UNIVERSIDADE FEDERAL DE PERNAMBUCO
CENTRO DE INFORMÁTICA
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

ANNA LUIZA CARACIOLO ALBUQUERQUE FERREIRA

FINE-TUNING DE LLMS PARA GERAÇÃO DE CÓDIGO MOJO

Recife
2024

ANNA LUIZA CARACIOLO ALBUQUERQUE FERREIRA

FINE-TUNING DE LLMS PARA GERAÇÃO DE CÓDIGO MOJO

Trabalho apresentado ao programa de
Graduação em Ciência da Computação do
Centro de Informática da Universidade Federal
de Pernambuco como requisito parcial para a
obtenção do grau de Bacharel em Ciência da
Computação.

Recife
2024

Ficha de identificação da obra elaborada pelo autor,
através do programa de geração automática do SIB/UFPE

Ferreira, Anna Luiza Caraciolo Albuquerque.
Fine-tuning de LLMs para geração de código Mojo / Anna Luiza Caraciolo Albuquerque Ferreira. - Recife, 2024.
33p. : il., tab.

Orientador(a): Luciano de Andrade Barbosa
Trabalho de Conclusão de Curso (Graduação) - Universidade Federal de Pernambuco, Centro de Informática, Ciências da Computação - Bacharelado, 2024.
Inclui referências.

1. geração de código. 2. IA gerativa. 3. Llama 2. 4. Code Llama. 5. fine-tuning. I. Barbosa, Luciano de Andrade. (Orientação). II. Título.

000 CDD (22.ed.)

ANNA LUIZA CARACIOLO ALBUQUERQUE FERREIRA

FINE-TUNING DE LLMS PARA GERAÇÃO DE CÓDIGO MOJO

Trabalho apresentado ao programa de Graduação em Ciência da Computação do Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação.

Apresentado em: 02/08/2024

Banca Examinadora:

Prof. Dr. Luciano Barbosa (Orientador)
Universidade Federal de Pernambuco

Prof. Dr. Vinicius Cardoso Garcia (Examinador Interno)
Universidade Federal de Pernambuco

Recife, 02 de agosto de 2024.

RESUMO

Esta pesquisa demonstra os processos de *fine-tuning* e de *prompt engineering* de modelos de aprendizagem de grande escala (LLMs) para geração de código na linguagem Mojo, desenvolvida pela Modular e lançada em versão inicial em 2023. Serão apresentadas as etapas de coleta e pré-processamento de dados, treinamento dos modelos e suas avaliações e apresentação dos dados gerados a partir dos modelos refinados e de um agente de chat comercial baseado em um modelo de linguagem de larga escala utilizando a estratégia de *few-shot learning*.

Palavras-chave: geração de código; IA gerativa; Llama 2; Code Llama; fine-tuning.

ABSTRACT

This research demonstrates the processes of fine-tuning and prompt engineering large language models (LLMs) for code generation in the language Mojo, developed by Modular and announced with a first version in 2023. It will be presented the steps of data collection and pre processing, fine-tuning of the chosen models and their evaluation and present the results generated by the fine-tuned models and by a commercial chat agent based on a large language model using the few-shot learning approach.

Keywords: code generation; generative AI; Llama 2; Code Llama; fine-tuning.

LISTA DE TABELAS

LISTA DE FIGURAS

LISTA DE CÓDIGOS

SUMÁRIO

1 Introdução

1.1 Motivação

LLMs tornaram-se um tópico popular por apresentarem capacidade interpretar linguagem natural e gerar resultados inéditos. Hoje, é possível ter acesso a modelos robustos para auxílio nas mais diversas tarefas, como gerar imagens a partir de linguagem natural, criar resumo de um texto, tradução de texto ou manter uma conversa com um agente de chat. Uma das áreas nas quais que também foram notados o impacto e o potencial de modelos de linguagem de grande escala é a de desenvolvimento de software. Hoje existem exemplos do uso de LLMs nesta área para as tarefas de geração de código, resumo de código e tradução de código [1].

Os modelos de linguagem de grande escala são treinados em bases com milhares de *tokens* e possuem bilhões de parâmetros, sendo esses os atributos responsáveis pelos resultados demonstrados de capacidade em interpretar linguagem natural. Entretanto, a escala desses modelos influencia diretamente nos recursos necessários para realizar seu o treinamento, consequentemente, o custo do uso de memória e processamento necessários para essa tarefa podem ser proibitivos para o desenvolvimento nessa área. A tarefa de fine-tuning demonstra o potencial de se fazer ajustes em LLMs para realização de tarefas específicas utilizando uma fração dos recursos necessários para desenvolver e treinar um modelo do zero, consequentemente reduzindo os custos para obter um modelo customizado. Desta forma, introduziremos nesta pesquisa o *fine-tuning* de modelos base para realizar uma tarefa específica, a de geração de código na linguagem Mojo. Além da tarefa de *fine-tuning* de modelos base, será feita a comparação dos modelos ajustados com a introdução de prompts em um modelo de linguagem de larga escala difundido e disponível comercialmente.

Para esta pesquisa, foram selecionadas três variantes da família de modelos Code Llama: o modelo base, Code Llama, o modelo especializado em Python, Code Llama Python, e o voltado para instruções, Code Llama Instruct. Estes três modelos são baseados no modelo Llama 2 e as variantes escolhidas são nas versões de 7 bilhões de parâmetros [2]. Já para a tarefa de aprendizagem baseada em contexto (*In-Context Learning*), o modelo escolhido foi o ChatGPT com o GPT-4.

Mojo é uma linguagem de programação baseada em Python desenvolvida pela empresa Modular. Python é a linguagem mais difundida atualmente para o desenvolvimento na área de aprendizagem de máquina. Entretanto, uma das principais barreiras encontradas hoje durante o desenvolvimento de modelos de aprendizagem de máquina em Python é o fato de ser uma linguagem lenta quando comparada com outras linguagens de programação populares [3]. Assim, a linguagem Mojo foi introduzida com a proposta de ter a usabilidade de Python e a performance de C, permitindo paralelismo, uso de tipos, controle o armazenamento, alocando valores diretamente em estruturas e meta-programação em tempo de compilação para escrever algoritmos independentes de hardware e reduzir redundância. A sua interoperabilidade e similaridade com

Python e a sua performance são as principais características que tornam Mojo uma linguagem de potencial promissor para a área de desenvolvimento de aprendizagem de máquina.

1.2 Objetivos

Esta pesquisa tem como objetivo utilizar modelos de aprendizagem de máquina (LLMs) para a geração de código na linguagem Mojo a partir de linguagem natural e trechos de código da linguagem. O escopo pricipal da pesquisa é o *fine-tuning* de LLMs de código para geração de código Mojo, ao final, é feita uma comparação de códigos Mojo gerados a partir *In-Context Learning* pelo ChatGPT.

2 Conceitos básicos

2.1 Large language models (LLMs)

Segundo Humza *et al.* (2023), a criação e propagação de modelos de linguagem de larga escala observados recentemente foram impulsionadas pelo uso de Transformers, maior capacidade computacional e pela disponibilidade de dados para treinamento dos modelos [4]. Nos últimos anos foram lançados para o público diversos LLMs propostos a realizar diferentes tarefas, como geração de imagem, processamento de linguagem natural e processamento de voz. Podemos destacar no escopo de processamento de linguagem natural o lançamento do ChatGPT em 2022 pela OpenAI, os modelos Llama, lançados pela Meta e o Gemini, introduzido pela Google em 2023.

2.1.1 Modelos de geração de código

LLMs de código são treinados em bases de larga escala contendo majoritariamente trechos de código, enquanto LLMs generalistas são treinadas em bases que contém textos em linguagem natural e que também podem conter trechos de códigos.[5] A capacidade de LLMs de transformarem instruções em linguagem natural em código fonte impulsionou o lançamento de ferramentas de IA para suporte ao desenvolvimento de código, como o GitHub Copilot e Amazon CodeWhisperer. Os assistentes de código têm se mostrado promissores para a otimização da rotina de desenvolvimento, oferecendo funcionalidades de conclusão de código, sugestão de melhoria e comentário de código.

A família de modelos Code Llama [2], introduzido em 2023, foi lançada em versões de diferentes tamanhos e com propostas de realização de tarefas de domínio específico. Os três modelos apresentados foram o Code Llama, Code Llama - Python e o Code Llama - Instruct. O Code Llama tem a capacidade de completar trechos de código e foi treinado com o objetivo de *code infilling*. O Code Llama - Python é a variante especializada em Python e voltada para a tarefa de completar código. A variante Code Llama - Instruct tem o objetivo de seguir instruções humanas.

2.1.2 Arquitetura

As arquiteturas de LLMs são em sua maioria baseadas na arquitetura de Transformers. A arquitetura original dos Transformers [6] é do tipo *encoder-decoder*, mas algumas LLMs gerativas utilizam a arquitetura *decoder-only* com objetivo de geração de resultados inéditos para o modelo. Os blocos *encoder* são responsáveis por transformar as sequências de entrada em uma outra representação da sequência em forma fixa. Já o *decoder* transforma a saída do *encoder* na sequência de saída do modelo. [7]

Os modelos Code Llama utilizam como modelo base o Llama 2, que por sua vez foi

derivado do Llama 1 baseado em um transformer auto-regressivo [8]. Um transformer auto-regressivo é capaz de realizar novas inferências baseado em inferências anteriores. O Llama 2 é um refinamento do modelo Llama 1 para o qual foram aplicadas as técnicas de *supervised fine-tuning* e *reinforcement learning with human feedback*.

2.1.3 Fine-tuning

Segundo Humza *et al.* (2023), foram desenvolvidas novas formas de realizar refinamento de modelos com menor custo computacional em comparação a fazer um treinamento completo de um modelo, sendo o conjunto destas técnicas chamado *parameter-efficient fine-tuning* [4]. Algumas destas técnicas são o uso de *adapters*, *prompt tuning* e *prefix tuning*. Adapters introduzem o conceito de módulos entre as camadas de um dado modelo e, durante o *fine-tuning*, apenas os parâmetros o adapter são atualizados e os pesos do modelo pré-treinado são mantidos [9].

De acordo com a pesquisa *Exploring Parameter-Efficient Fine-Tuning Techniques for Code Generation with Large Language Models*, foi observado que o método (LoRA) *Low-rank adaptation* obtém melhores resultados para LLMs quando comparado ao retreinamento do modelo inteiro e ao In-Context Learning (ICL) [10]. O método LoRa introduz adapters de baixa ordem às matrizes de peso pré-treinadas [11], reduzindo o número de parâmetros usados durante o treinamento e, consequentemente, acelerando o tempo de processamento. Por estes motivos, a técnica de *Low-rank adaptation* tem sido difundida para a tarefa de refinamento de LLMs.

2.2 Linguagem Mojo

A linguagem Mojo foi lançada em 2023 e oferece o Mojo Playground, ambiente virtual para desenvolvimento de programas Mojo, um SDK, documentação para uso da linguagem e uma comunidade crescente. Mojo é uma linguagem voltada para a área de aprendizagem de máquina e sitemas da alta performance e se propõe a ser mais rápido que Python. Para alcançar este objetivo, o seu desenvolvimento teve como duas das principais missões incluir inovações em componentes internos de compilador e prover suporte para aceleradores atuais e emergentes. [12]

Até o momento desta pesquisa, Mojo provê uma biblioteca padrão que implemeta os tipos e métodos básicos presentes em linguagens de programação de alto nível. Mojo implementa na sua bibliteca padrão recursos de matemática básicos, classes para representação de inteiros, booleanos, *strings*, vetores, tensores e *decorators*.

2.3 Métricas de avaliação

As métricas escolhidas para análise foram ROUGE [13] e FrugalScore [14].

2.3.1 ROUGE

A métrica ROUGE (*Recall-Oriented Understudy for Gisting Evaluation*) foi desenvolvida a partir da necessidade de medir a qualidade de um resumo de texto de forma automatizada, tendo em vista que esta tarefa consome tempo e esforço humano consideráveis quando feitas manualmente [13].

O ROUGE realiza uma comparação entre o resumo de texto candidato e resumos de referência. Existem variantes da medida de rouge score: O ROUGE-N é baseado na medida de *recall* e o *n* signifca o tamanho do n-grama sendo considerado para comparação de similaridade entre candidato e refêrencia. O ROUGE-L compara a subsequência comum mais longa (*longest common subsequence*) entre o resumo candidato e sua referência.

2.3.2 FrugalScore

O FrugalScore é uma métrica baseada em métricas para modelos de larga escala, como BERTScore, para avaliação de geração de linguagem natural (NLG) com a proposta de ser leve e performática. Mesmo sendo originalmente dedicada para avaliação de NLG, o seu cálculo é baseado em análise de similaridade de sequências e por esse motivo foi possível obter bons resultados de FrugalScore para comparação dos códigos gerados e suas respectivas referências.

O desenvolvimento do FrugalScore pode ser dividido principalmente em duas etapas: a primeira consiste na criação de um conjunto de dados de sequências anotadas com as métricas a serem aprendidas. A segunda etapa é a de treinamento de uma versão pequena de um modelo de linguagem sobre o conjunto de dados gerado na primeira etapa.

Segundo Eddine *et al.*(2021), o FrugalScore consegue manter em média 96.8% da perfomance da métrica original sob o qual foi baseado, sendo em média 24 vezes mais rápido.

3 Metodologia

Esta pesquisa foi realizada em cinco etapas: a coleta de amostras de código em Mojo, o pré-processamento do conjunto de dados, a verificação de códigos gerados pelos modelos originais antes do refinamento, *fine-tuning* das LLMs selecionadas previamente e avaliação dos modelos:

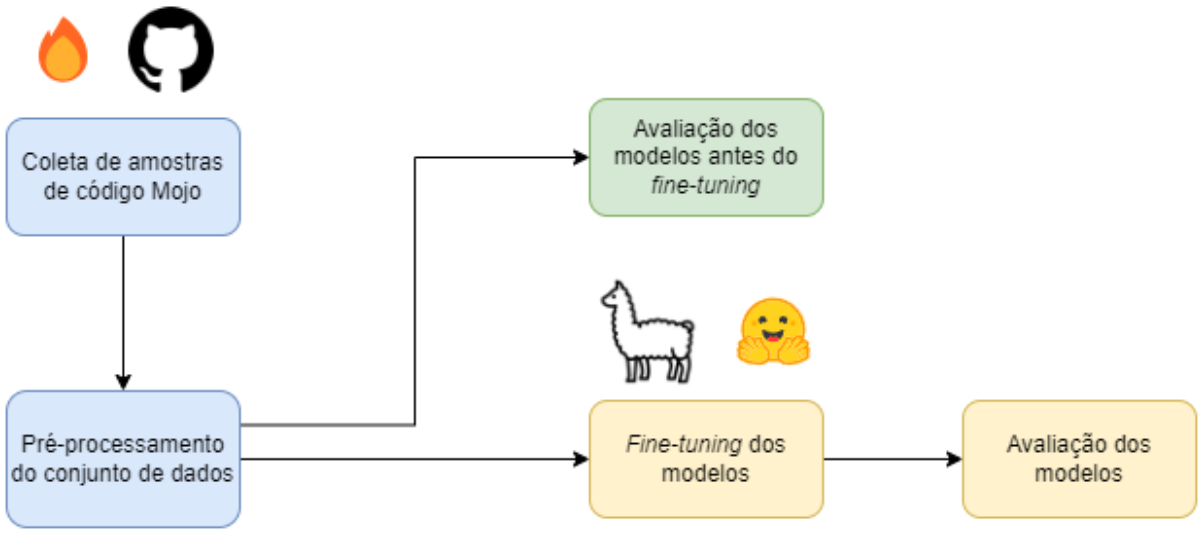

Figura 1: Pipeline de desenvolvimento da pesquisa

Cada variante do Code Llama possui um conjunto de tarefas as quais o modelo é capaz de realizar, na seguinte distribuição:

Tabela 1 - Tarefas realizadas por cada variante do Code Llama

Modelo	**Code completion**	**Infilling**	**Instructions/Chat**
Code Llama 7b	x	x	
Code Llama Python 7b	x		
Code Llama Instruct 7b	x	x	x

Para esta pesquisa foi realizado o fine-tuning de cada modelo Code Llama para capacitação em três tarefas distintas: O modelo Code Llama para a tarefa de *code infilling*, o modelo Code Llama Python para *code completion* e o modelo Code Llama Instruct para geração de código a partir de uma instrução em linguagem natural.

3.1 Coleta e pré-processamento de dados

Os dados necessários para a tarefa de fine-tuning foram coletados a partir de repositórios abertos no GitHub que contêm códigos em Mojo. O conjunto de dados foi construído principalmente a partir do repositório que contém o código-fonte de Mojo e a documentação da linguagem. Para

cada variante do Code Llama existe um padrão de entrada, desta forma, o conjunto de dados foi reproduzido e adaptado para o formato compatível com cada uma das tarefas propostas.

O modelo Code Llama voltado para a tarefa de *code infilling* espera o trecho de código incompleto, com a parte do meio do código faltante. Os registros desta amostra encontram-se no seguinte formato:

Código 1: Exemplo de prompt Fibonacci para Code Llama

```
prompt = "def fibonacci <FILL_ME> return fibonacci(n-2)
+ fibonacci(n-1)"
```

O modelo Code Llama voltado para a tarefa de *code completion* espera o trecho de código incompleto. Os registros desta amostra encontram-se no seguinte formato:

Código 2: Exemplo de prompt Fibonacci para Code Llama Python

```
prompt = "def fibonacci("
```

Para o conjunto de dados voltado à tarefa de instruções, o formato esperado de entrada é o seguinte:

Código 3: Exemplo de prompt Fibonacci para Code Llama Instruct

```
prompt = f" <s>[INST]{input}/INST]{output}</s>"
```

No conjunto de dados voltado para a terefa de instruções, o *input* representa a instrução passada para o agente e que descreve o problema que deve ser respondido e o *output* é o trecho de código correspondente a resposta esperada da instrução dada. Os trechos de código coletados são majoritariamente o próprio código-fonte da linguagem e muitos já possuem uma breve explicação sobre o código no repositório original, sendo o par explicação-código utilizado como *input* e *output* respectivamente no conjunto de dados formatado para instruções.

O conjunto dedicado para testes contém amostras de entradas nos formatos esperados por cada variante de Code Llama e suas respectivas referências, ou seja, as respostas que são esperadas para cada entrada.

Desta forma, foram utilizados três conjuntos de dados formatados para as tarefas de instrução, *infilling* e *code completion* que foram derivados do conjunto de dados original. Cada conjunto foi dividido em subconjuntos para treinamento, com 199 amostras, validação, com 40 amostras e teste, contendo 6 amostras. Os conjuntos estão hospedados e disponíveis no Hugging Face *datasets* no formato JSON.

3.2 Avaliação antes do fine-tuning

Antes de fazer o *fine-tuning* para cada tarefa proposta, foram realizados experimentos com os modelos originais para medir as métricas de ROUGE e FrugalScore, assim como analisar os trechos de códigos que cada modelo foi capaz de gerar para as tarefas de *infilling*, *completion* e instrução.

Os valores de ROUGE foram calculados para cada tarefa proposta sobre o mesmo conjunto de teste que será aplicado para os testes após o refinamento dos modelos, contendo 6 amostras de código em Mojo. Foram medidas 4 variações de ROUGE: rouge1, rouge2, rougeL e rougeLsum. Os valores de FrugalScore foram medidos de com relação aos três exemplos gerados para os algoritmos de soma de inteiros, retorno do n-ésimo termo da sequência de Fibonacci e cálculo da diferença entre conjuntos.

Para conduzir os experimentos localmente de inferência com os modelos base Code Llama 7b, Code Llama Python 7b e Code Llama Instruct 7b foi necessário realizar a solicitação de acesso aos modelos para a Meta [15] e os modelos são baixados através da API disponibilizada pela Meta no website Hugging Face. Os pesos e o *tokenizer* dos modelos serão mantidos para os testes com os modelos originais, antes do *fine-tuning*.

3.3 Fine-tuning dos modelos

A tarefa de *fine-tuning* dos LLMs selecionados é iniciada através do download dos modelos base que iremos utilizar, assim como na foi feito para o experimento com os modelos sem modificações. Em seguida, foram determinadas as configurações de cada modelo para o treinamento.

Para realizar o *fine-tuning* dos modelos selecionados, o tipo de processamento de texto escolhido foi o *causal language modeling*, comumente utilizado para tarefas de geração de texto [16]. Foi utilizada a API AutoModelForCausalLM para carregamento dos modelos Code Llama. O *tokenizer* é inferido a partir do modelo carregado através da API AutoTokenizer e algumas parametrizações são definidas de acordo com a tarefa a ser realizada. Para o treinamento, foram definidos os parâmetros pad_token e o padding_side. Através de experimentação foi escolhido o valor “left” de padding_side, que apresentou os melhores resultados gerados. Também foi utilizado um *data collator* para modelagem de linguagem com o objetivo de realizar o treinamento em *batches*. Alguns parâmetros de treinamento podem ser customizados através da API STFConfig e, para esta pesquisa, foram escolhidos alguns destes como o otimizador *PagedAdam8bit*, o número de épocas de treinamento e o *learning rate* constante de 2e-4.

Para o treinamento de modelos, um técnica de otimização é feita através do agrupamento de sequências de forma que estas fiquem com um tamanho consistente entre *batches*. Para o *fine-tuning* dos modelos Code Llama selecionados, foi utilizado um *token* reservado para o preenchimento das sequências com o objetivo de uniformizar os seus tamanhos. A distribuição do tamanho das amostras para treinamento é dispersa, como representada na Figura 2, e por

este motivo é necessário determinar um *token* especial como valor para o parâmetro *pad_token* do teokenizador. O *token* especial será inserido como preenchimento nas amostras que são menores que o tamanho de sequência utilizado durante o treinamento.

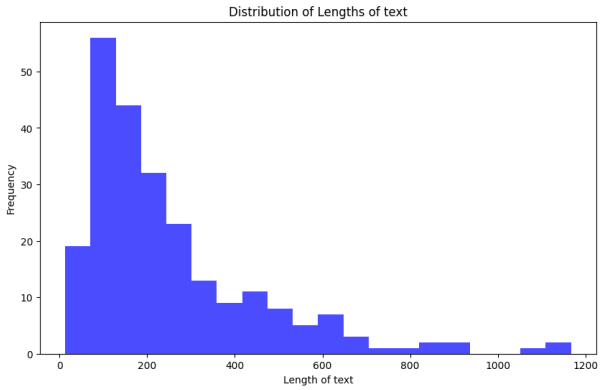

Figura 2: Distribuição de tamanho das amostras do conjunto de treinamento

O *fine-tuning* consiste de um novo treinamento do modelo pré-treinado sobre um conjunto de dado mais específico para o domínio. Entretanto, as LLMs selecionadas da família Code Llama possuem 7 bilhões de parâmetros e realizar *fine-tuning* desses modelos, mesmo sob um conjunto de dados relativamente pequeno, é uma tarefa que consome muitos recursos, sendo os principais memória e processamento. Para contornar esse desafio, foi utilizada a técnica de Low-Rank Adaptation (LoRa), que congela os pesos do modelo pré-treinado e faz a atualização dos pesos durante o treinamento através da decomposição da matriz de pesos original [17].

Os treinamentos dos três modelos Code Llama foram realizados no mesmo ambiente virtual do Google Colab, utilizando o recurso de GPU T4, 15 GB de RAM de GPU e 78.2 GB de disco. Os treinamentos foram realizados em 5 épocas, salvando o modelo obtido a cada época. O processamento apenas de treinamento dos modelos dura em média 10 minutos, utilizando GPU. O número de épocas foi determinado a partir de experimentação, sendo observado que, em média, na quinta época as perdas de treinamento passavam a ser menores que as perdas de validação, e seguiam diminuindo enquanto as perdas de validação aumentavam. Este comportamento é um indicativo de *overfitting* do modelo após uma determinada época e por isso o treinamento era finalizado após a quinta iteração.Ao longo de cada época de treinamento foram medidas as perdas sobre o conjunto de treino e de validação, sendo seus valores na a quinta época para cada variante refinada demonstrados na tabela a seguir:

Tabela 2 - Valores de perda na quinta época por modelo

Modelo	Perda de treino	Perda de validação
Code Llama 7b	0.472	1.079
Code Llama Python 7b	0.516	1.012
Code Llama Instruct 7b	0.511	1.015

3.4 Avaliação dos modelos

Foram feitas avaliações de cada variante refinada do Code Llama utilizando duas métricas: ROUGE e FrugalScore. As métricas de ROUGE foram calculadas sobre o subconjunto de teste que contém 6 amostras de entradas para cada modelo. Os códigos gerados a partir destas entradas são então comparados aos códigos de referência do conjunto de testes para obtenção dos valores de ROUGE. Assim como nos experimentos com os modelos sem modificações, foram medidas 4 variações de ROUGE: rouge1, rouge2, rougeL e rougeLsum.

A métrica de FrugalScore foi medida sobre uma amostra de três códigos gerados pelos modelos refinados e suas respectivas referências previamente definidas. Os algoritmos escolhidos para observação da capacidade de geração dos modelos e os quais foram avaliados utilizando o FrugalScore foram o cálculo da soma de inteiros, o retorno do n-ésimo termo da sequência de Fibonacci e o cálculo da diferença entre dois conjuntos.

4 Resultados e Análise

Nas tabelas 3 e 4, respectivamente, estão representados os resultados de ROUGE obtidos antes e após *fine-tuning* por cada variante de acordo com sua tarefa proposta:

Tabela 3 - Valores de avaliação de ROUGE para os modelos antes do fine-tuning

Modelo	rouge1	rouge2	rougeL	rougeLSum
Infilling	0.587	0.452	0.568	0.580
Completion	0.764	0.662	0.733	0.764
Instruct	0.296	0.185	0.275	0.285

Tabela 4 - Valores de avaliação de ROUGE para os modelos refinados

Modelo	rouge1	rouge2	rougeL	rougeLSum
Infilling	0.562	0.454	0.509	0.565
Completion	0.711	0.576	0.661	0.702
Instruct	0.511	0.398	0.492	0.508

Nas tabelas 5 e 6 encontram-se os valores de FrugalScore para os modelos antes e após o *fine-tuning*:

Tabela 5 - Valores de avaliação de FrugalScore para os modelos antes do fine-tuning

Modelo	Integer sum	Fibonacci	Set difference
Infilling	0.89	0.51	0.80
Completion	0.89	0.51	0.81
Instruct	0.17	0.19	0.0

Tabela 6 - Valores de avaliação de FrugalScore para os modelos refinados

Modelo	Integer sum	Fibonacci	Set difference
Infilling	0.89	0.97	0.81
Completion	0.89	0.95	0.81
Instruct	0.52	0.72	0.22

4.1 Tarefa de infilling de código (Code Llama 7b)

Para a avalição do modelo Code Llama 7b refinado foi utilizada a base de dados dedicada a testes, contendo 6 amostras. Foram gerados códigos baseados nas amostras que continham apenas as partes iniciais e finais do código de referência. Após o refinamento deste modelo sob a base de treinamento, os testes para a tarefa de *infilling* geraram os seguintes resultados:

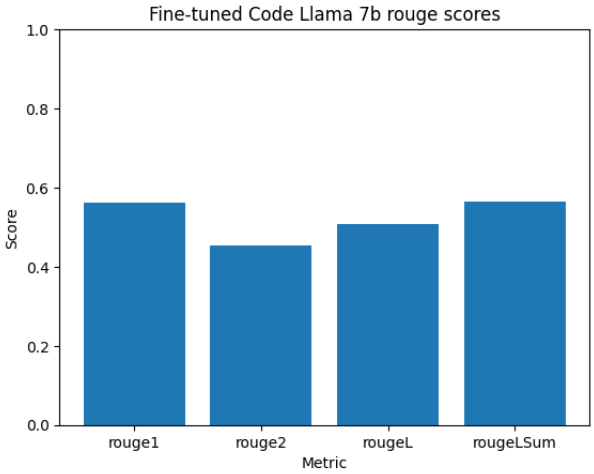

Figura 3: Valores de avaliação de ROUGE para o modelo Code Llama 7b refinado

Para observação e experimentação dos códigos gerados, foram criados três exemplos de código com o modelo original e com o modelo refinado. O primeiro exemplo gerado foi um programa que calcula a soma de dois números inteiros e a retorna. Na tabela a seguir é possível observar os códigos gerados antes e depois do *fine-tuning* do Code Llama para a tarefa de *infilling* de um trecho de código contendo um algoritmo de soma de inteiros:

Tabela 7 - Códigos gerados pelo Code Llama 7b antes e após refinamento para o algoritmo de soma de inteiros

Prompt: fn sum(a: Int, b: Int)	
Code Llama antes do fine-tuning	**Code Llama após fine-tuning**

fn sum(a: Int, b: Int)
Int
{
  a + b
}
fn main()
  Int
{
  sum(1,2)
	fn sum(a: Int, b: Int) -> Int:
	  return a + b
	fn main():
	  print(sum(1, 2))

Na tabela 8 estão representados os códigos que foram retornados pelo modelo Code Llama antes e após o *fine-tuning* para o cálculo do n-ésimo termo da sequência de Fibonacci:

Tabela 8 - Códigos gerados pelo Code Llama 7b antes e após refinamento para o algoritmo de cálculo do n-ésimo termo da sequência de Fibonacci

Prompt: fn fibonacci(return fibonacci(n - 2) + fibonacci(n - 1)	
Code Llama antes do fine-tuning	**Code Llama após fine-tuning**

fn fibonacci(
return fibonacci(n - 2) + fibonacci(n - 1)
n: u64,
) -> u64 {
if n == 0 {
return 0;
}
if n == 1 {
return 1;
}
	fn fibonacci(
	n: Int,
	) -> Int:
	if n < 2:
	return n;
	return fibonacci(n - 2) + fibonacci(n - 1)

O último exemplo solicitado para o Code Llama 7b para a tarefa de *infilling* foi para completar o código de cálculo da diferença entre conjuntos e os resultados encontram-se a seguir:

Tabela 9 - Códigos gerados pelo Code Llama 7b antes e após refinamento para o algoritmo de cálculo da diferença entre dois conjuntos

Prompt: fn difference(self, other: Self) -> Self: if e[] not in other: result.add(e[]) return result^	
Code Llama antes do fine-tuning	**Code Llama após fine-tuning**

fn difference(self, other: Self) -> Self:
var result = self.toSet()
for e in other:
if e[] not in other:
result.add(e[])
return result^
	fn difference(self, other: Self) -> Self:
	var result = Set[T]()
	for e in self:
	if e[] not in other:
	result.add(e[])
	return result^

Os resultados obtidos de FrugalScore para o modelo refinado Code Llama comparados estão representados na Figura 4 e foi observado que o código gerado para o algoritmo de retorno do n-ésimo Fibonacci está mais próximo do seu código de referência:

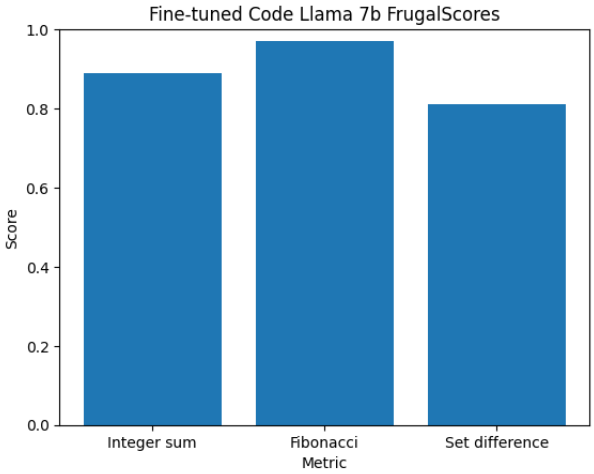

Figura 4: Valores de avaliação de FrugalScore para o modelo Code Llama 7b refinado

4.2 Tarefa de conclusão de código (Code Llama Python 7b)

Para a avaliação para a tarefa de *code completion* utilizando o conjunto de métricas ROUGE foi utilizado o conjunto dedicado para testes contendo 6 amostras. O conjunto é subdividido em trechos de código Mojo completos, demarcados como system_output, os mesmos trechos de código, desta vez incompletos, demarcados como user_input. Desta forma, utilizamos os trechos de user_input para geração com o modelo Code Llama Python que foi ajustado e mantivemos os trechos de system_output como as referências que usaremos para avaliação. Através da Figura 5 podemos observar a consistência entre os resultados apresentados para as métricas ROUGE, sendo todas acima de 0.5.

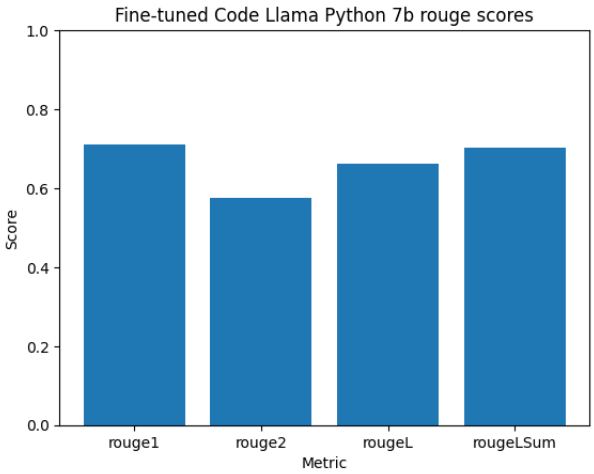

Figura 5: Valores de avaliação de ROUGE para o modelo Code Llama Python 7b refinado

Para o exemplo de cálculo de soma de inteiros, os resultados obtidos são demonstrados na tabela abaixo:

Tabela 10 - Códigos gerados pelo Code Llama Python 7b antes e após refinamento para o algoritmo de soma de inteiros

Prompt: fn difference(self, other: Self) -> Self: if e[] not in other: result.add(e[]) return result^	
Code Llama Python 7b antes do fine-tuning	**Code Llama Python 7b após fine-tuning**

fn sum(a: Int, b: Int) -> Int {
a + b
}
fn main() {
let a = 1;
let b = 2;
let c = sum(a, b);
	fn sum(a: Int, b: Int) -> Int:
	return a + b
	@parameter
	fn main() -> None:
	@parameter
	fn test_sum(a: Int, b: Int) ->

Na tabela 11 encontram-se os trechos de códigos gerados pelo modelo Code Llama Python para a tarefa de conclusão de código a partir de um trecho de código para cáculo do n-ésimo termo da sequência de Fibonacci antes e depois de realizar o *fine-tuning*:

Tabela 11 - Códigos gerados pelo Code Llama Python 7b antes e após refinamento para o algoritmo de cálculo do n-ésimo termo da sequência de Fibonacci

Prompt: fn fibonacci(	
Code Llama Python antes do fine-tuning	**Code Llama Python após fine-tuning**

fn fibonacci(n: u64) -> u64 {
if n < 2 {
return n;
}
let mut a = 0;
let mut b = 1;
	fn fibonacci(n: Int) -> Int:
	if n < 2:
	return n
	return fibonacci(n - 2) + fibonacci(n - 1)

Na tabela abaixo são demonstrados os resultados obtidos pela variante Code Llama Python para a tarea de conclusão de código dado como entrada um trecho parcial de um código que representa o cálculo da diferença entre conjuntos:

Tabela 12 - Códigos gerados pelo Code Llama Python 7b antes e após refinamento para o algoritmo de cálculo da diferença entre conjuntos

Prompt: fn difference(self, other: Self) -> Self: if e[] not in other: result.add(e[]) return result^	
Code Llama Python antes do fine-tuning	**Code Llama Python após fine-tuning**

fn difference(self, other: Self) -> Self:
var result = Set[T]()
for e in self:
if !other.contains(e):
result.add(e)
for e in other:
if !self.contains(e):
	fn difference(self, other: Self) -> Self:
	var result = Set[T]()
	for e in self:
	if not e in other:
	result.add(e[])
	return result ^

Os resultados obtidos de FrugalScoree com o modelo refinado Code Llama Python para a tarefa de conclusão de código estão comparados na Figura 6:

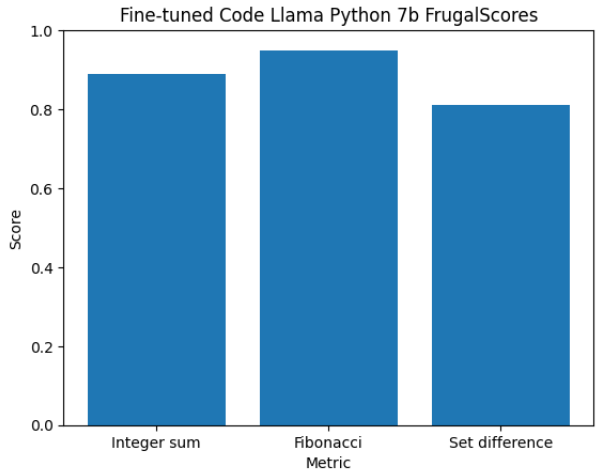

Figura 6: Valores de avaliação de FrugalScore para o modelo Code Llama Python 7b refinado

O modelo refinado atingiu mais de 0.8 de FrugalScore nos três experimentos, demonstrando uma alta similaridade entre o código que foi gerado pelo modelo refinado e o código de referência. Assim como no modelo refinado baseado no Code Llama 7b, o algoritmo de retorno do n-ésimo termo da sequência de Fibonacci apresenta o mais alto valor de FrugalScore, alcançando 0.95 de resultado.

4.3 Tarefa de geração baseada em instrução (Code Llama Instruct 7b)

O modelo voltado a instruções foi avaliado sobre a sua base de dados dedicada para teste. O formato de entrada desta variante do Code Llama é uma instrução em linguagem natural e

os resultados esperados são trechos de código. Desta forma, para avaliar as métricas de rouge score foi necessário utilizar os resultados gerados a partir da instrução e, como referência, foram utilizados os códigos completos que são esperados de acordo com a instrução passada.

Os valores de ROUGE foram medidos com base no conjunto de teste contendo 6 amostras para este modelo e os seus resultados são comparados na Figura 7:

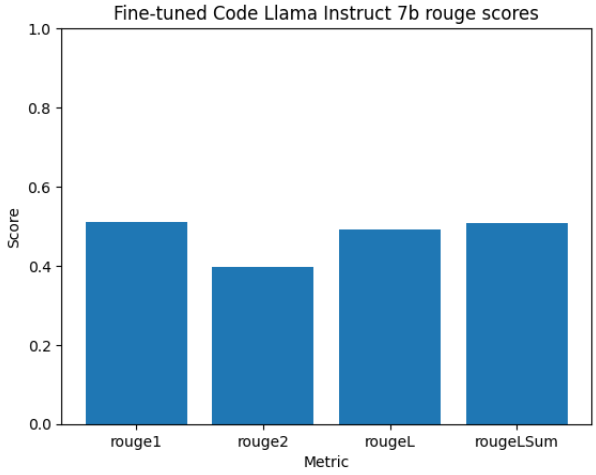

Figura 7: Valores de avaliação de ROUGE para o modelo Code Llama Instruct 7b refinado

O primeiro código gerado para análise do modelo a patir de instrução em linguagem natural foi o de soma de inteiros:

Tabela 13 - Códigos gerados pelo Code Llama Instruct 7b antes e após refinamento para o algoritmo de soma de inteiros

Prompt: Write a function to calculate the sum of two integers.	
Code Llama Instruct antes do fine-tuning	**Code Llama Instruct após fine-tuning**

def sum(a, b):
    return a + b
	Args:
	    a: An integer.
	    b: Another integer.
	Returns:
	    The sum of the two integers.
	@always_inline
	    fn _sum(a: Int, b: Int) -> Int:
	        return a + b
	        fn _sum(a: Int, b: Int) -> Int:
	            return a + b
	        return _sum

Na tabela a seguir, encontram-se os trechos de códigos resultantes a partir da instrução de geração de uma função para calcular o n-ésimo termo da sequência de Fibonacci:

Tabela 14 - Códigos gerados pelo Code Llama Instruct 7b antes e após refinamento para o algoritmo cálculo do n-ésimo termo da sequência de Fibonacci

Prompt: Write a function to calculate the nth fibonacci number.	
Code Llama Instruct antes do fine-tuning	**Code Llama Instruct após fine-tuning**

def fib(n):
  if n == 0:
  return 0
  elif n == 1:
  return 1
  else:
  return fib(n-1) + fib(n-2)
	fn fib[n: Int]() -> Int:
	  if n < 2:
	  return n
	  else:
	  return fib[n - 1]() + fib[n - 2]()
	# Test the function.
	print(fib[10]())

O último exemplar gerado foi a partir da intrução de cálculo da diferença entre conjunto. Na tabela 15 estão definidos os códigos gerados pelo modelo Code Llama Instruct 7b antes e após o *fine-tuning*:

Tabela 15 - Códigos gerados pelo Code Llama Instruct 7b antes e após refinamento para o algoritmo cálculo da diferença entre dois conjuntos

Prompt: Write a function to calculate the sum of two integers.	
Code Llama Instruct antes do fine-tuning	**Code Llama Instruct após fine-tuning**

fn symmetric_difference(self, other: Set[T]) -> Set[T]:
  return self.difference(other).union
	fn difference[T](a: Set[T], b: Set[T]) -> Set[T]:
	  # Create a new set.
	  var result = Set[T]()
	  # Iterate over the elements of the first set.
	  for e in a:
	  # If the element is not in the second
	  # set, add it to the result.
	  if e not in b:
	  result.add(e)
	  # Return the result.
	  return result ^[/[
	return difference(b, a)

Foi observado que os códigos gerados por esta variante possuem mais comentários dedicados à explicação dos trechos gerados.

Os resultados obtidos com o modelo refinado Code Llama Instruct apresentaram mais divergências com relação às referências, como observado pelos resultados de ROUGe e Fru-

galScore. As métricas de FrugalScore resultantes são comparadas na Figura 8, sendo perceptível que o algoritmo de cálculo do n-ésimo termo da sequência de Fibonacci também é o mais próximo da sua referência entre os três sob análise:

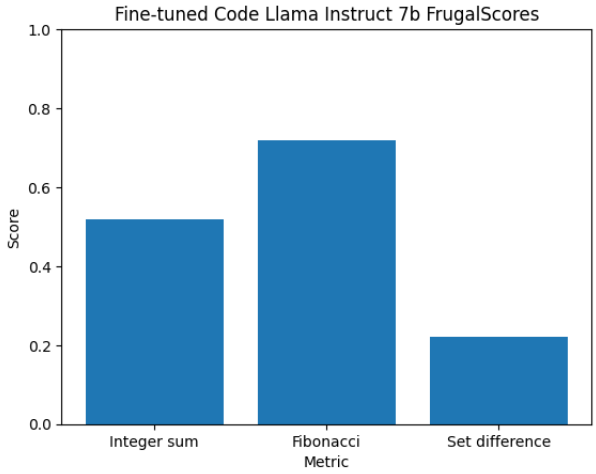

Figura 8: Valores de avaliação de FrugalScore para o modelo Code Llama Instruct 7b refinado

4.4 Comparação de resultados

Os modelos original e refinados para *infilling* e para *code completion* apresentaram resultados de FrugalScore semelhantes entre si:

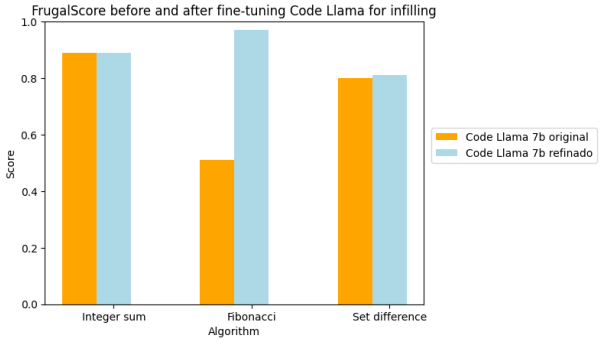

Figura 9: Comparação de FrugalScore entre Code Llama 7b original e refinado

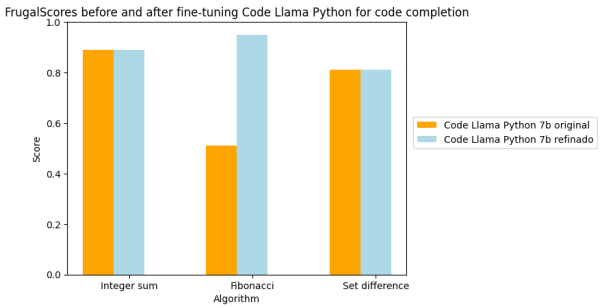

Figura 10: Comparação de FrugalScore entre Code Llama Python 7b original e refinado

Os resultados obtidos antes e após o *fine-tuning* para os algoritmos de soma de inteiros e o de diferença entre conjuntos demonstram que não houve uma melhoria relevante para ambas as variantes. Para o algoritmo de cálculo do n-ésimo termo da sequência de Fibonacci é possível observar a melhoria considerável dos códigos gerados para ambas as variantes do Code Llama após o refinamento.

No gráfico a seguir está ilustrada a diferença entre o modelo Code Llama Instruct antes e depois do *fine-tuning*:

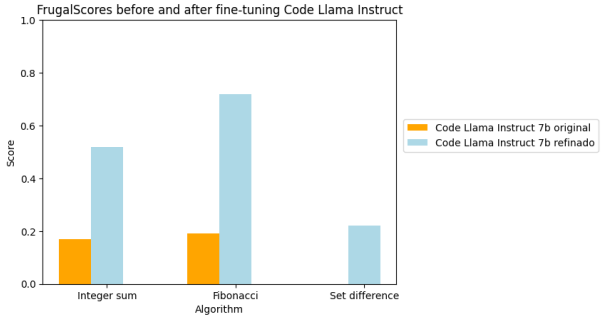

Figura 11: Comparação de FrugalScore entre Code Llama Instruct 7b original e refinado

Os resultados de FrugalScore para a tarefa de geração de código baseada em instrução em linguagem natural são mais baixos em comparação com os modelos que tiveram um código base de entrada. Também é considerável a melhoria do modelo após o refinamento para a tarefa de geração baseada em instrução, com os três algoritmos testados apresentando um aumento de FrugalScore em pelo menos 3 vezes.

Os resultados obtidos para os modelos refinados voltados para *code infilling* e *code completion* obtiveram resultados de FrugalScore acima de 0,8 para os três exemplos gerados. Um comportamento que foi notado foi nos resultados de ROUGE para a variante refinada do Code Llama Python foram consideravelmente mais altos que o Code Llama refinado, mas ambos apresentaram medidas de FrugalScore muito semelhantes entre si. Já o modelo voltado para instruções obteve resultados menos precisos. Entre os três modelos que foram refinados e testados, a variante do Code Llama Instruct apresentou a maior diferença de códigos gerados e os códigos de referência na linguagem Mojo.

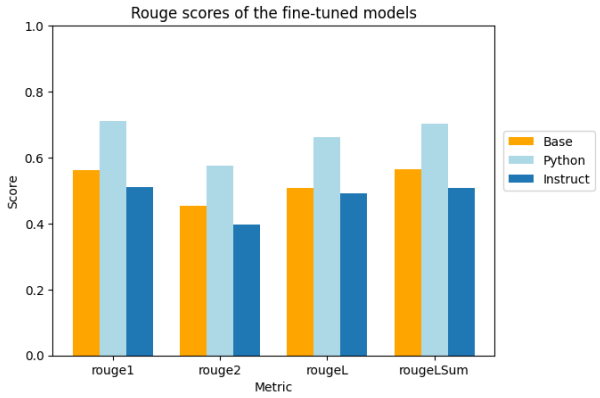

Figura 12: Valores de avaliação de ROUGE para os três modelos refinados

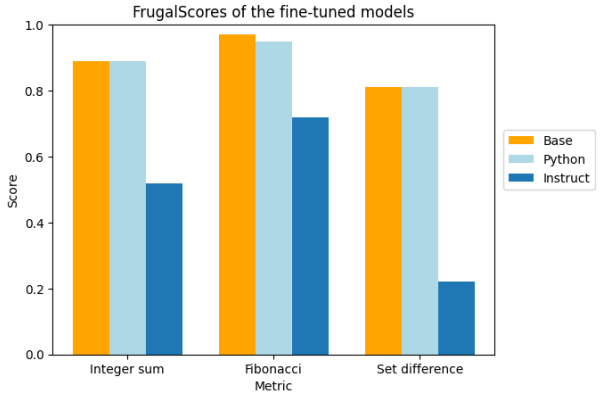

Figura 13: Valores de avaliação de FrugalScore para os três modelos refinados

Os modelos Code Llama e Code Llama Python geraram os códigos mais próximos das referências e os exemplos de soma de inteiros do modelo Code Llama refinado e o de diferença entre conjuntos do Code Llama Python refinado funcionam sem necessidade de alterações. Os algoritmos de cálculo do n-ésimo termo da sequência de Fibonacci apresentaram os melhores valores de FrugalScore para as três variantes do Code Llama refinadas e os códigos gerados através de *infilling* e *completion* não precisaram de modificações, já o gerado através de intrução necessita de poucas alterações para funcionar. Os outros exemplos gerados para ambas as variantes funcionam após pequenas modificações. A variante refinada no Code Llama Instruct para geração a partir de instrução em linguagm natural gerou trechos de códigos que necessitam maiores modificações para funcionar.

4.5 Análise

A segunda parte da pesquisa consiste na experimentação baseada em *prompt* e foi realizada utilizando o ChatGPT 4o (GPT-4o) através da técnica *few-shot learning*, que consiste na inserção de trechos de código na linguagem Mojo junto a instrução para geração de código com o objetivo de observar os resultados obtidos e compará-los aos dos modelos que foram refinados

anteriormente. A amostra que foi fornecida de exemplo foi a mesma amostra contendo 6 trechos de códigos de referência utilizada para executar os testes dos modelos anteriormente.

Foram realizados experimentos com os três algoritmos utilizados para análise dos modelos anteriores: soma de inteiros, retornar o n-ésimo termo da sequência de Fibonacci e calcular a diferença entre dois conjuntos. Após forncecer as 6 amostras de teste como exemplos para o ChatGPT 4o, as instruções foram passadas de acordo com a tarefa que seria realizada. Para as tarefas de *code completion* e *infilling* foram fornecidos os trechos de códigos base junto à instrução da tarefa em linguagem natural. Para a tarefa de geração baseada apenas em linguagem natural, apenas a instrução era fornecida. Para todos os experimentos, o ChatGPT 4o retornava explicações sobre o código que foi gerado. Não foi citado em suas respostas em qual linguagem foram escritos os códigos gerados, mas o ChatGPT 4o identificou que os exemplos dados para contexto são de uma linguagem similar a Rust.

Durante as pesquisas foram realizados testes com o ChatGPT 4o para geração de código Mojo a partir de contexto em linguagem natural sobre a linguagem Mojo. O contexto fornecido é um compilado de explicações sobre funcionalidades e implementação da linguagem Mojo retirada diretamente da documentação disponibilizada pela Modular. Os experimentos baseados em *few-shot learnig* utilizando o ChatGPT 4o apresentaram melhores resultados e menos esforço quando comparados aos experimentos de aprendizagem por contexto em linguagem natural. Desta forma, os resultados apresentados sobre o ChatGPT 4o nesta pesquisa são referentes às experimentações através de *few-shot learning*.

O ChatGPT 4o quando utilizado para a tarefa de *infilling* apresentou bons resultados, destacando a tarefa de calcular a soma de inteiros que foi corretamente inferida pelo ChatGPT 4o, assim como pelo Code Llama:

Código 4: Algoritmo de soma de inteiros gerado pelo ChatGPT 4o através de infilling

```
@always_inline
    fn sum(a: Int, b: Int) -> Int:
        return a + b
```

Também foram obtidos bons resultados os testar a geração de código baseada em instruções. Baseando-se nos exemplos fornecidos, o ChatGPT 4o conseguiu gerar códigos Mojo a partir de instruções em linguagem natural:

Código 5: Algoritmo de cálculo de soma de inteiros gerado pelo ChatGPT 4o baseado em instrução em linguagem natural

```
@always_inline
fn sum(a: Int, b: Int) -> Int:
    return a + b
```

Código 6: Algoritmo de cálculo de diferença entre dois conjuntos gerado pelo ChatGPT 4o baseado em instrução em linguagem natural

```
@always_inline
fn difference[T: element_trait]
(set1: Set[T], set2: Set[T]) ->
Set[T]:
    var result = Set[T]()
    for element in set1:

        if element not in set2:
            result.add(element)
        return result
```

Os testes para *code completion* utilizando o ChatGPT 4o apresentaram mais divergências em comparação aos resultados obtidos para as tarefas de *infilling* e de geração de código baseada em instrução em linguagem natural. Os códigos gerados baseados em *code completion* para os três exemplos solicitados apresentaram o uso de chaves para delimitação de um bloco de comandos ao invés da delimitação com o uso de dois pontos e indentação, como é esperado pela linguagem Mojo. Nos trechos de código a seguir estão representados os exemplos de geração para o cálculo de soma de inteiros e para o cálculo do n-ésimo termo da sequência de Fibonacci:

Código 7: Algoritmo de soma de inteiros gerado pelo ChatGPT 4o baseado em code completion

```
fn sum(a: Int, b: Int) -> Int {
    return a + b
}
```

Código 8: Algoritmo de cálculo do n-ésimo termo da sequência de Fibonacci pelo ChatGPT 4o baseado em code completion

```
fn fibonacci(n: Int) -> Int {
    if n <= 1 {
        return n
    }
    return fibonacci(n - 2) + fibonacci(n - 1)
}
```

Os códigos gerados através da tarefa de conclusão de código não estão de acordo com a sintaxe esperada da linguagem Mojo, mas apresentam a lógica correta para os cálculos solicitados.

5 Conclusão

5.1 Considerações Finais

Os modelos refinados baseado no Code Llama 7b e no Code Llama - Python 7b alcançaram bons resultados, levando em consideração o tamanho do conjunto de dados utilizado para o treinamento. O modelo refinado baseado no Code Llama - Instruct 7b gerou resultados mais distantes daqueles utilizados como referência. Entretanto, eram retornados em sua maioria com comentários documentando parte do código gerado e demonstraram precisar de poucos ajustes para se adequarem ao resultado esperado.

O ChatGPT 4o apresentou bons resultados ao utilizarmos a técnica de *few-shot learning*, que consiste em apresentar alguns exemplos daquilo que é esperado como resultado antes de fornecer a instrução desejada.

Os modelos Code Llama apresentam as vantagens de serem gratuitos para uso e possuir variantes de tamanho reduzido, sendo possível utilizar os modelos e refiná-los utilizando recusos limitados tanto localmente como através de ambientes virtuais de terceiros. Já o ChatGPT 4o apresentou resultados próximos aqueles obtidos através dos modelos refinados do Code Llama com menos esforço utilizando a técnica *few-shot learning*, mas possui a desvantagem de atualmente ser pago.

5.2 Limitações

Esta pesquisa utilizou uma pequena amostra de dados na linguagem Mojo, o que pode influenciar a capacidade de generalização dos modelos que foram refinados [18]. Também foi possível observar durante as experimentações que as respostas obtidas a cada interação com os modelos de larga escala refinados sob análise variam após treinamento. Por fim, os experimentos realizados têm como objetivo simplificar a observação e análise dos códigos gerados pelos modelos e avaliação dos resultados com base nas suas referências. Assim, não foi utilizado um *benchmark* formal para análise mais profunda da performance da linguagem Mojo para avaliação da sua velocidade, consumo de memória e nem acurácia.

5.3 Trabalhos Futuros

Este trabalho propõe o refinamento de modelos de aprendizagem profunda acessível em termos de recursos computacionais para geração de código Mojo. Esta linguagem encontra-se no início do seu desenvolvimento e aplicação, apresentando um planejamento de lançamento de novas funcionalidades no futuro. Desta forma, uma sugestão de trabalho futuro é o refinamento de LLMs para geração de código Mojo em um conjunto de dados expandido, que reflita as novas funcionalidades que serão lançadas e mais exemplares de códigos abertos que vêm sendo implementados pela comunidade de Mojo.

A proposta de Mojo é ter a usabilidade de Python provendo alta performace para tarefas de aprendizagem de máquina e IA e, caso alcance adesão da comunidade desta área, poderemos observar o aumento do interesse da transformação de códigos em Python para Mojo. Desta forma, outra sugestão é a investigação de LLMs de código para tradução de códigos Python para Mojo, explorando as possibilidades que esta linguagem oferece para otimização.

No momento de desenvolvimento desta pesquisa novos modelos com mais parâmetros e ainda mais capazes vêm sendo lançados ao público. Desta forma, esta pesquisa pode ser expandida para o estudo do refinamento de outros modelos, como Llama 3 ou Gemma, para a tarefa de geração de código. Por fim, é sugerido o investimento no uso de *benchmarks* para uma avaliação mais completa da capacidade de LLMs para geração de códigos em Mojo.

Referências

[1] Jiho Shin et al. “Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks”. Em: *arXiv preprint arXiv:2310.10508* (2023).

[2] Baptiste Roziere et al. “Code llama: Open foundation models for code”. Em: *arXiv preprint arXiv:2308.12950* (2023).

[3] Abhinav Nagpal e Goldie Gabrani. “Python for data analytics, scientific and technical applications”. Em: *2019 Amity international conference on artificial intelligence (AICAI)*. IEEE. 2019, pp. 140–145.

[4] Humza Naveed et al. “A comprehensive overview of large language models”. Em: *arXiv preprint arXiv:2307.06435* (2023).

[5] Juyong Jiang et al. “A Survey on Large Language Models for Code Generation”. Em: *arXiv preprint arXiv:2406.00515* (2024).

[6] Ashish Vaswani et al. “Attention is all you need”. Em: *Advances in neural information processing systems* 30 (2017).

[7] Aston Zhang et al. “Dive into Deep Learning”. Em: *arXiv preprint arXiv:2106.11342* (2021).

[8] Hugo Touvron et al. “Llama 2: Open foundation and fine-tuned chat models”. Em: *arXiv preprint arXiv:2307.09288* (2023).

[9] Ruidan He et al. “On the effectiveness of adapter-based tuning for pretrained language model adaptation”. Em: *arXiv preprint arXiv:2106.03164* (2021).

[10] Martin Weyssow et al. “Exploring parameter-efficient fine-tuning techniques for code generation with large language models”. Em: *arXiv preprint arXiv:2308.10462* (2023).

[11] Yuchen Zeng e Kangwook Lee. “The expressive power of low-rank adaptation”. Em: *arXiv preprint arXiv:2310.17513* (2023).

[12] Modular. *Why Mojo*. URL: `https://docs.modular.com/mojo/why-mojo` (acesso em 04/07/2024).

[13] Chin-Yew Lin. “Rouge: A package for automatic evaluation of summaries”. Em: *Text summarization branches out*. 2004, pp. 74–81.

[14] Moussa Kamal Eddine et al. “Frugalscore: Learning cheaper, lighter and faster evaluation metricsfor automatic text generation”. Em: *arXiv preprint arXiv:2110.08559* (2021).

[15] Meta. *Download Llama*. URL: `https://llama.meta.com/llama-downloads/` (acesso em 27/07/2024).

[16] Hugging Face. *Language Modeling with Transformers*. Accessed: 2024-07-28. 2024. URL: https://huggingface.co/docs/transformers/tasks/language_modeling.

[17] Edward J Hu et al. “Lora: Low-rank adaptation of large language models”. Em: *arXiv preprint arXiv:2106.09685* (2021).

[18] Haoran Yang et al. “Unveiling the generalization power of fine-tuned large language models”. Em: *arXiv preprint arXiv:2403.09162* (2024).