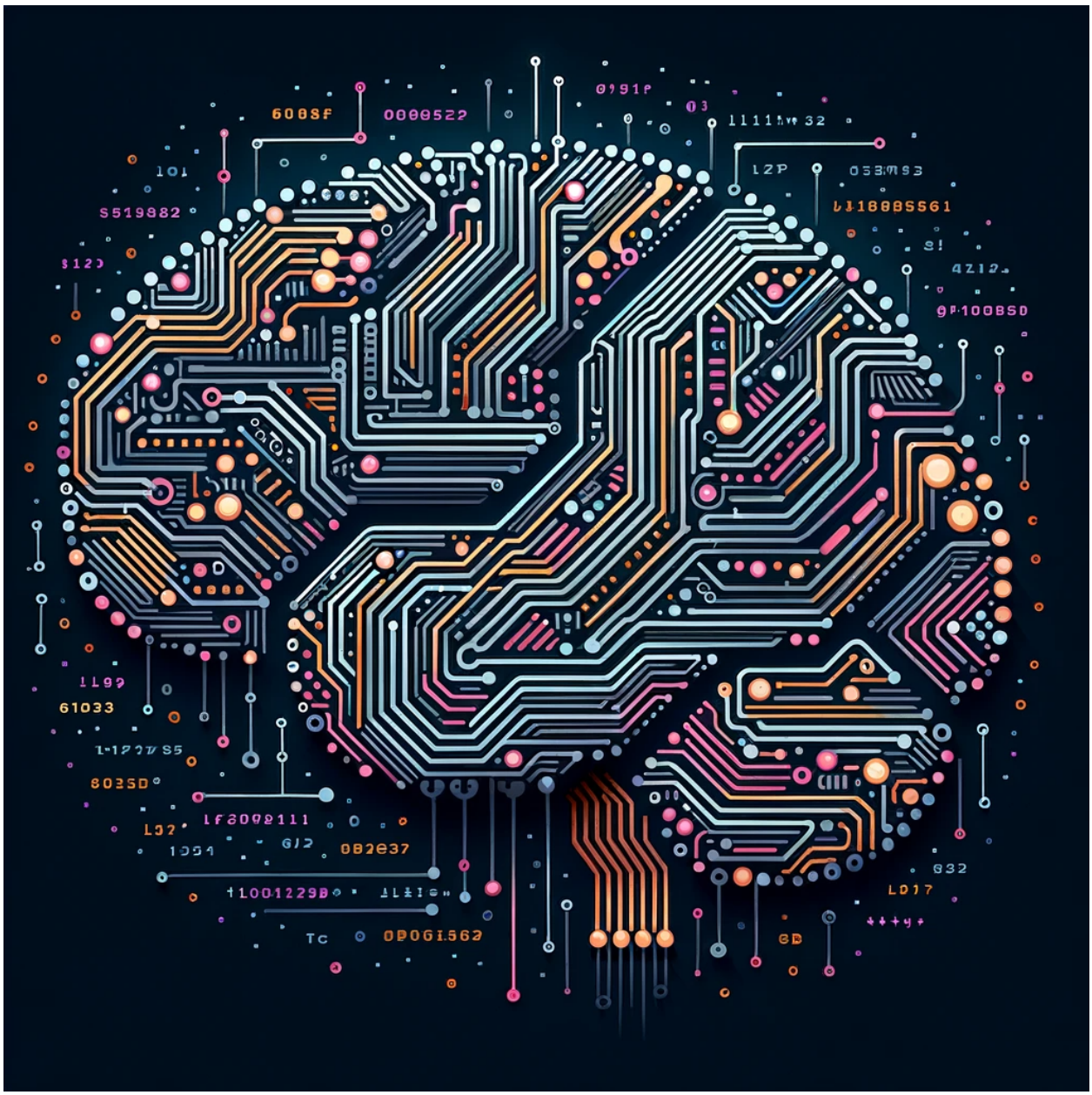
Traitement Automatique de langue Naturelle

NLP TD

INSÉREZ VOTRE TEXTE ICI



Partie 1: Text classification: prédire si la vidéo est une chronique comique

Tasks

Run la pipeline evaluate "python src/main.py evaluate". Elle devrait marcher sur la task "is_comic_video":

→ réponse Got accuracy 91.49497487437186%

```
pip install nltk
```

Installation du corpus français et des stopwords

```
nltk.download('punkt')
nltk.download('stopwords')
nltk.download('snowball_data')
```

Choix du Feature à partir de sklearn.feature_extraction.text:

Une petite recherche sur l'utilisation de chacune des features.

CountVectorizer : (par défaut)

- Meilleur pour : Les tâches où les fréquences brutes des termes (c'est-à-dire les comptes) sont importantes. Utile également pour les modèles de sacs de mots de base.
- Utilisations : Analyse de sentiment, modélisation de sujets, ou toute tâche où la simple présence ou absence d'un mot est significative.

TfidfVectorizer :

- Meilleur pour : Lorsque vous souhaitez prendre en compte l'importance d'un terme par rapport à sa fréquence dans l'ensemble des documents. Il contribue à réduire le poids des termes qui se produisent très fréquemment, les considérant moins informatifs que les termes qui se produisent dans une petite fraction du corpus.
- Utilisations : Classification de documents, regroupement, recherche d'informations et autres tâches où les mots qui sont fréquents dans un document, mais pas dans l'ensemble des documents, peuvent être plus informatifs.

HashingVectorizer :

- Meilleur pour : Les grands ensembles de données où l'utilisation de la mémoire est une préoccupation. Il est très efficace en termes de mémoire car il ne stocke pas le vocabulaire, mais il n'est pas aussi interprétable.
- Utilisations : Classification de texte ou regroupement avec de très grands ensembles de données. Des situations où vous avez besoin d'un vecteur de sortie de taille fixe, quel que soit le nombre de mots uniques dans l'entrée. Idéal lorsque la scalabilité est une préoccupation. Limitations : Le principal inconvénient est que vous ne pouvez pas récupérer les noms de fonctionnalités d'origine, ce qui rend le modèle plus difficile à interpréter. De plus, il existe une possibilité (bien que faible)

de collisions de hachage, où différents mots sont mappés sur la même valeur de hachage.

TfidfTransformer :

- Meilleur pour : Lorsque vous avez déjà une matrice de comptage (à partir de CountVectorizer ou d'une autre méthode) et que vous souhaitez lui appliquer une mise à l'échelle TF-IDF.
- Utilisations : Utilisations similaires à TfidfVectorizer, mais spécifiquement lorsque vous souhaitez un processus en deux étapes consistant d'abord à obtenir des comptages bruts, puis à les transformer.

Je décide de partir sur **TfidfVectorizer** car après quelques textes, me semble plus concluant à réaliser notre tâche.

Utilisation d'un tokenizer:

L'utilisation d'un tokenizer dans notre cas semble détériorer la performance de notre modèle. Il semble plus judicieux de le laisser de côté, du moins pour le moment.
La fonction pour stem notre texte se trouve dans src.features.extractions.

La fonction garde seulement la racine d'un mot, par exemple (manger, mangeant) devient mang.

Utilisation de **strip_accents**

L'utilisation de strip_accents à l'air d'avoir en général un effet positif sur notre modèle. Il permet comme son nom l'indique d'enlever les accents d'un mot. Ex: général → general

Choix du modèle:

Jusqu'ici j'ai ajouté / enlevé mes hyper-paramètres à la main. Mais il est possible pour accélérer la recherche d'utiliser un **gridsearch** qui s'occupera de tester toutes les combinaisons possibles.
Note : pour l'instant skipped

Après avoir fait des tests à la main, je me suis rendu compte qu'un **RandomClassifier** faisait de très bons résultats avec un un **taux d'accuracy > 92%**.