# L.A.R.A Documentation

Rédigé par : Louis Grenioux, Anna-Rose Lescure, Riad El Otmani

8 juin 2020

## Table des matières

# 1 Introduction

L.A.R.A est un logiciel complet mettant en œuvre un agent conversationnel entraîné sur des discussions issues de Facebook. Ce logiciel repose notamment sur les bibliothèques Deeplearning4J et JavaFX pour Java ainsi que sur les bibliothèques Tensorflow et Keras pour python. Toutes ces bibliothèques sont utilisées dans leurs plus récentes versions au mois de Juin 2020. L'intégralité du code est opensource et peut être consultée sur le github LaraProject.

L.A.R.A est découpé en plusieurs modules :

— Facebook Parser
— Natural Langage Processing
— RNN2Java
— Réseau récursif de neurones
— Interface graphique

Le fonctionnement de chaque module, leur utilisation ainsi que leurs liens avec les autres modules sont expliqués dans les moindres détails afin d'exposer toute la flexibilité du logiciel. Dans la dernière section, un tutoriel simple explique comment utiliser le logiciel sans aucunes ressources initiales avec les paramètres par défaut.

# 2 Facebook Parser

## 2.1 Objectifs et outils

Nous avons décidé de travailler tout au long de ce projet en ayant comme objectif d'entraîner notre programme sur nos données personnelles, extraites notamment de nos conversations Facebook. Ainsi le parser qui a été développé est l'un des modules critiques qui forment la version actuelle de L.A.R.A.

Son rôle est d'ordonner et organiser le dump de données que nous pouvons télécharger sur Facebook de manière à en faire un dataset d'entraînement et de validation valide pour les prochains modules qui composent le projet LARA.

## 2.2 Traitement des données

Le parser opère sur le dump FaceBook un ensemble d'opérations dont il est possible de voir le résultat dans cette partie.

### 2.2.1 Données initiales

Données extraites du fichier `dumpFacebook.json`.

```
...
{
    "sender_name": "Riad El Otmani",
    "timestamp_ms": 1579199706153,
    "content": "J'ai des trains \u00c3\u00a0 15",
    "type": "Generic"
},
{
    "sender_name": "Louis Popi",
    "timestamp_ms": 1579199562206,
    "content": "C'est une bonne marge de s\u00c3\u00a9curit\u00c3\u00a9",
    "type": "Generic"
}
...
```

### 2.2.2 Données finales

Données produites par le parser après traitement.

```
...
{
   "sender_name":"Louis Popi",
   "content":"c'est une bonne marge de securite",
   "conversationId":4,
   "subConversationId":27
```

```
},
{
   "sender_name":"Riad El Otmani",
   "content":"j'ai des trains a  15",
   "conversationId":3,
   "subConversationId":27
}
...
```

## 2.3 Programme

### 2.3.1 Constructeur de la classe `Parser`

Le parser Facebok est construit autour d'une classe `Parser`. Son constructeur initialise les variables de la classe avec les valeurs qui lui sont transmises et s'occupe de charger en mémoire le contenu du dump téléchargé sur Facebook. Voici son constructeur :

```
__init__(self, fileName, nbMessages, delayBetween2Conv, answerer, withTimestamp=True,
                                                                    debug=False)
```

- `fileName` : donne le chemin d'accès au fichier de données à parser qui sera utilisé par la suite.
- `nbMessage` : correspond aux nombres de messages qui vont être traités par le parser.
- `delayBetween2Conv` : indique une durée arbitraire qui permet de délimiter les conversations.
- `answerer` : indique la personne qui aux questions
- `withTimeStamp` : option qui permet l'utilisation ou non des timestamps.
- `debug` : mode debuggage.

### 2.3.2 La méthode `start`

La méthode `start` est l'une des méthodes les plus importantes de la classe ; elle ne prend aucun argument. Elle initialise deux listes `self.conversation['speakers']` et `self.conversation['messages']` qui contiendront respectivement la liste des participants à la conversation et son contenu.

Dans un premier temps, on remplit la liste `conversation.['speakers']` simplement en se basant sur le champ `conversation['participants']` du dump JSON. Vient ensuite le problème de la gestion des messages. Dans un premier temps, nous devons vérifier l'utilisabilité du dump.

Pour cela on itère sur chaque élément du dump et on vérifie que l'élément en question est bien un message en utilisant la methode `self.getMsg()`. Dans le cas où c'est bien un message, on peut l'ajouter dans la liste `self.conversation['message']`, dans le cas contraire on émet un message d'erreur.

Au vu de la grande variété possible des différents messages (text, emoji, réaction, image, vidéo, ...), nous devons passer par une phase de nettoyage en amont afin de réduire nos données à quelque chose d'utilisable par les autres modules et notamment pour le réseau de neuronnes.

On décide ainsi de trier les messages par leur timestamps afin de les ordonner temporellement sous forme de conversations. Pour réaliser cette séparation, on se base sur l'écart de temps entre les différents messages car on part du principe que le RNN sera plus performant en travaillant par conversations puisque les messages seront liés entre eux par le contexte de la conversation.

```
if abs(int(self.dataRaw['messages'][k]['timestamp_ms']) - timestamp) >
                                    self.delayBetween2Conv
```

Pendant la création de la conversation, repérée par son `id`, on vérifie certaines conditions pour assurer la validité de la conversation.

Ainsi, **une nouvelle conversation doit obligatoirement commencer par une question et la conversation précédente doit se terminer par une réponse**. On fera aussi attention à éviter les conversations de type monologue où il n'y a pas d'échange entre les deux participants. Quand une de ces conditions n'est pas vérifiée, on supprime simplement la conversation en question.

Voici une représentation simplifiée de l'attribution des `ids` par le parser :

```
(ID = 0 | SID = 0) Riad : Salut
(ID = 0 | SID = 1) Louis : Salut
(ID = 0 | SID = 2) Riad : ça va ?
(ID = 0 | SID = 3) Louis : oui et toi ?
(ID = 0 | SID = 3) Louis : tu fais quoi ?
[20 minutes plus tard]
(ID = 1 | SID = 0) Riad : je travaille sur le parser et toi ?
(ID = 1 | SID = 1) Louis : je m'occupe du nlp
```

### 2.3.3 La méthode `isAnswerer`

Voici les paramètres : `isAnswerer(self, name)`. Il s'agit d'une méthode simple qui renvoie le booléen résultant de cette opération : `name == self.answerer`.

### 2.3.4 La méthode `removeFullConv`

Voici les paramètres : `removeFullConv(self, conv_id)`. Cette méthode permet de supprimer une conversation entière grâce à son `id`.

### 2.3.5 La méthode `removeSubConv`

Voici les paramètres : `removeSubConv(self, conv_id, subconv_id)`. Cette méthode permet de supprimer un message d'une conversation grâce à son `id`.

### 2.3.6 La méthode `getMsg`

Voici les paramètres : `getMsg(self, k, conversationId, subConversationId)`. Cette méthode se charge de renvoyer un message structuré de type dictionnaire (tel que défini par Python).

Voici un extrait du code :

```
if self.withTimestamp:
        msg = {
        'sender\_name': self.dataRaw['messages'][k]['sender\_name'],
        'content': self.cleanMessage(self.dataRaw['messages'][k]['content']),
        'timestamp': self.dataRaw['messages'][k]['timestamp\_ms'],
        'conversationId': conversationId,
        'subConversationId': subConversationId
}
```

Si le champ `self.withTimestamp` vaut `False`, il suffit de retirer cette ligne. Dans le cas où le le message reçu n'est pas interprétable (message video, audio, réaction, ...) alors la fonction `self.cleanMsg` renvoie une erreur, ce qui entraîne la sortie d'un `try` et permet le catch de l'erreur dans le `except` ; le message est alors supprimé.

On renvoie finalement soit `msg` si tout s'est bien déroulé, soit l'`id` du message qui a causé l'erreur.

### 2.3.7 La méthode `cleanMsg`

Voici les paramètres : `cleanMessage(self, message)`. Cette méthode renvoie le message sous forme de chaîne de caractères après avoir subi quelques transformations. Le but étant de transformer les caractères encodés par Facebook et de supprimer les majuscules du texte.

### 2.3.8 La méthode `extract_time`

Voici les paramètres : `extract_time(self, msg)`. Cette méthode renvoie le timestamp associé au message. Dans le cas où celui-ci n'est pas défini, on renvoie la valeur `0`.

### 2.3.9 La méthode `getNbConversation`

Voici les paramètres : `getNbConversation(self)`. Cette méthode renvoie le nombre de messages enregistrés dans la lists `self.conversations['messages']`.

### 2.3.10 La méthode `finalDump`

Voici les paramètres : `finalDump(self, filename)`. Cette méthode crée le fichier `.json` final qui sera utilisé par les autres modules.

## 2.4 Utilisation

### 2.4.1 Le parser

Le parser a été conçu pour être utilisable le plus simplement possible. Pour cela une utilisation par ligne de commande a été implantée grâce à la librairie `argparse`. Après avoir installer les dépendances via

```
$ pip install -r requirements.txt
```

Il suffit d'ouvrir un terminal et de lancer la commande :

```
$ python3 parserFB.py inputFile outputFile answerer
```

Cette commande peut-être enrichie en donnant des arguments supplémentaires optionnels :

```
$ python3 parserFB.py inputFile outputFile answerer -nbMessages [int] --debug [True/False]
                  --withTimestamp [True/False] -delayBetween2Conv [int]
```

`parserFB.py` prend de multiples paramètres :
— `inputFile` : dossier de donéées de FaceBook (le dossier `messages` habituellement).
— `outputFile` : dossier de sortie du parser. Doit être préalablement créé.
— `answerer` : filtre les messages autour du compte spécifié. Par exemple, "Jean Dupont". Ainsi, le dataset pourra être utilisé pour faire un chatbot ayant le même style de message que cette personne.
— `--nbMessages` : utilisé pour spécifier une limite max dans le nombre de messages parsés. Utile lorsque le nombre de messages est énorme, ou pour faire un test sur un très petit dataset.
— `--debug` : sert à afficher diverses informations de débogage.
— `--withTimestamp` : sauvegarder l'heure d'envoi du message dans la sortie parsée.
— `--delayBetween2Conv` : seuil à partir de laquelle deux messages sont considérés comme faisant partie de conversations distinctes.
— `--export` : exporte le résultat du parsing dans un seul gros fichier .json

### 2.4.2 Les scripts BASH

Le script `script_parsing.sh` permet d'utiliser le parser sur toutes les conversations d'un dump Facebook. Son utilisation est la suivante :

```
./script_parsing.sh fb_folder out_folder answerer fb_parser
```

où
— `fb_folder` est le dossier de décompression du dossier Facebook ;
— `out_folder` est le dossier où les fichiers "parsés" seront stockés ;
— `answerer` est le nom de la personne qui répond aux questions ;
— `fb_parser` est le chemin vers le script python.
L'intégralité des fichiers qui ont été traités peuvent maintenant être lus par le NLP avec `script_java_format.sh` qui s'utilise comme suit :

```
./script_java_format.sh in_folder out_folder answerer java_lib
```

où
— `in_folder` est le dossier contenant les sorties du parser (précédemment appelé `out_folder`) ;
— `out_folder` est le dossier où seront stockés les résultat du NLP ;
— `answerer` est le nom de la personne qui répond aux questions ;
— `java_lib` est le chemin vers le `jar` du NLP (de lara).

# 3 Natural Langage Processing - NLP

## 3.1 Objectifs et outils

L'objectif du module NLP est double : il doit à la fois faire toute la partie de préparation des données qui seront ensuite mises à la disposition du RNN, et il doit faire toute la partie de langage processing en entraînant des algorithmes de Word2Vec. Ce package (`org.lara.nlp`), dont le code source individuel peut être retrouvé ici, est compilé avec le logiciel Maven et fait majoritairement appel à la bibliothèque open-source de référence en Deep Learning : Deeplearning4J.

## 3.2 La classe `Context`

La classe abstraite `Context` (`org.lara.nlp.context.Context`) permet de réaliser la préparation des données. Elle a été implémentée par deux classes filles : `Facebook`, qui exploite les données issues du parser et `Cornell`, qui exploite les données issues du corpus de Cornell. L'objectif est de fournir à partir de ces différentes sources deux listes de même taille : l'une contenant les questions et l'autre des réponses ; toutes deux parfaitement nettoyées (nous détaillerons ce nettoyage plus tard).

La méthode `init()` permet d'initialiser l'objet en remplissant ces deux listes. Les deux listes peuvent ensuite

être "tokenizées" avec `tokenize()` (cf. `RNN`), ou nettoyées avec `cleaning()` (cf. `Processer`).

Les données peuvent être exportées et importées avec les méthodes `save(String path_questions, String path_answers)` et `restore(String path_questions, String path_answers)`. Il est aussi possible d'exporter ces deux listes dans un même fichier via `exportData(String path)` (cette méthode sera utilisée dans pour le RNN).

#### 3.2.1 La classe `Cornell`

La classe `Cornell` implémente `Context` pour le corpus de Cornell. Son contructeur est

```
Cornell(String lines_filename, String conversations_filename, int min_length, int
                                    max_length)
```

où

— `min_length` et `max_length` sont des paramètres destinés à la classe `Processer` ;
— `lines_filename` est le chemin du fichier `movie_lines.txt` ;
— `conversations_filenames` est le chemin du fichier `movie_conversations.txt`.

Lors de l'appel à `init()` les listes `questions` et `answers` vont être remplies par tous les dialogues possibles apparaissant dans le corpus.

#### 3.2.2 La classe `Facebook`

La classe `Cornell` implémente `Context` à partir des données issues du parser grâce à la bibliothèque `org.json.simple`. Son constructeur est

```
Facebook(String json_filename, String answerer, int min_length, int max_length)
```

où

— `min_length` et `max_length` sont des paramètres destinés à la classe `Processer` ;
— `json_filename` est le chemin du fichier issu du parser ;
— `answerer` est le nom de la personne qui répond aux questions.

Lors de l'appel à `init()` les listes `questions` et `answers` vont être remplies en parcourant le fichier JSON avec les différentes variables exportées par le parser.

#### 3.2.3 La classe `Simple`

La classe `Simple` implémente `Context` à partir d'un format de donnée très simple (qui est d'ailleurs celui exporté par `exportData(String path)` évoquée précédemment) :

```
Question: ...
 Answer: ...
Question: ...
 Answer: ...
```

Son constructeur est

```
public Simple(String filename, int min_length, int max_length)
```

où

— `min_length` et `max_length` sont des paramètres destinés à la classe `Processer` ;
— `filename` est le fichier contenant les conversations.

Lors de l'appel à `init()` les listes `questions` et `answers` vont être remplies en parcourant le fichier de conversations.

#### 3.2.4 La classe `Processer`

La classe `Processer` est liée à la classe `Context`. Elle met en œuvre les méthodes permettant de nettoyer les données :

— `clean_text(String orig)` cette fonction centrale renvoie une chaîne de caractère où plusieurs changements ont été effectués :
  — les lettres du texte deviennent des minuscules ;
  — les URLs sont supprimées ;
  — la langue anglaise est simplifiée ("you're" devient "you are" par exemple) ;
  — les balises HTML sont enlevées ;

  - les emojis sont remplacés par leur signification (via la bibliothèque 'emoji4j' qui peut être retrouvée ici) ;
  - la ponctuation est traitée par des expressions régulières ;
  - les caractères de terminaison de ligne sont supprimés.
- `cleanQuestionsAnswers()` applique la fonction précédente à toutes les questions et les réponses ;
- `lengthFilter()` supprime les couples question/réponse où l'un ou l'autre a un nombre de mots inférieur à `min_length` ou supérieur à `max_length` ;
- `tokenize_sentence(String s)` applique la tokenisation en ajoutant les tokens `<START>` et `<END>` au début et à la fin de la phrase et en rajoutant le mot `<PAD>` pour que la phrase ait une longueur égale à `max_length` ;
- `tokenize()` applique `tokenize_sentence` à chaque question et à chaque réponse ;
- `process()` applique le filtre de longueur et le nettoyage à chaque question et à chaque réponse.

#### 3.2.5 Le parser LELU

LELU est un dataset disponible sur Kaggle qui regroupe 556,621 conversations issues de Reddit. Le script `parserLELU.py` qui peut être retrouvé dans le repository LaraProject/parserLELU ouvre les données de LELU et produit (via `stdout`) un fichier avec le format de `Simple`. Ce dataset sera utilisé pour l'entraînement du Word2Vec.

### 3.3 Les implémentations de l'algorithme de plongement lexical

Le NLP est aussi responsable de la tâche de plongement lexical (word embedding). Il implémente plusieurs algorithmes de `Word2Vec` via des classes très similaires mais sans héritage commun explicite. Ces classes utilisent pleinement la bibliothèque Deeplearning4J. Les classes suivantes appartiennent au package `org.lara.nlp.word2vec`.

#### 3.3.1 La classe `Glv` - Modèle GloVe

La classe `Glv` implémente le modèle GloVe. **Malheureusement, dans la version 6 de deeplearning4j, la classe glove a été supprimée**. Son constructeur était

```
Glv(ArrayList<String> sentences)
```

où

- `sentences` est la liste des phrases des données d'entraînement

Le réseau de neurones est initialisé avec les mêmes hyper-paramètres à chaque instance. La fonction `write_vectors(String path)` exporte les vecteurs dans un format lisible humainement. La fonction `getModel()` retourne le modèle. Le code de cette fonction est toujours présent parmi les commits du projet.

#### 3.3.2 La classe `Pv` - Paragraph Vectors

La classe `Pv` implémente le modèle de ParagraphVectors (ou Doc2Vec). Son constructeur est

```
Pv(WordVectors model)
```

où

- `model` est un modèle héritant de `WordVectors` issu de DL4J (tous les modèles présentés ici sont concernés).

La classe `Pv` possède un constructeur alternatif

```
Pv(String path)
```

qui permet de restaurer un modèle précédemment sauvegardé à l'aide de `save_model(String path)`. Les fonctions `write_vectors(String path)` et `getModel()` sont aussi présentes dans cette classe.

#### 3.3.3 La classe `Sv` - Sequence Vectors

La classe `Sv` implémente de l'extraction de caractéristiques abstraites pour des instances du type `Sequences` et `SequenceElements`, en utilisant les algorithmes SkipGram, CBOW ou DBOW. Son constructeur est

```
Sv(ArrayList<String> sentences)
```

où

- `sentences` est la liste des phrases des données d'entraînement

Tous les hyper-paramètres de l'algorithme sont fixés.

Le modèle peut être sauvegardé et restauré à l'aide du constructeur

```
Sv(String path)
```

et avec la fonction `save_model()`. Comme les autres classes, `Sv` possède les fonctions `write_vectors(String path)` et `getModel()`.

#### 3.3.4 La classe `W2v` - Word2Vec

La classe `W2v` est la classe la plus utilisée dans tout le projet. Elle met en œuvre l'algorithme Word2Vec de Google. Son constructeur est le suivant

```
W2v(ArrayList<String> words, int minWordFrequency, int iterations, int epochs, int
                                        dimension)
```

où

- `words` est la liste des phrases des données d'entraînement ;
- `minWordFrequency` est le nombre minimal d'occurences d'un mot pour qu'il soit considéré ;
- `iterations` est le nombre d'itérations de l'algorithme du `Word2Vec` ;
- `epochs` est le nombre d'époques dans l'entraînement du réseau de neurones ;
- `dimension` est la taille des vecteurs ;

Il est aussi possible de restaurer un modèle sauvegardé (via la fonction `save_model(String path)`) avec le constructeur

```
W2v(String path)
```

Comme les autres classes, `W2v` dispose des fonctions `write_vectors(String path)` et `getModel()`. En utilisant le script bash `gensim_convert.sh /chemin/du/fichier` sur le fichier issu de `write_vectors(String path)`, on convertit le fichier dans un format compatible avec le module gensim de Python via

```
from gensim.models import KeyedVectors
KeyedVectors.load_word2vec_format(fichier.txt, binary=False)
```

#### 3.3.5 Le script `fasttxt.py`

La réalisation du modèle Word2Vec avec l'approche de Deeplearning4J pouvant être très longue, nous proposons une alternative via python et le module `gensim` (évoqué précédemment). Elle implémente l'architecture FastText de Facebook à partir d'un fichier compréhensible par la classe `Simple`. Avant de poursuivre, il est nécessaire d'installer le module `gensim` via `$ pip install -r requirements.txt` (cf la section sur le RNN). Son utilisation est précisée par `$ python fasttext.py -h`.

- `--size` : définit la taille des vecteurs de mot (un plus grand vecteur permettra peut être plus de précision dans les prédictions, mais sera plus long à entraîner, et nécessitera plus de données pour un résultat satisfaisant).
- `--window` : taille de la fenêtre de messages utilisés pour donner du contexte à un message.
- `--minCount` : nombres d'occurences minimales d'un mot pour qu'il soit pris en compte dans le Word2Vec.
- `--workers` : nombre de fils d'exécution utilisés lors de l'apprentissage du Word2Vec
- `--epochs` : nombre d'époques d'entraînement, c'est-à-dire combien de fois l'apprentissage est affiné sur le set de données. Un nombre plus élevé nécessite plus de temps, mais peut apporter une meilleure précision du Word2Vec
- `--path` : dossier d'export des vecteurs de mots.
- `--modelPath` : dossier d'export du modèle FastText complet.

### 3.4 Compilation, tests et utilisation

Le module se compile grâce au gestionnaire de dépendance Maven. Il suffit d'une seule commande :

```
mvn package
```

Nous allons détailler ci-dessous les classes de test des fonctionnalités principales.

#### 3.4.1 Test de la classe `Cornell`

Il est possible de tester la classe `Cornell` via la classe `CornellTest` :

```
$ java -cp target/laraproject-*.jar org.lara.nlp.context.CornellTest movie_lines.txt
                    movie_conversations.txt cornell_export.txt
```

où

- `movie_lines.txt` et `movie_conversations.txt` proviennent du corpus de Cornell ;
- `cornell_export.txt` représente le chemin du fichier d'exportation du contexte.

Cette classe teste l'importation, le nettoyage et l'exportation.

### 3.4.2 Test de la classe `Facebook`

Il est possible de tester la classe `Facebook` via la classe `FacebookTest` :

```
$ java -cp target/laraproject-*.jar org.lara.nlp.context.FacebookTest parser_export.js
                            "Prenom Nom" facebook_export.txt
```

où

— `parser_export.js` est le fichier exporté par le parser de conversations Facebook ;
— `"Prenom Nom"` représente la personne qui répond aux questions ;
— `facebook_export.txt` représente le chemin du fichier d'exportation du contexte.

Cette classe teste l'importation, le nettoyage (la longueur maximale des phrases est de 40 par défaut) et l'exportation.

### 3.4.3 Test de la classe `Simple`

Il est possible de tester la classe `Simple` via la classe `SimpleTest` :

```
$ java -cp target/laraproject-*.jar org.lara.nlp.context.SimpleTest conversations.txt
                    min_length max_length conversations_export.txt
```

où

— `conversations.txt` est le fichier contenant les conversations ;
— `min_length` et `max_length` sont des paramètres destinés à la classe `Processer` ;
— `conversations_export.txt` représente le chemin du fichier d'exportation du contexte.

Cette classe teste l'importation, le nettoyage (avec les valeurs de longueur fournies) et l'exportation.

### 3.4.4 Test de la classe `W2v` avec `Cornell`

Il est possible de tester la classe `W2v` via la classe `W2vTest` à partir du corpus de Cornell :

```
$ java -cp target/laraproject-*.jar org.lara.nlp.word2vec.W2vTest movie_lines.txt
                    movie_conversations.txt word2vec_vectors.txt
```

où `word2vec_vectors.txt` est le chemin du fichier dans lequel seront écrits les vecteurs des mots du corpus (de dimension 100). Attention, ce test est particulièrement lent. Cette classe teste l'importation, le nettoyage, la tokenisation et l'exportation de la classe `Cornell` ainsi que l'entraînement et l'export pour la classe `W2v`. Cette classe de test pourrait se généraliser à `Facebook` et s'étendre à d'autres modèles (comme `Glv`) mais nous avons choisi de passer par la classe `Simple` pour cela.

### 3.4.5 Test de la classe `W2v` avec `Simple`

Il est possible de tester la classe `W2v` via la classe `W2vTest` à partir d'un fichier compris par `Simple` :

```
$ java -cp target/laraproject-*.jar org.lara.nlp.word2vec.W2vSimpleTest filename.txt
                    min_length max_length word2vec_vectors.txt
```

où `word2vec_vectors.txt` est le chemin du fichier dans lequel seront écrits les vecteurs des mots du corpus (de dimension 100). Attention, ce test est particulièrement lent. Cette classe teste l'importation, le nettoyage, la tokenisation et l'exportation de la classe `Simple` ainsi que l'entraînement et l'export pour la classe `W2v`.

# 4 Lien entre Python et Java - RNN2Java

## 4.1 Motivations

La motivation principale (imprévue) qui a mené à la création de ce module est qu'il fallait un lien entre le RNN et le reste de l'application écrite en Java (GUI, NLP). En effet, la bibliothèque DL4J se heurte (à l'heure actuelle) à un bug qui empêche l'importation de notre modèle de RNN depuis Keras (nous avons reporté le problème détaillé (sur le github `deeplearning4j`) aux développeurs et nous l'avons partiellement corrigé).

D'autre part, nous avions le désir d'implémenter les deux fonctionnalités suivantes :

— fournir une API simple permettant de discuter avec Lara ;
— permettre de détacher le module RNN afin de réaliser l'entraînement et les prédictions sur une machine distante (éventuellement plus puissante)

C'est dans ce contexte que s'inscrit le module `RNN2Java` (`org.lara.rnn` en Java et `python/main.py` en Python). Il est disponible de manière isolée ici. Cette implémentation est inspirée de l'implémentation de Zack Lalanne disponible https ://github.com/zlalanne/java-python-ipc-protobuf.

## 4.2 Objectifs et outils

L'objectif est donc de mettre en place une API ayant les propriétés suivantes :
— être facilement exploitable en Python et en Java ;
— être très simple et facilement extensible ;
Nous avons donc choisi de nous orienter vers l'outil protobuf de Google. La solution exploite des sockets (le script Python est le serveur et le script Java est le client) sur le port 9987.

## 4.3 Implémentation et classes

### 4.3.1 Protobuf

La structure de l'API protobuf est la suivante :

```
option java_package = "org.lara.rnn";

message Command {

  enum CommandType {
    SWITCH_PERSON = 0;
    ANSWER = 1;
    QUESTION = 2;
    SHUTDOWN = 3;
  }

  required CommandType type = 1;
  required string name = 2;
  required string data = 3;
}
```

Il y a donc 4 types de messages différents :
— le type `SWITCH_PERSON` qui indique quel modèle du RNN à charger (le numéro de la personne est dans le champ `data`) ;
— le type `ANSWER` qui représente une réponse (sous forme de chaîne de caractères dans le champ `data`) ;
— le type `QUESTION` qui représente une question (sous forme de chaîne de caractères dans le champ `data`) ;
— le type `SHUTDOWN` qui indiquera une demande d'extinction du serveur.

## 4.4 La classe `Server`

La classe `Server` permet d'assurer la communication côté Java. Son constructeur n'a aucun paramètre (il est hardcodé pour `localhost` par défaut)

`Server()`

Plusieurs méthodes sont alors accessibles afin d'utiliser l'API :
— `makeQuestion(String q)`, `makeShutdown()` et `makeSwitchPerson(int person)` créent les objets `Command` (objet généré par `protobuf`) conformément à la section précédente ;
— `send(Command cmd)` envoie la `Command` via le socket et retourne la `Command` reçue en réponse (il est possible de ne pas attendre la réponse en utilisant la méthode `send_without_answer(Command cmd)`) ;
— Les fonctions `sendQuestion(String q)`, `switchPerson(int person)` et `shutdownServer()` allient les deux types de méthodes précédemment décrits ;
— Les fonctions `openSock()` et `closeSock()` ouvrent et ferment la connexion.

## 4.5 Le script `python/main.py`

La première moitié du script traite de fonctions importées du RNN, nous n'en parlerons pas ici. La seule fonction utile est la fonction `main()`.

Cette fonction initialise dans un premier temps le `socket` qui écoute sur `localhost` au port `PORT`.

Le script exécute une boucle infinie. A chaque itération, elle analyse les données reçues via le socket et tente de les interpréter en tant qu'une `command` de l'API protobuf. Plusieurs cas de figures se présentent alors :
— Si la commande est de type `QUESTION` alors celle-ci est retournée sur `stdout` puis la réponse du RNN est envoyée à travers le `socket` et l'utilisateur est averti via `stdout`. (Cela se fait via la fonction `answer_command` qui construit l'objet de type `command` à partir du résultat de `answer`. Ce dernier réalise

la prédiction à partir du RNN (détails dans la partie RNN) en utilisant les modèles spécifiques à la personne actuelle (cf. plus bas)) ;
— Si la commande est de type `ANSWER` alors le serveur renvoie une erreur et s'arrête ;
— Si la commande est de type `SWITCH_PERSON` alors le script change la sauvegarde du modèle utilisé pour la prédiction. Celles-ci sont stockées dans `models` ;
— Si la commande est de type `SHUTDOWN`, la boucle principale est quittée (l'utilisateur est averti via `stdout`) et le serveur s'arrête.

## 4.6 Compilation, tests et utilisation

### 4.6.1 Compilation

Afin de compiler le code protobuf, il est nécessaire de l'installer sur le système d'exploitation. Par exemple, sous Ubuntu

```
$ sudo snap install protobuf --classic
```

Il faut ensuite compiler le code protobuf

```
$ make
```

Puis compiler le code Java avec Maven

```
$ mvn package
```

Puis installer les dépendances Python

```
$ pip install -r requirements.txt
```

Une fois toutes ces étapes effectuées, assurez-vous que les sauvegardes du RNN (cf la section RNN) sont enregistrées comme suit :

```
models/##/model_enc.h5
models/##/model_dec.h5
models/##/tokenizer.pickle
models/##/length.txt
```

où `##` est le numéro de la sauvegarde (commençant à 1).

### 4.6.2 Utilisation

Il faut tout d'abord lancer le serveur côté Python dans un premier terminal :

```
$ cd python
$ python main.py
```

Puis dans un second terminal, lancer le client Java :

```
$ java -cp target/laraproject-*.jar org.lara.rnn.ServerTest
```

Ceci va :
— Changer la personne
— Faire deux dialogues simples
— Éteindre le serveur
Vous pouvez visualiser les différentes interactions dans la sortie des deux terminaux.

# 5 Réseau de neurones récursif - RNN

Le réseau de neurones utilisé ici est basé sur le script fourni par Swapnil Ashok Jadhav dans son article "Marathi To English Neural Machine Translation With Near Perfect Corpus And Transformers" publié dans son Google Collab. Des modifications majeures ont été effectuées à la fois sur le modèle et sur ses hyper-paramètres.

## 5.1 Objectifs et outils

L'objectif est de mettre en œuvre en Python un modèle de Seq2Seq. Les contraintes sont les suivantes :
— Mettre à profit le NLP ;
— Utiliser des outils récents (TensorFlow 2, Keras 2, Python 3) ;
— Produire des résultats intéressants ;
Le code est disponible de manière isolé sur ce repository.

## 5.2 Fonctions, méthodes et modèle

Les variables et objets `tokenizer`, `questions`, `answers`, `vocab`, `model_w2v`, `embedding_matrix`, `maxlen_questions`, `maxlen_answers` et `VOCAB_SIZE` sont globaux et seront manipulés par la plupart des fonctions.

### 5.2.1 Traitement des données

Dans le cas où on utilise des données autre que celles du NLP, le script réalise tout d'abord un pré-traitement des données. Celui-ci passe par plusieurs étapes :

- le téléchargement des données via `import_data()` ;
- la fonction qui filtre (faiblement) les données et qui réalise la tokenisation
  - elle crée les listes `questions` et `answers` à partir des données téléchargées ;
  - elle enlève les types de données non souhaités ;
  - elle insère les tokens `<start>` et `<end>` au début et à la fin de chaque réponse.
- la fonction `clean_text(text)` est très similaire à celle de `org.lara.nlp.context.Processer` et est appliquée sur l'intégralité de `questions` et de `answers` via l'appel à `clean_everything()`.

Dans le cas contraire, il suffit d'appeler `use_custom_data(path, size)` où `path` est le chemin du fichier exporté par le NLP et `size` est le pourcentage de données utilisé.

### 5.2.2 Word Embedding

Le word embedding exploite les exportations des modèles issus de `org.lara.nlp` et plus précisément des méthodes `write_vectors(String path)` (converties via le script `gensim_convert.txt /chemin/vers/le/modèle.txt`).

- `load_word2vec(model_path, useFastText)` charge le modèle exporté par `org.lara.nlp` (cf explication ci-dessus) au format `gensim` (et plus précisément le format `KeyedVectors`). En utilisant `useFastText=True`, il est possible de charger directement une exportation du modèle `FastText` de `gensim` en précisant le préfixe de la sauvegarde (cette approche, plus lente, est beaucoup plus performante) ;
- les fonction `get_known_words()` et `fit_new_tokenizer()` vont créer un nouvel objet `tf.keras.preprocessing.text.Tokenizer` où le vocabulaire est issu de l'intersection entre le vocabulaire présent dans le dataset et celui du modèle Word2Vec (ceci diminue énormément la dimensionnalité) ;
- `create_embedding_matrix()` assemble la matrice d'intégration (embedding matrix) à partir du nouveau `tokenizer`. Cette matrice qui a autant de lignes que de mots dans `tokenizer` associe à chaque mot un vecteur. C'est une restriction de la matrice du modèle Word2Vec.

### 5.2.3 Modèle de l'encodeur-décodeur seq2seq

Le modèle seq2seq exploite 3 tableaux créés par la fonction `create_input_output()` :

- `encoder_input_data` qui contient les questions tokenizées par `tokenizer` (c'est-à-dire que la question a été transformée d'une suite de mots en une suite de nombres où chaque nombre représente (de manière unique) un mot). Cette suite de nombres est ensuite complétée par des zéros (pour que toutes les questions soient de taille fixe alignée sur la question la plus longue) : c'est l'action de padding ;
- `decoder_input_data` qui est issue des mêmes opérations que `encoder_input_data` mais effectuées sur `answers` ;
- `decoder_output_data` qui est issue des mêmes opérations que `encoder_input_data` mais effectuées sur `answers` où les phrases ont été privées de leur premier mot (le token `<start>`).

Le modèle seq2seq utilise 3 différentes couches (layers) :

- 2 layers d'entrée : `encoder_input_data` et `decoder_input_data` ;
- 2 layer d'intégration (embedding) : `encoder_embedding` et `decoder_embedding`. Ces layers utilisent la matrice d'intégration construite précédemment en guise de poids et ne participe pas à la propagation du gradient lors de l'entraînement ;
- 1 layer LSTM (Long-Short Term Memory) : `decoder_lstm`.

Le fonctionnement est alors le suivant :

- `encoder_input_data` entre dans `encoder_embedding` ;
- la sortie de `encoder_embedding` est mise à l'entrée du LSTM produisant alors 2 vecteurs d'état `h` et `c` ;
- `decoder_input_data` entre dans `decoder_embedding` ;
- la sortie de `decoder_embedding` est mise à l'entrée du LSTM (initialisé avec les 2 vecteurs d'état `h` et `c` précédents) produisant ainsi les séquences de sortie ;
- la sortie du LSTM est alors introduite dans un ultime layer dense possédant autant de neurones que le corpus possède de mots.

Ce modèle est compilé par la fonction `create_model(encoder_input_data, decoder_input_data, decoder_output_data, use_spatial_dropout, use_reccurent_dropout, use_batch_normalisation)` qui retourne le modèle ainsi que `encoder_inputs`, `encoder_states`, `decoder_embedding`, `decoder_lstm`, `decoder_dense`

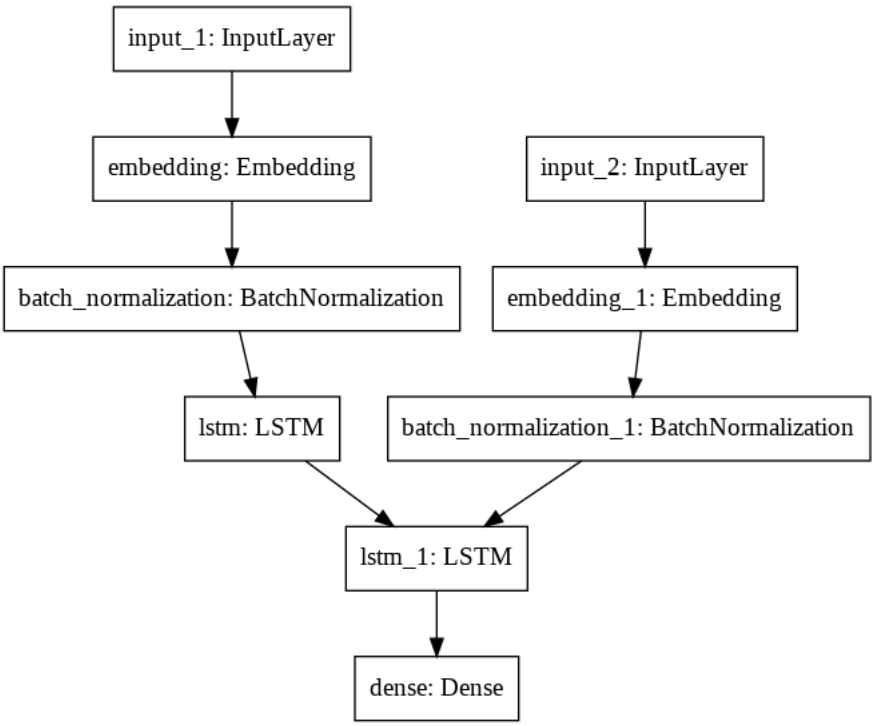


FIGURE 1 – Modèle du RNN

et `decoder_inputs` (nécessaires pour l'inférence). Les trois derniers arguments permettent de complexifier le modèle :

- `use_spatial_dropout` ajoute un dropout en sortie des deux layers d'embedding (il est désactivé par défaut) ;
- `use_reccurent_dropout` ajoute un dropout récurrent dans le LSTM (il est désactivé par défaut et nous prive de l'utilisation de CUDA) ;
- `use_batch_normalisation` ajoute un layer `BatchNormalization` au niveau des deux entrées (il est activé par défaut et nécessaire dans lors de l'utilisation du modèle FastText).

Le schéma récapitulatif du modèle a été exporté via `tf.keras.utils.plot_model`. L'entraînement peut être réalisé avec la fonction `train(use_spatial_dropout, use_reccurent_dropout, use_batch_normalisation)` avec l'optimiseur `NADAM` et la fonction de coût `sparse_categorical_crossentropy`.

#### 5.2.4 Disposition inférentielle

Le modèle inférentiel est légèrement complexe. Il est composé de deux modèles : le modèle inférentiel de l'encodeur `encoder_model` et le modèle inférentiel du décodeur `decoder_model`.

- le modèle de l'encodeur prend en entrée les mêmes types de donnée que l'encodeur évoqué précédemment et retourne les états (`h` et `c`) du LSTM ;
- le modèle du décodeur prend en entrée les mêmes type de donnée que le décodeur évoqué précédemment ainsi que deux états pour le LSTM.

En injectant les sorties de l'encodeur dans le décodeur on obtient les réponses. Ces deux modèles sont réalisés par la fonction `make_inference_models(encoder_inputs, encoder_states, decoder_embedding, decoder_lstm, decoder_dense, decoder_inputs)`.

Ainsi la fonction `ask_questions(enc_model, dec_model)` répond à une question de la manière suivante :

- les vecteurs d'états `h` et `c` sont obtenus à partir de `enc_model.predict` avec comme entrée la question tokenizée (par `str_to_tokens(sentence)` comme expliqué précédemment) ;
- un mot (ainsi que de nouveaux vecteurs d'états) est produit via `dec_model.predict` puis identifié avec une chaîne de caractère grâce à `tokenizer` ;
- ce procédé est répété (en utilisant les vecteurs d'états résultant à chaque fois) jusqu'à obtenir le token `<end>`.

#### 5.2.5 Sauvegarde des objets

Les différents objets clefs de cette implémentation peuvent être exportés (et importés) via les fonctions suivantes :

— `save_inference_model(path, encoder_inputs, encoder_states, decoder_embedding, decoder_lstm, decoder_dense, decoder_inputs)` qui sauvergarde le modèle inférentiel dans un fichier ;
— `save_tokenizer(path)` qui sauvegarde `tokenizer` via le module `pickle` dans un fichier donné ;
— `save_length(path)` qui sauvegarde la longueur maximale des questions et des réponses ;
— `load_inference_model(enc_file, dec_file)` qui retourne un modèle inférentiel à partir d'un fichier ;
— `load_tokenizer(tokenizer_file)` qui retourne un `tokenizer` à partir d'un fichier ;
— `load_length(length_file)` qui retourne la longueur maximale des questions et des réponses à partir d'un fichier.

## 5.3 Utilisation

### 5.3.1 Dépendances

Afin d'installer les dépendances Python, il est nécessaire d'exécuter cette commande :

```
$ pip install -r requirements.txt
```

### 5.3.2 Utilisation

Le script Python s'utilise via un système d'arguments qui peuvent être obtenus à partir de `$ python seq2seq.py -h`.

Ses options sont les suivantes :
— `--useFastText [True/False]` : est-ce que l'on utilise le Word2Vec de Facebook, FastText.
— `--downloadData [True/False]` : est-ce que le set de données doit être téléchargé.
— `--speak [True/False]` : active un mode interractif de test après que l'entraînement soit terminé.
— `--saveModel path` : dossier dans lequel soauvegarder le modèle.
— `--loadModel path` : dossier depuis lequel charger un modèle déjà entraîné.
— `--useSpatialDropout [True/False]` : utiliser ou non des couches de Dropout après les les couches d'embedding. Peut aider à réduire le surentraînement.
— `--useReccurentDropout [True/False]` : utiliser un dropout dans le LSTM. Effets variables, mais temps d'entraînement augmenté.
— `--useBatchNormalisation [True/False]` : utiliser des couches de Batch Normalisation. Peut aider à une meilleure généralisation du modèle, mais augmente la durée d'entraînement légèrement.
— `--vectorSize [True/False]` : taille des vecteurs de mot. Doit être consistant avec la taille des mots du Word2Vec.
— `--dataSize [True/False]` : fraction (en pourcentage) des données du dataset à utiliser pour l'entraînement. Par défaut, on utilise tout. Mettre une valeur différente de 100 est utile pour des tests rapides.
— `word2vec_model` : Paramètre **obligatoire**. Le Word2Vec utilisé pour l'entraînement.

```
usage: seq2seq.py [-h] --useFastText [True/False] [--downloadData [True/False]]
                  [--customData path] [--speak [True/False]] [--saveModel path]
                  [--loadModel path] [--useSpatialDropout [True/False]]
                  [--useReccurentDropout [True/False]]
                  [--useBatchNormalisation [True/False]] [--vectorSize size]
                  [--dataSize size]
                  word2vec_model
```

# 6 Interface graphique - GUI

## 6.1 Objectifs et outils

L'objectif de l'interface graphique (*Graphical User Interface*) est de recréer l'interface d'un système de communication tel que *Messenger* ou *WhatsApp* par exemple. Ce package (`org.lara.gui`), dont le code source individuel peut être retrouvé *ici*, est écrit à l'aide de la bibliothèque `JavaFX` et a été conçu en partie avec le logiciel *Scene Builder*. Il a pour mission d'être simple d'utilisation, et de permettre à l'utilisateur d'échanger avec une personne de son choix, ainsi que de changer d'interlocuteur et de choisir un surnom. La mise en page a été réalisée à l'aide de CSS.

## 6.2 Implémentation et classes

### 6.2.1 La classe `Main`

La classe `Main` permet de lancer l'interface graphique. Y sont définis les chemins des vues `Homepage.fxml` et `ChatFrame.fxml`. La classe `Main` est consitutée de la méthode usuelle d'un GUI : `start(Stage primaryStage)`.

### 6.2.2 La classe `HomepageCtrl`

La classe `HomepageCtrl` constitue le contrôleur de la vue `Homepage.fxml`, c'est-à-dire de la page d'accueil de l'interface graphique. Elle définit la méthode `chooseUsername(ActionEvent event)`, qui permet à l'utilisateur de définir un éventuel *username*, et qui permet le changement de vue, au profit de `ChatFrame.fxml`.

### 6.2.3 La classe `ChatFrameCtrl`

La classe `ChatFrameCtrl` constitue le contrôleur de la vue `ChatFrame.fxml`, c'est-à-dire de la fenêtre de chat de l'interface graphique. Elle définit les boutons et autres composants nécessaires, ainsi que les *handlers* associés :

- les méthodes `idLouis(ActionEvent event)`, `idAnna(ActionEvent event)`, `idRiad(ActionEvent event)`, et `idAntoine(ActionEvent event)` permettent de changer d'interlocuteur et d'effacer la conversation précédente, à partir de la méthode `idLara()` ;
- la méthode `exitLara(ActionEvent event)` permet d'éteindre le serveur et de quitter l'interface graphique ;
- il est possible d'envoyer un message en cliquant sur le bouton dédié, ou simplement en appuyant sur la touche *Entrée* du clavier, grâce aux méthodes `sendMessage(ActionEvent event)` et `sendKeyPressed(KeyEvent keyEvent)` qui utilisent la méthode `send()`.

### 6.2.4 Les vues

Les vues `Homepage.fxml` et `ChatFrame.fxml` permettent de définir la présentation de l'interface graphique.

### 6.2.5 La stylesheet `application.css`

Ce fichier CSS permet la mise en page des vues, et d'ainsi obtenir un résultat le plus proche possible des interfaces graphiques des systèmes de communication traditionnels.

## 6.3 Compilation, tests et utilisation

### 6.3.1 Compilation

Pour les utilisateurs d'*Oracle Java 8* ou d'une version plus récente, le JDK JavaFX est déjà compris dans le JRE, et il faudra alors simplement compiler les classes `Main`, `HomepageCtrl`, et `ChatFrameCtrl` à l'aide des instructions :

```
$ javac org.lara.gui.Main.java
$ javac org.lara.gui.HomepageCtrl.java
$ javac org.lara.gui.ChatFrameCtrl.java
```

Pour les utilisateurs d'*OpenJDK*, il faudra d'abord installer *OpenJFX* et l'inclure dans le classpath. Un tutoriel détaillé de l'installation est disponilble *ici*. L'instruction

```
$ sudo apt-get install openjfx
```

permet d'installer *OpenJFX* sous Ubuntu par exemple.

Pour les utilisateurs d'une version antérieure à *Oracle Java 8*, il faut télécharger le SDK JavaFX *ici*, et l'ajouter au classpath.

Ainsi, les utilisateurs d'*OpenJDK* ou d'une version de Java antérieure à *Java 8* compileront avec les instructions suivantes :

```
    $ javac -classpath "PATH_TO_JAVAFX_SDK/rt/lib/jfxrt.jar" org.lara.gui.Main.java
$ javac -classpath "PATH_TO_JAVAFX_SDK/rt/lib/jfxrt.jar" org.lara.gui.HomepageCtrl.java
$ javac -classpath "PATH_TO_JAVAFX_SDK/rt/lib/jfxrt.jar" org.lara.gui.ChatFrameCtrl.java
```

où il conviendra de remplacer `PATH_TO_JAVAFX_SDK` par le chemin du SDK JavaFX.

### 6.3.2 Utilisation

Les utilisateurs de *Oracle Java 8* ou d'une version plus récente lanceront le client Java par

```
$ java org.lara.gui.Main
```

Sinon, il conviendra d'utiliser l'instruction suivante :

```
$ java -classpath "PATH_TO_JAVAFX_SDK/rt/lib/jfxrt.jar:." org.lara.gui.Main
```

Il est alors demandé à l'utilisateur de saisir un `username`, puis de choisir un interlocuteur parmi quatre options. Les messages seront saisis dans le champ adapté, et peuvent être envoyés en appuyant sur le bouton *Send* ou sur la touche *Entrée* du clavier.

### 6.3.3 Utilisation avec Maven

Il est aussi possible de compiler le module via Maven

```
$ cd laraGUI
$ mvn compile
```

et de l'exécuter via

```
$ mvn javafx:run
```

# 7 Utilisation de LARA

Cette section détaille comment utiliser le logiciel à partir de 0.

## 7.1 Téléchargement des données Facebook

Afin de télécharger les conversations Messenger, il faut naviguer jusqu'à la page `Télécharger vos informations` (`Paramètres` ⟶ `Vos informations Facebook` ⟶ `Télécharger vos informations`). Une fois arrivé, il faut saisir `Format : JSON` et `Qualité des photos : faible`. Dans la liste `Vos données`, il faut uniquement cocher `Messages`.

Facebook vous enverra une notification une fois votre archive prête (cela peut mettre plusieurs heures) qu'il vous faudra télécharger afin de passer à la suite.

## 7.2 Mise en place de l'environnement Python

A partir de cette étape, nous imaginerons que nous travaillons dans le dossier `LARA`. Nous allons mettre en place un environnement virtuel pour nos différents scripts Python :

```
$ pip install virtualenv
   $ virtualenv venv
$ source venv/bin/activate
```

## 7.3 Téléchargement de LARA un installation des dépendances

Commençons par télécharger LARA :

```
git clone https://github.com/LaraProject/lara
```

après `$ cd lara`, installer protobuf et maven :

```
$ sudo snap install protobuf --classic
    $ sudo apt-get install maven
```

compiler le protobuf

```
$ make
```

installer les dépendances python :

```
$ pip install -r requirements.txt
```

compiler le package Java :

```
$ mvn package
```

et enfin `$ cd ..`

## 7.4 Parsing des données personnelles avec FacebookParser

Tout d'abord, il faut décompresser l'archive Facebook dans un dossier nommé `facebook-data`. Ensuite, il faut créer les dossiers de sorties et d'entrées des différents scripts :

```
$ mkdir parser_out
$ mkdir nlp_out
```

Puis il faut exécuter le parser Facebook sur chaque conversation :

```
$ ./lara/script_parsing.sh facebook-data parser_out "Prénom Nom" lara/parserFB.py
```

où "Prénom Nom" correspond au profil Facebook dont est issu le dump Facebook.

## 7.5 Traitement des données personnelles avec le NLP

Tout d'abord, il faut traiter toutes les conversations :

```
$ ./lara/script_java_format.sh parser_out nlp_out "Prénom Nom" lara/target/laraproject-*.jar
```

Vous pouvez ensuite tout regrouper dans un même fichier :

```
$ cat nlp_out/* > data_facebook_cleaned.txt
```

## 7.6 Traitement des données LELU avec le NLP

Cette section est très très longue. Il est possible de télécharger le fichier data_lelu_cleaned.txt (compressé avec lrzip) directement. Dans le cas contraire, voici les différentes étapes. Il faut tout d'abord télécharger ici le fichier `final_SPF_2.xml`. Puis télécharger le parser :

```
$ git clone https://github.com/LaraProject/parserLELU
$ mv final_SPF_2.xml parserLELU/
```

puis exécuter le script Python :

```
$ pip install lxml
$ python parserLELU/parserLELU.py > data_lelu_raw.txt
```

puis de le nettoyer à l'aide du NLP (étape la plus longue) :

```
$ java -cp lara/target/laraproject-*.jar org.lara.nlp.context.SimpleTest data_lelu_raw.txt 0
                                999999 data_lelu_cleaned.txt
```

## 7.7 Entraînement du modèle Word2Vec

Comme détaillé plus tôt, cette étape peut être réalisée de plusieurs manières différentes : avec Python ou avec Java. Il faut dans tous les cas (sauf si vous utilisez Colab) commencer par combiner les données Facebook et LELU :

```
$ cat data_lelu_cleaned.txt data_facebook_cleaned.txt > data.txt
```

### 7.7.1 Avec Java

On utilise la classe `W2vSimpleTest` comme suit :

```
$ java -cp lara/target/laraproject-*.jar org.lara.nlp.word2vec.W2vSimpleTest data.txt 0
                                999999 word2vec_vectors.txt
```

puis la convertir dans le bon format :

```
$ ./lara/gensim_convert.sh word2vec_vectors.txt
```

### 7.7.2 Avec Python

On utilise le script `fasttext.py` :

```
$ python lara/fasttext.py --path word2vec_vectors.txt data.txt
```

Il est possible (**recommandé**) d'exporter l'intégralité du modèle `FastText` de `gensim` avec l'argument `--modelPath path` où `path` est le préfixe (avec le chemin) des fichiers d'exportation.

### 7.7.3 Avec Python via Google Collab

Il suffit d'importer ce notebook qui contient toutes les instructions. **Inutile de télécharger** `data_lelu_cleaned.txt` **le notebook s'en charge lui même**.

## 7.8 Entraînement du RNN

### 7.8.1 Sur son ordinateur personnel

Il faut tout d'abord créer le dossier d'exportation `$ mkdir model_save`. Ensuite, il y a plusieurs approches disponibles pour le chargement du modèle Word2Vec :

— Si vous avez le fichier `word2vec_vectors.txt` vous pouvez utiliser l'approche 1, c'est à dire :

```
$ python lara/seq2seq.py --customData data_facebook_cleaned.txt --saveModel model_save
                         word2vec_vectors.txt --useFastText False
```

où `word2vec_vectors.txt` est le chemin du fichier exporté par le NLP ;

— Si vous avez les 7 fichiers de la forme `word2vec_model.bin*` vous pouvez utiliser l'approche 2

```
$ python lara/seq2seq.py --customData data_facebook_cleaned.txt --saveModel model_save
                           word2vec_model.bin --useFastText True
```

où `word2vec_model.bin` est le préfixe (comprenant le chemin) des 7 fichiers correspondant au modèle FastText exporté.

Il faut choisir entre l'approche 1 et l'approche 2. L'approche 2 est significativement plus performante mais plus lente (5h30 d'entraînement). Cette commande va entraîner le RNN et sauvegarder le modèle dans le dossier `model_save`. En cas de problème de taille de tensor, il est possible de diminuer la taille du dataset utilisée en spécifiant un pourcentage via `--dataSize`.

### 7.8.2 Via Google Collab

Il suffit d'importer ce notebook qui contient toutes les instructions.

## 7.9 Intégration des fichiers exportés par le RNN dans LARA

A partir des fichiers exportés par le RNN, il faut former l'arborescence suivante

```
lara/models/##/model_enc.h5
lara/models/##/model_dec.h5
lara/models/##/tokenizer.pickle
lara/models/##/length.txt
```

où `##` correspond au numéro de la personne (c'est numéro commence à 1, si aucun numéro n'a été explicitement attribué, il faut mettre le numéro existant +1).

## 7.10 Exécution et accès à l'interface graphique

Afin d'exécuter LARA, il faut ouvrir deux terminaux distincts. Attention dans chaque terminal veillez à être dans le bon dossier `$ LARA/lara` et d'utiliser l'environnement virtuel python `$ source ../venv/bin/activate`.

### 7.10.1 Premier terminal : le serveur

Pour allumer le serveur, il suffit d'exécuter le script python depuis le dossier `python` :

```
$ cd python
$ python main.py
```

En cas de problème de port déjà occupé, il peut être changé aux lignes suivantes `lara/src/main/java/org/lara/rnn/Server.java:11` (côté client) et `python/main.py:70` (côté serveur).

### 7.10.2 Second terminal : le client

Il suffit d'utiliser maven pour appeler le script JavaFX :

```
$ mvn javafx:run
```

Il vous est désormais possible de discuter avec les membres de LARA.