Rapport PJE C : Analyse de sentiments sur Twitter

Binôme: Simon DECOMBLE, Jules BOMPARD

Git: https://gitlab-etu.fil.univ-lille1.fr/bompard/bompard-decomble-pjec-site

A) Description générale du projet

1. Description de la problématique

Dans un monde qui est de plus en plus connecté, de nombreuses instances cherchent à quantifier l'opinion du grand public sur des produits, des évènements, ou encore des candidats à une élection. La Twittosphère est, depuis plusieurs années déjà, une source continue de feedback sur à peu près toutes les thématiques. Cependant, la volumétrie quotidienne des tweets est si grande qu'il est impossible humainement de parcourir tous les avis sur un produit et d'en déduire les proportions d'avis positifs et négatifs. L'homme doit donc recourir à la machine; et puisqu'il s'agit de données complexes traduisant le langage naturel, il est nécessaire de recourir à des algorithmes d'apprentissage automatique.

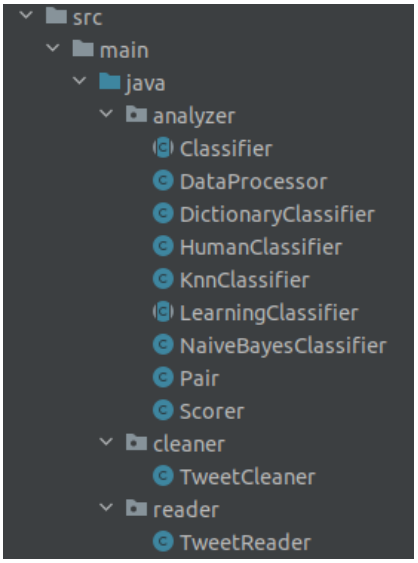
Notre projet se centre globalement sur la problématique qui découle de ce besoin: comment dégager le sentiment général d'un tweet ? Le sentiment général peut être positif, négatif ou neutre. Il s'agit donc d'un problème de classification avec en entrée des données textuelles. Le projet tente donc de répondre à un problème de traitement du langage naturel (NLP).

Dans celui-ci, nous avons donc implémenté la récupération de tweets via l'API Twitter, leur nettoyage avant traitement, puis leur classification en utilisant 3 types de classifieurs dont 2 utilisant des approches usuelles de machine learning.

2. Description générale de l'architecture de l'application

Back de l'application:

Il est composé de 3 modules principaux: reader, cleaner, analyzer.



Reader:
Ce module contient la classe qui requête l'API Search de Twitter pour en extraire le nombre de tweets voulus selon des mots clés fournis. Elle lit donc ces tweets et les écrit dans un fichier CSV avec d'autres métadonnées (ID, auteur, tweet, date, query, classe par défaut). Cette classe est appelée par le main ou l'interface quand la récupération de tweet est nécessaire.

Cleaner:
Ce module contient la classe de nettoyage des tweets, qui consiste à transformer le texte pour qu'il suive une norme propice aux algorithmes de classification textuelle. Les étapes seront détaillées dans la section dédiée.
Cette classe est appelée directement après la récupération des tweets.

Analyzer:
Ce module contient tous les composants nécessaires à l'analyse des textes et à la gestion des structures de données.

La classe DataProcessor contient toutes les fonctions utiles à l'extraction de données (gestion des CSV, extractions de mots et bigrammes) et à la génération de fichier d'entraînement et de test (le train-test split).

La classe Pair modélise un couple de données quelconque, elle nous a servi pour retourner deux valeurs de type différent dans certaines méthodes de classifieurs.

Les classes abstraites Classifier et LearningClassifier modélisent ensuite respectivement l'abstraction d'un classifieur et l'abstraction d'un classifieur apprenant. Ils permettent de stocker des propriétés communes aux classifieurs et aux apprenants (constantes, méthodes génériques). LearningClassifier est bien une sous-classe de Classifier.

Les 4 classifieurs concrets HumanClassifier, DictionaryClassifier, KnnClassifier et NaiveBayesClassifier permettent de classer des fichiers CSV de tweets.
HumanClassifier: il a servi à étiqueter à la main les classes réelles des tweets d'apprentissage (et de test) récupérés, il affiche sur la console le tweet et récupère notre vote (+/-/=) puis applique le numéro de classe correspondant dans le fichier CSV.
Les 3 autres sont des classifieurs basés respectivement sur la méthode par dictionnaire, par k plus proches voisins et par Bayes plus ou moins naïf.

La classe Scorer contient les méthodes de calcul de l'erreur empirique classique train/test (proportion de tweets de test mal étiquetés) et de l'erreur empirique moyenne obtenue par 10-fold (validation croisée, voir explication dans la section C).

Ces classes sont appelées dans notre application pour étiqueter les tweets récupérés et nettoyés, ainsi que pour obtenir les statistiques de classifications et l'évaluation de chacun des trois modèles (temps de calcul et précision). Au sein du module, les classifieurs se servent des méthodes statiques de DataProcessor pour manipuler les données. Scorer utilise les fichiers générés par les classifieurs afin de déterminer les erreurs de prédiction.

La majorité des modules du projet génèrent des fichiers CSV qui servent de support permanent à l'association tweet - classe (même si d'autres structures plus légères portent ces infos dans les algorithmes).

L'application est structurée selon le modèle MVC (Modèle, Vue, Contrôleur).

B) Détails des différents travaux réalisés

1. API Twitter

L'API Search de Twitter est utilisée pour requêter les tweets à partir de mots clés. Puisque nous travaillons en Java, il a été nécessaire d'utiliser un module Java qui interagit avec cette API: **twitter4j.**

Ensuite, nous avons paramétré un fichier **twitter4j.properties** pour accéder à l'API Twitter avec un compte développeur lors de l'exécution de notre code. Ce fichier comprend les clés d'accès: consumerKey, consumerSecret, acessToken et accessTokenSecret dont les valeurs renseignées sont celles fournies par Twitter une fois notre compte développeur validé.

Au niveau du code, les requêtes à l'API ont toutes lieu dans le module reader, dans la classe TweetReader, et plus précisément dans cette fonction.

```
public void getNTweetsByKeyWord(String keyWord, int nbTweets, String filename, boolean append)
    /*
    Recherche *nbTweets* contenant le mot clé *keyWord* en passant par l'API Twitter,
    puis les écrit dans le fichier *filename*
    *append* indique si on ajoute les nouveaux tweets au fichier ou si l'on écrase
     */
    Twitter twitter = TwitterFactory.getSingleton();
    Query query = new Query(keyWord + " +exclude:retweets");
    query.setCount(nbTweets);
    query.setLang("fr");
    QueryResult result = twitter.search(query);
```

On instancie un singleton TwitterFactory partir duquel on peut lancer la requête à l'API Search par la méthode search. La query construite contient le mot clé (ou les mots clés si on met plusieurs mots séparés par des espaces).

Il est possible d'indiquer des paramètres supplémentaires, c'est ce que nous avons fait afin de répondre au cahier des charges et afin de construire une base d'apprentissage la moins biaisée possible, en particulier nous avons procédé à:

- la désactivation des retweets (+exclude:retweets), afin d'éviter les doublons qui auraient été un biais dans l'apprentissage ou l'évaluation
- la fixation du nombre de tweets cherchés (setCount), ce nombre est limité à 100 de toute façon, mais la précision du paramètre est utile dans notre application. On peut donc requêter le nombre de tweets que l'on veut par rapport au(x) mot(s) clé.

- la limitation du langage utilisé au français (setLang), afin de se restreindre au Twitter francophone pour notre traitement du langage naturel (utile pour l'approche par dictionnaire notamment, et nécessite moins de volumétrie).

Nous n'avons pas rencontré de difficulté spécifique pour cette partie. Les limitations sur les retweets et la langue ont été faites a posteriori, par le constat de certains tweets récupérés utilisant des idéogrammes ou présents plusieurs fois car retweetés.

Une fois ces tweets récupérés, la suite de la fonction ci-dessus induit leur écriture dans un fichier CSV où ils pourront être récupérés pour nettoyage et apprentissage.

Le format de chacun des fichiers CSV, supports des données des tweets, est le suivant:

ID du tweet; username auteur; tweet; date de publication; requête Search exécutée; classe

2. Préparation de la base d'apprentissage

Nettoyage:
Le nettoyage a lieu dans le module cleaner, il repose sur les étapes suivantes:

- Suppression des mots dans les hashtags et des noms d'utilisateur. Cela dans le but de limiter le vocabulaire inutile dans la base. En effet, il semble raisonnable de penser que les usernames et thèmes des tweets ne reflètent pas le sentiment général.
- Suppression des valeurs monétaires et pourcentages, remplacement des chiffres par des XX. Idem, n'apportent pas d'information objective sur le sentiment.
- Suppression des URL pour la même raison.
- Ajout d'espaces avant et après la ponctuation. Cela dans le but de délimiter les mots et de ne pas prendre en compte la ponctuation faible au moment de l'apprentissage par exemple.

Ces étapes de nettoyage ont été réalisées par expressions régulières, grâce aux classes java Pattern et Matcher.

Nous avons rencontré une difficulté pour cette étape car dans l'approche par dictionnaires, les chaînes représentant les émoticônes à l'américaine (ex :-)) sont catégorisées comme positives ou négatives. Il a fallu donc les garder et ne pas ajouter d'espace avant et après les deux points suivi d'un tiret et d'une parenthèse par exemple. Cela a mené à une disjonction de cas assez conséquente basée sur des expressions régulières afin de n'ajouter un espace avant et après que dans les cas où l'on n'avait pas d'émoticône.

- Suppression des émojis encodés en UTF-8. Ils peuvent être assez représentatifs de l'émotion dans un tweet mais rajoutent pas mal de complexité aux algorithmes de classification, car ils accroissent le vocabulaire des tweets notamment. Dans le cadre de notre projet, il valait mieux les ignorer. Mais dans un élargissement du projet, il serait intéressant de les prendre en compte, en effectuant un tri sur ceux qui seraient pertinent à garder. Ce nettoyage a été permis par la fonction *removeAllEmojis* du module vdurmont/emoji-java .

Base de données:
Au niveau de la base de données, nous avons récupéré et nettoyé 308 tweets sur 32 thèmes (mots clés) assez variés: *Macron, sciences, covid, musique, football, Lille, gastronomie, arts, voiture, Youtube, télé, histoire, météo, animaux, littérature, soleil, vélo, décoration, politique, course, voyage, livre, jeux, cinéma, manger, chat, fleur, prix, futur, sport, peinture, Paris.* Avec donc environ 10 tweets par mot clé.

Initialement, nous avions 320 tweets non classés (-1 dans le dernier champ CSV) mais 18 ont été écartés soit car étaient des doublons (malgré l'absence de retweets), soit parce qu'ils contenaient une majorité de termes non francophones.

Nous avons étiqueté les 308 tweets à la main grâce à la classe HumanClassifier, certains tweets n'étaient pas évidents à catégoriser par manque de contexte ou par doute concernant le sarcasme et la neutralité de l'auteur concernant le sujet abordé.

Nous avons classifié 107 tweets comme négatifs, 131 comme neutres, 70 comme positifs. La dernière classe est donc légèrement sous-représentée mais c'est peut-être une conséquence du réseau social en lui-même. En effet, les thèmes utilisés pour la récolte de tweets ne sont pas nécessairement des sujets à polémique. Cependant, il y a probablement un biais introduit de notre part dans cette base. Ce biais aurait pu être atténué par une forte volumétrie de tweets mais la classification à la main est assez chronophage et les temps de calcul de nos algorithmes (surtout Bayes) sont déjà conséquents avec 300 tweets. Il est donc plus intéressant, pour le côté démonstration rapide, de se limiter à ces 308 tweets bien que la précision de prédiction s'en voit atténuée.

Une fois cette base générale mise en place, nous l'avons séparée aléatoirement en ⅔ de tweets d'entraînement et ⅓ en tweets de test (DataProcessor.trainTestSplit), comme souvent en machine learning. Cela nous a permis d'évaluer rapidement nos modèles avec apprentissage (KNN et Bayes). En calculant une erreur empirique sur l'échantillon de test, après apprentissage sur l'échantillon d'entraînement. Plus tard dans le projet, nous avons mis en place une évaluation de l'erreur par K-fold qui s'est appliquée cette fois à l'ensemble des 308 tweets étiquetés.

Nous avons donc 2 ensembles d'apprentissage, celui de 308 tweets et celui des ⅔ donc 205 tweets. L'existence des deux permettant d'effectuer une évaluation différente du modèle et de comparer certains résultats. Dans la suite, on nommera ces deux ensembles base classifiée (308 tweets) et base d'entraînement (205 tweets).

3. Algorithmes de classification

Nous avons mis en place 3 algorithmes de classification, dont 2 qui apprennent de la base d'entraînement.

a. Classifieur par mots clé / par dictionnaires

Ce classifieur est le plus naïf, on lui fournit un dictionnaire de mots positifs de la langue française et un dictionnaire de mots négatifs, qui sont des fichiers texte. Notre algorithme crée une liste (ArrayList) de mots positifs et une autre de mots négatifs à partir de ces fichiers. Ensuite, pour chaque tweet de la base classifiée, on en extrait les mots, on initialise

un score de sentiment à 0. Ensuite, on regarde pour chacun d'entre eux s'il apparaît dans la liste positive, si oui, le score est incrémenté de 1, s'il apparaît dans la liste négative, le score est décrémenté de 1. Une fois tous les mots du tweet parcourus, le signe du score fournit la classe du tweet: positif pour un score positif, négatif pour un score négatif, et neutre pour un score nul (autant de termes positifs que négatifs, ou uniquement des termes neutres).

La difficulté rencontrée pour cette méthode résidait dans le fait de décider d'un seuil de score à partir duquel on jugeait qu'un tweet n'était pas neutre. Un raisonnement basé sur l'erreur empirique (développé dans la partie C) nous a convaincu de laisser l'algorithme tel quel et de nous concentrer sur l'amélioration des méthodes suivantes basée sur un réel apprentissage. Le défaut étant que très peu de tweets sont classifiés comme neutres, ce qui contribue à la majorité de l'erreur.

b. K plus proches voisins (KNN)

Il s'agit de la première méthode de classification supervisée, nécessitant donc des données classifiées. On sépare donc l'algorithme en deux phases, celle d'entraînement (fit) et celle de classification / prédiction (predict). Dans le code, ces phases sont clairement séparées pour maximiser la clarté des procédures employées.

Pour cette méthode, la phase d'entraînement consiste seulement à extraire les tweets d'entraînement et leur classe pour les placer dans une liste analogue à un ensemble de points qui serviront de référence de comparaisons (distance) pour tous les tweets de test.

Lors de la phase de prédiction, pour chaque tweet, on calcule la distance à chacun des "points" de cette liste préparée, selon une fonction de distance précise. On cherche ensuite les K plus petites distances obtenues et les tweets correspondants sont donc les K plus proches voisins. Leur classe ayant été sauvegardée dans une liste également lors du fit, on extrait la classe majoritaire qui est celle prédite.

Le nombre de voisins K et la fonction de distance sont d'ailleurs les deux hyperparamètres du classifieur, que l'on peut choisir via notre application. La valeur de K à 5 semble la plus pertinente car elle évite toute égalité entre classes et la fonction de distance est à choisir entre la distance naïve (proportion de mots en commun) et la distance d'édition (nombre d'opérations élémentaires pour passer d'une chaîne à l'autre).

D'autres distances du package opensource simmetrics ont été testées, notamment la distance euclidienne, la similarité cosinus et leur implémentation de la distance d'édition (Levenshtein) mais l'intégration du .jar de simmetrics nous a posé problème lors de la liaison avec l'interface graphique, et ces distances ne donnaient pas une précision significativement supérieure à notre distance d'édition.

Les deux distances restantes ont donc été codées directement dans notre classifieur. La distance naïve compte donc chaque mot en commun dans les deux chaînes et divise ce nombre par le nombre total de mots. Ce qui donne la proportion de mots communs.
La distance d'édition est basée sur une approche de programmation dynamique qui utilise une table de distances entre les deux chaînes en fonction de si l'opération élémentaire sur le caractère est une insertion, une délétion, une substitution (si nécessaire). À chaque étape, la distance minimale obtenue par une de ces opérations est gardée pour la suite jusqu'à la fin

du mot. Cette approche a été inspirée par le travail de Xavier Dupré qui a implémenté cette approche en Python (http://www.xavierdupre.fr/blog/2013-12-02_nojs.html).

c. Bayes naïf et variantes

Dans cette deuxième approche d'apprentissage supervisé, c'est cette fois l'entraînement qui est le plus complexe et chronophage. Le calcul des probabilités (selon la distribution de l'échantillon d'entraînement) de présence de chaque classe et de présence / fréquence de chaque mot sachant une classe prend du temps mais il est ensuite stocké dans une structure qui peut être exploitée en temps constant pour prédire les classes des tweets de test.

L'approche utilisée lors de l'entraînement est celle qui nous a été inculqué lors du projet, en extrayant le vocabulaire des tweets d'entraînement. Les probabilités de présence de chaque classes sont stockées dans un table classProba. Les probabilités de présence ou fréquence de chaque mot sachant chacune des classes sont stockées dans une table probaMap associant un mot à une liste de 3 probabilités (proba sachant classe négative, sachant classe neutre, sachant classe positive).

La prédiction accède en temps constant à ces probabilités dans les structures qui lui sont fournies et réalise le calcul basé sur le théorème de Bayes avec l'hypothèse d'indépendance des features. On peut ainsi en extraire une probabilité d'appartenance à chaque classe pour chaque tweet de test (qui peuvent être affichées en mode verbose) et donc une classe prédite.

Les variantes mises en place sont:

- Le mode fréquence: cas où l'on prend en compte la fréquence des mots dans le tweet pour le calcul dans le théorème de Bayes, ce qui accentue la pondération du sentiment associé à certains mots représentatifs. Pour mettre en place cette option, il a suffit de placer les mots du tweet dans une autre structure: une liste (avec répétition donc) au lieu d'un ensemble (au sens mathématique). En appliquant ensuite le calcul, les termes présents plusieurs fois dans la liste induisent des produits supplémentaires.
- Le mode de limitation de mots: cas où l'on ne prend pas en compte les mots de moins de 3 lettres. Cette option est motivée par le fait que les mots d'au plus deux lettres ne portent souvent pas d'émotion (déterminants par exemple). Cependant, cela peut exclure certaines émoticônes à deux caractères qui sont, eux, très représentatifs d'un sentiment. Pour mettre en place cette option, il a suffi de supprimer les mots de 3 lettres dans le vocabulaire issu des tweets d'entraînement.
- Le type de n-grammes pris en compte: jusqu'ici on calculait la probabilité de présence de mots (ou unigrammes), mais si l'on prend en compte les ensembles de deux mots consécutifs (bigrammes), on enrichit le modèle. Cette option a donc été implémentée tout comme celle prenant en compte unigrammes comme bigrammes. Pour cela, l'approche reste identique mais il a fallu une fonction qui extrait les bigrammes au lieu de mots uniquement, et cela induit un nombre plus élevé de calculs de probabilités.

4. Interface graphique

a. Technologie utilisée

Deux approches différentes ont été travaillées au cours de ce projet concernant l'interface graphique.

Dans un premier temps, nous avons utilisé la bibliothèque native de java Swing. L'idée était de construire un menu central qui permettrait d'accéder à chacune des fonctionnalités, dans une architecture en étoile.

Le menu de l'application sous Swing

Cependant, nous avons décidé de changer de technologie. En effet, les possibilités d'affichage de graphiques avec Swing sont très limitées. Bien que nous ayons pu faire des pages de rentrée des paramètres, nous n'aurions jamais pu afficher les résultats de manière graphique. Le code produit en Swing est toujours disponible sur le premier dépôt.

Nous avons donc migré vers la technologie Jakarta EE. Elle permet de créer des applications web à partir du langage Java en fond, couplée à du HTML, CSS et Javascript pour l'affichage. Cette technologie est donc parfaitement adaptée à notre problématique, puisqu'elle permet d'allier l'utilisation de l'API Twitter4j en java, avec les avantages graphiques des pages HTML/CSS, permettant notamment la création de graphiques avec le langage Javascript.

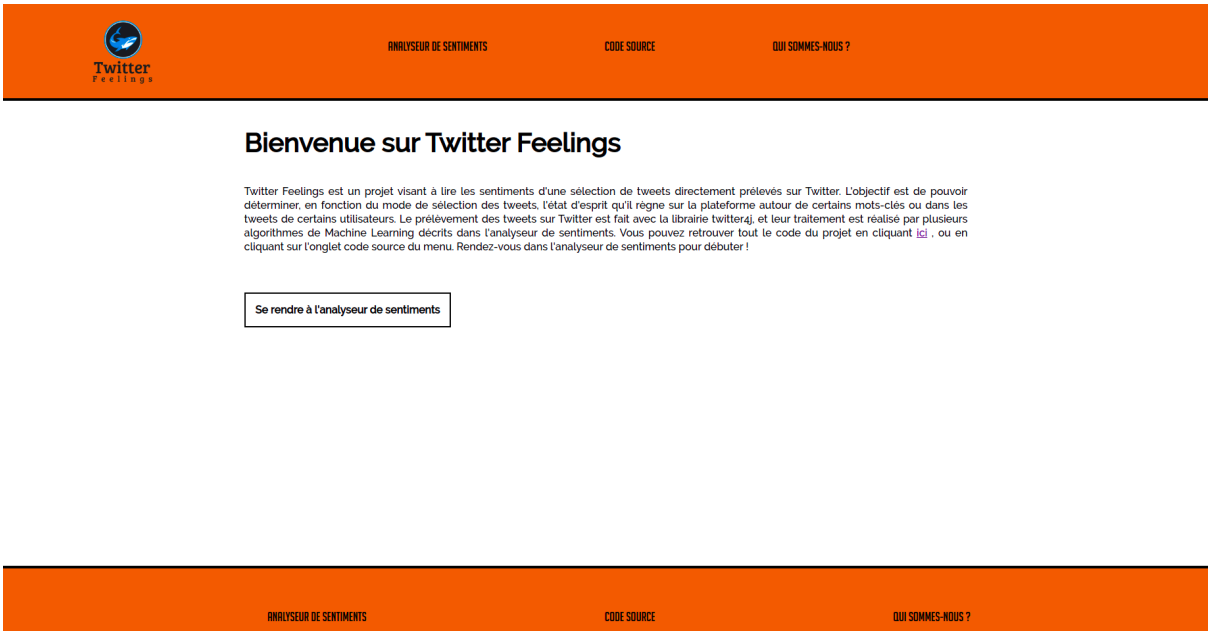
b. Architecture de l'application

L'application est structurée selon le modèle MVC (Modèle, Vue, Contrôleur).

Le modèle, soit les données et les objets Java servant à les exploiter, est représenté par les différents fichiers .csv ainsi que les classes du dossier beans dans les sources. Les fichiers .csv servent de bases de données, ils servent de réceptacles pour l'import des tweets ainsi que pour les tweets nettoyés. Le dossier beans contient toutes les classes Java de traitement, décrites dans la partie A-2.

La vue est représentée par le dossier WEB-INF du projet. Il contient tous les fichiers .jsp, qui sont des fichiers .html dans lesquels on peut ajouter du code java. Ce dossier contient également le fichier CSS de style du site, et un dossier contenant les images.

Enfin, le controller est représenté par le package servlets dans le dossier source. Les servlets sont les objets appelés lors de la demande d'affichage d'une page du site. Les contrôleurs de l'accueil et de la page qui-sommes-nous ne renvoie que les pages jsp statiques correspondantes. Le contrôleur de la page d'analyse contient quant à lui tout le programme qui pourrait être qualifié de "Main" pour le traitement des paramètres et la réalisation de l'analyse d'une sélection de tweets par les trois algorithmes.



Menu de l'application sous JEE

c. Architecture du site

Le site est composé d'un accueil, d'une page pour la sélection / analyse des tweets, et d'une page d'informations sur les créateurs. Deux menus sont présents sur chaque page, pour pouvoir naviguer de page en page sans devoir repasser par l'accueil à chaque fois. Il s'agit donc d'une architecture en graphe complet.

d. Page d’analyse

La page d’analyse est la seule à avoir une construction complexe. Elle dispose de deux vues différentes, la sélection des paramètres et l’analyse des tweets, qui sont déterminées par le contrôleur en fonction des paramètres indiqués. Si aucun paramètre n’est indiqué par la méthode GET, ou si un des paramètres n’est pas valide, la page de sélection est renvoyée. Dans le cas où des paramètres valides sont rentrés, la page d’analyse est renvoyée et le traitement est déclenché.

Les deux pages contiennent la même introduction, qui décrit la manière dont fonctionnent les algorithmes.

Pour la page sélection, un formulaire est présent par la suite, pour y renseigner les différents paramètres nécessaires. Il faut ensuite cliquer sur le bouton “sélectionner” pour démarrer les algorithmes.

Après exécution, la page d’analyse s’affiche. Sans compter l’introduction, elle est composée de trois parties. La première rappelle les paramètres qui ont été sélectionnés. La deuxième comporte un graphique en camembert par algorithme. Ils renseignent l’utilisateur sur les proportions de tweets dans chaque classe, ainsi que leur nombre. Il est également indiqué les temps d’apprentissage et de classification pour chacun des algorithmes. La troisième et dernière partie est un tableau contenant tous les tweets qui ont été classifiés, ainsi que leur classification selon l’algorithme.

e. Fonctionnalités disponibles

Les fonctionnalités disponibles à l’utilisateur se limitent aux critères de sélection des tweets. Celui-ci peut choisir entre sélectionner des tweets selon un ou plusieurs mots-clés (qui doivent tous être contenus dans le tweet) ou sélectionner des tweets émis par un utilisateur en particulier. Dans les deux cas, les tweets sont sélectionnés du plus récent vers le plus ancien. Un maximum de 100 tweets peuvent être prélevés.

Les trois algorithmes, sans choix, sont utilisés lors de l’analyse.

C) Résultats de la classification avec les différentes méthodes et analyse

Pour chacun des trois classifieurs, nous présenterons des tableaux de résultats et leur interprétation.

a. Classifieur par mots clé / par dictionnaires

Pour cette méthode, il n’y a pas d’hyperparamètres, on utilise le même dictionnaire. On peut par contre comparer les métriques obtenues sur la base classifiée et sur la base d’entraînement (⅔ des tweets classifiés).

Base de données	Nb tweets	Temps de classification	Temps moyen par tweet	Erreur empirique
Entraînement	205	62.5 ms	0.30 ms	0.47
Classifiée	308	84.2 ms	0.27 ms	0.49

La durée de classification est très faible, il n'y a en effet pas d'apprentissage. On met environ 0.3 ms à classifier un tweet. L'erreur est proche de la moitié dans les deux cas.

La durée mise dans les deux cas est proche, la différence est négligeable au vu de l'unité. Cette classification étant basée sur une structure externe fixe, la classification d'un tweet se fait en temps constant.

En comparant les deux bases, on se rend compte que l'erreur est légèrement moins élevée pour la base plus petite. La différence est moindre mais si l'on voulait l'expliquer, on pourrait dire que, puisque la méthode n'apprend pas et reste globalement naïve, alors introduire de nouveaux tweets augmente la probabilité de faire une erreur.

En investiguant, on se rend compte que peu de tweets sont classifiés comme neutres par cette approche, la stratégie de l'algorithme pourrait être perfectible.

Nous avons pensé que cette approche pouvait être améliorée en tenant compte de la longueur du tweet et d'un seuil de score à partir duquel on déclare le tweet positif comme négatif, mais les tentatives d'amélioration en changeant le seuil ont empiré l'erreur en réalité. L'erreur empirique de 0.5 sur la base classifiée montre déjà que l'on fait mieux que le hasard (qui fait une erreur moyenne de 0.66), donc nous avons préféré nous concentrer sur les méthodes suivantes plutôt que d'améliorer celle-ci.

b. K plus proches voisins (KNN)

Les hyperparamètres sont ici le nombre de voisins K pris en compte et la fonction de distance. Pour K on choisit de comparer les valeurs 5 et 7 qui imposent qu'il y ait une classe majoritaire parmi les voisins. Pour la fonction de distance, on compare les deux que nous avons implémentées : naïve et édition.

Le calcul de l'erreur empirique sur les 103 tweets de tests.
Le calcul de l'erreur moyenne par 10-folds se fait sur l'ensemble des 308 tweets classifiés.

K = 5, distance naïve:

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	0.02 ms	103	4.64 ms	0.679	0.678

K = 5, distance d'édition:

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	0.03 ms	103	54.60 ms	0.650	0.620

K = 7, distance naïve:

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	0.02 ms	103	3.66 ms	0.71	0.68

K = 7, distance d'édition:

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	0.03 ms	103	56.18 ms	0.71	0.68

Interprétations:

Le temps de fit est constant peu importe les hyperparamètres car il s'agit simplement de récupérer les tweets et leur classe et de les mettre dans la structure de donnée adéquate à la prédiction par KNN.

Effet de K:
L'augmentation de K n'induit pas d'augmentation significative du temps de classification. En effet, peu importe K, toutes les distances sont calculées. La différence d'un point de vue algorithmique est de trouver les 5 ou 7 plus petites valeurs, et cela n'implique quasiment pas de différence en terme de temps de calcul (2 parcours supplémentaires des points restants donc négligeable d'un point de vue algorithmique puisque les calculs de distance portent la majeure partie de la complexité).

En revanche, placer K à 7 induit une augmentation de 5% des 2 erreurs. On prend en compte trop de voisins, et comme les tweets d'entraînement sont trop peu nombreux, cela biaise le résultat et a un effet contre productif.

Effet de la fonction de distance:
La distance d'édition induit un temps de classification plus élevé (d'un facteur 10). Cela est cohérent puisque la technique est plus élaborée: la construction de la table de distance des

2 textes est plus coûteuse en temps que le fait de compter les mots en commun (distance naïve).

Pour K=5 (meilleure valeur de K), l'utilisation de la distance d'édition permet d'augmenter légèrement la précision de prédiction. Cependant, pour K=7, on ne remarque pas d'amélioration. On peut supposer que c'est le K à 7 qui contribue majoritairement à l'erreur.

Les meilleurs hyperparamètres semblent donc être K=5 et distance d'édition au vu de nos estimateurs. L'erreur calculée sur les 10 plis est légèrement inférieure à l'erreur empirique, cela est probablement dû au fait qu'il y ait moins d'échantillons de test et plus de points de références (270 environ à chaque pli, contre 200 pour l'erreur empirique). La présence de nombreux points d'entraînement étant cruciale pour KNN.

C'est seulement avec ces hyperparamètres que notre modèle fait mieux que le hasard sur ce nombre limité de données. En effet, un classifieur aléatoire ferait en moyenne une erreur de 0.66, nous sommes inférieur à cette valeur pour les deux types d'erreur uniquement avec les paramètres optimaux.

Discussion:
L'erreur reste très élevée et supérieure à notre premier classifieur qui n'apprend pourtant pas des données. Cette approche requiert beaucoup d'exemples pour avoir un ensemble de points caractéristiques des sentiments. En effet, le langage naturel est si varié que nos 200 tweets d'entraînement ne fournissent pas une base de points suffisante pour le KNN, d'autant plus qu'un biais est introduit par l'utilisation de certains mots clé pour la création de la base classifiée: deux tweets sur le même thème pourront être jugés plus proches l'un de l'autre car peuvent utiliser le même vocabulaire, alors qu'ils portent peut-être un sentiment différent.

D'ailleurs, quand notre base ne comptait que 100 tweets d'entraînement, les erreurs étaient supérieures à 70%, une multiplication par 2 des données d'entraînement a donc permis un progrès d'au moins 5%, ce qui est encourageant dans la perspective d'augmenter la taille du train set.

On cherchera donc plus de performances avec le modèle suivant basé sur le théorème de Bayes.

c. Bayes naïf et variantes

Les hyperparamètres modifiés sont la représentation (présence / fréquence), les mots pris en compte (tous / au moins 3 lettres) et la suite de mots pris en compte (unigrammes / bigrammes / unigrammes et bigrammes).

Présence, tous, unigrammes :

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	9.34 ms	103	0.19 ms	0.388	0.461

Présence, tous, bigrammes :

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	27.58 ms	103	0.38 ms	0.592	0.574

Présence, tous, uni+bigrammes :

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	41.99 ms	103	0.21 ms	0.388	0.461

Présence, 3 lettres, unigrammes :

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	8.35 ms	103	0.31 ms	0.476	0.471

Présence, 3 lettres, bigrammes :

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	18.5 ms	103	0.37 ms	0.592	0.574

Présence, 3 lettres, uni+bigrammes:

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	31.8 ms	103	0.25 ms	0.476	0.471

Fréquence, tous, unigrammes :

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	8.47 ms	103	0.17 ms	0.398	0.455

Fréquence, tous, bigrammes:

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	26.05 ms	103	0.51 ms	0.592	0.574

Fréquence, tous, uni+bigrammes :

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	39.2 ms	103	0.16 ms	0.398	0.455

Fréquence, 3 lettres, unigrammes :

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	8.59 ms	103	0.28 ms	0.466	0.481

Fréquence, 3 lettres, bigrammes :

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	19.7 ms	103	0.64 ms	0.592	0.574

Fréquence, 3 lettres, uni+bigrammes :

Nb tweet de fit	Temps de fit par tweet	Nb tweets classifiés	Temps de classification d'un tweet	Erreur empirique	Erreur par 10-fold
205	29.3 ms	103	0.33 ms	0.466	0.481

Cette fois, c'est le temps de classification qui est très faible et relativement constant peu importe les hyperparamètres. Cela s'explique par la complexité en O(1) de la prédiction, car il suffit d'accéder à la bonne case dans la structure de données pour obtenir la probabilité souhaitée. Les légers pics de durée observés s'expliquent par la taille de la structure, qui grandit quand on ajoute les bigrammes par exemple. Il a plus de clés donc l'accès mémoire est légèrement plus long.

Les valeurs de temps d'entraînement et d'erreurs dépendent par contre beaucoup de certains hyperparamètres.

Effet du type de représentation:
La prise en compte de la présence ou fréquence dans les tweets n'induit pas de modification du temps de fit.

Les 2 types d'erreurs fluctuent en fonction des autres paramètres, donc on ne peut conclure parfaitement sur l'effet de la représentation sur l'erreur. Dans le cas où l'on a les meilleurs résultats pour les autres paramètres (tous les mots, unigrammes), les erreurs sont légèrement plus faibles en utilisant la présence. Ce qui semble aller à l'encontre de l'intérêt initial d'ajouter le mode fréquence (on s'attendait à voir une amélioration du score).

Effet des mots considérés:
En ne prenant pas en compte les mots de moins de 3 lettres, on a un temps d'entraînement plus faible ce qui est logique puisqu'on calcule moins de probabilités.

En retirant les mots de moins de 3 lettres, on remarque une augmentation de l'erreur (5% environ). Cette option était censée augmenter la performance du modèle (le seul bémol étant la suppression d'émoticônes à l'américaine de moins de 3 caractères). L'explication résiderait alors peut-être dans la présence de corrélations (sans causalité fondée) entre certains mots très courts et un sentiment. Ce biais pourrait être réduit avec beaucoup plus de données d'entraînement et donc un plus riche vocabulaire.

Effet des n-grammes considérés:
La prise en compte de bigrammes augmente le temps d'entraînement, logique puisqu'on augmente la taille du vocabulaire et donc le nombre de probabilités à calculer.

Cependant, encore une fois, on n'améliore pas le score en prenant en compte les bigrammes. D'ailleurs, lorsqu'on les prend en compte sans les unigrammes, on remarque une augmentation nette des erreurs (de plus de 10%). La cause réside probablement encore dans le manque de données d'entraînement, puisque la combinatoire implique que les couples de mots consécutifs sont beaucoup plus nombreux que les unigrammes (de manière quadratique, une approche naïve supposant qu'on puisse avoir n'importe quel couple de mots consécutifs montre que si l'on a n mots dans un vocabulaire, on a 2 parmi n soit n(n-1)/2 bigrammes). Ainsi, une forte volumétrie serait nécessaire pour constater l'effet positif de cette option de prise en compte des bigrammes. Et encore plus si l'on voulait considérer les trigrammes.

Les deux types d'erreur présentent des valeurs proches sauf pour nos meilleurs résultats où l'erreur par K-fold est plus élevée. Nous ne trouvons pas d'explication de cette différence qui va à l'encontre des conclusions émises pour KNN. Cela est peut-être dû à des fluctuations statistiques et peut-être que l'erreur empirique est anormalement faible pour le train test split utilisé, le K-fold serait alors plus représentatif de la réelle erreur, puisqu'il moyenne les résultats issus de 10 séparations distinctes.

Bilan sur Bayes:
Les meilleurs hyperparamètres pour cette méthode sont les paramètres initiaux, les plus naïfs: présence, tous les mots, unigrammes. Ils impliquent un temps d'entraînement faible et des scores très satisfaisants pour le peu de données: aux alentour de 40% d'erreur. Mais attention, cela n'est valable que pour notre jeu de données de taille relativement limité pour une base d'apprentissage de langage naturel.

Une piste éventuelle est d'augmenter la base d'apprentissage, cependant, cela rendrait l'évaluation du modèle encore plus longue. Actuellement, pour certains hyperparamètres, le temps de fit des 200 tweets d'entraînement est de 8 secondes, ce qui induit une évaluation de l'erreur par 10-fold d'une durée de 1min30.

D) Conclusions

A travers ce projet, nous avons pu constater la difficulté de classifier du langage naturel humain, d'autant plus selon un critère qui est tout aussi humain: le sentiment.

Nous avons dû accéder à une source d'information riche: Twitter, à travers son API Search. Une fois ce lien effectué, nous avons pu récupérer des tweets selon des thématiques précises. Un processus de pré-traitement a été appliqué pour formater le texte et le rendre propice à la classification par différents algorithmes.

Le premier classifieur, par dictionnaire, se basait sur une connaissance humaine, à savoir le sentiment dégagé par certains mots. Sur les données récupérées, il classait correctement 1 tweet sur 2.

Ensuite, un classifieur basé sur la méthode des K plus proches voisins a été mis en œuvre. Cette fois, il apprenait des données d'entraînement étiquetées à la main. Son taux d'erreur de 65% s'est révélé plus faible que le premier classifieur, mais meilleur que le hasard tout de même. L'augmentation de la volumétrie d'entraînement semble une bonne piste pour faire tendre cette méthode vers de meilleurs résultats, en dépit d'un temps de calcul plus long.

Enfin, le dernier classifieur était basé sur l'approche Bayésienne, et repose sur la distribution des données d'entraînement en fonction du sentiment. Cette méthode s'est révélée très efficace malgré un nombre limité d'exemples, avec environ 40% d'erreur en moyenne. Certaines variantes ont été explorées afin d'améliorer le score du modèle naïf, cependant elles n'ont pas porté leurs fruits.
On suppose que ce constat est dû au nombre limité de tweets, les modèles de Natural Language Processing nécessitent souvent des milliers voire millions d'exemples pour être très performants. Nos algorithmes sont encore trop gourmands en temps et l'étiquetage d'un nombre si important de données est fastidieux. Cependant, le taux d'erreur de 40% seulement à cette échelle prouve la puissance de la classification Bayésienne naïve.