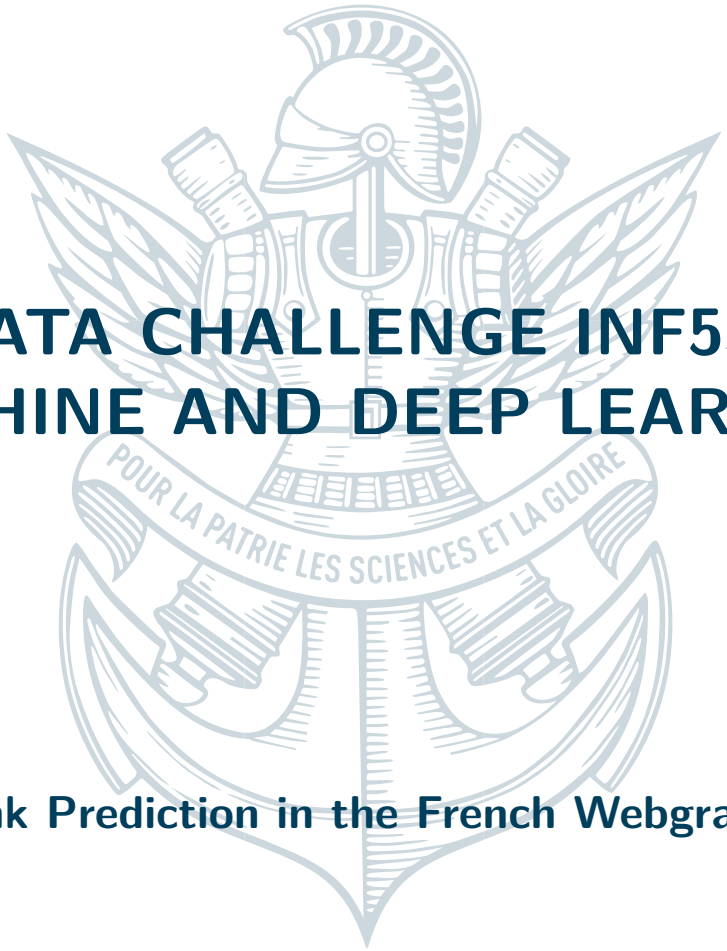

DATA CHALLENGE INF554
MACHINE AND DEEP LEARNING

Link Prediction in the French Webgraph

6 janvier 2020

Gabriel FAIVRE - Skandère SAHLI - Alban ZAMMIT
Team name : **AGS**

TABLE DES MATIÈRES

1
FEATURE ENGINEERING

1.1 Features sur la structure de graphe

On peut à l'aide du TrainSet créer un graphe partiel du web, à partir duquel on peut extraire des features interessants.

1.1.1 • A l'aide de la bibliothèque NetworkX

La bibliothèque NetworkX permet de représenter efficacement des graphes en Python et de leur appliquer de nombreux algorithmes classiques. Cela nous a permis de créer rapidement des features interessants pour le graphe :

- Pour le noeud source et le noeud target dont on cherche à prédire le lien, on peut essayer d'évaluer à quel point l'un ou l'autre est central dans le graphe. La notion de **k-core**, introduite par SEIDMAN dans [1] permet de bien capturer la position d'un noeud dans un graphe. Un k-core d'un graphe (non-orienté) G est un sous-graphe maximal de G où tous les noeuds ont sont de degré au moins k. Le **core number** d'un noeud V correpond alors au plus grand k des k-cores auxquels V appartient. Cette notion nous donne les features *SOURCE NODE CORE* et *TARGET NODE CORE*.
- Un autre feature intérressant est la popularité du noeud source et la popularité du noeud target. On peut évaluer cela avec le **PageRank** du noeud source et du noeud target, qui n'est autre que la valeur de la mesure invariante du graphe web considéré comme une chaine de Markov au noeud étudié [2]. Cette notion nous donne les features *SOURCE PAGERANK* et *TARGET PAGERANK*.
- On peut aussi, pour le noeud source, comme pour le noeud target, penser que le degré entrant ou sortant (et d'autres variantes) peut avoir une influence sur l'existence d'un lien. Cela donne les features *SOURCE NODE IN DEGREE*, *TARGET NODE IN DEGREE*, *SOURCE NODE OUT DEGREE*, *TARGET NODE OUT DEGREE*, *SOURCE AVERAGE NEIGHBORS DEGREE* et *TARGET AVERAGE NEIGHBORS DEGREE*
- On peut aussi regarder la longeur du plus court chemin dans le graphe partiel donné par le TrainSet, entre le noeud source et le noeud target à l'étude. Cela donne le feature *SHORTEST PATH LENGTH*.
- Enfin, on peut utiliser des index qui donnent une idée de la probabilité d'existence d'un lien entre le noeud source et le noeud target à partir du graphe partiel donné par le TrainSet. Ainsi, nous pouvons utiliser **l'index d'allocation ressource**, **le coefficient de Jaccard**, **l'index Adamic-Adar** et **l'attachement préférentiel** Ces index font l'objet de toute un champ d'étude de la théorie des graphes. Etant donné la concision de ce rapport, nous ne rentrons pas ici dans les détails. On pourra se référer aux articles [3] et [4]. Nous avons ainsi créé les features *RESSOURCE ALLOCATION INDEX*, *JACCARD COEFFICIENT*, *ADAMIC ADAR INDEX* et *PREFERENTIAL ATTACHMENT*.

1.1.2 • Avec un auto-encoder

Afin de capturer de manière plus générale la structure de graphe, nous souhaitions utiliser sa matrice d'adjacence. Etant donné le très grand nombre de noeuds dans la graphe (plus de 30000), nous devions réduire la dimension. Nous avons décidé d'utiliser un auto-encoder, en reprenant une idée utilisée dans l'article [5]. Pour chaque noeud, on obtient alors un vecteur de petite dimension qui correspond à la réduction de sa ligne dans la matrice d'adjacence (cf. figure 1)

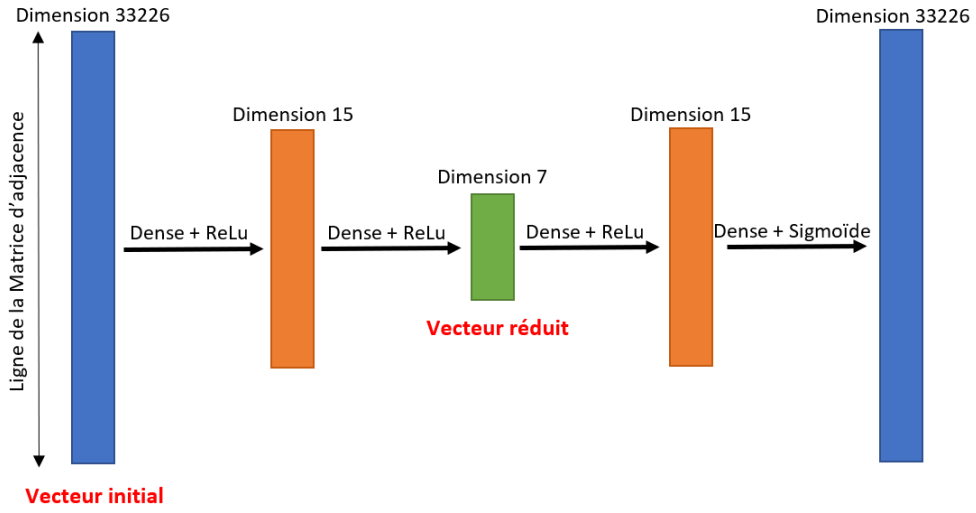

Figure 1 – Autoencodage de la Matrice d'Adjacence

1.1.3 • Node2Vec

Enfin, nous souhaitions obtenir un dernier encodage du graphe, cette fois à l'aide d'une bibliothèque prête à l'emploi : **Node2Vec**, décrite dans [**?**], qui considère le graphe Web comme une chaine de Markov, génére des chemins aléatoires qui se comportent comme des mots et sont ensuite vectorisés avec la bibliothèque *Word2Vec* (cf. partie suivante pour plus d'information).

Cependant, en raison de la grande taille du graphe, et de la capacité limitée de nos ordinateurs portables en ces périodes des fêtes, loin des serveurs de l'école, nous n'avons pas réussi à faire tourner l'algorithme (Au bout de 17h, seulement 17% de l'encodage était effectué...). Nous nous sommes donc passés de **Node2Vec**

1.2 NLP-based features

Notre deuxième idée pour créer des vecteurs est d'utiliser les textes associés à chaque page Web. Notre intuition est de dire que si deux pages Web parlent de la même chose, il est probable qu'il existe un lien entre les deux sites Web. Sur la base de cette intuition, nous avons utilisé trois techniques différentes pour construire des vecteurs basés sur les textes fournis.

1.2.1 • Nettoyage des textes

Pour pouvoir appliquer nos techniques de vectorisation nous avons du en premier lieu nettoyer les textes. D'une part nous nous sommes débarassés des symboles inutiles (tels que *,-,/,@...) et de la ponctuation. Nous avons aussi supprimé les chiffres et nombres qui n'apportent pas vraiment d'information sur le contenu de la page web. Une fois cela fait, nous avons supprimé tous les accents (é–>e) et avons mis toutes les lettres en minuscule. Enfin, nous avons supprimé les mots les plus fréquents de la langue française ainsi que tous les mots de moins de 3 lettres (mots de liaison n'apportant pas beaucoup d'information). Une fois ce traitement effectué, nous avons sauvegardé pour chaque noeud un nouveau fichier texte contenant le texte nettoyé (cela nous permet de gagner en efficacité temporelle et ne pas refaire le nettoyage à chaque fois). C'est à partir de ces nouveaux fichiers que nous avons utilisé nos méthodes de vectorisation.

1.2.2 • Bag of Words

Notre première idée a été d'utiliser la technique classique "bag of words". Cette méthode permet de créer un vecteur pour chaque texte appartenant à un corpus de texte. Ici, notre corpus de texte est l'ensemble des textes nettoyés. A partir de ce corpus, nous créons un vocabulaire (une liste de tous les mots présents dans le corpus), et ce vocabulaire sera notre base pour la vectorisation. Pour chaque texte associé à un noeud, le vecteur associé est créé en mettant dans la colonne associée à un mot du vocabulaire, le nombre d'occurence de ce mot dans le texte. On voit ici que nous obtiendrons des vecteurs très "sparse" (contenant beaucoup de 0). Le principal problème lié à cette méthode est la dimension aberrante du vecteur obtenu (>1.000.000 pour notre nettoyage). Pour remédier à ce problème, nous avons restreint notre vocabulaire aux mots de au plus 10 caractères (en effet, nous avons observé la présence de longue chaine de caractère ne correspondant pas à des mots). Nous avons aussi supprimé tous les mots apparaissant moins de 100 fois dans tout le corpus (leur effet est négligeable car ils apparaitront dans un texte en moyenne moins de 0.01 fois). Nous sommes ainsi passés d'une dimension supérieure à 1.000.000 à une dimension de 50.000. Pour réduire encore la dimension, nous avons utilisé une PCA et avons conservé les 15 premiers vecteurs pour conserver environ 85% de la variance

1.2.3 • Term Frequency - Inverse Document Frequency

Cette méthode est une variante de bag of words, mais qui prend en compte une combinaison de deux facteurs : la fréquence d'apparition du mot dans le texte considéré, mais aussi le nombre de texte du corpus dans lequel le mot apparaît. L'idée sous-jacente est que si un mot est présent dans un grand nombre de texte, alors il ne sera pas discriminant pour catégoriser un texte (exemple classique : un mot de liaison). Il faut aussi bien évidemment tenir compte de la fréquence d'apparition du mot dans le texte considéré. La vectorisation finale conserve la même base que Bag of Words, mais modifie la valeur du coefficient en le fixant à $tf_i * idf_i$ avec tf_i la fréquence d'apparition du mot i dans le texte considéré, et $idf_i = log(\frac{N}{n_i})$ avec N le nombre de texte dans le corpus et n_i le nombre de texte dans lequel le mot i apparait (voir [6]). Nous avons traité le problème de la dimension exactement de la même manière que pour Bag of Words.

1.2.4 • Doc2Vec

La dernière méthode utilisée est radicalement différente des deux autres. Nous quittons ici l'approche de représentation à partir de tout le vocabulaire du corpus, et utilisons une technique bien

plus évoluée qui généralise la méthode word2vec aux paragraphes. En résumé, cet algorithme utilise des fenêtres autour des mots permettant de tenir compte du contexte lors de la vectorisation, ce qui n'était pas du tout capturé par les deux vectorisations précédentes. Par soucis de concision, nous ne détaillerons pas le fonctionnement de cette méthode, et référons le lecteur à cet article [7]. En théorie, cette vectorisation correspond le mieux avec notre intuition initiale d'associer à des sites parlant d'un même sujet des vecteurs similaires. On peut donc espérer qu'elle nous apporte les meilleurs résultats.

2
SELECTION ET ENTRAINEMENT D'UN MODÈLE

2.1 Selection de la famille de modèle

Maintenant que nous avons un DataSet oprationnel, ils nous reste 2 lourdes taches : choisir les features les plus appropriés parmi ceux que nous avons, et choisir un modèle adapté.

Nous avons commencé par explorer tous les modèles possibles avec tous les features existants : SVM, k-NN Classifier, Random Forest, Naive Bayes, Multi-Layer Perceptron...

Ces expériences de débroussaillage nous ont montré que deux types de modèles étaient les plus à même de résoudre notre problème le plus efficacement, que ce soit en temps d'entrainement, ou en précison des prédictions : **Random Forest** et **Multi-Layer Perceptron**.

2.2 Une stratégie de vote pour la classification binaire

Au début, nous souhaitions choisir un unique modèle par Cross-Validation. Cependant, l'espace de recherche des hyper-paramètres est très grand, et nous n'aurions pas aboutis dans les temps. Nous nous sommes alors rappelé d'une remarque de M. SALHA en Petite Classe : on peut grandement améliorer la précision d'un classifieur en utilisant une **stratégie de vote** sur de nombreux classifieurs pas forcément optimaux, ce que semble d'ailleurs confirmer le papier [8].

Nous avons donc entrainé **6 modèles différents**, 3 *Random Forests* et 3 *Multi-Layer Perceptrons*, grâce à la bibliothèque *sk-learn* de Python. Leurs caractéristiques sont données en figure 2. On notera d'ailleurs qu'on évite l'overfitting pour les Random Forests en faisant des arbres peu profonds, et pour les réseaux de neuronnes en utilisant un validation set avec le paramètre *early_stopping* activé.

Ensuite, pour chacun de ces 6 modèles, afin de choisir les features les plus intéressants, nous avons créé 7 jeux de features différents :

- Tous les features
- Les features données par *NetworkX* et ceux par *Bag of Words*
- Les features données par *NetworkX* et ceux par *Term Frequency - Inverse Document Frequency*
- Les features données par *NetworkX* et ceux par *Doc2Vec*
- Les features données par *l'auto-encodeur* et ceux par *Bag of Words*
- Les features données par *l'auto-encodeur* et ceux par *Term Frequency - Inverse Document Frequency*
- Les features données par *l'auto-encodeur* et ceux par *Doc2Vec*

```
classifiers = [
    ("Random Forest 1",RandomForestClassifier(max_depth=5, n_estimators=100, n_jobs = 3,verbose = 1)),
    ("Random Forest 2",RandomForestClassifier(max_depth=10, n_estimators=100, n_jobs = 3, verbose = 1)),
    ("Random Forest 3",RandomForestClassifier(max_depth=15, n_estimators=100, n_jobs = 3, verbose = 1)),
    ("Neural Network 1",MLPClassifier(hidden_layer_sizes=(200,300,150), activation = 'relu',
        learning_rate='adaptive',learning_rate_init = 0.01, verbose=True, max_iter=20,
        early_stopping = True, validation_fraction = 0.05,tol = 1e-3,n_iter_no_change = 3)),
    ("Neural Network 2",MLPClassifier(hidden_layer_sizes=(150,200,300,150), activation = 'tanh',
        learning_rate='adaptive',learning_rate_init = 0.01, verbose=True, max_iter=20,
        early_stopping = True, validation_fraction = 0.05,tol = 1e-3,n_iter_no_change = 3)),
    ("Neural Network 3",MLPClassifier(hidden_layer_sizes=(150,200,300,400,150), activation = 'relu',
        learning_rate='adaptive',learning_rate_init = 0.01, verbose=True, max_iter=20,
        early_stopping = True, validation_fraction = 0.05,tol = 1e-3,n_iter_no_change = 3))]
```

Figure 2 – Caractéristiques des modèles entrainés pour le vote

Ainsi, nous avons entrainé et généré les prédictions de **42 modèles**. Nous avons alors organisé un vote entre ces 42 modèles, c'est-à-dire que pour chaque prédiction, on prend la prédiction majoritaire parmi les 42 modèles. L'idée sous-jacente est que si un modèle est robuste dans 90% des prédictions, mais peu fiable dans 10% des autres, ses erreurs seront compensées par le vote des autres prédicteurs, potentiellement fiable là où il ne l'est pas (il faut pour cela l'hypothèse de décorrelation entre les prédicteurs, ce qui est souligné dans [8], et que nous ne pouvons pas vérifier ici).

2.3 Résultats Numériques

Les F1-score de tous les modèles sur le TestSet, ainsi que leur vote sur différents sous-ensembles sont présentés figure 3

	Tous les features	NetworkX + BOW	NetworkX + TFIDF	NetworkX + Doc2Vec	AutoEncoder + BOW	AutoEncoder + TFIDF	AutoEncoder + Doc2Vec	Vote de tous les jeux de features par modèle
Random Forest 1	0,90665	0,907	0,90646	0,90642	0,88254			**0,90775**
Random Forest 2	0,91296	0,9129	0,91347	0,91165	0,88857			**0,91432**
Random Forest 3	0,91674	0,91476	0,91609	0,91327	0,88698			**0,91764**
Neural Network 1	0,91132	0,90316	0,92166	0,91684	0,87195			**0,91645**
Neural Network 2	0,90286	0,89862	0,91344	0,90926	0,87678			**0,91515**
Neural Network 3	0,89904	0,88706	0,92197	0,91633	0,86518			**0,92281**
Vote de tous les modèles par jeu de feature	**0,92052**	**0,91539**	**0,92219**	**0,91709**	**0,88901**	**0,901741**	**0,89036**	**0,92439**

Figure 3 – Résultat du vote sur les 42 modèles. On Affiche le F1-Score à chaque fois. En bas à droite, il s'agit du vote entre les 42 modèles. Comme attendu, le vote fait croitre la précision du modèle dans tous les cas. Les cases vides n'ont pas plus être remplies à cause de la limite de soumission sur Kaggle

Cette procédure de vote nous permet enfin de conclure : les features les plus interessants pour capturer la structure de graphe sont ceux de **NetworkX** et pour capturer le texte des pages ceux de **TFIDF**. Enfin, on voit que le modèle le plus efficace est **le Multi-Layer Perceptron le plus profond** (Neural Network 3). Cela nous confirme l'intuition que l'on a intérêt à ajouter des couches cachées si l'on veut accroître la précision des prédictions, ce que nous n'avons pas pu faire en pratique ici vu la limite de nos ordinateurs.

RÉFÉRENCES

[1] Stephen B. SEIDMAN. Network structure and minimum degree. *Social networks*, 5(3) :269–287, 1983.

[2] Bachir BEKKA. Le théorème de Perron-Frobenius, les chaines de Markov et un célèbre moteur de recherche. 2007.

[3] Tao ZHOU ; Linyuan LU ; Yi-Cheng ZHANG. Predicting missing links via local information. *The European Physical Journal B*, 71(4) :623–630, 2009.

[4] David LIBEN-NOWELL ; Jon KLEINBERG. The link-prediction problem for social networks. *Journal of the American society for information science and technology*, 58(7) :1019–1031, 2007.

[5] Hang LI ; Haozheng WANG ; Zhenglu YANG. Variation autoencoder based network representation learning for classification. *Student Research Workshop*, pages 56–61, 2017.

[6] Juan Ramos. Using TF-IDF to Determine Word Relevance in Document Queries. pages 1–2, 2003.

[7] Quoc LA ; Tomas MIKOLOV. Distributed representations of sentences and documents. *International conference on machine learning*, pages 1188–1196, 2014.

[8] Roberto BATTITI ; Anna Maria COLLA. Democracy in neural nets : Voting schemes for classification. *Neural Networks*, pages 691–707, 1994.