软件学报 ISSN 1000-9825, CODEN RUXUEW
Journal of Software [doi: 10.13328/j.cnki.jos.006940]

E-mail: jos@iscas.ac.cn
http://www.jos.org.cn
Tel: +86-10-62562563

面向代码搜索的函数功能多重图嵌入*

徐 杨, 陈晓杰, 汤德佑, 黄 翰

(华南理工大学 软件学院, 广东 广州 510006)

通信作者: 黄翰, E-mail: hhan@scut.edu.cn

摘 要: 如何提高异构的自然语言查询输入和高度结构化程序语言源代码的匹配准确度, 是代码搜索的一个基本问题. 代码特征的准确提取是提高匹配准确度的关键之一. 代码语句表达的语义不仅与其本身有关, 还与其所处的上下文相关. 代码的结构模型为理解代码功能提供了丰富的上下文信息. 提出一个基于函数功能多重图嵌入的代码搜索方法. 在所提方法中, 使用早期融合的策略, 将代码语句的数据依赖关系融合到控制流图中, 构建函数功能多重图来表示代码. 该多重图通过数据依赖关系显式表达控制流图中缺乏的非直接前驱后继节点的依赖关系, 增强语句节点的上下文信息. 同时, 针对多重图的边的异质性, 采用关系图卷积网络方法从函数多重图中提取代码的特征. 在公开数据集的实验表明, 相比现有基于代码文本和结构模型的方法, 所提方法的 *MRR* 提高 5% 以上. 通过消融实验也表明控制流图较数据依赖图在搜索准确度上贡献较大.

关键词: 代码搜索; 控制流图; 数据依赖图; 函数功能多重图

中图法分类号: TP311

中文引用格式: 徐杨, 陈晓杰, 汤德佑, 黄翰. 面向代码搜索的函数功能多重图嵌入. 软件学报. http://www.jos.org.cn/1000-9825/6940.htm

英文引用格式: Xu Y, Chen XJ, Tang DY, Huang H. Code-search-oriented Function Multigraph Embedding. Ruan Jian Xue Bao/Journal of Software (in Chinese). http://www.jos.org.cn/1000-9825/6940.htm

Code-search-oriented Function Multigraph Embedding

XU Yang, CHEN Xiao-Jie, TANG De-You, HUANG Han

(College of Software Engineering, South China University of Technology, Guangzhou 510006, China)

Abstract: How to improve the accuracy of matching between natural language query input and highly structured programming language source code is a fundamental concern in code search. Accurate extraction of code features is one of the key challenges to improving matching accuracy. The semantics expressed by statements in codes is not only relevant to themselves but also to their contexts. The structural model of the code provides rich contextual information for understanding code functions. This study proposes a code search method based on function multigraph embedding. By using an early fusion strategy, the study fuses the data dependencies of code statements into a control flow graph and constructs a function multigraph to represent the code. The multigraph explicitly expresses the dependency relationships of indirect predecessor and successor nodes that are lacking in the control flow graph through data dependencies and enhances the contextual information of statement nodes. At the same time, in view of the edge heterogeneity of the multigraph, this study uses the relational graph convolutional network to extract the features of the code from the function multigraph. Experiments on a public dataset show that the proposed method can improve the *MRR* by more than 5% compared with the existing methods based on code text and structural models. The ablation experiments also show that the control flow graph contributes more to the search accuracy than the data dependence graph.

Key words: code search; control flow graph (CFG); data dependence graph (DDG); function multigraph

---

* 基金项目: 广东省自然科学基金面上项目 (2020A1515010696, 2022A1515011491); 国家自然科学基金面上项目 (61876207, 62276103); 中央高校面上项目 (2020ZYGXZR014); 广东省财税大数据重点实验室开放基金 (2019B121203012)
收稿时间: 2022-05-09; 修改时间: 2022-10-04, 2022-12-18, 2023-02-16; 采用时间: 2023-03-27; jos 在线出版时间: 2023-07-26

搜索代码是利用自然语言查询语句搜索符合其描述的代码片段的行为[1], 是现代软件开发过程中最频繁的活动之一. 研究表明, 超过 90% 的开发者会去搜索代码以便复用现有的高质量代码, 从而减轻学习负担, 提高软件的生产效率和质量[2].

代码搜索的一个基本问题是, 如何提高异构的自然语言查询输入和高度结构化程序语言源代码的匹配准确度. 其中, 准确地理解源代码表达的结构和功能语义, 提取代码特征是关键之一. 源代码中, 每条语句所表达的功能语义不仅与其本身有关, 而且还与其所处的上下文相关, 不同的上下文导致语句具有不同的语义. 例如, 若 send 和 recv 两个函数调用处于循环结构中, 其表达的语义可能是进行拒绝服务攻击[3], 而若两个函数调用出现在顺序结构中, 其表达的功能是正常地收发数据. 因而, 代码搜索任务实质上是搜索实现某一或某些功能的函数或代码片段, 而不是其中的一两个语句.

现有的代码搜索方法可以分为两类: 基于信息检索的方法和基于深度学习的方法.

传统的基于信息检索的方法[4−6]将代码视作纯文本, 把代码搜索看作是代码和查询间的文本匹配, 这类方法存在两个问题: 一是这类方法只提取了代码文本的浅层语义特征, 若查询和代码使用不同的词汇表达相同的意图, 则可能匹配失败[1]; 二是这类方法用词袋模型表示代码, 没有关注代码中普遍存在的顺序、分支和循环等结构特征, 而这些结构特征是理解代码语句重要的上下文信息.

而基于深度学习的方法将深度学习的方法引入到代码搜索问题研究中, 近年来受到研究者广泛关注. 这类方法提取代码的潜在语义特征, 可以缓解基于信息检索的方法存在的前述第 1 个问题. 根据代码表示的不同, 可以将该类方法分为两类, 分别是基于代码文本特征的方法[7−9]、基于结构特征的方法[10−12]. 基于文本特征的方法仍然将代码看作纯文本, 也没有关注代码的结构特征, 仍然存在基于信息检索的方法存在的第 2 个问题. 而基于结构特征的方法, 除了关注代码文本外, 还从底层的代码实现细节中抽象出表示代码结构和功能的模型, 比如函数的抽象语法树 (abstract syntax tree, AST)、控制流图 (control flow graph, CFG)[13]和程序依赖图 (program dependence graph, PDG)[14], 为提取代码特征提供更多的语义信息. 其中, 程序依赖图 (PDG) 通过控制依赖图 (CDG) 表达代码中语句之间的控制依赖 (control-dependent) 关系, 但没有描述理解语句功能语义所需的重要上下文的顺序、分支、循环等结构特征. 抽象语法树 (AST) 仅表达了代码的语法特征. 而控制流图表达了代码的顺序、分支和循环结构特征, 为代码搜索提供了丰富的源代码结构和功能特征[15]. 但从目前使用控制流图作为代码结构表示的方法提供的实验结果看, 这些方法未取得理想的性能.

这里引出值得思考的问题: 控制流图是否有助于提取更准确的代码特征? 是否需要其他表示模型辅助控制流图更好地表示代码功能语义? 如果需要这种辅助模型, 如何融合不同的特征表示, 构建代表整个代码的特征, 即多模态特征融合问题. 此外, 如何从表示模型中准确提取表达代码功能语义的特征也是一个重要的问题. 即在设计特征提取方法时, 不仅要考虑代码语句的特征提取, 还要考虑语句上下文的特征提取.

针对函数级的代码搜索任务, 本文提出一个基于函数功能多重图嵌入的代码搜索方法, 称作 FuncGraphCS. FuncGraphCS 采用早期融合 (early fusion) 的策略[16], 将语句、控制流图和数据依赖图 3 种模态信息融合在一个有向多重图中, 作为函数代码的功能语义表示. 同时, 由于函数功能多重图中的边具有异质性 (heterogeneity), 本文采用关系图卷积网络 (RGCN)[17]提取该图的特征作为函数代码特征, 与查询语句的语义特征进行匹配, 完成代码搜索任务. 实验表明, FuncGraphCS 与现有典型的方法相比, 代码搜索的平均倒数排名 (*MRR*) 提高了 5% 以上, 表明函数功能多重图表示的上下文信息可以有效提高代码搜索的准确度, 这些上下文信息是代码搜索所需的重要信息. 同时, 通过消融实验也表明, 相比较数据依赖图, 控制流图对提取准确的代码特征贡献更大, 数据依赖图则可以辅助控制流图更好地表示功能语义.

本文第 1 节讨论了代码搜索的相关研究工作. 第 2 节介绍了函数的多重图表示. 第 3 节介绍本文的基于函数功能多重图嵌入的代码搜索方法. 第 4 节通过对比实验验证了所提方法的有效性, 并通过消融实验验证了各类特征对搜索准确度的贡献. 最后第 5 节总结全文.

1 相关研究

现有的代码搜索方法主要分为两类: 基于信息检索的方法和基于深度学习的方法.

基于信息检索的方法主要基于 TF-IDF、BM25 等算法. 为了提高检索的准确度, 一些方法针对性地提取了代码的属性信息 (如类名、方法调用) 和依赖关系以实现更细粒度的检索. 例如, Sourcer[4]利用 TF-IDF 和主题模型实现代码特征提取, 同时提取了代码中的实体和实体间关系, 利用代码依赖图[18]和 PageRank 算法[19]对搜索结果进一步排序. 类似地, CodeExchange[5]提供了代码复杂度等更细粒度的过滤条件, 有效缩小了搜索结果的范围. 此类方法存在的问题是, 只提取了代码文本的浅层语义特征, 需要代码中存在与查询语义匹配的词, 才能进行查询和代码的匹配. 为此, 有研究提出了查询扩展的方法. 例如, CodeHow[20]利用用户查询与 API 名称、API 描述的相似度, 寻找 Top 10 的 API 的词汇添加到查询中. Zhang 等人[6]通过用户搜索日志发现在查询中增加标识符 (主要是类名) 可以有效提高搜索准确度, 作者利用词向量相似度寻找与用户查询最相关的 5 个 API 类名扩展查询. Rahman 等人[21]同样认为 API 类名能有效提高准确度, 该文综合词向量、文本相似度和 PageRank 分数选择 Top K 的 API 类名扩展查询. Nie 等人[22]提出利用众智知识扩展查询, 对于每个查询, 从预先构建的问答对中寻找 Top K 个词扩展查询, Huang 等人[23]利用 GitHub Knowledge、代码演化信息等多种信息源提高查询扩展的多样性. 与现有查询扩展方法不同, 黎萱等人[24]通过增强代码描述的方式提高搜索准确度, 他们通过提取代码-描述语料库中代码的方法调用特征和代码结构特征, 增强代码库中的代码.

基于深度学习的方法又可以分为两类: 基于文本特征的方法和基于结构特征的方法.

基于文本特征的方法使用深度学习的相关方法从代码文本提取特征. DeepCS[7]最早将深度学习方法引入到代码搜索任务中, 该方法使用函数名称、函数内的 API 调用序列、函数代码文本作为特征. NCS[25]利用代码和注释无监督地训练词向量, 同时利用 TF-IDF 对词向量加权求和得到注释向量和代码向量并按相似度排序. UNIF[8]在 NCS 的基础上增加了有监督训练, 有效提高了性能. CARLCS[9]、SANCS[26]等模型认为现有方法使用两个不同的模型提取代码和注释特征, 缺乏特征交互, 因此利用注意力机制实现代码和注释的特征交互, 但这种方法需要实时计算注释和代码的交互特征, 无法离线计算代码特征, 极大降低了搜索效率. 得益于自然语言处理领域大规模预训练语言模型的发展, 代码领域也发展出了诸如 CodeBERT[27]、GraphCodeBERT[28]等预训练语言模型. 基于文本特征的方法仅将代码当作自然语言文本来处理, 忽视了代码的结构特征, 而这些结构特征表达了语句的功能语义. 而预训练模型需要庞大的训练数据, 并且模型参数多, 对内存、CPU、显卡等资源要求高, 训练和推理时间长.

基于结构特征的方法主要利用函数的抽象语法树、控制流图[13]和程序依赖图[14]等表示模型作为特征. MMAN[10]认为仅使用代码文本特征忽略了代码的结构特征, 因此提出使用代码文本、抽象语法树、控制流图 3 种表示模型作为特征, 分别使用 LSTM、Tree-LSTM[29]、GGNN[30]提取代码文本、抽象语法树、控制流图的特征, 最后通过中间融合策略得到最终的代码特征. MRNCS[12]认为利用多种模态的特征可以有效提高模型的性能, 同时该文还提出了简化语义树, 简化语义树去除了抽象语法树中对代码搜索任务无用的节点, 例如修饰符节点, 并通过中间融合策略[16]融合多模态特征, 最后得出结论, 使用简化语义树可以提高搜索准确度, 同时也验证了使用多模态特征可以提高准确度. DGMS[31]认为现有方法忽略了代码和查询文本的结构特征, 因此该文提出使用成分句法解析树表示查询文本, 使用增强的抽象语法树[32]表示代码, 然后使用关系图卷积网络[17]提取两种树的特征. 凌春阳等人[33]将软件项目表示为代码图, 结合信息检索和图嵌入方法学习代码图中的深层结构信息, 在提高搜索准确度的同时还降低了搜索响应时间. CRaDLe[11]认为神经网络难以提取抽象语法树的结构特征, 而控制流图中部分执行路径可能不会执行, 导致提取的代码特征存在偏差, 为此作者提出利用程序依赖图表示函数, 通过矩阵表示语句之间的依赖关系, 然后提取依赖特征, 并通过中间融合策略融合依赖特征和文本特征得到代码的特征向量. 与 CRaDLe 类似, 黄思远等人[34]也考虑了程序依赖图特征, 但该文通过构造节点子图序列和 Skip-Gram 算法提取图的特征. 与这两个研究不同, 本文认为语句顺序、分支和循环结构是表达函数功能的重要上下文信息, 控制依赖和数据依赖表达了语句之间的依赖关系, 但没有描述语句的顺序、分支、循环等结构特征. 从目前公开的实验结果看, 当前基于结构特征的方法的搜索准确度不高. 正如前述的分析, 本文认为现有方法在代码的表示模型方面大多使用的抽象语法树、程序依赖图等表示模型, 缺少描述顺序、分支、循环等结构特征, 影响了代码搜索的准确度.

表 1 从搜索粒度、数据表示模型、特征抽取方法、特征融合方法 4 个方面, 总结了现有的典型的代码搜索方

法的特点, 此外, 本文还汇总了各方法的代码公开情况. 其中, 特征融合方法指的是融合多模态特征的方法才具备的特点, “无”表示该方法没有使用多模态特征.

表 1 现有方法汇总

方法名称	搜索粒度	数据表示模型	特征提取方法	特征融合方法	源代码是否公开
Sourcer[4]	文件	语句词元序列、代码依赖图	TF-IDF、主题模型	无	否
CodeExchange[5]	文件	语句词元序列、代码属性	TF-IDF	无	否
CodeHow[20]	函数	语句的词元序列	TF-IDF	无	否
Zhang等人[6]	函数	语句的词元序列	TF-IDF	无	否
Rahman[21]	文件	语句的词元序列	TF-IDF	无	否
Nie等人[22]	文件	语句的词元序列	TF-IDF、主题模型	无	否
Huang等人[23]	文件	语句的词元序列	TF-IDF	无	否
黄思远等人[34]	函数	语句的词元序列	TF-IDF	无	否
黎萱等人[24]	函数	语句的词元序列	TF-IDF	无	否
DeepCS[7]	函数	方法名、API调用序列、语句词元序列	LSTM、MLP	向量拼接+MLP	是
NCS[25]	函数	语句词元序列	FastText	无	否
UNIF[8]	函数	语句词元序列	FastText、Attention	无	否
CARLCS[9]	函数	方法名、API调用序列、语句词元序列	CNN、Attention	Attention	是
SANCS[26]	函数	方法名、API调用序列、语句词元序列	WordEmbedding、Attention	无	是
CodeBERT[27]	函数	语句词元序列	Transformer	无	是
MMAN[10]	函数	语句词元序列、抽象语法树、控制流图	LSTM、Tree-LSTM、GGNN	向量拼接+MLP	是
凌春阳等人[33]	文件	代码图	TF-IDF、LINE[35]	无	否
CRaDLe[11]	函数	语句词元序列、语句的控制依赖和数据依赖矩阵	MLP、LSTM	向量拼接+LSTM	是
DGMS[31]	函数	增强的抽象语法树	RGCN	无	是
MRNCS[12]	函数	语句词元序列、简化语义树	WordEmbedding	向量相加	是

2 函数的多重图表示

正如本文前述的观点, 代码的每条语句所表达的语义不仅与其本身有关, 而且还与其所处的上下文相关. 使用什么表示模型表达语句及其上下文的特征, 特别是代码的功能特征, 是代码搜索任务首要解决的问题.

2.1 函数多重图定义

程序依赖图 (PDG)[14]是可选的表示模型之一. PDG 同时显示地表达了程序中存在的控制依赖 (control dependence) 和数据依赖关系 (data dependence). 控制依赖表达的是一个语句对另一个语句的控制依赖关系, 数据依赖描述了一个语句对另一个语句中值的依赖. 控制依赖和数据依赖可以描述代码语句的上下文特征. 但是, PDG 中的控制依赖, 仅描述了两两语句间控制依赖关系. 例如 if 代码块, PDG 的控制依赖描述了, if 代码块内的各语句以及嵌套在其中的代码块控制依赖于 if-condition 语句控制, 但 PDG 不描述 if 代码块内的不存在控制依赖关系, 但存在执行逻辑先后关系的语句间的关系, 没有完整地描述代码块内语句的顺序、分支和循环的结构关系, 而这些关系是理解语句功能的重要的上下文信息.

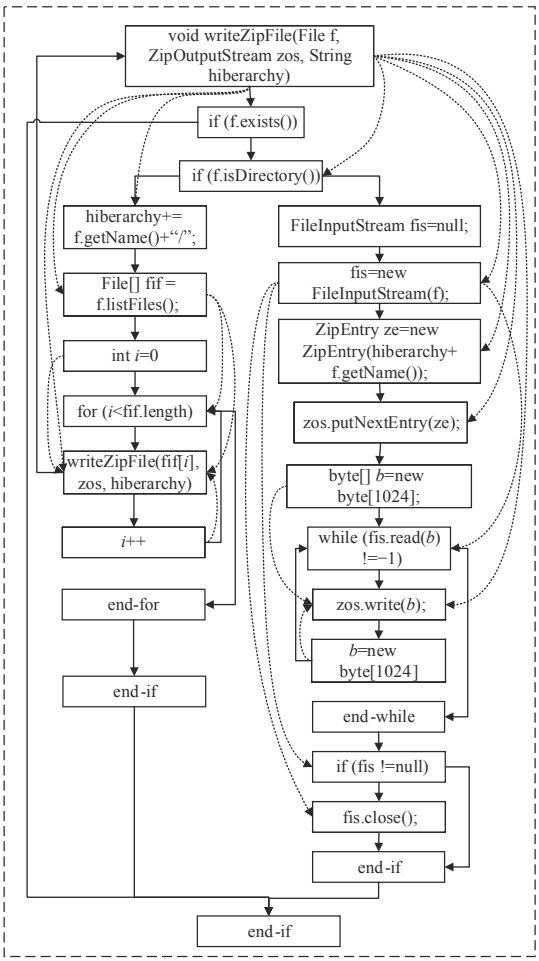
而控制流图, 它记录代码中语句的执行流程, 其中边表达了代码的顺序、分支和循环结构特征, 节点表示具有一些高度相关语句的代码块, 为代码搜索提供了丰富的源代码结构和功能特征[13], 而且 PDG 也是基于控制流图

提取出来的[14].

本文的函数代码特征, 是通过融合函数代码中的每一个语句特征构建的. 而每个语句特征的提取除了考虑语句的文本特征外, 还要考虑其上下文, 即与该语句发生关联的语句特征. 因此本文的控制流图是以函数内每个语句作为节点、以有向边连接节点构成的有向图, 有向边表示了语句的执行顺序, 包括了顺序、分支、循环. 但控制流图无法表示非直接前驱后继节点的依赖关系, 而这种依赖关系也是理解代码语句功能的上下文信息之一. 如图 1(a)代码中的语句 fis = new FileInputStream(f) 和 fis.close() 分别表示流的创建和关闭, 若不结合 fis = new FileInputStream(f) 语句则无法判断 fis.close() 的功能.

```
private void writeZipFile(File f, ZipOutputStream zos,
String hiberarchy) throws IOException {
   if (f.exists()) {
      if (f.isDirectory()) {
         hiberarchy+=f.getName()+"/";
         File[] fif=f.listFiles();
         for (int i=0; i<fif.length; i++) {
            writeZipFile(fif[i], zos, hiberarchy);
         }
      } else {
         FileInputStream fis=null;
         fis=new FileInputStream(f);
         ZipEntry ze=new ZipEntry(hiberarchy+
f.getName());
         zos.putNextEntry(ze);
         byte[] b=new byte[1024];
         while (fis.read(b) !=-1) {
            zos.write(b);
            b=new byte[1024];
         }
         if (fis != null)
            fis.close();
      }
   }
}
```

(a) 递归添加多个文件到压缩文件的函数



(b) 函数功能多重图

图 1 递归添加多个文件到压缩文件的函数及其函数功能多重图

本文使用数据依赖图 (data dependence graph, DDG), 来弥补控制流图无法表示非直接前驱后继节点的依赖关系的问题. 数据依赖图作为 PDG 子图, 是以函数内每个语句作为节点、以数据依赖边连接节点构成的有向图. 数据依赖边表达了变量使用语句对变量声明、赋值语句的依赖, 可以补充控制流图中无法表示的非直接前驱后继节点的依赖关系. 例如, 通过为图 1(a) 中的 fis = new FileInputStream(f) 和 fis.close() 显式地建立数据依赖边, 可以明确 fis.close() 语句所表达的功能是关闭文件输入流.

控制流图和数据依赖图中的节点都以语句为单位, 控制流图的节点集合包含了数据依赖图的节点集合. 基于这两个图的这种结构特征, 不同于现有的多模态方法采用中期融合 (intermediate fusion) 策略[15], 本文采用早期融

合[15]的策略, 将它们合并为一张有向多重图, 通过不同的边类型区分控制流图的边和数据依赖边.

总的来说, 本文的函数功能多重图是以函数内的语句作为节点、以控制流边和数据依赖边两种不同类型的有向边连接节点所构成的图. 正式地, 这里给出函数功能多重图的定义.

定义 1 (**函数功能多重图**). 给定函数代码 C 及其控制流图 $G_{CF}=(V_{CF},E_{CF})$ 和数据依赖图 $G_{DD}=(V_{DD},E_{DD})$, V_{CF}, E_{CF} 分别为控制流图的节点集合和有向边集合, V_{DD}, E_{DD} 分别为数据依赖图的节点集合和有向边集合. $G=(V,E)$ 为 C 的函数功能多重图. 其中,

- V 为节点集合, $V=V_{CF}$, 且 $V_{DD}\subseteq V$. 每个节点代表 C 中的一条语句.
- E 为图中有向边的集合. $e=(v_i,v_j,r)\in E$, v_i, v_j 为 V 中的任意两个节点. 其中, $r\in R=\{0,1\}$ 表示边 e 的类型. $r=0$ 表示边 e 的类型为控制流关系类型; $r=1$ 表示边 e 的类型为数据依赖关系类型.
  - 若有向边 $(v_i,v_j)\in E_{CF}$, 则 $(v_i,v_j,0)\in E$.
  - 若有向边 $(v_i,v_j)\in E_{DD}$, 且 $(v_i,v_j)\notin E_{CF}$, 则 $(v_i,v_j,1)\in E$.
  - 若有向边 $(v_i,v_j)\notin E_{CF}$, 则 $(v_i,v_j,0)\notin E$; 若有向边 $(v_i,v_j)\notin E_{DD}$, 则 $(v_i,v_j,1)\notin E$.

图 1(b) 展示了图 1(a) 的函数对应的函数功能多重图. 图中每个节点代表一个语句, 节点之间通过控制流边 (实线) 和数据依赖边 (虚线) 连接. 下面对一些特殊节点进行说明.

- 函数声明节点. 图 1(b) 中的顶端节点为函数声明节点. 函数的声明包含的函数名和参数列表提供了理解代码功能的信息, 因此本文将函数的声明作为控制流图的入口, 并与函数执行的第 1 个语句和其他有控制流关系的节点建立控制流边.
- 分支结构节点. 图 1(a) 的 if(f.isDirectory()) 判断条件语句, 在图 1(b) 的图中作为单独一个节点, 以便表达 if 条件与其块内语句的执行顺序. if(f.isDirectory()) 存在多条弧, 可以表示不同判断条件下的多条执行路径. 类似地, 其他分支结构 (elseif/else/switch)、循环结构 (for/for-each/while/do-while) 和特殊块 (synchronized /try/catch/finally) 的入口也单独作为一个节点.
- 循环结构节点. 图 1(b) 的 b = new byte[1024] 执行后返回到 while(fis.read(b)! = –1), 表示了循环结构, 其他循环结构也与 while 一致. 特别地, 图 1(a) 中的 for(int i = 0; i < fif.length; i++) 在图 1(b) 表示为 int i = 0、for(i<fif.length) 和 i++这 3 个节点, int i = 0 是 for(i<fif.length) 的前驱结点, i++是循环体内最后一个语句 writeZipFile(fif[i], zos, hierarchy) 的后继节点. 通过将 for(init_statement; test_statement; alter_statement) 拆分的方式表达了 init_statement、test_statement 和 alter_statement 的执行顺序, 是对 for 语句块的更细粒度的表示.
- 结束节点. 图 1(b) 还存在以“end-”开头的节点, 如 end-if、end-for, 这些节点在源代码中没有对应的语句, 用于显式表达控制结构和特殊块的结束, 包括 end-if、end-while、end-for、end-try、end-catch、end-finally、end-switch、end-synchronized.

对于数据依赖边, 图 1(b) 中 byte[] b = new byte[1024] 与 while (fis.read(b) != –1) 存在数据依赖, 但两个节点已经存在控制流边, 已经表达了这两个节点间的直接前驱后继的依赖关系. 因此不再增加数据依赖边. 本文对已存在控制流边连接的节点不再增加数据依赖边.

2.2 函数多重图的构建

根据定义 1, 函数多重图的构建可以以函数的 CFG 和 DDG 为基础, 如果两语句节点间不存在控制流边, 但存在数据依赖边, 则将数据依赖边添加到 CFG 中, 即可完成函数多重图的构建.

虽然 CFG 和 PDG (DDG) 早已成为代码分析与优化领域广泛应用的工具, 目前也存在针对不同语言的 CFG 和 PDG (DDG) 的构建工具. 但是受程序语言的不断发展、相关工具开发时目标各异等因素的影响, 目前的 CFG 和 DDG 构建工具仍然不完善, 比如针对 Python、JavaScript 等语言的 CFG 构建工具还不成熟, 更缺少 DDG 构建工具. 很多构建工具往往针对中间代码的表示 (intermediate code representation), 而不是面向源代码级别. 综合考虑多重图构建需求和本文使用的数据集, 本文使用 GitHub 中的开源工具 Progex (https://github.com/ghaffarian/progex) 生成代码的控制流图和数据依赖图作为基础, 通过将数据依赖图中的边添加到控制流图中, 实现函数功能

多重图的构建. 选择 Progex 基于以下原因: (1) Progex 是一个跨平台的开源工具, 可以在软件源代码级别提取诸如 CFG、PDG (DDG)、AST 等的多种程序表示. 其中, 其源代码级别构建 CFG 和 DDG 的特点, 正符合本文函数多重图的节点是源代码语句的定义; (2) Progex 以程序源代码文件作为输入, 可以将生成的各种程序表示导出为常见的文件格式. 导出常见的文件格式可以大量减少本文数据预处理的工作量.

由于两个语句间可以存在多个变量的依赖关系, Progex 生成的数据依赖图时, 对每个变量的依赖均生成一条以变量名为标签的边. 但本文使用数据依赖边是为了表达语句的上下文关联关系, 因此将两个语句间可能存在的多个依赖关系合并为一个依赖关系, 在多重图上表示为一条不带标签的数据依赖边. 如图 1(b) 所示的函数声明节点 void writeZipFile(File f, ZipOutputStream zos, String hierarchy) 与 ZipEntry ze = new ZipEntry(hierarchy + f.getName()) 节点间就存在两个变量的依赖关系, 图中仅使用一条边表示两节点间的数据依赖关系. 为了达到这个目的, 本文对 Progex 输出数据依赖图的源码进行了修改. 具体来说, 就是使用哈希表存储 Progex 算法生成的数据依赖图 (邻接表), 达到边去重的目的, 然后使用现有工具库直接转化为 JSON 格式文件输出.

函数功能多重图的构建如算法 1 所示. 算法以控制流图G_{CF}和数据依赖图G_{DD}作为输入. 第 3–11 行遍历控制流图所有的边, 并将每个节点和边添加到函数功能多重图G. 类似地, 第 12–16 行遍历了数据依赖图的所有边, 并将边添加到中G. 算法 1 遍历了控制流图和数据依赖图的所有边, 也即函数功能多重图的所有边, 故该算法的时间复杂度为$O(|E|)$. G存储了函数功能多重图所有节点和边, 故空间复杂度为$O(|V|+|E|)$.

算法 1. 构建函数功能多重图.

输入: 控制流图G_{CF}, 数据依赖图G_{DD};
输出: 函数功能多重图G.

1. **function buildFunctionMultigraph(**G_{CF}**,** G_{DD}**)**
2. 初始化存储函数功能多重图G为空邻接表;
3. **for** each edge $(v_i, v_j) \in E_{CF}$ **do**
4. **if** $v_i \notin V$ **then**
5. 添加节点v_i到V中;
6. **endif**
7. **if** $v_j \notin V$ **then**
8. 添加节点v_j到V中;
9. **endif**
10. 添加边$(v_i, v_j, 0)$到E中;
11. **endfor**
12. **for** each edge $(v_i, v_j) \in G_{DD}$ **do**
13. **if** $(v_i, v_j) \notin E_{CF}$ **then**
14. 添加边$(v_i, v_j, 1)$到E中;
15. **endif**
16. **endfor**
17. **return** G.

3 基于函数功能多重图嵌入的代码搜索

3.1 方法框架

本文提出的基于函数功能多重图嵌入的代码搜索方法 FuncGraphCS, 是将自然语言表述的查询与函数源代码

进行逐一匹配并排序的过程. 图 2 描述了 FuncGraphCS 方法的整体框架. 整个搜索过程主要包括 3 个步骤: 1) 函数特征提取; 2) 查询特征提取; 3) 查询和函数的匹配度计算并排序.

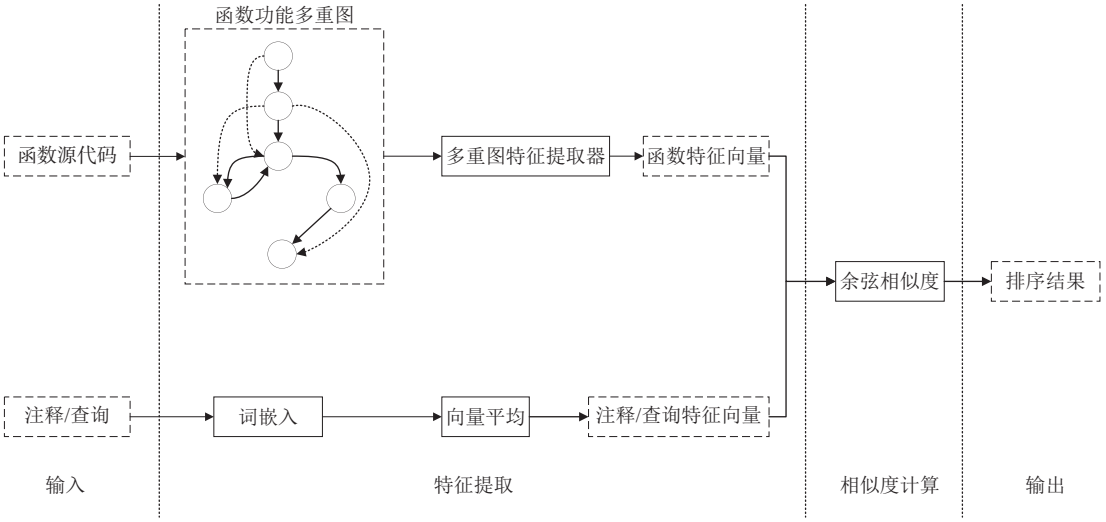


图 2 方法整体框架

在 FuncGraphCS 方法中, 每个函数代码被转换成一个多重图. 多重图融合了语句、控制流图和数据依赖 3 种模态的特征. 为了从多重图中提取函数特征, FuncGraphCS 设计一个多重图特征提取器来得到函数的特征向量. 在多重图特征提取器中, 首先使用词嵌入技术提取多重图中每个语句节点的特征, 然后使用关系图卷积网络提取语句节点的上下文特征, 以增强语句节点的特征, 最后将多重图中所有节点的特征融合, 得到表示函数的特征向量. 在训练和推理阶段, 多重图的特征提取方法是一致的. 代码库中函数代码的特征向量可以离线计算并存储, 以提高实时搜索的速度.

FuncGraphCS 采用自然语言处理的词嵌入技术提取用户查询的特征. 由于目前在代码搜索研究领域, 缺少查询-代码数据集. 因此, 在训练阶段, FuncGraphCS 以函数的文档注释代替用户查询. 具体地, FuncGraphCS 将文档注释分词得到词元序列, 再将每个词元的词向量平均, 得到注释的特征向量, 然后利用余弦相似度计算注释特征向量与函数代码特征向量的相似度, 并利用第 3.3 节中的损失函数提高两个向量的相似度. 在推理阶段, FuncGraphCS 利用训练好的注释特征提取模型, 提取查询的特征, 然后与代码库中的每一个函数特征向量计算相似度, 按照相似度从高到低返回查询结果.

3.2 函数特征提取

如前所述, 函数特征提取不仅要考虑每一条语句的特征提取, 还要考虑语句上下文的特征提取, 以及语句特征和上下文特征的融合以获取整个函数的特征. FuncGraphCS 将函数表示为多重图, 该表示模型表达了函数内语句及语句上下文 (所处的控制结构和数据依赖关系). 这样, 函数特征的提取转化为对多重图的特征提取. 图 3 描述了代码特征提取的整体流程. 具体地, 多重图的特征提取分为 3 个步骤: 1) 节点语句特征提取, 采用词嵌入技术提取代码的每个语句特征; 2) 节点语句上下文特征提取, 采用关系图卷积网络学习每个语句的上下文, 获得每个语句上下文语义增强的特征; 3) 图特征的生成, 融合每个节点增强特征, 得到整个图特征, 即函数特征. 下面详细介绍各步骤.

1) 节点语句特征提取

如图 3 的 Part1 所示, 每个节点以语句作为特征, 由于代码语句具有长度短的特点, 因此本文直接对每个语句分词, 得到词元序列, 并通过词嵌入技术将每个词元转化为向量, 再将词元序列的向量平均, 得到语句的特征向量. 本文使用该特征向量作为每个节点特征的初始值, 该特征向量仅包含语句本身特征, 不包含语句的上下文特征. 具体地, 记 $T=\langle t_1,t_2,\ldots,t_n\rangle$ 为分词后的语句的词元序列, 其中 n 为序列的长度, 则词元序列对应的词向量序列记为

$E_T = \langle e_{t_1}, e_{t_2}, \ldots, e_{t_n} \rangle$, 则语句的特征向量 e_{stmt} 为:

$$e_{stmt} = \frac{1}{n} \sum_{i=1}^{n} e_{t_i} \tag{1}$$

本文维护一个映射代码中的词元和向量的查找矩阵 M_C, 通过该表查找每个词元对应向量, 即 $e_{t_i} = lookup(t_i, M_C)$. 该矩阵在训练时首先随机初始化, 在训练中与其他参数一起更新. 该矩阵的大小是固定的.

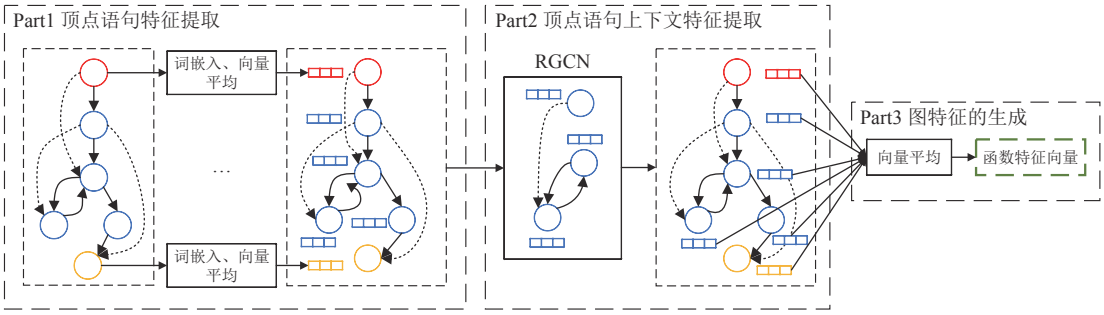


图 3 函数功能多重图的特征提取

2) 节点语句上下文特征提取

图神经网络在提取顶点的特征时可以通过聚合邻居节点的特征来更新本节点特征, 而函数中每个语句的功能与其上下文相关, 这种聚合邻居节点特征的特性正好可以提取语句的上下文特征. 具体地, 本文使用的函数功能多重图是包含两种不同类型边的有向异构图. 关系图卷积网络适用于具有多种边类型的有向异构图的特征提取, 该网络对不同类型的有向边设置不同的可训练参数, 通过这些参数聚合邻居节点的特征. 这样, 便可以综合邻居节点、边的类型和方向信息来提取节点的上下文特征. 关系图卷积网络的上述特性, 正好适用于本文的函数功能多重图. 因此, 本文选用关系图卷积网络作为图的特征提取器.

顶点语句上下文特征提取如图 3 的 Part2 所示. 关系图卷积网络的相关理论可以参考文献 [17]. 此处只对本文方法的一些设置进行说明, 未说明的设置与文献 [17] 一致. 若函数功能多重图中有 m 个节点, 则图中每个节点的初始特征向量为 $e_{stmt_i}, i = 1, 2, \ldots, m$, 即 $h_i^{(0)} = e_{stmt_i}$. 函数功能多重图中包含控制流边和数据依赖边, 即 R={0, 1}, 因而在关系图卷积网络的每一层中, 每个节点的特征通过这两种关系融合了其邻居节点的特征. 本文使用 2 层关系图卷积网络, 即 $L = 2$. 这里, 使用图深度学习框架 PyTorch Geometric (https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.RGCNConv) 提供的 RGCNConv 实现该网络. 此步骤后, 每个节点的特征向量包含了语句及其上下文特征.

3) 图特征的生成

如图 3 的 Part3 所示, 通过 Part2 得到每个节点的特征 $h_i^{(L)}$ 后, 本文通过对每个节点增强特征平均得到整个图的特征向量 h_{graph}, 该特征即函数特征:

$$h_{\text{graph}} = \frac{1}{m} \sum_{i=1}^{m} h_i^{(L)} \tag{2}$$

3.3 注释和查询特征提取与损失函数

在训练阶段本文使用每个函数的文档注释训练模型, 在推理阶段本文使用训练好的注释特征提取模型提取查询的特征.

如图 2 的下半部分所示, 注释/查询特征提取首先将注释或查询中每个词元转化为词向量, 再将所有的词向量平均得到特征向量. 具体地, 记注释或查询分词得到词元序列为 $Q = \langle q_1, q_2, \ldots, q_s \rangle$, 其中 s 为序列长度, 则词元序列对应的词向量序列为 $E_Q = \left\langle e_{q_1}, e_{q_2}, \ldots, e_{q_s} \right\rangle$, 则注释或查询的特征向量 h_Q 通过公式 (3) 得到:

$$h_Q = \frac{1}{s}\sum_{i=1}^{s} e_{q_i} \tag{3}$$

本文维护一个映射注释中的词元和向量的查找矩阵 M_Q, 通过该表查找每个词元对应的向量, 即 $e_{q_i} = lookup(q_i, M_Q)$. 该矩阵在训练时首先随机初始化, 在训练中与其他参数一起更新. 该矩阵的大小是固定的. 对于查询, 同样使用这个查找矩阵将词元转化为向量.

在训练阶段, 本文采用的损失函数如公式 (4) 所示, 其中 h_Q^+ 是代码对应的注释的特征向量, 即正样本, h_Q^- 是随机采样的注释特征向量列表, 其中 $h_{Q_j}^-$ 为第 j 个注释的特征向量. 具体地, 本文使用同一批次内其他代码的注释作为负样本列表. $sim(\cdot,\cdot)$ 为向量相似度计算函数, 本文使用余弦函数计算向量相似度. 训练目标是最小化公式 (4), 最小化公式 (4) 相当于最大化函数特征向量和其对应注释特征向量的相似度, 最小化与当前函数特征向量相似度最大的负样本注释特征向量的相似度. 本文采用 GitHub 上 codesnippetsearch 项目 (https://github.com/novoselrok/codesnippetsearch/blob/master/code_search/train.py) 提供的损失函数 (cosine_loss)实现.

$$loss = \max\left\{1 - sim\left(h_{\text{graph}}, h_Q^+\right) + \max_j sim\left(h_{\text{graph}}, h_{Q_j}^-\right), 0\right\} \tag{4}$$

4 实验分析

4.1 实验数据

实验数据来源于数据集 CodeSearchNet (https://github.com/github/CodeSearchNet)[36]. 该数据集包含了 Java、Python、Ruby、JavaScript、PHP、Go 这 6 种语言的代码搜索数据集. 由于除了 Java 语言, 其他语言没有合适的生成控制流图和数据依赖图的工具, 且其他 Java 语言的数据集已被预处理过, 无法解析为控制流图和数据依赖图, 因此本文只在 CodeSearchNet 的 Java 语言数据集上实验. 该数据集中存在一部分函数无法解析, 本文去除这部分数据. 具体的预处理规则及原因如下.

(1) 删除注释为非英文的样本: 本文仅处理使用英文撰写的查询.

(2) 删除注释长度小于 3 的样本: 本文认为注释长度小于 3 个词的难以准确表达函数的功能.

(3) 删除无法解析出控制流图和数据依赖的样本: 这类样本可能存在语法错误, 导致工具解析错误.

(4) 删除图节点数量大于 500 的样本: 去除这类样本主要是为了提高算法运行效率.

(5) 删除图节点数量为 0 的样本: 节点数量为 0 表明该函数方法体为空.

(6) 每个函数去除代码中的诸如 public、static 等关键字: 这些关键字广泛存在于代码中, 对区分不同代码的功能没有帮助.

(7) 将驼峰命名和蛇形命名风格的函数、变量、语句拆分为词元: 这两种风格的命名方式使用多个词元构造, 拆分有助于减小词表.

(8) 去除代码文本中的“.”“[]”等符号和整数、浮点数: 具体的数值存储在词表中时无法比较大小, 因此没有语义信息.

(9) 忽略函数中 Lambda 表达式等 Java JDK1.7 以后版本的新特性: 控制流图和数据依赖图生成工具 Progex 不支持 lambda 表达式等 JDK1.7 以后版本新特性的处理.

表 2 给出了处理后的数据集的样本数量和占比, 其中训练集、验证集、测试集是数据集发布者预先划分的. 后续所有实验均在该数据集上进行.

表 2 实验数据集

数据集	样本数量 (占比)
训练集	393 008 (91.64%)
验证集	12 608 (2.94%)
测试集	23 251 (5.42%)

4.2 评价指标及对比模型

由于代码搜索的目标是使与用户搜索意图相关性越大的结果排在结果列表越前面, 因此本文选择使用 *MRR* (mean reciprocal rank) 作为评价指标, 其计算公式如公式 (5):

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i} \tag{5}$$

其中, $|Q|$为测试集中查询的数量, 对于第 i 个查询, 该查询最相关的结果在结果列表的位置为 $rank_i$. 若最相关的结果排在越前面, 则 *MRR* 指标越高.

此外, 本文还使用准确率 *ACC* 作为评价指标. 在搜索场景下, 本文使用 *ACC*@1、*ACC*@5 和 *ACC*@10 作为指标. *ACC*@k 的含义是若答案在排在结果列表的前 k 位 (包含 k), 则计数器 *count* 加 1, 否则加 0, 然后将 *count* 除以查询的总数$|Q|$得到 Top k 准确率 ($count \leqslant |Q|$). 其计算公式如下:

$$ACC@k = \frac{count}{|Q|} \tag{6}$$

其中, *count* 为答案排在结果列表前 k 位 (包含 k) 的查询的数量. 可以看到, 该指标不关心结果在列表中的位置.

由于许多方法没有开源代码, 因此本文主要与公开了代码的方法比较. 具体地, 本文将 FuncGraphCS 与 NCS、DeepCS、NBoW、MRNCS、MMAN、DGMS 比较. 其中, NCS、DeepCS、NBoW 仅使用代码文本作为特征. NCS 使用 FastText 算法无监督训练得到的词向量搜索, DeepCS 使用方法名、API 调用序列和代码文本作为特征, NBoW 分别将代码和注释分词得到词元序列, 并将所有词元的词向量平均得到代码特征向量和注释特征向量. NBoW 是当前仅使用代码文本特征的方法中准确度最高的; MRNCS 融合了代码文本、简化语法树两种特征, 采用了 4 种树序列化方法. 实验中, 本文对比在 MRNCS 论文中准确度最高的 MM-SBT 方法; MMAN 方法融合了代码文本、抽象语法树、控制流图 3 种特征, 其中抽象语法树表示了代码的语法特征, 控制流图的每个节点代表一个语句. MMAN 和 MRNCS 通过中间融合策略融合多模态特征; DGMS 使用增强的抽象语法树表示代码, 使用成分句法解析树表示查询, 并使用 RGCN 提取函数和查询的特征, 同时使用注意力机制实现函数和查询的特征交互.

4.3 实验方法与参数设置

本文使用第 4.1 节中处理得到的训练集训练. 在训练过程中, 将每轮训练得到的模型在验证集中计算 *MRR* 指标, 保存 *MRR* 最高的模型, 再使用测试集对该模型进行测试. 本文测试时, 在测试集中随机选择 1 999 个函数样本, 加上该查询的答案, 组成大小为 2 000 的代码库用作实验评估.

训练时, 语句和注释的词向量的维度均为 128, 词表大小为 10 000, 即 $M_C \in \mathbb{R}^{10000\times128}, M_Q \in \mathbb{R}^{10000\times128}$. 关系图卷积网络中的隐藏层的维度为 256. 优化器选用 Adam[37], 学习率为 0.01, 训练轮数为 100. 所有实验均将随机数种子固定为 123 456. 代码语句最大长度设置为 15, 注释最大长度设置为 35, 超过最大长度的截断, 不足最大长度的填充“__pad”到最大长度. 本文使用 BPE 算法[38]构建词表并对语句进行分词, BPE 算法采用 CodeSearchNet (https://github.com/github/CodeSearchNet/blob/master/src/utils/bpevocabulary.py) 提供的实现. 这里, 词表规模和构建算法与实验对比方法 MRNCS、NBoW 设置相同.

推理时, 由于每个代码语句和注释的特征向量都是通过词向量平均的方式获得, 且推理时不需要像训练时构造批次, 代码语句和注释的最大长度可以不设置限制. 但在测试时本文仍将测试数据构造为批次, 因此沿用训练时的语句和注释长度设置.

本文的实验代码均使用 Python 3.7 开发. 本文的模型使用 PyTorch Geometric 2.0.2 (PyG) 实现. 实验对比方法中, NCS 使用 PyTorch 1.8.1 实现; DeepCS 和 MRNCS 使用了原作者的实现; NBoW 采用了 MRNCS 作者提供的实现; DGMS 也使用了 PyTorch Geometric 2.0.2 (PyG) 实现; MMAN 则使用了 DGL 0.7.2 实现. 实验环境如后文表 3 所示.

4.4 实验结果与分析

为了评估基于函数功能多重图嵌入代码搜索方法 FuncGraphCS 的有效性, 本文研究了以下 3 个问题.

• RQ1: FuncGraphCS 使用函数功能多重图表示函数, 是否比现有的基于文本特征、结构特征的深度学习方法准确度更高?

• RQ2: 控制流图是否有助于提取更准确的代码特征? 数据依赖图是否可以辅助控制流图更好地表示代码功能语义?

• RQ3: FuncGraphCS 使用早期融合的策略融合多模态特征, 与中间融合的策略相比如何?

RQ1: FuncGraphCS 使用函数功能多重图表示函数, 是否比现有的基于文本特征、结构特征的深度学习方法准确度更高?

为了验证这个问题的结果, 本文将 FuncGraphCS 与 NCS、DeepCS、NBoW 这 3 个基于文本特征的方法和 MMAN、MRNCS (MM-SBT)、DGMS 这 3 个基于结构特征的方法对比. 实验结果如表 4 所示 (加粗表示取得最好的结果).

表 3 实验环境清单

环境	型号
操作系统	Ubuntu 16.04
CPU	Intel(R) Xeon(R) CPU E5-2620 v4 @ 2.10 GHz
GPU	4× GeForce RTX 2080 Ti Rev. A
内存	64 GB

表 4 与现有基于代码文本、结构特征的方法的对比结果

方法	*MRR*	*ACC*@1	*ACC*@5	*ACC*@10
NCS	0.3667	0.2884	0.4543	0.4548
DeepCS	0.4607	0.3575	0.5793	0.6588
NBoW	0.5436	0.4471	0.6603	0.7245
MMAN	0.4942	0.3811	0.6294	0.7187
DGMS	0.4651	0.3395	0.6129	0.7056
MRNCS (MM-SBT)	0.5960	0.5030	0.7061	0.7678
FuncGraphCS (ours)	**0.6301**	**0.5249**	**0.7580**	**0.8282**

可以看到, FuncGraphCS 的 *MRR* 分别超过 NCS、DeepCS、NBoW 71.8%、36.8%、15.9%. 注意, FuncGraphCS 仅将 NBoW 模型的代码特征从代码文本换为函数功能多重图, 这表明函数功能多重图表示的上下文信息可以有效提高代码搜索的准确度.

同时, FuncGraphCS 的 *MRR* 分别超过 MRNCS (MM-SBT)、MMAN、DGMS 5.7%、27.5%、35.5%. 与 MRNCS (MM-SBT)、MMAN 和 DGMS 等基于结构特征的方法相比, FuncGraphCS 的方法取得了更高的准确度. 这表明函数功能多重图表示的上下文信息对代码搜索更重要.

注意到 MMAN 和 DGMS 的 *MRR* 与 NBoW 差距较大, 这两个方法在代码文本特征的基础上还增加了结构特征, 提供了更多的信息, 但 *MRR* 反而较低. 本文认为可能的原因是 MMAN 和 DGMS 的特征提取方法不适用于其表示模型. 表示模型与特征提取方法的适配仍旧是一个值得探讨的问题.

总的来说, 与典型的基于文本特征和结构特征的深度学习方法相比, 本文使用的函数功能多重图表示的上下文信息可以有效提高代码搜索的准确度, 这些上下文信息对代码搜索更重要.

RQ2: 控制流图是否有助于提取更准确的代码特征? 数据依赖图是否可以辅助控制流图更好地表示代码功能语义?

为了验证这个问题的结果, 本文比较仅使用控制流图 (FuncGraphCS-cfg)、仅使用数据依赖图 (FuncGraphCS-dd)、同时使用控制流图和数据依赖图 (FuncGraphCS) 的结果. 实验结果如表 5 所示.

从表 5 的实验结果可以看到, FuncGraphCS-cfg 相比表 4 中基于抽象语法树的方法 (MMAN、MRNCS) *MRR* 提升较大. 这表明, 如果采用合适的特征提取方法, 控制流图有助于提取更准确的代码特征.

表 5 与仅使用控制流图、数据依赖的方法的对比结果

方法	*MRR*	*ACC*@1	*ACC*@5	*ACC*@10
FuncGraphCS-cfg (ours)	0.6236	0.5220	0.7501	0.8210
FuncGraphCS-dd (ours)	0.5890	0.4806	0.7185	0.7897
FuncGraphCS (ours)	**0.6301**	**0.5249**	**0.7580**	**0.8282**

从表 5 也可以看到 FuncGraphCS 的 *MRR* 高于 FuncGraphCS-cfg, 但提升轻微, 表明 FuncGraphCS-cfg 在多重图中的贡献是主要的. 同时也表明数据依赖关系在多重图中贡献不大. 但对比表 4 的数据, FuncGraphCS-dd 的表现仍好于除 FuncGraphCS 和 MRNCS (MM-SBT) 外的其它方法, 说明数据依赖信息对代码搜索还是有较好效果的. 和多重图的基本设计思想——将数据依赖图作为对控制流图的一种补充, 仅将控制流图中无法表示的非直接前驱后继节点的依赖关系进行补充——是一致的. 而且 FuncGraphCS 表现好于 FuncGraphCS-cfg, 也说明数据依赖图表示的上下文信息可以辅助控制流图更好地表示功能语义, 但贡献有限.

RQ3: FuncGraphCS 使用早期融合的策略融合多模态特征, 与中间融合的策略相比如何?

为了验证这个问题的结果, 本文分别比较使用向量相加、向量拼接融合控制流图和数据依赖图的方法, 分别记为 FuncGraphCS-sum、FuncGraphCS-concat. 实验结果如表 6 所示.

表 6　不同多模态特征融合策略的对比结果

方法	*MRR*	*ACC*@1	*ACC*@5	*ACC*@10
FuncGraphCS-sum (ours)	0.626 0	0.515 8	0.755 9	**0.829 0**
FuncGraphCS-concat (ours)	0.619 9	0.513 3	0.753 2	0.823 8
FuncGraphCS (ours)	**0.630 1**	**0.524 9**	**0.758 0**	0.828 2

从表 6 可以看出, 相比向量拼接 (FuncGraphCS-concat) 的中间融合策略, 本文的早期融合策略 *MRR* 更高, 相比向量相加 (FuncGraphCS-sum) 的中期融合策略则差距很小. 对比 NBoW 和 FuncGraphCS, 发现 NBoW 仅使用代码文本特征的 *MRR* 就已经达到了 0.543 6. 这说明在代码搜索任务中, 语句的文本特征是代码特征最重要的部分. 表示语句上下文的结构特征可以提高搜索准确度, 但贡献比文本特征低. 而不同的多模态特征融合策略仅影响到不同结构特征的融合, 因此对搜索准确度影响不大.

5 总　结

本文提出了一个基于函数功能多重图嵌入的代码搜索方法. 在这个方法中, 本文使用早期融合的策略, 将代码语句的数据依赖关系融合到控制流图中, 构建函数功能多重图来表示代码. 该多重图通过数据依赖关系显式表达了控制流图中缺乏的非直接前驱后继节点的依赖关系, 增强了语句节点的上下文信息. 同时, 针对多重图的异质特性, 本文采用关系图卷积网络提取函数代码的特征.

实验表明, 本文方法的 *MRR* 比典型的基于文本特征和结构特征的深度学习方法高 5% 以上, 表明函数功能多重图表示的上下文信息可以有效提高代码搜索的准确度, 这些上下文信息对代码搜索更重要. 同时, 消融实验也表明了控制流图有助于提取更准确的代码特征, 数据依赖图可以辅助控制流图更好地表示功能语义. 最后, 本文还分析了中间融合策略与早期融合策略对搜索准确度的影响, 由于代码文本特征是代码特征最重要的部分, 不同的多模态特征融合策略仅影响到不同结构特征的融合, 因此对代码搜索准确度的影响不大.

由于目前没有查询-代码数据集, 因此本文使用注释-代码数据集训练, 但注释和查询不同, 注释是开发者对函数所表达的功能和相关细节的说明, 一般由开发者本人编写, 其表述一般能较准确地表达函数的特征, 而用户查询的表述相对随意模糊, 歧义较大. 这样, 导致基于注释数据训练得到的模型与实际应用场景要求存在差距. 对于这个问题, 本文认为有两个可行的思路, 一是运营大型代码版本管理仓库的相关企业 (如 GitHub、Gitee) 根据其查询日志整理并通过众包的方式标注; 二是整理小规模的注释-查询数据集, 通过训练文本改写模型的方式生成与查询表达相近的数据集. 与真实应用场景相符的数据集是促进代码搜索领域发展的重要基础, 是值得研究的重点问题之一.

References:

[1] Liu C, Xia X, Lo D, Gao CY, Yang XH, Grundy J. Opportunities and challenges in code search tools. ACM Computing Surveys, 2021, 54(9): 196. [doi: 10.1145/3480027]

[2] Wei M, Zhang LP. Research progress of code search methods. Application Research of Computers, 2021, 38(11): 3215–3221, 3230 (in Chinese with English abstract). [doi: 10.19734/j.issn.1001-3695.2021.04.0096]

[3] Li JJ, Liang ZY, Wei T, Mao J. A malicious behavior analysis method based on program semantic. Acta Scientiarum Naturalium Universitatis Pekinensis, 2008, 44(4): 537–542 (in Chinese with English abstract). [doi: 10.13209/j.0479-8023.2008.084]

[4] Linstead E, Bajracharya S, Ngo T, Rigor P, Lopes C, Baldi P. Sourcerer: Mining and searching internet-scale software repositories. Data Mining and Knowledge Discovery, 2009, 18(2): 300–336. [doi: 10.1007/s10618-008-0118-x]

[5] Martie L, LaToza TD, van der Hoek A. CodeExchange: Supporting reformulation of internet-scale code queries in context (T). In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Lincoln: IEEE, 2015. 24–35. [doi: 10.1109/ASE.2015.51]

[6] Zhang F, Niu HR, Keivanloo I, Zou Y. Expanding queries for code search using semantically related API class-names. IEEE Trans. on Software Engineering, 2018, 44(11): 1070–1082. [doi: 10.1109/TSE.2017.2750682]

[7] Gu XD, Zhang HY, Kim S. Deep code search. In: Proc. of the 40th Int'l Conf. on Software Engineering. Gothenburg: ACM, 2018. 933–944. [doi: 10.1145/3180155.3180167]

[8] Cambronero J, Li HY, Kim S, Sen K, Chandra S. When deep learning met code search. In: Proc. of the 27th ACM Joint Meeting on European Software Engineering Conf. and Symp. on the Foundations of Software Engineering. Tallinn: ACM, 2019. 964–974. [doi: 10.1145/3338906.3340458]

[9] Shuai JH, Xu L, Liu C, Yan M, Xia X, Lei Y. Improving code search with co-attentive representation learning. In: Proc. of the 28th Int'l Conf. on Program Comprehension. Seoul: ACM, 2020. 196–207. [doi: 10.1145/3387904.3389269]

[10] Wan Y, Shu JD, Sui YL, Xu GD, Zhao Z, Wu J, Yu P. Multi-modal attention network learning for semantic source code retrieval. In: Proc. of the 34th IEEE/ACM Int'l Conf. on Automated Software Engineering. San Diego: IEEE, 2019. 13–25. [doi: 10.1109/ASE.2019.00012]

[11] Gu WC, Li ZJ, Gao CY, Wang CZ, Zhang HY, Xu ZL, Lyu MR. CRaDLe: Deep code retrieval based on semantic dependency learning. Neural Networks, 2021, 141: 385–394. [doi: 10.1016/j.neunet.2021.04.019]

[12] Gu J, Chen ZM, Martin M. Multimodal representation for neural code search. In: Proc. of the 2021 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Luxembourg: IEEE, 2021. 483–494. [doi: 10.1109/ICSME52107.2021.00049]

[13] Allen FE. Control flow analysis. ACM SIGPLAN Notices, 1970, 5(7): 1–19. [doi: 10.1145/390013.808479]

[14] Ferrante J, Ottenstein KJ, Warren JD. The program dependence graph and its use in optimization. ACM Trans. on Programming Languages and Systems, 1987, 9(3): 319–349. [doi: 10.1145/24039.24041]

[15] Huo X, Li M, Zhou ZH. Control flow graph embedding based on multi-instance decomposition for bug localization. Proc. of the AAAI Conf. on Artificial Intelligence, 2020, 34(4): 4223–4230. [doi: 10.1609/aaai.v34i04.5844]

[16] Ramachandram D, Taylor GW. Deep multimodal learning: A survey on recent advances and trends. IEEE Signal Processing Magazine, 2017, 34(6): 96–108. [doi: 10.1109/MSP.2017.2738401]

[17] Schlichtkrull M, Kipf TN, Bloem P, van den Berg R, Titov I, Welling M. Modeling relational data with graph convolutional networks. In: Proc. of the 15th Int'l Conf. on the Semantic Web. Heraklion: Springer, 2018. 593–607. [doi: 10.1007/978-3-319-93417-4_38]

[18] Wilde N, Huitt R, Huitt S. Dependency analysis tools: Reusable components for software maintenance. In: Proc. of the 1989 Conf. on Software Maintenance. Miami: IEEE, 1989. 126–131. [doi: 10.1109/ICSM.1989.65203]

[19] Page L, Brin S, Motwani R, Winograd T. The PageRank citation ranking: Bringing order to the Web. Technical Report, Stanford: Stanford University, 1998.

[20] Lv F, Zhang HY, Lou JG, Wang SW, Zhang DM, Zhao JJ. CodeHow: Effective code search based on api understanding and extended boolean model (E). In: Proc. of the 30th IEEE/ACM Int'l Conf. on Automated Software Engineering (ASE). Lincoln: IEEE, 2015. 260–270. [doi: 10.1109/ASE.2015.42]

[21] Rahman MM, Roy C. Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics. In: Proc. of the 2018 IEEE Int'l Conf. on Software Maintenance and Evolution (ICSME). Madrid: IEEE, 2018. 473–484. [doi: 10.1109/ICSME.2018.00057]

[22] Nie LM, Jiang H, Ren ZL, Sun ZY, Li XC. Query expansion based on crowd knowledge for code search. IEEE Trans. on Services Computing, 2016, 9(5): 771–783. [doi: 10.1109/TSC.2016.2560165]

[23] Huang Q, Wu HG. QE-integrating framework based on GitHub knowledge and SVM ranking. Science China Information Sciences, 2019, 62(5): 52102. [doi: 10.1007/s11432-017-9465-9]

[24] Li X, Wang QX, Jin Z. Description reinforcement based code search. Ruan Jian Xue Bao/Journal of Software, 2017, 28(6): 1405–1417 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5226.htm [doi: 10.13328/j.cnki.jos.005226]

[25] Sachdev S, Li HY, Luan SF, Kim S, Sen K, Chandra S. Retrieval on source code: A neural code search. In: Proc. of the 2nd ACM SIGPLAN Int'l Workshop on Machine Learning and Programming Languages. Philadelphia: ACM, 2018. 31–41. [doi: 10.1145/3211346.3211353]

[26] Fang S, Tan YS, Zhang T, Liu YP. Self-attention networks for code search. Information and Software Technology, 2021, 134: 106542. [doi: 10.1016/j.infsof.2021.106542]

[27] Feng ZY, Guo DY, Tang DY, Duan N, Feng XC, Gong M, Shou LJ, Qin B, Liu T, Jiang DX, Zhou M. CodeBERT: A pre-trained model for programming and natural languages. arXiv:2002.08155v4, 2020.

[28] Guo DY, Ren S, Lu S, Feng ZY, Tang DY, Liu SJ, Zhou L, Duan N, Svyatkovskiy A, Fu SY, Tufano M, Deng SK, Clement C, Drain D, Sundaresan N, Yin J, Jiang DX, Zhou M. GraphCodeBERT: Pre-training code representations with data flow. arXiv:2009.08366, 2021.

[29] Tai KS, Socher R, Manning CD. Improved semantic representations from tree-structured long short-term memory networks. In: Proc. of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th Int'l Joint Conf. on Natural Language Processing (Vol. 1: Long Papers). Beijing: ACL, 2015. 1556–1566. [doi: 10.3115/v1/P15-1150]

[30] Li YJ, Tarlow D, Brockschmidt M, Zemel R. Gated graph sequence neural networks. arXiv:1511.05493v4, 2017.

[31] Ling X, Wu LF, Wang SZ, Pan GN, Ma TF, Xu FL, Liu AX, Wu CM, Ji SL. Deep graph matching and searching for semantic code retrieval. ACM Trans. on Knowledge Discovery from Data, 2021, 15(5): 88. [doi: 10.1145/3447571]

[32] Allamanis M, Brockschmidt M, Khademi M. Learning to represent programs with graphs. arXiv:1711.00740v3, 2018.

[33] Ling CY, Zou YZ, Lin ZQ, Xie B, Zhao JF. Approach to searching software source code with graph embedding. Ruan Jian Xue Bao/Journal of Software, 2019, 30(5): 1481−1497 (in Chinese with English abstract). http://www.jos.org.cn/1000-9825/5721.htm [doi: 10.13328/j.cnki.jos.005721]

[34] Huang SY, Zhao YH, Liang YM. Code search combining graph embedding and attention mechanism. Journal of Frontiers of Computer Science and Technology, 2022, 16(4): 844–854 (in Chinese with English abstract). [doi: 10.3778/j.issn.1673-9418.2010087]

[35] Tang J, Qu M, Wang MZ, Zhang M, Yan J, Mei QZ. Line: Large-scale information network embedding. In: Proc. of the 24th Int'l Conf. on World Wide Web. Florence: International World Wide Web Conferences Steering Committee, 2015. 1067–1077. [doi: 10.1145/2736277.2741093]

[36] Husain H, Wu HH, Gazit T, Allamanis M, Brockschmidt M. CodeSearchNet challenge: Evaluating the state of semantic code search. arXiv:1909.09436, 2020.

[37] Kingma DP, Ba J. Adam: A method for stochastic optimization. arXiv:1412.6980, 2017.

[38] Sennrich R, Haddow B, Birch A. Neural machine translation of rare words with subword units. In: Proc. of the 54th Annual Meeting of the Association for Computational Linguistics (Vol. 1: Long Papers). Berlin: ACL, 2016. 1715–1725. [doi: 10.18653/v1/P16-1162]

附中文参考文献:

[2] 魏敏, 张丽萍. 代码搜索方法研究进展. 计算机应用研究, 2021, 38(11): 3215–3221, 3230. [doi: 10.19734/j.issn.1001-3695.2021.04.0096]

[3] 李佳静, 梁知音, 韦韬, 毛剑. 一种基于语义的恶意行为分析方法. 北京大学学报(自然科学版), 2008, 44(4): 537–542. [doi: 10.13209/j.0479-8023.2008.084]

[24] 黎宣, 王千祥, 金芝. 基于增强描述的代码搜索方法. 软件学报, 2017, 28(6): 1405–1417. http://www.jos.org.cn/1000-9825/5226.htm [doi: 10.13328/j.cnki.jos.005226]

[33] 凌春阳, 邹艳珍, 林泽琦, 谢冰, 赵俊峰. 基于图嵌入的软件项目源代码检索方法. 软件学报, 2019, 30(5): 1481−1497. http://www.jos.org.cn/1000-9825/5721.htm [doi: 10.13328/j.cnki.jos.005721]

[34] 黄思远, 赵宇海, 梁燚铭. 融合图嵌入和注意力机制的代码搜索. 计算机科学与探索, 2022, 16(4): 844–854. [doi: 10.3778/j.issn.1673-9418.2010087]

徐杨(1970－), 男, 博士, 讲师, CCF 专业会员, 主要研究领域为智能化软件工程, 机器学习, 分布式计算.

汤德佑(1976－), 男, 博士, 副教授, CCF 专业会员, 主要研究领域为数据库, 高性能计算, 软件优化.

陈晓杰(1995－), 男, 硕士, 主要研究领域为深度学习, 智能化软件工程.

黄翰(1980－), 男, 博士, 教授, 博士生导师, CCF 杰出会员, 主要研究领域为微计算方法的理论基础及应用, 智能化软件工程.