"La génération de paraphrases de données d'entrainement pour une meilleure classification des intents d'un chatbot créé avec Rasa"

Cheramy, Jean

ABSTRACT

À l'ère d'Internet et de l'omniprésence numérique, les chatbots émergent et prennent place dans l'espace virtuel. Ils sont implantés dans les messageries instantanées ainsi que sur des sites web. Ils proposent leurs services aux internautes, mais également aux entreprises. Ainsi, leurs rôles vont de la recherche d'informations à la mise en place d'un service clientèle interactif. Dotés de compétences linguistiques, leur efficacité (ou « intelligence ») est bien souvent appréciée selon leur capacité à reconnaitre efficacement les intentions de l'utilisateur. En effet, une mauvaise décision de la part du chatbot quant aux volontés de l'internaute mène inévitablement à des dialogues de sourds, conduisant l'utilisateur à déserter la conversation. Afin de classifier correctement ces intentions, ou « intents », le chatbot dispose d'une base de données NLU (Natural Language Understanding). Cette dernière est composée d'énoncés reliés à des intents, permet...

CITE THIS VERSION

UCLouvain

FIAL
Un éventail de formations, une multitude d'horizons

Faculté de philosophie, arts et lettres

La génération de paraphrases de données d'entrainement pour une meilleure classification des intents d'un chatbot créé avec Rasa

Auteur : Jean Cheramy
Promoteur(s) : Thomas François
Lecteur(s) : Cédrick Fairon
Année académique 2018-2019
Master en Linguistique, finalité en traitement automatique du langage

Remerciements

En avant-propos, je souhaite remercier certaines personnes qui m'ont aidé à construire et rédiger ce mémoire.

Je remercie mon promoteur, Thomas François, pour le suivi et les conseils qu'il m'a prodigués durant ces deux dernières années.

Je remercie également M. Hubert Naets pour sa disponibilité, sa patience et ses conseils avisés.

Un mémoire n'est rien sans de bons relecteurs, je remercie donc mes amis Alexandre Henry et Marie Bayot pour leurs avis à la fois pertinents, lucides et indispensables. Je présente également des remerciements spéciaux à Clémence qui a eu à la fois le courage, et la bonté, de me supporter quotidiennement lors de la rédaction de ce mémoire. Ses relectures ainsi que celles de mes parents m'ont également été d'un grand secours.

Enfin, je remercie mes parents. Ces derniers m'ont toujours prodigué, avec amour, leur soutien et leur inébranlable confiance. Ils ont toujours eu à coeur d'offrir le meilleur pour leurs enfants. Ce mémoire, qui représente l'ultime étape de ma scolarité, leur est dédié.

Première partie

Table des matières

Introduction

Les chatbots sont des systèmes conversationnels automatiques dont la naissance remonte aux années 60. Ils sont actuellement en plein essor et profitent d'un climat particulièrement favorable. La très grande disponibilité d'Internet couplée à la généralisation des *smartphones* et des technologies de 4G en font un outil particulièrement visible dans la sphère technologique mondiale.

Ils peuvent être utilisés aussi bien par des particuliers que par des entreprises souhaitant intégrer un service client automatisé ou un assistant virtuel sur leur site web. Les chatbots sont en effet très facilement intégrés aux plateformes de messagerie instantanée telles que Messenger ou Telegram. Leur intégration au sein d'un site web est également aisé. L'apparition d'un effet de mode vers les années 2010 [1] a entrainé l'émergence de nombreux outils facilitant la création et la maintenance de tels systèmes. Certains, tels que *DialogFlow* [2], appartiennent à des entreprises privées. D'autres, tels que Rasa [3], sont *open source* et disponibles pour tout le monde.

Les chatbots permettent, par exemple, d'alléger la pression sur le service clientèle en répondant aux questions les plus fréquentes des internautes. Ces derniers en profitent également étant donné que ce type de service est disponible de façon continue et n'est pas tributaire d'un horaire. L'utilisateur est alors servi beaucoup plus rapidement et en toutes circonstances. Cependant, cette technologie ne vaut pas l'expertise propre à un être humain et se voit donc attribuer des tâches le plus souvent bien délimitées. En effet, les chatbots ne sont pas « intelligents » au sens humain du terme. Leur habileté à déterminer l'intention de l'utilisateur et à apporter une réponse pertinente est tributaire de la qualité du système, de façon générale, mais surtout de celle de la base de données.

Cette dernière répertorie des énoncés, des exemples linguistiques, et les attribue à une intention de communication bien particulière, un *intent*. Au plus ces données sont larges et diversifiées, au mieux le système pourra reconnaitre

1. Siri a été lancé en 2011.
2. https ://dialogflow.com/
3. https ://rasa.com/

efficacement les intentions de l'internaute. En effet, nous postulons que l'élaboration d'une base de données comportant des exemples variés et linguistiquement pertinents ne peut qu'être bénéfique pour la robustesse du chatbot. Le pouvoir de généralisation d'un tel système dépend en effet des données qu'il contient. Cependant, insuffler de la diversité linguistique à un système le plus souvent élaboré manuellement est une tâche très complexe. En effet, le développeur crée des exemples qui ne reflètent qu'un unique prisme linguistique. Le chatbot peut alors souffrir d'un manque de performance face à des énoncés structurellement différents, lexicalement étrangers ou provenant d'une strate sociolinguistique bien particulière. Les langues sont en effet composées de locuteurs bien différents, disposant chacun d'un répertoire linguistique particulier.

C'est à partir de ce constat que nous avons déterminé le sujet de notre recherche. Nous estimons qu'il serait profitable de générer des données variées de façon automatique. En effet, le gain de temps et d'énergie serait alors conséquent. Notre problématique consiste à envisager la génération automatique de paraphrases depuis des données d'entrainement afin d'améliorer les performances d'un classificateur d'*intents*.

Nous envisagerons d'abord notre recherche d'un point de vue théorique, en définissant explicitement ce qu'est un chatbot et en explorant les thématiques que sont la classification d'*intents*, les techniques de surgénération de données au sein des chatbots et la génération automatique de paraphrases. Nous l'envisagerons également d'un point de vue pratique en implémentant un système de génération automatique de paraphrases. Nous explorerons deux approches distinctes pour générer des paraphrases depuis les données d'entrainement : l'utilisation d'un dictionnaire de synonymes pour effectuer des remplacements lexicaux et la mise en place d'un système de remplacement de syntagmes depuis un grand corpus de paraphrases. Finalement, nous évaluerons la qualité des paraphrases générées et nous déterminerons les effets de nos approches lors d'une expérimentation. Cette dernière consistera en la création d'un chatbot minimal avec Rasa et l'utilisation des données créées automatiquement pour déterminer leur influence sur les performances des classificateurs d'*intents*.

Cette recherche a pour but l'exploration de la génération automatique de données en vue d'améliorer la capacité de généralisation et les performances d'un chatbot. Nous souhaitons démontrer que des techniques basées sur des données *open source*[4] peuvent se révéler intéressantes, de façon générale et non spécifique, pour l'optimisation de chatbots. En effet, l'avantage d'utiliser un dictionnaire de synonymes ou un corpus (général) de paraphrases est son pouvoir de généralisation. Si l'une de ces approches s'avère concluante pour l'amélioration des performances d'un chatbot spécifique, il est permis de supposer qu'elle le serait également pour d'autres.

4. Le dictionnaire de synonymes et le corpus de paraphrases.

Chapitre 1

État de l'art

Cet état de l'art a pour ambition de dépeindre l'ensemble du paysage technologique impliqué dans notre problématique. Celle-ci consiste à envisager la surgénération de paraphrases de données d'entrainement dans le but de constater ses effets sur la classification des *intents* d'un chatbot créé avec un outil *open source* : Rasa.

Nous commencerons par définir ce qu'est exactement un chatbot, son histoire, sa composition et son fonctionnement global. Nous nous intéresserons ensuite à la tâche de classification des *intents*, que nous définirons avec des fondements linguistiques. Nous présenterons par la suite les diverses techniques mises en œuvre pour créer des classificateurs dans le cadre de cette tâche.

Un aperçu des diverses techniques utilisées pour générer des données d'entrainement sera ensuite présenté. Nous nous focaliserons enfin sur la génération, automatique, de paraphrases.

1.1 Définition du chatbot

Un chatbot est « un agent conversationnel qui interagit avec des utilisateurs dans un certain domaine ou sur un thème précis avec des phrases en langue naturelle » (Huang et al., 2007) [1]. Cette définition générale pourrait sembler suffisante, mais il s'avère que la notion de chatbot est compliquée à définir, d'autant plus lorsqu'elle est mise en parallèle avec les systèmes de *Question-Answering* (QA), les systèmes de dialogues, les assistants personnels, etc. Cette section aura alors pour tâche de clarifier le sujet d'étude et de sélectionner ou d'élaborer une définition servant de base de travail.

1. A chatbot is a conversational agent that interacts with users in a certain domain or on a certain topic with natural language sentences.

1.1.1 Une dimension « intelligente », voire « vivante »

Selon Radziwill and Benton (2017), « les chatbots sont [...] une forme de logiciel conversationnel intelligent activé par un *input* linguistique (écrit, oral ou les deux). Ils donnent en sortie des réponses et ils peuvent également, à la demande, exécuter des tâches » [2]. La dimension de l'intelligence est ici mise en avant, l'incluant dans le domaine plus vaste de l'intelligence artificielle. Rahman et al. (2017) iront même plus loin en postulant qu'un chatbot « est une personne virtuelle qui peut efficacement parler à n'importe quel humain en utilisant des compétences textuelles interactives » [3]. Radziwill and Benton (2017) estiment quant à eux que les chatbots sont des « agents intelligents et beaucoup d'entre eux ont été développés avec des identités humaines et même des personnalités » [4]. Ces définitions relèvent davantage du fantasme que de la réalité : établir un parallèle entre un logiciel informatique et une personne, une identité humaine, voire même une personnalité, nous semble scabreux et difficilement soutenable. Il en va de même des travaux (Bapat et al., 2018) qui définissent les chatbots comme « des programmes vivant dans les applications de messagerie et permettant de créer une conversation avec un humain pour fournir un certain service » [5]. Leur attribuer une intelligence, un statut proche de l'individualité (*virtual person*, identité humaine) et les considérer comme des êtres vivants nous semble abusif et bien loin de la réalité. Cependant, la mode actuelle est de qualifier des outils de *machine learning* comme étant « intelligents ». Ce terme, qui semble d'ailleurs plus compliqué à définir que le chatbot (Warner, 2002), se prête assez mal, selon nous, à la description de logiciels informatiques. Nous pouvons cependant retenir qu'un chatbot, à défaut d'être humain, « stimule des conversations intelligentes avec des humains » (Manaswi, 2018) [6] et doit donc parler de façon naturelle.

1.1.2 En opposition à d'autres systèmes

Différencier le chatbot d'autres systèmes pourrait, a priori, être une tâche somme toute banale. Il n'en est cependant rien. Nous avons pu constater qu'une certaine confusion règne dans le domaine des agents conversationnels et qu'une typologie commune n'est pas partagée. Un essai a bien été tenté (Radziwill and

2. Chatbots are one class of intelligent, conversational software agents activated by natural language input (wich can be in the form of text, voice, or both). They provide conversational output in response, and if commanded, can sometimes also execute tasks.

3. Chatbot is a virtual person who can effectively talk to any human being using interactive textual skills.

4. [...] intelligent agents, many of which have been developed with human identities and even personalities.

5. chatbots [...] are computer programs living in messenger applications and emulating a conversation with a human to provide a certain service.

6. A chatbot stimulates intelligent conversations with humans [...]

Benton, 2017) et donne à voir plusieurs catégories.

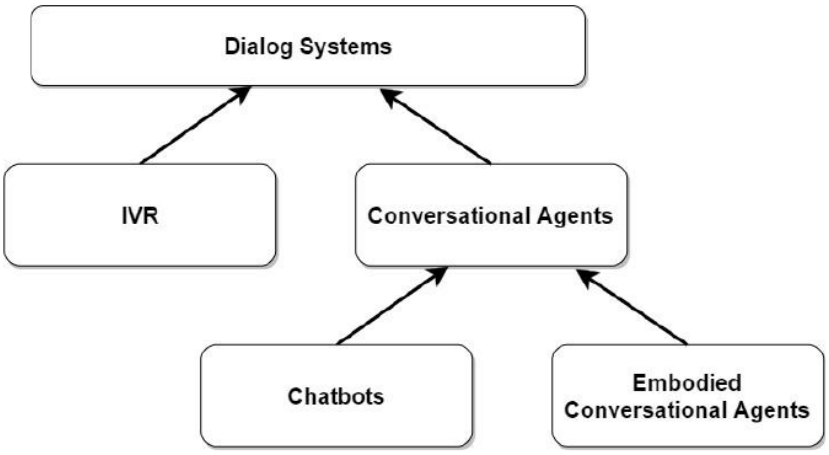


FIGURE 1.1 – Taxonomie (Radziwill and Benton, 2017)

— « Système de dialogue » serait un terme générique englobant les autres ;
— IVR représente les *Interactive Voice Response systems* (« press or say 1 for english ») qui ne comportent pas de dimension réellement conversationnelle ou « intelligente » ;
— Les agents conversationnels désigneraient à la fois les chatbots et les agents « embodied », représentés par un avatar (animal ou visage).

Cette typologie pourrait être satisfaisante si elle n'omettait pas les systèmes de *Question-Answering* (QA) et leurs déclinaisons (*single-turn* et *multi-turn*). En effet, les systèmes de QA se différencieraient du chatbot sur plusieurs points. Tout d'abord, l'objectif n'est pas le même. Un QA « offre une interface agréable à l'humain pour naviguer à travers des données et de la connaissance qui peuvent être encyclopédiques ou spécialisées » (Mensio et al., 2018) [7]. Le chatbot ne comporte pas, à première vue, cet objectif revendiqué de seulement donner accès à des données bien définies. Cependant, « généralement, un système QA est créé pour donner une réponse à une question spécifique (*single-turn*). Cependant, [certains QA] peu[vent][...] collecter les informations nécessaires à la génération d'une réponse finale qui compile les différents tours de parole (*multi-turn*) » (Mensio et al., 2018) [8]. La frontière entre chatbot et QA s'amincit de par la

7. QA systems offer a human friendly interface to navigate through knowledge, wich can range from encyclopedic to domain-specific.

8. Generally, a QA system is designed to provide an answer to a specific question once (so-called single-turn) [...]. However, [QA systems can also] collect the necessary information and generate a compelling final answer through multiple turns.

possibilité de créer un fil de discussion en plusieurs tours de parole.

D'autres appellations viennent ensuite se greffer à cela. En effet, Yu et al. (2016) estiment que « contrairement aux systèmes de dialogues orientés [en fonction d'un objectif conversationnel] (*goal-oriented*), les chatbots n'ont pas de but spécifique qui guide leurs interactions. [...] la même conversation ne pourrait pas arriver plus d'une fois [et] il n'y a pas de réelle aboutissement de la discussion » [9]. ELIZA (Weizenbaum et al., 1966) ou encore cleverbot [10] sont des exemples de ce type de chatbot qui n'a pas de parcours prédéfinis. Cette définition va à l'encontre de l'utilisation du terme chatbot sur Internet, là où ils pullulent sur les messageries instantanées en proposant des services bien définis et très orientés.

William Meisel distingue les assistants personnels généraux comme Siri des très nombreux chatbots plus spécifiques qu'il appelle les « assistants digitaux spécialisés » ou *specialized digital assistants* (Dale, 2016). Cortana (Windows), Siri (Apple), Alexa (Amazon) ou encore le Google assistant permettent de bloquer des dates dans un agenda, de lire et d'écrire des mails, de jouer de la musique et de contrôler des objets domotiques. Cependant, ils sont rarement mis en lien avec l'univers des chatbots et le terme « assistant personnel » leur est préféré. Nous ne les considérerons pas comme étant des chatbots à proprement parler pour certaines raisons : l'absence d'objectif clairement défini de ce type d'assistant, leur caractère très général (et non spécifique), etc.

De nombreuses définitions mettent aussi en relation le chatbot avec les *Software agents* ou encore les *Virtual Agents* (Kar and Haldar, 2016).

Il n'est pas aisé de différencier complètement ces systèmes qui tendent à partager certaines caractéristiques. Nous pouvons tenter de retenir qu'un chatbot serait un logiciel, pas obligatoirement muni d'un avatar, ayant comme objectif de tenir une conversation (en plusieurs tours de parole) en langue naturelle sur un sujet le plus souvent précis tout en se faisant passer pour un humain. Cette définition nous semble encore trop générale et il lui manque certains éléments.

1.1.3 Objectifs

Les chatbots, considérés comme ayant des objectifs précis, peuvent être utilisés dans de nombreuses tâches (Stöckl, 2017 ; Manaswi, 2018 ; Radziwill and Benton, 2017) :

— Réduire le délai de réponse aux questions ;
— Améliorer le service client ;
— Rechercher des informations ;
— Guider l'utilisateur sur un site ;

9. Unlike goal oriented dialog systems, chatbots do not have any specific goal that guides the interaction. [...] the same conversation will not occur more than once [...][and] there is no clear sense of when such a conversation is "complete".
10. https ://www.cleverbot.com/

— Commander des produits en ligne (Alexa d'Amazon).

Bien souvent, ils se limitent à ces tâches et ne sont pas conçus pour s'en éloigner. Une dimension de rapidité et d'efficacité émerge à de nombreuses reprises : un chatbot doit être rapide et efficace (Rahman et al., 2017). Ils « s'imposent comme étant la manière la plus pratique de s'occuper des consommateurs de façon opportune et satisfaisante » (Manaswi, 2018) [11].

Cependant, ces systèmes peuvent également servir à des fins malveillantes telles que la diffusion de *fake news*, l'augmentation artificielle du nombre de *followers* ou l'intimidation d'internautes exprimant certains idéaux politiques (Radziwill and Benton, 2017).

1.1.4 Interfaces

Les « communications peuvent être faites par le biais de l'écriture mais aussi avec un système de reconnaissance de la parole » (Stöckl, 2017) [12]. La reconnaissance de la parole peut être vue comme une surcouche au niveau textuel. En effet, son fonctionnement implique la conversion des sons en texte. De nombreuses plateformes de messagerie instantanée permettent à ces logiciels de facilement s'intégrer dans des interfaces : Messenger, Slack, Telegram ou WhatsApp. L'interface écrite est donc très souvent identique, elle se calque sur le fonctionnement des messageries instantanées.

1.1.5 Synthèse

Nous proposons d'introduire une définition explicite de ce qu'est un chatbot, d'après nos lectures et recherches sur le sujet :

> Un chatbot est un logiciel, muni ou non d'un avatar, ayant pour objectif de tenir une conversation de plusieurs tours de parole dans un but déterminé en utilisant les langues naturelles sur un sujet le plus souvent précis, le tout en donnant le sentiment à l'utilisateur de parler avec un humain.

Cette définition prend en compte les caractéristiques les plus importantes que nous avons trouvées. Elle est assez permissive par rapport aux ambigüités résiduelles avec, par exemple, un QA à plusieurs tours de parole qui pourrait se superposer assez bien, selon son comportement, à ces critères. Nous estimons cependant avoir affiné notre vision du sujet de recherche.

11. They are emerging as as the most convenient way of dealing with consumers in a timely and satisfactory manner.

12. These communications can be done by writing text with the keyboard or with a speech-recognition system.

1.2 Histoire des chatbots

L'origine des chatbots remonte à 1966 avec l'invention d'ELIZA par Weizenbaum et al. (1966). ELIZA imitait le comportement linguistique d'un psychologue, posant beaucoup de questions pour faire parler l'interlocuteur. Son système reposait principalement sur l'identification de mots-clés, des règles d'assemblage et une génération de phrases. L'objectif premier de ce type d'invention était de réussir le test de Turing : celui-ci nécessitait que le programme informatique soit capable de se faire passer pour un humain. Les systèmes tels qu'ELIZA n'étaient pas assez performants pour cela. De nombreuses tentatives eurent lieu sans pour autant passer le test.

Plus tard, Wallace (1995) mit au point ALICE (*Artificial linguistic internet computer entity*). Ce chatbot « n'est pas devenu emblématique pour ses capacités conversationnelles, mais parce qu'il a mené au développement du *Artificial Intelligence Markup Language* (AIML). AIML est utilisé pour déclarer des règles de *pattern-matching* qui lient les soumissions de l'utilisateur (mots et phrases) à des catégories. Ce format, basé sur le XML, est très utilisé de nos jours dans les plateformes et services de chatbots » (Radziwill and Benton, 2017) [13]. ALICE gagnera le célèbre prix Loebner en 2000 et en 2001. Par la suite, de nombreux chatbots seront développés sur la base du *framework* d'ALICE (Wu et al., 2008).

Le prix Loebner a été créé en 1990 par Hugh Loebner [14] avec pour principe d'organiser un concours sur la base du test de Turing. Pour rappel, ce dernier vise à déterminer si le chatbot est un robot ou un humain. Le test est réussi si des humains se font tromper et estiment avoir affaire à l'un de leurs congénères. Comme tout concours, celui-ci a ses critiques. Marvin Minsky n'y voit, quant à lui, qu'un coup de publicité [15]. Durant la même période, les robots IRC (*Internet Relay Chat*), similaires aux chatbots, étaient très courants (Stoecklé, 2017).

Aujourd'hui, les chatbots profitent de l'accessibilité grandissante et globale d'Internet, de la popularisation des *smartphones*, du wifi, des technologies de 4G et bientôt de 5G. C'est dans un monde ultraconnecté qu'ils prennent place et s'incorporent. Gupta et al. (2019) estiment que nous sommes dans « l'ère » du chatbot. Ces derniers, entre 2007 et 2015, ont été impliqués dans un tiers ou la moitié de toutes les interactions en ligne (Radziwill and Benton, 2017). Ils sont présents dans les messageries instantanées : WhatsApp (1,5 milliard d'utilisateurs), Messenger (1,3 milliard), Telegram (200 millions) mais également sur les sites web d'entreprises voire même de particuliers. Que ce soit sur tablette,

13. ALICE [...] becoming significant not for its conversational capabilities but because it led to the development of Artificial Intelligence Markup Language (AIML). AIML is used to declare pattern-matching rules that links user-submitted words and phrases with topic categories. It is eXtensible Markup Language (XML) based, and supports most chatbot platforms and services in use today.

14. https ://fr.wikipedia.org/wiki/Prix_Loebner

15. http ://boowiki.info/art/intelligence-artificielle/prix-uuloebner.html

ordinateur ou *smartphone*, tous les appareils y ont accès. Cette facilité d'intégration dans le matériau numérique confère à cette technologie l'opportunité de se propager et de toucher une grande part de la population.

William Meisel, figure connue dans l'univers du traitement automatique des langues, estime que le chiffre d'affaires de cette technologie devrait atteindre les 623 milliards de dollars en 2020 (Dale, 2016). Chris Messina, de Uber, déclarait que 2016 était l'année du commerce conversationnel. Satya Nadella, CEO de Microsoft, annonçait alors que les chatbots étaient le « next big thing ». Mark Zuckerberg, quant à lui, a proclamé que les chatbots étaient la solution à la surcharge des applications. Il y a, aujourd'hui, un très grand nombre de chatbots répondant à divers problèmes (Dale, 2016). Ainsi, sur *Pandarabots*[16], une plateforme de création de chatbots, sont recensés[17] plus de :

— 250 000 développeurs ;
— 300 000 chatbots ;
— 60 000 000 000 d'interactions.

Parallèlement à cette plateforme spécifique, Messenger accueille plus de 300 000 chatbots. Microsoft compte également des dizaines de milliers de développeurs sur Skype.

L'engouement est donc chiffré et semble d'autant plus imposant. Natya Nadella (Microsoft) ira jusqu'à dire, sur le modèle de Zuckerberg, que « chatbots are the new apps » (Brandtzaeg and Følstad, 2017). En effet, les conversations avec un chatbot semblent être, pour l'utilisateur, un moyen plus naturel d'interagir (Brandtzaeg and Følstad, 2017). Certains chatbots gagnent alors en renommée, tels que Mitsuku (prix Loebner de 2013 et 2016), DoNotPay (qui se charge d'aider les usagers à faire sauter les procès-verbaux), Babylon Health (donne des conseils médicaux), etc. BotList[18] donne à voir un très grand échantillon des chatbots disponibles et de leurs objectifs. Les raisons d'utiliser un chatbot peuvent être très nombreuses et très diversifiées. Cependant, elles se classent le plus souvent dans ces catégories (Brandtzaeg and Følstad, 2017) :

— Productivité ;
— Divertissement ;
— Social ;
— Nouveauté.

Les quatre systèmes apparentés au chatbot et connus mondialement sont : Siri, Alexa, Cortana et Google Assistant. Cependant, comme nous pouvons le constater, « ces quatre ambassadeurs de la technologie conversationnelle sont seulement le sommet de l'iceberg » (Dale, 2016)[19]. De plus, « le succès des chatbots

16. https ://home.pandorabots.com/home.html
17. Au mois de juin 2019.
18. https ://botlist.co/
19. But these four ambassadors for conversational technology are really just the tip of the iceberg.

est déjà une réalité en Asie. Sur WeChat, il est possible de faire des virements, de participer à des loteries, et d'acheter des tickets de cinéma via des bots » (Stoecklé, 2017).

> « Very soon we'll be in a world where some of those conversational partners we'll know to be humans, some we'll know to be bots, and probably some we don't know either way, and may not even care » (Dale, 2016).

1.3 Composition et fonctionnement d'un chatbot

Une brève présentation du fonctionnement d'un chatbot nous semble indispensable, bien que nous y reviendrons en profondeur par la suite. Les notions d'*intents* et d'entités seront brièvement expliquées avant de présenter l'importance de la base de données.

1.3.1 Général

Un chatbot est principalement composé de deux parties :

— Un composant chargé de traiter le « Natural Language Understanding » ou NLU et d'extraire les *intents*[20] (ou intentions d'utilisateur) et les *entities* (entités) ;

— Un module chargé de s'occuper du « Dialog Management ». Il a donc pour tâche l'exécution des actions, le maintien du dialogue et le suivi de la discussion.

Ce travail se concentrera sur la partie NLU. En effet, celle-ci est primordiale car elle permet de capter la volonté de l'utilisateur et donc de donner les bonnes indications pour la gestion du dialogue. L'extraction des entités survient généralement après la reconnaissance d'*intents* mais s'opère parfois simultanément (Xu and Sarikaya, 2013 ; Liu and Lane, 2016).

1.3.2 Intents

L'*intent* extrait par le système doit impérativement correspondre aux volontés de l'utilisateur. En effet, une mauvaise classification des *intents* peut mener à des discussions n'ayant que très peu de sens, le chatbot répondant à tort et à travers. Par exemple, si le chatbot a pour rôle de conseiller l'internaute sur l'achat d'une voiture, il devra être en mesure de classer des *intents* tels que :

— prix_voiture ;

— couleur_voiture ;

20. Nous utiliserons le terme anglais tout au long de ce mémoire afin de ne pas créer d'ambigüité.

— caractéristiques_voiture ;

Ces *intents* représenteraient l'intention de l'internaute de connaitre le prix, la couleur ou les caractéristiques d'une voiture. Le chatbot, étant capable de reconnaitre ces *intents* dans les messages d'un utilisateur, peut ensuite déclencher une action pour répondre au besoin. L'envoi d'une réponse ou une redirection internet en sont des exemples.

Une classification d'*intents* précise et robuste est donc primordiale pour être en mesure de proposer un chatbot efficace.

1.3.3 Entités

La recherche d'entités dans des messages de chatbot se rattache au domaine, bien étudié, de l'extraction d'entités nommées.

> « La tâche d'extraction entités nommées se compose de trois sous-tâches (noms d'entités, expressions temporelles, expressions de nombre). Les expressions à annoter sont les « identificateurs uniques » des entités (organisations, personnes, lieux), des heures (dates, heures) et des quantités (valeurs monétaires, pourcentages) » (Chinchor and Robinson, 1997) [21].

Cependant, dans le cadre de chatbots pouvant être inscrits dans des domaines très variés, la définition s'élargit aux besoins des développeurs. Les entités représentent en effet des variables qui sont nécessaires au bon fonctionnement de certaines fonctionnalités d'un chatbot. Par exemple, si la tâche de l'agent conversationnel est de proposer des restaurants, il devra extraire les caractéristiques qui intéressent l'utilisateur :

— Le type de cuisine : chinois, italien, français ;
— L'emplacement : Paris, Toulouse ;
— La date, l'heure.

Pour ce type de phrase : « Je souhaiterais manger chinois aujourd'hui », il doit être en mesure d'extraire [chinois] et [aujourd'hui] pour proposer des réponses adéquates. Cependant, pour pouvoir extraire les bonnes entités, il doit être avant tout capable de reconnaitre la bonne intention. La classification de ces dernières est permise grâce à un modèle entrainé sur base d'un jeu de données représentatif.

21. The Named Entity task consists of three subtasks (entity names, temporal expressions, number expressions). The expressions to be annotated are "unique identifiers" of entities (organizations, persons, locations), times (dates, times), and quantities (monetary values, percentages).

1.3.4 Base de données

Tout modèle de *machine learning* nécessite une base de données pour donner lieu à un entrainement. Plus ces données sont vastes et représentatives, plus les probabilités de classer correctement les messages s'améliorent. Généralement, la base de données s'organise autour des *intents* et donne à voir pour chaque *intent* des exemples de phrases le contenant. Pour un *intent* « prix_voiture », des exemples de phrase pourraient être :

— Quel est le prix de la [Mazda 6] [22] ?
— À combien est la [Mazda 6] ?
— Combien coute une [Peugeot 206] ?

L'intention est identique, mais la façon de la formuler peut fortement varier d'un point de vue lexical, orthographique, syntaxique (inversion), etc. Il est donc primordial de créer ou de trouver un jeu de données correspondant à ces *intents* et suffisamment vaste pour donner au modèle une bonne robustesse face aux variations linguistiques. L'inconvénient, avec ces données sur mesure, est qu'il faut bien souvent les créer manuellement. Cette démarche est couteuse, mais également très chronophage. Nous verrons par après que de nombreuses stratégies ont été mises en place pour pallier ce problème. C'est en effet grâce à la qualité de ces jeux de données qu'une bonne classification d'*intents* est possible. La section suivante définira plus exactement cette tâche et brossera un panorama des différentes stratégies mises en place dans la littérature pour mener à bien cette classification.

1.4 Classification des intents

La classification d'*intents* est un problème bien étudié. Il s'agit, en effet, d'un mécanisme essentiel au bon fonctionnement de tout système de dialogue. Il consiste en la classification d'un message dans des catégories d'intention préalablement déterminées. Ainsi, au message « quel est le prix de » est reliée l'intention de l'utilisateur de connaitre le prix d'un article. La bonne détection de cette intention dépend du résultat de la classification des *intents*. De cette façon, le chatbot devra reconnaitre l'*intent* « connaitre_prix » afin de donner la bonne réponse à l'utilisateur.

De nombreux modèles de *machine learning* ont été utilisés et testés pour résoudre ce problème. Nous poserons des fondements linguistiques à la notion d'*intent* avant de définir plus précisément la tâche de classification et les nombreuses méthodes employées dans la littérature pour la réaliser.

22. Les crochets indiquent que « Mazda 6 » correspond à une entité.

1.4.1 Intent

Il conviendra de revenir sur la définition de l'*intent* et d'y apporter un fondement linguistique. Comme explicité supra, est désigné par *intent* l'intention de l'utilisateur au sein d'une phrase. Celle-ci peut aller de « dire bonjour » à « demander le prix » ou à « demander le catalogue ». En suivant cette logique, la locution « Salut bot ! » renverra à l'*intent* « bonjour ». Le chatbot, une fois cette information extraite, pourra alors déclencher une action. Cela peut simplement être un message : « Hello ! Comment-puis-je t'aider ? ». Cette section a pour ambition de souligner le fondement linguistique des *intents* avec la théorie des actes de langages (Austin, 1975).

Selon Searle (1968), qui reprend et commente Austin, une distinction doit être établie entre les actes locutoires et illocutoires. Les premiers consistent en une signification (de la locution), c'est-à-dire les actes comportant un sens et une référence, tandis que les seconds comportent une « force ». L'exemple présenté est le suivant : « I am going to do it » peut avoir la force d'une promesse, d'une prédiction, d'une menace, d'un avertissement ou plus simplement d'une déclaration d'intention. Cette phrase n'a pourtant qu'un seul et unique sens littéral, il n'y aurait donc qu'un acte locutoire. Searle nous explique qu'il peut cependant contenir de nombreux actes illocutoires possibles. Afin d'y voir plus clair et pour distinguer les deux actes, Searle reprend un exemple donné par Austin :

- Locution : *He said to me "Shoot her !" meaning by "shoot" shoot and referring by « her » to her ;*
- Illocution : *He urged (or advised, ordered, etc.) me to shoot her ;*
- Locution : *He said to me "You can't do that" ;*
- Illocution : *He protested against my doing it.*

Searle met en relation l'acte illocutoire avec une déclaration d'intention. Nous nous intéresserons donc particulièrement à cette notion comportant une « force » et une « déclaration d'intention ».

Le sens de la phrase (acte locutoire) peut déterminer une force illocutoire pour différentes applications. Par exemple, « I hereby promise that I am going to do it » comportera toujours une force illocutoire reliée à la déclaration d'une promesse. Cette force peut en effet être incluse dans le sens de la phrase. De plus, il existe également des verbes illocutoires qui couvrent une classe très générale de forces illocutoires. Par exemple, « he told me to » couvre des forces reliées à « he ordered », « he commanded » ou « he requested ». La distinction entre le sens (l'acte locutoire) et la force (l'acte illocutoire) d'une même phrase n'est donc pas claire.

Les actes illocutoires peuvent être plus ou moins définis selon leurs forces illocutoires. Il serait possible de penser que les actes illocutoires respectent un continuum, mais la situation est plus complexe. En effet, sous la rubrique de

« forces illocutoires » sont regroupés plusieurs principes de distinction. Searle propose douze dimensions significatives (Searle, 1976) dans lesquelles les actes illocutoires peuvent varier (objectifs de l'acte, états psychologiques exprimés, relations au reste du discours).

Un acte illocutoire, l'unité basique de la communication linguistique humaine selon Searle (1976), est donc constitué de forces illocutoires synthétisant divers aspects tels que le but de l'acte, le placement conversationnel et le degré d'engagement pris. Ce type d'acte a été classifié par Austin en une classification de verbes. En effet, Austin a proposé la typologie suivante [23] :

- Les « verdictifs » prononcent un jugement : acquitter, calculer, décrire, analyser,... ;
- Les « exercitifs » formulent une décision en faveur ou à l'encontre d'une suite d'actions : ordonner, commander,... ;
- Les « commissifs » engagent le locuteur à une suite d'actions déterminées : promettre, faire le vœu de,... ;
- Les « expositifs » sont utilisés pour exposer des conceptions, conduire une argumentation : affirmer, nier, répondre,... ;
- Les « comportementaux » sont liés au comportement des autres : s'excuser, remercier, féliciter, défier,... ;

Cette typologie ne classifie donc pas les actes, mais les verbes. À la classification des verbes devrait donc correspondre la classification des actes. Cependant, Searle estime cette typologie critiquable et introduit la sienne (Searle, 1975). Cette dernière est une adaptation de celle d'Austin. Il ajoute qu'il est nécessaire de faire la distinction entre les verbes illocutoires et les actes illocutoires (les illocutions). Les illocutions sont une partie du langage, à ne pas confondre avec les langues, tandis que les verbes illocutoires font toujours partie d'une langue spécifique.

Nous nous intéressons surtout aux illocutions et à leur rapport au langage. Loin d'être une structure de surface tels que les verbes, l'illocution donne à voir une synthèse de différentes forces à l'œuvre au sein du langage qui constituent le sens « caché » ou « profond » d'un énoncé. Ainsi, la locution « avez-vous l'heure ? » comporte l'acte illocutoire (plus ou moins directif) pour que l'on donne l'heure et non une simple affirmation ou négation. L'acte illocutoire représente les actes effectués durant la locution (Vermersch, 2007). En effet, « parler c'est produire des effets ou des conséquences, mais aussi viser un but, rechercher l'obtention d'un résultat » (Vermersch, 2007). Les actes illocutoires, parfois visibles directement à travers des verbes illocutoires, comportent donc des forces, dont la variabilité est grande, qui composent le sens implicite et réel d'un énoncé. Ainsi, la locution « est-ce qu'il y a du sel » ne demande pas une

23. Cette source nous a été utile pour traduire les noms des différentes classes de la typologie présente dans Austin (1975). Source : http ://psydoc-fr.broca.inserm.fr/linguistique/actes.html

affirmation ou une négation pour réponse, mais bien une action, celle de recevoir la salière. C'est dans cette dimension de l'implicite que l'acte illocutoire prend sens et vie. C'est également dans ce sens qu'est utilisée la notion d'*intent*. Tout chatbot demande en effet à recevoir des illocutions, des demandes d'actes, qui répondent à un besoin d'agir. Parler n'est pas innocent, encore moins avec un agent conversationnel dont le but dans une conversation est originellement déterminé. La notion d'*intent* peut donc, à notre sens, être directement mise en lien avec celle d'illocution dans le sens précédemment explicité.

1.4.2 Classification

« Classifier consiste à définir des classes ; classer [(la classification)] est l'opération permettant de mettre un objet dans une classe définie au préalable » (Husson et al., 2016). Les différentes classes sont, en *machine learning*, apprises par le système d'après un jeu de données annotées. Une fois ces classes définies et leurs attributs calculés par le système, le modèle est en mesure d'attribuer une classe à des éléments nouveaux. La détection d'*intents* est le plus souvent vue comme un problème de classification. En effet, le message de l'utilisateur est porteur d'une intention qu'il faut être en mesure de replacer dans la bonne classe. Différentes étapes (Stöckl, 2017) permettant de représenter l'information textuelle sous la forme de nombres doivent être accomplies avant l'implémentation d'une technique de *machine learning* :

— Une phase de prétraitement est évidemment indispensable : normalisation de l'orthographe, tokenisation, suppression de la ponctuation, etc ;
— Construction de vecteurs : les algorithmes de *machine learning* ont en effet généralement besoin de nombres en entrée : les *words embeddings*[24] ;
— Une phase de visualisation des données est alors bienvenue pour s'en faire une meilleure idée ;

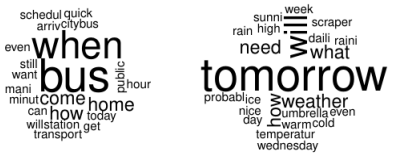


FIGURE 1.2 – Tagcloud (Stöckl, 2017)

24. TensorFlow ou GloVe par exemple.

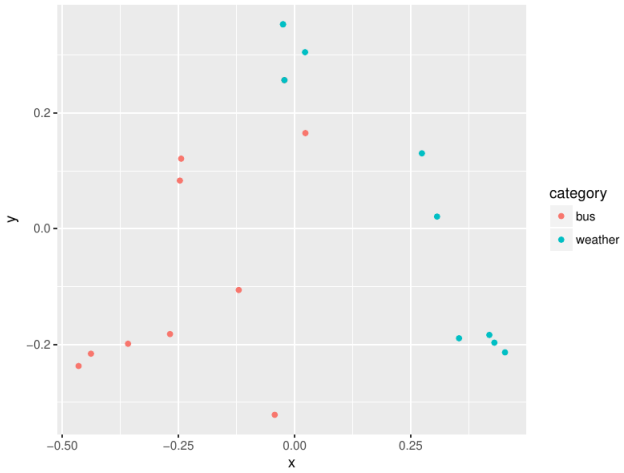


FIGURE 1.3 – Représentation des vecteurs sur un plan (Stöckl, 2017)

— Implémentation d'un algorithme de *machine learning* pour dériver des règles depuis les données d'entrainement dans le but de classer les nouveaux messages. Ces données textuelles transformées en nombres sont indispensables au bon fonctionnement des modèles probabilistes ou statistiques.

Nous définirons et expliquerons dans les sections suivantes les différentes techniques mises en œuvre dans la classification des *intents*. Des approches classiques telles que la classification Bayésienne ou les KNN ont d'abord été utilisées (Kamphaug et al., 2017) avant de laisser place aux modèles neuronaux et à leurs variantes (RNN, CNN). Enfin, les SVM ainsi que la régression logistique sont également des alternatives fiables. Il est également possible, marginalement, de trouver des systèmes déterministes, fonctionnant à l'aide de règles prédéterminées.

1.4.3 Classification naïve Bayésienne

Un classificateur naïf Bayésien met en place une classification probabiliste simple basée sur le théorème de Bayes avec indépendance (naïve) des hypothèses. Le théorème de Bayes est le suivant (Stöckl, 2017) :

$$P(C_b \mid x_1, x_2, ..., x_n) = \frac{P(C_b)\, P(x_1, x_2, ..., x_n \mid C_b)}{P(x_1, x_2, ..., x_n)}$$

$P(C_b)$	probabilité a priori de l'intent
$P(x_1, x_2, ..., x_n)$	probabilité a priori des mots
$P(C_b \mid x_1, x_2, ..., x_n)$	probabilité conditionnelle de l'intent selon x_i
$P(x_1, x_2, ..., x_n \mid C_b)$	probabilité conditionnelle de x_i selon l'intent C_b

Il est possible de transformer cette équation grâce à la notion d'indépendance des hypothèses (Stöckl, 2017) :

$$P(C_b \mid x_1, x_2, ..., x_n) = \frac{P(C_b) \prod_{i=1}^{n} P(x_i \mid C_b)}{P(x_1, x_2, ..., x_n)}$$

La catégorie sélectionnée est celle récoltant la probabilité la plus élevée. Ce classificateur « est naïf parce qu'il assume l'indépendance entre les caractéristiques, dans ce cas : les mots. Dit autrement : chaque mot est traité comme n'ayant aucun lien avec les autres mots de la phrase à classer » (gk_, 2017) [25]. Bien que l'indépendance est généralement une supposition problématique (Rish et al., 2001), ce type de classificateur a su rivaliser avec d'autres approches de *machine learning*. Il était en effet généralement utilisé lorsque de nombreux attributs devaient être considérés simultanément (Stöckl, 2017), dans ce cas-ci c'est le comptage de mots des séquences. Une implémentation possible (gk_, 2017) consiste à tokeniser et à « stemmer » la base de données NLU afin de pouvoir comparer les stemmes du message, alors vu comme un « bag of words », avec ceux de la base de données des classes (les *intents*). Il est ensuite possible,

```
Corpus words and counts: {'how': 3, 'ar': 1, 'mak': 2, 'see': 1,
'is': 2, 'can': 1, 'me': 1, 'good': 1, 'hav': 3, 'talk': 1,
'lunch': 1, 'soon': 1, 'yo': 1, 'you': 4, 'day': 4, 'to': 1,
'nic': 2, 'lat': 1, 'a': 5, 'what': 1, 'for': 1, 'today': 2,
'sandwich': 3, 'it': 1, 'going': 1}


Class words: {'goodbye': ['hav', 'a', 'nic', 'day', 'see', 'you',
'lat', 'hav', 'a', 'nic', 'day', 'talk', 'to', 'you', 'soon'],
'sandwich': ['mak', 'me', 'a', 'sandwich', 'can', 'you', 'mak',
'a', 'sandwich', 'hav', 'a', 'sandwich', 'today', 'what', 'for',
'lunch'], 'greeting': ['how', 'ar', 'you', 'how', 'is', 'yo',
'day', 'good', 'day', 'how', 'is', 'it', 'going', 'today']}
```

FIGURE 1.4 – Bases de données tokenisées et stemmées (gk_, 2017)

25. This classifier is "naive" because it assumes independence between "features", in this case : words. Said differently : each word is treated as having no connection with other words in the sentence being classified.

sur le modèle décrit ici, de déterminer la classe qui serait rattachée au message suivant : « good day for us to have lunch ? ». C'est donc une méthode populaire

```
match: day (0.25)
   match: to (1.0)
   match: hav (0.3333333333333333)
Class: goodbye  Score: 1.5833333333333333

   match: for (1.0)
   match: hav (0.3333333333333333)
   match: lunch (1.0)
Class: sandwich  Score: 2.333333333333333

   match: good (1.0)
   match: day (0.25)
Class: greeting  Score: 1.25
```

FIGURE 1.5 – Résultats (gk_, 2017)

pour la catégorisation de textes et, notamment, la classification d'*intents* (Stöckl, 2017). Pour un vecteur (représentation numérique d'un énoncé) donné, les probabilités qu'il appartienne à la liste d'*intents* sont calculées et la probabilité la plus haute sera retenue. C'est de cette façon que cette méthode est appliquée pour la classification d'*intents*.

1.4.4 K-nearest neighbors (KNN)

Les k plus proches voisins (*k nearest neighbors* ou KNN) est une méthode d'apprentissage supervisé non paramétrique permettant d'assigner à une observation inconnue une classe ou une catégorie calculée comme étant la plus proche depuis les catégories d'un jeu de données. Cette méthode est considérée comme étant l'une des plus simples en *machine learning*. Cover et al. (1967) donnent à voir un excellent article de cette technique pour la classification de *patterns*. Le texte donné par l'utilisateur au chatbot est vectorisé et placé dans l'espace des données d'entrainement. L'énoncé est alors considéré comme une cible. Cette méthode prend en compte les k points les plus proches de la cible et sélectionne la classe majoritaire pour déterminer sa classe la plus probable. Par exemple, si k = 5 et que l'on retrouve 4 fois l'*intent* « weather », alors la cible recevra comme *intent* « weather » (Stöckl, 2017). C'est de cette manière qu'est utilisé le modèle des KNN pour la classification d'*intents*. Cette prédiction est réalisée en une seule étape, il n'y a en effet pas besoin de créer un modèle (*lazy classifier*).

Cependant, la littérature actuelle regorge de méthodes innovantes qui surpassent ces approches classiques pour la classification d'*intents*. Les modèles actuellement utilisés sont, pour la plupart, apparentés aux modèles neuronaux et à leurs extensions.

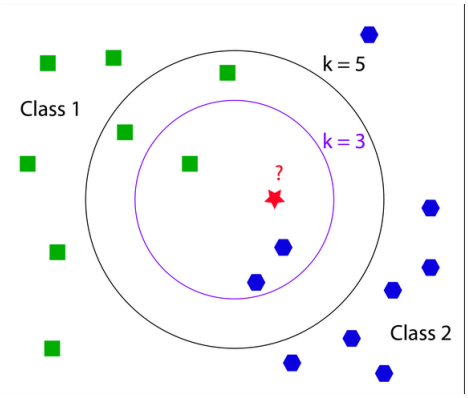


FIGURE 1.6 – KNN (Cover et al., 1967)

1.4.5 Modèles neuronaux

Nous décrirons dans cette section les différents types de modèles neuronaux qui ont été utilisés pour la tâche de classification des *intents*. Nous présenterons, dans l'ordre, le fonctionnement d'un réseau de neurones simple, le réseau de neurones artificiels récursifs (RNN) et le réseau neuronal convolutif (CNN).

1.4.5.1 Réseaux de neurones

Les réseaux de neurones artificiels sont une méthode de *machine learning* existant depuis des dizaines d'années. De récents progrès l'ont projetée au devant de la scène. Un réseau classique est composé de trois parties (Gershenson, 2003) :

— Une couche d'entrée qui prend les valeurs qu'on lui donne ;
— Des couches cachées reliées entre elles par des connexions ayant un poids (afin de simuler le fonctionnement des synapses d'un cerveau) ;
— Une couche de sortie qui donne le résultat final.

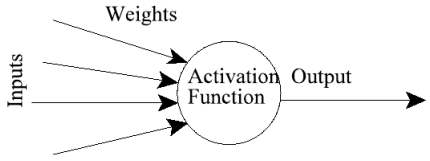


FIGURE 1.7 – Un neurone artificiel (Gershenson, 2003)

Un réseau de neurones modélise la relation entre des valeurs d'entrées et une de sortie. Ainsi, dans le cadre de la classification d'*intents*, les valeurs d'entrées

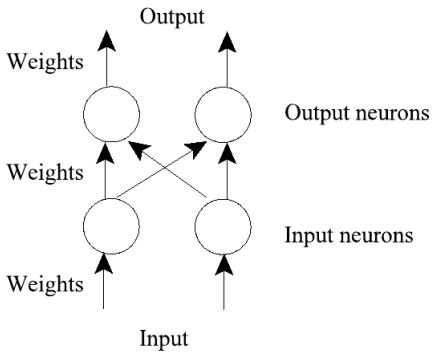


FIGURE 1.8 – Un réseau de 4 neurones artificiels (Gershenson, 2003)

sont les vecteurs des mots du texte et la sortie est composée des probabilités sur les classes déterminées grâce à une fonction *softmax*[26](Stöckl, 2017). L'*intent* est ensuite déterminé en fonction de ces probabilités. Pour en savoir plus sur son fonctionnement, nous ne pouvons que conseiller de lire la littérature (Jain et al., 1996). Ce type de technique est le plus souvent utilisé dans des méthodes plus complexes telles que les RNN.

1.4.5.2 Recursive neural network (RNN)

Les réseaux de neurones artificiels récursifs[27] sont très largement utilisés en reconnaissance de la parole ainsi qu'en traitement des langues de façon générale. Ces modèles séquentiels sont le plus souvent construits de manière à pouvoir reconnaitre les caractéristiques depuis des données et d'utiliser des *patterns* pour prédire le scénario suivant le plus probable. Leur efficacité apparait particulièrement dans des situations où le contexte est important. Ils se différencient des réseaux de neurones classiques par des boucles de « feedback » qui permettent de conserver l'information. Ils ont donc une forme de « mémoire » en plus. Ils sont composés de couches capables de recevoir un *input* et de générer un *output* à d'autres nœuds du réseau. À la différence des réseaux simples, un RNN est apte à faire passer l'information dans les deux sens (vers l'avant et vers l'*output*). Un problème majeur de ce type de réseau, partagé par les perceptrons multicouches (*multilayer perceptron MLP*), est ce que l'on nomme le « vanishing

26. « Elle transforme les nombres [...] en probabilités qui s'additionnent à 1. La fonction *softmax* produit un vecteur qui représente les distributions de probabilité d'une liste de résultats potentiels [les *intents*]. C'est aussi un élément central utilisé dans les tâches de classification [...] » (Uniqtech, 2018b).

27. Cette définition est construite à partir de Rouse (2018).

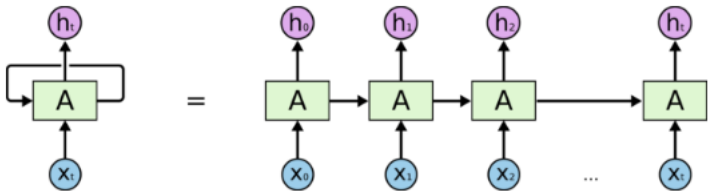


FIGURE 1.9 – RNN (Banerjee, 2018)

gradient problem » [28] qui génère des difficultés lors de l'entrainement.

> « Nous avons besoin que le signal circule correctement dans les deux sens : vers l'avant pour les prédictions, et dans le sens inverse pour propager les gradients vers l'arrière. Pour que cela se produise, [...] nous avons besoin que la variance des sorties de chaque couche soit égale à la variance de ses entrées, et que les gradients aient une variance égale avant et après le passage d'une couche dans le sens inverse. Dans les faits, il n'est pas possible de garantir les deux à moins que la couche ait un nombre égal de connexions d'entrée et de sortie, mais ils [Glorot and Bengio (2010)] ont proposé un bon compromis qui s'est avéré très efficace dans la pratique : les poids de connexion doivent être initialisés de manière aléatoire [(*Xavier initialization*)] » (Géron, 2017) [29].

En ce qui concerne la classification d'*intents*, plusieurs équipes de chercheurs ont exploité cette technique pour mener à bien la tâche. C'est le cas de Kamphaug et al. (2017) qui ont appliqué ce type de méthode dans un chatbot plus général et disposant de nombreux *intents*. Leur modèle est construit de sorte à capturer la structure complexe d'une phrase à de multiples niveaux d'abstraction. Leur méthode est la suivante :

— Une représentation vectorielle de chaque mot est créée grâce à un « embedding layer », les termes avec un sens similaire sont proches dans l'espace vectoriel ;

28. Au plus on ajoute des couches utilisant certaines fonctions d'activation, au plus les gradients de la fonction de perte se rapprochent de zéro, ce qui rend le réseau difficile à entrainer (Wang, 2019).

29. We need the signal to flow properly in both directions : in the forward direction when making predictions, and in the reverse direction when backpropagating gradients. For this to happen, the authors argue that we need the variance of the outputs of each layer to be equal to the variance of its inputs, and we also need the gradients to have equal variance before and after flowing through a layer in the reverse direction [...]. It is actually not possible to guarantee both unless the layer has an equal number of input and output connections, but they proposed a good compromise that has proven to work very well in practice : the connection weights must be initialized randomly [...].

— Le RNN, composé de cellules GRU [30] (une variante plus simple des LSTM [31]), empile les cellules existantes afin de créer une autre couche d'abstraction. Cette méthode permet d'apprendre des concepts plus complexes ;
— La partie finale du modèle consiste en deux couches de réseau de neurones totalement connectées. Un tel réseau entièrement connecté est adapté à une résolution flexible de problèmes de classification. En effet, cela est dû à la création d'entités abstraites dans des couches sous-jacentes et, le plus souvent, à l'utilisation de la fonction *softmax* pour la construction de la sortie du réseau.

Ils obtiennent des résultats convaincants qu'ils attribuent à la capacité du RNN d'extraire et de se souvenir de concepts abstraits apparaissant dans les séquences de mots, simplement en tentant de prédire quel mot apparaitra ensuite.

Mensio et al. (2018) ont utilisé cette méthode dans le cadre d'un système QA à plusieurs tours. Ce type de système de questions-réponses se rapproche sensiblement d'un chatbot car, contrairement à leurs homologues en simples tours, ils permettent de donner lieu à des discussions. Cette caractéristique complique d'ailleurs la tâche et, bien souvent, la solution de facilité est de proposer un mot-clé « stop » pour revenir à un état initial. De récentes recherches portent cependant sur le « tracking » du but de l'utilisateur. Le choix de Mensio et al. (2018) s'est porté sur les RNN afin de travailler sur des séquences de mots et de catégoriser la phrase d'après un set d'*intents* tout en prenant en compte le tour précédent. Ils estiment que les cellules GRU et LSTM (permettant de disposer d'une « mémoire ») sont des alternatives valides. Leur approche consiste en la concaténation du tour précédent à l'actuel pour mieux comprendre l'état de la conversation. Ils estiment en effet que le contexte est crucial pour les interactions de plusieurs tours de parole.

Liu and Lane (2016) ont étudié la détection d'*intents* conjointement à l'extraction d'entités. Ils estiment que la première tâche peut être considérée comme un problème de classification sémantique tandis que la seconde se rapporte davantage à une tâche d'étiquetage de séquences. Traditionnellement séparées dans le système, ces deux tâches ont été fusionnées dans un même modèle. Ils utilisent un RNN bidirectionnel à cellules LSTM dans l'encodeur. Le choix des cellules LSTM se justifie car elles sont plus efficaces dans les modélisations de relations longues distances. Cet encodeur lit la séquence de mots (dans les deux sens), génère des états cachés et les concatène. Le décodeur est, quant à lui, un RNN simple implémenté avec des cellules LSTM. Il génère un résultat représentant l'*intent*.

30. *Gated Recurrent Unit*.
31. *Long Short-Term Memory*. Ces cellules catégorisent les données dans des cellules « short term » et « long term » qui permettent au RNN de se souvenir de l'importance des différentes données au fur et à mesure des itérations. Ceci permet d'éviter d'oublier des données peu importantes mais surtout de garder les données essentielles (Hochreiter and Schmidhuber, 1997).

Ils ont ensuite ajouté un mécanisme d'attention pour ne pas perdre d'information car, dans un RNN, les états cachés portent une information qui, au fil des itérations (dans les deux sens) peut se dégrader voire même se perdre. Ce mécanisme d'attention est utile, à la fois dans la détection d'entités et dans la classification d'*intents* car, lorsqu'il est activé, les poids des états cachés se dégradent moins et permettent d'obtenir de meilleurs résultats. Si ce mécanisme n'est pas enclenché, une mise en commun des états cachés et une régression logistique prennent le relais pour la classification. Ils ont testé leur système sur les données ATIS (Hemphill et al., 1990) et ont obtenu de bons résultats. Ils estiment par ailleurs qu'il est bénéfique d'utiliser un seul composant pour ces deux tâches afin de simplifier le système et sa mise en place.

1.4.5.3 Convolutional neural network (CNN)

Un CNN [32] demande moins de prétraitement que d'autres méthodes. Cette technique est très utilisée pour la reconnaissance et la classification d'images (Prabhu, 2018). Comparativement aux RNN, un CNN est plus rapide car, lorsque le RNN calcule de façon séquentielle les mots d'une phrase, le CNN le fait simultanément. Il est donc possible d'obtenir des résultats similaires en abaissant les contraintes calculatoires, ce qui en fait un modèle intéressant pour le TAL [33]. Il semble l'être particulièrement pour les tâches de classification (Kim, 2014). Des chercheurs ont alors appliqué ce type de modèle à la classification

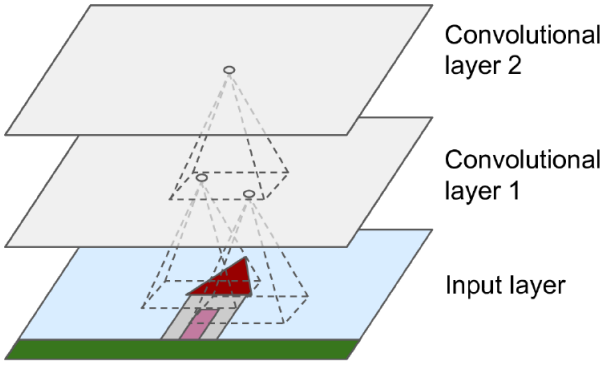


FIGURE 1.10 – CNN (Géron, 2017)

d'*intents* pour des chatbots. C'est le cas de Xu and Sarikaya (2013) qui ont mis au point un modèle CNN pour détecter les *intents* d'un message et pour localiser les entités. Leur classification d'*intents* diffère des réseaux de neurones simples car ces derniers convertissent la séquence de mots en vecteurs (avec un

32. La définition de ce système s'inspire de Bressler (2018).
33. Traitement Automatique du Langage

compteur *n-gram* par exemple) sur lesquels les couches sont construites. Dans leur modèle, les couches convolutives sont employées pour extraire directement des particularités depuis la séquence de mots. Ces traits (des vecteurs) sont ensuite additionnés pour obtenir une valeur de sortie : l'*intent*. Ces couches convolutives, et donc les traits qui sont dégagés des données, sont partagés par les deux tâches qui sont donc prises en charge simultanément.

1.4.6 Support-vector machine (SVM)

Les machines à vecteur support (*Support Vector Machines*) sont une méthode de *machine learning* très utilisée dans les classifications (melepe, 2017 ; Patel, 2017). Leur but est d'apprendre à placer une délimitation entre deux classes. La frontière choisie doit être aussi lointaine que possible des premiers éléments de chaque côté (appelés vecteurs supports), ceci afin de ne pas biaiser la classification et d'optimiser la capacité de généralisation. De façon générale, un SVM aura pour tâche de trouver un hyperplan qui sépare les catégories. Dès lors, un tel classificateur déterminera la classe du nouvel objet selon sa position dans l'hyperplan par rapport à la frontière. En effet, les hyperplans sont placés de sorte que les catégories sont séparées par un « fossé ». Les nouveaux objets sont placés sur le plan et la classification de l'*intent* s'effectuera en fonction du côté du fossé vers lequel « tombe » ou « penche » l'objet (Stöckl, 2017). Ce système peut générer ce type de frontières :

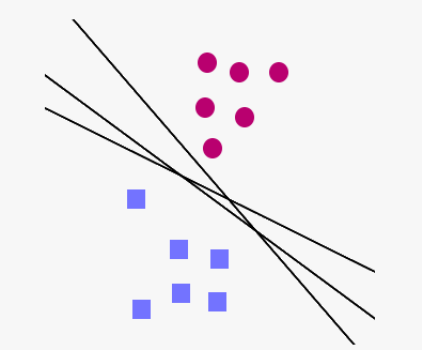


FIGURE 1.11 – SVM (melepe, 2017)

Dès lors, pour la tâche de classification d'*intents* (Stöckl, 2017), il faudra vectoriser l'entrée de l'utilisateur afin de pouvoir la replacer sur l'hyperplan. Ensuite, l'application de la procédure des SVM permettra de déterminer la catégorie vers laquelle penche le plus la cible. La catégorie ainsi trouvée correspond à l'*intent* le plus probable.

Bhargava et al. (2013) ont obtenu des résultats intéressants dans la prédiction d'*intents* en la traitant comme un problème de classification en utilisant des SVM ou comme une tâche de *tagging* avec des SVM-HMMs. Cette seconde approche allie donc les SVM avec les *Hidden Markov Model* (Rabiner, 1989), technique bien connue en TAL.

1.4.7 Régression logistique

La régression logistique (Gandhi, 2018) est l'une des techniques les plus connues en *machine learning*. Elle ressemble beaucoup à la régression linéaire, utilisée pour des tâches de prédiction, mais se distingue dans ses applications : la classification. Cette technique utilise une équation linéaire, dont la valeur prédite peut être comprise entre -infini et +infini, ainsi qu'une fonction sigmoïde, permettant de transformer le résultat de l'équation linéaire dans un intervalle compris entre 0 et 1 (ce qui la distingue de la régression linéaire). Ces deux valeurs représentent deux classes. Typiquement, si la valeur pour un nouvel objet est inférieure à 0.5, elle appartiendra à la classe 0.

L'équipe de création de Snips[34] (Coucke et al., 2018), un chatbot respectueux de la vie privée, a utilisé plusieurs modules :

— Un modèle de langue pour prédire les phrases les plus probables reconnues depuis la parole (nécessaire pour le *Spoken Language Understanding*, ou SLU) ;

— Un module NLU qui s'occupe notamment de la classification d'*intents*. Celle-ci est gérée par un *parser* déterministe (implémentation basée sur des *regex*[35] pour créer des *patterns* à reconnaitre) et, lorsqu'il ne fonctionne pas, il est remplacé par un *parser* probabiliste. Ce dernier est implémenté sous la forme d'une régression logistique permettant de reconnaitre plus aisément les variations grâce à sa capacité de généralisation. Ce modèle, binaire, peut cependant être utilisé pour classer des éléments non binaires :

> « Nous avons supposé ici que chaque caractéristique est binaire, c'est-à-dire que chaque entrée possède ou non une caractéristique. Les caractéristiques à valeur d'étiquette (par exemple, une caractéristique de couleur qui peut être rouge, verte, bleue, blanche ou orange) peuvent être converties en caractéristiques binaires en les

34. https ://github.com/snipsco/snips-nlu
35. expressions régulières.

remplaçant par des caractéristiques binaires telles que "color-is-red" » [36].

Ce modèle est entrainé à partir d'exemples de phrase pour chaque *intent*. Le graphique suivant illustre cette architecture.

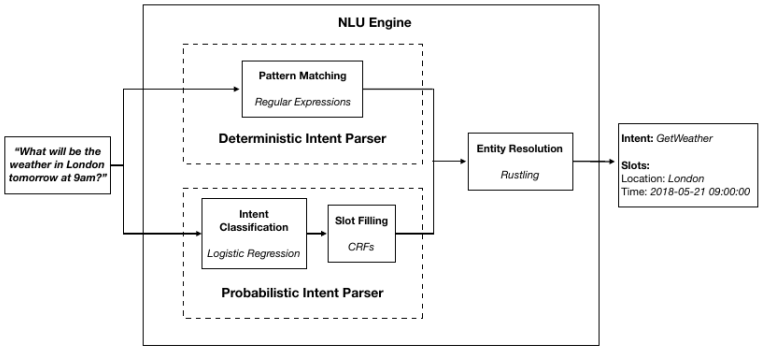


FIGURE 1.12 – Architecture de Snips (Coucke et al., 2018)

1.4.8 Par règles

Les approches par règles sont de loin les techniques les plus compliquées à mettre en œuvre. En effet, pour obtenir de bons résultats, il est nécessaire d'en créer suffisamment pour résister aux variations linguistiques. Cela peut prendre beaucoup de temps et cette méthode ne permet pas de généraliser facilement ou d'exporter les règles à d'autres applications. Elles sont à ce jour très peu présentes dans la classification d'*intents*. Cependant, il est possible d'en trouver dans certaines approches :

— Comme expliqué dans la section précédente, Snips dispose d'un *parser* déterministe, donc constitué d'une multitude de règles de ce format :

```
set the (?P<room>kitchen|hall|bedroom) lights to
```

L'élément « room » désigne l'entité et les différents noms de pièces sont les valeurs qui peuvent lui être affectées. Ce *parser* reconnaitra donc l'*intent* « Allumer_pièce » si la séquence reçue correspond à cette règle.

36. We have assumed here that each feature is binary, i.e. that each input either has a feature or does not. Label-valued features (e.g., a color feature which could be red, green, blue, white, or orange) can be converted to binary features by replacing them with binary features such as "color-is-red". Voir : https ://www.nltk.org/book/ch06.html

Évidemment, les langues étant très créatives, un *parser* probabiliste est prévu pour prendre le relais quand aucune règle ne correspond.

— Athreya et al. (2018) ont développé un projet de chatbot *open source*[37] utilisant un algorithme, *Rivescript*[38], basé sur des règles pour identifier les *intents*. Les règles ont été créées au préalable grâce à des conversations réelles d'utilisateurs. Ces *patterns* sont très similaires aux *regex* dans leur formalisme :

```
+[how][*](contribute|contributing|involved)...
```

1.4.9 Synthèse

Comme nous avons pu le constater, les techniques utilisées dans cette tâche de classification sont majoritairement supervisées. Cette supervision est principalement représentée par l'utilisation d'une base de données suffisamment pertinente et vaste pour entrainer ces modèles de *machine learning* (les approches par règles nécessitent aussi d'avoir accès à des données représentatives pour extraire des *patterns* pertinents). L'expérience d'utilisateur dépend largement de la performance du modèle NLU (notamment la classification d'*intents*) du chatbot. Celle-ci dépend principalement des données utilisées pour sa construction (Bapat et al., 2018). Le problème est cependant de taille, ce type de corpus doit en effet couvrir la diversité des requêtes possibles. Obtenir de telles données nécessite souvent de passer par une phase de création manuelle, étape longue et couteuse. Cependant, différentes techniques ont été mises au point pour accélérer ou automatiser cette collecte de données.

1.5 Surgénération de données dans les chatbots

Dans le domaine des chatbots, un enjeu crucial est d'avoir accès à des jeux de données suffisamment vastes et pertinents pour obtenir de bons résultats pour le modèle NLU. Cette recherche se cristallise de différentes manières que nous pourrions diviser en deux approches. La première consiste à utiliser un matériel linguistique préexistant afin de l'incorporer dans la base de données. Le *crowdsourcing* ainsi que l'acquisition de données depuis des forums en ligne en sont des exemples. Très peu de transformations, voire aucune, sont effectuées sur les données. La seconde approche vise à générer, depuis un jeu de données restreint, des paraphrases afin d'ajouter de la diversité linguistique et d'améliorer le modèle.

37. https ://github.com/dbpedia/
38. https ://www.rivescript.com/

1.5.1 Crowdsourcing

Faire appel aux locuteurs de la langue est une technique souvent utilisée dans de nombreuses applications en TAL. En effet, plus les données sont nombreuses et provenant de locuteurs différents (en âge, genre, origine et autres), plus la capacité de généralisation d'un modèle entrainé sur ces données sera grande.

Bapat et al. (2018) ont adopté cette technique pour la construction d'un chatbot doté de 9 *intents*. Ils insistent sur l'importance du module NLU, nécessaire au bon fonctionnement du chatbot. Ils ont alors demandé à des participants d'âges, de milieux et d'origines différents d'enrichir les données d'un petit corpus. Ils ont également mis en place des stratégies pour être certains d'avoir des phrases pertinentes (vérifier la langue, s'assurer qu'il n'y a pas eu de copier-coller de la phrase d'exemple avec un faible ajout). Leur méthode peut donc être décrite comme une méthode *end-to-end*[39] permettant de générer des données d'entrainement grâce aux locuteurs en leur demandant d'enrichir le corpus (via des paraphrases ou des reformulations locales) pour ensuite entrainer un classificateur d'*intents*. Ils ont obtenu des résultats très encourageants, avec pour exception une tâche pour laquelle les consignes données aux locuteurs étaient trop imprécises, et ils donnent des pistes d'amélioration :

- Donner des consignes claires et précises aux utilisateurs ;
- Utiliser un système GWAP (*game with a purpose*[40]) : c'est-à-dire un jeu qui permet aux utilisateurs d'ajouter uniquement des phrases équivalentes qui ne sont pas déjà présentes dans la base de données ;
- Utiliser des techniques d'enrichissement de phrases : l'auto-génération de valeurs de paramètres (tels que des lieux) et l'ajout de « extra noisy attachments » (ajouter des éléments superflus) aux requêtes d'utilisateurs pour mimer certains comportements linguistiques ;
- Mettre en place un chatbot et récupérer les échanges avec des utilisateurs réels. Cela nécessite d'avoir a priori un modèle NLU suffisamment bien entrainé.

Nikitina et al. (2018) ont également investi le *crowdsourcing* pour l'élaboration de données d'entrainement pour un chatbot. Leur projet était de créer un chatbot de réminiscence pour personnes âgées capable de discuter de leur passé. Les personnes âgées sont en effet souvent isolées et le projet était de créer une compagnie conversationnelle. Cependant, maintenir ce type de conversation n'est pas chose aisée. Ils ont décidé de se concentrer sur les types et les sujets d'apprentissage plutôt que de mener une interaction au sein d'un sujet ou de formuler des phrases. L'objectif n'est donc pas de générer un maximum de para-

39. L'apprentissage de bout en bout (*end-to-end*) consiste à former un système d'apprentissage éventuellement complexe en appliquant l'apprentissage par gradients à l'ensemble du système (Glasmachers, 2017).

40. http ://gameswithpurpose.org/about/

phrases, car les sujets abordés peuvent correspondre à des domaines très larges et diversifiés. Il s'agit avant tout de déterminer depuis les données du *crowdsourcing* une logique de conversation, des règles, à même de gérer des situations diverses. Le chatbot doit en effet être capable de poursuivre la discussion, poser des questions et inciter les personnes à parler. Le *crowdsourcing* leur a donc permis de créer des « dialog dataset » utilisés pour entrainer l'algorithme et pour inférer des modèles de conversation. Les dialogues ont été également enrichis sémantiquement.

1.5.2 Acquisition de données depuis des forums en ligne

Huang et al. (2007) constatent les difficultés de construction et d'adaptation à de nouveaux domaines des bases de données de chatbot. Ils ont alors concentré leurs recherches sur l'extraction de données depuis des forums en ligne. Ces derniers sont en effet une mine d'informations linguistiques. Avant leur travail, il n'y avait en quelque sorte que des données annotées par des humains (Abu Shawar and Atwell, 2003 ; Tarau and Figa, 2004), couteuses en temps et difficilement adaptables. Ils extraient les contenus des balises HTML :

```
<thread-title, reply>
```

Ils les classent ensuite par pertinence grâce à un SVM basé sur la qualité de leur contenu. Les résultats de leur approche sont intéressants et démontrent une bonne efficacité, des dizaines de milliers d'exemples sont stockés et permettent d'obtenir une meilleure précision. Ils estiment encore pouvoir améliorer la pertinence et l'efficacité de leur technique. Cette méthode a cependant montré qu'elle était capable de générer de nombreux exemples d'entrainement dans le domaine des chatbots [41].

Cette approche a également convaincu Wu et al. (2008) qui ont utilisé un modèle de classification basé sur le « rough set » ou « ensemble approximatif » (Pawlak, 1982) et la technique de l'« ensemble learning » ou « apprentissage par ensembles » (Ditterrich, 1997) pour prendre une décision. Pour chaque forum, des ensembles de classificateurs approximatifs sont d'abord construits et entrainés. Ils classifient ensuite les réponses du forum et sélectionnent les messages qui y sont liés dans la base de données [42]. Ce schéma résume son fonctionnement :

41. Il convient de spécifier que la définition de chatbot utilisée dans ce papier rejoint davantage celle concernant les « open domain ». Ces derniers ne visent pas tant à reconnaitre efficacement des intents prédéfinis mais plutôt à être en mesure de fournir une réponse appropriée à de très nombreuses questions d'utilisateurs. C'est pour cette raison qu'ils parlent plutôt de « base de connaissance » ou de « knowledge ». Cependant, il est intéressant de considérer les forums en ligne, sources de nombreuses variations langagières, comme matériel linguistique pour la création de bases de données de chatbot.

42. Il s'agit encore une fois d'une optique davantage orientée vers le « open domain ». Il n'y a donc pas, à proprement parler, d'annotation d'intents sur cette base de données créée.

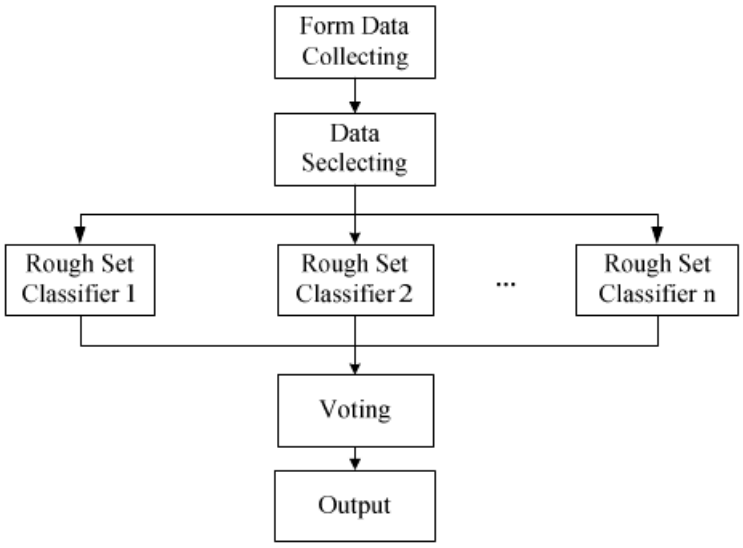


FIGURE 1.13 – Système d'extraction automatique de données (Wu et al., 2008)

Ils obtiennent de bons résultats mais estiment que beaucoup d'éléments peuvent affecter la qualité des réponses stockées. Ils considèrent alors seulement les caractéristiques structurelles ou propres au contenu, plus à même de permettre une reconnaissance efficace des séquences pertinentes. Ils caractérisent également les difficultés inhérentes à l'analyse de forums en ligne : des formats et des styles de forum différents ou des réponses d'utilisateurs qui ne citent pas le message auquel ils répondent. De plus, il convient de spécifier que :

> « les [données] extrait[e]s ne peuvent pas être ajouté[e]s directement à la base de connaissances. Un enjeu important est de concevoir une structure de stockage raisonnable pour améliorer l'efficacité de la récupération du module de gestion de dialogue »(Wu et al., 2008) [43].

1.5.3 Génération automatique de paraphrases

Il convient, avant toute chose, de définir ce qu'est exactement une paraphrase dans notre réflexion : « Les paraphrases sont des phrases ou des expressions qui véhiculent le même sens en utilisant des formulations différentes » (Bhagat and Hovy, 2013) [44]. Cette précision effectuée, nous introduisons différents tra-

43. Finally, extracted RRs are not suitable to be added directly to the knowledge base. It is a focus issue in the future that how to design a reasonable storage structure to improve the retrieval efficiency of the dialog management module.

44. Paraphrases are sentences or phrases that convey the same meaning using different wording.

vaux illustrant des techniques de génération de paraphrases dans le domaine des chatbots.

Face au manque de données, particulièrement à l'insuffisance de corpus de dialogues, Babkin et al. (2017) ont décidé d'appliquer des techniques non supervisées ou très peu supervisées dans le but d'affiner le modèle et sa couverture. Ils ont essayé, en plus, d'augmenter la base de données avec une génération automatique de paraphrases mais ont constaté des résultats instables. Ils estiment n'avoir pas pu approfondir cette voie par manque de temps. Leurs résultats sont modérés. Ils insistent sur le fait que cela fonctionne bien pour certains domaines mais qu'il n'est pas évident de transposer cette méthode à d'autres.

Gupta et al. (2019) travaillent sur la génération de paraphrases dans le cadre de la création d'une base de données d'entrainement de chatbots. Ils pensent en effet que ce domaine rencontre des problèmes dus à un manque de données, à la presque obligation de passer par une étape manuelle de création de corpus et à une perte de temps conséquente dans la mise en place de cette création de données. Ils estiment que l'ère du chatbot est arrivée et qu'il faut être en mesure de donner des outils efficaces dans le cadre de leur élaboration. Ils affirment que

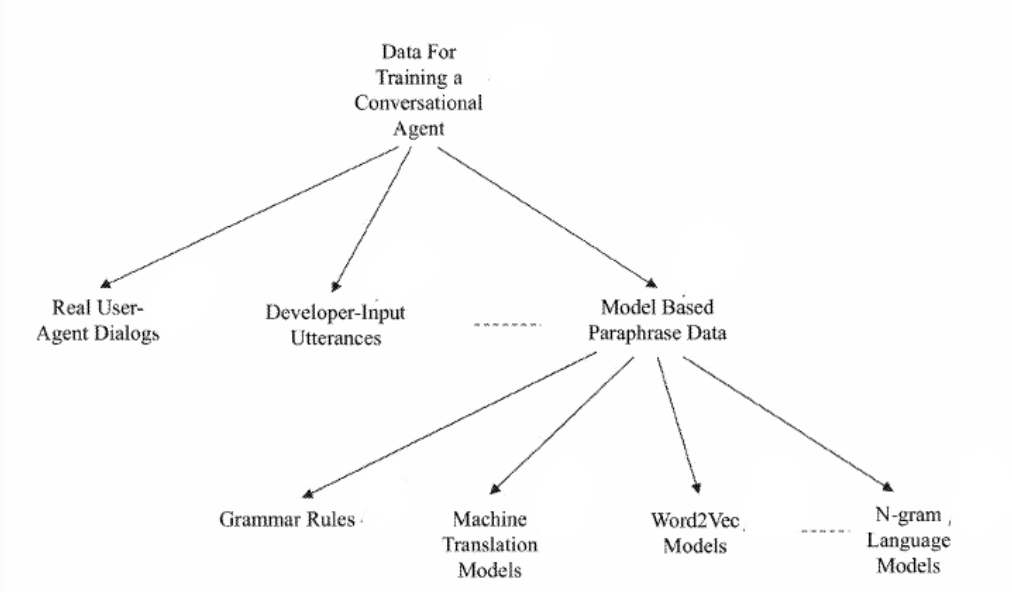


FIGURE 1.14 – Possibilités pour obtenir des données (Gupta et al., 2019)

trois cas de figure (voir illustration ci-dessus) sont possibles pour collecter ces précieuses données :

— L'ajout d'interactions réelles (*crowdsourcing*, forums) ;
— L'ajout manuel d'exemples par le développeur ;

— La génération automatique de paraphrases sémantiquement équivalentes (par règles, basé sur la traduction automatique).

C'est donc cette troisième méthode qu'ils ont décidé de mettre en œuvre en implémentant un modèle de génération de paraphrases basé sur la traduction automatique. Le procédé et le système mis en place visent à créer efficacement un agent conversationnel en générant rapidement des données d'apprentissage pour la création et la formation de modèles NLU. Leur application ressemble en tous points à notre approche de génération de paraphrases (basée sur des dictionnaires de synonymes) imaginée lors d'un stage chez *chatbot Plus*[45] durant l'été 2018. En effet, le procédé qu'ils ont imaginé est le suivant : pour chaque exemple d'énoncé que le développeur entre, le système génère de 10 à 100 paraphrases sémantiquement équivalentes que le développeur pourra librement ajouter au modèle selon leur pertinence. Le tout est implémenté dans une interface graphique qui permet au développeur de simplement cliquer sur les paraphrases pertinentes.

Cette approche, sans pour autant être totalement autonome, permet d'accélérer le processus de génération des données d'apprentissage et de garantir une bonne diversité dans le jeu de données. La qualité du modèle du chatbot s'en retrouve alors améliorée. De plus, il est bien plus simple pour le développeur de choisir des variations linguistiques que de les créer.

Leur modèle de génération de paraphrases combine à la fois les *n-grams*, des règles grammaticales et la traduction automatique pour créer un modèle statistique de création de paraphrases.

Nous avions déjà, avant de découvrir cet article, la volonté d'implémenter un générateur de paraphrases au sein de Rasa pour être en mesure de vérifier son efficacité dans la tâche de classification d'*intents*. Il a donc été à la fois très rassurant et inspirant de constater que cette idée était en voie d'exploration par une équipe de chercheurs. Nous ne pensons pas que notre travail perde de son originalité pour autant, de nombreuses variables se distingueront de l'approche menée par cette équipe :

— Le modèle de génération de paraphrases sera volontairement différent de celui implémenté dans cet article. Nous envisagerons en effet une approche automatique ;
— L'utilisation de Rasa permettra à la fois de tester cette technique dans un logiciel *open source* très utilisé, et de le proposer à l'équipe de développement s'il s'avère que les résultats sont significatifs ;
— Les données minimales d'entrainement du chatbot seront différentes ;
— Nous implémenterons notre système pour le français.

Pour mener à bien cette recherche, nous devons faire le choix d'une technique pour mettre en place la génération de paraphrases.

45. https ://www.chatbot-plus.com/

1.6 Génération de paraphrases

La génération de paraphrases fait partie des techniques permettant d'améliorer les bases de données pour certaines tâches en TAL, dont la classification d'*intents* dans les chatbots. Nous pouvons citer plusieurs autres domaines utilisant ce type de technique (Zhao et al., 2009) : le *Question-Answering* afin d'augmenter le nombre de questions ; la génération de langue naturelle pour le « polissage de textes » ; la simplification de textes ; la traduction automatique ainsi que le résumé automatique de textes pour améliorer le calcul de similarité. Cette section contient les grandes tendances et méthodes de la génération de paraphrases. Une technique retiendra notre attention pour l'implémentation finale.

1.6.1 Approches classiques

Certaines approches sont étudiées depuis longtemps [46] :

— Approche par règles : ce type de technique est daté et a été utilisé de nombreuses fois (Madnani and Dorr, 2010). Ces méthodes comportent des règles et des *patterns* qui sont soit créés à la main (McKeown, 1980 ; Zong et al., 2001) ou extraits automatiquement (Lin and Pantel, 2001 ; Barzilay and Lee, 2003 ; Zhao et al., 2008). Des règles de transformation peuvent également être utilisées avec l'algorithme Monte-Carlo (Chevelu et al., 2009) ;

— Approche par interlingua : un cadre sémantique est construit grâce à plusieurs étapes d'analyse morphologique, de *parsing* syntaxique et d'étiquetage sémantique depuis l'énoncé. Ce cadre se définit comme une interlingua, une réécriture est ensuite appliquée depuis ce résumé sémantique. Des problèmes apparaissent pour les phrases nominales, les prédicats et les couches supérieures du cadre sémantique (Glass et al., 1994). Le phénomène de mouvement, ou déplacement de groupes de mots au sein d'une phrase, pose beaucoup de problèmes. Un système assez complexe basé sur des règles est alors utilisé pour générer des paraphrases depuis le cadre sémantique précédemment créé. Concevoir ce type de générateur de langue naturelle (NLG) est loin d'être évident. Cette technique a été bien étudiée (Kozlowski et al., 2003 ; Power and Scott, 2005) ;

— Approche par thesaurus : ces approches basées sur l'utilisation de thesaurus sont anciennes et ont été largement étudiées (Bolshakov and Gelbukh, 2004 ; Kauchak and Barzilay, 2006 ; Hassan et al., 2007). Ces méthodes extraient d'abord tous les synonymes d'un terme avant de sélectionner le plus approprié selon le contexte de la phrase.

46. Ce résumé s'inspire librement de l'article de Zhao et al. (2009).

1.6.2 Statistical machine translation (SMT)

Quirk et al. (2004) ont appliqué la traduction automatique statistique (*statistical machine translation*, SMT) pour générer des paraphrases d'un exemple donné. Un grand corpus de phrases a été automatiquement extrait depuis des articles, traitant du même sujet, sur le web. Ces contenus ont été téléchargés depuis le HTML en isolant le contenu textuel. Un HMM supervisé permet de distinguer le contenu à télécharger de publicités ou d'éléments non pertinents. Ce type de données est très fréquent sur Internet, un corpus de 180 000 articles a donc pu être constitué. L'extraction de candidats paraphrases est exécutée grâce à la distance d'édition (ou *edit distance* de Levenshtein, 1966) [47], 140 000 paires sont trouvées. Ce système utilise un décodeur phrastique permettant de générer des paraphrases dans plusieurs domaines. Une phase de prétraitement est nécessaire : mots en minuscule, tokenisation, extraction d'entités nommées. La base de données permet ensuite, pour une phrase donnée, de proposer un ensemble de paraphrases. Ensuite, une procédure à base d'un trigram et du décodage de Viterbi détermine la paraphrase la plus probable. Le modèle de langue est un trigram entrainé sur plus de 24 millions de mots du corpus précédemment créé. Le post-traitement s'assure que la paraphrase trouvée n'est pas triviale et rétablit les valeurs des entités nommées et de la conjugaison. Le système est évalué grâce au taux d'erreur d'alignement et à deux juges humains. Ce modèle est en mesure de générer des synonymes, des remplacements locaux (« bush administration » pour « white house »), un réassortiment intra-phrastique. Il est cependant incapable de générer des alternatives structurelles (voix active et passive, inversion). Leurs résultats sont satisfaisants pour cette époque (2004), car ils sont préférés par les humains. Les chercheurs ont ensuite avancé qu'ils avaient besoin de plus grandes et de meilleures bases de données. Ils estiment que l'extraction des premières lignes des articles serait bien plus qualitative, car les journalistes ont tendance à résumer tout l'article dans les premières phrases.

1.6.3 Méthode non supervisée

Barzilay and Lee (2003) font reposer leur méthode sur un corpus comparable non annoté. Un set de *patterns* de paraphrases est ensuite représenté par un treillis de paires de mots qui permet de déterminer automatiquement de nouvelles paraphrases. Cette approche, selon eux, est très intéressante pour des systèmes de génération *text-to-text* (résumé, réécriture). Leur méthode est constituée de plusieurs étapes :

— Regroupement de phrases selon leur similarité (*sentence clustering*) ;

47. Cette mesure calcule le nombre de suppressions ou d'insertions de mots pour transformer une phrase en une autre. Elle permet donc de détecter des paraphrases assez proches. Ainsi, les auteurs ont même prévu d'exclure les paraphrases qui avaient pour seule différence la ponctuation.

— Alignement de plusieurs séquences des phrases dans un *cluster* donné ;
— Déterminer la meilleure manière d'insérer ou de supprimer des éléments ;
— Représenter grâce à des *n-grams* les similarités syntaxiques entre les phrases du *cluster* ;
— Déterminer quels mots sont des arguments devant être remplacés par des « slots » (ce sont généralement les zones de grande variabilité qui correspondent à des arguments ou entités) ;
— Phase de « matching lattice » où les différents treillis sont comparés pour confronter les valeurs des *slots* afin d'identifier les paires de paraphrases entre treillis ;
— Selon une phrase donnée, d'abord identifier quelle phrase d'un *cluster* lui est le plus proche (en trouvant le meilleur alignement grâce aux treillis). Si une paraphrase est trouvée, remplacer la phrase d'entrée par celle du corpus et lui substituer les valeurs des arguments avec ceux de la phrase originelle.

1.6.4 Sequence to Sequence (seq2seq)

Li et al. (2017) mettent en avant que la génération de paraphrases peut être formulée comme un « seqsSeq problem » [48]. L'un des plus gros *challenges* dans la génération de paraphrases est la définition d'une mesure évaluative. Il faut en effet calculer la similarité sémantique entre une phrase et ses paraphrases. L'« entropie croisée » (Shibuya, 2018) [49] traditionnelle en Seq2Seq n'est qu'une approximation qui affiche ses limites. Pour résoudre cela, il faudrait utiliser l'apprentissage par renforcement (*reinforcement learning*) pour guider l'entrainement du modèle Seq2Seq et utiliser des mesures telles que BLEU et ROUGE. Ces mesures ne sont pas pour autant parfaites pour représenter la similarité sémantique.

En pratique, ils ont adopté une approche *data-driven* pour entrainer un modèle capable d'effectuer une évaluation dans l'apprentissage pour la génération de paraphrases. Il est divisé en deux modules :

— Un générateur qui consiste en un modèle Seq2Seq avec un mécanisme d'attention ;
— Un évaluateur constitué d'un modèle de « deep matching » ainsi que d'un modèle d'attention décomposable.

Le générateur crée une paraphrase depuis un énoncé de départ et le donne à l'évaluateur qui en évalue le degré de similarité sémantique.

Les données utilisées sont celles du corpus Quora [50] et du Twitter URL pa-

48. ou « EncoderDecoder model » (Ma'amari, 2018)
49. Cross entropy.
50. https ://data.quora.com/First-Quora-Dataset-Release-Question-Pairs

raphrase Corpus [51].

Input : X = $[x_1, ..., x_s]$ avec une taille S
Output : Y = $[y_1, ..., y_t]$ avec une taille T (Y a le même sens que X)

Le but du modèle Seq2Seq est, selon une phrase X, d'entrainer un modèle G qui est capable de générer $Y = G(X)$ comme paraphrase. L'encodeur RNN transforme la phrase X en états cachés $H[h_0, ..., h_s]$ avec s le nombre d'états. Le décodeur RNN génère quant à lui une phrase sur base de ces états cachés. Dans les faits, le générateur et l'évaluateur sont entrainés alternativement. Il s'agit donc un RBM-IRL (*reinforcement by matching with inverse reinforcement learning*). Ce type de modèle peut utiliser efficacement les phrases générées pour entrainer l'évaluateur. Cette méthode offre de très bons résultats.

Gupta et al. (2018) ont opté pour la combinaison d'un modèle génératif profond (sous la forme d'un VAE [52]) avec des modèles Seq2Seq (implémenté avec des cellules LSTM) pour générer des paraphrases depuis une phrase en entrée. Les données utilisées sont celles des corpus MSCOCO [53] et Quora qui sont des bases de données de phrases accompagnées de leurs paraphrases. Ils ont obtenu de bons résultats. Les phrases générées ne sont pas seulement sémantiquement similaires aux phrases originelles, elles ont aussi réussi à capturer de nouveaux concepts liés à la phrase originelle.

1.6.5 Statistical paraphrase generation (SPG)

Zhao et al. (2009) sont les premiers à utiliser un générateur statistique de paraphrases. Ce dernier, selon eux, peut être aisément transformé afin d'être utilisable pour une application différente. Il exploite différentes ressources telles que des corpus de paraphrases, des *patterns* et des collocations pour pallier le manque de données (en 2009). Il peut être utilisé dans trois applications : la compression de phrases, la simplification de phrases et le calcul de la similarité entre phrases. Le système SPG proposé dans ce travail est composé de 3 modules :

- Un module de prétraitement de la phrase d'entrée (*POS tagging*, *parsing* des dépendances syntaxiques) ;
- Un module de *paraphrase planning* qui consiste en la sélection d'unités à paraphraser dans la phrase d'entrée et dans les paraphrases candidates ;
- Un module de génération de paraphrases qui désigne les unités optimales à utiliser dans la paraphrase grâce à un modèle statistique. Ce composant est en réalité composé d'un modèle de paraphrase, d'un modèle de la

51. https ://github.com/lanwuwei/Twitter-URL-Corpus
52. Un auto-encodeur variationnel consiste en un encodeur, un décodeur et une fonction de perte (Altosaar, 2019).
53. http ://cocodataset.org/#home

langue (tri-gram) et d'un modèle d'utilisabilité (permettant de contrôler l'adéquation, la fluence et l'utilisabilité).

Cette technique a obtenu de meilleurs résultats que la *baseline* de l'époque : une approche basée sur la traduction automatique.

1.6.6 Autres méthodes

Li et al. (2017) donnent à voir dans leur article de nombreuses autres techniques telles que :

- L'implémentation de cellules LSTM dans le modèle Seq2Seq pour élargir la capacité du modèle ;
- L'utilisation d'un vocabulaire additionnel pour restreindre le nombre de mots candidats durant la génération ;
- L'exploitation de l'information syntaxique pour la génération de paraphrases ;
- L'utilisation de données bilingues pour produire des paraphrases en pivotant sur une traduction partagée dans une autre langue ;
- L'utilisation de la traduction automatique neuronale pour générer des paraphrases grâce à la traduction arrière de paires de phrases bilingues ;
- L'utilisation des *generative adversarial networks* ou *rankGAN* (Lin et al., 2017).

1.7 Synthèse

L'état de l'art nous a permis de clarifier la notion de chatbot en confrontant les nombreuses définitions qui cohabitent dans ce domaine. Après avoir mis en avant diverses caractéristiques telles qu'une prétendue intelligence, les objectifs visés, les interfaces et les oppositions par rapport à d'autres systèmes similaires, nous avons pu élaborer notre propre définition du chatbot qui est, pour rappel, la suivante :

> Un chatbot est un logiciel, muni ou non d'un avatar, ayant pour objectif de tenir une conversation de plusieurs tours de parole dans un but déterminé en utilisant les langues naturelles sur un sujet le plus souvent précis, le tout en donnant le sentiment à l'utilisateur de parler avec un humain.

Nous avons ensuite retracé, brièvement, la longue histoire de cette technologie afin d'en offrir un aperçu fidèle et actuel. L'importance, chiffrée, de ces agents conversationnels a également été dévoilée.

Après avoir introduit la définition de notre sujet d'étude et retracé son histoire, nous nous sommes penché sur son fonctionnement général. Nous avons également décrit le concept d'*intent*, la tâche d'extraction des entités ainsi que

l'importance d'une base de données de bonne qualité. Il nous semblait en effet important d'introduire la technologie de façon globale et intelligible avant de nous plonger dans une description détaillée de l'état de l'art.

Nous avons, en premier lieu, porté notre attention sur la classification des *intents*, élément central de ce mémoire. Nous y avons décrit linguistiquement la notion d'*intent* afin de clarifier ce concept qui se situe au coeur de notre réflexion. Ensuite, le problème de la classification a été détaillé avant de décrire rigoureusement les différents modèles permettant de mener à bien cette tâche. Ces recherches nous ont permis de poser l'hypothèse suivante :

> La génération automatique de données supplémentaires dans la base de données d'un chatbot peut améliorer les performances de la classification des *intents*.

C'est donc naturellement que nous avons investi ce domaine afin de constater ce qui s'y était déjà fait. Parmi les différentes techniques que sont le *crowdsourcing*, l'acquisition automatique depuis des forums ou la génération de paraphrases, cette dernière solution a éveillé notre intérêt. Nous estimons que la mise en place d'un outil automatique de génération de paraphrases peut être une plus-value intéressante dans le domaine des chatbots. Cette méthode ne nécessite en effet aucune intervention humaine, contrairement au *crowdsourcing* et à l'acquisition depuis des forums.

Nous avons finalement investi le domaine de la génération automatique de paraphrases et avons expliqué les différentes technologies permettant de la mener à bien. Après réflexions, nous avons décidé d'implémenter un système basé sur un thesaurus, *ReSyf* (Billami et al., 2018) [54], afin de générer des paraphrases. Nous utilisons parallèlement le corpus de paraphrases *PPDB* (Ganitkevitch et al., 2013) afin de retrouver ou de générer des paraphrases depuis le NLU du chatbot. Ces deux techniques, bien séparées, seront comparées. Nous sommes bien conscient que ces méthodes ne sont pas les plus poussées ni même les plus efficaces de toutes. Cependant, nous estimons que leur implémentation dans le cadre de l'ajout de données dans le NLU d'un chatbot francophone est un parti pris original. L'anglais domine en effet les publications et autres innovations dans ce secteur. Nous désirons mettre à l'épreuve notre hypothèse avec ces systèmes afin de déterminer s'il est nécessaire de développer des modèles très complexes pour espérer obtenir des résultats probants.

54. Un dictionnaire de synonymes gradués.

Chapitre 2

Implémentations

Nous avons pu constater que la création d'un chatbot passe par une étape de construction d'une base de données. Nous avons expliqué en quoi cette étape était cruciale pour le bon fonctionnement du système et, particulièrement, pour la bonne classification des *intents*. Une grande diversité d'énoncés, de structures et de vocabulaire est importante pour l'élaboration d'un système robuste. Cependant, cette étape est très souvent manuelle ou bien supervisée par le développeur d'une façon ou d'une autre. La base de données est alors construite par le biais du prisme linguistique du développeur. Les données manquent alors inévitablement de diversité et de créativité.

Nous souhaitons dans cette section explorer l'efficacité, ou l'inefficacité, de méthodes automatiques d'ajout de paraphrases dans cette base de données. Nous espérons que ces procédures automatiques permettront d'insuffler de la diversité linguistique sans pour autant perdre en cohérence syntaxique ou sémantique. Objectif ambitieux que celui de mimer les capacités linguistiques d'un humain tout en profitant de ressources linguistiques variées.

Nous commencerons tout d'abord par décrire plus explicitement le fonctionnement de Rasa, les librairies qu'il utilise, un système de génération supervisé qu'il recommande ainsi que toute la procédure de création d'un chatbot. Nous expliquerons également les techniques qu'utilise Rasa pour classer les *intents*. Nous décrirons ensuite brièvement le cadre méthodologique et les données que nous utiliserons dans la construction et l'évaluation de notre programme.

Après avoir décrit les corpus et les autres données, nous plongerons dans le vif du sujet en décrivant les deux approches que nous avons explorées. La première consiste en l'utilisation d'un corpus de paraphrases pour la recherche et la construction d'énoncés pertinents à ajouter dans le NLU. La seconde, quant à elle, a pour objectif d'utiliser un dictionnaire de synonymes pour générer des paraphrases. Toutes les implémentations se feront en Python 3.7.

2.1 Fonctionnement de RASA

Rasa est un outil de création de chatbots *open source*, nous allons expliquer son fonctionnement de façon plus précise mais il ne nous sera pas possible d'atteindre l'exhaustivité de la documentation officielle dont nous nous inspirons [1].

2.1.1 Architecture générale

Rasa (Bocklisch et al., 2017) est un logiciel *open source* qui propose deux modules : Rasa NLU et Rasa Core. Le premier permet de mettre en place la compréhension du langage naturel tandis que le second s'occupe du « dialog management », la gestion des dialogues. L'objectif, selon les créateurs, est de combler le fossé entre la recherche et l'application et d'amener les récentes avancées du *machine learning* à des personnes inexpérimentées qui souhaitent implémenter des agents conversationnels. Les modules NLU et Core peuvent être utilisés ensemble ou séparément, la modularité étant un parti pris pour donner un maximum de liberté au développeur. Il est donc possible d'utiliser le module Core avec un autre système NLU et inversement. Le code est entièrement écrit en Python et il est possible de le modifier aisément.

Un agent conversationnel procède généralement de la même manière pour mener à bien la tâche qui lui est confiée. Le système que nous utiliserons, Rasa, procède comme suit :

— Réception du message de l'utilisateur, classification de l'*intent* et extraction des entités (NLU) ;
— Un *tracker* maintient la discussion et est notifié du message reçu ;
— Le « policy » reçoit l'état courant du *tracker* et choisit une action à prendre ;
— L'action est ensuite enregistrée dans le *tracker* et exécutée (la fonction peut générer l'envoi d'une réponse, un calcul ou autre chose).

2.1.1.1 Rasa NLU

Les phrases données en entrée au chatbot sont représentées par une mise en commun de vecteurs de mots pour chaque token. Pour ce faire, Rasa utilise (dans son modèle recommandé) des *words embeddings* préentrainés (*GloVe*). Selon eux, le classificateur d'*intents* est robuste. Les *intents* et autres entités sont alors extraits et donnés au module Core qui s'occupe de décider de la prochaine action selon ce qu'il a reçu. Rasa NLU peut être utilisé avec plusieurs modèles (ou *pipelines* [2] : *tensorflow_embedding* ou bien *spacy_sklearn*). Les deux ont

1. https ://rasa.com/
2. https ://rasa.com/docs/nlu/choosing_pipeline/

leurs propres avantages et spécificités : *spacy_ sklearn* utilise des *words embeddings* déjà préentrainés (*GloVe*) alors que *TensorFlow* les construit directement depuis les données du chatbot. *Spacy_ sklearn* devrait donc être préféré lorsque la base de données est assez faible alors que *TensorFlow* peut être utilisé avec beaucoup d'intérêt si la base de données disponible est suffisamment conséquente et pertinente. Nous allons nous intéresser plus spécifiquement à *spacy_ sklearn*, le *pipeline* recommandé :

- Tokenisation, *POS tagging* et vectorisation de chaque token grâce à *GloVe* pour créer une représentation de la phrase ;
- *Scikit-learn classifier* entraine ensuite un estimateur pour le jeu de données. Par défaut, il consiste en un *multiclass support vector* (type particulier de SVM) dans le cadre de la classification d'*intents*. Les entités sont reconnues grâce au module *ner_ crf* qui a pour tâche d'entrainer un *conditional random fiel* (CRF) qui utilise les tokens et les *POS tags*. D'autres composants permettent de gérer les *out-of-vocabulary* (OOV) et proposent de nombreuses personnalisations.

De façon générale, Rasa encourage les développeurs à entrainer leurs chatbots de façon interactive, en corrigeant ses erreurs. Cette méthode (*machine teaching*) permet d'améliorer significativement les résultats obtenus en « enseignant » les bonnes réponses à son chatbot. Rasa Core, pour sa part, ne reçoit que les *intents* et les entités reconnus.

2.1.1.2 Rasa Core

Ce module retiendra moins notre attention car il n'a pas d'influence sur la classification des *intents*. En effet, il intervient en aval.

La question du *dialogue management* est considérée comme un problème de classification. Pour chaque itération, Rasa Core prédit quelle action doit être exécutée depuis une liste. Il utilise pour ce faire les *intents* et les entités du message le plus récent. L'état du dialogue est conservé dans un *tracker object*, il en existe un seul par discussion.

Les données cruciales qu'utilise le Core sont les *stories*. Une *story* est un enchainement logique d'*intents* permettant de construire un fil de dialogue. Elles sont particulièrement puissantes dans les systèmes *goal oriented* car elles participent grandement à la bonne gestion du dialogue.

```
## conso path 1          -> nom de la story
* greet                 -> nom de l'intent attendu (bonjour)
  - utter_greet         -> action à exécuter (accueillir l'utilisateur)
* conso
  - utter_conso
  - utter_goodbye
```

Cette *story* prend donc en charge une discussion de ce type :

```
Utilisateur : - Bonjour Bot!
Chatbot : - Bonjour, comment puis-je vous aider?
U : - Je voudrais connaitre la consommation de cette voiture.
C : - Cette voiture consomme 4,7 litres pour cent kilomètres.
C : - Au revoir!
```

La classification des *intents* intervient donc en amont et donne au Core l'*intent* reconnu afin de décider de l'action suivante la plus probable.

2.1.2 Interface en ligne de commande

Rasa s'utilise très facilement en interface de lignes de commande. Depuis le répertoire principal du chatbot, différentes commandes, qui ont été largement simplifiées depuis 2018, peuvent être lancées :

```
rasa train      -> déclenche l'entrainement du modèle
rasa shell      -> déclenche le chatbot dans la console pour le tester
rasa test       -> contient de nombreuses options afin de tester son
                   chatbot et produire des sorties
...
```

2.1.3 Librairies utilisées par Rasa

Rasa ne réinvente pas la roue. Le système est composé d'une multitude de librairies et modules *open source* afin de reposer sur des bases solides. Ainsi est permise l'extraction d'entités grâce à *sklearn-crfsuite*[3], *spaCy*[4], *duckling*[5] ou *MITIE*[6] tandis que la classification des *intents* s'effectue grâce à *Spacy_sklearn* (et *GloVe*[7]) ou *TensorFlow*[8].

Il est également possible de lier son chatbot à des plateformes telles que Messenger, Telegram ou encore Slack.

2.1.4 Chatito

Il existe déjà un outil de génération de données d'entrainement pour Rasa mais il repose sur un principe bien différent. En effet, *chatito*[9] prend en entrée

3. https ://sklearn-crfsuite.readthedocs.io/en/latest/
4. https ://spacy.io/
5. https ://duckling.wit.ai/
6. https ://github.com/mit-nlp/MITIE
7. https ://nlp.stanford.edu/projects/glove/
8. https ://www.tensorflow.org/
9. https ://rodrigopivi.github.io/Chatito/

des blocs de texte définis par le développeur ou des entités afin de reconstituer des phrases similaires [10].

```
Unités:
@[city]                     -> entités
    ~[new york]
    ~[san francisco]
    ~[atlanta]

~[find]                     -> unités définies par l'utilisateur
    find
    i'm looking for
    help me find

Règle:
~[hi?] ~[please?] ~[find?] ~[restaurants] ~[located at] @[city] ~[city?] ~[thanks?]

Sortie:
"please places to eat located at atlanta thank you"
"hey plz i'm looking for places to eat located at atlanta city thanks"
```

Cette approche, selon Rasa, peut mener à un *overfitting* si trop d'entités sont utilisées [11]. De plus, elle nécessite l'intervention du développeur dans la création des jeux d'entités et des syntagmes à utiliser. La tâche peut dès lors devenir laborieuse si le chatbot est un peu complexe. Nous pensons nous différencier de cette approche par notre optique presque indépendante du développeur. La génération de données que nous avons mise en place fonctionne de façon automatique, depuis des ressources linguistiques telles qu'un corpus de paraphrases ou un dictionnaire de synonymes. Cependant, il reste à constater si la pertinence des données créées par *chatito*, provenant de décisions humaines, se retrouve également dans notre approche.

2.1.5 Processus de création d'un chatbot

Il n'est pas possible d'être aussi complet que le tutoriel officiel [12] mais nous résumerons ici, de façon synthétique, le processus de création d'un chatbot.

Pour les besoins de l'expérience et afin d'être en mesure de vérifier si nos implémentations sont significativement intéressantes, nous avons créé un chatbot assez simple. Le contexte est le suivant :

10. L'exemple suivant est directement repris de Chatito.
11. https ://rasa.com/docs/rasa/nlu/training-data-format/
12. https ://rasa.com/docs/rasa/user-guide/rasa-tutorial/

> L'utilisateur est à la recherche d'une voiture d'occasion et trouve un site web d'un garage automobile qui a pour spécialité la vente de véhicules d'occasion. À la pointe de la technologie, ce garage a intégré un chatbot à son site pour répondre plus facilement aux très nombreux internautes.

En ce qui concerne la sélection du véhicule, nous estimons qu'il serait laborieux de la faire par le biais du chatbot. En effet, les photos sont importantes et un utilisateur pourrait être déstabilisé de ne pas avoir accès (ou avoir un accès limité de par la petite taille de la fenêtre de messagerie) à cette dimension visuelle. Nous partons donc du principe que le chatbot sait de quelle voiture il s'agit car il s'activerait après la sélection de la voiture. Il se peut que ce schéma ne soit pas très réaliste mais l'intention première lors de la création de ce chatbot était surtout de mettre une dizaine d'*intents* sur un même niveau afin de tester la classification de Rasa. L'utilisation ou le caractère factice du chatbot n'importe donc pas.

2.1.5.1 Données nécessaires

Il est demandé au développeur de fournir des données, très généralement écrites à la main : les exemples d'entrainement NLU et les *stories*.

NLU Il est possible d'utiliser le format *Markdown* ou *JSON* pour créer ce fichier. Nous avons décidé d'utiliser dans un premier temps le format *Markdown*, très lisible et pratique pour des ajouts manuels.

```
## intent:greet          -> nom de l'intent
- hey                   -> un exemple d'entrainement
- hello
- bonjour
```

Nous y avons décrit 10 *intents* accompagnés chacun de 10 exemples d'entrainement créés à la main. C'est très peu mais cela permet déjà à Rasa de fonctionner un minimum, à condition de ne pas donner un *input* trop différent de ce qui est spécifié dans ce fichier. Les dix *intents* sont les suivants :

- Dire bonjour ;
- Dire au revoir ;
- Donner la consommation de la voiture ;
- Expliquer les difficultés ou facilités pour assurer l'auto ;
- Expliquer si le moment est propice pour l'acheter ou si la cote va descendre ;
- Donner les caractéristiques de la voiture, les options ;
- Donner les noms de marques vendues par le garage ;
- Donner les détails sur la garantie ;

— Donner le nom du carburant de la voiture ;
— Raconter une blague.

Il est possible de critiquer cette création manuelle, ex nihilo, des données d'entrainement mais notre optique est bien de démarrer d'un système primitif, doté d'un jeu de données faible mais pertinent. Notre expertise linguistique nous a guidé dans la rédaction de ce fichier. De plus, les développeurs de chatbot qui utilisent Rasa passent impérativement par cette phase manuelle. Nous ne pensons donc pas biaiser notre expérience. Une fois cette étape achevée, nous sommes passé à la construction des *stories*.

Stories Comme expliqué précédemment, les *stories* sont un outil puissant et indispensable dans Rasa. Le chatbot que nous avons créé est très simple, nous avons simplement mis les *intents* sur un même palier de discussion pour s'intéresser uniquement à leur classification. Pour ce faire, nous avons créé une dizaine de *stories* de ce format :

```
## assurance path 1          -> nom de la story
* greet                     -> nom d'un intent
  - utter_greet              -> nom d'une action (dire bonjour)
* assurance
  - utter_assurance
  - utter_goodbye

## bon-marche path 1
* greet
  - utter_greet
* bon-marche
  - utter_bon-marche
  - utter_goodbye
```

Ces *stories*, triviales, construisent un arbre de discussion (Annexes Fig A.2). Nous pouvons constater qu'il est basique. Il nous semble, pour les besoins de l'expérience, beaucoup plus simple d'utiliser un chatbot de ce type plutôt qu'un système plus complexe, aux ramifications diverses et étendues. Nous estimons en effet que si notre système améliore la classification de ces *intents* mis au même niveau, il devrait fonctionner pour tout *intent* à n'importe quel niveau de la discussion. La détection de l'*intent* se produit en effet avant que le module Core ne charge les *stories*. Cela signifie que si un mauvais *intent* est reconnu et qu'il n'y a pas de *stories* prévues pour ce cas de figure, le chatbot tombe dans une erreur. Si une *story* comporte l'*intent* mal classé, elle sera sélectionnée mais la réponse du chatbot ne sera dès lors pas pertinente par rapport à ce que l'utilisateur attendait.

NLU.json Nous avons estimé que l'ajout de nouvelles données d'entrainement serait beaucoup plus aisé dans un fichier *JSON* que *Markdown*. Nous avons alors procédé à une manipulation manuelle. Le fichier *nlu.md*, une fois l'entrainement effectué, entraine la création d'un fichier rassemblant ces données sous un format *JSON* (/models/NOM-DU-MODÈLE/nlu/training_data.json). Sachant que le format est expressément créé pour correspondre à ce qu'attend Rasa et qu'il est possible de renseigner un fichier *nlu.json* à la place du fichier *nlu.md*, nous avons simplement copié et renommé ce fichier json en *nlu.json* et l'avons mis à la place du fichier *nlu.md*. Le dossier data se présente alors de cette façon :

```
/data
    nlu.md
    stories.md

/data
    nlu.json
    stories.md
```

Cette manipulation nous permettra ensuite d'ajouter aisément des exemples d'entrainement dans un format simple d'usage.

2.1.5.2 Config

Le fichier *config.yml* permet de configurer les *pipelines* que le chatbot utilisera. Dans ce cas-ci et pour le NLU, nous chargeons le modèle préentrainé de *spaCy* (préalablement installé et lié au mot-clé « fr ») afin de reconnaitre les *intents*. Les paramètres du Core sont par défaut.

```
# Configuration for Rasa NLU.
# https://rasa.com/docs/rasa/nlu/components/
language: fr
pipeline: "pretrained_embeddings_spacy"

# Configuration for Rasa Core.
# https://rasa.com/docs/rasa/core/policies/
policies:
  - name: MemoizationPolicy
  - name: KerasPolicy
  - name: MappingPolicy
```

C'est donc dans ce fichier qu'il est possible de modifier le classificateur d'*intents* : soit *spaCy*, soit *TensorFlow*.

2.1.5.3 Domaine

Le domaine est un « fichier-synthèse ». Il reprend en effet l'essentiel de l'univers dans lequel le chatbot évoluera : la liste des *intents*, les actions, les *templates* (réponses du chatbot) et les entités.

```
intents:
  - greet
  - conso

actions:
- utter_greet
- utter_conso

templates:
  utter_greet:
  - text : "Salut! Comment puis-je t'aider?"

  utter_conso:
  - text : "Cette voiture consomme 4,7 litres pour cent kilomètres."
```

Nous ne nous soucions pas des entités dans notre cas, nous nous focalisons principalement sur les *intents*. De plus, les entités sont extraites après que la classification des *intents* ait eut lieu. Cette tâche fait partie du domaine de la reconnaissance d'entités nommées. C'est évidemment un problème important car il permet d'extraire des informations pertinentes à partir des énoncés. Un mémoire entier pourrait sans doute porter sur le sujet, nous préférons donc ne pas nous y aventurer et nous concentrer sur la classification des *intents*.

2.1.6 Rasa et la classification d'intents

Le blog de Rasa héberge depuis peu des articles traitant de sujets tels que la classification d'*intents*[13] ou l'extraction des entités en vue de faciliter leur compréhension par la communauté. Nous résumerons donc leurs explications afin d'offrir une vue synthétique mais précise de leur classificateur.

Rasa NLU utilise ce que l'on appelle un *pipeline*.

> « Un pipeline définit différents composants qui traitent un message utilisateur de manière séquentielle et conduisent finalement à la classification des messages utilisateur en *intents* et à l'extraction d'entités »[14].

13. https ://blog.rasa.com/rasa-nlu-in-depth-part-1-intent-classification/
14. https ://blog.rasa.com/rasa-nlu-in-depth-part-1-intent-classification/ : A pipeline defines different components which process a user message sequentially and ultimately lead to the classification of user messages into intents and the extraction of entities.

La classification des *intents* est donc gérée par un composant au sein de ce *pipeline*. Les deux composants utilisables sont :

— *Pretrained Embeddings (Intent_ classifier_ sklearn)*
— *Supervised Embeddings (Intent_ classifier_ tensorflow_ embedding)*

Pour simplifier la compréhension des différences entre les deux systèmes, Rasa a créé un organigramme pour que la communauté s'y retrouve aisément (Annexes Fig A.1).

2.1.6.1 Sklearn (spaCy)

Comme expliqué brièvement précédemment, le modèle recommandé est le premier. Ce classificateur utilise la bibliothèque *spaCy* pour charger des modèles de langue préentrainés qui sont ensuite utilisés pour représenter chaque mot du message en *words embeddings*. Ce sont des représentations vectorielles de mots, ce qui signifie que chaque mot est converti en un vecteur numérique. Les *words embeddings* capturent les aspects sémantiques et syntaxiques des mots. Des mots similaires doivent donc être représentés par des vecteurs similaires (Mikolov et al., 2013). Ces *words embeddings* sont spécifiques aux langues. Il faut donc choisir et télécharger le modèle de la langue utilisée et le lier au nom donné à Rasa dans le fichier de configuration (« fr »).

```
python -m spacy download fr_core_news_sm
python -m spacy link fr_core_news_sm fr
```

Ensuite, Rasa NLU prend la moyenne des *words embeddings* d'un message et effectue une recherche par grille pour trouver les meilleurs paramètres pour le SVM qui classe les moyennes des *embeddings* dans les différents *intents*.

Avantages Utiliser ces *embeddings* préentrainés permet de profiter des dernières avancées dans leur création afin de recourir à des *embeddings* plus puissants et sémantiquement pertinents. De plus, le SVM a plus de facilités à utiliser ce type de données pour classer les *intents* car les *embeddings* sont déjà entrainés. C'est pour ces raisons qu'il est le plus souvent utilisé au début du développement d'un chatbot, lorsqu'il y a peu de données, car il permet de donner une classification robuste.

Désavantages Les *words embeddings* ne sont pas disponibles pour toutes les langues car ils sont le plus souvent construits depuis de larges bases de données publiques. Toutes les langues n'en possèdent pas. Ils ne permettent pas non plus de couvrir des mots spécifiques à un domaine tels que des noms de produits ou des acronymes. C'est pour ces raisons qu'il est possible d'utiliser *TensorFlow* pour calculer directement les *embeddings* depuis ses propres données.

2.1.6.2 TensorFlow

Le classificateur *Intent_ classifier_ tensorflow_ embedding* a été développé par Rasa et crée les *words embeddings* depuis les données d'entrainement renseignées par l'utilisateur. Il est utilisé avec un composant, *intent_ featurizer_ count _ vectors*, qui compte la fréquence à laquelle des mots distincts des données d'entrainement apparaissent dans un message d'utilisateur. Il fournit ensuite le résultat au classificateur. Il est également possible d'utiliser un comptage de *n-grams* plutôt qu'un comptage de mots : cela a pour conséquence directe d'augmenter la complexité calculatoire du processus mais la classification des *intents* est alors plus robuste.

Avantages Les *words embeddings*, construits depuis les données NLU, sont adaptés au domaine, parfois bien spécifique, du chatbot. Il n'y a en effet pas de *words embeddings* manquants. De plus, cette technique est indépendante de la langue et supporte les *intents* multiples dans un même message. Par exemple, « Hi, how is the weather » [15] contient l'*intent* « greet » et « ask_weather ». Il est donc indiqué pour des utilisations plus avancées de Rasa. L'utilisation d'un large jeu de données ou la gestion d'*intents* multiples en sont des exemples.

Désavantages Étant donné que ce classificateur part de zéro et utilise uniquement les données d'entrainement, il a besoin d'un grand jeu de données. C'est le seul élément réellement bloquant.

2.2 Données

Nous présenterons, dans cette section, les différentes données qui entrent en jeu lors de l'expérimentation. C'est pourquoi nous décrirons brièvement la *baseline* à partir de laquelle nous appliquerons nos techniques de génération de paraphrases. La création ainsi que le contenu du jeu de tests seront décrits. Nous donnerons à voir le processus de création de ce corpus, plusieurs éléments statistiques et une brève analyse linguistique. Nous introduirons ensuite les bases de données de synonymes (*ReSyf*) et de paraphrases (*PPDB*).

2.2.1 Baseline

Nous avons déjà brièvement expliqué que nous avions créé de toutes pièces un jeu de données afin de constituer la *baseline* de notre expérience. Ce jeu de données est composé de 10 *intents* comprenant chacun 10 exemples d'énoncés. Nous avons essayé, tant bien que mal, d'insuffler de la diversité linguistique dans ces 10 exemples. Par exemple, pour l'*intent* « conso » :

15. Exemple tiré de : https ://blog.rasa.com/rasa-nlu-in-depth-part-1-intent-classification/

```
combien de L/100?
Quelle est la consommation de la voiture?
Elle consomme combien aux 100?
Elle consomme combien?
quelle est la consommation du véhicule ?
Peux-tu me donner la consommation de la voiture pour 100km
combien consomme la voiture
Le moteur consomme beaucoup?
Combien de litres aux 100?
chatbot, combien la voiture consomme?
```

C'est une tâche compliquée que de créer ce type de base de données. Il n'est pas aisé de créer des phrases très différentes de ce qui a déjà été écrit. Bien souvent, le développeur a tendance à penser le problème de la même manière et donc, à terme, de construire des phrases très similaires. Nous nous sommes inspirés du jeu de tests lorsque nous avions certaines difficultés à insérer des exemples pertinents et variés. Après avoir ajouté une phrase dans la *baseline* depuis ce jeu de données, nous y avons supprimé l'élément afin de ne pas biaiser l'expérience.

2.2.2 Jeu de tests

Afin de constituer rapidement un jeu de tests, nous avons lancé une collecte sur le réseau social Facebook. Nous y avons publié un sondage créé sur Google Forms (Annexes Fig D.1, D.2, D.3) [16] dans lequel nous expliquons ce qu'est un chatbot, le contexte dans lequel il se situerait (garage automobile) et ce que nous attendions de l'internaute. Nous avons collecté des données personnelles (sexe, age, langue maternelle) à des fins statistiques et avons ensuite demandé à l'internaute de renseigner une phrase qu'il serait susceptible d'utiliser pour obtenir 10 types d'information (les 10 *intents*).

La démarche est artificielle, l'internaute n'est pas inscrit dans un contexte réel et non biaisé, mais cela nous semble négligeable pour les raisons suivantes. Premièrement, il s'agit d'une collecte somme toute assez simple qui n'a pas pour vocation de créer un jeu de données représentatif d'une population. Il nous faut simplement des énoncés différents de ceux déjà renseignés dans le fichier NLU afin de constituer un jeu de tests. Pour cette raison, nous nous intéressons exclusivement à la pertinence linguistique des énoncés, et non à leur représentativité. Ensuite, il aurait en effet été possible de créer un chatbot simple et de le lier sans trop de difficultés à Messenger [17], le service de messagerie de Facebook, mais rien ne nous aurait alors assuré qu'un utilisateur renseigne un énoncé pour chaque *intent*. De plus, à moins de réellement disposer d'une base

16. Source de l'image utilisée dans l'entête de l'enquête : http ://penseeartificielle.fr/focus-reseau-neurones-artificiels-perceptron-multicouche/

17. La nouvelle version de Rasa rend ce type de connection assez simple.

de données de garage automobile et de donner à voir un chatbot fonctionnel, la démarche aurait sans doute été tout aussi artificielle, avec, en prime, une absence d'explication et de contexte. Le but de ce mémoire n'est évidemment pas de créer réellement un chatbot pour les garages automobiles.

Les données d'une cinquantaine d'internautes rassemblées, nous avons décidé de mettre fin à la collecte pour commencer le prétraitement. Nous avons utilisé un outil en ligne [18] afin de supprimer les doublons et de ne conserver que des phrases uniques. Nous avons pour cela introduit les énoncés de la *baseline* et avons ensuite ajouté ce jeu de tests. L'outil a donc supprimé tous les doublons des énoncés collectés par rapport à notre *baseline* et à ses propres données. Il faut spécifier que nous avons décidé d'ignorer la casse afin de n'avoir que des énoncés originaux. Il restait ensuite 414 énoncés sur 500, 86 ont donc été supprimés par cet outil. Il est à noter que les *intents* « greet » et « goodbye » ont particulièrement été impactés par la suppression des doublons. Cela est dû à la faible créativité et diversité linguistique qui les concerne par rapport aux autres *intents* qui, bien souvent, demandent des phrases plus longues, donc statistiquement plus propices à la variation linguistique.

Nous avons ensuite procédé à une phase manuelle de nettoyage qui a abaissé le nombre d'énoncés à 375. En effet, certains énoncés étaient absurdes, non pertinents par rapport à l'*intent* ou injurieux.

```
absurde : Tout ca c'est beau mais niveau tuning ca donne quoi?
                L'moteur c'est quoi sa cavalerie? (caractéristiques)
non pertinents ou discutables : elle boit combien au 100 ? (conso)
                                avez-vous les gr. de co2 par km? (conso)
ambigu : que consomme la voiture comme carburant? (conso et carburant)
```

Statistiques Nous n'avons pas collecté beaucoup de métadonnées, le but de cette petite enquête n'étant pas d'aboutir à une enquête sociolinguistique. Cependant, nous pouvons mettre en avant les informations que nous avons reçues.

Les répondants à l'enquête sont principalement jeunes (18-25 ; 25-30) et dépassent marginalement les 30 ans (Annexes Fig D.4). L'échantillon est composé de 44% d'hommes et de 52% de femmes (Annexes Fig D.5). Le français est la langue maternelle de tous les individus.

Description linguistique du corpus De façon générale, ce petit corpus contient des énoncés à l'orthographe variée et aux tournures syntaxiques diverses.

18. http ://www.textwidgets.com/dedupelist.html#startresults

Phénomène	**Exemples**
Interpellation	*dis chatbox, quelle est la consommation de la voiture ?*
Écriture orale	*ils vendent quels marques dans ce garage ?*
Inversion	*y a-t-il des éléments à prendre en compte pour le cout de l'assurance ?*
Mots-clés	*prix assurance / bonne occasion ? / description ?*
Longues phrases	*est-elle facile à assurer pour quelqu'un qui ne s'y connait pas forcément ou y a-t-il des petits « trucs » à savoir ?*
Phrases complexes	*est-ce le bon moment pour acheter en vue de la conjoncture du marché automobile actuelle et à venir ?*

Le jeu de données est assez varié. Certains interpellent le chatbot, posent leurs questions grâce à des inversions tandis que d'autres écrivent plus oralement. Des énoncés sont parfois réduits à un mot [19] ou alors constitués d'un très grand nombre d'éléments. Généralement simples, certaines phrases sont parfois beaucoup plus complexes en ce qui concerne le niveau de langue, les choix des mots et des tournures syntaxiques. Le tout est évidemment parsemé d'erreurs de langue : orthographe, conjugaison, etc. Nous avons également remarqué que les internautes tendent à se positionner sur un même niveau de langue. Cela se devine notamment grâce à l'emploi majoritaire du terme « voiture », plus formel que « auto » mais moins que « véhicule ».

Transformation en Markdown *Google Forms* nous permet d'exporter les données en *csv* afin de les lire sous *Excel* ou *Libre Office Calc*. Nous avons manuellement copié-collé les énoncés et avons créé un fichier NLU sous le format *Markdown*. C'est lors de cette transformation que nous avons supprimé les doublons et nettoyé le corpus.

2.2.3 Utilisation de l'API de ReSyf

ReSyf dispose d'une API Python que nous pouvons aisément utiliser [20]. Ses fonctions nous permettent de retrouver un terme et ses synonymes.

```
get_synonyms(lexicalRes, term, pos, sense_id=None):
```

Pour ce faire, il est nécessaire de donner en argument le mot et son étiquette morphosyntaxique. Nous devrons donc passer par une étape de *tagging*.

Les entrées de *ReSyf* (Annexes Fig B.1), en format simplifié, ressemblent à cela :

19. Certains semblent donc voir ce système comme une possibilité de recherche par mots-clés.
20. https ://gitlab.com/Cental-FR/resyf-package

```
    lexeme = "manger peu"
    partOfSpeech = "VER"
Synonymes:
    word = "manger peu"
    rank = 1
    word = "avoir un appétit d'oiseau"
    rank = 2
    word = "chipoter"
    rank = 3
    word = "picorer"
    rank = 4
```

La particularité de ce dictionnaire consiste en la gradation attribuée aux synonymes. Plus la valeur de *rank* est haute, plus le synonyme est difficile ou spécifique. Nous avons, dans nos premières approches, essayé de déterminer si une sélection de synonymes par rapport à leur rang donnait de meilleurs résultats. Cependant, il s'est avéré qu'en utilisant la ressource de cette façon, nous perdions toujours des synonymes pertinents au profit d'autres moins adaptés. Nous avons donc décidé d'utiliser tous les synonymes d'un même mot pour créer des paraphrases que nous évaluons par après pour sélectionner les meilleures.

2.2.4 PPDB

Le corpus de paraphrases *PPDB* (Ganitkevitch et al., 2013) est la base de données de paraphrases [21] que nous utiliserons.

> « *PPDB* est une base de données extraite automatiquement [22] contenant des millions de paraphrases dans 16 langues différentes. L'objectif de *PPDB* est d'améliorer le traitement automatique des langues en rendant les systèmes plus robustes à la variabilité linguistique et aux mots inconnus (Ganitkevitch et al., 2013) » [23].

Il existe des versions de cette base de données pour une mutitude de langues et d'applications. En effet, il est possible de choisir entre des données lexicales, phrasales et syntaxiques. Un éventail de tailles de corpus différentes est également proposé. Dans un premier temps, nous avons téléchargé le plus gros corpus *phrasal* pour le français. En effet, il nous semblait plus indiqué car il fait correspondre des unités phrastiques à d'autres. Les corpus lexicaux et syntaxiques ne comportent pas cette dimension qui nous parait importante :

21. http ://paraphrase.org/#/download
22. Les paraphrases sont classées grâce à un modèle de régression supervisée. Voir Pavlick et al. (2015).
23. PPDB is an automatically extracted database containing millions paraphrases in 16 different languages. The goal of PPDB is to improve language processing by making systems more robust to language variability and unseen words.

```
Exemples tirés du corpus Phrasal (taille S, paraphrases, français)

soutien communautaire ---> l'appui de la communauté
deux catégories de ---> deux types de
expansion du marché ---> de développement des marchés
des agents humanitaires ---> travailleurs humanitaires
es en sécurité ---> ne crains rien
```

Il comporte un très grand nombre d'énoncés (+100Go) accompagnés de leur paraphrase et de métadonnées. Nous avons décidé de modifier le format de ces données afin de gagner en efficacité calculatoire et en espace disque. Ainsi, nous avons créé une table *SQLite* comportant 2 informations : la phrase et sa paraphrase associée. La création de cette table a permis de diminuer le volume du corpus de façon significative (14Go). Son utilisation reste cependant laborieuse et très gourmande. Il faut en effet itérer sur la base de données et comparer l'énoncé avec chaque phrase afin de trouver une correspondance.

Souhaitant avant tout mettre au point un système viable d'un point de vue calculatoire, nous avons décidé d'utiliser une version réduite de ce corpus. Il est à noter que :

> « [...] la base de données [est disponible] en six tailles, de S à XXXL. S ne contient que les paires avec le plus haut score, pour une précision maximale, tandis que XXXL contient toutes les paires, pour un meilleur rappel. Le nombre de paraphrases double à chaque niveau et les plus grandes tailles contiennent les plus petites (Ganitkevitch et al., 2013) » [24].

Un corpus plus petit équivaut donc à utiliser de meilleures paraphrases mais à perdre de l'information qui pourrait être intéressante dans notre cas.

Exemple Une entrée de *PPDB* ressemble à ceci (Annexes Fig C.1) :

```
    [X] ||| cession des droits ||| transfert de droits ||| Abstract=0 Adjacent=0
    Alignment=0-0:1-1:2-2 CharCountDiff=1 CharLogCR=0.05407 ContainsX=1
GlueRule=0 Identity=0 Lex(e|f)=8.06829 Lex(f|e)=5.48712 Lexical=1 LogCount=0
Monotonic=1 PhrasePenalty=1 RarityPenalty=0 SourceTerminalsButNoTarget=0
SourceWords=3 TargetTerminalsButNoSource=0 TargetWords=3 UnalignedSource=0
UnalignedTarget=0 WordCountDiff=0 WordLenDiff=0.33333 WordLogCR=0 p(LHS|e)=0.28768
p(LHS|f)=0.27354 p(e|LHS)=15.91153 p(e|f)=1.32290 p(e|f,LHS)=1.42234
p(f|LHS)=15.61255 p(f|e)=1.03807 p(f|e,LHS)=1.12336
```

24. [...] into six sizes, from S up to XXXL. S contains only the highest-scoring pairs, for the highest precision, while XXXL contains all pairs, for highest recall. The number of paraphrases doubles with each increase in size, and larger sizes subsume smaller sizes.

Nous nous intéressons seulement aux deux énoncés : « cession des droits » et « transfert de droits » que nous sauvegardons dans une base de données *SQLite* (Annexes Fig C.2). Cette dernière, comportant près de 8 000 000 d'entrées, sera notre base de travail pour la première approche que nous avons implémentée. Celle-ci consiste en la recherche de paraphrases au sein du corpus *PPDB* afin d'augmenter notre base de données NLU.

2.3 Génération depuis PPDB

Nous avons décidé d'utiliser la ressource *PPDB* afin de déterminer s'il était possible d'obtenir des résultats satisfaisants pour la recherche ou la création de paraphrases. Avant toute chose, nous allons exposer les problèmes rencontrés avec le corpus *PPDB*, les solutions mises en place ainsi que les différentes transformations que nous y avons apportées. Nous détaillerons ensuite les trois procédures que nous avons utilisées pour l'exploitation de cette ressource dans le cadre de l'ajout de données dans le NLU d'un chatbot.

2.3.1 Transformation de PPDB en SQLite

Le corpus *PPDB* n'est pas facilement manipulable. En effet, il est assez volumineux [25] et est en texte brut. Nous avons donc décidé de le transformer en une base de données facilement utilisable : *SQLite*. Cela nous a également permis de grandement diminuer la taille du corpus.

Après avoir tenté d'utiliser efficacement le plus gros corpus de *PPDB* (+100Go), nous nous sommes dirigé vers le plus petit (4,3Go). En effet, il nous semblait plus réaliste et beaucoup moins exigeant de fonctionner avec celui-là. De plus, ce corpus reste très intéressant car il concentre les paraphrases les plus qualitatives tandis que le plus gros les contient toutes. Une première sélection a donc déjà été opérée lors de la création du corpus.

2.3.1.1 Mémoire vive dépassée

Bien que beaucoup moins large que le précédent, le corpus ne tient tout de même pas dans notre mémoire vive. Afin de pouvoir le charger et le transformer en une base de données *SQLite*, nous avons donc dû procéder à un bricolage. Nous avons divisé le corpus en une multitude de fichiers plus petits (composés de seulement 200 000 lignes) que nous avons stockés dans un dossier. Ces derniers pèsent près de 100Mo, il n'y a donc aucun problème à les charger dans notre *RAM*. Ensuite, nous avons écrit un script Python (Annexes Code F.1) qui lit

25. Cela dépend de la taille sélectionnée mais même le plus petit pèse tout de même plus de 4Go.

ces fichiers et écrit leur contenu dans la base de données *SQLite*. Une fois le dernier fichier lu, la base de données est complète.

```
split -l 200000 ppdb-1.0-s-phrasal

-l pour diviser sur le nombre de lignes
```

Le fichier *SQLite*, uniquement composé de paires phrase-paraphrase, est approximativement 10 fois moins gros que l'original. Nous avons décidé de ne pas conserver les nombreuses métadonnées renseignées dans le corpus car nous n'en avions pas l'usage.

2.3.1.2 Format des énoncés du corpus

Selon les approches utilisées, nous avions, ou non, besoin de prétraiter les données du corpus. En effet, dans l'optique de trouver une phrase dans le corpus afin de pouvoir utiliser sa paraphrase, il était nécessaire d'uniformiser les formats. Certaines différences étaient présentes dans le corpus :

— Ponctuation différente : [c'_est | guadeloupe_,_et | établissement_. | france_,] ;
— Espaces superflus en début et fin de phrase : « _je veux te voir_ ».

Afin de résoudre ces problèmes, nous avons écrit un script Python (Annexes Code F.3) qui effectue ce prétraitement en uniformisant les données du corpus avant de les écrire dans la base de données SQLite.

Cependant, après avoir exploité le corpus de différentes manières, nous nous sommes rendu compte que cette tokenisation préalable pouvait nous être bénéfique. Nous avons donc choisi de conserver les deux corpus, *ppdb.sqlite* (prétraité) et *ppdb-2.sqlite* (sans prétraitement), afin de comparer leurs effets sur la génération de paraphrases et, à terme, sur la classification des *intents*.

2.3.2 Approches envisagées

Nous avons essayé d'utiliser cette base de données à de nombreuses reprises et suivant diverses approches. Nous avons tout d'abord tenté, plus naïvement, de trouver les phrases de la base NLU dans le corpus. Cette approche, rapide, ne fonctionne que pour des énoncés communs tels que : « Je le veux bien » mais est inutile pour des phrases plus longues ou spécifiques à un domaine particulier. Nous avons également essayé d'utiliser certaines mesures telles que BLEU ou la distance d'édition pour sélectionner des phrases similaires. Mais c'est alors le sens qui ne correspondait pas à l'énoncé original. De plus, ces approches sont très gourmandes car il faut calculer ces mesures pour chaque entrée du corpus. Nous avons donc abandonné ces fonctions mais les avons laissées dans le programme final pour information.

Nous nous sommes ensuite dirigé vers trois approches que nous testerons dans le cadre de notre expérimentation.

La première consiste à identifier, dans le corpus, des syntagmes de l'énoncé original et de les remplacer par leurs paraphrases.

La deuxième vise à réduire le nombre d'énoncés pertinents grâce au calcul du nombre de caractères et de mots afin de ne garder que les plus pertinentes avant de calculer la similarité entre l'original et les phrases du corpus et de sélectionner les meilleures.

La troisième opère une fusion des deux autres approches : après avoir créé une liste des mots (noms, adjectifs et verbes) [26] de la *baseline*, nous recherchons dans le corpus les phrases en contenant au moins un et les sauvegardons. Nous procédons ensuite, si les phrases sélectionnées sont des sous-parties des phrases originales, au remplacement de ces syntagmes par leurs paraphrases.

Les trois approches présentées ici sont rassemblées dans un seul programme : *findInSqlite.py* (Annexes Code F.2).

26. Nous avons décidé de nous intéresser uniquement à ces catégories de mots car elles sont particulièrement chargées en sens.

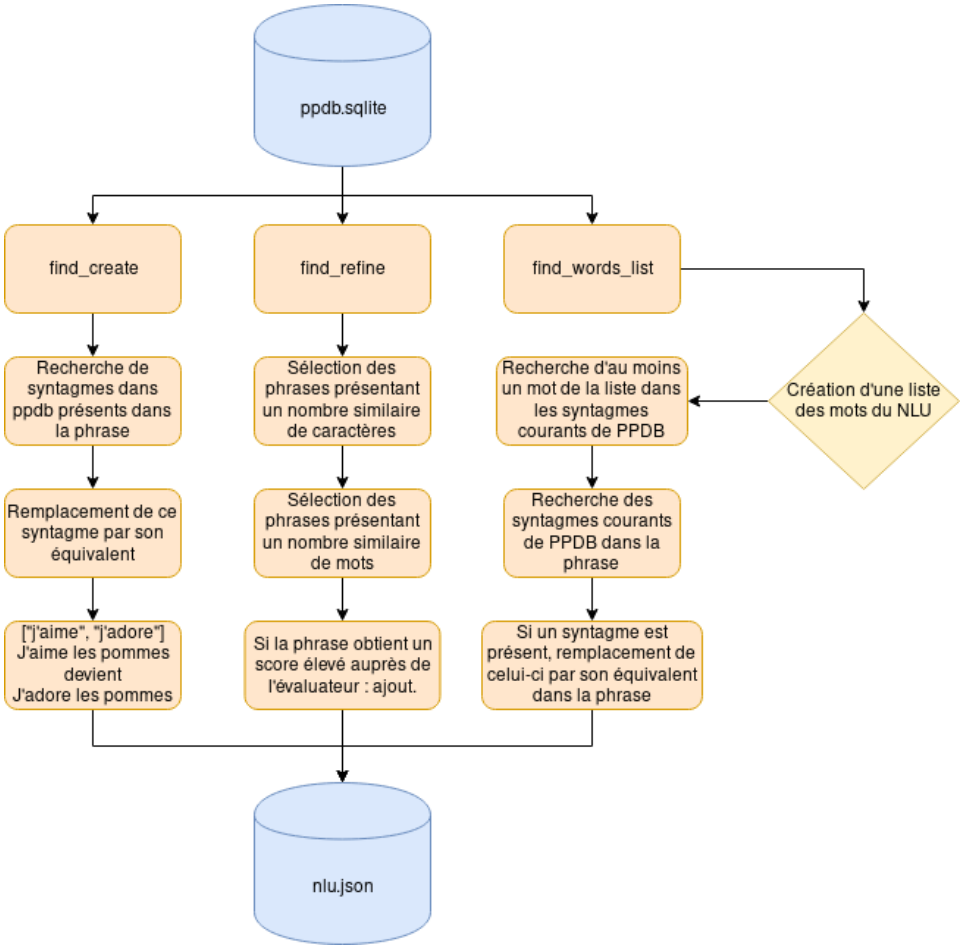


FIGURE 2.1 – Approches envisagées depuis le corpus PPDB

2.3.2.1 Première approche

Cette première approche (Annexes Code F.2), que nous avons baptisée *find_ create*, prend en argument la phrase courante du NLU et la ligne courante du corpus *PPDB*. Nous vérifions si la phrase courante contient le premier élément de la ligne de *PPDB*. Si c'est le cas, nous remplaçons cet élément par son équivalent.

```
je voudrais l'assurer, ce n'est pas trop difficile?     -> phrase du NLU
('je voudrais', 'je souhaite')                          -> sous-partie
je souhaite l'assurer, ce n'est pas trop difficile?     -> phrase reconstruite
```

Cette procédure n'est pas compliquée à implémenter mais souffre de plusieurs problèmes. Tout d'abord, elle demande beaucoup de ressource calculatoire car elle nécessite de très nombreuses comparaisons entre les 100 phrases du NLU et l'entièreté du corpus *PPDB*. Ensuite, elle génère de nombreux déchets tels que (pour le même exemple) :

```
('je voudrais', "j'aimerais que")
j'aimerais que l'assurer, ce n'est pas trop difficile?
('je voudrais', 'je voudrais que')
je voudrais que l'assurer, ce n'est pas trop difficile?
('je voudrais', 'permettez-moi')
permettez-moi l'assurer, ce n'est pas trop difficile?
```

Elle produit donc des paraphrases valides, intéressantes linguistiquement pour la diversité grâce aux ajouts de verbes et autres éléments différents, mais elle produit des déchets en grande quantité. Nous pensons donc que cette approche peut être intéressante s'il y a relecture et supervision humaine pour le choix des paraphrases [27]. Cependant, elle risque de diminuer significativement les performances en cas d'utilisation automatique. Il faudrait, pour automatiser entièrement le processus, intégrer un évaluateur robuste à la fois syntaxiquement et sémantiquement pour écarter les phrases « déchets ».

2.3.2.2 Deuxième approche

Notre deuxième approche, *find_ refine* (Annexes Code F.2), est celle par raffinement. Elle prend également en entrée la phrase courante du NLU et l'entrée courante du corpus *PPDB*. Elle fait ensuite appel à deux fonctions qui calculent respectivement si les deux énoncés comportent un nombre similaire de caractères et, si c'est le cas, s'ils comportent un nombre similaire de tokens. Si ces deux conditions sont remplies, la phrase courante de *PPDB* est évaluée par rapport à la phrase du NLU, grâce à un module qui calcule la similarité cosinus [28]. Si le

27. Le développeur peut alors simplement choisir des paraphrases pertinentes plutôt que de les créer, cela devrait donc simplifier la tâche.

28. Voir section 2.5.1

score est suffisamment élevé, la phrase est considérée comme étant valide et est alors ajoutée au NLU.

Cette approche comporte certains problèmes. En effet, le fait de ne sélectionner que des paraphrases comportant un nombre similaire de caractères et de tokens revient à restreindre le spectre linguistique. De plus, les longues phrases seront sans aucun doute pénalisées car le corpus *PPDB* ne comporte généralement que des phrases assez courtes. Il est également compliqué d'avancer que le calcul automatique de la similarité est suffisamment fiable pour ne sélectionner que des paraphrases pertinentes. Nous avons rencontré des problèmes calculatoires et logiques. Premièrement, l'utilisation de l'évaluateur (et donc le chargement d'un modèle assez lourd) ralentit considérablement le processus. Appliqué à autant de données, la perte de temps est vertigineuse. Ensuite, d'un point de vue logique, cette approche est moins pertinente car l'évaluateur ne mesure pas la similarité sémantique (du moins pas directement). Les phrases sélectionnées seraient alors rarement pertinentes. Enfin, ce type de recherche est bien trop naïf. En effet, même si le corpus est énorme, il semble très incertain de trouver des phrases pertinentes sur le domaine, très spécifique, de la vente de voitures. Nous renseignons donc cette approche mais ne l'utiliserons pas lors de l'expérimentation finale.

2.3.2.3 Troisième approche

Cette dernière approche, *find_words_list* (Annexes Code F.2), a pour objectif d'effectuer une première phase de sélection des entrées de *PPDB* en sélectionnant celles qui comportent au moins un mot d'une liste précédemment créée. Cette liste est construite grâce à une fonction *create_list* (Annexes Code F.2). Celle-ci prend en argument le nom du fichier NLU, le lit et, après une analyse des étiquettes morphosyntaxiques, sélectionne tous les mots qui sont soit des verbes, des noms ou des adjectifs. Ces derniers sont en effet chargés sémantiquement et sont donc plus susceptibles de donner lieu à une sélection plus pertinente au sein du corpus.

La liste de mots créée, la fonction *find_words_list* (Annexes Code F.2) vérifie si un mot de la liste est présent dans l'entrée de *PPDB*. Si c'est le cas, nous vérifions si le premier élément de l'entrée est compris dans la phrase originale. Si oui, nous remplaçons l'élément par son équivalent [29].

29. Une petit précision s'impose sans doute. Nous ne vérifions pas si le second élément de la ligne de *PPDB* est également présent dans la phrase courante pour la simple raison que *PPDB* dispose également d'une entrée pour lui.

```
Exemple fictif:
liste de mots = ["manger", "nourriture", "pomme", "poire"]
string = "je veux manger une pomme"

                    élément 1               élement2
current row = ["manger une pomme", "dévorer une pomme verte"]

-> Nous pouvons constater que les éléments de row contiennent au moins
   un mot de la liste. Nous regardons donc s'il est possible d'intervertir
   des éléments. C'est le cas:

resultat = "je veux dévorer une pomme verte"
```

Cette technique permet de réduire le nombre d'éléments à intervertir grâce à une première sélection opérée grâce à une liste de mots définie selon la *baseline* NLU. Il est donc possible de postuler que les transformations seront plus pertinentes que les autres approches car faisant partie d'un même domaine sémantique. Cependant, cette approche comporte plusieurs défauts. Le premier est qu'il n'est pas possible, si le fichier NLU est limité, d'avoir accès à des paraphrases pertinentes contenant des mots alors inconnus de la liste établie. Le second défaut intervient dans la mesure où certaines transformations n'auraient pas lieu. En effet, nous ne prenons en compte que les noms, les adjectifs et les verbes à l'infinitif pour la construction de la liste. Nous ne nous soucions pas des transformations des verbes conjugués :

```
Je voudrais -> je souhaiterais, j'aimerais,...
```

Les paraphrases courtes de ce type sont pourtant légion dans *PPDB*. Bien que limitée, cette approche peut tout de même être intéressante dans la mesure où l'on s'intéresse davantage à la recherche d'équivalence sémantique de sous-parties de l'énoncé original. Une génération de phrases structurellement identiques mais utilisant des syntagmes différents peut alors prendre place.

2.3.2.4 Limites

Ces trois approches ont toutes pour limite l'aspect calculatoire, la seconde particulièrement. Le corpus est en effet conséquent et effectuer ce type de comparaisons avec une *baseline* de 100 énoncés est très gourmand. La recherche des informations dans la base de données n'est pas aussi rapide qu'une recherche dans une structure de données stockée en mémoire vive. Il serait sans aucun doute profitable d'utiliser ces techniques de cette façon.

2.3.2.5 Synthèse

Nous exploitons le corpus *PPDB* de trois manières : une recherche de syntagmes d'un énoncé dans le corpus afin de générer une paraphrase en remplaçant le syntagme par son équivalent ; une approche par raffinements successifs grâce à des mesures de similarité ; une recherche de paraphrases depuis une liste de mots permettant de transformer un syntagme de l'énoncé par l'équivalent du corpus.

Ayant écarté la seconde implémentation, nous espérons que la première ou la troisième permette d'améliorer les résultats du classificateur d'*intents*. Chacune de ces techniques a ses défauts et limites mais il reste à démontrer leur efficacité lors de notre expérimentation.

2.4 Génération depuis ReSyf

Notre seconde implémentation consiste en la génération de paraphrases grâce au remplacement lexical depuis le dictionnaire de synonymes *ReSyf*. Cette approche par thesaurus, méthode classique dans le domaine de la génération de paraphrases, a pour but de générer des phrases équivalentes afin d'apporter de la diversité, principalement lexicale, aux données d'entrainement du chatbot. Cette section décrira d'abord le fonctionnement global du système avant de s'attarder plus précisément sur chacune de ses facettes. Nous discuterons finalement des limites de cette approche.

2.4.1 Fonctionnement

Le schéma suivant résume le fonctionnement global du système de génération de paraphrases depuis *ReSyf*. Après lecture et stockage du fichier dans une structure de données, nous recherchons les synonymes de la phrase courante et construisons des phrases similaires depuis ces listes de mots. Nous évaluons ensuite automatiquement le score de similarité que ces paraphrases entretiennent avec la phrase d'origine et écrivons les meilleures dans la base de données.

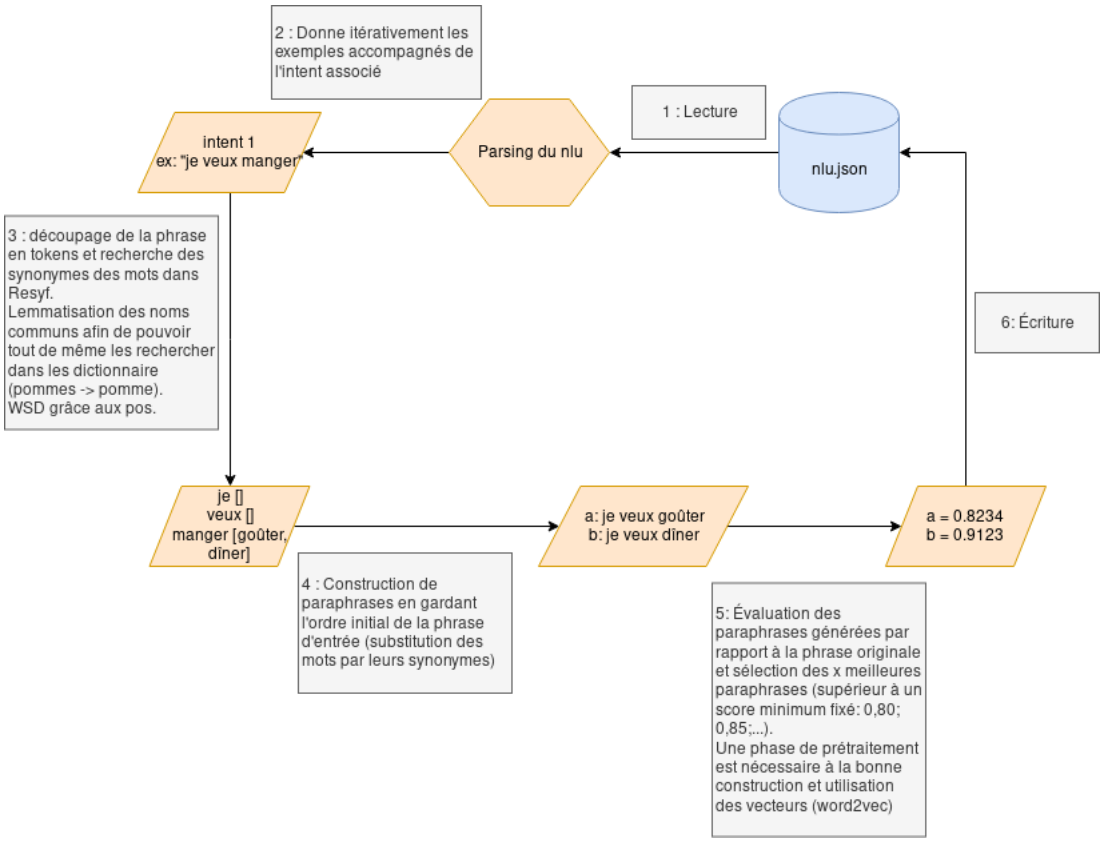


FIGURE 2.2 – Pipeline de la génération depuis ReSyf

2.4.1.1 MAIN.py

Ce programme (Annexes Code F.5) permet l'exécution de l'ensemble du processus de génération de paraphrases. Il faut lui donner en argument le nom du fichier NLU en format *JSON*.

```
python MAIN.py nlu.json
```

Lecture du NLU La première étape est évidemment de parser le fichier NLU et de stocker ces données dans une structure de données : un dictionnaire de listes. À chaque clef (l'*intent*) du dictionnaire correspond une liste de phrases (les exemples).

```
Dictionnaire['conso'] = ['Consommation', 'quelle est sa consommation',...]
```

Cette étape exclut les phrases de moins de 2 tokens pour la raison suivante : les exemples composés d'un seul mot sont sujets à un mauvais remplacement synonymique, l'énoncé n'étant composé d'aucun contexte. Nous utilisons en effet, par la suite, un *tagger* morphosyntaxique qui utilise le contexte de chaque mot pour déterminer son étiquette. Nous mettons également en place un évaluateur qui calcule un score de similarité entre deux phrases, les mots isolés risquent donc de biaiser cette étape. De plus, bien souvent, ces énoncés très simples ne comportent pas beaucoup de possibilités en matière de génération de paraphrases.

Appel à la génération Le programme appelle ensuite, itérativement, le module *generateResyf.py* en lui donnant en argument l'*intent*, l'exemple courant et le nom du fichier NLU [30].

2.4.1.2 generateResyf.py

C'est dans ce module (Annexes Code F.6) que s'articulent tous les autres processus afin de générer des paraphrases depuis un énoncé. Il est possible de diviser ce processus en quatre étapes principales :

- Recherche de synonymes pour les mots de la phrase ;
- Construction de paraphrases depuis le résultat de l'étape précédente ;
- Évaluation des paraphrases générées et sélection des meilleures ;
- Sauvegarde et écriture des meilleures paraphrases dans le fichier NLU si elles n'y sont pas déjà présentes.

Les trois premières étapes sont gérées, chacune, par un module spécifique.

30. Il devra en effet être réutilisé plus tard pour l'enregistrement des résultats dans ce même fichier NLU.

2.4.1.3 getSynonymsResyf.py

Avant d'entamer la recherche des synonymes (Annexes Code F.7), nous devons charger *ReSyf* grâce à son API.

```
lexicalRes = ReSyf.load(PATH/to/the/folder/of/resyf)
```

Nous procédons à la tokenisation, grâce au module *NLTK*[31], de la phrase qui a été donnée en argument au programme. Ensuite, afin de désambigüiser la nature des mots qui devront être recherchés dans *ReSyf*, nous mettons en place un étiquetage morphosyntaxique. Ce dernier est mis en place grâce au *Stanford-POSTagger*[32].

Nous avons constaté que les étiquettes morphosyntaxiques du *tagger* et de *ReSyf* n'étaient pas identiques. Le premier utilise les étiquettes du *TreeBank*[33] tandis que le second a son propre jeu d'étiquettes. Nous avons donc écrit une fonction qui convertit les étiquettes du *tagger* pour qu'elles correspondent à celles de *ReSyf*. C'est sur base de l'énoncé tokenisé et de la liste d'étiquettes que le reste du programme s'articule.

Les deux listes ont une longueur rigoureusement identiques. Lorsqu'une étiquette n'a pas pu être décelée par le *tagger*, la valeur à cet indice est *None*. De cette façon, le programme itère sur les listes et, si la valeur de l'étiquette est *None*, il ajoute le terme à la liste des résultats. Donc, si aucune étiquette n'est trouvée pour un mot, le programme conserve sa valeur afin de pouvoir reconstruire le phrase par après mais ne recherche pas de synonymes dans *ReSyf* car il manque l'étiquette morphosyntaxique. Si une étiquette a été identifiée, le processus continue.

Si l'étiquette correspond à un adjectif ou à un nom, une étape de lemmatisation a lieu. Celle-ci permet d'obtenir des résulats lorsque la forme identifiée est fléchie. En effet, *ReSyf* ne possède que les lemmes, les formes canoniques des termes.

```
Je veux en savoir plus sur ses options
-> savoir (V) ; plus (ADV) ; options (NC)

 savoir -> [savoir, voir, connaitre, comprendre, apprendre, imaginer]
 plus -> [plus, davantage, encore, surtout,...]
 options -> option -> []

 options a été lemmatisé mais aucun synonyme n'a été trouvé
```

31. https ://www.nltk.org/
32. Source : https ://stackoverflow.com/questions/44468300/how-to-pos-tag-a-french-sentence
33. http ://www.llf.cnrs.fr/Gens/Abeille/French-Treebank-fr.php

Nous nous rendons bien compte que cette approche n'est pas fidèle aux accords de la phrase mais cela ne nous semble pas primordial. En effet, lors de la lecture de notre jeu de tests, récolté sur Facebook, nous nous sommes rendu compte qu'une part significative de ces énoncés comportaient des erreurs d'accord (du pluriel principalement mais également du féminin). Nous pensons donc que la génération de ce type de paraphrase est intéressante pour couvrir cette diversité linguistique. La recherche de synonymes est permise grâce à l'API proposée par *ReSyf* :

```
result = ReSyf.get_synonyms(lexicalRes, term, postag)
```

Cette fonction renvoie un objet contenant les résultats de la requête. Il suffit ensuite d'itérer sur les synonymes et de les ajouter à la liste de résultats s'ils n'y sont pas déjà présents. Après avoir construit une liste de synonymes pour un mot, nous la stockons dans une autre liste afin de créer une matrice.

```
Exemple fictif:
[[Je]
 [veux]
 [en]
 [savoir, voir, connaitre, comprendre, apprendre, imaginer]
 [plus, davantage, encore, surtout,...]
 [sur]
 [ses]
 [options]
]
```

Une fois cette matrice construite, elle est retournée au module *generateResyf.py* qui appelle ensuite le module *makeSentences.py* afin de reconstruire des phrases depuis ces matrices.

2.4.1.4 makeSentences.py

Ce module (Annexes Code F.8) prend la matrice de synonymes en argument et, grâce à la librairie *itertools*, reconstruit des phrases [34]. Une fonction auxiliaire, *clean*, a été écrite pour nettoyer les résultats, des parenthèses s'ajoutaient en effet dans la phrase lors du processus de construction.

```
Je veux en savoir plus sur ses options
Je veux en voir plus sur ses options
Je veux en connaitre plus sur ses options
Je veux en savoir davantage sur ses options
```

34. Source qui a aidé à la rédaction du code : https ://stackoverflow.com/questions/48819566/generate-all-possible-sentences-from-n-lists-using-python

```
Je veux en comprendre plus sur ses options
Je veux en apprendre plus sur ses options
Je veux en imaginer encore sur ses options
Je veux en imaginer surtout sur ses options
```

Ce programme génère beaucoup de paraphrases. En effet, au plus il y a de synonymes, au plus il y aura de combinaisons possibles. C'est pour cette raison, et pour éviter un *overfitting* dans les données du NLU, que nous avons implémenté un évaluateur. Celui-ci a pour tâche de sélectionner les paraphrases les plus pertinentes. En effet, certains synonymes peuvent, dans le contexte, posséder un sens incongru ou inadapté. Nous avons donc essayé de minimiser ce phénomène.

2.4.1.5 evaluate.py

Nous avons d'abord procédé au calcul de la similarité entre phrases avec des mesures telles que Levenschtein ou BLEU. Cependant, les scores de similarité reflétaient assez mal l'équivalence sémantique que nous recherchions. Nous avons donc décidé d'écarter ces mesures basées principalement sur des critères formels et de calculer ce score grâce à des vecteurs préentrainés.

Afin de déterminer si les paraphrases générées sont pertinentes, nous avons décidé de créer un module (Annexes Code F.9) procédant à leur évaluation. Nous avons estimé que la librairie *gensim*[35] était une solution abordable et efficace. Pour que cette méthode fonctionne, il est nécessaire de charger un modèle de *words embeddings* préentrainés (*word2vec*) pour procéder à l'analyse des paraphrases.

Word2Vec La représentation de mots en vecteurs numériques donne des propriétés très intéressantes. Ainsi est-il possible de déterminer, grâce au calcul de la similarité cosinus, la proximité de deux mots dans l'espace vectoriel. De cette manière, la recherche de mots associés à « Sweden » donnera pour résultat une liste de pays avec en tête de liste « Norway » disposant d'un score de 0.76 [36].

> « Avec suffisamment de données, d'usages et de contextes, *word2vec* peut faire des suppositions très précises sur la signification d'un mot à partir de ses occurrences passées. Ces suppositions peuvent être utilisées pour établir l'association d'un mot avec d'autres mots (par exemple, « homme » est à « garçon » ce que « femme » est à « fille »), ou regrouper des documents et les classer par sujet. Ces ensembles peuvent servir de base à l'analyse des sentiments et aux recommandations dans des domaines aussi divers que la recherche scientifique,

35. https ://radimrehurek.com/gensim/index.html
36. https ://skymind.ai/wiki/word2vec

> la découverte juridique, le commerce électronique et la gestion des relations avec la clientèle » [37].

Dans notre cas, nous utilisons cette ressource pour déterminer la similarité, c'est-à-dire la proximité [38] dans le plan vectoriel, d'une phrase par rapport aux paraphrases générées. Nous utilisons un modèle particulier qui a été créé d'après le corpus « French Wikipedia XML » [39].

> « La taille de cet ensemble de données est d'environ 500 millions de mots ou 3,6 Go de texte brut. Les principaux paramètres de construction du modèle étaient les suivants : aucune lemmatisation, la tokenisation a été effectuée à l'aide de l'expression régulière « \W » (tout caractère qui n'est pas un mot divise la phrase en tokens) et le modèle a été construit avec 500 dimensions » [40].

Afin de pouvoir calculer ce score pour des phrases entières et non plus des mots isolés, nous avons utilisé la fonction *Word2Vec.n_similarity* de *gensim* [41]. Celle-ci calcule la similarité cosinus entre des ensembles de mots [42] :

```
Phrase : Donne-moi les caractéristiques principales du véhicule : 1


1 : Donne-moi les caractéristiques essentiel du véhicule  : 0.828
2 : Donne-moi les caractéristiques notable du véhicule  : 0.801
3 : Donne-moi les caractéristiques principales de cet abricot : 0.766
4 : Quelles sont les caractéristiques importantes de cette auto? : 0.674
5 : Donne-moi les caractéristiques de cet avion : 0.647
6 : Quelles informations importantes peux-tu me donner sur cette voiture? : 0.518
7 : Donne-moi des informations sur ce fruit : 0.329
8 : Donne-moi une pomme : 0.0344
```

Nous pouvons constater que cette mesure a ses faiblesses et qu'elle a tendance à se tromper. De fait, elle ne prend pas en compte le sens très différent de la troisième paraphrase. Elle a également des difficultés à classer correctement

37. https ://skymind.ai/wiki/word2vec : Given enough data, usage and contexts, *word2vec* can make highly accurate guesses about a word's meaning based on past appearances. Those guesses can be used to establish a word's association with other words (e.g. "man" is to "boy" what "woman" is to "girl"), or cluster documents and classify them by topic. Those clusters can form the basis of search, sentiment analysis and recommendations in such diverse fields as scientific research, legal discovery, e-commerce and customer relationship management.

38. Nous n'entendons pas par proximité une distance euclidienne minimale mais bien un faible cosinus d'angle. Voir section 2.5.1.1.

39. https ://zenodo.org/record/162792#.XS_BkxgyWuU

40. Ibid. : The size of that dataset is about 500 million words or 3.6 GB of plain text. The principal parameters for building the model were the following : no lemmatization was performed, tokenization was done using the "\W" regular expression (any non-word character splits tokens), and the model was built with 500 dimensions.

41. https ://tedboy.github.io/nlps/generated/generated/gensim.models.Word2Vec.n_similarity.html

42. Les fautes de langue dans ces exemples sont voulues. Les paraphrases générées ont en effet tendance à en comporter certaines.

la sixième paraphrase qui devrait obtenir un score bien plus élevé. Cependant, nous avons décidé de l'implémenter afin de déterminer son efficacité lors de notre expérimentation.

C'est lors de l'implémentation que nous avons rencontré un problème calculatoire.

RAM Notre ordinateur dispose de 4Go de *RAM*. L'utilisation la plus courante de *word2vec* nécessite de charger le modèle en mémoire vive. C'est lors de cette étape que nous avons rencontré inévitablement une *memory error*. En effet, nous ne disposions pas de suffisamment de *RAM*. La solution a été trouvée dans l'ajout de la fonctionnalité *mmap* lors du chargement du modèle.

```
w2v = models.Word2Vec.load("../trained_models/frwiki.gensim",mmap='r')
```

Mmap Cette fonction est très utile lorsque les ressources calculatoires sont limitées.

> « [mmap] implémente la pagination à la demande, car le contenu des fichiers n'est pas lu directement à partir du disque et n'utilise initialement pas du tout de *RAM* physique » [43].
> « Le mappage à l'aide de fichiers mappe une zone de la mémoire virtuelle du processus avec des fichiers, c'est-à-dire que la lecture de ces zones de mémoire entraine la lecture du fichier. C'est le type de mappage par défaut [44]. »

L'utilisation de cette fonctionnalité permet donc de stocker le modèle dans des fichiers sur le disque. La *RAM* n'est donc pas occupée par le modèle et, bien que plus lent, le processus s'exécute sans aucune erreur. Cette fonction est donc particulièrement utile lorsque les performances ne sont pas une priorité et que les ressources calculatoires sont limitées.

preprocess.py Afin de pouvoir être évaluées, la phrase ainsi que les paraphrases doivent obligatoirement passer par une étape de prétraitement (Annexes Code F.10). En effet, la construction et l'utilisation des vecteurs demandent la suppression de la ponctuation, le retrait des mots fonctionnels (pouvant biaiser le calcul de similarité), la mise en bas de casse et une tokenisation. Nous avons donc écrit un module qui gère ce prétraitement en prenant en argument une phrase et en retournant une liste de mots.

43. https ://en.wikipedia.org/wiki/Mmap : It implements demand paging, because file contents are not read from disk directly and initially do not use physical RAM at all.

44. https ://en.wikipedia.org/wiki/Mmap : File-backed mapping maps an area of the process's virtual memory to files ; i.e. reading those areas of memory causes the file to be read. It is the default mapping type.

```
"Je veux manger une pomme!"
[veux, manger, pomme]
```

Processus d'évaluation automatique Le processus se divise en plusieurs étapes :

- 1 : Chargement du modèle *word2vec* ;
- 2 : Prétraitement (supprimer les mots fonctionnels et la ponctuation, tokenisation, mise en bas de casse) de la phrase et construction du vecteur ;
- 3 : Retour d'une liste vide si aucune paraphrase n'est renseignée ou si la longueur du vecteur est de 0 ;
- 4 : Itération sur les paraphrases : prétraitement et construction du second vecteur ;
- 5 : Calcul de similarité entre le vecteur 1 (phrase originale) et le vecteur 2 (paraphrase) ;
- 6 : Stockage des paraphrases et de leur score par rapport au vecteur 1 dans un dictionnaire[score] = paraphrase ;
- 7 : Stockage dans une liste des x (maximum) paraphrases les plus pertinentes et exclusion des autres (afin de minimiser l'*overfitting* et éviter de remplir le NLU avec trop d'exemples similaires) ;
- 8 : Retourner la liste au programme *generateResyf.py* afin de l'écrire dans le fichier *JSON*.

Cette dernière étape marque la fin du processus de création de paraphrases pour un énoncé. C'est ensuite au tour de l'énoncé suivant d'être paraphrasé.

2.4.2 Limites de cette approche

Cette approche de génération basée sur un thesaurus possède plusieurs faiblesses.

Premièrement, elle ne gère ni la grammaire ni même la conjugaison. Nous nous sommes en effet principalement axé sur des remplacements lexicaux. Nous avons remarqué, dans le jeu de données récolté sur Facebook, que les messages des internautes n'étaient pas forcément grammaticaux. Certains ne possédaient même pas de verbe. Prendre en compte des modifications grammaticales aurait sans aucun doute été très compliqué à implémenter. Nous n'avons pas voulu complexifier le modèle en y ajoutant des règles de transformation. Notre hypothèse se concentre en effet sur l'efficacité, ou l'inefficacité, de cette méthode de génération, à remplacement lexical, dans le cadre de la classification d'*intents*.

Dans un deuxième temps, la recherche de synonymes est tributaire de la bonne identification des étiquettes morphosyntaxiques du *tagger*. S'il échoue à reconnaitre efficacement une étiquette, il ne sera pas possible de trouver les synonymes pertinents dans *ReSyf*. Dans certains cas, le mot concerné possède

des sens différents selon l'étiquette, une erreur peut alors mener à la génération de phrases inintelligibles.

Enfin, l'évaluation pose parfois problème dans l'attribution des scores aux paraphrases. En effet, ces dernières étant créées grâce à des synonymes, il arrive qu'une paraphrase moins pertinente reçoive un score plus élevé que d'autres plus appropriées. C'est un effet malheureusement prévisible car il n'est pas facile pour le modèle de faire correspondre son score de similarité à une évaluation humaine.

Cependant, cette approche a tout de même le mérite d'obtenir des premiers résultats de génération de paraphrases en utilisant un seul thesaurus. Nous avons choisi cette méthode car les données sont disponibles sur Internet en libre accès. De plus, dans le cadre d'un chatbot disponible sur le web, il nous semble avant tout important de lui offrir une large couverture lexicale. En effet, à l'instar du langage SMS qui résulte d'une adaptation à un média (le téléphone mobile), nous estimons pouvoir partir du postulat que les internautes parlent d'une manière spécifique aux chatbots. Nous avons en effet pu constater que certains communiquent avec des mots-clés ou avec des phrases très courtes ou nominales, ce qu'ils ne feraient en aucun cas avec un opérateur humain.

2.5 Évaluation

L'approche basée sur *ReSyf* dispose d'une phase d'évaluation de la qualité des paraphrases générées. Nous décrirons dans cette section différentes techniques et mesures permettant de procéder à ces évaluations et nous détaillerons les raisons de l'utilisation de *word2vec*. Ensuite, nous expliquerons brièvement l'importance des mesures humaines pour juger de la pertinence et de la qualité des résultats.

Bien que ce ne soit pas l'objectif de ce travail, nous estimons qu'il est intéressant de donner un rapide aperçu des différentes façons d'évaluer, globalement, un chatbot. Nous n'utiliserons pas ces mesures et cadres évaluatifs mais les renseignons tout de même, pour information.

2.5.1 Évaluation des paraphrases générées

Cette section présente les différentes techniques et mesures permettant de juger de la qualité de paraphrases générées.

2.5.1.1 Mesures calculées

Certaines mesures demandent à être calculées par le système :

— BLEU (Papineni et al., 2002) : cette mesure traditionnellement utilisée en traduction automatique utilise les concordances entre les phrases originales et les paraphrases en utilisant la pénalité de brièveté (*brevity*

penalty, ou BP) et la précision des *n-grams* (*n-grams precision*) (Gupta et al., 2018) ;

— METEOR (Banerjee and Lavie, 2005) : cette mesure, également utilisée en traduction automatique, utilise le *stemming* et les synonymes (souvent WordNet) pour ensuite calculer un score basé sur la précision d'*unigrams* (Gupta et al., 2018) ;

— TER (*Translation Error Rate*) : cette mesure est basée sur le nombre de modifications (insertions, suppressions, substitutions) requises par un humain pour convertir le résultat en une paraphrase de référence (Gupta et al., 2018).

— La précision de remplacement(rp), le taux de remplacement(rr) et la F-mesure de remplacement(rf) (Zhao et al., 2009). Estimant que BLEU ne correspond pas à leur approche (SPG), ils ont introduit une nouvelle façon d'évaluer les paraphrases selon l'idée que l'une d'entre elles devrait contenir autant d'unités correctes de remplacement qu'il est possible d'avoir ;

— La similarité cosinus, calculée grâce aux *words embeddings* de *word2vec*. Cette mesure permet de calculer la similarité de documents indépendamment de leurs tailles. Elle mesure le cosinus entre les deux vecteurs (représentant deux documents) au sein d'un espace multidimensionnel. L'avantage de cette technique est qu'elle n'est pas dépendante de la distance euclidienne entre les deux documents. À partir du moment où l'angle est faible, la similarité est forte [45].

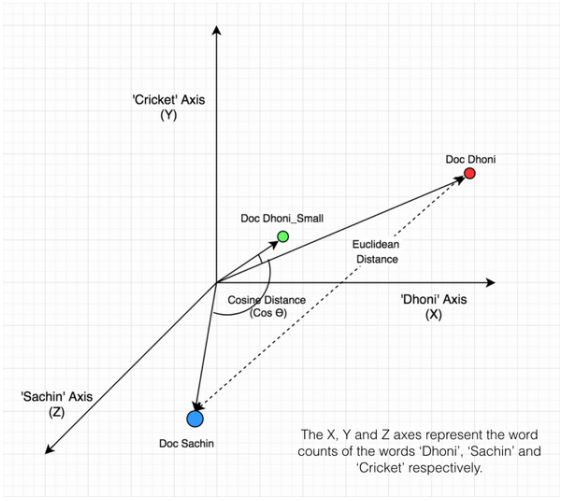


FIGURE 2.3 – Similarité cosinus (*Cosine similarity*) (Uniqtech, 2018a)

45. https ://www.machinelearningplus.com/nlp/cosine-similarity/

2.5.1.2 Mesures humaines

Une évaluation humaine est souvent nécessaire pour juger de la pertinence des résultats obtenus. Différents critères sont alors pris en compte :

- La pertinence et la lisibilité (échelles de 1 à 5, voir Gupta et al. (2018)) ;
- L'adéquation (Zhao et al., 2009) ;
- La fluence (Zhao et al., 2009) ;
- L'utilisabilité (Zhao et al., 2009) ;
- Vérification manuelle pour vérifier si les unités de remplacement sont correctes ou incorrectes (Zhao et al., 2009) ;
- Caractère naturel de l'énoncé (Nikitina et al., 2018).

Nous nous baserons sur ces mesures afin d'évaluer la qualité des paraphrases générées par nos approches.

2.5.2 Évaluation du chatbot

De nombreux travaux proposent des cadres évaluatifs ou diverses mesures pour estimer la qualité d'un chatbot. Nous ne mettrons pas en place l'une de ces évaluations dans le cadre de notre expérimentation. En effet, notre chatbot est volontairement très simple afin d'isoler et d'étudier la classification des *intents*. Cela n'aurait donc que peu de sens d'évaluer la qualité du système dans sa globalité. Cependant, nous estimons qu'un rapide tour d'horizon de ce domaine est profitable, afin de se faire une idée plus précise de ce que l'on peut attendre de ce type de système.

Walker et al. (1997) ont proposé un cadre évaluatif : PARADISE (*PARAdigm for DIalogue Evaluation*). Ce dernier utilise le coefficient *jappa*, calculé à partir d'une matrice résumant la façon dont le système a fonctionné. Il utilise également les couts de dialogue, les réponses inappropriées ou répétées et la satisfaction de l'utilisateur. En bref, la mesure de performance PARADISE est une fonction qui prend en compte le succès de la tâche (k) et le cout du dialogue (ci). Elle a, selon ses concepteurs, de nombreux avantages :

- Elle peut évaluer la performance à n'importe quel niveau, car k et ci peuvent être calculées à partir de n'importe quelle sous-tâche du dialogue ;
- k permet de mesurer le succès partiel de l'achèvement d'une tâche ;
- La performance peut combiner des mesures de cout objectives ou subjectives ;
- Elle utilise la satisfaction de l'utilisateur (ils étaient alors les premiers à la prendre en compte) ;
- Il est possible d'associer cette performance avec des stratégies individuelles de dialogue car k prend aussi en compte la complexité de la tâche. Des comparaisons peuvent donc être faites sur base de ce critère.

Bien qu'intéressant, ce cadre évaluatif est daté et ne correspond peut-être plus aux nouveaux agents conversationnels.

Kaleem et al. (2016) estiment qu'il est compliqué d'évaluer des systèmes de dialogue (Martinez et al., 2008). Il n'y a en effet pas de standard d'évaluation. De plus, certaines unités d'évaluation d'un système ne peuvent pas forcément être transposées à un autre. PARADISE ne serait pas, selon eux, adapté à des systèmes modernes et plus avancés.

Moller et al. (2009) ont, quant à eux, avancé une taxonomie mettant en avant des critères de qualité du service et de qualité de l'expérience. Les premiers seraient basés sur des critères objectifs, alors que les seconds seraient plus subjectifs (expérience d'utilisateur). Il ressort cependant de tout cela que le critère « d'utilisabilité » est plus important que ceux de « naturalité » et de « flexibilité ». La satisfaction d'utilisateur est le plus souvent évaluée grâce à des questionnaires utilisant l'échelle de Likert. Les critères objectifs pourraient quant à eux être calculés grâce aux *logs* du système. L'évaluation n'est utile selon eux que si elle permet de comprendre les processus sous-jacents. Ils ont donc défini un cadre du type *Goals Questions Metrics* :

— *Goals* : lister les buts principaux du système ;
— *Questions* : dériver des questions précises depuis les buts pour déterminer s'ils sont atteints ;
— *Metrics* : décider de ce qui doit être mesuré pour répondre aux questions. Les mesures peuvent être objectives ou subjectives.

Le GQM est un modèle hiérarchique du style *top-down*. C'est donc un cadre adaptable à chaque système qui permet de clarifier les buts recherchés et les façons de calculer les résultats.

Shawar and Atwell (2007) ont déterminé plusieurs mesures pour évaluer globalement un chatbot :

— Cout de la tâche en temps écoulé ou en tours nécessaires. Cela permet de déterminer la performance du système pour achever sa tâche (mesure partagée avec Nikitina et al., 2018) ;
— L'habileté à trouver des réponses à une entrée d'utilisateur (mesures d'efficacité du dialogue), taux d'erreur pour les reconnaissances de phrases (identification d'un *intent*) ;
— Degré de « raisonnabilité » des réponses (mesures de qualité du dialogue) : raisonnable, moyen, etc ;
— Satisfaction de l'utilisateur.

Ils concluent leur travail en insistant sur le fait qu'il ne faut pas adopter une méthodologie d'évaluation selon un standard établi, mais plutôt s'adapter à l'application et aux besoins d'utilisateur.

Muischnek and Müürisep (2018) ont quant à eux créé 5 étiquettes (échelle de Lickert) permettant d'évaluer la réponse du chatbot ; ils se sont inspirés des

travaux de Shawar and Atwell (2007) :

— Approprié ;
— Partiellement approprié ;
— Moyen ;
— Partiellement inapproprié ;
— Inapproprié.

Ils évaluent la mise en relation entre la question d'utilisateur et l'*intent* trouvé, cela en considérant dans ce cas qu'il peut y avoir des *intents* très proches sémantiquement. Ils mettent aussi en avant qu'un *challenge* actuel est de pouvoir prendre en charge les erreurs grammaticales de l'entrée de l'utilisateur : erreurs orthographiques, signes diacritiques manquants, etc.

Radziwill and Benton (2017) ont compilé des articles de 1990 à 2017 (32 papiers et 10 articles) traitant de ce sujet afin de générer des matrices comportant les attributs et les caractéristiques d'évaluation d'un chatbot. Ils ont pu constater que les attributs se regroupaient selon les normes ISO 9241 :

— Efficacité (*effectiveness*) ;
— Bon fonctionnement (*efficiency*) ;
— Satisfaction.

Les matrices disponibles dans cet article sont globales et très générales. Elles constituent dès lors une excellente base sur laquelle il est possible de travailler pour établir sa propre matrice de critères d'évaluation selon ses besoins.

Braun et al. (2017) ont quant à eux étudié spécifiquement les performances des modèles NLU de plusieurs services tels que : LUIS [46], Rasa, Watson [47] et Dialogflow. Ils ont concentré leurs efforts autour de la classification des *intents* et la recherche des entités. Pour évaluer les performances des différents systèmes, ils ont calculé le nombre de *true positives* , de *false positives*, de *false negatives* et de *true negatives* pour ensuite pouvoir calculer la précision, le rappel (*recall*) et la F-mesure. Ils en déduisent ensuite le meilleur système pour la tâche évaluée. Une telle méthode est simple à mettre en œuvre mais possède, selon eux, plusieurs limitations :

— Les performances ne seront pas égales selon les domaines ;
— La conclusion ne peut pas être générale, il n'est pas possible de dire qu'un système est toujours meilleur qu'un autre.

2.5.2.1 Synthèse des mesures évaluatives

Cette section décrit, en premier lieu, les différentes mesures et techniques, automatiques ou manuelles, permettant de juger de la qualité des paraphrases générées. C'est la similarité cosinus, calculée grâce à *word2vec*, qui a retenu notre

46. https ://www.luis.ai/home
47. https ://www.ibm.com/watson/how-to-build-a-chatbot

attention pour l'implémentation. Les mesures manuelles, quant à elles, nous seront utiles pour procéder à une analyse linguistique des résultats obtenus.

Par contre, nous ne nous soucierons pas des mesures évaluatives globales. En effet, nous ne jugerons pas de la qualité de notre chatbot, ce dernier étant originellement simple. Ce bref panorama des techniques évaluatives est présent à titre informatif, afin de décrire les différentes techniques utilisées dans ce domaine.

En ce qui concerne l'évaluation globale des performances de nos différentes approches, nous utiliserons un cadre évaluatif préexistant et incorporé dans Rasa. Une fonctionnalité très pratique consiste à exécuter la commande suivante :

```
rasa test nlu -u TEST-FILE --model models/NAME-OF-THE-MODEL
```

Rasa a en effet intégré un évaluateur automatique qui prend comme argument un jeu de tests (TEST-FILE) [48] afin d'évaluer les performances du modèle [49]. Différents résultats ressortent de ce processus :

- Une *confusion matrix* qui permet de visualiser les collisions entre les différents *intents* et de se rendre compte des problèmes de classification [50] ;
- Un histogramme de la distribution des *intents* selon leurs probabilités de prédiction ;
- Les résultats des *precision, recall, accuracy* et F-mesure pour la classification des *intents* et l'extraction des entités. Nous ne nous intéresserons qu'aux *intents*. Ces mesures permettent de comparer aisément les performances des différents modèles, et donc de leur modèle NLU associé ;
- Un fichier d'erreurs est créé, il permet de voir quels énoncés ne sont pas bien classés par le modèle.

2.6 Synthèse

Nous avons détaillé, dans ce second chapitre, la méthodologie et les implémentations que nous avons mises en oeuvre pour répondre à notre problématique qui vise à évaluer les effets d'une génération automatique de paraphrases dans le cadre de la classification des *intents* dans Rasa.

Le premier volet de ce chapitre consistait à expliquer de façon détaillée le fonctionnement de Rasa, son architecture jusqu'au processus de création d'un chatbot. Nous avons également mis en avant les deux classificateurs d'*intents* de Rasa et avons décrit leurs limites et avantages respectifs.

48. Le jeu de données récolté sur Facebook.

49. Entrainé avec les différents NLU. À un fichier NLU correspondra un modèle bien précis qui sera ensuite évalué.

50. Elle permet en effet de voir combien de phrases appartenant à l'*intent* A sont classées sous B ou C.

Nous avons ensuite présenté les différentes variables qui interviendront dans la construction des résultats ainsi que les diverses données qui prendront place dans cette expérimentation.

Par la suite, nous avons détaillé le fonctionnement de notre première approche basée sur le corpus *PPDB*. Celle-ci vise à : rechercher des syntagmes d'un énoncé dans le corpus afin de générer une paraphrase en remplaçant le syntagme par son équivalent ; adopter une approche par raffinements successifs grâce à des mesures de similarité ; rechercher des paraphrases depuis une liste de mots permettant de transformer un syntagme de l'énoncé par l'équivalent du corpus.

Enfin, nous avons exposé notre seconde approche qui vise à générer des paraphrases grâce à un dictionnaire de synonymes : *ReSyf*.

Les limites de nos approches ont été décrites mais il reste à évaluer empiriquement leur efficacité ou inefficacité lors d'une expérimentation.

Nous avons dès lors décrit les différentes techniques et mesures permettant de mener à bien deux types d'évaluation : juger de la qualité des paraphrases générées et décider de la qualité globale du chatbot. Bien qu'ayant décrit les deux domaines, nous ne nous concentrerons que sur les premières mesures évaluatives, automatiques et manuelles, permettant de juger de la qualité des phrases générées. Finalement, nous avons exposé le cadre évaluatif intégré dans Rasa. Nous utiliserons celui-ci afin de déterminer si la surgénération de paraphrases, selon les différentes techniques mises en oeuvre, influence positivement ou négativement les résultats globaux de la classification des *intents*.

Chapitre 3

Résultats

Ce chapitre présentera les résultats de nos approches visant à infirmer ou affirmer l'hypothèse selon laquelle la génération automatique d'énoncés similaires améliorerait la classification des *intents* d'un chatbot créé avec Rasa.

Il nous est donc nécessaire de créer un cadre évaluatif pour déterminer l'incidence de notre méthode sur la classification des *intents*. Pour ce faire, il convient d'identifier les éléments que nous ferons varier ainsi que les données que nous utiliserons. Nous agirons sur plusieurs leviers afin de tester nos différentes approches dans plusieurs configurations. Nous déterminerons ensuite l'impact, positif ou négatif, de notre expérimentation sur les résultats. Une discussion suivra afin d'apporter une explication aux scores obtenus par les méthodes mises en place.

3.1 Leviers évaluatifs

Nous présenterons dans cette section les différentes variables qui nous permettront de générer des jeux de données différents. Nous ferons varier le classificateur, la méthode de génération, le score minimum demandé par l'évaluateur, le nombre de paraphrases sélectionnées et le prétraitement du corpus de paraphrases.

3.1.1 Rasa

Nous comparerons les performances des classificateurs de *TensorFlow* et *spacy_sklearn*. Les deux fonctionnent différemment, il semble donc intéressant de déterminer dans quelle mesure notre méthode influence leurs résultats.

3.1.2 Méthodes de génération

Nous mettrons en parallèle l'utilisation d'une recherche de paraphrases au sein du corpus *PPDB* avec une génération basée sur le dictionnaire de synonymes *ReSyf*. La première approche se divise en deux techniques : la création de paraphrases grâce à la substitution de syntagmes intraphrastiques par les paraphrases du corpus ; l'utilisation d'une liste de mots créée depuis le NLU afin de procéder aux remplacements de syntagmes chargés sémantiquement.

3.1.3 Paramètres

Nous ferons varier, pour l'approche basée sur *ReSyf*, le score minimum de similarité et le nombre de paraphrases sélectionnées. Ces paramètres prendront diverses valeurs afin de détecter les effets qu'ils peuvent avoir sur la sélection des paraphrases et, à terme, sur la classification des *intents*.

En ce qui concerne l'approche utilisant le corpus *PPDB*, nous sommes originellement parti du principe qu'il était nécessaire de normaliser le format des énoncés. Le corpus était en effet tokenisé et disposait d'espaces supplémentaires. Cependant, nous pensons qu'il peut être intéressant, pour la construction des vecteurs, d'utiliser le corpus *PPDB* originel. Nous allons donc créer une seconde base de données SQLite (*ppdb-2.sqlite*) sans utiliser le prétraitement qui modifiait le format. Ainsi pourrons-nous déterminer les différences entre les paraphrases générées par ces deux corpus.

3.2 Paraphrases générées

3.2.1 Depuis ReSyf

La génération de paraphrases depuis *ReSyf*, sur le jeu de données de 10 *intents* et 100 exemples, s'effectue en 20 minutes (Intel I5 2410). Nous avons généré 5 bases de données NLU en faisant varier différents paramètres : le score de similarité de l'évaluateur ainsi que le nombre maximum de paraphrases sélectionnées par exemple. Il est possible de déterminer dans quelle mesure ces paramètres ont permis l'ajout de données. En effet, la taille du fichier de la *baseline* est de 9,3 Kio.

Nom	Similarité	Paraphrases	Taille (Kio)
nlu_0	sim $>$ 0.80	15	41
nlu_1	sim $>$ 0.85	10	28,5
nlu_2	sim $>$ 0.85	5	19,2
nlu_3	sim $>$ 0.90	5	16,3
nlu_4	sim $>$ 0.95	5	10

Les différences de taille sont dues aux différents paramètres. En effet, au plus le score de similarité minimum est bas, au plus le nombre de paraphrases sélectionnées est élevé. De plus, définir un plus grand nombre de paraphrases sélectionnées gonfle inévitablement la taille du fichier final. Nous avons décidé d'essayer 5 paramétrages différents. Du plus tolérant, *nlu_0.json*, au plus stricte, *nlu_4.json*. Nous avons privilégié l'ajout de maximum 5 paraphrases par exemple. Nous estimons ainsi limiter tout effet d'*overfitting*.

3.2.1.1 Qualité des paraphrases

A priori, les résultats les plus pertinents doivent être contenus dans les fichiers créés par les paramétrages les plus strictes.

Nous pouvons constater que le fichier *nlu_4.json*, trop sévère dans ses sélections, n'a incorporé que 4 énoncés. Ce paramétrage n'est pas pertinent étant donné qu'il rejette de trop nombreuses paraphrases. De plus, les énoncés acceptés ne diffèrent que légèrement des phrases originelles, l'intérêt linguistique est donc faible. C'est pour ces raisons que nous ne l'utiliserons pas lors de l'expérimentation.

La base de données *nlu_3.json*, quant à elle, a incorporé près de 60 paraphrases pour l'ensemble des *intents*. Il est possible de classer les paraphrases en plusieurs catégories que nous utiliserons par la suite.

Typologie des paraphrases générées

1. **Pertinentes**	
• "La garantie couvre quoi exactement ?"	*"La garantie couvre quoi **précisément** ?"*
• "Je veux en savoir plus sur ses options"	*"Je veux en **voir** plus sur ses options"*
• "Quel type d'assurance doit-on prévoir pour un jeune conducteur ?"	*"Quel type d'assurance doit-on **envisager** pour un jeune conducteur ?"*
• "je souhaite évaluer le prix de l'assurance"	*"je souhaite **estimer** le prix de l'assurance"*
2. **Non naturelles**	
• "Son prix va-t-il diminuer ?"	*"Son prix va-t-il **réduire** ?"*
• "Je voudrais l'assurer, ce n'est pas trop difficile ?"	*"Je voudrais l'assurer , ce n'est **point** trop difficile ?"*
• "Peux-tu me donner la consommation de la voiture pour 100km"	*"Peux-tu me **fournir** la consommation de la voiture pour 100km"*
• "je souhaite évaluer le prix de l'assurance"	*"je souhaite **quantifier** le prix de l'assurance"*
3. **Déchets**	
• "C'est un bon rapport qualité-prix ?"	*"C'est un bon **rapport sexuel** qualité-prix ?"*
• "Comment faire une bonne affaire ?"	*"Comment faire une **bonasse** affaire ?"* "*Comment faire une **louable** affaire ?"*
• "comment assurer la voiture et à quel prix ?"	*"comment **ménager** la voiture et à quel prix ?"*

La première classe regroupe les différents énoncés linguistiquement corrects et pertinents par rapport à l'*intent* associé. La deuxième contient les paraphrases qui, sans être mauvaises, semblent éloignées de ce qu'un utilisateur pourrait employer. Le caractère naturel y fait donc défaut. La dernière classe regroupe les énoncés qui ne sont ni corrects linguistiquement, ni pertinents par rapport à l'*intent*. Nous les nommons les « déchets ».

Le fichier *nlu_ 3.json*, après analyse linguistique des paraphrases générées [1], contient 33,3% de paraphrases pertinentes, 28,3% de non naturelles et 38,3% de déchets. Nous pouvons donc déjà constater que l'approche par thesaurus génère des énoncés qu'il serait nécessaire de supprimer. Cependant, il est à noter que près de 34% du produit de la génération est valide linguistiquement et pertinent par rapport à l'*intent*. Il convient de spécifier que cette technique de génération procède par remplacement synonymique, les phrases générées sont dès lors très similaires. Qui plus est, notre évaluateur a tendance à privilégier les phrases qui ne diffèrent pas de trop. Il y a donc peu de paraphrases qui contiennent

1. En nous appuyant sur des notions telles que l'utilisabilité, la pertinence, la fluence et la grammaticalité. Une évaluation manuelle par des juges humains aurait été souhaitable mais, au vu du nombre de paraphrases à annoter, cette démarche aurait été longue et couteuse. Nous avons donc pris le parti de réaliser cette étape nous-même. Le sens des remplacement synonymiques a donc été évalué en fonction du sens global de la phrase, du contexte sémantique, du caractère naturel de la production, etc.

plusieurs changements synonymiques. Nous pouvons également postuler que des modifications lexicales trop importantes peuvent augmenter les probabilités de modifier le sens ou la pertinence des énoncés.

Nous décrivons dans le tableau suivant un récapitulatif des données générées grâce à cette approche.

Nom	Nombre	Pertinentes	Non naturelles	Déchets	Total
nlu_0	291	0.25	0.22	0.53	1
nlu_1	179	0.29	0.21	0.50	1
nlu_2	90	0.344	0.233	0.422	1
nlu_3	60	0.333	0.283	0.383	1
nlu_4	/	/	/	/	/

Nous pouvons constater que plus le nombre de paraphrases est élevé, plus les déchets représentent une part importante des données générées. Cette tendance s'explique aisément. En effet, au plus le nombre de paraphrases est élevé, au plus le nombre de synonymes différents d'un même mot augmente. Cependant, bien que synonymes, ces différents mots s'inscrivent le plus souvent dans des contextes plus ou moins proches. Ainsi, utiliser tous les synonymes d'un mot revient à sélectionner également ceux qui ne sont pas pertinents dans le contexte donné. Le nombre de déchets augmente donc de cette façon. De plus, nous avons pu constater que les synonymes d'un mot fréquent dans les données NLU, « bon », génère des synonymes très éloignés du contexte tels que : « délicieux » ou « succulent ». Étant donné sa fréquence élevée dans la base de données, utiliser davantage de synonymes de ce mot revient à gonfler instantanément le nombre de déchets.

Le paramétrage générant le plus haut taux de paraphrases pertinentes revient à la base de données se situant entre les zones « strictes » et « tolérantes ». Cependant, il produit tout de même davantage de déchets que son homologue plus stricte.

3.2.1.2 Hypothèses

Après analyse des données générées, nous pouvons avancer plusieurs hypothèses. Étant donné que la méthode consiste en un remplacement synonymique, nous postulons que les modifications valides apportées aux énoncés sont limitées à des changements lexicaux. La créativité linguistique de cette méthode est donc relativement réduite. En effet, l'évaluateur tend à privilégier les phrases qui ne s'éloignent beaucoup, structurellement et formellement, de l'énoncé de départ. Il est donc possible que cette approche n'apporte pas suffisamment de diversité aux données et, au contraire, mène à une saturation des données par des exemples répétitifs et peu variés.

De plus, cette technique génère également des données moins pertinentes ou erronées. Le risque de dégrader les performances du classificateur d'*intents* est donc élevé. Nous pouvons déjà avancer que l'implémentation d'un évaluateur automatique sémantique pourrait, sinon pallier ce défaut, du moins minimiser son importance. Cette évaluation automatique ne pourrait cependant pas égaler une révision manuelle par un juge humain.

Bien que possédant des défauts importants, cette technique de génération permet de conserver la structure syntaxique et d'insuffler une diversité lexicale aux données. Le modèle voit alors sa couverture lexicale être augmentée. Reste à déterminer de quelle façon sont modifiées les performances du classificateur.

Notre expérimentation permettra de déterminer l'influence réelle de ce type de données augmentées sur la classification des *intents*.

3.2.2 Depuis PPDB

La génération de paraphrases depuis le corpus *PPDB* sur le même jeu de données est bien plus longue que la précédente approche. En effet, la fonction *find_ create* a demandé 3H30 de calculs et la seconde approche, *find_ words_ list*, a demandé, quant à elle, plus de 30 heures pour s'achever. La différence de durée entre les deux fonctions s'explique par les multiples comparaisons supplémentaires demandées par *find_ words_ list*[2].

Nom	Corpus	Taille (Kio)
nlu-create-ppdb.json	ppdb	78
nlu-create-ppdb-2.json	ppdb-2	102
nlu-wordList-ppdb.json	ppdb	89
nlu-wordList-ppdb-2.json	ppdb-2	37

Nous pouvons directement constater que ces fichiers sont plus volumineux que leurs homologues précédemment analysés. En effet, le corpus *PPDB*[3] est composé d'une multitude de phrases accompagnées de leur paraphrases. Ces énoncés peuvent n'être que des syntagmes tels que :

```
"j'adore les"    ->    "j'aime les"
```

Dès lors, au vu de la taille du corpus et du très grand nombre de remplacements intraphrastiques possibles, il semble logique que de nombreuses paraphrases soient générées. Qui plus est, nous n'avons pas utilisé le module d'évaluation pour ces approches. En effet, celui-ci, en plus d'ajouter une contrainte calculatoire élevée, ne semblait pas convenir à l'évaluation de paraphrases incorporant

2. Les comparaisons en question sont celles entre les mots des entrées de *PPDB* et les mots de la liste créée depuis le NLU.

3. ppdb.sqlite est le corpus prétraité tandis que ppdb-2.sqlite est le corpus original.

parfois des structures bien différentes de la phrase originelle. Nous craignions qu'il évalue mal les courtes phrases paraphrasées, celles-ci pourraient en effet lui sembler trop différentes de l'énoncé de départ. À nouveau, l'implémentation d'un évaluateur automatique sémantique pourrait être une solution.

3.2.2.1 Qualité des paraphrases

Nous décrirons dans cette section la qualité des paraphrases produites par les deux approches basées sur le corpus *PPDB*. Étant données les tailles importantes des fichiers produits, nous ne présenterons pas de statistiques [4]. Cependant, nous compilerons les différents types de transformation que nous avons identifiés lors de l'exploration manuelle des résultats.

Nous avons recensé les mécanismes de transformation linguistique que ces approches permettent d'effectuer. Après avoir analysé, pour chaque approche, les deux bases de données [5], nous avons estimé que les résultats d'une même approche relevaient des mêmes types de transformation [6], indépendamment du type de corpus utilisé (prétraité ou non). Nous les détaillons dans les typologies suivantes.

4. Cela nécessiterait beaucoup de temps pour évaluer manuellement la pertinence des énoncés produits.

5. Pour rappel, nous avons utilisé en parallèle le corpus *PPDB* original, *ppdb_2.sqlite*, et un autre prétraité : *ppdb.sqlite*.

6. Nous n'avons en effet pas décelé de différence majeure entre les deux fichiers de résultats, si ce n'est que l'un des deux en contient davantage.

Typologie de *find_ create*

1. **Inversions**	
• "Elle est facile à assurer ?"	*"**est-elle** facile à assurer ?"*
• "Où puis-je l'acheter pour le moins cher ?" ?	*"où **je peux** l'acheter pour le moins cher ?"*
2. **Changement de la personne**	
• "Combien avez-vous de marques ?"	*"combien **as-tu** de marques ?"*
3. **Remplacement de groupes verbaux**	
• "je tient à l'assurer, ce n'est pas trop difficile ?"	*"**Je voudrais l'assurer**, ce n'est pas trop difficile ?"*
• "c'est une voiture essence ou diesel ?"	*"**il s' agit là d'un** voiture essence ou diesel ?"*
• "est-ce une bonne offre ?"	*"**s' agit-il d'une** bonne offre ?"*
4. **Remplacement de groupes nominaux**	
• "Quel type d'assurance doit-on prévoir pour un jeune conducteur ?"	*"**quels genres de assurance** doit-on prévoir pour un jeune conducteur ?"*
• "La voiture est sous garantie après l'achat ?"	*"**le véhicule** est sous garantie après l'achat ?"*
5. **Remplacement de groupes interrogatifs**	
• "Combien de marques vendez-vous ?"	*"**quelle quantité de** marques vendez-vous ?"*
6. **Remplacement de locutions prépositives**	
• "La voiture est sous garantie après l'achat ?"	*"la voiture est sous garantie **suite à** l'achat ?"*
7. **Ajout/remplacement adverbial**	
• "Je veux des informations sur la voiture"	*"je veux **juste** des informations sur la voiture"*
• "Où puis-je l'acheter pour le moins cher ?"	*"**comment** puis-je l'acheter pour le moins cher ?"*
8. **Non naturelles**	
• "combien consomme la voiture"	*"**quelle quantité** consomme la voiture"*
• "Le moteur consomme beaucoup ?"	*"**l'élément moteur** consomme beaucoup ?"*
• "Je veux des informations sur la voiture"	*"je veux des informations **à propos la** voiture"*
• "Je veux en savoir plus sur ses options"	*"je veux **plus amples informations sur** ses options"*
9. **Déchets**	
• "comment assurer la voiture et à quel prix ?"	*"comment **faire en sorte** la voiture et à quel prix ?"*
	*"**comme ça que** assurer la voiture et à quel prix ?"*
• "Quand dois-je acheter pour qu'elle soit une bonne occasion ?"	*"**lorsqu'on dois-je** acheter pour qu'elle soit une bonne occasion ?"*
• "je souhaite évaluer le prix de l'assurance"	*"**j'adresse évaluer** le prix de l'assurance"*
• "quelle est la consommation du véhicule ?"	*"**quelle est consommateur** du véhicule ?"*
• "à un de ces 4"	*"**à a a** de ces 4"*
• "Elle consomme combien aux 100 ?"	*"elle consomme **combien d' argent** aux 100 ?"*
• "quelle est la consommation du véhicule ?"	*"**what is the** consommation du véhicule ?"*
• "Combien de marques vendez-vous ?"	*"**how many** marques vendez-vous ?"*
• "Où puis-je l'acheter pour le moins cher ?"	*"**où est-ce que je l'acheter** pour le moins cher ?"*
• "Elle consomme combien aux 100 ?"	*"**l'oratrice** consomme combien aux 100 ?"*

Nous pouvons constater que cette première typologie, qui concerne l'approche *find_ create*, dispose de nombreux mécanismes de transformation. Ces derniers permettent des modifications à la fois structurelles (inversion) ou syntagmatiques (remplacement de groupes). Des changements de personne ainsi que des ajouts adverbiaux ont également été identifiés. Autant de modifications intéressantes dans le but d'insuffler de la diversité linguistique dans le jeu de données en conservant le sens des énoncés.

Cependant, il n'est pas rare de rencontrer des paraphrases qui manquent de naturel ou qui sont des déchets. En effet, le corpus *PPDB* contient un très grand nombre d'entrées (près de 8 000 000). La probabilité d'utiliser une entrée peu pertinente pour effectuer les transformations est alors loin d'être négligeable. De plus, il est permis de se demander si la présence de certains éléments dans le corpus fait sens. Par exemple, « combien » a été transformé en « how many ». Nous pouvons en effet émettre des doutes sur la légitimité d'une traduction anglaise dans un corpus de paraphrases françaises. D'autres problèmes surgissent tels que « à » qui devient, de façon incompréhensible, « à a a » ou encore « puis-je » qui est transformé en « est-ce que ». Ce genre de transformations incongrues génère un grand nombre de déchets en plus de ceux, plus logiques, qui consistent en l'utilisation de certains syntagmes dans des contextes qui leur sont habituellement étrangers.

Nous allons ensuite nous intéresser à la qualité des paraphrases générées par la seconde approche. Nous dressons également une typologie des mécanismes mis en oeuvre pour la génération de données.

Typologie de *find_ words_ list*

1. **Changement de déterminant**	
• "La garantie est-elle bien de 12 mois ?"	*"**cette** garantie est-elle bien de 12 mois ?"*
2. **Remplacement de groupes nominaux**	
• "Le prix de l'assurance est-il élevé ?"	*"**le cout** de l'assurance est-il élevé ?"*
• "Puis-je avoir des informations sur la voiture ?"	*"puis-je avoir des informations sur **le véhicule** ?"*
• "c'est une voiture essence ou diesel ?"	*"c'est **une auto** essence ou diesel ?"*
• "Quand dois-je acheter pour qu'elle soit une bonne occasion ?"	*"quand dois-je acheter pour qu'elle soit **l'occasion idéale** ?"*
• "comment assurer la voiture et à quel prix ?"	*"comment assurer **la bagnole** et à quel prix ?"*
• "quels sont ses points faibles ?"	*"quels sont ses **faiblesses** ?"*
• "c'est une voiture essence ou diesel ?"	*"c'est **une caisse** essence ou diesel ?"*
• "Quelles sont les conditions de la garantie ?"	*"quelles sont **les modalités** de la garantie ?"*
• "quel est le carburant du véhicule ?"	*"quel est **le combustible** du véhicule ?"*
• "Dois-je me préparer à payer une assurance très chère ?"	*"dois-je me préparer à payer **une police d'assurance** très chère ?"*
3. **Remplacement de locutions prépositives**	
• "Puis-je avoir des informations sur la voiture ?"	*"puis-je avoir des informations **concernant la** voiture ?"*
• "Je veux des informations sur la voiture"	*"je veux des **renseignements concernant la** voiture"*
4. **Ajouts**	
• "Elle a quoi comme options ?"	*"elle a quoi comme options **possibles** ?"*
• "Quelles sont les conditions de la garantie ?"	*"quelles sont **les modalités et conditions** de la garantie ?"*
5. **Non naturelles**	
• "Vous vendez quoi comme voitures ?"	*"vous vendez **quoi comme des** bagnoles ?"*
• "quel est le carburant du véhicule ?"	*"**quel est combustible du** véhicule ?"*
• "Je veux des informations sur la voiture"	*"je veux des renseignements **au sujet la** voiture"*
• "Combien de litres aux 100 ?"	*"**combien de de gallons** aux 100 ?"*
6. **Déchets**	
• "Est-ce que la voiture sera sous garantie ?"	*"est-ce que **ia** voiture sera sous garantie ?"*
• "quel est le type de carburant ?"	*"quel est **le gars** de carburant ?"*
• "Le moteur consomme beaucoup ?"	*"**le force motrice** consomme beaucoup ?"*
• "quelle est la consommation du véhicule ?"	*"quelle est **la des consommateurs** du véhicule ?"*

Cette approche, pour rappel, utilise une liste de mots du NLU chargés sémantiquement afin de n'utiliser que les entrées de *PPDB* qui en contiennent au moins un. Étant donné que la liste est composée principalement de noms, d'adjectifs ou de verbes à l'infinitif, les remplacements de groupes nominaux sont légion. Nous avions pris cette décision afin de n'utiliser que des éléments très proches sémantiquement.

Nous avons également identifié la présence, marginale, de remplacement de locutions prépositives ainsi que d'ajouts d'adjectifs. La modification des déterminants apparait également comme un mécanisme secondaire de cette approche.

De nombreux déchets ou phrases manquant de naturel sont également créés. Cependant, ces derniers, contrairement à ceux générés par la précédente approche, sont le plus souvent dus à l'utilisation incongrue de syntagmes dans des contextes inappropriés. Par exemple, « type » est parfois remplacé par « gars » dans un contexte qui ne permet pas à ce synonyme d'être pertinent. Des déchets inexpliqués, devant provenir d'erreurs au sein du corpus, apparaissent également tels que « la » transformé en « ia ». Un grand nombre de déchets est également provoqué par des équivalences sémantiques douteuses. Par exemple, « consommation » est mis en relation synonymique avec « des consommateurs ». Bien que ces deux éléments partagent une relation de sens évidente, ils ne sont pas pour autant équivalents. Ce type de déchet est très présent.

3.2.2.2 Hypothèses

Après analyse des résultats de la génération et de la qualité des paraphrases créées, nous pouvons poser plusieurs hypothèses.

Nous estimons que les paraphrases générées depuis *PPDB* sont linguistiquement plus intéressantes que celles créées grâce à *ReSyf*. En effet, les mécanismes de remplacement ne se limitent pas à des transformations synonymiques mais comprennent des changements de déterminants, de personne, des modifications de groupes nominaux et verbaux, des transformations de locutions prépositives et des ajouts d'adjectifs ou d'adverbes.

Nous pouvons donc estimer que la diversité linguistique de ces bases de données y est plus importante. Cette dimension est, à notre sens, particulièrement intéressante dans le cadre de la bonne classification d'*intents*. Nous posons donc l'hypothèse selon laquelle ces données peuvent améliorer plus efficacement les performances du classificateur.

Cependant, les déchets y sont nombreux. Leur influence peut donc contrebalancer la diversité linguistique précédemment énoncée. Nous craignons donc une stagnation des performances du classificateur à cause de la qualité hétérogène des paraphrases créées. Cette inquiétude concerne également les paraphrases créées depuis *ReSyf*. En effet, la sélection de synonymes peut parfois poser problème et générer des paraphrases incongrues.

3.2.3 Synthèse sur la qualité des paraphrases générées

Nous avons présenté les résultats de la génération des approches basées sur *ReSyf* et *PPDB*.

La première procède par remplacements synonymiques et limite le nombre de paraphrases générées grâce à un évaluateur qui ne sélectionne que les meilleures (selon la similarité cosinus). La seconde exploite un grand corpus de paraphrases afin de transformer des phrases en des équivalents valides et linguistiquement variés.

Ces deux approches permettent d'insuffler, chacune à sa manière, davantage de diversité aux données. La première permet d'élargir la couverture lexicale de la base de données et donc, à terme, de permettre la bonne reconnaissance des *intents* selon des formulations lexicales diversifiées. La seconde, quant à elle, est plus intéressante car elle permet la mise en place de mécanismes variés. Ces derniers vont de l'inversion aux remplacements de groupes nominaux et verbaux en passant par l'ajout d'adjectifs ou d'adverbes et de la modification de déterminant. Cependant, ces deux approches ont leurs limites.

Nous avons constaté que ces deux techniques exposent leur faiblesse à limiter le nombre de mauvaises paraphrases (déchets, non pertinentes). En effet, la première sélectionne parfois des synonymes qui, dans le contexte de l'énoncé, ne font pas sens ou manquent de naturel. Ces paraphrases représentent une part non négligeable du produit de la génération. La seconde, quant à elle, génère différents types de déchets. Certains sont dûs au manque de pertinence contextuelle de certaines transformations. Ce phénomène était attendu. Par contre, *PPDB* contient des équivalences linguistiques critiquables ou des traductions anglaises de syntagmes français. De nombreux déchets sont alors produits suite à leur utilisation.

Après avoir été confronté à ces problèmes, nous souhaitons avancer l'idée selon laquelle un évaluateur sémantique [7] et syntaxique robuste devrait permettre de filtrer et de limiter le nombre de déchets générés. En effet, les problèmes surviennent tant au niveau syntaxique que sémantique.

Nous souhaitons également introduire la possibilité d'une approche hybride, utilisant à la fois *ReSyf* et *PPDB*. Une telle implémentation, combinée à l'utilisation d'un évaluateur robuste, devrait permettre d'exploiter les qualités des deux méthodes. Cependant, avant toute chose, nous allons les évaluer séparément lors d'une expérimentation.

Celle-ci a pour but de déterminer l'influence, positive ou négative, de la génération de paraphrases de données d'entrainement afin de classer les *intents* d'un chatbot.

7. La *soft cosine similarity* mériterait d'être implémentée afin de déterminer sa capacité à sélectionner des phrases équivalentes sémantiquement.

3.3 Effets sur la classification d'intents

Nous allons, dans cette section, présenter les résultats de notre expérimentation [8]. Dans un premier temps, nous décrirons les performances de classification d'*intents* d'un classificateur minimal (*dummy*) afin de déterminer les performances d'un mauvais classificateur. Il nous semble en effet utile de pouvoir comparer ces résultats avec ceux de la *baseline* ou des données générées. Nous présenterons ensuite les résultats de la *baseline* et des jeux de données créés par nos différentes approches. L'objectif de cette expérimentation est de déterminer les effets de ces techniques sur la classification des *intents*. Nous présenterons les scores obtenus pour les deux classificateurs que nous avons déjà présentés : *sklearn_spacy* et *TensorFlow*.

3.3.1 Dummy

Afin de pouvoir comparer les résultats de la *baseline* et des bases de données créées lors de notre expérimentation, nous avons créé un *dummy* chatbot. Nous avons donc évalué les performances du chatbot avec un jeu de données très restreint, rendant son comportement normalement imprévisible. Nous souhaitions en effet calculer les performances du classificateur pour un tel modèle. Celui-ci, fonctionnant avec une base NLU (Annexes Fig E.3) très faible (3 exemples par *intent*), nous permet de déterminer comment fonctionne Rasa avec ce type de données et à quel point la *baseline* et nos jeux de données influencent les résultats du classificateur. Les résultats sont très mauvais pour *sklearn_spacy* avec une F-mesure global de 0.10 mais un score, particulièrement bon, de 0.71 pour le classificateur de *TensorFlow* [9].

Les performances du classificateur basé sur *TensorFlow* posent question. En effet, la documentation de Rasa déconseille de l'utiliser sans disposer de larges jeux de données [10]. Pourtant, il semble beaucoup plus robuste avec seulement plusieurs mots-clés par *intent*. Nous avançons l'hypothèse selon laquelle ces performances s'expliquent par la pertinence et l'unicité des mots-clés utilisés dans le NLU ainsi que par la différenciation sémantique assez nette entre les différents *intents* et la relative facilité qui en découlerait pour désambigüiser le choix d'*intent*. Ces résultats s'expliqueraient également par la forte présence des mots-clés dans le jeu de tests, nous avons en effet utilisé des termes très généraux dans le NLU. Nous pouvons donc supposer que, contrairement à *sklearn_spacy*, ce classificateur rencontre des difficultés pour généraliser son pouvoir de classifica-

8. Pour des raisons pratiques, nous ne mettons pas les captures d'écran et les autres annexes de résultats dans ce mémoire (sauf ceux, servant d'exemple, de la *baseline*). Elles sont disponibles dans un dossier informatique, pour plus de facilité.

9. Nous ne nous attendions pas du tout à obtenir d'aussi bons résultats avec seulement 3 exemples par *intent*.

10. https ://rasa.com/docs/rasa/nlu/choosing-a-pipeline/

tion. En effet, *sklearn_ spacy*, utilisant des modèles de langue préentrainés, est capable de faire des relations entre diverses notions. Il ne semble pourtant pas être en mesure de donner de bons résultats avec uniquement des mots-clés. En effet, sa performance globale correspond à la probabilité qu'aurait chaque intent d'être tiré au sort (1 chance sur 10).

Classificateur	F-mesure moyenne
sklearn_spacy	0.10
TensorFlow	0.71

3.3.2 Baseline

La *baseline*, composée de 10 exemples pour chaque *intent*, obtient une F-mesure honorable de 0,588 pour le classificateur *sklearn_ spacy* et un score, particulièrement bon, de 0.85 avec TensorFlow. Nous renseignons les résultats détaillés et les matrices de confusion pour la *baseline* (Annexes Fig E.1, Fig E.2, Fig E.4, Fig E.5) à titre d'exemple.

Classificateur	F-mesure moyenne
sklearn_spacy	0.588
TensorFlow	0.85

L'objet de notre expérimentation consiste à déterminer si la génération automatique de données depuis nos différentes approches améliore ou dégrade ces performances. Nous veillerons ensuite à commenter et discuter les résultats finaux.

3.3.3 Résultats des données générées

Nous avons procédé à l'évaluation des résultats de nos implémentations. Le processus est relativement simple. Nous avons fait varier le classificateur [11] pour chaque base de données, produisant ainsi 2 modèles par NLU. Nous avons ensuite appliqué l'évaluation automatique de Rasa sur chacun de ces modèles grâce à cette commande :

```
rasa test nlu -u TEST-FILE --model models/NAME-OF-THE-MODEL
```

Les résultats sont constitués d'un récapitulatif des performances, d'une matrice de confusion, d'un histogramme de la distribution des *intents* selon leurs probabilités de prédiction et d'un *log* comprenant les différentes phrases ayant été classées sous un mauvais *intent*. Nous nous concentrerons sur la F-mesure globale pour chaque modèle afin de déterminer les effets des données sur la classification des *intents*.

11. En modifiant sa valeur dans le fichier de configuration.

NLU	**sklearn_spacy**	**TensorFlow**
Baseline	**0.588**	**0.85**
NLU-0-ReSyf	0.59	0.88
NLU-1-ReSyf	0.58	0.878
NLU-2-ReSyf	0.562	0.852
NLU-3-ReSyf	0.565	0.85
nlu-create-ppdb	0.588	**0.91**
nlu-create-ppdb-2	**0.618**	0.878
nlu-wordList-ppdb	0.59	0.86
nlu-wordList-ppdb-2	0.575	0.86

Nous pouvons constater que, globalement, les nouvelles données n'améliorent et ne dégradent pas les performances des classificateurs.

Les données produites par *ReSyf* semblent particulièrement inefficaces pour améliorer la classification des *intents* avec les deux classificateurs. Pour *sklearn_spacy*, les performances stagnent quand elles ne diminuent pas. Par contre, pour *TensorFlow*, la tendance est inverse. Ces résultats ne sont cependant pas concluants. Nous pensons que les simples transformations synonymiques ne sont pas suffisantes pour insuffler de la diversité linguistique pour ce type de tâche. De plus, les déchets produits ont sans doute tendance à diminuer la pertinence des données produites. Cette approche, isolée, ne semble donc pas convenir à l'amélioration automatique de la classification des *intents*.

Cependant, les résultats engendrés par la génération de paraphrases depuis *PPDB* semblent bien plus intéressants. En effet, la tendance qui s'en dégage est clairement positive. Le modèle créé grâce à la fonction *find_create* depuis le corpus *PPDB* prétraité offre des résultats très encourageants. En effet, le modèle gagne 6 points de F-mesure par rapport à notre *baseline* avec le classificateur *TensorFlow*. De plus, le modèle créé avec la même fonction sur le corpus *PPDB* original offre des résultats supérieurs à ceux de la *baseline* pour le classificateur de *sklearn_spacy*. Cette amélioration est moins importante mais semble tout de même intéressante. Nous estimons que ces résultats pourraient sans aucun doute être améliorés avec l'implémentation d'un évaluateur sémantique et syntaxique robuste. En effet, les nombreux déchets doivent certainement avoir tendance à abaisser la performance des modèles.

3.3.4 Synthèse

Ce dernier chapitre a été l'objet du processus d'évaluation. Nous y avons décrit les leviers évaluatifs avant de donner une description linguistique des paraphrases générées. Les différents problèmes émaillant ce processus ont été largement commentés. En effet, nous avons pu constater que la proportion de

déchets et de phrases peu pertinentes est importante pour les deux approches (*ReSyf* et *PPDB*).

La première met en oeuvre un remplacement synonymique pour générer des paraphrases. Cette approche est limitée par la performance de l'évaluateur automatique de sélectionner des énoncés pertinents. En effet, certains synonymes sont utilisés dans des contextes qui ne leur appartiennent habituellement pas. Étant donné que les bases de données générées ne sont pas trop grandes, nous avons pu donner un aperçu statistique de leur distribution. Nous avons ensuite dressé l'hypothèse selon laquelle cette technique serait sans doute peu concluante lors de l'expérimentation. Nous avons pu vérifier qu'elle était justifiée et nous maintenons que la seule transformation synonymique n'est pas efficace pour impacter significativement les performances d'un classificateur d'*intents*.

La seconde approche, basée sur *PPDB*, s'est avérée plus prolixe dans la génération d'énoncés. Il faut en effet souligner que le corpus de paraphrases est conséquent. Grâce à cela, de nombreux mécanismes de transformation ont été identifiés : l'inversion, le changement de personne, le remplacement de groupes verbaux et nominaux, le remplacement de groupes interrogatifs et de locutions prépositives, l'ajout ou le remplacement adverbial, le changement de déterminant et certains ajouts d'adjectifs ou de noms. Nous avons ensuite discuté des problèmes et limites de cette approche qui consistent, eux aussi, en la génération de déchets ou de phrases peu pertinentes. Leur origine est liée à certains aspects critiquables du corpus (présence de traductions, équivalences sémantiques douteuses) mais aussi à la technique de substitution qui peut remplacer des syntagmes par d'autres moins adaptés au contexte. Cependant, nous avons posé l'hypothèse selon laquelle cette démarche pouvait améliorer les performances du classificateur d'*intents* et nous avons pu la valider lors de notre expérimentation. En effet, cette méthode, grâce à la diversité linguistique qu'elle génère, nous semblait plus prometteuse.

Nous avons ensuite présenté les performances des classificateurs et avons montré les résultats obtenus sur un très petit jeu de données (*dummy*) afin de déterminer le comportement des classificateurs avec une base de données minimale. C'est alors que le classificateur basé sur *TensorFlow* nous a surpris en affichant de très bons résultats. Il lui suffisait en effet de quelques mots-clés pour désambigüiser et classer les *intents* avec une certaine précision. Nous estimons qu'il est capable de donner lieu à cette performance grâce à la pertinence des mots-clés utilisés et à leur unicité, à l'hétérogénéité sémantique des différents *intents* (pas de chevauchement sémantique) et au fait que les mots-clés utilisés, très courants, se retrouvent dans de nombreux énoncés du jeu de tests. Cependant, *TensofFlow* s'est toujours démarqué lors de nos évaluations. Nous estimons donc que la consigne donnée aux développeurs par Rasa, qui consiste à n'utiliser ce classificateur que si nous disposons de plus de 1000 données d'entrainement,

n'est pas particulièrement pertinente, du moins dans notre cas. Nous aurions même tendance à conseiller l'utilisation de ce *pipeline* pour la classification des *intents* avec Rasa.

Nous avons ensuite présenté et analysé les résultats. Ceux-ci démontrent que notre démarche peut se révéler intéressante dans le cadre de l'amélioration de la classification des *intents* d'un chatbot avec Rasa. En effet, bien que la plupart des modèles que nous avons créé n'engendre pas de réelles modifications des performances, certains se montrent suffisamment pertinents pour améliorer les résultats de façon significative. Il serait sans aucun doute intéressant de procéder à cette expérimentation avec un jeu de tests bien plus imposant et représentatif. Cette évaluation pourrait souligner l'utilité et l'efficacité de notre démarche. Dans tous les cas, l'utilisation de l'approche basée sur *PPDB* devrait être privilégiée, d'autant plus si elle est utilisée avec le classificateur d'*intents* utilisant *TensorFlow*.

Conclusion

L'objectif de notre recherche était de déterminer l'influence et la pertinence d'une génération automatique de paraphrases de données d'entrainement afin d'améliorer les performances d'un classificateur d'*intents*. Nous avions le souhait de démontrer que des approches généralistes, se basant sur un corpus et un dictionnaire de synonymes généraux, pouvaient tout de même apporter des résultats satisfaisants. En effet, les chatbots peuvent traiter des domaines très variés et spécifiques. Il est dès lors inenvisageable de trouver et d'utiliser des données spécialisées pour chacune de ces applications.

Nous avons en premier lieu théorisé chacune des facettes de notre recherche. Ainsi, nous avons défini, en profondeur, le chatbot et avons retracé son histoire. Nous avons ensuite brièvement expliqué le fonctionnement général de ce type de système. Par après, nous avons détaillé la tâche de classification des *intents* ainsi que les diverses méthodes permettant de la mettre en place. Par la suite, nous avons postulé que la surgénération de données était une piste intéressante dans le but d'améliorer les performances du classificateur. Nous avons ensuite expliqué les différentes techniques permettant de la mettre en oeuvre. Notre préférence s'est portée vers la génération automatique de paraphrases. En effet, cette méthode a le mérite de ne pas dépendre d'une supervision humaine (*crowdsourcing*) ou de la qualité de données trouvées sur Internet (acquisition depuis des forums). Nous avons ensuite détaillé les différentes techniques permettant de mettre en place ce type de génération.

Les notions théoriques bien établies, nous avons débuté l'implémentation de notre projet et avons expliqué en détail le fonctionnement de Rasa, le processus de création d'un chatbot, les données et les méthodes de classification utilisées. Les données de notre *baseline* et du jeu de tests ont également fait l'objet de descriptions détaillées. Ensuite, nous avons présenté les ressources principales que nous avons exploitées : *ReSyf* et *PPDB*. Nous avons finalement détaillé nos implémentations avant d'introduire les différentes mesures d'évaluation.

L'analyse des résultats de notre démarche a pris place dans le dernier chapitre. Nous y avons d'abord expliqué les différents leviers évaluatifs que nous allions activer. Ensuite, la qualité des paraphrases générées a été évaluée pour

les deux approches. Nous avons mis en avant les différences entre les mécanismes permis par *ReSyf* et ceux engendrés par *PPDB*. Nous avons procédé à l'évaluation finale lors d'une expérimentation en conditions réelles. Nous avons donc créé un chatbot minimal et avons produit de nombreuses bases de données, en faisant varier les différents leviers évaluatifs, afin d'avoir matière à comparer.

Les résultats, mitigés, nous ont permis de déterminer que de nombreuses configurations ne produisent aucune différence significative par rapport à la *baseline*. Cependant, certains modèles ont démontré qu'une amélioration des performances était possible pour notre approche basée sur *PPDB*. Ces résultats, malgré les limites de nos méthodes, sont encourageants. Un intérêt supplémentaire de ce travail réside également dans l'exploitation de ressources francophones. En effet, les recherches entreprises dans ce domaine concernent bien souvent l'anglais, il nous semblait donc intéressant de nous orienter vers le français. Ceci étant dit, il s'avère que la méthode basée sur *PPDB* pourrait s'appliquer aux autres langues prises en charge par ce corpus.

Les limites posées par nos approches ont sans doute contribué à la stagnation générale des performances. En effet, la génération de bruit, de déchets et de phrases peu pertinentes intervient dans une proportion non négligeable lors des processus de création de nouvelles données. Ces déchets, ajoutés automatiquement, dégradent la qualité du modèle. Nous estimons que minimiser ce phénomène devrait permettre d'asseoir les résultats positifs que nous avons décelés et de contribuer à l'augmentation de la stabilité des modèles. En effet, des comportements inattendus (reconnaissances incongrues d'*intents*) pourraient survenir à cause de la pertinence hétérogène des paraphrases.

Nous souhaitons introduire plusieurs idées d'amélioration. Premièrement, nous pensons que le calcul de la *soft cosine similarity* devrait donner des résultats intéressants. Cette mesure semble en effet prometteuse pour déterminer la similarité sémantique entre deux phrases [12]. La similarité cosinus que nous utilisons ne comprend pas cette dimension sémantique.

En second lieu, l'utilisation d'un corpus syntaxique de *PPDB* devrait permettre d'insuffler davantage de diversité structurale dans les données. Nous pensons que l'utiliser conjointement à celui employé dans cette recherche devrait permettre de générer des résultats intéressants. Nous pensons également qu'une approche hybride, utilisant à la fois *PPDB* et *ReSyf*, devrait permettre de profiter des avantages des deux approches. Dans tous les cas, nous privilégierions l'utilisation des plus petits corpus de *PPDB*. En effet, ceux-ci contiennent déjà des équivalences sémantiques critiquables, les plus gros corpus doivent en contenir davantage. Nous souhaitons en effet maximiser la probabilité de générer une paraphrase valide.

Ensuite, l'implémentation d'un évaluateur robuste à la fois sémantiquement

12. https ://www.machinelearningplus.com/nlp/gensim-tutorial/

et syntaxiquement devrait permettre de filtrer plus efficacement les données pour en exclure le bruit. L'utilisation de la *soft cosine similarity* est une première piste que nous envisagerions.

Enfin, il serait sans doute intéressant d'utiliser *WordNet*[13] ou *JeuxdeMots*[14] pour intégrer les notions d'hyperonymie et d'hyponymie afin de sélectionner les synonymes les plus pertinents de *ReSyf*. Nous estimons également que l'adoption d'une optique différente par rapport à cette problématique pourrait laisser entrevoir de nouveaux champs de recherche. En effet, nous avons pu constater lors de la collecte d'énoncés que les internautes adoptent, le plus souvent, un même niveau de langue. Dès lors, peut-être serait-il judicieux de procéder à une enquête linguistique poussée pour valider cette hypothèse et construire un générateur qui prenne cet aspect en compte.

Cette recherche partait du postulat qu'une base de données plus vaste et plus diversifiée devrait permettre d'améliorer la classification des *intents* d'un chatbot. Nous avons démontré que cette approche était valide méthodologiquement et qu'elle pouvait apporter une amélioration substantielle aux performances. Ces résultats sont d'autant plus intéressants qu'ils n'ont pas nécessité de révision humaine pour la sélection des paraphrases. Nous pensons cependant qu'il reste à implémenter un système permettant de filtrer efficacement les énoncés générés afin de construire des modèles stables et robustes. Réduire la production de bruit lors de la génération pourrait, à terme, faire de cette méthode un outil précieux pour la création de bases de données de chatbots.

13. La ressource WOLF est l'équivalent francophone de WordNet : http ://alpage.inria.fr/ sagot/

14. http ://www.jeuxdemots.org/jdm-accueil.php

Bibliographie

Abu Shawar, B. and Atwell, E. (2003). Machine learning from dialogue corpora to generate chatbots. *Expert Update journal*, 6(3) :25–29.

Altosaar, J. (2019). Tutorial - what is a variational autoencoder ? *En ligne https ://jaan.io/what-is-variational-autoencoder-vae-tutorial/.*

Athreya, R. G., Ngonga Ngomo, A.-C., and Usbeck, R. (2018). Enhancing community interactions with data-driven chatbots–the dbpedia chatbot. In *Companion Proceedings of the The Web Conference 2018*, pages 143–146. International World Wide Web Conferences Steering Committee.

Austin, J. L. (1975). *How to do things with words*. Oxford university press.

Babkin, P., Chowdhury, M. F. M., Gliozzo, A., Hirzel, M., and Shinnar, A. (2017). Bootstrapping chatbots for novel domains. In *Workshop at NIPS on Learning with Limited Labeled Data (LLD)*.

Banerjee, S. (2018). An introduction to recurrent neural networks. En ligne https ://medium.com/explore-artificial- intelligence/an-introduction-to-recurrent-neural-networks-72c97bf0912.

Banerjee, S. and Lavie, A. (2005). Meteor : An automatic metric for mt evaluation with improved correlation with human judgments. In *Proceedings of the acl workshop on intrinsic and extrinsic evaluation measures for machine translation and/or summarization*, pages 65–72.

Bapat, R., Kucherbaev, P., and Bozzon, A. (2018). Effective crowdsourced generation of training data for chatbots natural language understanding. In *International Conference on Web Engineering*, pages 114–128. Springer.

Barzilay, R. and Lee, L. (2003). Learning to paraphrase : an unsupervised approach using multiple-sequence alignment. In *Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology-Volume 1*, pages 16–23. Association for Computational Linguistics.

Bhagat, R. and Hovy, E. (2013). What is a paraphrase ? *Computational Linguistics*, 39(3) :463–472.

Bhargava, A., Celikyilmaz, A., Hakkani-Tür, D., and Sarikaya, R. (2013). Easy contextual intent prediction and slot detection. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 8337–8341. IEEE.

Billami, M. B., François, T., and Gala, N. (2018). Resyf : a french lexicon with ranked synonyms. In *Proceedings of the 27th International Conference on Computational Linguistics*, pages 2570–2581.

Bocklisch, T., Faulkner, J., Pawlowski, N., and Nichol, A. (2017). Rasa : Open source language understanding and dialogue management. *arXiv preprint arXiv :1712.05181*.

Bolshakov, I. A. and Gelbukh, A. (2004). Synonymous paraphrasing using wordnet and internet. In *International Conference on Application of Natural Language to Information Systems*, pages 312–323. Springer.

Brandtzaeg, P. B. and Følstad, A. (2017). Why people use chatbots. In *International Conference on Internet Science*, pages 377–392. Springer.

Braun, D., Hernandez-Mendez, A., Matthes, F., and Langen, M. (2017). Evaluating natural language understanding services for conversational question answering systems. In *Proceedings of the 18th Annual SIGdial Meeting on Discourse and Dialogue*, pages 174–185.

Bressler, D. (2018). Building a convolutional neural network for natural language processing ? *En ligne https ://towardsdatascience.com/how-to-build-a-gated- convolutional-neural-network-gcnn-for-natural-language-processing-nlp-5ba3ee730bfb*.

Chevelu, J., Lavergne, T., Lepage, Y., and Moudenc, T. (2009). Introduction of a new paraphrase generation tool based on monte-carlo sampling. In *Proceedings of the ACL-IJCNLP 2009 Conference Short Papers*, pages 249–252. Association for Computational Linguistics.

Chinchor, N. and Robinson, P. (1997). Muc-7 named entity task definition. In *Proceedings of the 7th Conference on Message Understanding*, volume 29, pages 1–21.

Coucke, A., Saade, A., Ball, A., Bluche, T., Caulier, A., Leroy, D., Doumouro, C., Gisselbrecht, T., Caltagirone, F., Lavril, T., et al. (2018). Snips voice platform : an embedded spoken language understanding system for private-by-design voice interfaces. *arXiv preprint arXiv :1805.10190*.

Cover, T. M., Hart, P., et al. (1967). Nearest neighbor pattern classification. *IEEE transactions on information theory*, 13(1) :21–27.

Dale, R. (2016). The return of the chatbots. *Natural Language Engineering*, 22(5) :811–817.

Ditterrich, T. (1997). Machine learning research : four current direction. *Artificial Intelligence Magzine*, 4 :97–136.

Gandhi, R. (2018). Introduction to machine learning algorithms : Logistic regression. En ligne https ://hackernoon.com/introduction-to -machine-learning-algorithms-logistic-regression-cbdd82d81a36.

Ganitkevitch, J., Van Durme, B., and Callison-Burch, C. (2013). Ppdb : The paraphrase database. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics : Human Language Technologies*, pages 758–764.

Géron, A. (2017). *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems.* " O'Reilly Media, Inc.".

Gershenson, C. (2003). Artificial neural networks for beginners. *arXiv preprint cs/0308031*.

gk_ (2017). Text classification using algorithms. En ligne https ://chatbotslife.com/text- classification-using-algorithms-e4d50dcba45.

Glasmachers, T. (2017). Limits of end-to-end learning. *arXiv preprint arXiv :1704.08305*.

Glass, J. R., Polifroni, J., and Seneff, S. (1994). Multilingual language generation across multiple domains. In *Third International Conference on Spoken Language Processing*.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Gupta, A., Agarwal, A., Singh, P., and Rai, P. (2018). A deep generative framework for paraphrase generation. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Gupta, A., Daly, T., and Ban, T. (2019). Method and system for generating a conversational agent by automatic paraphrase generation based on machine translation. US Patent App. 15/667,283.

Hassan, S., Csomai, A., Banea, C., Sinha, R., and Mihalcea, R. (2007). Unt : Subfinder : Combining knowledge sources for automatic lexical substitution. In *Proceedings of the Fourth International Workshop on Semantic Evaluations (SemEval-2007)*, pages 410–413.

Hemphill, C. T., Godfrey, J. J., and Doddington, G. R. (1990). The atis spoken language systems pilot corpus. In *Speech and Natural Language : Proceedings of a Workshop Held at Hidden Valley, Pennsylvania, June 24-27, 1990*.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8) :1735–1780.

Huang, J., Zhou, M., and Yang, D. (2007). Extracting chatbot knowledge from online discussion forums. In *IJCAI*, volume 7, pages 423–428.

Husson, F., Lê, S., and Pagès, J. (2016). *Analyse de données avec R*. Pratique de la statistique. Presses Universitaires de Rennes.

Jain, A. K., Mao, J., and Mohiuddin, K. (1996). Artificial neural networks : A tutorial. *Computer*, 3 :31–44.

Kaleem, M., Alobadi, O., O'Shea, J., and Crockett, K. (2016). Framework for the formulation of metrics for conversational agent evaluation. In *RE-WOCHAT : Workshop on Collecting and Generating Resources for Chatbots and Conversational Agents-Development and Evaluation Workshop Programme (May 28 th, 2016)*, page 20.

Kamphaug, Å., Granmo, O.-C., Goodwin, M., and Zadorozhny, V. I. (2017). Towards open domain chatbots—a gru architecture for data driven conversations. In *International Conference on Internet Science*, pages 213–222. Springer.

Kar, R. and Haldar, R. (2016). Applying chatbots to the internet of things : Opportunities and architectural elements. *arXiv preprint arXiv :1611.03799*.

Kauchak, D. and Barzilay, R. (2006). Paraphrasing for automatic evaluation. In *Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics*, pages 455–462. Association for Computational Linguistics.

Kim, Y. (2014). Convolutional neural networks for sentence classification. *arXiv preprint arXiv :1408.5882*.

Kozlowski, R., McCoy, K. F., and Vijay-Shanker, K. (2003). Generation of single-sentence paraphrases from predicate/argument structure using lexico-grammatical resources. In *Proceedings of the second international workshop*

on Paraphrasing-Volume 16, pages 1–8. Association for Computational Linguistics.

Li, Z., Jiang, X., Shang, L., and Li, H. (2017). Paraphrase generation with deep reinforcement learning. *arXiv preprint arXiv :1711.00279*.

Lin, D. and Pantel, P. (2001). Discovery of inference rules for question-answering. *Natural Language Engineering*, 7(4) :343–360.

Lin, K., Li, D., He, X., Zhang, Z., and Sun, M.-T. (2017). Adversarial ranking for language generation. In *Advances in Neural Information Processing Systems*, pages 3155–3165.

Liu, B. and Lane, I. (2016). Attention-based recurrent neural network models for joint intent detection and slot filling. *arXiv preprint arXiv :1609.01454*.

Ma'amari, M. (2018). Nlp | sequence to sequence networks| part 2|seq2seq model (encoderdecoder model). En ligne https ://towardsdatascience.com/nlp-sequence-to -sequence-networks-part -2-seq2seq-model-encoderdecoder-model-6c22e29fd7e1.

Madnani, N. and Dorr, B. J. (2010). Generating phrasal and sentential paraphrases : A survey of data-driven methods. *Computational Linguistics*, 36(3) :341–387.

Manaswi, N. K. (2018). Developing chatbots. In *Deep Learning with Applications Using Python*, pages 145–170. Springer.

Martinez, F. F., Blázquez, J., Ferreiros, J., Barra, R., Macias-Guarasa, J., and Lucas-Cuesta, J. M. (2008). Evaluation of a spoken dialogue system for controlling a hifi audio system. In *2008 IEEE Spoken Language Technology Workshop*, pages 137–140. IEEE.

McKeown, K. R. (1980). Paraphrasing using given and new information in a question-answer system. *Technical Reports (CIS)*, page 723.

melepe (2017). Un peu de machine learning avec les svm. En ligne https ://zestedesavoir.com/tutoriels/1760 /un-peu-de-machine-learning-avec-les-svm/#2-cest-lheure-de-lentrainement-.

Mensio, M., Rizzo, G., and Morisio, M. (2018). Multi-turn qa : A rnn contextual approach to intent classification for goal-oriented systems. In *Companion Proceedings of the The Web Conference 2018*, pages 1075–1080. International World Wide Web Conferences Steering Committee.

Mikolov, T., Chen, K., Corrado, G., and Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv :1301.3781*.

Moller, S., Engelbrecht, K.-P., Kuhnel, C., Wechsung, I., and Weiss, B. (2009). A taxonomy of quality of service and quality of experience of multimodal human-machine interaction. In *2009 International Workshop on Quality of Multimedia Experience*, pages 7–12. IEEE.

Muischnek, K. and Müürisep, K. (2018). Collection of resources and evaluation of customer support chatbot. In *Human Language Technologies–The Baltic Perspective : Proceedings of the Eighth International Conference Baltic HLT 2018*, volume 307, page 30. IOS Press.

Nikitina, S., Daniel, F., Baez, M., and Casati, F. (2018). Crowdsourcing for reminiscence chatbot design. *arXiv preprint arXiv :1805.12346*.

Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. (2002). Bleu : a method for automatic evaluation of machine translation. In *Proceedings of the 40th annual meeting on association for computational linguistics*, pages 311–318. Association for Computational Linguistics.

Patel, S. (2017). Chapter 2 : Svm (support vector machine)–theory. En ligne https ://medium.com/machine-learning-101 /chapter-2-svm-support-vector-machine-theory-f0812effc72.

Pavlick, E., Rastogi, P., Ganitkevitch, J., Van Durme, B., and Callison-Burch, C. (2015). Ppdb 2.0 : Better paraphrase ranking, fine-grained entailment relations, word embeddings, and style classification. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2 : Short Papers)*, pages 425–430.

Pawlak, Z. (1982). Rough sets. *International journal of computer & information sciences*, 11(5) :341–356.

Power, R. and Scott, D. (2005). Automatic generation of large-scale paraphrases. *In : 3rd International Workshop on Paraphrasing (IWP2005)*.

Prabhu (2018). Understanding of convolutional neural network (cnn)-deep learning. En ligne https ://medium.com/@RaghavPrabhu/ understanding-of-convolutional-neural-network- cnn-deep-learning-99760835f148.

Quirk, C., Brockett, C., and Dolan, W. (2004). Monolingual machine translation for paraphrase generation. In *Proceedings of the 2004 conference on empirical methods in natural language processing*, pages 142–149.

Rabiner, L. R. (1989). A tutorial on hidden markov models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2) :257–286.

Radziwill, N. M. and Benton, M. C. (2017). Evaluating quality of chatbots and intelligent conversational agents. *arXiv preprint arXiv :1704.04579*.

Rahman, A., Al Mamun, A., and Islam, A. (2017). Programming challenges of chatbot : Current and future prospective. In *2017 IEEE Region 10 Humanitarian Technology Conference (R10-HTC)*, pages 75–78. IEEE.

Rish, I. et al. (2001). An empirical study of the naive bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46.

Rouse, M. (2018). recurrent neural networks. En ligne https ://searchenterpriseai.techtarget.com /definition/recurrent-neural-networks.

Searle, J. R. (1968). Austin on locutionary and illocutionary acts. *The philosophical review*, 77(4) :405–424.

Searle, J. R. (1975). A taxonomy of illocutionary acts. *University of Minnesota Press, Minneapolis*.

Searle, J. R. (1976). A classification of illocutionary acts. *Language in society*, 5(1) :1–23.

Shawar, B. A. and Atwell, E. (2007). Different measurements metrics to evaluate a chatbot system. In *Proceedings of the workshop on bridging the gap : Academic and industrial research in dialog technologies*, pages 89–96. Association for Computational Linguistics.

Shibuya, N. (2018). Demystifying cross-entropy. En ligne https ://towardsdatascience.com/demystifying- cross-entropy-e80e3ad54a8.

Stoecklé, L. (2017). l'app fatigue et les chatbots. En ligne https ://medium.com/@addventa/quest-ce-qu-un-chatbot-127c9b85dedb.

Stöckl, A. (2017). Classification of chatbot inputs. En ligne https ://www.researchgate.net/profile/Andreas_Stoeckl2/publication/318661551_Classification_of_Chatbot_Inputs/links/597642e8a6fdcc8348aa52e4/Classification-of-Chatbot-Inputs.pdf.

Tarau, P. and Figa, E. (2004). Knowledge-based conversational agents and virtual storytelling. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 39–44. ACM.

Uniqtech (2018a). Cosine similarity – understanding the math and how it works (with python codes). En ligne https ://www.machinelearningplus.com/nlp/cosine-similarity/.

Uniqtech (2018b). Understand the softmax function in minutes. En ligne https ://medium.com/data-science-bootcamp/ understand-the-softmax-function-in-minutes-f3a59641e86d.

Vermersch, P. (2007). Approche des effets perlocutoires. *Expliciter*, 71 :1–23.

Walker, M. A., Litman, D. J., Kamm, C. A., and Abella, A. (1997). Paradise : A framework for evaluating spoken dialogue agents. *arXiv preprint cmp-lg/9704004*.

Wallace, R. (1995). Artificial linguistic internet computer entity (alice).

Wang, C.-F. (2019). The vanishing gradient problem. En ligne https ://towardsdatascience.com/the-vanishing -gradient-problem-69bf08b15484.

Warner, M. (2002). Wanted : A definition of intelligence. Technical report, CENTRAL INTELLIGENCE AGENCY WASHINGTON DC CENTER FOR THE STUDY OF INTELLIGENCE.

Weizenbaum, J. et al. (1966). Eliza—a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1) :36–45.

Wu, Y., Wang, G., Li, W., and Li, Z. (2008). Automatic chatbot knowledge acquisition from online forum via rough set and ensemble learning. In *2008 IFIP International Conference on Network and Parallel Computing*, pages 242–246. IEEE.

Xu, P. and Sarikaya, R. (2013). Convolutional neural network based triangular crf for joint intent detection and slot filling. In *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*, pages 78–83. IEEE.

Yu, Z., Xu, Z., Black, A. W., and Rudnicky, A. (2016). Chatbot evaluation and database expansion via crowdsourcing. In *Proceedings of the chatbot workshop of LREC*, volume 63, page 102.

Zhao, S., Lan, X., Liu, T., and Li, S. (2009). Application-driven statistical paraphrase generation. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP : Volume 2-Volume 2*, pages 834–842. Association for Computational Linguistics.

Zhao, S., Wang, H., Liu, T., and Li, S. (2008). Pivot approach for extracting paraphrase patterns from bilingual corpora. In *Proceedings of ACL-08 : HLT*, pages 780–788.

Zong, C., Zhang, Y., Yamamoto, K., Sakamoto, M., and Shirai, S. (2001). Approach to spoken chinese paraphrasing based on feature extraction. In *NLPRS*, pages 551–556. Citeseer.

Deuxième partie

Annexes

Table des matières

Annexe A

Rasa

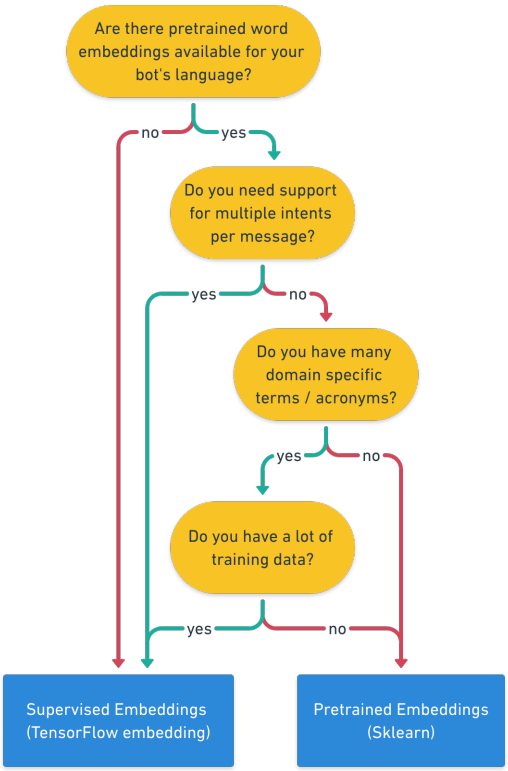


FIGURE A.1 – Organigramme de Rasa pour le choix d'un classificateur

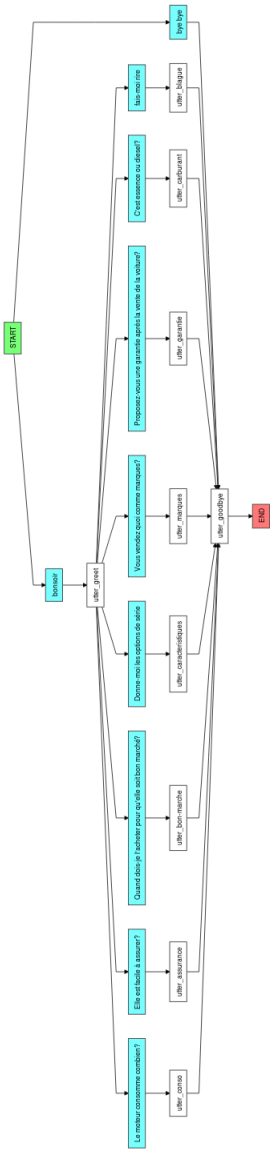


FIGURE A.2 – Stories

Annexe B

ReSyf

```
<Lemma type="Form">
    <feat att="lexeme" val="manger peu"/>
    <feat att="partOfSpeech" val="VER"/>
</Lemma>
<Sense id="manger-peu_VER_MONO">
    <SenseExample id="manger-peu_VER_MONO">
        <feat att="type" val="synonyms"/>
        <feat att="annotation" val="manually"/>
        <feat att="score_lemma" val="--"/>
        <feat att="score_sense" val="--"/>
        <feat att="word" val="manger peu"/>
        <feat att="rank" val="1"/>
    </SenseExample>
    <SenseExample id="manger-peu_VER_MONO">
        <feat att="type" val="synonyms"/>
        <feat att="annotation" val="manually"/>
        <feat att="word" val="avoir un appétit d'oiseau"/>
        <feat att="rank" val="2"/>
    </SenseExample>
    <SenseExample id="manger-peu_VER_MONO">
        <feat att="type" val="synonyms"/>
        <feat att="annotation" val="manually"/>
        <feat att="word" val="chipoter"/>
        <feat att="rank" val="4"/>
    </SenseExample>
    <SenseExample id="manger-peu_VER_MONO">
        <feat att="type" val="synonyms"/>
        <feat att="annotation" val="manually"/>
        <feat att="word" val="picorer"/>
        <feat att="rank" val="3"/>
    </SenseExample>
```

FIGURE B.1 – Exemple d'entrée de *ReSyf*

Annexe C

PPDB

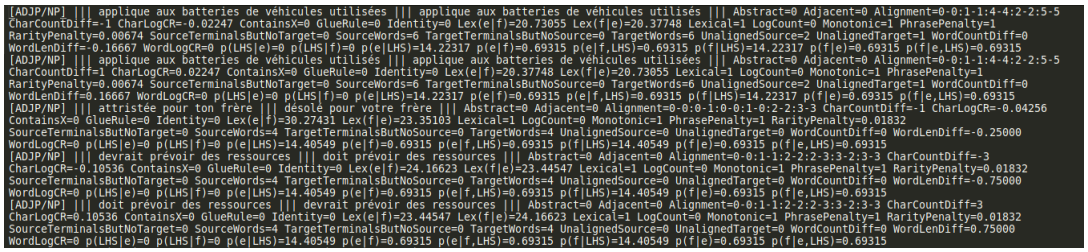


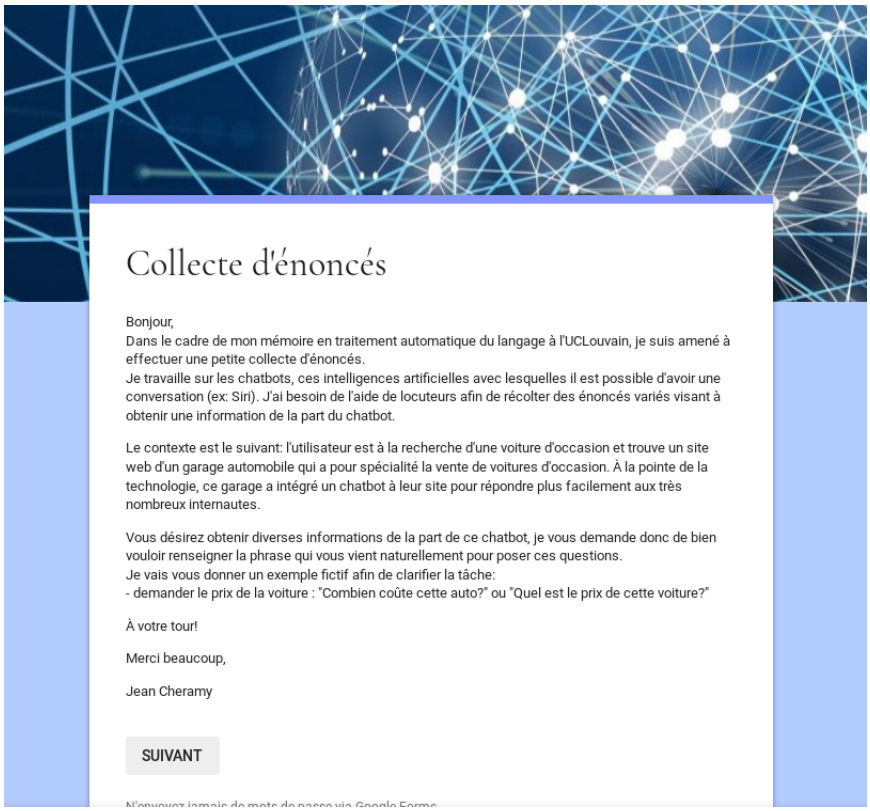
FIGURE C.1 – Exemple d'entrées de *PPDB* en texte brut

4173856	il n'existe actuellement	il n'y a actuellement
4173857	il n'y a actuellement	il n'existe actuellement
4173858	il s'est joint	il s'est joint à
4173859	il s'est joint à	il s'est joint
4173860	l'appui de leur	soutien de leurs
4173861	la conciliation.	de conciliation.

FIGURE C.2 – Exemple d'entrée SQLite de *PPDB*

Annexe D

Récolte d'énoncés



Collecte d'énoncés

Bonjour,
Dans le cadre de mon mémoire en traitement automatique du langage à l'UCLouvain, je suis amené à effectuer une petite collecte d'énoncés.
Je travaille sur les chatbots, ces intelligences artificielles avec lesquelles il est possible d'avoir une conversation (ex: Siri). J'ai besoin de l'aide de locuteurs afin de récolter des énoncés variés visant à obtenir une information de la part du chatbot.

Le contexte est le suivant: l'utilisateur est à la recherche d'une voiture d'occasion et trouve un site web d'un garage automobile qui a pour spécialité la vente de voitures d'occasion. À la pointe de la technologie, ce garage a intégré un chatbot à leur site pour répondre plus facilement aux très nombreux internautes.

Vous désirez obtenir diverses informations de la part de ce chatbot, je vous demande donc de bien vouloir renseigner la phrase qui vous vient naturellement pour poser ces questions.
Je vais vous donner un exemple fictif afin de clarifier la tâche:
- demander le prix de la voiture : "Combien coûte cette auto?" ou "Quel est le prix de cette voiture?"

À votre tour!

Merci beaucoup,

Jean Cheramy

SUIVANT

N'envoyez jamais de mots de passe via Google Forms.

FIGURE D.1 – Description de la récolte

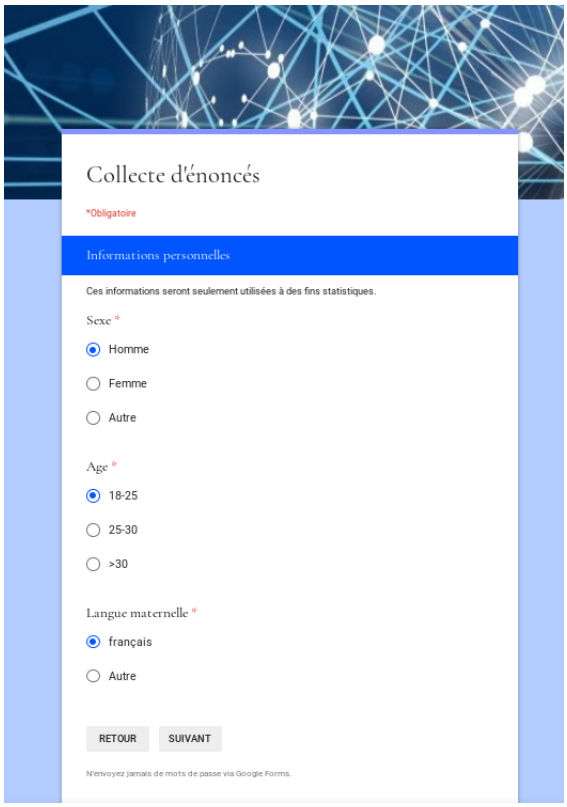


FIGURE D.2 – Demande d'informations personnelles

FIGURE D.3 – Collecte d'énoncés

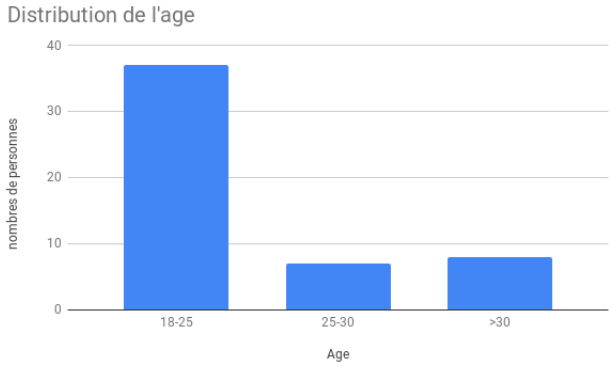


FIGURE D.4 – Âge

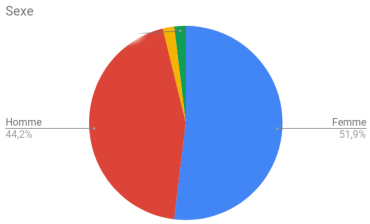


FIGURE D.5 – Sexe

Annexe E

Exemples de résultats

```
2019-08-10 14:32:28 INFO     rasa.nlu.test  - F1-Score:  0.5781978248056998
2019-08-10 14:32:28 INFO     rasa.nlu.test  - Precision: 0.6102069134737012
2019-08-10 14:32:28 INFO     rasa.nlu.test  - Accuracy:  0.5755968169761273
2019-08-10 14:32:28 INFO     rasa.nlu.test  - Classification report:
                precision    recall  f1-score   support

      assurance       0.33      0.41      0.37        44
         blague       0.51      0.80      0.62        35
     bon-marche       0.47      0.42      0.45        45
caracteristiques       0.54      0.69      0.61        45
      carburant       0.55      0.61      0.58        36
          conso       0.69      0.72      0.70        46
       garantie       0.73      0.49      0.59        45
        goodbye       0.63      0.48      0.55        25
          greet       0.50      0.22      0.31         9
        marques       1.00      0.64      0.78        47

      micro avg       0.58      0.58      0.58       377
      macro avg       0.60      0.55      0.55       377
   weighted avg       0.61      0.58      0.58       377
```

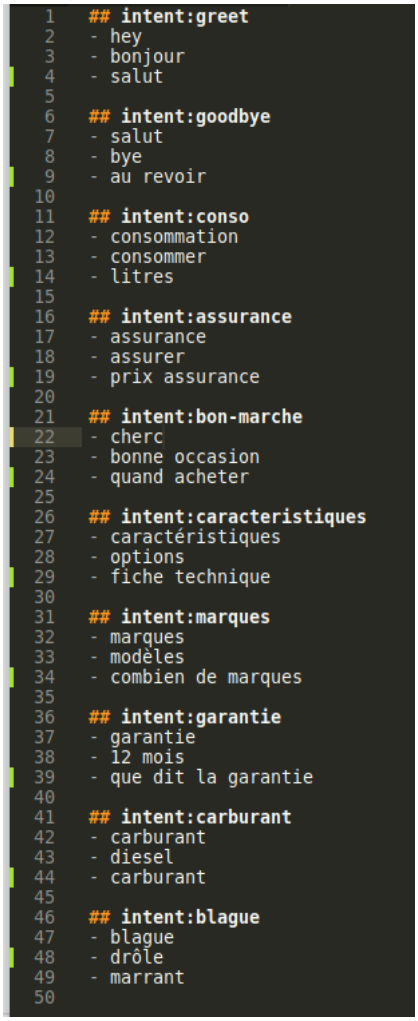
FIGURE E.1 – Résultats de la *baseline* avec *sklearn*

```
2019-08-10 14:34:38 INFO     rasa.nlu.test  - F1-Score:  0.8534379019218725
2019-08-10 14:34:38 INFO     rasa.nlu.test  - Precision: 0.869592148574794
2019-08-10 14:34:38 INFO     rasa.nlu.test  - Accuracy:  0.8488063660477454
2019-08-10 14:34:38 INFO     rasa.nlu.test  - Classification report:
                precision    recall  f1-score   support

                     0.00      0.00      0.00         0
      assurance       0.78      0.73      0.75        44
         blague       0.89      0.97      0.93        35
     bon-marche       0.85      0.76      0.80        45
caracteristiques       0.92      0.78      0.84        45
      carburant       0.81      0.97      0.89        36
          conso       0.87      0.98      0.92        46
       garantie       0.89      0.91      0.90        45
        goodbye       0.88      0.60      0.71        25
          greet       1.00      0.56      0.71         9
        marques       0.90      0.94      0.92        47

      micro avg       0.85      0.85      0.85       377
      macro avg       0.80      0.74      0.76       377
   weighted avg       0.87      0.85      0.85       377
```

FIGURE E.2 – Résultats de la *baseline* avec *TensorFlow*



```
## intent:greet
- hey
- bonjour
- salut

## intent:goodbye
- salut
- bye
- au revoir

## intent:conso
- consommation
- consommer
- litres

## intent:assurance
- assurance
- assurer
- prix assurance

## intent:bon-marche
- cherc
- bonne occasion
- quand acheter

## intent:caracteristiques
- caractéristiques
- options
- fiche technique

## intent:marques
- marques
- modèles
- combien de marques

## intent:garantie
- garantie
- 12 mois
- que dit la garantie

## intent:carburant
- carburant
- diesel
- carburant

## intent:blague
- blague
- drôle
- marrant

```

FIGURE E.3 – NLU de Dummy

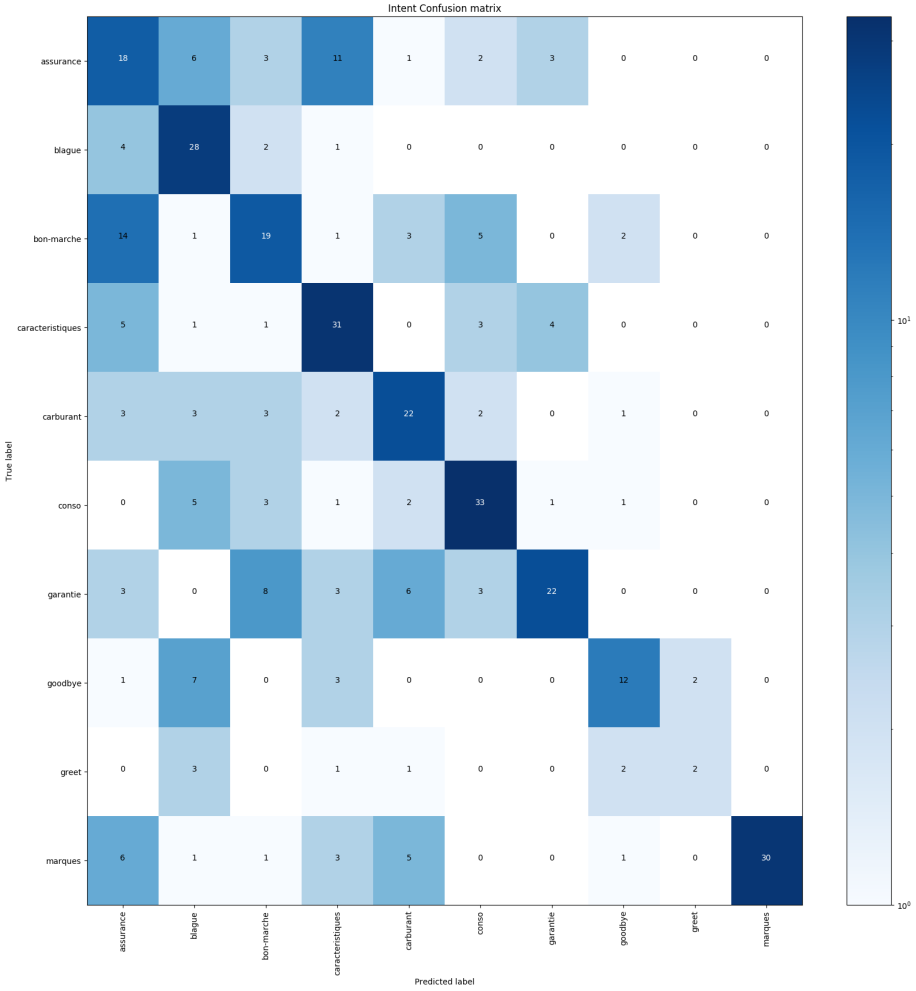


FIGURE E.4 – *Confusion matrix* de la *baseline* avec *sklearn*

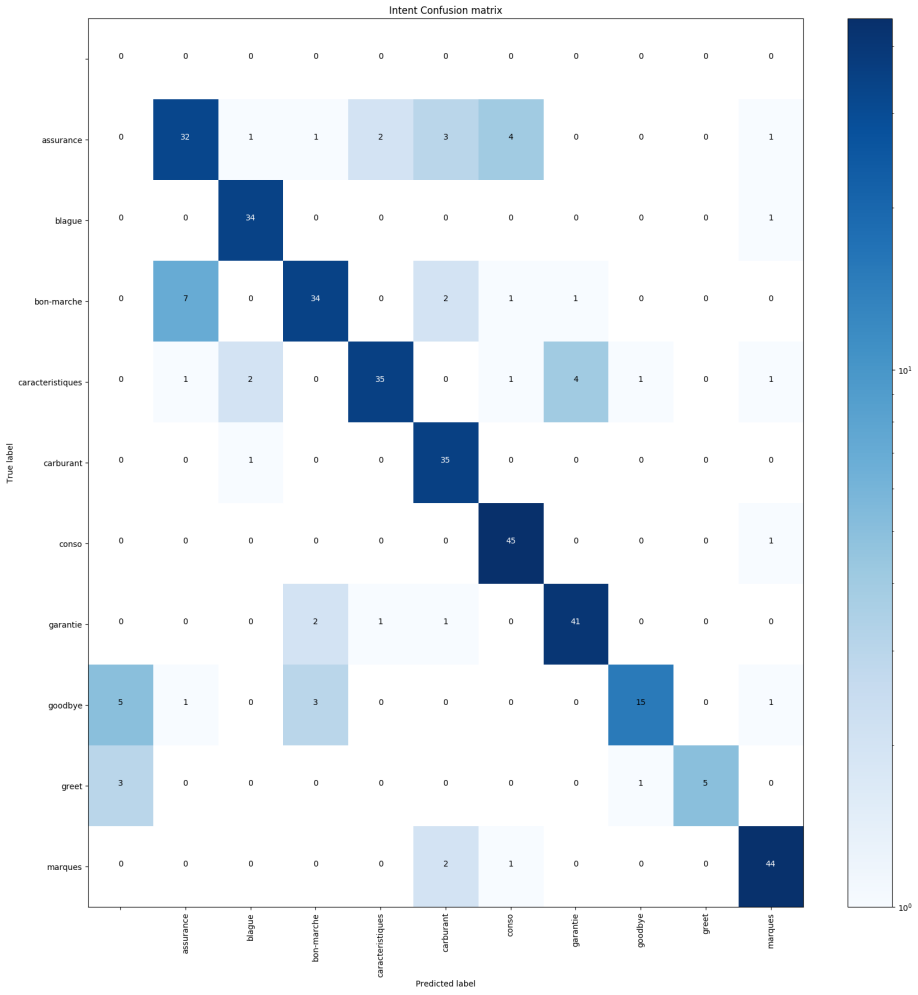


FIGURE E.5 – *Confusion matrix* de la *baseline* avec *TensorFlow*

Annexe F

Code Python

F.1 PPDB

F.1.1 extractData.py

Listing F.1 – extractData.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
#==================================
# Author: Jean Cheramy
# Date : August 2019
#==================================
# Because of some computational problems, we had
# to split the original ppdb database in several
# smaller files:
#
# split -l 200000 ppdb-1.0-s-phrasal
#
# the -l option is useful to split the data every 200 000 lines
# the -b (bytes) options cut the sentences and corrupt the data
#
# Doing this we had no more "memory error" or other problems.
# So all the smaller files are grouped in a directory.
# We read them and we transform the data (plain text) in
# a sqlite database because we had not enough RAM to handle the
# entire corpus (4,3 Go). This transformation into sqlite seemed
# to be a good compromise between performance and easy to use.
# We know that it's not very fast but we wanted to write
# a code wich can be understood easily.
#==================================
# We have to give a name for the database "ppdb.sqlite" for example
# the command line is:
#
#     python extractData.py /PARAPHRASES_S ppdb.sqlite
```

```
29 #
30 #==================================
31 # We have to thank some persons:
32 # https://medium.com/analytics-vidhya/programming-with-databases-in
       -python-using-sqlite-4cecbef51ab9
33 # https://www.quora.com/If-SQL-is-a-high-level-language-why-is-it-
       so-fast
34 # These artices helped us in the redaction of this code.
35 #==================================
36 import re
37 import os
38 import sqlite3
39 import sys
40 import time
41
42 #==================================
43 # MAIN Function
44 #==================================
45 # Here we create the database and we read
46 # the ppdb directory (with splitted corpus).
47 # After that we read them, we split the rows
48 # and we preprocess the data before adding it
49 # in our database.
50 #==================================
51 def create_sqlite(DB_name, Dir):
52
53   # We create the database
54   conn = sqlite3.connect(DB_name)
55   cursor = conn.cursor()
56   print("Db opened")
57   cursor.execute('''CREATE TABLE TEST
58            (SENTENCE           TEXT    NOT NULL,
59            PARAPHRASE            TEXT     NOT NULL);''')
60
61   # We read the directory and save the names of the files
62   # in a list
63   liste = os.listdir(Dir)
64
65   # We instantiate the PATH to which we are going to add the names
66   # of the files
67   PATH = os.path.dirname(os.path.realpath(__file__)) + "/" + Dir
68
69   # For each file in the list, we create the PATH.
70   for file in liste:
71     file = PATH + "/" + file
72
73     # We open the PATh and we read the lines, the lines are like
74     # that:
75     # [X] ||| cession des droits ||| transfert de droits |||
       Abstract=0 Adjacent=0 Alignment=0-0:1-1:2-2 CharCountDiff=1
       CharLogCR=0.05407 ContainsX=1
76     # GlueRule=0 Identity=0 Lex(e|f)=8.06829 Lex(f|e)=5.48712
```

```
    Lexical=1 LogCount=0 Monotonic=1 PhrasePenalty=1 RarityPenalty
    =0 SourceTerminalsButNoTarget=0
77  # SourceWords=3 TargetTerminalsButNoSource=0 TargetWords=3
    UnalignedSource=0 UnalignedTarget=0 WordCountDiff=0 WordLenDiff
    =0.33333 WordLogCR=0 p(LHS|e)=0.28768
78  # p(LHS|f)=0.27354 p(e|LHS)=15.91153 p(e|f)=1.32290 p(e|f,LHS)
    =1.42234 p(f|LHS)=15.61255 p(f|e)=1.03807 p(f|e,LHS)=1.12336
79  #
80  # Here, we just want to keep the first sentence and the
81  # associated paraphrase.
82  with open(file) as f:
83      lines = f.readlines()
84
85      # So, for each line, we split on the ||| separator and
86      # we keep the string values that interest us
87      for line in lines:
88          array = line.split('|||')
89          var = array[1]
90          var2 = array[2]
91
92          # We preprocess them and add them in the db
93          var = preprocess(var)
94          var2 = preprocess(var2)
95          cursor.execute("insert into TEST values ( ?, ?)", (var,
     var2))
96
97  #We commit and iterate over the different files
98  conn.commit()
99
100 # We close the db when it is finished
101 conn.close()
102
103 #==================================
104 # PREPROCESS Function
105 #==================================
106 # Here we preprocess the strings because there
107 # are some problems like:
108 # added spaces, ponctuation not well postionned,...
109 # We use regex (substitution) to handle it and
110 # modify the data.
111 # This preprocessing is useful to create a database more
112 # 'natural' and potentially more useful to match nlu sentences.
113 #==================================
114 def preprocess(string):
115   string = str(string)
116
117   # We remove spaces at the beginning and at the end of the string
118   string = string.strip()
119
120   # We remove some useless spaces and we put the ponctuation to the
121   # right place.
122   if "'" in string:
```

```
123     string = re.sub(r"(.+)'\s(.+)", r"\1'\2", string)
124   if "," in string:
125     string = re.sub(r"(.+)\s,(\s.+)", r"\1,\2", string)
126     string = re.sub(r"(.+)\s,", r"\1,", string)
127   if "." in string:
128     string = re.sub(r"(.+)\s([.;?!])", r"\1\2", string)
129
130   return string
131
132
133 # Here we just call the fonction and we mesure
134 # the time needed.
135 if __name__ == '__main__':
136   start_time = time.time()
137
138   DB_name = sys.argv[2]
139   Dir = sys.argv[1]
140   create_sqlite(DB_name, Dir)
141
142   print("%s s" % (time.time() - start_time))
```

F.1.2 findInSqlite.py

Listing F.2 – findInSqlite.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  #==================================
4  # Author: Jean Cheramy
5  # Date : August 2019
6  #==================================
7  # We have to give to this program the name of
8  # the nlu file ('nlu.json') and the name of the
9  # sqlite database ('ppdb.sqlite').
10 #
11 #       python findInSqlite.py nlu.json ppdb.sqlite
12 #
13 #==================================
14 # The sqlite database contains the PPDB corpus (S size)
15 # in a easy to use format:
16 #
17 #     [Phrase, Paraphrase]
18 #
19 # The nlu file is in JSON and contains the nlu baseline that we
20 # want to improve with a generation, or retrieval, of paraphrases.
21 # This programs tries to find or generate paraphrases from the
22 # nlu sentences.
23 # We tried different approaches that we will comment below.
24 #==================================
25 import re
26 import os
27 import sqlite3
28 import time
```

```
import nltk
import difflib
import sys
import json
import spacy
import os
from nltk.tag import StanfordPOSTagger
from evaluate1 import evaluator
from preprocess1 import preprocess

#==================================
# MAIN function
#==================================
# In this function we firstly create a
# liste of words and another liste of
# sentences from the nlu.
# This words list would be useful to retrieve
# paraphrases that contains them to check
# if it is a good paraphrase.
#==================================
def find_sqlite(file, db):
  # we call the create_list function, it takes some time
  # because it creates a list of words (with POS tags) and a
  # dictionnary[intent] = [exemple1, exemple2, exemple3,...]
  liste, sentences = create_list(file)

  # We retrieve a list of intents (keys of the sentences
    dictionnary)
  intents = sentences.keys()

  # We instantiate some results lists
  RESULTATS_1 = []
  RESULTATS_2 = []
  RESULTATS_3 = []

  # we connect to the database
  conn = sqlite3.connect(db)
  cursor = conn.cursor()
  cursor = conn.execute("SELECT SENTENCE, PARAPHRASE from TEST")
  print("Processing...")

  # For each row of the ppdb file
  for row in cursor:
    # For each intent
    for intent in intents:
      # For each sentence, we try different approaches
      for string in sentences[intent]:

      # We try our approaches after having done
      # a preprocessing: we transform the current sentence
      # in lower case (because there is no MAJ in the db)
        string = string.lower()
```

```
80
81      #==============================
82      # Approach N 1
83      #==============================
84      # In this approach we try to generate paraphrases
85      # from the sqlite. To do that we transform our sentence
86      # with some local modification.
87      # For exemple, if the sentence is:
88      # "je voudrais l'assurer, ce n'est pas trop difficile?"
89      # and the current row is: ('je voudrais', 'je souhaiterais')
90      # We transform the sentence like that:
91      # "je souhaiterais l'assurer, ce n'est pas trop difficile?"
92      #
93      # See the comments of the function for more details
94      #==============================
95        results_1 = find_create(string, row)
96        if(not results_1):
97          pass
98        else:
99          RESULTATS_1.append(results_1)
100
101     #==============================
102     # Approach N 2
103     #==============================
104     # In this approach we try to retrieve paraphrase
105     # from the sqlite with some refinements.
106     # We start with computing the char lenght of the
107     # db paraphrases and we keep ones that are close to
108     # the sentence lenght.
109     # We compute the words lenght and we do the same thing.
110     # After that we compute a similarity score for each
111     # paraphrase and we keep the best ones (in relation to the
112     # current sentence). See the comments of the function for
113     # more details
114     #==============================
115     # DEPRECATED APPROACH
116     # not efficient and takes too much time to process
117     #==============================
118       # results_2 = find_refine(string, row)
119       # if(not results_2):
120       #   pass
121       # else:
122       #   print(results_2)
123       #   final_sentences = evaluator(string, results_2)
124       #   if(final_sentences):
125       #     print(final_sentences)
126       #     RESULTATS_2.append(final_sentences)
127       #   else:
128       #     pass
129
130     #==============================
131     # Approach N 3
```

```
132       #============================
133       # In this function we try to recover paraphrases
134       # with another approach. We built a words list
135       # wich only take names, infinitive verbs and adjectives
136       # (semantically useful words).
137       # If the row contains one of the list, we check if the
138       # value of the row is located into the sentence. If so
139       # we replace it by its paraphrase.
140       # ex: "J'aime les pommes vertes"
141       #   [les pommes vertes, les fruits verts]
142       #   "J'aime les fruits verts"
143       #
144       # This approach seems very similar to the first one.
145       # It's true. The main difference is that we focus here on
146       # some tags which contain more information.
147       #============================
148         results_3 = find_words_list(string, liste, row)
149         if(not results_3):
150           pass
151         else:
152           RESULTATS_3.append(results_3)
153
154       if(RESULTATS_1):
155         save_json("nlu1-bis.json", RESULTATS_1, intent)
156         RESULTATS_1 = []
157       # if(RESULTATS_2):
158       #   save_json("nlu2.json", RESULTATS_2, intent)
159       #   RESULTATS_2 = []
160       if(RESULTATS_3):
161         save_json("nlu3-bis.json", RESULTATS_3, intent)
162         RESULTATS_3 = []
163
164   conn.close()
165 #==============================
166 # Function 1: save into json
167 #==============================
168 # This function is useful to save the results
169 # of the searching functions into a JSON file.
170 # We decided to do that in JSON because it's easier
171 # to check if a sentence is already, or not, in the
172 # file. It's also easy to read it and to parse it.
173 # We have to give it the name of the filename, the
174 # array of sentences and the current intent.
175 #==============================
176 def save_json(filename, array, intent):
177
178   with open(filename, 'r+', encoding='utf-8') as json_file:
179     data = json.load(json_file)
180
181     # For each sentence, we instantiate a boolean to False
182     # and we preprocess the sentence(tokenized, remove punctuation)
183     for sentence in array:
```

```
184       indice = False
185       tokenized = preprocess(str(sentence))
186
187       # we iterate on the nlu sentences and we also preprocess them
188       # to compare the tokenized good-example with the nlu
189       # sentences.The preprocess step would not be necessary if
190       # there were not so many sentences composed of the same words
191       # but still different (punctuation, spaces,...)
192       for item in data["rasa_nlu_data"]["common_examples"]:
193         nlu_sentence = preprocess(str(item["text"]))
194
195         # If tokenized sentences are equal, we give
196         # the True value to the boolean (because that means
197         # that the sentence is already in there and we do not
198         # want to add it and overfit the database)
199         if(tokenized == nlu_sentence):
200           indice = True
201         else:
202           pass
203
204       # If the value is True, we pass.
205       # else, we add the sentence (original one, not the tokenized)
206       # into the nlu file
207       if(indice==True):
208         pass
209       else:
210         data["rasa_nlu_data"]["common_examples"].insert(0, {"intent
      ": intent, "text": sentence})
211         with open(filename, 'w', encoding='utf-8') as json_file:
212           json.dump(data, json_file, ensure_ascii=False, indent=2)
213
214 #================================
215 # Function 2: find and create
216 #================================
217 # This function reads the current row
218 # of the sqlite database and, if the content
219 # of the row is also in the sentence, we
220 # replace it with the paraphrase
221 # row = ["j'aime", "j'adore"]
222 # ex: "j'aime les pommes"
223 # -> "j'adore les pommes"
224 #================================
225 def find_create(sentence, row):
226
227   # The try-except statement is useful to avoid some bug
228   # that we did not understand. We thought that it was because
229   # of the presence of some special character in the row elements
230   # but, after having escaped them, there was still some bugs
231   # (but much less than before!)
232   # We do not escape the second elem of the row because it creates
233   #some char troubles in the rewriting (add some '\' in the string)
234   try:
```

```
235     string1 = re.escape(row[0])
236     string2 = row[1]
237 
238     # If the current sentence of the row is in the original
        sentence
239     # we create a new sentence with the paraphrase
240     # We don't run this process in the inverse order because,
241     # in the db, the paraphrase is also an entry sentence
242     #       ["j'adore", "j'aime beaucoup"]
243     #       ["j'aime beaucoup", "j'adore"]
244 
245 
246     # We build a regex to match the first elem of the row
247     # we just check if the string is independant of others words
248     # with the \b (match limits of words)
249     regex =  r"\b" + string1 + r"\b"
250 
251     # If we match the regex into the sentence, we replace the
252     # value of the row by the second one.
253     # Here we have a paraphrase
254     if re.search(regex, sentence):
255       paraphrase = re.sub(regex, string2, sentence)
256       return paraphrase
257     else:
258       pass
259 
260   except:
261     print("still a bug... in find_create")
262 
263 #==============================
264 # Function 3: find with refine
265 #==============================
266 # This function searchs paraphrases of the
267 # current sentence by refinement. It
268 # starts by computing the char similarity
269 # (in number) between the sentence and the paraphrase
270 # If the difference is too big, we do not select
271 # the paraphrase.
272 # If it is similar, we compute the token similarity
273 # (number of words) and and we do the same thing.
274 #-----------------------------------------
275 # We decided to try this approach because computing
276 # Levenchtein, edition distance or BLEU needed too much time.
277 # So we wanted to restrain the number of 'relevant' paraphrases
278 # to give to the evaluator without using too much processus.
279 #-------------------------------------------------
280 # After that, we evaluate the relatedness
281 # and we keep the best ones (we do this 1.126)
282 #================================
283 # This approach is problematic because we only look
284 # at similar sentences (so the linguistic diversity
285 # is not very good). Moreover, computing the
```

```
286 # similarity is not always effective.
287 #================================
288 # This function is not efficient.
289 #================================
290 def find_refine(string, row):
291   first_results = []
292   interim_results = []
293   length_string = len(nltk.word_tokenize(string))
294 
295   # computing the char similarity
296   # (in number) between the sentence and the paraphrase
297   if(similar_char_len(string, row[0])==True):
298     # compute the token similarity
299     # (number of words)
300     if(similar_tokens_len(string,row[0])==True):
301       return row[0]
302 
303 #================================
304 # Function 4: refine with words + create
305 #================================
306 # In his function we use the list of words created
307 # from the nlu file. This list is supposed to be representative
308 # of the semntic content of the nlu (because it is composed
309 # of names, adjectives and verbs).
310 # We firstly refine the corpus by searching sentences that contain
311 # at least one word of the list.
312 # After that we try to generate some sentences
313 # (same goal that the find_create function)
314 #================================
315 def find_words_list(string, liste, row):
316   results = []
317   length_string = len(nltk.word_tokenize(string))
318 
319   # if there is just one word, we pass
320   if(length_string== 1):
321     pass
322   else:
323     # We check if the current row contains a word of the liste
324     for word in liste:
325       if(word in row[0] or word in row[1]):
326         print("ok")
327         # If is is True, we check if the first elem or the second
    one
328         # is inside the current sentence. If it is, we create a
    paraphrase
329         # by replacing the element by its equivalent.
330         # We return the result
331         try:
332 
333           # We do not escape the second elem of the row because it
    creates
334           # some char troubles in the rewriting (add some '\' in
```

```
    the string)
335           string1 = re.escape(row[0])
336           string2 = row[1]
337 
338           # If the current sentence of the row is in the original
    sentence
339           # we create a new sentence with the paraphrase
340           # We don't run this process in the inverse order because,
341           # in the db, the paraphrase is also an entry sentence
342           #       ["j'adore", "j'aime beaucoup"]
343           #       ["j'aime beaucoup", "j'adore"]
344 
345 
346           # We build a regex to match the first elem of the row
347           # we just check if the string is independant of others
    words
348           # with the \b (match limits of words)
349           regex =  r"\b" + string1 + r"\b"
350 
351           # If we match the regex into the sentence, we replace the
352           # value of the row by the second one.
353           # Here we have a paraphrase
354           if re.search(regex, string):
355             paraphrase = re.sub(regex, string2, string)
356             print(paraphrase)
357             return paraphrase
358           else:
359             pass
360 
361         # We launch an exception because we had some troubles with
362         # special characters,...
363         except:
364           print("wrong character in find_words_list l.342, it is
    not supported")
365       else:
366         pass
367 
368 #==============================
369 # Function 5: similar tokens
370 #==============================
371 # In his function we decide if a sentence
372 # is similar to another in terms of tokens
373 # length
374 #==============================
375 def similar_tokens_len(string, paraph):
376 
377   # We use the nltk module to tokenize
378   tokens_s = nltk.word_tokenize(string)
379   tokens_p = nltk.word_tokenize(paraph)
380 
381   # We save the lengths in variables
382   string_len = len(tokens_s)
```

```
383   parahrase_len = len(tokens_p)
384
385   # We instantiate the minimum tokens needed.
386   # If a sentence is smaller, that does not fit.
387   minimum = (string_len/2)+(string_len/4)
388
389   # Idem for the maximum
390   maximum = string_len+(string_len/4)
391
392   # If the second sentence is between with min and max, we
393   # return True
394   if(parahrase_len>minimum and parahrase_len<maximum):
395     return True
396   else:
397     return False
398
399 #================================
400 # Function 6: similar chars length
401 #================================
402 # In his function we decide if a sentence
403 # is similar to another in terms of chars length
404 #================================
405 def similar_char_len(string, phrase):
406
407   # We create variables that contain the
408   # number of chars of the two sentences
409   string_len = len(string)
410   phrase_len = len(phrase)
411
412   # We decide of the min and max (based on
413   # the sentence that we would paraphrase)
414   minimum = (string_len/2)+(string_len/4)
415   maximum = string_len+(string_len/4)
416
417   # if the char length of the paraphrase fits, we
418   # return True
419   if(phrase_len>minimum and phrase_len<maximum):
420     return True
421   else:
422     return False
423
424 #================================
425 # Function 7: create list of words/sentences
426 #================================
427 # This function is very useful because we build
428 # the list of words (semantically important) of the nlu
429 # that would be used later in another function.
430 # We also build the sentences list (from the nlu file, again)
431 # that would be used by every function of this program.
432 #================================
433 # Thank you to stakoverflow for the help:
434 # https://stackoverflow.com/questions/34692987/cant-make-stanford-
```

```
    pos-tagger-working-in-nltk
#================================
# We just have to give the JSON file of the nlu
#================================
def create_list(file):

  # We have to use the Stanford Tagger (because it is very
    efficient for the french)
  jar = '/home/gandalf/MEMOIRE/IMPLEMENTATION/GENERATION/stanford-
    postagger-full-2018-10-16/stanford-postagger-3.9.2.jar'
  model = '/home/gandalf/MEMOIRE/IMPLEMENTATION/GENERATION/stanford
    -postagger-full-2018-10-16/models/french.tagger'
  java_path = "C:/Program Files/Java/jdk1.8.0_121/bin/java.exe"
  os.environ['JAVAHOME'] = java_path
  array_pos = []
  pos_tagger = StanfordPOSTagger(model, jar, encoding='utf8' )
  #============================
  # Step1 : pos tagging of the sentence
  #============================

  # We open the nlu file and we load the data
  dictionnaire = {}
  sentences = []
  array = []
  with open(file, 'r+', encoding='utf-8') as json_file:
    data = json.load(json_file)
    intent1 = data["rasa_nlu_data"]["common_examples"][0]["intent"]

    # Here we read the data in the nlu and we save it into
    # a data structure : a dictionnary of listes. The result
    # will associate a key (an intent) with the related sentences
    # dictionnaire["conso"] = ["consommation?", "que
    # consomme-t-elle?",...]
    for example in data["rasa_nlu_data"]["common_examples"]:
      intent = example["intent"]
      text = example["text"]
      if(intent==intent1):
        tokens = nltk.word_tokenize(text)
        if(len(tokens)==1):
          pass
        else:
          array.append(text)
      else:
        dictionnaire[intent1]=array
        array=[]
        intent1 = intent
        array.append(text)
      # We tokenize the current sentence and we tag
      # this sequence of token
      # we append a list of postags (and their words) in
      # this format: [["chien", NC], ["manger", VINF]]
      wordsList = nltk.word_tokenize(text)
```

```
483       res = pos_tagger.tag(wordsList)
484       array_pos.append(res)
485
486   # We instantiate a list
487   final_array = []
488
489   # For each item in the postag array -> item = [["le", DT],
490   # ["chien", NC],...]
491   for item in array_pos:
492     # x -> ["chien", NC]
493     for x in item:
494       # Here we check if the current tag's token
495       # is a name, a infinitive verb of an adjective
496       # if so, we add it to the final list
497       if("NC" or "ADJ" or "VINF") == x[1]:
498         if (x[0] in final_array):
499           pass
500         else:
501           final_array.append(x[0])
502       else:
503         pass
504   json_file.close()
505
506   # We return the list of words and the list of sentences
507   return final_array, dictionnaire
508
509 if __name__ == '__main__':
510   start_time = time.time()
511   file = sys.argv[1]
512   db = sys.argv[2]
513   find_sqlite(file, db)
514
515   print("%s min" % ((time.time() - start_time)/60))
516
517 #====================================
518 # ARCHIVES
519 #====================================
520 #==============================
521 # exact match
522 #==============================
523 # Dummy function where we try to find
524 # the sentence in the database and to
525 # return the linked paraphrase.
526 # Works for common sentences (I like,...)
527 # but not for specific domains (cars,...)
528 # so we do not use it
529 # We keep it here for information
530 #==============================
531 def exact_match(string, ligne):
532   phrase = str(ligne[0])
533   if(string == phrase):
534     return str(ligne[1])
```

```
535   else:
536     return None
537
538
539 #================================
540 # SOME SIMILARITY METRICS
541 #================================
542 # We just let them for information.
543 # We implemented them, we used them
544 # and we tested them but that was not really
545 # efficient.
546 #================================
547
548
549 #================================================
550 # Levenchtein
551 # https://python.gotrained.com/nltk-edit-distance-jaccard-distance/
552 #================================================
553 def edition_distance(string, ligne):
554   value = nltk.edit_distance(string, ligne)
555   return(value)
556
557 #================================================
558 # https://stackoverflow.com/questions/6690739/high-performance-
      fuzzy-string-comparison-in-python-use-levenshtein-or-difflib
559 #------------------------------------------------
560 # "difflib.SequenceMatcher uses the Ratcliff/Obershelp algorithm it
561 # computes the doubled number of matching characters divided by the
562 # total number of characters in the two strings" Ghassen Hamrouni
563 # (see the link above)
564 #================================================
565 def difflib_seq_match(string, ligne):
566   seq = difflib.SequenceMatcher(None,string,ligne)
567   diff = seq.ratio()*100
568   return(diff)
569
570 #================================================
571 # BLEU
572 # https://www.nltk.org/_modules/nltk/translate/bleu_score.html
573 #================================================
574 def BLEU_Indice(string, ligne):
575   bleu = nltk.translate.bleu_score.sentence_bleu(string, ligne,
        weights = (0.5, 0.5))
576   return(bleu)
```

F.1.3 preprocess1.py

Listing F.3 – preprocess1.py

```
1 #!/usr/bin/env python
2 # -*- coding: utf-8 -*-
3 #================================
4 # Author: Jean Cheramy
```

```
# Date : August 2019
#==================================
# This function is useful to preprocess
# sentences:
# tokenize, lower case, remove punctuation
# and stop-words
#==================================
# thanks to:
# https://www.geeksforgeeks.org/removing-stop-words-nltk-python/
# https://stackoverflow.com/questions/265960/best-way-to-strip
# -punctuation-from-a-string
#==================================
import nltk
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, sent_tokenize
import time

def preprocess(sentence):

  # create a stop-words variable for french
  stop_words = set(stopwords.words('french'))

  # strip punctuation
  sentence = re.sub(r'[^\w\s]','',sentence)

  # we tokenize the sentence
  tokenized = sent_tokenize(sentence)
  wordsList = []

  # For each token we check if it is a stop word or not
  # if it is not, we add it to the wordlist
  for token in tokenized:
      token = token.lower()
      wordsList = nltk.word_tokenize(token)
      wordsList = [w for w in wordsList if not w in stop_words]

  # we return the tokenized sentence in lower case, without
    punctuation and stop-words
  return wordsList

if __name__ == '__main__':
  start_time = time.time()
  phrase = "Je veux! manger un'e' Pomme"
  print(preprocess(phrase))
  print("%s min" % ((time.time() - start_time)/60))
```

F.1.4 evaluate.py

Listing F.4 – evaluate.py

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
```

```
#==================================
# Author: Jean Cheramy
# Date : August 2019
#==================================
# Here we use the gensim library to
# compute the cosine similarity of the paraphrases
# in order to choose the best ones.
#==================================
# Thanks to Hubert Naets for the help and advices
# Source : https://zenodo.org/record/162792#.XS_BkxgyWuU
# Thanks to
# https://www.machinelearningplus.com/nlp/gensim-tutorial/
# https://radimrehurek.com/gensim/similarities/docsim.html
# #gensim.similarities.docsim.Similarity
# https://www.machinelearningplus.com/nlp/cosine-similarity/
#==================================
import nltk
import time
from gensim import corpora,models, similarities
from preprocess1 import preprocess
from gensim.matutils import softcossim
from gensim import corpora
import gensim
import gensim.downloader as api
from gensim.test.utils import datapath

# This function takes in argument the sentence and its
# paraphrases. It loads Word2Vec model to compute the cosine
# similarity between the sentences and the paraphrases.
def evaluator(phrase, paraphrases):


  # We load the model with the "mmap" option, less fast but our RAM
  # can handle it
  w2v = models.Word2Vec.load("../trained_models/frwiki.gensim",mmap
    ='r')

  # We need to preprocess inputs (tokenization, remove punctuation,
  # lower case,...) because we'll create some vectors
  text1 = preprocess(phrase)



  # We create the first vector of the original sentence
  vec1 = []
  for token in text1:
    if token in w2v.wv.vocab:
      vec1.append(token)

  # If the vector is empty, we return an empty list
  if(len(vec1)==0):
    return False
```

```
54
55  # else, we compute the similarity of the sentences
56  else:
57    dictionnaire = {}
58    for paraphrase in paraphrases:
59
60      # we preprocess the current paraphrase
61      text2= preprocess(paraphrase)
62
63      vec2 = []
64
65        # We build the vector of the paraphrase
66      for token in text2:
67        if token in w2v.wv.vocab:
68          vec2.append(token)
69
70      # If the vector is not empty, we compute the similarity
71      if(len(vec2)!=0):
72
73        # We use the n_similarity function (compute the cosine
74        # similarity score)
75        sim = w2v.wv.n_similarity(vec1,vec2)
76
77        #===================================
78        # Here we can modify the value in the condition
79        # to check if it changes something in the results
80        # If the computed score is bigger, we add the paraphrase
81        # to the dictionnary with key = similarity score
82        #===================================
83        if(sim > 0.80):
84          dictionnaire[sim] = paraphrase
85        else:
86          pass
87
88    # All scores are computed, we build an array
89    # with the keys of the dictionnary (similarity scores)
90    keys_array = dictionnaire.keys()
91    i = 0
92    sentences_array = []
93
94    # We iterate in the reverse order and we save only the 10
95    # best paraphrases (with best scores of similarity)
96    # Only 10 because we do not want to overfit our model with
97    # too many similar sentences
98    for key in sorted(keys_array, reverse=True):
99      if(i == 10):
100       return sentences_array
101     else:
102       sentences_array.append(dictionnaire[key])
103     i = i +1
104
105   # We return these best paraphrases
```

```
106     return sentences_array
107
108
109 if __name__ == '__main__':
110   start_time = time.time()
111   phrase = "Je veux manger une pomme"
112   paraphrases = "Je veux acheter une pomme"
113   print(evaluator(phrase, paraphrases))
114
115   print("%s min" % ((time.time() - start_time)/60))
```

F.2 ReSyf

F.2.1 MAIN.py

Listing F.5 – MAIN.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  #==================================
4  # Author: Jean Cheramy
5  # Date : August 2019
6  #==================================
7  # We have to give to this program the name of
8  # the nlu file ('nlu.json').
9  #
10 #       python MAIN.py nlu.json
11 #
12 #==================================
13 # This is the main program to generate paraphrases
14 # It reads the nlu in JSON and call the generateResyf module
15 #==================================
16 import sys
17 import nltk
18 import json
19 import time
20 from generateResyf import generate_common_examples
21
22 #=================================================
23 # Step 1: read nlu file (JSON) and store data in
24 #     a hashtable in RAM
25 #=================================================
26
27 # We compute the time of the process (for information)
28 start_time = time.time()
29
30 # We instantiate a dictionnary and the filename variable
31 dictionnaire = {}
32 filename = str(sys.argv[1])
33
34 # We open the nlu and we load the data. We save the first intent
```

```
# and we instantiate an array (liste in python)
with open(filename, 'r+', encoding='utf-8') as json_file:
  data = json.load(json_file)
  intent1 = data["rasa_nlu_data"]["common_examples"][0]["intent"]
  array = []

  # for each sentence in the data, we check if the current intent
  # is the same as the saved intent.
  for example in data["rasa_nlu_data"]["common_examples"]:
    intent = example["intent"]
    text = example["text"]

    # If it is the same intent, we tokenize the sentence
    # to check the len (nb of tokens) of the sentence.
    # If there is just one word, we pass (not relevant to
    # generate sentence from only one word). If the len is
    # bigger, we add the sentence into an array.
    if(intent==intent1):
      tokens = nltk.word_tokenize(text)
      if(len(tokens)==1):
        pass
      else:
        array.append(text)

    # If the intent is not the same, that means that
    # we already read all the sentences of the saved intent
    # so we save the array of sentences in a dictionnary with
    # the key = intent.
    # We change the value of the saved intent, we clear the array
    # and we save the sentence in the array
    else:
      dictionnaire[intent1]=array
      array=[]
      intent1 = intent
      array.append(text)

# We close the json file
json_file.close()

#=================================================
# Step 2: call generateResyf until nlu is totally read
#=================================================

# We build an array of dictionnary keys
key_array = dictionnaire.keys()

# For each intent in the array of keys
# and for each sentence in the "key partition"
# of the dictionnary, we call the Generate function
for intent in key_array:
  for sentence in dictionnaire[intent]:
    generate_common_examples(intent, sentence, filename)
```

```
87
88 # We print the time needed for the process
89 print("%s min" % ((time.time() - start_time)/60))
```

F.2.2 generateResyf.py

Listing F.6 – generateResyf.py

```
1  # !/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  # ========================================
4  # Author: Jean Cheramy and Alexandre Robin
5  # mail: cheramy.jean.emile@gmail.com
6  # mail: robin.alexandre@icloud.com
7  # Date: July and August 2018; V2 August 2019
8  # For: Chatbot PLus SPRL, Bruxelles.
9  # Modified in August 2019 for the Master thesis
10 # ========================================
11 # MAIN program to create examples from nlu
12 # ========================================
13 # This function is called in MAIN.
14 # 3 arguments are given: intent, sentence and nlu filename
15 # ========================================
16 import time
17 import json
18 from makeSentences import make_sentences
19 from getSynonymsResyf import get_synonyms_resyf
20 from evaluate import evaluator
21 from preprocess import preprocess
22 import re
23 import nltk
24 from nltk.tokenize import word_tokenize, sent_tokenize
25
26 def generate_common_examples(intent, text, filename):
27   #=================================================
28   # Step 1: we get synonyms from the original sentence
29   #     ex : "je veux manger"
30   #     ex: ["je", "on"],["veux"],["manger", "diner", "gouter"]]
31   #=================================================
32
33   tokenized = nltk.word_tokenize(text)
34
35   if(len(tokenized)<2):
36     pass
37   else:
38     matrix = get_synonyms_resyf(text)
39
40     #=================================================
41     # Step 2: We make similar sentences, return a list of strings
42     # ex: ["je veux diner", "on veux gouter", "je veux manger",...]
43     #=================================================
44     common_examples = make_sentences(matrix)
45
```

```
46
47      #=================================================
48      # Step 3: We evaluate paraphrases and we only keep the
49      # best ones. The original sentence is "text" and its
50      # paraphrases are in common_examples.
51      #=================================================
52
53      if(len(common_examples)!=0):
54        good_examples = evaluator(text, common_examples)
55
56        #=================================================
57      # Step 4: We save results and we only add sentences that
58      # are not yet in the nlu
59      #=================================================
60      # We open and we read the nlu json_file and we write the
61      # results in a pretty print
62      # in UTF8. The output is like this (fictive example):
63      #    {
64      #      "intent": "reserver_voiture",
65      #      "text": "je veux acheter cette voiture"
66      #    },
67        #=================================================
68
69        # If no good examples was found by the evaluator, pass
70        if(len(good_examples) == 0):
71          pass
72
73        # else, we open the nlu file, we load the data
74        else:
75          with open(filename, 'r+', encoding='utf-8') as json_file:
76            data = json.load(json_file)
77
78            # For each good sentence, we instantiate
79            # a boolean to False
80            # and we preprocess the sentence
81            # (tokenized, remove punctuation)
82            for sentence in good_examples:
83              indice = False
84              tokenized = preprocess(sentence)
85
86              # we iterate on the nlu sentences and we also
87              # preprocess them to compare the tokenized
88              # good-example with the nlu sentences. The
89              # preprocess step would not be necessary if
90              # there were not so many sentences composed of
91              # the same words but still different
92              # (punctuation, spaces,...)
93              for item in data["rasa_nlu_data"]["common_examples"]:
94                nlu_sentence = preprocess(item["text"])
95
96                # If tokenized sentences are equal, we give
97                # the True value to the boolean
```

```
98                if(tokenized == nlu_sentence):
99                  indice = True
100               else:
101                 pass
102
103             # If the value is True, we pass.
104             # else, we add the sentence (original one,
105             # not the tokenized) into the nlu file
106             if(indice==True):
107               pass
108             else:
109               data["rasa_nlu_data"]["common_examples"].insert(0, {"
    intent": intent, "text": sentence})
110               with open(filename, 'w', encoding='utf-8') as
    json_file:
111                 json.dump(data, json_file, ensure_ascii=False,
    indent=2)
112
113 if __name__ == '__main__':
114   start_time = time.time()
115   intent = "conso"
116   exemple = "Je veux connaitre la consommation de cette voiture?"
117   filename = "nlu.json"
118   generate_common_examples(intent, exemple, filename)
119
120   print("%s min" % ((time.time() - start_time)/60))
```

F.2.3 getSynonymsResyf.py

Listing F.7 – getSynonymsResyf.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  #==================================
4  # Author: Jean Cheramy
5  # Date : August 2019
6  #==================================
7  # This function searchs synonyms of the words
8  # of the sentence. It builds a matrix (array of arrays)
9  # and return it to the generate function.
10 #==================================
11 # It needs the Resyf API
12 # https://gitlab.com/Cental-FR/resyf-package
13 # ReSyf is the dictionnary of synonyms
14 #==================================
15 import time
16 import ReSyf
17 import nltk
18 from nltk.tag import StanfordPOSTagger
19 import spacy
20 import os
21
22
```

```
#==============================
# Function 1 : get_syns_resyf
#==============================
# Get synonyms of words of a sentence
#==============================
def get_synonyms_resyf(sentence):
  sentence = str(sentence)

  # Firstly we load the resyf ressource and we
  # tokenize the sentence
  lexicalRes = ReSyf.load('/home/gandalf/MEMOIRE/IMPLEMENTATION/
    GENERATION/resyf-package-master')
  wordsList = nltk.word_tokenize(sentence)

  #==============================
  # Step1 : pos tagging of the sentence
  #==============================
  pos = stanford_tag(wordsList)
  resyf_tags = []
  final_matrix = []

  #==============================
  # Step 2: converting pos tags to resyf tags
  #     because they do not use the same typology
  #==============================
  for x in pos:
    tag = convert_pos(x[1])
    resyf_tags.append(tag)

  #==============================
  # Step 3: search synonyms in resyf with pos tag and
  #     lemmatized words
  #==============================
  j = 0
  sen_len = len(wordsList)

  while(j<sen_len):

    # we instantiate term and postag variables
    # these variables are linked (term = être; postag = VER)
    term = wordsList[j]
    postag = resyf_tags[j]

    j = j+1

    # This list will contain all the synonyms of a word
    # ex: manger -> syns = ['manger, bouffer', 'gouter', 'diner']
    syns = []

    # if postag is None (if the word is not a
    # verb, a name, an adjective or an adverb),
    # we just add the term to the array
```

```
74      if(postag == None):
75        syns.append(term)
76
77      # else, if the word is a name or an adjective, we lemmatize it.
78      # we do that because we want to find synonyms of words
79      # even if there are slight spelling differences
80      # voitures -> voiture -> resyf ['voiture',
81      # 'bagnole', 'auto',...]
82      # Indeed, resyf does not have "voitures" as entry
83      # And, linguistically, this approach is viable because there is
84      # a lot of spelling errors and transformation in user messages.
85      # So it fits to our goal of diversity.
86      else:
87        if(postag ==  "N" or postag == "ADJ"):
88          term = lemmatize(term)
89
90        # We use ReSyf functions (see gitlab), it gives us an object
91        # and we iterate on it.
92        result = ReSyf.get_synonyms(lexicalRes, term, postag)
93        key = result.keys()
94        for elem in key:
95          lenght = len(result[elem]['synonyms'])
96          i = 0
97
98          # We iterate on synonyms of a word
99          while(i<lenght):
100           mot = result[elem]['synonyms'][i].feat['word']
101
102           # We tried to use synonyms depending on the "rank", the
103           # gradued value (of difficulty) of the word but
104           # that didn't help a lot so we comment it.
105           # rang = int(result[elem]['synonyms'][i].feat['rank'])
106           # if rang < or > ...
107
108           # If the current word is not in the list, we add it
109           if(mot not in syns):
110             syns.append(mot)
111           else:
112             pass
113           i=i+1
114
115     # If the list is empty, we just add the original term
116     if not syns:
117       syns.append(term)
118
119     # We add the syns list to the matrix
120     final_matrix.append(syns)
121
122   # We return the matrix to the generate function
123   return final_matrix
124
125 #==============================
```

```
126 # Function 2: convert pos
127 #==============================
128 # Here we convert pos of Standford to
129 # resyf pos.
130 #==============================
131 # Thanks here for the list of pos of Treebank
132 #http://www.llf.cnrs.fr/Gens/Abeille/French-Treebank-fr.php
133 #==============================
134 def convert_pos(pos):
135   if ("N" or "NC" or "NP") == pos:
136     pos = "NC"
137   elif pos == "VINF" :
138     pos = "VER"
139   elif ("ADJ" or "A") == pos:
140     pos = "ADJ"
141   elif "ADV" == pos:
142     pos = "ADV"
143   else:
144     pos = None
145   return pos
146
147 #==============================
148 # Function 3: get pos
149 #==============================
150 # Here we use the Standford tagger to get french tags of our
151 # sentences. This tagger seemed to be the best for the french
152 # so we use it.
153 #==============================
154 # Thanks here for the help
155 # https://stackoverflow.com/questions/44468300/how-to-pos-tag-a-
      french-sentence
156 #==============================
157 def stanford_tag(sentence):
158   jar = '/home/gandalf/MEMOIRE/IMPLEMENTATION/GENERATION/stanford-
        postagger-full-2018-10-16/stanford-postagger-3.9.2.jar'
159   model = '/home/gandalf/MEMOIRE/IMPLEMENTATION/GENERATION/stanford
        -postagger-full-2018-10-16/models/french.tagger'
160   java_path = "C:/Program Files/Java/jdk1.8.0_121/bin/java.exe"
161   os.environ['JAVAHOME'] = java_path
162   pos_tagger = StanfordPOSTagger(model, jar, encoding='utf8' )
163   res = pos_tagger.tag(sentence)
164   return res
165
166 #==============================
167 # Function 4: lemmatize
168 #==============================
169 # Here we lemmatize words with spacy
170 #==============================
171 # Thanks to ksayeh
172 # https://stackoverflow.com/questions/13131139/lemmatize-french-
      text
173 #==============================
```

```
def lemmatize(word):
  nlp = spacy.load('fr')
  doc = nlp(word)
  for token in doc:
      return(token.lemma_)

if __name__ == '__main__':
  start_time = time.time()
  phrase = "Je veux acheter un vélo"
  print(get_synonyms_resyf(phrase))
  print("%s min" % ((time.time() - start_time)/60))
```

F.2.4 makeSentences.py

Listing F.8 – makeSentences.py

```
# !/usr/bin/env python
# -*- coding: utf-8 -*-
# ========================================
# Author: Alexandre Robin and Jean Cheramy
# mail: cheramy.jean.emile@gmail.com
# mail: robin.alexandre@icloud.com
# Date: July and August 2018
# V2: August 2019
# For: Chatbot PLus SPRL, Bruxelles.
# ========================================
# This package takes a matrix of synonyms in argument and
# gives sentences in output. For example, for the matrix:
# [["je"],["veux"],["manger", "diner", "gouter"]],
# the output will give:
# Je veux manger/ Je veux diner/ je veux gouter
# ========================================
# Thanks to stackoverflow:
# https://stackoverflow.com/questions/48819566/generate
# -all-possible-sentences-from-n-lists-using-python
# ========================================
import time
import itertools;

#The function takes the matrix in argument
def make_sentences(arrayOfarray):
    allSynonyms = arrayOfarray;
    sentences = [];
    sentence = [];
    sentencesFinal = [];

    # This function will split a sub array and construct sentences
    # with the itertools package.
    def split_sub_table():
        i = 0;
        temporary = '';
        sentences.append(allSynonyms[0]);
        allSynonyms.remove(allSynonyms[0]);
```

```
38          while i < len(allSynonyms):
39              if i == len(allSynonyms) - 1:
40                  sentence.append(list(itertools.product(sentences[i
    ], allSynonyms[i])));
41              else:
42                  temp = list(itertools.product(sentences[i],
    allSynonyms[i]));
43                  sentences.append(temp);
44              i += 1;
45          tab = sentence[0];
46          for table in tab:
47              sentencesFinal.append(clean(table));
48
49      # This function clean the tuple (because there are () in
50      # the string and we don't need them).
51      def clean(tab):
52          string = '';
53          for tupl in tab:
54              if type(tupl) is str:
55                  string = string + tupl + " "
56              else:
57                  string = string + clean(tupl)
58          return string;
59      split_sub_table();
60      return sentencesFinal;
61
62  if __name__ == '__main__':
63      start_time = time.time()
64      matrix = [["je"]["veux"]["manger", "diner", "gouter"]]
65      print(make_sentences(matrix))
66      print("%s min" % ((time.time() - start_time)/60))
```

F.2.5 evaluate.py

Listing F.9 – evaluate.py

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3  #==================================
4  # Author: Jean Cheramy
5  # Date : August 2019
6  #==================================
7  # Here we use the gensim library to
8  # compute the cosine similarity of the paraphrases
9  # in order to choose the best ones.
10 #==================================
11 # Thanks to Hubert Naets for the help and advices
12 # Source : https://zenodo.org/record/162792#.XS_BkxgyWuU
13 # https://www.machinelearningplus.com/nlp/gensim-tutorial/
14 # https://radimrehurek.com/gensim/similarities/docsim.html
15 # #gensim.similarities.docsim.Similarity
16 # https://www.machinelearningplus.com/nlp/cosine-similarity/
17 #==================================
```

```
import nltk
import time
from gensim import corpora,models, similarities
from preprocess import preprocess
from gensim.matutils import softcossim
from gensim import corpora
import gensim
import gensim.downloader as api
from gensim.test.utils import datapath

# This function takes in argument the sentence and its
# paraphrases. It loads Word2Vec model to compute the cosine
# similarity between the sentences and the paraphrases.
def evaluator(phrase, paraphrases):

  # We instantiate the results list
  sentences_array = []

  # If there is no paraphrases, we return an empty list
  if(len(paraphrases)==0):
    return sentences_array

  # We load the model with the "mmap" option, less fast but our RAM
  # can handle it
  w2v = models.Word2Vec.load("../trained_models/frwiki.gensim",mmap
    ='r')

  # We need to preprocess inputs (tokenization, remove
  # punctuation, lower case,...) because we'll create some vectors
  text1 = preprocess(phrase)

  # We create the first vector of the original sentence
  vec1 = []
  for token in text1:
    if token in w2v.wv.vocab:
      vec1.append(token)

  # If the vector is empty, we return an empty list
  if(len(vec1)==0):
    return sentences_array

  # else, we compute the similarity of the sentences
  else:
    # This dictionnary will be used to save results
    dictionnaire = {}

    # for each paraphrase, we preprocess it and build
    # a vector of tokens
    for paraphrase in paraphrases:
      text2= preprocess(paraphrase)
      vec2 = []
      for token in text2:
```

```
69          if token in w2v.wv.vocab:
70            vec2.append(token)
71
72        # If the vector is not empty, we compute the similarity
73        if(len(vec2)!=0):
74
75          # We use the n_similarity function (compute the cosine
76          # similarity score)
77          sim = w2v.wv.n_similarity(vec1,vec2)
78
79          #==================================
80          # Here we can modify the value in the condition
81          # to check if it changes something in the results
82          # If the computed score is bigger, we add the paraphrase
83          # to the dictionnary with key = similarity score
84          #==================================
85          if(sim > 0.80):
86            dictionnaire[sim] = paraphrase
87          else:
88            pass
89
90      # All scores are computed, we build an array
91      # with the keys of the dictionnary (similarity scores)
92      keys_array = dictionnaire.keys()
93      i = 0
94
95      # We iterate in the reverse order and we save only the 10
96      # best paraphrases (with best scores of similarity)
97      for key in sorted(keys_array, reverse=True):
98        if(i == 15):
99          return sentences_array
100       else:
101         sentences_array.append(dictionnaire[key])
102       i = i +1
103
104     # We return these best paraphrases to the generateResyf
        function
105     return sentences_array
106
107 if __name__ == '__main__':
108   start_time = time.time()
109   phrase = "Donne-moi les caractéristiques principales du véhicule"
110   paraphrases = ['Donne-moi les caractéristiques notable du vé
        hicule ', 'Je veux connaitre les caractéristiques de la voiture
        ']
111   evaluator(phrase, paraphrases)
112   print("%s min" % ((time.time() - start_time)/60))
```

F.2.6 preprocess.py

Listing F.10 – preprocess.py

```
1 #!/usr/bin/env python
```

```
# -*- coding: utf-8 -*-
#==================================
# Author: Jean Cheramy
# Date : August 2019
#==================================
# This function is useful to preprocess
# sentences:
# tokenize, lower case, remove punctuation
# ans stop-words
#==================================
# thanks to:
# https://www.geeksforgeeks.org/removing-stop-words-nltk-python/
# https://stackoverflow.com/questions/265960/best-way-to-strip-
# punctuation-from-a-string
#==================================
import nltk
import re
from nltk.corpus import stopwords
from nltk.tokenize import word_tokenize, sent_tokenize
import time

def preprocess(sentence):

  # create a stop-words variable for french
  stop_words = set(stopwords.words('french'))

  # strip punctuation
  sentence = re.sub(r'[^\w\s]','',sentence)

  # we tokenize the sentence
  tokenized = sent_tokenize(sentence)
  wordsList = []

  # For each token we check if it is a stop word or not
  # if it is not, we add it to the wordlist
  for token in tokenized:
      token = token.lower()
      wordsList = nltk.word_tokenize(token)
      wordsList = [w for w in wordsList if not w in stop_words]

  # we return the tokenized sentence in lower case, without
    punctuation and stop-words
  return wordsList

if __name__ == '__main__':
  start_time = time.time()
  phrase = "Je veux! manger un'e' Pomme"
  print(preprocess(phrase))
  print("%s min" % ((time.time() - start_time)/60))
```

UNIVERSITÉ CATHOLIQUE DE LOUVAIN
Faculté de philosophie, arts et lettres
Place Blaise Pascal, 1 bte L3.03.11, 1348 Louvain-la-Neuve, Belgique | **www.uclouvain.be/fial**