# Comprendre et développer un chatbot

Paul Regus

© 2018

## Les ChatBots, qu'est-ce que c'est ?

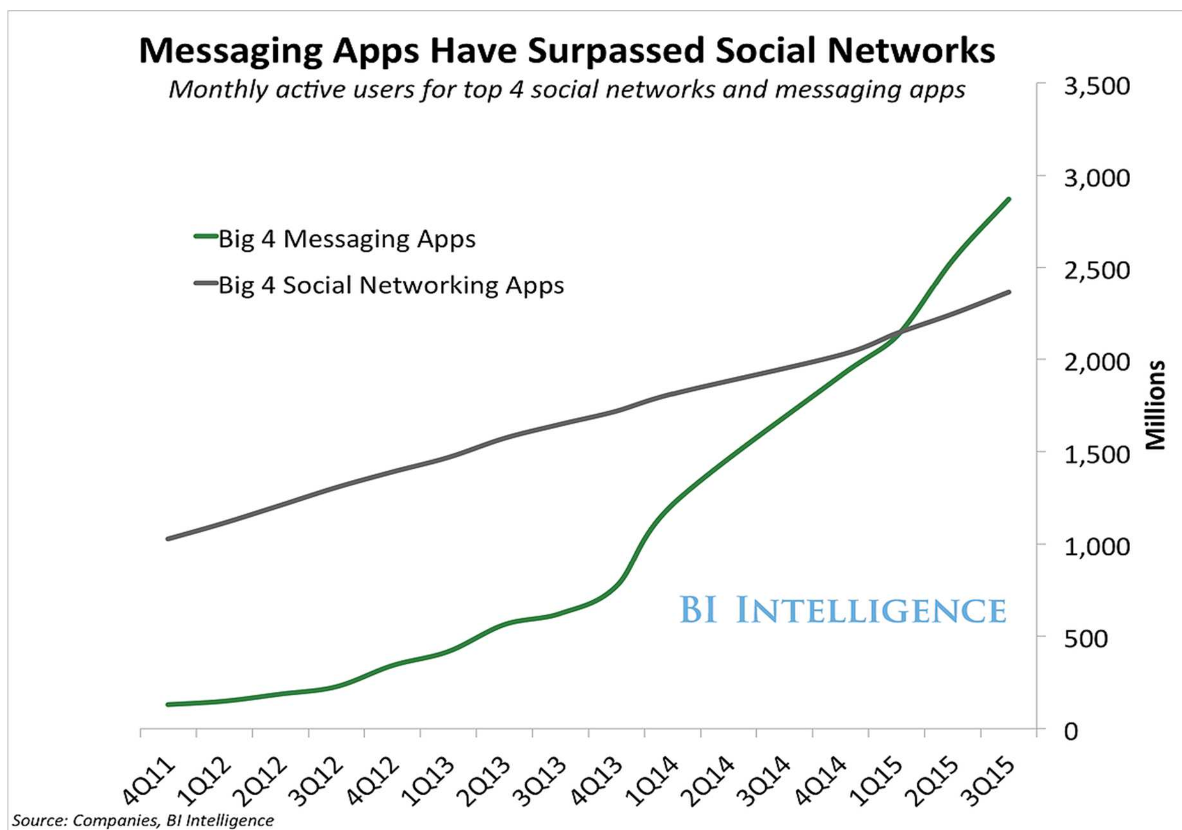
Un ChatBot est un logiciel pouvant communiquer avec quelqu'un par le biais d'un langage naturel. Ils peuvent servir à répondre à nos questions, sur le temps, les informations, notre agenda, mais aussi nous commander des produits ou des services, comme un taxi, un billet, des vêtements etc…

Longtemps vibrante, la scène des agents conversationnels n'a connu un véritable essor que récemment début 2016. Il faut dire que cette volonté de parler avec des machines remonte à loin, depuis le père de l'informatique Alan Turing et son célèbre test.

Pourquoi un tel essor aujourd'hui ? Tant que la poussière soulevée par cette explosion n'est pas retombée, il est difficile de répondre à cette question. Néanmoins, en assistant aux meet-ups, en rencontrant les créateurs de start-ups de ChatBots, en suivant les explications des GAFA (Google, Amazon, Facebook, Apple) concernant leurs investissements dans ce domaine, plusieurs grands types de réponses se dessinent et se répètent :

**> Aller là où les consommateurs sont, et sont accessibles.**

A la mi-2015, le nombre d'utilisateurs sur les 4 applications de messageries les plus utilisées dépassait le nombre d'utilisateurs des 4 applications de réseaux sociaux les plus importantes.




Pourtant le nombre d'utilisations commerciales et marketing des applications de messageries est bien plus réduit que sur les réseaux sociaux. Cette asymétrie d'utilisation, alors même que les opportunités sont équivalentes, a amené un certain nombre d'acteurs à se positionner sur ce marché. Pour y accéder, il

fallait pouvoir entrer dans un format similaire à ce qui est offert actuellement et donc passer par des ChatBots.

- **Un spin-off des applications mobiles.**

La moitié des utilisateurs n'ont téléchargé presqu'aucune app en 2016. Le marché des Apps mobiles est devenu mature, et le calcul de ROI (Return On Interest) des applications mobiles est plus facile à faire. Une analyse fréquente de ce chiffre est que les utilisateurs ont déjà trop d'applications dans leurs smartphones et ne sont pas vraiment prêts à en installer d'autres facilement.

Les ChatBots sont parfois considérés comme une version cloud des applications mobiles. Le bot Uber déployé sur Messenger par exemple permet de faire essentiellement la même chose que l'application Uber, commander un chauffeur d'un point A à un point B, mais ne nécessite pas de télécharger l'application. En ce sens les ChatBots sont un peu aux applications ce que les sites web sont aux programmes locaux sur votre machine. Reste que le changement de paradigme vers une interface textuelle constitue un véritable enjeu d'UX (Experience Utilisateur) pour lequel aucune solution standard n'existe pour l'instant. Ce changement de paradigme est aussi une chance car il apporte une véritable facilité d'interaction avec le service.

**> Des coûts de développements (à priori) plus réduits.**

Une grande partie des coûts induits par le développement d'une application mobile est directement reliée à l'interface utilisateur plus qu'au développement des fonctionnalités attendues par l'App. Or, les applications de messageries ont toutes ou presque désormais développé des modules permettant de s'interfacer rapidement à leurs services.

Résultat : pour créer un ChatBot, plus besoin de développer un front. Il suffit de développer directement le ChatBot. Certaines plateformes (comme BotFuel ou API.ai) permettent déjà de déployer un ChatBot commun pour toutes les différentes interfaces, et d'autres vous permettent de créer des ChatBots basiques facilement comme ChatFuel, ou FacebookMessenger , parfois sans même avoir besoin de passer par du code (même si les possibilités restent encore limitées encore aujourd'hui en 2018).

Reste que les ChatBots constituent un problème fondamental de design et un challenge pour l'UX, et que ce challenge peut avoir un coût important et un impact sur le ROI du projet.

## Et l'intelligence artificielle dans tout ça ?

Pour bien comprendre comment l'intelligence artificielle permet d'étendre massivement le champ des possibles pour les ChatBots, il faut comprendre leur principe de fonctionnement.

Globalement, les services clients ont deux fonctions : récupérer des informations des clients et les guider en menant avec eux une conversation.

Dans la plupart des démarchages clients et quelques services clients, les opérateurs soit naviguent au sein d'arbres de conversations qui « scriptent » l'échange, soit se basent sur des phrases pré-construites qui cadrent les éléments communiqués. Ce script ou cette base permet une standardisation, et une facilité de formation des opérateurs.

Un ChatBot permet la navigation automatisée parmi l'ensemble de ces réponses possibles, ainsi que la récupération des informations contenues dans la conversation. Certes, il existe des ChatBots qui permettent de générer leurs propres phrases (on parle alors de Natural Language Generation). Mais ces ChatBots restent à l'heure actuelle un défi du domaine de la recherche plus que de l'industrialisation.

Là où un opérateur utilise son jugement pour savoir vers quelle branche s'orienter, un ChatBot utilisera un ensemble de règles pour naviguer dans l'arbre de conversation. Dans le cas où le nom et le prénom ont été demandés au client, c'est le jugement de l'opérateur qui permettra de savoir si le client a refusé de donner cette information, et donc qu'il faut justifier cette demande

dans la réponse, ou si le message transmis contient bien un nom et un prénom et qu'il faut dans la réponse demander les prochaines informations. Dans le cas d'un ChatBot, ce jugement humain sera remplacé par un ensemble de règles permettant d'analyser les réponses des utilisateurs en fonction du contexte. On retrouve la même équivalence entre le jugement d'un opérateur et un ChatBot dans la récupération des informations clients.

C'est dans l'établissement de ces règles que le machine learning (ou apprentissage automatique) peut assister le travail de développement de ChatBot et permettre l'obtention de bots plus intelligents et plus agréables à qui parler pour les clients. Le machine learning peut intervenir de différentes manières, à la fois pour déterminer la bonne attitude et la bonne réponse pour le client, et pour déterminer quelles sont les éléments d'information à récupérer dans les messages des clients, et à quels champs ils s'appliquent (dans ce cas on parle de Natural Langage Understanding ou NLU).

Le machine learning peut être utilisé comme une première couche d'analyse pour interpréter les messages clients et permettre une programmation de concepts complexes à décrire finement.

Par exemple, pour mener la conversation et savoir quoi répondre, on pourrait utiliser une représentation word2vec ou doc2vec (pré-entrainé) pour tester la présence de concepts dans le message visiteur et baser les règles établies par les développeurs sur ces concepts. Sinon, on pourrait directement imaginer un modèle de machine learning prenant un message visiteur en entrée et retournant le message à répondre.

On retrouve la même dualité d'approche dans l'extraction des informations clients. On peut se servir de "pos-tagger" pré-entrainés (ou étiqueteur morphosyntaxique) qui caractérisent la nature du mot (nom adjectif verbe etc…) et permettent à des développeurs de se guider sur cette nature pour établir les règles à appliquer. Par exemple, un pos-tagger permet de coder un algorithme de récupération de prénom en testant la nature des mots de la phrase. Sinon, cette information peut directement être rendue par un modèle de machine learning qui a été entrainé sur un historique de messages clients traités et les informations extraites correspondantes.

Cette utilisation du machine learning permet d’inclure une certaine forme d’intelligence qui facilite grandement la vie des utilisateurs et des développeurs. Sans elle, les développeurs devraient lister l’intégralité des synonymes lorsqu’ils cherchent un mot dans une phrase. Le client devrait choisir des formulations de phrases spécifiques et l’expérience qu’il aurait du service reviendrait à cliquer sur les phrases qu’on lui propose et qui lui conviennent le mieux. Sans cette utilisation du machine learning, les développeurs devraient lister une à une toutes les manières de donner son nom, et tous les noms et prénoms possibles pour pouvoir les récupérer.

## Comment créer un ChatBot intelligent?

Pour créer un ChatBot intelligent, c'est-à-dire un ChatBot incluant du machine learning, il faut des données sur lesquelles apprendre ou un modèle pré-entrainé. Dans le cas de pos-tagger ou de word2vec, ces modèles peuvent être pré-entrainés sur de larges corpus. De tels modèles pré-entrainés peuvent être trouvés pour le Français sur l’excellent site de Jean-Philippe Fauconnier.

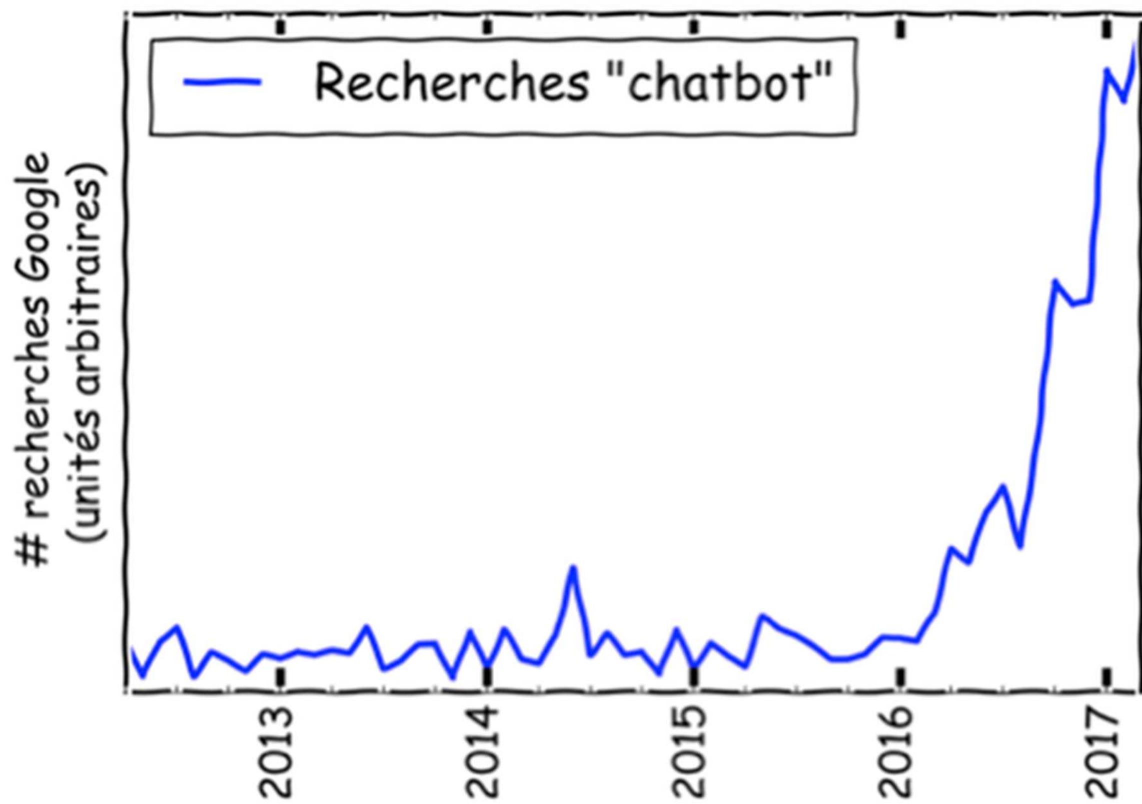
Dans le cas contraire où on attend une application du machine learning spécifique au problème que le ChatBot doit résoudre, des données spécifiques sur ce traitement doivent être recueillies. La qualité du ChatBot dépendra directement de la qualité de ces données. Aussi, pour obtenir ces données, il faut donc qu’il existe un service de qualité, même temporaire. Pour cela, un arbre de conversation permettant de définir les parcours clients classiques doit être développé, et les opérateurs doivent autant que possible rester au sein de cet arbre en répondant au client. Ce processus sera plus facile à appliquer pour le démarchage client que pour les services clients, puisque les arbres de conversations y sont plus largement utilisés. Dans le cas où aucun arbre n’a déjà été développé, la meilleure pratique consiste à définir cet arbre de façon AGILE en modifiant et complétant un premier arbre de conversation par rapport au retour d’expérience de son utilisation par les conseillers. Une fois cet arbre défini et complet, les opérateurs seront ensuite en charge de naviguer au sein

de cet arbre de conversation pour guider les clients. L’historique de traitement des clients constituera ensuite une base de données qui pourra permettre le développement de modèles de machine learning. Pour faciliter l’apprentissage, il faudra avoir un processus le plus stable possible et que toutes ses modifications soient enregistrées, documentées et mises en relation avec les processus qui existaient avant ces changements.

Cette base de données nous permettra ensuite de développer les modèles de machine learning spécifiques aux problèmes que nous cherchons à résoudre...

# Développer un Chatbot

Comment un programme peut-il apprendre par lui-même à répondre à des questions qu’on ne lui a jamais posées ? Quels algorithmes se cachent derrière les chatbots ?




Introduction

La création d'un chatbot à même d'avoir une conversation cohérente dans n'importe quel contexte est le Graal de l'intelligence artificielle. Ce chatbot générique fait l'objet d'un effort de recherche intense, soutenu par les chercheurs des universités et grands groupes tels que Google, Facebook, Amazon, et quelques autres… Pour s'adapter à n'importe quel contexte ou interlocuteur, un tel agent conversationnel doit générer ses réponses mot par mot ; on parle de chatbot génératif. Le problème de cette approche est qu'elle est trop générale et les chatbots peinent à avoir une conversation cohérente ou même à produire des phrases syntaxiquement valides avec les modèles actuels.

Un problème plus abordable consiste à construire un chatbot adapté à un contexte donné, qui construit ses réponses à l'aide d'un ensemble de phrases qu'on lui aura donné par avance. Cette approche, que nous avons choisie, garantit des réponses grammaticalement correctes et simplifie la tâche d'apprentissage pour l'algorithme, car elle permet l'entraînement d'un modèle sur un échantillon d'entraînement plus petit que les immenses jeux de données nécessaires dans le cas d'un chatbot génératif.

Le modèle que l’on veut construire doit être à même de produire une réponse cohérente au dernier message de son interlocuteur, tout en prenant en compte l’historique de la conversation.

La formalisation du problème pose plusieurs questions :

● Quelles données sont nécessaires ? Comment représenter les conversations pour qu’un modèle statistique puisse les appréhender ?

● Quels modèles d’apprentissage automatique peuvent traiter ce problème efficacement ?

● Comment évaluer les performances du chatbot ? Autrement dit comment quantifier la plausibilité d’une réponse proposée par le chatbot et la cohérence de la conversation au vu des réponses du chatbot ?

## De la conversation au modèle statistique

Pour un ordinateur, une conversation n’est qu’une série de lettres sans structure. On doit d’abord trouver un moyen de simplifier le texte, puis de transformer les conversations en vecteurs tout en conservant la structure du langage, avant d’entraîner le modèle de génération de réponse; enfin, on doit choisir une métrique d’évaluation du modèle pour sélectionner automatiquement le plus performant.

### Pré-traitement

Nous avions 25 000 conversations à notre disposition pour l’entraînement du chatbot. C’est énorme … et très peu à la fois. En effet la plupart des quelques 20 000 mots différents qui apparaissent dans notre corpus de conversation n’apparaissent qu’une fois. Cette observation confirme une fois de plus la loi empirique de Zipf (cf. Figure 1), qui implique notamment que la plupart des

mots d'un texte ou corpus n'apparaissent qu'une ou deux fois. Le deuxième mot le plus présent, par exemple, est typiquement deux fois moins représenté du premier.

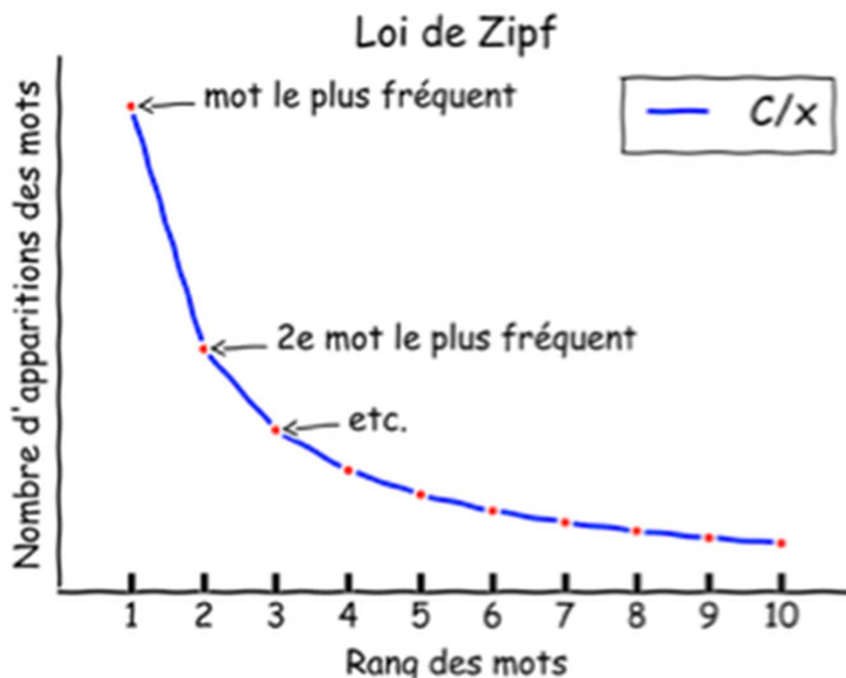



**FIGURE 1: LOI EMPIRIQUE DE ZIPF : LA FRÉQUENCE D'APPARITION DE MOTS SUIT UNE LOI INVERSE DU CLASSEMENT**

Comme notre modèle ne peut pas apprendre à répondre à des phrases composées de mots qu'il n'a presque jamais vu au cours de son entraînement, il faut l'aider un peu. C'est la phase de prétraitement du texte, qui est aussi nécessaire que riche.

Au cours de cette étape, on cherche à réduire la complexité du texte en appliquant certaines des opérations suivantes :

● Suppression des mots vides (et, le, de, ...) qui sont omniprésents et ne sont pas nécessaires à la compréhension du texte

● Remplacement de tous les mots rares (moins de 5 apparitions typiquement) à une balise ‘<OOV>’ (« Out Of Vocabulary »)

● Normalisation : c’est l’opération qui consiste à la conjugaison, les marques de nombre, de genre, … On distingue :

o La racinisation qui se base sur des dictionnaires : e.g. peuvent → pouvoir

o La lemmatisation qui applique des règles d’amputation des mots : maisons → maison

● Association de la fonction grammaticale aux mots –- c’est l’étiquetage morpho-syntaxique -- pour ne pas confondre le joint d’étanchéité et le fait de joindre quelqu’un par exemple

● Étiquetage de certains mots. Par exemple une séquence du type xxxx@xxx.com gagne à être estampillée par une balise <email>, autrement chaque adresse serait perçue comme un nouveau mot à ajouter au vocabulaire.

● Séparation de la phrase en unités lexicales. Cette étape consiste à transformer la chaîne de caractères qui correspond à une phrase en une liste de mots. Elle implique des choix sur la gestion des tirets, la suppression de la ponctuation, etc.

Il faut noter que ces opérations sont partiellement destructrices pour le texte, elles peuvent même le rendre inintelligible par endroit. Le choix des opérations de pré-traitement à appliquer doit donc résulter d’une inspection approfondie d’exemples et de l’impact sur les performances finales du

modèle. On remarque que l'ordre des opérations de prétraitement joue un rôle, et que des opérations préliminaires de nettoyage du texte sont souvent nécessaires pour retirer des balises HTML, remplacer des acronymes spécifiques, traduire certains mots, verbaliser des émoticônes, …

En Python, la bibliothèque NLTK possède les corpus et fonctionnalités nécessaires à toutes ces tâches, même en français ! Voici un exemple de transformation typique du texte au cours du prétraitement, sur le paragraphe que vous venez de lire (« Il faut noter que … ») :

```
faut not oper partiel destructric pour text elle peuvent mem rendr inintelligibl endroit choix oper pre trait a appliqu doit donc result dun inspect approfond dexempl limpact le perform final model lordr oper pre trait jou rol oper preliminair nettoyag text souvent necessair pour retir balis html remplac acronym specif traduir certain mot verbalis emoticon
```

<u>Un exemple plus simple, mais détaillé :</u>

Exemple de pré-traitement d'une phrase :

initial: `Bonjour.Il est 15h mon numéro de tel c'est le 06 00 00 00 00.`

tokenization: `['bonjour', '.', 'il', 'est', '15h', 'mon', 'numero', 'de', 'tel', 'c', "'", 'est', 'le', '06', '00', '00', '00', '00', '.']`

stopwords: `['bonjour', '.', '15h', 'numero', 'tel', "'", '06', '00', '00', '00', '00', '.']`

tagging: `bonjour . _HOUR_ numero tel ' _PHONENUMBER_ .`

## Vectorisation

Une fois le texte préparé, il est nécessaire d’associer un vecteur à chaque conversation dont on veut déterminer la réponse. Il existe plusieurs méthodes de vectorisation de textes. Par ordre de complexité croissante on peut mentionner le “sac de mots”, la TF-IDF, GloVe, word2vec et paragraph2vec.

Figure 2 : word2vec. Cette représentation capture une partie des relations sémantiques.

Il faut noter que ces représentations ne tiennent pas compte de l'ordre des mots. En effet elles produisent les mêmes vecteurs pour deux phrases qui ont les mêmes mots, même dans un ordre différent. C'est un problème auquel on peut palier en utilisant des modèles séquentiels tels que des réseaux de neurones récurrents (voir notamment seq2seq), qui seront évoqués plus loin.

Une approche plus simple que nous avons régulièrement exploitée pour résoudre ce problème consiste à introduire les « n-grammes ». Dans cette approche on ne considère plus seulement la présence individuelle des mots pour déterminer la représentation d'un message, mais également les paires de mots, triplets, etc. On obtient alors un lexique très important ; par exemple mille mots différents peuvent générer jusqu'à un million de paires, et un milliard de triplets ! Cette source de complexité est modérée en ne considérant que les combinaisons les plus fréquentes : dans notre cas, les 300 n-grammes les plus fréquents étaient suffisants pour obtenir les meilleurs résultats.

L'étape de vectorisation est plus compliquée dans le cas de la conversation, car il ne faut pas seulement choisir la méthode de représentation des phrases de chaque interlocuteur mais la manière dont on représente l'ensemble de la conversation. Pour cela nous avons utilisé essentiellement la représentation TF-IDF des messages des visiteurs, tandis que pour représenter les phrases des agents nous avons opté pour une approche différente.

## L'astuce simplificatrice : segmentation des phrases de l'opérateur

Le chatbot doit reproduire les phrases formulées par les opérateurs du chat. Mais ces phrases sont très diverses, surtout au vu de notre maigre ensemble de 25 000 conversations. Pour constituer un ensemble de phrases cohérentes nous avons segmenté les réponses des opérateurs pour dégager quelques phrases-types. Dans le cas où les opérateurs suivent un arbre de conversation – et c'était bien notre cas – l'historique de données est naturellement propice à cette segmentation.

Cette étape nous a permis de constituer une base d'une centaine de phrases-type. Le problème se réduit alors à une "simple" classification, où chaque conversation est représentée par un vecteur, et chaque réponse opérateur correspond à une classe.

## Modélisation

Nous avons testé plusieurs modèles pour prédire la phrase-type à répondre à un message de visiteur; voici une présentation des modèles les plus pertinents.

### Un modèle d'extraction de conversations similaires

Pour chaque prédiction, ce modèle simple recherche le message visiteur le plus proche dans le corpus d'entraînement, et donne la réponse du corpus associée [3]. Explicitement :

- On détermine la représentation TF-IDF à partir des phrases « visiteurs » du corpus d'entraînement

- Pour chaque nouvelle phrase de visiteur dont on veut prédire la réponse, on détermine sa représentation TF-IDF

- On calcule la similarité cosinus de cette phrase avec chaque phrase « visiteur » du corpus d'entraînement

- On prédit la réponse qui a été donnée à la phrase la plus similaire dans le corpus d'entraînement.

Ce modèle est en réalité équivalent à un modèle du plus proche voisin pour la distance cosinus. Il donne des résultats étonnamment proches de notre meilleur modèle au vu de sa simplicité (~50% d'exactitude, voir la discussion du prochain paragraphe).

## Une régression logistique

Pour le problème de classification que nous avons formulé, il existe une multitude modèles. Nous avons obtenu les meilleures performances avec une simple régression logistique non régularisée, ce que suggère une relation linéaire entre les variables que nous avons créées.

Il faut noter que les performances de ce modèle dépendent de manière cruciale du découpage en « n-grammes », dans la mesure où la régression logistique ne tient compte ni de l'ordre de mots ni de même de leur co-occurrence !

Nous avons entraîné une régression logistique sur la représentation TF-IDF de dimension 300 qui mêle unigrammes, bigrammes et trigrammes. Cette régression nous permet d'atteindre une exactitude de l'ordre de 65%, dont la valeur est discutée ci-après.

## Un réseau de neurones récurrent (LSTM)

Dans le domaine du traitement automatique du langage naturel, les modèles rois sont souvent les réseaux de neurones récurrents. Ceux-ci, en tant que réseaux de neurones, sont à même de déceler des interactions complexes entre les mots. Mais ils sont de plus capables de traiter des phrases de longueur variable, tout en tenant compte de l'ordre des mots qu'ils analysent. Ces modèles ne seront pas présentés ici, mais d'excellents sites et ouvrages en détaillent le fonctionnement.

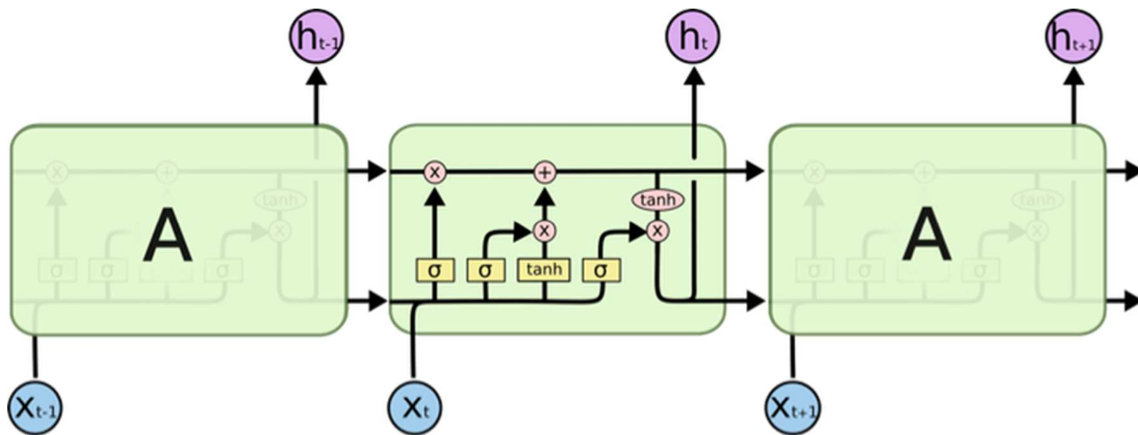



Figure 3: Schéma d'un LSTM. Tiré du blog de Chris Olah

Quelques expérimentations avec un LSTM (Long-Short Term Memory, un type de réseau de neurones récurrent) nous ont permis d'obtenir des performances légèrement meilleures que la régression logistique du paragraphe précédent (~68 % d'exactitude). Cependant l'utilisation d'un LSTM pour la prédiction requiert près d'une seconde par réponse en utilisant les 4 cœurs d'un ordinateur portable. Son utilisation ne sera pertinente que si le serveur du chatbot est à même de maintenir toutes les conversations en parallèle avec un temps de réponse raisonnable.

Nous n'avons fait qu'effleurer les réseaux de neurones récurrents. En effet, il en existe de nombreux types différents (Gated Recurrent Unit, seq2seq, …), dont chaque implémentation doit être ajustée minutieusement pour obtenir les hyper-paramètres optimaux. De plus, pour la classification de texte les réseaux de neurones récurrents sont parfois appliqués caractère par caractère ! Autrement dit on peut aussi entraîner un modèle à analyser les messages des visiteurs lettre par lettre pour prédire la réponse.

## Evaluation

Évaluer les performances d’un algorithme de machine learning est toujours un point délicat ; mais pour un chatbot, ce sujet est encore plus épineux que d’habitude. Une métrique couramment utilisée, nommée BLEU, compare les séquences de mots entre les phrases prédites et les phrases à prédire. Le problème est qu’une telle métrique attribue un score de 0 à la phrase « Je vous prie de bien vouloir patienter une minute » lorsque la phrase à prédire est « Pourriez-vous attendre un instant s’il-vous-plaît ? », alors que le sens est similaire. On peut repousser légèrement cette limitation en comparant les représentations word2vec des deux phrases au lieu de comparer les mots eux-mêmes ; les similarités sémantiques sont alors prises en compte. Cependant, une étude approfondie a montré qu’aucune de ces métriques ne s’approche d’une évaluation humaine.

Nous avons donc opté pour une métrique très simple : l’exactitude (i.e. le pourcentage de phrases-type correctement prédites) vis-à-vis de nos phrases type. Cette métrique souffre du fait que prédire une phrase sémantiquement proche de la réponse réelle n’est pas valorisé. Néanmoins c’est la métrique qui nous paraissait la plus représentatives de la qualité des réponses du chatbot, par comparaison avec des essais de conversations.

Notre meilleur modèle atteint donc une exactitude de plus de 65%. Ce chiffre peut paraître assez modeste, alors que des essais nous ont permis de vérifier que le chatbot reproduit bien le comportement des agents dans une conversation. Les erreurs qu’il commet correspondent souvent à une réponse sémantiquement proche de la bonne réponse, et qui en réalité pourrait être considérée valide.

## En résumé

L’entraînement d’un chatbot est un problème de d’apprentissage automatique très spécifique. Il comporte des problématiques propres de découpage des messages, pré-traitement du texte, vectorisation, modélisation, et même en ce qui concerne l’évaluation des performances, le

problème semble particulièrement épineux. Pour chacune de ces étapes nous avons passé un temps important à vérifier la cohérence entre l'entrée et la sortie, et la construction d'une chaîne de traitement idoine nous a permis d'optimiser chaque opération facilement.

Notre meilleur modèle a été évalué à l'aune de plusieurs métriques, notamment son exactitude, ainsi qu'au regard de sa cohérence dans des conversations tests que nous menions nous-même. Au final, nous avons obtenu les meilleures performances avec un pré-traitement assez standard, une représentation TF-IDF de dimension 300 qui incorpore les bigrammes et trigrammes, et un modèle de régression logistique.

Ce modèle permet d'avoir des conversations avec le chatbot qui sont très proches de celles des opérateurs. Il ne reste plus qu'à le brancher au vrai chat : ce sera le sujet du troisième et dernier paragraphe, dans lequel nous analyserons l'utilisation d'un chatbot en condition de production et son interface avec une API.

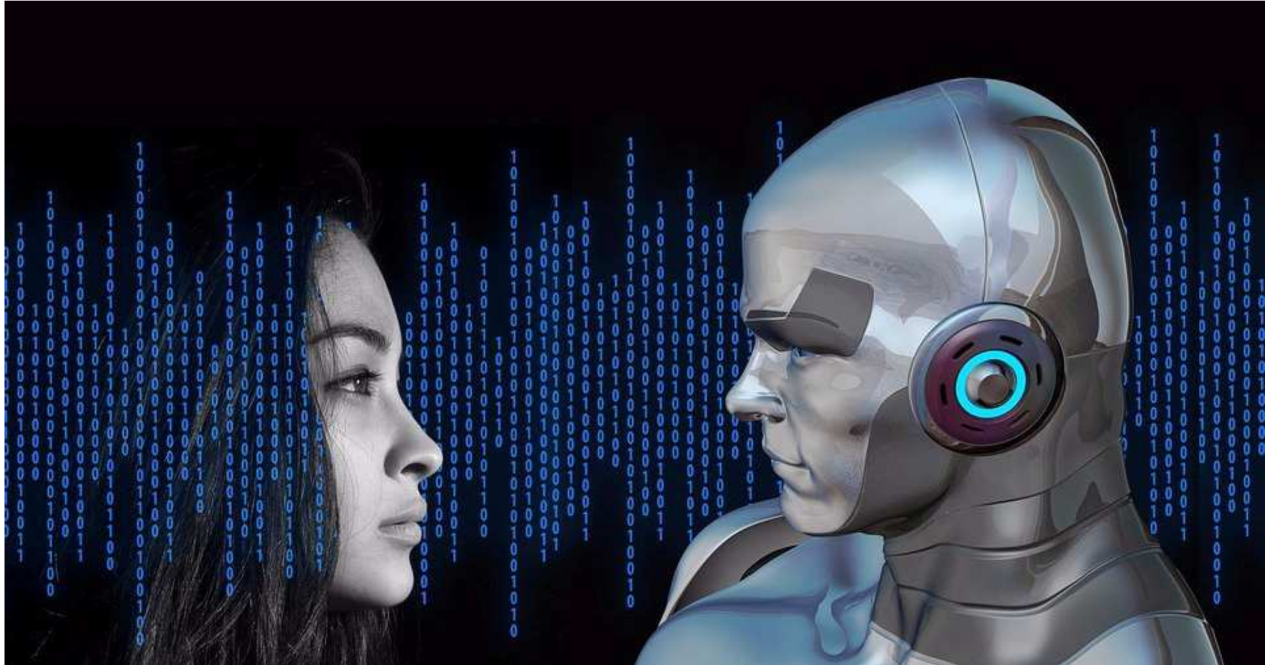
## Bonus : Donnez une bouche et des oreilles à votre chatbot !

Bonne nouvelle : il est assez simple de doter notre nouveau chatbot de la parole ! Voici un exemple d'un script python qui prononce une phrase (« Salut, ça va ? ») via votre micro, écoute votre réponse, et vous permet de vérifier qu'il vous a bien compris en répétant votre phrase. Il repose sur gTTS et SpeechRecognition.

```python
from gtts import gTTS
from playsound import playsound
import speech_recognition

def speak(text):
    filename = 'record.mp3'
    tts = gTTS(text=text, lang='fr')
    tts.save(filename)
    playsound(filename)
    print(text)

recognizer = speech_recognition.Recognizer()

def listen():
    with speech_recognition.Microphone() as source:
        recognizer.adjust_for_ambient_noise(source)
        audio = recognizer.listen(source)
        return recognizer.recognize_sphinx(audio, language='fr-FR')
    except speech_recognition.UnknownValueError:
        print("Could not understand audio")
    return ""

speak("Salut ça va ?")
speak("Tu as dit " + listen())
```

Ce module fonctionne aussi en français, et il suffit d’ajouter ce genre de fonctionnalités à l’interface du chatbot pour pouvoir discuter avec lui de vive voix.

# Chatbot à la demande avec les APIs



On désigne par API (application programming interface) un ensemble d'outils permettant à deux applications de s'interfacer et de fonctionner de concert.

Les APIs connaissent depuis quelques années un regain d'intérêt dans le monde de l'IT. De plus en plus de compagnies proposent des APIs ouvertes au public : citons celles bien connues de Google, Facebook et Twitter.

Bien que cette idée ne date pas d'hier, son succès actuel s'explique par plusieurs facteurs. Parmi ceux-ci figure l'apparition des smartphones. IHS Markit estime que 6 milliards de ces petits appareils seront en circulation en 2020, soit une croissance de 50% par rapport à 2016.

Avec la diversité des systèmes d'exploitation dont ces smartphones sont pourvus (Android, iOS, Windows, etc.), de plus en plus de développeurs d'applications mobiles optent pour la stratégie du back-end as a service

(BaaS). Cette architecture s'appuie sur le paradigme client-serveur et consiste à positionner le back-end de l'application sur un serveur distant et de ne mettre à disposition du mobile uniquement que le front, les deux étant interfacés au moyen d'une API.

On peut généraliser le phénomène aux autres objets connectés, avec le développement de la communication inter-machine, où les APIs sont devenues essentielles. De manière plus générale, toutes les grandes tendances innovantes de l'IT (Cloud, Big Data, IoT, etc.) favorisent l'utilisation des APIs et les placent au centre de leurs dispositifs.

Aujourd'hui, une véritable économie autour des APIs est en train de prendre forme, que d'aucuns nomment le B2D (business to developer). Les fournisseurs d'API permettent à des communautés de développeurs de bâtir de nouvelles applications autour de leurs services et d'étendre leurs possibilités. En mettant ainsi à disposition ces APIs, les fournisseurs bénéficient d'une rente supplémentaire. Les développeurs qui consomment ces services, quant à eux, s'épargnent l'effort de développer des briques compliquées et peuvent les intégrer directement dans leurs solutions.

## APIs et data science

S'agissant de la science des données, les APIs s'avèrent être très utiles, et ceci à plus d'un titre. Elles constituent d'abord un très bon moyen d'enrichir son historique de données. Les APIs simplifient aussi l'architecture générale en permettant plus de modularité entre les sources de données et les modèles d'une part, et entre le modèle et le front-end d'autre part.

Enfin, les APIs apportent de l'agilité dans le développement et l'industrialisation des modèles de data science. En effet, ces modèles ont généralement leur propre cycle de développement (préparation des données, entraînement du modèle, validation, amélioration, etc.) qui est

indépendant de celui du reste de l'environnement applicatif, ils s'intègrent donc mieux dans un projet modulaire.

## API REST

Avec l'augmentation du nombre d'APIs et leur succès croissant, en particulier les APIs publiques, les fournisseurs sont confrontés à de nouvelles problématiques. Tout d'abord, de nombreuses APIs aux fonctions connexes ou identiques adoptent des designs radicalement différents les unes des autres, et il arrive parfois que les plus récentes réinventent la roue. De plus, devant le nombre important et la diversité des clients de ces APIs, il est impératif d'assurer leur scalabilité pour satisfaire toute la demande. Sans compter que beaucoup de ces APIs seront amenées à durer relativement longtemps, elles doivent donc dès l'origine être conçues pour être faciles à maintenir et à faire évoluer.

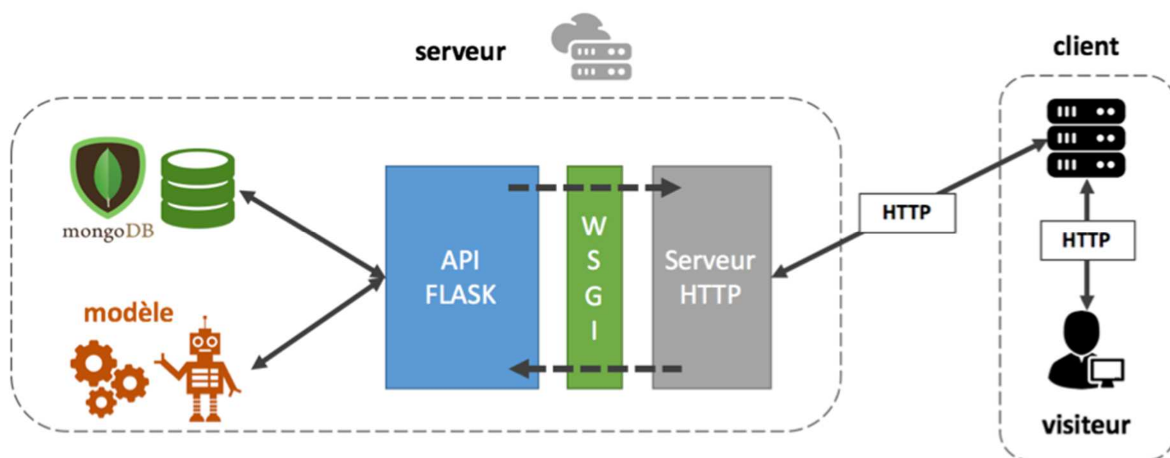
Si bien qu'une nécessité de standardisation s'est fait ressentir très tôt. Ceci a donné lieu à l'apparition d'une multitude de philosophies de standards de développement. Une de celles qui ont eu le plus de succès est connue sous l'acronyme REST (representational state transfer) : c'est un modèle d'architecture basé sur le protocole HTTP, qui induit tout un ensemble de règles et de bonnes pratiques.

## Tutoriel Flask

Pour toutes ces raisons, nous avons choisi de mettre en place une API REST dans l'architecture de notre chatbot. Nous avons utilisé pour cela la célèbre librairie Flask de Python. Flask est un framework de développement web qui se veut minimaliste, et qui confère à son utilisateur une liberté de conception, très appréciable pour réaliser une API. Flask repose sur le protocole WSGI (web server gateway interface), qui permet d'interfacer l'application Python (notre API) et le serveur HTTP.

Notre API a pour fonction de fournir une abstraction du modèle au front-end. On obtient une architecture modulaire dont les trois principales briques constitutives (back-end, API, front-end) peuvent être développées séparément.

Dans la suite de ce tutoriel, nous admettrons avoir un modèle de chatbot sérialisé prêt à l'emploi au format pickle (model.pk). Ce modèle produit une réponse du bot en fonction d'une conversation en entrée (une liste ordonnée des messages à un instant donné).




La première chose à faire est d'instancier une application Flask (**app**). C'est cette instance qui, grâce à WSGI, va pouvoir traiter les requêtes réceptionnées par le serveur web.

Nous avons aussi utilisé la base de données MongoDB. Par souci de simplification, une classe DBManager a été créée pour gérer l'interaction avec MongoDB. Sachez toutefois qu'il existe une librairie (Flask-PyMongo) proposant un connecteur entre Flask et MongoDB.

Nous stockons le modèle sérialisé dans la variable **model**.

```python

from flask import Flask, request, redirect, url_for, jsonify
from chatbot.utils import DBManager, Conversation
import pickle

app = Flask(__name__)
db = DBManager(app)

model = pickle.load('model.pk')

```

*Design*

Avant d'aller plus loin dans le code, il nous faut réfléchir au design de l'API, en respectant au mieux les principes REST. Le paragraphe qui suit est un peu complexe, donc restez concentrés !

Le paradigme REST met en œuvre deux concepts fondamentaux que sont **les ressources et les représentations**. Une ressource désigne tout objet, abstrait ou concret, identifié par un URL qui lui est propre (deux ressources ne peuvent avoir le même URL à moins d'être identiques). On appelle représentation toute description complète ou partielle obtenue après accès à la ressource grâce à l'URL associé. Une ressource peut avoir plusieurs représentations différentes. Une représentation peut aussi servir à mettre à jour une ressource, on parle de changement d'**état de la ressource**.

Dans l'optique de notre chatbot, un utilisateur de ce service doit pouvoir réaliser deux actions : mettre à jour l'état d'une conversation (la conversation est créée si elle n'existe pas déjà), et obtenir la dernière réponse du bot.

Si on transpose ce schéma dans le paradigme REST, on peut alors définir une ressource comme étant une conversation, et une représentation de cette ressource comme étant le dernier message de cette conversation (qui sera théoriquement toujours un message du bot, ce dernier répondant toujours à son interlocuteur).

Pour ce qui est de l’URL de chaque conversation, il sera tout simplement composé de l’adresse de l’API et d’un identifiant unique, séparés d’un slash.

A présent, il nous faut un moyen de distinguer les deux actions que l’on souhaite implémenter (mise à jour et obtention de la représentation d’une conversation). L’architecture REST étant basée sur le protocole HTTP, nous allons associer un verbe HTTP à chacune de ces actions. Voici quelques-uns des verbes HTTP les plus communs, ainsi que leur fonction :

- GET : obtention d’une représentation de la ressource ;
- POST : sert entre autres à créer une ressource sous-jacente à la ressource ciblée dans la requête ;
- PUT : mise à jour totale de la ressource ;
- PATCH : mise à jour partielle de la ressource.

Nous choisirons le verbe PATCH pour la mise à jour des conversations et GET pour avoir accès au dernier message bot de chaque conversation.

*Code*

Une fois notre API conceptualisée, nous pouvons enfin nous atteler au code Python! La première fonction à implémenter est celle qui permet de mettre à jour une conversation. Nous utilisons pour cela le décorateur Python **app.route** fourni par Flask. Ce décorateur va intercepter toute requête PATCH dont l’URL contient l’identifiant de la conversation considérée et va appeler la fonction **update_conversation**. Cette fonction va insérer le message dans la conversation en base (ou créer la conversation si elle n’existe pas). Deux cas se présentent alors : si le message inséré est de type bot, le code réponse 200 est renvoyé (il signifie simplement que le serveur a traité la requête avec succès). Sinon, une redirection est effectuée avec un code réponse 303 pour accéder à la fonction **get_answer** (voir ci-après).

```python
@app.route('/<string:id>', methods=['PATCH'])
def update_conversation(id):
    message = request.get_json()['message']
    d = {"message": message}
    db.upsert(id, d)
    if message['type'] == 'bot':
        return jsonify({}), 200
    else:
        return redirect(url_for('get_answer', id=id), 303)
```

Pour ce qui est de l’accès à une représentation (équivalent ici au dernier message de type bot de la conversation), nous développons une fonction analogue nommée **get_answer**. Celle-ci va appliquer le modèle sur la conversation considérée, et générant un message bot sui sera renvoyé au client au format JSON, avec un code 200.

```python
@app.route('/<string:id>', methods=['GET'])
def get_answer(id):
    messages = db.find(id)
    conv = Conversation(messages)
    pred = model.predict_conversation(conv)
    return jsonify(answer=pred[0]), 200
```

Enfin, ces quelques lignes de code mettront l’API en marche.

```python
if __name__ == '__main__':
    app.run()
```

Bonne programmation à toutes et tous !

## FIN