srt记录

第一次组会---2023.1.12

TODO:

任务背景

当前的这个模型是基于transformer的自然语言处理预训练模型。数据处理的时候把整个程序代码当作文本在词表中找到对于ID，形成一个行向量，直接投喂到模型中。实际上这就损失了很多程序语言特有的结构化信息，所以学长考虑把程序语言特有的一些结构化特征加入到模型中，初步考虑学长想到了AST(抽象语法树)作为程序的描述。有了AST信息，我们可以获得如下受益：

> - AST一些关键序列、结点信息可以和原来的向量合并作为输入序列
> - 用于生成代码时根据AST一些规则生成符合语法的代码
> - 在数据训练时对一些不鲁棒、不高效的程序数据直接删掉

AST生成工具

给定代码文本以及语言label，生成指定形式的AST返回。可以参考开源代码以及相关书籍。

完成过程

idea1：每一种程序语言找到对于的编译器前端

查找了关于AST生成的算法，我是想在python环境下写，查了查python环境下有ast类，可以将python代码生成ast。于是我就想一个一个语言查相应的库，最后汇总起来。一开始我C语言查到了pycparser,是一个专门在python环境下解析C语言的库，最后通过学长找到了astexplorer开源项目，该项目做的工作就是把不同语言编译器前端整合到一起。

> 问题：ast不统一，没有统一的处理接口，后续研究不方便

idea2：ANTLR4开源软件

ANTLR4是业界比较权威的语法解析软件，而且能够用统一的框架支持多个语言，不同语言用不同语法文件描述，最后用ANTLR统一处理。

> 在尝试了有两周之后发现antlr网上教程较少，不同语法文件生成的parsetree接口不一样，不知道如何使用。对树遍历原理不清楚。

idea3：treesitter

treesitter是在搜索代码生成相关的论文时找到的，网上有一个相关的教程，在学长帮助下发现可以使用。一开始想要通过遍历语法树的方式生成JSON格式的ast，最后发现treesitter有统一的接

口sexp()可以生成lisp格式的ast，于是通过python字符串的操作转化为了list格式的ast。

> 完成情况：
> 例如：

```
def factorial(n):
    factorial = 1
    for i in range(1, n + 1):
        factorial *= i
    return factorial
```

解析后生成ast列表：

```
[['module ', ['function_definition name: ', ['identifier'], ' parameters: ',
['parameters ', ['identifier']], ' body: ', ['block ', ['expression_statement ',
['assignment left: ', ['identifier'], ' right: ', ['integer']]], ' ', ['for_statement
left: ', ['identifier'], ' right: ', ['call function: ', ['identifier'], ' arguments:
', ['argument_list ', ['integer'], ' ', ['binary_operator left: ', ['identifier'], '
right: ', ['integer']]]], ' body: ', ['block ', ['expression_statement ',
['augmented_assignment left: ', ['identifier'], ' right: ', ['identifier']]]]], ' ',
['return_statement ', ['identifier']]]]]]
```

> **使用方式**
>
> setting.ipynb文件内code2ast函数，参数为language和filepath。

第二次组会---2023.2.7

TODO:
在完成list格式的ast生成后，我发现不同语法生成的ast在结构和结点命名上都有区别，因此下一步任务就是建立不同语法文件的映射字典。一开始我打算人工通过代码来比对，学长说这样不行，还是要从不同语言的语法文件中去找相关的信息。但是发现语法文件是一个JSON格式，结构单一但内容复杂。所以可以用递归的方式统一地处理JSON格式的文件来映射为python字典。

> 完成情况：
> grammar.ipynb文件内存有将各个语言grammar.json文件解析后得到的字典。

第三次组会---2023.2.11

TODO:

测试语法正确性

每个语法文件都有examples文件夹，内部有.tex文件，在treesitter官网上查看自动化验证语法正确性的方法，用python编写自动化验证程序

> 完成情况：
> 检查下载的treesitter解析器能否通过自带的corpus测试，具体测试内容和方法在test.ipynb

代码预处理——源代码拆分

将带有注释的整个源代码文件作为输入，利用treesitter生成的ast提供的结构性信息将含有多个类、函数以及引用类库的源文件拆分为多个完整的注释+类的形式，引用类库部分可以省略。**目的是为了增强模型代码生成能力**

> 完成情况：
> 由于不同语言节点不一样，所以必须要一个一个进行调研，这花费我很长时间。我准备建立两个词典：

```
#分割代码时根据文本内容和生成的ast进行分析，找到需要提取的节点名称。由于不同语言有差异，故建立字典
#该字典为不同语言的主体节点
dict_codebody={'c': ['function_definition', 'struct_specifier'],
 'cpp': ['function_definition', 'namespace_definition', 'class_specifier', 'template_declaration', 'struct_specifier'],
 'c_sharp': ['namespace_definition', 'class_declaration', 'interface_declaration', 'struct_declaration'],
 'go': ['type_declaration', ' method_declaration', 'function_declaration'],
 'java': ['class_declaration', 'interface_declaration'],
 'php': ['class_declaration', 'expression_statement', 'interface_declaration', 'function_definition'],
 'python': ['function_definition', 'class_definition'], 'ruby': ['class', 'module', 'method'],
  'rust': ['trait_item', 'struct_item', 'impl_item', 'enum_item', 'function_item', 'type_item', 'mod_item'],
   'tsx': ['lexical_declaration', 'export_statement', 'expression_statement', 'function_declaration', 'class_declaration', 'interface_declaration'],
   'fortran': ['module', 'subroutine', 'function', 'submodule', 'program'],
   'kotlin': ['call_expression', 'function_declaration', 'prefix_expression', 'class_declaration', 'object_declaration', ''],
   'cuda': ['function_definition', 'template_declaration', 'namespace_definition', 'struct_specifier', 'class_specifier', 'type_definition'],
   'scala': ['class_definition', 'object_definition', 'function_definition', 'trait_definition']}

#该字典为不同语言主体的前缀，需要在body的前面作为提示信息
dict_codehint={'c': ['comment', ' type_definition', 'declaration'],
'cpp': ['comment', 'type_definition', 'declaration'], 'c_sharp': ['comment', 'global_statement'],
'go': ['comment', 'var_declaration'], 'java': ['block_comment', 'line_comment'],
'php': ['php_tag', 'namespace_definition', 'namespace_use_declaration', 'comment'],
 'python': ['comment', 'expression_statement-string'], 'ruby': ['comment'],
 'rust': ['line_comment', 'block_comment'], 'tsx': ['comment'], 'fortran': ['comment'],
 'kotlin': ['comment'], 'cuda': ['comment'], 'scala': ['comment']}
```

建立词典的代码为,该代码输出源代码相应信息，帮助更快地找到重要的节点名称：

```
import json
import os
file="./data"
for root,dirs,files,in os.walk(file):
  #将jsonl文件转化为str文本
  for file in files:
    text=[]
    path=os.path.join(root, file)
    language=file.split('_')[0]
    if language not in supported_languages:
      continue
    else:
      print(language+'\n')
      with open(path,'r') as f:
        for line in f:
          #每一行读取后都是一个json，可以按照key去取对应的值
          obj=json.loads(line)
          text.append(obj['code'])
      #分割代码
      methods_list=[]
      code_list=[]
      for i in text:
        methods_list.append(split_code(dict_codegeex2ts[language],i))
        code_list.append(i)
      #写文件
      newroot='./code'
      string=language+'.jsonl'
      path=newroot+'/'+string
      with open(path,'w',encoding='utf-8') as f:
        for i in text:
          obj['code']=i
          obj['code_cleaned']=methods_list
        f.write(json.dumps(obj))
        f.close
        print('Success\n')
```

```
from tree_sitter import Language, Parser
#解析代码获取parsetree
def getast_tree(language,text):
    LANGUAGE=Language('build/my-languages.so', language)
    parser=Parser()
    parser.set_language(LANGUAGE)
    tree=parser.parse(bytes(text,"utf8"))
    return tree
def search(node,language):
    nodes=set()
    node_childs = node.children
    if node_childs==[]:
        return set()
    for i in node_childs:
        nodes.add(i.type)
        nodes=nodes|search(i,language)
```

```
    return nodes
def test_text(language,text):
    tree=getast_tree(language,text)
    nodes=set()
    nodes=search(tree.root_node,language)
    for i in nodes:
        if 'comment' in i:
            return True
    return False
#对根节点的第一层节点进行搜索，将class、function节点加入
def dfs(node,language):
        node_childs = node.children
        if node_childs == []: return
        method_list=[]
        mod=""
        for i in range(len(node_childs)-1,-1,-1):
            if node_childs[i].type in dict_codebody[language]:
                method_list.append(node_childs[i])
                mod='body'
            elif node_childs[i].type in dict_codehint[language]:
                if mod=='body':
                    method_list.append(node_childs[i])
            else:
                mod='trash'
        return method_list
def split_code(language,code):
    "返回所有函数和函数节点信息"
    tree=getast_tree(language,code)
    method_nodes=[]
    method_nodes=dfs(tree.root_node,language)
    code_list = code.split('\n')
    methods_list = []
    method=""
    for n in range(len(method_nodes)-1,-1,-1):
        if method_nodes[n].type in dict_codehint[language]:
            method+=method_nodes[n].text.decode('utf-8') # bytes转str
            method+='\n'
        else:
            method+= method_nodes[n].text.decode('utf-8')
            if test_text(language,method):
                methods_list.append(method)
            method=""
    return methods_list
```

一共写了三版，第二版能够做到只把和函数类直接相关的注释放在函数前边，第三版做到了将前缀和内部没有注释的函数直接筛除掉。

grammar.json文件语法树比对

利用树匹配相关算法对不同语法文件的字典进行匹配，最终目的是提供统一的ast格式，不过鉴于任务可能比较复杂，一开始可以先通过树匹配算法找到类似的结构。任务可能比较复杂，长期工作。

> 完成情况：任务较为复杂，学长将其细化

第四次组会---2023.2.16

研究代码相似度指标
TODO:读一篇论文.这篇论文里有提到Jaccard similarity指标，用来判断两段代码的相似度.
需要做的是：

1. 跑一跑Jaccard similarity，看一下是否能很好反应代码相似性
2. 代码转化成AST后，是否已有或设计一个指标来判断AST间的相似性

> 完成情况：
> 通过和ChatGPT的聊天，我快速地了解到了这一算法，并且深入地了解了这个工作的任务是near-duplicate。并通过Chatgpt了解了常见的算法以及python库，甚至还直接给出了示例代码。

ChatGPT给出的基于 Jaccard 相似度改进的相似度算法。以下是其中的一些例子：

- SimHash：SimHash 是一种基于 Jaccard 相似度的哈希算法，用于计算文本、图像等数据的相似度。它通过将数据表示为一个二进制向量，并计算这些向量的哈希值来衡量它们之间的相似程度。
- Locality-Sensitive Hashing (LSH)：LSH 是一种基于哈希函数的相似度计算方法，适用于高维数据集。它可以将相似的数据映射到同一哈希桶中，从而提高查询效率。
- Shingle-based Jaccard Similarity：Shingle-based Jaccard Similarity 是一种基于 Jaccard 相似度的文本相似度计算方法。它将文本表示为多个词汇序列，称为 shingle。然后，使用 Jaccard 相似度计算这些 shingle 的相似程度。
- Min-Hashing based Jaccard Similarity：除了 MinHash Jaccard 相似度之外，还有其他基于 MinHash 的相似度计算方法，如 Weighted Min-Hashing。

这些相似度计算方法都是基于 Jaccard 相似度的基础上进行改进和扩展，可以适用于不同类型的数据和应用场景，具有不同的优缺点。

ChatGPT给出的相关的库：

- Python 的 "sklearn" 库：提供了 MinHashing 和 Shingle-based Jaccard Similarity 的实现。可用于处理文本、图像等类型的数据。
- Python 的 "datasketch" 库：提供了 MinHash 相关的实现，支持多个哈希函数的计算，也支持 Jaccard 相似度的计算。
- Python 的 "pybktree" 库：提供了基于 Levenshtein 距离和 Jaccard 相似度的相似度计算。
- C++ 的 "SimHash" 库：提供了 SimHash 算法的实现，可用于计算文本、图像等类型的数据的相似度。
- Java 的 "Simmetrics" 库：提供了多种字符串相似度计算方法，包括 Jaccard 相似度、Levenshtein 距离、Smith-Waterman 等。

实际测试发现，这几种方法中LSH和SimHash并没有充分利用到AST信息，Shingle-based jaccard在取样的过程中将代码看作是普通文本，也没有利用AST信息，所以最后我决定将AST的前序序列利用MinHash Jaccard相似度进行计算。

为了利用AST信息，我找到了几种基于AST的代码相似度检测方法：
1、基于编辑距离
2、基于子树匹配
3、基于特征向量
4、基于神经网络
我尝试着实现了几种，发现效果不好，进而我又找到了几种开源软件

Moss：Moss（Measure of Software Similarity）是一个跨语言的代码相似性度量工具，它可以检测多种编程语言的代码相似性，包括 C、C++、Java、Python、JavaScript、Ruby 等。Moss 的工作原理是将代码转换成抽象语法树（AST），然后使用基于编辑距离的算法计算 AST 之间的相似度。

Plagiarism Checker：Plagiarism Checker 是一个基于云端的代码重复检测工具，它可以检测多种编程语言的代码相似性，包括 C、C++、Java、Python、JavaScript 等。Plagiarism Checker 的工作原理是将代码转换成抽象语法树（AST），然后使用基于子树匹配的算法计算 AST 之间的相似度。

CodeMatch：CodeMatch 是一个跨语言的代码相似性度量工具，它可以检测多种编程语言的代码相似性，包括 C、C++、Java、Python、JavaScript、PHP 等。CodeMatch 的工作原理是将代码转换成向量表示，并使用向量相似度度量算法计算代码之间的相似度。

问题：

1. 这个任务的目的是什么
2. 如何客观评价自己设计的指标
3. 是否需要把这些方法都遍历一遍做对比

回答：
1、找到能够较好反映代码相似度的指标。

2、观察代码翻译的结果，看和代码相似度指标是否有相关性。(可以用来强化代码翻译模型)

结果：测试完MinHash和SimHash的AST和CODE两种版本，测试结果**均不理想**。测试代码见code_similarity.ipynb，测试结果见./data/code_translation

分析可能原因：

1. 计算相关性均单独计算了各个语言文件，没有统一。
2. 测试文件本身的数据问题
3. 自己实现的代码相似度算法有问题
4. 处理文件的代码有问题，概率较小