UASB03 – Certificat de spécialisation Analyste de données massives : Projet final

Daniel PONT
février 2020

Remerciements

Je tiens ici à exprimer ma vive gratitude à l'équipe pédagogique pour la qualité de son enseignement et sa disponibilité sur l'ensemble de la formation. Au moment de rédiger ce rapport, j'ai notamment une pensée particulière pour :

- Mme Ndèye Niang-Keita, M. Vincent Audigier et M.Nicolas Thome (STA211) ;
- MM. Philippe Rigaux, Nicolas Travers et Raphaël Fournier S'niehotta (NFE204) ;
- MM. Michel Crucianu, Pierre Cubaud et Marin Ferecatu (RCP216).

Un grand merci également à toutes les personnes du CNAM qui ont permis de rendre le certificat de spécialisation « Analyste de données massives » accessible en enseignement à distance.

Je souhaite également témoigner ma reconnaissance à pôle emploi (particulièrement à Mmes Chantal CHALVIDAN et Marie-Noëlle BADOL) pour le financement de ma formation.

"The world is full of obvious things which nobody by any chance ever observes."

– Sherlock Holmes

"We show that easily accessible digital records of behavior, Facebook Likes, can be used to automatically and accurately predict a range of highly sensitive personal attributes including: sexual orientation, ethnicity, religious and political views, personality traits, intelligence, happiness, use of addictive substances, parental separation, age, and gender."

– Michal Kosinski, David Stillwell, and Thore Graepel

Sommaire

1. Introduction

Ce document est un rapport rédigé dans le cadre du projet final UASB03, concluant le certificat de spécialisation « Analyste de données massives » organisé par le CNAM Paris.

Le sujet d'étude est la **détection du risque qu'un utilisateur Twitter souffre de dépression en analysant les messages qu'il a postés sur le réseau**.

Dans ce premier chapitre, nous présentons les motivations du projet puis définissons les données et les traitements. Nous abordons ensuite la question de la scalabilité et précisons les objectifs de l'étude.

1.1 Motivations

L'idée du projet trouve sa source dans les travaux du Dr Michal Kosinsky et plus précisément dans son papier "Private traits and attributes are predictable from digital records of human behavior (2013)" rédigé avec les Drs David Stillwell et Thore Graepel. Les résultats auxquels sont parvenus ces chercheurs ont trouvé un large écho dans la presse internationale. Ainsi, le Guardian publia le 11 mars 2013, un article intitulé "Facebook users unwittingly revealing intimate secrets, study finds" dans lequel on peut lire la conclusion suivante :

"Researchers were able to accurately infer a Facebook user's race, IQ, sexuality, substance use, personality or political views using only a record of the subjects and items they had 'liked' on Facebook – even if users had chosen not to reveal that information."

Cette information a été par la suite reprise dans de nombreux médias tel le magazine Time qui dans son article "Here's Proof That Facebook Knows You Better Than Your Friends", paru le 12 janvier 2015, résume ainsi la méthodologie employée par les chercheurs et les résultats obtenus :

"They [researchers at the University of Cambridge and Stanford University] asked 86,220 people on Facebook to complete a 100-question personality survey that determined where they stood on the so-called Big Five traits: openness, conscientiousness, extraversion, agreeableness and neuroticism. They then analyzed their Facebook Likes to generate a model in which Likes were linked to the traits.[...] On average, people on Facebook had 227 Likes, and this was enough information for the computer to be a better predictor of personality than an average human judge (in other words, a friend), and almost as good as a spouse. The more Likes, the better the computer got. It only took 10 Likes for the computer to outperform a work colleague, for instance, 70 to do better than a friend, and 150 to outscore a family member."

Au-delà des nombreuses questions éthiques soulevées par ces recherches (la tristement célèbre officine cambridge analytica est censée les avoir utilisées pour influencer les votes lors de l'élection présidentielle américaine de 2016), elles mettent en lumière un phénomène fascinant : la possibilité de prédire précisément, à l'aide d'algorithmes d'apprentissage automatique, non seulement les traits de caractères, mais également les opinions politiques ou l'orientation sexuelle d'un utilisateur Facebook sans que celui-ci les ait déclarés explicitement.

Ces résultats nous ont amenés aux interrogations suivantes :

- Si il est possible de prédire, à l'aide d'algorithmes d'apprentissage automatique, avec une grande précision, des traits de personnalité intimes d'un utilisateur de réseau social, en se basant sur uniquement sur son activité en ligne, peut-on également prédire son état de santé psychologique ?
- Comment identifier rapidement et efficacement, sur un réseau social, une quantité suffisante d'utilisateurs souffrant d'une pathologie particulière, pour créer un modèle supervisé (ou réaliser une étude médicale plus classique)?

Nous avons choisi de traiter ces deux interrogations au travers d'un problème concret :

1. Récupérer sur twitter les messages d'utilisateurs déclarant souffrir de dépression et ceux d'utilisateur n'en souffrant (a priori) pas ;
2. Construire en partant des données textuelles pré traitées, un modèle de classification binaire supervisé puis évaluer sa précision.

Dans les deux sous-paragraphes qui suivent, nous répondons à deux questions : pourquoi twitter ? Pourquoi la dépression?

1.1.1 Pourquoi Twitter et pas Facebook ou un autre réseau social ?

Nous avons écarté immédiatement Facebook : comme l'explique le Dr Kosinsky sur son site, la publication de son papier en 2013 a conduit Facebook, dans les deux semaines qui ont suivi, à interdire l'accès publique aux likes de ces utilisateurs.

Twitter avec ses 145 millions d'utilisateurs quotidiens s'est révélé un choix naturel :

- Il s'agit d'un des réseaux sociaux les plus populaires avec un nombre d'utilisateurs élevé présentant des profils très variés ;
- Le format des "tweets" - messages limités à 140 caractères - se prêtent bien à une représentation Word2Vec telle qu'abordée en TP RCP216 . Cette approche permet de travailler avec des vecteurs numériques compactes (typiquement ayant une dimension de l'ordre de la centaine) qui peuvent facilement être traités ;
- Les API d'accès à Twitter même avec un compte gratuit comportant des limitations (notamment en terme de fréquence d'interrogation) offre des fonctionnalités riches et permettent d'obtenir facilement et rapidement un volume de données important. A titre d'exemple nous avons pu télécharger l'ensemble des tweets postés par plus de 5000 utilisateurs en une dizaine d'heures.

1.1.2 Pourquoi la dépression ?

Selon l'OMS, la dépression est la première cause de morbidité et d'incapacité dans le monde. En 2015, plus de 300 millions de personnes souffraient de cette maladie. Son étude, sa prise en charge et le développement de traitements efficaces sont donc des enjeux majeurs de santé publique. D'un point de vue purement pratique, le nombre (malheureusement) élevé de personnes touchées facilite le recueil d'échantillons de population par rapport à une pathologie plus rare.

1.2 Données

Il convient d'abord de préciser la définition d'« utilisateur Twitter souffrant de dépression ». Il s'agit dans notre étude d'une personne :

- ayant **déclarée explicitement** souffrir de dépression au sens clinique du terme (dépression simple, dépression bipolaire, dysthymie...) **dans la description de son profil twitter** ;
- abonné à au moins un compte traitant de dépression et/ou de la santé mentale.

A contrario, nous considérerons qu'un utilisateur twitter ne souffre pas de dépression si il s'agit d'une personne qui dans la description de son profil :

- ne déclare pas souffrir de troubles psychologiques ;
- s**e définit** comme étant **heureux et/ou optimiste**

Nous reviendrons sur ces définitions de façon plus approfondie dans la suite du document, mais elles soulèvent d'emblée une question cruciale : quelle est **la validité et la véracité des données** ? Plus précisément peut-on procéder à une analyse de données pertinente avec un étiquetage basé sur des diagnostiques « auto-déclarés » publiés sur des comptes Twitter ?

Des éléments de réponse intéressants à cette question résident dans le papier Detecting depression and mental illness on social media: an integrative review publié en 2017 par Sharath Chandra Guntuku, David B Yaden et al.

Cet article expose les résultats d'une méta-étude portant sur la prédiction des maladies psychologiques via les réseaux-sociaux. On peut notamment y lire :

"A number of studies use publicly accessible data. 'Self-declared' mental illness diagnosis on Twitter (identified through statements such as 'I was diagnosed with depression today') is one such source of publicly-available data. We review seven studies of this kind. Helping to facilitate studies of this kind, a Computational Linguistics and Clinical Psychology (CLPsych) workshop was started in 2014 to foster cooperation between clinical psychologists and computer scientists. 'Shared tasks' were designed to explore and compare different solutions to the same prediction problem on the same data set"

Concernant les méthodes prédictives utilisées, l'article explique :

"Automated analysis of social media is accomplished by building predictive models, which use 'features,' or variables that have been extracted from social media data. For example, commonly used features include users' language encoded as frequencies of each word, time of posts, and other variables [...]. Features are then treated as independent variables in an algorithm (e.g. Linear Regression [4] with built in variable selection [5], or Support Vector Machines (SVM)) [6] to predict the dependent variable of an outcome of interest (e.g. users' mental health)."

Finalement les auteurs de l'étude évalue la précision du diagnostique obtenu avec ce type de méthodes de la manière suivante:

"social media-based screening may reach prediction performance somewhere between unaided clinician assessment and screening surveys; however, no study to date has assessed social-media-based prediction against structured clinical interviews".

Au vu de cet article, la constitution d'un jeu de données, avec un étiquetage basé sur des diagnostiques « auto-déclarés » publiés sur des comptes twitter, visant à étudier une maladie

comme la dépression est donc possible : c'est un sujet de recherche ouvert. De plus les travaux publiés qui ont déjà été menés dans ce sens, utilisent pour analyser les données des modèles (ex : SVM) étudiés en STA211/RCP216. La création, la validation puis l'utilisation d'un de ces modèles sont abordées dans le sous-chapitre suivant.

Une caractéristique fondamentale du projet est que les messages Twitter ou « tweets » avec lesquels nous travaillons ne sont pas issus d'un jeu de donnés pré-constitué : en terme de données nous partons de zéro. Autrement dit, l'acquisition et l'étiquetage des données font partie intégrante du travail réalisé . Dans le chapitre 2, nous détaillerons

- La procédure permettant de télécharger les tweets d'utilisateurs susceptibles d'appartenir au groupe traité (« déprimé ») et au groupe de contrôle (« non déprimé ») ;
- La procédure d'étiquetage en elle-même, son intérêt, ses limitations ainsi que des pistes pour l'améliorer.

1.3 Traitement

Le traitement réalisé dans le cadre de notre étude est une **classification binaire** portant sur **des données textuelles**. Un individu (au sens statistique du terme) correspond à un utilisateur Twitter ayant posté au moins 50 messages sur le réseau social. Les données d'entrée sont l'ensemble des textes de ses messages, les éventuels contenus multimédias associés (photos, vidéos, liens hypertextes,…) étant ignorés. Le but est de déterminer à quelle classe :« déprimé »/ « non déprimé », l'individu appartient.

La mise en œuvre de cette classification, détaillée dans la suite du document, comporte:

1. une phase d'analyse exploratoire des données (cf. chapitre 3);
2. un pré-traitement (cf. chapitre 4);
3. la création d'un modèle avec un algorithme d'apprentissage automatique supervisé (régression logistique) basé sur un échantillon de 1000 individus « déprimés » et 1000 individus « non déprimés » (cf. chapitre 5.1);
4. la validation de ce modèle sur un jeu de données de test avec 600 individus « déprimés » et 600 « non déprimés » distincts de ceux utilisés dans la phase d'apprentissage (cf. chapitre 5.2);
5. l'application du modèle sur un flux twitter (cf. chapitre 6).

Un élément important à souligner concerne la notion de « détection du risque » dans l'intitulé du sujet. il ne s'agit pas ici d'analyser les tweets d'un utilisateur dans un intervalle de temps [t1,t2] pour prédire le fait que cet utilisateur, à une date t>t2 souffre de dépression. Il s'agit en fait simplement de considérer l'ensemble des tweets d'un utilisateur pour déterminer si il souffre ou a souffert de dépression.

Une discussion plus détaillée sur la question de la **dimension temporelle des tweets** est présentée dans le chapitre 2.1.3.

1.4 Scalabilité

Lors du développement du projet, nous n'avons disposé que d'un simple PC portable d'entrée de gamme, ce qui a exclu une expérimentation en vraie grandeur sur des données massives stockées dans un système distribué. Néanmoins :

- Le code Spark / Scala présenté dans la suite du document pour créer et valider un modèle supervisé sur un échantillon limité (de l'ordre du millier d'individus) peut être employé sur des données beaucoup plus volumineuses (dizaines de milliers d'individus). La seule différence majeure en terme de code, dans ce dernier cas, consiste à lire les fichiers de données stockés non pas sur un répertoire local mais sur un système HDFS.
- La question de la scalabilité se pose non seulement lors de la phase de création du modèle (avec un dataset dont la dimension est potentiellement élevée) mais dès la phase d'acquisition et d'étiquetage des profils Twitter. Afin de constituer un dataset exploitable, nous devons traiter un volume de données important. A titre indicatif, l'ensemble des tweets de 5000 utilisateurs abonnés au compte DBSAlliance téléchargés selon la méthode détaillée dans l'annexe 9.1.6 , incluant tous les détails fournis par l'API Twitter représente plus de 30 Go. Ces données brutes sont réduites pour ne garder que les informations pertinentes, insérées dans un moteur de recherche Elasticsearch puis filtrées à l'aide de requêtes booléennes.

Nous reviendrons sur ces points dans le chapitre 7 (passage à l'échelle) où nous présenterons entre autres une architecture inspirée de MapReduce.

1.5 Objectifs

Le projet UASB03 décrit dans ce document offre l'opportunité de mettre en œuvre les compétences méthodologiques et technologiques acquises respectivement en :

- NFE204 (recherche d'information textuelle avec Elasticsearch, MapReduce) ;
- STA211 (analyse exploratoire, méthodes de classification supervisées) ;
- RCP216 (Spark Datafames, SparkMLlib, Spark Streaming, Spark NLP, Word2Vec).

L'objectif de ce projet est d'appliquer ces compétences afin d'évaluer dans quelle mesure :

- Il est possible d'identifier rapidement et à moindre coût sur Twitter un nombre important de personnes souffrant d'une maladie (ici la dépression). Ceci peut s'avérer utile dans le cadre d'une étude épidémiologique ou pour rechercher des volontaires acceptant de tester un traitement thérapeutique ;
- Les méthodes d'apprentissage supervisés utilisées avec succès par Kosinsky et al. pour définir la personnalité d'un utilisateur Facebook à partir de des « Likes » constituent une voie d'expérimentation prometteuse pour détecter si un utilisateur Twitter présente un risque de souffrir de dépression.

2. Acquisition et étiquetage des données

L'acquisition et l'étiquetage des données se compose de quatre parties :

1. Création d'une liste d'utilisateurs souffrant de dépression ;
2. Création d'une liste d'utilisateurs ne souffrant pas de dépression ;
3. Téléchargement des tweets des utilisateurs appartenant aux deux listes précédentes ;
4. Création d'un jeu de données d'apprentissage et d'un jeu de données de test.

2.1 Utilisateurs Twitter souffrant de dépression

2.1.1 Acquisition de la liste des utilisateurs souffrant de dépression

Une des spécificités du projet est de partir de données brutes, non étiquetées. L'un des défis est donc de pouvoir étiqueter ces données de manière fiable. Le fait de considérer qu'un utilisateur Twitter souffre de dépression sous prétexte que les mots-clés "depression" ou "depressed" apparaissent dans les messages qu'il a postés n'est clairement pas satisfaisant. D'une part, l'utilisateur ne parle pas forcément de lui mais il peut parler de quelqu'un d'autre. Ensuite, il faut tenir compte du contexte du message, du fait que son auteur peut employer l'ironie, le second degré ou faire preuve d'exagération :

```
I won't be able to go to Justin Bieber's concert next week. Soooooo depressed right now :'(
```

D'autre part, l'idée d'effectuer des recherches directes avec l'API Twitter, non pas avec de simples mots-clés isolés mais avec des expressions beaucoup plus spécifiques pour éliminer toute ambiguïté - ex. : "I was diagnosed with depresssion today" se heurte à un problème pratique très simple : le nombre de résultats retournés par l'API (dans sa version gratuite) est limité à 100 ce qui est nettement insuffisant.

Pour résoudre ces problème, nous nous sommes posés deux questions :

1. Comment identifier rapidement un nombre d'utilisateurs élevé susceptible de souffrir de dépression ?
2. En supposant qu'on dispose de l'ensemble des tweets d'un lot d'utilisateurs (disons quelques milliers) constituant des candidats potentiels, comment distinguer de façon efficace et précise ceux qui souffrent vraiment de dépression ?

La réponse à la première question est relativement simple : il suffit de récupérer la liste des abonnés d'un compte twitter traitant du sujet ; par exemple celui-ci, qui comporte près de 30 000 followers :

Une fois récupérée les tweets des abonnés, on peut les importer dans une instance Elasticsearch. On dispose alors d'outils de recherche textuelle sophistiqués qui sont étudiés dans l'UE NFE204.

Nous nous sommes initialement limités à 5000 utilisateurs ayant posté chacun au moins 50 messages, et avons importé dans Elasticsearch les informations principales :

- user_id ;
- user_name ;
- user_description;
- user_location ;
- text;
- created_at ;

Puis nous avons effectué une requête très basique pour récupérer les tweets contenant le mot-clé "depression" qui a retournée 10000 résultats.

Afin de faciliter l'analyse des résultats, nous les avons exporté dans une feuille Excel.

Initialement nous pensions nous focaliser sur la colonne "text" contenant le message des tweets, mais en fait la colonne la plus intéressante dans une optique d’étiquetage s'est révélée

être la colonne "user_description', car elle permet d'un simple coup d'œil d'identifier trois grandes catégories d'utilisateurs :

- Des utilisateurs souffrant de dépression et qui le mentionnent explicitement dans leur profil :

```
#mentalhealth Blogger| #Depression and #Anxiety Sufferer| Making a community devoted to
helping each other with mental illness
```

- Des utilisateurs dont un des proches souffre ou a souffert de dépression :

```
 A Filipino living with someone who has depression. Provides inspiration and insights to
those who are companions of people w/ mental health issues. #MHAdvocate
```

- Des thérapeutes, ou des établissements/associations traitant la dépression :

```
The SA Federation for Mental Health. Our mission is: To work with the community to achieve
the highest possible level of mental health
```

Au vu des résultats préliminaires ci-dessus, nous avons retenu la procédure suivante :

1. téléchargement des profils (id, nom, description, emplacement géographique) d'utilisateurs abonnés à un compte traitant de la dépression (ex : "dbsalliance"). **A ce stade, nous récupérons uniquement des profils, pas les messages postés**, ce qui diminue fortement le volume de données à télécharger (cf chapitre 7 : passage à l'échelle);
2. import des profils par lot dans un index d'une instance Elasticsearch locale avec l'API bulk ;
3. exécution d'une requête Elasticsearch booléenne sur le champ description du profil utilisateur :

   contient ("depression" ou "depressed ou "bipolar" ou "dysthymia" ou "survivor') et

   ne contient pas ("therapist" ou "psychiatrist" ou "federation" ou "program")

   le but de cette requête est de filtrer au maximum les profils pour ne garder que les utilisateurs malades en éliminant ceux qui sont concernés par le sujet pour d'autres raisons. Les profils retournés par la requête sont exportés dans une feuille Excel ;
4. validation manuelle de la feuille Excel (une colonne « exclude », permet en la cochant d'exclure les profils non pertinents) ;
5. extraction à partir de la feuille Excel de la liste des utilisateurs validés (ou plutôt de leurs identifiants) au format json.

Le code implémentant cette procédure est présentée dans l'annexe 9.1. Nous avons choisi de l'écrire en python en raison de la richesse des bibliothèques disponibles (API Twitter , Elasticsearch , lecture/écriture de fichiers Excel ,...) et de leur facilité d'utilisation.

NB : le traitement réel est un peu plus compliqué que le principe exposé ci-dessus : pour obtenir suffisamment de données, les scripts python que nous avons développés :

- récupèrent les abonnés non pas depuis un seul compte traitant de la dépression mais depuis plusieurs : "depressionarmy", "depressionnote", "dbsalliance", "NIMHgov" ;
- créent non pas un seul index d'Elasticsearch mais autant d'indexs que de comptes dont on liste les abonnés ;
- génèrent autant de feuilles Excel que d'index Elasticsearch ;
- éliminent les doublons (un même utilisateur peut être abonné à plusieurs comptes - ex : "dbsalliance" et "NIMHgov") .

2.1.2 Efficacité et validité de l'étiquetage

Afin d'avoir une meilleure idée de l'**efficacité de l' étiquetage**, examinons un extrait significatif d'une feuille Excel générée avec la procédure décrite au paragraphe précédent :

	A	B	C	D	E	F
1	id	name	screen_name	location	description	exclude
2	[redacted]			Etats-Unis	Christian, Bipolar 1, depression, mania, suicide survivor, recovered from eating disorder, wife, mother, friend	
3	983437478160936960	Karla Smith Behavioral Health	KSBHRecovery	O'Fallon, IL	Karla Smith Behavioral Health provides treatment and support to individuals and families affected by mental illness, substance use and suicide grief.	x
4	943583432013205504	CounselorsAssociates	CounselorsAssoc	Maryville, IL and Shiloh, IL	Counseling & Therapy Practice. Specialties: Adults, Children, Couples, Law Enforcement Officers. Areas of Concern: Depression, Anxiety, Grief and Trauma	x
5	[redacted]			Sacramento, USA	I live with, depression/ panic/anxiety/PTSD/ agoraphobia. Last year I completed a Medical Asst. Program with a 4.0.Keep fighting, your worth it!	
6				Montrose, CO	I'm hooked on good role play pods & fiction! I enjoy movies, music & pets. Hoping to have fun & meet people. Fighting bipolar disorder alone. #SickNotWeak	
7				Pottstown, PA	Grateful for my #recovery from #alcoholism & #BPD. @EaglevilleHosp showed me what was possible. Treating my #depression, #anxiety and #ptsd. #KeepTalkingMH	
8	963992693591261185	TadafumiKato	TadafumiKato		Lab Head in RIKEN studying neurobiology of bipolar disorder. Editor-in-Chief of Psychiatry and Clinical Neurosciences (PCN), an official journal of JSPN.	x
9	3237691431	The Quinn Project	project_quinn	Logan, UT	We raise awareness of the issues of bullying and cyberbullying and its impact on today's teens. We also want to prevent teen suicide.	x
10	[redacted]			Republic of the Philippines	A Filipino living with someone who has depression. Provides inspiration and insights to those who are companions of people w/ mental health issues. #MHAdvocate	x

Une remarque s'impose au vu de cet extrait : le filtrage des profils à l'aide de requêtes Elasticsearch n'est pas entièrement suffisant. Il est difficile, même en affinant les critères, de distinguer de façon totalement automatique :

- un compte twitter associé à une personne physique, de celui d'une organisation *(ligne 2 : « Christian, Bipolar 1 ...»*vs *ligne 3 « Karla Smith Behvorial Health...») ;*
- un utilisateur souffrant de dépression, d'un utilisateur concerné indirectement *(ligne 5* vs *ligne 10)* ;
- un malade d'un thérapeute / chercheur *(ligne 7* vs *ligne 8)*

C'est la raison pour laquelle un examinateur humain doit finaliser l'étiquetage. C'est l'approche que nous avons adoptée en cochant manuellement les profils à écarter (« x » dans la dernière colonne de la copie d'écran ci-dessus).

En comptant 10s par profil, il faut moins de 3 heures pour examiner 1000 individus. En considérant qu'entre 25 et 50 % des individus présélectionnés par les requêtes Elasticsearch seront retenus, l'étiquetage d'un échantillon de taille comparable à celui utilisé pour les données d'apprentissage prend entre 6 et 12 heures.

Concernant **la validité des données étiquetées,** une question intéressante est de savoir dans quelle mesure une personne déclarant souffrir de dépression a effectivement été diagnostiquée par un professionnel de santé.

Une première solution est d'exécuter des requêtes Elasticsearch très spécifiques portant non seulement sur la description des utilisateurs mais également sur le contenu de leurs messages. Cela suppose de disposer d'un large volume de données et nécessite un passage à l'échelle (c'est l'approche abordée au chapitre 7.1)

Une deuxième solution consiste à contacter directement les utilisateurs (pré)étiquetés et à leur demander de répondre à un questionnaire en ligne. Cela suppose :

1. de mettre en place un site web avec le questionnaire et une base de données pour stocker les résultats. Si le nombre d'utilisateurs susceptibles de répondre au sondage n'est pas trop important (disons 10 000), une simple base MySQL et quelques pages PHP suffisent ;
2. d'envoyer automatiquement, avec un script, des invitations à la liste d'utilisateurs Twiter (pré)étiquetés, par exemple avec la fonction PostDirectMessage de l'API Twitter .

Cette deuxième solution nécessite un développement sortant du périmètre de notre étude.

2.1.3 Composante temporelle des tweets

Dans le cadre de ce projet, plus précisément lors de la construction du modèle de classification nous prenons en compte l'ensemble des tweets d'un utilisateur pour déterminer si il souffre ou a souffert de dépression.

Une question posée par cette approche est la suivante : considérons un utilisateur pour lequel on dispose de l'historique sur une période relativement longue - disons 5 ans ou plus. Durant cette période il a connu une épreuve personnelle - ex. : un accident, un deuil - qui l'a amené à souffrir d'une dépression réactionnelle pendant 6 mois. Peut-on arriver à détecter cet épisode, sachant que les tweets qui y sont relatifs ne constitue probablement qu'une minorité des messages de l'historique, qu'ils sont en quelque sorte noyés dans la masse des tweets ?

Notre réponse se base sur les critères d'étiquetage (cf. chapitre 2.1.1). Si un utilisateur mentionne explicitement dans son profil qu'il souffre de dépression et que par ailleurs, il est abonné à au moins un compte traitant de la question, on peut légitimement penser que c'est un problème important qui l'a marqué ou qui est récurrent dans sa vie. Par conséquent l'hypothèse qu'on puisse en détecter les traces dans ses tweets semble vraisemblable.

Une dernière réflexion sur la composante temporelle : il pourrait s'avérer intéressant d'analyser l'évolution de la fréquence des tweets dans le temps. Par exemple, dans le cas d'un utilisateur souffrant de troubles bipolaires (i.e. alternant des phases d'hyper activité et de dépression), il semble naturel d'imaginer qu'on observe des périodes pendant lequel cette personne "twitte beaucoup" et des périodes pendant lesquelles elle "twitte peu" ou pas. Nous n'avons pas pu, faute de temps, intégrer ce type de données dans le modèle de classification.

2.2 Utilisateurs Twitter ne souffrant pas de dépression

Afin de récupérer une liste d'utilisateurs Twitter ne souffrant (a priori) pas de dépression, nous avons employé la procédure suivante :

1. Connexion à un flux Twitter (avec l'API python tweepy) et récupération des profils utilisateurs (id, nom, description, emplacement géographique) dont la description contient les mots clés « optimist » ou « happy ». Ces profils sont stockés dans un fichier Excel .
2. Validation « manuelle » du fichier Excel selon les critères suivant :
    - l'utilisateur doit se présenter comme étant une personne physique (et non une organisation ,une entreprise, etc. …) ;
    - l'utilisateur ne doit pas avoir créé son compte uniquement pour partager des informations sur un sujet spécifique ou promouvoir une cause (ex : compte d'un fan consacré à son idole, compte dédié à un sport / loisir, compte utilisé ouvertement des fins de militantisme politique, etc. ...) ;
    - la description de l'utilisateur doit spécifier explicitement que la personne est heureuse et/ou optimiste (ex : « optimist student », « happy husband », etc.) ;
    - la description de l'utilisateur ne doit pas mentionner de problèmes psychologiques (« anxiety », « depression », …) ;
    - en cas de doute sur un des quatres points précédents, le profil est exclu .
3. Extraction à partir de la feuille Excel de la liste des utilisateurs validés (ou plutôt de leurs identifiants) au format json .

Le code de l'étape 1 est présenté dans l'annexe 9.1.5, celui de l'étape 3 dans l'annexe 9.1.4

Nous avions envisagé initialement :

- d'importer les profils téléchargés à l'étape 1) dans un index Elasticsearch ;
- de filtrer ces profils avec des requêtes booléennes dans le but de limiter le nombre de profils à valider à l'étape 2)

En pratique cela s'est révélé difficile, les critères à spécifier étant trop généraux pour être formalisés de façon efficace. Au vu des résultats peu concluants, nous avons écarté cette option. La validation des profils est donc réalisée « manuellement » et non sur la base d'un ensemble pré-sélectionné automatiquement. Cela limite le volume de données traitées : concrètement il a fallu environ 20h de travail pour retenir 1600 profils.

De plus on ne peut garantir de façon absolue qu'une personne dont le profil contient les mots clés « optimist » ou « happy ne souffre pas en réalité de dépression. Fort heureusement les personnes malades sont minoritaires. En supposant que le taux de dépression chez les utilisateurs Twitter reflète la prévalence de la dépression dans la population générale, le risque de sélectionner un utilisateur malade sur un tirage aléatoire est inférieur à 1/10. Nous avons fait l'hypothèse que ce risque est encore plus faible avec un sélection aléatoire parmi des personnes se présentant comme heureuse ou optimiste.

En résumé la méthode que nous avons présentée ici n'est pas idéale mais elle a le mérite de fournir une première base d'étude.

2.3 Récupération des tweets postés par une liste d'utilisateurs

Le code permettant de récupérer les tweets postés par une liste d'utilisateurs s'effectue en deux étapes :

1. Un premier script python (cf. annexe 9.1.6) télécharge un fichier json pour chaque utilisateur (le nom du fichier correspond à l'identifiant Twitter de l'utilisateur). Il est appliqué :
   - sur la liste des utilisateurs souffrant de dépression (groupe traité) ;
   - sur la liste des utilisateurs n'en souffrant pas (groupe de contrôle).

   Les fichier json produits contiennent non seulement les messages mais aussi toutes les informations connexes fournies par l'API Twitter (date d'envoi, fichiers multimédias joints, description de l'utilisateur ayant écrit le message, langue du message, etc.).
2. Un second script python (cf. annexe 9.1.7) génère à partir de chaque fichier .json un fichier .txt :
   - listant les textes des messages (avec un message par ligne) ;
   - sans les informations connexes.

Il serait bien sûr possible de fusionner les deux étapes, et d'optimiser le traitement en ne téléchargeant que le texte des messages à l'étape 1).

Nous avons préféré récupérer et conserver le maximum de données sur chaque utilisateur afin de pouvoir au besoin procéder à des analyses complémentaires.

Pour donner un ordre de grandeur, les fichiers textes contenant (uniquement) les messages de 3200 utilisateurs Twitter, obtenus à l'issue de l'étape 2 , occupent 600Mo d'espace disque.

2.4 Création des jeux de données d'apprentissage et de test

A partir des 3200 fichiers .txt créés avec le traitement exposé au paragraphe précédent, on constitue :

- un jeu de données d'apprentissage avec :
  - 1000 fichiers de messages d'utilisateurs « déprimés »
  - 1000 fichiers de messages d'utilisateurs « non déprimés »
- un jeu de données (distinctes) de test avec :
  - 600 fichiers de messages d'utilisateurs « déprimés »
  - 600 fichiers de messages d'utilisateurs « non déprimés »

3. Analyse exploratoire

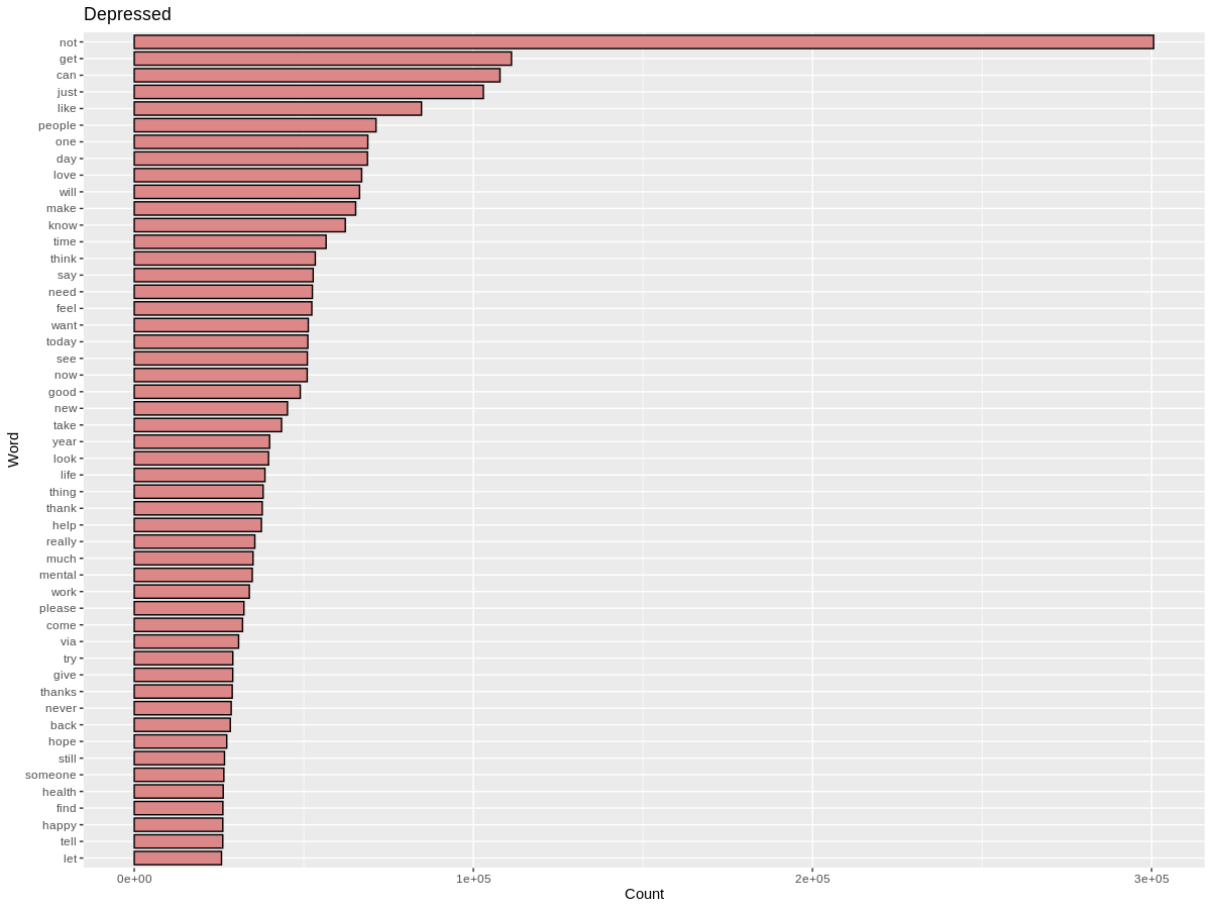
3.1 Termes les plus utilisés

Afin d'obtenir les termes les plus utilisés dans les tweets, nous exécutons le traitement Spark/ Scala suivant (cf. code à l'annexe 9.2.1):

1. lecture des fichiers textes contenant les messages des tweets et création d' un RDD ;
2. suppression des stops words et lemmatisation des messages avec le traitement présenté dans le TP RCP216 : « Fouille de données textuelles » ;
3. création d'un dataframe *tweetsDF* comportant la colonne « text » à partir des messages lemmatisés ;
4. utilisation d'une fonction d'aggrégation (implémentée nativement avec les dataframes Spark) pour compter les termes et export du résultat dans un fichier csv:

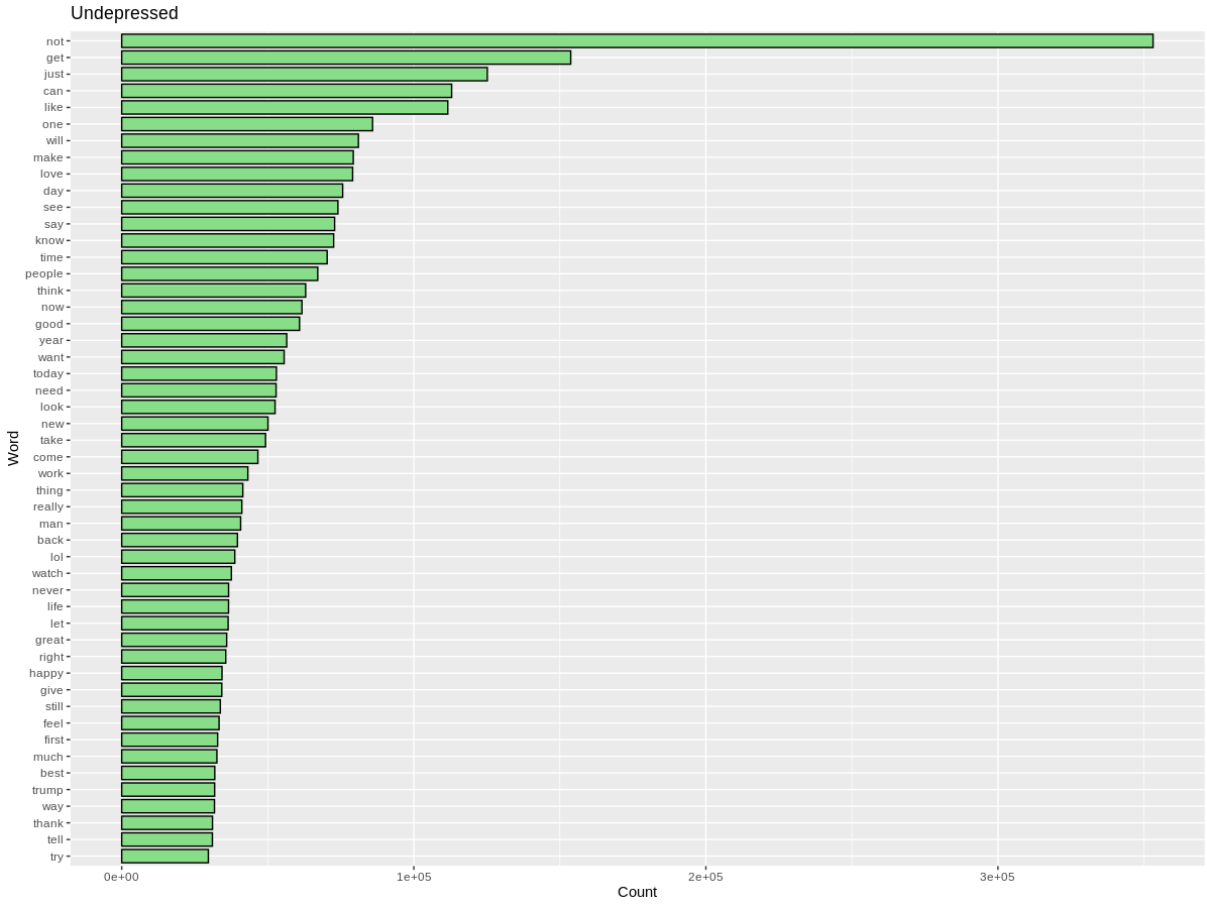
```
val wordsDF = tweetsDF.explode("text","word")((text: String) => text.split(" "))
val wordCountDF = wordsDF.groupBy("word").count().sort($"count".desc)
wordCountDF.coalesce(1).write.format("csv").option("header", "true").csv(outputFile)
```

Nous pouvons alors charger le fichier csv dans Rstudio et visualiser les 50 termes les plus utilisés sous forme de diagrammes en barre (cf. annexe 9.2.1). Pour les utilisateurs Twitter souffrants de dépression, nous obtenons le diagramme suivant :



Nous notons la présence des termes « *mental* » et « *health* », respectivement en 33 et 46ème position.

Pour les utilisateurs ne souffrant pas de dépression, le diagramme obtenu est :

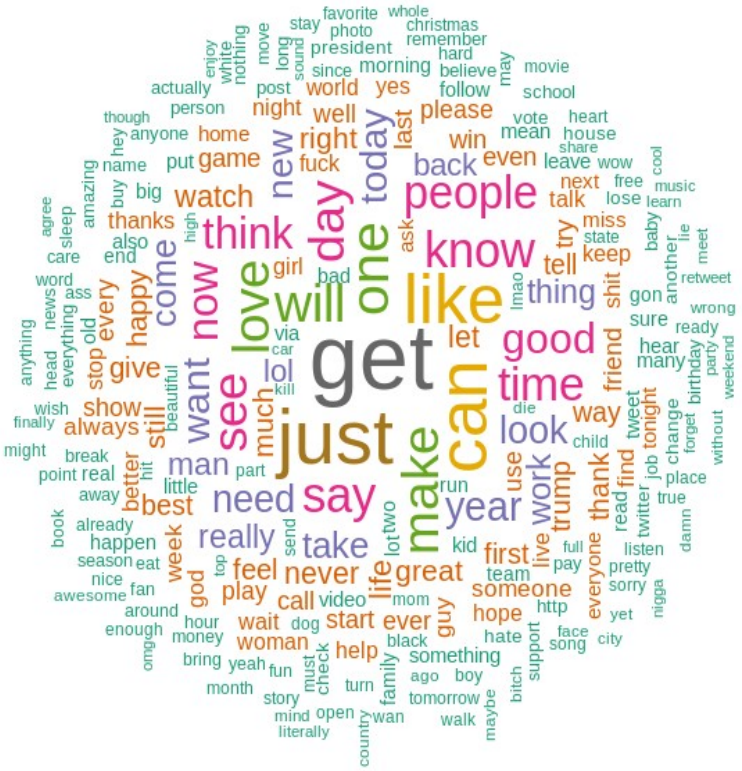


Les termes « mental » et « health » sont ici absents, en revanche nous remarquons la présence des termes tels que « lol », « great » et « best ».

Une autre façon de représenter la fréquence d'apparition des termes dans les tweets consiste à dessiner des nuages de mots. Pour produire les graphiques ci-dessous, nous avons retenus les 250 termes les plus employés (en omettant « not », peu significatif) et les avons traités avec la bibliothèque R wordcloud (cf. annexe 9.2.1) :

Utilisateurs souffrants de dépression :

Utilisateurs ne souffrant pas de dépression :



Nous notons, pour les utilisateurs du premier groupe, en périphérie du nuage, la présence des termes « depression », « bipolar », « disorder », « suicide », « struggle » absents chez les utilisateurs du second groupe.

Une dernière remarque : l'utilisation de Spark/Scala pour compter les termes permet de travailler avec un volume de texte important. Le fichier csv résultant est lui de taille suffisamment réduite (25000 lignes) pour être exploité avec R.

3.2 Recherche des thèmes principaux (LDA)

Suite à l'analyse, au paragraphe précédent, des termes les plus employés dans les tweets, nous avons remarqué des différences significatives entre les utilisateurs « déprimés » et les « non déprimés ». Des termes relatifs à la dépression sont clairement présents dans un cas et absents (ou en tout cas nettement plus rares) dans l'autre. Pour approfondir cette étude, il est intéressant d'étudier non seulement les termes, considérés individuellement, hors de tout contexte mais aussi les thèmes principaux abordés dans les messages Twitter.

Cette recherche des thèmes principaux est réalisée en appliquant une des techniques d'analyse sémantique latente abordées dans le cours RCP216 « Fouille de données textuelles ». Nous avons choisi l'allocation de Dirichlet latente (LDA) car elle donne de bons résultats et est intégrée dans la version dataframe de l'API Mllib. Une alternative aurait été d'appliquer une décomposition en valeurs singulières (Singular Value Decomposition, SVD) mais celle-ci requiert la version RDD de l'API Mllib, plus ancienne.

Le code, présenté dans l'annexe 9.2.2, nous a permis d'extraire les dix sujets suivants, triés par ordre de prépondérance décroissante :

	Utilisateurs « déprimés »	Utilisateurs « non déprimés »
1	better, yes, lol, day, miss, not, time, work, world, well	shit, good, feel, girl, make, just, not, one, twitter, bitch
2	not, care, right, like, feel, can, now, bad, make, need	tonight, not, lmao, today, call, see, little, money, party, get
3	tweet, twitter, thanks, god, night, **suicide**, not, baby, morning, join	read, gon, day, story, anyone, win, not, give, live, awesome
4	thank, wait, not, change, much, can, new, real, life, make	lol, look, not, always, see, sure, happen, just, really, watch
5	one, love, not, fuck, best, word, just, check, amazing, shit	not, will, can, love, wait, year, play, believe, find, baby
6	**mental**, not, **depression**, just, start, **health**, think, **illness**, free, can	week, god, happy, fuck, birthday, not, life, time, true, can
7	beautiful, die, little, man, girl, tonight, sleep, kid, nice, wow	wow, via, another, lose, friend, sorry, like, get, well, yeah
8	good, back, not, stay, hard, come, day, today, ago, year	not, need, let, now, beautiful, wrong, run, agree, damn, mean
9	may, not, hear, say, send, ask, true, really, love, time	not, yes, kid, people, know, like, big, care, something, want
10	not, get, break, heart, trump, just, vote, sorry, sign, enjoy	thank, thanks, get, great, not, guy, free, photo, miss, vote

Nous constatons ici la présence, chez les utilisateurs souffrant de dépression, de sujets relatifs à la maladie (n°3 et n°6) qui n'apparaissent pas chez les autres utilisateurs. Ces résultats, tout comme celui du sous-chapitre précédent, tendent à confirmer l'hypothèse selon laquelle il existe des motifs textuels identifiables discriminant les deux groupes.

3.3 Analyse de sentiment (SparkNLP)

SparkNLP, qui fait l'objet d'un TP RCP216, offre des fonctionnalités très riches, notamment la capacité de fournir une estimation du sentiment « positif » ou « négatif » qui se dégage d'une phrase.

Nous avons utilisé cette bibliothèque pour implémenter l'algorithme suivant (cf. annexe 9.2.3) :

- calcul des « sentiments » sur les 50 tweets les plus récents de chaque utilisateur du jeu d'apprentissage(*)
- agrégation des sentiments par catégorie « positive », « negative », « na » (not available)

L'idée était de vérifier si les « sentiments » des tweets émis par les utilisateurs « déprimés » étaient plus « négatifs » que ceux des utilisateurs « non déprimés ».

Nous avons obtenu les résultats suivants :

Utilisateurs souffrant de dépression

```
+---------+-----+
|sentiment|count|
+---------+-----+
| negative|22795|
| positive|18391|
|       na| 5029|
+---------+-----+
```

Utilisateurs ne souffrant pas de dépression

```
+---------+-----+
|sentiment|count|
+---------+-----+
| negative|22403|
| positive|18763|
|       na| 5194|
+---------+-----+
```

La différence entre les deux groupes n'est pas flagrante. Cela tient peut être au fait que nous avons fourni en entrée à SparkNLP des tweets dont nous avions retiré préalablement les « stop words » et qui avaient déjà été lemmatisés. Ce point mériterait un examen plus approfondi.

En tout cas, un des aspects extrêmement intéressant de SparkNLP est de permettre, grâce à des pipelines prédéfinis l'écriture de traitements complexes avec très peu de code.

Ainsi l'analyse de sentiment proprement dite, telle qu'implémentée dans ce chapitre, se résume à cinq lignes:

```
val sentimentPipelineModel = PretrainedPipeline("analyze_sentiment").model

val finisherSentiment = new Finisher().setInputCols("document","sentiment")

val pipelineSentiment = new Pipeline().setStages(Array(sentimentPipelineModel,finisherSentiment))
val modelSentiment = pipelineSentiment.fit(tweetsDF)
val sentimentTweetsDF = modelSentiment.transform(tweetsDF).select("text","finished_sentiment")
```

*(*) Nous nous sommes limités à 50 tweets par utilisateur pour des raisons de temps de traitement. Parmi les 50 tweets, certains (par exemple ceux ne comportant qu'une émoticône) sont filtrés suite à la lemmatisation (car le message résultant est vide). Le total des « sentiments » calculé pour chaque groupe n'est donc pas de 50000 (50 tweets*1000 utilisateurs) mais est plus proche des 46000.*

4. Prétraitement

Dans notre projet le prétraitement consiste, à partir des fichiers textes contenant les messages twitter, à construire des Dataframes Spark dans un format adapté à la classification supervisée présentée au chapitre 5. Les différentes étapes du prétraitement sont détaillés dans le paragraphe ci-dessous. Le code correspondant est l'objet de l'annexe 9.3

4.1 Lecture des fichiers de messages Twitter

Nous disposons en entrée d'un fichier texte par utilisateur et le nom du fichier correspond à l'identifiant Twitter de cet utilisateur.

Afin de lire l'ensemble des fichiers de chaque groupe («déprimé » / « non déprimé ») nous utilisons la fonction `SparkContext.wholeTextFiles` qui retourne des pairs (nom de fichier, contenu du fichier). Il suffit alors de rajouter un label correspondant au groupe (1 = «déprimé » , 0 = « non déprimé ») pour obtenir une structure du type :

(identifiant utilisateur, texte des messages, classe cible)

4.2 Lemmatisation et suppression des stop words

La lemmatisation et réalisée avec des bibliothèques du projet Stanford NLP, regroupées dans le jar suivant : http://cedric.cnam.fr/~ferecatu/RCP216/tp/tptexte/lsa.jar.

Nous retirons également des messages Twitter les « stops words » ne présentant pas d'intérêt pour la classification. La liste des « stops words » à filtrer est basée sur celle contenue dans le zip suivant : http://cedric.cnam.fr/~ferecatu/RCP216/tp/tptexte/deps.zip. Nous avons supprimé de cette liste les négations (« no », « nor », « not ») car un tweet tel que « happy » n'a clairement pas la même signification que le tweet « not happy».

4.3 Création d'un modèle Word2vec

Dans le cadre de notre projet RCP216, nous avons effectué une analyse de sentiment sur des revues IMDB et comparé à cette occasion deux méthodes de représentation textuelle : matrice TF-IDF et vecteurs Word2vec. Les deux approches nous ont permis d'obtenir des classifications d'une précision voisines l'une de l'autre mais Word2vec présente un gros avantage : celle de travailler en dimension nettement réduite. Un dataframe TF-IDF comportant plusieurs milliers de colonnes, peut avantageusement être remplacé par un dataframe Word2vec n'en comportant que 100. C'est donc cette dernière solution que nous avons choisie ici. Nous aurions pu utiliser un modèle Word2vec prédéfini mais avons préféré construire un modèle spécifique, adapté à nos données, construit sur la base des tweets que nous avions téléchargés (cf. annexe 9.3.1)

4.4 Application du modèle Word2vec

Nous avons appliqué le modèle Word2vec (cf annexe 9.3.2) en calculant:

- le vecteur Word2vec représentant chaque message Twitter (moyenne des vecteurs représentant chaque terme du message) ;
- la moyenne, par utilisateur, des vecteurs Word2vec sur l'ensemble de ses messages.

5. Création et validation d'un modèle supervisé

5.1 Création d'un modèle supervisé (régression logistique)

A ce stade du projet nous disposons d'un jeu de données d'apprentissage avec :

- 2000 individus (1000 du groupe traité, 1000 du groupe de contrôle)
- 100 variables explicatives (les composantes du vecteur Word2vec associé à chaque individu : ce vecteur synthétise le contenu textuel des messages Twitter)
- 1 variable cible binaire (1 : « déprimé », 0 : « non déprimé »)

Plusieurs méthodes de classification sont envisageables : il nous a paru raisonnable de sélectionner la régression logistique pour les raisons suivantes :

- les variables explicatives (composantes de vecteurs Word2vec), par la construction même du modèle Word2Vec (qui compresse les informations), ne sont pas redondantes et le risque d'avoir des valeurs aberrantes est limité;
- il est naturel de créer un modèle initial simple en supposant que les deux classes de la variable cible sont linéairement séparables, avant d'essayer de construire des modèles plus complexes, plus lourds (en termes de ressources, de temps d'apprentissage) ;
- la régression logistique permet d'obtenir non seulement une prédiction brute, binaire, mais également les probabilités conditionnelles associées à chaque classe, intéressantes pour évaluer la fiabilité d'une prédiction.

Le code Spark / Scala implémentant la création du modèle est présenté à l'annexe 9.4.1.

Deux points sont à souligner dans la modélisation que nous avons développée.

Premièrement, l'algorithme recherche les valeurs optimales des hyper paramètres (nombre maximum d'itérations, régularisation, paramètre elastic net) en parcourant une grille de valeurs possibles :

```
val paramGrid = new ParamGridBuilder().
                    addGrid(logReg.elasticNetParam, Array(0.0,  0.2,  0.4, 0.6, 0.8, 1)).
                    addGrid(logReg.maxIter, Array(50,100,150,200,250)).
                    addGrid(logReg.regParam, Array(0.01,  0.1,  0.5)).build()
```

Deuxièmement chaque jeu d' hyper paramètres de la grille est évaluée avec une validation croisée k-fold (k=5) :

```
// Définition du scaler pour centrer les variables initiales
val scaler = new StandardScaler().setInputCol("features").setOutputCol("scaledFeatures")
                    .setWithStd(false).setWithMean(true)

// Définition du pipeline (scaler puis régression logistique)
val pipeline = new Pipeline().setStages(Array(scaler,logReg))

// Définition de l'instance de CrossValidator : à quel estimateur l'appliquer,
//  avec quels (hyper)paramètres, combien de folds, comment évaluer les résultats
val cv = new CrossValidator().setEstimator(pipeline).setEstimatorParamMaps(paramGrid).setNumFolds(5).
                    setEvaluator(new BinaryClassificationEvaluator())

// Construction et évaluation par validation croisée des modèles correspondant
//   à toutes les combinaisons de valeurs de (hyper)paramètres de paramGrid
val cvLogRegModel = cv.fit(trainDF)
```

L'objet *cvLogRegModel* produit par la validation croisée contient le meilleur modèle (généré avec la combinaison de valeurs optimales pour les hyper paramètres et construit sur l'intégralité des données d'apprentissage).

La combinaison d'hyper paramètres optimale, qu'on peut de extraire de *cvLogRegModel*, est la suivante :

maxIter	regParam	elasticNetParam
50	0.01	0

Nous remarquons que cette combinaison est celle présentant les termes de régularisation (*regParam* et *elasticNetParam*) ayant les valeurs les plus faibles de la grille.

Par ailleurs si on applique *cvLogRegModel* sur les données d'apprentissage, on obtient les résultats ci-dessous :

Score AUC	Précision
0.94	87 %

Bien sûr ces résultats en tant que tels ne signifient pas grand-chose : il convient de valider le modèle sur des données différentes de celles utilisées pendant l'apprentissage. C'est l'objet du paragraphe suivant.

5.2 Validation du modèle

Afin de valider le modèle construit au paragraphe précédent, nous l'appliquons sur le jeu de données de test (qui comporte 600 individus « déprimés » et 600 « non déprimés », distincts de ceux présents dans le jeu d'apprentissage). Le code Spark/Scala effectuant ce traitement est reproduit dans l'annexe 9.4.2. Les résultats sont les suivants :

Score AUC	Précision
0.96	89 %

Nous observons que le modèle produit une classification de performance comparable sur le jeu de données de test et sur le jeu de données d'apprentissage. Il n'y a donc pas de sur-apprentissage (ce qui est consistant avec les faibles valeurs de régularisation observées dans le paragraphe précédent) ni de sous-apprentissage important (la performance de la classification est dans les deux cas élevée) .

Le score AUC (0.96) et la précision obtenus (89%) sur les données de test semblent indiquer que le modèle est bien capable de distinguer un utilisateur Twitter **déclarant dans son profil souffrir de dépression** , d'un utilisateur **déclarant dans son profil être optimiste/heureux** en analysant uniquement le texte de leurs messages respectifs.

En revanche, ces résultats ne permettent pas à a eux seuls, de savoir dans quelle mesure :

- Les utilisateurs Twitter déclarant explicitement souffrir de dépression dans leurs profils sont représentatifs de l'ensemble des utilisateurs Twitter atteints par cette maladie ;
- Les utilisateurs se présentant comme optimistes / heureux constituent un groupe de contrôle pertinent ;

De plus nous ne pouvons pas exclure la présence dans le modèle de biais, liés à la méthode de recueil et d'étiquetage des données que nous avons employée. Le fait par exemple que les utilisateurs que nous avons étiquetés comme « déprimés » sont tous abonnés à au moins un des comptes de la liste (*depressionarmy*, *depressionnote*, *dbsalliance*, *NIMHgov)* alors que les « non déprimés » ont été recueillis depuis un flux et à priori ne sont pas liés les uns aux autres peut avoir un impact non négligeable sur la classification.

Une première solution pour répondre à ces interrogations consisteraient à évaluer la performance prédictive du modèle sur d'autres échantillons d'utilisateurs Twitter dont le diagnostique (« déprimé » / « non déprimé), validé par une autorité médicale, est connu.

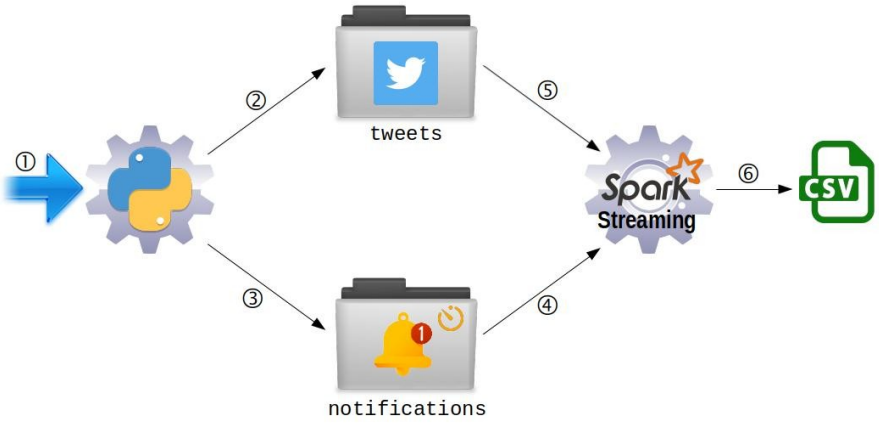
Une autre option serait d'appliquer notre modèle sur un échantillon d'utilisateurs Twitter sélectionnés aléatoirement puis de faire confirmer les prédictions par des experts spécialisés dans le domaine. Afin de constituer un tel échantillon, nous pouvons par exemple appliquer le modèle sur un flux twitter : c'est l'objet du chapitre suivant.

6. Application du modèle sur un flux Twitter

Afin d'appliquer le modèle de classification supervisée sur un flux Twitter, nous avons mis en œuvre le principe de « separation of concerns ». Concrètement nous avons dissocié l'acquisition des données depuis le flux Twitter de l'application du modèle supervisé en développant deux tâches distinctes :

- La partie acquisition de données est réalisée par un script python basé sur la bibliothèque tweepy(*) (déjà employée dans le projet en raison de sa facilité d'utilisation). Le code python de cette partie acquisition est présenté à l'annexe 9.5.1 ;
- La partie application du modèle (cf. annexe 9.5.2) est implémenté à l'aide d'un programme Spark / Scala basé sur l'API Spark Streaming.

L'intégration des deux tâches est représentée dans la figure ci-dessous :



Étape	Description
1	La tâche python, connectée à un flux Twitter, récupère les tweets contenant le mot clé « depression »
2	Si l'auteur d'un tweet mentionnant le mot clé « depression » a un historique Twitter suffisant, la tâche python récupère ses 50 derniers messages et les stocke sous forme d'un fichier .txt dans le répertoire « *tweets* ». Le nom de ce fichier correspond à l'identifiant Twitter de l'utilisateur, ex : « 534607320.txt » .
3	La tâche python crée dans le répertoire « *notifications* » un fichier contenant l'identifiant Twitter de l'utilisateur dont elle vient de télécharger les messages. Ce fichier permet d'avertir la tâche Spark qu'une classification doit être réalisée.
4	La tâche Spark surveille, grâce à l'API Streaming, l'apparition de fichiers dans le répertoire « *notifications* ». Elle lit dans ces fichiers les identifiants Twitter des utilisateurs à traiter.
5	Pour chaque identifiant Twitter lu dans un fichier du répertoire « *notifications* », la tâche Spark : 1. lit dans le répertoire « *tweets* » le fichier .txt contenant les messages de l'utilisateur correspondant ; 2. applique le modèle de classification.
6	Les résultats de chaque classification sont sauvegardés dans un fichier csv.

Une question naturelle se pose lorsqu'on examine l'algorithme ci-dessus : pourquoi créer un fichier de notification distinct du fichier contenant les messages Twitter ? La tâche Spark Streaming ne pourrait-elle pas surveiller directement le répertoire « *tweets* » ?

(*) Depuis la version 2.0, Spark n'intègre plus le support de l'API Twitter. Une extension existe, Apache Bahir, mais elle n'est pas compatible avec la version stable la plus récente de Spark.

La principale raison de choix est liée à la difficulté d'extraire l'identifiant utilisateur à partir du nom du fichier contenant les tweets. En effet, déterminer le nom du fichier en cours de traitement par Spark Streaming n'est pas trivial :

https://stackoverflow.com/questions/29031276/spark-streaming-dstream-rdd-to-get-file-name

Il s'avère plus simple en pratique de lire les identifiants utilisateurs dans des fichiers distincts avec le code suivant :

```
// création du flux avec une fenêtre de 10 secondes
val ssc = new StreamingContext(sc, Seconds(10))

// scrute dans le répertoire "notifications" l'apparition de fichiers
// chaque fichier de notification contient l'identifiant d'un utilisateur Twitter à traiter
val lines = ssc.textFileStream("notifications")
lines.foreachRDD(rdd => {
  if ((rdd != null) && (rdd.count() > 0) && (!rdd.isEmpty()) ) {
    // déclenche une classification pour chaque identifiant utilisateur lu dans les fichiers de notification
    rdd.collect().foreach(user_id => processUserTweets(user_id,stopWords,w2vModel,cvLogRegModel))
  }
})

// démarrage du traitement du flux
ssc.start()
ssc.awaitTermination()
```

Cette solution présente d'autres avantages :

- le fichier de notification est créé **après** le fichier contenant les tweets. Lorsque la tâche Spark détecte une notification, toutes les données nécessaires pour appliquer la classification sont disponibles. A contrario, le fait de déclencher une classification dès le moment où Spark détecte l'apparition d'un fichier de tweets peut causer des problèmes de synchronisation (Spark pourrait réaliser le traitement sur un fichier incomplet, en cours d'écriture) ;
- le téléchargement des tweets et leur classification sont indépendants (et pas forcément exécutés sur la même machine) : on peut par exemple télécharger un volume de tweets importants pour réaliser des analyses en différé et ne réaliser une classification « en temps réel » que sur un échantillon des données récupérées.

Nous avons testé le code présenté plus haut sur un PC portable d'entrée de gamme. La tâche python a produit environ 600 fichiers de notification en 10 minutes. Durant le même intervalle de temps, la tâche Spark n'en a traité que la moitié.

Dans le cadre d'un passage à l'échelle, il est possible de déployer ce type de solution sur une grappe de serveurs. Cela suppose :

- de rendre le code plus robuste (purge du répertoire *notifications*, suppression ou historisation sur un support adapté des tweets après classification, inhibition de la classification pour les utilisateurs postant fréquemment des messages, etc. …) ;
- d'équilibrer les ressources entre les tâches python d'acquisition et les tâches Spark de classification (pour ne pas récupérer plus de données que ce qui peut être traité, ou l'inverse) ;
- de s'assurer que les tweets d'un même utilisateur ne sont pas traités simultanément par plusieurs nœuds (par exemple en partitionnant l'ensemble des utilisateurs Twitter à prendre en compte et en distribuant le résultat du partitionnement entre les différents nœuds chargés de l'acquisition).

7. Passage à l'échelle

7.1 Acquisition des données

La question du passage à l'échelle se pose non seulement lors de la phase de création du modèle (avec un dataset dont la dimension est potentiellement élevée) mais dès la phase d'acquisition et d'étiquetage des données. Avant d'aller plus loin, rappelons brièvement la méthode que nous avons employée pour récupérer les textes des tweets postés par 1600 utilisateurs « déprimés » , ainsi que le volume des données associées :

Étape	Description	Volume de données (ordre de grandeur)
1	Récupération de la liste des identifiants d'utilisateurs Twitter abonnés à des comptes traitant de la dépression *(pour simplifier on ne considère dans la suite du tableau qu'un seul compte, « dbsalliance », qui nous a permis d'obtenir 500 individus « déprimés » sur les 1600 de notre étude).*	
2	**Téléchargement des profils utilisateurs (id, nom, description, emplacement géographique)** à partir de la liste des identifiants.	10 Mo
3	Insertion des profils dans une instance Elasticsearch locale.	
4	Filtrage des profils par une requête Elasticsearch sur le champ « description ». Les résultats sont exportés dans une feuille Excel.	100 Ko
5	Validation manuelle des profils dans la feuille Excel.	
6	Téléchargement des tweets postés par les utilisateurs dont le profil a été validé (uniquement le texte des messages sans les information connexes : date d'envoi, pièces jointes, description de l'utilisateur ayant écrit le message,etc.).	100 Mo

Le principe fondamentale de cette chaîne de traitements est d'appliquer une série de filtres afin de diminuer drastiquement le nombre d'utilisateurs dont les tweets doivent être téléchargés :

Étape	Filtre	Nombre d'utilisateurs
0	Tous les utilisateurs Twitter	300 000 000
1	Utilisateurs abonnés au compte « dbsalliance »	30 000
4	Utilisateurs abonnés au compte « dbsalliance » **filtrés par requête Elasticsearch sur le champ « description » de leurs profils**	2 000
5	Utilisateurs abonnés au compte « dbsalliance » validés manuellement sur la base de la présélection effectuée par la requête Elasticsearch	500

Les seules informations utilisées à des fins de filtrage sont les **profils utilisateurs pas le contenu des messages**.

Cette solution a l'avantage de réduire suffisamment le volume des données pour qu'elles puissent être traitées sur une seule machine.

Elle présente néanmoins un inconvénient : les descriptions que les utilisateurs Twitter rédigent dans leurs profils sont souvent très succinctes - ex : « 25 years old. #depression sufferer ».

Par conséquent, la description ne permet souvent pas, à elle seule, de déterminer si l'utilisateur a fait l'objet d'un diagnostique officiel (par un médecin, un psychologue, etc.).

Afin de résoudre ce problème une possibilité est de s'inspirer du Reddit Self-reported Depression Diagnosis (RSDD) dataset qui comprend les posts sur le forum Reddit d'environ 9 000 utilisateurs ayant déclaré avoir reçu un diagnostique de dépression («utilisateurs diagnostiqués») et environ 107 000 utilisateurs témoins appariés. Pour sélectionner les «utilisateurs diagnostiqués», la technique mise au point par les auteurs du RSDD a été de rechercher dans les messages la présence de patterns très spécifiques.

Cette technique peut être implémenté dans notre projet à l'aide de requêtes Elasticsearch portant sur le contenu textuel des tweets – voici un un exemple simple :

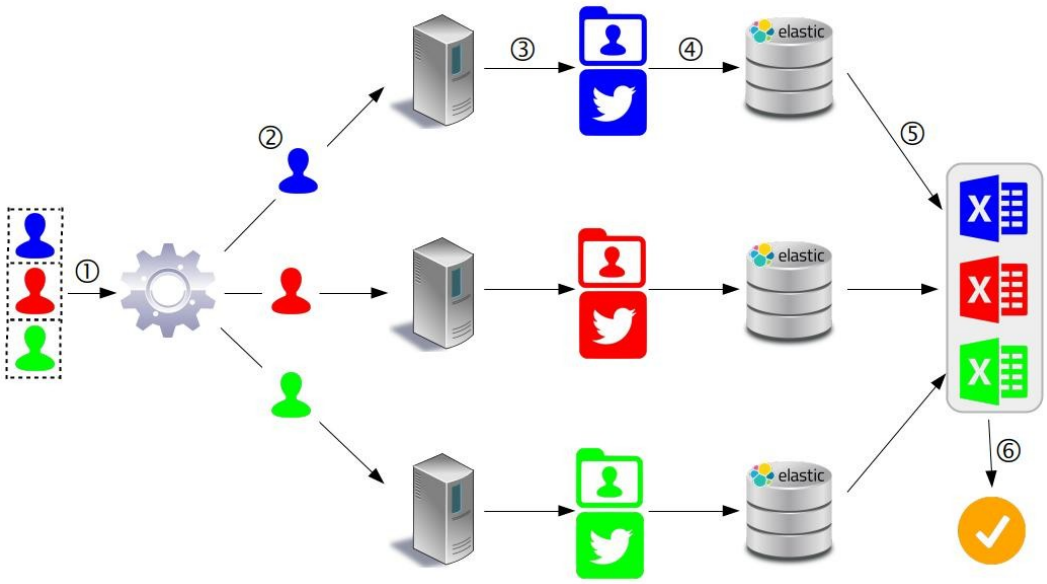
```
GET tweets/_search
{
    "query": {
        "query_string" : {
            "query" : "(-RT +I +was +diagnosed +with +depression ) OR (-RT +I +was +diagnosed +bipolar)",
            "fields" : ["texte_message"]
        }
    }
}
```

N.B. : le « -RT » a pour but d'éliminer les « ReTweets » (messages relayés par un utilisateur et dont il n'est pas forcément l'auteur). Les termes préfixés par un « + » sont obligatoirement présents dans les résultats et y apparaissent dans le même ordre que celui de la requête.

Une telle requête nécessite de disposer des tweets de chaque utilisateur à étiqueter. Cela entraîne une forte augmentation du volume de données devant être récupérées et traitées par rapport à la solution décrite au début du chapitre.

Concrètement, dans le cas de DBSAlliance, il faut télécharger les messages postés par l'ensemble des 30 000 abonnés du compte et pas seulement des 500 utilisateurs sélectionnés sur la base de leur profil.

Afin de gérer le passage à l'échelle, une possibilité intéressante est de mettre en œuvre une architecture inspirée de MapReduce, comme celle représentée dans le schéma de la page suivante.



Étape	Description
1	Un premier processus (unique, localisé sur n'importe quel machine de la grappe) récupère la liste des identifiants d'utilisateurs Twitter abonnés à des comptes tels que "dbsalliance" , "depressionarmy", "depressionnote",etc. : cette liste peut comporter plusieurs dizaines de milliers d'identifiants.
2	Après avoir éliminé les doublons, le processus divise la liste des identifiants en n sous-listes de taille égale et les transmet aux n machines de la grappe
3	Chaque machine télécharge le profil et les messages des utilisateurs Twitter dont les identifiants lui ont été transmis.
4	Chaque machine insère, dans une index Elaticsearch local, les informations relatives aux profils utilisateurs et les messages associés – ex : **(id_utilisateur, nom_utilisateur, description_utilisateur, texte_message)**
5	Une requête Elasticsearch spécifique - ex : "(-RT +I was +diagnosed +with depression) OR (-RT +I was +diagnosed +bipolar)" – portant sur le champ *texte_message* permet de pré-sélectionner les utilisateurs. Les résultats sont exportés dans un répertoire partagé sous forme de fichiers Excel (un fichier par machine de la grappe).
6	Les feuilles Excel sont validées manuellement en prenant en compte à la fois la description du profil et le texte du (ou des) message(s) mentionnant un diagnostique.

Remarques :

- le traitement exécuté par chaque machine peut être implémenté, moyennant adaptations, avec les scripts python que nous avons déjà développés (cf. annexe 9.1) ;
- les étapes 1 et 2 (récupération puis découpage de la liste des identifiants utilisateurs à traiter) sont réalisées par un processus unique localisé sur une seule machine. Cela ne pose pas de problème en terme de montée en charge car ces étapes sont négligeables en terme de temps d'exécution et de ressources matérielles consommées par rapport au travail réalisé sur chaque machine lors des étapes 3,4 et 5 ;
- la parallélisation des étapes 3 à 5 permet de soutenir une forte croissance de la volumétrie des données par un ajout, en proportion, de machines dans la grappe de serveurs ;
- cette architecture assure un partitionnement équilibré des données (chaque machine reçoit le même nombre d'identifiants utilisateurs à traiter) mais elle ne gère pas la réplication, sujet abordé dans le prochain paragraphe. Elle n'intègre pas non plus de mécanisme de reprise sur panne « à chaud » : si un dysfonctionnement empêche une

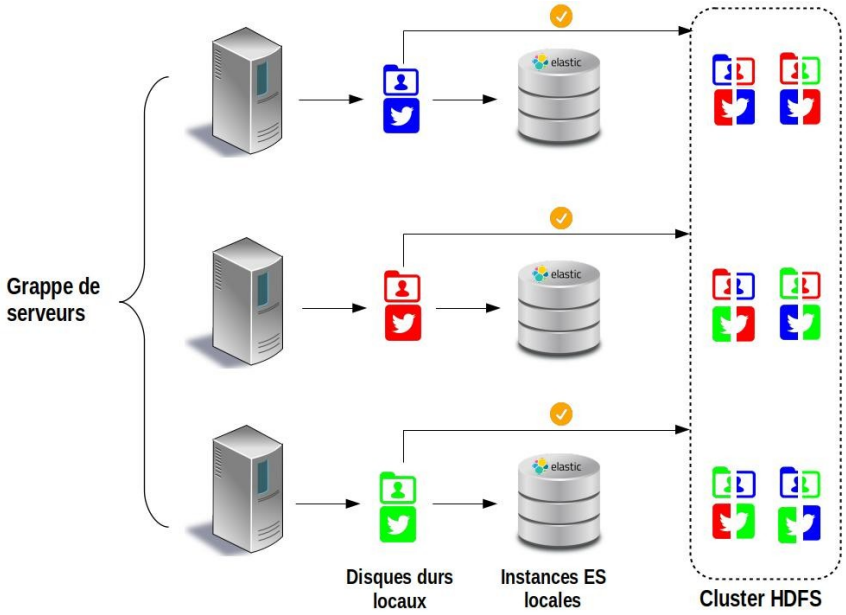
ou plusieurs machines de réaliser le traitement qui lui a été confié, il faut resoumettre au système les données d'entrée (identifiants utilisateurs) n'ayant pas été traitées ;

- une structure similaire (sans les étapes 4 à 6) permet de télécharger en parallèle les tweets des utilisateurs du groupe de contrôle (utilisateurs « non deprimés »).

7.2 Stockage des données

L'architecture proposée dans le paragraphe précédent permet un passage à l'échelle mais n'offre pas un stockage sécurisé des données. En effet, chaque machine télécharge les profils utilisateurs ainsi que les tweets sur son système de stockage local et les importe dans une instance Elasticsearch elle aussi locale. Par conséquent si une machine tombe en panne, les données qu'elle contient sont perdues.

Une solution consiste à sauvegarder les données des utilisateurs Twitter sélectionnés à l'issue de phase d'acquisition (étape 6 du paragraphe précédent) dans un cluster HDFS :



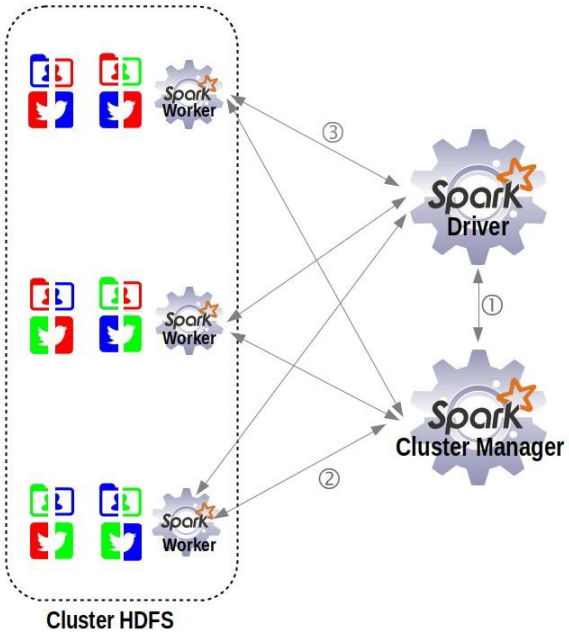
Les fichiers sauvegardés dans HDFS (cf. annexe 9.6.2) sont découpés en blocs de données et ces blocs de données sont redondés entre les différents nœuds du cluster (cf. annexe 9.6.1), comme indiqué dans le schéma ci-dessus. Ce cluster peut être hébergé sur les serveurs utilisés pour télécharger les tweets ou sur une grappe distincte.

N.B. :

- regrouper les instances Elasticsearch locales dans un cluster global afin de répliquer les index n'est pas absolument nécessaire dans la mesure où ces index sont utilisées uniquement pour filtrer les données dans la phase d'acquisition, pas dans les phases de traitement analytique ultérieures ;
- au-delà de la question de la réplication, **l'utilisation de HDFS permet de sauvegarder un volume de tweets important (potentiellement supérieur à la capacité de stockage d'une seule machine) dans un système adapté aux traitements Spark** (cf. chapitre suivant).

7.3 Traitements Spark

Dans le cadre du passage à l'échelle, les traitements Spark peuvent être exécutés (en particulier la création du modèle supervisé) sur un cluster HDFS tel que celui décrit au paragraphe précédent. Le fonctionnement est illustré par le schéma suivant :



Les *workers* (esclaves) sont des nœuds de calcul Spark localisés sur les *datanodes* HDFS et coordonnés par le *driver Spark* (maître). Pour fonctionner dans un environnement distribué, le d*river Spark* a besoin d'un c*luster Manager* chargé d'allouer les ressources sur le cluster.

Le *driver* et le *cluster manager* peuvent fonctionner sur la même machine ou sur des machines différentes, à l'intérieur ou à l'extérieur du cluster.

L'exécution d'un programme comporte les étapes suivantes :

1. Le *driver Spark* se connecte au *cluster manager* et obtient un ensemble d'*executors* Les *executors* sont les processus situés sur chaque nœud qui stockent les données et effectuent les traitements ;
2. Le code du programme est envoyé aux *executors ;*
3. Les tâches à traiter sont envoyées aux *executors* qui les exécutent et retournent le résultat au *driver.*

L'idée centrale d'une telle architecture est de découper les traitements de telle sorte qu'ils puissent être exécutés en parallèles sur les nœuds du cluster. C'est cette parallélisation qui assure la scalabilité du système.

8. Conclusion

Au travers de ce projet nous avons tenté de répondre à deux questions :

1. Est-il possible d'identifier rapidement et à moindre coût sur Twitter un nombre important de personnes souffrant d'une maladie (ici la dépression) en vue de réaliser des études (épidémiologiques, tests thérapeutiques, ...) ?
2. Les méthodes d'apprentissage supervisés utilisées avec succès par Kosinsky et al. pour définir la personnalité d'un utilisateur Facebook à partir de des « Likes » constituent-elles une voie d'expérimentation prometteuse pour détecter si un utilisateur Twitter souffre de dépression ?

Concernant la première question (acquisition des données) : nous avons montré (chapitre 2.1) qu'il était possible, en quelques jours de travail, avec des moyens limités (compte développeur Twitter gratuit, PC portable d'entrée de gamme), de constituer un jeu de données de l'ordre du millier d'individus. La méthode employée consiste à :

- récupérer des profils utilisateurs de personnes abonnées à des comptes traitant de la maladie ;
- insérer ces profils dans une instance Elasticsearch ;
- exécuter une requête Elasticsearch booléenne sur la description des profils pour identifier les utilisateurs malades et éliminer au maximum les autres catégories d' utilisateurs ;
- valider « manuellement » les profils pré-sélectionnés par le filtrage précédent.

A titre d'exemple, cette méthode nous a permis, en récupérant les 30 000 abonnés du compte DBSAlliance de sélectionner 500 personnes (issues de 2000 profils retournés par la requête Elasticsearch) **déclarant souffrir d'une forme de dépression clinique** (simple, bipolaire, dysthymie).

Nous avons ensuite décrit (chapitre 7.1) comment affiner ce processus de sélection en recourant à une grappe de serveurs dans le but d'identifier des personnes **déclarant avoir reçu un diagnostique médical**. Le principe est globalement le même mais implique une parallélisation des traitements, des requêtes Elasticsearch plus spécifiques (portant non pas seulement sur les profils des utilisateurs mais aussi sur le contenu de leurs messages) et un volume de données important nécessitant un passage à l'échelle.

Le point critique posé par ce mode de sélection est celui de la **validité et de la véracité des données** car les personnes retenues l'ont été uniquement sur la base de leurs déclarations. Une façon de prendre en compte ce problème est d'envoyer aux utilisateurs sélectionnés, via l'API Twitter, des messages pour les inviter à répondre à un questionnaire en ligne afin de disposer d'informations plus précises. L'envoi du message et la gestion du questionnaire peuvent être réalisés automatiquement avec l'API Twitter, un simple site Web, des pages PHP et une base MySQL.

Ce type de solution, dont la réalisation sort du cadre du projet UASB03, peut également servir à construire un groupe de contrôle plus fiable que celui nous avons constitué (cf. chapitre 2.2).

Concernant la deuxième question (développement d'une classification automatique supervisée) : nous avons créé avec Spark un modèle de régression logistique basé sur un échantillon de 1000 individus « déprimés » et 1000 individus « non déprimés » à partir des textes de l'ensemble des messages postés par ces individus sur Twitter. Les textes ont été pré traités (suppressions de « stop words », lemmatisation) et représentés sous forme de vecteurs Word2vec (de taille 100) afin de réduire la dimension des données et accélérer l'apprentissage. Nous avons ensuite évalué le modèle sur un jeu de test comportant 600 individus « déprimés » et 600 individus « non déprimés » distincts de ceux utilisés pendant la phase d'apprentissage. Cette modélisation, mise en œuvre avec des échantillons de tailles limitées est applicable dans un contexte de données massives (cf. chapitres 7.2 et 7.3)

Les résultats obtenus (score AUC de 0.96, précision de 89 %) sont intéressants mais posent la question de la représentativité des utilisateurs dont nous avons recueillis les données et celle de la généralisation du modèle. Ce dernier permet-il par exemple d'identifier une personne souffrant de dépression mais n'abordant pas ouvertement le sujet dans ses tweets ? Répondre à ce type d'interrogation nécessite clairement des études complémentaires (cf. chapitre 5.2). En tout cas il s'agit d'une voie de recherche prometteuse car une fois au point, un modèle de classification tel que celui que nous développé peut facilement être utilisé pour analyser un flux twitter en temps réel (cf. chapitre 6) et détecter les utilisateurs à risque parmi une population importante.

D'un point de vue purement pédagogique ce projet nous a permis d'explorer les différentes problématiques de l'UASB03 :

- **V**olume : les données susceptibles d'être traitées sont vraiment massives ;
- **V**ariété : les données d'entrées ne sont pas des données numériques « propres » spécifiquement rassemblées dans un dataset afin de répondre au problème posé mais des données textuelles de qualité inégale pour lesquelles il faut recourir aux techniques de fouilles enseignées dans les différentes UE ;
- **V**élocité : le projet inclut le traitement d'un flux twitter ;
- **V**alidité : la validité des données est évaluée en combinant les outils de recherche textuelle présentés dans l'UE NFE204 pour cibler et réduire les données pertinentes à examiner puis ultimement en procédant à une validation humaine.

De plus le sujet traité nous a offert l'opportunité de mettre en œuvre les compétences et technologies abordées respectivement dans les UE STA211, NFE204 et RCP216:

- analyse exploratoire des données, pré traitements, classification supervisée ;
- Elasticsearch, MapReduce ;
- Spark ML, Spark Streaming, Sark NLP, Word2Vec, LDA.

Enfin le travail réalisé a couvert l'ensemble des grandes étapes indispensables à la réalisation d'un projet big data, en intégrant notamment l'acquisition et l'étiquetage des données, une question à la fois non triviale et fondamentale.

9. Annexes

9.1 Annexe 1 : acquisition et étiquetage des données

9.1.1 Téléchargement d'une liste de profils d'utilisateurs Twitter abonnés à des comptes traitant de la dépression

```
import twitter
import json
import datetime

#########################################################################################################################
# code permettant d'exporter dans des fichiers json une liste de profils d'utilisateurs twitter
# (id, name, screen_name, location, description) abonnés à des comptes traitant de la dépression
#########################################################################################################################

# Wrapper python de l'API twitter
api = twitter.Api(     consumer_key= "XXXXX",consumer_secret = "XXXXX", access_token_key = "XXXXX",
                       access_token_secret = "XXXXX",sleep_on_rate_limit=True)

# liste des comptes twitters ont on souhaite exporter les abonnés
twitter_accounts_about_mental_health = [
      {"id" : "3093201981", "name":"depressionarmy" },   {"id" : "862029310470877185", "name":"depressionnote" },
      {"id" : "94458238", "name":"dbsalliance" },        {"id" : "39250316", "name":"NIMHgov" }]

# écrit sur une ligne d'un fichier json un résumé (id, nom, description,…) du profil twitter de l'utilisateur dont l'id est
# passé en entrée
def write_user_profile(file=None,user_id=None,end_of_line=' },\n'):
    try:
        user_profile= api.GetUser(user_id=user_id)
        if not user_profile.protected:
            file.write('{ "id" : '+json.dumps(user_profile.id_str)+', ')
            file.write('"name" : '+json.dumps(user_profile.name)+', ')
            file.write('"screen_name" : '+json.dumps(user_profile.screen_name)+', ')
            file.write('"location" : '+json.dumps(user_profile.location)+', ')
            file.write('"description" : '+json.dumps(user_profile.description)+end_of_line)
    except Exception as e:
        print(e)

# exporte dans un fichier json, les profils twitter d'une liste d'utilisateurs
def dump_users_profiles(filename=None,user_ids=None):
    print(str(datetime.datetime.now())+" write file : "+filename)
    nb_users = len(user_ids)
    if nb_users==0:
        print("    empty file")
        return
    with open('data/'+filename, 'w+') as f:
        f.write('[\n')
        for i in range(0, nb_users - 1):
            user_id = user_ids[i]
            write_user_profile(file=f,user_id=user_id)
        user_id = user_ids[nb_users - 1]
        write_user_profile(file=f,user_id=user_id,end_of_line=' }\n]')

# récupère les profils d'utilisateurs abonnés à un compte twitter spécifié en entrée et les exporte dans des fichiers json
# (la récupération se fait par tranches de 5000 profils, chaque tranche est stockée dans un fichier distinct)
def dump_followers_profiles(account=None):
    cursor=-1
    results = api.GetFollowerIDsPaged(user_id=account["id"],cursor=cursor)
    cursor =results[0]
    follower_ids=results[2]
    filename = account["name"]+"_"+str(cursor)+".json"
    dump_users_profiles(filename=filename,user_ids=follower_ids)
    while cursor>0:
        results = api.GetFollowerIDsPaged(user_id=account["id"],cursor=cursor)
        cursor = results[0]
        follower_ids=results[2]
        filename = account["name"]+"_"+str(cursor)+".json"
        dump_users_profiles(filename=filename,user_ids=follower_ids)

# parcourt une liste de compte twitters, et pour chacun d'eux exporte dans des fichiers json la liste de ses abonnés
def main():
    for account in twitter_accounts_about_mental_health:
        print("==> "+str(datetime.datetime.now())+' dump_followers_profiles("',account["name"],'")')
        try:
            dump_followers_profiles(account)
        except Exception as e:
            print(e)

if __name__== "__main__":
    main()
```

9.1.2 Import en bloc de profils d'utilisateurs Twitter dans une instance Elasticsearch avec l'API bulk

```
import json, os, elasticsearch, datetime

from elasticsearch import Elasticsearch
from elasticsearch import helpers

#########################################################################################################################
# Code permettant d'importer en bloc de profils d'utilisateurs Twitter dans une instance Elasticsearch avec l'API bulk
# On crée un index elasticsearch par compte Twitter traitant de la dépression :
#    "depressionarmy", "depressionnote", dbsalliance", "NIMHgov"
#  NB : Un profil utilisateur contient les informations suivantes : (id, name, screen_name, location, description)
#########################################################################################################################

# répertoire contenant les profils d'utilisateurs Twitter au format json
directory_in ="data"

# interface elasticsearch
es = elasticsearch.Elasticsearch()

i=1

print("Start bulk import : ", datetime.datetime.now())

for filename in os.listdir(directory_in):
    if filename.endswith(".json"):
        print(str(datetime.datetime.now())+" : "+filename)
        # si le fichier se termine par ",\n"
        # on remplace les 2 derniers chars par "\n]"
        # de façon à ce qu'il respecte la syntaxe d'un tableau json
        with open(directory_in+"/"+filename, 'rb+') as f:
            f.seek(-2,os.SEEK_END)
            last_chars=f.read().decode('utf-8')
            if(last_chars==',\n'):
                f.seek(-2,2)
                f.truncate()
                f.write('\n]'.encode())
        # extraction du nom de l'index elastic search depuis le nom du fichier
        # ex: "NIMHgov_1555896783124150987.json" ==> "NIMHgov"
        # NB: le téléchargement des profils se fait par tranches de 5000
        # pour un compte twitter avec N abonnées, le téléchargement produit donc N/5000 fichiers
        # le "1555896783124150987" dans le nom de fichier ci-dessus correspond à l'identifiant de la tranche
        pos_fin = filename.rfind("_")
        index_name = filename[:pos_fin].lower()
        # creation du fichier bulk et import dans Elasticsearch
        with open(directory_in+"/"+filename) as json_file_in:
            data_out = json.load(json_file_in)
            nb_data = len(data_out)
            ids = range(i,i+nb_data)
            actions = [
                {
                    "_index" : index_name,
                    "_type" : "profiletwitter",
                    "_id" : ids[j],
                    "_source" : data_out[j]
                }
                for j in range(0, nb_data)
            ]
            helpers.bulk(es,actions)
            i+=nb_data
print("End bulk import : ", datetime.datetime.now())
```

9.1.3 Exécution d'une requête Elasticsearch booléenne et export des résultats dans des fichiers Excel

```
import elasticsearch
import tablib
import datetime
import os

#################################################################################################################
# Code permettant d'exécuter une requête Elasticsearch booléenne et d'exporter le résultat dans un fichier Excel
# (la même requête est exécutée sur plusieurs indexes, on crée un fichier Excel résultat par index)
#################################################################################################################

output_directory='es_search_result'

index_names= [ "depressionarmy" ,"depressionnote" , "dbsalliance", "nimhgov" ]

if not os.path.isdir(output_directory) :
        os.mkdir(output_directory)

es = elasticsearch.Elasticsearch()

for index_name in index_names :
        print(str(datetime.datetime.now())+" query : "+index_name)
        res = es.search(index=index_name, body=
                {
                  "query": {
                    "bool": {
                      "must" : [
                        {
                          "terms": {
                            "description": [
                              "depression",
                              "depressed",
                              "bipolar",
                              "bpd",
                              "survivor",
                              "suicide",
                              "diagnosed",
                              "dysthymia"
                            ]
                          }
                        }
                      ],
                      "must_not" : [
                        {
                          "terms": {
                            "description": [
                              "therapist",
                              "psychiatrist",
                              "organization",
                              "fundation",
                              "NPO",
                              "clinic"
                            ]
                          }
                        }

                      ]
                    }
                  }
                }
        , size=10000)  #Nombre maximum d'enregistrements retournés par la requête

        print("Got %d Hits:" % res['hits']['total']['value'])


        data = tablib.Dataset(headers=['id','name','screen_name', 'location','description', 'exclude'])
        for hit in res['hits']['hits']:
                col1 = hit["_source"]["id"]
                col2 = hit["_source"]["name"]
                col2 = col2.replace('\n', ' ')
                col2 = col2.replace('\t', ' ')
                col3 = hit["_source"]["screen_name"]
                col3 = col3.replace('\n', ' ')
                col3 = col3.replace('\t', ' ')
                col4 = hit["_source"]["location"]
                col4 = col4.replace('\n', ' ')
                col4 = col4.replace('\t', ' ')
                col5 = hit["_source"]["description"]
                col5 = col5.replace('\n', ' ')
                col5 = col5.replace('\t', ' ')
                col6 = ' '
                row=(col1,col2,col3,col4,col5,col6)
                data.append(row)

        with open(output_directory+'/'+index_name+'.xlsx', 'wb') as f:
                f.write(data.export('xlsx'))
```

9.1.4 Extraction des identifiants utilisateurs validés depuis des feuilles Excels

```
import datetime
import os
import json

################################################################################
# Code permettant de récupérer dans les fichiers Excels des profils utilisateurs
# les identifiants de ceux qui ont été validés manuellement
# (i.e. lignes  où la colonne exclude n'a pas été cochée avec un "x")
# NB : les identifiants sont stockés dans une map pour éliminer les doublons
# avant d'être exporté dans un fichier json
################################################################################

input_directory  = 'checked_accounts'
output_directory  = 'filtered_results'

if not os.path.isdir(output_directory) :
        os.mkdir(output_directory)

# Map utilisée pour éliminer les doublons
dict_id_count= {}

for filename in os.listdir(input_directory):
        print(str(datetime.datetime.now())+" : "+filename)
        df = pd.read_excel(input_directory+"/"+filename)
        df=df[df['exclude']!='x']
        for row in df.itertuples(index=True):
                id=getattr(row, "id")
                count = 1
                if (id in  dict_id_count) :
                        count = dict_id_count[id] +1
                dict_id_count[id]=count
json.dump( dict_id_count, open( output_directory+'/dict_id_count.json', 'w' ))
```

9.1.5 Récupération dans un flux Twitter des utilisateurs dont la description contient les mots clés « happy » ou « optimist »

```
import tweepy
from tweepy import OAuthHandler
from tweepy import Stream
from tweepy.streaming import StreamListener
import json
import re
import sys
import os
import tablib
import datetime

#############################################################################################
# Code permettant de récupérer dans un flux twitter les profils utilisateurs dont les
# descriptions contiennent les mots clés "happy" ou "optimist"
# Les profils sont exportés dans un fichier Excel
#############################################################################################


keywords =["optimist","happy"]
words_re = re.compile("|".join(keywords))

output_directory='results'
if not os.path.isdir(output_directory) :
        os.mkdir(output_directory)

class MyStreamListener(StreamListener):
        counter=0
        data = tablib.Dataset(headers=['id','name','screen_name', 'location','description', 'exclude'])
        dict_users = {}

        # sur réception d'un tweet :
        def on_status(self, status):
                try:
                        # vérifie que que le tweet contient la description de l'utilisateur qui l'a émis
                        if status.user.description :
                                # vérifie que l'utilisateur n'a pas déjà été récupéré
                                if not status.user.id in self.dict_users:
                                        user_description=status.user.description.lower()
                                       # vérifie que la description de l'utilisateur contient les mots "optimist" ou "happy"
                                        if words_re.search(user_description):
                                                        # export du profil utilisateur dans un fichier Excel
                                                        self.dict_users[status.user.id]=status.user.id
                                                        col1 = str(status.user.id)
                                                        col2 = status.user.name
                                                        col3 = status.user.screen_name
                                                        col4 = status.user.location
                                                        col5 = status.user.description
                                                        col6 = ' '
                                                        row=(col1,col2,col3,col4,col5,col6)
                                                        self.data.append(row)
                                                        self.counter=self.counter+1
                                                        if (self.counter%100==0):
                                                                print(str(datetime.datetime.now())+" : ",self.counter)
                                                                with open(output_directory+'/users.xlsx', 'wb') as f:
                                                                        f.write(self.data.export('xlsx'))
                                                                        if(self.counter>=10000):
                                                                                sys.exit()
                except Exception as e:
                        print(e)

if __name__ == "__main__":
        auth = OAuthHandler("XXXXXXXX", "XXXXXXXX",)
        auth.set_access_token("XXXXXXXX","XXXXXXXX",)
        myStreamListener = MyStreamListener()
        myStream = tweepy.Stream(auth = auth, listener=myStreamListener)
        # l'API twitter  nécessite de filtrer les tweets en précisant un ou plusieurs mots clés.
        # nous définissons le filtre le moins restrictif possible avec le mot clé "the"
        # afin de capter le maximum de tweets
        myStream.filter(languages=["en"], track=["the"])
```

9.1.6 Téléchargement des tweets postés par une liste d'utilisateurs donnée

```
import twitter
import json
import shutil
import sys
import datetime
import os

#######################################################################################################################
# Code permettant de télécharger les tweets postés par une liste d'utilisateurs dans des fichiers json
# (un fichier par utilisateur)
#######################################################################################################################

api = twitter.Api(  consumer_key="xxxxx", consumer_secret="xxxxx", access_token_key="xxxxx",
                    access_token_secret="xxxxx",sleep_on_rate_limit=True)

output_directory = 'tweets'

# récupère l'ensemble des tweets postés par un utilisateur dont l'id est passée en paramètre
def getUserTweets(api=None, user_id=None):
    print("==> "+str(datetime.datetime.now())+" getUserTweets, user_id=",user_id)
    try:
        tweets = api.GetUserTimeline(user_id=user_id, count=200)
        if not tweets:
            return []
        earliest_tweets = min(tweets, key=lambda x: x.id).id
        while True:
            tweets_partial = api.GetUserTimeline(
                user_id=user_id, max_id=earliest_tweets, count=200
            )
            if not tweets_partial :
                break
            new_earliest = min(tweets_partial, key=lambda x: x.id).id
            if new_earliest == earliest_tweets:
                break
            else:
                earliest_tweets = new_earliest
                print("getting tweets before:", earliest_tweets)
                tweets += tweets_partial
    except Exception as e:
        print(e)
        tweets = []
    return tweets

# sauvegarde l'ensemble des tweets postés par une liste d'utilisateurs dont les
# ids sont passés en paramètre dans des fichiers json (un fichier par utilisateur)
def dumpUsersTweets(str_ids=None):
    for str_id in str_ids:
        id=int(str_id)
        tweets = getUserTweets(api=api, user_id=id)
        if tweets and len(tweets)>=50:
            filename=output_directory+'/'+str_id+'.json'
            print("write file (",filename,")")
            with open(output_directory+'/'+str_id+'.json', 'w+') as f:
                nbTweets = len(tweets)
                f.write('[')
                for i in range(0, nbTweets-1):
                    f.write(json.dumps(tweets[i]._json))
                    f.write(',\n')
                f.write(json.dumps(tweets[nbTweets-1]._json))
                f.write('\n')
                f.write(']')
        else:
            print("Not enough tweets")

def main():
    if not os.path.isdir(output_directory) :
        os.mkdir(output_directory)
    dict_id_count = {}
    # lecture de la liste des ids d'utilisateurs dont on souhaite télécharger les tweets
    # ces ids ont été stockés dans une map (dictionary en python) de façon à éliminer les doublons
    with open('dict_id_count.json') as json_data:
        dict_id_count = json.load(json_data)
    dumpUsersTweets(str_ids=dict_id_count.keys())

if __name__== "__main__":
    main()
```

9.1.7 Extraction du texte des tweets

```
import os, json, datetime

directory_in ="tweets"
directory_out="tweets_text"

################################################################################
# Code permettant l'extraction du texte des tweets.
# A partir de chaque fichier json (contenant l'intégralité des tweets d'un utilisateur plus des informations connexes),
# on génère un fichier contenant uniquement le texte des messages
# (avec une ligne par message)
################################################################################

def main():
    if not os.path.isdir(directory_out) :
        os.mkdir(directory_out)
    print("Start text extraction : ", datetime.datetime.now())
    for filename in os.listdir(directory_in):
        if filename.endswith(".json"):
            data_out = []
            with open(directory_in+"/"+filename) as json_file_in:
                data_in = json.load(json_file_in)
                for elt_json_in in data_in:
                    # regroupe le texte du tweet sur une seule ligne
                    text=elt_json_in['text'].replace("\n", " ")+"\n"
                    data_out.append(text)
            with open(directory_out+"/"+filename.replace(".json",".txt"),"w") as text_file_out:
                text_file_out.writelines(data_out)
    print("End text extraction : ", datetime.datetime.now())

if __name__== "__main__":
    main()
```

9.2 Annexe 2 : analyse exploratoire

9.2.1 Termes les plus utilisés

Code Spark/Scala :

```
import com.cloudera.datascience.lsa._
import com.cloudera.datascience.lsa.ParseWikipedia._
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.Row

sc.setLogLevel("ERROR")

def wordCount(inputDirectory: String,  stopWords:Set[String],outputFile : String) = {
        // lecture, lemmatisation et supppresion des stopWords
        val tweetsRDD = spark.sparkContext.wholeTextFiles(inputDirectory).
                flatMap(l => l._2.split("\n").map(tweet => (l._1.split("/").last,tweet ) ) ).
                mapPartitions(iter => {
                        val pipeline = ParseWikipedia.createNLPPipeline();
                        iter.map{ case(user,tweet) =>
                             (user,ParseWikipedia.plainTextToLemmas(tweet, stopWords, pipeline).mkString(" "))};
                }).
                filter(t => t._2.size >= 1).
                cache()

        val tweetsDF = spark.createDataFrame(tweetsRDD).toDF("user","text")
        val wordsDF = tweetsDF.explode("text","word")((text: String) => text.split(" "))
        val wordCountDF = wordsDF.groupBy("word").count().sort($"count".desc)
        wordCountDF.show(50)
        wordCountDF.coalesce(1).write.format("csv").option("header", "true").csv(outputFile)
}

val stopWords = sc.broadcast(ParseWikipedia.loadStopWords("deps/lsa/src/main/resources/stopwords.txt")).value

wordCount("data/input/train/depressed",stopWords,"data/output/word_count_depressed")
wordCount("data/input/train/undepressed",stopWords,"data/output/word_count_undepressed")
```

Code R :

```
library("worldcloud")
library("RcolorBrewer")
library("ggplot2")

depressed <- read.csv(file="word_count_depressed.csv", header=TRUE, sep=",")

depressed <- depressed[1:250,]

#affichage du bar plot
ggplot(data=depressed[1:50,], aes(x=reorder(word,count), y=count, fill=word)) +
  geom_bar(colour="black", fill="#DD8888", width=.8, stat="identity") +
  coord_flip() +
  guides(fill=FALSE) +
  xlab("Word") + ylab("Count") +
  ggtitle("Depressed")


#affichage du word cloud
set.seed(1234)
wordcloud(words = depressed$word, freq = depressed$count, min.freq = 1,
          max.words=250, random.order=FALSE, rot.per=0.35,
          colors=brewer.pal(8, "Dark2"))


undepressed <- read.csv(file="word_count_undepressed.csv", header=TRUE, sep=",")

undepressed <- undepressed[1:250,]

#affichage du bar plot
ggplot(data=undepressed[1:50,], aes(x=reorder(word,count), y=count, fill=word)) +
  geom_bar(colour="black", fill="#88DD88", width=.8, stat="identity") +
  coord_flip() +
  guides(fill=FALSE) +
  xlab("Word") + ylab("Count") +
  ggtitle("Undepressed")


#affichage du word cloud
set.seed(1234)
wordcloud(words = undepressed$word, freq = undepressed$count, min.freq = 1,
          max.words=250, random.order=FALSE, rot.per=0.35,
          colors=brewer.pal(8, "Dark2"))
```

9.2.2 Recherche des thèmes principaux (LDA)

```
import com.cloudera.datascience.lsa._
import com.cloudera.datascience.lsa.ParseWikipedia._
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.Row
import org.apache.spark.ml.feature.{HashingTF, IDF, Tokenizer}
import org.apache.spark.ml.clustering.LDA
import org.apache.spark.ml.feature.{CountVectorizer, CountVectorizerModel}
import scala.collection.mutable.WrappedArray

sc.setLogLevel("ERROR")


def buildLDATopics(inputDirectory: String, stopWords:Set[String],outputFile:String) = {

        val tweetsRDD = spark.sparkContext.wholeTextFiles(inputDirectory).
                flatMap(l => l._2.split("\n").map(tweet => (l._1.split("/").last,tweet ) ) ).
                mapPartitions(iter => {
                        val pipeline = ParseWikipedia.createNLPPipeline();
                        iter.map{ case(user,tweet) =>
                             (user,ParseWikipedia.plainTextToLemmas(tweet, stopWords, pipeline))};
                }).
                filter(t => t._2.size >= 1).
                cache()

        // dataframe construit à partir de tweetsRDD
        val tweets = spark.createDataFrame(tweetsRDD).toDF("user","text")

        // création d'une matrice de type TF IDF

        // 1) calcul TF avec CountVectorizer
        val cvModel: CountVectorizerModel = new CountVectorizer()
                                .setInputCol("text")
                                .setOutputCol("rawFeatures")
                                .setVocabSize(2000)
                                .setMinDF(2)
                                .fit(tweets)



        // extraction du vocabulaire (tableau de String) à partir du CountVectorizeModel
        val vocab = cvModel.vocabulary

        val tweetsTF= cvModel.transform(tweets)

        // 2) calcul IDF
        val idfModel = new IDF()
                        .setInputCol("rawFeatures")
                        .setOutputCol("features")
                        .fit(tweetsTF)


        val tweetsTFIDF = idfModel.transform(tweetsTF)

        tweetsTFIDF.show()

        // création du modèle LDA avec 10 topics
        val nbTopics=10

        val lda = new LDA().setK(nbTopics).setMaxIter(10)
        val model = lda.fit(tweetsTFIDF)

        // les mots définissant les topics sont représentés par un tableau d'entier correspondant
        // à leurs indices dans le vocabulaire
        // la fonction toWords reconstitue la liste de mots à partir des indices et la place dans une String
        val toWords = udf( (x : WrappedArray[Int]) => { x.map(i => vocab(i)).mkString(", ") })
        val topics = model.describeTopics(nbTopics)
                .withColumn("topicWords", toWords(col("termIndices")))


        val topicsWords = topics.select("topicWords")
        topicsWords.show()
        topicsWords.coalesce(1).write.format("csv").option("header", "true").csv(outputFile)
}

val stopWords = sc.broadcast(ParseWikipedia.loadStopWords("deps/lsa/src/main/resources/stopwords.txt")).value

buildLDATopics("data/input/train/depressed",stopWords,"data/output/topicsDepressed.csv")
buildLDATopics("data/input/train/undepressed",stopWords,"data/output/topicsUndepressed.csv")
```

9.2.3 Analyse de sentiment (SparkNLP)

```
import com.cloudera.datascience.lsa._
import com.cloudera.datascience.lsa.ParseWikipedia._
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.Row
import org.apache.spark.ml.Pipeline
import com.johnsnowlabs.nlp.pretrained.PretrainedPipeline
import com.johnsnowlabs.nlp.Finisher
import scala.collection.mutable.WrappedArray

sc.setLogLevel("ERROR")


def evaluateSentiment(inputDirectory: String, stopWords:Set[String], outputFile:String) = {

        val tweetsRDD = spark.sparkContext.wholeTextFiles(inputDirectory).
                flatMap(l => l._2.split("\n").map(tweet => (l._1.split("/").last,tweet ) ) ).
                mapPartitions(iter => {
                        val pipeline = ParseWikipedia.createNLPPipeline();
                        iter.map{ case(user,tweet) =>
                             (user,ParseWikipedia.plainTextToLemmas(tweet, stopWords, pipeline).mkString(" "))};
                }).
                filter(t => t._2.size >= 1).
                cache()

        // dataframe construit à partir de tweetsRDD
        val tweetsDF = spark.createDataFrame(tweetsRDD).toDF("user","text")
        tweetsDF.show()

        // L'analyse de sentiment produite par SparkNLP (colonne "finished_sentiment")
        // est un tableau (avec un résultat par phrase du texte analysé)
        // Cela pose problème pour l'export des résultats au format CSV.
        // On convertit donc le tableau en String avec la fonction ci-dessous
        // NB: dans notre cas les signes de ponctuation sont supprimés lors de la lemmatisation
        // chaque tweet est donc traité par SparkNLP comme une phrase unique et le tableau
        // ne comporte qu'un seul élément
        val fromArrayToString = udf( (x : WrappedArray[String]) => { x.mkString(", ") })


        val sentimentPipelineModel = PretrainedPipeline("analyze_sentiment").model
        val finisherSentiment = new Finisher().setInputCols("document","sentiment")
        val pipelineSentiment = new Pipeline().setStages(Array(sentimentPipelineModel,finisherSentiment))
        val modelSentiment = pipelineSentiment.fit(tweetsDF)
        val sentimentTweetsDF = modelSentiment.
                                transform(tweetsDF).
                                select("text","finished_sentiment").
                                withColumn("sentiment", fromArrayToString(col("finished_sentiment"))).
                                select("text","sentiment")

        sentimentTweetsDF.show()
        val sentimentCountDF = sentimentTweetsDF.groupBy("sentiment").count().sort($"count".desc)

        sentimentCountDF.show()

        sentimentTweetsDF.coalesce(1).write.format("csv").option("header", "true").csv(outputFile)
}

val stopWords = sc.broadcast(ParseWikipedia.loadStopWords("deps/lsa/src/main/resources/stopwords.txt")).value

// On ne garde pour chaque utilisateur que les 50 derniers tweets (répertoire "input_truncated"):
// cela diminue le temps de traitement et évite que les sentiments des utilisateurs ayant posté
// de nombreux messages soient sur représentés
evaluateSentiment("data/input_truncated/train/depressed",stopWords,"data/output/sentimentAnalysisDepressed.csv")
evaluateSentiment("data/input_truncated/train/undepressed",stopWords,"data/output/sentimentAnalysisUndepressed.csv")
```

9.3 Annexe 3 : prétraitement

9.3.1 Création d'un modèle Word2vec

Le modèle Word2vec est construit à partir des fichiers de messages Twitter crées lors de la phase d'acquisition et d'étiquetage des données. Ces fichiers sont répartis dans deux répertoires, un pour le groupe « depressed », un pour le groupe « undepressed ».

Le script unix suivant permet de fusionner les fichiers de chaque répertoire :

```
cat undepressed/*.txt > undepressed.txt
cat depressed/*.txt > depressed.txt
```

La création du modèle Word2vec est alors réalisée avec le traitement Spark/Scala ci-dessous:

```
import com.cloudera.datascience.lsa._
import com.cloudera.datascience.lsa.ParseWikipedia._
import org.apache.spark.ml.feature.Word2Vec

val dataDir="data/input/"

// lecture des tweets du goupe "depressed"
val tweetsDepressed=sc.textFile(dataDir+"depressed.txt")
// lecture des tweets du goupe "undepressed"
val tweetsUndepressed=sc.textFile(dataDir+"undepressed.txt")

// fusion de l'ensemble des tweets en  1 seul RDD : tweetsAll
val tweetsAll=tweetsDepressed.union(tweetsUndepressed)

val stopWords = sc.broadcast(ParseWikipedia.loadStopWords("deps/lsa/src/main/resources/stopwords.txt")).value

// création du RDD corpus en lemmatisant les tweets
val corpus = tweetsAll.mapPartitions(iter => {
           val pipeline = ParseWikipedia.createNLPPipeline();
           iter.map{ tweet  =>  ParseWikipedia.plainTextToLemmas(tweet, stopWords, pipeline)};
       })

// transformation du RDD corpus en dataframe
val corpusDF = corpus.toDF("text")

// création du modèle Word2vec à partir de corpus
val word2Vec = new Word2Vec().setInputCol("text").setOutputCol("features").setVectorSize(100).setMinCount(0)
val w2vModel = word2Vec.fit(corpusDF)

// enregistrement du modèle Word2vec
w2vModel.save("data/output/w2vModel")
```

9.3.2 Lemmatisation, suppression des stop words et application du modèle Word2vec

```
import org.apache.spark.ml.feature.Word2Vec
import org.apache.spark.ml.feature.Word2VecModel
import com.cloudera.datascience.lsa._
import com.cloudera.datascience.lsa.ParseWikipedia._
import org.apache.spark.ml.stat.Summarizer
import org.apache.spark.sql.Dataset
import org.apache.spark.sql.Row

sc.setLogLevel("ERROR")

def buildDataframe(inputDirectory: String,  stopWords:Set[String],w2vModel: Word2VecModel,label:Double) : Dataset[Row] = {

        // lecture des tweets, suppression de stop words et lemmatisation
        val tweetsRDD = spark.sparkContext.wholeTextFiles(inputDirectory).
                flatMap(l => l._2.split("\n").map(tweet => (label,l._1.split("/").last,tweet ) ) ).
                mapPartitions(iter => {
                        val pipeline = ParseWikipedia.createNLPPipeline();
                        iter.map{ case(label,user,tweet) =>
                             (label,user,ParseWikipedia.plainTextToLemmas(tweet, stopWords, pipeline))};
                }).
                filter(t => t._3.size >= 1).
                cache()

        val tweetsDF = spark.createDataFrame(tweetsRDD).toDF("label","user","text")

        // application du modèle word2vec au texte :
        // ceci se traduit par la création d'un nouveau dataframe
        // comportant une colonne "features"
        // représentant le texte sous forme vectorielle
        val tweetsFeaturesDF = w2vModel.transform(tweetsDF).
                                        groupBy($"label",$"user").
                                        agg(Summarizer.mean($"features").alias("features"))

        return tweetsFeaturesDF
}

val stopWords = sc.broadcast(ParseWikipedia.loadStopWords("deps/lsa/src/main/resources/stopwords.txt")).value
val w2vModel = Word2VecModel.load("data/output/w2vModel")

val trainDepressedDF=buildDataframe("data/input/train/depressed",stopWords,w2vModel,1.0)
val trainUndepressedDF=buildDataframe("data/input/train/undepressed",stopWords,w2vModel,0.0)
val trainDF = trainDepressedDF.union(trainUndepressedDF)
trainDF.write.parquet("data/output/trainDF")


val testDepressedDF=buildDataframe("data/input/test/depressed",stopWords,w2vModel,1.0)
val testUndepressedDF=buildDataframe("data/input/test/undepressed",stopWords,w2vModel,0.0)
val testDF = testDepressedDF.union(testUndepressedDF)
testDF.write.parquet("data/output/testDF")
```

9.4 Annexe 4 : création et validation d'un modèle supervisé

9.4.1 Création d'un modèle supervisé (régression logistique)

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.ml.classification.LogisticRegression
import org.apache.spark.ml.{Pipeline, PipelineModel}
import org.apache.spark.ml.tuning.{CrossValidator, ParamGridBuilder}
import org.apache.spark.ml.evaluation.BinaryClassificationEvaluator
import org.apache.spark.ml.feature.StandardScaler
import org.apache.spark.ml.param.ParamMap

// dataPath      : emplacement du fichier contenant les données
// logReg        : modele de regression logistique à entrainer
// paramGrid     : grille de (hyper)paramètres utilisée pour grid search
// cvLogRegModelPath: emplacement du fichier ou sera sauvegardé le modèle de régression logistique
def logRegTrain(dataPath: String, logReg: LogisticRegression, paramGrid: Array[ParamMap], cvLogRegModelPath: String) = {
    // Chargement des données d'apprentissage
    val sqlContext = new SQLContext(sc)
    val trainDF = sqlContext.read.parquet("data/output/trainDF")
    val count = trainDF.count()


    // Définition du scaler pour centrer les variables initiales
    val scaler = new StandardScaler().
                            setInputCol("features").setOutputCol("scaledFeatures").setWithStd(false).setWithMean(true)

    // Définition du pipeline (scaler puis régression logistique)
    val pipeline = new Pipeline().setStages(Array(scaler,logReg))

    // Définition de l'instance de CrossValidator : à quel estimateur l'appliquer,
    //  avec quels (hyper)paramètres, combien de folds, comment évaluer les résultats
    val cv = new CrossValidator().setEstimator(pipeline).setEstimatorParamMaps(paramGrid).setNumFolds(5).
                            setEvaluator(new BinaryClassificationEvaluator())

    // Construction et évaluation par validation croisée des modèles correspondant
    //   à toutes les combinaisons de valeurs de (hyper)paramètres de paramGrid
    val cvLogRegModel = cv.fit(trainDF)

    // Afficher les meilleures valeurs pour les (hyper)paramètres
    println("PARAMS : "+cvLogRegModel.getEstimatorParamMaps.zip(cvLogRegModel.avgMetrics).maxBy(_._2)._1)

    // Enregistrement du meilleur modèle
    cvLogRegModel.write.save(cvLogRegModelPath)

    // Calculer les prédictions sur les données d'apprentissage
    val resTrain = cvLogRegModel.transform(trainDF)

    // Calculer et afficher AUC sur données d'apprentissage
    println("AUC : "+ cvLogRegModel.getEvaluator.evaluate(resTrain))

    // Calculer la précision
    resTrain.createOrReplaceTempView("resTrain")
    spark.sql("SELECT count(*)/"+count+ " FROM resTrain  WHERE label == prediction").show()
}

// Initialisation du modèle de régression logistique
val logReg = new LogisticRegression().setFeaturesCol("scaledFeatures").setLabelCol("label")

// Construction de la grille de (hyper)paramètres utilisée pour grid search
// Une composante est ajoutée avec .addGrid() pour chaque (hyper)paramètre à explorer
val paramGrid = new ParamGridBuilder().
                            addGrid(logReg.elasticNetParam, Array(0.0,  0.2,  0.4, 0.6, 0.8, 1)).
                            addGrid(logReg.maxIter, Array(50,100,150,200,250)).
                            addGrid(logReg.regParam, Array(0.01,  0.1,  0.5)).build()


// Apprentissage du modèle
logRegTrain("data/output/trainDF", logReg, paramGrid,"data/output/cv_logreg_model")
```

9.4.2 Validation du modèle

```
import org.apache.spark.sql.SQLContext
import org.apache.spark.ml.tuning.CrossValidatorModel


// dataPath      : emplacement du fichier contenant les données
// cvLogRegModelPath: emplacement du fichier contenant le modèle de régression logistique
//                    obtenu par validation croisée
def logRegTest(dataPath: String, cvLogRegModelPath: String) = {
    // Chargement des données de test
    val sqlContext = new SQLContext(sc)
    val testDF = sqlContext.read.parquet(dataPath)
    val count = testDF.count()

    // Chargement du modèle SVM
    val cvLogRegModel =  CrossValidatorModel.load(cvLogRegModelPath)

    // Calculer les prédictions sur les données de test
    val resultats = cvLogRegModel.transform(testDF)

    // Afficher 10 lignes complètes de résultats (sans la colonne features)
    // la colonne "probability" contient un vecteur :
    // [probabilité label = 0 (undepressed), probabilité label = 1 (depressed)]
    resultats.select("label", "probability", "prediction").show(10, false)

    // Calculer et afficher AUC sur données de test
    println("AUC : "+cvLogRegModel.getEvaluator.evaluate(resultats))

    // Calculer et afficher la précisions
    resultats.createOrReplaceTempView("resultats")
    spark.sql("SELECT count(*)/"+count+ " FROM resultats  WHERE label == prediction").show()

    //Afficher un résumé détaillé
   resultats.summary().show()
}

// Evaluation du mdèdle SVM avec les données de test
logRegTest("data/output/testDF", "data/output/cv_logreg_model")
```

9.5 Annexe 5 : application du modèle sur un flux Twitter

9.5.1 Récupération des données dans un flux Twitter

```
import os, socket, json, sys, twitter, tweepy
from tweepy import OauthHandler, Stream
from tweepy.streaming import StreamListener

#############################################################################################################
# Code permettant :
# 1) de détecter dans un flux Twitter les utilisateurs postant un message contenant le terme 'depression'
# 2) d'exporter les 50 derniers tweets de ces utilisateurs
# 3) de notifier Spark qu'une classification "déprimé/non déprimé" doit être effectuée
#############################################################################################################

# Listener du flux twitter
class TweetsListener(StreamListener):

        def set_twitter_api(self, twitter_api):
                self.twitter_api = twitter_api

        # répertoire scruté par Spark Streaming
        # les fichier de ce répertoire contiennent uniquement l'id d'un utilisateur sur lesquels on souhaite
        # effectuer la classification
        # la création d'un fichier dans ce répertoire, détecté par Spark Streaming, déclenche la classification
        def set_notifications_directory(self, notifications_directory):
                self.notifications_directory = notifications_directory

        # répertoire contenant les tweets d'utilisateurs sur lesquels la classification doit être appliquée
        def set_tweets_directory(self, tweets_directory):
                self.tweets_directory = tweets_directory

        # cette fonction est appelée quand un utilisateur twitter poste un message contenant le terme "depression"
        # cf. filtrage du flux plus bas : "twitter_stream.filter(languages=["en"], track=['depression'])"
        def on_status(self, status):
                try:
                        user_id = str(status.user.id)
                        print(user_id)
                        tweets = self.twitter_api.GetUserTimeline(user_id=user_id, count=50)
                        # on ne prend en compte que les utilisateurs ayant posté au moins 50 messages
                        if tweets and len(tweets)==50 :
                                data_out = []
                                for tweet in tweets:
                                        text = tweet._json['text'].replace("\n", " ")+"\n"
                                        data_out.append(text)
                                # création dans le répertoire 'tweets' d'un fichier :
                                # 1) dont le nom correspond à l'id de l'utilisateur ayant posté le tweet
                                # 2) contenant les textes des 50 derniers messages postés par l'utilisateur
                                with open(self.tweets_directory+"/"+user_id+".txt", 'w', encoding='utf-8') as text_file_out:
                                        text_file_out.writelines(data_out)
                                # création dans le répertoire 'notifications' d'un fichier contenant uniquement
                                # l'id de l'utilisateur pour avertir Spark qu'une classification doit être effectuée
                                with open(self.notifications_directory+"/"+user_id, 'w', encoding='utf-8') as f:
                                        f.write(user_id)
                except Exception as e:
                        print(e)

        def on_error(self, status):
                print(status)
                return True

# initialisation du traitement du flux twitter
def captureTweets():
        notifications_directory = "notifications"
        if not os.path.isdir(notifications_directory) :
                os.mkdir(notifications_directory)
        tweets_directory = "tweets"
        if not os.path.isdir(tweets_directory) :
                os.mkdir(tweets_directory)

        #authentification
        auth = OAuthHandler(consumer_key, consumer_secret)
        auth.set_access_token(access_token, access_secret)
        # initialisation du listener
        tweetsListener = TweetsListener()
        tweetsListener.set_twitter_api(twitter.Api(consumer_key="XXXXX", consumer_secret="XXXXX",
                access_token_key="XXXXX",access_token_secret="XXXXX",sleep_on_rate_limit=True))
        tweetsListener.set_notifications_directory(notifications_directory)
        tweetsListener.set_tweets_directory(tweets_directory)
        twitter_stream = Stream(auth,tweetsListener)
        twitter_stream.filter(languages=["en"], track=['depression'])

if __name__ == "__main__":
        captureTweets( )
```

9.5.2 Traitement des données avec Spark Streaming

```
import org.apache.spark.ml.feature.Word2Vec
import org.apache.spark.ml.feature.Word2VecModel
import com.cloudera.datascience.lsa._
import com.cloudera.datascience.lsa.ParseWikipedia._
import org.apache.spark.ml.stat.Summarizer
import org.apache.spark.sql.SQLContext
import org.apache.spark.ml.tuning.CrossValidatorModel
import org.apache.spark._
import org.apache.spark.streaming._
import org.apache.spark.streaming.StreamingContext._

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
// Code Spark  qui:
// 1) scrute (avec Spark Streaming) le répertoire "notifications" dans lequel apparaissent les fichiers contenant
//   les identifiants d'utilisateurs à traiter
// 2) lit, dans le répertoire "tweets", le fichier contenant le texte des messages de l'utilisateur à traiter
// 3) à partir des messages postés par l'utilisateur applique la classification "deprime/non deprime"
// 4) écrit le résultat dans un fichier csv
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////


// Effectue la classification sur un utilisateur dont l'identifiant (user_id) est passe en paramètre.
// La classification est basee sur le texte des 50 derniers tweets de l'utilisateur.
// Les tweets sont stockes dans le fichier "tweets/[user_id].txt".
// Le resultat de la classification est stocke dans dans le fichier "resultats/[user_id].csv".
def processUserTweets(user_id:String,stopWords:Set[String],w2vModel: Word2VecModel, cvLogRegModel: CrossValidatorModel) = {
  val path = "tweets/".concat(user_id).concat(".txt")

  val tweetsRDD = spark.sparkContext.textFile(path).
                flatMap(text => text.split("\n").map(tweet => (user_id,tweet ) ) ).
                mapPartitions(iter => {
                        val pipeline = ParseWikipedia.createNLPPipeline();
                        iter.map{ case(user,tweet) =>
                             (user,ParseWikipedia.plainTextToLemmas(tweet, stopWords, pipeline))};
                }).
                filter(t => t._2.size >= 1).
                cache()

  val tweetsDF = spark.createDataFrame(tweetsRDD).toDF("user","text")


  val tweetsFeaturesDF = w2vModel.transform(tweetsDF).
                                      groupBy($"user").
                                      agg(Summarizer.mean($"features").alias("features"))


  val resultats = cvLogRegModel.transform(tweetsFeaturesDF)
                               .select("user", "prediction")
        resultats.show()

  resultats.coalesce(1).write.format("csv").option("header", "true")
                .csv("resultats/".concat(user_id).concat(".csv"))
}


val w2vModel = Word2VecModel.load("data/output/w2vModel")
val stopWords = sc.broadcast(ParseWikipedia.loadStopWords("deps/lsa/src/main/resources/stopwords.txt")).value
// lecture du modèle de classification (regression logistique)
val cvLogRegModel =  CrossValidatorModel.load("data/output/cv_logreg_model")

// création du flux avec une fenêtre de 10 secondes
val ssc = new StreamingContext(sc, Seconds(10))

// scrute dans le répertoire "notifications" l'apparition de fichiers
// chaque fichier de notification contient l'identifiant d'un utilisateur Twitter à traiter
val lines = ssc.textFileStream("notifications")
lines.foreachRDD(rdd => {
  if ((rdd != null) && (rdd.count() > 0) && (!rdd.isEmpty()) ) {
    // déclenche une classification pour chaque identifiant utilisateur lu dans les fichiers de notification
    rdd.collect().foreach(user_id => processUserTweets(user_id,stopWords,w2vModel,cvLogRegModel))
  }
})

// démarrage du traitement du flux
ssc.start()
ssc.awaitTermination()
```

9.6 Annexe 6 : passage à l'échelle

9.6.1 Configuration des nœuds HDFS

La configuration intégrale d'un système HDFS peut s'avérer complexe. Nous présentons ici deux des principaux fichiers de configuration, présents sur chaque nœud du cluster, dans le répertoire hadoop/etc :

- *core-site.xml* (fichier spécifiant l'hôte et le port du système HDFS ainsi que l'emplacement des données temporaires)

```
<configuration>
    <property>
        <name>fs.defaultFS</name>
        <value>hdfs://master.local:9000</value>
    </property>
    <property>
        <name>hadoop.tmp.dir</name>
        <value>/home/hadoop/hadooptmpdata</value>
    </property>
</configuration>
```

- *hdfs-site.xml* (fichier spécifiant le **coefficient de réplication**, l'emplacement du répertoire où le NameNode va stocker l'historique des transactions et où les DataNode vont stocker leurs blocs)

```
<configuration>
    <property>
        <name>dfs.replication</name>
        <value>2</value>
    </property>
    <property>
        <name>dfs.name.dir</name>
        <value>/hadoop/hdfs/namenode</value>
    </property>
    <property>
        <name>dfs.data.dir</name>
        <value>/hadoop/hdfs/datanode</value>
    </property>
</configuration>
```

9.6.2 Lecture / écriture dans HDFS

Nous avons mis en œuvre des scripts python pour récupérer les messages Twitter. Il est donc naturel d'utiliser une bibliothèque du même langage pour enregistrer ces messages dans un système HDFS. Plusieurs options existent, une des plus utilisées est le package https://pypi.org/project/hdfs/. Nous présentons ci-dessous un exemple illustrant les fonctions de lecture / écriture avec cette API.

```
import hdfs

client = hdfs.InsecureClient("http://master.local:50070")


# Crée un répertoire sous la racine HDFS
client.makedirs("/twitter")

# Ecrit un fichier texte ligne à ligne
with client.write("/twitter/tweets_1234.txt", overwrite=True) as writer:
     #  ATTENTION : l'appel à "write()" prend en entrée des bytes et non une string
    writer.write(b"message1\n ")
    writer.write(b"message2\n ")

# Affiche le contenu du répertoire twitter
client.list("/twitter")

# Lit un fichier ligne à ligne
with client.read("/tweets_1234.txt") as reader:
    for line in reader:
        print(line)
```

NB : dans notre projet les scripts python sont utilisés pour acquérir les données. Le traitement analytique lui est réalisé avec Spark, qui dès le départ a été créé pour fonctionner avec HDFS. La seule différence de code entre la lecture d'un fichier stocké sur disque local par rapport à la lecture de ce même fichier stocké sur un système Hadoop avec Spark, réside dans le chemin du fichier :

```
// lecture dans HDFS
val tweets1234DS = spark.read.textFile("hdfs://master.local:9000/twitter/tweets_1234.txt")
```

9.7 Annexe 7 : logiciels utilisés

Logiciel	Version
Système d’exploitation : Ubuntu Desktop 64-bit	19.10
Java Oracle (JDK et JRE)	8u31
Spark	2.4.4
Scala	2.12.10
Python	3.7.4
Elasticsearch	7.5
R	3.6.1
Rstudio	1.2