Analyse à grande échelle de sentiments sur Twitter avec Pyspark

Clément Delteil (DELC12110004)
Thomas Sirvent (SIRT10080007)

14 décembre 2022

Résumé

Dans ce devoir, nous allons explorer les différentes problématiques du *Natural Language Processing*, ou NLP, à travers une analyse de sentiment de tweets récupérés sur le réseau social Twitter. Comme présenté dans l'article sujet de ce devoir [1], nous travaillerons avec Apache Spark afin de tester différentes techniques et ainsi déterminer lesquelles sont les plus intéressantes. Pendant tout ce devoir, nous nous mettrons dans un contexte de Big Data, où nous traiterons des volumes de données importants, prouvant encore une fois l'utilité de Spark et son implémentation en Python, PySpark.

1 Introduction

Les opinions que nous exprimons sur Internet sont des mines d'or, on décrit des choses que nous aimons, détestons, ou bien nous parlons de nous-mêmes, laissant aux autres la possibilité d'en apprendre plus sur nous. Toutes ces informations que nous partageons peuvent être utilisées à des fins d'amélioration continue, d'innovation ou tout simplement de nous comprendre en tant que consommateurs. Les entreprises, pour ne citer qu'elles, sont donc très favorable à la recherche et au développement des techniques de traitement et d'analyse de texte.

Cependant, les problèmes de NLP sont difficiles à traiter, principalement dû à la complexité de notre langage. En effet, un cas assez simple illustrant cette complexité est l'exclamation "J'en meurs d'envie !", un humain comprendra sans doute que c'est une phrase positive, ou négative selon l'intonation (sarcasme). Un ordinateur lui, s'il ne peut utiliser que des méthodes de bases comme la consultation d'un dictionnaire, conclura toujours sur la négativité. C'est donc un domaine qui est constamment en évolution et qui stimule la recherche.

Nous verrons ensemble dans ce devoir quelles sont les bonnes pratiques afin de tirer parti au mieux des techniques disponibles actuellement, tout en prenant en compte la problématique du Big Data.

2 Résumé de l'article

Motivé par les mêmes arguments, l'article présente une méthode parallélisée et distribuée de l'algorithme des *k-Nearest Neighbours* ou kNN, pour l'analyse de sentiment de tweets. Afin de pouvoir assurer l'utilisation de cet algorithme avec des données massives, ils utilisent également un *Bloom Filter*, qui est une structure de données probabiliste dont le principe est similaire au hachage. Ses avantages principaux sont multiples : on est certain de l'absence d'un élément (mais pas de sa présence), et sa taille est fixe, peu importe le nombre d'éléments qu'on lui ajoute. C'est ce dernier, exprimant une certaine compression de la donnée, qui a attiré les chercheurs à utiliser cette structure. L'incertitude que l'on a sur la présence d'un élément présente bien évidemment un désavantage, mais leurs tests ont prouvé que les imprécisions que ce défaut apporte sont largement compensées par le gain en temps d'exécution. Il y a également un léger gain de stockage, mais étant donné qu'un *Bloom Filter* est crée pour chaque caractéristique (comprendre pour chaque mot différent dans l'ensemble de tweets), ce gain n'est pas significatif.

L'article décrira son expérimentation en 4 étapes.

2.1 Extraction des caractéristiques

Étape vitale du processus des problèmes NLP, ici, on prendra en compte la dimension réseau social dans le jeu de données. En effet, les tweets ne sont pas des textes littéraires, il faut donc enlever les URLs les hashtags, et le mot "RT" (retweet) afin de rendre les caractéristiques plus simples à analyser et à comparer. Cependant, la ponctuation, si assez présente, est gardée comme caractéristique, car elle peut être une forme d'expression. Une fois les données traitées, l'article identifie les caractéristiques en 3 catégories selon leurs fréquences d'apparitions dans le jeu d'entraînement : *high-frequency,*

content, regular.

2.2 Feature Vector Construction

Cette étape consiste à maintenir une structure de données qui maintient pour chaque caractéristique, agissant comme clé, d'obtenir une valeur, qui sera la liste de tweets contenants la caractéristique concernée. L'union de ces listes deviendra le vecteur de caractéristiques. Pour chacune d'entre elles, on divisera en deux parties ce vecteur pour créer une liste pour l'entraînement et une pour les tests.

2.3 Distance Computation

Cela correspond à l'utilisation de l'algorithme kNN. Pour tout élément sujet, on fera un calcul de distance Euclidienne entre différents vecteurs qui possèdent des similarités, et on regroupe les k éléments les plus proches.

2.4 Sentiment classification

Pour finir, grâce aux éléments rassemblés dans l'étape précédente, on assigne à l'élément sujet le label de la majorité des k vecteurs.

L'aspect parallélisation apparaît à travers l'utilisation de fonctions Spark (*map, flatMap*) qui exprime la méthode *MapReduce*. *Map* correspondra à construire des paires clés-valeurs où la clé est le *label*, et les valeurs seront les distances entre features. L'étape implicite du *Shuffle* regroupera toutes ces paires selon les clés, et *Reduce* choisira les k distances les plus proches et fera un vote majoritaire afin de déterminer le *label* correspondant.

3 Jeux de données

3.1 Sentiment140

Sentiment140 [2] est un jeu de données très connu quand il s'agit d'analyser les sentiments de tweets. Il a été créé par Alec Go, Richa Bhayani et Lei Huang, qui étaient étudiants en informatique à l'université de Stanford. Depuis, il a été utilisé dans un certain nombre d'articles scientifiques et sert ainsi de point de référence pour comparer la précision des modèles.

Le jeu de données comprend 1600000 tweets sous le format suivant :

- la polarité du tweet (0 = négatif, 2 = neutre, 4 = positif)
- l'ID du tweet
- la date du tweet
- la requête utilisée
- l'utilisateur qui a tweeté
- le texte du tweet

Pour le calcul de la polarité, plutôt que de passer par un long processus d'annotation par un humain, ils ont supposé que tous les tweets contentant des émoticônes positives, comme *:)* sont positifs, et que les tweets contenant des émoticônes négatives, comme *:(*, sont négatifs.

3.2 Custom

Dans l'idée de rendre ce projet plus concret et proche des problématiques réelles de l'analyse sentimentale, nous avons décidé de récolter en parallèle nos propres Tweets via l'API de Twitter et ainsi de créer notre jeu de données.

Pour ce faire, nous avons utilisé la librairie *Tweepy* [3], qui facilite la communication avec l'API de Twitter. Lors de la recherche de tweets sur le réseau social Twitter, nous pouvons passer en paramètre de la requête des paramètres afin d'aiguiller la rechercher et d'éviter de récolter des informations non-pertinentes. Notre requête fut la suivante :

Listing 1 Requête API

```
("elon musk" OR "@elonmusk" OR "elonmusk")
-is :retweet lang :en
```

Nous avons choisi de nous intéresser à la personne d'Elon Musk, car c'est un personnage clivant, qui du fait, de son récent rachat de Twitter, suscite beaucoup de réactions et donc de tweets potentiellement marqués émotionnellement parlant.

Plus en détails, notre requête va récupérer tous les tweets en anglais contenant soit "elon musk" soit "@elonmusk" soit "elonmusk" et qui ne sont pas des retweet.

Nous avons ensuite planifié l'exécution de notre script Python à intervalles réguliers sur une période d'environ 3 semaines pour récolter près de XX tweets.

4 Pré-traitement

Dans cette section, nous ne distinguerons pas les deux jeux de données à notre disposition, on considérera alors que toutes ces étapes ont été appliquées à chacun d'eux.

4.1 Nettoyage

Nous avons appliqué les étapes classiques de nettoyage de données ainsi que quelques autres liées aux propriétés uniques des tweets.

Parmi les étapes classiques, nous avons :
- supprimé la ponctuation comme *. , ! $ ( ) * % @*
- supprimé les URLs
- supprimé les mots vides ("a", "the", "is", ...)
- transformé toutes les lettres en minuscule
- découpé le texte en unité lexicale élémentaire (Tokenization)
- réduit les mots à leur forme radicale si possible (Stemming)

Plus précisément, pour le cas du caractère *'@'*, nous avons dû le traiter comme un cas à part, car, dans les tweets, il est généralement suivi du nom d'un autre utilisateur. Or, ce nom d'utilisateur n'apporte pas d'indications sur le sens émotionnel d'un tweet. Nous avons donc systématiquement supprimé la chaîne de caractères collée à un *'@'*.

Aussi, nous avons choisi le *Stemming* plutôt que la *Lemmatization* car, d'après nos recherches, il est beaucoup plus rapide. En effet, il s'agit d'un algorithme assez simple basé sur les cas en comparaison à la *lemmatization*, qui est beaucoup plus coûteuse, mais n'offre pas d'amélioration proportionnelle à l'augmentation du temps de calcul.

Toutes ces étapes nous permettent de drastiquement réduire le nombre de caractéristiques présentes à la fin de la pipeline et ainsi d'éviter le sur-ajustement du modèle pour prioriser la généralisation.

4.2 Extraction des caractéristiques

Dans cette section, nous allons présenter toutes les caractéristiques que nous avons tenté d'extraire à partir des données en entrée. Toutes n'ont pas été conservées pour le modèle final (voir 6).

4.2.1 HashingTF vs CountVectorizer

HashingTF et *CountVectorizer* peuvent tous deux être utilisés pour générer les vecteurs de fréquence des termes. Cependant, ils diffèrent sur quelques points.

HashingTF convertit les documents en vecteurs de taille fixe. La dimension par défaut des caractéristiques est de 262144. Les termes sont mappés en indices en utilisant une fonction de hachage. La fonction de hachage utilisée est MurmurHash 3. Les fréquences des termes sont calculées par rapport aux indices mappés.

Tandis que CountVectorizer, lui, va tout d'abord générer un vocabulaire. Cette étape sera suivie de l'ajustement du modèle CountVectorizer. Au cours du processus d'ajustement, CountVectorizer sélectionnera les principaux mots de VocabSize classés par fréquence de terme. Le modèle produira un vecteur qui pourra être utilisé par la suite.

En somme, HashingTF est moins coûteux, car il utilise une fonction de hachage, mais est irréversible et peut causer des collisions, tandis que CountVectorizer, lui, est déterministe et réversible, mais est beaucoup plus coûteux.

4.2.2 IDF

Les vecteurs passent ensuite par la fonction IDF (*Inverse Document Frequency*) qui va, selon la formule ci-dessous, filtrer les termes qui n'apparaissent pas un nombre minimum de fois.

$$idf = log(\frac{m+1}{d(t)+1}) \tag{1}$$

Où m est le nombre total de documents, et $d(t)$ le nombre de documents qui contiennent le terme t.

4.2.3 N-gramme

Pour donner plus de contexte au modèle, on se sert des *sacs de mots* ou *N-gramme* pour regrouper les mots en groupe de 2, 3, 4 voire plus de mots consécutifs. C'est ce qu'on appelle des bi-grammes, trigrammes et quadrigrammes.

4.2.4 ChiSqSelector

En fin de pipeline, nous avons aussi expérimenté la sélection des meilleures caractéristiques à l'aide la fonction *ChiSqSelector* de Spark. En effet, avec la création des N-gramme présentés plus haut, le nombre de caractéristiques commençait à devenir beaucoup trop grand. Nous avons donc décidé de sélectionner les meilleures afin de réduire l'espace de recherche.

5 Modèles

5.1 Logistic Regression

La version Spark de la régression logistique est plus efficace que celle implémentée dans Scikit-learn, car elle parallélise la descente de gradient.

Plus précisément, Spark utilise une méthode dite de descente de gradient stochastique en mini-lot. Pour justifier ce choix d'implémentation, nous allons rappeler les différentes méthodes de descentes de gradient qui existent.

5.1.1 Descente de Gradient par Lots

La descente de gradient par Lots, alias, "Batch Gradient Descent", calcule de le gradient de la fonction de coût par rapport aux paramètres θ pour l'ensemble du jeu de données d'apprentissage :

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{2}$$

Comme nous devons calculer les gradients pour l'ensemble du jeu de données pour effectuer une seule mise à jour, la descente de gradient par lots peut être très lente et est intraitable pour les jeux de données qui ne tiennent pas en mémoire. Elle ne permet pas non plus de mettre à jour le modèle en ligne i.e. avec de nouveaux exemples à la volée.

5.1.2 Descente de Gradient Stochastique

La descente de gradient stochastique, alias, "Stochastic Gradient Descent", en revanche, effectue une mise à jour des paramètres pour chaque exemple d'apprentissage $x^{(i)}$ et label $y^{(i)}$:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i)}; y^{(i)}) \tag{3}$$

La descente de gradient par lots effectue des calculs redondants pour les grands ensembles de données, car elle recalcule les gradients pour des exemples similaires avant chaque mise à jour des paramètres. La descente de gradient stochastique élimine cette redondance en effectuant une mise à jour à la fois. Elle est donc généralement plus rapide et peut également être utilisée pour l'apprentissage en ligne.

5.1.3 Descente de Gradient par Mini-Lot

La descente de gradient par mini-lot, alias, "Mini-batch Gradient Descent", prend finalement le meilleur des deux mondes et effectue une mise à jour pour chaque mini-lot de n exemples d'apprentissage :

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{4}$$

À chaque itération, Spark échantillonne un sous-ensemble (*miniBatchFraction*) des données totales afin de calculer une estimation du gradient. L'échantillonnage et le calcul de la moyenne des sous-gradients sur ce sous-ensemble sont effectués en utilisant le standard MapReduce à chaque itération. Par défaut, le paramètre *miniBatchFraction* a une valeur de 1.0 [4].

Ci-dessous, vous trouverez l'architecture simplifiée présentée lors du sommet Spark 2017.

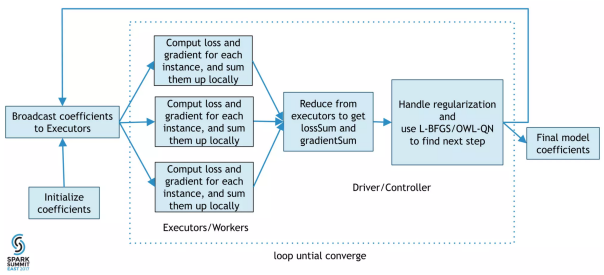

FIGURE 1 – Distribution du calcul du gradient [5]

Si l'on résume trivialement sous la forme d'instructions, on a :

1. Sélectionnez un échantillon de données
2. Calculez le gradient sur chaque ligne de l'échantillon.
3. Agréger le gradient
4. Retour à l'étape 1

5.2 Support Vector Machines

D'après nos recherches, il semblerait que le modèle SVM implémenté dans Spark, utilise le même type d'optimisation et de parallélisation présentée pour le modèle de régression logistique. C'est-à-dire que le processus parallélisé est la descente de gradient au sein du modèle en lui-même [6].

On peut par ailleurs noter qu'à l'heure où nous écrivons ce rapport, seul le *kernel* linéaire est disponible. En effet, nous avons fouillé le forum de développement de Spark où les développeurs peuvent demander des ajouts à librairie ou bien proposer leurs propres implémentations des algorithmes, et nous sommes tombés sur une publication de 2017 où un membre expliquait que les *kernels* non-linéaires étaient difficiles à distribuer. Les méthodes naïves nécessitent beaucoup de communication. Pour obtenir cette fonctionnalité dans Spark, ils doivent d'abord devoir faire une recherche de fond appropriée et rédiger un bon design. Il ajoutait aussi que d'autres algorithmes ML étaient sans doute plus demandés et nécessitaient encore des améliorations (à la date de ce commentaire). Les ensembles d'arbres étaient les premiers et les plus importants dans son esprit [7].

5.3 Naive Bayes

Nous n'avons pas trouvé de ressources parlant de la parallélisation du classificateur naïf bayésien. Cependant, étant donné que le modèle considère toutes les caractéristiques indépendantes et qu'il se sert du Théorème de Bayes qui dit que :

$$P(\mathbf{A}|\mathbf{B}) = \frac{P(\mathbf{B}|\mathbf{A})P(\mathbf{A})}{P(\mathbf{B})} \tag{5}$$

i.e. trouver la probabilité de A étant donné B.

Les probabilités de chaque attribut peuvent être calculées indépendamment grâce à l'hypothèse d'indépendance. Nous pensons que le calcul des probabilités de chaque caractéristique se fait séparément et est organisé par le système MapReduce. La division du travail accélère ainsi les calculs et permet de traiter plus facilement de grands ensembles de données.

6 Evaluation

6.1 Précision des modèles

Les modèles ont été évalués en local pour une question de simplicité. Vous trouverez ci-dessous dans la Table 1 la comparaison des modèles présentés dans la section 5. Cette comparaison se base sur la métrique de précision et évalue les modèles dans chacun des scénarios de caractéristiques imaginés.

Comme le montre nos résultats expérimentaux, c'est la *Régression Logistique* qui est le modèle le plus adapté à nos données avec une précision de 80.8% dans le dernier scénario. Aussi, nous avons expérimenté dans chacun des scénarios, différentes combinaisons d'hyperparamètres. Lorsque nous expliquions la différence entre *HashingTF* et *CountVectorizer* dans la Section 4.2.1, nous avons précisé que la première peut causer des collisions. C'est cette différence qui explique selon nous les différences de précisions que l'on peut noter entre les scénarios avec HashingTF contre CountVectorizer, qui est certes plus long, mais ne perd aucune informations.

Dans le dernier scénario, voici plus en détails les paramètres de nos fonctions d'extraction d'hyperparamètres.

— CountVectorizer : taille du vocabulaire = 2^{14}
— IDF : Fréquence minimum = 5
— N-gram : 1-Gram, 2-Gram, 3-Gram
— ChiSqSelector : Top 2^{14} features soit 16384

Soit un total 49152 features avant la sélection puis 16384 après le passage par ChiSqSelector dans la pipeline.

Plus en détails, vous trouverez ci-dessous deux visualisations des résultats du modèle de régression logistique.

Dans un premier temps, nous avons affiché les courbes ROC pour chacun des scénarios avec le modèle de régression logistique. Vous pouvez retrouver en abscisse le taux de faux positifs et en ordonnée le taux de vrais positifs. Les modèles présentés en légende sont dans le même ordre que dans le tableau 1. On peut remarquer que ce n'est pas le modèle avec la plus grande précision qui obtient la plus grande valeur AUC.

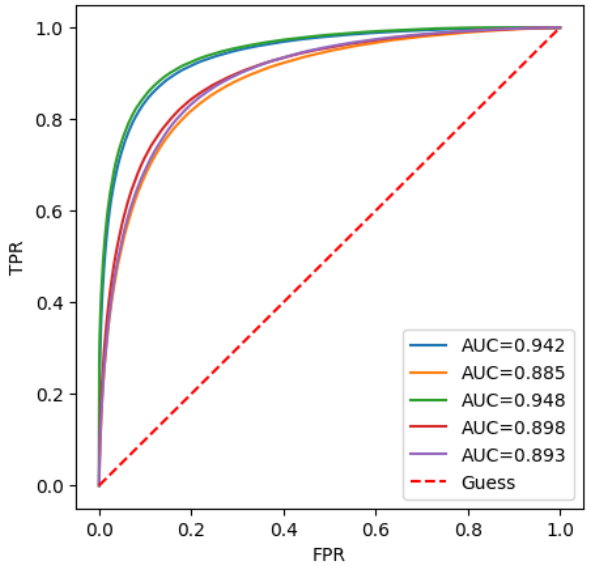

FIGURE 2 – ROC Curves

Plus en détails, si on se concentre sur le modèle de régression logistique dans le dernier scénario, on peut visualiser sa matrice confusion.

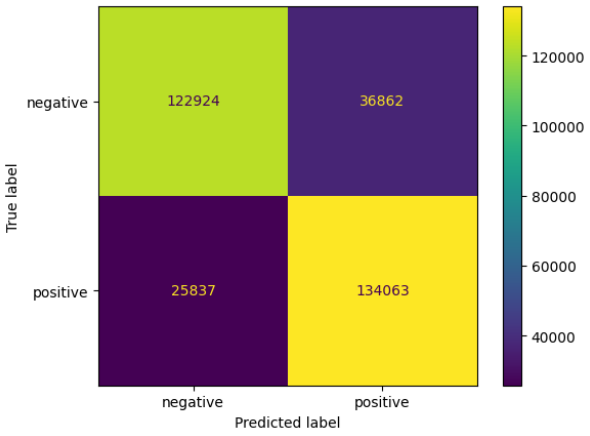

FIGURE 3 – Matrice de confusion

Features	Logistic Regression	Naive Bayes	SVM
Hashing TF-IDF + 1-Gram	73.4	72.5	75.8
Hashing TF-IDF + 1-Gram	77.7	74.9	77.8
CountVectorizer TF-IDF + 1-Gram	76.5	75.8	78.3
CountVectorizer TF-IDF + 1-Gram	79.3	76.8	79.5
CountVectorizer TF-IDF + 1-2-3-Gram + ChisQSelector	80.8	78.7	80.4

TABLE 1 – Précision des modèles dans chaque scénario

D'après cette matrice de confusion, on peut remarquer que ce modèle est plutôt optimiste concernant l'émotion d'un tweet. En effet, il a labelisé 36 862 tweets comme étant positifs alors qu'ils étaient négatifs contre seulement 25 837 tweets positifs prédits comme négatifs de l'autre côté de la matrice.

Étant donné que l'on a vu qu'une bonne précision n'induisait pas forcément que le modèle était le meilleur pour toutes les métriques, vous trouverez dans la Table 2 le récapitulatif de toutes les métriques pour tous les modèles dans le dernier scénario de caractéristiques.

6.2 Passage à l'échelle

6.2.1 Google Cloud

Pour illustrer le passage à l'échelle, nous nous sommes servi de la plateforme Google Cloud. En effet, celle-ci propose un essai gratuit de 90 jours et un crédit de 300 $, afin d'expérimenter avec leurs ressources l'exécution de scripts Python, notebooks Jupyter, ou R. La configuration de Spark est déjà faite, donc on peut mettre en place un environnement très rapidement.

Une fois l'essai gratuit commencé, il faut commencer par créer un espace de stockage en ligne, appelé *Bucket*. Une fois initialisé, il faut y déposer tous les datasets et scripts que nous allons utiliser. Quand nous voudrons utiliser ces ressources, il faudra utiliser un chemin relatif propre au *Bucket*, commençant par "gs ://".

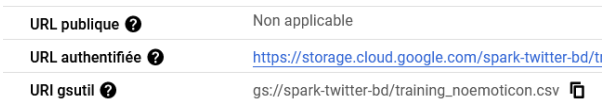

FIGURE 4 – Chemin relatif retrouvable dans les propriétés du fichier désiré

Ensuite, nous pouvons taper dans la barre de recherche "Dataproc" et activer ce module pour créer notre cluster. Il nous sera demandé d'indiquer notre région. Si possible, il faut sélectionner celle la plus proche de nous, car les ressources disponibles varient selon ce choix.

Lors de la création du cluster, nous pouvons choisir son type :
- 1 maître - N esclaves (standard)
- un seul noeud (pour les traitements à faible échelle)
- 3 maîtres - N esclaves (pour avoir une architecture tolérante aux pannes)

La première option nous convient très bien. Nous pouvons laisser la plateforme configurer la nature des noeuds automatiquement, mais il est intéressant d'y jeter un coup d'oeil, car nous pouvons vraiment choisir une configuration "à la carte". On a le choix entre des CPUs de différentes générations, et de différents constructeurs. De plus, il y a également un choix de GPUs NVIDIA, bien que moins fourni. Pour les CPUs, nous avons le choix entre des profils variés : usage général, optimisé pour les calculs, ou optimisé pour l'usage de grande capacité de mémoire. Cela fait varier le nombre de processeurs à disposition, et l'espace mémoire dont on dispose, variant de quelques gigaoctets à des dizaines de téraoctets.

Nous avons opté pour la configuration suivante :
- Noeud maître
  4 processeurs virtuels, 16 Go de RAM
- Noeud workers
  *6 * 4 processeurs virtuels, 15 Go de RAM*

Pour finaliser la configuration, il est important d'activer certaines fonctionnalités si nous voulons utiliser Jupyter Notebook. Cela n'a pas été notre mode de fonctionnement principal, mais il est intéressant de le mentionner.

Une fois les ressources allouées, nous obtenons donc l'ensemble de noeuds que nous avions choisi.

Metrics	Logistic Regression	Naive Bayes	SVM
False Negative	25 837	37 465	37 600
False Positive	36 862	30 725	24 934
True Negative	122 924	129 061	122 186
True Positive	134 063	122 435	134 966
Accuracy	0.808	0.787	0.804
Precision	0.809	0.787	0.806
Recall	0.808	0.787	0.804
F1-Score	0.808	0.787	0.805

TABLE 2 – Tableau récapitulatif

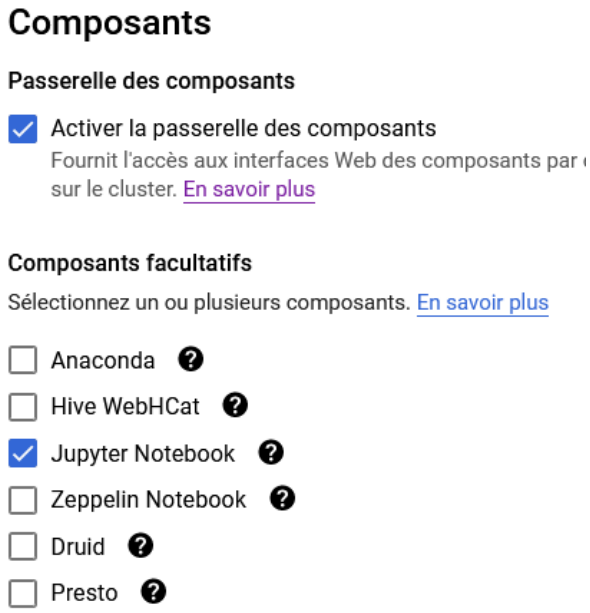

FIGURE 5 – Paramètres activables pour l'utilisation de Jupyter : Activer la passerelle des composants et Jupyter Notebook

Concrètement, cette plateforme nous permettra de tester des jeux de données importants sans pour autant monopoliser nos machines locales. Une fois le cluster en marche, il suffit d'aller dans la section *Interface Webs* et aller sur *Jupyter* ou *JupyterLab* pour commencer à coder nos processus. Dans notre cas, nous avons souvent eu des problèmes de maintien de connexion avec Jupyter, nous nous sommes donc focalisé sur la fonctionnalité *Envoyer un job*, où nous pourrons indiquer un script Python présent dans notre *Bucket* et le faire fonctionner de manière similaire à la commande *spark submit*.

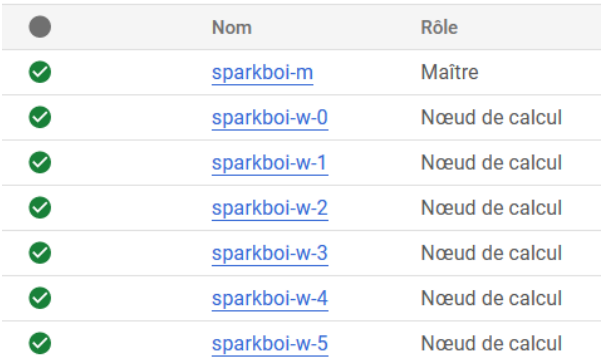

FIGURE 6 – Listes des noeuds de notre cluster

6.2.2 Résultats

Comme lors du Devoir #1, nous avons défini un facteur d'échelle établi à 1 pour 233 Mo pour la taille du jeu de données et de 1 pour 54 secondes pour le temps d'exécution. Nous avons par la suite dupliqué le jeu de données jusqu'à atteindre un facteur d'échelle de 43 soit près de 10,1Go. Vous trouverez ci-dessous le tableau récapitulatif des valeurs ainsi qu'une graphique démontrant la réussite du passage à l'échelle.

Dataset size (Go)	SF	Execution time (s)	SF
0,23	1	54	1,00
1,173	5	144	2,67
2,347	10	277	5,13
3,521	15	445	8,24
4,695	20	567	10,50
5,869	25	725	13,43
7,043	30	839	15,54
8,217	35	1027	19,02

TABLE 3 – Résultats expérimentaux

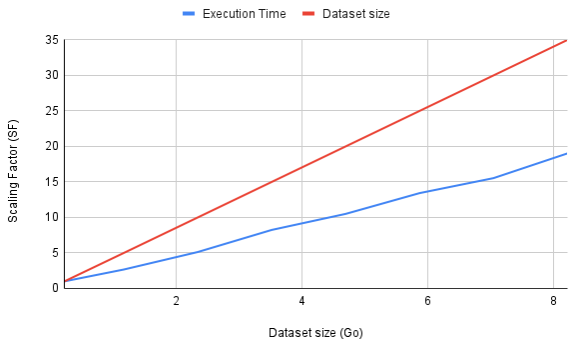

FIGURE 7 – Mise à l'échelle de la classification

7 Spark vs Scikit-Learn

Afin d'appuyer l'utilité des outils comme Spark dans les problèmes de Big Data, il est intéressant de mentionner que comparer à Spark, qui distribue les jeux de données, Scikit-Learn n'a pas été capable de traiter nos jeux de données. Le premier problème a été la taille de ces jeux, sans optimisations, Scikit-Learn (utilisé avec Pandas) ne peut pas charger les jeux de données dans un DataFrame. Nous avons poussé les tests en appliquant des optimisations comme l'inférence des types de données, permettant d'éviter au programme de déduire la nature des données présentes, car en interne, cela implique du pré-traitement qui prend de la mémoire.

Dans un deuxième temps, même si nous avons pu charger le DataFrame, Scikit-Learn ne réussissait pas à entraîner les modèles, pour des raisons similaires. Encore une fois, nous avons poussé la réflexion en transformant les DataFrames en *Sparse Matrix*, ou matrice creuse, permettant un gain en mémoire important, au prix des méta-données qui étaient présentes dans le DataFrame. Malheureusement, malgré tout cela, le dernier défaut que nous avons rencontré nous a convaincu qu'il était impossible d'utiliser Scikit pour les données massives. En surmontant nos deux premiers obstacles, nous arrivons à entraîner nos modèles, cependant, nous avons jamais pu les utiliser, tellement le processus était long.

Cela appuie donc la supériorité de Spark dans le cadre des données massives, qui arrive à entraîner des modèles et à trouver leurs hyperparamètres en quelques heures seulement.

8 Spark Streaming

Un autre pan du projet a été d'expérimenter l'extension Spark Streaming. En effet, c'est une ressource très largement utilisée dans le monde professionnel, car elle permet aux *Data Engineer* et *Data Scientist* de traiter des données en temps réel provenant de diverses sources, notamment de Kafka.

Pour aller plus loin, nous avons implémenté un processus *ETL* (Extract-Transform-Load) complet afin de s'inscrire pleinement dans le paradigme du Big Data. De l'extraction via la connexion à l'API Twitter, la transformation via le traitement et l'utilisation d'un modèle pré-entraîné, mais aussi le chargement via MongoDB ou Delta Lake.

8.1 Apache Kafka

Apache Kafka est un magasin de données distribuées optimisé pour l'ingestion et le traitement de données en continu en temps réel. Les données en continu sont générées en permanence par des milliers de sources de données, qui envoient généralement les enregistrements de données simultanément. Une plateforme de streaming doit gérer cet afflux constant de données et les traiter de manière séquentielle et incrémentale [8].

Vous trouverez ci-dessous la structure globale d'un cluster Kafka.

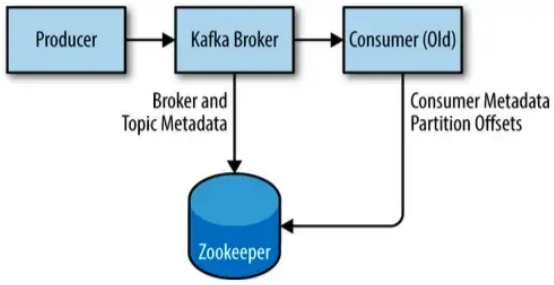

FIGURE 8 – Kafka Cluster [9]

Ce n'est pas ici le sujet principal, mais très rapidement voici les fonctions de chaque composant du cluster.

Le *Producer* est la partie du cluster qui publie ou écrit des données dans les *topics* des différentes partitions. Les *producers* savent automatiquement quelles données doivent être écrites dans quelle partition et quel *broker*. L'utilisateur n'a pas besoin de spécifier le *broker* et la partition.

Un *broker* Kafka reçoit des messages de la part des *producer* et les stocke sur le disque en fonction d'un décalage unique. Il permet aux *consummers* de récupérer

les messages par *topic*, partition et décalage.

Un *consumer* Kafka agit comme un utilisateur final ou une application qui récupère les données des serveurs Kafka dans lesquels les *producers* Kafka publient des messages en temps réel. Pour récupérer efficacement les messages en temps réel, les *consumers* Kafka doivent s'abonner aux *topics* respectifs présents dans les serveurs Kafka.

Enfin, *ZooKeeper*, lui, en opposition à Kafka, ne gère pas les données, mais gère les différents brokers disponibles. Il réagit quand il y a une panne et garde en mémoire une liste des *topics* disponibles dans le cluster.

8.2 Implémentation

Vous trouverez ci-dessous en Figure 9, la structure complète du processus ETL.

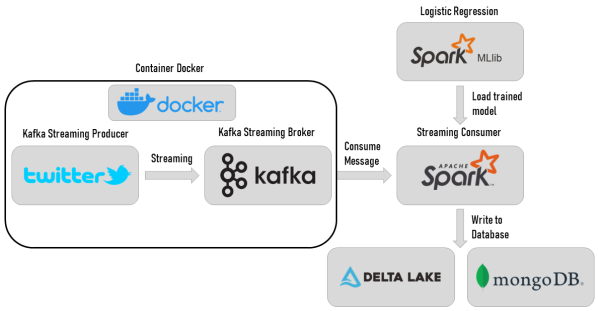

FIGURE 9 – Pipeline

Afin de faciliter l'exécution et le déploiement de l'application, Apache Kafka se lance en local dans un container Docker. Nous avons fait ce choix, car cela permet d'éviter les problèmes de versions entre les systèmes d'exploitation, les logiciels, etc.

Une fois l'instance de Kafka lancée, nous pouvons créer un topic nommé "twitter". En parallèle, nous avons écrit un script python nommé producer.py qui, de son côté, va se connecter à l'API Twitter, écouter les tweets et les envoyer dans notre topic précédemment créé. Enfin, il nous faut maintenant un *consumer* qui va s'abonner au topic twitter et récupérer le contenu via Spark Streaming. C'est lui qui va charger le modèle pré-entraîné pour faire des prédictions sur les données en sortie du topic. Une fois les prédictions faites sur le batch de données actuelles, nous pouvons les charger dans l'environnement de stockage de notre choix. Dans notre cas, MongoDB ou Delta Lake.

C'est intéressant de voir que toutes les fonctions habituelles utilisées pour afficher une partie d'un DataFrame, etc, ne fonctionnent plus dans le cadre du Streaming. Tout peut se faire de façon asynchrone. On peut spécifier si l'on veut qu'une tâche s'effectue directement ou bien après une autre via le paramètre *"awaitTermination"*. On n'écrit plus la totalité des données dans la base de données, mais nous opérons par *batch* via la fonction *foreachBatch*.

```
Batch: 2
-------------------------------------------
+--------------------+----------+
|        cleaned_data|prediction|
+--------------------+----------+
|[litmormon, zeroh...|       1.0|
|[work, careful, t...|       1.0|
|[closer, with, pa...|       0.0|
|[ctg, is, hiring,...|       1.0|
|[is, there, a, wa...|       0.0|
|[improve, push, s...|       0.0|
|[radio, bit, note...|       0.0|
|[return, police, ...|       0.0|
|[there, are, many...|       1.0|
```

FIGURE 10 – Prédictions effectuées sur un batch

Du fait des contraintes du temps réel, nous devons traiter les données et effectuer les prédictions le plus rapidement possible. Sinon, les tweets vont s'accumuler et nous allons prendre du retard. C'est pour cela que nous avons choisi de garder l'avant-dernier scénario des caractéristiques, car la création des 1-2-3 Gram est un processus trop long.

9 Livrables

Vous trouverez avec ce rapport, le support de présentation ainsi que différents sous-dossiers avec nos Jupyter Notebook, les fichiers Python pour faire des prédictions en temps réel avec le modèle pré-entraîné.

Vous trouverez dans le dossier *Notebooks*, l'ensemble des modèles utilisé lors de ce travail avec les résultats en mémoire.

Si vous souhaitez tester le modèle en temps réel de votre côté, nous avons laissé dans le dossier *ETL_Spark_Streaming* un fichier d'instructions nommé *readme.md*. Il contient les instructions exhaustives pour lancer localement un container Docker avec Apache Kafka, créer un topic Twitter, se connecter à l'API Twitter et faire des prédictions en tempsréel. Cependant, cela requiert d'installer beaucoup d'éléments.

10 Conclusion

Ce devoir a été extrêmement enrichissant. En plus de répondre à la problématique de base à travers la résolution d'un problème NLP, nous avons pu explorer des pistes supplémentaires comme la prédiction en temps réel et l'utilisation de machines fournies à travers le Cloud. Cela a été très formateur de se confronter à ces différentes méthodes.

Nous avons vu que l'analyse de sentiment réussit avec succès le passage à l'échelle, tout en gardant de très bonnes performances, et que plusieurs modèles peuvent s'appliquer : le kNN, comme montré dans l'article, ou bien la régression logistique ou le *Naïve Bayes*.

Pour conclure, ce projet, qui aurait pu s'arrêter à la simple comparaison des modèles, a été complexifié de différentes manières. Il nous a permis de gagner en confiance avec l'utilisation de l'outil PySpark, d'exécuter du code dans le Cloud, mais aussi d'implémenter un processus ETL complet tel qu'il est implémenté en entreprise. Cela nous a permis de nous rapprocher d'une problématique réelle et qui pourra très certainement nous aider dans notre recherche de stage de fin d'études. Merci pour ce sujet !

Références

[1] Nodarakis N. et al. « Large scale sentiment analysis on twitter with spark ». In : *CEUR Workshop Proceedings* 1558 (2016).

[2] Alec Go, Richa Bhayani et Lei Huang. « Twitter sentiment classification using distant supervision ». In : *CS224N project report, Stanford* 1.12 (2009), p. 2009.

[3] URL : https://www.tweepy.org/.

[4] Apache. *Spark/gradientdescent.scala at master · Apache/Spark*. URL : https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/optimization/GradientDescent.scala#L178.

[5] *Scaling Apache Spark MLlib to Billions of Parameters : Spark Summit East talk by Yanbo Liang*. YouTube, fév. 2017. URL : https://www.youtube.com/watch?v=5m9Rcu3YHps.

[6] Apache. *Spark/svm.scala at master · Apache/Spark*. Oct. 2021. URL : https://github.com/apache/spark/blob/master/mllib/src/main/scala/org/apache/spark/mllib/classification/SVM.scala#L134.

[7] *Spark's MLlib SVM classification to include kernels like Gaussian / (RBF) to find non linear boundaries*. Nov. 2014. URL : https://issues.apache.org/jira/browse/SPARK-4638.

[8] *What is Apache Kafka ?* 2022. URL : https://aws.amazon.com/fr/msk/what-is-kafka/.

[9] Neha Narkhede, Gwen Shapira et Todd Palino. *Kafka : The Definitive Guide Real-Time Data and Stream Processing at Scale*. 1st. O'Reilly Media, Inc., 2017. ISBN : 1491936169.