Publié par Aurore De Amaral et Jonathan Raffre

Big Data NLP scala

Il y a 1 an -Temps de lecture 20 minutes

Exploration de données textuelles en Scala

Ce matin, en prenant son café, Nina (nom anonymisé), data ingénieur de métier, aimerait rendre le partage de code plus efficace entre les data scientists et elle. Or, Nina utilise Spark en Scala et ses collègues Python (ou SQL, R...), pour leurs analyses de tweets, ce qui complique la communication et la transformation des règles métiers... Elle aimerait donner des outils avec le langage qu'elle utilise, pour faciliter la coopération et prendre en compte le langage cible en production dès la conception, qui est Scala dans notre exemple.

Rappel du billet précédent

Dans le précédent billet, nous avons vu comment explorer efficacement en Scala. L'exploration de données est une étape préliminaire à tout projet de Data Science. Il est effectué par un professionnel de la data, afin de vérifier l'état des données, valider des hypothèses ou inférer des corrélations et ainsi donner une visualisation rapide au métier. Nous avons vu les principaux outils pour nettoyer la donnée, la filtrer, la visualiser. En effet, l'exploration de données vise trois buts bien définis :

- charger/transformer les données ;
- comprendre la donnée et tenter d'en extraire ou créer des *features* (caractéristiques de la donnée) au besoin ;
- visualiser ces *features.*

Nous avons décidé de choisir la librairie Spark et l'outil Zeppelin pour répondre à tous ces objectifs. Ce précédent billet se confrontait à de l'exploration de données tabulaires, avec un schéma déjà défini. Mais l'article qui suit va se pencher sur des données non structurées et plus précisément sur du texte en langage naturel.

Deuxième exemple

Le traitement du langage naturel

Le traitement (automatique) du langage naturel (appelé **TAL** ou plus communément **NLP**, pour son acronyme anglais correspondant à *Natural Language Processing*) est une tâche qui s'articule autour de la donnée texte visant à faire comprendre des énoncés humains à la machine. Plus globalement, elle sert à transformer du **texte** en **informations**. Nous n'allons pas rentrer dans la définition en détail du sujet, ce billet présuppose que les personnes veulent simplement s'approprier des outils en Scala pour leurs usages de TAL. Et Nina le sait déjà. Néanmoins, nous passerons brièvement sur quelques définitions précises permettant la compréhension du code que nous exposerons en exemple.

Avec quel outils ?

Nina compte utiliser des outils clés en main, comme l'équivalent moderne *SpaCy* ou le très reconnu *NLTK* en Python. Le but est de trouver des outils qui font les tâches communes de TAL en Scala. Et qui auraient des modèles pré-entraînés, pour une exploration rapide. Elle en a trouvé des dizaines, mais en voici cinq qui sont mûrs pour des tâches exploratoires :

- Spark ML est la librairie de *machine learning* de Spark, qui recouvre quelques tâches sommaires de TAL comme la tokenisation ou les *n-grammes* de mots. Elle possède bien évidemment plus de *features* orientées *machine learning* que ne possèdent pas toujours les autres.
- Stanford CoreNLP est une des librairies Java les plus répandues dans le domaine du TAL. Maintenue par l'université de Stanford et de nombreux contributeurs enthousiastes, elle englobe de nombreuses tâches et supporte beaucoup de langages (dont le français).
- Spark NLP est une librairie Scala qui permet d'enrichir Spark avec des fonctionnalités de traitement du langage et des modèles pré-entraînés en anglais. Elle a donc toutes les qualités de Spark ML avec de nombreuses tâches de TAL.
- Deeplearning4J est une librairie Java pour le *deep learning*. Elle n'a pas beaucoup d'éléments pour le TAL : citons la tokenisation, le sac de mots et la vectorisation.
- Apache Open NLP est une librairie Java créée par la fondation Apache qui contient de nombreuses tâches de TAL et permet d'importer facilement ses modèles, mais n'en procure pas beaucoup par défaut, et pas encore en français.

Les règles et modèles nécessaires à une bonne efficacité du TAL nécessitent beaucoup d'investissement et de données, incitant les entreprises et certaines universités à ne pas partager ce travail, ce qui explique le peu de diversité de ces modèles dans la communauté *open source*.

Son choix s'est porté sur *Spark ML* et *Stanford CoreNLP* associé avec le *wrapper* CoreNLP de Databricks, qui couvre la Simple API de Stanford CoreNLP. La librairie Stanford CoreNLP et le wrapper seront cités par la suite comme « CoreNLP ». Ces trois librairies sont complémentaires et il faut souvent combiner une librairie de *machine learning* avec une librairie de TAL pour faciliter l'exploration de données. Cela permet d'embarquer facilement une première phase exploratoire et une phase de *feature engineering*. Celle-ci sera souvent gérée par des modèles maison en production.
De plus, son choix s'est porté en priorité sur une solution contenant déjà des modèles pré-entraînés sur du texte français, ce qui est le cas de la librairie Stanford CoreNLP.

N.B.: Dû à la licence GPLv3 de la librairie Stanford CoreNLP, nous tenons à vous prévenir que son utilisation à des fins commerciales peut être restrictive dans certains cas (embarqué, matériels à destination d'utilisateurs finaux...).

Exemple traité

Pour l'exemple, nous allons traiter un livre en français, tiré d'un livre disponible sur le site du projet Gutenberg.
Le but est aussi de voir des types de texte étant rédigés dans un style différent de la littérature, le texte choisi étant assez proche du langage parlé (mais du siècle dernier !), puisqu'il compile plusieurs lettres de poilus à leur famille.
Attention toutefois, cet exemple ne représente pas forcément le type de données que l'on peut avoir en entreprise (anciennes, contexte particulier).

Analyse du texte « brut » et création des phrases

Nina a donc reçu pour objectif de faire une analyse globale des lettres des poilus, pour en tirer potentiellement des tendances, comme le champ sémantique d'un corpus de cette époque.

Spark va donc l'assister dans sa tâche, celui-ci permettant de charger des textes.

Penchons-nous rapidement sur la structure du texte : un texte n'étant pas une donnée structurée, il va falloir la structurer. En général, on découpe le texte en phrases. Trois types de format textuel représentent des phrases :

1. Le texte possède des retours à la ligne (une ou plusieurs) qui découpent le texte en phrases (solution la plus structurée)
2. Le texte possède des retours à la ligne (une ou plusieurs) qui découpe un groupe de phrases (généralement, des paragraphes, solution structurée)

3. Le texte peut posséder des retours à la ligne mais n'apporte aucune information sur la formation des phrases (textes balisés ou retours non significatifs, solution non structurée)

Le but maintenant est de savoir comment charger ces différentes formes de texte. La première solution est assez simple à appliquer.

```
import org.apache.spark.sql.SparkSession
import spark.implicits._
import org.apache.spark.sql.functions._

val sc: SparkContext
val spark = SparkSession.builder
  .master("local")
  .appName("data-exploration")
  .getOrCreate()

val sentences = spark.read.textFile("s3a://jra-
xebia/datasets/lettres_poilus.txt").as[String]
    .map(_.trim())
    .filter(_.nonEmpty)
    .withColumn("id", monotonically_increasing_id())
    .orderBy("id")
```

Toutefois, ce qu'a Nina n'est pas structuré ainsi. En effet, en voici un extrait :

```
Avoir vécu trente-trois ans avec l'angoisse de ne pas voir venir le
jour
de gloire tant rêvé, avec l'humiliation de transmettre aux enfants la
honte d'être des Français diminués, moins fiers, moins libres que
leurs
grands-pères, avoir souffert de cela silencieusement, mais
profondément,
avec toute l'élite de mon pays, et voir soudain resplendir l'aube de
la
résurrection alors que je suis encore jeune et fort et que mon sang
est
prêt à jaillir, heureux, pour tous les sacrifices.

Je suis satisfait d'avoir été utile et même nécessaire à Nancy dans un
moment difficile, où les événements n'auraient pas eu le même
caractère
```

```
si mes fonctions avaient été détenues par un homme ayant moins de
sang-froid et d'esprit de décision. J'aurais été affecté s'il m'avait
fallu quitter Nancy, moins d'un mois après mon arrivée, alors que le
danger était grand et que j'avais beaucoup à faire.
```

Elle va donc devoir ruser un peu, le texte n'ayant pas de retours à la ligne significatifs, ce qui est le troisième cas, et découper notre texte avec la librairie CoreNLP.

```
val data = spark.read.textFile("s3a://jra-
xebia/datasets/lettres_poilus.txt").as[String]
    .map(_.trim())
    .filter(_.nonEmpty)
    .withColumnRenamed("value", "text")
    .withColumn("filename", lit("lettres_poilus.txt"))

val globalData = data.groupBy("filename")
    .agg(
        concat_ws(" ", collect_list(data("text"))).as("text")
    )
    .select(explode(ssplit(col("text"))).as("sentences"))
    .withColumn("id", monotonically_increasing_id())
    .orderBy("id")
```

Pour le second cas, nous nous basons sur la première méthode, mais il faut effectuer une seconde application pour transformer les paragraphes en phrase.

Avec le code ci-dessus, Nina possède maintenant un tableau de phrases (qui est un `DataFrame` Spark), et pour avoir une idée de celles-ci, on peut utiliser `df.sample()` qui va en donner un échantillon.

```
globalData.sample(false, 0.0034).show()

+--------------------------------------------------------------------
--------------------------------------------------------------------
-------------------------------------------------+------------+
|sentences
|id          |
+--------------------------------------------------------------------
```

```
--------------------------------------------------------------------------
-------------------------------------------------+------------+
|A mes camarades, je demande de croire avec quelle fierté je me suis
trouvé parmi eux et quelle affection j'avais vouée à notre cher
régiment.                                                |128849019086|
|C'est toi-même, C'est toi dans le petit chez nous....
|128849019196|
|J'ai eu des heures de découragement et de lassitude.
|128849019300|
|Vous trouveriez également dans mes papiers une sorte de testament qui
ne ferait que développer ce que je vous ai dit plus haut en une ligne.
|128849019348|
|espérons que, dans les mois à venir, nous nous amuserons bien de
cette lettre.
|128849019468|
|Adieu, cher père, je vous embrasse tous du plus profond de mon coeur
et surtout ma maman chérie.
|128849019509|
|Je finis mon séjour le 22 Juin prochain.
|128849019900|
|Et vous, quelle est votre déduction de tout cela?
|128849020218|
|_Lettre écrite à sa soeur par le Sergent Jacques-Etienne-Benoist DE
LAUMONT, du 66e Régiment d'Infanterie, tombé au champ d'honneur, le 25
Septembre 1915, à Agny-les-Arras.                |128849020358|
|Tu retourneras dans ton pays, en Alsace redevenue française, et tu te
diras si tu es à Thann ou à Strasbourg, c'est que tes fils auront
contribué à rendre à la France nos chères provinces.|128849021112|
+--------------------------------------------------------------------------
--------------------------------------------------------------------------
-------------------------------------------------+------------+
```

Premier nettoyage des phrases

Le texte possède quelques phrases en anglais, que Nina aimerait supprimer car elles ne font pas parties du texte originel (entêtes du projet Gutenberg). On voit aussi que les phrases commencent par une lettre majuscule. Ce sont des informations peu pertinentes après découpage en phrase. Nina va retirer les majuscules avec une `udf` (User Defined Function), fonction personnalisée qui permet de définir des transformations en fonction de son besoin.

```
val lower_first = udf((sentence: String) => {
    if (sentence == null || sentence.length == 0 ||
!sentence.charAt(0).isUpper) sentence
    else {
        val sb = new StringBuilder(sentence.length).append(sentence)
        sb.setCharAt(0, sentence.charAt(0).toLower)
        sb.toString
    }
})

val english_sentences =
globalData.filter(col("sentences").contains("the")) // solution naïve,
le mieux est d'utiliser TOUS les stopwords anglais

val datas = globalData.except(english_sentences).select(col("id"),
lower_first(col("sentences")).as("sentences"))
```

Analyse et extraction de feature : les tokens

Une phase importante en exploration de données textuelles est d'avoir un aperçu des tokens, *bigrammes* et de leur distribution au sein du corpus, ce qui donnerait à Nina une première idée des sujets que contiennent les lettres. Elle pourra ainsi faire ressortir un champ lexical de son corpus, repérer des synonymes, antonymes etc. Il faut donc commencer par découper en token chacune des phrases, en utilisant la librairie CoreNLP..

```
import com.databricks.spark.corenlp.functions._ //utilisé dans les
prochains appels à CoreNLP

val withTokens = datas.withColumn("tokens",
tokenize(col("sentences")))

withTokens.show(5)

+------------+--------------------+--------------------+
|          id|           sentences|              tokens|
+------------+--------------------+--------------------+
|128849019057|pour toi, ma chèr...|[pour, toi, ,, ma...|
|128849019081|a MES PARENTS Si ...|[a, MES, PARENTS,...|
|128849019200|     il est tard....|[il, est, tard, ...]|
```

```
|128849019476|c'est avec plaisi...|[c'est, avec, pla...|
|128849019915|sois forte, ne te...|[sois, forte, ,, ...|
+------------+--------------------+--------------------+
```

Pour avoir des tokens et moins de bruit sémantique, on peut transformer chacun des tokens dans leur version lemmatisée. En linguistique, le terme de lemme est important : c'est la variante dictionnaire d'un token. Malheureusement, on ne pourra pas le faire fonctionner en français avec CoreNLP, car il n'y a aucun modèle existant pré-entraîné. Nous allons tout de même montrer l'exemple de code, car il s'applique sur d'autres langues (en anglais notamment).

```
val withLemmes = datas.withColumn("lemmas", lemma(col("sentences")))
```

À partir de là, Nina peut commencer à faire de la statistique sur notre texte. Pour rappel, on distingue trois valeurs importantes :

- *Distribution* : Calcule une mesure sur une classe de valeur
- *Répartition* : Calcule la différence entre chaque distribution
- *Corrélation* : Trouve des mesures quantifiées soulignant un lien entre deux ou plus de valeurs

Pour avoir une distribution des tokens du corpus, classés par les plus vus en premier, elle se retrouve avec une liste. Cependant Nina ne peut pas encore en tirer une base solide pour ses calculs statistiques à cause des nombreux *stopwords* et ponctuation présents. Il va donc falloir « nettoyer » un peu la donnée, avec la librairie Spark.

```
withTokens
    .select(explode(col("tokens")).as("token"))
    .groupBy("token")
    .count()
    .orderBy(col("count").desc)
    .show

+-----+-----+
|token|count|
+-----+-----+
|    ,| 3528|
```

```
|    .| 2014|
|   de| 1525|
|   et| 1034|
|   je|  934|
|   la|  845|
|  que|  830|
|    à|  819|
|   le|  767|
| vous|  549|
| pour|  476|
|  les|  438|
|   en|  429|
| nous|  388|
|   j'|  357|
|   ne|  350|
|    _|  341|
|  pas|  336|
|  qui|  333|
|   un|  310|
+-----+-----+
```

Nettoyage des tokens : Suppression des stopwords et ponctuations

Pour supprimer les tokens qui ne vont apporter que du bruit dans les futures prédictions ou écritures de règles, et ainsi avoir un meilleur aperçu des distributions de mots, Nina va appliquer une liste de *stopwords* à supprimer. Pour cela, nous allons utiliser la librairie MLLib. Les dernières versions de Spark (Spark 2.0+) possèdent une liste de stopwords français correspondant à la liste de la base de données PostgreSQL. Néanmoins, cette liste n'est pas complète, Nina a donc fait le choix d'en rajouter d'autres, dont une partie de la ponctuation.

```
import org.apache.spark.ml.feature.StopWordsRemover

val remover = new StopWordsRemover()
    .setStopWords(StopWordsRemover.loadDefaultStopWords("french")
        ++ Array("a", "l'", "c'", "d'", "j'", "les", "c'est", "cette",
"tous", "tout", "avoir", "être", "aller", "...", "'", ",", ".", "!",
```

```
"_", "?", ":", ";", "-LRB-", "-RRB-", "#", "`"))
    .setInputCol("tokens")
    .setOutputCol("filtered_tokens")

val withoutStopwords = remover.transform(withTokens)
```

Extraction de features : Le retour

Maintenant que Nina a retiré la plupart du bruit de ses données textuelles et analysé son corpus, elle peut maintenant extraire des *features* plus cohérentes. À cause du processus de découpage de CoreNLP, il manque encore quelques *stopwords*. Mais elle peut déjà partir sur cette base.

Distribution des tokens

```
withoutStopwords
    .select(explode(col("filtered_tokens")).as("token"))
    .groupBy("token")
    .count()
    .orderBy(col("count").desc)
    .show

+------------+-----+
|token       |count|
+------------+-----+
|plus        |264  |
|bien        |208  |
|si          |195  |
|comme       |128  |
|Lettre      |119  |
|champ       |112  |
|tombé       |112  |
|écrite      |110  |
|mort        |109  |
|d'honneur   |104  |
|faire       |104  |
|France      |102  |
|fait        |91   |
|coeur       |88   |
|vie         |84   |
|lettre      |84   |
```

```
|qu'il       |83   |
|car         |82   |
|chère       |82   |
|devoir      |80   |
+------------+-----+
```

Pour avoir les tokens les moins récurrents, il suffit de retirer l'appel à `.desc` dans le `orderBy`.

Pour avoir les tokens les plus récurrents par ligne, il suffit de grouper aussi par l'`id` de la phrase.

Les n-grammes

On peut avoir les *bigrammes* et *trigrammes* avec la librairie Spark MlLib en se basant sur la colonne de tokens filtrés.

```
import org.apache.spark.ml.feature.NGram

val bigram = new
NGram().setInputCol("filtered_tokens").setOutputCol("ngrams").setN(2)
val trigram = new
NGram().setInputCol("filtered_tokens").setOutputCol("ngrams").setN(3)

val withBigram = bigram.transform(withoutStopwords)
val withTrigram = trigram.transform(withoutStopwords)

withBigram.select("ngrams").show(2, truncate = false)

+------------------------------------------------------------------------
-------------------------------------------+
|ngrams
|
+------------------------------------------------------------------------
-------------------------------------------+
|[Union Pères, Pères Mères, Mères dont, dont fils, fils morts, morts
Patrie, ... Paris VIe, VIe édité, édité livre] |
|[paris 29, 29 Octobre, Octobre 1921]
|
+------------------------------------------------------------------------
-------------------------------------------+
```

Catégorisation grammaticale

La catégorisation grammaticale est une application qui transforme un token en une paire (token, catégorie), cette catégorie modélisant la nature du token (déterminant, adverbe, nom, ponctuation...).

Elle marche en français avec CoreNLP, avec l'étiquetage basé sur le standard du Penn Treebank version 3. Pour la catégorisation grammaticale et la reconnaissance d'entités nommées que nous verrons plus loin, nous devons étiqueter la phrase complète, et non les tokens séparés.

```
val withPos = datas.withColumn("pos", pos(col("sentences")))

// Sortie :
|128849018883|paris, le 29 Octobre 1921. |[NNP, ,, DT, CD, NNP, CD, .]
// Soit : nom commun, virgule, déterminant, nombre cardinal, nom
commun, nombre cardinal, point
```

Reconnaissance d'entités nommées

La reconnaissance d'entités nommées n'est pas encore implémenté avec le modèle français donné par Stanford. Nous avons utilisé celui produit par le projet « Europeana Newspaper« . Les libellés utilisés sont des libellés *chunkés* par token selon le format IOB de Ramshaw & Marcus. Voici le résultat avec l'utilisation du modèle pré-entraîné et CoreNLP :

```
val withNe = datas.withColumn("ne", ner(col("sentences")))

// Sortie :
|128849018887|_ Saint-Gaudens, samedi 26 Septembre 1914. | [O, I-LIEU,
O, O, NUMBER, O, DATE, O]
```

Petite explication sur l'étiquetage au format IOB : Le tag préfixé `B-` symbolise le début d'une série d'étiquette, le symbole `O-` la fin d'une série d'étiquette. Enfin, `I-`, présent dans l'exemple, le reste. S'il n'y a qu'un token avec la même étiquette, pas besoin de mettre `I-`.

Vous remarquerez donc que la reconnaissance de dates n'est pas assez pertinente (le mieux serait d'avoir l'étiquette `DATE` pour la série de tokens `samedi 26 septembre 1914`, plus précisément : `B-`

`DATE, I-DATE, I-DATE, O-DATE`). En effet, même jusqu'à la dernière version, celle-ci n'existe qu'en anglais dans CoreNLP.

Création de features

Nina a maintenant les libellés dont elle a besoin pour faire de la répartition. Ainsi, elle va pouvoir enfin créer ses champs lexicaux. Elle va ainsi faire de la répartition par type d'entité nommée : voir les personnes les plus récurrentes du corpus. Cela donne un premier rapport des entités les plus mentionnées dans le texte. L'idée du champ lexical serait d'appliquer ces informations aussi sur les lieux. Enfin, Nina aura aussi recours à la vectorisation, permettant de détecter les synonymes et les antonymes.

Distribution par type d'entité nommées

Il faut avoir un *DataFrame* qui contient une colonne contenant un tableau de tokens, et un autre un tableau des libellés d'entité nommée correspondant, avec Spark.

```
val mergeColumns = udf((tokens: WrappedArray[String], ner:
WrappedArray[String]) => {
    (tokens zip ner).map(t => Array(t._1, t._2))
})

val tokensWithNe = sentencesWithNeAndTokens
    .select(mergeColumns(col("tokens"), col("ne")).as("merged"))
    .select(explode(col("merged")).as("entities"))
    .select(
        col("entities")(0).as("token"),
        col("entities")(1).as("named_entity")
    )
```

On a ainsi un *DataFrame* avec pour chaque ligne, son token et son libellé entité nommée. Maintenant, il faut filtrer par personne, et trier par les tokens les plus récurrents.

```
tokensWithNe.filter(col("named_entity").contains("PERS"))
    .groupBy(col("token"))
    .count()
    .orderBy(col("count").desc)
```

```
+--------+-----+
|   token|count|
+--------+-----+
|    Dieu|   58|
|   Maman|   25|
|   Henri|   23|
|    Juin|   16|
| Parents|   14|
|  Patrie|   13|
| Juillet|   13|
|Français|   11|
|   Louis|    9|
|  Marcel|    9|
|   André|    9|
| ...    | ... |
+--------+-----+
```

Il est aussi tout à fait possible d'améliorer ce code et de s'aider du `DataFrame` contenant la liste sans les *stopwords*. Ainsi, on élague les faux positifs d'entités personnes, qui sont en réalité des *stopwords*.

Vectorisation

La vectorisation (*word embedding*) est une approche de *machine learning* qui transforme tous les mots en un vecteur et les place dans un espace vectoriel représentant notre environnement. Les vecteurs proches seront des synonymes, les vecteurs opposés des antonymes.

On peut ainsi rapidement faire ses premiers tests de *features* de synonymes grâce à la vectorisation, notion très présente et utilisée de nos jours pour le traitement du langage. On peut faire un *word embedding* de nos tokens avec Spark ML en utilisant les *transformers*, et avoir en résultat des tokens synonymes. Nina a forcé la *seed* (graine) et une fenêtre de tokens réduite (par défaut 5) car il y a beaucoup de petites phrases. Nina va chercher les synonymes de « papa », à partir du `DataFrame` contenant les tokens filtrés :

```
import org.apache.spark.ml.feature.Word2Vec
import org.apache.spark.ml.linalg.Vector

// Définition du Word2Vec
val word2vec = new Word2Vec()
  .setInputCol("filtered_tokens")
```

```
  .setOutputCol("word2vec")
  .setSeed(42)
  .setWindowSize(3)
  .setMinCount(0)

// Entraînement du modèle Word2Vec avec notre corpus de tokens filtrés
val model = word2vec.fit(withoutStopwords.select("filtered_tokens"))
val result = model.transform(withoutStopwords)

// Application du modèle pour trouver les synonymes
model.findSynonyms("Papa", 20)

+----------+------------------+
|      word|        similarity|
+----------+------------------+
|     faire|0.7007346384322458|
|  adresser|0.6470895444147055|
|      bout|0.6349059664249204|
|      père| 0.615115274363701|
|       qu'|0.5969694204633825|
|       âme|0.5803198376992913|
|     calme|0.5800298983678057|
|       ici|0.5635688401774845|
|     belle| 0.563515918901308|
|    venger|0.5505874781021223|
|     Maman| 0.548059331277339|
|allemandes|0.5473994405466066|
|     écris|0.5438000211002777|
|    Perret|0.5330740098236204|
| remercier|0.5319559814015427|
| existence|0.5313522019194579|
|      Mère|0.5307856370822023|
|    chères|0.5277326390977163|
|     tante|0.5211589802541844|
|     brume|0.5193665746500904|
+----------+------------------+
```

Le modèle donne des résultats qui semble mitigés, mais certains semblent avoir du « sens ». Pour avoir de meilleurs résultats, il faudrait entraîner la vectorisation sur un plus grand nombre de données.

Conclusion

Grâce à Spark MLLib et CoreNLP, Nina est en possession de tous les outils pour faire une première exploration avancée des données, ce qui lui a permis de tirer de premières conclusions sur celles-ci.

Il faut toutefois retenir que les modèles de TAL (hors *machine learning*) traitant du français directement disponibles sont rares : Nous avons dû améliorer le code de Spark CoreNLP et ajouter un modèle français pour CoreNLP entrainé pour la reconnaissance d'entités nommées sur des journaux (l'un des rares modèles disponibles sur Internet), ce qui n'était pas adapté au corpus épistolaire que nous avons vu en exemple. Il peut être nécessaire d'entrainer son propre modèle en fonction du type de texte que l'on traite pour avoir de bonnes performances sur la reconnaissance du texte.

Enfin, avec notre précédent billet, nous avons essayé de représenter les tâches statistiques appliquées à l'exploration de données, les différentes étapes et démontrer qu'il est tout à fait possible de les faire en Scala.