Rapport projet Embeddings et classification de sentiments

Info804

Huynh Raphaëlle - Amarouche Samy

Boudra

22 avril 2025

SOMMAIRE

Introduction

Fondements théoriques

La représentation vectorielle des mots ou *word embedding*, désigne un ensemble de techniques permettant de transformer les mots issus du langage naturel en vecteurs numériques continus dans un espace à N dimensions. Cette transformation a pour but de manipuler du langage humain, naturellement ambigu et non structuré, sous une forme permettant aux algorithmes d'exploiter ces données.

L'idée principale est que la proximité géométrique dans l'espace vectoriel encode la proximité sémantique : deux mots aux significations proches ou ayant des usages similaires dans des phrases auront des vecteurs proches les uns des autres selon une métrique donnée (souvent la distance cosinus ou euclidienne).

Il existait différentes méthodes pour représentés les mots avant les embeddings, elles étaient représentées sous forme symbolique que nous approfondirons plus tard. Ces méthodes symboliques étaient très limités donc pour pallier à cela des méthodes d'apprentissage de représentations distribuées (les embeddings) ont été développées. Elles consistent à apprendre un vecteur dense (par exemple en 50, 100 ou 300 dimensions) pour chaque mot, de manière à capturer à la fois sa syntaxe et sa sémantique.

Ces modèles apprennent la représentation d'un mot en fonction des mots qui l'entourent. Deux variantes principales existent :

- **CBOW (Continuous Bag of Words)** : prédit un mot cible à partir de son contexte environnant.
- **Skip-Gram** : fait l'inverse, c'est-à-dire prédit le contexte à partir d'un mot donné.

Par exemple, avec la phrase : « le chat dort sur le canapé », le modèle Skip-Gram prend « chat » comme mot cible, et essaie de prédire les mots « le », « dort », « sur ».

Modèles basés sur la co-occurrence

Ces méthodes construisent une matrice de co-occurrence (combien de fois deux mots apparaissent dans une même fenêtre de texte), puis factorisent cette matrice. Le plus connu de ces modèles est GloVe (Global Vectors for Word Representation), qui cherche à apprendre des vecteurs tels que la relation entre deux mots soit encodée dans une simple opération vectorielle.

Les embeddings possèdent plusieurs propriétés algébriques et sémantiques intéressantes :

- **Similarité sémantique** : les mots « voiture », « véhicule », « automobile » seront proches les uns des autres.
- **Analogies linguistiques** : les relations comme « Paris est à la France ce que Rome est à l'Italie » peuvent être modélisées par des relations vectorielles.
- **Compactness** : les vecteurs sont de faible dimension, facilitant le calcul et la visualisation.
- **Réutilisabilité** : les embeddings appris sur un grand corpus (ex : Wikipedia, Common Crawl) peuvent être réutilisés dans de nombreuses tâches (classification, traduction, etc.).

Dans le cadre de notre projet nous allons approfondir ce qu'est cbow et skipgram.

Le modèle CBOW : prédiction d'un mot à partir de son contexte

Le modèle CBOW repose sur une intuition simple : un mot peut être déduit de son environnement. L'idée consiste donc à prédire un mot cible en se basant sur les mots qui l'entourent, appelés mots de contexte. Cette approche revient à modéliser la probabilité d'apparition d'un mot donné, à condition de connaître les mots adjacents.

Exemple

« Le chat dort sur le canapé. »

Si l'on souhaite prédire le mot "dort" avec une fenêtre de contexte de taille 2, le modèle prendra en entrée les mots : *Le*, *chat*, *sur*, *le*. Le mot cible à prédire est *dort*. Ces mots de contexte sont représentés par des vecteurs dans un espace dense, puis moyennés ou sommés afin de produire un vecteur unique de contexte global. Ce vecteur est ensuite utilisé pour prédire le mot central par le biais d'une fonction de probabilité, souvent une softmax sur l'ensemble du vocabulaire.

Le modèle CBOW est généralement plus rapide à entraîner, car chaque paire d'entrée produit une unique sortie. De plus, il est relativement robuste au bruit présent dans le corpus, en raison de la combinaison des vecteurs de contexte, qui atténue l'influence de mots non informatifs. Toutefois, cette approche présente une limite importante : elle néglige l'ordre des mots, ce qui peut réduire la richesse de l'information capturée dans certains contextes syntaxiques.

Le modèle Skip-Gram : prédiction du contexte à partir d'un mot

Le modèle Skip-Gram, quant à lui, inverse la logique du CBOW. Il prend un mot cible comme entrée et cherche à prédire les mots de son contexte. En d'autres termes, il modélise la probabilité que des mots apparaissent autour d'un mot donné dans une fenêtre de taille fixée.

Exemple

« Le chat dort sur le canapé. »

Le mot cible étant *dort* et la fenêtre de taille 2, le modèle générera plusieurs paires d'entraînement :

- (*dort*, *Le*)
- (*dort*, *chat*)
- (*dort*, *sur*)
- (*dort*, *le*)

Chaque paire correspond à une tâche de prédiction indépendante, où le modèle doit deviner un mot de contexte à partir du mot central. Cette méthode augmente considérablement la quantité de données disponibles pour l'apprentissage, ce qui constitue un avantage notable, surtout pour les mots rares. En effet, même un mot peu fréquent peut être associé à plusieurs contextes, ce qui améliore la qualité de sa représentation vectorielle.

Néanmoins, cette multiplication des paires augmente aussi le temps de calcul requis, rendant le modèle plus coûteux en ressources, en particulier sur de très grands corpus. De plus, comme CBOW, Skip-Gram ne tient pas compte de l'ordre des mots dans le contexte.

Nous approfondirons la partie mathématique et la comparaison de ces 2 modèles plus tard.

Histoire et évolution des Embeddings au court du temps

Avant la création des *embeddings*, les mots étaient représentés sous forme symbolique ce qui posait plusieurs problèmes majeurs.

One-hot encoding

Dans cette méthode chaque mot est représenté par un vecteur binaire de dimension égale à la taille du vocabulaire. Un seul élément est activé à la position correspondant au mot, tous les autres étant à zéro.

Bag-of-Words (BoW)

Cette méthode construit une représentation à partir de la fréquence d'apparition de chaque mot dans un document. Le vecteur résultant est aussi de dimension et ne prend pas en compte l'ordre des mots. Il ne capture donc pas la syntaxe ni les dépendances grammaticales.

TF-IDF (Term Frequency – Inverse Document Frequency)

Cette méthode améliore BoW en pondérant chaque mot par sa fréquence dans un document et par sa rareté dans le corpus global. Bien qu'elle apporte une certaine

finesse, elle reste une représentation statique, sans réelle compréhension contextuelle du mot.

Latent Semantic Analysis (LSA) – Années 1990 à ~2003

L'une des premières tentatives de capturer les relations sémantiques entre les mots à partir d'un corpus est le Latent Semantic Analysis (LSA). Cette méthode repose sur la construction d'une matrice de co-occurrence des mots (ou des termes) dans les documents. L'objectif est ensuite de réduire la dimension de cette matrice via une décomposition en valeurs singulières (SVD).

LSA cherche à projeter les mots dans un espace sémantique latent où les similarités de sens sont reflétées par la proximité vectorielle. Bien que novateur, LSA présente plusieurs limites : il est linéaire (ne capture pas la non-linéarité des relations linguistiques), statique (ne prend pas en compte le contexte dynamique), et peu scalable aux grands corpus.

Cependant, cette approche a jeté les bases de la vectorisation des mots et de la sémantique distributionnelle, un principe fondamental du TAL : *« le sens d'un mot est donné par le contexte dans lequel il apparaît »*.

Neural Probabilistic Language Model (Bengio et al., 2003)

En 2003, Yoshua Bengio et ses collègues ont introduit un modèle révolutionnaire : le Neural Probabilistic Language Model (NPLM). Pour la première fois, un réseau de neurones était utilisé pour prédire la probabilité d'un mot donné son contexte, tout en apprenant une représentation vectorielle dense de chaque mot.

Ce modèle est considéré comme l'un des tout premiers modèles de language modeling neuronal. Il résout plusieurs problèmes des méthodes précédentes :

- Il apprend conjointement les vecteurs de mots et la fonction de probabilité.
- Il permet une généralisation plus fine aux contextes jamais vus pendant l'entraînement.

Cependant, en pratique, ce modèle était encore trop coûteux en calcul à l'époque pour des applications à grande échelle.

Word2Vec (Mikolov et al., 2013)

Le véritable tournant a lieu en 2013 avec l'introduction de Word2Vec par Tomas Mikolov et ses collègues chez Google. Ce modèle a permis une révolution dans l'apprentissage des embeddings, grâce à une combinaison d'efficacité computationnelle et de performance sémantique.

Word2Vec se décline en deux architectures principales : CBOW (Continuous Bag of Words) et Skip-Gram, que nous avons abordées plus tôt. Ces modèles sont non

supervisés, entraînés sur de grands corpus textuels, et produisent des vecteurs dans lesquels la proximité géométrique reflète la similarité sémantique. Word2Vec a notamment rendu célèbre l'idée que des relations sémantiques peuvent être capturées par des opérations vectorielles :

vec(roi) - vec(homme) + vec(femme) ≈ vec(reine)

Les vecteurs produits sont statiques (un mot = un vecteur unique), mais Word2Vec a posé les fondations des modèles d'embeddings modernes.

GloVe (Global Vectors for Word Representation) – Pennington et al., 2014

En 2014, les chercheurs de Stanford (Pennington, Socher et Manning) ont proposé GloVe, une alternative à Word2Vec, visant à combiner les avantages des approches basées sur la co-occurrence globale (comme LSA) et des modèles prédictifs (comme Word2Vec).

GloVe construit une matrice de co-occurrence entre les mots et cherche à approximer les rapports de co-occurrence en factorisant cette matrice selon une fonction log-linéaire. Contrairement à Word2Vec qui traite les contextes localement, GloVe exploite la globalité du corpus pour capturer les relations sémantiques.

Les vecteurs produits sont tout aussi efficaces que ceux de Word2Vec, avec une structure plus explicite, notamment pour des tâches d'analogie ou de classification sémantique.

FastText – Facebook AI Research, 2016

En 2016, Facebook AI Research (FAIR) a introduit FastText, une extension de Word2Vec qui inclut des informations morphologiques. Chaque mot est représenté non seulement par un vecteur mais également par les vecteurs de ses n-grammes de caractères.

Cela permet à FastText :

- De mieux gérer les mots rares ou inconnus (OOV – Out Of Vocabulary),
- De capturer des similarités morphologiques utiles dans des langues riches (comme l'allemand ou le turc),
- D'avoir une meilleure performance sur des corpus moins volumineux.

FastText est aujourd'hui encore utilisé dans des contextes industriels, notamment pour des systèmes de recommandation ou de classification de texte.

Embeddings contextuels : ELMo, BERT et au-delà (à partir de 2018)

L'une des plus grandes limites des modèles précédents (Word2Vec, GloVe, etc.) est qu'ils attribuent **un vecteur unique par mot**, quelle que soit sa signification dans le

contexte. Or, de nombreux mots sont polysémiques (ex. *banc* : meuble ou institution financière).

C'est pourquoi, à partir de 2018, une nouvelle génération de modèles dits contextualisés a émergé :

- ELMo (Embeddings from Language Models) : introduit des embeddings dynamiques issus d'un modèle LSTM bidirectionnel. Chaque mot a un vecteur qui dépend du contexte dans la phrase.
- BERT (Bidirectional Encoder Representations from Transformers, 2018) : révolutionne le domaine en utilisant une architecture Transformer, permettant d'apprendre des représentations contextuelles très riches. BERT est pré-entraîné sur de grandes quantités de textes (via des tâches de masquage) puis adapté à des tâches spécifiques (classification, QA, etc.).

Avec BERT et ses successeurs (RoBERTa, ALBERT, GPT, T5, etc.), les représentations vectorielles deviennent profondes, dynamiques et contextualisées, ouvrant la voie à des applications bien plus performantes, du résumé automatique à la traduction neuronale, en passant par la génération de texte.

Résumé

Année	Méthode	Description
2003	**Latent Semantic Analysis (LSA)**	Utilise une décomposition SVD de la matrice de co-occurrence.
2008	**Neural Probabilistic Language Model (Bengio et al.)**	1er réseau neuronal pour prédire la probabilité d'un mot donné un contexte.
2013	**Word2Vec (Mikolov et al.)**	Utilise Skip-Gram et CBOW pour apprendre rapidement des vecteurs sémantiques.
2014	**GloVe (Pennington et al.)**	Modèle global basé sur la co-occurrence, entraîné via une factorisation.
2018	**Contextual embeddings (ELMo, BERT, etc.)**	Représentation dynamique dépendant du contexte dans la phrase.

Principes mathématiques sous-jacents

Calcul de probabilités conditionnelles

Dans les modèles de type CBOW (Continuous Bag of Words) ou Skip-Gram, l'objectif est d'estimer des probabilités conditionnelles entre les mots et leur contexte.

- CBOW cherche à estimer la probabilité du mot cible *wt* donné un contexte (environnant) :

$P(wt \mid wt-k, \ldots, wt+k)$

- Skip-Gram, à l'inverse, cherche à prédire le contexte à partir du mot central :

$P(wt-k, \ldots, wt+k \mid wt)$

L'objectif de l'apprentissage est donc d'ajuster les poids d'un petit réseau de neurones afin de maximiser ces probabilités.

Fonction de perte

Une fonction de perte très utilisée est la négation de la log-vraisemblance, notamment dans Skip-Gram :

$L = -\sum_{t=1 \text{ à } T} \log P(w\_context \mid wt)$

Cependant, le calcul du softmax sur l'ensemble du vocabulaire est coûteux, car il nécessite une normalisation sur tous les mots possibles. Pour pallier cela, deux stratégies sont souvent utilisées :

- Negative Sampling : on ne met à jour que quelques mots négatifs (tirés au hasard) au lieu de tout le vocabulaire.
- Hierarchical Softmax : utilise une structure en arbre binaire pour réduire le coût de calcul du softmax.

Descente de gradient

L'apprentissage repose sur la descente de gradient stochastique (SGD). Le processus suit les étapes suivantes :

1. Initialisation des poids (vecteurs de mots) aléatoirement.
2. Pour chaque itération :
    - Calcul de la perte sur un exemple (ou mini-lot).
    - Calcul du gradient de la perte par rapport aux vecteurs.
    - Mise à jour des vecteurs dans la direction qui minimise la perte.

La formule classique de mise à jour des poids est :

$\theta = \theta - \eta \cdot \nabla\theta L$

où :

- θ représente les paramètres du modèle (vecteurs de mots).
- η est le taux d'apprentissage.
- $\nabla\theta L$ est le gradient de la perte par rapport à ces paramètres.

Fonctionnement du projet

Structure du projet

Dans un premier temps nous allons présenter la structure de notre projet pour éclaircir son utilisation.

```
PROJET/
├── data/                    # Données d'entrée
├── src/                     # Code source principal
│   ├── lib/                 # Bibliothèque de fonctions
│   │   ├── __init__.py
│   │   ├── analogie.py          # Opérations d'analogie sémantique
│   │   ├── cosinus.py           # Similarité cosinus
│   │   ├── negative_sampling.py     # Stratégie de negative sampling
│   │   ├── subsampling.py           # Réduction de la fréq des mots fréquents
│   │   └── visual_embedding.py      # Visualisation des embeddings
│   │
│   └── model/               # Modèles de Word2Vec
│       ├── __init__.py
│       ├── ann.py               # Approximate Nearest Neighbors
│       ├── cbow.py              # Modèle CBOW
│       ├── skipgram.py          # Modèle Skip-Gram
│       ├── textprocessor.py         # Traitement du texte
│       └── utils.py             # Fonctions utilitaires
├── embeddings_cbow_10.npy         # Embeddings CBOW (10 %)
├── embeddings_cbow_30.npy         # Embeddings CBOW (30 %)
├── embeddings_skipgram_10.npy
├── embeddings_skipgram_30.npy
├── main.py                  # Point d'entrée principal (exécution script)
├── main.ipynb               # Notebook principal d’expérimentation
├── README.md                    # Présentation du projet
└── requirement.txt              # Liste des dépendances
```

Model contient nos différentes class objects que nous utilisons pour le projet. Il contient donc les class ANNClassifier ANN.py pour la classification, cbow.py pour notre embeddings cbow, skipgram.py pour notre embeddings skipgram et textprocessor.py qui contient une class pour réaliser le prétraitement des données.

Le dossier lib contient la plus part des fonctions demandés dans le projet et le fichier utils.py contient des fonctions utilitaires au bon fonctionnement du projet.

Les fichier embedding_etc.py contiennent l'enregistrement des modèles entrainés de nos embeddings sous différentes valeurs d'apprentissages en fonction de la taille du dataset chois par exemple 10 pour 10% du dataset.

Les fichiers Jupyter contiennent principalement des tests et les fichiers .py restant sont des fichiers d'exécution de code avec des pipelines prédéfinis.

Comment utiliser le projet

Voir le readme pour les prérequies.
Pour entrainer les embeddings :
python3/python train_cbow.py ou train_skipgram *il faut aussi modifier le contenu du fichier directement par exemple pour entrainer le modele avec 30% du dataset il faut modifier* corpus = corpus[:int(0.3 * len(corpus))] *0.3 étant 30% du dataset et modifier le nom de sauvegarde du modele d'embeddings*

Pour faire la classification simple :
python3/python train_ann_embedding.py ou train_ann_google.py en fonction de ce on veut étudier ou non les embeddings cbow / skipgram ou google. Pour les embeddings il faut modifier le fichier train_ann_embedding.py en mettant le modèle qu'on veut tester en remplaçant le nom du fichier par le bon fichier de l'embedding voulu skipgram_embeddings = np.load(« embeddings_cbow_30.npy ») ici le nom de la variable n'a pas d'importance car elle stock seulement l'embedding.
Pour entrainer le modele de google il faut modifier la même ligne corpus = corpus[:int(0.3 * len(corpus))] *0.3 étant 30% du dataset pour modifier le vocabulaire de l'embedding*.

Pour faire la classification multiple : Idem que la classification simple mais avec les fichiers train_annMulti_emb.py et train_annMulti_google.py

Préparation des données

Pipeline de prétraitement

Nous avons conçu une classe ***TextPreprocessor*** dédiée à cette tâche ce trouvant dans le fichier textprocessor.py . Ce pipeline suit plusieurs étapes clés :

- Normalisation du texte :
  - Passage en minuscules pour uniformiser le texte.
  - Suppression de tous les caractères spéciaux (ponctuation, chiffres, symboles) à l'aide d'expressions régulières :
- Tokenisation :
  - Découpage du texte en mots (tokens) via str.split().
  - Suppression des stopwords (mots très fréquents mais peu informatifs comme "the", "and", etc.), chargés une seule fois depuis nltk.corpus.stopwords.,
- Gestion des mots rares :
  - Calcul des fréquences d'apparition des mots avec collections.Counter.
  - Filtrage des mots ayant une fréquence inférieure à un seuil (min_freq=5), afin de réduire le bruit et la taille du vocabulaire.

Construction du vocabulaire

Une fois le texte nettoyé et filtré, nous avons généré deux dictionnaires pour gérer les représentations numériques :

- word2idx : associe un mot à un identifiant unique
- idx2word : associe un identifiant à son mot d'origine

Cela permet d'encoder et décoder les textes facilement pour les phases d'apprentissage et d'évaluation.

Prétraitement pour la classification

Dans le cadre de la classification de sentiments sur le jeu de données IMDB :

- Les textes ont été nettoyés par la fonction clean_text (similaire à TextPreprocessor mais adaptée au format CSV).
- Chaque texte est transformé en liste de mots.
- Les étiquettes ("positive" / "negative") sont converties en labels binaires (1 / 0).

Construction de vocabulaire et mapping des mots indice

Dans la classe Textprocessor

Un compteur (Counter) est utilisé pour calculer la fréquence de chaque mot dans la liste de tokens obtenue.

Ensuite, une liste filtered est construite en ne gardant que les mots ayant une fréquence supérieure ou égale à min_freq. Cela permet de filtrer les mots rares, qui ne sont pas inclus dans le vocabulaire final.

Le vocabulaire (self.vocab) est ensuite mis à jour avec ces mots filtrés sous forme d'un set.

Le word2idx (dictionnaire) est créé, qui associe chaque mot à un indice unique (calculé via enumerate sur la liste des mots triée). Cela permet de donner un indice unique à chaque mot dans le vocabulaire.

Le idx2word est l'inverse de word2idx, il associe chaque indice à son mot correspondant.

Génération des paires d'apprentissages

Les fonctions generate_cbow_data et generate_skipgram_data dans le fichier utils.py génèrent des paires de données d'apprentissage pour deux modèles d'apprentissage de représentations de mots : CBOW (Continuous Bag of Words) et Skip-gram. Voici comment chacune de ces fonctions génère les paires d'apprentissage .

Fonction generate_cbow_data (pour le modèle CBOW) :

Le modèle CBOW prédit un mot cible à partir de son contexte. Pour cela, on génère des paires de contexte (les mots voisins) et le mot cible (au centre de la fenêtre).

Pour chaque position dans le texte, le mot central (cible) est sélectionné.

Les mots voisins dans la fenêtre de taille window (de part et d'autre du mot cible) sont collectés comme contexte.

Chaque paire (contexte, cible) est ensuite ajoutée aux listes X et y :

X contient les mots du contexte, c'est-à-dire les indices des mots voisins.

y contient le mot cible, c'est-à-dire l'indice du mot central.

Le résultat est renvoyé sous forme de tableaux numpy X et y.

Exemple (avec window=2 pour une fenêtre de taille 2) : Si le texte encodé est [1, 2, 3, 4, 5], avec window=2 :

Pour le mot à l'indice 2 (cible = 3), les contextes sont les indices [1, 2, 4, 5].

La paire sera (context: [1, 2, 4, 5], target: 3).

Le modèle CBOW apprend à prédire le mot central (cible) à partir des mots de son contexte.

Fonction generate_skipgram_data (pour le modèle Skip-gram) :

Le modèle Skip-gram fonctionne dans l'autre sens par rapport à CBOW. Il utilise un mot central (cible) pour prédire ses mots contextuels. Ce modèle fonctionne également en utilisant une fenêtre de contexte.

Pour chaque mot dans la séquence de tokens encodés (encoded_tokens), ce mot devient le mot cible (target).

Une fenêtre de taille aléatoire est choisie autour du mot cible. Cette taille de fenêtre est choisie de manière aléatoire entre 1 et window, ce qui permet d'ajouter une certaine variabilité dans la taille de la fenêtre de contexte.

Pour chaque mot dans cette fenêtre, à l'exception du mot cible lui-même, une paire (cible, contexte) est générée.

La paire (target, context) est ajoutée à une liste pairs.

Enfin, la liste de paires est convertie en un tableau numpy et renvoyée.

Exemple (avec window=2 pour une fenêtre de taille entre 1 et 2, et un texte encodé [1, 2, 3, 4, 5]) :

Si le mot à l'indice 2 est choisi comme cible (3), une fenêtre aléatoire de taille 1 ou 2 est choisie autour de lui. Si la fenêtre aléatoire choisie est de taille 2, les contextes seront [2, 4] (pour un target=3).

Les paires seront (3, 2) et (3, 4).

Implémentation des techniques d'optimisation

Génération d'échantillons négatifs (generate_negative_samples) :

Cette fonction génère des échantillons négatifs pour l'échantillonnage négatif, une technique utilisée pour entraîner des modèles comme Word2Vec. Voici un aperçu détaillé de ton code pour cette fonction :

Entrée :

vocab_size : La taille du vocabulaire, nécessaire pour générer des exemples négatifs en choisissant au hasard un indice dans l'espace du vocabulaire.

positive_pairs : Liste de paires positives (target, context) représentant les relations réelles entre les mots cibles et les mots contextuels dans le corpus.

num_negative : Le nombre d'exemples négatifs à générer pour chaque paire positive.

Processus :

Ajout des paires positives : À chaque itération, la paire (target, context) est d'abord ajoutée aux échantillons comme étant positive, avec un label 1.

Génération des paires négatives :

Un negative_context est généré de manière aléatoire entre 0 et vocab_size. Si le mot négatif généré est le même que le mot cible (target), il est renvoyé et un autre échantillon est généré pour éviter cette répétition.

Ces échantillons négatifs sont ajoutés à la liste des paires avec un label 0.

Retour :

La fonction retourne une liste de paires, où chaque paire est associée à un label (1 pour les exemples positifs, 0 pour les négatifs), ce qui est essentiel pour l'entraînement de modèles utilisant la technique d'échantillonnage négatif.

Optimisation :

La génération des exemples négatifs optimise l'entraînement en réduisant le nombre de paires à traiter tout en maintenant une certaine diversité dans les exemples.

La vérification de la répétition (while negative_context == target) garantit que les exemples négatifs ne contiennent pas de contexte identique au mot cible, ce qui rend l'entraînement plus pertinent.

Sous-échantillonnage des mots fréquents (subsample_frequent_words) :

Cette fonction applique un sous-échantillonnage aux tokens du texte pour réduire la dominance des mots fréquents, ce qui est crucial pour l'entraînement efficace des modèles de langage. Voici l'analyse de cette fonction :

Entrée :

tokens : Liste des tokens du corpus.

threshold : Seuil pour déterminer quand un mot est considéré trop fréquent. Le seuil est fixé à 1e-5 par défaut.

Processus :

Calcul des fréquences : Le code utilise Counter pour compter la fréquence des mots dans tokens, puis calcule la fréquence totale du corpus.

Calcul des probabilités de conservation : La probabilité de conserver chaque mot est calculée à l'aide de la formule :

$$P(w) = (\sqrt{(t/f(w))} + 1) \times (t/f(w))$$

Cela réduit la probabilité de conserver les mots les plus fréquents et empêche leur domination dans le modèle.

Sous-échantillonnage : Chaque mot est comparé à un nombre aléatoire et est conservé en fonction de sa probabilité de conservation.

Retour :

La fonction retourne une liste des tokens après sous-échantillonnage, où les mots fréquents ont moins de chances d'être conservés.

Les Embeddings

Implémentation de cbow

Le modèle CBOW apprend à prédire un mot cible à partir des mots de son contexte. Le contexte est représenté par une fenêtre de mots autour du mot cible dans une séquence donnée.

Les étapes principales de l'implémentation :

Données d'entrée : La fonction data_generator est utilisée pour fournir des mini-lots de données (batchs) contenant des contextes (X) et des mots cibles (y).

Création du modèle : Le modèle CBOW utilise une couche d'embedding pour transformer les indices des mots en vecteurs denses, puis applique une réduction moyenne pour obtenir une représentation du contexte.

Sortie : La sortie du modèle est un vecteur représentant les probabilités des mots du vocabulaire, qui permet de prédire un mot cible en fonction du contexte.

Initialisation du modèle:

- vocab_size : La taille du vocabulaire, c'est-à-dire le nombre total de mots uniques dans ton corpus.
- embedding_dim : La dimension des vecteurs d'embedding, qui détermine la "richesse" de la représentation vectorielle.
- context_size : La taille de la fenêtre du contexte autour du mot cible (combien de mots autour du mot cible seront utilisés pour prédire ce dernier).
- use_ann : Un paramètre qui détermine si le modèle doit utiliser un réseau de neurones pour prédire la sortie (True par défaut, ce qui signifie qu'un perceptron multi-couches est utilisé).
- optimizer et learning_rate : Le type d'optimiseur à utiliser et le taux d'apprentissage.

Choix de l'optimiseur:

La fonction get_optimiseur retourne l'optimiseur sélectionné en fonction du paramètre optimizer_name. Elle permet de choisir entre SGD (stochastic gradient descent) et Adam.

Méthode _build_model :

- Embedding Layer : La couche Embedding transforme chaque mot du contexte (qui est un indice) en un vecteur dense de dimension embedding_dim.
- Lambda Layer : La couche Lambda est utilisée pour calculer la moyenne des vecteurs de contexte sur la dimension des mots, ce qui donne une représentation moyenne du contexte.
- Couche de sortie : Si use_ann=True, une couche Dense est ajoutée avec une activation softmax pour prédire la probabilité de chaque mot du vocabulaire comme étant le mot cible.
- Compilation du modèle : Si use_ann=True, le modèle est compilé avec une fonction de perte categorical_crossentropy (adaptée aux classifications multiclasse) et l'optimiseur sélectionné.

Méthode train :

- Vérifie que le modèle utilise un perceptron (si use_ann=False, il ne peut pas être entraîné).
- Appelle la méthode fit pour entraîner le modèle sur les données générées, sur un nombre spécifié d'époques.

Méthode get_embeddings :

Cette méthode permet d'extraire les poids de la couche d'embedding, qui sont les vecteurs d'embedding appris par le modèle pour chaque mot dans le vocabulaire.

Méthode summary :

Cette méthode renvoie un résumé du modèle, ce qui est utile pour inspecter l'architecture et les paramètres du modèle.

Implémentation de skipgram

Le modèle skipgram est un peu près similaire au modèle cbow la différence se porte surtout sur le calcul de similarité mais aussi sur les données entrées.

Entrées :

input_word : le mot cible (target).

context_word : le mot du contexte (context).

Embedding :

Les deux entrées passent par la même couche d'embedding (partagée).

Chaque mot est transformé en un vecteur de dimension embedding_dim.

Calcul de la similarité :

On calcule le produit scalaire entre les vecteurs du mot cible et du mot contexte via une couche Dot.

Le produit scalaire mesure à quel point les deux mots sont proches dans l'espace vectoriel.

Sortie :

La similarité passe dans une fonction d'activation sigmoïde.

La sortie est une probabilité entre 0 et 1 indiquant si le contexte est correct pour le mot cible.

Perte utilisée :

binary_crossentropy car il s'agit d'une classification binaire ("ce contexte est correct" ou "il ne l'est pas").

Visualisation des embeddings

La visualisation est disponible sur la présentation et le code dans le fichier visualisation_embedding.py on peut choisir de visualiser avec pca ou tsne.

Utilisation de la similarité cosinus

L'utilisation de la similarité cosinus entre les mots a été aussi utilisé dans le projet, on peut voir des rendus dans la présentation mais aussi le code dans le fichier cosinus.py

Entrainement et test

Dans un premier temps nous avons testé nos différentes fonctions et class dans le fichier test_embeddings.py afin de voir si la pipeline était fonctionnelle. Ensuite nous avons entrainé nos modèles d'embedding avec les fichiers train approprié à 10% du dataset et 30% du dataset et nous avons visualisé les résultats. Nous n'avons pas pu faire d'autres entrainements à cause du temps d'entrainement de nos modèles.

La classification

Architecture du modèle

Pour réaliser un modèle de classification nous avons créé une classe ANNClassifier disponible dans le fichier ann.py.

Ce code définit un classifieur de documents en utilisant un réseau de neurones artificiels (ANN : Artificial Neural Network) basé sur des vecteurs d'embedding de mots.

Chaque document est résumé sous forme d'un vecteur moyen des embeddings de ses mots, puis classé en deux classes (classification binaire).

Entrée :

Un document est représenté comme un vecteur unique obtenu en faisant la moyenne des vecteurs d'embedding de ses mots.

Architecture :

Entrée : un vecteur de taille embedding_dim.

Couche dense 1 : 128 neurones + activation ReLU.

BatchNormalization : normalise les activations pour accélérer l'entraînement et stabiliser le réseau.

Dropout : 40% des neurones sont désactivés pour éviter le surapprentissage.

Couche dense 2 : 64 neurones + activation ReLU.

BatchNormalization.

Dropout : 30%.

Sortie : 1 neurone avec activation sigmoïde pour prédire la probabilité d'appartenir à la classe 1.

Optimiseur :

Adam avec un taux d'apprentissage donné (learning_rate).

Perte utilisée :

binary_crossentropy, adaptée à la classification binaire.

ANNClassifier.__init__ :

Initialise le classifieur en créant le modèle (self.model) et en stockant la matrice d'embedding et le dictionnaire word2idx.

ANNClassifier._build_model :

Crée un modèle Sequential avec des couches Dense, BatchNormalization et Dropout.

Compile le modèle avec Adam et la fonction de perte binary_crossentropy.

ANNClassifier._document_to_vector :

Transforme un document (liste de mots) en un vecteur moyen de ses mots présents dans la matrice d'embedding.

Si aucun mot du document n'est trouvé, retourne un vecteur nul.

ANNClassifier.prepare_features :

Applique _document_to_vector à une liste de documents pour créer la matrice d'entrée pour le réseau (X).

ANNClassifier.train :

Coupe les données en ensemble d'entraînement/test.

Entraîne le modèle avec early stopping (arrêt anticipé si pas d'amélioration sur val_loss après un certain nombre d'époques).

Affiche les résultats avec classification_report (précision, rappel, f1-score).

Comparaison entre cbow et skipgram

Test réalisés – Embeddings avec 20 newGroups avec la classification et IMDB

- cbow

Pourcentage de données utilisés	Accuracy	Loss	Val_accuracy	Val_loss	Test_accuracy
10 %	0.5945	0.6642	0.5764	0.6759	0.58
30 %	0.7013	0.5683	0.6762	0.5974	0.67

- skipgram

Pourcentage de données utilisés	Accuracy	Loss	Val_accuracy	Val_loss	Test_accuracy
10 %	0.6608	0.6152	0.6585	0.6129	0.65
30 %	0.6727	0.6043	0.6732	0.6038	0.68

Comparaison avec l'embedding de google

Test réalisés – Avec Embedding de Google

Pourcentage de données utilisés	Accuracy	Loss	Val_accuracy	Val_loss	Test_accuracy
10 %	0.8289	0.3857	0.8399	0.3627	0.83
30 %	0.8334	0.3794	0.8331	0.3709	0.84
100 %	0.8330	0.3817	0.8439	0.3547	0.84

Conclusion

Nous avons pu conclure que le modèle cbow est meilleur avec les Database plus grande tandis que skipgram est plus performant avec de petit dataset notamment du au fait qu'il est meilleur pour la précision des mots rares. En revanche il est beaucoup plus a entrainé que le modèle cbow car il ne génère qu'une pair à la fois

Aspect	**CBOW**	**SkipGram**
Vitesse d'entraînement	Plus rapide (car moyenne du contexte)	Plus lent (une pair à la fois)
Besoin en données	Besoin de plus de données pour de bonnes représentations	Meilleur avec peu de données
Performance avec peu de données	Moins bon	Meilleur
Qualité des embeddings pour mots rares	Moins bon	Meilleur

La classification multiclasse

Architecture du modèle

Le but du modèle ANNClassifierMultiClass est de classer des documents ou des séquences de mots en plusieurs classes (par exemple : 3 classes si nb_classes=3).

Chaque document est représenté comme une séquence d'indices (les indices correspondent aux mots dans le dictionnaire word2idx).

Chaque indice est transformé en vecteur d'embedding à partir d'une matrice d'embedding pré-entraînée (embedding_matrix).

Architecture du modèle

Input Layer :

Reçoit une séquence d'indices (de longueur fixée input_length, souvent 100).

Embedding Layer :

Convertit chaque indice en son vecteur d'embedding.

trainable=False, les embeddings sont fixes (on n'ajuste pas les vecteurs pendant l'entraînement).

GlobalAveragePooling1D :

Prend tous les vecteurs d'un document et en fait une moyenne, produit un seul vecteur fixe par document.

Ça réduit la séquence à un vecteur unique, facile à traiter ensuite.

Dense Layer (64 neurones) :

Première couche de neurones fully connected.

Activation ReLU pour apprendre des représentations non-linéaires.

Dropout (0.5) :

Pour réduire l'overfitting en forçant le modèle à ne pas dépendre d'un petit sous-ensemble de neurones.

Dense Output Layer (nb_classes neurones) :

Produit un vecteur de taille égale au nombre de classes.

Activation softmax, transforme les scores en probabilités de classes.

Compilation du modèle

Optimiseur : Adam avec un learning_rate=1e-3.

Fonction de perte : sparse_categorical_crossentropy (adaptée aux étiquettes qui sont des entiers, pas des one-hot vectors).

Entraînement

Le jeu de données est divisé en :

Entraînement (train)

Validation (val)

Test (test)

Un early stopping est appliqué :

Surveille la val_loss.

Si la perte de validation ne diminue pas pendant patience epochs, l'entraînement s'arrête.

Les poids de classe peuvent être calculés automatiquement pour mieux traiter un déséquilibre entre les classes (compute_class_weight).

Préparation des données

Utilisation de pad_sequences pour que toutes les séquences aient exactement input_length mots.

Remplacement des indices trop grands pour éviter d'aller chercher un vecteur qui n'existe pas dans l'embedding.

Prédiction et Évaluation

Pour prédire :

Le modèle sort des probabilités pour chaque classe.

On prend la classe avec la probabilité la plus haute (np.argmax).

Pour évaluer :

Affichage de l'accuracy et de la perte (loss) sur un jeu de test.

Un rapport de classification (classification_report) est généré :

Précision, rappel, F1-score par classe.

Prétraitement des données

Chargement et Découpe des Documents

Les fichiers texte (.txt) présents dans le dossier data/archive sont lus un par un.

Chaque fichier représente un ensemble de phrases associé à un label :

Le label est déterminé par le nom du fichier sans l'extension (.txt).

Les documents sont découpés en phrases individuelles selon une règle basée sur la ponctuation :

Une séparation a lieu après un point (.) ou un point d'interrogation (?), suivi d'un espace.

La découpe est réalisé grâce à une expression régulière

Chaque phrase est considérée comme un document à part entière, associé au même label que le fichier d'origine.

Prétraitement Textuel

Le prétraitement est effectué via la classe TextPreprocessor avec un seuil de fréquence minimale fixé à 5 (min_freq=5).

Nettoyage et Tokenisation

Pour chaque phrase/document :

Passage en minuscules.

Suppression des signes de ponctuation.

Découpage du texte en mots/tokens.

Les résultats sont stockés sous forme de listes de tokens.

Construction du Vocabulaire

L'ensemble des tokens est rassemblé.

Le vocabulaire est filtré pour ne conserver que les mots apparaissant au moins 5 fois dans le corpus.

Chaque mot retenu reçoit un index unique (word2idx).

Encodage des Documents

Chaque phrase (tokenisée) est transformée en une séquence d'indices correspondant aux mots présents dans le vocabulaire.

Les mots absents du vocabulaire sont ignorés ou mappés à un index spécial (UNK, si géré par TextPreprocessor).

Construction des jeux de données

X : ensemble des documents encodés sous forme de séquences d'entiers.

y : ensemble des labels numériques correspondant à chaque document.

Un dictionnaire label2idx mappe chaque label textuel à un index numérique.

Séparation des données

Les données sont séparées de manière stratifiée :

80% des données pour l'ensemble d'entraînement + validation, 20% pour le test.

Puis 75% des données d'entraînement pour le train, et 25% pour la validation.

Cela garantit une répartition équilibrée des classes entre les ensembles.

Préparation finale pour l'entraînement

Les séquences encodées sont paddées :

Longueur cible : 100 tokens (input_length=100).

Complément avec des zéros pour les séquences plus courtes (padding après la séquence).

Troncature après 100 mots pour les séquences plus longues.

Calcul des poids de classes

Les poids sont calculés avec compute_class_weight pour compenser le déséquilibre entre les différentes classes.

Ces poids sont utilisés pendant l'entraînement pour pénaliser proportionnellement les erreurs sur les classes minoritaires.

Comparaison entre cbow et skipgram

Cbow 10%

	precision	recall	f1-score	support
0	0.08	0.68	0.15	10938
1	0.00	0.00	0.00	7904
2	0.00	0.00	0.00	4952
3	0.00	0.00	0.00	4847
4	0.00	0.00	0.00	4464
5	0.08	0.07	0.07	5954
6	0.03	0.25	0.06	3248
7	0.00	0.00	0.00	5596
8	0.00	0.00	0.00	4981
9	0.00	0.00	0.00	6003
10	0.11	0.00	0.00	6343
11	0.07	0.10	0.09	7839
12	0.00	0.00	0.00	5173
13	0.08	0.00	0.00	7290
14	0.07	0.03	0.04	6758
15	0.09	0.04	0.05	9616
16	0.17	0.00	0.00	8148
17	0.00	0.00	0.00	11825
18	0.00	0.00	0.00	9101
19	0.00	0.00	0.00	5750
accuracy			0.07	136730
macro avg	0.04	0.06	0.02	136730
weighted avg	0.04	0.07	0.03	136730

Cbow 30%

	precision	recall	f1-score	support
0	0.00	0.00	0.00	10938
1	0.00	0.00	0.00	7904
2	0.00	0.00	0.00	4952
3	0.00	0.00	0.00	4847
4	0.00	0.00	0.00	4464
5	0.00	0.00	0.00	5954
6	0.21	0.01	0.02	3248
7	0.00	0.00	0.00	5596
8	0.00	0.00	0.00	4981
9	0.00	0.00	0.00	6003
10	0.11	0.01	0.01	6343
11	1.00	0.00	0.00	7839
12	0.04	1.00	0.07	5173
13	0.00	0.00	0.00	7290
14	0.00	0.00	0.00	6758
15	0.00	0.00	0.00	9616
16	0.00	0.00	0.00	8148
17	0.15	0.00	0.00	11825
18	0.00	0.00	0.00	9101
19	0.00	0.00	0.00	5750
accuracy			0.04	136730
macro avg	0.08	0.05	0.01	136730
weighted avg	0.08	0.04	0.00	136730

Skipgram 10 %

	precision	recall	f1-score	support
0	0.00	0.00	0.00	10938
1	0.06	0.50	0.11	7904
2	0.00	0.00	0.00	4952
3	0.00	0.00	0.00	4847
4	0.00	0.00	0.00	4464
5	0.08	0.03	0.05	5954
6	0.24	0.01	0.02	3248
7	0.00	0.00	0.00	5596
8	0.00	0.00	0.00	4981
9	0.00	0.00	0.00	6003
10	0.00	0.00	0.00	6343
11	0.00	0.00	0.00	7839
12	0.00	0.00	0.00	5173
13	0.00	0.00	0.00	7290
14	0.06	0.10	0.08	6758
15	0.00	0.00	0.00	9616
16	0.00	0.00	0.00	8148
17	0.09	0.11	0.10	11825
18	0.07	0.32	0.12	9101
19	0.04	0.03	0.03	5750
accuracy			0.07	136730
macro avg	0.03	0.06	0.03	136730
weighted avg	0.03	0.07	0.03	136730

Skipgram 30%

	precision	recall	f1-score	support
0	0.00	0.00	0.00	10938
1	0.00	0.00	0.00	7904
2	0.00	0.00	0.00	4952
3	0.00	0.00	0.00	4847
4	0.00	0.00	0.00	4464
5	0.00	0.00	0.00	5954
6	0.05	0.07	0.06	3248
7	0.00	0.00	0.00	5596
8	0.00	0.00	0.00	4981
9	0.00	0.00	0.00	6003
10	0.00	0.00	0.00	6343
11	1.00	0.00	0.00	7839
12	0.00	0.00	0.00	5173
13	0.00	0.00	0.00	7290
14	0.06	0.06	0.06	6758
15	0.00	0.00	0.00	9616
16	0.00	0.00	0.00	8148
17	0.00	0.00	0.00	11825
18	0.07	0.92	0.13	9101
19	0.00	0.00	0.00	5750
accuracy			0.07	136730
macro avg	0.06	0.05	0.01	136730
weighted avg	0.07	0.07	0.01	136730

On peut constater que le modèle a énormément de mal à apprendre. Cela peut être du à la simplicité des embeddings ou bien même à un trop grand nombres de classes. On pourrait appliquer comme solution d'essayer avec des embeddings entrain avec plus de données ou bien de diminuer le nombre de classes et de les généralisés par exemple faire une catégorie sport au lieu de moto / baseball / hockey etc afin de faciliter la tâche de l'apprentissage. Nous n'avons pas eu le temps d'explorer cette solution qui nous paraît tout à fait pertinente.

Comparaison avec l'embedding de google

Nous n'avons pas pu faire tourner le script pour entrainer le modèle avec l'embedding de google dû au trop grand nombres de calculs à réaliser. Malgré l'essaie de plusieurs méthodes pour optimiser le code notamment en diminuant les batch_size au maximum en diminuant la input_lenght nous n'avons pas pu le faire tourner. Il fallait donc se tourner vers une solution avec une carte graphique que nous ne possédons pas. Nous aurions pu nous tourner vers une solution avec google collabs mais la connexion internet que nous avons ne nous a pas permis d'uploader les données assez rapidement. Juste pour le modèle d'embedding de google cela prenait 5h.

Conclusion

Il reste encore des solutions à explorer pour améliorer la classification multiclasse de notre modèle. Les résultats ne nous ont pas permis de faire un comparatif de performance entre les 2 modèles d'embeddings.

Les problèmes rencontrés

Disponible avec les solutions dans la présentation.