UMONS
Université de Mons

Faculté des Sciences

Projet de premier master en sciences informatiques
Rapport

Prototype de chatbot assistant l'écriture de code propre exploitant les grands modèles de langage et le traitement du langage naturel

---

Élève :
Nicolas Gatta

Directeur de projet :
Stéphane Dupont

Année académique 2023 - 2024

Table des matières

1 Introduction

Ce projet en sciences informatiques se déroule dans le cadre de la première année du diplôme de master en sciences informatiques à finalité spécialisée en Artificial Intelligence and Data Analytics. Les objectifs principaux de ce projet sont de permettre d'apprendre à se documenter, de s'informer, de savoir gérer son temps, d'apprendre de nouveaux concepts et outils ainsi que de faire preuve d'indépendance.

Le thème choisi pour ce projet est intitulé "Intelligence Artificielle pour les interfaces utilisateurs de type chatbot". Ce thème a pour but de se concentrer sur le développement d'un logiciel permettant à l'utilisateur de parler à l'aide d'une interface de saisie à l'ordinateur.

Ce thème offrant un éventail vaste de possibilités, il a été décidé que ce projet allait se concentrer sur le développement d'un chatbot complété d'une interface avec la capacité d'aider l'utilisateur à corriger son code au niveau de la syntaxe ainsi que de l'aider à appliquer les principes du Clean Code. En plus de cela, il inclura des fonctionnalités telles que la tenue de conversations.

Considérant la quantité de documentation disponible en anglais dans le domaine de l'intelligence artificielle et des interfaces utilisateurs de type chatbot, il a été naturel de choisir cette langue pour le développement intégral du projet. Cette décision a simplifié la recherche d'informations pertinentes et de technologies utilisables pour les chatbots qui sont souvent publiées en anglais. Ainsi, développer le projet de zéro en anglais a permis d'exploiter au mieux les différentes documentations et de faciliter l'intégration de bibliothèques et d'outils de développement qui sont majoritairement conçus et documentés en anglais.

Les objectifs de ce travail vont être d'une part, d'approfondir les connaissances liées aux concepts d'intelligence artificielle et de chatbot, et d'autres par, d'améliorer les compétences liées à la conception et l'implémentation de code tout en respectant les principes de Clean Code. Ce rapport a pour objectif de fournir un aperçu du projet en permettant au lecteur de découvrir les tenants et aboutissants grâce à la division du rapport en plusieurs sections distinctes.

La section 2 a pour but d'aider le lecteur qui n'a pas les connaissances sur les concepts centraux de ce projet, de les présenter de manière détaillée et simple. La section 3 liste les technologies qui ont été nécessaires pour la réalisation de ce projet tout en y incluant des explications sur leur fonctionnement. La section 4 donne de manière graphique tout le processus de fonctionnement du logiciel ainsi que la structure de la base de données. La section 5 aborde la description complète du logiciel en passant par les détails techniques de l'implémentation des différentes technologies jusqu'au manuel de l'utilisateur. La section 7.2 fournit une description détaillée de la performance du logiciel en utilisant diverses métriques telles que la rapidité de réponse. Enfin, la section 9 conclut le travail réalisé en récapitulant de manière concise les différentes observations et en offrant des pistes permettant une amélioration possible du logiciel.

2 Préambule

L'objectif de cette section est de permettre aux lecteurs de se familiariser avec les concepts qui composent la pierre triangulaire de ce projet ainsi que de servir d'introduction à des éléments souvent négligés qui jouent pourtant un rôle crucial dans le monde du développement informatique. Les trois concepts qui composent la pierre triangulaire sont le Clean Code, la rigueur syntaxique dans le développement de logiciels et les chatbots.

2.1 Le Clean Code : L'art de la programmation propre

2.1.1 Introduction au Clean Code

Le Clean Code est un concept popularisé par l'ingénieur logiciel Robert C. Martin grâce à son livre "Clean Code : A Handbook of Agile Software Craftsmanship"[1]. Ce livre représente bien plus que de simples conventions de programmation, il incarne une philosophie que tout développeur logiciel devrait suivre. Il a pour but de placer la lisibilité, la simplicité et la maintenabilité au cœur de la conception des programmes. A la base de cette approche de la part de Robert C. Martin se trouve la conviction que le code source devrait non seulement permettre à un programme de fonctionner, mais également de permettre aux développeurs de facilement faire transiter l'information dans le présent ou le futur. De plus, il aurait pour effet d'améliorer la collaboration au sein des équipes, d'accélérer la détection d'erreurs et de faciliter la maintenance du logiciel tout au long de son cycle de vie.

2.1.2 Principes fondamentaux

Les principes du Clean Code orientent la production de code informatique vers une qualité supérieure et une compréhension plus aisée. En suivant les principes ci-dessous, les développeurs créent un code source qui favorise une communication claire et efficace entre les membres de l'équipe :

1. **Lisibilité**
   Le premier principe du Clean Code est la lisibilité. Le code doit être rédigé de manière à ce que sa compréhension soit immédiate. Cela passe par des noms de variables explicites, une indentation cohérente et une organisation logique du code.

2. **Simplicité et Clarté**
   Le deuxième principe du Clean Code est la simplicité et la clarté. Les solutions simples sont préférées aux solutions complexes. Il est entendu par là que tout un chacun doit pouvoir comprendre le code et non pas seulement l'auteur.

3. **Modularité**
   Le troisième principe du Clean Code est la modularité. Ce principe encourage la division de code en modules distincts. Chaque module doit avoir une responsabilité unique et bien définie qui permet ensuite de faciliter la réutilisation du code et permet une maintenance plus aisée.

4. **Éviter la redondance**
   Le quatrième principe du Clean Code est d'éviter la redondance. Ce principe va de pair avec le principe de modularité, il faut éviter à tout prix des morceaux de codes répétitifs et si le cas se présentait les encapsuler dans des fonctions ou des classes.

5. **Maintenabilité**
   Le cinquième principe du Clean Code est la maintenabilité. Les développeurs doivent pouvoir comprendre rapidement le fonctionnement du code existant et y apporter si possible des modifications sans introduire de bugs inattendus.

6. **Tests unitaires**
   Le sixième principe du Clean Code est les tests unitaires. Des tests bien écrits garantissent que le code fonctionne comme prévu et permettent de détecter rapidement un problème après une modification.

7. **Élégance**
   Le septième principe du Clean Code est l'élégance. Un code élégant est à la fois fonctionnel et agréable à lire, il montre l'habilité du développeur dans son art.

2.1.3 Exemple : Impacts du Clean Code sur la compréhension du code

Dans l'exemple ci-dessous, on peut constater que les noms des variables **x**, **y** et **result** ne reflètent pas de manière optimale la nature de ces variables dans la partie Listing 1 – Sans Clean Code. De plus, on peut aussi remarquer que le nom de la fonction **fn** n'est pas représentatif de ce que celle-ci effectue.

Pour corriger ce problème, il faut effectuer des changements sur les éléments cités ci-dessus pour respecter les principes du Clean Code et obtenir le résultat qui se trouve dans la partie Listing 2 – Avec Clean Code :

1. **x** → **nombre1**
2. **y** → **nombre2**
3. **result** → **somme**
4. **fn** → **additionner**

```
def fn(x, y):
result = x + y
return result
```

Listing 1 – Sans Clean Code

```
def additionner(nombre1, nombre2):
    somme = nombre1 + nombre2
    return somme
```

Listing 2 – Avec Clean Code

2.2 Rigueur syntaxique dans le développement de logiciels

2.2.1 Définition de la syntaxe

Avant de définir ce qu'est la rigueur syntaxique, il est important de définir ce qu'est la syntaxe. La syntaxe est définie comme des séquences et combinaisons de caractères nécessaires pour créer du code correctement structuré. En effet, la syntaxe ne fait que donner une structure et de l'ordre dans les instructions d'un code, mais celui-ci dépend grandement de la sémantique pour donner une signification à ces instructions. Cependant, la syntaxe est importante lors de la création de code et ne pas la respecter conduit à s'exposer à des erreurs de syntaxe qui vont bloquer le programme lors de son exécution.[2]

2.2.2 Définition de la rigueur syntaxique

De par la définition dans la section 2.2.1, la définition de la rigueur syntaxique découle tout naturellement. La rigueur syntaxique dans le contexte du développement logiciel fait référence à la conformité stricte du code à une syntaxe spécifique du langage de programmation actuellement utilisé. Il est question de respecter les règles et conventions établies pour s'assurer de la lisibilité, la maintenabilité et la compréhension du code.

2.2.3 Importance de la rigueur syntaxique

L'importance de la rigueur syntaxique se retrouve très fréquemment associée au Clean Code et pour cause, ils partagent tous deux énormément de points. En conséquence, il est difficile de les dissocier. La liste des éléments composant la rigueur syntaxique est ainsi très similaire à celle que l'on peut retrouver pour le Clean Code :

1. **Lisibilité**
   Une syntaxe claire et cohérente permet une plus grande aisance lors de la lecture du code et donc facilite la collaboration au sein de l'équipe de développement.

2. **Prévention des erreurs**
   La rigueur syntaxique contribue à prévenir de potentiels bugs liés à une mauvaise structure du code ou du moins permet de réduire les risques que cela se produise.

3. **Conformité aux standards**
   La rigueur syntaxique pousse les développeurs à respecter les conventions de codage recommandées par la communauté du langage en question.

4. **Facilitation des modifications**
   La rigueur syntaxique permet d'avoir un code bien formaté, ce qui permet aux développeurs de se repérer plus facilement dans le code pour opérer des modifications.

5. **Maintenabilité**
   La rigueur syntaxique permet une maintenabilité du code plus aisée et donc une réduction des coûts associés lors de l'évolution du programme.

2.2.4 Exemple : Impacts de la rigueur syntaxique sur la lisibilité du code

Lorsque la rigueur syntaxique est respectée, le code devient plus lisible et compréhensible pour les développeurs. Grâce aux exemples ci-dessous, la raison pour laquelle la rigueur syntaxique est si importante devient évidente. On peut voir les effets du non-respect de la syntaxe dans Listing 3 – Sans rigueur syntaxique. Il est plus compliqué de lire le code à cause d'une syntaxe un peu hasardeuse alors que dans Listing 4 – Avec rigueur syntaxique, le contenu est plus agréable visuellement.

```
def calculer_resultat(x, y,
    ↪ operation){
    resultat = 0
    if operation == 'add': resultat
        ↪ = x + y
    else:
        if operation == 'sous':
            ↪ resultat = x - y
        else:
            if operation == 'multi':
                ↪ resultat = x * y
            else: resultat = x / y
    return resultat
```

Listing 3 – Sans rigueur syntaxique

```
def calculer_resultat(x, y,
    ↪ operation):

    if operation == 'add':
        return x + y
    if operation == 'sous':
        return x - y
    if operation == 'multi':
        return x * y
    if operation == 'div':
        return x / y

    return Aucun
```

Listing 4 – Avec rigueur syntaxique

2.3 Les chatbots : La révolution des interactions Homme-Machine

2.3.1 Introduction aux chatbots

Un chatbot est un type d'intelligence artificielle qui incarne l'interaction entre l'homme et la machine. Fonctionnant sur le même principe qu'un programme informatique, un chatbot a pour but principal de créer et maintenir des conversations avec son utilisateur par le biais de texte ou de la voix. De plus, il fait preuve de la capacité de comprendre et de répondre dans une ou plusieurs langues à l'aide d'algorithme de traitement du langage naturel qui sera abordé plus en détail dans la section 3.1. [3]

2.3.2 La Révolution des interfaces Homme-Machine

L'intégration de l'intelligence artificielle dans notre vie quotidienne a doucement amené à une nouvelle révolution aussi appelée la quatrième révolution industrielle. L'un des points clés de l'entrée dans cette nouvelle révolution est l'apparition des chatbots qui permettent une interaction plus intuitive et facile entre l'utilisateur et la machine en éliminant les barrières techniques qui pouvaient exister. Maintenant, chacun peut se permettre d'utiliser des chatbots sans avoir de compétences techniques dans le domaine. Cela a été rendu possible grâce à deux éléments clés. Le premier élément est l'utilisation de technologie permettant à des programmes informatiques de mieux comprendre et répondre aux requêtes provenant du langage humain. Le deuxième élément est une interface simple pour permettre à monsieur et madame Tout-le-Monde d'utiliser ces nouvelles technologies.

2.3.3 Exemple : ChatGPT

Un exemple concret illustrant bien l'avènement des chatbots est la création et la mise à disposition de chatGPT. En effet, même si chatGPT est une nouvelle technologie, il reste très simple d'utilisation grâce à son interface plus que basique, mais remplissant sa fonction à merveille comme représenté sur la Figure 1 – Interface de chatGPT.

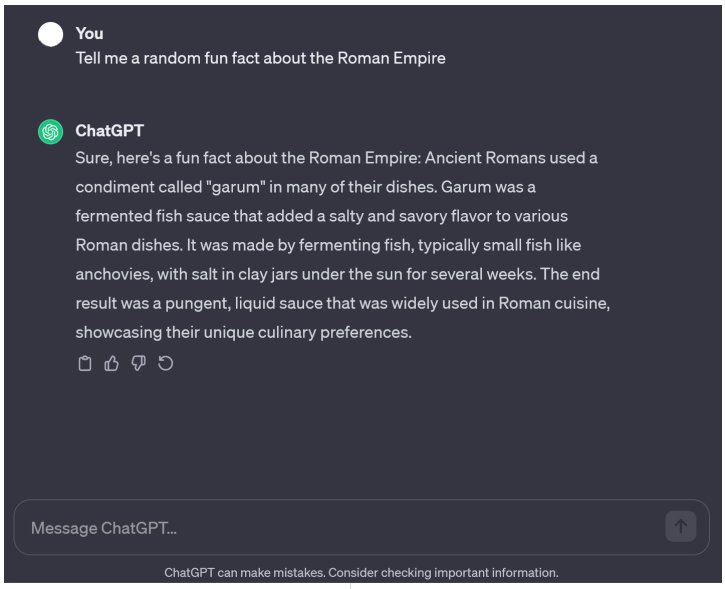


FIGURE 1 – Interface de chatGPT

2.4 Les grands modèles de langage : La nouvelle génération de chatbot

2.4.1 Introduction aux grands modèles de langage

Les grands modèles de langage ou Large Language Model (LLM) sont des systèmes d'intelligence artificielle qui sont développés pour comprendre, générer et interagir avec le langage humain. Ils ont pour but principal d'imiter la capacité humaine à comprendre et à produire du texte à tel point qu'il devient difficile de faire la différence entre l'humain et la machine. Ces modèles sont basés sur des architectures de réseaux de neurones qui leur permettent de traiter et d'apprendre de vastes quantités de données textuelles.
Un LLM peut être entraîné sur des ensembles de données immenses leur permettant d'apprendre de plus en plus du langage humain. Le fonctionnement d'un des grands modèles de langage nommé BERT sera abordé en détail dans la section 3.3.

2.4.2 L'impact des grands modèles de langage

L'arrivée des LLM a marqué une étape significative dans le domaine de l'intelligence artificielle. Ceux-ci ont apporté des améliorations notables dans la compréhension du langage naturel permettant ainsi des implications profondes pour divers secteurs tels que l'éducation, la technologie et le service client. Depuis maintenant quelques années, les LLM font partie intégrante de nos vies. En effet, de plus en plus de sites web mettent en avant l'utilisation d'assistant virtuel pour aider le client lors de ses achats en ligne. On peut citer des sites web comme Amazon, Facebook ou Google qui utilisent cette technologie pour offrir au client une expérience personnalisée et des conversations complexes.

2.4.3 Les architectures des grands modèles de langage

Dans le domaine des grands modèles de langage, on en trouve une vaste diversité, et il n'est pas exagéré de dire qu'un nouveau modèle émerge chaque jour. Chacun de ces modèles peut être classé dans l'une des trois catégories suivantes : [4][5][6]

— **Les décodeurs** : Les architectures de décodeur se concentrent principalement sur la génération de textes. Elles prennent en entrée une séquence de mots et effectue une prédiction pour le mot suivant dans la séquence en se basant uniquement sur les mots précédents. Cela est effectué jusqu'à générer un texte complet. L'un des portes étendard de cette architecture est le modèle GPT[1] qui a été développé par OpenAI. Ce modèle est l'un des LLM les plus avancés et il a été largement discuté pour sa capacité à générer du texte d'une qualité proche de celle d'un humain.
Il est utilisé dans diverses applications, allant de la rédaction automatique à la création de contenu et à l'assistance conversationnelle.

1. GPT : Generative Pre-trained Transformer

— **Les encodeurs** : Les architectures d'encodeur analysent et transforment le texte en vecteur de caractéristiques qui capturent le sens des mots dans leur contexte. Contrairement aux décodeurs, les encodeurs ne sont pas conçus pour générer du texte mais pour comprendre et encoder les informations. Le modèle le plus connu de ce type d'architecture est le modèle BERT[1] qui a été créé par Google.
Le modèle BERT a révolutionné la façon dont les moteurs de recherche comprennent les requêtes linguistiques en améliorant significativement la pertinence des résultats de recherche basé sur le contexte du langage naturel.

— **Les encodeur-décodeurs** : Ces architectures combinent les fonctions des encodeurs et des décodeurs pour effectuer des tâches complexes de transformation de texte comme de la traduction automatique. Le texte d'entrée est d'abord encodé pour former une représentation intermédiaire qui sera par la suite décodée en une nouvelle séquence de texte. Le modèle qui représente au mieux cette architecture est le modèle T5[2] qui est développé par Google. Le modèle T5 interprète toutes les tâches de traitement du langage comme un problème de transformation de texte en texte. Cela lui permet de réaliser des tâches telles que la traduction, la correction et le résumé de textes.

1. BERT : Bidirectional Encoder Representations from Transformers
2. T5 : Text-To-Text Transfer Transformer

3 Technologies

Cette section vise à expliquer les principes de fonctionnements des différents composants exploités tout au long de ce projet. C'est une partie cruciale du document qui permet d'explorer en détails les différentes architectures, technologies et frameworks intégrés dans le projet.

3.1 Natural Language Processing

3.1.1 Introduction

Le Natural Language Processing (NLP) ou Traitement Automatique du Langage Naturel (TALN) en français est une sous-discipline du domaine mêlant intelligence artificielle et linguistique. Cette technologie est dédiée à la compréhension des mots, phrases ou expressions dans les langues humaines par un ordinateur.

Un langage naturel est un langage qui est parlé ou écrit par des êtres humains dans un but de faire transiter des informations entre eux. Ce terme est apparu grâce à l'apparition d'utilisateurs voulant communiquer avec un ordinateur sans pour autant apprendre le langage spécifique de la machine.

On peut conclure que la technologie du NPL est apparue pour aider les personnes qui n'ont pas les connaissances qui permettent l'interaction avec un ordinateur en utilisant le langage machine, mais qui possèdent des connaissances assez poussées dans un langage humain que pour communiquer avec une autre personne, qui est dans ce cas remplacée par un ordinateur. [7]

3.1.2 Les niveaux de langage dans le Natural Language Processing

La compréhension du traitement du langage naturel (Natural Language Processing, NLP) peut être une tâche complexe en raison de la diversité des éléments linguistiques impliqués. Il est donc très souvent utile de décomposer le langage en ses principales composantes linguistiques. Cette approche permet de gérer plus efficacement le traitement du langage et de faciliter l'interaction entre ses différents aspects.

Cette manière de procéder provient des recherches réalisées dans le domaine de la psycholinguistique[1] qui suggèrent que le traitement du langage se fait de manière dynamique. L'un des exemples les plus parlants pour illustrer le dynamisme d'une langue réside dans la présence de mots aux sens multiples. Grâce à la décomposition en différentes composants linguistiques[2], un sens peut être privilégié par rapport aux autres. Il est essentiel de comprendre ce que sont ces différents composants linguistiques utilisés par la technologie NLP. Il est important de noter que le premier composant qu'est la phonologie n'est utilisée que dans le cas d'une utilisation vocale du NLP. [8]

1. Psycholinguistique : La psycholinguistique est un domaine de recherche relativement récent qui se donne pour objectif de mettre au jour les mécanismes impliqués dans l'utilisation du langage plus spécifiquement dans la production, la compréhension et l'acquisition du langage. [9]
2. Composant linguistique : Partie fondamentale du langage qui aide à comprendre et à analyser sa structure et son fonctionnement

1. **Phonologie**
   Ce premier niveau concerne l'interprétation des sons provenant de la parole telle que les mots. Il utilise trois règles qui sont respectivement la phonétique[1], le phonème[2] et la prosodie[3].

2. **Morphologie**
   Ce deuxième niveau se concentre sur l'analyse de la structure et la formation des mots qui est un domaine essentiel pour la maîtrise des conjugaisons, des accords et des mots composés. Par exemple, le mot pré-enregistrement peut être analysé en deux parties distinctes : la première qui est "pré" qui signifie "avant" et la deuxième qui est "enregistrement" qui est "l'action d'enregistrer". On peut aussi prendre l'exemple du verbe "parler" sous différentes formes tel que "je parle", "tu parlais" et "il parlera". Ces variations illustrent comment la conjugaison adapte le verbe selon le temps et la personne. En ce qui concerne les accords, ils sont visibles quand des adjectifs ou des articles s'accordent en genre et en nombre avec les noms qu'ils déterminent comme dans la phrase "la voiture noire".

3. **Lexique**
   Ce troisième niveau se concentre sur l'étude des mots pris individuellement pour leur attribuer une signification précise. Cependant, il est important de noter que cette étape peut associer plusieurs sens à un même mot, il faudra utiliser le contexte dans lequel le mot est utilisé pour permettre de lever toute ambiguïté de manière définitive.

4. **Syntaxique**
   Ce quatrième niveau se concentre sur l'analyse des mots qui composent une phrase pour en comprendre la structure grammaticale. Cela résulte en une représentation de la phrase qui met en évidence les relations entre les mots.

5. **Sémantique**
   Ce cinquième niveau permet de déterminer les significations possibles d'une phrase en se concentrant sur les interactions entre les significations des mots dans la phrase. Cela permet de désambiguïser[4] le sens de certains mots grâce au contexte de la phrase et permettre de fixer un sens au mot qui semble logique pour la représentation de la phrase.

6. **Discours**
   Ce sixième niveau comparé au niveau syntaxique et sémantique œuvre sur des textes plus longs qu'une phrase. Il se concentre sur les propriétés du texte dans son ensemble et le sens que celui-ci essaie de transmettre.

7. **Pragmatique**
   Ce septième niveau se concentre sur l'utilisation intentionnelle de certains types de langage dans des situations bien précises et utilise le contexte global du contenu pour établir une meilleure compréhension du texte.

3.1.3 Comment fonctionne le Natural Language Processing

La technologie du Natural language Processing utilise trois grands processus qui sont respectivement le prétraitement des données, l'extraction de caractéristiques et la modélisation.

La première étape est le prétraitement des données. Avant qu'un modèle ne traite le texte pour une tâche spécifique, il est nécessaire de prétraiter les données pour améliorer les performances du modèle. On retrouve une variété de techniques diverses qui peuvent être utilisées lors de ce prétraitement :

1. **Analyse syntaxique**
   L'analyse syntaxique est un processus qui a pour but de déterminer le début et la fin de chaque phrase dans le document. Elle est grandement utilisée pour la segmentation de texte qui permet de diviser un texte en un ensemble de phrases.

2. **Racinisation/Lemmatisation**
   La racinisation et la lemmatisation sont deux processus linguistiques utilisés pour simplifier les mots dans le traitement du langage naturel. La racinisation convertit les mots en leur forme de base alors que la lemmatisation va plus loin en prenant en compte le contexte grammatical et sémantique pour ramener les mots à leur forme canonique.

3. **Analyse sémantique**
   L'analyse sémantique vise à comprendre la signification des mots et des phrases dans leur contexte spécifique. Cela inclut l'interprétation des expressions idiomatiques[1], savoir à qui ou à quoi se réfèrent les mots comme "il" ou "elle", et voir comment différentes idées ou phrases sont liées entre elles.

4. **Analyse morphosyntaxique**
   L'analyse morphosyntaxique consiste à identifier quel rôle chaque mot joue dans une phrase, comme déterminer s'il s'agit d'un nom, d'un verbe ou d'un adjectif. Ce processus aide à comprendre comment les mots sont connectés et organisés pour former des phrases qui ont du sens.

5. **Suppression des mots vides**
   La suppression des mots vides est un processus qui permet d'enlever les mots qui n'apportent que peu d'informations sur le texte telles que "un", "le", "une", etc.

6. **Tokenisation**
   La tokenisation est un processus qui consiste à décomposer un texte en segments plus petits comme des mots, des morceaux de phrases ou des caractères individuels appelés tokens. Une fois l'obtention des différents tokens, ceux-ci reçoivent un identifiant unique pour faciliter le traitement informatique du texte.

1. Phonétique : Partie de la linguistique qui étudie les sons de la parole.
2. Phonème : Unité distinctive de prononciation dans une langue.
3. Prosodie : Ensemble des traits oraux d'une expression verbale d'un locuteur.
4. Désambiguïser : Faire disparaître l'ambiguïté.

En prenant comme exemple la Figure 2 – Fonctionnement de la tokenisation. On commence avec des données de bases qui sont composées d'un ensemble de textes appelés corpus : "This is a sentence" et "This is another big sentence". Une fois les textes dans le système, les phrases sont séparées en une liste de mots uniques auxquelles sont assignés un identifiant unique à chacun (rectangle de gauche). Ensuite, il est possible de transformer les phrases du corpus en une liste de chiffres dans laquelle chaque chiffre représente un mot et la liste complète des chiffres représente la phrase (rectangle de droite).

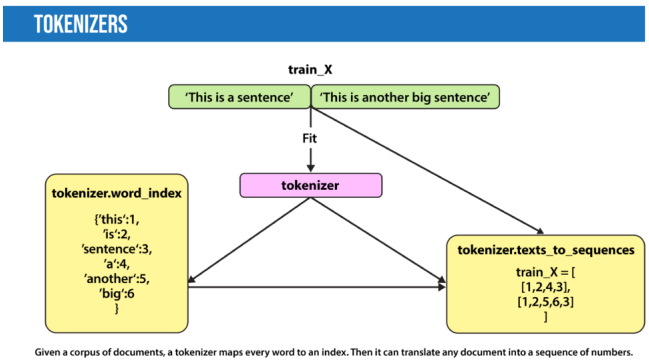


FIGURE 2 – Fonctionnement de la tokenisation [10]

La deuxième étape est l'extraction de caractéristiques. Elle s'effectue à la suite de la tokenisation du texte pour obtenir les séries de chiffres qui décrivent le texte. Il est ensuite possible de trouver des caractéristiques en utilisant différentes méthodes telles que le sac de mots ou le TF-IDF.

1. **Sac de mots**
   La méthode du sac de mots consiste à compter le nombre d'occurrences d'un mot ou d'un groupe de mots dans un document. Pour cela, la tokenisation doit être réalisée au préalable, comme cela devrait déjà être effectué dans la partie prétraitement. Ensuite, on procède au décompte du nombre d'apparitions de chaque mot dans la phrase comme illustré dans la Figure 3 – Fonctionnement du sac de mots.

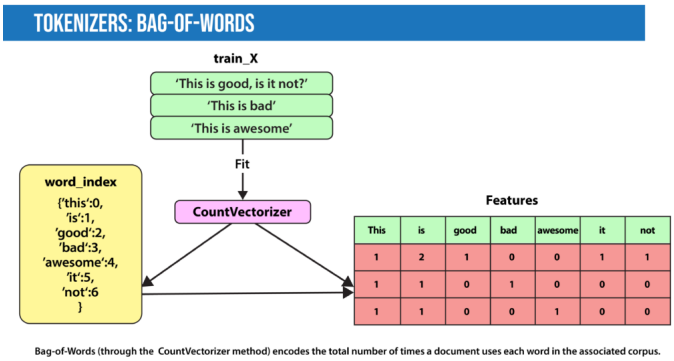


FIGURE 3 – Fonctionnement du sac de mots [10]

1. Expression idiomatique : Phrase où les mots ensemble ont un sens différent de ce qu'ils disent littéralement

2. **TF-IDF**
La méthode du TF-IDF (Term Frequency-Inverse Document Frequency) affine l'approche du sac de mots en intégrant une pondération spécifique pour chaque mot, augmentant ainsi la précision de la représentation textuelle. Le TF-IDF est composé de deux termes. Le premier terme TF (Term Frequency) qui mesure la fréquence d'un mot précis dans un document ce qui est calculé par le nombre de fois qu'un mot apparaît dans ce document divisé par le nombre total de mots dans le document. Le deuxième terme IDF (Inverse Document Frequency) sert à diminuer l'importance des mots qui apparaissent fréquemment dans les documents pour ainsi réduire leur impact. Il est calculé en faisant le logarithme du nombre total de documents divisé par le nombre de documents contenant le mot en question. Pour calculer le TF-IDF il suffit juste de multiplier le TF et IDF comme représenté sur la Figure 4 – Fonctionnement du TF-IDF.

Term Frequency (TF) :

$$\mathrm{TF}(t,d) = \frac{\text{Nombre de fois que le terme } t \text{ apparaît dans le document } d}{\text{Nombre total de termes dans le document } d} \tag{1}$$

Inverse Document Frequency (IDF) :

$$\mathrm{IDF}(t,D) = \log\left(\frac{\text{Nombre total de documents } |D|}{\text{Nombre de documents contenant le terme } t}\right) \tag{2}$$

TF-IDF :

$$\text{TF-IDF}(t,d,D) = \mathrm{TF}(t,d) \times \mathrm{IDF}(t,D) \tag{3}$$

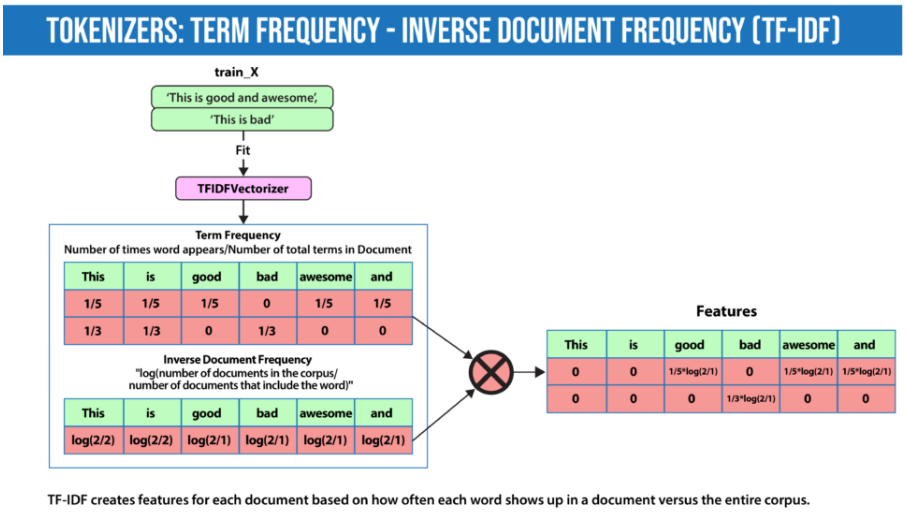


FIGURE 4 – Fonctionnement du TF-IDF [10]

La troisième et dernière étape concerne l'application de modèles d'apprentissages tels que les réseaux neuronaux, les SVM[1] ou des méthodes utilisant des statistiques pour permettre la classification, la traduction automatique ou encore la génération de textes. [10] [11]

1. **Réseau de neurones artificiels**
   Le réseau de neurones artificiels est une structure adaptative inspirée de la nature, plus précisément du fonctionnement du cerveau humain. Celui-ci est composé d'éléments appelés neurones[2] qui utilisent la plasticité[3] de manière similaire à sa contrepartie humaine, ce qui permet la modification de l'agencement des connexions pour se spécialiser dans une tâche précise. Le réseau de neurones fonctionne en traitant des entrées qui passent successivement à travers plusieurs couches de neurones avant de produire une sortie, comme sur la Figure 5 – Fonctionnement du réseau de neurones. [12]

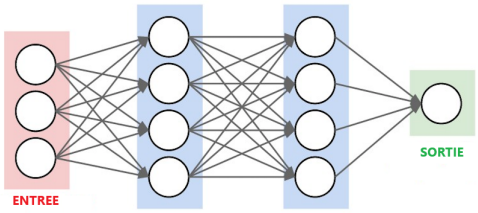


FIGURE 5 – Fonctionnement du réseau de neurones [13]

2. **SVM**
   Les machines à vecteurs de support (SVM) sont des modèles d'apprentissage supervisé fréquemment utilisés pour la classification. Le principe des SVM est de chercher un hyperplan[4] qui sépare de manière optimale les échantillons. Malencontreusement, il est très commun que les données ne soient pas linéairement séparables, c'est-à-dire qu'il n'existe pas d'hyperplan qui puisse les diviser parfaitement. Pour résoudre ce problème, les SVM utilisent une technique appelée le kernel trick. Cette méthode permet de projeter les données dans un espace de dimension supérieure où elles deviennent séparables par un hyperplan. Cette projection se fait avec des fonctions de noyau qui permettent de traiter des relations complexes entre les échantillons sans avoir besoin de les transformer explicitement dans un espace de dimension supérieure tel que représenté sur la Figure 6 – Fonctionnement du Kernel Trick. [14]

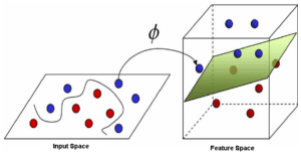


FIGURE 6 – Fonctionnement du Kernel Trick [15]

1. SVM ou Support Vector Machine : Algorithme de classification utilisant les vecteurs.
2. Neurone : Élément spécialisé dans le traitement, la réception et la transmission d'informations.
3. Plasticité : Capacité du cerveau à changer, se remodeler, se réorganiser, dans le but de s'adapter à de nouvelles situations. [16]
4. Hyperplan : Objet qui permet de séparer un espace en deux parties distinctes.

3.1.4 Comment représenter les caractéristiques des données ?

Dans le domaine du traitement du langage naturel, il est crucial de bien représenter les caractéristiques extraites des ensembles de données pour qu'elles soient traitées et comprises par les algorithmes. Quatre approches de représentation des caractéristiques des données existent : les représentations denses, les représentations creuses, les représentations contextuelles et les représentations non-contextuelles. En pratique, il est possible de combiner ces différents types de représentations pour créer un hybride tels que des représentations denses non-contextuelles ou des représentations denses contextuelles. [17] [18] [19]

— Les représentations denses (Dense Representations) désignent des vecteurs continus où chaque dimension encode une caractéristique. Dans ce type de représentation, chaque composante du vecteur contient explicitement une valeur. Par exemple, une représentation d'un groupe de personnes d'âges différents est considéré comme dense, car il est peu probable d'avoir des valeurs manquantes. Il est possible de voir la représentation dense sous forme de matrice dans la partie gauche de la Figure 7 – Représentation dense et sparse

— Les représentations creuses (Sparse Representations) désignent des vecteurs dans lesquels les éléments sont principalement des valeurs nulles, mais l'où on ne stocke que des valeurs explicites non nulles. Ce type de représentation est particulièrement utile pour gérer des données de grandes dimensions qui sont naturellement creuses, comme les données textuelles où chaque dimension pourrait représenter la présence ou l'absence d'un mot spécifique dans un dictionnaire. Il est possible de voir la représentation dense sous forme de matrice dans la partie droite de la Figure 7 – Représentation dense et sparse

— Les représentations contextuelles sont des vecteurs de caractéristiques générés de manière à ce que le sens d'un mot ou d'une caractéristique soit influencé par son contexte immédiat. Ces représentations sont dynamiques et peuvent changer en fonction de la phrase ou du texte dans lequel l'élément est utilisé. Ce type de représentation est souvent utilisé par les modèles de langage comme BERT pour permettre une meilleure analyse du contexte dans le cas du traitement du langage naturel.

— Les représentations non contextuelles sont statiques et ne varient pas en fonction du contexte. Chaque instance d'un mot ou d'une caractéristique a toujours la même représentation indépendamment de son utilisation dans différentes phrases. Des technologies telles que Word2Vec ou GloVe sont des exemples de représentations denses non contextuelles.

FIGURE 7 – Représentation dense et sparse [17]

3.2 Analyseur syntaxique

3.2.1 Introduction

Dans le domaine de l'informatique, le parseur également connu sous le nom d'analyseur syntaxique est un composant essentiel présent dans la plupart des compilateurs.[1] Sa fonction principale est d'analyser les données en entrée qui sont sous la forme d'instructions séquentielles provenant du code source puis de repérer les erreurs potentielles à l'aide de divers mécanismes. [20]

3.2.2 Comment fonctionne un analyseur syntaxique

L'analyseur syntaxique se compose de trois éléments, chacun gérant une étape différente du processus d'analyse. Ces trois étapes sont les suivantes :

1. **L'analyse lexicale**
   L'analyse lexicale est un composant essentiel dans le processus de compilation d'un langage de programmation. Il a pour but de segmenter le code source en petits morceaux appelés jetons ou tokens pour ensuite permettre une classification de chaque élément. Chaque jeton est une brique élémentaire de la grammaire du langage comme les mots clés, les identifiants, les opérateurs et les littéraux. Ceux-ci sont essentiels pour que le compilateur comprenne et traite le code de manière appropriée comme illustré sur la Figure 8 – Classification des tokens. De plus, l'analyse lexicale va supprimer tous les éléments non essentiels, les commentaires et les caractères d'espacement.

FIGURE 8 – Classification des tokens [20]

2. **L'analyse syntaxique**
   L'analyse syntaxique a pour but de valider si la séquence de jetons respecte les règles syntaxiques du langage. Pour cela, l'utilisation d'un arbre syntaxique est essentielle et permet la représentation de la structure du code comme présenté sur la Figure 9 – Arbre syntaxique. Cet arbre permet une visualisation des différentes relations entre les jetons tout en respectant les règles grammaticales du langage. Si le code source enfreint ces règles, des erreurs de syntaxe sont signalées pour indiquer les parties du code qui doivent être corrigées.

1. Compilateur : Un programme qui traite les instructions écrites dans un langage de programmation donné pour les traduire en langage machine

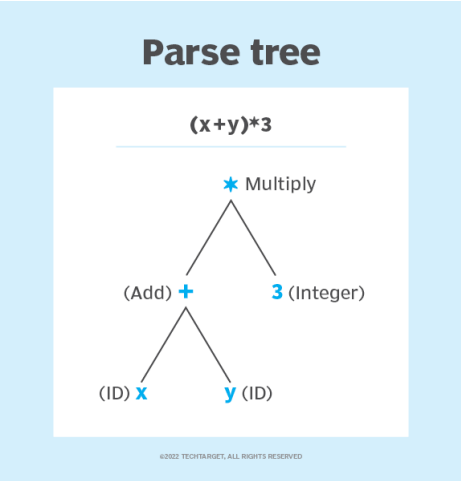


FIGURE 9 – Arbre syntaxique [20]

3. **L'analyse sémantique**
   L'analyse sémantique vérifie la cohérence de l'arbre de dérivation avec la sémantique du langage en effectuant des tâches telles que la vérification des types, la résolution des variables et la détection d'erreurs sémantiques. Elle garantit que les variables sont déclarées avant leur utilisation et que les fonctions sont appelées correctement. Au final, son objectif est de détecter les erreurs non capturées par l'analyse syntaxique et qui empêcherait le programme de s'exécuter correctement.

3.2.3 Quels sont les différents types d'analyseur syntaxique ?

Pour accomplir les différentes tâches de l'analyseur syntaxique, il existe différentes méthodes qui vérifient les étapes dans différents ordres. Il existe deux principaux types d'analyseur syntaxique :

1. **L'analyseur descendant (top-down) :**
   Les analyseurs descendants commencent le processus en partant de la règle la plus générale dans la grammaire du langage de programmation et descendent jusqu'aux symboles terminaux.
2. **L'analyseur ascendant (bottom-up) :**
   Les analyseurs ascendants commencent le processus en partant des symboles terminaux dans la grammaire du langage de programmation et remontent jusqu'à la règle la plus générale.

Il existe deux types de variation possible pour les analyseurs syntaxiques :

1. **L'analyseur LL (Left-to-Right, Leftmost Derivation) :**
   L'analyseur LL parcourt l'entrée de gauche à droite en utilisant une dérivation la plus à gauche. Cela signifie qu'à chaque étape de l'analyse, l'analyseur choisit la règle la plus à gauche pour étendre l'arbre de dérivation.
2. **L'analyseur LR (Left-to-Right, Rightmost Derivation) :**
   L'analyseur LR analyse également l'entrée de gauche à droite, mais en utilisant une dérivation la plus à droite. Cela signifie qu'il choisit toujours la règle la plus à droite lorsqu'il étend l'arbre de dérivation.

3.3 BERT : Bidirectional Encoder Representations from Transformers

3.3.1 Introduction

Dans le domaine du traitement du langage naturel, une révolution a doucement fait son apparition. Le modèle dénommé BERT développé par Google a permis une avancée significative dans ce domaine en permettant une compréhension plus profonde et contextuelle du langage humain. Contrairement aux modèles précédents qui traitent les mots séquentiellement, BERT analyse les mots dans le contexte de la phrase. BERT permet de saisir des nuances complexes et des dépendances syntaxiques entre les mots tout en offrant une compréhension du texte bien plus riche.

3.3.2 Comment fonctionne BERT

Le modèle BERT fonctionne grâce à quatre mécanismes : la bidirectionnalité qui permet de comprendre le langage, l'architecture des Transformers qui traite les mots dans leur contexte global, le NSP qui permet de prédire la phrase suivante et le fine-tuning qui est une étape permettant au modèle BERT de se spécialiser sur une tâche précise. [21] [22] [23]

Le premier mécanisme utilisé dans le modèle BERT est la bidirectionnalité. La majorité des modèles avant l'arrivée de BERT traitaient le texte de manière séquentielle de gauche à droite. BERT quant à lui essaye de comprendre le contexte d'un mot en tenant compte des mots voisins à celui-ci. À cette fin, le modèle utilise une technique appelée "Masked Language Model" (MLM) qui est utilisée durant l'entrainement du modèle. Celle-ci consiste à dissimuler un mot dans une phrase pour inciter le modèle à utiliser les mots de part et d'autre du mot dissimulé pour prédire le mot caché en utilisant les deux directions de gauche à droite et de droite à gauche. Il est possible de voir un exemple de ce fonctionnement sur la Figure 10 – MLM "how are you doing today" dans laquelle le mot "you" a été masqué et le but du modèle est de retrouver quel mot aurait la plus grande probabilité de s'y trouver en utilisant les mots autour pour déterminer un contexte.

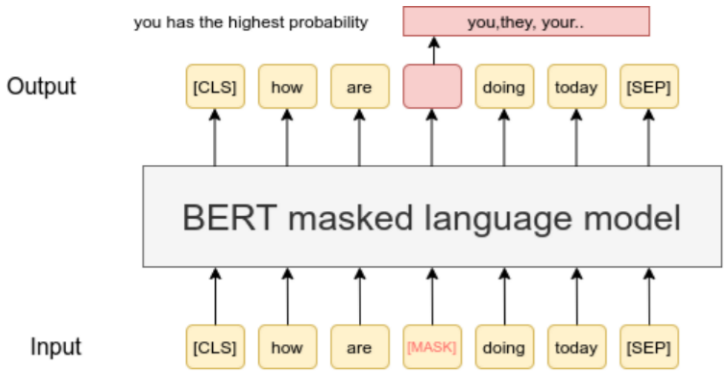


FIGURE 10 – MLM "how are you doing today" [21]

Le deuxième mécanisme central du modèle BERT repose sur les architectures des Transformers qui sont conçues spécifiquement pour traiter des séquences de données.
Ces architectures utilisent des mécanismes d'attention pour évaluer l'importance relative et le contexte de chaque mot dans une phrase tout en améliorant la compréhension globale du contexte et réduisant les ambiguïtés. L'architecture typique d'un Transformer est exposée sur la Figure 11 – La structure codeur-décodeur de l'architecture Transformer et est décrite ci-dessous :

1. **Inputs et Input Embedding** : Les entrées sont converties en vecteurs denses non-contextuels. Ces vecteurs permettent au modèle de traiter efficacement les informations.
2. **Positional Encoding** : Comme les Transformers ne traitent pas les données séquentiellement, ils utilisent un "encodage positionnel" pour tenir compte de la position des mots dans la phrase.
3. **Multi-Head Attention** : Cette couche permet au modèle de se concentrer sur différentes parties de la phrase pendant qu'il traite l'information. Chaque "tête" d'attention peut se concentrer sur différentes relations entre les mots, comme leur contexte ou leur signification grammaticale.
4. **Add & Norm** : Après chaque couche d'attention et chaque couche suivante, une opération d'addition (Add) est effectuée pour combiner les informations de différentes couches, suivie d'une normalisation (Norm) pour stabiliser l'apprentissage du réseau.
5. **Feed Forward** : Application de transformations linéaires et de non-linéarités à chaque position d'entrée.
6. **Output Embedding et Output Probabilities** : Les sorties sont finalement transformées en probabilités à l'aide de la fonction softmax, qui aide à déterminer le mot le plus probable suivant dans la séquence.

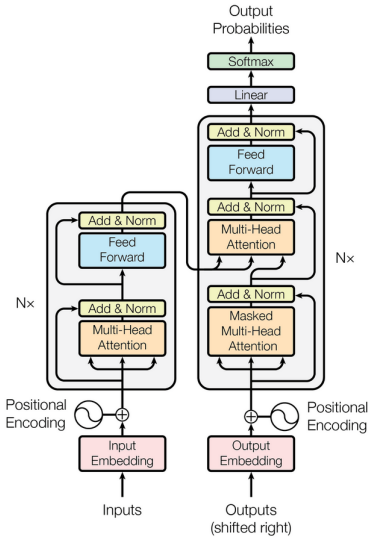


FIGURE 11 – La structure codeur-décodeur de l'architecture Transformer

Le troisième mécanisme présent dans le modèle BERT est le "Next Sentence Prediction" (NSP) qui fait partie des mécanismes d'entraînement du modèle. Ce mécanisme a été conçu pour permettre au modèle de comprendre la relation entre deux phrases, ce qui est crucial pour des tâches telles que la réponse aux questions de l'utilisateur. Ce mécanisme fonctionne en fournissant durant l'entraînement du modèle des paires de phrases et le modèle doit alors prédire si la seconde phrase dans la paire est la suite logique de la première phrase. Pour environ 50% des paires d'apprentissage, la seconde phrase est effectivement la suite de la première et pour les autres 50%, la seconde phrase est juste une phrase au hasard. Grâce à l'exemple de la Figure 12 – Exemple de paire de phrases, on peut observer que des tokens ont été rajoutés pour que le modèle sache où se trouve le début avec le [CLS] ainsi qu'un token [SEP] pour marquer la fin d'une phrase.

FIGURE 12 – Exemple de paire de phrases

Le quatrième mécanisme est le fine-tuning qui est une étape cruciale pour spécialiser un modèle BERT pré-entraîné. Initialement, BERT est entraîné sur un large corpus de texte tel que Wikipédia pour récolter le maximum d'informations. Cela permet de comprendre la langue de manière générale grâce à des tâches comme la prédiction de mots masqués et la prédiction de la prochaine phrase. Lors du fine-tuning, il est nécessaire d'utiliser un modèle de base qui est un modèle BERT dans notre cas, auquel il va falloir ajouter une dernière couche pour ainsi permettre une spécialisation du modèle pour une ou des tâches spécifiques comme de la classification de texte, de la reconnaissance d'entités nommées, ou de la génération de réponses aux questions des utilisateurs. Par exemple, pour une tâche de classification, on ajoute une couche qui va être spécialement utilisée pour la tâche en question puis le modèle est entraîné sur un ensemble de données. Durant l'entraînement, le modèle ajuste les différents paramètres internes pour optimiser la performance sur cette tâche précise. Ce processus permet au modèle BERT de transférer les connaissances acquises lors de l'entraînement initial à des applications plus ciblées tout en améliorant sa précision et son efficacité dans des domaines spécialisés.

4 Modélisation conceptuelle

La section sur la modélisation conceptuelle vise à représenter de manière compréhensible les différentes étapes de conceptions d'un logiciel qui seront ensuite présentées à d'autres personnes qui ne sont pas nécessairement sensibles à la technologie.

4.1 BPMN (Le Business Process Model and Notation)

4.1.1 Qu'est-ce qu'un Business Process Model and Notation ?

Le BPMN est une norme de modélisation des processus métier[1] qui fournit une base graphique commune à toute une équipe. Cela permet aux différentes personnes composant l'équipe d'avoir une base commune pour comprendre, documenter et optimiser le processus métier. [24]

4.1.2 Représentation du projet en schéma BPMN

Le schéma présenté dans la Figure 13 – Schéma BPMN du logiciel illustre le fonctionnement de l'application développée pour ce projet. Elle est divisée en deux principales composantes : le back-end[2] est responsable du traitement des entrées utilisateur, et le front-end[3] fournit une interface intuitive pour faciliter l'interaction avec le chatbot.

Lors du démarrage du programme, celui-ci attend que l'utilisateur interagisse avec l'interface. Une fois les entrées confirmées par l'utilisateur, celles-ci sont affichées dans l'interface et envoyées au back-end pour traitement.

Dans le back-end, les entrées sont analysées pour décider si elles doivent être divisées en plusieurs segments qui subiront diverses transformations tels que la suppression des mots vides et de la ponctuation. Après ces modifications, le programme extrait les caractéristiques des entrées pour prédire la catégorie de réponse que le chatbot devrait utiliser. Si le programme ne trouve pas de réponse adéquate, un message d'excuses est envoyé à l'utilisateur. Si une réponse est identifiée, le programme détermine l'action correspondante pouvant être une réponse standard ou nécessite une analyse plus approfondie du code pour signaler des erreurs de syntaxe ou suggérer des améliorations en rapport avec le Clean Code. Le processus se termine lorsque l'utilisateur décide de quitter l'application.

1. Processus métier : une activité ou un ensemble d'activités mis en place pour réaliser une tâche, un projet ou atteindre un objectif [25]
2. Front-end : Partie visible de l'application avec laquelle les utilisateurs interagissent, comprenant l'interface utilisateur.
3. Back-end : Partie de l'application qui opère en arrière-plan, gérant les données et les logiques métier nécessaires pour traiter les requêtes des utilisateurs

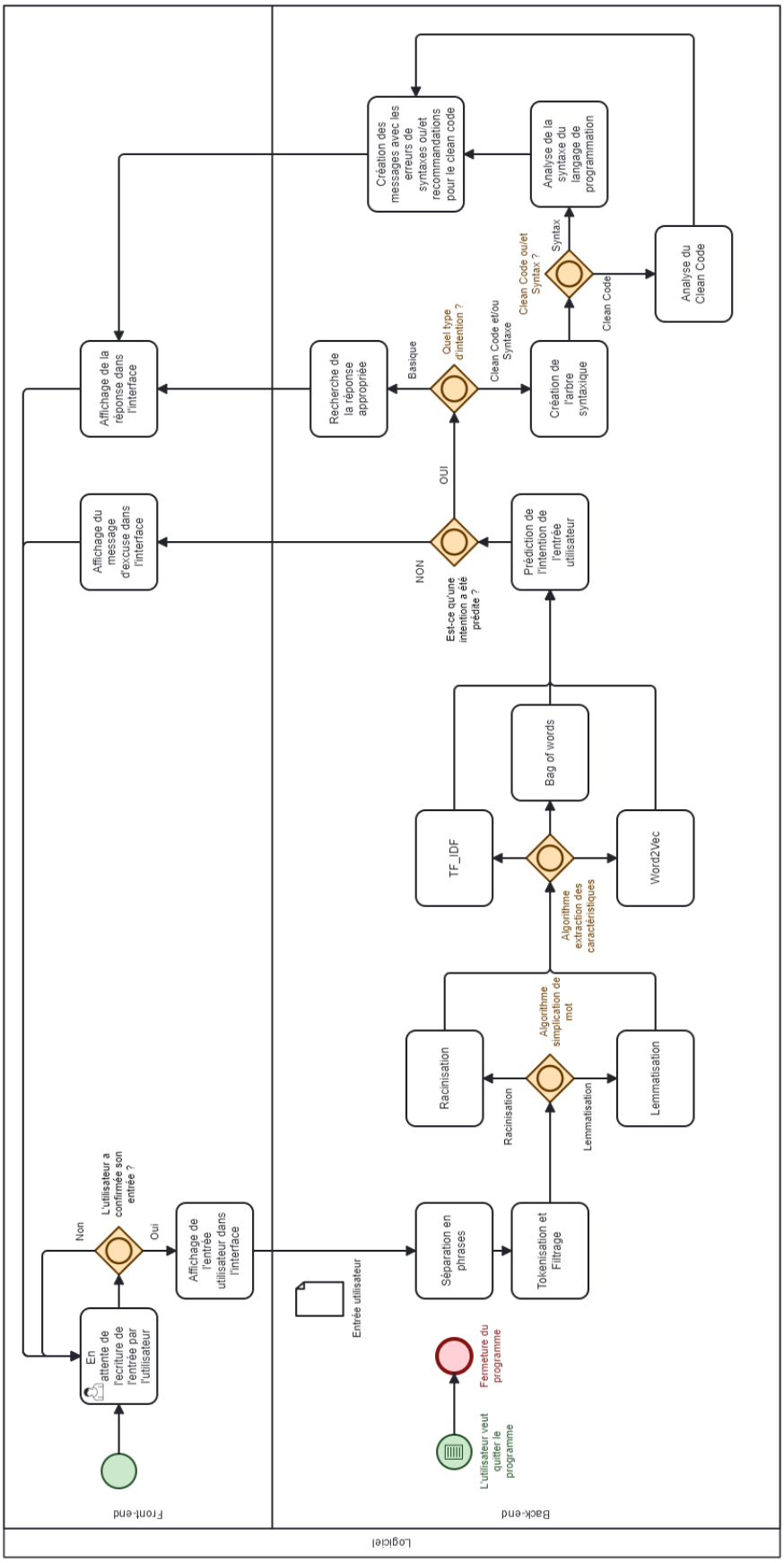


FIGURE 13 – Schéma BPMN du logiciel

5 Structure du logiciel

Cette section fournit une description complète de la structure du projet qui a été réalisée en relation avec le thème choisi. Elle comprendra une explication des fonctionnalités supportées par le logiciel ainsi que des explications sur le pseudocode des algorithmes principaux développés à partir de zéro. Comme expliqué durant dans l'introduction de ce projet, les fonctionnalités ont été développées pour être utilisées avec la langue anglaise.

5.1 Fonctionnalités

Les fonctionnalités développées pour ce projet sont des fonctionnalités qui sont très souvent présentent dans les projets de ce type. Le chatbot possède des fonctionnalités basiques telles que répondre à l'utilisateur et traiter l'entrée de celui-ci.

Ces capacités de traitement reposent sur des technologies de traitement du langage naturel qui donnent au chatbot la possibilité de diviser le texte de l'utilisateur en phrases ainsi que d'effectuer la tokenisation. De plus, il peut appliquer d'autres transformations sur les entrées pour éliminer les éléments superflus grâce à la lemmatisation et à la racinisation. Les fonctionnalités d'extraction de caractéristiques comme le "bag of words", TF-IDF ou Word2Vec sont également implémentées. Pour traiter toutes ces données, le chatbot utilise des modèles de deep learning tels que BERT ou des réseaux de neurones simples, ce qui lui permet de comprendre le contexte et de choisir une réponse adaptée à l'entrée de l'utilisateur.

Le système est également doté d'une fonction d'analyse syntaxique qui lui permet de construire des arbres syntaxiques pour examiner le code entré par l'utilisateur. Cette fonction détecte les erreurs de syntaxes et suggère des améliorations pour rendre le code plus lisible.

Pour faciliter l'utilisation pendant le développement, plusieurs fonctionnalités ont été intégrées à l'interface graphique pour permettre d'entraîner des modèles, de les tester et de consulter le fichier contenant toutes les intentions du chatbot.

5.2 Explication des pseudocodes

Cette section va permettre de découvrir la façon dont ont été implémentés les algorithmes principaux. Chaque sous-section est dédiée à l'explication détaillée du pseudocode d'un algorithme spécifique dans l'ordre de son exécution dans le programme. Cela permettra une compréhension plus claire et précise du fonctionnement interne du projet.

5.2.1 Séparation en phrases

L'algorithme présenté ci-dessous est le premier algorithme à se déclencher lors de la réception de l'entrée de l'utilisateur. Il est séparé en deux étapes principales qui vont permettre d'extraire de l'entrée de l'utilisateur, les différentes phrases ainsi que le bloc de code si celle-ci en contient un.

1. **L'extraction du code et du langage informatique**
   Cette fonction permet de chercher dans une entrée utilisateur la présence d'un bloc de code sous format Markdown c'est-à-dire entourée de "‘. S'il détecte un bloc de code, il le retire de l'entrée utilisateur puis, la fonction va permettre l'extraction du langage informatique si l'utilisateur l'a précisé. Pour ce qui est du texte restant, il est préservé pour les différentes analyses ultérieures.
2. **La segmentation en phrase**
   Cette fonction qui survient après l'extraction du code et du langage informatique va permettre de segmenter la partie conservée de l'entrée de l'utilisateur en phrases. Pour cela, l'utilisation d'une expression régulière a été nécessaire pour identifier les fins de phrases marquées par un point, point d'exclamation, point d'interrogation ou un retour à la ligne tout en écartant les abréviations.

Algorithm 1: Segmenter les phrases, extraire le code intégré et identifier le langage de programmation de l'entrée utilisateur

Result: Renvoie un dictionnaire avec les phrases, la langue et le code extrait de l'entrée utilisateur
Input: Une chaîne de caractères pouvant contenir du texte et du code
Function `segment_sentences(`*user_input*`)` **:**
    phrases_segmentées ← `extract_language_and_code(`*user_input*`)`
    user_input ← diviser user_input en phrases en utilisant l'expression régulière
    for *segment dans phrases_segmentées["user_input"]* **do**
        if *segment* ≠ *""* **then**
            Ajouter segment à phrases_segmentées["user_input"]
    return phrases_segmentées
Private Function `extract_language_and_code(`*user_input*`)` **:**
    Initialiser code, langue à Aucun
    pattern ← expression régulière pour les blocs de code
    correspondance ← rechercher pattern dans user_input
    if *correspondance trouvée* **then**
        langue ← premier groupe de correspondance
        code ← deuxième groupe de correspondance
        user_input ← extraire le texte en dehors du bloc de code de user_input
    return *dictionnaire avec langue, code, et le user_input modifié*

5.2.2 Tokenisation et filtrage

L'algorithme ci-dessous présente la manière dont la tokenisation et le filtrage se déroulent dans le programme. Cette partie se déclenche lors de la réception d'une entrée de l'utilisateur après que celle-ci ait été prétraitée par **l'algorithme 1**. Il a pour but de faire un deuxième prétraitement sur les différentes phrases pour les rendre plus facilement utilisables par les algorithmes en utilisant trois grandes étapes qui sont la tokenisation, la suppression des mots vides et la suppression des ponctuations.

1. **Suppression des ponctuations**
   Lorsqu'une phrase est reçue, l'une des premières étapes est de retirer les ponctuations qui dans notre cas n'apporte pas plus d'information pour l'analyse.
2. **Tokenisation**
   Lors de la tokenisation la phrase est transformée en une liste contenant les mots de la phrase.
3. **Suppression des mots vides**
   Avant de renvoyer la liste comprenant les mots de la phrase pour que celle-ci soit analysée, les mots vides sont retirés car ceux-ci constituent des informations parasites.

Algorithm 2: Tokenisation et Filtrage des Phrases

```
Fonction tokenize_and_filter_sentence(phrase) :
    phrase ← remove_punctuation(phrase)
    jetons ← tokenize(phrase)
    if enlever mots vides then
        jetons ← remove_stop_words(jetons)
    return jetons
Private Function load_stop_words() :
    mots_vides ← Aucun
    chemin_fichier ← Chemin vers le fichier
    Ouverture du fichier contenant les mots vides
    foreach ligne dans fichier do
        Ajouter mots vides dans la liste
    return mots_vides
Private Function tokenize(phrase) :
    jetons ← Utilisation d'un regex pour séparer la phrase en mots
    return jetons
Private Function remove_stop_words(jetons) :
    mots_filtrés ← Retrait des mots vides dans jetons
    return mots_filtrés
Private Function remove_punctuation(phrase) :
    traducteur ← Retrait des ponctuations
    return phrase.traduire(traducteur)
```

5.2.3 Porter Stemmer

Le Porter Stemmer fonctionne en appliquant une série de règles de remplacement à des mots pour en retirer les affixes communs.

1. Suppressions initiales
   - Si le mot se termine par 'sses', remplacer par 'ss'
   - Si le mot se termine par 'ies', remplacer par 'i'
   - Si le mot se termine par 'ss', ne rien faire
   - Si le mot se termine par 's', supprimer 's'
   - Si le m $>$ 0 et que le mot se termine par 'eed', remplacer par 'ee'
   - Si le mot contient une voyelle et que le mot se termine par 'ed', supprimer 'ed'
   - Si le mot contient une voyelle et que le mot se termine par 'ing', supprimer 'ing'
   - **Si l'une des étapes précédentes a été exécutée alors**
     - Si le mot se termine par 'at', remplacer par 'ate'
     - Si le mot se termine par 'bl', remplacer par 'ble'
     - Si le mot se termine par 'iz', remplacer par 'ize'
     - Si le mot se termine une double consonne qui n'est pas 'l', 's' ou 'z', supprimer la dernière lettre
     - Si le m = 1 et se termine par une consonne, une voyelle puis une consonne, rajouter 'e'
   - Si le mot possède une voyelle et qu'il se termine par un 'y', remplacer par 'i'
2. Traitement des terminaisons plus complexes
   - Les remplacements de suffixes 'ational' → 'ate', 'tional' → 'tion', 'enci' → 'ence', 'anci' → 'ance', 'izer' → 'ize' , 'abli' → 'able', 'alli' → 'al', 'entli' → 'ent', 'eli' → 'e', 'ousli' → 'ous', 'ization' → 'ize', 'ation' → 'ate', 'ato' → 'ate', 'alism' → 'al', 'iveness' → 'ive', 'fulness' → 'ful', 'ousness' → 'ous', 'aliti' → 'al', 'iviti' → 'ive' , et 'biliti' → 'ble' .
3. Traitement des suffixes résiduels
   - Supprimer 'ative', 'full', 'ness' si ces suffixes sont présents
   - Si le mot se termine par 'icate' ou 'iciti', remplacer par 'ic'
   - Si le mot se termine par 'alize' ou 'ical', remplacer par 'al'
4. Réductions finales
   - Suppression des terminaisons tel que 'al', 'ance', 'ence', 'er', 'ic', 'able', 'ible', 'ant', 'ement', 'ment', 'ent', 'ion', 'ou', 'ism', 'ate', 'iti', ous', 'ive' et 'ize'
5. Suppression finale de 'e'
   - Supprimer 'e' final si le mot contient plus d'une syllabe après cette suppression
   - Supprimer un 'l' si jamais celui-ci est dédoublé à la fin du mot

Algorithm 3: Étape 1 de l'algorithme de racinisation de Porter

Function `step_1_a(`*mot*`)` **:**
- **Input :** Un mot
- **Output:** Mot après suppression des suffixes pluriels
- **if** *mot se termine par "sses" ou "ies" ou "ss" ou "s"* **then**
  - **return** *mot avec suffixe modifié*
- **return** *mot*

Function `step_1_b(`*mot*`)` **:**
- **Input :** Un mot
- **Output:** Mot après traitement des formes en -ed, -ing
- **if** *mot se termine par "eed" et* `determine_m(`*mot sans suffixe*`)` > 0 **then**
  - **return** *mot avec suffixe modifié*
- **if** *mot se termine par "ed" et mot avec suffixe en moins contient une voyelle* **then**
  - **return** `step_1_2b(`*mot avec suffixe modifié*`)`
- **if** *mot se termine par "ing" et mot avec suffixe en moins contient une voyelle* **then**
  - **return** `step_1_2b(`*mot avec suffixe modifié*`)`
- **return** *mot*

Function `step_1_2b(`*mot*`)` **:**
- **Input :** Un mot
- **Output:** Mot après application des règles supplémentaires de l'étape 1b
- **if** *mot se termine "at" ou "bl" ou "iz"* **then**
  - **return** *mot avec suffixe modifié*
- **if** *mot ne se termine ni par "s", ni par "z", ni par "l" et se termine par une double consonne* **then**
  - **return** *mot moins la dernière lettre*
- **if** *m == 1 et se termine par une consonne un voyelle et une consonne* **then**
  - **return** *mot + "e"*
- **return** *mot*

Function `step_1_c(`*mot*`)` **:**
- **Input :** Un mot
- **Output:** Mot après changement de 'y' en 'i' si nécessaire
- **if** *mot se termine par "y" ou "i" et mot sans suffixe contient une voyelle)* **then**
  - **return** *mot avec suffixe modifié*
- **return** *mot*

Algorithm 4: Étape 2 de l'algorithme de racinisation de Porter

Function `step_2(`*mot*`)` **:**
- **Input :** Un mot
- **Output:** Mot modifié après application des transformations de suffixe
- **if** *mot se termine par 'ational', 'tional',... et* `determine_m(`*mot sans suffixe*`)` *> 0* **then**
  - **return** *mot avec suffixe modifié*
- **return** *mot*

Algorithm 5: Étape 3 de l'algorithme de racinisation de Porter

Function `step_3(`*mot*`)` **:**
- **Input :** Un mot
- **Output:** Mot modifié après application des transformations de suffixe
- **if** *mot se termine par "icate" ou "ative" ou "alize" ou "iciti" ou "ical" ou "ful" ou "ness" &* `determine_m(`*mot sans suffixe*`)` *> 0* **then**
  - **return** *mot avec suffixe modifié*
- **return** *mot*

Algorithm 6: Étape 4 de l'algorithme de racinisation de Porter

Function `step_4(`*mot*`)` **:**
- **Input :** Un mot
- **Output:** Mot modifié après application des transformations de suffixe
- **if** *mot se termine par 'al', 'ance', 'ence',... et* `determine_m(`*mot sans suffixe*`)` *> 1* **then**
  - **return** *mot avec suffixe modifié*
- **return** *mot*

Algorithm 7: Étape 5 de l'algorithme de racinisation de Porter

Function `step_5_a(`*mot*`)` **:**
- **Input :** Un mot
- **Output:** Mot après suppression possible d'un 'e' terminal
- **if** `determine_m(`*mot sans suffixe*`)` *> 1 et mot se termine par 'e'* **then**
  - **return** *mot avec suffixe modifié*
- **if** *mot se termine par 'e' et* `determine_m(`*mot sans suffixe*`)` *== 1 et !*`end_with_cvc(`*mot sans suffixe*`)` **then**
  - **return** *mot avec suffixe modifié*
- **return** *mot*

Function `step_5_b(`*mot*`)` **:**
- **Input :** Un mot
- **Output:** Mot après suppression possible de la dernière consonne en double
- **if** *mot se termine par 'l' et* `determine_m(`*mot sans suffixe*`)` *> 1 et* `end_with_double_consonant(`*mot*`)` **then**
  - **return** *mot avec suffixe modifié*
- **return** *mot*

5.2.4 Sac de mots

La fonction ci-dessous permet de convertir une phrase qui a été au préalable tokenisée en un vecteur où chaque indice correspond à un mot spécifique du vocabulaire. Pour chaque mot du vocabulaire, un indice lui sera associé dans le vecteur qui contiendra le nombre de fois que ce mot apparaît dans la phrase analysée.

Algorithm 8: Convertit une phrase en vecteur sac de mots

Result: Renvoie un tableau numpy représentant la phrase sous forme de vecteur de fréquences de mots
Input: Une chaîne de caractères à convertir en représentation sac de mots
Function `extraire_caracteristiques(`*phrase*`)` **:**
    representation_sac_de_mots ← initialise un tableau de zéros de taille égale à la longueur du vocabulaire ;
    for *mot dans texte de phrase prétraité* **do**
        if *mot est dans le vocabulaire* **then**
            indice ← trouve l'indice du mot dans le vocabulaire;
            incrémente representation_sac_de_mots à l'indice de 1;
    return representation_sac_de_mots;

5.2.5 TF_IDF

L'algorithme ci-dessous permet de convertir une phrase qui a été au préalable tokenisée en un vecteur où chaque indice correspond à un mot spécifique du vocabulaire. Contrairement au sac de mots, le TF_IDF prend en compte la fréquence d'apparitions des mots pour ainsi réduire l'impact des mots qui appraissent fréquemment.

Algorithm 9: Extraction des Caractéristiques et Calcul de Fréquence Documentaire

Fonction `ExtraireCaracteristiques(`*phrase*`)` **:**
    phrase_pretraitee ← `PretraiterTexte(`*phrase*`)`
    vecteur_tf_idf ← tableau de zéros (longueur de vocab)
    foreach *mot dans phrase_pretraitee* **do**
        if *mot dans vocab* **then**
            tf ← nombre de fois mot dans phrase_pretraitee / longueur de phrase_pretraitee
            idf ← log du nombre total de document / `CalculerFreqDoc(`*mot*`)`
            vecteur_tf_idf[index du mot] ← tf × idf
    return vecteur_tf_idf
Private Function `CalculerFreqDoc()` **:**
    foreach *doc dans docs* **do**
        foreach *mot dans doc* **do**
            freq_doc[mot] ← freq_doc[mot] + 1
    return freq_doc

5.2.6 Analyseur de la syntaxe du langage de programmation

L'algorithme ci-dessous permet de parcourir un arbre syntaxique pour trouver les erreurs de syntaxe ou les éléments manquants dans le code qui pourraient l'empêcher de compiler. L'arbre syntaxique est obtenu grâce à l'utilisation d'une bibliothèque externe nommée Tree-sitter qui a été développée spécifiquement pour la création d'arbres syntaxiques pour les langages de programmation.

Algorithm 10: Trouver des problèmes de syntaxe dans un arbre syntaxique

```
Input  : syntax_tree (Tree)
Output: Liste d'éléments
descriptions ← empty set;
todo ← Contient le premier élément de l'arbre syntaxique;
while todo n'est pas vide do
    node ← retire le premier élément du todo;
    for child in node.children do
        if Si il y a une erreur dans le child then
            Ajoute l'erreur et la ligne dans la description
        else if Si il manque un élément dans le child then
            Ajoute le missing et la ligne dans la description
        Ajouter le child au todo;
if descriptions est vide then
    return "I didn't detect syntax errors in your code.";
Trie les descriptions dans l'ordre des lignes;
return descriptions;
```

5.2.7 Analyseur de Clean Code

L'algorithme ci-dessous permet de parcourir un arbre syntaxique pour y trouver des recommandations à faire au niveau du Clean Code. Ces recommandations sont basées sur les conventions des langages supportés actuellement et peuvent varier de la simple longueur du nom d'une variable ou d'une fonction au nombre de lignes ou de paramètres dans une fonction.

Algorithm 11: Analyse d'un arbre syntaxique pour les problèmes de code propre

```
Data: syntax_tree (Tree), language (str)
Result: Liste ou chaîne des problèmes de codage
Function describe_clean_code_problems(syntax_tree, language) :
    descriptions ← empty set
    todo ← Contient le premier élément de l'arbre syntaxique
    while todo n'est pas vide do
        node ← retire le premier élément du todo
        for child in node.named_children do
            if child est de type "function_declarator" then
                if Le nombre de paramètre est trop important then
                    descriptions ← une recommandation sur le nombre de
                      paramètre de la fonction
                if Le nom de la fonction ne respecte pas les conventions then
                    descriptions ← une recommandation sur le nom de la fonction
                if Le nombre de ligne de la fonction est trop important then
                    descriptions ← une recommandation sur la grandeur de la
                      fonction
            else if child est de type "identifier" ou "field_identifier" then
                if Le nom de la variable ne respecte pas les conventions de no then
                    descriptions ← une recommandation sur le nom
                if Le nom de la variable est plus petit que 3 caractères then
                    descriptions ← une recommandation sur la longueur du nom
            else if child est de type "class_declaration", "struct_specifier"] then
                if Le nom de la classe ou de la structure ne respecte pas les
                  conventions then
                    descriptions ← une recommandation sur le nom
                if Le nom de la classe ou de la structure est plus petit que 3
                  caractères then
                    descriptions ← une recommandation sur la longueur du nom
            ajoute le child dans le todo
    if descriptions est vide then
        return "No recommendations to make for Clean Code"
    Trie les descriptions dans l'ordre des lignes
    return descriptions
```

6 Entraînement du chatbot

Cette section présente une des phases cruciales de la création d'un chatbot qui n'est d'autre que son entraînement. L'entraînement du chatbot à pour but de lui permettre de répondre de manière efficace et intelligente aux utilisateurs. Cependant, il n'est pas uniquement question de cela dans cette section mais aussi de tout ce qui entoure l'entraînement tel que la gestion des données utilisées, leur stockage, les méthodes d'entraînement employées, et le développement de modèles de compréhension du langage naturel qui a été traité dans la section 3.1. L'objectif final étant de créer un chatbot qui peut comprendre le contenu textuel, mais aussi de permettre si possible au chatbot de comprendre le contexte et la nuance des dialogues.

6.1 Gestion des données pour l'entraînement

Il serait difficile de parler d'entraînement sans parler des données. En effet, les données sont un aspect essentiel de l'entraînement qui ne peut simplement pas fonctionner sans celle-ci. Pour le chatbot lié à ce projet, les données provenant du site web Kaggle ont été utilisées comme base avant de les agrémenter de plusieurs nouveaux éléments pour répondre aux besoins de ce projet. Une fois les données en main, celles-ci ont été stockées dans un fichier JSON qui est surnommé **la base de connaissances** car ce fichier contient tout ce que le chatbot est censé pouvoir comprendre. L'exemple ci-dessous représente une sorte d'intention stockée qui est ici la catégorie "Greeting" qui se réfère à la salutation.

```
"intents": [
  {
    "tag": "Greeting",
    "patterns": [
      "Hi",
      "Hi there",
      "Hola",
      "Hello",
      "Hello there",
      "Hya",
      "Hya there"
    ],
    "responses": [
      "Hi human !",
      "Hello human !",
      "Hola human !"
    ],
    "module": "",
    "class": "",
    "function": "",
    "parameters": []
  }
]
```

"intents" : Un tableau contenant plusieurs objets, chacun représentant une intention spécifique que le chatbot peut reconnaître.

"tag" : Un identifiant unique pour chaque intention, utilisé pour classer les différents types de requêtes ou de questions posées par les utilisateurs.

"patterns" : Un tableau de phrases ou expressions que les utilisateurs sont susceptibles d'utiliser pour exprimer l'intention correspondante.

"responses" : Un tableau de réponses que le chatbot peut utiliser pour répondre à l'intention détectée.

"module" : Nom du module de traitement, si applicable. Laissez vide si non applicable.

"class" : Nom de la classe utilisée dans le module, si applicable. Laissez vide si non applicable.

"function" : Nom de la fonction à exécuter dans la classe spécifiée, si applicable. Laissez vide si non applicable.

"parameters" : Un tableau des paramètres nécessaires pour la fonction, si applicable. Laissez vide si non applicable.

6.2 Classification des intentions

La classification des intentions est une composante clé dans le développement des chatbots. Elle consiste à identifier l'intention derrière les phrases saisies par les utilisateurs afin que le chatbot puisse répondre de manière appropriée.

6.2.1 Fonctionnement de la classification

Le fonctionnement de la classification suit une structure bien précise qui est décrite ci-dessous en cinq étapes. De plus, la Figure 14 – Exemple Simple d'un classificateur d'intention offre un schéma simplifié du fonctionnement de la classification.

1. **Réception de la requête :** Le chatbot reçoit une phrase de l'utilisateur.
2. **Prétraitement des données :** La phrase est préparée pour l'analyse en effectuant la suppression de la ponctuation, la mise en minuscule des lettres et l'élimination des mots inutiles.
3. **Analyse et comparaison :** La phrase traitée est ensuite comparée aux différents modèles de la base de connaissances.
4. **Identification de l'intention :** À l'aide d'algorithmes de traitement du langage naturel, le système détermine l'intention la plus probable en fonction des similitudes avec les données stockées.
5. **Sélection de la réponse :** Une fois l'intention identifiée, le chatbot sélectionne une réponse appropriée parmi celles associées à l'intention détectée.

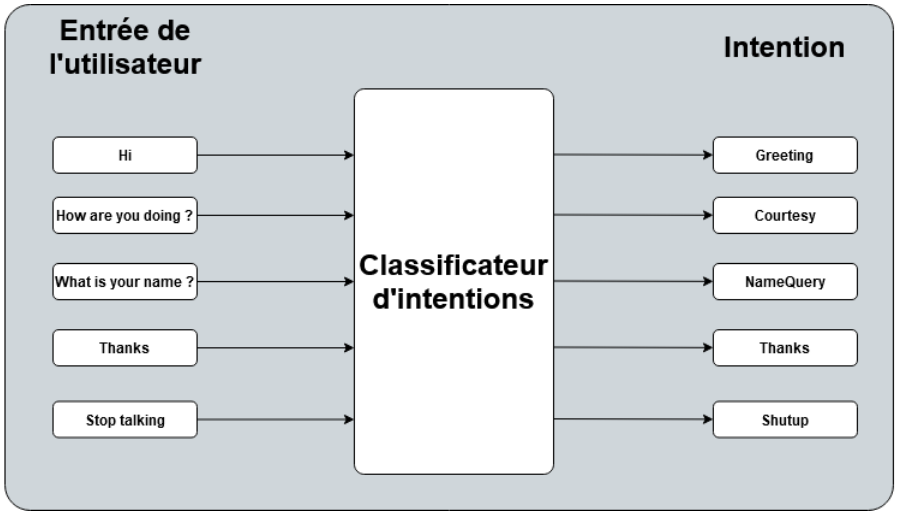


FIGURE 14 – Exemple simple d'un classificateur d'intentions

6.2.2 Ressources nécessaires

Pour que la classification des intentions soit efficace, elle nécessite :

— Une **base de connaissances** qui peut-être un fichier contenant de multiples informations comme dans l'exemple ci-dessus.

— Des **algorithmes de traitement du langage naturel (NLP)** capable de comprendre et d'analyser le texte.

6.2.3 BERT pour la classification des intentions

Cette section aborde l'utilisation du modèle BERT pour la tâche de classification des intentions grâce au modèle pré-entrainé "bert-base-uncased" provenant du site web HuggingFace.

1. **Préparation des données** :
   - Les données doivent être représentées comme une paire (texte, étiquette) dans laquelle le texte représente une phrase et l'étiquette représente l'intention.
   - Le texte est prétraité en le séparant en phrases multiples, en retirant les ponctuations et les mots vides.
2. **Tokenisation** :
   - Toutes les paires (texte, étiquette) provenant de la base de connaissances sont alors tokeniser grâce à l'utilisation de la classe **BertTokeniser** présente dans les ressources pour l'utilisation du modèle BERT de HuggingFace.
3. **Entraînement** :
   - La dernière étape est d'entraîner le modèle en utilisant la classe **BertForSequenceClassification** qui est initialisée avec le poids pré-entraîné du modèle "bert-base-uncased". BertForSequenceClassification va alors ajouter une dernière couche de classification avant de commencer l'entraînement.
   - L'entraînement s'effectue grâce à l'utilisation de la classe **Trainer** qui est une classe spécialement conçue pour l'entraînement des transformers.
4. **Classification de l'intention**
   - Lorsqu'un utilisateur envoie son entrée, celle-ci va être tokenisée et prétraitée avant d'être envoyée dans le modèle BERT spécialisé pour la classification des intentions.
   - La sortie est une probabilité pour chacune des intentions.

6.2.4 TF-IDF pour la classification des intentions

Cette section aborde l'utilisation de la méthode TF-IDF alliée à un réseau de neurones pour la tâche de classification des intentions.

1. **Préparation des données** :
   - Le texte est prétraité en le séparant en phrases multiples, en retirant les ponctuations et les mots vides.
2. **Vectorisation des Textes** :
   - Le texte prétraité est alors transformé en vecteurs TF-IDF grâce à la classe **TFIDF** et dont le fonctionnement a été détaillé dans la section 3.1.3
3. **Entraînement du réseau de neurones** :
   - On effectue les deux étapes précédentes sur toutes les données de la base de connaissances pour créer des paires (vecteur, étiquette) où le vecteur représente un texte vectorisé/prétraité et l'étiquette représente l'intention avant de les envoyer dans le réseau de neurones pour l'entraînement.
4. **Classification des intentions** :
   - Lorsqu'un utilisateur envoie son entrée, celle-ci va être prétraitée et vectorisée avant d'être envoyée au réseau de neurones pour la classification des intentions.
   - La sortie est une probabilité pour chacune des intentions.

6.2.5 Word2Vec pour la classification des intentions

Cette section aborde l'utilisation de la méthode Word2Vec alliée à un réseau de neurones pour la tâche de classification des intentions.

1. **Préparation des données** :
   — Le texte est prétraité en le séparant en phrases multiples, en retirant les ponctuations et les mots vides.
2. **Encodage des Mots** :
   — Le texte est transformé en vecteurs Word2Vec qui est un vecteur de mot représentant la phrase grâce à l'utilisation de la classe **Word2Vec**.
3. **Moyennage des Vecteurs** :
   — Il est alors nécessaire de faire la moyenne des mots du vecteur pour obtenir une représentation vectorielle de la phrase.
4. **Entraînement du réseau de neurones** :
   — On effectue les trois étapes précédentes sur toutes les données de la base de connaissances pour créer des paires (vecteur, étiquette) où le vecteur représente un texte prétraité, vectorisé et moyenné. L'étiquette représente l'intention avant de les envoyer dans le réseau de neurones pour l'entraînement.
5. **Classification des intentions** :
   — Lorsqu'un utilisateur envoie son entrée, celle-ci va être prétraitée, vectorisée et moyennée avant d'être envoyée au réseau de neurones pour la classification des intentions.
   — La sortie est une probabilité pour chacune des intentions.

6.2.6 Bag of Words (BoW) pour la classification des intentions

Cette section aborde l'utilisation de la méthode Word2Vec alliée à un réseau de neurones pour la tâche de classification des intentions.

1. **Préparation des données** :
   — Le texte est prétraité en le séparant en phrases multiples, en retirant les ponctuations et les mots vides.
2. **Vectorisation des Textes** :
   — Le texte est transformé en vecteurs de fréquence de mots. Chaque texte est représenté par un vecteur dont les dimensions correspondent aux termes du vocabulaire, avec des valeurs correspondant à la fréquence de chaque terme grâce à l'utilisation de la classe **BagOfWords**.
3. **Entraînement du réseau de neurones** :
   — On effectue les deux étapes précédentes sur toutes les données de la base de connaissances pour créer des paires (vecteur, étiquette) où le vecteur représente un texte prétraité et vectorisé. L'étiquette représente l'intention avant de les envoyer dans le réseau de neurones pour l'entraînement.
4. **Classification des intentions** :
   — Lorsqu'un utilisateur envoie son entrée, celle-ci va être prétraitée et vectorisée avant d'être envoyée au réseau de neurones pour la classification des intentions.
   — La sortie est une probabilité pour chacune des intentions.

7 Évaluation

Cette section vise à présenter une évaluation complète du logiciel en utilisant l'évaluation qualitative et quantitative. L'objectif est de déterminer dans quelle mesure le logiciel répond aux attentes des utilisateurs issus de divers horizons professionnels ainsi que de déterminer grâce à des métriques si le logiciel répond correctement à la problématique.

7.1 Évaluation Qualitative

L'évaluation qualitative a été menée en prenant un échantillon de dix personnes qui sont divisées en trois catégories : cinq personnes travaillant dans l'informatique, deux personnes qui sont totalement extérieures au domaine de l'informatique et enfin trois étudiants en informatique. Chaque participant a été invité à tester le logiciel et à évaluer son expérience selon trois critères principaux : la facilité d'utilisation, le design de l'interface, et la qualité des réponses du chatbot. Pour chaque critère, trois options étaient disponibles pour évaluer : "Bien", "Satisfaisant" et "Mauvais". En plus de cela, les participants ont eu la possibilité de laisser des commentaires pour justifier leur décision.

7.1.1 Critère : Facilité d'utilisation

Ce critère évalue si l'interface de l'application est intuitive et simple à naviguer pour l'utilisateur.

Facilité d'utilisation			
	Bien	**Satisfaisant**	**Mauvais**
Professionnel	1	4	0
Externe	0	2	0
Étudiant	2	1	0

TABLE 1 – Table des évaluations pour la facilité d'utilisation

Commentaires des utilisateurs :

- "L'interface est facile à utiliser, ce qui simplifie grandement l'expérience."
- "L'utilisation est aussi simple que celle d'un chatbot classique, ce qui rend le processus très intuitif."
- "J'ai eu du mal à comprendre qu'il fallait utiliser le bouton 'code' pour permettre au chatbot de localiser le code."
- "Une version française serait fortement appréciée pour faciliter l'utilisation par des francophones."

7.1.2 Critère : Design de l'interface

Ce critère se concentre sur l'esthétique visuelle de l'interface utilisateur et l'ergonomie globale. Il prend en compte l'harmonie des couleurs, la lisibilité des textes et la disposition logique des éléments.

Design de l'interface			
	Bien	**Satisfaisant**	**Mauvais**
Professionnel	3	2	0
Externe	2	0	0
Étudiant	3	0	0

TABLE 2 – Table des évaluations pour le design de l'interface

Commentaires des utilisateurs :

— "Le design de l'interface est simple et les couleurs sont douces pour les yeux "

— "J'aurais aimé avoir la possibilité de changer vers un mode plus lumineux."

7.1.3 Critère : Qualité des réponses du chatbot

Ce critère mesure la pertinence et l'exactitude des réponses fournies par le chatbot. Il évalue la capacité du chatbot à comprendre les demandes des utilisateurs et à fournir des réponses qui sont correctes. La mesure de la qualité peut également inclure la vitesse de réponse du chatbot.

Qualité des réponses du chatbot			
	Bien	**Satisfaisant**	**Mauvais**
Professionnel	2	2	1
Externe	0	2	0
Étudiant	2	1	0

TABLE 3 – Table des évaluations pour la qualité du chatbot

Commentaires des utilisateurs :

— "Le chatbot m'a l'air assez limité sur les sujets auxquels il peut répondre."

— "Le chatbot a des réponses simples, je m'attendais à quelque chose comme ChatGPT"

— "J'ai été agréablement surpris de voir la vitesse à laquelle le chatbot répondait, et ce même lors de la correction de code."

— " La qualité pour un prototype est assez bonne, le chatbot répond rapidement même si limité dans sa base de connaissances. Continue comme ça !"

7.2 Évaluation Quantitative

L'objectif principal de l'analyse quantitative est de comparer les différents modèles utilisables pour la création d'un chatbot dans le cadre de ce projet afin de déterminer lequel offre les meilleures performances. Tous les entraînements ont été effectués sur un ordinateur disposant d'un Ryzen 7 3800X à 16 coeurs (de 3.9 GHz à 4.5 GHz) et de 32 GO de RAM.

Pour permettre cette comparaison, il a été nécessaire d'entraîner une multitude de modèles en utilisant pour chacun des paramètres optimaux qui ont été trouvés en utilisant une manière de procéder par essais-erreurs. Cette méthode a permis au fil des essais d'identifier des paramètres optimaux pour les modèles en termes d'épochs, de taille de lot, de taux d'apprentissage et de taille cachée. Les graphiques générés à la fin de chaque entraînement de modèle ont été utilisés pour ajuster ses paramètres en les comparant aux courbes de la Figure 15 – Modèle des taux d'apprentissages. Un échantillon des derniers graphiques générés est présent dans la Figure 16 – Comparaison des courbes d'entraînement.

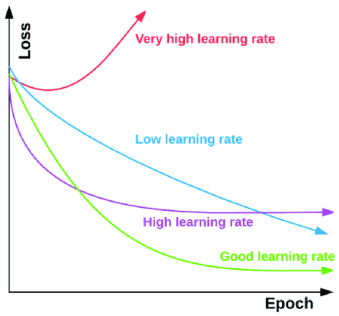
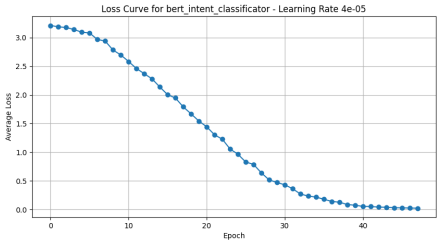
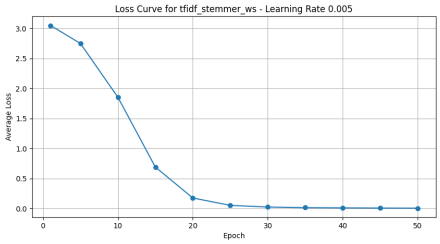


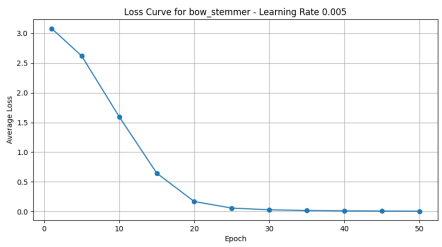
FIGURE 15 – Modèle des taux d'apprentissages [26]



(a) Bert



(b) TFIDF avec mots vides



(c) BagOfWords

FIGURE 16 – Comparaison des courbes d'entraînement

Une fois les paramètres optimaux trouvés pour chaque modèle, ils sont testés grâce à deux métriques qui mesurent les performances sur des données connues et inconnues. Chaque modèle et les résultats obtenus pour ces deux métriques sont résumés dans le Tableau 4 – Résultats d'entraînement des différents modèles :

— **Les données connues (Précision d'Entraînement)** : Les données connues sont celles qui figurent dans la base de connaissances définie dans la section 6.1. Cette métrique vise à déterminer si le modèle fonctionne correctement sur les données sur lesquelles il a été entraîné.

— **Les données inconnues (Précision du Test)** : Les données inconnues ne sont pas présentes dans la base de connaissances, mais elles sont suffisamment similaires pour que le modèle puisse normalement les reconnaître. Cette métrique permet de déterminer si le modèle peut identifier des similarités entre les données qu'il connaît et celles qu'il ne connaît pas. Cela est crucial étant donné que nos utilisateurs viennent de divers horizons, il peut y avoir de légères variations dans les entrées fournies par les utilisateurs par rapport aux données dans la base de connaissances.

Modèle	**Prétraitement**	**Extracteur**	**Avec mots vides ?**	**Epochs**	**Taille de lot**	**Taux d'apprentissage**	**Taille caché**	**Précision d'Entraînement**	**Précision du Test**
BERT	Aucun	BERT	Aucun	40	16	0.00004	Aucune	100.0%	82.19%
NeuralNet	Stemmer	Word2Vec GRAM	NO	1000	16	0.003	16	100.0%	28.77%
NeuralNet	Lemmatizer	Word2Vec GRAM	NO	1000	16	0.003	16	86.83%	23.29%
NeuralNet	Stemmer	Word2Vec CBOW	NO	1000	16	0.003	16	100.0%	23.29%
NeuralNet	Lemmatizer	Word2Vec CBOW	NO	1000	16	0.003	16	98.2%	20.55%
NeuralNet	Stemmer	Word2Vec GRAM	YES	1000	16	0.003	16	100.0%	20.55%
NeuralNet	Lemmatizer	Word2Vec GRAM	YES	1000	16	0.003	16	100.0%	20.55%
NeuralNet	Stemmer	Word2Vec CBOW	YES	1000	16	0.003	16	100.0%	17.81%
NeuralNet	Lemmatizer	Word2Vec CBOW	YES	1000	16	0.003	16	100.0%	30.14%
NeuralNet	Stemmer	TFIDF	NO	50	16	0.005	16	100.0%	38.36%
NeuralNet	Lemmatizer	TFIDF	NO	50	16	0.005	16	100.0%	31.51%
NeuralNet	Stemmer	TFIDF	YES	50	16	0.005	16	100.0%	38.36%
NeuralNet	Lemmatizer	TFIDF	YES	50	16	0.005	16	100.0%	34.25%
NeuralNet	Stemmer	BagOfWords	NO	50	16	0.005	16	100.0%	38.36%
NeuralNet	Lemmatizer	BagOfWords	NO	50	16	0.005	16	100.0%	36.99%
NeuralNet	Stemmer	BagOfWords	YES	50	16	0.005	16	100.0%	46.58%
NeuralNet	Lemmatizer	BagOfWords	YES	50	16	0.005	16	100.0%	32.88%

TABLE 4 – Résultats d'entraînement des différents modèles

On peut observer dans le tableau ci-dessus que BERT surpasse les autres modèles de classification des intentions grâce à sa capacité à comprendre les contextes bidirectionnels des mots dans une phrase, ce qui améliore significativement la précision. Son aptitude à généraliser sur de nouveaux ensembles de données, combinée avec les dernières avancées en architecture de transformers, en fait un outil extrêmement fiable et efficace pour identifier correctement les intentions des utilisateurs.

8 Discussion

Dans cette section, il sera question d'explorer en détail les différentes facettes du projet. La lumière sera mise sur les limites inhérentes à l'approche actuelle ainsi que les possibilités d'améliorations futures.

8.1 Les limites du projet

Ce projet bien qu'il exploite les grands modèles de langage et la technologie de traitement du langage naturel (NLP) présente des limites bien visibles notamment en ce qui concerne sa base de connaissances.

La première limite présente dans le projet est la base de connaissances. En effet, cette base est constituée uniquement d'un fichier regroupant toutes les données nécessaires au chatbot pour la classification des intentions. L'utilisation d'un fichier comme celui-ci pose plusieurs problèmes, notamment en ce qui concerne la capacité du chatbot à comprendre des variations de phrases ayant le même sens, mais présentant des différences syntaxiques importantes. De plus, la capacité de réponse du chatbot qui est limitée car elle dépend des réponses pré-enregistrées dans ce fichier. En conséquence, le chatbot ne peut pas générer de réponses par lui-même. Tout cela amène à une efficacité amoindrie du chatbot dans les scénarios de communication plus complexes ou moins prévisibles.

La deuxième limite est tout ce qui concerne la maintenance et la mise à jour de la base de connaissances. Étant donnée que les mises à jour ne peuvent se faire que manuellement, il peut s'avérer très laborieux de venir ajouter de gros conglomérats de données dans le fichier actuel. Cela deviendra de plus en plus difficile au fur et à mesure que le champ d'application du chatbot s'élargit car cela amènerait à faire des erreurs d'encodage.

La troisième limite de ce projet concerne principalement l'analyse des erreurs de syntaxe et la recommandation de pratiques de Clean Code. Ces fonctionnalités reposent sur l'utilisation d'un arbre syntaxique, ce qui peut se révéler rapidement limité face à des codes complexes ou l'utilisation d'annotations de haut niveau qui sont courantes dans de nombreux langages de programmation. De plus, l'absence d'une base de données dynamique pour intégrer les nouvelles recommandations de Clean Code rend nécessaire une mise à jour manuelle. Cette tâche peut devenir particulièrement fastidieuse surtout lorsque le système prend en charge des dizaines de langages de programmation différents qui possèdent tout des règles de Clean Code différents. De plus, les recommandations de Clean Code qui sont actuellement en vigueur ne seront très certainement plus les mêmes d'ici une dizaine d'années.

La dernière limite de ce projet est directement liée à la langue que le programme peut traiter. En effet, à l'heure actuelle, le programme ne peut gérer que les entrées de l'utilisateur en anglais, ce qui limite l'utilisation des personnes ne parlant pas cette langue.

Ces limites soulignent l'importance de penser à des améliorations pour le futur du projet. Dans la section suivante, il sera discuté des améliorations qui peuvent être implémentées pour surmonter ces obstacles.

8.2 Les pistes d'améliorations

Pour surmonter les obstacles évoqués précédemment, il est nécessaire d'explorer les pistes d'améliorations visant à amener ce projet vers une version plus pérenne dans le temps. Les pistes d'amélioration proposées ci-dessous visent à changer le programme qui est dans un mode statique vers un mode dynamique. En mode statique, les principes du programme tel que la base de connaissances ou les règles de Clean Code ne changent pas avec le temps. Cependant, le mode dynamique confère au programme la capacité de s'adapter et d'évoluer avec le temps. Cela permettrait d'enrichir la base de connaissances grâce au données provenant des interactions avec l'utilisateur ainsi que de mettre à jour les principes du Clean Code si besoin.

La première amélioration suggérée concerne la base de connaissances qui est actuellement dans un état statique. Il serait bénéfique de permettre au système d'apprendre continuellement des interactions avec les utilisateurs et leurs feedbacks. Cette habilité d'apprendre du programme permettrait de rendre les réponses plus dynamiques et adaptatives. Cependant, pour garantir la qualité des réponses du programme, il sera essentiel de développer un système de validation qui filtrera les données inexactes ou inappropriées avant leur intégration dans la base de connaissances.

La deuxième amélioration possible concerne l'analyseur syntaxique et la recommandation du Clean Code qui utilise tous deux les arbres syntaxiques. Bien que cela soit efficace pour analyser la syntaxe, cette méthode montre des limites pour l'analyse du Clean Code.
La solution serait de mettre en place des modèles d'intelligence artificielle qui seront spécialisés dans la suggestion de recommandation pour le Clean Code ainsi que dans la correction de syntaxe. Cela permettrait d'évoluer de méthodes statiques vers des méthodes dynamiques qui pourront s'adapter en fonction du langage et de l'époque.

La troisième amélioration possible concerne la gestion des langues de l'utilisateur. Afin de rendre le chatbot plus accessible et efficace, il est important d'intégrer un système de détection automatique de la langue. Cela permettra au chatbot de comprendre et de répondre dans la langue choisie par l'utilisateur sans que ce dernier ait à la spécifier. Il serait aussi possible d'implémenter une fonctionnalité de traduction automatique en temps réel. Cela permettrait alors d'entraîner le chatbot sur une seul langue et de traduire tout ce que l'utilisateur transmet vers la langue comprise par le chatbot puis inversement lors de l'envoi de la réponse vers l'utilisateur. Cette capacité multilingue améliorera non seulement l'expérience utilisateur, mais élargira également la portée du chatbot à une audience internationale.

Grâce à ces trois améliorations, on peut ainsi espérer avoir un système plus efficace, réactif et pérenne dans le temps.

9 Conclusion

Ce projet a conduit à la réalisation d'un prototype de chatbot destiné à soutenir l'utilisateur dans la rédaction de code en utilisant le traitement du langage naturel et les arbres syntaxiques. Ce logiciel offre un aperçu des possibilités dans le domaine des chatbots et de l'assistance à l'écriture de code propre, ce qui répond parfaitement à l'objectif principal du projet qui était de développer un "Prototype de chatbot assistant l'écriture de code propre et exploitant le langage naturel". De plus, ce projet laisse entrevoir la prochaine étape dans la continuité du projet qui permettra la création d'un chatbot avancé exploitant les grands modèles de langage.

Cependant, malgré sa capacité à répondre à la problématique initiale, le prototype est limité par sa base de connaissances actuelles et nécessite des mises à jour manuelles pour évoluer. Il est également important de souligner que le chatbot peut rencontrer des difficultés à interpréter les entrées des utilisateurs en fonction du contexte.

En conclusion, bien que le développement du projet ait bien progressé, ce programme reste dans un état de prototype en raison des contraintes de temps typiques pour un projet de cette envergure. Il est judicieux de souligner que l'exploration des domaines des chatbots, du traitement du langage naturel et des arbres syntaxiques s'est révélée très enrichissante. Finalement, le travail de recherche effectué dans ces différents domaines durant le développement du logiciel a grandement contribué à renforcer mes connaissances tout en me donnant l'envie d'approfondir mes recherches sur ces différents sujets.

10 Références

[1] Robert C Martin. *Clean code : a handbook of agile software craftsmanship*. Pearson Education, 2009.

[2] MozDevNet. Syntaxe - glossaire mdn : Définitions des termes du web : Mdn. `https://developer.mozilla.org/fr/docs/Glossary/Syntax`.

[3] Eleni Adamopoulou and Lefteris Moussiades. An overview of chatbot technology. In *IFIP international conference on artificial intelligence applications and innovations*, pages 373–383. Springer, 2020.

[4] Les modèles de séquence-à-séquence - hugging face. `https://huggingface.co/learn/nlp-course/fr/chapter1/7`.

[5] Les modèles basés sur le encodeur - hugging face. `https://huggingface.co/learn/nlp-course/fr/chapter1/5`.

[6] Les modèles basés sur le décodeur - hugging face. `https://huggingface.co/learn/nlp-course/fr/chapter1/6`.

[7] Abhimanyu Chopra, Abhinav Prashar, and Chandresh Sain. Natural language processing. *International journal of technology enhancements and emerging engineering research*, 1(4) :131–134, 2013.

[8] Elizabeth D Liddy. Natural language processing. 2001.

[9] Marie Labelle. Trente ans de psycholinguistique. *Revue québécoise de linguistique*, 30(1) :155–176, 2001.

[10] deeplearning.ai. Natural language processing (nlp) - a complete guide. `https://www.deeplearning.ai/resources/natural-language-processing/`.

[11] Vineet Raina, Srinath Krishnamurthy, Vineet Raina, and Srinath Krishnamurthy. Natural language processing. *Building an Effective Data Science Practice : A Framework to Bootstrap and Manage a Successful Data Science Practice*, pages 63–73, 2022.

[12] Enzo Grossi and Massimo Buscema. Introduction to artificial neural networks. *European journal of gastroenterology & hepatology*, 19(12) :1046–1054, 2007.

[13] Imagerie vétérinaire – metron software. `https://www.eponamind.com/fr/neural-networks-deep-learning/`.

[14] Rohith Gandhi. Support vector machine - introduction to machine learning algorithms. `https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47`, Jul 2018.

[15] Svm - data analytics. `https://dataanalyticspost.com/Lexique/svm/`, Dec 2018.

[16] Amélie Dindouve and Yves Vandermeeren. Apprentissage moteur et plasticité cérébrale chez le patient hémiparétique après avc : L'apprentissage bimanuel est-il amélioré par dual-tdcs, comparé à une stimulation placebo ?

[17] Induraj. What are sparse features and dense features ? https://induraj2020.medium.com/what-are-sparse-features-and-dense-features-8d1746a77035, Feb 2023.

[18] Aindriya Barua. Contextual v/s non-contextual word embedding models for hindi named entity recognition. https://medium.com/@barua.aindriya/contextual-v-s-non-contextual-word-embedding-models-for-hindi-named-entity-recogn Dec 2021.

[19] Imeshadilshani. Words as vectors - sparse vectors vs. dense vectors. https://medium.com/@imeshadilshani212/words-as-vectors-sparse-vectors-vs-dense-vectors-18e2084ad312, Dec 2023.

[20] Ben Lutkevich. What is a parser ? definition, types and examples. https://www.techtarget.com/searchapparchitecture/definition/parser, Jul 2022.

[21] Radovan Vojtko. Bert model – bidirectional encoder representations from transformers. https://quantpedia.com/bert-model-bidirectional-encoder-representations-from-transformers/, Jul 2023.

[22] Cameron Hashemi-Pour and Ben Lutkevich. What is the bert language model ? : Definition from techtarget.com. https://www.techtarget.com/searchenterpriseai/definition/BERT-language-model, Feb 2024.

[23] Rani Horev. Bert explained : State of the art language model for nlp. https://towardsdatascience.com/bert-explained-state-of-the-art-language-model-for-nlp-f8b21a9b6270, Nov 2018.

[24] Mark von Rosing, Stephen White, Fred Cummins, and Henk de Man. Business process model and notation-bpmn. http://www.omg.org/news/whitepapers/Business_Process_Model_and_Notation.pdf, 2015.

[25] NeoLedge. Définition : Processus métier. https://www.neoledge.com/fr/a-propos/glossaire/definition-processus-metier/, May 2019.

[26] Comparative analysis of recurrent neural network architectures for reservoir inflow forecasting. https://www.researchgate.net/figure/Changes-in-the-loss-function-vs-the-epoch-by-the-learning-rate-40_fig2_341609757.

Table des figures

Liste des tableaux

List of Algorithms