MICHELUCCI Romain
TRACOL Théo
SINGLAN Nina

Traitement Automatique du Texte en IA

# Rapport projet :

## Présentation du projet :

Afin de mettre en œuvre ce que nous avons appris dans ce cours, nous avons décidé de travailler sur la détection de *Cyber Troll*.

« *Le terme troll désigne, dans le jargon de l'internet, un personnage malfaisant dont le but premier est de perturber le fonctionnement des forums de discussion* », ainsi, il existe différents types de troll :

- Troll débutant : Le troll débutant (appelé aussi « troll qui s'ignore ») est une manière de troller par ignorance de la *nétiquette* (i.e. les règles de conduite et de politesse) et du fonctionnement technique, sans véritable intention de nuire.
- Troll bête : Persuadé d'avoir une opinion valable sur tout, d'être de bonne foi, le troll bête prend l'apparence d'un message véritable.
- Troll méchant : Son but est, consciemment, de tuer les forums. Il cumule tous les types détaillés plus haut.

Ainsi, la détection de troll est un problème récurrent dans la modération des forums.

Afin de réussir à détecter les trolls, nous avons décider d'utiliser un Neural Network que nous avons entrainé sur un ensemble de données annotées.

## Description des tâches :

Dans un premier temps, nous avons téléchargé et prétraité l'ensemble des données sur lesquels nous avons choisis de travailler.

Ensuite, nous avons essayé de mettre en place différents « *Classifier* » afin de voir lequel était le plus compétent pour répondre au problème posé.

Enfin, nous pourrions résumer notre travail en 4 parties :

1. Prétraitement
2. Conversion des données en nombres
3. Entrainement du classifier et prédiction
4. Evaluation des modèles

Bien évidemment, nous mettrons aussi en place différents graphiques afin de visualiser les données et les résultats obtenus par nos classifiers.

MICHELUCCI Romain
TRACOL Théo
SINGLAN Nina

Traitement Automatique du Texte en IA

**Conception et implémentation :**

Dans cette partie nous détaillerons pas à pas les différentes étapes que nous avons suivies lors de notre expérimentation.

Pour commencer, nous avons importé le jeu de données. Et afin de pouvoir le manipuler, nous avons transformé le fichier csv en « *DataFrame* ». Ce qui nous a permis par la suite d'utiliser les méthodes de ***pandas*** afin d'observer et de comprendre les différentes *features* ainsi que d'exploiter le matériel sur lequel nous travaillerons par la suite.

La première partie de notre travail a consisté à supprimer les colonnes/*features* inutiles :

- Nous avons donc commencé par chercher s'il existait des colonnes dont le contenu était nul. Nous en avons trouvé deux : annotation/notes et extras.
- Ensuite nous avons cherché à supprimer les colonnes composées d'une unique valeur. Quatre d'entre elles se sont révélées correspondre à cette description : metadata/sec taken, metadata/last updated by, metadata/status, metadata/evaluation. Elles étaient donc inutiles dans l'algorithme de classification.
- Pour finir nous nous sommes rendu compte que deux autres colonnes n'étaient pas pertinentes : metadata/lastupdates at et metadata/first done at car elles étaient composées uniquement de timestamps.

Lorsque ce tri fut fait, il ne nous restait que 2 colonnes pertinentes pour la classification : content (composée du texte) et annotation/label (composée de 0 [Not Cyber-Troll] ou 1 [Cyber-Troll]).

L'étape suivante de notre projet a été le prétraitement du texte.

Le texte peut contenir des nombres, des caractères spéciaux ou encore des espaces indésirables. C'est pourquoi nous les avons retirés du texte à l'aide de *WordNetLemmatizer*.

Après avoir appliqué notre méthode au contenu, la visualisation nous a permis de confirmer que nous pouvions passer à l'étape suivante.

L'analyse de données nous a permis de convertir le texte afin qu'il soit compréhensible par l'algorithme de classification.

Différentes approches existent. Nous avons choisi d'utiliser le « *Bag of Words Model* » (Sac de mots en français) qui consiste à transformer le texte en représentation numérique.

MICHELUCCI Romain
TRACOL Théo
SINGLAN Nina

Traitement Automatique du Texte en IA
Pour être plus précis et mieux comprendre son utilisation voici un exemple tiré de notre projet : (en théorie)

a) Phrase extraite du jeu de donnée :

« *i just deleted all the people that i have on my facebook off myspace Damn if everyone switched i could just delete the account* »

b) Liste construite à partir des mots de la phrase :

« *i* », « *just* », « *deleted* », « *all* », « *the* », « *people* », « *that* », « *i* », « *have* », « *on* », « *my* », « *facebook* », « *off* », « *myspace* », « *Damn* », « *if* », « *everyone* », « *switched* », « *i* », « *could* », « *just* », « *delete* », « *the* », « *account* »

c) La liste est représentée en « sacs de mots » : chaque clef est un mot distinct tandis que sa valeur est le nombre d'occurrences du mot dans la phrase (ou groupe de phrases)

{« i » : 3, « just » : 2, « deleted » : 1, « all » : 1, « the » : 2, « people » :1, « that » :1, « have » :1, « on » : 1, « my » :1, « facebook » : 1, « off » :1, « myspace » :1, « Damn » : 1, « if » :1, « everyone » :1, « switched » :1, « could » : 1, « delete » :1, « account » :1}

Dans notre cas, le modèle va tout d'abord définir le vocabulaire (c'est l'ensemble des mots distincts provenant de la phrase extraite et arrangé comme un vecteur).

Ensuite le vecteur de vocabulaire est utilisé pour convertir la phrase en un vecteur de fréquence. (Comme expliqué ci-dessus)

Output: array([3, 2, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])

Cependant l'utilisation de ce modèle ne suffit pas.

Il ne prend pas en compte le fait que le mot peut aussi avoir une fréquence importante dans d'autres cas d'utilisations (d'autres documents).

TFIDF permet de résoudre ce problème en multipliant la fréquence d'un mot par la fréquence inverse du document. C'est pourquoi nous l'avons implémentée.

Lorsque ces étapes ont été réalisées nous avons pu nous attaquer à l'implémentation des différents modèles de classification.

Nous avons décidé de tester 5 modèles différents dans le but de trouver le plus optimal d'entre tous.

En effet, en tentant de répondre à la question « Quel modèle devons-nous utiliser ? », nous nous sommes rendu compte qu'il était impossible de prédire lequel d'entre eux serait le plus efficace sans les expérimenter au préalable.

C'est pourquoi nous avons mis en place les plus connus d'entre eux et ceux qui nous paraissaient les plus appropriés.

Dans l'ordre nous avons testé : *MLPClassifier*, *LogisticRegression*, *KNeighborsClassifier*, *LinearSupportVectorMachineClassifier*, *DecisionTreeClassifier* et finalement *RandomForestClassfier*.

- Nous avons donc débuté par l'implémentation du **MLPClassifier** (Multi-layer Perceptron).
  Ce modèle consiste en plusieurs couches de neurones : une couche d'entrée (*input layer*), une ou plusieurs couches cachées (*hidden layers*) et une couche de sortie (*output layer*). Il utilise la technique de back propagation qui a pour but d'optimiser la fonction log-loss en utilisant différentes méthodes (LBGFGS ou la descente de gradient stochastique)

  En adaptant les données à un modèle basique avec les hyperparamètres réglés par défaut (*un hyperparamètre est un paramètre du modèle utilisé. Ces derniers sont définis avant le processus d'apprentissage*), la précision obtenue est de 0.84.

  Ensuite afin d'optimiser ces résultats nous avons tenté de trouver quels seraient les meilleurs hyperparamètres.

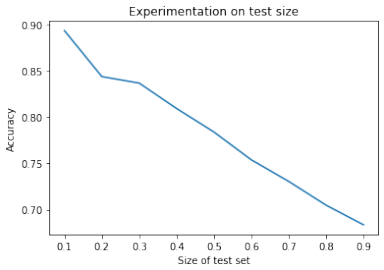
  Pour ce faire nous avons utilisé la méthode *GridSearchCV* proposée par *scikit-learn*.
  Cependant il est intéressant de noter qu'au fil de notre développement nous n'avons pas toujours utilisé cette méthode.
  En effet, il existe aussi *RandomizedSearchCV* et d'autres méthodes de *cross-validation* plus appropriées selon le modèle étudié.

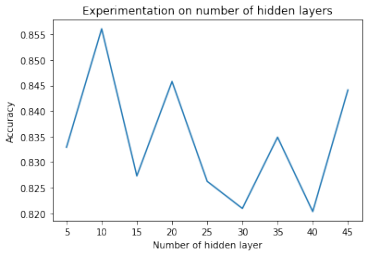
  Avant d'expliquer la différence entre ces deux méthodes il semble important de définir ce qu'est la cross-validation (validation croisée en français) : « La validation croisée permet d'évaluer les performances de l'estimateur. Apprendre les paramètres de prédiction d'une fonction et les tester sur le même jeu de données pose un problème. Le modèle ne ferait que répéter les labels qu'il a vu et un score parfait serait obtenu sur le train set alors qu'il n'arriverait à prédire aucune nouvelle donnée. C'est ce qu'on appelle l'*overfitting* (sur-apprentissage en français dont la définition pourrait être la suivante : *modèle prédictif trop spécialisé qui s'adapte trop bien au training set mais qui ne se généralise que trop peu*) »

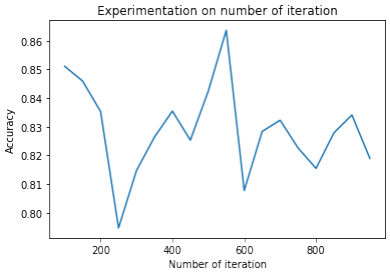
Pour éviter cela, on garde une partie des données pour l'ensemble de test ce qui permettra au modèle de s'entrainer sur de nouvelles données (le jeu de données est séparé en deux parties le *train set*, sur lequel on entraine le modèle, et le *test set* sur lequel le modèle pourra rencontrer des données dont il ne connait pas le label).

Pour revenir aux principales méthodes d'estimations, ces dernières implémentent toutes les deux des méthodes visant à optimiser par *cross validation* les résultats obtenus lors de la recherche des hyperparamètres.
*GridSearchCV* ne va pas tester tous les paramètres contrairement à *RandomizedSearchCV* mais va plutôt le faire sur un nombre donné de paramètres (n_iter).
En comparant les deux (ce sera fait plus tard pour le *RandomForestClassifier*) on se rend compte que *GridSearchCV* obtient un meilleur score final mais a besoin de plus de temps d'exécution alors que *RandomizedSearchCV* sera plus rapide mais obtiendra une estimation plus faible.
C'est pourquoi, selon les cas, il est plus intéressant d'utiliser l'une de ses deux méthodes plutôt que l'autres. Tout dépend de nos besoins.

Cependant nous n'avons pas réussi à obtenir de résultat optimal pour cette classification car la capacité de nos ordinateurs ne nous permettait pas d'exécuter les méthodes *GridSearch* ni même *RandomizedSearch* dans un intervalle de temps raisonnable (entre 24 et 30 heures !).



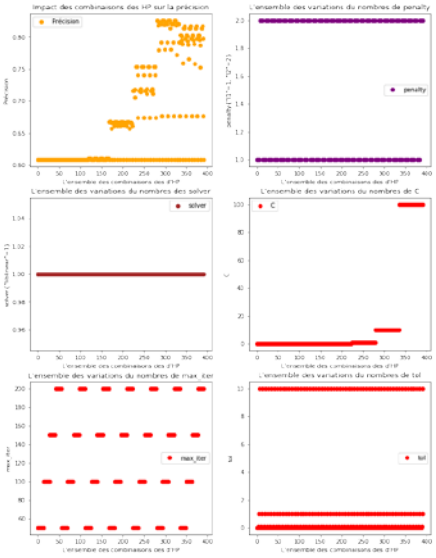

- Dans un second temps, la **régression logistique** est le modèle de classification que nous avons choisis d'utiliser.
  Il s'agit d'un modèle statistique prenant en compte les probabilités d'obtenir une certaine classe plutôt qu'une autre. Il utilise une équation de régression linéaire pour produire une sortie binaire discrète correspondant aux labels.

  Le modèle par défaut obtient une moyenne de 0.76.

MICHELUCCI Romain
TRACOL Théo
SINGLAN Nina

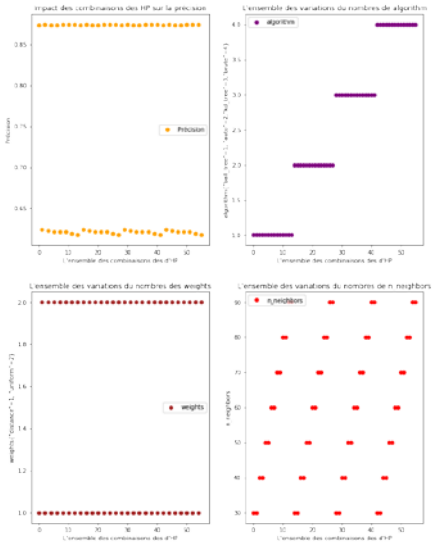


Dans ce cas, afin d'optimiser le résultat obtenu, nous avons comparé le *GridSearchCV* et la méthode de cross-validation spécifique proposée par *scikit-learn* : *LogisticRegressionCV*.
La *GridSearchCV* permet d'obtenir un résultat plus précis (0.87) mais la *LogisticRegressionCV* permet quant à elle d'obtenir un résultat presque aussi précis (perte de 0.01) en un temps plus court.



- **KneighborsClassifier** se base sur la méthode des k plus proches voisins. Cette méthode permet de résoudre le problème de classification suivant : on dispose d'un ensemble de points répartis en deux classes distinctes. Un nouveau point est placé. Nous ne connaissons pas sa classe. L'objectif et de déterminer à quelle classe ce point appartient en connaissant la classe d'appartenance des autres points. (Cela est souvent réalisé à l'aide de calculs distances euclidiennes)
  La précision de base obtenue est de 0.70 environ.

  Après utilisation de la méthode (magique) de la *GridSearchCV* la précision moyenne obtenue est de 0.95 !
  Ce qui en fait le meilleur modèle pour le moment.



MICHELUCCI Romain
TRACOL Théo
SINGLAN Nina

Traitement Automatique du Texte en IA

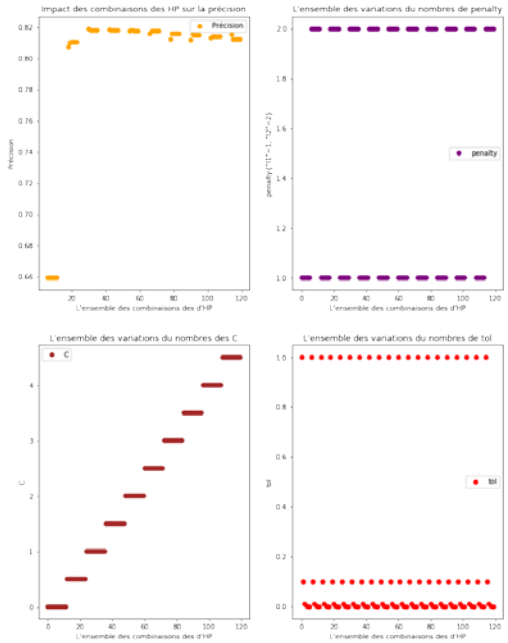
- Ensuite nous nous sommes attaqués au **LinearSupportVectorMachineClassifier.** C'est un algorithme de classification semblable à celui du SVM (Support Vector Machine) qui consiste à trouver les hyperplans/frontières permettant de différencier/séparer les objets de classe. Il permet de mieux s'adapter à un grand nombre d'échantillon.

  Il s'agit du modèle le plus recommandé pour la classification de texte.
  Nous avons trouvé plusieurs points expliquant pourquoi il le serait :
  - Le texte est linéairement séparable
  - Le texte a peu de *features* non pertinentes donc beaucoup d'entrées à traiter

  L'aspect linéaire du SVM permet de créer une classification rapide et efficace.

  Pour preuve en utilisant la méthode de base le résultat obtenu est de 0.84.

  Après avoir appliqué les hyperparamètres obtenus par l'estimateur de la *GridSearchCV*, nous avons conçu une méthode permettant de calculer la précision moyenne et la précision maximale du modèle avec les hyperparamètres optimisés. Nous obtenons une précision maximale de 0.85 environ et une précision moyenne de 0.84.



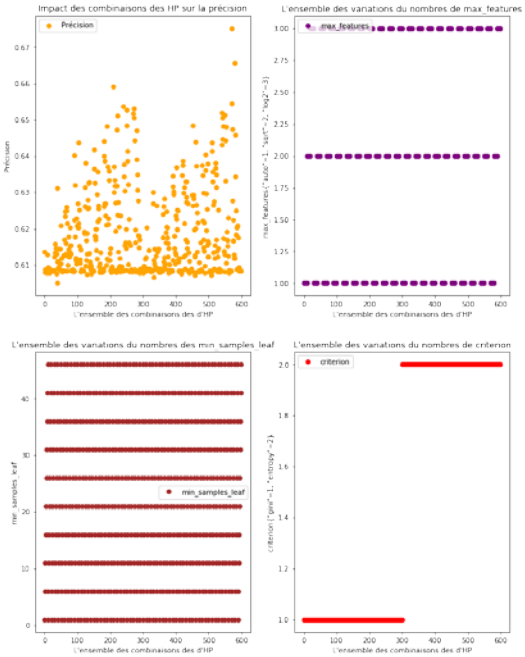

- Avant de finir nous avons testé le **DecisionTreeClassifier**. Cette classification a pour but de créer un modèle qui prédit la valeur d'une variable en apprenant des règles de décision (déduites des caractéristiques données).

L'arbre obtenu est constitué de nœuds qui correspondent à la valeur d'un attribut testé, de branches qui correspondent au résultat du test et de feuilles (nœuds terminaux) qui prédisent le résultat (il s'agit des labels)

L'avantage de cette méthode est qu'elle est assez simple et que l'arbre de décision généré peut être visualisé. De plus il semble parfaitement correspondre au problème de notre projet puisque le résultat doit être binaire : Cyber-Troll or not. (Yes/No type)

Cependant, un des désavantages de ce type de modèle est que l'*overfitting* est facilement atteint. Un autre (observé) est que l'arbre de décision est instable. En effet, de faibles variations peuvent amener à un arbre de décision complètement différent.

C'est pourquoi pour finir nous avons choisi une « version améliorée » de *DecisionTree* pour pallier les problèmes cités ci-dessus :




- Le **RandomForestClassifier**.

Pour rappel, ce modèle est composé d'un ensemble d'arbres de décision issu d'un sous ensemble sélectionné de manière aléatoire et à partir d'un ensemble d'apprentissage (le train set). Il fait la moyenne des scores des différents arbres afin de choisir la classe finale de l'objet testé.

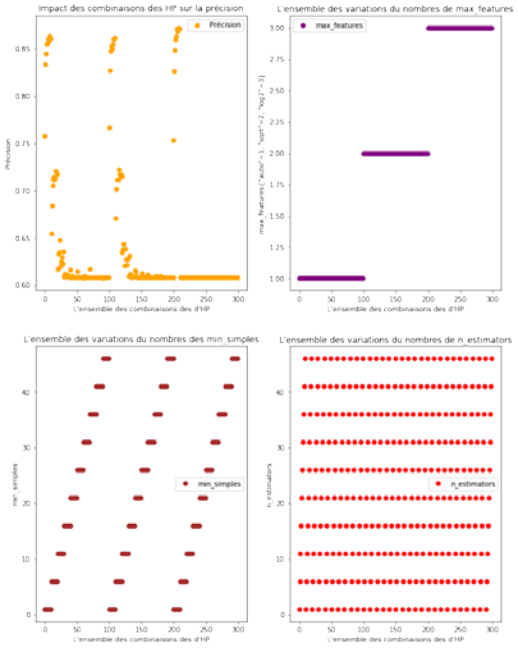
L'un des avantages de *RandomForest* c'est qu'il est capable d'estimer quelles *features*/variables sont les plus importantes lors de la classification.
Il donne généralement une grande précision et surtout permet de contrôler l'*over-fitting* (problème rencontré précédemment).

MICHELUCCI Romain
TRACOL Théo
SINGLAN Nina

Traitement Automatique du Texte en IA

On obtient une classification avec une précision de 0.90 environ.

Les hyperparamètres fournis par *RandomizedSearchCV* permettent d'atteindre un score trop peu satisfaisant de 0.76.
Cependant après avoir testé ceux fournis par *GridSearchCV* la précision atteinte est de 0.92 !




Pour chacun des modèles précédents, excepté pour le *MLPClassifier*, nous avons implémenté une fonction qui calcule la moyenne des performances avec les hyperparamètres. Ces fonctions nous permettent d'obtenir une valeur plus représentative des performances que ce que nous permet l'*accuracy score*.
Cette fonction n'a pas été implémenté pour le *MLPClassifier* car, encore une fois, le temps d'exécution était beaucoup trop long.

**<u>Analyse des erreurs :</u>**

La meilleure précision que nous avons pu obtenir est de 0.95, soit 95%. Ce qui signifie tout de même qu'il reste au moins 5% d'erreurs.
Comment expliquer ces erreurs ?
Dans un premier temps, il semble nécessaire de signaler que les réseaux de neurones arrivent très rarement à une précision de 100% et que, quand cela arrive, c'est souvent du au phénomène d'*overfitting*. Donc, le résultat que nous obtenons est assez satisfaisant, bien qu'améliorable.

Afin d'obtenir une précision encore meilleure, il aurait fallu entrainer nos modèles sur un plus grand jeu de donnée, ce qui aurait permis aux modèles de devenir encore plus performant.

Or, comme précisé plus haut, les réseaux de neurones ne permettent pas un résultat fiable à 100% car, malgré l'entrainement, certaines phrases

« trollesques » ne ressemblent à aucune autres. Ainsi, quelque soit le nombre de « règles » que la machine peut avoir identifiée pour décrire un *Cyber-Troll*, il peut quand même exister des exceptions : des *Cyber-Troll* inédits et donc, non-identifiables par le réseau de neurones.

**Améliorations possibles :**

Il nous aurait été possible de tester plus de classifier afin d'avoir une vue encore plus globale des résultats.

Il aurait également pu être intéressant, avec plus de temps, de tester nos résultats sur de plus grands ensembles de données afin de voir si nos Classifier avait réellement été bien entrainés. De plus, un plus grand jeu de données nous aurait très certainement permis d'obtenir des résultats plus fiables.

Enfin, il aurait été enrichissant de mettre en place un plus grand nombre de méthode de visualisation des données.

**Conclusion :**

Pour conclure ce rapport, nous dirons simplement que nous avons réussi à mener à bien les tâches que nous nous étions fixées.

MICHELUCCI Romain
TRACOL Théo
SINGLAN Nina

Traitement Automatique du Texte en IA

# Sources :

Jeu de données : https://dataturks.com/projects/abhishek.narayanan/Dataset%20for%20Detection%20of%20Cyber-Trolls

Définition du cyber troll : https://www.commentcamarche.net/faq/3610-qu-est-ce-qu-un-troll-informatique

Inspirations : https://stackabuse.com/text-classification-with-python-and-scikit-learn/

Randomized Search VS Grid Search : https://scikit-learn.org/stable/auto_examples/model_selection/plot_randomized_search.html

SVM : http://www.cs.cornell.edu/people/tj/publications/joachims_98a.pdf

https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/

Decision Tree : https://towardsdatascience.com/decision-tree-classification-de64fc4d5aac

https://chrisalbon.com/machine_learning/trees_and_forests/visualize_a_decision_tree/

Logistic Regression : https://towardsdatascience.com/logistic-regression-classifier-8583e0c3cf9

https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegressionCV.html

KNN : https://www.datacamp.com/community/tutorials/k-nearest-neighbor-classification-scikit-learn

LinearSVC : https://scikit-learn.org/stable/modules/generated/sklearn.svm.LinearSVC.html

Doc scikit-learn: https://scikit-learn.org/stable/user_guide.html