

# Projet industriel

---

Amélioration des modèles IA génératifs de texte

**Serge Téhé, Soulaïman Marsou, Jules Singer**

Année 2023–2024

Projet réalisé en partenariat avec Sopra Steria

Encadrants industriels : Tabara Mbaye, Alice Petit, Jules Duarte Vala, Patrick Meyer

Encadrant académique : Hervé Panetto



# Déclaration sur l'honneur de non-plagiat

**Je soussigné(e),**

**Nom, prénom : Téhé, Serge**

**Élève-ingénieur(e) régulièrement inscrit(e) en 3<sup>e</sup> année à TELECOM Nancy**

**Numéro de carte de l'étudiant(e) : 32026573**

**Année universitaire : 2023–2024**

**Auteur(e) du document, mémoire, rapport ou code informatique intitulé :**

**Amélioration des modèles IA génératifs de texte**

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

**Fait à Nancy, le 28 février 2024**

**Signature :**





# Déclaration sur l'honneur de non-plagiat

Je soussigné(e),

Nom, prénom : Marsou, Soulaïman

Élève-ingénieur(e) régulièrement inscrit(e) en 3<sup>e</sup> année à TELECOM Nancy

Numéro de carte de l'étudiant(e) : 32119921

Année universitaire : 2023–2024

Auteur(e) du document, mémoire, rapport ou code informatique intitulé :

Amélioration des modèles IA génératifs de texte

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

Fait à Nancy, le 28 février 2024

Signature :





# Déclaration sur l'honneur de non-plagiat

**Je soussigné(e),**

**Nom, prénom : Singer, Jules**

**Élève-ingénieur(e) régulièrement inscrit(e) en 3<sup>e</sup> année à TELECOM Nancy**

**Numéro de carte de l'étudiant(e) : 31905961**

**Année universitaire : 2023–2024**

**Auteur(e) du document, mémoire, rapport ou code informatique intitulé :**

**Amélioration des modèles IA génératifs de texte**

Par la présente, je déclare m'être informé(e) sur les différentes formes de plagiat existantes et sur les techniques et normes de citation et référence.

Je déclare en outre que le travail rendu est un travail original, issu de ma réflexion personnelle, et qu'il a été rédigé entièrement par mes soins. J'affirme n'avoir ni contrefait, ni falsifié, ni copié tout ou partie de l'œuvre d'autrui, en particulier texte ou code informatique, dans le but de me l'accaparer.

Je certifie donc que toutes formulations, idées, recherches, raisonnements, analyses, programmes, schémas ou autre créations, figurant dans le document et empruntés à un tiers, sont clairement signalés comme tels, selon les usages en vigueur.

Je suis conscient(e) que le fait de ne pas citer une source ou de ne pas la citer clairement et complètement est constitutif de plagiat, que le plagiat est considéré comme une faute grave au sein de l'Université, et qu'en cas de manquement aux règles en la matière, j'encourrais des poursuites non seulement devant la commission de discipline de l'établissement mais également devant les tribunaux de la République Française.

**Fait à Nancy, le 28 février 2024**

**Signature :**





# Projet industriel

---

Amélioration des modèles IA génératifs de texte

**Serge Téhé, Soulaïman Marsou, Jules Singer**

**Année 2023–2024**

Projet réalisé en partenariat avec Sopra Steria

Serge Téhé, Soulaïman Marsou, Jules Singer  
[yohou.tehe@telecommancy.eu](mailto:yohou.tehe@telecommancy.eu), [soulaiman.marsou@telecommancy.eu](mailto:soulaiman.marsou@telecommancy.eu), [jules.singer@telecommancy.eu](mailto:jules.singer@telecommancy.eu)

TELECOM Nancy  
193 avenue Paul Muller,  
CS 90172, VILLERS-LÈS-NANCY  
+33 (0)3 83 68 26 00  
[contact@telecommancy.eu](mailto:contact@telecommancy.eu)

SOPRA STERIA  
6 avenue Kleber  
75016, Paris, France  
+33 (0)1 40 67 29 29

Encadrant industriel : Tabara Mbaye, Alice Petit, Jules Duarte Vala, Patrick Meyer

Encadrant académique : Hervé Panetto



## **Remerciements**

*Nous tenons à remercier les ingénieurs de Sopra Steria, Tabara Mbaye, Alice Petit et Jules Duarte Vala, pour leur accompagnement et leurs précieux conseils tout au long de ce projet. Leur expertise et leur soutien ont été d'une aide inestimable et ont grandement contribué à la réussite de notre travail.*

*Nous souhaitons également remercier notre encadrant académique, Hervé Panetto et plus globalement Telecom Nancy et Sopra Steria pour avoir rendu possible cette collaboration nous donnant l'opportunité de travailler sur ce projet.*

*Cette expérience a été enrichissante à bien des égards et nous sommes reconnaissants à toutes les personnes impliquées pour leur contribution significative à ce projet.*

*Soulaïman, Serge et Jules.*



# Table des matières

<b>Remerciements</b>	<b>ix</b>
<b>Table des matières</b>	<b>xi</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Présentation de l'entreprise</b>	<b>3</b>
<b>3 Présentation du projet</b>	<b>4</b>
<b>4 État de l'art</b>	<b>5</b>
4.1 Généralités sur les LLMs . . . . .	5
4.1.1 Introduction . . . . .	5
4.1.2 Principe des transformateurs . . . . .	5
4.1.2.1 Encodeur . . . . .	5
4.1.2.2 Décodeur . . . . .	7
4.1.2.3 Classificateur linéaire et softmax final pour probabilités de sorties	7
4.2 Méthodes d'apprentissage . . . . .	8
4.2.1 Introduction aux méthodes d'apprentissage des LLMs . . . . .	8
4.2.2 Phase d'alignement . . . . .	9
4.2.2.1 Affinement supervisé(Supervised fine-tuning) . . . . .	9
4.2.2.2 Apprentissage par renforcement à partir des retours des humains(Reinforcement learning from human feedback RLHF) . . . . .	11
4.3 Méthodes d'amélioration . . . . .	11
4.3.1 Introduction à l'amélioration des modèles . . . . .	11
4.3.2 <i>Prompt engineering</i> . . . . .	12
4.3.3 Compression . . . . .	14
4.3.3.1 Quantification (alias quantization) . . . . .	15
4.3.3.2 Elagage (alias pruning) . . . . .	17
4.3.4 Le format et l'environnement d'exécution . . . . .	18

4.3.4.1	ONNX . . . . .	19
4.3.4.2	GGUF . . . . .	19
4.3.4.3	DeepSparse . . . . .	20
4.3.5	Conclusion sur l'amélioration des modèles . . . . .	21
4.4	Méthodes de comparaisons des LLMs . . . . .	21
4.4.1	Introduction à la comparaison des LLMs . . . . .	21
4.4.2	Organisation actuelle de l'évaluation des LLMs . . . . .	22
4.4.3	Présentations de quelques métriques . . . . .	23
4.4.3.1	Taille des LLMs . . . . .	23
4.4.3.2	Rappel . . . . .	24
4.4.3.3	Précision . . . . .	24
4.4.3.4	F1 score . . . . .	25
4.4.3.5	Perplexité . . . . .	25
4.4.3.6	Pass@k . . . . .	26
4.4.3.7	Zero-shot learning . . . . .	27
4.4.3.8	CodeBERTScore . . . . .	28
4.4.4	Conclusion méthodes de comparaisons . . . . .	29
<b>5</b>	<b>Cas d'application</b> . . . . .	<b>30</b>
5.1	Introduction . . . . .	30
5.1.1	Objectifs et contexte du cas d'application . . . . .	30
5.1.2	Le choix du modèle . . . . .	31
5.1.3	Les données . . . . .	31
5.2	Évaluation et comparaison des modèles . . . . .	32
5.2.1	Introduction . . . . .	32
5.2.2	Choix des métriques d'évaluation . . . . .	32
5.2.3	Mise en place d'une base de données . . . . .	33
5.2.4	Développement d'un système d'évaluation des modèles . . . . .	34
5.2.4.1	Génération des requêtes . . . . .	34
5.2.4.2	Génération des tests unitaires . . . . .	34
5.2.4.3	Évaluation du modèle . . . . .	35
5.3	Apprentissage . . . . .	35
5.3.1	Le prompt . . . . .	35
5.3.2	Formattage des données . . . . .	36
5.3.3	Entraînement . . . . .	36

5.3.3.1	Paramètres Lora . . . . .	36
5.3.3.2	Paramètres d'entraînement . . . . .	36
5.3.3.3	Résultats des entraînements . . . . .	37
5.3.3.4	Résultats et analyse des performances des modèles fine-tune . . . . .	38
5.4	Optimisation . . . . .	39
5.4.1	Introduction . . . . .	39
5.4.2	Le prompt . . . . .	39
5.4.3	Les environnements d'exécution . . . . .	39
5.4.4	Les différentes compressions . . . . .	41
5.5	Difficultés rencontrées et pistes d'améliorations . . . . .	43
<b>6</b>	<b>Évolution du Gantt</b>	<b>45</b>
<b>7</b>	<b>Conclusion</b>	<b>47</b>
<b>Bibliographie / Webographie</b>		<b>49</b>
<b>Liste des illustrations</b>		<b>53</b>
<b>Listings</b>		<b>55</b>
<b>Annexes</b>		<b>58</b>
<b>A</b>	<b>Génération des requêtes Cypher</b>	<b>59</b>
<b>B</b>	<b>Génération des tests unitaires</b>	<b>60</b>
<b>C</b>	<b>Évaluation des modèles</b>	<b>61</b>
<b>D</b>	<b>Charge de travail du projet</b>	<b>62</b>
<b>Résumé</b>		<b>63</b>
<b>Abstract</b>		<b>63</b>



# 1 Introduction

À l'ère de la transformation numérique, les modèles de langage à grande échelle (LLMs) se sont établis comme une technologie de pointe, influençant de manière significative le développement et l'innovation dans le secteur de l'intelligence artificielle. Leur impact sur l'économie et la société est profond, offrant des perspectives prometteuses pour les entreprises et les organisations cherchant à exploiter le potentiel du numérique. Ces modèles représentent non seulement une avancée technologique majeure, mais ils sont également porteurs d'enjeux économiques et sociaux considérables, nécessitant une attention particulière et une compréhension approfondie.

Économiquement, l'adoption des LLMs par les entreprises peut conduire à une transformation radicale de leurs opérations, de leur relation client et de leur capacité à innover. Ces modèles offrent des possibilités de personnalisation à grande échelle, d'automatisation intelligente et d'analyse de données textuelles, ouvrant ainsi la voie à de nouveaux services et produits. Cependant, l'intégration de ces technologies avancées s'accompagne d'investissements significatifs, non seulement en termes de coûts financiers pour le développement et la maintenance, mais aussi en termes de ressources humaines pour la formation, l'adaptation et la gestion de ces systèmes.

Face à ces coûts potentiels et à la complexité inhérente à l'implémentation des LLMs, la nécessité d'une recherche approfondie et d'une connaissance solide du domaine est primordiale. Comprendre en détail les mécanismes, les applications et les limites des LLMs est essentiel pour prendre des décisions éclairées et rentables. Cette compréhension permet aux entreprises d'évaluer correctement les avantages et les risques associés à ces technologies, d'identifier les meilleures stratégies d'implémentation et d'optimiser leur retour sur investissement.

De plus, en raison de leur nature évolutive et de leur relative immaturité, les LLMs sont caractérisés par des avancées technologiques fréquentes qui peuvent rapidement transformer le champ des possibilités et remettre en question les anciens résultats. Cette dynamique souligne l'importance d'une veille technologique active et d'une capacité d'adaptation rapide pour les entreprises souhaitant rester compétitives. Une recherche et une mise à jour constantes des connaissances dans ce domaine sont donc cruciales pour maintenir une compréhension à jour et exploiter efficacement ces technologies.

Sur le plan social, les implications des LLMs sont également significatives. Ils soulèvent des questions éthiques majeures, notamment en matière de respect de la vie privée, de biais algorithmiques et de l'impact sur l'emploi. Ces enjeux doivent être pris en compte dans le développement et l'application des LLMs, afin d'assurer une utilisation responsable de ces technologies.

Les LLMs constituent donc un domaine clé d'innovation avec des implications économiques et sociales significatives. Ce rapport doit permettre à Sopra Steria une compréhension approfondie de ces modèles, facilitant ainsi des décisions éclairées sur leur intégration future.



## 2 Présentation de l'entreprise

Sopra Steria, fusion de Sopra (1968) et Steria (1969) depuis 2014, s'impose aujourd'hui comme la deuxième entreprise de services du numérique (ESN) française et se classe parmi les cinq premiers acteurs du secteur en Europe. Au fil des années, l'entreprise a su s'étendre et asseoir sa présence à l'international, opérant désormais dans près de 30 pays. Cette expansion géographique est le reflet d'une croissance économique constante, illustrée par un chiffre d'affaires impressionnant de 5,1 milliards d'euros en 2022.

L'équipe de Sopra Steria, avec près de 55 000 collaborateurs déployés à travers le monde vise à guider et accompagner ses clients dans leurs projets numériques. En mettant l'accent sur l'innovation et la qualité, Sopra Steria se positionne comme un partenaire pour les entreprises cherchant à accomplir leur transition numérique.

Sopra Steria offre principalement deux services. D'abord, l'entreprise excelle dans le conseil en développement de logiciels, couvrant un large éventail de secteurs tels que l'aéronautique, la défense, la finance, les transports, les assurances, le secteur public et bien d'autres. Cette expertise multidisciplinaire permet à Sopra Steria de proposer des solutions sur mesure, allant du conseil stratégique à la conception et au déploiement de solutions logicielles innovantes.

Ensuite, Sopra Steria se distingue par son offre en tant qu'éditeur de logiciels spécialisés. L'entreprise développe des solutions ciblées pour répondre aux besoins spécifiques de divers secteurs, notamment la banque, les ressources humaines et l'immobilier. Ces logiciels sont conçus pour optimiser les opérations de leurs clients, apportant ainsi une valeur ajoutée significative à leur activité.

Enfin, Sopra Steria met l'accent sur l'innovation en faisant de l'intelligence artificielle, la blockchain, l'IoT, le cloud une priorité dans leur développement. Cet engagement envers l'innovation permet à l'entreprise de rester compétitive dans un marché en constante évolution et de répondre aux besoins changeants de ses clients. Par ailleurs, la collaboration avec des institutions académiques telles que Telecom Nancy souligne l'importance que Sopra Steria accorde à la R&D (recherche et développement), ainsi qu'à l'établissement de partenariats stratégiques pour le partage des connaissances et l'exploration de nouvelles frontières technologiques.

Cette approche, intégrant savoir-faire spécifique à chaque secteur, avancées technologiques et partenariats stratégiques, positionne Sopra Steria comme un acteur clé de la transformation numérique.

### **3 Présentation du projet**

Titre du projet : Amélioration des modèles IA générative de texte

Dans un contexte marqué par une demande croissante de la part des clients de Sopra Steria pour la mise en place d'agents conversationnels, le projet "Amélioration des modèles IA génératifs de texte" a été proposé. L'explosion de l'intelligence artificielle et l'émergence de nouveaux modèles de langages (LLMs - Large Language Models) ont généré un intérêt significatif, mais également mis en lumière un besoin crucial de compréhension et de maîtrise de ces technologies encore très récentes. Face à ces enjeux, le projet vise à réaliser une recherche approfondie sur les LLMs, afin de faciliter la future intégration de ces modèles chez les clients de Sopra Steria.

Les objectifs du projet industriel sont multiples. Tout d'abord, il s'agit de contribuer à l'optimisation des modèles génératifs existants, en terme de qualité de génération d'une part mais aussi en terme de temps d'inférence et de ressources utilisées. Ensuite, le projet vise à acquérir une vision plus claire et complète des différents modèles génératifs, ainsi que des méthodes disponibles pour leur amélioration. Parallèlement, il est essentiel de définir des protocoles d'évaluation rigoureux pour mesurer l'efficacité de ces différentes méthodes. Enfin, le projet aboutira à la production d'un document de synthèse, qui recensera l'ensemble des recherches menées et servira de référence pour les futurs développements dans ce domaine.

Le déroulement du projet industriel s'est articulé en deux phases principales. La première phase a consisté à établir un état de l'art des LLMs, en examinant d'abord les modèles dans leur ensemble, puis en se focalisant plus précisément sur les méthodes d'optimisation, d'apprentissage et de comparaison. Cette phase a permis de poser les fondations théoriques et techniques nécessaires à la poursuite du projet. La seconde phase a été consacrée à l'application pratique des connaissances acquises. Elle a impliqué l'entraînement de modèles en utilisant les méthodes étudiées lors de la première phase, ainsi que la comparaison de leurs résultats. Cette approche expérimentale a eu pour but de valider les théories et les techniques explorées, tout en fournissant des informations précieuses pour les développements futurs de l'entreprise.

Pour mener à bien ce projet, les compétences requises étaient multiples. Il fallait faire preuve de recul et d'esprit critique concernant les modèles, les méthodes d'amélioration et d'évaluation. Trouver ou créer des méthodes efficaces pour tester les modèles était également essentiel, tout comme une bonne organisation pour gérer la charge de travail. Pour faire face aux difficultés du projet, nous avons bénéficié de l'accompagnement de trois ingénieurs en intelligence artificielle de Sopra Steria : Tabara Mbaye, Alice Petit et Jules Duarte Vala. Leur expertise, leurs conseils et le temps qu'ils ont passé pour suivre les avancées ont été déterminants pour le succès du projet.

# 4 État de l'art

## Introduction

Cet état de l'art a pour but de proposer une introduction avancée aux Large Language Models (LLMs). Elle est destinée à toute personne ayant la volonté d'approfondir ses connaissances dans le but de se mettre à la pratique en entraînant ses premiers modèles. Ce chapitre sera décomposé en quatre grandes parties :

- Généralités sur les LLMs
- Méthodes d'apprentissage
- Méthodes d'amélioration
- Méthodes de comparaisons

## 4.1 Généralités sur les LLMs

### 4.1.1 Introduction

Cette partie a pour objectif de présenter la structure des LLMs ainsi que les différents mécanismes utilisés pour mieux aborder les futures recherches.

### 4.1.2 Principe des transformeurs

Un transformeur est un ensemble de réseaux neuronaux composés d'un encodeur et d'un décodeur dotés de capacité d'auto-attention lui permettant d'extraire le sens d'une séquence de texte et de comprendre les relations entre les mots et phrases qui composent cette séquence [18].

#### 4.1.2.1 Encodeur

L'encodeur permet d'extraire les relations entre les mots qui composent une séquence de texte afin de comprendre son sens. Nous allons expliquer les différents éléments constituant l'architecture de l'encodeur (voir partie gauche de la figure 4.1)

#### Incorporation(Embedding)

En entrée de l'encodeur, nous transformons la séquence de texte en langage naturel en vecteurs de nombres réels. En effet, L'incorporation de mots consiste à représenter les mots d'un vocabulaire par des vecteurs de nombres réels de sorte à réduire la dimension des vecteurs tout en capturant le

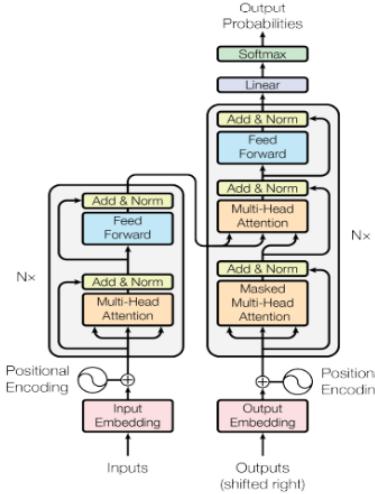


FIGURE 4.1 – Architecture des transformateurs [17]

contexte et les similarités sémantiques d'un mot. On utilise généralement le résultat d'une couche de neurones comme incorporation, cela revient à multiplier la matrice d'incorporation par la représentation one-hot [43] d'un mot du vocabulaire. Soit  $n$  la longueur maximale de la séquence de texte,  $N$  la taille du vocabulaire,  $d_{model}$  la dimension des incorporations ; on a alors les matrices des incorporations telles que présentées sur la figure 4.2.

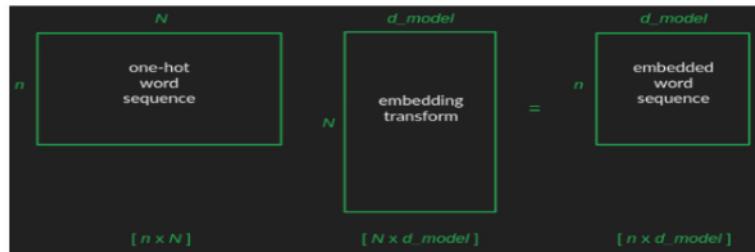


FIGURE 4.2 – Matrice des incorporations [21]

### Encodage positionnel

L'encodage positionnel consiste à injecter des informations de position dans les incorporations afin de permettre au module attention de comprendre l'ordre de chaque élément de la séquence en entrée [23] car les incorporations n'encodent pas les positions relatives de chaque mot dans la séquence.

### Mécanisme d'attention

Le mécanisme d'attention permet au modèle de se concentrer sur les éléments précis de la séquence d'entrée qui sont importants pour un mot afin de comprendre les relations entre les mots qui composent cette séquence. Pour mettre en place ce mécanisme, nous avons besoin d'une "query", d'une "key" et d'une "value". Une "query" représente le mot pour lequel on veut calculer l'attention, les "keys" représentent les différents mots de la séquence en entrée et une value contient l'information d'un mot. A chaque vecteur, est associé une "query", une "key" et une "value" représentant différentes projections de ce vecteur. Le produit entre une "query" et une "key" définit l'attention qu'il faut mettre sur cette "key" ; On applique ensuite la fonction softmax au produit entre les "queries" et les "keys" pour transformer les valeurs sous forme de probabilités [17]. Ainsi le modèle arrive à comprendre les relations entre les différents mots de la séquence.

### Attention multi-tête

Une tête d'attention (head attention) est un mécanisme qui permet au modèle de se concentrer sur différentes parties de l'entrée donnée au modèle. Une tête d'attention permet de calculer les poids d'attention entre les différentes parties de l'entrée. Ensuite, ces poids sont utilisés pour pondérer les informations, ce qui permet au modèle de donner plus d'importance à certaines parties de l'entrée lors de la prise de décision.

On utilise plusieurs têtes d'attention (multi-head attention) pour permettre au modèle de savoir se focaliser sur plusieurs mots simultanément. En effet, dans le cadre de l'utilisation d'une seule tête d'attention (single head attention), la fonction softmax amplifie les valeurs élevées et rabaisse les valeurs faibles ce qui fait qu'une tête d'attention aura plus tendance à se focaliser sur un seul mot. Avec ce mécanisme on arrive à savoir tous les mots en relation avec un mot donné dans la séquence.

#### 4.1.2.2 Décodeur

Le décodeur (figure 4.1 partie gauche) est auto-régressif [19], il permet de générer le prochain mot d'une séquence. Il commence par un jeton de démarrage et prend en entrée une liste des sorties précédentes, ainsi que les sorties de l'encodeur qui contiennent les informations d'Attention de l'entrée. Le décodeur arrête le décodage lorsqu'il génère le jeton de fin [23].

##### Attention multi-têtes masquées

Un masque est technique consistant à cacher certaines informations accessibles par un modèle à un moment donné. On applique l'attention multi-têtes aux entrées avec des masques pour que le mécanisme d'attention n'utilise pas les informations cachées qui représentent les futurs mots à prédire. Le masque permet d'appliquer une valeur d'attention fixée à l'infini négative pour les futurs mots à prédire de sorte que la fonction softmax mette toutes ces valeurs à 0 [39]. La figure 4.3 est une illustration de l'attention multi-têtes masquée.

```
(1, 0, 0, 0, 0, ..., 0) => (<sos>)
(1, 1, 0, 0, 0, ..., 0) => (<sos>, 'Bonjour')
(1, 1, 1, 0, 0, ..., 0) => (<sos>, 'Bonjour', 'le')
(1, 1, 1, 1, 0, ..., 0) => (<sos>, 'Bonjour', 'le', 'monde')
(1, 1, 1, 1, 1, ..., 0) => (<sos>, 'Bonjour', 'le', 'monde', '!')
```

FIGURE 4.3 – Illustration de l'attention multi-têtes masquée [39]

On applique ensuite le mécanisme d'attention multi-têtes entre le vecteur sortant de la couche précédente (masked multi-head attention) et les vecteurs sortant de l'encodeur [39]. Ici, la "query" est générée par le vecteur de la couche précédente ; Les "keys" et les "values" sont générées par les vecteurs sortant de l'encodeur.

#### 4.1.2.3 Classificateur linéaire et softmax final pour probabilités de sorties

La sortie de la couche "Feedforward" passe par une couche linéaire qui agit comme classificateur de classes où le nombre de classes est le nombre de mots du vocabulaire. On applique ensuite la fonction softmax pour produire les scores de probabilités. Ce fonctionnement, représentant la sortie du décodeur, est illustré sur la figure 4.4.

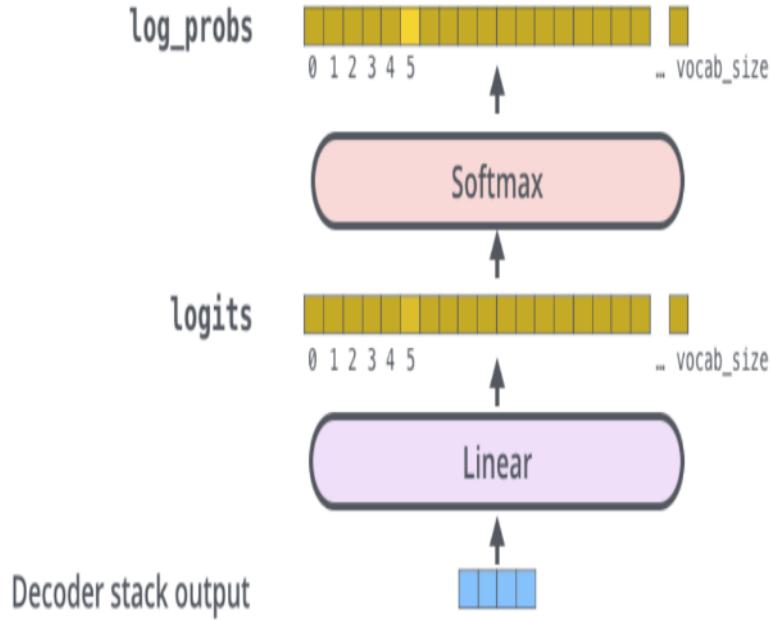


FIGURE 4.4 – Sortie du décodeur [16]

## 4.2 Méthodes d'apprentissage

### 4.2.1 Introduction aux méthodes d'apprentissage des LLMs

On distingue différentes phases d'entraînements des LLMs (figure 4.5).

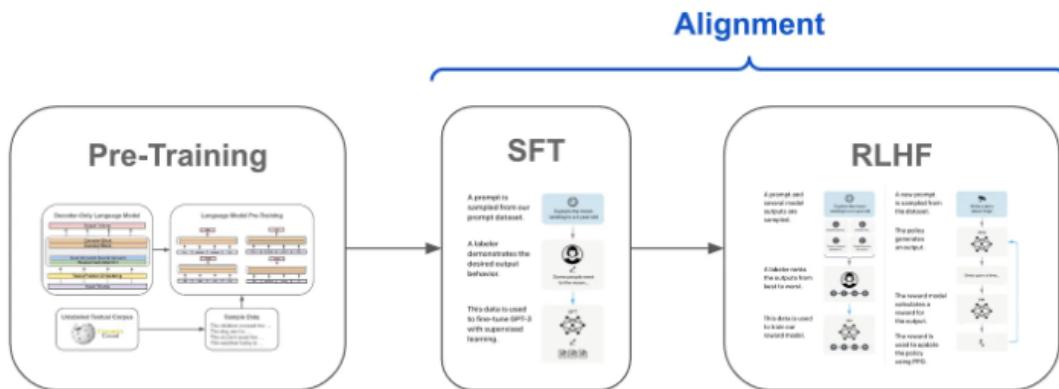


FIGURE 4.5 – Stades d'entraînement des LLMs [17]

Les LLMs sont d'abord **pré-entraînés** sur des corpus de texte très volumineux provenant de plusieurs sources hétérogènes comme le montre le jeu de données utilisé pour le modèle Llama (voir figure 4.6).

Ensuite, on peut utiliser ces LLMs pré-entraînés pour les spécialiser sur des tâches spécifiques ce qui constitue la **phase d'alignement**. Notre objectif pour ce projet étant de développer un LLM spécialisé sur un cas d'application spécifique qui est la génération de code en langage cypher, nous allons donc nous focaliser sur les méthodes de la phase d'alignement.

Dataset	Sampling prop.	Epochs	Disk size
CommonCrawl	67.0%	1.10	3.3 TB
C4	15.0%	1.06	783 GB
Github	4.5%	0.64	328 GB
Wikipedia	4.5%	2.45	83 GB
Books	4.5%	2.23	85 GB
ArXiv	2.5%	1.06	92 GB
StackExchange	2.0%	1.03	78 GB

FIGURE 4.6 – jeu de données du modèle Llama [31]

## 4.2.2 Phase d'alignement

### 4.2.2.1 Affinement supervisé(Supervised fine-tuning)

Le processus du "supervised fine-tuning" (SFT) consiste à entraîner un modèle pré-entraîné pour apprendre des motifs et nuances présents dans de nouvelles données labelisées [17]. Ces données labelisées représentent les données du cas d'application sur lequel on veut spécialiser le LLM. Contrairement à la phase de pré-entraînement, le fine-tuning nécessite moins de ressources et le dataset est de petite taille (figure 4.7). On distingue plusieurs méthodes de fine-tuning :

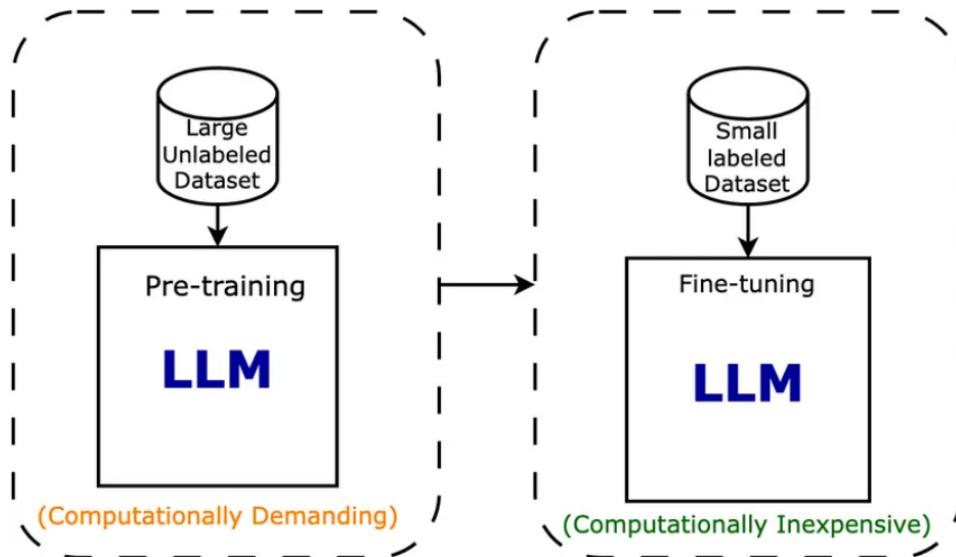


FIGURE 4.7 – Phases de pré-entraînement et de fine-tuning [17]

### Affinement complet(full fine-tuning)

Il s'agit d'un processus selon lequel, tous les poids du modèle sont mis à jour lors du processus d'entraînement. Il requiert beaucoup de mémoire et beaucoup de ressources comme le pré-entraînement pour traiter et stocker tous les gradients et optimiseurs du modèle [54]. Toutefois, il y a eu récemment le développement d'un nouvel optimiseur de type LOMO (Low memory optimizer) qui permet de réduire l'utilisation de la mémoire pour les mises à jour de tous les poids et composants d'un modèle LLM dans le cadre du full fine-tuning [35].

### Affinement efficace des paramètres(Parameter efficient tuning PEFT)

C'est une technique de transfert learning permettant de réduire l'utilisation de la mémoire et des

ressources de calculs en réduisant le nombre de paramètres entraînables. Cette méthode permet également de garder les informations précédemment apprises par le LLM en gelant la plus part des paramètres du modèle pré-entraîné [54]. On distingue différentes méthodes de PEFT :

### **Adaptation de bas rang(Low-rank adaptation Lora)**

La méthode Lora consiste à fixer les poids du modèle pré-entraîné et d'entrainer les poids de certaines couches du modèle pour permettre au modèle d'apprendre et/ou de se spécialiser une tâche tout en conservant les informations précédemment apprises (figure 4.8). Les nouveaux poids résultant de l'entraînement permettent de mettre à jour les poids du modèle pré-entraîné en utilisant 2 matrices de rang  $r$  très inférieures à la dimension de la matrice des poids du modèle pré-entraîné ce qui réduit le nombre de paramètres à stocker en mémoire. Par exemple, si la matrice des poids du modèle pré-entraîné est de dimension  $n^*d$ , avec la méthode Lora on stocke  $n^*r+r^*d$  paramètres au lieu de  $n^*d$  paramètres.

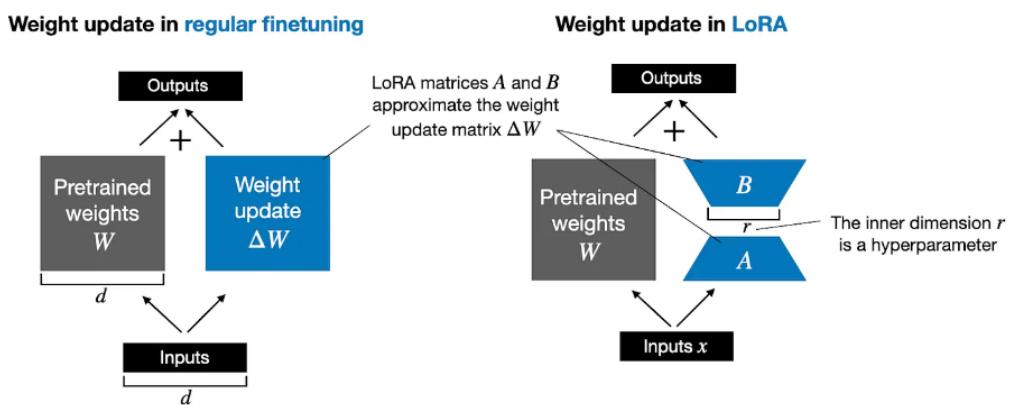


FIGURE 4.8 – Méthode Lora [51]

### **Affinement des préfixes(Prefix Tuning)**

C'est une méthode consistant à injecter des préfixes représentant une tâche spécifique au modèle pour qu'il ait un comportement adapté à la tâche voulue [52].

### **Couches adaptatives**

C'est le fait d'ajouter des couches de réseau de neurones au LLM pour améliorer ses performances pour le cas d'application tout en réduisant le nombre de paramètres à stocker en mémoire. On fixe les poids du modèle pré-entraîné et on entraîne les paramètres des couches rajoutées [61] (figure 4.9).

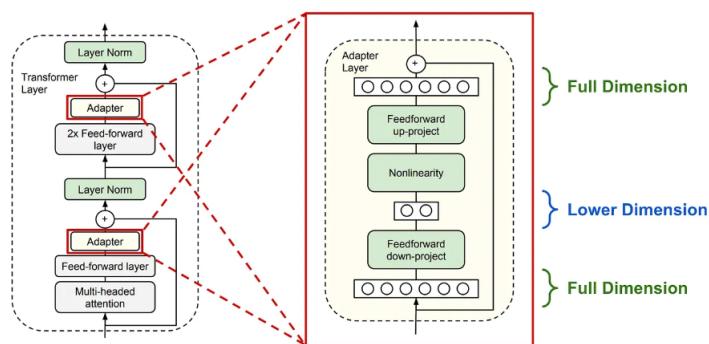


FIGURE 4.9 – Couches adaptatives [61]

#### 4.2.2.2 Apprentissage par renforcement à partir des retours des humains(Reinforcement learning from human feedback RLHF)

Le principe RLHF consiste à utiliser l'apprentissage par renforcement pour permettre au modèle d'avoir le comportement attendu en prenant en compte les retours des humains. En effet, le LLM génère plusieurs sorties différentes pour chaque entrée et des humains classent les sorties selon leur qualité [62]. La qualité est représentée par un score que le modèle cherche à maximiser en apprenant une politique pour matcher le comportement attendu (figure 4.10).

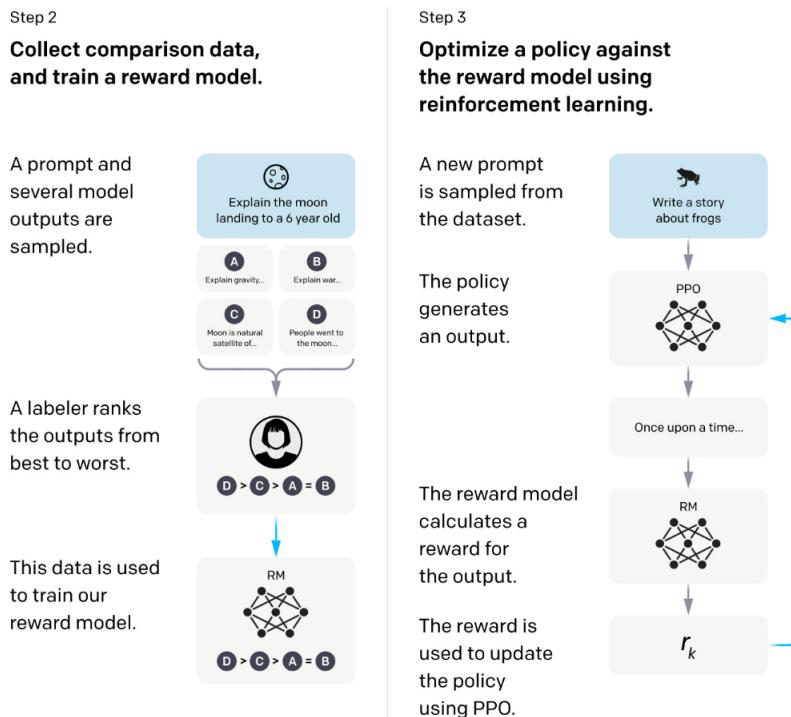


FIGURE 4.10 – Principe RLHF [62]

## 4.3 Méthodes d'amélioration

### 4.3.1 Introduction à l'amélioration des modèles

Nous supposons que nous avons un modèle de langage qui a été entraîné dans un but précis. En fonction du cas d'utilisation, les modèles peuvent nécessiter beaucoup de ressources de calcul, de mémoire et de stockage. Les plus gros LLMs ont tous été mis en production sur des environnements avec beaucoup de ressources et souvent très coûteux. Cependant, pour des cas d'utilisation particuliers, on pourrait avoir besoin des LLMs sur des environnements beaucoup plus petits comme un ordinateur personnel ou un système embarqué dans un satellite.

Dans d'autres cas d'utilisation où l'on aurait accès à beaucoup de ressources, il est naturel de se demander comment peut-on maximiser les performances. La question que l'on se pose donc est la suivante.

De quelle manière peut-on améliorer les modèles pour maximiser les performances tout en optimisant les ressources nécessaires ?

A cette question, nous proposons plusieurs axes de réflexion que l'on développera tout au long de

cette section :

- **Prompt engineering** : la forme du texte et la vectorisation obtenue qui est donné en entrée d'un LLM jouent un rôle très important dans les performances du modèle. Il existe des méthodes pour optimiser l'entrée qui est envoyé dans le modèle.
- **La compression** : c'est un ensemble de technique permettant d'optimiser le modèle dans le but d'alléger sa taille et les ressources nécessaires à son utilisation, tout en ayant une dégradation minimale. C'est ce qui permet à des domaines comme l'IoT de profiter de ces nouvelles technologies.
- **Le format du modèle et l'environnement d'exécution** : en choisissant un format optimisé à un environnement d'exécution précis, on peut accélérer le chargement du modèle ainsi que l'inférence. Tous les modèles sont exportés dans un format particulier lors de la mise en production. Il existe des environnements adaptés pour CPU comme pour GPU.

#### 4.3.2 *Prompt engineering*

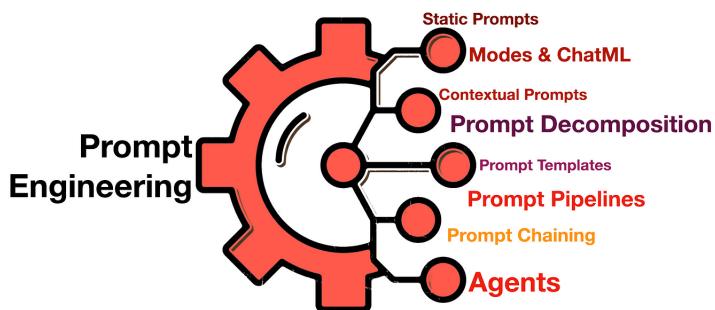


FIGURE 4.11 – Le *prompt engineering* et les disciplines associées. [28]

Le *prompt engineering* (figure 4.11) est une discipline dans laquelle on cherche à utiliser des techniques qui optimisent et enrichissent les données en entrée du modèle dans le but de maximiser les performances. Ces techniques peuvent se concentrer sur le formatage des données, l'utilisation d'un contexte que l'on concatène, ou même la modification ou l'ajout d'information dans le vecteur d'entrée ("vector embedding" en anglais).

L'avantage de cette discipline est qu'elle permet d'améliorer les performances tout en ne modifiant pas le modèle lui-même.

Il faut savoir également que dans le domaine du *prompt engineering*, on identifie deux types d'entrées :

- *Hard prompts* : Ce sont les entrées écrites en langage naturel. Ces entrées comptent un nombre discret de tokens.
- *Soft prompts* : Ce sont les entrées générées par un autre modèle. Elles sont générées directement sous forme de vecteurs. Ils ont l'avantage d'être plus performants, mais ont l'inconvénient de perdre en explicabilité.

Nous allons présenter quelques méthodes d'amélioration dans les paragraphes suivants. Il faut garder en tête que la discipline étant nouvelle, les termes se mélangent parfois un peu et un même terme

peut signifier plusieurs choses différentes. Voici quelques références pour comprendre les différentes significations : [28][40][53].

Pour approfondir vos recherches et vous éclaircir sur le sujet, vous pouvez vous renseigner sur le site web suivant pour plus de détails sur les différentes méthodes de *prompt engineering* : [22].

**Example-based Prompt Engineering** Il existe des techniques de *prompt engineering* qui ajoutent des exemples en entrée. Ces données supplémentaires vont aider le modèle à mieux comprendre ce qu'on attend de lui. Sur la figure 4.12, on peut observer ce principe à droite dans la partie "Few-shot Prompt", dans laquelle on fournit plusieurs exemples entrées/sorties au modèle au préalable.

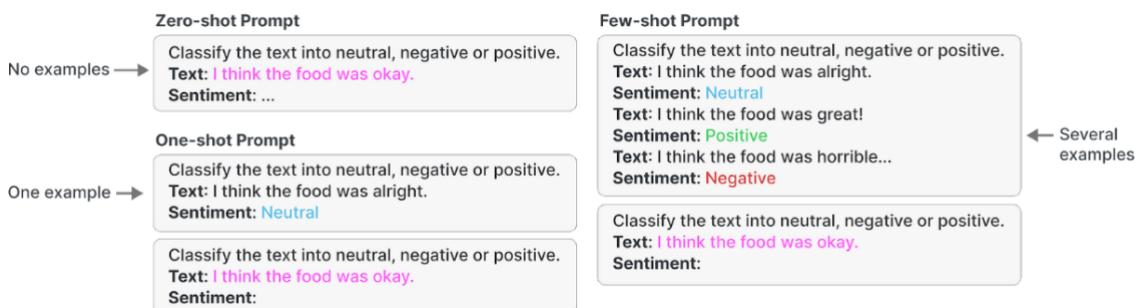


FIGURE 4.12 – Illustration d'entrées enrichies avec des exemples. [29]

**Thought-based Prompt Engineering** Pour certaines résolutions de problème, il est parfois plus efficace de découper la tâche que l'on veut résoudre en sous-tâches. C'est ce que l'on peut tous faire naturellement lorsque l'on essaie de résoudre un problème avec chatGPT de OpenAI. Lorsque la réponse à une sous-tâche est générée, celle-ci est renvoyée en entrée pour la tâche suivante. On peut observer les différentes méthodes de "Thought-based" sur la figure 4.13.

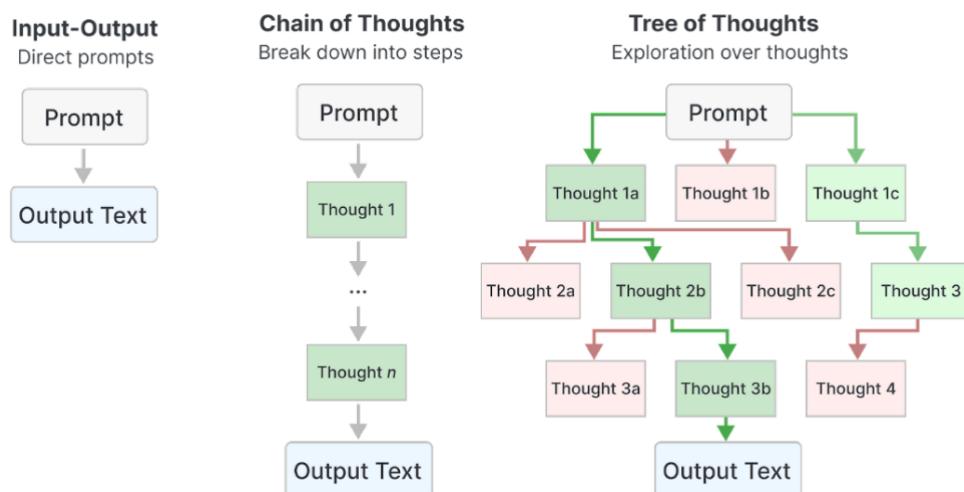


FIGURE 4.13 – Différentes méthodes dites "Thought-based". [29]

**Retrieval Augmented Generation** Pour certains problèmes complexes qui ont besoin de beaucoup d'informations et de connaissances, il est possible de construire un modèle basé sur des res-

sources externes. Ainsi, en donnant accès à des données externes, cela augmente la fiabilité et la cohérence des réponses. La méthode RAG permet de mettre en place cette idée (figure 4.14). [44]

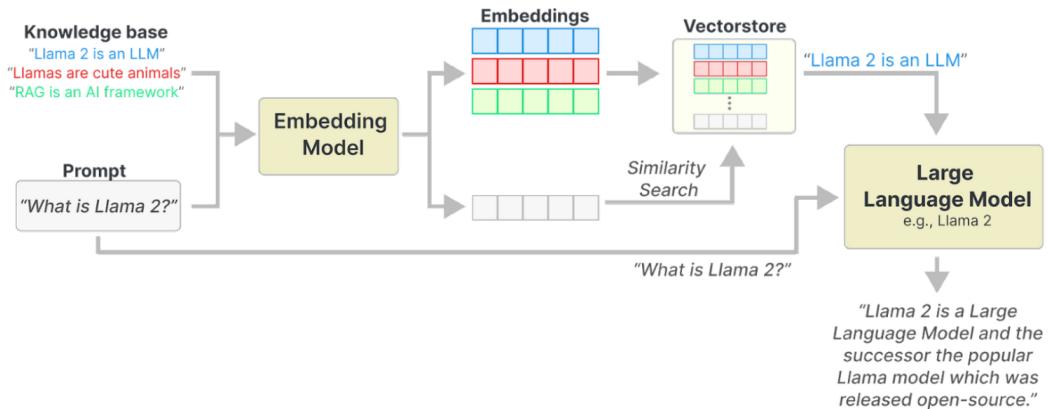


FIGURE 4.14 – Illustration de la méthode RAG. [29]

### 4.3.3 Compression

La compression de LLM est un ensemble de techniques qui permet d'obtenir un modèle plus compact et plus léger pour pouvoir être utilisé dans un environnement à faible ressource. Cela implique aussi une meilleure inference, et une latence plus faible.

La figure 4.15 représente l'état de l'art des méthodes de compression à jour :

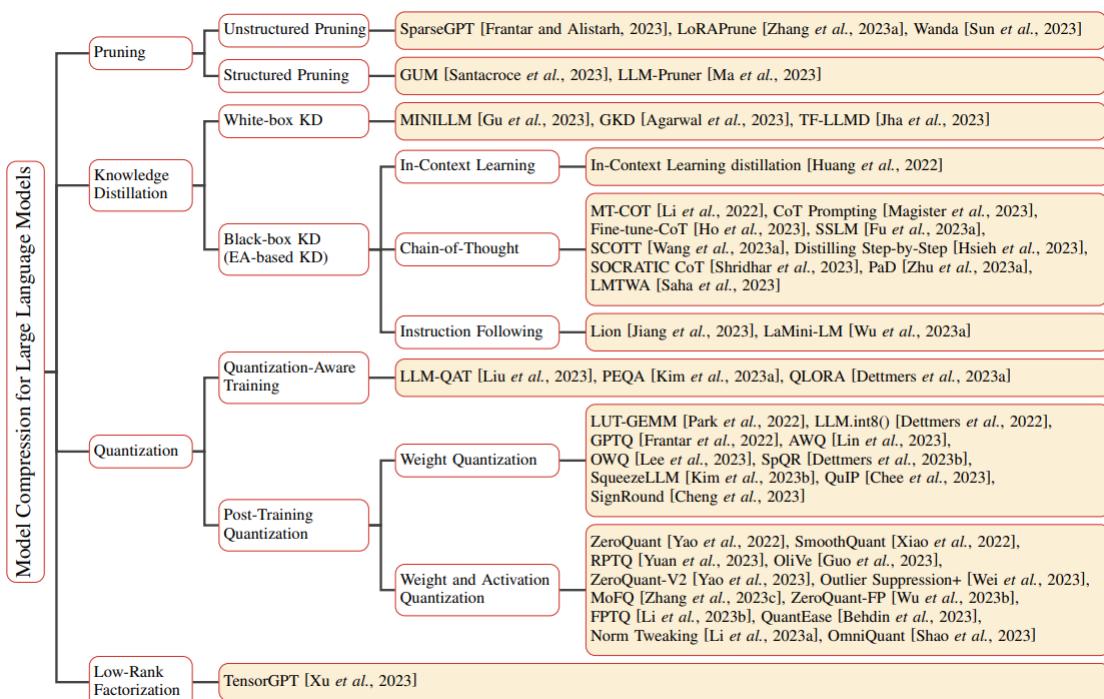


FIGURE 4.15 – Les catégories de compression [64]

La compression des modèles de langage est un domaine très récent et qui est à ce jour encore sous-traité. Vous pouvez suivre cette constante évolution grâce à un état de l'art exhaustif des méthodes

de compression des LLMs dans le répertoire GitHub *HuangOwen/Awesome-LLM-Compression* [10]. Vous y trouverez les articles scientifiques les plus récents ainsi que les codes existants.

Dans notre état de l'art, nous traiterons des méthodes de quantification ("quantization" en anglais) et des méthodes d'élagage ("pruning" en anglais). Nous ne traiterons pas des deux autres catégories qui sont présentées dans la figure ?? : "Knowledge Distillation" et "Low-Rank Factorization". Pour approfondir vos recherches, veuillez vous référer à l'article scientifique *A Survey on Model Compression for Large Language Models* [64].

#### 4.3.3.1 Quantification (alias quantization)

Les poids d'un réseau de neurones sont généralement représentés par des nombres flottants codés en 32 bits ou 16 bits. La quantification consiste à réduire le nombre de bits nécessaires pour stocker les poids. Cette méthode va permettre de réduire considérablement la taille des réseaux. Ces réseaux seront donc plus légers et plus rapides en calcul. Il existe des quantifications en 16, 8, 5, 4, 3 bits (et même 1 bits).

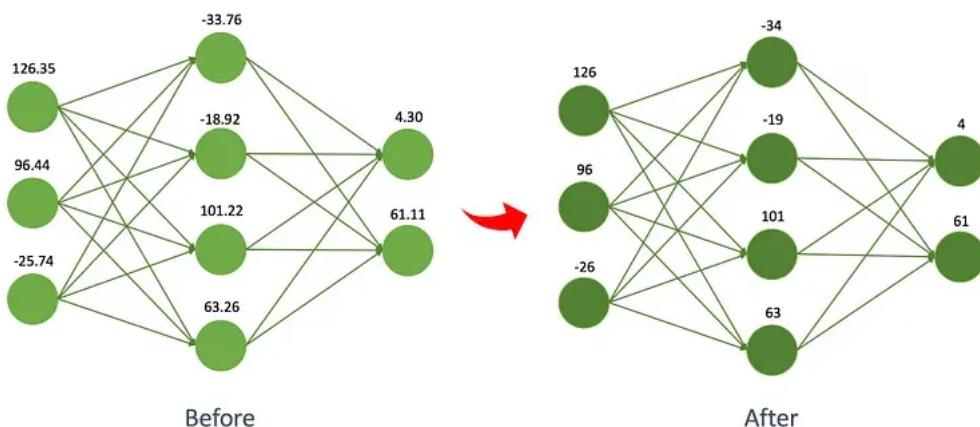


FIGURE 4.16 – Illustration de la quantification dans un réseau de neurones [34]

Information intéressante à savoir, certaines des méthodes quantifient uniquement les poids tandis que d'autres quantifient également l'activation.

Dans la littérature, la quantification est considérée comme la meilleure méthode de compression.

On peut séparer les méthodes en deux catégories : la quantification pendant l'entraînement (Quantization-Aware Training) que l'on dénommera **QAT** et la quantification après entraînement (Post-Training Quantization) que l'on dénommera **PTQ**.

Pour la suite, nous vous présenterons quelques méthodes de quantifications et vous trouverez également un tableau non exhaustive d'un ensemble de méthodes que vous pourriez utiliser pour vos besoins en pratique.

**QLoRA : une méthode QAT** QLoRA [56] est une méthode de quantification qui se décompose en deux étapes. Tout d'abord le modèle est quantifié en 4 bits pour pouvoir charger le modèle avec peu de ressources. Ensuite, l'entraînement LoRA est effectué en précision 32 bits. Le passage en haute précision permet d'entraîner efficacement les adaptateurs LoRA. Il est possible de varier les paramètres pour changer la précision initiale, ou pour choisir quels paramètres du modèle sont ciblés

parmi les poids et l'activation. En pratique, pour utiliser QLoRA, une façon de faire est d'utiliser une combinaison des bibliothèques bitsandbytes [2] et PEFT [1] pour pouvoir respectivement quantifier le modèle et l'entraîner avec la méthode LoRA. Pour vous initier à la pratique, vous trouverez un notebook illustrant la méthode QLoRA en utilisant ces deux précédentes bibliothèques. Vous le trouverez dans un cours d'introduction sur les LLMs *Fine-Tune Your Own Llama 2 Model in a Colab Notebook* [41] proposé par Maxime Labonne. Ce notebook fonctionne sur les environnements gratuits T4 GPU de Google Colab.

**llama.cpp : une méthode PTQ** Le projet llama.cpp [13] a conçu un format de modèle en C/C++ qui se nomme le GGUF. Voir la section 4.3.4.2 pour plus d'informations. Ce qui nous intéresse ici, ce sont les méthodes de quantification proposées par llama.cpp applicable sur les modèles au format GGUF. Pour plus d'informations sur le fonctionnement des quantifications de llama.cpp, voir [36]. Ces méthodes ont retenu notre attention par leur rapidité d'exécution et leur facilité de mise en place. Le code étant réalisé en C/C++, il s'adapte à l'environnement utilisé et semble être moins sujet aux bugs (notamment aux bugs de dépendance entre les librairies).

**Omniquant : une méthode à mis chemin** Dans l'article scientifique correspondant [59], les auteurs expliquent que les méthodes QAT permettent d'avoir de meilleures performances, tandis que les méthodes PTQ sont plus efficaces en ressources et en temps. Omnipquant [11] a donc vu le jour pour essayer d'allier les points forts des méthodes QAT et PTQ.

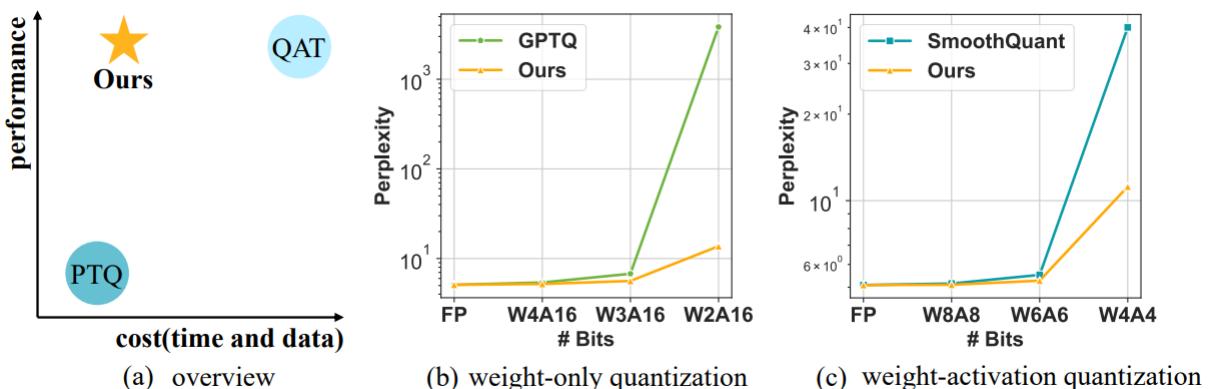


FIGURE 4.17 – Vu d'ensemble des résultats d'Omniquant montrant la capacité à obtenir les performances des méthodes QAT tout en ayant l'efficacité des méthodes PTQ. [59]

**En pratique** Vous trouverez dans le tableau suivant, une liste non exhaustive des méthodes de quantification les plus pertinentes et les plus simples d'utilisation que l'on a rencontrée durant nos recherches.

Méthode	Catégorie	Sources
bitsandbytes	PTQ	[2][8]
QLoRA	QAT	[56] [41]
GGUF	PTQ	[13]
AWQ	PTQ	[4][14]
AutoGPTQ	PTQ	[15][3] [25]
Omniquant	QAT/PTQ	[11]
Intel Neural Compressor	QAT,PTQ	[9]

**Performances** Pour donner un aperçu des performances possibles grâce à la quantification, voici un exemple dans la figure 4.18 évaluant la perplexité en fonction d'un modèle de base et de son homologue quantifié en différentes tailles. Attention, la perplexité n'est pas la seule métrique qu'il faut observer pour évaluer un modèle. Les résultats dans la figure 4.18 sont donc à prendre avec recul. Concernant les résultats, on peut voir que la quantification en 8 bits permet d'obtenir la même perplexité avec une taille de modèle divisé par deux. Les modèles quantifiés semblent avoir d'impressionnantes performances au vu de l'économie en ressources qui est réalisée.

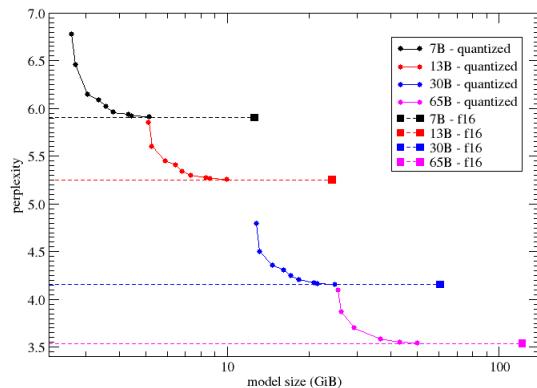


FIGURE 4.18 – Comparaison de la perplexité entre différentes quantification d'un même modèle [42]

#### 4.3.3.2 Elagage (alias pruning)

Dans un modèle, il y a plusieurs paramètres redondants qui n'ont pas, ou ont très peu, d'effet sur les performances. En les éliminant, on obtient un modèle beaucoup plus léger et peu coûteux en ressources de stockage, CPU et RAM. De plus, la baisse de performance est minimisée.

L'élagage (“Pruning” en anglais) d'un LLM est une technique permettant de réduire la taille et la complexité d'un modèle en éliminant des parties pas/peu nécessaires ou redondantes.

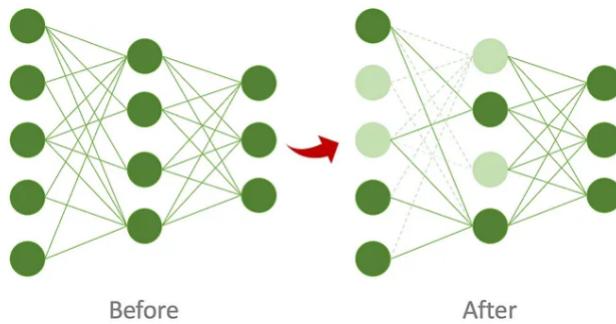


FIGURE 4.19 – Illustration de l'élagage dans un réseau de neurones [38]

Deux catégories d'élagages sont connues à ce jour : l'élagage structuré (Structured Pruning) , et l'élagage non structuré (Unstructured Pruning) . Ces méthodes se différencient dans les paramètres qu'elles ciblent et dans la structure finale obtenue du modèle.

L'élagage a souvent besoin de entraînement pour compenser une perte de précision.

**Élagage structuré** Dans l’élagage non structuré, on considère les paramètres individuellement et ne prend pas en compte la structure interne. Les paramètres ou les neurones sont ciblés grâce à un seuillage. Ils sont alors mis à zéro. Cela crée des irrégularités dans le modèle. Cette méthode d’élagage nécessite souvent un réentraînement du modèle pour regagner de la qualité.

**Élagage non structuré** L’élagage structuré consiste à éliminer des composantes entières de la structure du modèle, comme des neurones, des canaux, ou des couches. Ce type d’élagage permet de cibler des ensemble de paramètres, ce qui permet d’optimiser le modèle tout en conservant la structure globale.

**LLM Pruner : Une méthode d’élagage structuré** LLM pruner [27] est une méthode d’élagage structuré qui permet un élagage de toute taille. Elle a été conçue pour les modèles multi-tâches dans le but de conserver cette dernière caractéristique.

**En pratique** Vous trouverez dans le tableau suivant, une liste non exhaustive des méthodes d’élagage que l’on a rencontrées durant nos recherches.

Méthode	Catégorie	Source
LLM Pruner	Structuré	[27]
SparseGPT	Non structuré	[24]
LoRAPrune	Non structuré	[46]
LLM Surgeon	Structuré & Non structuré	[58]
Wanda	Structuré & Non structuré	[45]

#### 4.3.4 Le format et l’environnement d’exécution

L’environnement le plus populaire pour le développement des LLMs est l’environnement Python à l’aide de certaines librairies comme HuggingFace ou Tensorflow. HuggingFace est une start-up qui a tout fait pour faciliter l’accès à l’IA. On retrouve sur leur site web un large éventail de LLMs pré-entraînés en libre accès. C’est une plate-forme de partage qui propose également une librairie python *transformers* qui facilite la création, le développement, l’entraînement et l’exécution des LLMs au format PyTorch.

Après la phase d’entraînement, lorsque l’on obtient le modèle qui répond à nos besoins, l’étape suivante est alors la mise en production. Cependant, les LLMs au format PyTorch ne sont pas les plus efficaces à l’exécution. D’autres formats ont alors vu le jour pour optimiser l’inférence. Il existe des formats conçus pour une optimisation sur CPU ou GPU, pour certaines architectures particulières, ou même en fonction de certaines caractéristiques des modèles (pour les modèles très éparques par exemple).

Nous allons donc voir différents formats et environnements d’exécution que l’on a pu rencontrer durant nos recherches.

- ONNX
- GGUF (ex-GGML)
- DeepSparse

#### 4.3.4.1 ONNX

Le format ONNX a été conçu pour maximiser les performances en inférence par l'optimisation des calculs au niveau de l'*hardware*. C'est un format standard pour représenter les modèles de Machine Learning.

En pratique, il est possible de convertir au format ONNX les modèles basés sur la bibliothèque *transformers* en utilisant l'API Optimum [12]. Ainsi tout modèle pouvant être chargé à partir de la bibliothèque *transformers* peut être convertit au format ONNX. Le format est également accessible pour les autres formats de modèle à l'aide de bibliothèques variés. Une recherche sur un moteur de recherche devrait vous guider en fonction de vos besoins.

L'API Optimum permet également de gérer l'inférence (plus d'informations sur HuggingFace [5]), mais propose aussi un outil pour réaliser une quantification des modèles ONNX et d'autres optimisations.

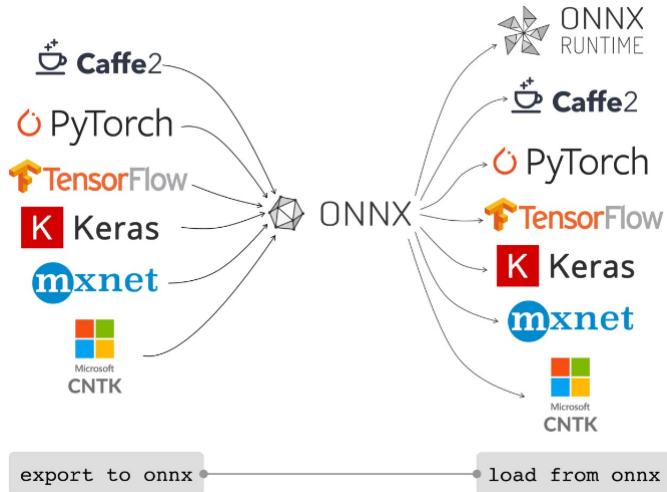


FIGURE 4.20 – Illustration de la portabilité du format ONNX. [30]

#### 4.3.4.2 GGUF

GGUF, qui est un nouveau format qui cherche à résoudre les problèmes de son petit frère le GGML, est un format qui convertit les modèles en pur langage C/C++ pour optimiser l'inférence.

La motivation du projet a été, à la base, de permettre l'inférence de LLM LLaMa sur Macbook. Aujourd'hui, GGUF est un format populaire pour sa facilité d'utilisation, son efficacité en inférence, ainsi que ses méthodes de quantification efficaces. Il supporte aujourd'hui beaucoup plus de modèles qu'initialement, comme LLaMa 2, Bloom, GPT-2 ou encore Mistral7B par exemple.

On peut de plus quantifier un modèle GGUF facilement en suivant les instructions sur le README du répertoire GitHub du projet *llama.cpp*. [13]

Un autre point positif du format GGUF est l'économie d'espace disque une fois quantifié. En effet, on peut voir dans la figure 4.21 les tailles des modèles une fois quantifiés en utilisant le paramètre "Q4\_0" du code source du projet. Les modèles ne font plus qu'un tiers de leur taille initiale.

Pour plus d'information, veuillez vous redirigez vers la page GitHub du projet "llama.cpp" [13].

Model	Original size	Quantized size (Q4_0)
7B	13 GB	3.9 GB
13B	24 GB	7.8 GB
30B	60 GB	19.5 GB
65B	120 GB	38.5 GB

FIGURE 4.21 – Illustration des ressources nécessaires pour un LLM au format GGUF quantifié en 4bits. ([source](#))

#### 4.3.4.3 DeepSparse

DeepSparse est un environnement d'exécution conçu pour optimiser l'inférence sur CPU de modèles très éparses. [7] Pour obtenir ces modèles éparses, DeepSparse nous invite à utiliser SparseML. [6]

En combinant DeepSparse et SparseML, on obtient un outil tout en un qui permet de compresser un modèle à l'aide d'une méthode d'élagage, de l'extraire en ONNX, puis d'optimiser l'inférence en profitant du caractère épars du modèle.

DeepSparse a également l'avantage de proposer une banque de modèles pré-élagué (BERT, YOLOv5, ResNet-50, etc). Ces modèles peuvent être récupérés pour être ré-entraînés.

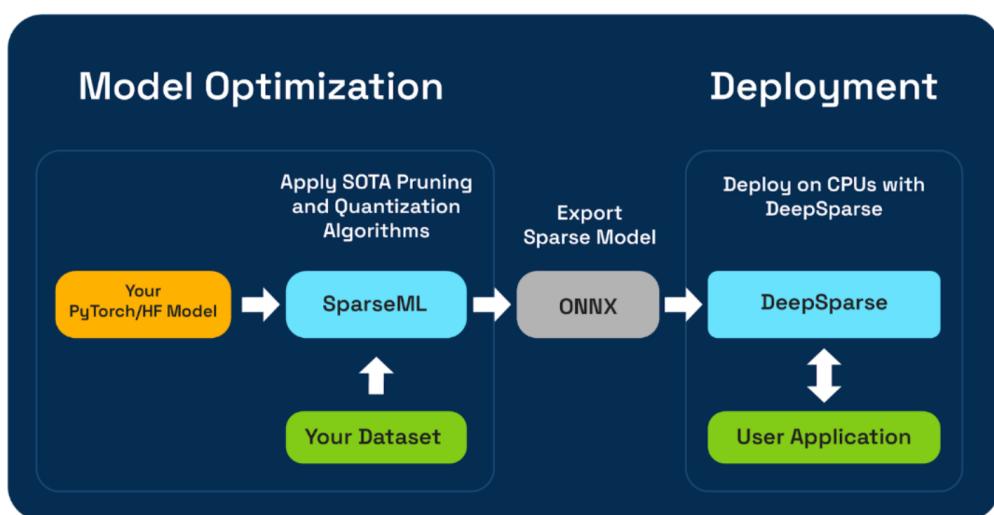


FIGURE 4.22 – Flux de travail disponible via l'outil DeepSparse. [7]

DeepSparse propose trois différentes APIs :

- Engine : Une API de bas niveau. Elle permet de compiler un modèle ONNX. On y passe des "tensors" en entrée. Et elle nous renvoie des données bruts en sortie.
- Pipeline : Ajoute une couche de pré- et post- processing à l'API Engine. On lui donne un vecteur en entrée, et elle nous renvoie la prédiction.
- Server : Utilise une API REST. On envoie le vecteur via http, et la prédiction nous est renvoyée.

Pour plus d'information sur DeepSparse, voir la page GitHub du projet. [7]

### 4.3.5 Conclusion sur l'amélioration des modèles

Nous avons pu voir différentes manières d'optimiser un LLM. Ce n'est bien évidemment pas une liste exhaustive des méthodes d'amélioration, mais nous avons traités en surface les trois grands axes importants : le *prompt engineering*, les environnements d'exécution, les méthodes d'élagage et de quantification.

Comme dit précédemment, la compression des modèles de langage est un domaine très récent et qui est à ce jour encore sous-traité. Vous pouvez approfondir vos connaissances grâce au dépôts GitHub suivant : *HuangOwen/Awesome-LLM-Compression* [10].

## 4.4 Méthodes de comparaisons des LLMs

### 4.4.1 Introduction à la comparaison des LLMs

Depuis quelques années et d'autant plus depuis la sortie de ChatGPT en novembre 2022, de nombreux LLMs ont émergé et de nombreux autres émergent fréquemment. Avec l'augmentation du nombre de modèles disponibles, cela devient complexe de savoir quel modèle est approprié à son cas d'utilisation, quel modèle est "le meilleur" pour son cas d'utilisation, pour une raison assez évidente : les méthodes de comparaison ne sont actuellement pas assez standardisées et fiables. Le domaine est en effet en pleine croissance et beaucoup d'entreprises investissent pour créer des LLMs. Les découvertes, améliorations, optimisations et résultats sont donc fréquents, mais cela pose plusieurs problématiques pour réussir à comparer et évaluer ces LLMs.

Premièrement, déterminer quels critères comparer pour les LLMs s'avère complexe, car chaque critère peut avoir des impacts tant positifs que négatifs selon le contexte d'utilisation, et l'effet de certains d'entre eux peut être difficile à mesurer. En effet, les résultats de la génération de texte par ces modèles ne sont pas toujours clairs ni binaires.

Deuxièmement, la question de l'obsolescence des méthodes de comparaison se pose. Dans un domaine qui évolue rapidement, comment maintenir le rythme avec les progrès technologiques fréquents et significatifs ?

Enfin, un problème majeur réside dans le manque d'investissement dans l'amélioration de ces méthodes. Les entreprises étant les principaux acteurs des avancées technologiques dans ce domaine, elles tendent à se concentrer sur les aspects les plus rentables, notamment le développement de leur propre LLM, plutôt que sur l'amélioration des méthodes de comparaison.

Pourtant, il est essentiel de disposer de méthodes de comparaison fiables et standardisées pour les LLMs, car cela bénéficierait à tous les acteurs impliqués. Premièrement, cela garantirait la qualité des LLMs avant leur mise sur le marché, en termes de sécurité, exactitude et fiabilité.

De plus, des méthodes de comparaison plus approfondies pourraient prévenir des pratiques où certaines entreprises optimisent leurs LLMs uniquement pour exceller dans les benchmarks qui évaluent actuellement les modèles, sans que cela se traduise par une performance réelle équivalente. Cette pratique, déjà observée, souligne la nécessité d'avoir des méthodes plus fiables.

Enfin, l'établissement de méthodes de comparaison standardisées offrirait une plus grande transparence et assurance aux utilisateurs, qu'ils soient des entreprises ou des particuliers, quant à la performance des LLMs qu'ils choisissent d'utiliser.

Le problème actuel est qu'il existe effectivement des benchmarks pour comparer les LLMs, mais il en existe beaucoup, qui utilisent des critères différents, des méthodes différentes, et ce n'est pas du tout clair de savoir lesquels existent et lesquels regarder pour prendre une décision, ni de savoir s'ils sont réellement fiables.

#### 4.4.2 Organisation actuelle de l'évaluation des LLMs

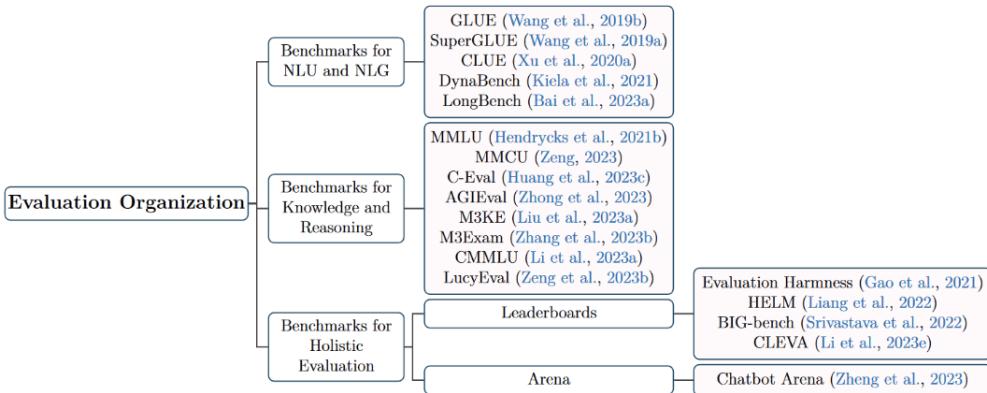


Figure 6: Overview of LLM evaluation organization.

FIGURE 4.23 – Les différentes types de benchmarks [63]

Actuellement, l'évaluation de la performance se traduit par l'utilisation de benchmarks variés, chacun ciblant des aspects différents des capacités des LLMs. La figure 4.23 fournit un aperçu structuré des catégories d'organisation d'évaluation pour les LLMs.

Il y a d'abord les benchmarks pour la compréhension du langage naturel (NLU) et la génération de langage naturel (NLG) : Cette catégorie regroupe les outils d'évaluation conçus pour tester la capacité des LLMs à comprendre et à produire du texte de manière cohérente et pertinente. Des benchmarks tels que GLUE (General Language Understanding Evaluation) et SuperGLUE sont des suites de tests qui mesurent la compréhension du langage à travers une variété de tâches, telles que l'analyse de sentiments ou la reconnaissance d'entités nommées. D'autres, comme CLUE (Chinese Language Understanding Evaluation), DynaBench et LongBench, fournissent des plateformes pour tester les LLMs dans des langues spécifiques ou sur des tâches à plus long terme.

Ensuite, il existe des benchmarks pour la connaissance et le raisonnement : Cette catégorie comprend des benchmarks qui évaluent la capacité des LLMs à stocker des connaissances et à effectuer des raisonnements complexes. Des outils comme MMLU (Massive Multitask Language Understanding) et MMCU (Multitask Multilingual Common Understanding) proposent des défis multilingues et multidisciplinaires. D'autres, tels que C-Eval, AGIEval et M3KE, mettent l'accent sur l'évaluation de l'intelligence artificielle générale et la compréhension de la connaissance mondiale.

Enfin, les benchmarks pour l'évaluation holistique : Cette section regroupe des benchmarks qui visent à fournir une évaluation globale des LLMs, en tenant compte de tous les aspects de leur performance. Ces outils cherchent à fournir une vue d'ensemble, plutôt que de se concentrer sur des tâches spécifiques. Les Leaderboards et les arènes de compétition sont des exemples de plates-formes où différents LLMs peuvent être comparés directement en fonction de leurs performances globales.

Ces benchmarks sont essentiels pour évaluer de manière les capacités des LLMs mais aucun ne prend en compte la sécurité comme facteur et aucun ne se suffit à lui-même car ils ne répondent qu'à certains critères précis.

#### 4.4.3 Présentations de quelques métriques

Lorsqu'il s'agit de comparer un modèle de langage à grande échelle (LLM), plusieurs approches peuvent être envisagées. D'un côté, il est possible d'opter pour une comparaison quantitative, en s'appuyant sur des métriques de performance appliquées à des tâches spécifiques comme la classification, la génération de texte, la traduction, etc. Ces métriques numériques incluent la précision, le rappel, le score F1, la perplexité, le temps d'inférence, entre autres. De l'autre côté, une comparaison qualitative peut être réalisée grâce à une évaluation humaine portant sur la qualité des réponses générées par le modèle, évaluant des critères tels que la pertinence, la cohérence et la fluidité du texte. La comparaison des coûts associés à l'entraînement et à l'utilisation du modèle représente également un critère important.

Certains LLMs, spécifiquement entraînés pour des domaines précis, peuvent être évalués à travers une comparaison adaptée au domaine en question. Par exemple, dans le domaine de la génération de code, HumanEval propose des tests basés sur des problèmes informatiques spécifiques à résoudre.

D'autres approches de comparaison incluent l'examen des questions éthiques liées à l'utilisation du modèle ou la présence de biais dans les données d'entraînement. Ces aspects cruciaux offrent une vision plus complète de la performance et de l'applicabilité d'un LLM dans divers contextes.

Cette partie consiste donc à présenter quelques métriques intéressantes et fréquemment utilisées lors de l'évaluation des LLMs. [57] [55] [20]

##### 4.4.3.1 Taille des LLMs

La taille d'un modèle exprimé en termes de nombre de paramètres est une métrique qui sert à quantifier la complexité du modèle. Les paramètres sont les éléments que le modèle apprend à partir des données d'entraînement pour effectuer des tâches spécifiques, comme la classification, la prédiction, la génération de texte...

Trois points importants à noter sur la taille des LLMs :

- Capacité d'apprentissage : un modèle avec un plus grand nombre de paramètres a généralement une plus grande capacité d'apprentissage, ce qui signifie qu'il peut théoriquement modéliser des relations plus complexes dans les données. Cela peut être bénéfique pour des tâches complexes ou lorsque les données d'entraînement sont très variées et riches.
- Risque de surajustement (overfitting) : alors qu'un modèle de grande taille peut mieux apprendre des données complexes, il y a aussi un risque accru de surajustement, c'est-à-dire qu'il peut apprendre des détails et du bruit spécifiques aux données d'entraînement qui ne se généralisent pas bien aux données inédites.
- Ressources nécessaires : les modèles plus grands nécessitent plus de mémoire et de puissance de calcul, tant pour l'entraînement que pour l'inférence. Cela peut poser des contraintes en termes de coûts matériels et de temps de calcul, et peut limiter l'utilisation de tels modèles sur des dispositifs avec des ressources limitées.

Il a été clairement observé une corrélation entre la taille du modèle et les résultats. Par exemple pour GPT-3, on observe une augmentation des résultats au fur et à mesure que le nombre de paramètres augmente, même lorsque le modèle a déjà beaucoup de paramètres.

#### 4.4.3.2 Rappel

La métrique de rappel (ou recall en anglais) est un indicateur de la capacité d'un modèle à identifier correctement toutes les instances positives d'une classe spécifique. Le rappel est défini comme le nombre de vrais positifs divisé par la somme des vrais positifs et des faux négatifs.

Pour expliquer cela plus clairement, considérons les termes suivants :

Vrai Positif (VP) : le nombre de cas où le modèle a correctement prédit la classe positive.

Faux Négatif (FN) : le nombre de cas où le modèle a incorrectement prédit la classe négative alors que la véritable classe était positive.

La formule du rappel est donc la suivante :

$$\text{recall} = \frac{\text{Vrai Positif}}{\text{Vrai Positif} + \text{Faux Négatif}} \quad (4.1)$$

Le rappel est particulièrement important dans des situations où il est crucial de ne pas rater les cas positifs. Par exemple, dans le diagnostic médical, un rappel élevé signifie que le modèle a correctement identifié la plupart des patients malades, minimisant le risque de ne pas traiter une personne atteinte d'une maladie. Un autre exemple pourrait être la détection de fraude, où il est préférable de détecter autant de transactions frauduleuses que possible, même au risque de générer des faux positifs.

Cependant, il est important de noter que se concentrer uniquement sur le rappel peut entraîner une augmentation du nombre de faux positifs, car le modèle pourrait être biaisé en faveur de la prédiction de la classe positive. Pour cette raison, le rappel est souvent utilisé en conjonction avec la précision (le nombre de vrais positifs divisé par le nombre total de cas positifs prédits) et la mesure F1, qui est la moyenne harmonique de la précision et du rappel. La mesure F1 fournit un équilibre entre la précision et le rappel, offrant une vue d'ensemble de la performance du modèle. [37]

#### 4.4.3.3 Précision

La métrique de précision est un autre indicateur utilisé pour évaluer la performance des modèles en IA. Elle mesure la proportion de prédictions positives qui sont réellement correctes. La précision est calculée en divisant le nombre de vrais positifs par la somme des vrais positifs et des faux positifs.

La formule de la précision est donc la suivante :

$$\text{precision} = \frac{\text{Vrai Positif}}{\text{Vrai Positif} + \text{Faux Positif}} \quad (4.2)$$

La précision est particulièrement importante dans les contextes où les coûts des fausses alarmes sont élevés. Par exemple, dans le filtrage des courriels électroniques, une haute précision signifie

que peu de courriers légitimes sont incorrectement marqués comme spam, ce qui est préférable pour l'expérience utilisateur.

Cependant, tout comme pour le rappel, se concentrer uniquement sur la précision peut être trompeur. Un modèle pourrait simplement prédire très peu de cas positifs, et par conséquent, ceux qu'il prédit correctement pourraient donner une précision élevée, même si de nombreux vrais positifs sont manqués (faux négatifs). Pour cette raison, la précision est souvent évaluée en conjonction avec le rappel pour obtenir une image complète de la performance du modèle. [37]

#### 4.4.3.4 F1 score

Le F1 Score est une métrique qui combine la précision et le rappel pour évaluer la performance des modèles en IA. Il est particulièrement utile lorsque vous souhaitez trouver un équilibre entre la précision et le rappel, surtout si l'on suspecte un déséquilibre entre les coûts des faux positifs et des faux négatifs.

Le F1 Score est le double du produit de la précision et du rappel divisé par la somme de la précision et du rappel. Sa formule est donc :

$$\text{F1 Score} = 2 \times \frac{\text{precision} \times \text{rappel}}{\text{precision} + \text{rappel}} \quad (4.3)$$

Cette formule donne une moyenne harmonique de la précision et du rappel, et le score F1 atteindra sa meilleure valeur à 1 (performance parfaite) et sa pire à 0. Un F1 Score élevé indique donc que le modèle a une bonne précision ainsi qu'un bon rappel.

Le F1 Score est particulièrement important dans des situations où les classes ne sont pas bien équilibrées, c'est-à-dire quand il y a un grand déséquilibre entre le nombre d'exemples pour chaque classe. Par exemple, dans une situation où les cas positifs sont rares (comme dans la détection de fraude), un modèle qui prédit simplement que tous les cas sont négatifs pourrait avoir une précision et un rappel faibles, mais il serait peu utile dans la pratique. Le F1 Score aiderait dans ce cas à identifier les modèles qui gèrent correctement les cas positifs rares tout en évitant un nombre excessif de faux positifs. [33] [37]

#### 4.4.3.5 Perplexité

La perplexité (PPL) est une métrique utilisée pour évaluer la performance des modèles de langage probabilistes. Elle mesure à quel point un modèle de langage est surpris lorsqu'il reçoit de nouvelles données, autrement dit, sa capacité à prédire un échantillon. Une perplexité faible indique que le modèle prédit les échantillons avec une haute probabilité, ce qui signifie qu'il a bien appris la distribution des mots du langage qu'il modélise.

La formule de la perplexité est la suivante :

$$\text{PPL}(X) = \exp \left( -\frac{1}{t} \sum_i^t \log p_{\theta}(x_i | x_{<i}) \right) \quad (4.4)$$

où :

$X$  représente une séquence de mots ou de jetons.

$t$  est la longueur de la séquence.

$p(x < i)$  est la probabilité que le modèle attribue au jeton  $x_i$  sachant les jetons précédents  $x < i$

$\exp$  est la fonction exponentielle.

$\log$  est le logarithme naturel.

La perplexité est calculée en prenant l'exponentielle de l'opposé de la moyenne du logarithme des probabilités attribuées par le modèle à chaque jeton de la séquence, donnée par le modèle. Cela revient à prendre l'exponentielle de l'entropie croisée entre la distribution de probabilité prédite par le modèle et la véritable distribution des données.

La perplexité est utilisée pour comparer différents modèles de langage : un modèle avec une perplexité plus basse sur un ensemble de données de test est considéré comme ayant une meilleure performance, car il prédit mieux les mots inconnus. Elle est particulièrement utile pour des tâches comme la génération de texte ou la traduction automatique, où il est important que le modèle puisse prédire avec précision la probabilité des séquences de mots. [50] [26]

#### 4.4.3.6 Pass@k

La métrique pass@k est une mesure d'évaluation spécifiquement conçue pour les modèles de génération de code, le modèle est chargé de produire du code source qui résout un problème donné. L'idée derrière cette métrique est de tester la capacité du modèle à générer des échantillons de code qui non seulement semblent corrects syntaxiquement, mais qui sont également fonctionnellement valides, c'est-à-dire qu'ils réussissent des tests unitaires spécifiés pour le problème en question.

Le fonctionnement de la métrique pass@k est le suivant :

Le modèle est confronté à un problème de programmation pour lequel il doit générer du code. Pour une valeur de  $k$  donnée, le modèle produit  $k$  échantillons différents de code qui tentent de résoudre le problème. Les  $k$  échantillons de code sont ensuite soumis à une série de tests unitaires qui vérifient leur exactitude. Si au moins un des  $k$  échantillons réussit tous les tests unitaires, le problème est considéré comme "résolu" par le modèle selon la métrique pass@k. En pratique, les valeurs couramment utilisées pour  $k$  sont pass@1 et pass@10. Pass@1 indique si le premier échantillon de code généré passe tous les tests unitaires, tandis que pass@10 offre une vision plus souple de la performance du modèle, car elle permet au modèle jusqu'à 10 tentatives pour produire un code valide.

Cette métrique est particulièrement utile car elle reflète la capacité pratique d'un modèle à générer du code utilisable dans des situations réelles.

La figure 4.24 présente un tableau comparatif de plusieurs modèles de génération de code évalués en utilisant la métrique pass@k. Cette image illustre comment les modèles sont comparés en termes de leur capacité à générer du code qui réussit les tests unitaires pour deux ensembles de problèmes de programmation différents, HumanEval et MBPP.

Pour chaque modèle, le tableau montre la taille du modèle (en milliards de paramètres, noté B) ainsi que les performances obtenues sur les métriques pass@1, pass@10, et pass@100. Ces performances sont exprimées en pourcentage, représentant la proportion de problèmes résolus parmi les échantillons de code générés.

Par exemple, regardons le modèle "GPT-4" sur l'ensemble de problèmes HumanEval. Il obtient une performance de pass@1 de 67.0%, ce qui signifie que pour 67.0% des problèmes, le premier échantillon

Model	Size	HumanEval			MBPP		
		pass@1	pass@10	pass@100	pass@1	pass@10	pass@100
code-cushman-001	12B	33.5%	-	-	45.9%	-	-
GPT-3.5 (ChatGPT)	-	48.1%	-	-	52.2%	-	-
GPT-4	-	<b>67.0%</b>	-	-	-	-	-
PaLM	540B	26.2%	-	-	36.8%	-	-
PaLM-Coder	540B	35.9%	-	88.4%	47.0%	-	-
PaLM 2-S	-	37.6%	-	88.4%	50.0%	-	-
StarCoder Base	15.5B	30.4%	-	-	49.0%	-	-
StarCoder Python	15.5B	33.6%	-	-	52.7%	-	-
StarCoder Prompted	15.5B	40.8%	-	-	49.5%	-	-
LLAMA 2	7B	12.2%	25.2%	44.4%	20.8%	41.8%	65.5%
	13B	20.1%	34.8%	61.2%	27.6%	48.1%	69.5%
	34B	22.6%	47.0%	79.5%	33.8%	56.9%	77.6%
	70B	30.5%	59.4%	87.0%	45.4%	66.2%	83.1%
CODE LLAMA	7B	33.5%	59.6%	85.9%	41.4%	66.7%	82.5%
	13B	36.0%	69.4%	89.8%	47.0%	71.7%	87.1%
	34B	48.8%	76.8%	93.0%	55.0%	76.2%	86.6%
CODE LLAMA - INSTRUCT	7B	34.8%	64.3%	88.1%	44.4%	65.4%	76.8%
	13B	42.7%	71.6%	91.6%	49.4%	71.2%	84.1%
	34B	41.5%	77.2%	93.5%	57.0%	74.6%	85.4%
UNNATURAL CODE LLAMA	34B	<b>62.2%</b>	<b>85.2%</b>	<b>95.4%</b>	<b>61.2%</b>	<b>76.6%</b>	86.7%
CODE LLAMA - PYTHON	7B	38.4%	70.3%	90.6%	47.6%	70.3%	84.8%
	13B	43.3%	77.4%	94.1%	49.0%	74.0%	87.6%
	34B	53.7%	82.8%	94.7%	56.2%	76.4%	<b>88.2%</b>

FIGURE 4.24 – comparatif des modèles générateur de code avec pass@k

de code généré par le modèle a réussi tous les tests unitaires. Pour pass@10, la performance n'est pas spécifiée dans le tableau.

Le tableau fournit également des données pour d'autres modèles comme "PaLM-Coder", "StarCoder", "LLama 2", "Code LLama", et des variantes de "Code LLama" avec des tailles de modèle allant de 7B à 70B. Chaque modèle présente des performances variables en pass@1, pass@10 et pass@100, ce qui donne une idée de leur efficacité à résoudre les problèmes de programmation à différents niveaux de tentatives.

Cet exemple illustre clairement comment la métrique pass@k est utilisée pour évaluer la capacité des modèles de génération de code à produire du code fonctionnel et valide. Il permet aux utilisateurs de comparer l'efficacité des modèles en fonction de leur taille et de leur architecture.[49]

#### 4.4.3.7 Zero-shot learning

Le zero-shot learning est une métrique qui évalue la capacité d'un modèle d'apprentissage automatique à généraliser à de nouvelles tâches ou à des catégories d'objets qu'il n'a jamais vues auparavant durant la phase d'entraînement. Plutôt qu'une métrique traditionnelle comme l'exactitude ou le F1 score, le zero-shot learning est une approche ou un paradigme d'apprentissage qui met au défi les modèles de faire des prédictions sur des données pour lesquelles ils n'ont pas été explicitement préparés.

Dans le contexte des modèles de langage ou de vision par ordinateur, un modèle capable de zero-shot learning peut, par exemple, reconnaître des images d'animaux qu'il n'a jamais rencontrés dans son jeu de données d'entraînement ou comprendre des requêtes textuelles dans des domaines de connaissances sur lesquels il n'a pas été directement formé.

Le zero-shot learning repose généralement sur l'hypothèse que le modèle a appris une représentation riche et générale des données durant l'entraînement qui peut être appliquée à de nouveaux contextes. Cela est souvent rendu possible grâce à des techniques comme l'apprentissage par transfert, où des connaissances acquises sur une tâche sont transférées à une autre tâche, ou l'incorporation de connaissances externes, comme une base de données conceptuelle ou un modèle de langage pré-entraîné qui comprend une large variété de concepts.

Pour évaluer un modèle sur sa capacité à effectuer du zero-shot learning, les chercheurs présentent au modèle des tâches qu'il n'a pas vues auparavant et mesurent sa performance, souvent en utilisant des métriques standardisées comme la précision, mais dans un contexte où les exemples de ces tâches spécifiques sont absents de l'ensemble de données d'entraînement. [60]

#### 4.4.3.8 CodeBERTScore

CodeBERTScore est une métrique utilisée pour évaluer la qualité du code généré par un modèle d'IA en le comparant à un code de référence. La procédure d'évaluation se déroule comme suit :

On commence avec deux morceaux de code : un code de référence, qui est considéré comme une solution correcte au problème posé, et un code généré par le modèle d'IA que l'on souhaite évaluer.

Chaque code est encodé séparément en utilisant le modèle d'IA CodeBERT, un modèle basé sur BERT spécialement entraîné pour comprendre le langage de programmation.

Après l'encodage, on calcule la similarité cosinus paire à paire entre les tokens significatifs (tokens non ponctuation) des deux codes. La similarité cosinus mesure la similarité entre deux vecteurs dans un espace vectoriel, ce qui, dans ce contexte, reflète la similarité sémantique entre les tokens des deux morceaux de code.

À partir de la matrice de similarité obtenue, trois scores sont calculés : la Précision, le Rappel et le F-Score. La Précision mesure la proportion de tokens dans le code généré qui correspondent correctement à un token dans le code de référence, tandis que le Rappel mesure la proportion de tokens dans le code de référence qui ont été correctement repris dans le code généré. Le F-Score est la moyenne harmonique entre la Précision et le Rappel, offrant un équilibre entre ces deux mesures. La figure 4.25 illustre le fonctionnement de la métrique.

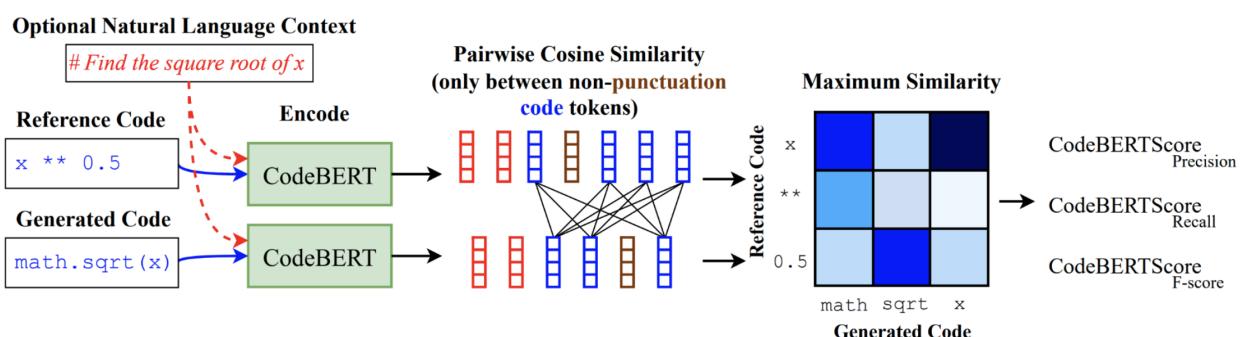


FIGURE 4.25 – fonctionnement de la métrique codeBERTScore

Cependant, il est important de noter que, en date de janvier 2024, cette métrique ne fonctionne que pour l'analyse syntaxique sur les langages que CodeBERT ne connaît pas. CodeBERT a été entraîné sur les langages de programmation les plus courants, tels que Java, JavaScript, Python, C et C++. C'est donc sur ces langages que l'analyse sémantique fonctionnera. [47] [48]

La figure 4.26 compare deux candidats de code - l'un équivalent et l'autre non équivalent au code de référence - et illustre comment le CodeBERTScore peut préférer un candidat équivalent en termes de fonctionnalité par rapport aux autres métriques qui pourraient ne pas capturer la sémantique du code de manière aussi efficace :

<div style="border: 1px solid black; padding: 5px; width: 100%;"> <p><b>Reference:</b></p> <pre>int f(Object target) {     int i = 0;     for (Object elem: this.elements) {         if (elem.equals(target)) {             return i;         }         i++;     }     return -1; }</pre> </div>	<div style="border: 1px solid black; padding: 5px; width: 100%;"> <p><b>Non-equivalent candidate:</b></p> <pre>boolean f(Object target) {     for (Object elem: this.elements) {         if (elem.equals(target)) {             return true;         }     }      return false; }</pre> </div>	<div style="border: 1px solid black; padding: 5px; width: 100%;"> <p><b>Equivalent candidate:</b></p> <pre>int f(Object target) {     for (int i=0; i&lt;this.elements.size(); i++) {         Object elem = this.elements.get(i);         if (elem.equals(target)) {             return i;         }     }     return -1; }</pre> </div>
<p>(a) The ground truth reference – find <i>the index</i> of target in <code>this.elements</code>.</p>	<p>(b) Preferred by BLEU &amp; CrystalBLEU – find <i>whether or not</i> target is in <code>this.elements</code>.</p>	<p>(c) Preferred by CodeBERTScore – find <i>the index</i> of target in <code>this.elements</code>.</p>

FIGURE 4.26 – Illustration de la métrique codeBERTScore

#### 4.4.4 Conclusion méthodes de comparaisons

Les méthodes de comparaison jouent un rôle crucial dans l'évaluation des modèles d'intelligence artificielle, y compris les LLMs. La diversité des métriques met en évidence l'importance d'une évaluation multifacettes pour capturer les différentes dimensions de performance des modèles. Chacune de ces métriques apporte des informations complémentaires.

Il est à noter que toutes les métriques disponibles pour l'évaluation des modèles d'intelligence artificielle ne sont pas exhaustivement traitées dans l'état de l'art présenté. Cependant, les métriques les plus connues et les plus largement utilisées ont été abordées.

La partie de recherche consacrée à l'étude de ces métriques est essentielle pour la phase suivante du projet, qui consiste à créer notre propre benchmark pour le cas d'application spécifique que nous envisageons. En comprenant les avantages et les limites des métriques existantes, nous pouvons élaborer une méthode d'évaluation qui soit adaptée aux besoins précis de notre application.

# 5 Cas d'application

## 5.1 Introduction

### 5.1.1 Objectifs et contexte du cas d'application

Le cas d'application vise à mettre en pratique l'ensemble des connaissances et des méthodologies explorées durant l'état de l'art. L'objectif principal est d'appliquer concrètement certaines de ces méthodes à un cas réel, afin d'évaluer leur efficacité et d'en déduire des informations pertinentes sur leur performance.

Dans le cadre spécifique de notre projet, Sopra Steria a exprimé le souhait de se concentrer sur la génération de requêtes Cypher. Ce choix repose sur le constat que les LLMs sont actuellement peu ou pas entraînés spécifiquement sur ce langage de requête. Cette situation présente une opportunité de mesurer et, potentiellement, d'améliorer la capacité des LLMs à générer des requêtes Cypher, en raison de la marge de progression importante qui est envisageable dans ce domaine. Cypher est un langage de requête de graphe principalement utilisé avec la base de données Neo4j, un système de gestion de base de données de graphes. Voici deux exemples très basique de requêtes Cypher pour illustrer ces propos :

```
1 // Retourne la personne s'appelant Julie
2 MATCH (a:Person {name : "Julie"}) Return a;
3
4 // Retourne les personnes ayant un lien avec Pierre
5 MATCH (a:Person) -[:KNOWS]-> (b:Person{name: "Pierre"}) Return a;
```

Pour mener à bien ce cas d'application, plusieurs étapes clés ont été définies et suivies avec rigueur. Tout d'abord, il a été nécessaire de sélectionner un modèle de LLM adapté à nos besoins, capable de générer du langage Cypher.

Une fois le modèle choisi, l'étape suivante a consisté à le préparer et à l'entraîner à l'aide d'un jeu de données soigneusement constitué pour couvrir une gamme variée de scénarios de requêtes Cypher. L'entraînement du modèle a été complété par l'application de diverses méthodes d'optimisation identifiées lors de l'étude de l'état de l'art, dans le but d'améliorer ses performances.

Enfin, pour évaluer de manière objective et précise l'efficacité du modèle entraîné, un système d'évaluation personnalisé a été mis en place. Ce système a permis de mesurer la qualité des requêtes Cypher générées par le modèle. Cette démarche d'évaluation a joué un rôle crucial pour valider les performances du modèle et pour extraire des conclusions significatives sur les méthodes utilisées.

### 5.1.2 Le choix du modèle

Nous devions choisir un modèle qui soit performant et adapté à nos besoins. Notamment, l'une des contraintes était la compatibilité des modèles avec les différentes méthodes de compression. Nous avons identifié une architecture de modèle adéquate : LLaMa. En effet, cette architecture est compatible avec la plupart des méthodes qui ont été recherchées et Llama est un des modèles open source ayant les meilleures performances sur les résultats étudiés pendant l'état de l'art. De plus, Llama se décompose en plusieurs modèles dont un qui a été entraîné spécifiquement sur de la génération de code : **CodeLLaMa-7B-Instruct**. C'est un modèle spécialisé en génération de code sur plusieurs langages différents, et qui a été entraîné avec des entrées en langage naturel. Ce modèle existe également en taille 13b et 34b, mais nous avons choisi le modèle 7b pour limiter les ressources utilisées car l'objectif ici n'est pas d'avoir un résultat qui pourrait être utilisé en production mais de passer d'un cas théorique à un cas pratique. [32]

### 5.1.3 Les données

La construction et l'évaluation de nos modèles s'appuient sur un jeu de données spécialement conçu à cet effet. Ce jeu est constitué d'un ensemble de paires, chaque paire comportant une instruction en langage naturel et la requête Cypher correspondante, attendue comme sortie suite à l'instruction donnée au modèle.

Initialement, nous avons élaboré notre jeu de données en nous basant sur deux sources principales : la documentation de Cypher, qui fournit plusieurs exemples, et un jeu de données open source axé sur le thème des films. Cette démarche nous a rapidement confrontés à l'importance cruciale d'un jeu de données de qualité. En effet, nous avons rencontré plusieurs obstacles. Le premier était que la majorité des entrées étaient centrées sur des instructions cinématographiques, entraînant un risque de sur-entraînement du modèle sur ce domaine spécifique et compromettant sa capacité à généraliser à d'autres domaines. Autrement dit, le modèle risquait d'exceller sur les données entraînées tout en étant inefficace sur tout le reste. Le second problème relevait de la présence de doublons, de requêtes approximatives ou d'instructions incohérentes dans le jeu de données open source, ce qui aurait "empoisonné" le modèle avec des données de mauvaise qualité.

Face à ces constats et malgré le temps considérable requis, nous avons pris la décision de reconstruire entièrement le jeu de données. Nous avons diversifié les sujets et sélectionné uniquement les meilleures entrées des deux sources initiales, en complétant notre jeu avec des instructions et des requêtes générées manuellement par nos soins.

Le jeu de données finalisé comprend 285 entrées. Nous avons opté pour une division en deux parties : environ 85% des entrées forment le jeu d'entraînement, utilisé pour entraîner les modèles, tandis que les 15% restants constituent le jeu de test, déployé lors de l'évaluation des modèles. Cette séparation en deux jeux de données distincts et fixes répond à deux objectifs. D'une part, elle garantit que tous les modèles soient entraînés et évalués sur les mêmes données, assurant ainsi que les variations de performance d'un modèle à l'autre résultent bien des paramètres et méthodes employés, et non de différences dans la qualité des jeux de données. D'autre part, cette décision simplifie l'interaction avec notre base de données utilisée pour les tests, dont le fonctionnement est expliqué dans la partie [5.2.3].

## 5.2 Évaluation et comparaison des modèles

### 5.2.1 Introduction

Dans cette partie du rapport, nous abordons l'aspect crucial de l'évaluation des LLMs que nous développons dans le cadre de notre cas d'application. L'objectif est de mettre en place un ensemble d'outils et de méthodologies pour évaluer les performances de nos modèles. Cette évaluation joue un rôle essentiel, car elle nous permettra de déterminer l'efficacité des modèles que nous allons entraîner, et ainsi en tirer des déductions sur certaines méthodes utilisées. Ce processus s'est effectué en plusieurs étapes.

Premièrement, nous avons choisi des métriques d'évaluation pour nos modèles, c'est-à-dire un ensemble de critères spécifiques pour mesurer et comparer les performances des différents modèles. Ces critères ont été sélectionnés d'un compromis d'une part pour refléter les aspects les plus pertinents des performances des LLMs et d'autre part par rapport aux moyens que nous avions à disposition.

Puis nous avons mis en place une base de données permettant de requêter du code Cypher. Cette base de données joue un rôle central dans l'évaluation des modèles, en permettant de réaliser des requêtes spécifiques et d'obtenir des résultats qui serviront à évaluer les performances de nos modèles en conditions réelles.

Enfin, nous avons développé un notebook qui intègre ces éléments, fournissant un cadre structuré pour l'évaluation automatique des modèles. Ce notebook est l'outil central et final, permettant de simplifier et de standardiser le processus d'évaluation, garantissant ainsi une approche méthodique et reproductible pour tous les modèles.

Cette partie du rapport détaille donc la mise en place d'un système d'évaluation complet pour nos modèles de LLMs, ainsi que la présentation des résultats obtenus sur les modèles testés.

### 5.2.2 Choix des métriques d'évaluation

Pour établir des métriques, nous nous sommes appuyés sur l'état de l'art et sur l'ensemble des recherches menées, ce qui nous a permis de connaître et de comprendre les métriques applicables à l'évaluation de nos modèles. Nous avons décidé d'utiliser trois métriques principales.

En premier lieu, le temps d'inférence, qui correspond au temps nécessaire pour que le modèle génère sa réponse. Cette métrique est cruciale, car un modèle affichant de bons résultats mais nécessitant un temps d'inférence excessif ne serait pas utilisable pour un client.

La deuxième métrique est l'utilisation de codeBERTScore, qui a été présentée lors de notre revue de l'état de l'art. Comme indiqué précédemment, codeBERTScore ne peut pas être utilisé pour analyser sémantiquement ou fonctionnellement le code Cypher, car CodeBERT n'a pas été entraîné sur ce langage. Néanmoins, cette métrique s'avère utile pour examiner les requêtes générées d'un point de vue syntaxique, ce qui permet d'évaluer dans quelle mesure la requête générée se rapproche syntaxiquement de la requête de référence.

La troisième et dernière métrique est un pourcentage de tests unitaires réussis par le modèle. Cette mesure est fondamentale pour évaluer la fonctionnalité du code généré, qui est la principale finalité de la génération de code. À notre avis, c'est une métrique indispensable pour déterminer quel modèle

produit les meilleures requêtes d'un point de vue sémantique.

Ces trois métriques forment donc notre benchmark, qui servira de cadre de référence pour l'évaluation de nos modèles.

### 5.2.3 Mise en place d'une base de données

Pour évaluer efficacement nos modèles, une des métriques essentielles implique l'exécution de requêtes sur une base de données, suivie d'une comparaison des résultats pour déterminer si le test unitaire est réussi ou non. À cette fin, nous avons opté pour l'utilisation de Neo4j Aura, un service qui permet de créer et d'interroger une base de données Neo4j dans le cloud à l'aide du langage Cypher.

La principale difficulté résidait dans le besoin de concevoir une base de données capable de fournir des résultats pertinents pour toutes les requêtes présentes dans notre jeu de test. Cela impliquait de créer un ensemble complexe de tables, de relations et de données pour couvrir l'intégralité du jeu de test, une tâche considérable qui aurait demandé beaucoup plus de temps que ce que notre planning et l'avancement prévu du projet pouvaient permettre. Pour surmonter cet obstacle, nous avons décidé de limiter les entrées de notre jeu de test à des domaines spécifiques, tels que le cinéma (films, réalisateurs, acteurs, producteurs, dates de sortie, etc.) et autour des produits et des commandes. Cette approche a deux avantages : elle vérifie que le modèle, bien qu'entraîné sur une variété de sujets, est capable de générer des requêtes dans un domaine non abordé pendant l'entraînement, et elle simplifie la création de la base de données en réduisant le nombre de relations et de tables nécessaires.

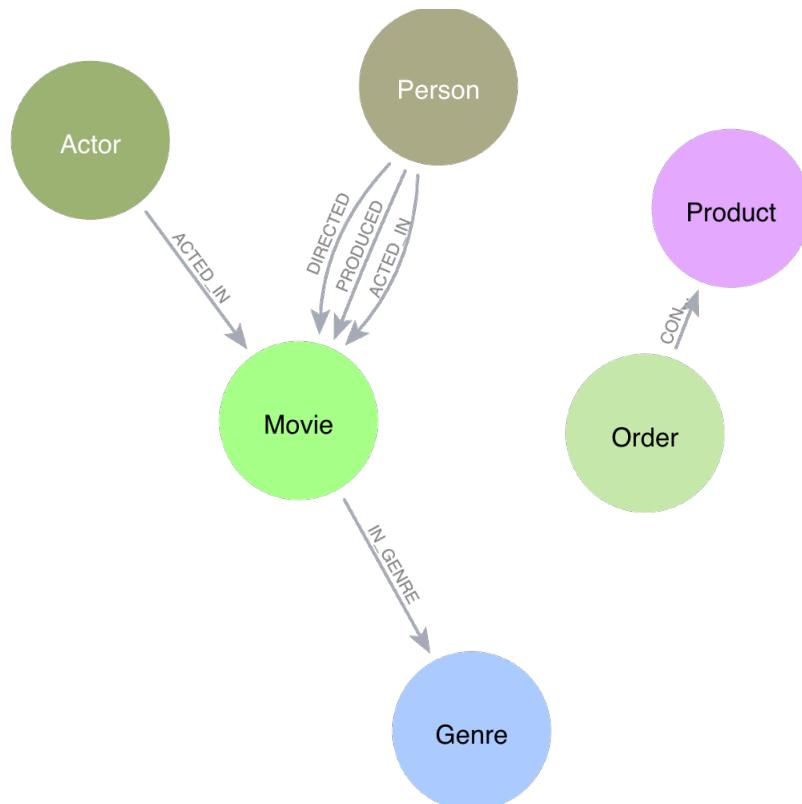


FIGURE 5.1 – Schéma de la base de données

La figure 5.1 montre le schéma final de notre base de données Neo4j. Chaque cercle représente une table, tandis que chaque flèche illustre une relation entre deux tables. Ce schéma offre une vue

d'ensemble claire de la structure de la base de données, facilitant ainsi l'exécution et l'évaluation des requêtes générées par nos modèles.

En plus de la structuration de la base de données, nous avons pris soin de peupler chaque table et chaque relation avec des données fictives soigneusement conçues. Cette démarche était indispensable pour garantir que notre base puisse fournir des résultats adéquats pour l'ensemble des tests unitaires de notre jeu de test.

## 5.2.4 Développement d'un système d'évaluation des modèles

Les éléments précédents doivent donc être assemblés pour créer un système qui, selon un modèle en entrée, donne une évaluation correspondant au calcul de nos métriques en fonction du jeu de test en sortie.

### 5.2.4.1 Génération des requêtes

Le code en annexe (A) sert de préparation pour l'évaluation des performances de nos LLMs, en générant des requêtes basées sur un jeu de données de test. Le jeu de données, chargé depuis un fichier Excel nommé 'Jeu\_test.xlsx', contient deux colonnes principales : 'instruction', qui inclut les prompts en langage naturel, et 'output', qui contient les requêtes Cypher de référence attendues en sortie.

La fonction 'generate\_requests()' parcourt chaque instruction du jeu de données pour générer une requête candidate correspondante à l'aide du modèle. Pour chaque instruction, la fonction mesure le temps d'inférence, c'est-à-dire le temps nécessaire pour que le modèle génère la requête à partir de l'instruction. Ce temps est calculé en enregistrant le moment précis avant et après l'appel à la fonction generate(), qui est responsable de la génération de la requête basée sur l'instruction fournie. Les requêtes candidates générées ainsi que les temps d'inférence sont stockés dans des listes pour une utilisation ultérieure.

Le résultat final de cette fonction est un triplet composé des requêtes de référence, des requêtes candidates générées par le modèle, et des temps d'inférence. Ces éléments servent de base pour réaliser les métriques nécessaires à l'évaluation complète du modèle.

### 5.2.4.2 Génération des tests unitaires

Le code en annexe (B) illustre la mise en place d'une suite de tests unitaires pour évaluer la capacité des LLMs à générer des requêtes Cypher correctes et fonctionnelles. Les tests sont basés sur la comparaison entre des requêtes Cypher de référence et des requêtes générées par les modèles, en utilisant la base de données Neo4j pour exécuter ces requêtes et comparer leurs résultats. Voici une explication détaillée des deux fonctions principales :

Fonction create\_unit\_tests() :

Cette fonction construit une suite de tests unitaires de manière dynamique. Elle parcourt simultanément les listes de requêtes de référence (reference\_queries) et les requêtes candidates générées par le modèle (candidate\_queries). Pour chaque paire de requêtes, elle crée une fonction de test qui sera exécutée pour comparer les deux requêtes à l'aide de la fonction execute\_cypher\_comparison().

Chaque fonction de test est ajoutée à la suite de tests. Cette dernière représente l'ensemble des validations que les requêtes générées par le modèle doivent passer pour être considérées comme correctes.

#### Fonction execute\_cypher\_comparison()

La fonction execute\_cypher\_comparison() est responsable de l'exécution des requêtes Cypher dans une base de données Neo4j et de la comparaison de leurs résultats. Elle établit d'abord une connexion à la base de données à l'aide des identifiants fournis. Ensuite, pour chaque paire de requêtes, elle exécute les requêtes dans la base de données Neo4j et compare leurs résultats. Si les résultats obtenus de la requête testée ne correspondent pas à ceux de la requête de référence, le test échoue, signalant une incohérence ou une erreur dans la requête générée par le modèle.

#### 5.2.4.3 Évaluation du modèle

Dans l'annexe (C), on exécute finalement la fonction d'évaluation du modèle. Cette fonction calcule les résultats des tests unitaires, les résultats de codeBERTScore (Précision, rappel, F1 et F3 score) ainsi que le calcul de la moyenne du temps d'inférence. Ces résultats sont formatés puis enregistrer dans un fichier qui nous permet de garder un historique des résultats des différents modèles évalués.

## 5.3 Apprentissage

### 5.3.1 Le prompt

La première étape pour l'apprentissage a été de choisir un prompt adéquat pour le modèle CodeLLaMa-7B-Instruct-hf.

Nous savons que LLaMa 2 avait été entraîné à l'aide d'un prompt sous la forme suivante :

```
1 <s>[INST] <>SYS>>
2 system_prompt
3 <>/SYS>>
4
5 user_prompt [/INST] model_answer </s>
```

Nous savons également que CodeLlama est un modèle LLaMa 2 qui a été ré-entraîné. Nous avons donc décidé d'utiliser ce prompt.

Pour optimiser la génération et éviter que le modèle ne génère de commentaires en plus du code, nous avons défini la variable system\_prompt de la manière suivante :

```
1 system_prompt = "Provide answers in Cypher for Neo4j graph databases. Do not
   give comments. Do not give explanation. Create a Cypher statement to answer
   the following question:"
```

Cette chaîne de caractère a été la meilleure que l'on ait trouvée dans notre cas d'utilisation pour que le modèle génère principalement du code et évite les commentaires.

### 5.3.2 Formattage des données

Après avoir effectuer un pretraitement du dataset créé, nous avons transformé le dataset de type **pandas** en un type **Dataset** de huggingface pour les besoins d'apprentissage. Ensuite nous avons concaténé l'input et l'output pour former une séquence que notre modèle doit apprendre à générer (figure 5.2).

```
def formatting_prompts_func(dataset):
    output_texts = []
    for i in range(len(dataset)):
        text = dataset['instruction'][i] + dataset['output'][i] + "</s>"
        output_texts.append(text)
    return output_texts
```

FIGURE 5.2 – Formattage des données

### 5.3.3 Entraînement

Nous avons utilisé une méthode PEFT(voir page 10) qui est la low-rank adaptation(Lora) pour notre travail car cette méthode était la plus utilisée et donnait de meilleurs résultats d'après la littérature.

#### 5.3.3.1 Paramètres Lora

Nous avons principalement appliquer la méthode Lora sur les couches query, key et value du composant multi-attention de notre modèle. Il faut noter que plus le rang r des matrices augmente, plus le nombre de paramètres entraînables augmente. Après plusieurs test, nous avons retenu un rang r=16 et un coefficient lora\_alpha=32, ce qui correspond à 18% des paramètres du modèle choisi (7milliards de paramètres) qui sont entraînables. Pour ces valeurs fixées, nous avons également tester différentes valeurs de lora\_dropout qui représente le pourcentage de neurones qui faut désactiver pour chaque couche en vue d'éviter le sur-apprentissage.

#### 5.3.3.2 Paramètres d'entraînement

Nous avons principalement testé les hyperparamètres suivants :

- Coefficient d'apprentissage pour contrôler la taille des pas que l'optimiseur effectue lors de la mise à jour des poids du modèle pendant l'entraînement.
- Paramètre de régularisation (weight decay) pour éviter le surapprentissage en ajoutant une pénalité sur les poids du modèle pendant l'entraînement.
- Nombre d'epochs correspond au nombre de fois que l'ensemble du jeu de données est passé à travers le modèle pendant l'entraînement.
- Batch size correspond au nombre d'échantillons des données d'entraînement utilisées dans une itération pour calculer la perte et mettre à jour les poids du modèle.

Ensuite, nous avons effectué l'entraînement avec la classe **SFTtrainer** de Hugginface qui a été conçu pour finetune des LLMs pré-entraînés. Enfin, tous les modèles finetune testés ont été envoyés sur hugginface pour les tests sur les métriques d'évaluation.

### 5.3.3.3 Résultats des entraînements

Nous avons retenu 3 modèles(finetuningqkv5, llamacfinetunelast, finetuningqvlast) qu'on a push sur huggingface et dont on peut voir les différences à travers les loss (pertes) sur les données d'entraînement et de test (figures 5.3, 5.4). On constate qu'il n'y a pas de sur-apprentissage lors de l'entraînement de ces modèles. En principe, le meilleur modèle est celui qui a la plus petite loss (le modèle en rouge sur le graphique).

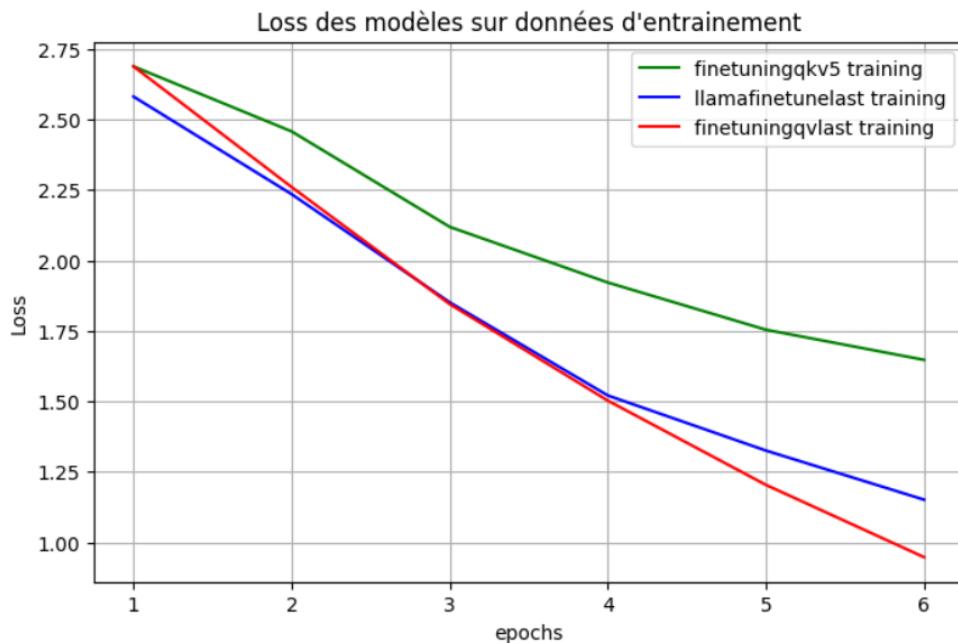


FIGURE 5.3 – Loss sur données d'entraînement

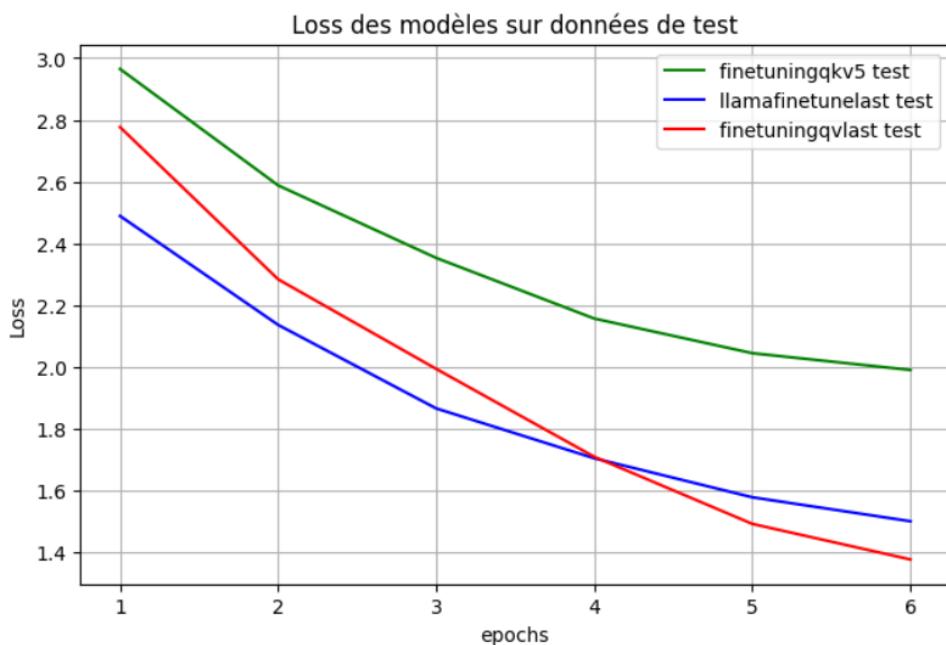


FIGURE 5.4 – Loss sur données de test

### 5.3.3.4 Résultats et analyse des performances des modèles fine-tune

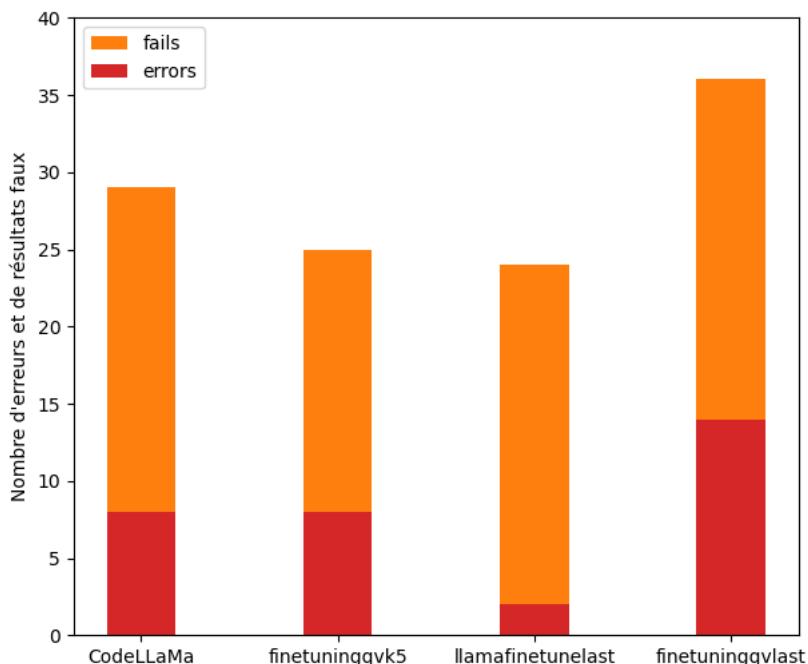


FIGURE 5.5 – Erreurs et échecs des modèles sur 43 tests

On constate sur ce graphique(voir figure 5.5) que le modèle ayant la plus petite loss (finetuningqvlast) sur les données de test à la fin de l’entraînement n’est pas celui qui a les meilleures performances en termes de bonnes requêtes générées. Aussi, on constate que les 2 autres modèles fine-tune ayant des loss un peu plus élevées sur les données de test donnent de meilleurs résultats que le modèle CodeLlama de base. Cela pourrait s’expliquer par le fait qu’on teste l’égalité des résultats entre une requête générée par le modèle finetune et la requête attendue. Par exemple, certains tests du modèle finetune ayant la plus petite loss donnait un résultat de la forme [1,2] or la requête attendue était [2,1]; Vu qu’on faisait le test  $[1,2] == [2,1]$ , le test nous disait que la requête générée par le modèle finetune est fausse. En outre, le fait que le contexte donné au prompt ne soit pas optimisé peut poser problème : On pourrait par exemple définir les noms des noeuds de la base de données sur lesquels on veut effectuer les requêtes dans le prompt. En somme, on pourrait dire que les tests unitaires ne sont pas représentatifs de la fonction de perte (loss). En d’autres termes ce n’est pas parce que la loss du modèle sur les données de test est très petite que le modèle passera plus de tests unitaires. Sur le graphique(voir figure 5.6), on peut aussi voir les métriques indiquant les similarités sémantiques pour les modèles fine-tune et le modèle de base. On peut constater que le modèle qui génère le plus de requêtes sémantiquement similaires aux requêtes attendues est le modèle fine-tune nommé llamafinetunelast qui, est d’ailleurs le meilleur modèle sur les tests unitaires.

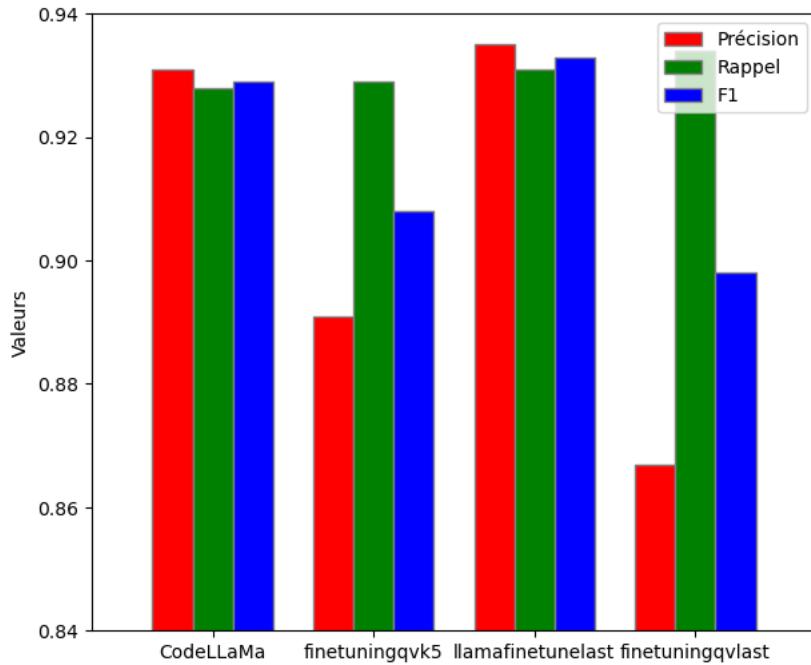


FIGURE 5.6 – Métriques pour similarités sémantiques

## 5.4 Optimisation

### 5.4.1 Introduction

Dans cette section, nous allons vous présenter les démarches réalisées et les résultats obtenus lors de la mise en pratique des méthodes d'amélioration découvertes lors de l'état de l'art. Ayant pris du retard sur l'apprentissage du modèle de notre cas d'utilisation, les tests ont été principalement réalisés sur le modèle CodeLLaMa-7B-Instruct-hf.

### 5.4.2 Le prompt

La première étape d'optimisation a été d'utiliser un prompt adéquat pour le modèle CodeLLaMa-7B. En effet, nous avons passé nos premiers tests avec un prompt naïf qui comportait uniquement la requête de l'utilisateur.

Cependant, lorsque l'on est passé au prompt que l'on utilise pendant l'entraînement (décris à la section 5.3.1), nous avons pu voir à travers nos tests que les requêtes étaient générées beaucoup plus rapidement. Pour une requête générée en **14** secondes avec le prompt sans forme particulière, la requête prenait **4** secondes avec le prompt au format présenté ci-dessus (tests réalisés sur colab T4 GPU). Le prompt joue donc un rôle très important dans l'optimisation du modèle.

### 5.4.3 Les environnements d'exécution

Concernant l'inférence des modèles, nous avions prévus de tester 6 environnements d'exécution. Vous trouverez dans le tableau suivant une liste de tests des environnements. Ce tableau indique le

modèle utilisé, l'environnement utilisé et indique également si nous avons réussi à faire fonctionner le modèle dans l'environnement concerné.

Modèle	Environnement	Test réussi	Raison de l'échec
CodeLLaMa	Pytorch	Oui	-
CodeLLaMa	ONNX	Oui	-
CodeLLaMa	TensorFlow Lite	Non	LLaMa non supporté.
CodeLLaMa	OpenVINO	Non	Erreur non identifiée.
CodeLLaMa	DeepSparse	Non	"text-generation" non supportée.
CodeLLaMa	DeepSpeed	Non	Erreur non identifiée.
CodeLLaMa	GGUFe	Oui	-

Comme vous pouvez le voir dans le tableau, nous avons réussi à faire fonctionner uniquement les modèles au format ONNX et GGUF. Nous vous présenterons donc les résultats de CodeLLaMa-7B-Instruct-hf au format PyTorch, ONNX et GGUF.

Pour tester les performances des différentes quantifications, nous avons utilisé la fonction de test qui se trouve en annexe (B) sur le modèle CodeLLaMa au format original PyTorch, puis au format GGUF et ONNX. Vous pouvez vous référer à la section précédente 5.2 pour plus d'informations sur le fonctionnement des tests.

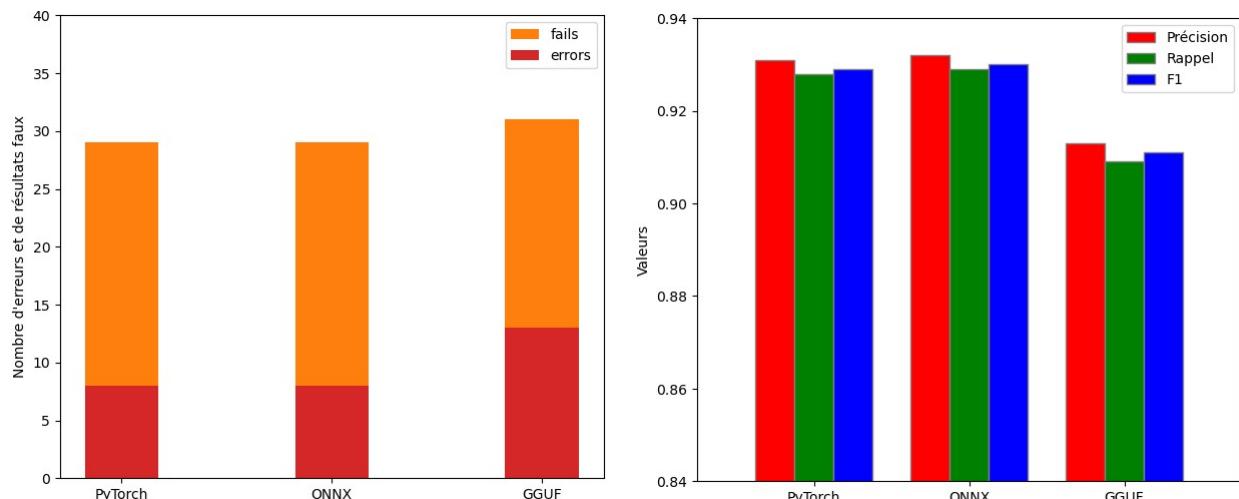


FIGURE 5.7 – Résultats des tests unitaires et de codeBERTScore [5.2] sur le modèle CodeLLaMa-7B-Instruct-hf au format PyTorch, ONNX et GGUF.

**Performances** La figure 5.7 nous montre que la conversion au format GGUF a dégradé légèrement le modèle. On a 5 tests unitaires de moins qui sont validés, et la précision donnée par codeBERTScore a diminué de presque 2%. Pour le modèle ONNX, rien d'anormal à signaler, le modèle semble garder des performances identiques.

Pour le cas de la conversion en GGUF, rien dans la littérature n'indique une dégradation des modèles convertis. Nous nous sommes penchés sur la question, mais nous ne sommes tombés sur aucune discussion à ce propos sur le web. Nous avons utilisé la commande suivante fournie en clonnant le projet GitHub llama.cpp :

```
1 python llama.cpp/convert.py <local_model_path> --outfile <outmodel>.gguf
```

Ce n'est en théorie qu'une conversion en C/C++ du modèle PyTorch. Nous restons donc sur une question à laquelle nous ne répondrons pas : quelle peut-être la cause de la dégradation des performances, et est-ce un cas isolé ?

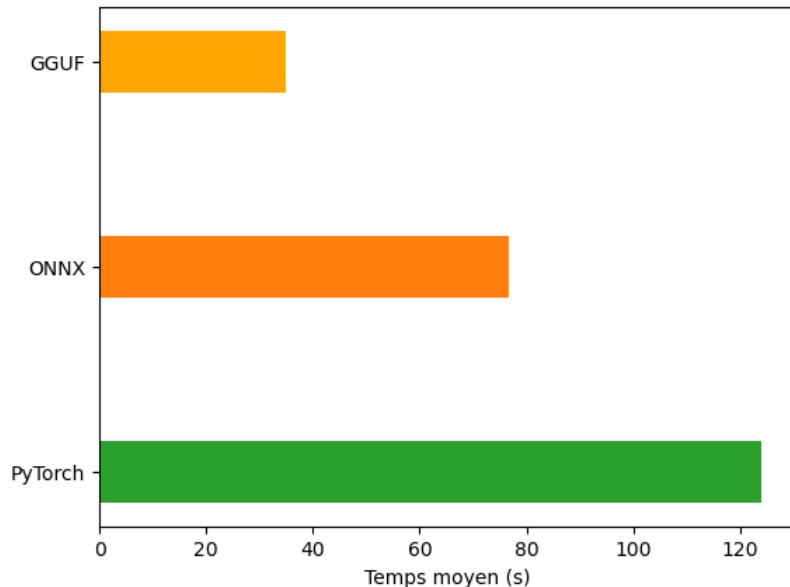


FIGURE 5.8 – Résultats des temps d’inférence moyen sur CPU [5.2] sur le modèle CodeLLaMa-7B-Instruct-hf au format PyTorch, ONNX et GGUF.

**Temps d’inférence** Pour le temps d’inférence moyen, nous avons fait les tests sur CPU. Nous voyons que les formats ONNX et GGUF sont beaucoup plus optimisés pour l’inférence que le modèle au format PyTorch. Ils accélèrent l’inférence. Enfin, on peut remarquer que le format GGUF est celui qui présente les meilleurs temps d’inférence sur CPU.

**Conclusion** Le format PyTorch est le bon format pour manipuler et entraîner un modèle. Le format ONNX est un bon premier format pour utiliser les modèles en inférence tout en ne modifiant pas les performances. Enfin, le format GGUF est le plus optimisé en temps pour l’inférence, cependant il faudra faire bien attention à réévaluer son modèle pour vérifier les performances après conversion.

#### 5.4.4 Les différentes compressions

Pour la compression des modèles, nous avons essayé de faire fonctionner des quantifications et des élagages sur le modèle CodeLLaMa-7B-Instruct-hf. Voici un tableau récapitulatif de nos tests :

Modèle	Catégorie	Nom de la méthode	Test réussi
CodeLLaMa	Quantification	bitsandbytes	Non
CodeLLaMa	Quantification	AutoGPTQ	Non
CodeLLaMa	Quantification	ONNX Quantization	Non
CodeLLaMa	Quantification	GGUF Quantization	Oui
CodeLLaMa	Quantification	Omniquant	Non
CodeLLaMa	Elagage	LLM Pruner	Non

La seule méthode de compression que l’on a réussi à faire fonctionner a été la quantification des modèles au format GGUF grâce au code fourni par le projet llama.cpp. Nous n’avons pas réussi à

faire fonctionner les autres méthodes par manque de temps. L'implémentation des différentes méthodes n'étant pas assez mature, beaucoup d'erreurs apparaissaient lors de l'exécution. Certaines des méthodes demandaient beaucoup trop de mémoires RAM pour fonctionner, d'autres nous indiquaient des erreurs de type lors des calculs effectués dans les programmes utilisés sur nos modèles et un certain nombre des méthodes avaient des problèmes de dépendance compliqués à résoudre. Les actions que l'on a entrepris pour contourner ces différents types d'erreurs n'ont abouti qu'au mieux sur l'apparition d'autres erreurs après le contournement de la première. Une examination plus approfondie des fichiers source aurait pu permettre de faire fonctionner certaines de ces méthodes. Cependant, nous n'avons pas réussi à allouer plus de temps pour la résolution des bugs.

Nous avons donc testé le modèle CodeLLaMa au format GGUF quantifié en 8, 5, 4, et 3 bits.

Pour tester les performances des différentes quantifications, nous avons utilisé toujours la même fonction de test qui se trouve en annexe (B) sur les différents modèles obtenus.

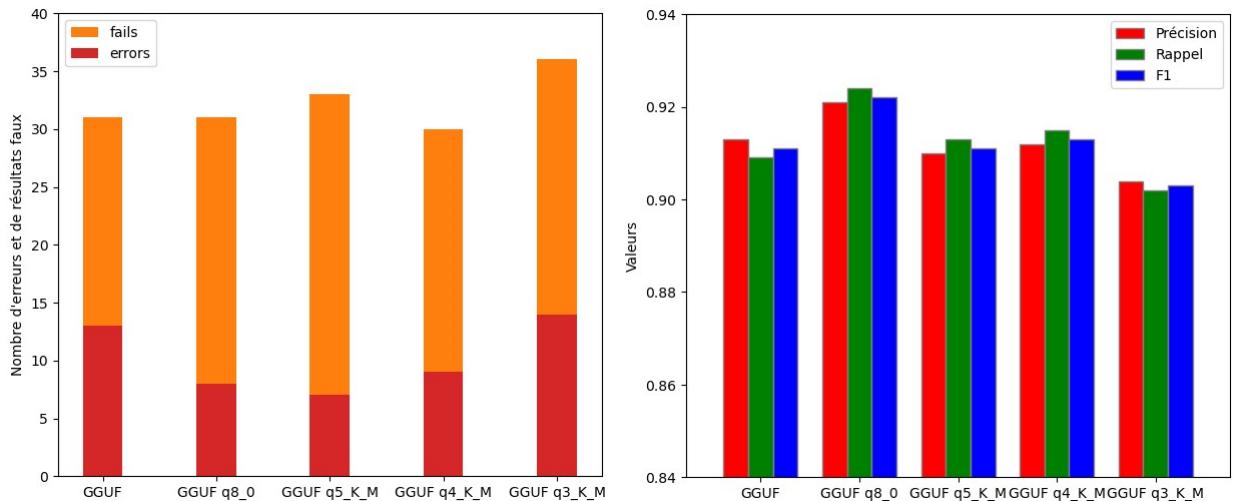


FIGURE 5.9 – Résultats des tests unitaires et de codeBERTScore [5.2] sur le modèle CodeLLaMa-7B-Instruct-hf au format GGUF et sous différentes quantifications. Les mention sous la forme "q5\_K\_M" correspondent aux différents paramètres de la méthode de quantification issue du code du projet llama.cpp. [13]

**Performances et temps d'inférence** En observant les figures 5.9 et 5.10, on peut remarquer que la quantification ne dégrade pas significativement le modèle. C'est un peu moins vrai pour la quantification en 3 bits qui affiche le nombre de tests unitaires manqué le plus haut et une baisse de 1% de précision.

Un résultat surprenant est l'amélioration des résultats lors de quantification en 8 bits. En effet, on voit dans la figure 5.9 que la quantification en 8 bits améliore la précision de 1%. De plus, on observe de meilleures performances également dans les tests unitaires. On voit que 5 tests sont passés d'erreur à échec comparé aux résultats du modèle sans quantification.

Au niveau des temps d'inférence, on peut voir que plus le modèle est quantifié, plus la réponse est rapide.

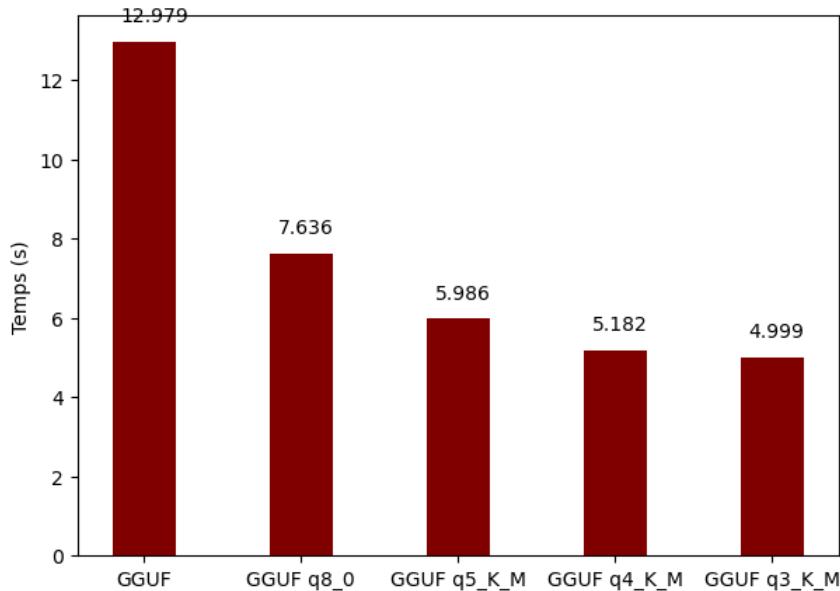


FIGURE 5.10 – Temps d’inférence moyen [5.2] sur le modèle CodeLLaMa-7B-Instruct-hf au format GGUF et sous différentes quantifications. Voir la description à 5.9 pour plus d’informations sur les mentions de la forme “q5\_K\_M”.

**Conclusion** La quantification n’impacte pas significativement la qualité du modèle tout en étant plus efficace en temps et en ressources. Parfois, le modèle semble obtenir de meilleures performances d’après nos résultats. Quantifier un modèle sous différentes précisions est donc une bonne initiative à entreprendre lors d’une phase d’optimisation d’un modèle.

## 5.5 Difficultés rencontrées et pistes d’améliorations

Au cours de ce projet, notre groupe a été confronté à plusieurs défis, qui ont enrichi notre expérience et nous ont permis de mieux comprendre les complexités inhérentes à l’application pratique de l’intelligence artificielle.

### Difficultés rencontrées

- Méconnaissance initiale : Pour certains membres du groupe, cette expérience a été une première interaction avec l’intelligence artificielle dans son ensemble, nécessitant un effort pour se familiariser avec les concepts de base. De plus la méconnaissance de l’environnement et les fonctionnalités de IBM Watson à parfois nécessité un temps d’adaptation.
- Utilisation de Cypher : Le choix de travailler avec le langage de requête Cypher a introduit des difficultés supplémentaires. Sa moindre popularité a limité notre accès à des méthodes récentes optimisées pour des langages plus largement utilisés. De plus, l’application d’un langage de requête dans nos tests a rendu le processus plus complexe que pour des langages de programmation classiques, comme Python. La nécessité d’obtenir des résultats spécifiques via des interactions avec une base de données a ajouté une couche de complexité à la mise en place des tests.
- Passage de la théorie à la pratique : La transition de concepts et résultats théoriquement “parfait” durant l’état de l’art à leur application pratique a été jalonnée d’imprévus, de bugs et de problèmes de compatibilité. Notre progression a souvent été marquée par des avancées

suivies de reculs, rendant le processus moins fluide et plus difficile que prévu.

#### Pistes d'amélioration

- Enrichissement du jeu de données : Une première piste d'amélioration consisterait à augmenter la qualité et la quantité de notre jeu de données si nous avions plus de temps sur le projet. Cela permettrait d'affiner l'entraînement de notre modèle et d'améliorer sa capacité à générer des requêtes Cypher précises et pertinentes.
- Exploration de nouvelles méthodes et modèles : Il serait également judicieux de continuer à explorer d'autres méthodes et peut-être d'expérimenter avec d'autres modèles que celui que nous avons utilisé (Llama). L'adoption de différentes approches pourrait révéler des optimisations potentielles ou des alternatives plus efficaces pour notre cas d'application.
- Renforcement de l'expérience pratique : Enfin, une familiarisation plus approfondie avec les outils d'IA et une meilleure préparation à la résolution de problèmes techniques inattendus pourraient considérablement atténuer les difficultés rencontrées lors du passage de la théorie à la pratique. Cela implique une mise à jour continue de nos connaissances et une adaptation agile face aux défis techniques.

## 6 Évolution du Gantt

Les différentes phases du projet présentées dans le Gantt ont été respectés. Cependant, nous avons eu un retard sur le premier livrable. Il a été repoussé jusqu'à la fin du projet.



FIGURE 6.1 – Les trois Gantt réalisés tout au long du projet.



FIGURE 6.2 – Légende des éléments utilisés dans les Gantt précédents.

## 7 Conclusion

Le projet "Amélioration des modèles IA génératifs de texte" a généré une plus-value pour l'ensemble des parties prenantes concernées.

Pour Sopra Steria, les résultats de nos recherches constituent une mine d'informations, qui se traduira par un gain de temps considérable et qui pourra être utilisé comme une aide à la décision pour l'adoption des LLMs dans de futurs projets.

Pour notre équipe étudiante, ce projet a été l'occasion de faire preuve de rigueur dans le travail de recherche et de se confronter à la complexité de la mise en pratique de ces technologies. Les compétences acquises en intelligence artificielle, et tout particulièrement sur les LLMs, constituent un atout professionnel indéniable pour notre avenir, en plus du savoir-faire développé en gestion de projet (prévision projet, préparation des réunions, rédaction des comptes rendus, répartition des tâches...).

Quant à l'objectif principal du projet qui était d'apporter une meilleure vision sur les méthodes d'apprentissage, d'optimisation et de comparaison des LLMs, il a été pleinement atteint en apportant des informations riches et en appliquant des cas concrets pour tester et améliorer les modèles. Les résultats sur le cas d'application, notamment en termes de temps d'inférence et de métriques de performance, montrent que nous avons pu améliorer le modèle de base avec des méthodes sélectionnées.

Ce document étant le résultat de notre travail, celui-ci permettra également de partager notre démarche et nos conclusions avec le lecteur. Des conclusions que l'on peut résumer en quelques points. La première chose est que l'entraînement d'un LLM permet d'améliorer significativement la génération. Ensuite, l'utilisation d'un format de modèle et d'un environnement spécifique permet d'optimiser la mise en production en accélérant l'inférence. Enfin, la quantification permet une accélération supplémentaire de l'inférence en allégeant le modèle sans dégradation significative. L'élagage, bien qu'elle soit moins populaire, a les mêmes objectifs que la quantification : alléger et accélérer le modèle avec un minimum de dégradation.

En prenant en compte la valeur ajoutée de ce projet pour l'ensemble des parties prenantes, nous pouvons considérer que le projet est une réussite, d'autant plus qu'il s'inscrit dans une tendance importante de l'utilisation croissante d'agents conversationnels et de LLMs, assurant son utilité et sa pertinence sur le long terme.



# Bibliographie / Webographie

- [1] *Get started : PEFT*. HuggingFace. URL : <https://huggingface.co/docs/peft/index>. 16
- [2] *How-to guide : quantization*. HuggingFace. URL : [https://huggingface.co/docs/accelerate/usage\\_guides/quantization](https://huggingface.co/docs/accelerate/usage_guides/quantization). 16
- [3] *How-To Guide : quantization with autoGPTQ*. HuggingFace. URL : <https://huggingface.co/docs/transformers/main/en/quantization#autogptq>. 16
- [4] *How-To Guide : quantization with AWQ*. HuggingFace. URL : <https://huggingface.co/docs/transformers/main/en/quantization#awq>. 16
- [5] *ONNX Runtime*. Hugging Face. URL : [https://huggingface.co/docs/optimum/v1.2.1/en/onnxruntime/modeling\\_ort](https://huggingface.co/docs/optimum/v1.2.1/en/onnxruntime/modeling_ort). 19
- [6] *SparseML Libraries for applying sparsification recipes to neural networks with a few lines of code, enabling faster and smaller models*. GitHub, 2020. URL : <https://github.com/neuralmagic/sparseml>. 20
- [7] *DeepSparse Sparsity-aware deep learning inference runtime for CPUs*. GitHub, 2021. URL : <https://github.com/neuralmagic/deepsparse?tab=readme-ov-file>. 20, 53
- [8] *bitsandbytes*. GitHub, 2022. URL : <https://github.com/TimDettmers/bitsandbytes>. 16
- [9] *Intel® Neural Compressor An open-source Python library supporting popular model compression techniques on all mainstream deep learning frameworks (TensorFlow, PyTorch, ONNX Runtime, and MXNet)*. GitHub, 2022. URL : <https://github.com/intel/neural-compressor>. 16
- [10] *Awesome LLM compression research papers and tools to accelerate LLM training and inference*. GitHub, 2024. URL : <https://github.com/HuangOwen/Awesome-LLM-Compression>. 15, 21
- [11] *OmniQuant : Omnidirectionally Calibrated Quantization for Large Language Models*. GitHub, Août 2023. URL : <https://github.com/OpenGVLab/OmniQuant>. 16
- [12] *Hugging Face Optimum*. GitHub, Juillet 2021. URL : <https://github.com/huggingface/optimum/blob/main/README.md>. 19
- [13] *llama.cpp Inference of Meta's LLaMA model (and others) in pure C/C++*. GitHub, Juillet 2023. URL : <https://github.com/ggerganov/llama.cpp>. 16, 19, 42, 54
- [14] *AWQ : Activation-aware Weight Quantization for LLM Compression and Acceleration*. GitHub, Juin 2023. URL : <https://github.com/mit-han-lab/llm-awq>. 16
- [15] *AutoGPTQ An easy-to-use LLM quantization package with user-friendly APIs, based on GPTQ algorithm (weight-only quantization)*. GitHub, Mars 2023. URL : <https://github.com/AutoGPTQ/AutoGPTQ>. 16

- [16] Jay Alammar. *The Illustrated Transformer*. URL : <https://jalammar.github.io/illustrated-transformer/>. 8, 53
- [17] Niki Parmar Jakob Uszkoreit Llion Jones Aidan N. Gomez Lukasz Kaiser Illia Polosukhin Ashish Vaswani, Noam Shazeer. *Attention Is All You Need*. URL : <https://arxiv.org/abs/1706.03762>. 6, 8, 9, 53
- [18] AWS. *Que sont les grands modèles de langage (LLM)*? URL : <https://aws.amazon.com/fr/what-is/large-language-model/>. 5
- [19] AWS. *Que sont les modèles autorégressifs*? URL : <https://aws.amazon.com/fr/what-is/autoregressive-models/>. 7
- [20] Burak Ceylan. "Large Language Model Evaluation in 2024 : 5 Methods". URL : <https://research.aimultiple.com/large-language-model-evaluation/>. 23
- [21] Aman Chadha. *Primers • Transformers*. URL : <https://aman.ai/primers/ai/transformers/#embeddings>. 6, 53
- [22] DAIR.AI. *Prompt Engineering Guide*. URL : <https://www.promptingguide.ai/techniques>. 13
- [23] devoteam. *Attention is all you need : comprendre le traitement naturel du langage avec les modèles Transformers*. URL : <https://france.devoteam.com/paroles-dexperts/attention-is-all-you-need-comprendre-le-traitement-naturel-du-langage>. 6, 7
- [24] Denis Kuznedelev Elias Frantar. *SparseGPT*. GitHub, Mars 2023. URL : <https://github.com/IST-DASLab/sparsegpt>. 18
- [25] Torsten Hoefler Dan Alistarh Elias Frantar, Saleh Ashkboos. *GPTQ : Accurate Post-Training Quantization for Generative Pre-trained Transformers*. Octobre 2022. URL : <https://arxiv.org/abs/2210.17323>. 16
- [26] Hugging face. "Perplexity of fixed-length models". URL : <https://huggingface.co/docs/transformers/perplexity>. 26
- [27] Horsee Gongfan Fang, Ikko Eltociear Ashimine. *LLM-Pruner On the Structural Pruning of Large Language Models*. GitHub, July 2023. URL : <https://github.com/horseee/LLM-Pruner?tab=readme-ov-file>. 18
- [28] Cobus Greyling. *Eight Prompt Engineering Implementations*. URL : <https://cobusgreyling.medium.com/eight-prompt-engineering-implementations-fc361fdc87b>. 12, 13, 53
- [29] Maarten Grootendorst. *3 Ways To Improve Your Large Language Model*. Septembre 2023. URL : <https://www.maartengrootendorst.com/blog/improving-l1ms/>. 13, 14, 53
- [30] Sagar Gyanchandani. *What is ONNX?* Auriga. URL : <https://aurigait.com/blog/onnx-onnx-runtime-and-tensorrt/>. 19, 53
- [31] Gautier Izacard Xavier Martinet Marie-Anne Lachaux Timothee Lacroix Baptiste Rozière Namman Goyal Eric Hambro Faisal Azhar Aurelien Rodriguez Armand Joulin Edouard Grave Guillaume Lample Hugo Touvron, Thibaut Lavril. *LLaMA : Open and Efficient Foundation Language Models*. URL : <https://arxiv.org/pdf/2302.13971.pdf>. 9, 53
- [32] HumanEval. "Big Code Models Leaderboard". URL : <https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard>. 31
- [33] Jaid. "F is for F1 Score". URL : <https://jaid.io/blog/f-is-for-f1-score/>. 25

- [34] Jeremy Jouvance. *The LLM Revolution : Boosting Computing Capacity with Quantization Methods*. Medium, 4 juillet 2023. URL : <https://blog.gopenai.com/the-l1m-revolution-boosting-computing-capacity-with-quantization-methods>. 15, 53
- [35] Tengxiao Liu Qinghui Gao Qipeng Guo Xipeng Qiu Kai Lv, Yuqing Yang. *Full parameter Fine-tuning for Large Language models with limited resources*. URL : <https://arxiv.org/pdf/2306.09782.pdf>. 9
- [36] Kawrakow. *Pull request : k-quants1684*. GitHub, 3 juillet 2023. URL : <https://github.com/ggerganov/llama.cpp/pull/1684>. 16
- [37] TOM KELDENICH. "Recall, Precision, F1 Score – Explication Simple Métrique en ML". URL : <https://inside-machinelearning.com/recall-precision-f1-score/>. 24, 25
- [38] Jan Marcel Kezmann. Medium, Décembre 2022. 17, 53
- [39] KiKaBen. *Transformer's Encoder-Decoder*. URL : <https://kikaben.com/transformers-encoder-decoder/>. 7, 53
- [40] Tizian Kronsbein. *Language Learning Models and differentiating the art of prompt design vs. the science of prompt engineering*. DAIN Studios, 21 Juillet 2023. URL : <https://dainstudios.com/insights/prompt-engineering-prompt-design/>. 13
- [41] Maxime Labonne. *Fine Tune Your Own Llama 2 Model in a Colab Notebook.html*. GitHub Pages, 2023. URL : [https://mlabonne.github.io/blog/posts/Fine\\_Tune\\_Your\\_Own\\_Llama\\_2\\_Model\\_in\\_a\\_Colab\\_Notebook.html](https://mlabonne.github.io/blog/posts/Fine_Tune_Your_Own_Llama_2_Model_in_a_Colab_Notebook.html). 16
- [42] latent\_variable. *K quantization vs perplexity*. Reddit, 2023. URL : [https://www.reddit.com/r/LocalLLaMA/comments/1441jnr/k\\_quantization\\_vs\\_perplexity/](https://www.reddit.com/r/LocalLLaMA/comments/1441jnr/k_quantization_vs_perplexity/). 17, 53
- [43] SII Lille. *Data Science – One Hot Encoding*. URL : <https://medium.com/@sii-lille/data-science-one-hot-encoding-c59e82b3f0e7>. 6
- [44] Meta. *Retrieval Augmented Generation : Streamlining the creation of intelligent natural language processing models*. 28 Septembre 2020. URL : <https://ai.meta.com/blog/retrieval-augmented-generation-streamlining-the-creation-of-intelligent-language-processing-models>. 14
- [45] Ikko E. Ashimine Mingjie Sun. *Pruning LLMs by Weights and Activations*. GitHub, Juin 2023. URL : <https://github.com/locuslab/wanda>. 18
- [46] Chunhua Shen Zhen Yang Linlin Ou Xinyi Yu Bohan Zhuang Mingyang Zhang, Hao Chen. *LoRAPrune : Pruning Meets Low-Rank Parameter-Efficient Fine-Tuning*. GitHub, Mai 2023. URL : <https://arxiv.org/abs/2305.18403>. 18
- [47] Shuyan Zhou Uri Alon Sumit Agarwal Graham Neubig. "CodeBERTScore". URL : <https://github.com/neulab/code-bert-score?tab=readme-ov-file>. 29
- [48] Shuyan Zhou Uri Alon Sumit Agarwal Graham Neubig. "CodeBERTScore : Evaluating Code Generation with Pretrained Models of Code". URL : <https://arxiv.org/pdf/2302.05527.pdf>. 29
- [49] Prof Bill Buchanan OBE. "AI Aims For Perfect Coding And With Multiple Solutions". URL : <https://medium.com/asecuritysite-when-bob-met-alice/ai-aims-for-perfect-coding-and-with-multiple-solutions-ab3202542b8e>. 27
- [50] Priyanka. "Perplexity of Language Models". URL : <https://medium.com/@priyankads/perplexity-of-language-models-41160427ed72>. 26

- [51] Sebastian Raschka. *Practical Tips for Finetuning LLMs Using LoRA (Low-Rank Adaptation)*. AHEAD OF AI, 19 Novembre 2023. URL : <https://magazine.sebastianraschka.com/p/practical-tips-for-finetuning-l1ms>. 10, 53
- [52] Sebastian Raschka. *Understanding Parameter-Efficient LLM Finetuning : Prompt Tuning And Prefix Tuning*. AHEAD OF AI, 30 Avril 2023. URL : <https://magazine.sebastianraschka.com/p/understanding-parameter-efficient>. 10
- [53] IBM research blog. *What is prompt-tuning?* URL : <https://research.ibm.com/blog/what-is-ai-prompt-tuning>. 13
- [54] SuperAnnotate. *Fine-tuning large language models (LLMs) in 2024*. URL : <https://www.superannotate.com/blog/l1m-fine-tuning>. 9, 10
- [55] Ayush Thakur. *How to Evaluate, Compare, and Optimize LLM Systems*. URL : <https://wandb.ai/ayush-thakur/l1m-eval-sweep/reports/How-to-Evaluate-Compare-and-Optimize-LLM-Systems--Vm1ldzo0NzgyMTQz>. 23
- [56] Ari Holtzman Luke Zettlemoyer Tim Dettmers, Artidoro Pagnoni. *QLoRA : Efficient Finetuning of Quantized LLMs*. URL : <https://arxiv.org/abs/2305.14314>. 15, 16
- [57] Gyan Prakash Tripathi. *How to Evaluate a Large Language Model (LLM)?* URL : <https://www.analyticsvidhya.com/blog/2023/05/how-to-evaluate-a-large-language-model-l1m/>. 23
- [58] Mart van Baalen Yuki M. Asano Tijmen Blankevoort Tycho F.A. van der Ouderaa, Markus Nagel. *The LLM Surgeon*. Décembre 2023. URL : <https://arxiv.org/abs/2312.17244>. 18
- [59] Zhaoyang Zhang Peng Xu Lirui Zhao Zhiqian Li Kaipeng Zhang Peng Gao Yu Qiao Ping Luo Wenqi Shao, Mengzhao Chen. *OmniQuant : Omnidirectionally Calibrated Quantization for Large Language Models*. Août 2023. URL : <https://arxiv.org/abs/2308.13137>. 16, 53
- [60] Wikiwand. "Zero-shot learning". URL : [https://www.wikiwand.com/en/Zero-shot\\_learning](https://www.wikiwand.com/en/Zero-shot_learning). 28
- [61] Cameron R. Wolfe. *Easily Train a Specialized LLM : PEFT, LoRA, QLoRA, LLaMA-Adapter, and More.* URL : <https://cameronrwolfe.substack.com/p/easily-train-a-specialized-l1m-peft>. 10, 53
- [62] Cameron R. Wolfe. *LLaMA-2 from the Ground Up*. URL : <https://cameronrwolfe.substack.com/p/llama-2-from-the-ground-up>. 11, 53
- [63] Zishan Guo Renren Jin Chuang Liu Yufei Huang Dan Shi Supryadi Linhao Yu Yan Liu Jiaxuan Li Bojian Xiong Deyi Xiong. "Evaluating Large Language Models : A Comprehensive Survey". URL : <https://arxiv.org/abs/2310.19736>. 22, 54
- [64] Yong Liu Can Ma Weiping Wang Xunyu Zhu, Jian Li. *A Survey on Model Compression for Large Language Models*. Août 2023. URL : <https://arxiv.org/abs/2308.07633>. 14, 15, 53

# Liste des illustrations

4.1	Architecture des transformeurs [17] . . . . .	6
4.2	Matrice des incorporations [21] . . . . .	6
4.3	Illustration de l'attention multi-têtes masquée [39] . . . . .	7
4.4	Sortie du décodeur [16] . . . . .	8
4.5	Stades d'entraînement des LLMs [17] . . . . .	8
4.6	jeu de données du modèle Llama [31] . . . . .	9
4.7	Phases de pré-entraînement et de fine-tuning [17] . . . . .	9
4.8	Méthode Lora [51] . . . . .	10
4.9	Couches adaptatives [61] . . . . .	10
4.10	Principe RLHF [62] . . . . .	11
4.11	Le <i>prompt engineering</i> et les disciplines associées. [28] . . . . .	12
4.12	Illustration d'entrées enrichies avec des exemples. [29] . . . . .	13
4.13	Différentes méthodes dites "Thought-based". [29] . . . . .	13
4.14	Illustration de la méthode RAG. [29] . . . . .	14
4.15	Les catégories de compression [64] . . . . .	14
4.16	Illustration de la quantification dans un réseau de neurones [34] . . . . .	15
4.17	Vu d'ensemble des résultats d'Omniquant montrant la capacité à obtenir les performances des méthodes QAT tout en ayant l'efficacité des méthodes PTQ. [59] . . . . .	16
4.18	Comparaison de la perplexité entre différentes quantification d'un même modèle [42]	17
4.19	Illustration de l'élagage dans un réseau de neurones [38] . . . . .	17
4.20	Illustration de la portabilité du format ONNX. [30] . . . . .	19
4.21	Illustration des ressources nécessaires pour un LLM au format GGUF quantifié en 4bits. ( <a href="#">source</a> ) . . . . .	20
4.22	Flux de travail disponible via l'outil DeepSparse. [7] . . . . .	20

4.23	Les différentes types de benchmarks [63] . . . . .	22
4.24	comparatif des modèles générateur de code avec pass@k . . . . .	27
4.25	fonctionnement de la métrique codeBERTScore . . . . .	28
4.26	Illustration de la métrique codeBERTScore . . . . .	29
5.1	Schéma de la base de données . . . . .	33
5.2	Formattage des données . . . . .	36
5.3	Loss sur données d'entraînement . . . . .	37
5.4	Loss sur données de test . . . . .	37
5.5	Erreurs et échecs des modèles sur 43 tests . . . . .	38
5.6	Métriques pour similarités sémantiques . . . . .	39
5.7	Résultats des tests unitaires et de codeBERTScore [5.2] sur le modèle CodeLLaMa-7B-Instruct-hf au format PyTorch, ONNX et GGUF. . . . .	40
5.8	Résultats des temps d'inférence moyen sur CPU [5.2] sur le modèle CodeLLaMa-7B-Instruct-hf au format PyTorch, ONNX et GGUF. . . . .	41
5.9	Résultats des tests unitaires et de codeBERTScore [5.2] sur le modèle CodeLLaMa-7B-Instruct-hf au format GGUF et sous différentes quantifications. Les mention sous la forme "q5_K_M" correspondent aux différents paramètres de la méthode de quantification issue du code du projet llama.cpp. [13] . . . . .	42
5.10	Temps d'inférence moyen [5.2] sur le modèle CodeLLaMa-7B-Instruct-hf au format GGUF et sous différentes quantifications. Voir la description à 5.9 pour plus d'informations sur les mentions de la forme "q5_K_M". . . . .	43
6.1	Les trois Gantt réalisés tout au long du projet. . . . .	45
6.2	Légende des éléments utilisés dans les Gantt précédents. . . . .	46

# Listings

A.1	Génération des requêtes Cypher par le modèle . . . . .	59
B.1	Génération des tests unitaires . . . . .	60
C.1	Evaluation du modèle . . . . .	61



## **Annexes**



## A Génération des requêtes Cypher

```
1 # Jeu de donnees de test
2 dataset = pd.read_excel('Jeu_test.xlsx')
3 dataset.columns = [ 'instruction' , 'output' ]
4
5 def generate_requests():
6     instructions = dataset[ 'instruction' ] # instructions du jeu de donnees
7     references = dataset[ 'output' ] # requetes references du jeu de donnees
8     candidates = [] # requetes candidates, generees par le modele
9     inference_times = [] # enregistrement des temps d'inference pour chaque
10    requete
11
12    for instruction in instructions:
13        start_time = time.time()
14        candidate_request = generate(instruction) # generation de la requete
15        end_time = time.time()
16
17        inference_time = end_time - start_time # calcul du temps d'inference
18        inference_times.append(inference_time) # insertion du temps d'inference
19
20        candidates.append(candidate_request) # insertion de la requete generee
21
22    return references , candidates , inference_times
```

Listing A.1 – Génération des requêtes Cypher par le modèle

## B Génération des tests unitaires

```
1 def create_unit_tests():
2     tests_suite = unittest.TestSuite()
3     for reference_query, candidate_query in zip(reference_queries,
4         candidate_queries):
4         test_function = lambda ref_q=reference_query, cand_q=candidate_query:
5             execute_cypher_comparison(ref_q, cand_q)
6         tests_suite.addTest(unittest.FunctionTestCase(test_function))
7
8     return tests_suite
9
10 def execute_cypher_comparison(reference_query, tested_query):
11     # connexion a la base de donnees
12     uri = "neo4j+s://a4c4a346.databases.neo4j.io:7687"
13     user = "neo4j"
14     password = "xxxxxxxx"
15     driver = GraphDatabase.driver(uri, auth=(user, password))
16
17     with driver.session() as session:
18         expected_result = session.run(cypher_query) # resultat de la requete de
19         # reference
20         tested_result = session.run(tested_query) # resultat de la requete
21         # generee par le modele
22         assert tested_result == expected_result, f"Test echoue: {tested_result}"
23         n'est pas égal à {expected_result}" # verification de l'egalite des resultats
```

Listing B.1 – Génération des tests unitaires

## C Évaluation des modèles

```
1 def evaluate_model_performance(references, candidates, inference_times):
2     # Execution des tests unitaires
3     unit_test_runner = unittest.TextTestRunner()
4     unit_tests_suite = create_unit_tests()
5     unit_test_results = unit_test_runner.run(unit_tests_suite)
6     total_unit_tests = unit_test_results.testsRun
7     unit_test_errors = len(unit_test_results.errors)
8     unit_test_failures = len(unit_test_results.failures)
9     successful_tests = total_unit_tests - unit_test_errors - unit_test_failures
10
11    # Analyse syntaxique (CodeBERTScore)
12    P, R, F1, F3 = score(candidates, references, lang='en', verbose=True)
13
14    # Temps d'inference
15    average_inference_time = sum(inference_times) / len(inference_times)
16
17    # Formattage du texte à enregistrer
18    output_text = f"Résultats du modèle - {model_path} - :\n"
19    output_text += f"Test unitaires - Nombre de tests : {total_tests} | éPassés : {correct} | Erreurs : {errors} | Echecs : {failures}\n"
20    output_text += f"Ressemblance syntaxique - ÉPrécision : {P.mean():.3f} | Rappel : {R.mean():.3f} | F1 score : {F1.mean():.3f} | F3 score : {F3.mean():.3f}\n"
21
22    output_text += f"Temps d'inference moyen : {average_inference_time:.3f} secondes\n"
23
24    # Ecriture du résultat dans le fichier
25    with open("results.txt", "w", encoding="utf-8") as file:
26        file.write(output_text)
27
28    return output_text
```

Listing C.1 – Evaluation du modèle

## D Charge de travail du projet

Étapes / Membre	Soulaïman	Serge	Jules
<b>État de l'art</b>			
Généralités sur les LLMs	20h	30h	20h
Méthodes d'apprentissage	05h	60h	00h
Méthodes d'optimisation	60h	00h	00h
Méthodes de comparaison	00h	00h	65h
<b>Cas d'application</b>	55h	60h	63h
<b>Gestion de projet</b>			
Rédaction du rapport	18h	16h	15h
Rédaction des CRs	3h	1h	04h30
<b>TOTAL</b>	161h	167h	167h30

## Résumé

Ce projet industriel, fruit d'une collaboration en Sopra Steria et Telecom Nancy, visait à explorer et à améliorer la capacité des modèles de langage à grande échelle (LLMs). Une première partie du projet a été consacrée à l'état de l'art, c'est-à-dire à la recherche sur les méthodes d'apprentissage, d'optimisation et de comparaison. La seconde partie a consisté à appliquer nos recherches sur un cas d'application. Ce dernier, choisi par Sopra Steria, a pour objectif de générer des requêtes Cypher, un langage de requête associé aux bases de données Neo4j. Face à la reconnaissance que les LLMs sont souvent peu entraînés sur des langages de niche tels que Cypher, une opportunité a été identifiée de par la forte marge de progression existante.

Nous avons ensuite sélectionné un modèle adapté, dans notre cas CodeLLama. Puis, nous l'avons entraîné avec un jeu de données spécifiquement conçu pour couvrir une variété de scénarios de requêtes Cypher et nous avons appliqué des méthodes d'optimisation identifiées lors de notre revue de l'état de l'art. Ce processus a été répété à plusieurs reprises afin d'obtenir des modèles avec entraînées et optimisées sur différentes méthodes, permettant ainsi de visualiser des différences de résultat selon les choix effectués.

Pour évaluer la performance des modèles, nous avons développé un système d'évaluation personnalisé, fondé sur des tests unitaires, ainsi que sur le temps d'inférence et sur une analyse syntaxique.

Ce projet a permis d'élargir notre compréhension des LLMs, en mettant en lumière de nouvelles perspectives et approches pour leur entraînement et optimisation.

**Mots-clés : intelligence artificielle, LLM, optimisation, Cypher**

## Abstract

This industrial project, a collaboration between Sopra Steria and Telecom Nancy, aimed to explore and enhance the capabilities of large language models (LLMs). The first part of the project was devoted to the state of the art, meaning research on learning methods, optimization, and comparison. The second part consisted of applying our research to a use case. This case, chosen by Sopra Steria, aimed to generate Cypher queries, a query language associated with Neo4j databases. Given that LLMs are often poorly trained on niche languages such as Cypher, an opportunity was identified due to the significant room for improvement.

We then selected a suitable model, in our case, codeLLama. Next, we trained it with a dataset specifically designed to cover a variety of Cypher query scenarios, and we applied optimization methods identified during our review of the state of the art. This process was repeated several times to obtain models that were trained and optimized using different methods, thereby allowing us to visualize differences in results based on the choices made.

To evaluate the performance of the models, we developed a customized evaluation system, based on unit tests, as well as on inference time and syntactic analysis.

This project broadened our understanding of LLMs, highlighting new perspectives and approaches for their training and optimization.

**Keywords : artificial intelligence, LLM, optimization, Cypher**