Java 代码补全方案报告

1. 任务介绍

随着大模型技术的飞速发展，大模型在帮助程序员完成编码任务上取得了巨大提升。根据当前代码的上下文自动生成代码片段（FIM，Fill In the Middle）就是其中的一项主要任务。它能够节省研发人员编程成本，提高编码效率。

2. 模型选择

此次项目以 Deepseek-Coder-1.3B 模型作为基础预训练模型，采用 **LLaMA Factory** 训练工具对其进行微调。使得模型能够根据当前 Java 代码的上下文自动生成代码片段。

Deepseek-Coder在代码补全任务上的 prompt 格式为："<｜fim_begin｜>" + prefix_code + "<|fim_hole|>" + suffix_code + "<|fim_end|>"。其中，<|fim_begin|>是补全前缀，<｜fim_end｜>是补全后缀，<｜fim_hole｜>是模型有待补全的内容。prompt 示例如下：

```
<｜fim_begin｜>def quick_sort(arr):
    if len(arr) <= 1:
        return arr
    pivot = arr[0]
    left = []
    right = []
<｜fim_hole｜>
        if arr[i] < pivot:
            left.append(arr[i])
        else:
            right.append(arr[i])
    return quick_sort(left) + [pivot] + quick_sort(right)<｜fim_end｜>
```

3. 数据收集与清洗

3.1 数据来源

由于此次任务是对 Java 代码进行补全，所以本项目从 huggingface Java-GitHub-Codes 这个数据集中进行采样，以构造相应的 prompt 和 response。Java-GitHub-Codes 数据集一共包含了 640 万份 Java 代码，同时记录了代码仓库来源、开源许可证类别、代码长度等信息。Java-GitHub-Codes 数据集示例如下：

code	repo_name	path	language	license	size
package io.github.ageofwar.telejam.updates; ...	AgeOfWar/Telejam	src/main/java/io/github/ageofwar/telejam/updates/UpdateReader.java	Java	mit	4138

3.2 数据清洗

在 Java-GitHub-Codes 数据集中，由于是直接从 GitHub 上获取的，没有经过任何处理，所以在许多代码开头存在与代码本身无关的 license 注释。于是，在数据清洗阶段会先检测这些大段的 license 注释，并将其去除。处理代码如下所示：

```
def remove_leading_comments(java_code):
    # 去除开头的 /**/ 多行注释
    java_code = re.sub(r'^/\*[\s\S]*?\*/', '', java_code, flags=re.MULTILINE)
    # 去除开头的多个 // 单行注释
    java_code = re.sub(r'^//.*', '', java_code, flags=re.MULTILINE)
    # 去除多余的空行
    java_code = re.sub(r'^\s*$', '', java_code, flags=re.MULTILINE)
    return java_code.lstrip()
```

其次，由于 GitHub 上的代码来源于不同的操作系统（Windows, Linux, macOS），所以代码字符串中的换行表示有所不同：\r\n, \n, \r。在数据清洗过程中，将换行表示统一转换为 "\n"，以便于后续 prompt 构造及字符处理。此外，将制表符 \t 统一转换为 4 个空格。处理代码如下：

```
code = row['code'].replace('\r', '').expandtabs(4)
```

另外，为了确保 Java 代码的质量，避免空的、没有实际内容的代码字符串加入训练集。在数据清洗过程中，将过滤掉代码有效行数小于 10 行的 Java 代码。同时，由于资

源限制，将过滤掉字符数大于 8k 的 Java 代码。统计代码有效行数的处理代码如下：

```
def count_non_empty_lines(code_str):
    lines = code_str.strip().splitlines()
    return sum(1 for line in lines if line.strip())
```

3.3 Prompt 制作

对于每份 Java 代码字符串，首先将其按照每行转化为字符串列表，列表中的每项就是每行的代码。从中随机截取连续的若干行作为有待补全的代码，剩下的代码行作为上下文 prompt 输入。最后，拼接补全前缀、后缀和待补全的特殊标识符。此外，为了确保有足够的上下文提示让大模型进行代码补全，有待补全的代码行数不能超过总代码行数的 20%。prompt 制作代码如下：

```
def generate_prompt_response(code: str, max_hole_lines: int = 6,
max_hole_ratio: float = 0.2):
    bos = "<｜fim▁begin｜>"
    eos = "<｜fim▁end｜>"
    hole = "<｜fim▁hole｜>"
    lines = code.splitlines()
    num_lines = np.random.randint(1, min(max_hole_lines,
int(len(lines)*max_hole_ratio)) + 1)  # 随机选择挖取的行数
    start_line = np.random.randint(0, len(lines) - num_lines)  # 随机选择起
始行
    # 构造 response
    response = '\n'.join(lines[start_line:start_line + num_lines])
    if start_line + num_lines < len(lines):
        response += '\n'
    # 构造 prompt
    prompt = '\n'.join(lines[:start_line]) + '\n' + hole +
'\n'.join(lines[start_line + num_lines:])
    prompt = bos + prompt + eos
    return prompt, response
```

3.4 训练集构造

为了让大模型学习到更广泛的上下文和代码结构模式，提高其泛化能力。在构造训练集时，一份 Java 代码将生成多个 prompt 和 response，即挖取多个不同的代码块。最终，本项目一共构造了 10 万条 prompt 和 response，每份 Java 代码构造 5 条记录，训练集和验证集按照 99：1 进行划分。prompt 和 response 示例如下：

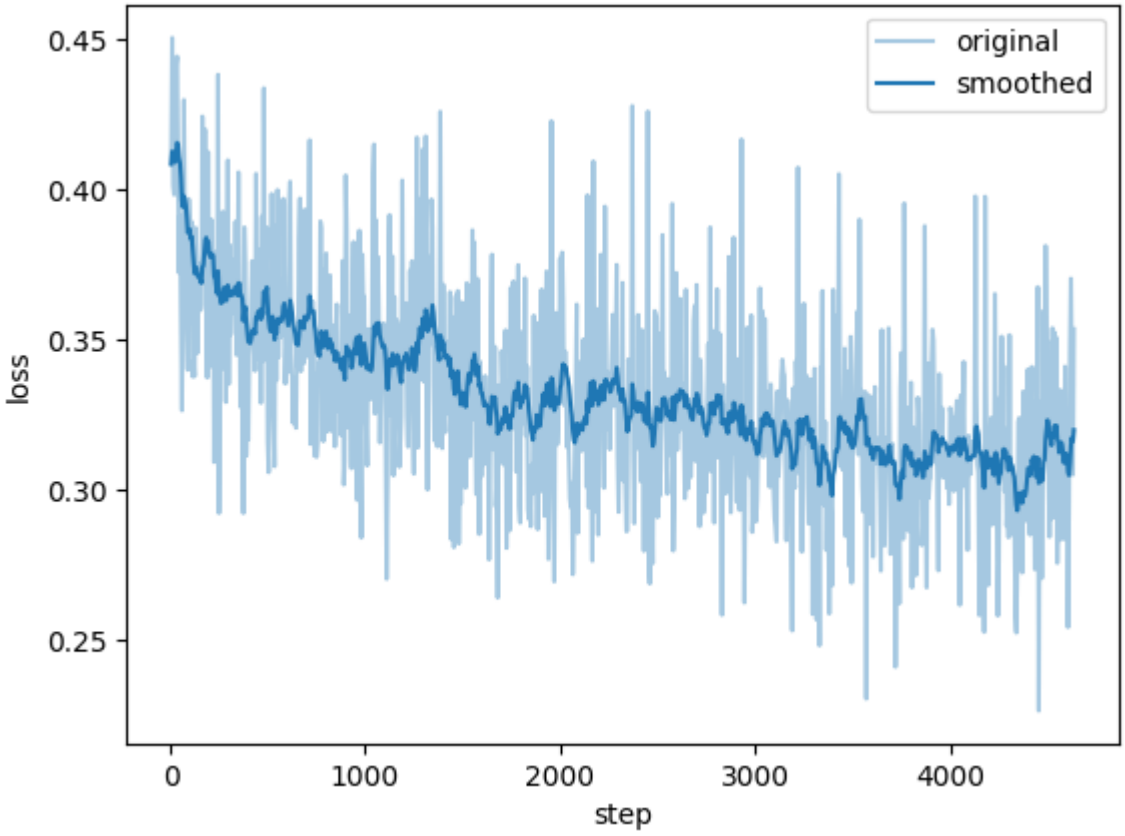
```
{
   "prompt": "<｜fim▁begin｜>package cherry.foundation.generator.test;\n\nimport lombok.EqualsAndHashCode;\nimport lombok.Getter;\nimport lombok.Setter;\nimport lombok.ToString;\n\n@Getter\n@Setter\n@EqualsAndHashCode(callSuper = true)\n@ToString(callSuper = true)\npublic class GroupForm extends GroupFormBase {\n<｜fim▁hole｜>}<｜fim▁end｜>",
   "response": "\n    private static final long serialVersionUID = 1L;\n\n"
}
```

4. LoRA 微调

本项目采用 LLaMA Factory 框架的 LoRA 的方式，对大模型进行监督微调（SFT）。由于训练集中已经构造好了 prompt 模板，所以在 LLaMA Factory 中微调大模型时，对话模板选择 empty，即不额外构造 prompt 模板。各项重要的训练参数如下所示：

GPU	微调方法	训练阶段	学习率	epoch	batch size
RTX 4090 x1	LoRA	SFT	5e-5	3	8
optim	**计算类型**	**截断长度**	**梯度累积**	**LoRA Alpha**	**LoRA Rank**
adamw	bf16	2048	8	16	8

LLaMA Factory 微调训练大模型的损失变化图如下所示，可以看出损失逐步下降，训练正常进行。



5. 模型评估

本项目对模型采用的评估指标为 Exact Match 方法 ,即对模型补全结果与实际挖空的正确答案之间的对比，在过滤一些无意义的字符后，若结果一致，则补全分数越高。由于实际开发过程中 ,用户倾向于接受的代码补全行数往往比较少 ,因此在评估过程中，只关注补全结果前六行的代码准确性。最终将前 6 行的评估分数取平均值作为最终的分数。EM 评估方法示例如下，generate 和 reference 一共有 3 对（ "123" ，"1as24b" ，"kghj78" ）是完全一样的，所以准确率为 60%。

```
generate = ["123", "abc", "1as24b", "kghj78", "1"]
reference = ["123", "ab", "1as24b", "kghj78", "2"]
```

大模型训练微调后，与基础预训练模型，在验证集上的评估结果对比如下所示。其中 ,block_1 - block_6 分别表示 1 - 6 行代码补全的评估结果。可以看出微调之后的大模型，在 Java 代码补全任务上，性能得到了明显的提升。

```
// 微调后
{
    "block_1": 0.448,
    "block_2": 0.3152317880794702,
    "block_3": 0.259581881533101,
    "block_4": 0.19363395225464192,
    "block_5": 0.17475728155339806,
    "block_6": 0.11904761904761904
}

// 微调前
{
    "block_1": 0.29,
    "block_2": 0.20397350993377483,
    "block_3": 0.15331010452961671,
    "block_4": 0.11140583554376658,
    "block_5": 0.0825242718446602,
    "block_6": 0.03571428571428571
}
```