
Rapport final

FINE TUNING D'UN LLM POUR UN CAS D'USAGE SPÉCIFIQUE

PROJET INFO NUM 5

Vendredi 12 Avril 2024

Tuteur entreprise :
Thomas COSYN

Membres de l'équipe:
Ayman MOUMEN
Marouane MAAMAR
Samer LAHOUD

Contents

1 Remerciements :	3
2 Introduction :	4
3 Contexte et problématique :	4
3.1 Problématique initiale :	4
3.2 Echange avec OnePoint :	4
4 Objectifs fixés et cas d'usages identifiés :	5
4.1 Objectifs fixés :	5
4.2 Qu'est-ce que le SAS ?	5
4.3 Livrables attendus	6
4.4 Contraintes du projet :	6
4.5 Cas d'usages identifiés :	6
5 Analyse des points clés :	8
5.1 Code LLMs :	8
5.1.1 Evaluation des solutions déployées actuellement :	8
5.1.2 Evaluation des performances des modèles OpenSource :	9
5.2 Méthodes de Fine tuning :	9
5.2.1 L'architecture des LLM (Transformers)	9
5.2.2 Fine-tuning des LLM	11
5.2.3 État de l'Art des Méthodes de Fine-Tuning	11
5.2.4 Méthodes de Fine-Tuning : LoRA et PEFT	11
5.3 Méthodes d'évaluation :	15
5.3.1 Métrique traditionnelles	15
5.3.2 CodeBLEU	15
5.3.3 CodeBERTScore	17
5.3.4 GPT-4	20
5.3.5 Perspectives d'amélioration	22
6 Shéma d'ensemble et finalité prévue :	24
6.1 Shéma d'ensemble :	24
6.2 Diagramme de Gantt :	25
7 Résultats et discussion :	26
7.1 Paramètres fixés pour le fine-tuning :	26
7.2 Fine tuning sur le langage SAS :	27
7.2.1 Récupération et prétraitement du Dataset :	27
7.2.2 Le process de finetuning :	28
7.2.3 Résultats obtenus :	29
7.3 Fine tuning pour l'assistance à la résolution d'erreurs et débogage :	30
7.3.1 Le dataset utilisé :	30
7.3.2 Le process de Fine-Tuning:	31
7.3.3 Résultats et évaluation :	32

8 Conclusion et perspectives d'amélioration :	33
8.1 Problèmes rencontrés :	33
8.2 Perspectives d'amélioration :	34
8.3 Conclusion générale	34

1 Remerciements :

Nous tenons à exprimer nos sincères remerciements à plusieurs personnes qui ont grandement contribué à la réalisation de ce projet de finetuning d'un modèle de langage naturel (LLM) pour le langage SAS ainsi que pour le débogage et la résolution d'erreurs en Python.

Tout d'abord, nous adressons nos plus chaleureux remerciements à M. Thomas Cosyn, notre tuteur d'entreprise, pour sa précieuse guidance et son suivi attentif tout au long du projet. Sa disponibilité, ses conseils éclairés et les ressources matérielles et logicielles qu'il a mises à notre disposition, notamment les licences GPT-4, ont été essentielles pour mener à bien notre travail d'évaluation des modèles. Ses réunions régulières et son accompagnement nous ont permis de surmonter les obstacles rencontrés et d'avancer dans notre démarche avec confiance.

Nous exprimons également notre gratitude envers Mme Céline Hudelot, la responsable de la mention IA et membre du jury lors de la première soutenance de mi-projet. Ses retours constructifs sur notre avancement ainsi que les ressources supplémentaires qu'elle nous a fournies ont été d'une grande aide pour approfondir nos connaissances et améliorer la qualité de notre travail, notamment lors des difficultés rencontrées dans l'évaluation du LLM.

Enfin, nous remercions chaleureusement M. Paul Bizouard pour avoir facilité notre accès au cluster d'environnement de finetuning doté de cartes graphiques V100s, ressources cruciales pour mener à bien nos expérimentations de finetuning. Sa diligence et la rapidité de sa réponse à notre demande ont grandement contribué à la réussite de notre projet.

Nos remerciements s'étendent également à l'ensemble de nos professeurs de la mention IA à CentraleSupélec pour leur encadrement et leur enseignement de qualité qui ont posé les bases essentielles à la réussite de ce projet. Leurs conseils avisés et leur expertise ont été précieux tout au long de notre parcours académique et professionnel.

De plus, nous exprimons notre profonde gratitude envers l'équipe de One Point pour nous avoir accueillis dans leurs locaux et pour leur soutien constant tout au long du projet. Leur environnement propice à la recherche et au développement ainsi que leur collaboration précieuse ont été des facteurs déterminants dans la réussite de ce projet.

Nous sommes reconnaissants envers toutes ces personnes pour leur soutien indéfectible et leur contribution à la réussite de ce projet.

2 Introduction :

Le Groupe OnePoint occupe une place prépondérante dans le paysage des technologies de l'information et des services numériques en France. Fondée sur une vision novatrice et une expertise approfondie, cette entreprise s'est érigée en un acteur majeur du conseil, contribuant de manière significative à la transformation digitale des entreprises. Au cœur de ses activités, OnePoint déploie des solutions innovantes et des stratégies sur mesure pour répondre aux défis complexes de l'écosystème numérique contemporain.

3 Contexte et problématique :

3.1 Problématique initiale :

Initialement, le projet consiste en un **"fine-tuning" d'un modèle de langage LLM léger**, tel que distilGPT, **sur un ensemble de données spécifique**. L'objectif est d'explorer différentes techniques de "fine-tuning" et d'évaluer les performances du modèle ainsi ajusté. Les compétences développées au cours du projet devraient inclure la compréhension théorique des LLM, l'utilisation de bibliothèques open source telles que PEFT, LoRa, et évaluation des performances des modèles obtenus.

Le projet se déroule à 50% en présentiel dans les locaux de l'entreprise à Paris, avec des points de suivi hebdomadaires et des présentations régulières aux parties prenantes, notamment les leaders et partenaires de One Point. Le but était de nous donner l'opportunité de monter en compétence grâce à l'accompagnement par des experts de One Point, tout en découvrant le monde du conseil. En fin de projet, nous devons avoir créé un livrable démonstratif qui pourra être utilisé comme exemple concret pour les clients.

3.2 Echange avec OnePoint :

Afin de cibler efficacement le vaste projet de fine-tuning de LLM, il était impératif de définir un cas d'usage spécifique qui répondrait aux besoins et aux intérêts particuliers de OnePoint. Le défi résidait dans l'identification d'une problématique claire et pertinente, offrant ainsi une direction précise pour le projet.

Nous avons alors entrepris l'exploration de différents cas d'usage potentiels, en mettant l'accent sur la recherche d'une problématique engageante et susceptible d'intéresser OnePoint.

Dans le cadre de l'analyse approfondie des opportunités de projet, nous avons entrepris une démarche structurée en classifiant les idées par thème, dans un fichier excel ([Lien du document](#)), allant de l'utilisation des modèles de LLM dans le secteur financier (tel que l'utilisation d'un LLM pour classer la pertinence des cryptomonnaies en se fondant sur l'analyse des opinions exprimées sur les réseaux sociaux) à leur application dans le domaine de l'énergie. Notre objectif était de cerner des projets potentiellement innovants et alignés sur les domaines d'intérêt de OnePoint.

4 Objectifs fixés et cas d’usages identifiés :

4.1 Objectifs fixés :

L’objectif principal de ce projet était initialement d’approfondir la compréhension des techniques de fine-tuning appliquées aux modèles de génération de texte actuels tout en évaluant leurs performances. Après des échanges avec Thomas, notre encadrant de projet, de nouveaux objectifs ont été fixé, ces derniers étant plus pertinent pour les clients de OnePoint. Voici les principaux objectifs fixés :

- **Exploration de l’état de l’art des méthodes de fine-tuning** : Cet objectif est le premier qui a été fixé. Le but était d’examiner et comprendre les méthodes de fine-tuning existantes appliquées aux LLM légers, y compris les outils open source tels que PEFT, LoRa et QLoRa, afin de sélectionner les approches les plus adaptées au contexte du projet. Ce travail nous a permis de fixer le reste des objectifs.
- **Entraîner un LLM pour le code en langage SAS** : Lors de l’une de nos discussions avec Thomas, il nous a parlé d’un des clients de OnePoint qui rencontre des difficultés de migration de code SAS. C’est ainsi que résoudre ce problème est devenu l’un des objectifs de ce projet. Le but est de Fine Tuner un LLM sur une base de donnée de codes en langage SAS afin d’avoir un modèle qui performe au moins aussi bien que les modèles existant pour ce qui est de la génération de codes en SAS. Cet objectif était bien sûr accompagné de sous-objectifs, tels que trouver une base de données adaptée à ce travail. Après cela, un sous-objectif d’annoter cette base de données en utilisant le moins de ressources possible est survenu.
- **Entraîner un LLM pour aider au debugging de codes** : Un deuxième cas d’usage que Thomas a trouvé intéressant est l’entraînement d’un LLM pour accompagner l’utilisateur lors de tâches de débogage. Tout comme le modèle pour le langage SAS, il est attendu que ce modèle de débogage performe au moins aussi bien que les modèles existants pour cette tâche.
- **Rechercher et utiliser des méthodes d’évaluation appropriées aux modèles entraînés** : Vu la complexité de ces tâches, utiliser les métriques traditionnelles telles que BLEU ou ROUGE n’était pas une option. Le quatrième objectif était alors de faire des recherches des méthodes de l’art et trouver une ou des méthodes d’évaluation qui pourraient donner, de façon fiable et adaptée, un score aux modèles entraînés.

4.2 Qu’est-ce que le SAS ?

SAS (Statistical Analysis System) est un langage de programmation spécialisé largement utilisé dans le domaine de l’analyse statistique et de la manipulation de données. Créé initialement pour traiter et examiner de grandes quantités de données, SAS propose diverses fonctionnalités pour manipuler, analyser et représenter des données. Eléments distinctifs incluent une syntaxe claire et bien organisée, des fonctionnalités avancées pour le traitement de données, ainsi que des outils intégrés pour la modélisation statistique et la création de rapports.

Malgré de nombreuses options open source émergentes, SAS reste populaire pour diverses raisons. En premier lieu, il possède une histoire ancienne et une renommée bien établie dans le domaine de l’analyse de données, ce qui le rend très crédible aux yeux des entreprises et des institutions. En outre, SAS offre une gamme étendue de services logiciels allant de l’analyse de données à la Business Intelligence en incluant l’analyse prédictive, ce qui le rend séduisant pour les entreprises cherchant une solution intégrale.

Les applications de SAS sont variées et englobent une grande diversité d'industries et de domaines d'application. Il est fréquemment employé dans les domaines de la santé, des finances, du commerce de détail et des sciences sociales afin de réaliser des analyses de données, élaborer des modèles prédictifs, produire des rapports et des graphiques, et prendre des décisions stratégiques basées sur des données empiriques. Pour résumer, SAS demeure un instrument indispensable pour ceux requérant une plateforme robuste et fiable afin de gérer et analyser d'importantes quantités de données.

4.3 Livrables attendus

Un [repo GitHub](#) détaillé contenant les différents fichiers de code et notebooks utilisés pour le fine-tuning des modèles.

Le fichier README devrait être exhaustif. Il devrait comprendre une liste des outils utilisés, y compris la capacité mémoire du GPU, les temps d'inférence, les paramètres de fine-tuning, les hypothèses, et les contraintes spécifiées. En plus de cela, il devrait contenir un guide pas à pas du projet, expliquant comment exécuter le code et l'utiliser efficacement. Chaque étape du processus de fine-tuning devrait être détaillée, couvrant la méthodologie employée, le choix des paramètres, ainsi qu'une évaluation approfondie des résultats obtenus. Ceci fournira un exemple concret et utilisable pour les clients de OnePoint.

Cela permettra aux équipes de OnePoint de modifier et de mettre à jour plus facilement les modèles et les bases de données. Le repo Git pourrait éventuellement être adapté à d'autres cas d'utilisation à l'avenir.

4.4 Contraintes du projet :

Suite aux réunions avec le tuteur plusieurs contraintes ont été établies pour guider notre avancement dans le cadre du projet :

1. **Disponibilité et Utilisation des Ressources :** Un point essentiel discuté était la nécessité de planifier et de réserver en avance les ressources de l'environnement Mydocker doté d'un GPU puissant, fourni par l'école, pour garantir un accès fluide aux capacités de calcul et de stockage requises. Après avoir pris contact avec Paul Bizouard, architecte digital dans la DiSI de CentraleSupélec, nous avons obtenu un seul GPU, une Nvidia V100 avec une mémoire VRAM de 32 Go.
2. **Documentation Méthodologique :** La nécessité de documenter minutieusement chaque étape du processus de fine-tuning dans des notebooks a été soulignée. Cette documentation détaillée garantira la clarté et la reproductibilité des démarches entreprises.
3. **Communication et Présentation des Résultats :** La communication régulière avec le tuteur du projet et la préparation de présentations claires et concises pour les parties prenantes dès l'obtention de résultats significatifs ont été définies comme des impératifs pour assurer la validation continue et la transparence du projet.

4.5 Cas d'usages identifiés :

Suite à nos échanges avec OnePoint, comme déjà mentionné, Thomas s'est montré particulièrement enthousiaste vis-à-vis des projets liés à la digitalisation et à la programmation. Au travers de cette exploration, deux concepts ont émergé comme étant particulièrement pertinents pour les besoins de OnePoint.

- Tout d'abord, comme évoqué précédemment, les LLM entraînés pour le débogage du code, avec un accent particulier sur cette fonctionnalité, promettent d'optimiser significativement les processus de développement et de maintenance des codes. Cette approche présente ainsi des avantages tangibles pour les équipes travaillant sur des projets de digitalisation.
- Ensuite, comme mentionné plus haut, la deuxième idée majeure se concentre sur le fine-tuning de modèles de LLM spécifiques pour le langage SAS. Cette proposition découle des défis de migration de code SAS rencontrés par OnePoint dans un contexte client. En adaptant les LLM aux particularités du langage SAS, nous visons à surmonter les obstacles actuels et à améliorer la compatibilité entre différentes versions de code, facilitant ainsi la transition pour le client.

La section 'Analyse des points clés' justifiera en détail nos choix pour ces cas d'usages.

5 Analyse des points clés :

5.1 Code LLMs :

- Qu'est ce qu'un CODE LLM ?

Un code LLM est utilisé pour générer du code et aider à la programmation. Il peut comprendre et produire du code dans plusieurs langages de programmation, aidant les développeurs dans des tâches telles que la complétion de code, le débogage, la rédaction de documentation et même la génération de code basé sur des descriptions en langage naturel. Le code LLM est principalement entraîné sur un dataset de codes sur différents langages de programmation, Ce genre de modèles est formé sur de vastes ensembles de données et peuvent générer des extraits de code ou des programmes complets basés sur des instructions de saisie. Ils constituent une avancée significative dans le domaine du développement de logiciels, permettant aux programmeurs de travailler plus facilement et plus efficacement sur des projets complexes et réduisant les erreurs de codage.

5.1.1 Evaluation des solutions déployées actuellement :

Afin de justifier les cas d'usages identifiés et leurs pertinences pour le groupe OnePoint, nous avons effectué un état de l'art des solutions existantes en soulignant leurs avantages et leurs limitations. Il existe plusieurs interfaces déployées actuellement qui assistent les développeurs et programmeurs dans leurs projets, nous avons décidé de résumer les avantages et limitations des différentes interfaces dans le schéma ci-dessous :





				
Avantages	<ul style="list-style-type: none"> • Complète le code basé sur le contexte donné • Prise en charge de plusieurs langages 	<ul style="list-style-type: none"> • Génère un code basé sur les instructions de l'utilisateur • Prise en charge de plusieurs langages • Explique les erreurs rencontrés • Dispose d'une interface web pour tout utilisateur 	<ul style="list-style-type: none"> • Complète le code basé sur le contexte donné • Prise en charge de plusieurs langages 	<ul style="list-style-type: none"> • Complète le code basé sur le contexte donné • Explique les erreurs et effectue le "débogage" • Dispose d'une interface web pour tout utilisateur • Prise en charge de plusieurs langages
Limitations	<ul style="list-style-type: none"> • Besoin d'être intégré dans un IDE (pas pratique pour LINUX/UNIX) • Ne peut pas déboguer ou expliquer les erreurs • Quelques problèmes d'hallucinations et de codes mal générés 	<ul style="list-style-type: none"> • Faible face aux demandes d'algorithmes complexes • Beaucoup de problèmes d'hallucinations et de codes mal générés • Version préliminaire donc faible performance 	<ul style="list-style-type: none"> • Besoin d'être intégré dans un IDE (pas pratique pour LINUX/UNIX) • Ne peut pas déboguer ou expliquer les erreurs • Quelques problèmes d'hallucinations et de codes mal générés • Faible face aux demandes d'algorithmes complexes 	<ul style="list-style-type: none"> • Quelques problèmes d'hallucinations et de codes mal générés • Exclusif aux membres GPT+ • Rumeurs d'infractions de la confidentialité des données partagées

Figure 2: Avantages et limitations des interfaces assistantes dans le développement de code

Il existe bien sûr d'autres interfaces qui ont été déployées pour assister dans le développement de codes, nous avons choisi de comparer les interfaces les plus utilisées et les plus connues.

5.1.2 Evaluation des performances des modèles OpenSource :

Nous souhaitons rappeler dans cette section que l'une des contraintes auquel nous devons faire face est d'utiliser des modèles OpenSource pour nos cas d'usage. Durant nos recherches nous avons quelques modèles code LLMs évalués sur HumanEval et MBPP (des métriques d'évaluation pour juger la qualité et la justesse des codes python générés, la section 'Méthodes d'évaluation' abordera ça en détail) et nous avons pu conclure les résultats suivantes :

Model	HumanEval	MBPP
LLaMA-7B	10.5	17.7
LaMDA-137B	14.0	14.8
LLaMA-13B	15.8	22.0
CodeGen-16B-Multi	18.3	20.9
LLaMA-33B	21.7	30.2
CodeGeeX	22.9	24.4
LLaMA-65B	23.7	37.7
PaLM-540B	26.2	36.8
CodeGen-16B-Mono	29.3	35.3
StarCoderBase	30.4	49.0
code-cushman-001	33.5	45.9
Mistral-7B-Instruct	32.1	47.5
StarCoder	33.6	52.7
StarCoder-Prompted	40.8	49.5

Table 1: Comparaison des Modèles OpenSource et ClosedSource sur HumanEval et MBPP en pass@1 [1]

On remarque que StarCoder et StarCoder-Prompted se démarquent du lot avec des scores assez élevés, ce sont de bons modèles, toutefois, en raison du grand nombre de paramètres des modèles et de leur taille conséquente et vu l'environnement dont nous disposons (TESLA T4 16GB VRAM) sur Google Colab (avant qu'on y est accès à l'environnement MyDocker avec une Tesla V100 32GB VRAM), nous avons décidé de commencer des premiers tests avec Mistral-7B-Instruct qui a réussi à avoir des scores assez proches de StarCoder.

Un autre aspect à souligner, pour mettre en évidence l'intérêt du fine-tuning dans le débogage et l'assistance à la résolution d'erreurs, est que le modèle Starcoder a été entraîné uniquement sur l'auto-complétion et la génération de code, mais pas sur le débogage ni sur l'assistance à la résolution de problèmes. De plus, il n'a été fine-tuned que pour le langage Python, ce qui nous ouvre des possibilités de fine-tuning sur d'autres langages de programmation et de mesurer les performances correspondantes pour la continuité du projet.

5.2 Méthodes de Fine tuning :

5.2.1 L'architecture des LLM (Transformers)

Les Transformers, en particulier les modèles de langage, reposent sur une architecture dénommée "décodeur" (Decoder) pour effectuer des tâches de génération de texte. Cette architecture est fondamentale pour les tâches de modélisation du langage naturel et a été révolutionnaire dans l'avancée des performances des modèles de traitement du langage.

Le schéma du décodeur Transformer illustre un réseau de neurones récurrents qui, contrairement aux modèles récurrents traditionnels, utilise une attention multi-têtes. Cette attention permet au modèle d'apprendre des dépendances longue-distance et d'exploiter des informations contextuelles à partir de l'ensemble de la séquence d'entrée pour générer des prédictions de mots suivants. Cette architecture se compose de plusieurs couches d'attention, de normalisation, et de réseaux de neurones entièrement connectés, favorisant ainsi l'apprentissage de représentations abstraites du langage.

L'architecture des modèles "decoder-only" comme Mistral se compose généralement des éléments suivants :

1. **Positional Encoding** : Introduit pour informer le modèle sur la position des mots ou des éléments dans la séquence. Cela permet au modèle de comprendre la séquence dans son contexte.
2. **Multi-Head Self-Attention Layers** : Ces couches permettent au modèle de comprendre les relations entre différents éléments de la séquence en les comparant les uns aux autres. Elles permettent une compréhension contextuelle des mots ou des symboles dans une séquence.
3. **Couches Feedforward** : Ces couches sont responsables de la transformation non linéaire des représentations apprises par les couches d'attention.
4. **Output Layers** : Ces couches finales génèrent les prédictions pour la séquence en sortie, que ce soit pour de la traduction, de la génération de texte ou d'autres tâches similaires.

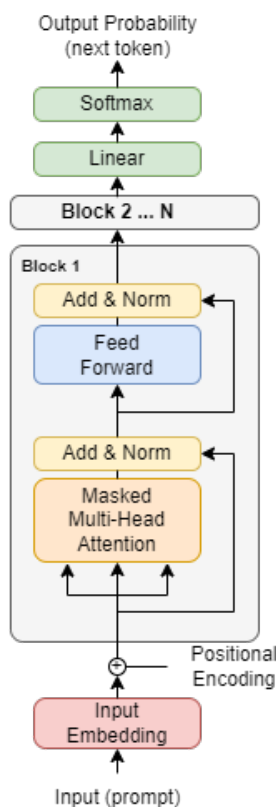


Figure 3: Architecture du transformer decoder

[2]

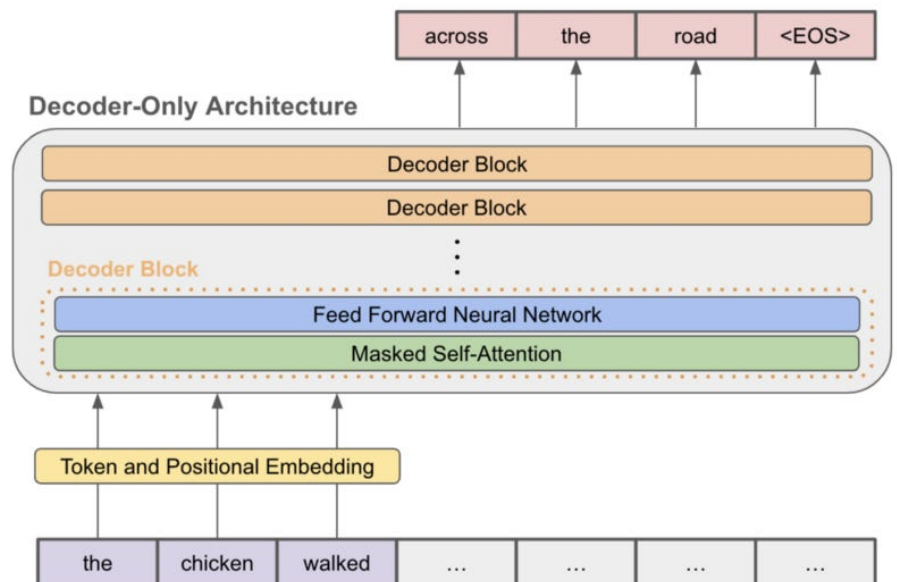


Figure 4: Architecture du transformer decoder-only [3]

5.2.2 Fine-tuning des LLM

Le fine-tuning des LLM représente l'adaptation d'un modèle pré-entraîné à des données ou à des tâches spécifiques. Pour un modèle de langage, cette procédure implique généralement l'ajustement des paramètres du modèle afin de mieux s'aligner avec des données cibles plus spécifiques. Ces données peuvent concerner un domaine textuel particulier ou des types spécifiques de tâches linguistiques telles que la traduction, la génération de texte ou la compréhension de texte.

Ce processus de fine-tuning offre la possibilité d'optimiser les performances d'un modèle pour des tâches spécifiques sans exiger un entraînement complet à partir de zéro. En ajustant minutieusement les poids pré-entraînés du modèle, cette technique permet une adaptation ciblée, favorisant ainsi des résultats améliorés et plus précis pour des contextes ou des objectifs linguistiques définis.

5.2.3 État de l'Art des Méthodes de Fine-Tuning

Le domaine du fine-tuning des modèles de langage a évolué vers une multitude de techniques innovantes, visant à améliorer les performances et l'adaptabilité des modèles pré-entraînés sur des tâches spécifiques. Parmi les avancées remarquables :

- **Adaptive Fine-Tuning** : Cette approche dynamise le processus de fine-tuning en adaptant les taux d'apprentissage pour différentes parties du modèle. Elle favorise ainsi un ajustement plus précis et efficace lors de l'entraînement sur des données spécifiques.
- **Few-Shot Learning** : Les modèles, tels que GPT-3, ont démontré une capacité étonnante à exécuter des tâches avec un très petit nombre d'exemples d'entraînement. Cette méthode repousse les limites traditionnelles du fine-tuning en exploitant des stratégies d'apprentissage à partir de très peu de données.
- **Knowledge Distillation** : Cette approche implique la compression de modèles pré-entraînés plus grands pour les adapter à des modèles plus petits et plus efficaces tout en conservant leurs performances. Cela les rend plus adaptés à des déploiements dans des environnements contraints en ressources.
- **Adapter Layers** : L'ajout de couches supplémentaires spécifiques à la tâche (adapter layers) permet de cibler des aspects particuliers d'une tâche sans altérer les couches pré-entraînées. Cela offre une flexibilité accrue dans l'adaptation des modèles à des tâches spécifiques.
- **Multitask Fine-Tuning** : Cette approche combine plusieurs tâches d'apprentissage pour améliorer la capacité du modèle à généraliser. Elle lui permet de mieux comprendre et synthétiser différentes structures linguistiques, renforçant ainsi sa polyvalence.
- **Learning Rate Schedules** : La variation des taux d'apprentissage tout au long du processus de fine-tuning peut grandement améliorer la convergence du modèle et sa capacité à apprendre des représentations plus précises, optimisant ainsi ses performances.

5.2.4 Méthodes de Fine-Tuning : LoRA et PEFT

Parmi les techniques de la méthode PEFT [5] on trouve :

- **Adapters** : Les adaptateurs consistent à ajouter des modules ou des couches spécifiques à une tâche à un modèle pré-entraîné sans modifier de manière extensive ses paramètres initiaux. Ces

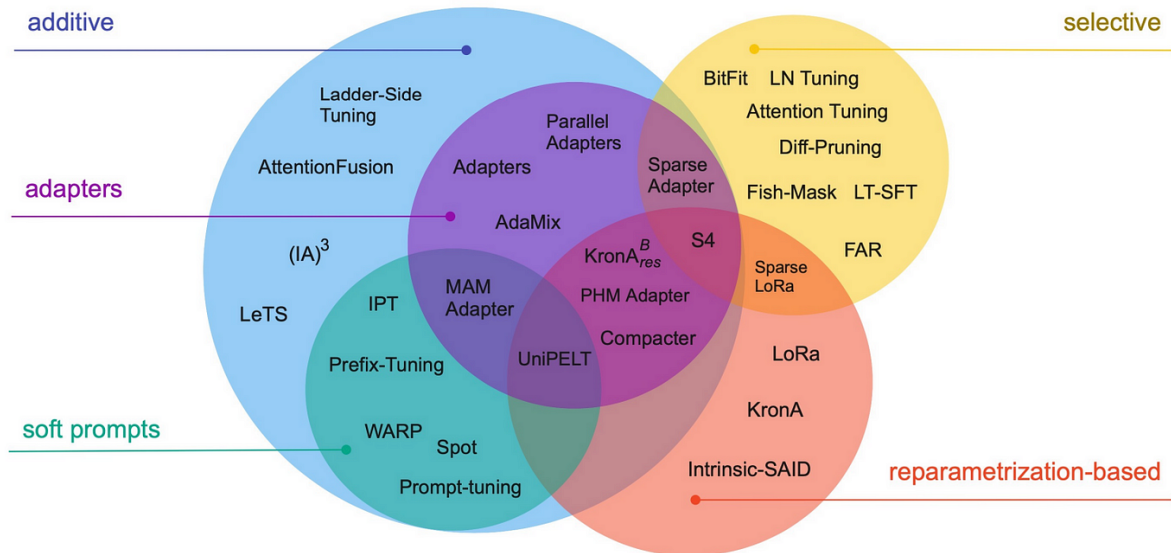


Figure 5: méthodes de PEFT [4]

adaptateurs permettent au modèle de se spécialiser pour une tâche spécifique tout en préservant les connaissances acquises lors de la pré-entraînement. Ils servent de modules compacts pouvant être insérés à différentes parties du réseau pour s'adapter à différentes tâches sans nécessiter de réentraînement approfondi.

- **Soft prompting** : Cette technique implique l'utilisation d'indications souples guidant l'attention du modèle vers certains aspects des données d'entrée. Au lieu d'indications rigides, les indications souples offrent une approche plus flexible en permettant au modèle de prêter doucement attention à des informations ou des indices spécifiques présents dans l'entrée, améliorant ainsi l'adaptabilité à différentes entrées.
- **Reparametrization** : La reparamétrisation modifie les paramètres du modèle lors du fine-tuning pour l'adapter à une tâche spécifique. Cela peut impliquer des modifications de certains aspects de l'architecture du modèle ou la modification de paramètres de manière spécifique à la tâche. Cela aide à adapter le comportement du modèle aux exigences de la tâche de fine-tuning sans altérer de manière extensive l'architecture initiale du modèle.
- **LoRA (Low Rank Attention)**
LoRA [6] est une méthode de fine-tuning qui vise à réduire la complexité des calculs d'attention dans les modèles de langage. Plutôt que de calculer l'attention entre toutes les paires de mots, LoRA utilise une approximation de rang faible pour réduire la complexité temporelle et spatiale des opérations d'attention. Cela permet de réduire la charge computationnelle lors du fine-tuning, tout en préservant les performances du modèle.

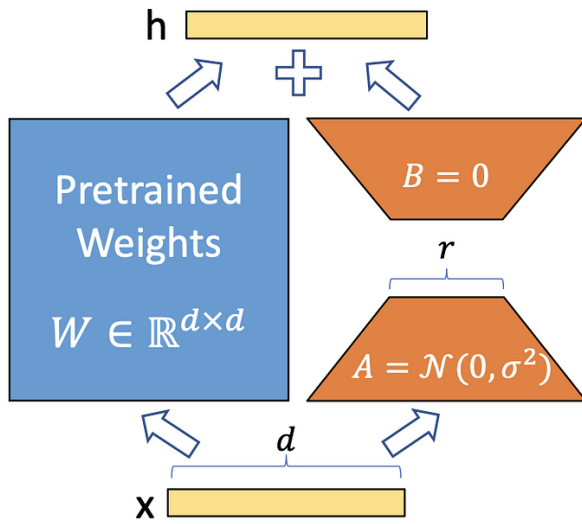


Figure 6: Lora 1 [6]

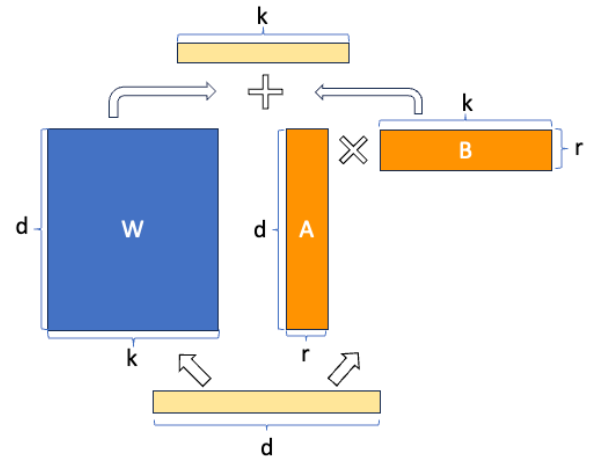


Figure 7: Lora 2 [6]

PEFT (Parameter Efficient Fine-Tuning)

PEFT se concentre sur l'optimisation des ressources lors du fine-tuning des modèles de langage. Cette méthode cherche à réduire la taille des modèles tout en maintenant leurs performances. Elle explore des techniques telles que la quantification de poids, la compression et la régularisation pour réduire le nombre de paramètres tout en préservant la capacité de généralisation du modèle.

l'équation de reparamétrisation :

$$W_0 + \Delta W = W_0 + BA \quad (1)$$

avec W_0 ($d \times k$), A ($d \times r$) et B ($r \times k$) et $r \ll d, k$

Parameter efficient fine-tuning (PEFT)

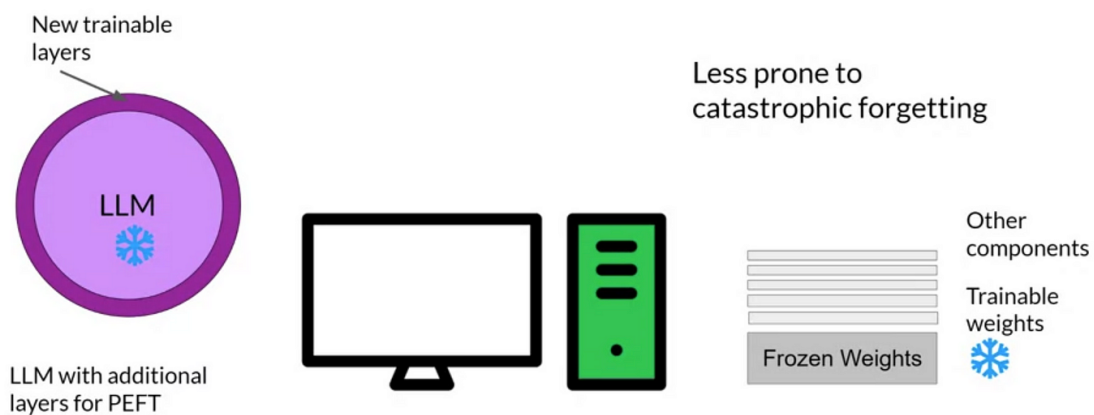


Figure 8: PEFT [7]

Avantages et Applications

Ces méthodes, LoRA et PEFT, offrent des avantages significatifs pour le fine-tuning des modèles de langage, notamment en termes d'efficacité computationnelle, de réduction des ressources nécessaires et de maintien des performances du modèle. Elles sont particulièrement utiles dans des environnements où

les contraintes de mémoire et de puissance de calcul sont importantes, tout en étant essentielles pour des déploiements à grande échelle.

Impact sur le Fine-Tuning

L'intégration de LoRA et PEFT dans le processus de fine-tuning ouvre la voie à des modèles plus efficaces et plus adaptés à une utilisation pratique. Ces techniques permettent d'exploiter les modèles de langage pré-entraînés tout en limitant la charge computationnelle et la consommation de ressources, ce qui les rend pertinents pour des applications variées, notamment dans les domaines nécessitant des modèles légers et rapides, comme le traitement de langage naturel sur des appareils mobiles ou des systèmes embarqués.

5.3 Méthodes d'évaluation :

L'évaluation des LLM constitue un domaine dynamique où diverses méthodes ont émergé pour mesurer la qualité et la pertinence de leurs sorties générées. Chaque méthode d'évaluation apporte son propre éclairage, offrant des perspectives distinctes sur la performance des LLM. Un grand nombre de métriques est disponible pour évaluer les modèles de langage naturel dans divers domaines, mais toutes ne sont pas appropriées pour évaluer les modèles entraînés sur le code informatique. Dans cette section, nous abordons la sélection des métriques les plus pertinentes pour évaluer les modèles de langage naturel entraînés sur le code, en mettant en lumière celles qui capturent le mieux la complexité et les particularités du langage informatique.

5.3.1 Métrique traditionnelles

1. **BLEU** : Le BLEU Score, développé en 2002 par Papineni et al., est une métrique essentielle pour évaluer la qualité des traductions automatiques. Le BLEU Score mesure la similarité en comparant les n-grammes (séquences de n mots) du texte généré avec ceux des solutions de référence. Alors que BLEU mesure efficacement la qualité de la traduction dans les langues naturelles, il néglige l'importance des mots-clés, de la structure syntaxique et de la correction sémantique cruciale dans l'évaluation de la synthèse de code[8].
2. **ROUGE** : Le ROUGE Score, a été développé pour évaluer la qualité des résumés automatiques de textes. L'évaluation par ROUGE, similaire au processus BLEU, consiste à générer un résumé automatique du texte source et à le comparer avec des résumés humains de référence, en évaluant le rappel des n-grammes et des séquences de mots pour attribuer des scores basés sur la couverture du contenu, utilisant notamment ROUGE-N pour les n-grammes et ROUGE-L pour la plus longue séquence commune. Tout comme la BLEU métrique BLEU, il n'est pas adapté aux modèles entraînés sur le code[9].

5.3.2 CodeBLEU

Aperçu :

La métrique CodeBLEU [10] est une mesure d'évaluation utilisée pour évaluer la qualité des codes sources générés par des modèles de génération automatique de code. Elle s'inspire de la métrique BLEU et prend en compte plusieurs aspects de la qualité du code généré, notamment la similarité des n-grammes entre le code prédit et le code de référence, la pondération de certains tokens (mots clés), la similarité syntaxique et l'analyse de la circulation des données dans le code.

Le processus d'évaluation de la métrique CodeBLEU implique la comparaison des sorties générées par un modèle de génération de code avec des références humaines. Plus précisément, il mesure à quel point le code généré est similaire au code de référence, en tenant compte à la fois de la syntaxe et de la sémantique du code.

Le CodeBLEU offre plusieurs avantages significatifs :

1. **Évaluation Objective** : Il fournit une mesure standardisée et objective de la qualité du code généré, permettant ainsi une comparaison précise entre différents modèles et approches.
2. **Aspect Multidimensionnel** : Contrairement à certaines mesures traditionnelles qui se concentrent uniquement sur la syntaxe ou la similarité des tokens, le CodeBLEU prend en compte divers aspects du code, tels que la syntaxe, la sémantique et la circulation des données.

Processus d'Évaluation :

CodeBLEU est une métrique d'évaluation complexe qui prend en compte plusieurs aspects de la qualité du code synthétisé. Tout d'abord, BLEU est calculé selon la méthode standard de Papineni et al. (2002), mesurant la correspondance des n-grammes entre la traduction candidate et les références. Cependant, pour tenir compte des particularités du langage de programmation, BLEU est étendu avec BLEUweight, une adaptation qui pondère les correspondances de n-grammes en attribuant des poids différents aux mots-clés et aux autres tokens.

Ensuite, la correspondance syntactique est évaluée à l'aide de l'appariement des structures d'arbre de syntaxe abstraite (AST) générées à partir du code candidat et des références. Cette composante, appelée Matchast, exploite la structure arborescente naturelle du code en comparant les sous-arbres des AST pour évaluer la similarité syntaxique.

Enfin, la correspondance sémantique est évaluée à l'aide de l'analyse des flux de données dans le code candidat et les références. Cette composante, appelée Matchdf, utilise des graphes de flux de données pour représenter les dépendances entre les variables dans le code. La similarité entre ces graphes permet d'évaluer la cohérence sémantique du code synthétisé.

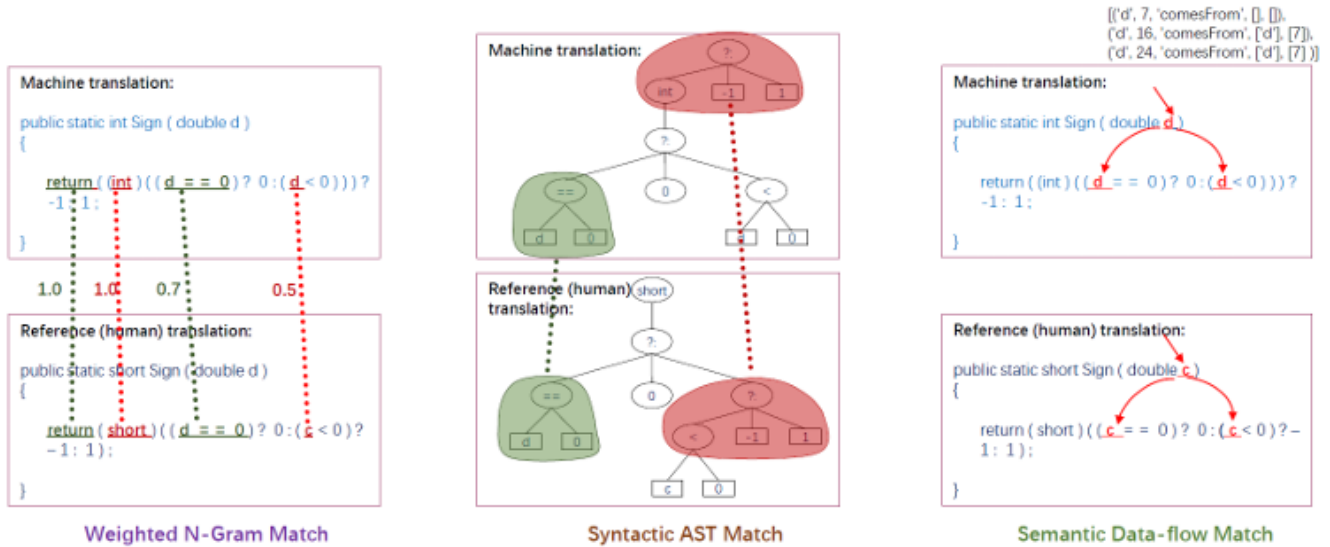
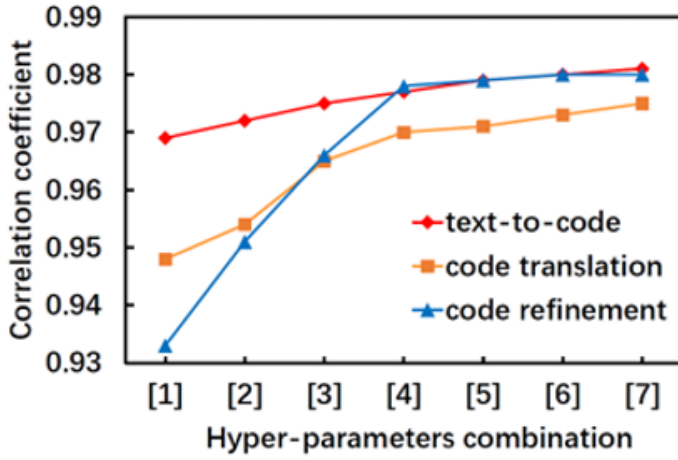


Figure 9: Weighted n-gram match, syntactic AST match, and semantic data-flow match [10]

$$\text{CodeBLEU Score} = \alpha \cdot \text{BLEU} + \beta \cdot \text{BLEU}_{\text{weight}} + \gamma \cdot \text{Match}_{\text{ast}} + \delta \cdot \text{Match}_{\text{df}} \quad (2)$$

Les coefficients de pondération α , β , γ et δ sont utilisés pour ajuster l'importance relative de chaque composante dans le calcul du score final de CodeBLEU. Ils sont généralement choisis empiriquement en fonction des besoins spécifiques de l'application et des performances désirées du modèle. Ils peuvent être optimisés en utilisant des techniques d'ajustement des hyperparamètres sur un ensemble de validation ou en se basant sur des connaissances préalables du domaine. Une approche courante consiste à effectuer

une recherche par grille ou une recherche aléatoire pour trouver les valeurs qui maximisent la performance de la métrique d'évaluation sur un ensemble de données de validation.



Combination	$\alpha, \beta, \gamma, \delta$
[1]	0.40, 0.40, 0.10, 0.10
[2]	0.35, 0.35, 0.15, 0.15
[3]	0.30, 0.30, 0.20, 0.20
[4]	0.25, 0.25, 0.25, 0.25
[5]	0.20, 0.20, 0.30, 0.30
[6]	0.15, 0.15, 0.35, 0.35
[7]	0.10, 0.10, 0.40, 0.40

Figure 10: Correlation coefficients between CodeBLEU and human scores with different hyper-parameters [10]

Chaque composante de CodeBLEU contribue à évaluer différents aspects de la qualité du code, de la syntaxe à la sémantique, en tenant compte des spécificités du langage de programmation. Le score final de CodeBLEU est calculé en combinant ces différentes mesures à l'aide de coefficients de pondération.

Défis:

CodeBLEU présente néanmoins quelques limites. En effet, les langages supportés par ce dernier sont Java, JavaScript, C#, PHP, C, C++, Python, Go, Ruby et Rust. Cependant, CodeBLEU ne prend pas en charge le langage SAS, ni même le langage SQL, qui lui est similaire. Pour contourner ce problème, nous avons choisi d'opter pour le langage Python. Ce dernier se démarque comme le choix le plus pertinent pour évaluer les scripts similaires à ceux de SAS en utilisant CodeBLEU, en raison de son utilisation répandue en science des données et en analyse, ainsi que de sa similarité en termes de facilité d'utilisation et de lisibilité. L'écosystème Python, avec ses bibliothèques dédiées à la manipulation, à l'analyse et à la visualisation des données, en fait un candidat solide pour les tâches couramment effectuées avec SAS.

5.3.3 CodeBERTScore

Aperçu :

CodeBERTScore, une métrique d'évaluation pour la génération de code, qui s'appuie sur BERTScore. Contrairement à BERTScore qui encode uniquement les tokens générés, CodeBERTScore encode également le langage naturel précédant le code généré, modélisant ainsi la cohérence entre le code généré et son contexte en langage naturel[11].

Processus d'Évaluation :

CodeBERT évalue un modèle de génération de code en comparant le code généré avec une implémentation de référence en utilisant une fonction de métrique $f : Y \times Y \rightarrow \mathbb{R}$, où Y représente l'ensemble des codes

possibles. Cette fonction de métrique compare le code généré \hat{y} avec le code de référence y^* . Une valeur plus élevée de $f(\hat{y}, y^*)$ indique que le code généré est plus précis par rapport au code de référence.

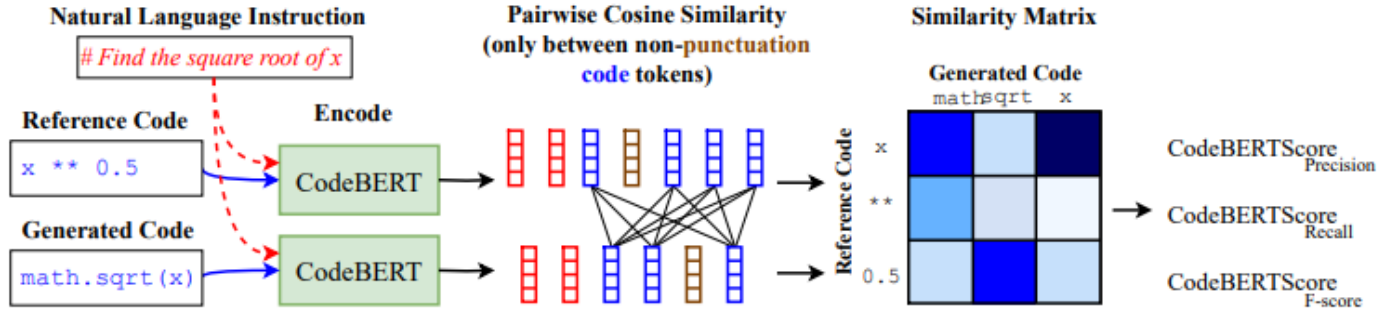


Figure 11: Diagram illustrating CodeBERTScore [11]

L'évaluation est effectuée en utilisant BERTScore, une méthode basée sur BERT pour l'évaluation des sorties de génération de code. BERTScore encode séparément la phrase candidate (la prédiction) et la phrase de référence (la vérité terrain) en utilisant un modèle basé sur BERT. Ensuite, il calcule la similarité cosinus entre chaque vecteur de la séquence candidate et chaque vecteur des séquences de référence. Sur la base de ces scores de similarité, BERTScore calcule la précision au niveau de la phrase en prenant le maximum des scores de similarité pour chaque vecteur candidat, et la rappel en prenant la moyenne des scores de similarité maximum pour chaque vecteur de référence. Le score final est le score F1, calculé comme la moyenne harmonique de la précision et du rappel.

$$\text{CodeBERTScoreP} = \frac{1}{|\hat{y}^{[m]}|} \sum_{\hat{y}_j \in \hat{y}^{[m]}} \max_{y_i^* \in y^{*[m^*]}} \text{sim}(y_i^*, \hat{y}_j) \quad (3)$$

$$\text{CodeBERTScoreR} = \frac{1}{|y^{*[m]}|} \sum_{y_i^* \in y^{*[m^*]}} \max_{\hat{y}_j \in \hat{y}^{[m]}} \text{sim}(y_i^*, \hat{y}_j) \quad (4)$$

$$\text{CodeBERTScoreF1} = \frac{2 \cdot \text{CodeBERTScoreP} \cdot \text{CodeBERTScoreR}}{\text{CodeBERTScoreP} + \text{CodeBERTScoreR}} \quad (5)$$

CodeBERTScore est une adaptation de BERTScore pour l'évaluation de la génération de code. Contrairement à BERTScore, il encode le contexte (l'instruction ou le commentaire en langage naturel) avec chaque fragment de code généré et de référence, mais n'utilise pas le contexte encodé dans le calcul final de similarité. Au lieu de calculer uniquement le score F1, CodeBERTScore calcule également le score F3 pour donner plus de poids au rappel par rapport à la précision. Le modèle sous-jacent utilisé pour l'encodage est un modèle similaire à BERT, pré-entraîné spécifiquement pour le langage de programmation, appelé CodeBERT.

$$\text{CodeBERTScoreF3} = \frac{10 \cdot \text{CodeBERTScoreP} \cdot \text{CodeBERTScoreR}}{9 \cdot \text{CodeBERTScoreP} + \text{CodeBERTScoreR}} \quad (6)$$

La représentation des jetons se fait en concaténant le contexte avec chaque fragment de code, puis en les encodant avec le modèle BERT-like. Ensuite, la similarité cosinus est calculée entre les jetons encodés des fragments de code générés et de référence pour obtenir les scores de similarité. Les scores de similarité sont utilisés pour calculer la précision, le rappel et les scores F1 et F3, en prenant le maximum à travers les lignes et les colonnes de la matrice de similarité, puis en faisant la moyenne.

Metric	Java		C++		Python		JavaScript	
	τ	r_s	τ	r_s	τ	r_s	τ	r_s
BLEU	.481	.361	.112	.301	.393	.352	.248	.343
CodeBLEU	.496	.324	.175	.201	.366	.326	.261	.299
ROUGE-1	.516	.318	.262	.260	.368	.334	.279	.280
ROUGE-2	.525	.315	.270	.273	.365	.322	.261	.292
ROUGE-L	.508	.344	.258	.288	.338	.350	.271	.293
METEOR	.558	.383	.301	.321	.418	.402	.324	.415
chrF	.532	.319	.319	.321	.394	.379	.302	.374
CrystalBLEU	.471	.273	.046	.095	.391	.309	.118	.059
CodeBERTScore	.553	.369	.327	.393	.422	.415	.319	.402

Figure 12: Correlation entre CodeBERT et d'autres métriques [11]

Actuellement les modèles CodeBERT publiés couvrent quatre langage de programmation : Java, Python, C, C++, et JavaScript. Cependant, les modèles spécifiques à chaque langue sont comparés à CodeBERT-base dans la figure suivante. En général, CodeBERT-base atteint des performances proches d'un modèle spécifique à chaque langue. Cependant, dans la plupart des expériences HumanEval et des mesures de corrélation, l'utilisation du modèle spécifique à chaque langue est bénéfique. Ces résultats montrent que les modèles spécifiques à chaque langue sont souvent préférés s'ils sont disponibles, mais que CodeBERT-base peut quand même fournir des performances proches même sans pré-entraînement spécifique à chaque langue.

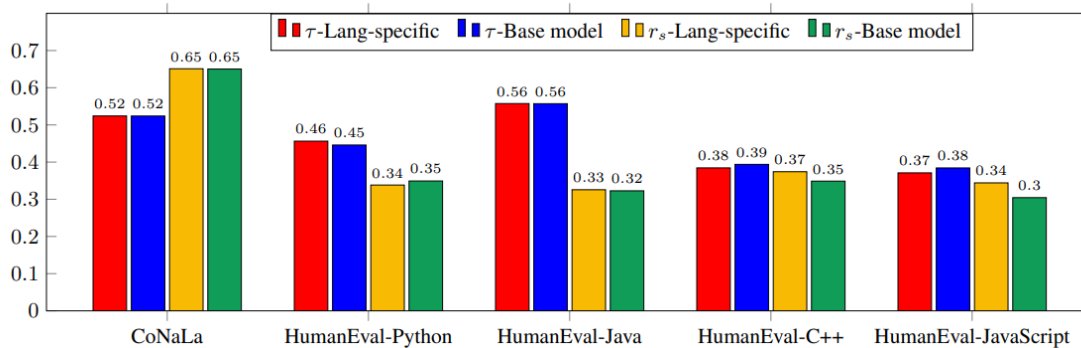


Figure 13: Le test de Kendall-Tau et de Spearman sur l'ensemble de développement de différents ensembles de données avec le modèle pré-entraîné spécifique à chaque langage (Langue spécifique) et avec le modèle de base CodeBERT (Modèle de base). [11]

Pour information, le test de Kendall Tau et le test de Spearman sont deux mesures de corrélation non paramétriques utilisées pour évaluer la relation entre deux ensembles de données. Ces tests sont souvent

utilisés lorsque les données ne sont pas distribuées de manière normale ou lorsque les relations entre les variables ne sont pas linéaires.

Le test de Kendall Tau mesure la similarité de l'ordre relatif des paires d'observations entre deux ensembles de données. Il calcule le coefficient de corrélation de Kendall Tau, qui varie de -1 à 1, où -1 indique une corrélation négative parfaite, 1 indique une corrélation positive parfaite, et 0 indique aucune corrélation.

Le test de Spearman est similaire au test de Kendall Tau, mais utilise les rangs des données plutôt que les données brutes. Il calcule le coefficient de corrélation de Spearman, qui mesure à quel point les rangs des données sont liés entre elles. Encore une fois, ce coefficient varie de -1 à 1, où -1 indique une corrélation négative parfaite, 1 indique une corrélation positive parfaite, et 0 indique aucune corrélation.

Ces tests sont largement utilisés dans différents domaines de la recherche pour évaluer les relations entre les variables lorsque les conditions requises par les tests paramétriques ne sont pas satisfaites.

Défis:

CodeBERTScore présente plusieurs limitations. Tout d'abord, il nécessite l'utilisation d'une carte graphique (GPU) pour calculer la métrique, tandis que des métriques traditionnelles telles que BLEU peuvent être calculées avec un processeur (CPU) seul. Cela ajoute une exigence matérielle à l'évaluation des modèles de code, alors que la plupart des approches précédentes sont moins exigeantes sur le plan computationnel, par exemple en comptant les n-grammes. Cependant, étant donné que l'entraînement et le test des modèles neuronaux nécessitent de toute façon une carte graphique, on peut raisonnablement supposer qu'une telle ressource est disponible.

De plus, les modèles de type BERT de base sont des encodeurs uniquement et non autorégressifs, ce qui signifie qu'ils n'exigent qu'un seul "passage avant" pour générer leurs prédictions, contrairement aux modèles encodeur-décodeur (par exemple, T5) et décodeurs uniquement (par exemple, GPT-3) qui doivent générer séquentiellement chaque token, nécessitant un passage avant pour chaque token de sortie. Ainsi, la consommation de temps supplémentaire par les modèles encodeurs uniquement, tels que BERT, est négligeable, surtout lors de l'évaluation de modèles générateurs NL à Code encodeur-décodeur ou décodeur uniquement.

Un autre point à considérer est que CodeBERTScore repose sur un modèle BERT de base solide, alors que des méthodes telles que BLEU n'ont pas beaucoup de "parties mobiles" ou d'hyperparamètres à régler. Cependant, il s'agit principalement d'un avantage, car CodeBERTScore peut être encore amélioré à l'avenir en utilisant des modèles de base plus robustes.

5.3.4 GPT-4

Aperçu :

Une autre méthode d'évaluation que nous avons utilisée est L'utilisation d'un autre LLM plus puissant (GPT-4) pour évaluer notre modèle (**Annexe 1 présente le prompt utilisé pour évaluation**). Cette nouvelle métrique est basée sur les capacités de génération des LLM tel que GPT-4. Ce dernier est comparé aux métriques traditionnelles d'évaluation dans un papier écrit en Octobre 2023 intitulé "Revisiting Instruction Fine-tuned Model Evaluation to Guide Industrial" [12].

Contrairement aux métriques traditionnelles qui se concentrent sur des aspects spécifiques tels que la correspondance avec des références ou des comparaisons de préférences humaines, cette nouvelle métrique évalue la qualité des réponses générées par les modèles de langage eux-mêmes.

Scorers	SUM	CAT	CIT	TFA
ROUGE	0.28	0.22	0.57	+513.9 %
BScore	0.21	0.22	0.13	+49.0 %
SBERT	0.25	0.29	0.43	+86.3 %
RM	0.20	0.28	0.29	-44%
GPT4	0.45	0.68	0.77	+2.1 %
GPT-3.5	0.42	-0.19	0.48	+9.5 %
Human	0.54	-	-	+12.0 %

Figure 14: Corrélation entre GPT, les scores humains et les métriques traditionnelles. [12]

Les avantages de cette méthode d'évaluation incluent une évaluation plus fiable de la performance des modèles IFT sur une variété de tâches génératives, ainsi qu'une meilleure adaptation aux exigences industrielles.

Processus d'Évaluation :

La méthode d'évaluation des modèles IFT (Instruction Fine-Tuning) repose sur l'utilisation de différentes métriques, telles que ROUGE, BScore, SBERT, et le modèle de récompense OpenAssistant Reward Model. Ces métriques sont utilisées pour évaluer la performance des modèles par rapport à des tâches génératives diverses. De plus, deux nouvelles exigences pour les métriques utilisées pour évaluer les modèles IFT sont introduites : la comparabilité entre les tâches (CAT) et l'agnosticisme vis-à-vis de la tâche et du format (TFA). CAT impose que les scores des métriques soient cohérents sur un ensemble diversifié de tâches génératives, tandis que TFA définit la nécessité pour les métriques de démontrer une robustesse aux variations dans les formats de sortie. En mettant en évidence les lacunes des métriques existantes pour répondre à CAT et TFA, il est démontré que l'utilisation de modèles de langage comme agents de notation est une alternative d'évaluation viable pour les modèles IFT.

Pour évaluer les performances des modèles IFT, les chercheurs utilisent cette nouvelle métrique en fournissant un stimulus d'entrée au modèle et en observant la réponse générée. Ensuite, ils attribuent une note à cette réponse en fonction de sa qualité perçue, sur une échelle de 0 à 10. Cette évaluation se fait sans référence à des réponses de qualité humaine ou à des préférences humaines, ce qui permet d'éliminer les biais potentiels introduits par l'utilisation de références externes.

Défis :

L'utilisation d'un LLM comme GPT pour évaluer un autre LLM présente certaines limites à prendre en compte. Tout d'abord, les LLM sont souvent entraînés sur des ensembles de données volumineux et divers, ce qui peut les rendre plus généraux mais aussi moins spécialisés dans des domaines spécifiques.

Cela signifie que leur capacité à évaluer précisément la performance d'autres LLM dans des tâches spécialisées ou spécifiques peut être limitée. De plus, l'utilisation d'un LLM pour évaluer un autre LLM peut introduire un certain degré de circularité dans l'évaluation, car les deux modèles peuvent partager des caractéristiques similaires en termes d'architecture et de formation, ce qui peut potentiellement biaiser les résultats de l'évaluation. Ainsi, bien que les LLM puissent fournir des évaluations utiles dans de nombreux cas, il est important de prendre en compte ces limitations lors de leur utilisation pour évaluer d'autres LLM.

5.3.5 Perspectives d'amélioration

Évaluation non supervisée avec correction en aller-retour :

Le concept de "round-trip correctness" (RTC) est une méthode d'évaluation non supervisée pour mesurer les capacités des modèles de langage de grande taille (LLM) dans le domaine du code informatique. RTC permet d'évaluer la capacité d'un modèle à générer du code à partir de descriptions en langage naturel, ainsi que sa capacité à éditer du code existant. La méthode d'évaluation repose sur la mesure de la similarité sémantique entre le code original et le code généré par le modèle [13].

Pour évaluer la capacité de synthèse de code, RTC utilise une tâche de synthèse de code en contexte qui ne nécessite pas de descriptions en langage naturel en entrée. Le modèle est invité à décrire de manière concise une région de code à l'aide de langage naturel, puis à générer du code implémentant cette description. Pour évaluer la capacité d'édition de code, RTC utilise une tâche d'édition de code où le modèle doit générer une description d'une modification apportée au code, puis implémenter cette modification.

Les métriques utilisées pour évaluer RTC comprennent la similarité exacte, mesurée par la correspondance exacte entre le code original et le code généré, ainsi que la métrique BLEU pour évaluer la similarité entre les descriptions en langage naturel. De plus, RTC utilise des oracles basés sur des tests pour évaluer la correction fonctionnelle du code généré.

Les avantages de RTC résident dans sa capacité à évaluer les modèles de manière non supervisée, sans nécessiter de jeux de données annotés par des humains. De plus, RTC permet d'évaluer la performance des modèles sur un large éventail de domaines de code, ce qui complète les benchmarks étroits existants et permet d'élargir les évaluations à de nouvelles tâches sans nécessiter d'annotations humaines coûteuses. Enfin, RTC est fortement corrélé avec les métriques existantes sur les benchmarks étroits, ce qui en fait une méthode d'évaluation fiable pour les LLM dans le domaine du code informatique.

Table 2. RTC_{pass} vs standard pass@1 metric.

	pass@1	RTC_{pass} (%)	L_M^{pass} (%)
HumanEval (Chen et al., 2021)			
PaLM 2-S	19.5	8.3	-0.2
PaLM 2-S+	29.3	10.6	3.3
PaLM 2-S*	37.6	18.3	9.8
Gemini Nano 2	33.4	18.9	3.7
Gemini Pro	67.7	28.5	12.0
ARCADE (Yin et al., 2022)			
PaLM 2-S	5.5	2.7	—
PaLM 2-S+	8.2	3.5	—
PaLM 2-S*	15.3	6.5	—
Gemini Nano 2	14.4	7.7	—
Gemini Pro	18.3	11.1	—

Figure 15: Correlation entre RTC et pass@1 [13]

Les limitations de l'évaluation de la correction en aller-retour (RTC) résident dans plusieurs aspects. Tout d'abord, la qualité de la mesure RTC dépend de la fonction de similarité utilisée. Une mesure faible de la similarité peut donner des résultats arbitraires. Ensuite, la performance des tâches avant et arrière est liée : si le modèle M ne peut pas fournir des échantillons pertinents, on ne peut pas mesurer la capacité du modèle arrière M^{-1} . Enfin, la RTC suppose que les modèles de langage sont correctement entraînés et ajustés aux instructions. Dans un contexte adversarial, un modèle avant peut ignorer l'instruction et répéter son entrée, tandis que le modèle arrière peut copier la sortie du modèle avant, obtenant ainsi une RTC parfaite. Bien que cela soit peu probable pour les modèles entraînés de manière conventionnelle, cela peut se produire si les modèles sont spécifiquement entraînés avec l'objectif de la RTC.

Développement d'unités de test pour une métrique d'évaluation plus robuste :

Dans la perspective d'améliorer la qualité et la fiabilité de notre travail, une piste d'amélioration serait le développement d'unités de test pour une métrique d'évaluation plus robuste. En introduisant des unités de test spécifiques, nous pourrions obtenir une évaluation plus approfondie et précise.

Par exemple, une unité de test pourrait évaluer la capacité du modèle à générer du code SAS conforme aux meilleures pratiques de programmation, telles que l'utilisation appropriée des macros, la gestion des erreurs et la cohérence de la syntaxe.

De plus, des unités de test pourraient être conçues pour évaluer la lisibilité et la maintenabilité du code généré. Par exemple, on pourrait mesurer la clarté du code en utilisant des métriques telles que la complexité cyclomatique ou la longueur des fonctions, afin de s'assurer que le code produit est facilement compréhensible et modifiable par les programmeurs SAS.

En intégrant ces unités de test dans le processus d'évaluation du modèle, nous pourrions obtenir une vision plus holistique de ses performances, permettant ainsi une identification plus précise des domaines où des améliorations sont nécessaires. En conséquence, cela contribuerait à renforcer la robustesse et la qualité des LLM.

6 Schéma d'ensemble et finalité prévue :

6.1 Schéma d'ensemble :

Notre solution, "OnePoint CodeTuner AI", offrira à OnePoint la capacité d'utiliser un modèle léger 'finetuned' sur des langages de programmation sans avoir recours à des API tierces (ce qui pourrait compromettre la confidentialité des données transmises à ces API de modèles). Pour notre projet, nous avons choisi de restreindre cette approche au finetuning sur le langage de programmation SAS ainsi que sur le débogage des erreurs. Notre travail de "pipeline" pourra ensuite être étendu à des modules LoRa pour les projets futurs.

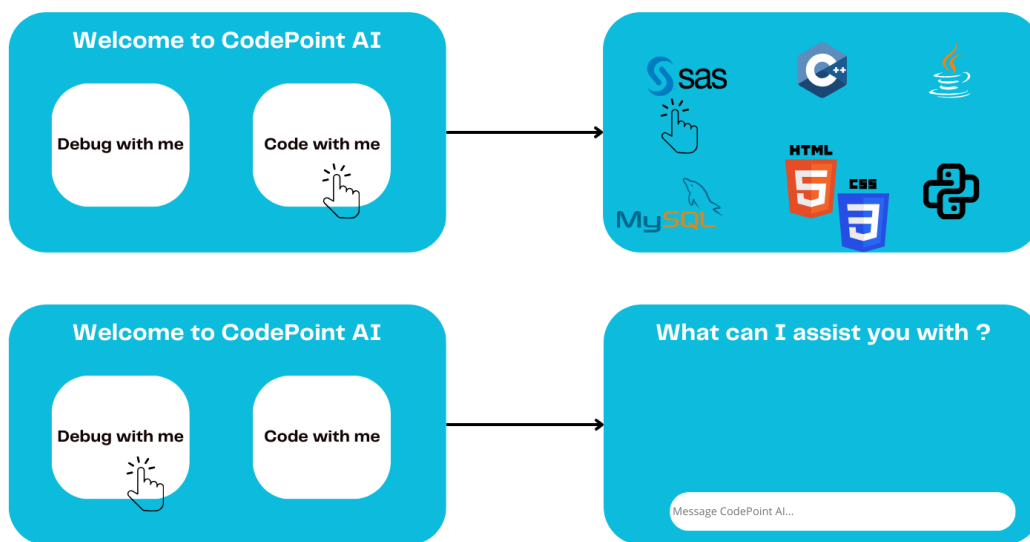


Figure 16: Schéma descriptif de la solution envisagée

Fonctionnalités Clés:

- **Personnalisation Profonde:** Les utilisateurs pourront personnaliser les modèles de langage pour qu'ils s'alignent mieux avec la syntaxe et les paradigmes spécifiques de leur choix de langage de programmation.
- **Interface Utilisateur Intuitive:** Inspirée de l'interface de GPT, notre interface sera conçue pour être familière aux développeurs, leur permettant de saisir des prompts et des requêtes de manière naturelle et directe.
- **Potentiel Commercial:** OnePoint CodeTuner AI sera développé avec une vision commerciale, en prévoyant la possibilité de le proposer à des utilisateurs externes et d'étendre ainsi notre impact sur le marché.
- **Documentation Exhaustive:** Le projet final serait accompagné d'une documentation détaillée, expliquant non seulement comment utiliser l'interface, mais aussi comment les modèles ont été ajustés et comment ils peuvent être déployés efficacement.
- **Prêt pour le Déploiement:** Nous nous engageons à fournir des modèles bien documentés et prêts à l'intégration dans les pipelines de développement de logiciels existants, assurant ainsi une transition en douceur et une mise en œuvre rapide.

6.2 Diagramme de Gantt :

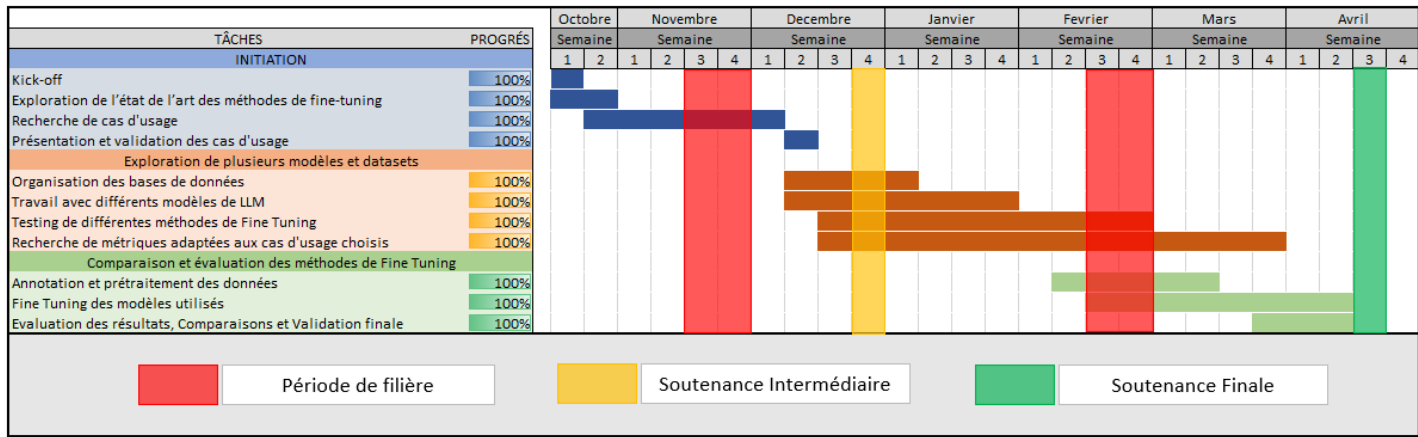


Figure 17: Gantt Chart du projet

7 Résultats et discussion :

Après avoir fait les recherches et la documentation sur l'état de l'art et techniques de FineTuning on a décidé de mettre en pratique ces techniques .

Pour répondre aux critères du cahier des charges, le modèle Mistral 7B Instruct a été sélectionné pour sa légèreté et ses performances parmi les Lightweight Language Model (LLM) decoders. Pour adapter ce modèle à notre projet, nous avons employé la méthode LoRA avec la bibliothèque PEFT. Ces méthodes ont été appliquées aux différents ensembles de données SAS et StackOverFlow, visant à améliorer les capacités du LLM en matière de génération de code et d'assistance à la résolution de problématiques de programmation.

Le processus de Fine tuning commence par un prétraitement et une préparation minutieuse des données d'entraînement pour s'adapter au format du modèle LLM utilisé. Ensuite, le modèle pré-entraîné a été chargé avec les bibliothèques PEFT, Lora et SFT Trainer pour l'entraînement.

Pendant l'entraînement, les fonctions de perte ont été affichées à chaque itération pour suivre la progression de l'apprentissage. Le modèle fine-tuned a été enregistré à la fin de cette phase. Les performances du modèle fine-tuned ont été évaluées par rapport au modèle de base lors de la phase de l'évaluation.

7.1 Paramètres fixés pour le fine-tuning :

Après plusieurs itérations expérimentales pour optimiser les paramètres d'entraînement au contexte de notre infrastructure, nous avons établi un ensemble de configurations qui ont favorisé les meilleures performances du modèle. Ces paramètres ont été uniformément adoptés pour l'intégralité du projet afin de garantir la cohérence et la répétabilité des résultats.

Paramètres d'entraînement adoptés :

- Quantization: Une quantification sur 4 bits a été activée (Load in 4 bit = True), permettant une réduction significative de l'empreinte mémoire sans altérer la précision du modèle.
- LoRA Parameters: Nous avons fixé lora-alpha à 16 et lora-r à 64, offrant un équilibre entre l'expressivité et la complexité du modèle. Le lora-dropout a été défini à 0,1 pour aider à prévenir le surajustement pendant l'entraînement.
- Bias: Nous avons choisi de ne pas intégrer de biais dans les ajustements de LoRA, signalé par la valeur bias = "None".
- Type de Tâche: Le type de tâche a été spécifié en tant que CAUSAL-LM, alignant ainsi le modèle pour la génération de texte séquentielle.
- Accumulation de Gradient: Une accumulation de gradient unitaire (Gradient Accumulation = 1) a été retenue pour maintenir une mise à jour directe des gradients pendant l'apprentissage.
- Taille de Batch: Une Batch Size = 1 a été déterminante pour gérer les contraintes de la mémoire disponible, bien que ce choix impose un compromis par rapport à la généralisation des apprentissages.
- Taux d'Apprentissage: Un taux d'apprentissage standard $2e^{-4}$.
- Nombre d'Epoch : 8 Epochs.

- **Fonction de Perte:** La nature de la tâche de génération de code SAS s'aligne avec l'utilisation de la Cross-Entropy Loss comme fonction de perte, car elle mesure efficacement la différence entre les distributions de probabilités prédites par le modèle et les distributions réelles des tokens dans les séquences de code SAS.

Ces paramètres ont été rigoureusement testés et ajustés pour assurer que le modèle affiné répondrait avec précision aux cas d'utilisation spécifiques de notre projet, tout en restant flexible et robuste face à diverses structures de données.

7.2 Fine tuning sur le langage SAS :

7.2.1 Récupération et prétraitement du Dataset :

Durant la phase initiale de notre projet, nous avons identifié un dataset prometteur connu sous le nom de "THE STACK", accessible via la plateforme HuggingFace ([voir The Stack - SAS Data](#)). Ce dataset se distingue par sa richesse, contenant des exemples de code source dans plus de 300 langages de programmation, y compris SAS, ce qui en faisait un candidat idéal pour notre objectif de fine-tuning d'un modèle capable de générer du code SAS.

Néanmoins, une exploration plus approfondie a révélé des limitations significatives pour notre use case spécifique. Les colonnes du dataset recensaient des informations telles que le dépôt d'origine du code, sa taille, et le langage de programmation correspondant, mais elles ne comportaient pas les annotations nécessaires - c'est-à-dire les instructions ou les demandes associées à chaque extrait de code.

Face à ce défi, notre équipe a entrepris un processus de nettoyage et de tokenisation du dataset en vue de son annotation. Ce travail préparatoire avait pour objectif d'estimer le coût de l'annotation, une étape cruciale compte tenu des ressources limitées du projet. Initialement, nous avions envisagé d'utiliser l'API Gemini pour cette tâche, mais des restrictions d'accès en Europe nous ont obligés à opter pour une alternative viable : l'API de GPT-3.5.

Le processus d'annotation s'est donc déroulé en plusieurs étapes. Après avoir tokenisé les extraits de code, nous avons utilisé l'API de GPT-3.5 pour annoter les données en batches, en veillant à assurer la précision et la pertinence des annotations par rapport aux extraits de code. Le détail de ces étapes de nettoyage et d'annotation est documenté dans notre dépôt de code ([voir le notebook associé](#)), où nous avons également fourni un résumé visuel des points clés sous forme de figure.

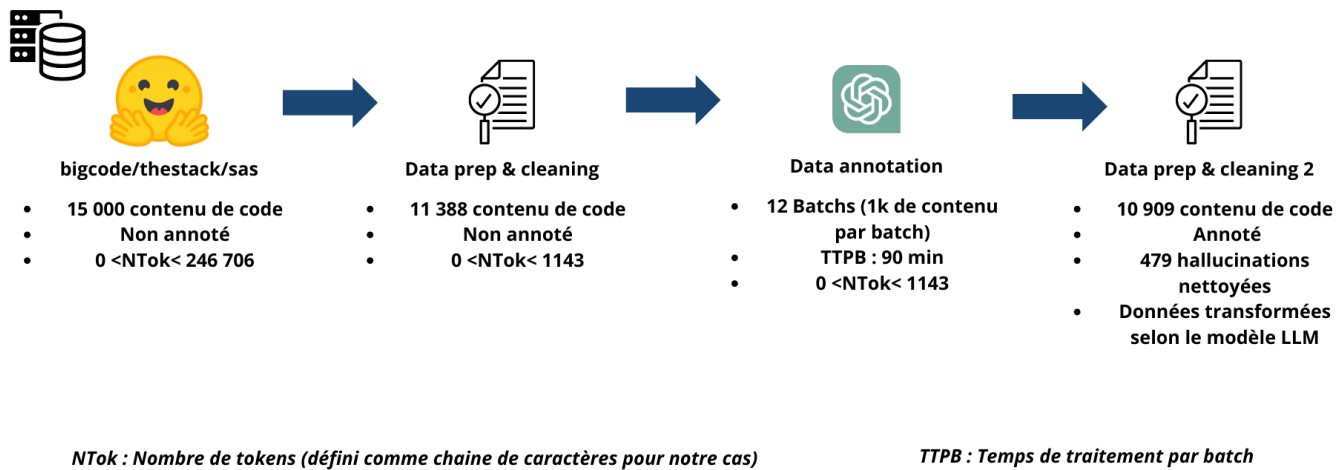


Figure 18: Pipeline de préparation des données

Suite au preprocessing et au formatting des données adaptées au langage du modèle de langage léger (LLM) ciblé pour le fine-tuning, et après avoir divisé notre dataset en 2 ensembles (training set : 8727 contenu de codes, testing set : 2182 contenu de codes) nous avons pris soin de conserver ce travail. L'ensemble des données prétraitées a été sauvegardé et mis à disposition sur la plateforme Hugging Face, assurant ainsi leur accessibilité pour une utilisation ultérieure. Les données peuvent être consultées et téléchargées via le lien suivant : ([MaamarM/SAS-training](#))

7.2.2 Le process de finetuning :

nous avons concentré nos efforts sur deux modèles distincts : Llama2-7b et Mistral-7b-Instruct-v02. Afin de gérer efficacement le temps et les ressources à disposition, chaque modèle a été affiné en utilisant 50% de notre ensemble de données d'entraînement, à l'exception de Llama qui a eu l'opportunité d'être fine-tuné sur la totalité du set.

Ce processus sélectif nous a permis d'observer une amélioration substantielle des performances, comme en témoigne la réduction de la fonction de perte de 1.6 à 0.21 au cours des itérations d'entraînement. La mention de cette décroissance est pertinente, car elle illustre non seulement l'efficacité de notre fine-tuning, mais sert également de baromètre quantitatif de l'apprentissage du modèle.

En ce qui concerne les paramètres d'entraînement, les spécifications établies précédemment ont été respectées, assurant ainsi une standardisation de l'entraînement pour une évaluation cohérente. Les résultats obtenus après cette phase d'ajustement fin du modèle sont récapitulés dans le tableau suivant, détaillant le temps consacré au fine-tuning pour chaque modèle : Tous les modèles fine tuned ont été

Modèle	Pourcentage du Training Set	Temps de Fine-tuning
Llama2-7b	50%	3.84 heures
Mistral-7b-Instruct-v02	50%	10.34 heures
Llama2-7b (Full Set)	100%	7.10 heures

Table 2: Temps de Fine-tuning pour chaque modèle

sauvegardés et déployés sur HuggingFace pour de futures utilisations ([voir lien des modèles finetuned](#)).

7.2.3 Résultats obtenus :

Nous avons procédé à l'évaluation des modèles (fine tuned sur 50 % du train set '4363 lignes') sur le set de test, sauf GPT-3.5 où on s'est servi de l'API. Pour une comparaison uniforme et contrôlée, seules les 100 premières lignes du test set ont été utilisées pour comparer le code source aux scripts générés par chaque modèle. Le tableau ci-dessous illustre le temps d'inférence nécessaire pour chaque modèle afin de générer 100 scripts basés sur les 100 premières instructions du test set : Ensuite, pour évaluer de manière

Modèle	Temps d'Inférence pour 100 Scripts
Llama2-7b	3 heures
Mistral-7b-Instruct-v02	3 heures
GPT-3.5	25 minutes

Table 3: Temps d'inférence pour générer 100 scripts

exhaustive la performance de nos modèles, nous les avons comparés entre eux (avant et après fine tuning) ainsi qu'au modèle GPT-3.5, en utilisant plusieurs métriques spécifiques au domaine de la génération de code. Ces métriques incluent CodeBLEU, CodeBERTScore (décomposé en Précision, Rappel, F1, et F3), et les réponses générées en utilisant GPT-4 pour un prompt spécifique disponible dans notre répertoire GitHub ([voir les notebooks d'évaluation](#)).

Le tableau comparatif ci-dessous présente les résultats obtenus :

Modèle	CodeBLEU	CBSP	CBSR	CBSF1	CBSF3	GPT-4
Llama2-7b	0.18	0.11	0.12	0.10	0.10	0.20
Llama2-7b finetuned	0.20	0.67	0.66	0.66	0.66	0.4
Mistral-7b-Instruct-v02	0.20	0.71	0.71	0.71	0.71	0.47
Mistral-7b-Instruct-v02 finetuned	0.21	0.76	0.73	0.74	0.73	0.60
GPT-3.5	0.23	0.75	0.75	0.74	0.75	0.71

Table 4: Comparaison des performances des modèles avant et après le fine-tuning sur différentes métriques

Les résultats présentés dans le tableau montrent des améliorations significatives dues au fine-tuning sur les modèles Llama2-7b et Mistral-7b-Instruct-v02, même en considérant les restrictions imposées lors de l'évaluation.

• Interprétation des Résultats :

- Amélioration significative après fine-tuning :

Llama2-7b montre une amélioration notable après fine-tuning, notamment dans les métriques CodeBERTScore (CBSP, CBSR, CBSF1, CBSF3) qui passent d'environ 0.10-0.12 à 0.66-0.67, et en CodeBLEU de 0.18 à 0.20. Cela démontre une amélioration de la précision et de la pertinence du code généré par rapport aux attentes initiales.

Mistral-7b-Instruct-v02 présente également une amélioration après fine-tuning, avec une augmentation dans toutes les métriques, y compris une montée en CodeBLEU de 0.20 à 0.21 et une augmentation substantielle dans les métriques CodeBERTScore et GPT-4.

- Comparaison avec GPT-3.5 :

Malgré le fine-tuning, GPT-3.5 montre des performances supérieures en termes de CodeBLEU et de scores CodeBERT. Cependant, les écarts se réduisent après le fine-tuning, en particulier pour Mistral-7b-Instruct-v02 finetuned qui atteint presque les niveaux de GPT-3.5 dans les métriques de CodeBERT et surpasse sa performance initiale en GPT-4. Le score de GPT-4 pour Mistral-7b-Instruct-v02 après fine-tuning (0.60) se rapproche de celui de GPT-3.5 (0.71), illustrant une nette amélioration de la capacité du modèle à générer des réponses pertinentes et contextuellement appropriées.

- Considérations sur les contraintes de l'évaluation :

Les performances de Mistral-7b-Instruct-v02 pourraient être légèrement sous-estimées en raison de la contrainte de la régénération des 1000 tokens, où il doit souvent régénérer l'instruction, ce qui peut impacter sa capacité à produire un code complet et optimisé dans ces conditions. Malgré l'utilisation de seulement 50% du dataset pour le fine-tuning, les améliorations observées attestent de l'efficacité de l'adaptation des modèles à leurs tâches spécifiques, suggérant que des résultats encore meilleurs pourraient être atteints avec un entraînement plus complet et moins restrictif.

- **Conclusion :**

Les résultats obtenus témoignent de la valeur du fine-tuning dans l'amélioration des capacités de génération de code des modèles de langage, même sous des contraintes non idéales. Ces données renforcent l'argument en faveur d'investissements continus dans le fine-tuning ciblé pour optimiser davantage les modèles LLM dans des applications spécifiques comme la génération de code SAS.

7.3 Fine tuning pour l'assistance à la résolution d'erreurs et débogage :

Dans cette section, nous aborderons le travail de fine-tuning réalisé avec les modèles Mistral-7b-instruct et Llama2 pour créer l'assistant chatbot dédié à l'aide aux développeurs dans la résolution d'erreurs et de problèmes de code Python et autres langages de programmation. Ce travail vise à améliorer la capacité des modèles choisis à fournir des réponses précises et utiles aux développeurs dans leurs problèmes de programmation, sans avoir à aller rechercher dans la documentation et passer beaucoup de temps pour trouver la solution.

7.3.1 Le dataset utilisé :

Avant de passer au finetuning, un travail de recherche a été réalisé pour trouver le dataset le plus adapté à notre objectif de finetuning. Nous avons réussi à extraire et préparer un large ensemble de données de Stack Overflow pour plusieurs questions et réponses sur des problèmes de programmation, disponible sur le lien suivant : [Dataset Stackoverflow](#), Le dataset contient 20M lignes et est composé de questions publiées sur [Stackoverflow](#) et plusieurs réponses pour chaque question ainsi que leurs scores (le score correspondant au nombre de Upvotes pour la réponse publiée).

On a fait un premier travail de preprocessing de ce dataset et de cleaning et normalisation afin de le rendre en un format adapté à notre modèle Mistral-7b-Instruct, on a aussi filtré le dataset pour ne garder que les réponses les plus pertinentes pour chaque question/thread. Cependant on a eu des problèmes d'insuffisance de mémoire lié à la taille du dataset avec notre environnement dédié au finetuning.

Nous avons donc décidé de filtrer notre dataset et garder uniquement les Question/Réponses en relation avec le langage Python. Nous avons ainsi réussi à réduire considérablement la taille de notre dataset à 17.5k lignes et nous l'avons ensuite divisé en train et test pour organiser notre pipeline de finetuning. A l'issue de ce travail nous avons réussi, en ajustant les hyperparamètres de finetuning pour notre modèle

Mistral-7b ainsi que Llama2 7b, à lancer notre process de finetuning et à générer des outputs pour notre dataset de test qui est composé de 100 lignes.

7.3.2 Le process de Fine-Tuning:

7.3.1 Fine-Tuning Initial sur un Sous-Ensemble du Dataset

Nous avons commencé par effectuer un premier cycle de fine-tuning sur un sous-ensemble représentant 60 % du dataset. Cette approche nous a permis d'explorer les hyperparamètres et d'optimiser les performances du modèle tout en respectant les contraintes de ressources computationnelles. En effet ce premier finetuning a duré 5h et nous a permis d'avoir un aperçu sur la qualité du finetuning et valider les étapes de preprocessing du dataset ainsi qu'avoir une idée sur le temps de training et d'inférence.

7.3.2 Optimisation des Hyperparamètres

Pendant cette phase, nous avons identifié et ajusté les hyperparamètres cruciaux tels que le learning rate, la taille du batch, le nombre d'epochs et expérimenté avec plusieurs différents modèles (Llama2, Gemma et plusieurs versions de Mistral 7b). L'objectif était d'améliorer la convergence du modèle et d'optimiser sa capacité à générer des réponses pertinentes, nous a également expérimenté avec plus de valeurs des hyperparamètres comme les paramètres α et r de Lora avec d'identifier les paramètres optimaux pour notre fine-tuning.

Voici les paramètres optimaux identifiés différents de ceux cités dans la partie 7.1 ainsi que leur effet sur le processus de fine-tuning :

- **Numéro d'Époques d'Entraînement** (`num_train_epochs = 8`) : Le nombre d'époques détermine le nombre de fois que l'ensemble des données d'entraînement est parcouru par le modèle. Un nombre optimal d'époques permet d'obtenir un équilibre entre l'apprentissage suffisant et l'évitement du surapprentissage, on a effectivement remarqué des situations de surapprentissage lorsqu'on dépasse 8 epochs.
- **Taille de Batch par Appareil** (`per_device_train_batch_size = 4`) : Il s'agit du nombre d'exemples d'entraînement traités simultanément sur chaque périphérique (GPU/TPU). Une taille de batch appropriée affecte la vitesse de convergence du modèle et l'utilisation efficace des ressources matérielles. On a donc réussi à accélérer le temps d'entraînement pour ce fine-tuning en prenant `batch-size = 4`
- **Pas d'Accumulation du Gradient** (`gradient_accumulation_steps = 2`) : Cette valeur contrôle le nombre de mises à jour du gradient avant la rétropropagation du gradient. L'accumulation du gradient peut aider à stabiliser l'entraînement, en particulier avec de petites tailles de batch.
- **Taux d'Apprentissage** (`learning_rate = 2e-3`) : Le taux d'apprentissage détermine la taille des pas effectués lors de la mise à jour des poids du modèle pendant l'entraînement. Un taux d'apprentissage optimal influence la vitesse de convergence et la qualité des gradients calculés.

7.3.3 Fine-Tuning sur l'Ensemble Complet du Dataset

Après avoir déterminé les hyperparamètres les plus efficaces, nous avons procédé à un second cycle de fine-tuning sur l'ensemble complet du dataset. Cette étape était essentielle pour maximiser les performances du modèle en exploitant l'intégralité des données disponibles. Ce travail a été appliqué

pour les 2 modèles dans les notebooks dédiés **Mistral_7b Finetuning** et **Llama2_7b Finetuning** disponibles sur GitHub :

- **Mistral_7b Finetuning** : [lien au notebook Fine-tuning Mistral-7b](#)
- **Llama2_7b Finetuning** : [lien au notebook Fine-tuning Llama2](#)

Le tableau récapitulatif suivant réument les durées de fine-tuning pour chacun des modèles :

Modèle	Pourcentage du Training Set	Temps de Fine-tuning
Mistral-7b-Instruct-v02	60%	1.43 heures
Mistral-7b-Instruct-v02	100%	2.51 heures
Llama2-7b	100%	2:48 heures

Table 5: Temps de Fine-tuning pour chaque modèle

Après fine-tuning, nous avons push les modèles fine-tuned aux repos Hugging face dédiés afin de faciliter l'importation et l'inférence de ces modèles :

- **Mistral-7b finetuned** : [Fine-tuned Mistral7b model](#)
- **Llama2 finetuned** : [Fine-tuned Llama2 model](#)

On présentera dans la section suivante les résultats de chacun de ces travaux

7.3.3 Résultats et évaluation :

Après la réalisation du Fine-tuning pour les différents modèles et les sauvegarder sur Hugging Face, nous avons fait un premier travail d'inférence de nos modèles fine-tuned nous avons testé avec différentes prompts pour évaluer qualitativement les performances de notre modèle dans les instuctions demandées (voir [notebook](#)). Notre modèle réponds aux instructions données mais pour juger judicieusement les performances de notre modèle, nous aurons besoin d'utiliser les métriques d'évaluation définies précédemment: Code-Bert, Code-Bleu et en utilisant GPT 4 comme LLM-Judge en adaptant le Prompt de scoring à la nature de notre modèle [\[14\]](#)

Nous avons donc générer des scripts, en rélisant une inférence sur notre dataset de test qui contient 100 instructions, qui vont servir de matériel d'évaluation pour chacun de nos modèles finetuned, Ce travail est élaboré dans [ce notebook](#)

Nous avons ensuite, comme fait pour le Use case SAS, évalué les réponses générées par nos modèles à l'aide des métriques d'évaluations définies et à l'aide du dataset test. Nous avons comparé les performances de chaque modèle avant et après Fine-tuning ainsi que les modèles entre eux.

Le tableau récapitulatif suivant résume les résultats obtenus pour chacune des métriques :

Le tableau comparatif ci-dessous présente les résultats obtenus qu'on peut retrouver dans les notebooks suivants : [Notebooks d'évaluation](#) :

Modèle	CodeBLEU	CBSP	CBSR	CBSF1	CBSF3	GPT-4
Mistral-7b-Instruct-v02 (base)	0.73	0.75	0.96	0.84	0.93	0.83
Mistral-7b-Instruct-v02 finetuned	0.83	0.76	0.98	0.88	0.96	0.9
Llama2-7b finetuned	0.72	0.76	0.98	0.85	0.95	0.82

Table 6: Comparaison des performances des modèles avant et après le fine-tuning sur différentes métriques

On remarque que après fine-tuning Mistral-7b améliore considérablement ses performances par rapport aux instructions du test set et il passe d'un score de 0.83 à un score de 0.9 avec la métrique d'évaluation par GPT 4. Alors que le modèle finetuned Llama2 a des performances comparables à Mistral-7b avant fine tuning. Cela peut être dû au fait que Llama2 est initialement moins performant que Mistral-7b sur ce type de tâche spécifique. Néanmoins, grâce au fine-tuning, nous avons pu observer une amélioration significative des performances de Llama2, le rapprochant ainsi des résultats obtenus avec Mistral-7b. Cette démonstration illustre l'efficacité du processus de fine-tuning pour adapter et améliorer les capacités des modèles de langage naturel selon des besoins spécifiques, même pour des modèles initialement moins performants.

8 Conclusion et perspectives d'amélioration :

8.1 Problèmes rencontrés :

Au cours du développement de notre projet de fine-tuning pour les modèles de langage dédiés : SAS et Débogage Python, notre équipe a rencontré plusieurs défis qui ont influencé notre progression et notre stratégie de déploiement. Voici les principales difficultés auxquelles nous avons dû faire face lors des différents étapes du projet :

- **Annotation du Dataset SAS** : Nous avons rencontré des difficultés lors de l'annotation du dataset SAS. Initialement, nous avons tenté d'utiliser Gemini pour annoter gratuitement, mais son accès était restreint en Europe, nécessitant l'utilisation d'un VPN. Cependant, le VPN présentait des problèmes de stabilité, interrompant souvent les processus d'annotation en cours.
- **Limitations de Ressources** : Nous avons été confrontés à des erreurs de type "CUDA out of memory", ce qui limitait nos capacités à utiliser des paramètres plus exigeants pour l'entraînement des modèles. Les sessions d'entraînement prenaient également beaucoup de temps (9 heures ou plus). Nous avons tenté de résoudre ce problème en explorant la possibilité d'obtenir un accès à un environnement avec plus de mémoire ou en combinant l'utilisation de plusieurs GPU pour exécuter des batches en parallèle, mais ces solutions n'étaient pas réalisables.
- **Exécution des Codes et Temps d'Inférence** : Le temps d'inférence des modèles fine-tuned était souvent long pour la génération de scripts SAS, en raison de la limitation de 1000 tokens_max imposée par le modèle. Certains scripts dépassaient cette limite, rendant les réponses incomplètes. Malgré cette contrainte, nous avons réussi à obtenir des scores très proches de ceux obtenus avec GPT-3.5.
- **Qualité du Dataset** : Le dataset n'était pas parfaitement et complètement nettoyé. Certaines annotations n'étaient pas correctes, car le modèle utilisé pour générer les annotations parfois réécrivait le code de manière incorrecte ou reformulait simplement les instructions, au lieu de fournir des annotations adéquates. Malheureusement, nous avons identifié ce problème tardivement, après avoir effectué le fine-tuning et l'évaluation, ce qui nous a empêchés de revenir en arrière et de refaire le preprocessing du dataset.
- **Problèmes de Prétraitement (Preprocessing)** : Les délimiteurs utilisés dans le prétraitement n'étaient pas toujours clairs pour le modèle à finetuner, ce qui générait des problèmes supplémentaires lors du preprocessing des données.

- **Choix de Métriques Pertinentes** : Nous avons eu des difficultés à trouver des métriques pertinentes pour évaluer nos modèles après le fine-tuning. Beaucoup de travail de recherche a été nécessaire pour sélectionner les métriques appropriées à utiliser dans notre évaluation.

8.2 Perspectives d'amélioration :

Les défis et les leçons apprises au cours de ce projet ouvrent la voie à plusieurs perspectives d'amélioration pour de futures recherches et développements dans le domaine du fine-tuning des modèles de langage naturel (LLM) pour le langage SAS et la résolution d'erreurs en Python :

- **Gestion des Ressources et de la Mémoire** : Investiguer des stratégies plus efficaces pour gérer les contraintes de ressources, telles que l'optimisation des paramètres du modèle pour réduire la consommation de mémoire et l'exploration de solutions d'infrastructure adaptées pour l'entraînement des modèles tels que l'utilisation de plusieurs GPUs en parallèle. Des ressources supplémentaires nous permettront d'explorer d'autres modèles plus performants que ceux qu'on a utilisés, tel que Gemma le récent LLM développé par Google.
- **Amélioration de la Qualité du Dataset** : Mettre en place des processus plus rigoureux de nettoyage et de validation du dataset afin d'identifier et éliminer les erreurs d'annotation et d'améliorer la cohérence des données utilisées pour le fine-tuning.
- **Développement de Métriques Spécifiques** : Continuer à explorer et à développer des métriques spécifiques adaptées à l'évaluation des modèles fine-tuned pour des tâches de résolution d'erreurs et de génération de scripts, en tenant compte des contraintes et des exigences particulières de chaque tâche et application de Finetuning.
- **Optimisation du Prétraitement (Preprocessing)** : Raffiner les techniques de prétraitement des données en identifiant et en clarifiant les formats appropriés de tokenisation pour garantir une meilleure compatibilité avec les modèles à entraîner et éviter les erreurs qu'on a identifiées lors de l'évaluation des modèles générateurs de script SAS.

8.3 Conclusion générale

Les recherches approfondies, la documentation minutieuse et la mise en pratique du fine-tuning sur des modèles de langage ont généré des améliorations significatives dans nos compétences de fine-tuning. Ces progrès se sont concrétisés par des résultats remarquables dans la génération de code Python, Finance, et même dans des langages de programmation moins connus comme OPL. L'utilisation de méthodes avancées telles que LoRA, Qlora et PEFT a joué un rôle essentiel en affinant les performances de nos modèles pour des tâches spécifiques. Ces résultats attestent de l'efficacité de nos techniques de fine-tuning sur des jeux de données plus simples et légers, démontrant ainsi notre capacité à atteindre les objectifs fixés avec notre tuteur de projet.

Le projet Infonum (Finetuning d'un LLM sur un cas d'usage précis) en choisissant comme cas d'usage le langage SAS et la résolution d'erreurs en Python a été une expérience enrichissante et formatrice. À travers ce projet, nous avons pu explorer les défis techniques et méthodologiques liés à l'adaptation et à l'optimisation des modèles de langage pour des tâches spécifiques dans le domaine du développement logiciel.

L'analyse des résultats obtenus lors du fine-tuning de Mistral-7b et Llama2 nous a permis de constater les

améliorations significatives apportées aux performances des modèles après adaptation sur notre dataset spécifique. Nous avons également identifié des défis majeurs tels que la gestion des ressources, l'importance de l'étape de prétraitement du dataset qui influe sur sa qualité et ainsi les performances du modèle et le choix des métriques appropriées pour l'évaluation des modèles fine-tuned.

Malgré les obstacles rencontrés, ce projet nous a permis d'acquérir de précieuses compétences en matière de prétraitement des données, d'entraînement des modèles et d'évaluation des performances. Nous avons également pris conscience de l'importance cruciale de la collaboration avec des experts et des mentors pour surmonter les difficultés techniques et méthodologiques.

En guise de perspectives futures, nous envisageons d'approfondir nos recherches sur l'optimisation du processus d'annotation, le développement de métriques spécifiques et l'exploration de nouvelles techniques de prétraitement des données pour améliorer la qualité des modèles fine-tuned dans des environnements complexes comme celui du langage SAS.

En conclusion, ce projet nous a permis d'explorer les frontières passionnantes du fine-tuning des LLM pour des applications pratiques et nous a motivés à poursuivre nos efforts pour améliorer les capacités des modèles de langage naturel dans des contextes spécifiques de développement logiciel. Nous remercions tous ceux qui ont contribué à la réussite de ce projet et sommes impatients de continuer à explorer ces domaines passionnants de recherche et d'innovation.

References

- [1] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim *et al.*, “Starcoder: may the source be with you!” *arXiv preprint arXiv:2305.06161*, 2023.
- [2] S. Flender, “Lora: Revolutionizing large language model adaptation without fine-tuning,” Feb 2024. [Online]. Available: <https://mlfrontiers.substack.com/p/lora-revolutionizing-large-language>
- [3] C. R. Wolfe, “Understanding the open pre-trained transformers (opt) library,” Jun 2022. [Online]. Available: <https://towardsdatascience.com/understanding-the-open-pre-trained-transformers-opt-library-193a29c14a15>
- [4] S. Kaushik, “Efficient model fine-tuning for llms: Understanding peft by implementation,” Jul 2023. [Online]. Available: <https://medium.com/@shivansh.kaushik/efficient-model-fine-tuning-for-llms-understanding-peft-by-implementation-fc4d5e985389>
- [5] V. Lialin, V. Deshpande, and A. Rumshisky, “Scaling down to scale up: A guide to parameter-efficient fine-tuning,” *arXiv preprint arXiv:2303.15647*, 2023.
- [6] E. J. Hu, Y. Shen, P. Wallis, Z. Allen-Zhu, Y. Li, S. Wang, L. Wang, and W. Chen, “Lora: Low-rank adaptation of large language models,” *arXiv preprint arXiv:2106.09685*, 2021.
- [7] J. Khetan, “Introducing parameter-efficient fine-tuning (peft),” Oct 2023. [Online]. Available: <https://medium.com/@jyotikhetan2/introducing-parameter-efficient-fine-tuning-peft-e1188943d7fc>
- [8] C. Callison-Burch, M. Osborne, and P. Koehn, “Re-evaluating the role of bleu in machine translation research,” in *11th conference of the european chapter of the association for computational linguistics*, 2006, pp. 249–256.
- [9] C.-Y. Lin, “Rouge: A package for automatic evaluation of summaries,” in *Text summarization branches out*, 2004, pp. 74–81.
- [10] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “Codebleu: a method for automatic evaluation of code synthesis,” *arXiv preprint arXiv:2009.10297*, 2020.
- [11] S. Zhou, U. Alon, S. Agarwal, and G. Neubig, “Codebertscore: Evaluating code generation with pretrained models of code,” *arXiv preprint arXiv:2302.05527*, 2023.
- [12] M. Faysse, G. Viaud, C. Hudelot, and P. Colombo, “Revisiting instruction fine-tuned model evaluation to guide industrial applications,” *arXiv preprint arXiv:2310.14103*, 2023.
- [13] M. Allamanis, S. Panthaplackel, and P. Yin, “Unsupervised evaluation of code llms with round-trip correctness,” *arXiv preprint arXiv:2402.08699*, 2024.
- [14] A. Roucher, “Using llm-as-a-judge for an automated and versatile evaluation - hugging face open-source ai cookbook,” 2024. [Online]. Available: https://huggingface.co/learn/cookbook/llm_judge

Annexes

Annexe 1 : Prompt considéré pour l'évaluation avec GPT 4

GPT Scoring Prompt:

You are a helpful assistant that helps evaluate the quality of a SAS script response to a given instruction.

Answer by awarding a score between 0 and 10 to the response, where 0 means the response is completely inappropriate and 10 means the response is very good.

A response that is acceptable should never be awarded less than 6 out of 10.

Answer based on the following criteria:

1. Is the script syntactically correct?
2. Does the script accurately perform the task as per the instruction?
3. Is the script efficiently written, using SAS best practices?
4. Is the script clear and maintainable?

Output format :

Rate the response to the following instruction: {instruction}

SAS Script: {script}

Annexe 2 : Problème de répétition de l'instruction en inférence pour Mistral Instruct

Input

```
# Ignore warnings
```

```
import logging
```

```
logging.set_verbosity(logging.CRITICAL)
```

```
# Prompt for the model
```

```
prompt = """
```

```
Create a new data table called (sports) which has only three columns from the (cars) data set: make, type, and msrp.
```

```
Keep only those observations where type is equal to (Sports) and msrp is greater than $100,000.
```

```
Then, create another data table called (suv) which has the same three columns. Keep only those observations where type is equal to (SUV) and msrp is greater than $60,000.
```

```
Finally, create a new data table called (expensive) by row-binding (sports) and (suv).
```

```
Print (expensive) to see the results.
```

```
"""
```

```
# Pipeline setup for text generation
```

```
from transformers import pipeline
```

```

model = 'your-model-name'
tokenizer = 'your-tokenizer-name'
pipe = pipeline(task="text-generation", model=model, tokenizer=tokenizer,
max_length=1000)
result = pipe(f"[INST]_{prompt}_[/INST]")
print(result[0]['generated_text'])

```

Output

[INST] Create a new data table called (sports) which has only three columns from the (cars) data set: make, type, and msrp. Keep only those observations where type is equal to (Sports) and msrp is greater than \$100,000. Then, create another data table called (suv) which has the same three columns. Keep only those observations where type is equal to (SUV) and msrp is greater than \$60,000. Finally, create a new data table called (expensive) by row-binding (sports) and (suv). Print (expensive) to see the results. [/INST]

To create the new data tables 'sports' and 'suv' based on the given conditions from the 'cars' dataset, you can use the following code:

```

# Create sports data table
sports <- subset(cars, type == "Sports" & msrp > 100000) %>%
  select(make, type, msrp)

# Create suv data table
suv <- subset(cars, type == "SUV" & msrp > 60000) %>%
  select(make, type, msrp)

# Create expensive data table by row-binding sports and suv
expensive <- rbind(sports, suv)

# Print the expensive data table
print(expensive)

```