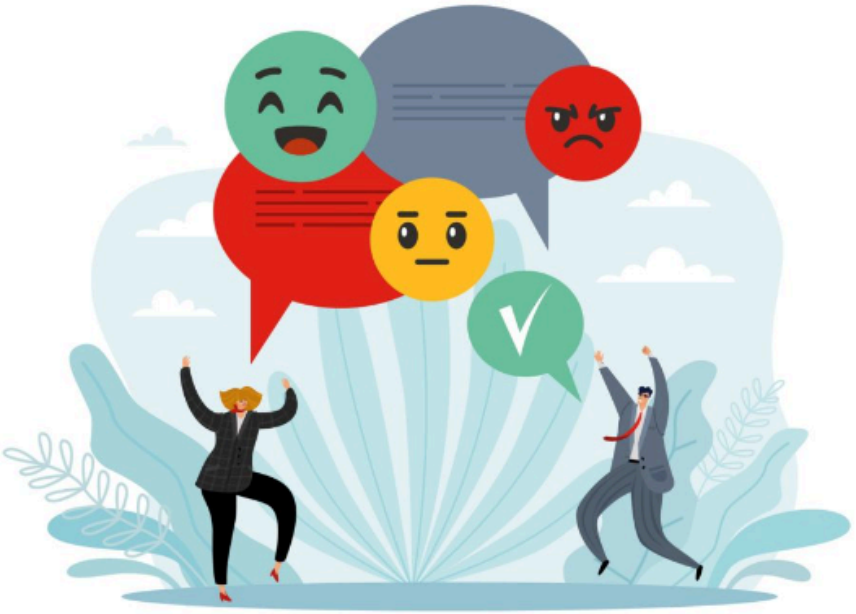
Luan Dechery,
Enrique Nasseh Freitas

NLP : Analyse de Sentiment avec des données de Twitter



M1 MIASHS

13/11/2024

Table des matières

Introduction

Dans le cadre du cours de NLP (Natural Language Processing) du Master MIASHS, on est confronté à utiliser les éléments vus pendant les séances de cours pour mettre en place un projet d'analyse de texte. Plusieurs projets ont été proposés, notre groupe a décidé d'attaquer celui intitulé "French Twitter Sentiments Analysis".

L'analyse de sentiments est une des techniques de NLP qui vise à classifier un texte selon sa polarité (/sentiment). Cette polarité peut être négative, positive ou encore neutre dans certains cas. C'est une méthode d'apprentissage automatique qui rentre dans le champ d'apprentissage supervisé car nécessite de données labellisées pour construire ainsi un classifieur qui puisse prédire la polarité du texte.
Cette technique d'apprentissage machine se montre particulièrement efficace pour traiter les données provenant des réseaux sociaux et pour avoir un échantillon de l'opinion ressentie par une population, en temps réel. Des plateformes comme X (ancien Twitter) deviennent d' importantes sources d'informations pour observer des messages sur des sujets variés, comme les événements d'actualités, les marques ou des figures publiques. Étudier le sentiment dans ces contextes est très intéressant pour détecter des tendances, des fluctuations d'opinion et qui peut donc servir comme outil de prise de décisions pour plusieurs entreprises.

Pour réaliser cette tâche de classification nous disposons d'un jeu de données composé de 1.5 millions de lignes et 2 colonnes. Chaque ligne ayant un texte, tweet en français et sa polarité (/sentiment) correspondant, 0 pour négatif et 1 pour positif. Nous n'avons pas d'informations supplémentaires sur les tweets, on sait juste qu'ils ont été traduits de l'anglais au français.
Plusieurs modèles seront entraînés et comparés pour identifier celui qui s'adapte au mieux à notre problème et ainsi pouvoir l'améliorer pour obtenir des résultats plus pertinents. *Une des principale difficulté est de bien paramétrer notre modèle afin qu'il puisse utiliser les bons mots lors de la prédiction de la classe du tweet. Pour cela, on passera par différentes étapes de nettoyage et ajustement.*

Le projet se divisera en 4 grandes étapes, la première étant le pre-processing qui consiste à nettoyer notre jeu de données. Ensuite, on fera la transformation en matrice numérique (embedding) pour pouvoir réaliser les analyses statistiques. Après, on attaquera la création et la comparaison des différents modèles afin de définir le modèle le plus performant. Enfin, la présentation des résultats, avec l'aide d'éléments visuels tels que les tableaux et les graphiques pour rendre la compréhension plus facile.

Pre-processing

Avant de commencer à faire le nettoyage des données, il faut comprendre la structure d'un tweet. Le tweet est un texte dit libre, c'est-à-dire que la personne qui rédige la phrase peut utiliser n'importe quelle caractère dans n'importe quel ordre. Ceci souligne quelques caractéristiques spécifiques aux tweets qui sont importantes d'avoir en tête lors du traitement de ces textes.

- L'utilisation des majuscules : dans le cas des tweets, les majuscules ne sont utilisées que pour déterminer le début d'une phrase ou d'un nom. Mais aussi pour donner de l'intensité à un mot ou une phrase. (Ex: "Je suis TROP content")

- Des caractères spéciaux : comme les tags et les nombres. Ces caractères ne rajoutent aucune aide lors de la compréhension des phrases, ils seront donc supprimés. (Ex: “@User123 je suis 1000x d’accord avec toi !”)
- Lettres d’autres alphabets : (Ex: “这是一句话”) et des espaces vides (Ex: “non je te comprends pas”). Ces caractères étant inutiles ou inexploitables par notre modèle seront enlevés pour ne pas avoir de problème d’analyse.
- Les emojis. Ils sont souvent très utilisés mais même s’ils apportent plus d’informations sur le texte, on a décidé de les supprimer pour deux raisons. Une est qu’ils peuvent surchargés encore plus notre modèle et il existe d’autres éléments dans la phrase qui sont plus utiles que les emojis pour comprendre le contexte. (Ex: “aujourd’hui je suis heureux :) ”)
- Répétition d’une lettre pour donner plus d'importance à un mot, mais sans forcément rajouter du contexte. (Ex: “J’ai troooop faim”)

Une fois que toutes les caractéristiques ont été énumérées, on va procéder à leur suppression. On a décidé de les enlever pour ainsi avoir un texte qui ressemble plus à un texte “classique” et aussi car on a jugé que ces caractéristiques ne rendent pas le contexte plus enrichissant. Autrement dit, on a filtré le texte pour le rendre plus simple mais sans perdre de profondeur. On aura donc une analyse plus rapide et performante. Par rapport aux emojis, nous avons exécuté un code qui nous a indiqué une présence de 0.08% d’emojis dans tous les tweets de notre dataset, donc nous avons décidé de les supprimer.

Après avoir filtré le texte, on peut attaquer les techniques de réduction des mots comme la lemmatisation et le stemming.

La lemmatisation est un processus qui consiste à réduire un mot à sa forme dite *lemme*, celle qui apparaît dans le dictionnaire. Il faut savoir que cette technique prend en compte la grammaire et le contexte autour du mot pour identifier sa fonction dans la phrase. Ce processus prend un peu plus de temps mais apporte une précision plus importante.

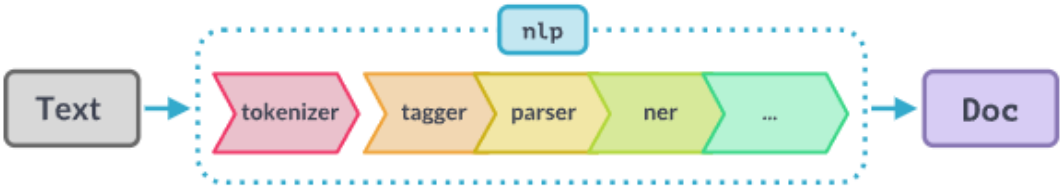
Le stemming est une méthode qui a pour caractéristique la suppression des suffixes et préfixes d’un mot. Ceci dit, l’objectif de ce procès nous renvoie que le radical d’un mot sans prendre en compte le contexte et sa grammaire. Ce qui peut engendrer des problèmes de transformation. NLP Processing Pipeline

Après avoir fait un prétraitement de nos données initiales, nous allons passer à l’étape de la lemmatisation du texte. La lemmatisation est une technique de traitement de texte qui consiste à transformer un mot dans sa forme “canonique” appelé *lemme.* (cf annexe 1), contrairement au stemming, la lemmatisation nous permet d’obtenir une forme standardisée des mots qui pour la suite nous aide lors de l’analyse, en réduisant la taille du vocabulaire du document ou du corpus étudié.

Pour procéder à cette lemmatisation nous allons utiliser le package “spaCy” qui est une bibliothèque de traitement de langage naturel sur Python très populaire pour son efficacité et ses outils. Dans le package, nous rencontrons plusieurs fonctions dont la fonction *nlp()*. Cette fonction produit un pipeline, dont chaque composante nous aide à analyser ou à transformer d’une certaine manière.

Cette fonction ensuite nous renvoie un Doc, contenant toutes ces informations. Pour pouvoir s’en servir de cet outil nous devons d’abord entrer un modèle déjà entraîné. (*nlp = spacy.load(“fr_core_news_sm”)*). Ce modèle va définir ensuite comment les composantes sont exécutées. Chaque partie de ce modèle a une signification, ‘fr’ pour la langue du modèle entraîné, ‘core’ pour exécuter toute la pipeline (contrairement à ‘dep’), ‘news’ pour le type de corpus que le modèle à été entraîné (alternatif à ‘web’) et finalement ‘sm’ pour la taille du modèle (alternatifs :

‘md’, ‘lg’, ‘trf’). Malheureusement pour le modèle français nous sommes limités à utiliser le modèle entraîné avec des données provenant de ‘news’ étant donné que le modèle ‘web’ n'existe pas encore.



Une fois que cette fonction à été exécutée nous avons un texte qui est lemmatisé et que nous allons pouvoir apporter aux modèles pour passer à l’embedding.

Embedding

Les embeddings sont des représentations numériques et vectorielles des mots (/phrases). Les ordinateurs n’ayant pas accès à nos connaissances humaines du langage naturel, nous sommes obligés de passer par cette étape de “quantification” des mots pour qu’ils puissent travailler dessus et les analyser. Il existe plusieurs formes d 'embeddings, des embeddings de mots , de contexte, de phrases… Chacun de ces cas va donc représenter numériquement nos données textuelles d’une façon spécifique et peuvent servir à des buts différents. Les deux principales que nous allons voir ici sont tf-idf et Word2Vec.

Word2Vec est une technique d’embedding développée par des ingénieurs de Google, très populaire couramment et qui nous donne au final des vecteurs denses qui encodent le contexte du mot dans l’espace vectoriel. Cela veut dire qu’à la fin de Word2Vec nous avons accès à des vecteurs qui ont un sens sémantique proche de notre langage naturel, par exemple, le vecteur “homme” serait proche du vecteur “femme” et “chien” proche de “chat”. Word2Vec arrive à avoir ces résultats en combinant deux méthodes différentes; l’une qui prédit un mot à partir de son contexte (CBOW) et une autre qui prédit le contexte à partir d’un mot central (Skip-Gram).

Tf-idf quant à lui est une méthode plus simple et qui ne considère pas le contexte des mots pour les transformer en vecteur. Elle évalue plutôt l’importance d’un terme dans un document par rapport à une collection de documents. Cette méthode nous produit à la fin également des vecteurs numériques mais cette fois peu denses (sparse). C’est une méthode bien adaptée pour les analyses basiques et aussi très utilisée pour faire de la classification. (cf. annexe 2).

Nous avons exécuté les deux méthodes pour ce projet, cependant nous avons eu plus de succès avec la méthode tf-idf donc nous avons décidé de la garder.

Analyse

Après avoir réalisé ces premières transformations, on peut commencer à faire quelques tests. On débute en testant le nombre de lignes sur chaque classe et on se rend compte qu'elles sont bien équilibrées (771604 lignes dans la classe "négative" et 755120 lignes dans la classe "positive"). On continue en réalisant la suppression des stop-words (à l'aide du package nltk) et un graphique appelé nuage de mots pour faire ressortir les mots clés de chaque classe. (cf. annexes 3 et 4)

Pour réaliser cette tâche de classification, nous avons à notre disposition plusieurs modèles prédictifs. Chaque modèle utilise une méthode de prédiction particulière, et l'objectif est d'identifier le modèle, ainsi que ses paramètres, le plus adapté à notre jeu de données pour avoir des prédictions les plus robustes possible.

Notre base de données se divisera donc en 2 grandes parties, Train et Test. L'échantillon train comporte la plus grande majorité des individus (généralement 66%) (cf. annexe 5). Le modèle s'entraîne sur ces données afin de faire ressortir des tendances pour ainsi mettre en place les prédictions. Chaque modèle est ajustable à travers des paramètres arbitraires. L'idée est de faire une comparaison entre chaque modèle et chaque liste de paramètres afin de trouver le couple parfait qui répond au mieux à notre demande.

⚠️ Il est important de noter que l'étape d'embedding se fait concrètement au moment après la division de notre jeu de données, on n'utilise que la partie Train pour entraîner nos modèles pour l'embedding, dans le cas contraire ça serait comme entrainer nos modèles connaissant la moyenne totale de nos données, ceci peut entraîner de *l'overfitting*. Il ne faut pas que le subset Train soit influencé par les données de Test.

Avant de commencer à présenter les résultats, nous allons expliquer le fonctionnement et l'utilisation de chaque modèle.

Gauss:

Ce modèle se base sur la distribution de la loi normale et est souvent utilisé pour modéliser des données suivent cette même loi. C'est-à-dire, des données symétriques autour de la moyenne et dont l'allure de la courbe ressemble à une cloche (distribution gaussienne). Il va utiliser la moyenne et la variance du jeu de données pour estimer la probabilité d'appartenir à une classe données. La probabilité d'appartenir à une classe est estimée à travers la formule de Bayes (cf. annexe 6):

Sur python, 2 paramètres peuvent être ajustable pour avoir une meilleur performance du modèle:

- *priors:* La probabilité à priori d'appartenir à une classe. Si ce paramètre est nul, la probabilité est calculée à partir des données.
- *var_smoothing:* Une valeur arbitraire ajoutée à la variance pour éviter d'avoir une division par 0, et ainsi améliorer la stabilité du modèle. Souvent utilisé pour des données très dispersées.

Le modèle gaussien est très utilisé et est un des principaux modèles prédictifs. Pourtant, dans notre cas, il n'est pas très intéressant car nos données ne sont pas continues et ne suivent pas une distribution gaussienne.

Bernoulli:

Ce modèle est utilisé pour des données données binaires (0 ou 1) et qui suivent une distribution de Bernoulli. Il est très efficace pour des situations de classification de texte où on s'intéresse de savoir si un groupe de mots est présent ou non dans un document, pour ainsi prédire sa classe. Le modèle estime donc la probabilité qu'un tweet appartienne à une classe selon la présence ou absence d'un mot ou un groupe de mots.

Sur python, on peut ajuster 3 paramètres pour obtenir des meilleurs résultats:

- *alpha:* Le paramètre de lissage de Laplace sert à gérer les mots qui sont très peu présents dans certaines classes du jeu de données. Plus cette valeur est élevée, moins les mots rares vont influencer. Plus cette valeur est faible, plus le modèle sera sensible aux mots rares.
- *binarize:* Un paramètre utilisé pour transformer les valeurs d'entrée en valeurs binaires. Très important pour la réalisation d'un modèle de Bernoulli dans le cas où nos données ne sont pas de format binaire.
- *fit_prior:* Paramètre qui contrôle si le modèle doit prendre en compte les probabilités à priori de chaque classe dans les données d'entraînement. Dans notre cas les classes sont équilibrées donc ce paramètre n'a pas d'importance.

Régression Logistique:

Ce modèle est aussi un modèle de classification binaire qui estime la probabilité d'un échantillon à appartenir à une classe donnée. Contrairement à la régression linéaire, qui utilise comme base une ligne droite, la régression logistique se base sur une fonction sigmoïde pour transformer la sortie en une probabilité. Ce modèle utilise la fonction de perte de la vraisemblance pour optimiser ses paramètres.

Sur python il existe 4 paramètres réglable pour optimiser les résultats:

- *C:* Inverse de la force de régularisation. Plus sa valeur est petite, plus on évite le sur-apprentissage.
- *solver:* Paramètre qui spécifie l'algorithme utilisé lors de l'optimisation des coefficients du modèle.
- *penalty:* Type de régularisation utilisé par le modèle (L2: "ridge", pénalité proportionnelle au carré des coefficients. L1: "lasso", la somme des valeurs absolues des coefficients)
- *max_iter:* Nombre d'itérations maximal que le modèle réalisera pour trouver les coefficients optimaux.

Chaque modèle possède des caractéristiques qui s'adaptent au mieux aux jeux de données. L'objectif est de tester chaque modèle avec des valeurs différentes pour chaque paramètres afin de déterminer le couple *modèle+paramètres* le plus approprié à notre demande.

Résultats

Une fois vu le fonctionnement des différents modèles, nous allons réaliser plusieurs tests sur l'ensemble des paramètres pour chaque algorithme afin d'évaluer la performance de chacun d'entre eux. Nous avons choisi arbitrairement une valeur pour chaque paramètre, l'objectif est de définir l'algorithme qui s'adapte et prédit au mieux la valeur de notre "label".

Méthode de Gauss:

On n'a pas pu réaliser de test pour ce modèle car il implique l'utilisation d'une matrice dite "dense". Pourtant pour faire la transformation de notre matrice, une grande quantité de mémoire RAM est nécessaire. Malheureusement la quantité de RAM disponible sur les services gratuits disponibles en ligne tel que Google Colab, ne nous permet pas de réaliser cette tâche. On a essayé de passer par des alternatifs comme, diviser la base de données en plusieurs sous-échantillons mais sans succès. Mais comme vu dans la partie précédente, on juge que cette méthode n'est pas adaptée à notre jeu de données.

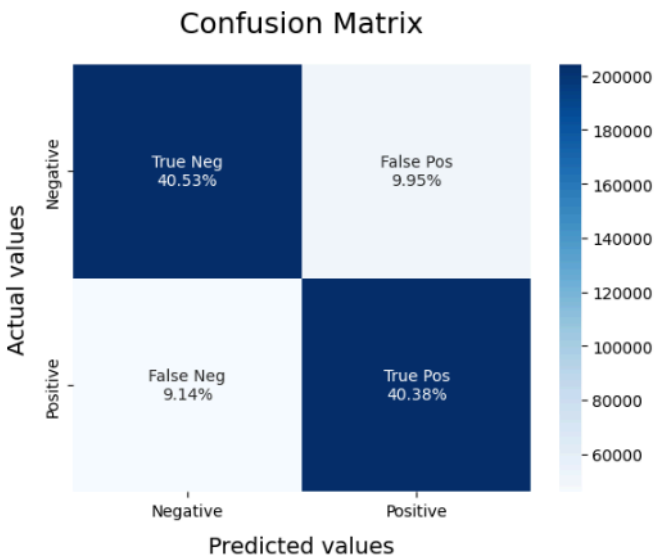
Méthode de Bernoulli et de Régression Logistique:

Pour répondre à notre demande, on a donc comparé 2 modèles prédictifs différents, la méthode de Bernoulli et la régression logistique. Pour cela, on a fait une boucle qui teste plusieurs paramètres choisis de manière arbitraire, et stock la précision (ou *accuracy*) dans un tableau. Ensuite, on peut donc identifier le couple *modèle* + *paramètres* qui maximise la performance.

	Model	alpha	Parameters	Accuracy
0	BernoulliNB	0.5	-	0.775981
1	BernoulliNB	1.0	-	0.775477
2	BernoulliNB	1.5	-	0.775001
3	LogisticRegression	-	C=0.1, penalty=l1, solver= liblinear	0.784040
4	LogisticRegression	-	C=0.1, penalty=l2, solver= liblinear	0.794051
5	LogisticRegression	-	C=1.0, penalty=l1, solver= liblinear	0.808483
6	LogisticRegression	-	C=1.0, penalty=l2, solver= liblinear	0.809152
7	LogisticRegression	-	C=10.0, penalty=l1, solver= liblinear	0.784139
8	LogisticRegression	-	C=10.0, penalty=l2, solver= liblinear	0.799269
9	LogisticRegression	-	C=0.1, penalty=l2, solver= sag	0.794057
10	LogisticRegression	-	C=1.0, penalty=l2, solver= sag	0.809142
11	LogisticRegression	-	C=10.0, penalty=l2, solver= sag	0.799261

On identifie le couple *modèle:* Régression Logistique et *paramètres:* C=1; penalty=l2; solver=liblinear. Ces valeurs nous exposent une précision d'environ 81%, ce qui est un résultat assez satisfaisant. Nous pouvons en déduire des tendances grâce aux paramètres sélectionnées. La valeur faible de C nous permet de faire le compromis entre sous-apprentissage et le sur-apprentissage. On l'aperçoit quand la valeur de C est plus petite (0.1 -> sur-apprentissage) et plus grande (10 -> sous-apprentissage). De plus, contrairement au solver 'liblinear' qui est efficace pour des petits data-sets, le solver 'sag' est très utile et rapide pour des data-sets plus lourds, ce qui est notre cas. Enfin, la valeur de penalty ne peut pas changer car l'utilisation du solveur *'sag'* implique que penalty soit égale à 'l2'. Les valeurs prises par les différents paramètres semblent être cohérentes avec le résultat obtenu.

Pour obtenir une vision plus complexe des résultats avec le couple performant vu précédemment, on peut passer par une matrice de confusion pour vérifier que notre modèle est efficace dans la prédiction de l'ensemble des classes.



Conclusion

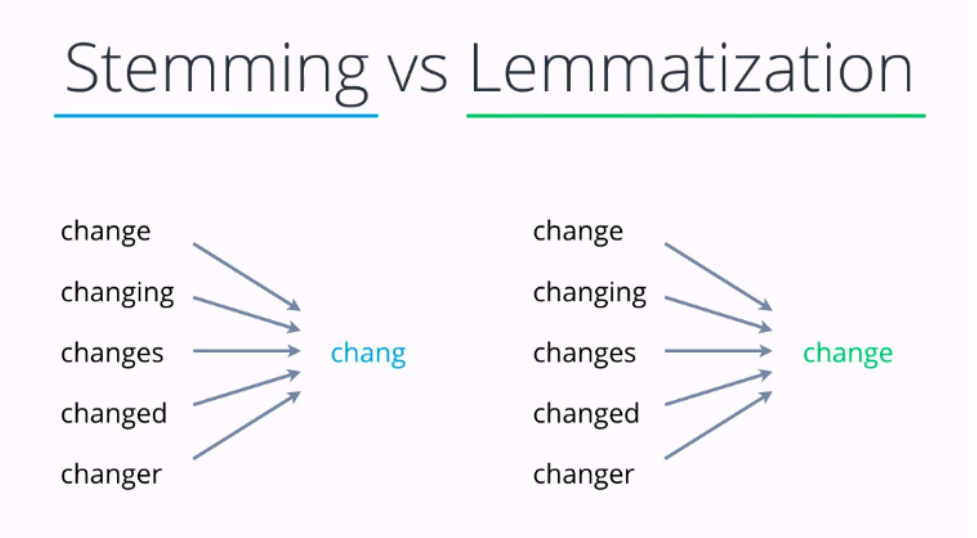
Ce projet d'analyse de sentiment avec des données de Twitter visait à créer un modèle qui arrive à classifier automatiquement le sentiment des messages, une tâche essentielle pour comprendre l'opinion des utilisateurs de réseaux sociaux. Grâce à ces techniques nous avons réussi à aboutir à un modèle qui arrive à discriminer les émotions des tweets avec un taux de 81% ce qui est prometteur. Nous avons pu faire ressortir des tendances à travers l'analyse statistique de l'ensemble des tweets, et créer un modèle qui prédit le label du tweet basé sur son contenu.

Toutefois des pistes d'amélioration subsistent. Il faut savoir que la mise en place du modèle prédictif est assez coûteuse en temps et en mémoire. Nous n'avons donc pas pu utiliser tous les modèles qu'on aimerait comme le modèle de Gauss et d'autres à cause des limites physiques de computation. Enfin, d'autres modèles prédictifs plus complexes pourraient être utilisés, tels que la méthode des K-plus proches voisins, les arbres de décision et la forêt aléatoire. Mais la limitation en temps et surtout en mémoire ne nous a pas permis d' explorer à fond ces méthodes.

Ce résultat encourageant montre la pertinence de notre approche et nous motive à explorer des pistes d'amélioration, comme l'enrichissement des données et le raffinement des hyperparamètres, pour viser une précision encore plus élevée. Ces efforts futurs pourraient ainsi aboutir à un modèle encore plus robuste et précis, offrant un outil précieux pour l'analyse des sentiments en ligne.

Annexes

Annexe 1 :

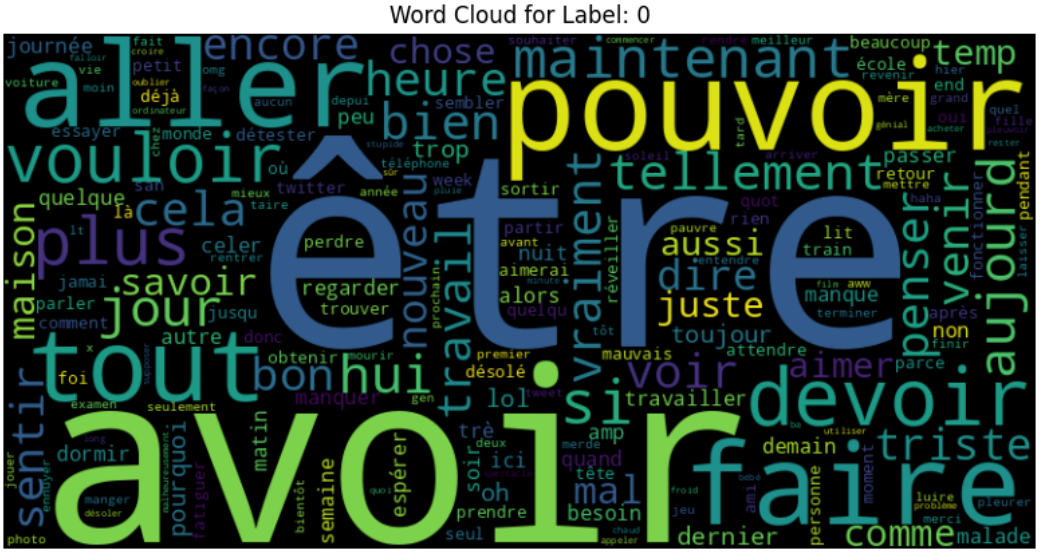


Annexe 2 :

$$w_{i,j} = tf_{i,j} \times \log\left(\frac{N}{df_i}\right)$$

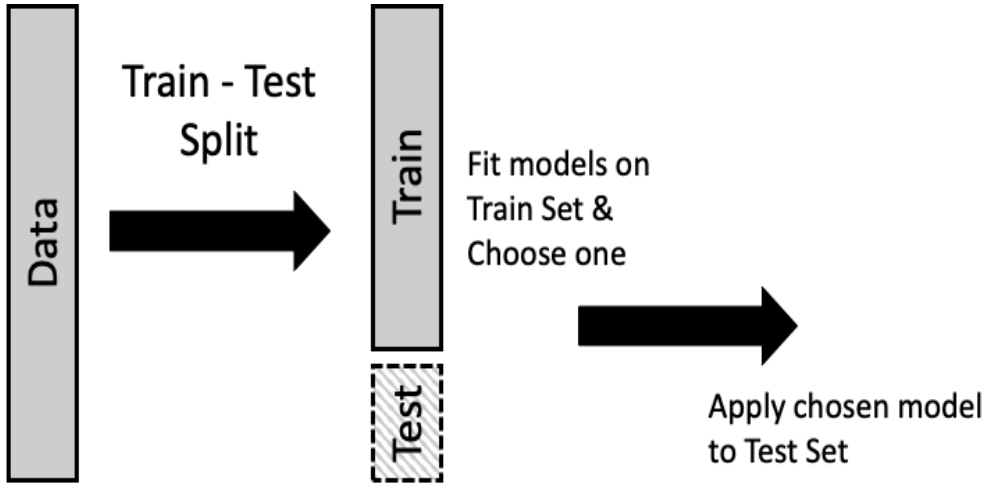
$tf_{i,j}$ = number of occurrences of i in j
df_i = number of documents containing i
N = total number of documents

Annexe 3 :



Annexe 4 :

Annexe 5



:

Annexe 6 :

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$