Sorbonne Université

Cours de serveurs conversationnels

Développement d'un chatbot sur la vie au Moyen-Âge

Auteur : Kouassi Didier Kouamé

Professeur : Claude Montacié

Master 2 Langue & Informatique

14 mars 2025

Résumé

Ce rapport présente la conception et la mise en œuvre d'un agent conversationnel spécialisé dans le thème de la vie au Moyen Âge. L'agent analyse les intentions (*intents*) de l'utilisateur, telles que *"Comment était organisée la société médiévale ?"*, et fournit des réponses précises en exploitant des paramètres spécifiques (*slots*). Avec 24 intentions prédéfinies, dont certaines ont été suggérées par une IA générative (ChatGPT), l'agent extrait des informations pertinentes pour répondre aux questions des utilisateurs. Ce projet s'inscrit dans l'évolution des modèles de langage (LLM) et des technologies d'intelligence artificielle, visant à simuler des interactions humaines complexes et à fournir des réponses adaptées et contextualisées.

Mots-clés : Chatbot, Intentions, Slots, prompts, LLM.

Table des matières

Table des figures

Liste des tableaux

Chapitre 1

Introduction

L'intelligence artificielle commence avec des philosophes comme Leibniz avec sa machine de calcul théorique appelée *Calculus ratiocinator* en 1666 Leibniz (1880), concept qu'il ne poursuivra pas mais qui sera repris au XIX^{e} siècle par d'autres scientifiques comme George Boole. Il imagine en effet un procédé automatique couplant ce langage et un algorithme capable de vérifier la vérifier la véracité d'une information et également d'énoncer des théories. Il s'agit là des prémisses de ce qu'on appelle la machine à raisonner. Au XX^{e} siècle, les scientifiques commencent à explorer l'idée d'un cerveau électronique pour faire allusion à l'ordinateur doté d'un processeur. Par la suite, avec l'évolution de l'informatique et des microprocesseurs, ils parviennent à créer des interfaces hommes-machines ou *IHM* permettant l'interaction entre l'homme et la machine à travers un ordinateur à l'écrit, ou d'un robot dans le domaine de la robotique industrielle. De là naît le "dialogueur" ou encore "l'agent conversationnel" qu'on appelle en anglais *chatbot*. L'agent conversationnel est un programme interagissant avec un humain par l'intermédiaire de dialogues en langage naturel. Il capable d'effectuer beaucoup de tâches comme bavarder, rechercher des informations, la création et la maintenance de profils utilisateurs (ressources humaines, gestions des risques pour des assurances, etc.). Ces dialogueurs au départ fonctionnaient par un ensemble de règles et de sorties, où des correspondances de motifs étaient faites. Ils étaient en effets programmés sur la base d'un système de règle qu'on appelait "système expert". Et également ils fonctionnaient par apprentissage automatique à partir d'une base de dialogues ou corpus de dialogues. De nos jours, la plupart des agents conversationnels ne fonctionnent plus sur cette base comme *ChatGPT* (OpenAI 2023) sont boostés avec de l'intelligence artificielle.

L'IA est donc un ensemble de théories et de techniques mises en œuvres en vue de réaliser des machines capables de simuler les cognitives de l'homme. Pour améliorer donc le fonctionnement des *chatbots* avec l'IA, on a les grands de modèles de langage (*LLM*) qui sont la suite des modèles de langages qui existent déjà. Ils s'agit des modèles statistiques, basés sur une architecture neuronale, d'une suite de symboles qu'on appelle des *tokens* qui sont entraînés sur de vastes ensembles de données. L'architecture neuronale est un ensemble de plusieurs couches de neurones interconnectées qui apprennent à identifier les formes des données d'entrée et génèrent des formes des données de sortie. Les formes sont donc des objets à identifier et le token est une suite de lettres. Pour la création de notre agent conversationnel sur la vie au Moyen-Âge, basé sur des intentions (*intents*) de utilisateur et des paramètres spécifiques (*slots*) associées aux intentions, nous construirons des requêtes d'analyse du prompt pour détecter les *intents* et les associer à leurs valeurs de *slots* respectives pour que le modèle puisse fournir des réponses adaptées aux requêtes de l'utilisateur pour recevoir une information spécifique comme sur l'organisation de la société au moyen-âge par exemple.

Chapitre 2

Choix de l'application

2.1 Agent conversationnel : La vie au Moyen-Âge

L'agent conversationnel qui est mis en place est un agent couvrant en partie le sujet de la vie au Moyen-Âge. C'est-à-dire que l'agent, par l'analyse *d'intents* de l'utilisateur fournira des réponses selon l'information que l'utilisateur veut sur la vie au Moyen-Âge. Par exemple si l'utilisateur pose la question *"Comment est organisée la société médiévale ?"*, l'agent conversationnel fournit des réponses concernant cet *intent* spécifiquement, "1. Organisation de la société médiévale" en extrayant des valeurs du *slot* associées à cet *intent*. En tout, nous avons 24 intentions.
Un corpus de dialogue représentatif n'a pas pu être implémenté pour représenter les dialogues entre l'utilisateur et le serveur conversationnel pour chaque sous-tâche, d'où l'absence de modèle HTA et de classification de dialogue. Il n'y a donc pas des unités de dialogues spécifiques. Il faut aussi dire que certaines intentions et *slots* ont été suggérées par une IA générative, en l'occurrence *ChatGPT*.
En mode prompt donc, le chatbot fourni des réponses, mais elles auraient pu être affinées avec un corpus de dialogues, des unités de dialogues et encore mieux avec des branchements entre ces dialogues de sorte qu'il cherche des informations dans 2 ou 3 dialogues pour fournir des réponses enrichies.
L'application est disponible sur Google Colab est fonctionnelle. Le tableau des intentions et slots est présenté sur la page suivante.

Table 2.1 : Tableau récapitulant les intentions et leurs *slots* associés

Numéro d'Intention	**Intention**	***Slots***
1	Organisation de la société médiévale	Seigneur, Chevalier, Paysan, Moine, Artisan
2	Mode de vie des seigneurs et chevaliers	Alimentation, Logement, Activités quotidiennes
3	Vie des paysans et artisans	Travail, Outils, Conditions de vie
4	Fonctionnement des châteaux forts	Défense, Architecture, Pièce principale
5	Religions et croyances	Croyances, Religieux, Pratiques
6	Apprentissage et éducation	Écoles monastiques, Universités, Maîtres et élèves
7	Commerce et échanges économiques	Monnaies, Routes commerciales, Marchés médiévaux
8	Maladies et médecine médiévale	Types de maladies, Remèdes, Médecins et guérisseurs
9	Rôles et place des femmes	Noblesse, Paysannerie, Monastères, Métiers
10	Tournois et divertissements	Joutes, Banquets, Jeux populaires
11	Guerre et stratégies militaires	Armures, Armes, Techniques de combat
12	Justice et droit féodal	Tribunaux, Châtiments, Serments
13	Religion et hérésie	Croisades, Inquisition, Saints et pèlerinages
14	Villes et villages médiévaux	Urbanisme, Rôles des corporations, Artisanat
15	Construction des cathédrales et églises	Architecture, Art gothique, Métiers impliqués
16	Nourriture et cuisine médiévale	Plats typiques, Ingrédients utilisés, Techniques de cuisson
17	Vie quotidienne des moines et moniales	Règles monastiques, Travail, Prière
18	Fonctionnement du système féodal	Suzeraineté, Hommage, Fiefs
19	Pirates et bandits au Moyen Âge	Routes dangereuses, Répression des brigands
20	Superstitions et magie	Sorcellerie, Astrologie, Amulettes
21	Art et musique médiévale	Instruments, Peinture, Littérature
22	Épidémies et famines	Peste noire, Disettes, Réactions des populations
23	Coutumes et mariages	Mariage nobles vs paysans, Traditions, Dotes
24	Sortir de la conversation	

Chapitre 3

Modélisation des productions de l'utilisateur

Pour détecter donc les intentions de l'utilisateur en mode prompt, les *intents* sont analysés en les mettant dans un dictionnaire avec un numéro d'intention attribué. Le numéro d'intention permettra de confirmer l'intention de l'utilisateur quand il saisit sa requête.

```
Posez votre question (ou tapez 'quit' pour arrêter) : Quelle était l'organisation de la société au moyen âge ?
Intent trouvé : 1. Organisation de la société médiévale
Numéro d'intention extrait : 1
```

Figure 3.1 : Exemple 1 de détection de l'intention "1. Organisation de la société médiévale"

Pour cette première requête posée au LLM "Quelle était l'organisation de la société au Moyen-âge ?" on voit que l'*intent* est trouvée ainsi que son numéro. Son numéro est extrait avec une expression régulière. Cela ainsi fait, on peut donc tester la détection d'intentions sur d'autres questions comme : *"Quel était le mode de vie des seigneurs et des chevaliers au Moyen Âge ?" "À quoi servaient les châteaux forts au Moyen Âge ?" "Comment l'éducation était-elle dispensée au Moyen Âge ?"*

Sur ce point, on obtient presque toujours de bonnes réponses *d'intents* et aussi de *slots* et de valeurs de *slots* associées.
Il faut préciser que la détection d'intentions est nettement améliorée car à implémenter un mappage des mots clés pour chaque intention. C'est-à-dire qu'on a mots-clés différents associés aux intentions qui accélèrent la correspondance intentions - questions de l'utilisateur.

C'est par exemple :

```
keywords_mapping = {
    ("organisation", "structure", "hiérarchie") : "1. Organisation de la
société médiévale",
    ("seigneur", "seigneurs", "noble", "vassal", "chevalerie") : "2. Mode
de vie des seigneurs et chevaliers",
    ("paysan", "paysans", "agriculture", "fermier", "terres") : "3. Vie
des paysans et artisans"
}
```

Ces mots-clés vont permettre d'associer plus facilement les questions avec leur intention

respective, écrive par exemple *"Quelle est l'organisation de la société au Moyen-Âge ?" ou simplement "organisation" ou "structure" ou "hiérarchie"* il y a toujours une correspondance avec le même *intent*.

```
Modèle chargé sur le device : cpu
Détection d'intention et extraction de slots avec le modèle.
Tapez 'quit' ou 'exit' pour terminer.

Posez votre question (ou tapez 'quit' pour arrêter) : organisation
Intent trouvé : 1. Organisation de la société médiévale
Numéro d'intention extrait : 1
```

Figure 3.2 : Exemple 2 de détection de l'intention "1. Organisation de la société médiévale"

```
Détection d'intention et extraction de slots avec le modèle.
Tapez 'quit' ou 'exit' pour terminer.

Posez votre question (ou tapez 'quit' pour arrêter) : Quelle était la hiérarchie de la société au moyen âge ?
Intent trouvé : 1. Organisation de la société médiévale
```

Figure 3.3 : Exemple 3 de détection de l'intention "1. Organisation de la société médiévale"

Également, le texte est converti en minuscule pour une correspondance insensible à la casse.

Ensuite on fait une correspondance entre les intentions détectées avec les mots-clés ou non avec leurs *slots* respectifs. Les *slots* qui sont des paramètres associés aux *intents* qui vont permettre de décrire l'*intent* en question. C'est ici qu'un corpus de dialogues avec des situations de dialogues aurait pu être fait pour que le modèle décrive exactement *l'intent* avec le *slot* adapté, parce que le système de correspondance actuel *intent* - *slot* fonctionne certes, mais donne des réponses trop générales lorsque le modèle les utilise pour répondre à une question.

```
def main():
intent_to_slots_and_keywords = {
     1: ("Organisation de la société médiévale", ["Seigneur",
"Chevalier", "Paysan", "Moine", "Artisan"]),
     2: ("Mode de vie des seigneurs et chevaliers", ["Alimentation",
"Logement", "Activités quotidiennes"]),
     3: ("Vie des paysans et artisans", ["Travail", "Outils", "Conditions
de vie"]),
}
```

Au-dessus on la correspondance des *intents* aux *slots*.

Une fois les intentions et les *slots* associés, on est en mesure d'extraire les valeurs de ces *slots* pour répondre à la question posée par l'utilisateur en fonction de son intention.

```
Détection d'intention et extraction de slots avec le modèle.
Tapez 'quit' ou 'exit' pour terminer.

Posez votre question (ou tapez 'quit' pour arrêter) : Quelle était la hiérarchie de la société au moyen âge ?
Intent trouvé : 1. Organisation de la société médiévale
Numéro d'intention extrait : 1
Slots disponibles :  ['Seigneur', 'Chevalier', 'Paysan', 'Moine', 'Artisan']
réponses attendues :  Le seigneur est le propriétaire des terres.
réponses attendues :  Un chevalier est un combattant lourdement armé.
réponses attendues :  Les paysans travaillent la terre et produisent de la nourriture.
réponses attendues :  Les moines vivent dans des monastères et suivent une règle religieuse stricte.
réponses attendues :  Les artisans sont des travailleurs spécialisés, comme des forgerons, des tisserands ou des potiers.
```

Figure 3.4 : Slots extraits pour l'*intent* "1. Organisation de la société médiévale"

Dans cette sortie, il ne devrait pas avoir écrit les valeurs des *slots* et "réponses attendues" car elles étaient affichées volontairement par un `print` pour observer les sorties du modèle et les réponses attendues qui sont dans une base de donnée `JSON`.

Chapitre 4

Modélisation des productions du serveur

4.1 LLM en mode prompt

Ici on fait en sorte que le modèle réponde non pas avec ses connaissances apprises mais à partir de connaissances provenant d'une base de données `JSON` qui contient les descriptions des *slots* trouvés. Le modèle avec lequel le travail a été fait est essentiellement avec `Llama-3.2-1B-Instruct` qui n'a certes pas beaucoup de paramètres mais qui donne quand même des réponses pertinentes.

```
def generate_full_narrative_response(intent_name, slots, tokenizer,
model, device):
     """ génère une réponse narrative complète en utilisant toutes les
valeurs des slots pour l'intention donnée  """
     prompt = f"Donne un aperçu complet sur {intent_name}. "
     print("Slots disponibles : ", slots)
     for slot in slots:
         description = extract_slot(slot, intent_name, tokenizer, model,
device)
         reponses_attendues = description
         prompt += f"{slot}: {description}. "
     response = call_LLM(prompt, tokenizer, model, device)
     llm_response = response
     print("réponses du modèle : ", llm_response)
```

Prompt : `prompt = f"Donne un aperçu complet sur {intent_name}. "`

On pourrait tester d'autres formulations de prompts mais celle-ci marche plutôt bien et ça prend du temps au modèle pour générer une réponse.

Le modèle de Llama *instruct* de 3 milliards de paramètres semble mieux générer des réponses cohérentes en rapport avec la question et l'intention posée en allant chercher les réponses dans une base de donnée. Mais parfois, que ce soit le modèle à 1 milliards ou 3 milliards de paramètres, on a constaté que le modèle hallucine un peu. C'est-à-dire qu'il donne les réponses pertinentes de la base de données mais en rajoute un peu provenant de

ses connaissances à lui. Cela arrive le plus souvent quand la température est proche de 1 c'est-à-dire 0.8 ou 0.9 par exemple. Mais lorsqu'on règle celle-ci très proche de 0 par exemple `temperature=0.2` les hallucinations sont réduites.

Il faut également régler le bon nombre de *tokens* à générer par le modèle, plus celui-ci est élevé plus le modèle va fournir plus de réponses plus il est susceptible d'halluciner s'il dépasse les données disponibles dans sa base de données.

```
Slots disponibles :  ['Seigneur', 'Chevalier', 'Paysan', 'Moine', 'Artisan']
réponses attendues :  Le seigneur est le propriétaire des terres.
réponses attendues :  Un chevalier est un combattant lourdement armé.
réponses attendues :  Les paysans travaillent la terre et produisent de la nourriture.
réponses attendues :  Les moines vivent dans des monastères et suivent une règle religieuse stricte.
réponses attendues :  Les artisans sont des travailleurs spécialisés, comme des forgerons, des tisserands ou des potiers.
Setting `pad_token_id` to `eos_token_id`:128001 for open-end generation.
réponses du modèle :  Donne un aperçu complet sur Organisation de la société médiévale. Seigneur: Le seigneur est le propriétaire des te
rres.. Chevalier: Un chevalier est un combattant lourdement armé.. Paysan: Les paysans travaillent la terre et produisent de la nourritu
re.. Moine: Les moines vivent dans des monastères et suivent une règle religieuse stricte.. Artisan: Les artisans sont des travailleurs
spécialisés, comme des forgerons, des tisserands ou des potiers.. 1.  **Organisation de la société médiévale** : La société médiévale ét
ait une société organisée en trois classes principales : les seigneurs, les paysans et les artisans. Les seigneurs possédaient les terre
s, les paysans travaillaient la terre et produisaient de la nourriture, tandis que les artisans étaient des travailleurs spécialisés. 2.
  **Seigneurs** : Les seigneurs
```

Figure 4.1 : Réponse du modèle à la question "Quelle était la hiérarchie de la société au moyen âge ?"

Chapitre 5

Table de données

Pour la table de données construite, elle est en format JSON. Elle comprend les intentions principales, les *slots* et les valeurs associées à ces *slots* nécessaires pour le modèle. Le fichier est `database.json`. Ce sont des données statiques qui peuvent être enrichies à tout moment.

```
"Organisation de la société médiévale": {
     "Seigneur": "Le seigneur est le propriétaire des terres.",
     "Chevalier": "Un chevalier est un combattant lourdement armé.",
     "Paysan": "Les paysans travaillent la terre et produisent de la
nourriture.",
     "Moine": "Les moines vivent dans des monastères et suivent une règle
religieuse stricte.",
     "Artisan": "Les artisans sont des travailleurs spécialisés, comme
des forgerons, des tisserands ou des potiers."
     },
     "Mode de vie des seigneurs et chevaliers": {
     "Alimentation": "Viande, pain, légumes, soupe.",
     "Logement": "Châteaux et manoirs, grandes chambres avec lits en
bois.",
     "Activités quotidiennes": "Chasse, entraînement militaire, gestion
des terres."
     }
```

Pour une raison encore difficile à trouver mais sûrement à cause d'une mauvaise correspondance, le modèle ne renvoie pas de résultats sous prétexte que certaines intentions n'ont pas de *slots* définis alors que les *slots* sont bien définies dans le JSON. Il faudrait analyser chaque intention définie dans JSON et chaque intention dans le code alors qu'il s'agit des mêmes.

Chapitre 6

Déploiement du serveur

Les ressources développées sont déployées sur Google Colab ainsi que tous les fichiers. Les dernières conversations sont également enregistrées dans un fichier `log` pour une analyse des conversations.

Chapitre 7

Évaluation des réponses aux questions du LLM et conclusion

Nous avons donc travaillé avec le modèle `Llama 3.2-1B-Instruct` *instruct* qui est un modèle qui pour objectif de suivre des instructions précises. Il est en effet optimisé, c'est-à-dire qu'il entraîné pour répondre aux requêtes directes de manière concise et efficace. Pour l'évaluer d'un point de vue humain avant d'utiliser des métriques, on constate que le modèle répond très généralement plutôt bien aux questions posées par l'utilisateur, c'est-à-dire qu'il donne au minimum l'information que l'utilisateur souhaite avoir sur la vie au Moyen-Âge concernant des *intents* spécifiques pour `Llama-3.2-1B-Instruct`.

Pour évaluer correctement le modèle, il faudrait utiliser un *benchmark* spécifique de compréhension et de génération de texte. Dans ce projet, notre tâche étant une tâche de réponse à des questions, on pourrait utiliser le SQuAD Rajpurkar et al. (2016) pour tester la compréhension du texte par le modèle.

Mais on pourrait aussi évaluer tout simplement utilisant la similarité cosinus par exemple pour comparer la similarité entre les sorties générées par le modèle et les réponses attendues qu'on connaît déjà qui sont les valeurs des *slots*.

```
    # Stocker la réponse générée par le LLM dans la variable llm_response
    llm_response = response
    print("Réponses du modèle : ", llm_response)

    # Évaluer la similarité entre la réponse du modèle et la réponse
attendue
    similarity_score = evaluate_similarity(llm_response,
reponses_attendues)
    print(f"Similarité entre la réponse du modèle et la réponse attendue :
{similarity_score:.2f}")
```

On peut donc faire en sorte que le taux de similarité soit calculé pour une liste de questions-réponses et ainsi voir à peu près où se situe les performances du modèle même si ce n'est pas une méthode parfaite. Les résultats sont donc des similarités sont enregistrées dans fichier `similarity_results.csv` et on peut générer un graphique pour observer le taux de similarité de chaque réponse du modèle par rapport à une question posée.

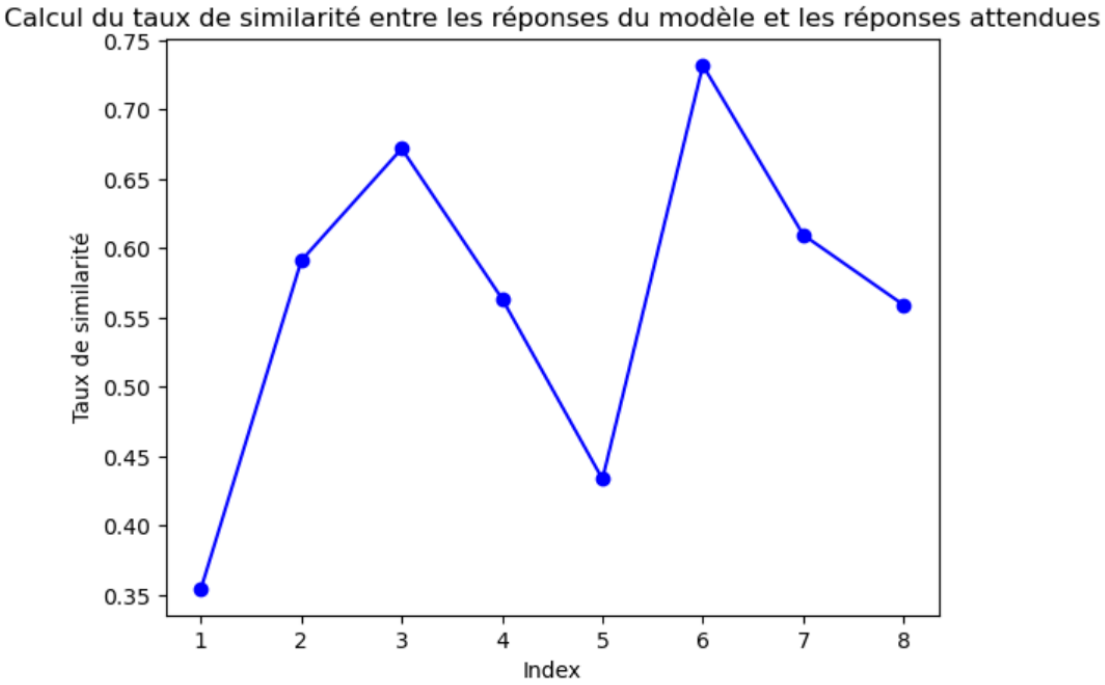

Figure 7.1 : Taux de similarités des réponses générées par le modèle et des réponses attendues (valeurs des slots)

Dans l'ensemble, on remarque que la comparaison des taux de similarités de 6 réponses du modèle donc de 6 réponses attendues sont supérieures à 0.5 soit 50% , ce qui veut dire que le modèle fourni des réponses pertinentes dans la plupart des cas. Un faible taux s'explique par :

- **Valeurs de *slots* non détectées :** si des valeurs de slots ne sont pas correctement détectées, le modèle peut fournir une réponse générique ou incorrecte, ce qui réduit la similarité avec la réponse attendue.
- **Répétition de mots :** le modèle peut répéter des mots qu'il a générés, ce qui fausse la similarité entre sa réponse et la réponse attendue.
- **Hallucinations du modèle :** le modèle peut "halluciner" en ajoutant des informations supplémentaires issues de sa base de connaissances, ce qui entraîne une divergence par rapport à la réponse attendue.

Dans une perspective d'amélioration des réponses de notre agent conversationnel, on pourrait essayer de :

- **Améliorer les prompts :** avec des techniques de *prompt engineering* au modèle pour générer une réponse.
- **Passage à un modèle supérieur :** tester des modèles plus performants, comme `Llama 3.2-3B-Instruct` ou d'autres modèles plus grands, pour observer l'amélioration des réponses.
- **Vectoriser la base en RAG :** mettre en place une vectorisation de la base en utilisant l'architecture RAG (Retrieval-Augmented Generation), ce qui permettrait d'améliorer la pertinence des réponses en combinant recherche et génération.
- **Finetuning du modèle :** fine-tuner le modèle avec nos descriptions spécifiques, bien que cela nécessite une architecture matérielle adaptée pour l'entraînement.

Bibliographie

Leibniz, G. W. (1880), 'Dissertatio de arte combinatoria (1666)', *Die philosophishe Schriften. Berlin* pp. 27–102.

Rajpurkar, P., Zhang, J., Lopyrev, K. and Liang, P. (2016), 'Squad : 100,000+ questions for machine comprehension of text', *arXiv preprint arXiv :1606.05250* .