

**Louanès HAMLA**

(Promotion 2020)



# RAPPORT DE STAGE

## ING4

STAGE TECHNIQUE DE DEUXIEME ANNEE DE  
CYCLE D'INGENIEUR



|  |  |
|--|--|
|  <div style="text-align: center;"> <b>Rapport de Stage</b><br/>       Cycle Ingénieur • 2ème année<br/>       2018/2019     </div>   |  |
| <b>Elève ingénieur</b><br><br><b>Prénom :</b> Louanès<br><b>Nom :</b> HAMLA  | <b>Majeure d'enseignement</b><br><br><input checked="" type="checkbox"/> SI <input type="checkbox"/> OCRES<br><input type="checkbox"/> SE <input type="checkbox"/> FI<br><input type="checkbox"/> EN <input type="checkbox"/> SA |
| <b>Entreprise d'accueil</b><br><br><b>Nom :</b> BeesApps<br><br><b>Adresse :</b><br>34, rue de Saint-Petersbourg<br>75008 Paris<br>France<br><br><b>Engagement de confidentialité (NDA) :</b> <input type="checkbox"/> oui <input checked="" type="checkbox"/> non<br><b>Rapport à remettre au tuteur de stage à l'issue de la correction :</b> <input checked="" type="checkbox"/> oui <input type="checkbox"/> non<br><br><b>Tuteur de stage (prénom / nom) :</b> Christopher MESQUITA<br><b>Téléphone du tuteur :</b> +33 6 14 62 22 16<br><b>Courriel du tuteur :</b> christopher.mesquita@beesy.com<br><br><b>Signature du Tuteur de Stage et cachet de l'entreprise (obligatoire) :</b><br><br><div style="text-align: center;"> <br/>  </div> |  |
| <b>Description de la mission</b><br><br><ul style="list-style-type: none"> <li>- Gestion et suivi du support client</li> <li>- Training et amélioration de l'IA FR</li> <li>- Formation de l'IA EN</li> <li>- Optimisation du processus de training</li> </ul>   |  |
| <b>Période du stage</b> (durée effective) : 89 jours (18 semaines, dont 6 jours fériés)<br><b>Du</b> 15 avril 2019 <b>au</b> 15 août 2019  |  |

# Remerciements

Avant tout développement sur cette expérience professionnelle, il me semble plus que légitime de commencer ce rapport de stage par des remerciements.

Je tiens à remercier dans un premier temps, toute l'équipe pédagogique de l'ECE et les intervenants du service responsable des stages, qui ont accepté et rendu possible le début de mon stage en respectant les délais impartis.

Je remercie profondément Monsieur David CHEVENEMENT, CEO de Beesapps qui m'a formé et accompagné tout au long de cette expérience professionnelle enrichissante, avec énormément de patience et de pédagogie, ayant su partager sa vision et son expérience d'ingénieur aguerri.

Aussi, j'aimerais adresser mes remerciements à :

Monsieur Christopher MESQUITA, Co-founder et responsable des ventes, pour son encadrement en tant que tuteur tout au long de ce stage.

Monsieur Jean-Charles RAGONS, Lead Software Engineer, pour ses précieux conseils et sa bonne humeur.

Mesdames Dina CHEVENEMENT et Cholé VEAUUVY, Responsables du pôle Marketing pour leur accueil sympathique et leur coopération professionnelle tout au long de ces 4 mois d'apprentissage.

# Sommaire

|      |   |    |
|------|---|----|
| I.   | Contexte.....                                     | 5  |
| a-   | A propos de l'entreprise.....                     | 5  |
| b-   | Missions et tâches confiées .....                 | 5  |
| c-   | Qu'est ce qu'un chatbot .....                     | 6  |
| II.  | Fonctionnement de Beesy .....                     | 7  |
| a-   | Rasa Core & Rasa NLU .....                        | 7  |
| b-   | Fonctionnalités de Beesy .....                    | 10 |
| III. | Modélisation et Training .....                    | 12 |
| a-   | Configuration du modèle utilisé .....             | 12 |
| b-   | Processus de training .....                       | 14 |
|      | 1- Traitement des conversations à valider         |    |
|      | 2- Lancement d'un training                        |    |
|      | 3- Traitement des flags                           |    |
| c-   | Evaluation et Analyse .....                       | 16 |
|      | 1- Script Intent results                          |    |
|      | 2- Script Entity errors                           |    |
| d-   | Gestion des patterns.....                         | 20 |
| IV.  | Training en grande quantité .....                 | 21 |
| a-   | Générateur Excel .....                            | 21 |
| b-   | Script Extractor .....                            | 22 |
| V.   | Conclusion .....                                  | 28 |
| a-   | Etat de l'étude et apport pour l'entreprise ..... | 28 |
| b-   | Potentiel suite de l'étude .....                  | 28 |

# I – Contexte

## 1- A PROPOS DE L'ENTREPRISE D'ACCUEIL

Beesapps est une Start-up Parisienne conçue pour organiser et prioriser la charge de travail des managers d'entreprises. Ils ont développé un chatbot au courant de l'année 2019 : Beesy.

Beesapps apporte une solution collaborative aux managers d'entreprises, leur permettant de mieux structurer leurs plans d'action, la gestion de leurs projets, ainsi que leurs suivis de réunion.

L'entreprise existe depuis plus de huit ans sur plateforme web et 50 000 users utilisent aujourd'hui Beesaaps comme principal outil de prise de note.

J'ai eu l'opportunité d'expérimenter l'ambiance « start-up » où l'organisation du travail de chaque employé de la boîte est plus ou moins imbriquée, et directement liée à la croissance et l'évolution de l'entreprise. En effet nous étions divisés en différents « pôles », un pôle marketing qui gère la partie communication et faire-valoir de l'entreprise, un pôle commercial, qui consiste à récupérer de nouveaux clients en leur proposant des démos de la solution (webinars) et à fidéliser les utilisateurs, et le pôle technique, qui gère la récupération, le stockage, l'affichage et la sécurisation des données avec une partie Intelligence Artificielle qui s'occupe du chatbot : Beesy.

Beesy est l'assistant virtuel (bot) qui permet aux utilisateurs d'avoir accès plus facilement à leur notes et de créer plus rapidement des actions. Implémenté sur MS Teams (après avoir signé un partenariat avec Microsoft au courant de l'année 2019), Skype et via Mail, on communique avec Beesy en langage naturel, l'enjeu est de s'adapter à la façon dont les utilisateurs expriment leurs intentions.

Doté de 3 intelligences artificielles, il existe beaucoup de fonctionnalités que Beesy est capable de gérer. On reviendra sur ces fonctionnalités un peu plus tard.

## 2- MISSIONS ET TACHES CONFIEES

Ce stage technique fût mon premier stage au sein d'une entreprise IT et j'ai pu apprendre énormément de choses, aussi bien sur le point technique que sur le travail en entreprise en général. Mes journées commençaient à 9h30 et se finissaient à 18h (avec 2 h de pause entre 12h et 14h). Ma mission principale, outre la gestion des demandes de support de la part des utilisateurs, était de me charger du training quotidien et de l'amélioration continue de Beesy. En étudiant différentes pistes d'amélioration, mon rôle était plus orienté R&D en IA conversationnel (Etudes du Framework utilisé, quels paramètres du modèle configurer ? Evaluer la performance actuelle du chatbot, et surtout comment l'entraîner de manière plus rapide et efficace ?).

Cela peut se résumer de la sorte :

- Lvl 1 - Support
- Lvl 2 - Suivi des demandes d'IA, training et modélisation quotidienne
- Lvl 3 - Identification des patterns d'erreurs / Identification des phrases type à générer
- Lvl 4 - Optimisation des paramètres du training de l'IA pour améliorer le taux de reconnaissance, diminuer overfitting.

### 3- QU'EST-CE QU'UN CHATBOT ?

Par définition, un chatbot est un système de communication entre Humain et Machine via langage naturel. Le marché des chatbots a rapporté 994,5 millions de dollars en 2014, et est estimé à monter à 8 milliards, si bien que 80% des entreprises utiliseront cette technologie d'ici 2022. Le chatbot facilite la communication entre le client et l'entreprise, son rôle d'assistant virtuel permet aux clients d'utiliser la solution - proposée par l'entreprise – à tout moment (24h/24) de manière plus fluide, rapide et efficace. En effet on entraîne le bot à reconnaître des requêtes utilisateurs fréquentes et on enclenche des actions appropriées à cette détection. On cherche à comprendre l'intention de l'utilisateur (intent) afin d'y fournir une réponse (utter) adéquate.

Le Natural Language Processing (NLP) est par définition le process de manipuler des données textuelles et de les rendre analysable. Le Natural Language Understanding (NLU) vise à approfondir la compréhension du texte par la machine, on cherche à comprendre l'intention de la phrase. Ces deux approches de Machine Learning permettent de déduire quelle action effectuer en fonction de la phrase analysée. Aussi, pour permettre au chatbot de répondre à plusieurs reprises et d'échanger avec l'utilisateur sous forme de conversation, on utilise une technique de Machine Learning appelée : Dialog Management ; on entraîne le chatbot sur des conversations prédéfinies, dites « stories ». Ce dernier devient de plus en plus précis et fait moins d'erreurs au fur et à mesure qu'on lui fournit des exemples de stories.

L'avantage du chatbot : C'est une technologie au quelle on fait appel directement via un canal de communication (E-mail, WhatsApp, Messenger, Slack, Skype etc ...). C'est-à-dire qu'il n'est pas demandé à l'utilisateur d'installer un exécutable ou autre fichier. Il faut uniquement importer le bot comme contact (via une fiche de contact) et on peut ainsi échanger avec le bot.

## II- Fonctionnement de Beesy

### A- Rasa Core & Rasa NLU

**Setup Rasa :** On utilise comme open source Framework : Rasa NLU et Rasa Core

IA 1 : Dialog Management (Rasa Core)

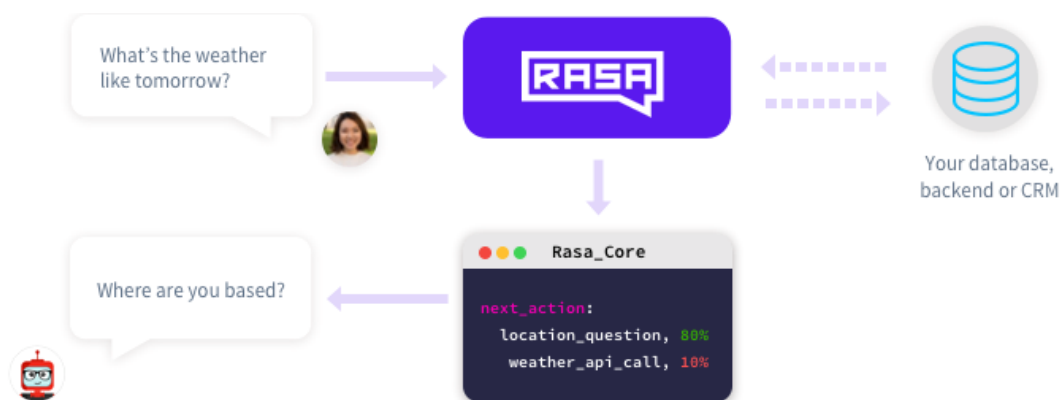
IA 2 : Intent classification (Rasa NLU)

IA 3 : Entity prédiction (Rasa NLU)

#### IA 1 Dialog Management :

Comme son nom l'indique, cette IA est responsable de la gestion du dialogue entre l'User et le Chatbot. Sur un panel d'actions que le chatbot peut effectuer, il y'a un panel de réponses (appelées « utter ») à renvoyer à l'utilisateur.

#### Exemple weather chatbot :



Le but de cette IA est donc de savoir quelle utter utilisé en fonction de la phrase qui a été envoyé par l'utilisateur, quelle prochaine action à accomplir par le bot.

→ Ok, mais alors comment savoir quelle action effectuer en fonction de la phrase reçue par le chatbot ? C'est à ce moment que l'IA 2 intervient.

## **IA 2 : Intent classification**

Un intent est tout simplement l'intention qui définit cette phrase. Il peut y avoir une centaine de différentes façons d'écrire une phrase qui porte la même intention.

Ex : donne-moi ma liste de tâche, je veux que tu me liste mes actions à faire, ma todolist stp, etc... →

L'intention ici est d'afficher une liste d'action.

L'idée est donc de regrouper toutes les possibilités de phrases exprimant la même intention dans un fichier portant un nom d'Intent spécifique. Il existe par conséquent autant d'intentions à identifier que de fonctionnalités assurées par le chatbot (Chaque intent détecté enclenche un type d'action différent).

## **IA 3 : Entity prediction :**

Une fois que l'intention est détectée, on s'intéresse à ce qui compose la phrase :

Affiche-moi les actions de « Paul » et Affiche-moi les actions d'« Elsa » sont en réalité deux réponses différentes pour le chatbot même s'ils partagent la même intention qui est : Actionlist (Afficher une liste de tâche)

En effet, l'intent permet de choisir quel type d'action effectuer, et les entités/slot permettent d'apporter plus d'information à la réponse fournie par le bot.

Exemple :

« Quelles sont les actions à faire ? »

→ Intent : Afficher une liste d'action

→ Par défaut, s'il n'y a pas d'entité, on affichera les actions de l'utilisateur en question, et les actions pour aujourd'hui.

« Dis-moi ce que David doit faire pour la semaine prochaine »

Owner                      Deadline

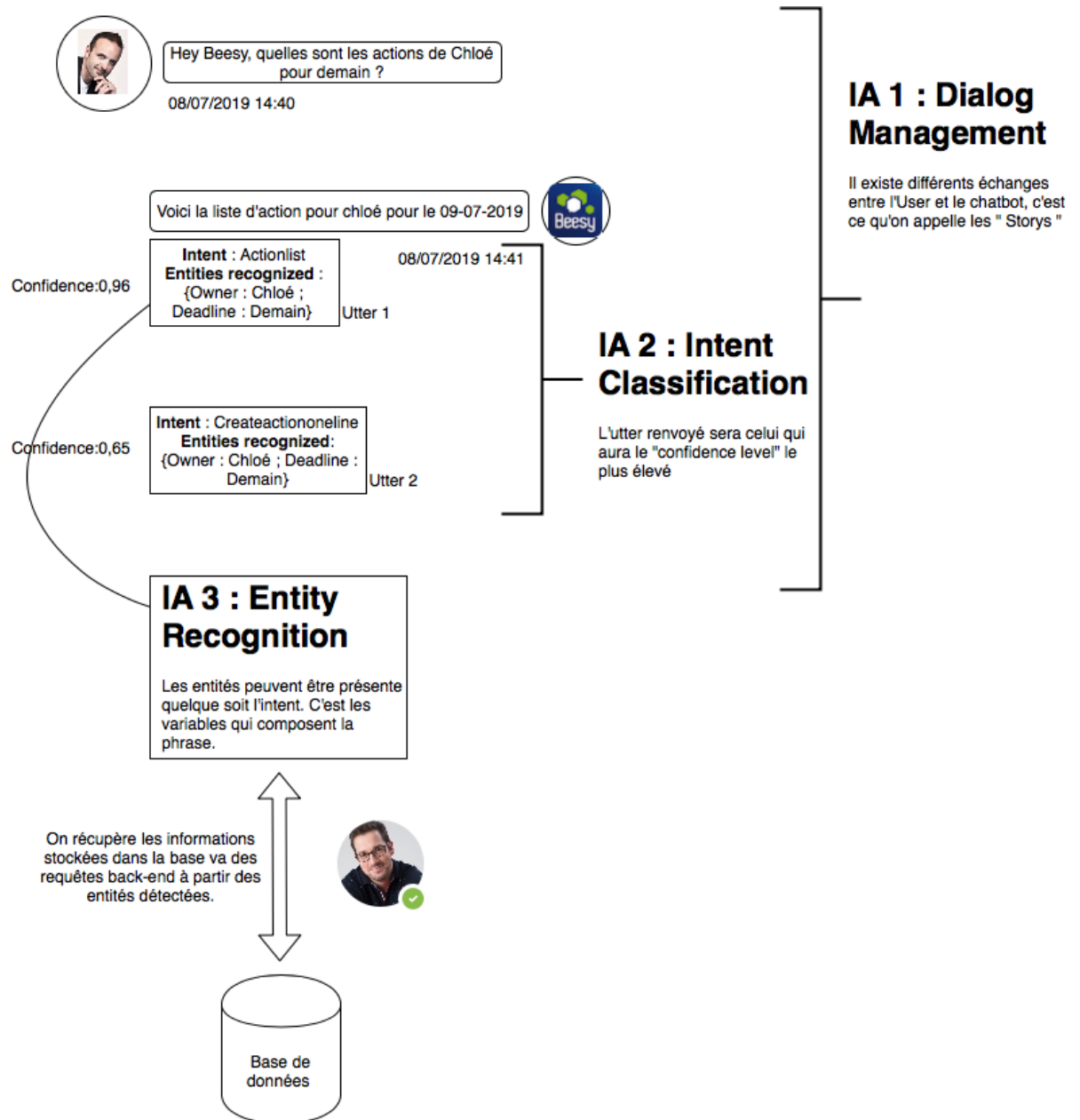
→ Intent : Afficher une liste d'action

→ Entités détectées : {Owner : David, Deadline : pour la semaine prochaine}

Le type d'action est le même mais les requêtes back end pour récupérer les informations changent en fonction des entités qui composent la phrase.



Voici un schéma récapitulant comment ces différents algorithmes de ML sont imbriqués :



## B- FONCTIONNALITES DE BEESY :

En l'occurrence, Beesy est entraîné sur 48 fichiers d'intents, vous trouverez ci-dessous les principaux intents nécessitant une explication un peu plus détaillée, et les autres intent [en annexe 1A](#).

### Intents type :

1-Actionlist : —> *Affiche les actions d'une personne, d'un projet*

- a- Actionlist CALL —> *Affiche les appels à passer*
- b- Actionlist DOCUMENT—> *Affiche les documents à rédiger*
- c- Actionlist EMAIL —> *Affiche les emails à envoyer*
- d- Actionlist MEETING —> *Affiche les meetings*

→ Pourquoi mettre des intents pour chaque type d'action alors qu'il existe une entité « actiontype » qui permet justement de détecter l'action que l'utilisateur souhaite afficher/créer ?

L'action type est un nom commun : « appel », « document » ; si on souhaite afficher une liste d'appels, on peut très bien avoir une phrase en intent actionlist qui accomplit cette tâche :

« Quels sont les [appels](actiontype: appel) que je dois passer ? » ; « Affiche moi ma liste de [documents](actiontype: document) à rédiger »

Maintenant si l'utilisateur exprime son intention avec un verbe, il nous est impossible de tagger l'entité actiontype :

« Qui dois-je appeler ? » ; « que dois-je rédiger [cette semaine](deadline) »

Ces phrases n'ayant pas d'actiontype défini, sont celle qui se trouvent dans les fichiers d'intent attribués par type d'action : actionlistcall ; actionlistdocument, createactiononlineemail .. etc

2-Create action —> *Crée une action en plusieurs étapes sous forme de dialogue (story)*

3-Create action one line —> *Crée une action du premier coup*

- a-Create ACTION ONE LINE CALL —> *Crée une action de type appel*
- b-Create action one line DOCUMENT —> *Crée une action de type document*
- c-Create action one line EMAIL —> *Crée une action de type mail*
- d-Create action one line MEETING —> *Crée une action de type meeting*

NB : On souhaite différencier les types d'actions afin de permettre à l'utilisateur de mieux organiser son temps en fonction de l'action qu'il a à accomplir. En effet, rédiger un document, appeler quelqu'un ou écrire un email prendra plus de temps que de faire une simple action. Nous pouvons définir l'actiontype directement en createactiononline, mais tout comme pour actionlist, il est impossible de récupérer le type d'action s'il n'est pas explicitement écrit dans la phrase. D'où l'utilité de créer des intents spécifique pour détecter les différents types d'actions.

#### 4-Createnote —> Rédige un compte rendu à partir d'un mail ou d'un Word

NB : Mailto Note

*Le Mail to Note est l'une des plus impressionnante fonctionnalités de Beesy. L'utilisateur prend des notes lors d'une réunion, il écrit tout ce qu'il juge être important sans pour autant se soucier de la mise en forme. Une fois la réunion terminée, il ne reste plus qu'à envoyer ses notes à Beesy en spécifiant dans l'objet du mail : crée-moi le compte rendu de cette note (ou n'importe quelle autre phrase permettant de déclencher l'intent createnote »). L'IA analyse ligne par ligne les phrases qui lui ont été envoyées, en structurant le compte rendu en différents paragraphes (en fonction des puces numérotées qu'elle croise : 1-) ordre du jour ; 2-) etc ..) et surtout : elle distingue ce qui est une « action » à faire, en l'attribuant à la bonne personne et en y mettant la bonne deadline (Si la deadline est dépassée, elle apparaîtra en rouge dans le compte rendu) ; d'une simple remarque. Les remarques sont gardées, l'idée est de mettre en avant les actions à réaliser afin d'avoir un compte rendu proprement mis en forme, bien organisé, permettant une visibilité du plan d'action rapide et efficace. Tout cela en 2 minutes, le compte rendu est renvoyé par mail à l'utilisateur. (Il peut y insérer le logo de son entreprise s'il le souhaite)*

6-Garbage → Renvoie qu'il n'est pas sûr que l'utilisateur s'adresse bien à lui Ex : « salut JC tu as vu le match d'hier ? »

7-Conversation → La phrase écrite par l'utilisateur contient des éléments qui permet d'actionner une des fonctionnalités de Beesy mais il manque des éléments pour prendre une décision. Au lieu de renvoyer quelque chose sans être sûr que c'est bien ce que l'utilisateur cherche à obtenir, on préfère répondre par l'utter conversation qui demande à l'utilisateur de nous donner plus d'information. Ex : « compte rendu Microsoft » ; est ce qu'on veut créer le compte rendu (createnote) ? ou bien l'exporter (noteexport) ?

*Derrière chaque « conversation » se cache une action que l'utilisateur n'a simplement pas su exprimer.*

**Garbage** → Toutes les phrases n'ayant pas de verbes d'action et n'ayant pas de mots clés liés à un autre intent

**Conversation** → Que les phrases ayant un mots clés capable d'enclencher la détection de plusieurs autres intents (on demande plus d'informations pour savoir quel intent choisir).

#### Beesy est entraîné sur 20 entités (Annexe 1B)

Les entités peuvent prendre 2 valeurs : text ou categorical.

- Text pour owner, project, goal et tout ce qui ne peut avoir qu'une seule valeur, définit dans la phrase. On écrit directement le nom de l'entité [.....](owner) ;[.....](project) dans les phrases de training
- Categorical pour toutes les entités qui peuvent avoir plusieurs valeurs : Actiontype = {appel ; email ; document etc ..} ou encore Actioncategory qui peut prendre différentes valeurs parmi : urgentes, importantes, terminées etc ... On doit spécifier la valeur de l'entité dans les phrases de training

Ex : Rédiger rapport de stage pour demain c'est [urgent](**actioncategory : urgentes**) etc ..

Les valeurs des entités peuvent avoir plusieurs formulations, pour toutes les détecter on stock tous les synonymes de ces valeurs dans un fichier nlu\_synonyms.md

Ainsi on peut s'adapter au vocabulaire des utilisateurs et aussi gérer les fautes de typos :

# Synonym reunion :

- Réunion
- Rendez-vous
- Rdv
- Meeting

De cette manière, il ne reste plus qu'à préciser le nom de l'entité liée à la valeur « reunion » dans les phrases de training de nos fichiers nlu\_data

## III- Modélisation et Training

Actuellement nous avons environ 50 000 phrases sur lesquels l'IA est formée. Le training est consacré à former l'IA sur une bonne classification des intents et sur la détection des entités que nous avons définies.

### 1- CONFIGURATION DU MODELE UTILISE

2 fichiers sont primordiaux pour la bonne exécution des fonctions générées par rasa core et nlu. Le premier fichier contenant la configuration de notre modèle : nlu\_config\_model.yml

```
1 language: "fr"
2
3 pipeline:
4   - name: "nlp_spacy"
5   - name: "tokenizer_spacy"
6   - name: "intent_featurizer_count_vectors"
7   - name: "ner_crf"
8   - name: "ner_synonyms"
9   - name: "ner_spacy"
10  - name: "intent_classifier_tensorflow_embedding"
11
```

Ce dernier regroupe tous les composants de ML utilisés.

Le second fichier: domain.yml décrit l'univers dans lequel le chatbot est défini, c'est-à-dire une liste des entités, intents, requêtes utilisateurs qu'il est censé recevoir, ce qu'il est censé effectuer comme actions, comment répondre et quelles informations stockées.

Ce qui compose la pipeline :

- **Tensorflow**

On utilise pour la classification d'intent : intent\_classifier\_tensorflow\_embedding, permettant de compter (from scratch) le nombre d'occurrence des mots par intent, et baser ainsi la classification par rapport à la présence de certains mots spécifique pour chaque fichier d'intent. (Ex : on rencontre souvent le mot « envoyer » dans les fichiers de training pour l'intent createactiononlineemail).

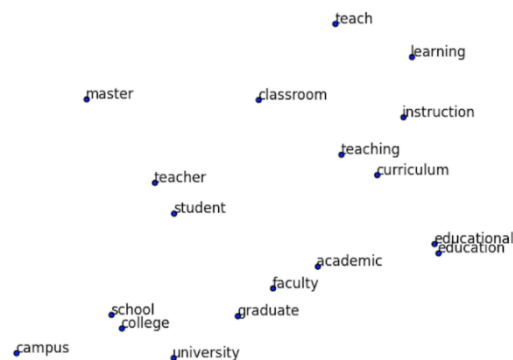
Avant de pouvoir compter le nombre d'occurrence il faut pouvoir distinguer les mots composant une phrase (tokens) et les transformer en vecteur. Ceci est géré par l'intent\_featurizer\_count\_vectors :



Il existe des paramètres que l'on peut ajuster simplement en indiquant la nouvelle valeur et le nom du paramètre en question en dessous du composant correspondant. Par exemple pour l'intent\_classifier\_tensorflow\_embedding, on peut augmenter le paramètre : embedding\_dimension.

L'embedding permet de trouver les similarités entre différents mots. Chaque mot est transformé en vecteur grâce à l'intent\_featurizer\_count\_vector. Ensuite à partir de l'embedding matrix, on stock les index des mots qui ont des similarités, qui sont souvent précédés ou qui précèdent un même mot. Ainsi en augmentant la dimension de la matrice, on prend en considération un plus grand nombre de mots. On augmente ainsi la robustesse du modèle, permettant de mieux gérer la détection d'intent à faible volume.

Il est possible de visualiser les relations entre les mots d'une même matrix en appliquant des algos de « dimensionality reduction ». Par exemple, ci-dessous on arrive à voir des data points représentant des mots appartenant à la même embedding matrix.



- **Spacy**

Nous avons recours à Spacy, on initie la bibliothèque simplement en introduisant nlp\_spacy comme composante du modèle permettant de faire appel à des fonctions appropriées à l'analyse de texte (processing), on utilise ner\_spacy pour son importante base d'entités « entraînés » sur des noms et prénoms, nous octroyant une meilleure détection d'Owner. Bien évidemment spacy fournit aussi un tokenizer, permettant de correctement séparer les mots de telle sorte à gérer les prénoms composés, les initiales et autres.

- **Named Entity Recognition (NER)**

On utilise également ner\_crf comme Named entity recognizer, nous permettant de définir nos propres entités (d'où la nomenclature des phrases de trainings). En effet il ne suffit plus qu'à marquer entre crochets les entités qu'on souhaite détecter, suivi du nom de l'entité en question entre parenthèse. (Ex : [lundi](deadline), [Maxence](owner) etc ..). Plus l'IA 3 aura croisé ces entités dans les fichiers de trainings, plus le bot prédira correctement cette entité lorsqu'il la croquera dans une nouvelle phrase écrite par les utilisateurs. Un des paramètres intéressants de cette composante est le « BILOU » tagging. En effet le ner\_crf nous permet de définir nous-même nos entités, avec le bilou tagging, on peut donner plus de détails :

Exemple : Si notre entité est composée de 3 mots :

« Rédige moi le cahier des charges, [au plus tard demain](deadline) concernant le projet [Microsoft](project) »

On peut préciser la position des mots présents dans l'entité de la manière suivante :

B : Beginning (pour le premier mot composant l'entité)

I : Inside (pour tous les autres mots à l'intérieur de l'entité, entre le premier et le dernier)

L : Last (pour le dernier mot composant l'entité)

O : Outside (signifiant que le mot n'est pas à l'intérieur d'une entité)

U : Unique (lorsqu'il n'y a qu'un seul mot pour l'entité)

On utilise le **BILOU** tagging pour avoir plus d'information sur la position des tokens constituant une entité, on précise d'abord la position suivie du nom de l'entité en question. En l'appliquant sur notre phrase nous aurons une tout autre nomenclature :

« Rédige(O) moi(O) le(O) cahier(O) des(O) charges(O) [au](B-deadline) [plus](I-deadline) [tard](I-deadline) [demain](L-deadline) concernant(O) le(O) projet(O) [Microsoft](U-project) »

On utilise également ner\_duckling pour tout ce qui est détection de date, mais on ne le spécifie pas dans la pipeline, on l'utilise en serveur externe. Puisque nous avons différents types de date comme entités : createddate, deadline ; fromdate, on doit d'abord filtrer nos entités avant de les passer à duckling afin de confirmer que c'est en effet bien une date (et non des chiffres correspondants à un salaire, une distance etc ..).

## B-PROCESSUS DE TRAINING

C'est un process en plusieurs étapes :

- 1- Traiter les conversations à valider
- 2- Lancer un training
- 3- Traiter les flagged

### 1-Traiter les conversations à valider


BeesApps a mis au point une plateforme qui permet d'avoir toutes les phrases que les users écrivent à Beesy. Avec des cases qui montrent l'Intent et les entités qui ont été prédites :

Admin

Beesy.me

ownCloud

Jira

Beesy

ADMIN

LOGOUT

| Query  | Valid ?             | Flag ?               | Close ?               | Intent             | Intent % | SI Deadline        | Duckling Text   | SI Owner | SI Project | SI Channel | SI Actioncategory | SI Actiontext | SI Actiontype | SI Project Template | SI Goal |
|--|---------------------|----------------------|-----------------------|--------------------|----------|--------------------|-----------------|----------|------------|------------|-------------------|---------------|---------------|---------------------|---------|
| Text   | Text                | Text                 | Text                  | Text               | Number   | Text               | Text            | Text     | Text       | Text       | Text              | Text          | Text          | Text                | Text    |
| former l'a avec les phrases en anglais de Jesus la semaine ... | <a href="#">Add</a> | <a href="#">Flag</a> | <a href="#">Close</a> | creationactionline | 0.94     | la semaine proc... | la semaine p... | jesus    |            |            |                   |               |               |                     | 1       |

La première étape de training consiste donc à passer sur ces phrases qui correspondent à la partie « test dataset » en ML. Nous sommes en train de vérifier la capacité de Beesy à prédire des nouvelles phrases qui ne sont pas forcément présentes dans le dataset de training. Il suffit de passer sur ces phrases en cliquant sur « add » si l'intent est les entités ont correctement étaient prédites ou sur « flag » si l'IA s'est trompée. Si elle s'est trompée, on copie colle la phrase erronée dans le bon fichier d'intent en marquant toutes les entités que l'IA aurait dû prédire. On corrige en quelque sorte la fausse prédiction de l'IA.

## 2-Lancer un training

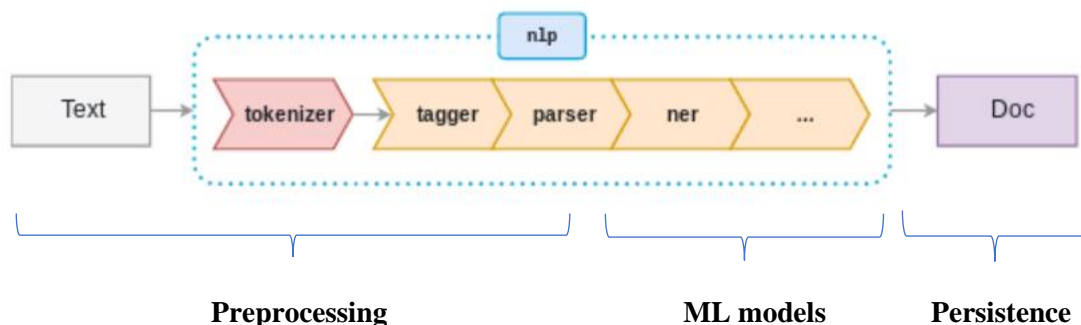
Le Framework Rasa est doté de plusieurs fonctions préconfigurées qui s'activent lorsqu'on exécute le fichier train.py. La première chose à faire est de récupérer la configuration du modèle sous forme d'objet contenant le chemin d'accès aux fichiers de training et le chemin où est stocké le modèle lorsque le training est fini.

La fonction "load\_data" intégré dans rasa nlu permet de lire nos fichiers de training à partir du path que l'on a précisé. Cette fonction renvoie un « TrainingData Object » qui regroupe l'intention et les entités prédites par les différents composants de la pipeline, pour les différentes phrases du dataset et le niveau de confiance de la prédiction.

```
{
  "text": "quels sont les actions de Kévin",
  "intent": "actionlist",
  "entities": [
    {
      "start": 25,
      "end": 29,
      "value": "Kévin",
      "entity": "owner",
      "extractor": "ner_crf",
      "confidence": 0.854,
      "processors": []
    }
  ]
}
```

Comment s'effectue le training ?

Cela commence par la création d'un « Trainer Object » qui prend en argument la configuration du modèle et construit une pipeline regroupant différents composants, où chaque composant est responsable d'une tâche NLP. La fonction Trainer. Train itère sur chaque composant de la pipeline en appliquant la fonctionnalité de ce dernier.



D'abord il y'a l'étape de « Preprocessing », où notre dataset composé de toutes les phrases des fichiers de training (data nlu) est restructurer/séparer (en tokens) pour récupérer les informations pertinentes à la prédiction. Ensuite on fait appel aux modèles de ML (défini dans le fichier config.yml) pour extraire les intents et les entités. Enfin l'étape de « Persistence » qui stock les résultats sous forme de document : « Interpreter Object ».

La sortie finale de la pipeline est ce qu'on appelle l'interpreter, un objet qui génère les fichiers contenant les résultats de chacun des modèles de ML. On retrouve parmi ces fichiers sauvegardés :

Trainingdata.json contenant l'objet training data (vu plus haut)

Entity\_synonyms.json contenant tous les synonymes rencontrés avec leurs valeurs d'entité correspondantes.

Intent\_classifier\_tensorflow\_embedding.pkl (composant nlp pour compter les occurrences des tokens)

Intent\_featurizer\_count\_vectors.pkl (composant nlp pour transformer un token en vecteur)

*NB : l'extension pkl signifie que le fichier a été créé par « pickle » un module python qui permet aux objets d'être sérialisés en fichiers sur disque puis désérialisés dans le programme au moment de l'exécution.*

### 3-Traiter les flagged

Après avoir lancer un training, il faut appliquer la 3<sup>ème</sup> étape du process : Corriger les erreurs de prédictions. On regroupe toutes les phrases mal prédites par l'IA dans un onglet « flagged conversations ».

On teste les phrases flaggées une par une en les envoyant à Beesy pour vérifier si l'IA a corrigé ses erreurs de prédictions après le training (maintenant que les phrases flaggées ont été ajouté au dataset de training).

- Si l'IA ne se trompe plus sur la détection d'entités/intent de la phrase, on peut la « Close », la retirer de l'onglet « flagged conversations »
- Si l'IA se trompe encore, on cherche la phrase en question dans nos fichiers de training et on double le nombre de phrases similaires existantes.

Si la phrase flaggée n'est toujours pas correctement prédite, on double son nombre d'occurrence dans les fichiers de training, on passe ainsi d'1 phrase à 2, de 2 à 4, de 4 à 8.

Au-delà de 8 phrases sans corriger l'erreur, on considère que c'est un **pattern**, et qu'il faudra trouver un autre moyen de le corriger.

*En doublant les phrases, il faut toujours garder en tête d'écrire des phrases possédant la même structure (le même pattern) mais qu'il faut néanmoins bien changer le contenu de la phrase. L'idée est de généraliser le plus possible les phrases de training.*

## C- Evaluation et analyse

Afin d'évaluer l'impact des modifications qui ont été apporté à l'IA, des scripts permettent d'avoir des résultats sur la prédiction d'Intent, et sur la prédiction d'entités.

Deux scripts ont été mis au point : Intent\_Results et Entity\_Error\_Prediction pour évaluer les prédictions du chatbot sur notre dataset de training. On peut ainsi vérifier si les phrases introduites dans les fichiers de trainings seront correctement détectées par l'IA si un utilisateur les écrit. De cette façon plus nos fichiers de trainings sont volumineux avec des phrases généralement distinctes, plus la prédiction du bot est précise.

*NB : Des clients canadiens utilisant Beesy, ont une différente manière de prendre rendez-vous : « Dis Beesy, peux-tu me cédule une réunion pour demain ? ». Après avoir ajouté des phrases contenant le verbe cédule dans le fichier d'intent : createactiononlinemeeting, le bot est à présent capable de correctement prédire ce genre de requête. L'idée est donc de gérer toutes les différentes manières de communiquer avec Beesy en les ajoutant au dataset de training.*

On cherche donc à savoir si le nombre d'itération d'une requête suffit à une bonne prédiction, ou s'il faut ajouter plus de phrases similaires au dataset de training.



## a-Script Intent Result

On prend toutes les phrases présentes dans chaque fichier, on les fait passer sur l'IA et on regarde si l'intent prédit par l'IA correspond bel et bien au fichier de training.

On écrit les résultats sur un fichier csv, on récupère d'une part l'intent prédit par l'IA et l'intent que nous avons défini. On les compare, si les intents sont similaires, cela signifie que la prédiction est correcte, on ajoute alors un « OK » dans la colonne de résultat. Si l'intent prédit ne correspond pas à l'intent défini dans les fichiers, on note « NOK » dans la colonne de résultats.

```
def collect_nlu_intent(intent_results): # pragma: no cover

    #Build a list of the values we want to retrieve
    intent_values = [{"text": r.message, "intent": r.target, "prediction": r.prediction, "confidence": r.confidence}]
    csv.register_dialect('myDialect', quoting=csv.QUOTE_ALL, skipinitialspace=True)
    csvfile = csv.writer(open("intent_results.csv", "w"), dialect='myDialect')
    csvfile.writerow(["Confidence", "Target", "Prediction", "Resultat", "Text"])
    result = 0

    for r in intent_results:
        #Distinguish when the AI prediction is correct
        if r.prediction == r.target:
            result = "OK"
        else :
            result = "NOK"
        csvfile.writerow([r.confidence, r.target, r.prediction, result, r.message])
```

## b-Entity error Script (Annexe 2)

Nous devons aussi évaluer la capacité du chatbot à détecter les entités. Encore une fois nous allons récupérer toutes les phrases du dataset de training sous formes de texte (sans tagger les entités), on les envoie à l'IA 3 et on vérifie si toutes les entités présentes dans la phrase ont été correctement détecter.

On souhaite construire un tableau qui pour chaque phrase, affiche l'entité prédite et l'entité qui aurait dû être prédite. On construit une 4<sup>ème</sup> colonne qui représente le résultat de la comparaison.

**Définition des résultats :**

```
def check_result(prediction, target):

    if prediction == target :
        if target == "vide":
            return 'not in sentence'
        else :
            return 'OK'
    elif prediction != target :
        if prediction == 'vide' :
            return 'NOT PREDICTED'
        if target == 'vide':
            return 'Wrongly_Predicted'

        for word in target:
            if word in prediction :
                return 'MissPredicted'

        else :
            return 'Wrongly_Predicted'
```

**Not Predicted (NP) :** Ce qu'on a entraîné (targetté) et qui n'a pas été prédit par l'IA

**MissPredicted (MP) :** Ce que l'IA a correctement prédit mais qui n'est pas exactement écrit de la même manière que dans nos fichiers de training (ça arrive souvent avec les articles avant les deadlines qui ne sont pas forcément prédit par l'IA alors que la date en elle-même a bien été prédite)

**Ex :** (ce qui a été tagger dans les fichiers de training : pour la semaine prochaine ; et ce qui a été prédit par l'IA : la semaine prochaine)

**Wrongly predicted (WP) :** Ce qui ne devait pas être prédit mais que l'IA a quand même prédit.

Ex : « [Isabelle](owner) doit rédiger les documents pour marion [d'ici demain] (deadline) », l'IA prédit marion comme owner au lieu d'Isabelle.

**OK :** Ce qui devait être prédit et que l'IA a correctement prédit.

Les chiffres intéressants pour l'évaluation de l'IA :

|  |                    |
|--|--------------------|
| Total de ce qui a été déclaré comme entité   | Nb Target declared |
| Déclaré comme entité mais pas correctement prédit  | NP + MP            |
| Non déclaré comme entité mais l'IA l'a prédit  | WP                 |
| Total de ce qui a été prédit par l'IA  | OK + WP + MP       |
| (Ce qui a été déclaré comme entité mais pas correctement prédit) / (tout ce qui a été déclaré comme entité). : NP+MP/nb Target | Recall             |
| (Ce qui n'a pas été déclaré comme entité et que l'IA a prédit) / (tout ce qui a été prédit). : WP/(OK+WP+MP)                   | Precision          |

Un des calculs le plus utilisé pour évaluer la performance d'un modèle de machine Learning est le F1-Score.

$$F1 - Score = 2 * (Recall * Precision) / (Recall + Precision)$$

*Plus le F1-score est élevé, plus notre modèle est performant.*

Ainsi après chaque modification apportée aux fichiers de trainings, on lance une évaluation et on construit via des tableaux dynamiques les classeurs Excel regroupant ses informations. On peut ainsi mesurer l'évolution de l'apprentissage du bot au long terme.

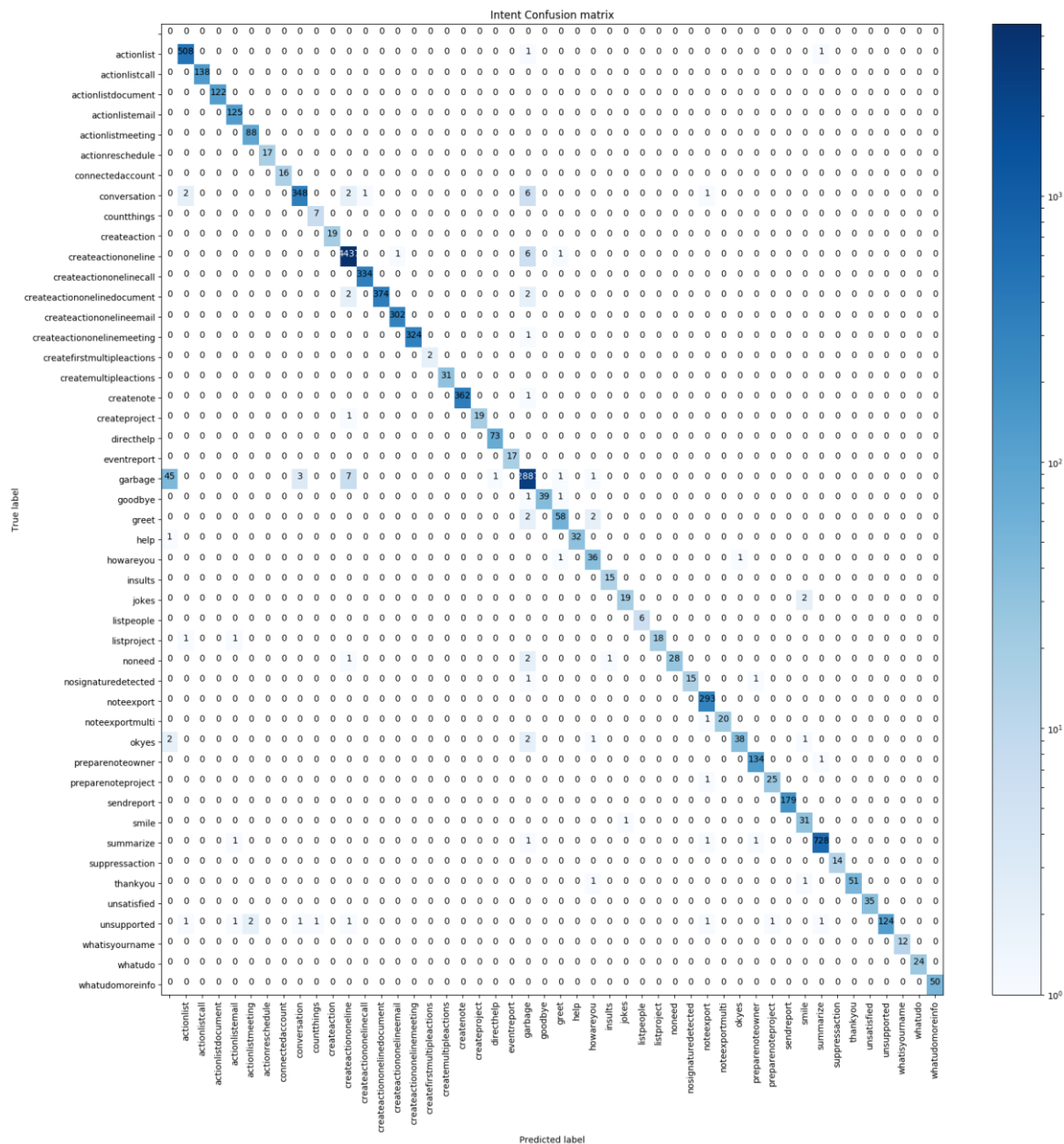
- **Multiclassification**

En utilisant les termes de machine Learning, l'intent classification est définie comme un apprentissage supervisé. Dans le sens où pour des entrées X1, X2, .., Xn (nos phrases), nous avons C1, C2, .. Cn classes pour catégoriser nos entrées (nos 48 fichiers d'intent). On parle donc de multiclassification puisque nos entrées sont prédites dans une des 48 classes possibles.

En ce qui concerne les entités, on parle de « Multi-label » puisque nos entrées (owner, deadline, project .. etc) peuvent être présentes dans n'importe quelle classe parmi les 48.

- **Confusion Matrix**

La matrice de confusion est un outil souvent utilisé lorsque l'on cherche à évaluer un modèle de machine learning. Pour un problème de multiclassification, c'est ce qui nous permet de savoir combien d'entrées ont été correctement prédites (Certes notre tableau croisé dynamique du script `intent_result` nous fournit aussi cette information, mais cela est clairement plus visible sur une matrice de confusion). Et le script d'évaluation fournit par Rasa construit directement cette matrice.



En ordonnée : Les entrées = Target = Ce qui a été entraîné = True Labels

En Abscisse : Prediction Labels = Ce qui est prédit par l'IA

Ainsi bien, la diagonale correspond à tout ce qui a été correctement prédit par l'IA.

## C-Gestion des patterns

Les modèles de Machine Learning qui permettent l'apprentissage de l'IA se basent sur des calculs mathématiques en rapport à la détection de pattern. C'est-à-dire que lorsque l'IA croise plusieurs phrases de training ayant la même structure pour une entité en particulier (comme des deadlines après chaque « pour le »), elle comprend que c'est un « pattern », une redondance. Par conséquent, lorsque l'IA rencontre une nouvelle phrase (écrite par un user) ayant une structure similaire, elle arrivera à prédire l'entité car elle a reconnu la structure sur laquelle elle a été entraîné.

De cette détection de patterns, découle un problème auquel il faut faire face : La dépatternisation. En effet l'IA a une facilité à reconnaître des patterns, si bien que pour certain cas de figure, elle n'arrive pas à faire la différence :

Exemple :

Si on entraîne l'IA sur des deadlines en fin de phrase :

- Envoyer la présentation de l'assistant [pour demain](deadline)
- Rédiger le guide support [d'ici demain] (deadline)

Lorsqu'elle rencontre une phrase contenant une date à la fin qui n'est pas un deadline :

- Faire le résumé du processus de training [avant mardi](deadline) à partir de ce qui a été vu à la réunion de jeudi

L'IA prédira « jeudi » comme deadline, alors que la bonne deadline est mardi.

➔ *Que faire dans ce genre de situation ?*

Il faut dépatterniser, c'est-à-dire ajouter un bon nombre de phrases contenant le pattern en entraînant l'IA sur les bonnes entités à prédire, afin qu'elle comprenne par exemple qu'une date en fin de phrase n'est pas forcément une deadline.

On ajoute autant de phrases contenant le pattern à dépatterniser, l'idée est de trouver le juste milieu entre ce qu'elle a déjà assimilé (deadline en fin de phrase) et ce qu'on cherche à lui faire apprendre (date normale en fin de phrase). On ne voudrait pas se retrouver à trop dépatterniser, à défaut de perdre ce qui est déjà assimilé (ne plus reconnaître les deadlines en fin de phrase).

En effet trop dépatterniser revient à créer un nouveau pattern, comme dans l'exemple ci-dessus, si on entraîne l'IA que sur des dates normales en fin de phrase, on aura créé un pattern de dates normale qui empêchera la détection de deadline en fin de phrase.

## IV – Training en grande quantité

### a-Generator

Pour justement gérer la dépatternisation sur des patterns qui ont un trop grand nombre de phrase à contrer, on utilise des méthodes qui permettent de générer un grand nombre de phrases.

- Construire les modèles Excel :

A partir du fichier Excel « Training Generator FR/EN », on peut reproduire la structure d'une phrase. On retrouve des parties de phrases dans chaque feuille de classeur (verbes d'action, owner, date etc.). Il suffit de créer un modèle pour le pattern qu'on cherche à contrer.

Exemple :

« Faire le devis pour Antoine » : Antoine, ici, n'est pas l'owner de l'action.

Nous avons détecté un pattern où l'IA prédit des « owner » dès que la phrase contient un prénom précédé d'un « pour » (sans doute après l'avoir entraîné sur des phrases du type : « action pour [Chris](owner) » etc ..).

Le but est donc de construire un modèle pouvant générer des phrases avec des prénoms après un « pour » sans que ce soit des owners.

➔ Choisir Feuilles de classeur : Action text + pour + ownernoentity

Il est très simple de générer des milliers de phrases avec Excel, il suffit uniquement de glisser vers le bas nos colonnes, voilà pourquoi il faut être très vigilant à ne pas créer de pattern en entraînant sur trop de phrase d'un modèle en particulier.

Pour éviter la création de nouveaux patterns : il faut entraîner par paquet de 100 phrases générés puis voir les résultats, si le pattern n'est pas corrigé, entraîner avec des paquets de 200 phrases, puis des paquets de 500.

Il est préférable d'utiliser différentes sortes de modèle pour contrer un même pattern (**toujours généraliser le training avec différentes structures de phrases**).

## b-Script Extractor :

Beesapps possède une base de données contenant environ 6,7 millions de phrases. L'idée est de récupérer ces phrases en les catégorisant directement par langue et type d'actions afin de pouvoir entraîner l'IA avec. Le gros avantage de ce processus est qu'on utilise des phrases qui ont été écrites par de vrais utilisateurs. Contrairement au générateur Excel, ces phrases seront beaucoup plus diverses, permettant un training encore plus généralisé.

### ➔ Comment fonctionne le script ?

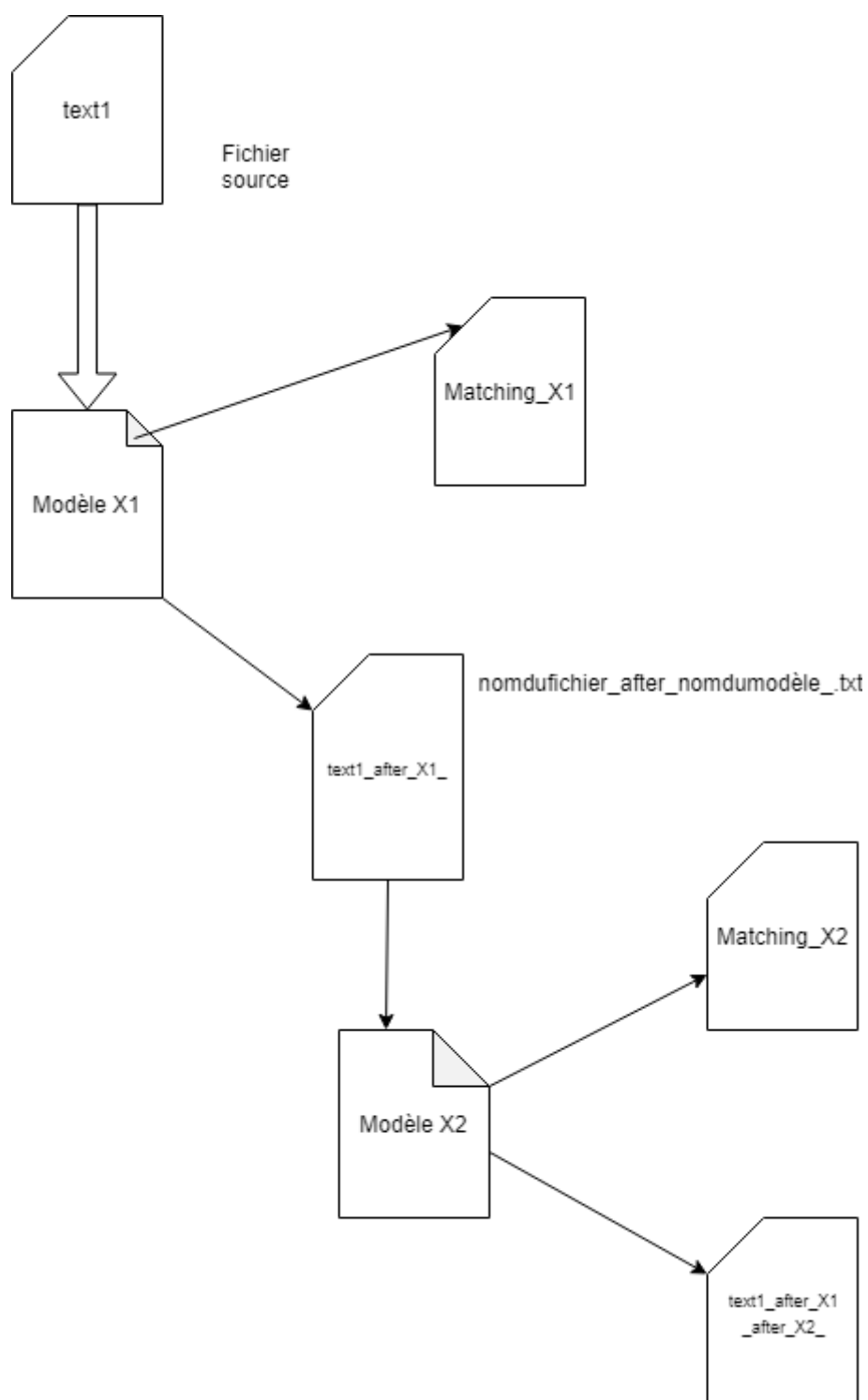
On récupère les phrases des utilisateurs de beesy stockées dans la base de données principale de Beesapps par paquet de 50 000. Ces phrases contiennent des actions online ainsi que du Garbage, des calls, des emails etc .. dans différentes langues. En effet l'application existe depuis maintenant presque 8 ans (7 mois après la sortie de l'iPad), un peu plus de 280 000 projets et 570 000 notes ont été écrites. Il faut les filtrer, on va donc construire différents « modèles » qui agiront comme des filtres pour chaque type d'action, de langue (beesapps existe en une vingtaine de langues), d'entités à extraire de notre fichier de phrases source.

Tout cela est dynamique, A partir d'un fichier à filtrer (50 000 phrases), on passe sur le premier modèle et on récupère 2 nouveaux fichiers :

- un fichier qui récupère toutes les phrases contenant les mots présents dans un modèle en particulier (Matching\_nomdumodèle)
- un autre fichier contenant toutes les phrases du fichier à filtrer hormis celles se trouvant dans le fichier Matching\_nomdumodèle (c'est le fichier extract, on extrait les phrases qui match le modèle X du fichier à filtrer).

Le deuxième modèle passe ainsi sur le fichier extrait du premier modèle, afin de ne pas repasser sur les phrases qui ont déjà matché le 1<sup>er</sup> modèle.

## Schéma récapitulatif du Script



Puisqu'on veut récupérer des phrases pour entraîner l'IA EN, on commence déjà par filtrer les langues :

Le premier modèle est le modèle chinois, on extrait tout ce qui est écrit en chinois du fichier à filtrer, et le fichier extrait devient le nouveau fichier à filtrer du modèle suivant : le modèle japonais, qui à son tour crée un nouveau fichier extrait. Après ces 2 modèle, le fichier extrait ne contient ni du chinois ni du japonais. Ainsi de suite, on fait passer autant de modèle que l'on souhaite, en obtenant à chaque fois un fichier contenant les phrases similaires à celle du modèle en question et un fichier extrait contenant les phrases restantes du fichier à filtrer.

Nous voulons récupérer des phrases à copier-coller directement dans nos fichiers de training. Voici les modèles que l'on utilise :

- 01 – Chinois
- 01 – Japonais
- 02 – Arabe
- 02 – grec
- 02 – Russe
- 02 – Thaï
- 02 – Turc
- 03 – Dutch
- 03 – Allemand
- 03 – Norvégien
- 03 – Slovénien/Tchèque/Polonais (Autres langues qui ne sont pas énormément utilisées)
- 04 – Italien
- 04 – Portugais
- 05 – Espagnol
- 06 – Français
- 08 – Name → Afin de récupérer toutes les phrases anglaises susceptible de contenir un owner
- 09 – Date → Afin de récupérer toutes les phrases anglais susceptible de contenir une deadline
- 21 – RDV → Afin de récupérer toutes les phrases d'action de type réunion
- 22 – DOC → Afin de récupérer toutes les phrases d'action de type document
- 23 – CALL → Afin de récupérer toutes les phrases d'action de type appel
- 24 – EMAIL → Afin de récupérer toutes les phrases d'action de type email



- 25 – AMBIGU → Afin d'extraire toutes les phrases ayant un verbe d'action mais susceptible d'être du Garbage
- 26 – ACTION → Afin de récupérer toutes les phrases contenant des verbes d'action oneline (**On peut copier-coller le fichier matching\_25ACTION directement en createactiononeline puisqu'il n'y a pas d'entités à tagger**)
- 27 – DO → Beesy ajoute parfois des « do : » au début des phrases que les utilisateurs écrivent, on souhaite extraire ces phrases pour pouvoir supprimer les « Do ».

**Après être passé sur tous ces modèles, le dernier fichier extract contient uniquement des remarques, que l'on peut copier-coller directement dans le fichier « Garbage ».**

*NB : Plus les modèles sont riches, plus nous sommes sûrs de bien filtrer les phrases de la base de données, il y a donc toujours intérêt à perfectionner les modèles.*

➔ Comment perfectionner les modèles ?

Il suffit tout simplement de lancer le script, et vérifier que les fichiers extracts ne contiennent pas de phrases correspondantes à un modèle en particulier. Si on croise une phrase qui n'a pas été filtrée, on rajoute les mots qui composent cette phrase comme nouvelles lignes du modèle correspondant. Car le cœur du script est de comparer chaque ligne du modèle avec celle du fichier à filtrer, et si les mots du modèle sont « inclus » dans la phrase du fichier à filtrer, c'est une phrase qu'il faut extraire. Ainsi bien, plus nos modèles sont volumineux, plus les matching\_files seront volumineux et plus les fichiers extracts seront petits de taille.

Le script a été codé en python et est composé de 4 fonctions principales.

Fonction qui permet de dynamiquement passer du fichier à filtrer au fichier extrait, et d'appeler chaque modèle pour le même fichier.

```
def dynamic_search(path_models,path_file):
    debug = 1
    for i,file in enumerate(gen1) :
        if debug:
            print(file)
        file_extract = file
        for c,models in enumerate(sorted(gen2)) :
            create_matching_file(models,file_extract)
            #unique_match(models,file_extract)
            create_extract_file(models,file_extract)
            if debug:
                print("this is the file at the beginning of loop {0} : {1}\n".format(c,file_extract))

            file_extract = "ex_"+models.replace(".txt","_")+file_extract

            if debug:
                print("this is the extract file : {0}\n after modele : {1}\n".format(file_extract,models))

        if debug :
            print('Out of file loop {}'.format(i))
```

Fonction qui permet de créer le fichier Matching regroupant toutes les phrases contenant les mots du modèle.

```
def create_matching_file(modele,file_to_filter):
    #Create a file where we'll append the matching sentences after each model
    debug = 0
    open('Match/matching_'+modele.replace(".txt","_")+file_to_filter, 'w').close()

    with open("Files/"+file_to_filter,"r") as file2: #file that you want to filter
        for sentence in file2:
            already_written = False
            with open("Models/"+modele,"r") as model:
                for line_model in model:
                    if already_written == False:
                        if write_matching_sentence(line_model,sentence,modele,file_to_filter) == "Found Match":
                            if debug:
                                print(line_model)
                                print("This is the sentence that matches the model : {}\n".format(sentence))
                            already_written = True
```

Fonction qui permet de faire la comparaison entre la phrase du fichier à filtrer et la ligne du modèle. On transforme les mots de la phrase et du modèle en « set de mot » et si le set de mots du modèle est contenu dans le set des mots de la phrase, on écrit la phrase dans le fichier Matching en question.

```
def write_matching_sentence(line_model,sentence,modele,file_to_filter):

    debug = 0
    mots_clés = [x for x in line_model.strip("\n").split(" ")]
    mots_phrase = [y for y in sentence.strip("\n").split(" ")]
    if debug:
        print("those are our words in model and in file_to_filter split and turned into a list :\n model : {}".format(mots_clés))
    if set(mots_clés).issubset(mots_phrase) == True:

        with open('Match/matching_'+modele.replace(".txt","_")+file_to_filter,"a") as f:
            f.write("{}\n".format(sentence.lower().strip("\n")))
        return "Found Match"
```

Fonction qui permet de créer le fichier extract.

```
def create_extract_file(modele,file):
    debug = 0
    #create a file that will store the original file_to_filter without sentences of matching_file
    open("Files/ex_"+modele.replace(".txt","_")+file, 'w').close()
    with open("Files/"+file,"r") as file2: #file that you want to filter
        for sentence in file2:
            with open('Match/matching_'+modele.replace(".txt","_")+file,'r') as matching_sentences:
                for line in matching_sentences:
                    if debug:
                        print("this line completes the model requirements : {} \n".format(line))
                    if line.strip("\n").lower() == sentence.strip("\n").lower():
                        sentence = sentence.replace(sentence,"")

            with open("Files/ex_"+modele.replace(".txt","_")+file,'a') as new_content :
                new_content.write(sentence.lower())
```

Résultats au 04/07/2019 :

Sur 10 000 phrases, on arrive à récupérer 15 % de Garbage et environ 20% d'action oneline sans entités que l'on peut directement copier-coller au dataset de training.

## V- Conclusion

### a- Etat de l'étude et apport pour l'entreprise

J'ai pu apprendre beaucoup de chose dans un domaine aussi intéressant que l'IA conversationnel. Je suis aussi ravi d'avoir pu contribuer au lancement du chatbot anglais, ce qui m'a permis de voir la différence entre entrainer un chatbot qui possède déjà un dataset important et un chatbot qui n'a pas encore de training data. Il a fallu dans un premier temps traduire les fichiers training data FR en anglais en vérifiant que les nlu\_synonyms concordent avec ceux de chaque fichier de training. Nous avons utilisé la sortie du translator fournit par google comme entrée à un très court script qui permet de justement remplacer les mots traduit par google ne correspondant pas aux valeurs instanciées au domaine (domain.yml) anglais. En effet il faut que les entités et synonymes traduits et ceux définis dans la configuration du modèle anglais concordent.

A mon arrivée, pour entrainer l'intent Garbage, il fallait récupérer des rapports Word sur google susceptible de contenir beaucoup de Garbage et les envoyer par mail au bot pour que ces phrases apparaissent sur la plateforme de conversations à valider. L'IA FR possédait 30 000 phrases au total, il y en a maintenant un peu plus de 50 000, et on peut récupérer directement une infinité de Garbage depuis l'extracteur. On peut aussi récupérer des actions basiques (sans entités) avec l'extracteur, cela a d'ailleurs permis d'entrainer l'IA EN (un modèle extracteur anglais a aussi été mis au point), de tel sorte à pouvoir augmenter le nombre de « training samples ».

### b- Suite de l'étude

Le script extractor doit encore être améliorer, l'idée est de récupérer des phrases déjà taguées (respectant la nomenclature) afin de directement les ajouter au dataset de training. Il faudra dans un premier temps isoler toutes les phrases contenant des entités - certaines entités sont précédées de mots spécifiques (pour une deadline : avant le, d'ici, pour le etc ..) – en ajoutant une autre condition dans l'algorithme pour gérer l'ordre des mots contenus dans les modèles. Pour l'instant on vérifie si les « sets » de mots du modèle sont inclus dans les « sets » de mots de la phrase à filtrer, ils peuvent ainsi être à différentes positions de la phrase, il faudrait donc ajouter la possibilité de comparer des suites de mots ou des sets de mots en fonction des modèles. Pour les distinguer on peut exécuter la fonction qui gère des suites de mots uniquement si le modèle possède un symbole particulier (si le symbole n'apparaît pas dans le nom du modèle, on procède normalement via les sets de mots). Dans un second temps, il faudra arriver à ajouter des crochets et le nom de l'entité correspondant au modèle. Pour cela il faut arriver à détecter la position de l'entité dans la phrase, ou peut être directement prendre le token qui suit les mots définis dans le modèle. Néanmoins, cela reste problématique pour les phrases pouvant contenir plusieurs entités.

# ANNEXES



**Fiche d'évaluation "Entreprise" à compléter par le maitre de stage**  
à insérer dans le rapport de stage

**Fiche d'évaluation du stage**  
**Cycle Ingénieur • 2<sup>ème</sup> année**  
**2018/2019**

Etudiant : Louanes HAMLA

Majeure : SI

Entreprise : BeesApps

Tuteur de stage : Christopher MESQUITA

☎ du tuteur : +33 6 14 62 22 16

Email du tuteur : christopher.mesquita@beesy.com

| Possibilité d'accorder des ½ points   | Note                                     |
|---|--|
| <b>Technique</b> <ul style="list-style-type: none"> <li>• Difficulté de l'étude</li> <li>• Résultats obtenus / respect du CDC</li> <li>• Qualité du travail (méthodologie, réalisation, dossier ...)</li> </ul> <b>Total Note technique</b>   | 4/4<br>5/6<br>8/10<br><b>17/20</b>       |
| <b>Qualités humaines</b> <ul style="list-style-type: none"> <li>• Autonomie et sens des responsabilités</li> <li>• Esprit d'initiative et créativité</li> <li>• Intégration / sociabilité au sein de l'équipe</li> <li>• Comportement / Attitude</li> </ul> <b>Total note Qualités Humaines</b> | 6/7<br>3/5<br>5/5<br>3/3<br><b>17/20</b> |
| <b>TOTAL</b>  | <b>34/40</b>                             |
| Soit  | 17/20                                    |

Observations :

Louanes a parfaitement su répondre aux attentes techniques du stage. Son comportement a été irréprochable, son intégration parfaite et rapide.

A Paris, le 09/09/2019

Signature du Tuteur de stage  
(obligatoire)

*Christopher Mesquita*

Cachet de l'entreprise d'accueil  
(obligatoire)

BeesApps SAS  
Capital de 23000€ Véronique sur seine  
N° 547 793 TVA FR 07 538747703

## Annexe 1A Intents utilisés par Beesy

- Connected Account → *Renvoie le compte avec lequel l'User est connecté*
- Create multiple action ( ) → *Crée plusieurs actions à partir d'une liste d'action*
- Directhelp → *Redirige vers [support@beesapps.com](mailto:support@beesapps.com) lorsque l'utilisateur demande de parler à une vraie personne. Ex : « Je souhaite communiquer avec un être humain »*
- Goodbye → Ex: « bye beesy »
- Greet → Ex: «Salut »
- Help → *Identifie que l'User a besoin d'aide «En quoi puis je vous aider ? »*
- Howareyou → Ex : « Comment vas-tu »
- Insults
- Jokes → Ex : « Raconte-moi une blague »
- Noneed → Ex : « Tu ne sers à rien »
- Noteexport → *Envoie une note par mail Ex : « envoie-moi la note partenariat Microsoft »*
- Noteexportmulti → *Envoie plusieurs notes par mail Ex : « envoie-moi mes dernières notes »*
- Okyes → Ex : « ok cool »
- Preparenoteproject → *Prépare la réunion d'un projet, renvoie une note regroupant toutes les actions liées à ce projet.*
- Sendreport → *Envoie un rapport journalier/hebdomadaire (un suivi)*
- Smile → Ex : « haha » ; « Lol » Beesy répond qu'il est content de vous avoir fait rire
- Summarize → *Envoie rapport d'action / bilan d'activité d'une personne /projet par mail*
- Thankyou → Ex : « Merci »
- Unsatisfied → Ex : « Tu ne sers à rien »
- Unsupported → *Renvoie que l'action demandé n'est pas encore gérée par Beesy*
- Whatisyourname → Ex : « Comment tu t'appelles ? »
- Whatudo → Ex : « Que sais tu faire ? » Beesy répond par la liste des tâches qu'il peut effectuer
- Whatudomoreinfo → Ex : « Soit plus précis » Beesy donne des exemples de phrases que l'utilisateur peut écrire

-Actionreschedule → *Décaler une liste d'action à une autre date Ex : "Reporte moi les actions de [louanès](owner) de [lundi](deadline) pour [la semaine prochaine](fromdate) »*

-Createproject → *Permet de créer un projet via conversation Ex : « Crée moi le projet [beesy ia](project) »*

-Nosignaturedetected → *Demande à l'User de configurer sa signature mail*

*NB : Lorsque les utilisateurs envoient des mails à beesy, il faut ajouter une mise en forme particulière à sa signature afin que beesy sache que c'est la fin du mail et qu'il ne doit prendre en considération que ce qui se trouve avant cette mise en forme. (Voir procédure ajouter sa signature dans le cahier de réponse du support)*

-Preparenoteowner → *Prépare une réunion individuelle*

*NB : Prépare une note regroupant toutes les actions en communs/assignés de l'utilisateur et de la personne avec qui on souhaite faire la réunion.*

*Ex : « prépare-moi la réunion individuelle avec Paul »*

## Annexe 1B Liste des entités

- Owner → Propriétaire de l'action
- Project → Le nom d'un projet
- Projecttemplate → Modèle sur lequel se baser pour créer un nouveau projet
- Goal → Nom d'un objectif
- Deadline → Echéance, Date à ne pas dépasser
- Fromdate → Date à laquelle on décale une action
- Channel → Via quelle channel l'User souhaite communiquer (Ex : email)
- Note → Nom d'une note
- Notetemplate → Modèle sur lequel se baser pour créer une nouvelle note
- Nbelement → Nombre d'élément que l'utilisateur a indiqué ; Ex : « donne moi les [10](nbelement) derniers [commentaires] (objecttype:commentaire)
- Actiontype → Type de l'action (appel, document, audio, dessin etc ..)
- Actioncategory → Catégorie de l'action : Urgente, terminée, dépassée
- Objecttype → Type d'objet que l'utilisateur demande (Note, projet, commentaire, action)
- Reporttype → Type de rapport (rapport journalier, rapport hebdomadaire)
- Orderby → Utilisé pour afficher ce que l'utilisateur demande de manière trié (trié par note, par sujet, par priorité, par propriétaire ou par opportunité)
- Actionall → Permet de détecter si l'User veut toutes les actions (terminées, dépassés et en cours)
- Sendto → Personne à qui envoyer le bilan d'activité dans l'intent summarize
- Searchtext → Recherche ce mot en particulier parmi toutes les actions ; Ex : « quelles sont mes actions contenant le mot [client](searchtext)
- Createddate → Date à laquelle l'action a été créée ; Ex : « quelles sont les actions créées [aujourd'hui](createddate)
- Ownerspecific → Permet de détecter si l'User veut les actions de tout le monde dans un projet ; Ex : « Les actions [de tout le monde](ownerspecific : pour tout le monde) »



## ANNEXE 2

## Entity Error Script

```
def EvaluateEntitiesPrediction(interpreter, test_data):
    """Runs the model for the test set and extracts entity
    predictions and tokens.
    """
    debug = 0
    with open("bavage.csv", "w", newline='') as filee :
        csv.register_dialect('myDialect', quoting=csv.QUOTE_ALL, skipinitialspace=True)
        writer = csv.writer(filee, dialect='myDialect')
        head = []
        head.append("Text")
        head.append("Wrong Prediction")
        head.append("Entity predicted")
        head.append("Entity targeted")

        writer.writerow(head)
    with open('entity_error_script2.csv', 'w') as file :
        csv.register_dialect('myDialect', quoting=csv.QUOTE_ALL, skipinitialspace=True)
        csvfile = csv.writer(file, dialect='myDialect')
        entity_predictions, tokens = [], []
        check_entity = {}
        header = []

        header.append("Text")
        header.append("Intent classification")
        list_entities_model = getListOfEntitiesFromModel('domain_fr.yml')
        for entity in list_entities_model:
            header.append(entity + " : Target")
            header.append(entity + " : Prediction")
            header.append(entity + " : Result")

        csvfile.writerow(header)
        list_wrong_predictions=[]
        list_targets=[]

        for sentence in test_data.training_examples:
            if debug:
                print(sentence.data['intent'])
            row=[]
            row.append(sentence.text.replace(" ", "").replace("-", "").replace(".", "").replace("@", "").replace(
                res = interpreter.parse(sentence.text, only_output_properties=False)
                extracted_entities = extract_entities(res)
                row.append(sentence.data['intent'])
                for entity in header[2:] :
                    if "Target" in entity:
                        target_entity = entity
                        a=get_target(sentence,entity)
                        #Create List of dict to keep track of what are the targets in all our entities
                        list_targets.append({"target":a, "entity":entity})
                        row.append(a)
                    elif "Prediction" in entity:
                        b=get_prediction(extracted_entities,entity)
                        row.append(b)
                    elif "Result" in entity:
                        row.append(check_result(b,a))
                        # Find Wrong predictions and store them into a List of dict to keep track of the entity
                        if check_result(b,a) == 'Wrongly Predicted':
                            list_wrong_predictions.append({"target":a,"Wrong prediction":b,"entity": entity})
                if list_wrong_predictions:
                    writer.writerow(check_bavage(list_wrong_predictions,list_targets,sentence))
                    #Reset the Lists to empty to make place for next sentences, otherwise it will append on them
                    list_wrong_predictions = []
                    list_targets = []
                csvfile.writerow(row)
```

```
def get_target(sample,entity):
    value = 0
    entity = entity.replace(' : Target','')
    #print(sample.data.keys())
    if "entities" in sample.data.keys():
        dict_target_trained = {x['entity']:x['value'] for x in sample.data["entities"]}
        #print(dict_target_trained)
        if entity in dict_target_trained.keys():
            return dict_target_trained[entity].replace(" - ","-").replace("' ','").replace("@","").replace("/ " / "
        else :
            return 'vide'
    else :
        return 'vide'

def get_prediction(sample,entity):
    value=0
    entity = entity.replace(' : Prediction','')
    dict_entities_extracted = {x['entity']:x['value'] for x in sample} #Create dictionary of entities extracted
    if entity in dict_entities_extracted.keys() :
        #print(entity, dict_entities_extracted,dict_entities_extracted.keys())
        #print(dict_entities_extracted[entity])
        return dict_entities_extracted[entity].replace(" - ","-").replace("' ','").replace("@","").replace("/ " / "
    else :
        return 'vide'
```

# Architecture Réseau

