte.splitlines())
       texte =
unicodedata.normalize('NFD',texte).encode('utf-8','ignore').decode('utf-8','ignore')
       texte = texte.lower()
       rm_ponctuation = str.maketrans('','', string.punctuation)
       texte = texte.translate(rm_ponctuation)
       txt_ss.append(texte)

   return txt_ss
```

Une première étape de nettoyage s'opère grâce à la fonction ci-dessus. Elle effectue les opérations suivantes pour chaque texte d'une liste donnée :

- supprimer les sauts de lignes qui se traduisent par l'apparition d'un "\n" (carriage return en UTF-8)

- normaliser tous les caractères en effectuant un encodage puis un décodage, toujours en UTF-8, pour filtrer les caractères non-supportés qui subsisteraient (d'où les "ignore" en paramètre).

- mettre chaque caractère en minuscule.

- supprimer la ponctuation (en la remplaçant par un caractère vide) à travers une fonction de traduction de NLTK qui contient chaque élément de ponctuation.

Une fois cette fonction appliquée au corpus, la liste en sortie contient toujours une chaîne de caractère par document, mais avec tous les mots les uns à la suite des autres sans ponctuation :

```
"I.M. : Intelligence Matérielle : L'émergence d'une connexion entre matériaux de construction et l'Internet des objets."

>>>

“im intelligence matérielle l’émergence d’une connexion entre matériaux de construction et l’internet des objets”
```

Néanmoins, pour pouvoir effectuer du NLP sur ce corpus, il faut lui faire subir une transformation clé, appelée “tokenisation”. Le but de cette étape est de découper une chaîne de caractères en une liste de chaînes de caractères appelées “tokens”, Un “token” se définit comme un ensemble de caractères qui va être associé à un identifiant qui lui est propre (et donc pouvoir le reconnaître facilement dans les différents documents) et dont on va chercher par la suite à évaluer l’impact sur la sémantique d’un texte.

Dans la plupart des cas, un “token” correspond à un mot du texte (appelé alors un “terme”), rendant cette étape assez simple à conceptualiser et effectuer (l’algorithme se contente alors de découper les chaînes de caractères séparées par des espaces). Cependant, pour une meilleure analyse sémantique, il est souvent intéressant en parallèle de combiner plusieurs termes en un unique “token” (les noms de ville en sont un bon exemple : “Saint”, “Denis” et “Saint Denis”). Cet aspect sera abordé plus précisément dans la section suivante.

```
def token_pour_tfidf(txt):

    stop_words = set(stopwords.words("french"))

    tk = [mot for mot in txt.split() if (len(mot)>2) and (mot not in stop_words)]

    blank = ""
    n_tk = [token for token in tk if (token not in blank)]

    stemmer=FrenchStemmer()
    n_tk = [stemmer.stem(token) for token in n_tk]

    return n_tk
```

La fonction ci-dessus regroupe l’étape de la “tokenisation”, effectuée grâce au module NLTK, qui contient un algorithme de tokenisation ainsi que d’autres opérations de filtrage des tokens obtenus :

- Importer les “stopwords” de la langue française, intégrés au module NLTK une fois téléchargés avec le script dédié (voir la section “modules employés”). Ces “stopwords” sont une liste de termes très communs dans la langue choisie, et qui ne sont donc pas porteurs de sens vis à vis du texte (comme les articles par exemple).

- Filtrer les tokens, de manière à supprimer ceux de moins de 3 caractères, ou bien s’ils appartiennent à la liste des “stopwords”.

- Supprimer les tokens vides résiduels.

La dernière transformation des tokens opérée par cette fonction est la “racinisation” “stemming” en anglais). Cette étape consiste à supprimer les éventuels suffixes et/ou préfixes d’un token, pour n’en garder que la “racine” au sens sémantique. Cette étape empêche certains termes de la même famille d’être comptabilisés sous un autre token, susceptibles de créer des doublons et donc de fausser leur importance au sein d’un texte (les accords grammaticaux par exemple, comme “mangé”, “mangés” ou encore “mangées” seront tronqués à “mang”)
Ici, c’est la fonction FrenchStemmer() du module Leff French Stemmer qui sera employée. C’est en réalité un modèle pré-entraîné à “raciniser” des mots français, et qui donne donc ici des résultats très corrects. Le contenu des textes en sortie de cette fonction est alors semblable à ceci :

```
"im intelligence matérielle l'émergence d'une connexion entre matériaux de construction et
l'internet des objets"

>>>

[“im”,”intelligence”,”matérielle”,”l’émergence”,”connexion”,”matériaux”,”construction”,”l’int
ernet”,”objets”]
```

Une “Data Frame” sera créée par la suite, qui contiendra pour chaque token “racinisé”, sa version complète, de manière à y faire appel plus tard, quand il faudra restituer les termes les plus significatifs par cluster (de cette manière, quand le terme “racinisé” sera appelé, on pourra à la place ressortir le terme complet pour la visualisation par l’utilisateur).

```
vocab = pd.DataFrame({'words': ss_racin}, index = racin)
```

A l’issue de toutes ces étapes, on obtient une liste, constituée elle même d’une liste par document, c’est à dire l'ensemble de ses “tokens”. Le corpus est enfin prêt pour être analysé en profondeur.

TF-IDF ET VECTORISATION

Une fois le corpus constitué d'une liste contenant des textes "tokenisés" (qualifiés de "sac de mots"), un algorithme nommé Term Frequency - Inverted Frequency Index Vectorizer ou TF-IDF Vectorizer va se charger d'associer un vecteur à un texte en fonction du "sac de mots" qui le définit.

```
tfconv = TfidfVectorizer(input = 'content',max_df=0.8, min_df=0.2,max_features =
9000000,ngram_range=(1,3),
token_pattern=None,analyzer='word',preprocessor=None,lowercase=False,
tokenizer=token_pour_tfidf).fit(sac)

corpus_vect = tfconv.transform(sac)
```

Dans la fonction ci-dessus, certains paramètres ont été définis pour affiner la pertinence des résultats vis à vis des "tokens":

- **max_df=0.8** indique que les termes communs à hauteur de plus de 80% ne sont pas comptabilisés. (termes trop fréquents pour être intéressants d'un point de vue sémantique)
- **min_df=0.2** indique que les termes communs à hauteur de moins de 20% ne sont pas comptabilisés. (termes trop rares pour définir de manière forte la sémantique d'un document).
- **max_features = 90 000 000** indique que le vocabulaire, c'est à dire le nombre de "tokens" uniques ne peut dépasser 90 000 000 termes (ici, la valeur est volontairement très élevée afin de ne pas limiter le traitement d'un nombre important de documents).
- **ngram_range(1,3)** signifie qu'à partir d'un "token" donné, le vocabulaire va être complété d'autres "tokens" formés d'une addition du "token" de base (d'indice n dans le sac de mots) et des "tokens" d'indices n+1,n+2, et n-1,n-2, afin d'inclure d'éventuels groupes de termes qui auraient une valeur sémantique intéressante :

```
"les GAFAM affirment que l'intelligence artificielle possède un grand potentiel"

>>> "l'intelligence"  [n]
>>> "affirment l'intelligence"  [n-1,n]
>>> "l'intelligence artificielle" [n,n+1] (valeur plus intéressante sémantiquement)
>>> "GAFAM affirment l'intelligence"  [n-2,n-1,n]
>>> "affirment l'intelligence artificielle" [n-1,n,n+1]
>>> "intelligence artificielle possède" [n,n+1,n+2]
```

TF-IDF Vectorizer récupère les "tokens" de tous les textes du corpus en les organisant dans plusieurs matrices, une par document (texte), en donnant pour chaque "token" un index qui lui est propre (commun à tous les textes) et son indice TF-IDF qui va définir son "poids" dans le document donné.

Le but de cette méthode est d'attribuer à chaque terme une valeur numérique, facilement manipulable par les algorithmes de Machine Learning.

L'indice TF-IDF se compose de deux mesures différentes : Term Frequency, la mesure de la fréquence du terme dans un document donné, et Inverse Document Frequency, la mesure de l'importance sémantique du terme dans le même document.

La raison pour laquelle on construit l'indice TF-IDF sur ces deux valeurs est basée sur les observations a posteriori suivantes : si un terme T est fréquent dans un document donné, on peut a priori affirmer que ce mot est important pour définir le sens de ce document. En réalité, certains termes qui reviennent souvent au sein du document, comme les articles (le, la, les, etc…) n'apportent quasiment aucun sens du point de vue de sa sémantique. C'est pourquoi on utilise le "Inverse Document Frequency", c'est à dire la mesure de la rareté du terme dans le corpus tout entier pour pondérer l'indice de fréquence. Ainsi, si un terme T a un indice de fréquence dans un document donné élevé, mais que ce terme est également très commun (donc un indice de rareté bas) dans le reste du corpus, ce terme n'est sûrement pas intéressant d'un point de vue de la sémantique du document en question.

D'un point de vue mathématique, la mesure du TF-IDF se définit comme la multiplication des indices TF et IDF. Dans la formule ci-dessous, t désigne le terme donné, d désigne le document en question, et D désigne quant à lui le corpus de documents.

$$tfidf(t, d, D) = tf(t, d) \cdot idf(t, D)$$

TF est alors calculé comme ceci, sur la base de la "fréquence" du terme t dans le document d (en l'occurrence, le nombre d'occurrences du terme sur le nombre de termes total du document) :

$$tf(t, d) = log(1 + freq(t, d))$$

Pour TF-IDF, le calcul est un logarithme appliqué sur une fraction, avec en numérateur le nombre N de documents au sein du corpus D, et en dénominateur le nombre de documents du même corpus où le terme t apparaît. Ainsi, si un terme est commun au sein du corpus, son "importance" tendra vers 0, et tendra dans le cas inverse vers 1.

$$idf(t, D) = log(\frac{N}{count(d \in D : t \in d)})$$

Cet indice TF-IDF obtenu pour chaque terme d'un document, ce dernier se verra attribuer un vecteur de n-dimensions (où n est le nombre de termes total dans le corpus, appelé alors dictionnaire). Considérons l'exemple ci-dessous :

```
phrase_1 = "l'intelligence artificielle repose sur les réseaux de neurones"

phrase_2 = "l'efficacité des réseaux de neurones repose sur leur entraînement"
```

La matrice avec les indices TFIDF par terme et par document se compose comme ceci :

terme	phrase_1	phrase_2
l'intelligence	2	0
artificielle	2	0
repose	1	1
réseaux	1	1
neurones	1	1
efficacité	0	2
entraînement	0	2

Dans cet exemple, on remarque que l'indice TF-IDF prends trois valeurs possibles :

- Si le terme t est absent dans un document, TF-IDF vaut 0
- Si le terme t est propre à un document, TF-IDF vaut 2 (ce qui indique une forte valeur sémantique)
- Si le terme t est présent dans tous les documents du corpus, TF-IDF vaut 1 (ce qui indique une valeur peu intéressante pour la sémantique, mais qui va augmenter leur similarité)

Ensuite, on définit pour chaque document un vecteur de n-dimensions (en l'occurrence, il ya 7 termes dans le dictionnaire du corpus, soit un vecteur à 7 dimensions).

```
V(phrase_1) = [(2,2,1,1,1,0,0)]

V(phrase_2) = [(0,0,1,1,1,2,2)]
```

La dernière étape est d'obtenir la distance entre ces différents vecteurs, c'est à dire leur similarité, qu'il sera possible d'exploiter pour la représenter graphiquement. Dès lors, le calcul de la distance euclidienne paraît être la méthode la plus simple et efficace, puisque nous disposons directement de deux vecteurs. Seulement, si l'on possède des vecteurs de plus de 2 dimensions, ce calcul ne traduit pas toujours la similarité de manière juste, étant donné qu'il y a énormément de dimensions à prendre en compte.

La méthode la plus adaptée est celle de la similarité cosinus ("cosine distance'" en anglais).

$$\cos\theta = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|}$$

La formule permet d'obtenir le cosinus de l'angle θ par la division du produit scalaire de A et B sur la multiplication des normes des vecteurs A et B.
Là où la distance euclidienne se base sur des coordonnées qu'il faut obtenir (x,y, etc...), le calcul de la similarité cosinus est indépendant du nombre de dimensions des vecteurs d'entrée, ce qui en fait un choix particulièrement adapté.

De plus, le cosinus étant compris dans l'intervalle [-1,1], il n'y a quasiment aucune opération à effectuer dessus pour obtenir des valeurs exploitables pour la restitution graphique :

```
distance = 1 - cosine_similarity(matrix)

   MDS()

   mds = MDS(n_components=2, dissimilarity="precomputed", random_state=1)

   pos = mds.fit_transform(distance)
```

Le code ci-dessus exploite la fonction cosine_similarity du module Scikit Learn, que l'on applique à tous les vecteurs en sortie de TF-IDF Vectorizer. Puis, pour obtenir une traduction de ces vecteurs dans un graphique à deux dimensions, le programme fait appel à l'algorithme MDS, servant à opérer une réduction dimensionnelle sur chaque indice de similarité.

Enfin, la dernière étape cruciale est de regrouper les documents sous un nombre choisi de "clusters", et en extraire les features caractéristiques.

K-MEANS : CLASSIFICATION NON-SUPERVISÉE

K-Means, ou partitionnement en K-moyennes en français est un algorithme de classification, qui, pour un entier donné K et un ensemble de points résoudra le problème de les “découper” en K groupes (souvent appelés “clusters”, d’où l’emploi du terme clusterisation). Il est souvent utilisé pour la classification de données non-labellisées, soit une classification non-supervisée. C’est le cas dans NLPvs.

Le code ci-dessous sert à appeler la version de K-Means de Scikit Learn (s’exécutant sur le CPU), et l’entraîner sur le corpus vectorisé de la section précédente. On y définit également le nombre “n” de clusters.

```
modele_custom = skcl.KMeans(n_clusters=n, init='k-means++').fit(corpus_vect)
```

Dans le cas où l’utilisateur choisit de créer un modèle accéléré par CUDA, et s’il possède le hardware compatible, une ligne de code similaire s’exécute à la place, faisant appel cette fois-ci à la version de K-Means de CuML, qui s’exécute sur un GPU Nvidia, en exploitant ses coeurs CUDA.

```
modele_custom = cucl.KMeans(n_clusters=n, init='k-means++').fit(corpus_vect)
```

Ici, le but de K-Means est de trouver K centroïdes dans l’espace géométrique des éléments à clusteriser (ici, les textes vectorisés), et pour chaque centroïde, évaluer la similarité des descripteurs (les termes contenus dans la matrice de chaque document) entre tous les éléments les plus proches de ce centroïde. Le but est d’obtenir pour chaque ensemble la similarité la plus forte possible, et donc atteindre un minimum de divergence entre les features qui définissent les vecteurs des différents textes.

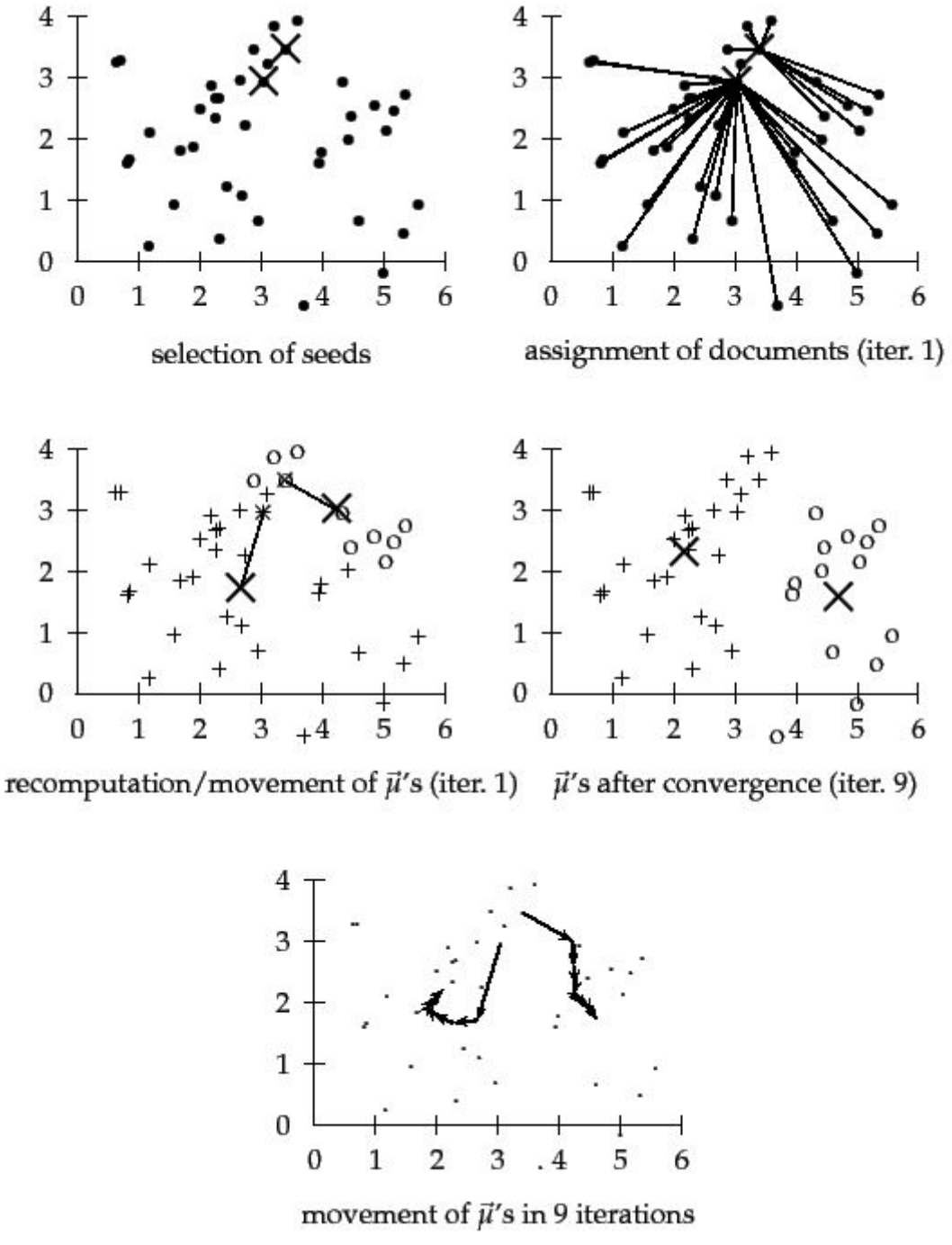
À chaque itération, les coordonnées des centroïdes seront ajustées de manière à atteindre à l’issue de suffisamment d’itération une convergence, un minimum local de la divergence mentionnée ci-dessus.

L’étape ayant la plus importante de l’algorithme est le choix initial de la position des centroïdes, puisqu’ils vont déterminer quel minimum local sera atteint (cette étape est par ailleurs appelée “initialisation”). Plusieurs hypothèses sont possibles pour trouver ces centroïdes, notamment celle du partitionnement aléatoire, qui va choisir de manière aléatoire (avec un “seed” fixe) chaque centroïde parmi les points en entrée, et ensuite procéder aux itérations, ou encore la méthode de Forgy, qui choisit les centroïdes depuis le point central entre k points en entrée choisis aléatoirement (k = nombre de centroïdes).

Seulement, ces méthodes ont le défaut de reposer sur un “seeding” libre, c’est à dire sans considérer le risque que deux centroïdes soient trop proches les uns des autres, ce qui fausserait les minimums locaux trouvés lors de la convergence.

C’est pourquoi NLPvs choisit la méthode “K-Means++” comme méthode d’initialisation. Cette dernière est qualifiée de “careful seeding”, car elle effectue un placement des centroïdes avant de commencer les itérations de K-Means. Pour ce faire, un seul centroïde est choisi au lancement, puis la distance entre ce centroïde et tous les autres points est calculée. Les centroïde suivant seront choisis parmi tous ces autres points, toujours de manière aléatoire, mais selon une probabilité proportionnelle au carré de leur distance du premier centroïde. Ensuite, une fois tous les centroïdes établis, l’algorithme procède aux itérations.

Cette méthode empêche les centroïdes d'être trop rapprochés les uns des autres, et donc de se chevaucher, et augmente de ce fait la probabilité que le minimum local lors de la convergence soit le minimum global de la fonction.



Fonctionnement général de K-means appliqué au NLP
Source : https://nlp.stanford.edu/IR-book/html/htmledition/k-means-1.html

Une fois la clusterisation effectuée, le programme récupère la liste des features de chacun des clusters, et va restituer les premiers termes de cette liste.

Ces termes sont les plus communs parmi les différents documents du cluster (et qui ont influencé le plus le calcul de similarité à la base de la constitution de ce cluster). Dans la fonction ci-dessous, les quatres premiers ont été choisis :

```
for ind in order_centroids[i, :4]:
            mot = (' %s' % vocab.loc[tm[ind].split('
')].values.tolist()[0][0]).encode('ascii','ignore').decode('utf-8','ignore')

            mots_par_clusters.append(mot)
```

Le rôle de la variable mot est de récupérer chaque terme (ind) d'indice 0 à 3, et de rechercher dans la Data Frame créée lors de la "tokenisation" (cf. De la chaîne de caractères aux tokens) son équivalent complet, puisque les termes en sortie de K-means sont toujours racinisés. Ainsi, c'est le terme complet qui est restitué à l'utilisateur, et qui figurera sur le diagramme sortant du programme.

Enfin, une couleur choisie aléatoirement sera attribuée à chaque cluster dans cette représentation graphique, de manière à ce que chaque couleur soit distincte (la plus éloignée de la précédente choisie).

SAUVEGARDE DES MODÈLES ET RÉUTILISATION

Une fonctionnalité intéressante de NLPvs est la sauvegarde d'un modèle pré-entraîné sur un certain corpus, et de pouvoir le réutiliser sur un autre corpus, afin d'économiser du temps de calcul, puisqu'il n'y a pas à calculer la position des centroïdes à nouveau, ni à constituer un nouveau vocabulaire pour la vectorisation.

```
joblib.dump(modele_custom,(path + '/model.sav'))
joblib.dump(tfconv,(path + '/vect.sav'))
joblib.dump(n,(path + '/nb_clust.sav'))
joblib.dump(vocab,(path + '/vocab.sav'))
joblib.dump(termes,(path + '/termes.sav'))
```

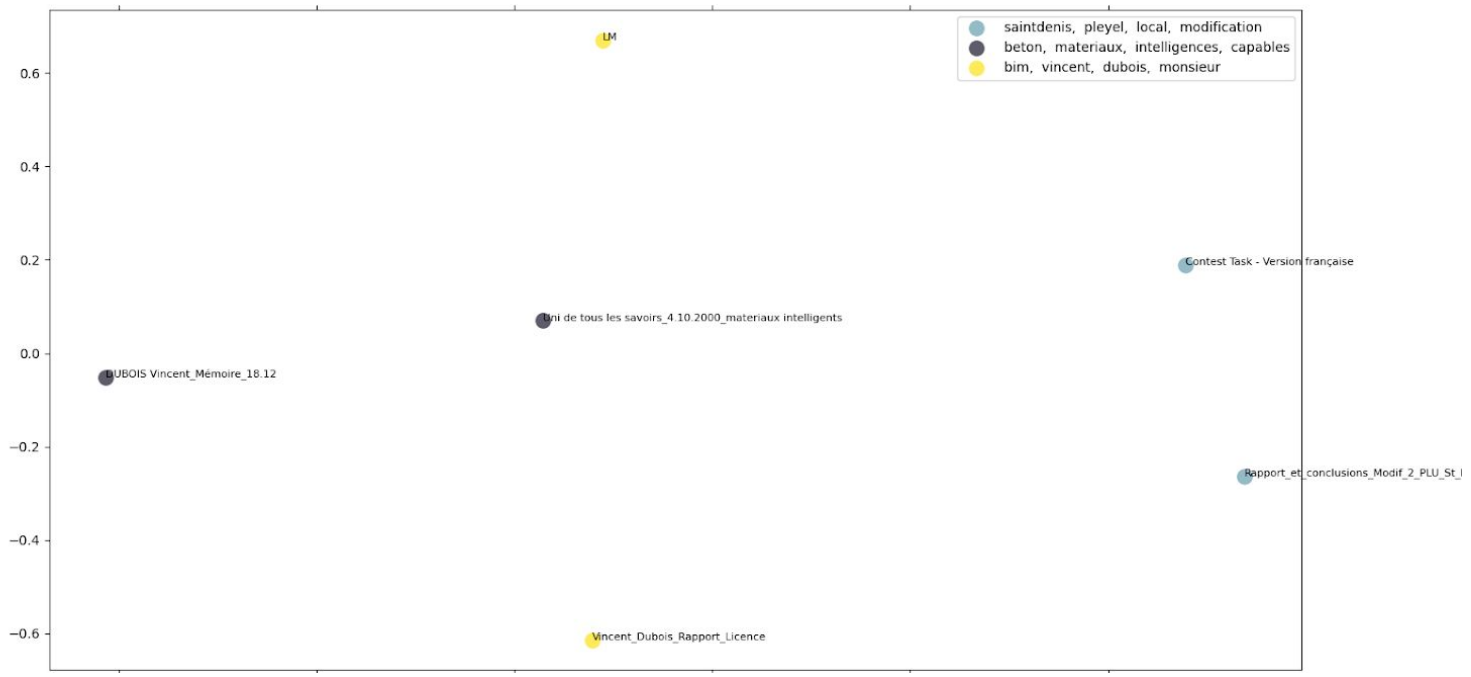
Chaque modèle créé est automatiquement sauvegardé dans le répertoire /MODEL sous un sous-dossier nommé selon la date du jour et s'il est configuré pour être exécuté sur le CPU ou le GPU. Ce sous-dossier contient:

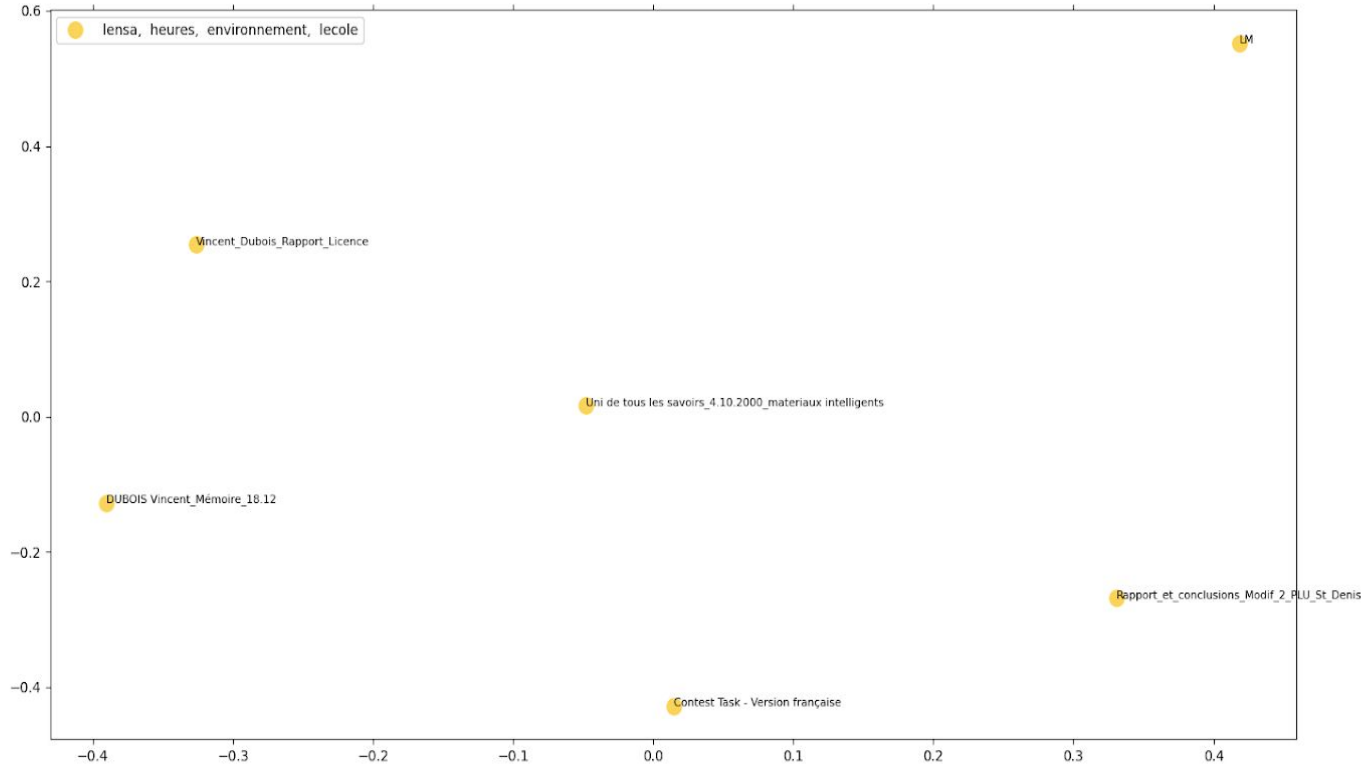
- model.sav, une sauvegarde du modèle K-Means pré-entraîné, contenant les données des centroïdes extraites du corpus d'origine.
- vect.sav, une sauvegarde du vectoriseur (TF-IDF), servant à vectoriser le corpus cible à partir du même vocabulaire que le corpus d'origine
- nb_clust.sav, le nombre de clusters demandé lors de l'entraînement du modèle K-Means
- vocab.sav, la Data Frame de sauvegarde des "tokens" racinisés et leur équivalent complet
- termes.sav, le vocabulaire complet du vectoriseur TF-IDF (la liste des termes sans les indices TF-IDF).

Ainsi, il est possible de les recharger plus tard, et d'employer seulement la fonction predict() de K-Means. Cette fonction permet d'affecter chacun des textes du nouveau corpus (qui sera vectorisé avec le même vectoriseur que le corpus d'origine) au cluster le plus proche parmi ceux stockés dans le modèle et générés à partir du corpus d'origine.

```
for texte in txt_vec:

    resultat = modele.predict(texte)
```





Les graphiques de la page ci-dessus montrent les différences entre la classification d'un même corpus de documents, avec deux modèles différents.

Dans le premier cas, un nouveau modèle a été créé et entraîné spécifiquement pour ce corpus de document à classifier. Les clusters et les vecteurs de chaque textes ont été générés en conséquence.

Dans le second cas, le corpus à classifier a été traité par un modèle pré-entraîné sur un corpus différent. On peut observer deux phénomènes :

- Tous les documents appartiennent à un même cluster. Dans le cas présent, le corpus qui a servi à générer les clusters étant très différent sémantiquement de celui que l'on veut classifier, l'attribution aux différents clusters n'est pas représentative du contenu des documents.

  Il est plus judicieux de réutiliser un modèle pré-entraîné sur un corpus sémantiquement assez proche de celui d'origine.

- La position des documents vis à vis des autres est différente. C'est dû au fait qu'il faille réutiliser le même vectoriseur qui a servi pour le corpus d'origine pour vectoriser le nouveau corpus. Cela provoque la non-prise en compte de tous les "tokens" ne faisant pas partie de son vocabulaire. Les vecteurs associés à chaque document en sont affectés, ce qui explique la différence de placement.

  Si le vocabulaire du corpus d'origine est bien plus pauvre que celui du nouveau corpus (ce qui est le cas ici), la position des documents sera altérée en conséquence. En revanche, l'inverse ne poserait pas de soucis, puisqu'il y aurait moins de risque de tomber sur un "token" inconnu du vocabulaire.

DOCUMENTATION

La plupart de la documentation concernant le code source et les aspects technologiques est accessible via le dépôt GitHub du projet NLPvs (cf lien ci-dessous). Néanmoins, les ressources web concernant l'approfondissement des notions clés présentées dans ce rapport sont listées ci-dessous :

- GitHub : Dépôt du projet NLPvs
  https://github.com/VincDub/NLPvs

- Université de Stanford, "K-Means"
  https://nlp.stanford.edu/IR-book/html/htmledition/k-means-1.html

- Université de Stanford, "Tokenization"
  https://nlp.stanford.edu/IR-book/html/htmledition/tokenization-1.html

- Scikit Learn : "TfidfVectorizer documentation"
  sklearn.feature_extraction.text.TfidfVectorizer — scikit-learn 0.23.0 documentation

- MonkeyLearn, : "What is tf-idf ?"
  https://monkeylearn.com/blog/what-is-tf-idf/

- Wikipédia US : "k-means++":
  https://en.wikipedia.org/wiki/K-means%2B%2B