

Projet 7

Développez une preuve de concept

Parcours Ingénieur Machine Learning

Soutenance le 23 Octobre 2021

QITOUT Kenza

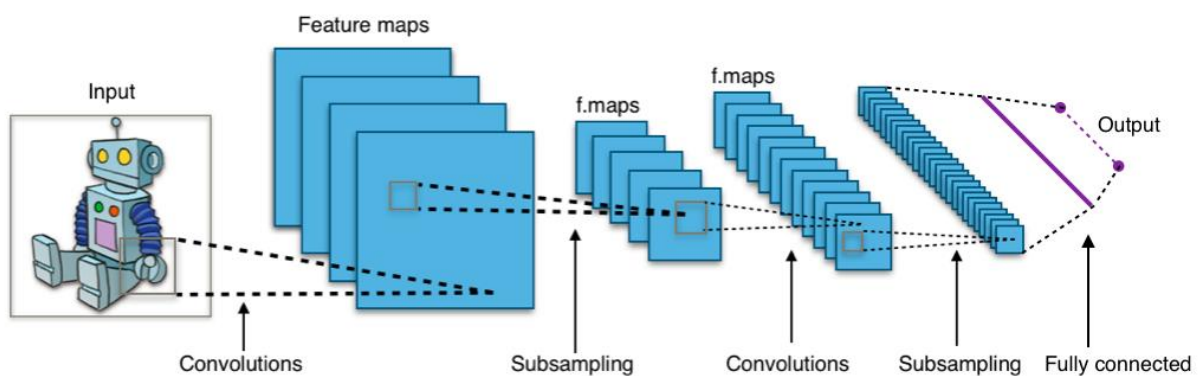


Figure 1 : https://en.wikipedia.org/wiki/Convolutional_neural_network

Mentor et Evalueur : Maïeul Lombard

Table des matières

Introduction.....	1
Présentation de la problématique	1
a) Choix des données utilisées	1
b) Méthode baseline : OneVersusRest et SGDClassifier.....	2
Evolution de la classification de données textuelles.....	2
a) Représentation des données : TF-IDF VS Word Embeddings.....	3
b) Modèle d'apprentissage : SGDClassifier VS Réseau de neurones convolutionnels	5
Implémentation de l'algorithme de la nouvelle méthode	7
a) Transformation des données textuelles.....	7
b) Recherche des hyperparamètres optimaux par validation croisée	8
c) Evaluation du modèle.....	9
Conclusions.....	10

Introduction

L'objectif de cette étude est d'identifier les évolutions de la Data Science pour la résolution d'un problème concret, de tester de nouvelles méthodes et de développer une preuve de concept en réalisant une veille thématique via la recherche de sources pertinentes, fiables et récentes sur les avancées dans le domaine étudié. En effet, les algorithmes et les modèles utilisés en Machine Learning et plus généralement en Data Science sont en constante évolution.

Nous nous plaçons ici dans le cadre d'un problème de classification de données textuelles : contrairement aux autres données, les données textuelles nécessitent des traitements spécifiques afin de transformer le texte en features exploitables et structurées (par exemple sous forme vectorielle) sur lesquelles nous pouvons appliquer des modèles classiques de Machine Learning. Pour résoudre un problème de classification de textes, un premier algorithme a déjà été mis au point sur des données textuelles nettoyées et normalisées. L'objectif est de trouver une nouvelle méthode pour améliorer la méthode initiale et améliorer les performances du modèle.

Présentation de la problématique

Nous nous plaçons dans le cadre d'un problème de classification de données textuelles : Stack Overflow est un célèbre site de questions-réponses liées au développement informatique. Un utilisateur peut poser une question sur ce site et à l'aide de plusieurs tags, faciliter l'accès aux réponses des autres utilisateurs. Cependant, contrairement aux utilisateurs expérimentés, il est difficile pour les utilisateurs débutants de trouver des tags pertinents à associer à une question. C'est pourquoi il nous est demandé de développer un système de suggestion de tags associés à un message pour le site Stack Overflow.

a) Choix des données utilisées

Les données ont été récupérées sur l'outil d'export[1] de données de Stack Overflow grâce à une requête SQL limitant les messages aux plus récents (depuis le 01/01/2017) et aux plus populaires (nombre de vues supérieur à 10) sans que le contenu du message soit vide.

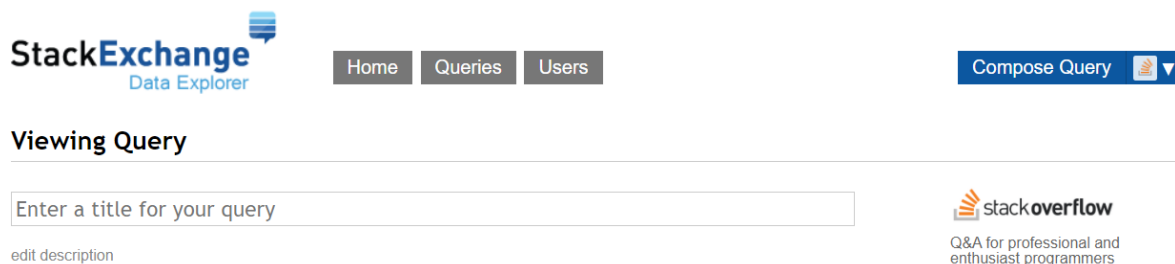


Figure 2 : Site de l'outil d'export de Stack Overflow

Le Dataset final qui sera utilisé pour les analyses est composé de 3 variables et de 48 447 lignes : les variables 'Body' et 'Title' correspondant au corps et au titre des messages donnés par l'utilisateur et la variable 'Tags' correspondant à la liste des Tags associée à chaque message. Les données des messages ont été traitées par différentes techniques spécifiques :

- Suppression des balises HTML
- Suppression de la ponctuation et des majuscules
- Récupération uniquement des noms et des noms propres dans les messages avec le package **Spacy**
- Suppression des **stopwords** correspondant à la liste des 145 mots les plus fréquents contenus dans les messages et des stopwords de la langue anglaise avec la librairie **NLTK**


- Suppression des nombres seuls, des mots à une lettre et des espaces

Ce jeu de données était composé initialement de 12 466 Tags différents dont 5 439 n'apparaissant qu'une seule fois. La liste des Tags associée à chaque message a été réduite aux 100 Tags les plus fréquents sur tout le DataFrame (ce qui représente 48.64 % des Tags utilisés).

b) Méthode baseline : OneVersusRest et SGDClassifier

La méthode baseline traite la prédiction des Tags à partir des messages nettoyés (variable qualitative à transformer) comme un problème de classification multi-labels. Les features utilisables ont été extraites par le module **TfidfVectorizer** appliqué aux messages avec **tf_idf_vec.fit_transform**. Nous obtenons un vecteur des fréquences d'apparition de chaque mot présent dans le corpus de messages, pondérées par un indicateur de similarité (si ce mot est commun ou rare dans tous les messages). Cette transformation **tf-idf** permet de prendre en compte l'importance d'un mot dans chaque message par rapport à l'ensemble du corpus de messages.

Le code baseline applique ici une approche supervisée aux variables '*Body*' et '*Title*' pour prédire la variable '*Tags*' transformée en une matrice par un **dummies** des 100 Tags Stack Overflow les plus fréquents. Le DataFrame a été séparé en jeux de données d'entraînement (70% du jeu de données) et en jeux de données test (30% du jeu de données) avec la fonction **train_test_split** de **scikit-learn** :



Tags	.net-core	algorithm	amazon-web-services
0 python	0	0	0
1 python	0	0	0
2 c++	0	0	0
3 r	0	0	0

4 rows × 99 columns

Figure 3 : Transformation de la variable '*Tags*' pour passer à un problème de classification multi-labels

Le modèle utilisé est l'approche one-versus-rest **OneVsRestClassifier** en utilisant le modèle du classifieur de descente de gradient stochastique **SGDClassifier** (classification multi-labels avec plus de 2 classes Tags différents pour les messages). La recherche des hyperparamètres optimaux du modèle ('*estimator__alpha*' et '*estimator__penalty*') a été effectuée via une validation croisée avec **GridSearchCV** sur le jeu de données d'entraînement et le calcul des performances du modèle sur le jeu de données test. Le modèle optimal présente un **score d'Accuracy** de 0.25 avec un temps de calcul de 0.09 seconde pour la variable '*Body*' et un **score d'Accuracy** de 0.20 avec un temps de calcul de 0.04 seconde pour la variable '*Title*'.

Cependant, cette méthode ne semble pas très performante au vu des résultats obtenus. Les résultats sont peu satisfaisants, il est donc nécessaire de chercher et de mettre en pratique une nouvelle méthode plus récente pour prédire les Tags Stack Overflow à partir des messages en représentant mieux le sens des mots.

Evolution de la classification de données textuelles

Avec l'augmentation continue des données disponibles aujourd'hui en Data Science, les problèmes liés au traitement du langage sont souvent liés à des problèmes de classification de textes qui impliquent souvent la prédiction d'une ou plusieurs étiquettes associées à un document [2]. Les premiers travaux du traitement automatique du langage (**NLP** : Natural Language Processing) datent des années 1950 principalement aux Etats-Unis avec le développement de la traduction automatique, jusqu'à aujourd'hui avec les modèles

d'intelligence artificielle de Microsoft et Alibaba[3]. Le traitement automatique du langage naturel imite la compréhension des mots, avec par exemple le modèle BERT lancé en 2018 par Google qui est un modèle de langage qui peut créer des modèles de pointe pour diverses tâches comme la réponse aux questions[4].

La classification multi-labels est très présente dans de nombreux problèmes modernes comme :

- Comprendre le sentiment du public à partir des médias sociaux
- Détection des e-mails spam et non-spam
- Catégorisation des articles de presse dans des sujets définis

a) Représentation des données : TF-IDF VS Word Embeddings

La classification de texte est un problème d'apprentissage supervisé avec des données étiquetées où chaque document est associé à un ou plusieurs labels. Cependant, les données textuelles de la même manière que les données visuelles nécessitent un pré-traitement. Cette première étape de nettoyage et de normalisation des données ne sera pas traitée ici (stopwords, stemming, lemming, ...). La suivante consiste à extraire à partir des données textuelles des features exploitables par les algorithmes de classification de Machine Learning. Un texte n'est qu'une suite de caractères binaires et n'a pas de sens pour un algorithme. Pour cela, il faut transformer le texte en vecteur pour être utilisable par l'algorithme d'apprentissage. Il existe différentes façons de représenter les documents d'un corpus de texte.

La méthode la plus simple est de créer un **bag of words** qui construit un vecteur binaire et considère uniquement la présence et l'absence du mot (ou un vecteur du nombre d'occurrences). Il est également possible d'affiner cette représentation en tenant compte de l'importance des mots sur l'ensemble des documents : la métrique **tf-idf** (Term-Frequency - Inverse Document Frequency) donne un poids plus important aux termes les moins fréquents qui feront la différence entre les documents avec la formule $idf_i = \log \frac{|D|}{|\{d_j: t_i \in d_j\}|}$. Le poids tf-idf est ainsi calculé comme la fréquence du n-gram x idf(n-gramme).

Cependant, une représentation des textes par cette méthode de comptage aboutit à des matrices de grande dimension mais essentiellement vides. Nous créons des « matrices creuses » : les mots n'étant pas présents dans tous les textes du corpus, cette représentation vectorielle est surtout composée de 0, ce qui représente un gaspillage de ressources et un possible biais dans les résultats (les 0 sont en majorité). Cela peut expliquer les résultats peu satisfaisants obtenus avec la méthode baseline. Les documents ont été traités comme un ensemble de mots indépendants les uns des autres, sans sens entre eux.

Une autre manière de représenter les mots et les documents est le **Word Embeddings** (ou plongement de mots) : la position d'un mot dans l'espace vectoriel est apprise à partir du texte et est basée sur les mots qui l'entourent, ce qui produit une représentation vectorielle dense[2]. Cette représentation distribuée du texte est l'une des principales avancées pour les méthodes d'apprentissage en profondeur pour le traitement du langage.

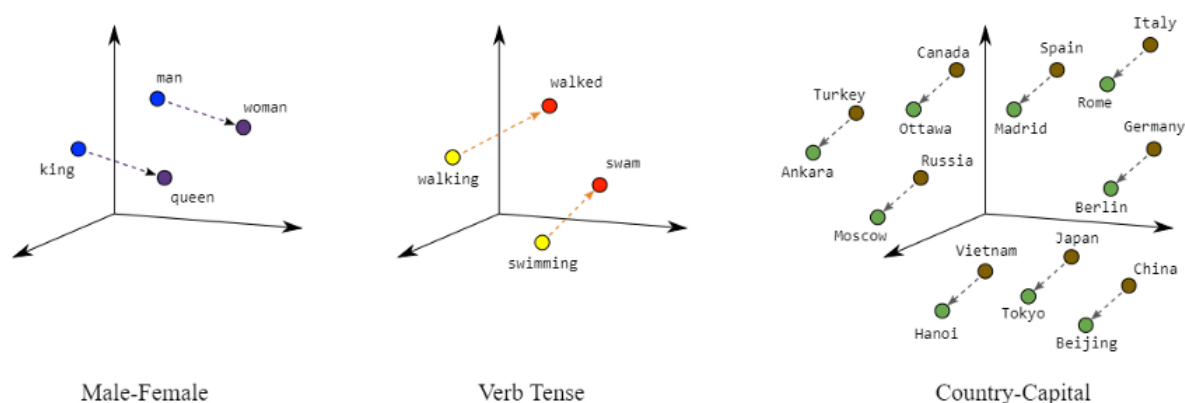


Figure 4 : Analogies entre les mots de sens similaire par le Word Embeddings[5]

L'hypothèse principale sur laquelle s'appuie cette méthode est la *distributional hypothesis* qui permet de prendre en compte l'environnement de mots dans lequel se trouve le mot étudié. En effet, le Word Embeddings représente les mots ayant le même sens avec une représentation similaire. Les mots sont représentés sous forme de vecteurs à valeur réelle dans un espace prédéfini où le sens des mots les rapproche. Chaque mot est mappé sur un vecteur où les valeurs du vecteur sont apprises. La représentation similaire entre des mots utilisés de la même façon permet de capturer leur sens. De plus, cette technique réduit également la dimension du vecteur (puisque la taille du vecteur fixe étant un paramètre à définir sera inférieure à la taille du vocabulaire) en comparaison avec les vecteurs obtenus avec les bag of words qui sont généralement vides et sans relation entre les mots[6].

Pour expliquer le principe du Word Embeddings, prenons un exemple: un problème de classification des aliments (« bon » ou « mauvais »). Chaque critique est composée d'un certain nombre de mots qui forment le vocabulaire de la taille de 5000 mots. L'objectif du Word Embeddings est de projeter l'ensemble des mots d'un vocabulaire de taille V dans un espace vectoriel continu composé des vecteurs des mots de taille N plus petite. Cette méthode s'appuie sur un réseau de neurones à une couche cachée. Dans notre exemple, l'objectif est de calculer pour chaque mot un vecteur de taille fixé, ici la matrice de taille 4 sur 5000 correspondant à la **matrice d'Embeddings W** qui contient les plongements de tous les mots du vocabulaire :

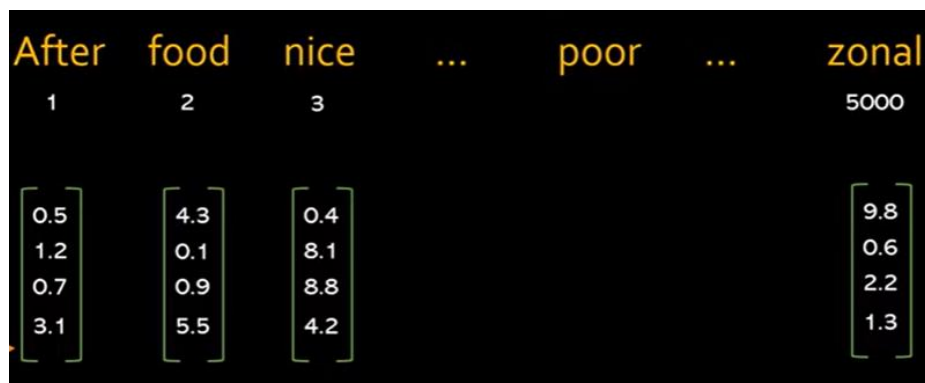
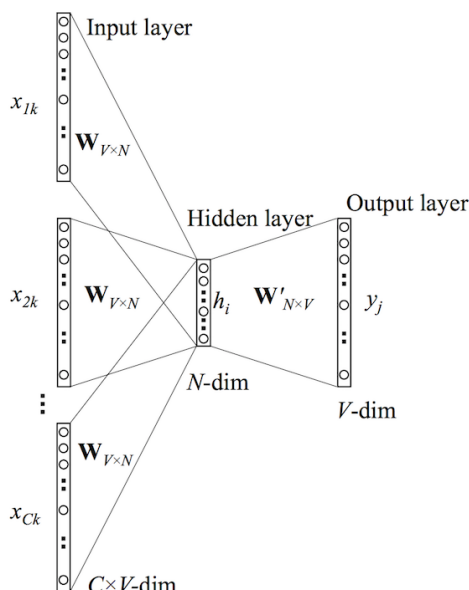


Figure 5 : Exemple de Word Embeddings : Vecteur des valeurs de chaque mot[7]

Chaque vecteur associé à chaque mot sera composé de valeurs proches à un autre vecteur de mot similaire.

La première étape est d'encoder chaque mot de chaque texte du corpus par rapport aux autres mots. Au début de l'entraînement, la matrice d'Embeddings W est initialisée avec des poids aléatoires pour les mots nice » et « food » et est multipliée au vecteur de taille 1×4 provenant d'un encodage one-hot du mot.



Ce produit scalaire permet de récupérer le vecteur à 1 dimension correspondant à la colonne du mot étudié dans la matrice d'Embeddings (couche cachée du réseau de neurones de taille définie N). Cette couche cachée est une application linéaire entre les entrées et la matrice d'Embeddings W qui permet le calcul et la comparaison de la couche de sortie Y grâce à la fonction d'activation. Les valeurs des vecteurs de chaque mot et donc de la matrice d'Embeddings W seront ensuite modifiées par rétropropagation (descente de gradient). Cette approche est semblable à un réseau de neurones à 1 couche et est présentée plus en détails après[8].

Figure 6 : Principe du Word Embeddings[9](cas du Continuous Bag-of-Words (CBOW) qui prend une fenêtre autour du mot et prédit le mot en sortie à l'inverse du modèle skip-gram)

On obtient pour notre exemple la matrice d'Embeddings suivante : les mots « nice » et « good » ont des sens proches et se retrouvent avec des valeurs similaires, de la même manière que « poor » et « weak ».

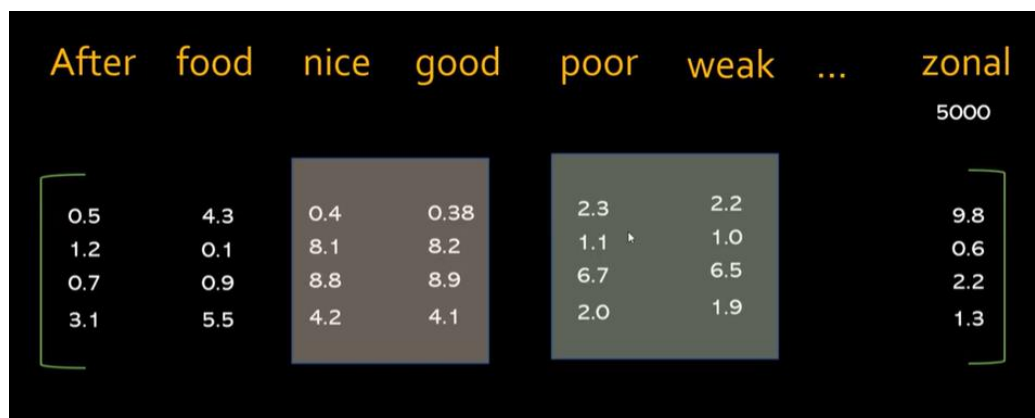


Figure 7 : Exemple de Word Embeddings : Matrice d'Embeddings final[7]

La taille de la couche cachée est fixée, ce qui permet de compresser le corpus vers un dictionnaire de vecteurs denses de dimension inférieure à la taille du vocabulaire initiale. Chaque mot est représenté par un point dans l'espace d'Embeddings et ces points sont appris en fonction des mots qui entourent le mot cible, ce qui permet de tenir compte du sens des mots (regroupement local des mots similaires).

b) Modèle d'apprentissage : SDGClassifier VS Réseau de neurones convolutionnels

Un problème de classification consiste à assigner un label à un document. Pour cela, l'ensemble des documents a été préalablement transformé en jeu de données sous forme vectorielle. Il existe de nombreux modèles d'apprentissage, tels que le SVM, le Bagging, le Boosting, la régression logistique... Le modèle utilisé dans cette étude est un algorithme de classification **SDGClassifier**. Il s'agit d'un classifieur linéaire avec formation **SDG**, c'est-à-dire qu'il implémente des modèles linéaires régularisés avec apprentissage par **descente de gradient stochastique**[10]. Ce classifieur est performant pour des problèmes d'apprentissage à grande dimension avec des données rares et éparées, ce qui est souvent le cas pour la classification de textes[10][11].

La descente de gradient stochastique est une méthode d'optimisation pour les problèmes d'optimisation sans contrainte. Cette méthode se rapproche du vrai gradient $E(w, b)$ en considérant un seul exemple d'entraînement à la fois. Le SDGClassifier implémente une routine d'apprentissage SDG qui prend en charge différentes fonctions de perte et de pénalités pour la classification avec l'algorithme qui itère sur les exemples d'entraînement et adapte les paramètres du modèle selon la formule : $w \leftarrow w - \eta [\alpha \frac{\partial R(w)}{w} + \frac{\partial L(w^T x_i + b, y_i)}{w}]$. Le paramètre b est mis à jour sans régularisation avec un taux d'apprentissage plus petit et avec une décroissance supplémentaire pour les matrices creuses[11]. Cependant, les inconvénients de ce modèle sont que le SDG nécessite des hyperparamètres comme le paramètre de régularisation et le nombre d'itérations et est sensible à la mise à l'échelle des features[12].

Les réseaux de neurones sont présents dans plusieurs domaines impliquant la classification, la régression et même les modèles génératifs. Ils sont très fréquemment utilisés dans la vision par ordinateur, la reconnaissance vocale et le traitement du langage naturel (NLP). En effet, ils traitent des problèmes sous forme de séquences, comparables aux mots qui sont des séquences de lettres. De plus, le Word Embeddings se fait conjointement avec un modèle de réseau de neurones pour la classification de textes. Les réseaux de neurones profonds sont utilisés pour reconnaître des patterns complexes et des relations qui peuvent exister entre les données, ce qui est idéal pour étudier du texte et les relations entre les mots. L'architecture d'un réseau de neurones est codifiée avec une couche d'entrée, des couches cachées et une couche de sortie avec les relations qui peuvent exister entre les couches :

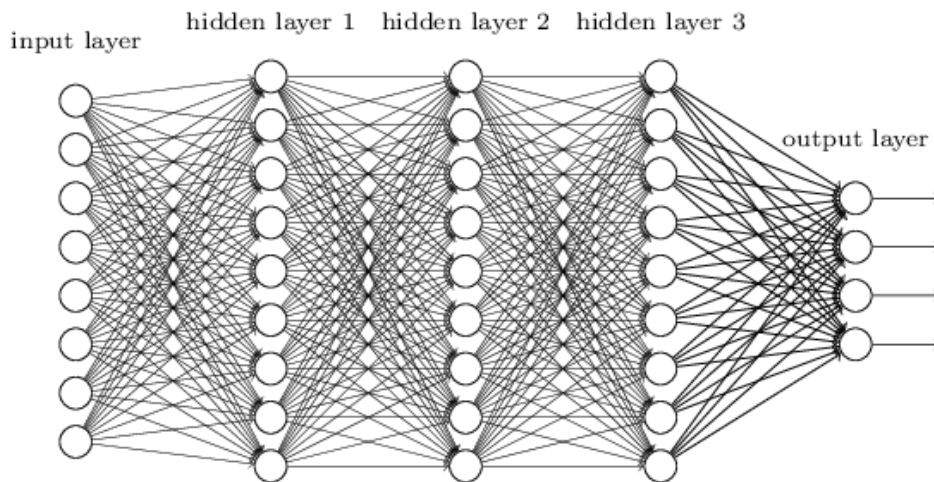


Figure 8 : Réseau de neurones profonds[2]

La formule d'une couche à l'autre est : $o_j = f(\sum_i w_{i,j} a_i + b_j)$, où la couche de nœuds a sert d'entrée à la couche de nœuds o (produit de chaque nœud d'entrée a_i par un poids w en ajoutant un biais b). la fonction f est la fonction d'activation sigmoïde pour des problèmes de classification binaire. Les poids w initialisés aléatoirement sont entraînés par le modèle par rétropropagation avec des méthodes d'optimisation comme la descente de gradient pour minimiser l'erreur en sortie du réseau calculée et la sortie souhaitée déterminée par une fonction de perte (entropie croisée binaire dans notre cas)[8].

Il existe diverses approches pour le traitement du langage. Ici, l'objectif est de trouver d'autres méthodes de classification de textes pour améliorer les performances. Une première étape était la représentation des données en essayant de donner un sens aux mots avec le Word Embeddings. Pour choisir quelle méthode utilisée, je me suis appuyée sur l'article de C. Wang *et al.* (2020)[13] qui compare et recommande d'utiliser les **réseaux de neurones convolutionnels CNN** par rapport aux **modèles LSTM bidirectionnel BiLSTM**.

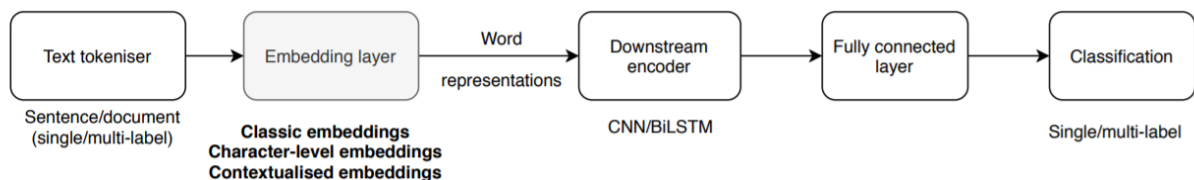


Figure 9 : Architecture de l'exploitation de diverses Word Embeddings pré-entraînées pour la classification de texte dans les modèles de réseaux de neurones[13]

Ce résultat est également démontré dans l'article de M. David et S. Renjith (2021)[14] qui compare les **réseaux de neurones convolutionnels (CNN)** et **récurrents (RNN)**. Les modèles CNN donnent en effet des performances élevées pour la classification de textes en utilisant le Word Embeddings.

Les plongements de mots sont placés en entrée des réseaux de neurones et la couche d'Embeddings est ajustée de manière supervisée avec l'algorithme de rétropropagation. Les textes des documents doivent être préparés en spécifiant la taille de l'espace vectoriel. Chaque mot va être appris comme une entrée dans une séquence, pouvant rendre l'apprentissage du modèle long mais avec un Word Embeddings ciblé.

En 2012, lors de la compétition annuelle ImageNet Challenge, Geoffrey Hinton et son équipe battent les modèles précédents en utilisant un réseau de neurones convolutionnels (**AlexNet**)[15]. Ils sont capables d'extraire des features apprises automatiquement à partir de chacune des entrées et d'entraîner un classifieur dessus (l'erreur de classification est minimisée pour optimiser les paramètres du classifieur et des features). En effet, le travail de convolution de la couche d'entrée est utilisé pour calculer la sortie, avec l'existence de connexions locales entre une région à l'entrée du réseau et un neurone à la sortie. Un CNN est composé de différents types de couches, chacune jouant un rôle dans un ordre établi. Ils reprennent

l'idée des filtres convolutionnels avec leurs coefficients qui sont appris comme paramètres par descente de gradient.

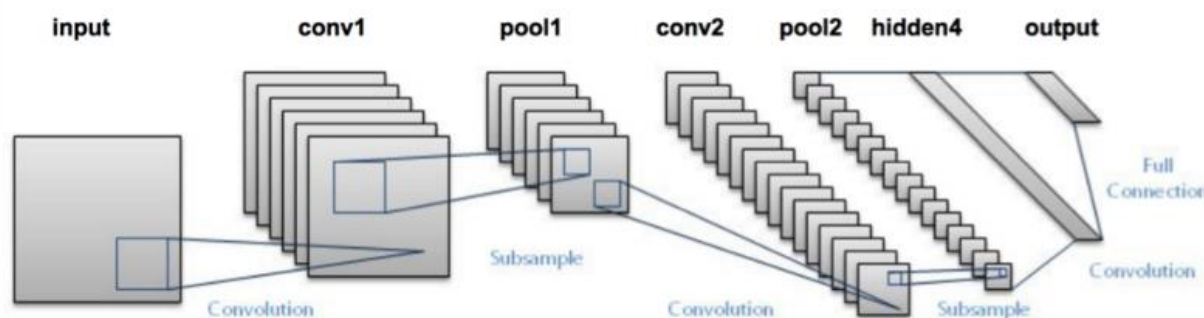


Figure 10 : Réseau de neurones convolutionnels[2]

La couche de **Convolution** détecte des features en utilisant un filtrage par convolution : elle va extraire une features map (garde que certains points) à partir des données en entrée, ce qui réduit le nombre de paramètres et le gradient évanescet. Un nombre X de filtres dans les mêmes couches génèrent X features map en sortie en parallèle. La couche de **Pooling** applique une opération de maximum local aux features map qui deviennent plus petites. Ce rééchantillonnage par le maximum réduit le nombre de paramètres et évite le surapprentissage. La couche **Fully-connected** reçoit un vecteur en entrée et modifie les valeurs pour en produire un nouveau. La dernière couche renvoie le vecteur des probabilités finales associées à chaque classe en utilisant une fonction d'activation. Dans le cas d'une classification multi-labels, il s'agit d'une fonction d'activation sigmoïde pour une classification binaire. La rétropropagation du gradient calcule le gradient de l'erreur de chaque neurone de la dernière vers la première couche. Cela permet de corriger les erreurs selon l'importance de la contribution de chaque élément aux erreurs. Plus il y a de couches, plus le réseau de neurones est profond, plus la couche de Convolution est loin de la sortie du réseau et plus les features apprises sont sophistiquées.

Implémentation de l'algorithme de la nouvelle méthode

L'objectif ici est de prédire la liste de Tags à partir des messages des utilisateurs :

	Body	Title	Tags
0	statement extract sections docx autonumbering ...	docx auto pythondocx	python
1	xarray dataset regression variables dimension ...	xarray operations dataset	python
2	number complexity optimizations check bigo pro...	prime bigo	c++
3	meaning statement section interpretations form...	meaning dot lmy	r
4	recognizer opencvcontrib line modules thing la... attributeerror attribute createlbphfacerecognizer		python, python-3.x
5	trouble ngbootstrap dropdown component depende...	peer dependency popperjs	jquery, npm
6	vuetify vstepper vue router requirements step...	vuetify vstepper vue router	javascript, vue.js

Figure 11 : DataFrame utilisé (7 premières lignes des 3 variables d'intérêt)

L'algorithme de la méthode utilisée est tiré d'un article de Nikolai Janakiev (2021)[8] a été modifié pour s'adapter à notre problème. On retrouve cette méthode dans d'autres articles tels que l'un des articles de Jason Brownlee[16] (2017) disponible sur <https://machinelearningmastery.com/>.

a) Transformation des données textuelles

Les messages et les Tags sont transformés en données exploitables pour passer à un problème de classification multi-label. L'encodage des messages s'est fait via la fonction `process_text`[17].

J'ai utilisé la fonction **Tokenizer** basée sur le nombre de mots à prendre en compte dans les messages (ici, 5000) et appliquée sur la liste des messages contenus dans 'Body' et 'Title' avec **fit_on_texts**. Cette étape va vectoriser le corpus de messages en une liste d'entiers, chaque entier correspondant à une valeur dans un dictionnaire encodant l'ensemble du corpus où les clés du dictionnaire sont les mots eux-mêmes :

```
statement extract sections docx autonumbering pythondocx text docx autonumbering
[191, 2759, 1115, 1, 1, 1, 26, 1, 1]
```

*Figure 12 : Encodage du premier message de 'Body' en utilisant le mappage effectué sur l'ensemble des messages avec **fit_on_texts***

Pour s'assurer que les messages soient de la même longueur, j'ai appliqué la fonction **pad_sequences** sur les vecteurs de messages pour les compresser à une longueur maximale *maxlen* correspondant à la longueur maximale des séquences sur l'ensemble des données :


```
array([[ 191, 2759, 1115, ..., 0, 0, 0],
       [ 1, 368, 1125, ..., 0, 0, 0],
       [ 11, 593, 1324, ..., 0, 0, 0],
       ...,
       [1489, 8, 48, ..., 0, 0, 0],
       [ 646, 562, 1, ..., 0, 0, 0],
       [ 6, 458, 522, ..., 0, 0, 0]], dtype=int32)
```

Figure 13 : Matrice de l'encodage des messages obtenue à partir de la variable 'Body'

J'ai également défini *vocab_size* qui est le nombre total de mots présents, correspondant à la taille du vocabulaire dans le **Tokenizer** utilisé pour encoder les documents.

J'ai également encodé la variable 'Tags' qui est une colonne composée d'un ou plusieurs Tags ou classes. Pour traiter ces données, j'ai transformé les chaînes de caractères de Tags en listes et appliqué la fonction **fit_transform** de **MultiLabelBinarizer**[18] sur la liste des listes des Tags. On passe donc d'une liste de classes à une matrice binaire de 100 colonnes de 0 et de 1 (1 indiquant la présence du Tag). La liste du nom des colonnes correspondant aux noms des Tags a été récupérée avec la fonction **classes_** de **MultiLabelBinarizer** :

Tags	
0	python
1	python
2	c++
3	r



```
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0]])
```

Figure 14 : Transformation de la variable 'Tags' pour passer à un problème de classification multi-labels

Pour la suite de l'analyse, j'ai séparé mon jeu de données en jeux de données d'entraînement et de test (représentant 30% du jeu de données). Les approches traditionnelles à un seul label pour la stratification des données ne parviennent pas à fournir une séparation homogène entre les nouveaux ensembles de données dans le cadre d'une stratification multi-labels[19]. J'ai utilisé la fonction **iterative_train_test_split** de **skmultilearn** selon une stratification homogène de la matrice des Tags présents[20].

b) Recherche des hyperparamètres optimaux par validation croisée

Le modèle utilisé ici est un réseau de neurones convolutionnels (CNN). Avant de l'implémenter, j'ai défini sa structure composée d'une couche de convolution et d'une couche fully-connected. Ce modèle prend en entrée les séquences de longueur *maxlen* et les classe dans les 100 classes de Tags. Il renvoie un vecteur des probabilités d'appartenir à chacune des 100 classes.

Je vais détailler la fonction du modèle CNN. Je crée d'abord un réseau de neurones vides avec **Sequential** et j'ajoute les couches au fur et à mesure. J'ajoute la couche **Embeddings** au réseau définie comme la première couche d'un réseau :

```
model.add(Embedding(vocab_size, embedding_dim, input_length=maxlen))
```

L'**Embeddings** a un vocabulaire de taille *vocab_size*, une taille d'entrée des séquences *maxlen* et un espace d'Embeddings de dimension *embedding_dim* (espace vectoriel dans lequel les mots sont intégrés). Il définit la taille des vecteurs de sortie de cette couche pour chaque mot[21]. Cette couche **Embeddings** nécessite en entrée des entiers où chacun d'eux correspondent à un seul token ayant une représentation spécifique dans l'Embeddings[16], ce qui a été réalisé par la fonction **fit_on_texts** de **Tokenizer**. Ces vecteurs sont aléatoires au début de l'entraînement mais deviennent significatifs au fur et à mesure dans le réseau.

La couche de Convolution utilise *num_filters* filtres de taille *kernel_size*, détecte les features en utilisant un filtrage par convolution et a une fonction d'activation **ReLU**, c'est-à-dire qu'elle est associée à une couche de correction ReLu qui transforme les valeurs négatives reçues en entrées en 0, sans modifier les valeurs positives ($ReLU(x) = \max(0, x)$). J'ai associé la couche de convolution à une couche de Pooling. Une couche **Dropout** permet de sélectionner aléatoirement un taux *rate* de neurones qui seront ignorés pendant l'entraînement du modèle. Le réseau devient ainsi moins sensible aux poids spécifiques des neurones, ce qui réduit le surapprentissage. La couche Fully-connected calcule un vecteur de taille *units* et est associée à une couche de correction ReLu. Cette couche reçoit un vecteur en entrée et modifie les valeurs pour produire un nouveau vecteur. La dernière couche renvoie un vecteur de taille 100 avec la fonction d'activation '*sigmoid*' : $sigmoid(x) = \frac{1}{(1+e^{-x})}$ [22]. Cette fonction renvoie un résultat proche de 0 pour les petites valeurs (<-5) et un résultat proche de 1 pour les grandes valeurs (>5), ce qui est le plus approprié pour un problème de classification multi-labels binaire. Cette couche de sortie va générer des valeurs comprises entre 0 et 1. Toutes ces couches sont ensuite compilées via un **optimizer Adam** avec un taux d'apprentissage *lr* et une fonction de perte d'entropie croisée binaire '**binary_crossentropy**', qui est utilisée pour des problèmes de classification binaire.

Le modèle étant défini, j'ai fait une recherche des hyperparamètres optimaux en utilisant le classifieur **KerasClassifier** et en définissant une grille de paramètres *param_grid* pour réaliser une validation croisée avec **GridSearchCV**[23]. Le processus **GridSearchCV** va construire et évaluer un modèle pour chaque combinaison de paramètres. La validation croisée est utilisée pour évaluer chaque modèle individuellement en optimisant l'**Accuracy**. Ce modèle de réseau de neurones utilisant beaucoup de ressources et mettant beaucoup de temps, j'ai réalisé la validation croisée sur 2 **folds** pour les 8 hyperparamètres (pour un total de 16 fits), j'ai fixé la plupart des hyperparamètres et fait varier la taille de l'espace vectoriel pour le Word Embeddings :

```
dict(num_filters=[128], kernel_size=[5], units=[32, 64], vocab_size=[vocab_size], embedding_dim=[50, 100], maxlen=[maxlen], epochs = [30], batch_size = [20], lr = [0.001], rate = [0.0, 0.5])
```

c) Evaluation du modèle

Le modèle a été entraîné sur le jeu de données *X_train* lors de la validation croisée. Pour comparer cette méthode avec la méthode baseline, j'ai calculé les performances du modèle sur de nouvelles données avec le jeu de données test *X_test*. Il existe de nombreuses metrics pour évaluer la qualité d'un modèle[24], telles que le Jaccard score (coefficient de similarité), le rappel (le ratio $\frac{tp}{(tp+fn)}$ où *tp* : nombre de vrais positifs et *fn* : nombre de faux négatifs), la précision (le ratio $\frac{tp}{(tp+fp)}$ où *tp* : nombre de vrais positifs et *fn* : nombre de faux positifs), le support (nombre d'occurrences dans chaque classe **y_true**), le F-beta score (moyenne harmonique pondérée de la précision et du rappel), ...[25]

Pour ce problème de classification multi-labels, la metrics la plus intéressante à étudier est le **score Accuracy** qui calcule la précision de la prédiction, c'est-à-dire l'ensemble des labels prédits doivent correspondre aux vrais labels. L'objectif ici est de suggérer des Tags à l'utilisateur et les faux négatifs et les faux positifs ne sont pas importants pour ce type de problème. En effet, ils n'auront pas une grande incidence sur l'utilisateur (contrairement à des problèmes liés à la santé ou à la sécurité) qui pourra, une fois les Tags proposés par le modèle en fonction de son message, faire lui-même un choix.

J'ai calculé l'**Accuracy** sur la prédiction faite sur le jeu de données test **X_test** et le temps de calcul. L'idée d'évaluer le modèle sur un jeu de données de test est de pouvoir généraliser le modèle et mesurer les performances du modèle dans des situations inconnues à l'entraînement.

J'ai récupéré les hyperparamètres optimaux obtenus par la validation croisée : 'embedding_dim': 100, 'rate': 0.5 pour une **Accuracy** de 0.423465 pour la variable 'Body' et 'embedding_dim': 50, 'rate': 0.5 pour une **Accuracy** de 0.334795 pour la variable 'Title' :

Layer (type)	Output Shape	Param #
embedding_17 (Embedding)	(None, 579, 100)	6973600
conv1d_17 (Conv1D)	(None, 575, 128)	64128
global_max_pooling1d_17 (Glo	(None, 128)	0
dropout_17 (Dropout)	(None, 128)	0
flatten_17 (Flatten)	(None, 128)	0
dense_34 (Dense)	(None, 32)	4128
dense_35 (Dense)	(None, 100)	3300
Total params: 7,045,156		
Trainable params: 7,045,156		
Non-trainable params: 0		

Figure 15 : Résumé des paramètres et de la taille des vecteurs pour chaque couche du CNN pour 'Body'

Conclusion

Le Word Embeddings inclus dans un réseau de neurones convolutionnels améliore les performances du modèle, ce qui est une meilleure approche pour la suggestion de Tags. Contrairement aux méthodes simples de tf-idf, la technique du Word Embeddings permet de créer des caractéristiques de textes dans un espace multidimensionnel, plus représentatif de la sémantique des mots, ainsi qu'une réduction de la taille de la représentation vectorielle.

Cette approche est une bonne piste pour ce type de problème de classification de textes. Le modèle utilisé ici a une architecture simple. Il est possible d'apporter des améliorations en terme d'optimisation des hyperparamètres ou structure des modèles (RNN, LSTM, RCNN). Une autre approche serait d'utiliser des modèles de Word Embeddings pré-entraînés tels que **glove**[21], **word2vec**[16], **BERT**[13], ... Ces modèles ont été entraînés sur des jeux de données très volumineux et permettent d'utiliser des espaces d'Embeddings pré-calculés[8], ce qui peut potentiellement encore améliorer les performances du modèle par rapport à la méthode baseline. Cependant, bien que ces techniques soient efficaces et performantes, elles prennent plus de temps pendant l'entraînement et représentent un outil de modélisation assez difficile à appréhender et donc complexe.

Bibliographie

- [1] Stack Overflow, "StackExchange." [Online]. Available: <https://data.stackexchange.com/stackoverflow/query/new>.
- [2] S. Bansal, "A Comprehensive Guide to Understand and Implement Text Classification in Python." <https://www.analyticsvidhya.com/blog/2018/04/a-comprehensive-guide-to-understand-and-implement-text-classification-in-python/>.
- [3] "Traitement automatique des langues." https://fr.wikipedia.org/wiki/Traitement_automatique_des_langues.
- [4] K. T. Jacob Devlin, Ming-Wei Chang, Kenton Lee, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," 2019. <https://arxiv.org/abs/1810.04805>.
- [5] "Embeddings: Translating to a Lower-Dimensional Space." <https://developers.google.com/machine-learning/crash-course/embeddings/translating-to-a-lower-dimensional-space>.
- [6] J. Brownlee, "What Are Word Embeddings for Text?," 2017. <https://machinelearningmastery.com/what-are-word-embeddings/>.
- [7] *Word embedding using keras embedding layer | Deep Learning Tutorial 40 (Tensorflow, Keras & Python)*. .
- [8] "Practical Text Classification With Python and Keras." <https://realpython.com/python-keras-text-classification/#keras-embedding-layer>.
- [9] "Word embedding." https://fr.wikipedia.org/wiki/Word_embedding#.
- [10] "SGDClassifier." https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.SGDClassifier.html.
- [11] "Descente de gradient stochastique." <https://scikit-learn.org/stable/modules/sgd.html#implementation-details>.
- [12] "Stochastic Gradient Descent." <https://scikit-learn.org/stable/modules/sgd.html>.
- [13] C. Wang, P. Nulty, and D. Lillis, "A Comparative Study on Word Embeddings in Deep Learning for Text Classification," *ACM Int. Conf. Proceeding Ser.*, no. December, pp. 37–46, 2020, doi: 10.1145/3443279.3443304.
- [14] M. S. David and S. Renjith, "Comparison of word embeddings in text classification based on RNN and CNN," *IOP Conf. Ser. Mater. Sci. Eng.*, vol. 1187, no. 1, p. 012029, 2021, doi: 10.1088/1757-899x/1187/1/012029.
- [15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks Alex," *Adv. Neural Inf. Process. Syst.*, pp. 25, 1097–1105, 2012.
- [16] J. Brownlee, "Deep Convolutional Neural Network for Sentiment Analysis (Text Classification)." <https://machinelearningmastery.com/develop-word-embedding-model-predicting-movie-review-sentiment/>.
- [17] "Tokenization and Text Data Preparation with TensorFlow & Keras." <https://www.kdnuggets.com/2020/03/tensorflow-keras-tokenization-text-data-prep.html>.
- [18] "MultiLabelBinarizer." <https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MultiLabelBinarizer.html>.
- [19] "Stratification." <http://scikit.ml/stratification.html>.
- [20] "iterative_stratification." http://scikit.ml/api/skmultilearn.model_selection.iterative_stratification.html#module-

skmultilearn.model_selection.iterative_stratification.

- [21] J. Brownlee, "How to Use Word Embedding Layers for Deep Learning with Keras." <https://machinelearningmastery.com/use-word-embedding-layers-deep-learning-keras/>.
- [22] "Activation functions." <https://keras.io/api/layers/activations/>.
- [23] J. Brownlee, "How to Grid Search Hyperparameters for Deep Learning Models in Python With Keras." <https://machinelearningmastery.com/grid-search-hyperparameters-deep-learning-models-python-keras/>.
- [24] "Precision, recall, F_score, support." https://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html#sklearn.metrics.precision_recall_fscore_support.
- [25] "Deep dive into multi-label classification...! (With detailed Case Study)." <https://towardsdatascience.com/journey-to-the-center-of-multi-label-classification-384c40229bff>.