華中科技大學

图神经网络实验报告

专　　业：

班　　级：

学　　号：

姓　　名：

电　　话：

邮　　箱：

独创性声明

本人郑重声明本报告内容，是由作者本人独立完成的。有关观点、方法、数据和文献等的引用已在文中指出。除文中已注明引用的内容外，本报告不包含任何其他个人或集体已经公开发表的作品成果，不存在剽窃、抄袭行为。

特此声明！

作者签名：

日期：　年　月　　日

成　　绩	
教师签名	

目 录

1 概述

本课程选择的任务是“基于图神经网络的代码漏洞检测”，漏洞检测是保护软件系统免受网络安全攻击的一项重要技术，通过在源代码中定位易受攻击的函数来进行代码防护。当前，安全专家主要利用动态模糊测试、符号执行和静态代码审计来发现漏洞。然而，这些技术各有局限性，难以提供全面有效的解决方案。例如，动态模糊测试面临着代码覆盖不足和初始测试种子选择问题，这可能导致重要漏洞未被发现。符号执行因路径爆炸和复杂的约束求解问题难以扩展到实际的复杂程序中，导致其在大规模应用场景中表现有限。静态代码审计虽然不依赖运行环境，可以发现潜在的逻辑漏洞，但通常需要人类专家的专业知识，且随着程序复杂度的提升，其审计效率和准确性也会显著下降。

随着深度学习的快速发展，很多研究开始探讨神经网络在自动化漏洞识别上的潜力，在学习方法上，将源代码视为与自然语言类似的序列进行处理。然而，源代码比自然语言更讲究结构性和逻辑性，并且存在诸多异质表示方式（例如抽象语法树、数据流、控制流等）。这里介绍一些常见的源代码特征表示方法。

AST（抽象语法树）是一种用树状结构表示程序代码语法结构的数据结构。在 AST 中，每个节点代表程序的一个语法元素，AST 的结构会忽略源代码中不影响语法的元素（如空格、注释），专注于程序的语法和逻辑结构，使其比原始代码更简洁和抽象。

CFG 图（控制流图）是一种用图结构来表示程序控制流的工具，主要用于编译器、静态分析和代码优化中。CFG 展示了程序中各个基本块（Basic Block）的执行顺序以及控制流之间的跳转关系。节点表示程序中的一个基本块，边表示控制流的可能路径。

PDG（程序依赖图）是一种表示程序代码中语句和操作间依赖关系的图形结构。它通过节点和边来表示程序的控制依赖和数据依赖。控制依赖表示某些代码执行的条件依赖于其他代码的执行结果；数据依赖则表明某些变量或数据依赖于前面代码的计算结果。

DFG（数据流图）是一种用于表示程序中数据依赖关系的图结构。它专注于描述程序中各个操作之间的数据传递关系，图中的节点代表计算操作，边表示数据依赖关系，指示数据从一个操作或变量传递到下一个操作。

这些特征表示以图作为数据结构，因此采用图神经网络作为深度学习框架进行建模，通过消息传播进行特征聚合，能够细粒度的完成类型判断、缺陷检测等任务。基于此，本实验采用一种基于图神经网络的漏洞检测模型 Devign[1]，将漏洞检测设计为图分类任务。通过将源代码编码成具有综合语义的联合图结构，使用门控图神经网络进行消息传播，进而捕获高阶特征，通过卷积池化对图进行分类，完成源代码中敏感函数的预测。

2 背景知识

图神经网络（GNN）是用来处理图类型数据的神经网络，GNN 的目标是学习每个节点 v 的表示，而每个节点的表示由该节点的特征、与该节点连接的边的特征、该节点的邻居表示和它邻居节点的特征计算得到。GNN 的核心思想是通过消息传递机制对图中的节点进行嵌入表示的学习。在每一层中，节点通过与其邻居节点的特征交互来更新自己的表示，使得经过多层消息传递后，节点的表示能够捕捉到远距离的邻居信息。关于 GNN 在图数据上的常见任务有链路预测、节点分类、图生成等。

图神经网络的研究与图嵌入技术息息相关，图嵌入是将图（Graph）结构数据映射到低维向量空间的技术，目标是将图的结构信息和节点特征转化为低维向量表示，使得在嵌入空间中，相似的节点或结构在空间中彼此接近。传统的图嵌入算法主要有以下几种：（1）基于矩阵分解的方法，通过分解邻接矩阵、拉普拉斯矩阵或一些相似矩阵来获得低维表示；（2）基于随机游走的方法：利用随机游走生成节点序列，并将图嵌入问题转化为类似于词嵌入的任务。通过在随机游走路径上生成的节点序列，可以捕获节点之间的局部和全局关系；（3）基于无监督或对比学习的方法：通过对比学习或无监督学习方式，学习到节点或子图的结构表示。而图神经网络通过信息传递机制（Message Passing）来学习节点的表示，使得节点嵌入能够综合邻居节点的信息。GNN 方法通常具有很强的表达能力，适合处理大规模图数据。

关于 GNN 的应用场景非常广泛，它可以用来进行社交网络分析，也可以在推荐领域中挖掘用户和商品的协同关系，还可以对知识图谱执行实体分类、关系预测等任务，除此之外，图神经网络在生物智能、代码智能等交叉领域也有很多应用。尽管 GNN 在图结构数据上表现优异，但仍面临一些挑战。首先是计算复杂度问题，随着图规模的增大，模型的计算开销显著增加；其次是过平滑问题，在深层网络中节点表示趋于相同，导致信息丢失。此外，图的异质性和动态性的建模仍然是一个开放性问题。

随着大语言模型（Large Language Model,LLM）的兴起，LLM 凭借其良好的语义理解能力和文本生成能力在各个领域均得到了广泛应用，图和大模型的结合也是目前图神经网络研究中一个重要的分支。目前图大模型的范式可以大致分为以下四种：（1）LLMs as Prefix: 将大模型作为 GNN 的增强器，通过利用 LLM 良好的语义理解和文本生成能力，为 Graph 提供语义信号从而增强图学习和推理。（2）GNN as Prefix: 利用 GNN 良好的捕捉和理解结构信息的能力，为大模型提供协同信号，从而增强大模型推理能力。（3）LLM-Graph Integration：通过联合训练、对比学习等方式连接图的协同信号和大模型的语义信号。（4）Only LLM：通过提示工程或指令微调的方式，借助自然语言/图的文本表示/编程语言来表示图，让大模型更好的理解图的信息。

由于我目前的科研工作主要关注图和大模型的结合，下面我将根据我所接触到的一些新颖的技术方案对这些知识进行归纳总结。

2.1 Graph-LLM For RecSys

在推荐领域中，图神经网络常常用来处理复杂的用户-物品交互数据，挖掘用户和商品之间的关系。基于图神经网络的协同推荐范式往往存在可解释性差、冷启动推荐等问题，此外，由于数据的稀疏性和噪声带来的影响，图推荐无法避免长尾效应，推荐过程中存在流行度偏差。为了更好地解决图推荐中存在的问题，现在许多工作利用大模型良好的语义理解和文本生成能力，将大模型和图神经网络结合进行推荐。

范式一：LLMs as Prefix

这里以 Wei W 等人的 LLMRec[2]工作为例进行说明。该工作聚焦在解决侧信息用来辅助推荐时出现的数据噪声、数据异构型以及数据不完整性问题，提出了采用三种图增强策略的 LLMRec 框架。图 2-1 为 LLMRec 框架图。

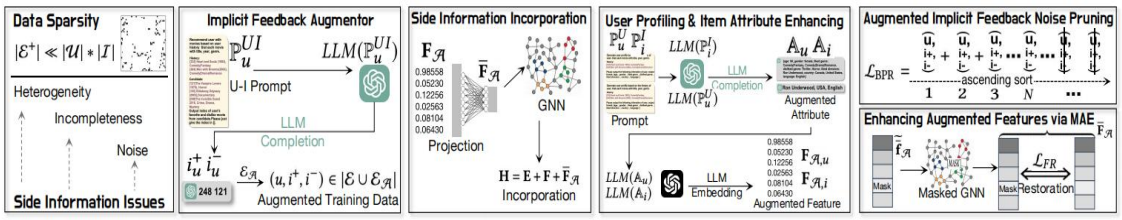

图 2-1 LLMRec 模型框架图

在数据增强方面，使用 LLM 利用用户和商品的文本信息生成语义特征，并通过提示工程的方式为每个用户进行正负商品采样，得到交互增广子图。为了将语义特征映射到图神经网络的协同空间，作者引入了一个可学习的 MLP 映射器。然后作者基于语义特征和采样得到的增广图开展了两个图降噪学习任务：一个是根据正负样本对计算 bpr 损失，进而去除噪音边；另一个是通过掩码生成任务进行特征增强。

范式二：GNN as Prefix

这里我以 Zhang Y 等人的 BinLLM[3]工作为例。在将 LLM 适配于推荐系统时，整合协同信息至关重要。该方法通过类似文本的编码方式，以一种精简的方式整合协同信息。BinLLM 将来自外部模型的协同嵌入转换为二进制序列，这是一种 LLMs 能够直接理解和操作的特定文本格式，从而实现了 LLMs 直接使用类似文本格式的协同信息。此外，BinLLM 还提供了使用点十进制表示法压缩二进制序列的选项，以避免过长的长度。图 2-2 为 BinLLM 的模型框架图。

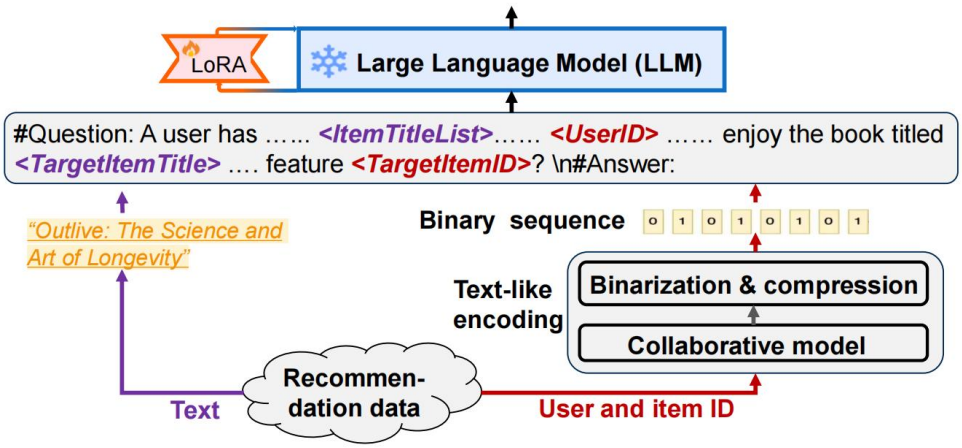

图 2-2 BinLLM 模型框架图

范式三：LLM-Graph Intergration

在这里我以 Ren X 等人的 RlmRec[4]工作为例，该方法通过利用大语言模型从文本角度挖掘用户行为偏好以及商品语义特征，并且利用最大化互信息的方式将文本信号和来自于图神经网络的协同信号增强对齐，从而有效促进算法学习到的表征质量。作者分别基于对比学习和生成学习两种训练范式设计了相应的表征

对齐方案。图 2-3 为 RlmRec 的模型框架图。

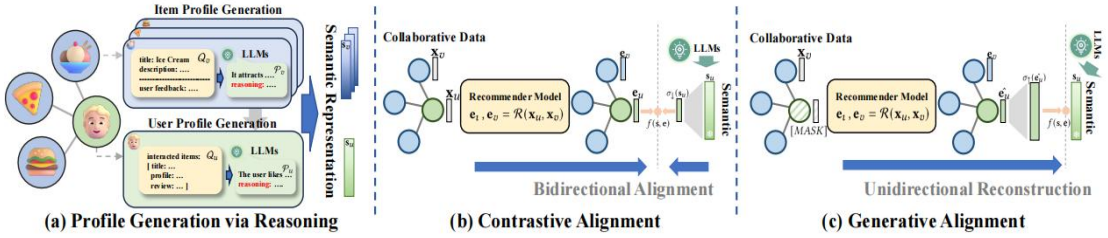

图 2-3 RlmRec 模型框架图

2.2 Graph-LLM For RAG

传统的 RAG 模型往往是基于文本相似度进行检索召回的，这种方式存在以下弊端：（1）稀疏检索挑战：当相关信息分散到多个文档中时，在大型数据集中识别到正确的信息段落有时是比较困难的；（2）缺乏上下文意识和关系建模能力，无法保持不同实体及其相互关系之间的一致性；（3）不支持结构化知识：对知识图谱或数据库等结构化知识无法有效利用；（4）更新难度较大：基于文本相似性的检索系统通常依赖于大规模向量索引结构来加速检索。如果知识库频繁更新，向量索引需要重新构建或大量调整，代价较高。

使用 Graph 来进行向量检索将会缓解以上的问题，基于 Graph 的 RAG 检索增强技术主要有以下几个优势：（1）多跳推理能力：通过消息传播，模型可以获得上下文中更深层的语义，这种深层匹配可以增强模型的推理生成能力；（2）关系建模能力：通过图检索，RAG 可以利用实体之间的结构化关系，进行更加精准的内容筛选；（3）高效的知识更新与管理：图结构可以通过增删节点和边来快速更新知识库的内容。

下面是一些基于 Graph 的 RAG 技术实例。

（1）G-Retriever[5]

该论文提出了一种 G-retriever 方法，它结合了 GNN、LLM、RAG，通过软提示进行高效微调以增强图理解能力。模型的检索生成主要分为四个阶段：（1）索引阶段：使用预训练的语言模型生成节点和图的嵌入来初始化 RAG，然后将这些嵌入存储在最近邻的数据结构中。（2）检索阶段：对于检索，对查询采取

与索引阶段相关的编码策略，来确保对文本信息的一致性处理。然后，识别与当前查询最相关的节点和边（即基于查询与每个节点或边的相似性，产生一组“相关的节点/边”）。（3）子图构建阶段：利用 PCST 算法构建一个尽可能包含多的相关节点和边的子图，同时保证图的大小是可管理的。核心目标是过滤掉与查询无关的节点和边，且保证图的大小转换成自然语言后是可以被输入到 LLM 中进行处理。（4）生成阶段：对于检索到的子图，一方面利用 GAT 进行图编码，另一方面将检索到的子图转换为文本格式，和查询一起输出到 LLM 进行字编码；最后结合两边的输出，一起输入到 LLM 中，生成答案。

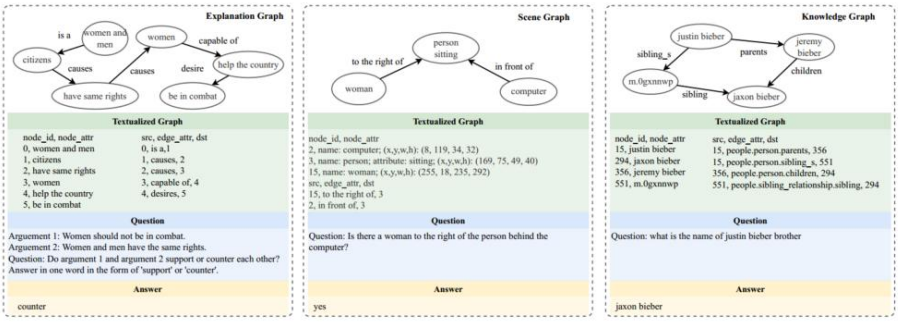

图 2-4 G-Retriever 模型框架图

（2）LightRAG[6]

现有 RAG 系统面临检索效率低下、信息相关性不足以及对新数据适应能力差等问题，严重限制了其应用范围。针对这些挑战，Zirui Guo 等人推出的 LightRAG 结合了图结构与双层检索机制，提供了有效解决方案。首先，图结构的引入使 LightRAG 能够精准捕捉实体间复杂依赖关系，提升信息理解的全面性。其次，双层检索策略同时处理具体与抽象查询，确保响应的相关性与丰富性。此外，LightRAG 具备快速适应新数据的能力，保证动态环境下的高效与准确。

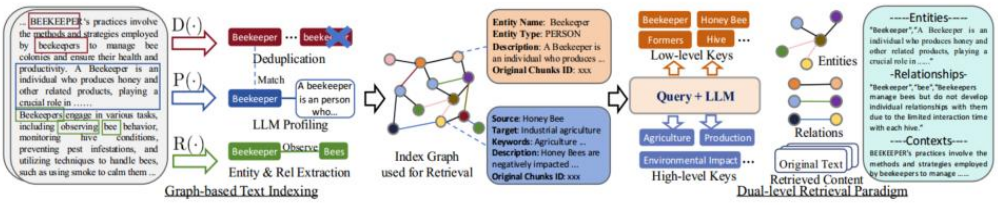

图 2-5 LightRAG 模型框架图

3 问题定义

大多数基于机器学习进行漏洞检测的方法往往是在源文件或应用程序的粗粒度级别上开展的，即判断一个源文件或者应用程序是否存在潜在漏洞。本实验从函数这种细粒度级别来分析漏洞代码，并将敏感函数识别看做一个二分类任务，通过将源代码表示为图结构，训练图神经网络来确定函数是否存在漏洞（是否为敏感函数）。关于源代码的图结构表示，本实验采用代码属性图（CPG）作为数据结构，CPG 整合了 AST，CFG，PDG 到一种数据结构当中。因此它是一种由多种关系（AST egde/CFG edge/PDG edge）构成的有向多重图。下面给出问题的符号定义：

数据样本可以表示为$((c_i, y_i) \mid c_i \in C, y_i \in Y), i \in \{1,2,...,n\}$，其中 C 表示源代码中的函数集，$Y = \{0,1\}^n$ 为函数的标签集，对于每个函数c_i，可以编码成一个多边图$g_i(V, X, A) \in G$，V 代表图的节点集，设 m 为节点的个数，$X \in R^{m*d}$ 为节点的特征矩阵，每一个节点$v_j \in V$ 的特征表示是一个 d 维的张量$x_j \in R^d$，$A \in \{0,1\}^{k*m*m}$ 为邻接矩阵，其中 k 表示边的关系种类数。$e_{s,t}^p \in A$ 如果等于 1 则表示节点v_s和v_t通过类型 p 的边相连。训练模型的目标是学习一个从G 到Y 映射函数f：$G->Y$ 来预测函数是否是敏感的，预测函数f 可以通过交叉熵损失函数得到：

$$\min \sum_{i=1}^{n} L(f(g_i(V, X, A), y_i \mid c_i)) + \lambda \omega(f)$$

4 模型概览

模型主要由三部分组成：（1）复合代码语义的图嵌入层：用来将源代码编码成具有综合语义的联合图结构。（2）门控图循环层：通过消息传播来聚合邻居节点的信息，从而学习每个节点的特征。（3）卷积池化模块：提取有用的节点特征进行图分类。

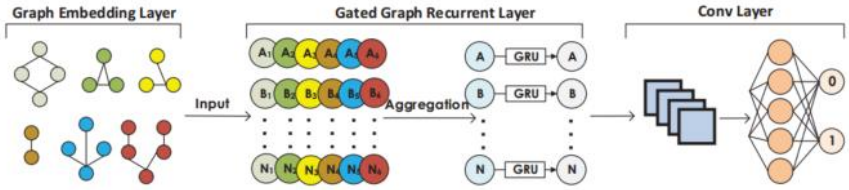

图 4-1 Devign 模型框架图

4.1 图嵌入

对于源代码的语义表示，比较常见的有 AST、控制流、数据流图等，此外，本模型还考虑了源代码的自然序列，它凭借平滑的结构以人类可读的方式捕获代码 token 序列之间的关系。

具体来说，使用 joern 工具来获得代码属性图，将函数 c_i 用联合图 g_i 表示，其中四类子图共享同一组节点 $V = V^{ast}$，这里的每个节点 $v \in V$ 都有两个属性：Code 和 Type。Code 属性通过使用预训练的 word2vec 模型对源代码进行编码来表示，Type 属性用来表示节点的类型，然后将 Code 和 Type 两种特征表示拼接在一起作为节点 v 的初始特征 x_v。图 4-2 形象的反映了这种图嵌入表示。

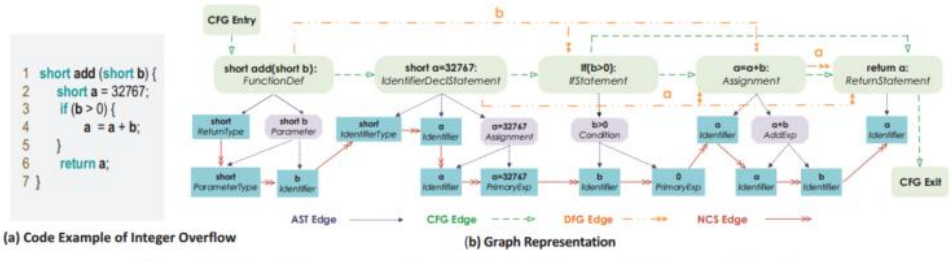

图 4-2 源代码的图嵌入表示

4.2 消息传播

图神经网络的核心思想是通过邻域聚合更新节点的特征表示。模型选用门控图递归网络（gated graph recurrent network）来学习节点的嵌入。对于给定的图嵌入表示 $g_t(V,X,A)$,图中的每个节点 $v_j \in V$，使用特征向量 x_j 来作为节点的初始状态 $h_j^{(1)} \in R^z$，$z \geq d$，即 $h_j^{(1)} = [x_j^T, 0]^T$，这里填充额外的 0 是为了对齐门控图神经网络的隐层大小。设 T 为门控图递归网络的总时间步，在时间步 t（t≤T）中，对于某个节点 j，在与关系 p 下的所有邻居节点聚合后的结果为：

$$a_{j,p}^{(t-1)} = A_p^T (W_p [h_1^{(t-1)^T}, ..., h_m^{(t-1)^T}] + b)$$

其中，A_p 表示关系 p 下的邻接矩阵，$W_p \in R^{z*z}$ 和 b 分别为权重和偏置。

接下来是通过一个门控循环单元（GRU）进行信息整合，公式如下：

$$h_j^{(t)} = GRU(h_j^{(t-1)}, AGG(\{a_{j,p}^{(t-1)}\}_{p=1}^k)$$

它首先将节点 v 在该时间步的所有关系下通过消息传播得到的特征向量进行整合，这里的 AGG 可以使任何函数（Sum/Mean/Max...），然后将前一时刻的隐藏状态和聚合后的邻居信息输入到 GRU 中，得到当前时刻节点的隐藏状态。

经过 T 个时间步的更新后，$H_i^{(T)} = \{h_j^{(T)}\}_{j=1}^m$ 作为节点集 V 的表示。

4.3 卷积池化

图分类的标准方法是全局收集所有生成的节点嵌入，通过线性加权求和等方式获得图的嵌入表示，用于预测图分类。本模型设计了一个 Conv 模块来选择与当前图任务相关的节点和特征集。通过一维卷积和密集神经网络来学习与图任务相关的特征，以获得更有效的预测。

定义 $\sigma(\cdot)$ 算子如下：$\sigma(\cdot) = MAXPOOL(\mathrm{Re}\,lu(CONV(\cdot)))$

设 l 为卷积层的个数，本模型定义 $l = 2$，通过卷积池化得到特征：

$$Z_i^{(1)} = \sigma([H_i^{(T)}, x_i]), ..., Z_i^{(l)} = \sigma(Z_i^{(l-1)})$$
$$Y_i^{(1)} = \sigma(H_i^{(T)}), ..., Y_i^{(l)} = \sigma(Y_i^{(l-1)})$$

最后将两个特征输入到 MLP 当中，然后对乘法结果进行平均聚合，以后使用 Sigmoid 激活函数将其映射到[0,1]的范围内，得到节点的预测结果。

$$\hat{y}_i = Sigmoid(AVG(MLP(Z_i^{(l)}) \otimes MLP(Y_i^{(l)})))$$

4.4 训练流程

这里主要对模型的 embed process 和 train process 进行代码分析。数据清洗是使用 joern 工具实现的，因此不再过度展开。

关于图嵌入，源代码在 main.py 中的 embed_task 函数进行了集成封装，该函数训练了用于编码节点中文本的 word2vec 模型，并对节点的文本进行编码，保存预处理后的数据集。图 4-3 是图嵌入的工作流程图。

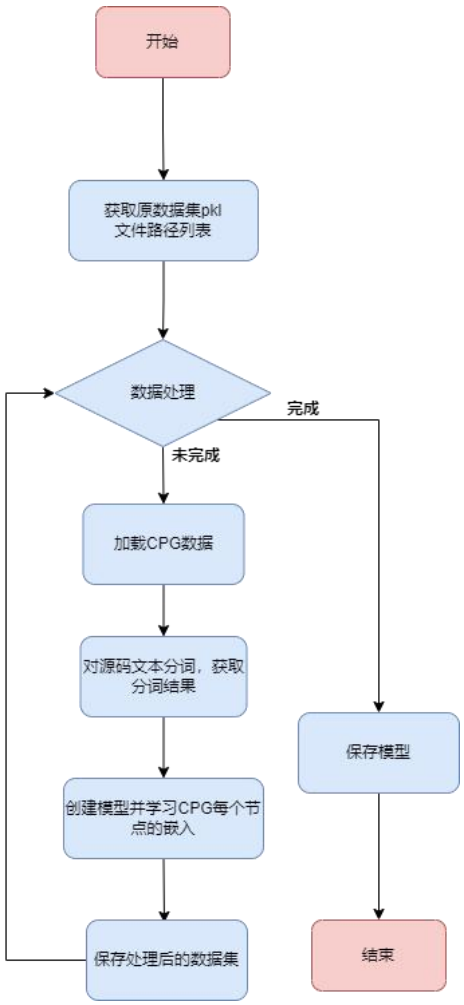

图 4-3 embed_task 工作流程图

关于模型的训练，源代码在 main.py 中的 process_task 函数中进行了集成封装。首先从配置文件中加载模型训练的超参数，然后定义模型、加载数据集，然后使用早停机制开始训练，EarlyStopping 是一种训练模型时常用的正则化策略，用于防止过拟合并节省训练时间，简单来说就是设定一个 patience 来最大允许验证性能在多少个连续 epoch 中没有提升，当超过耐心范围后触发停止训练。

另外值得一提的是，在源代码中 Train 类的调用方法中，传入的数据集迭代器并不是 Dataloader，而是使用自定义的 LoaderStep 类对 Dataloader 封装后的对象，在 LoaderStep 类中嵌套调用了自定义的 Stats 用于计算损失和准确率。此外，Train 中又调用了自定义的 History 类来进行控制台的打印和日志文件写入。图 4-4 展示了训练的工作流程图。

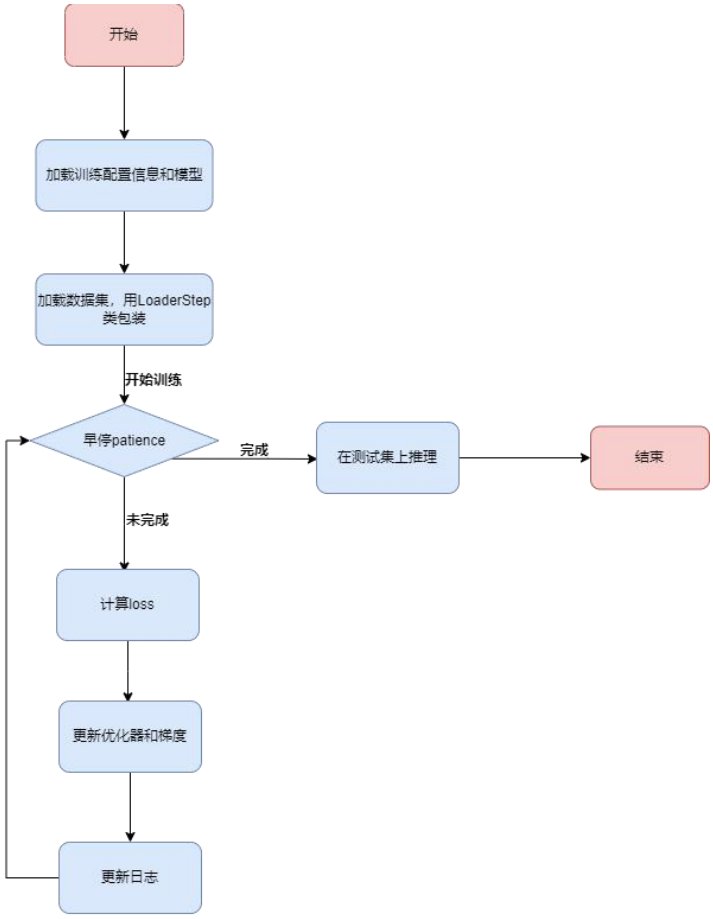

图 4-4 process_task 工作流程图

5 模型设计

本实验我主要从图嵌入和图池化操作上对已有代码进行了改进，下面是关于本实验所提供代码的一些缺陷：（1）关于图的节点表示，代码只提取了 CPG 图中的 AST 边，单一的代码语义表示往往不能够很好的反映代码的原始特征；（2）在图嵌入时所使用的词嵌入模型为 word2vec，然而该模型主要用于对自然语言进行词嵌入，对编程语言的理解能力较差，得到的图嵌入不能很好的反映代码的原始特征；（3）图池化操作方面，实验并没有提供 readout 部分的代码，我对原论文中的池化进行了复现，值得注意的细节是 1d 卷积中 channel 维的确定以及对 Y,Z 两个特征应分别设计 Conv 参数。下面我将对上面三个模型提升策略展开详细的介绍。

5.1 图的节点表示

实验提供的原数据经过 joern 工具处理后会得到相应的 cpg 图，cpg 图其实包含了多种类型的边，原论文只从中提取了 AST 边用于训练和推理，我从中额外提取了 cfg 和 dfg 类型的边进行了实验。下面是一些技术细节：

（1）原代码在 embed_task 函数中通过引用 nodes_to_input 函数来构造节点的特征表示和邻接矩阵，它提供了 edge_types 参数，在这里我们可以传一个边的类型列表（例如：edge_types=[‘Ast’,’Cfg’]），并对 nodes_to_input 函数进行改造，返回一个 Dataset 数组，从而构建多种关系图。

原来的 nodes_to_input 函数：

```
    def nodes_to_input(nodes, target, max_nodes, keyed_vectors, edge_type):
        nodes_embedding = NodesEmbedding(max_nodes, keyed_vectors)
        graphs_embedding = GraphsEmbedding(edge_type)
        label = torch.tensor([target]).float()

        return Data(x=nodes_embedding(nodes), edge_index=graphs_embedding(nodes), y=label)
```

改动以后的 nodes_to_input 函数：

```
    def nodes_to_input(nodes, target, max_nodes, edge_types):
```

```
        nodes_embedding = NodesEmbedding(max_nodes, keyed_vectors)
        nodes_emd = nodes_embedding(nodes)
        ret = []
        label = torch.tensor([target]).float()
        for edge_type in edge_types:
            graphs_embedding = GraphsEmbedding(edge_type)
            ret.append(Data(x=nodes_emd, edge_index=graphs_embedding(nodes), y=label))

        return ret

```

（2）训练参数需要进行调整。在加载数据集的时候，每个 batch 为一个由 Dataloader 组成的列表，因此映射到 device 上需要对 batch 的每个元素分别操作，此外 label 标签只需要用一个 Dataloader 元素的 y 作为 step 的 label 参数即可。

原始的 LoaderStep 类的__call__函数：

```
    def __call__(self, step):
            self.stats = stats.Stats(self.name)

            for i, batch in enumerate(self.loader):
                batch.to(self.device)
                stat: stats.Stat = step(i, batch, batch.y)
                self.stats(stat)

            return self.stats
```

调整以后：（这里以建立两种类型图，即 batch 有两个元素为例）

```
    def __call__(self, step):
            self.stats = stats.Stats(self.name)

            for i, batch in enumerate(self.loader):
                batch[0].to(self.device)
                batch[1].to(self.device)
                stat: stats.Stat = step(i, batch, batch[0].y)
                self.stats(stat)

            return self.stats
```

（3）门控神经网络在对每个关系图进行特征聚合之后，还需要聚合来自不同关系子图的信息。这里我设计了一个门控图循环网络来集成这些操作，在初始化的时候需要传入隐藏层大小和图神经网络的层数。下面以聚合两种关系图为例：

```
class GatedGraphRecurrentLayer(nn.Module):
    def __init__(self, hidden_size, num_layers):
        super(GatedGraphRecurrentLayer, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers
        self.conv_ast = GCNConv(hidden_size, hidden_size)
        self.conv_cfg = GCNConv(hidden_size, hidden_size)
        self.gru = nn.GRU(hidden_size, hidden_size, 1, batch_first=True)

    def agg(self, a_ast, a_cfg):
        return a_ast + a_cfg

    def forward(self, x, edge_ast, edge_cfg):
        h = x
        for _ in range(self.num_layers):
            a_ast = self.conv_ast(h, edge_ast)
            a_cfg = self.conv_cfg(h, edge_cfg)
            out, h = self.gru(self.agg(a_ast, a_cfg).unsqueeze(1), h.unsqueeze(0))
            h = h[-1, :, :]
        return h
```

5.2 图的节点表示

实验对图节点的文本信息进行编码是采用 word2vec 实现的，这里我改用 codebert 和 code-llama 模型重新进行图嵌入，经过实验测试，通过增强图嵌入的特征表示可以有效的提升图分类性能。

具体来说，需要对 prepare 阶段的 NodesEmbedding 类进行调整。这里主要介绍如何根据节点的文本信息得到相应的 embedding。下面是 get_code_embedding 函数的代码实现：

```
def get_code_embedding(code):

    # 将代码进行 Tokenize
    inputs = tokenizer(code, return_tensors='pt', truncation=True, padding=True, max_length=512).to(torch.device('cuda'))

    # 获取模型输出（返回的包含多个部分，通常我们使用最后一层的 hidden states）
    with torch.no_grad():
        outputs = model(**inputs)
```

```
9.
10.         # 通过模型输出获取代码的向量表示
11.         # 输出的 `last_hidden_state` 是一个 tensor，形状为 (batch_size, sequence_length, hidden_size)
12.         code_embedding = outputs.last_hidden_state.mean(dim=1)  # 取平均得到句子的向量表示
13.
14.         return code_embedding
```

编码后的特征张量还要加一维用来表示节点类型。使用的编码模型不同，最终得到的图节点特征形状也会不同。例如，codebert 的隐藏层大小为 768,编码后的特征向量为 769；code-llama 的隐藏层大小为 4096，编码后得到的特征向量大小为 4097。

更改了编码模型后的 NodesEmbedding 类的代码实现如下：

```
1.     class NodesEmbedding:
2.         def __init__(self, max_nodes: int):
3.             self.max_nodes = max_nodes
4.             assert self.max_nodes >= 0
5.             self.target = torch.zeros(self.max_nodes, 4096 + 1).float().to(torch.device('cuda'))  # co
    debert:768+1
6.
7.         def __call__(self, nodes):
8.             embedded_nodes = self.embed_nodes(nodes)
9.             self.target[:embedded_nodes.size(0), :] = embedded_nodes
10.
11.             return self.target
12.
13.         def embed_nodes(self, nodes):
14.             embeddings = torch.zeros(0, 4096 + 1).float().to(torch.device('cuda'))  # codebert:768+1
15.             batch_size = 512
16.             node_code_list = []
17.             node_type_list = []
18.             for n_id, node in nodes.items():
19.                 # Get node's code
20.                 node_code = node.get_code()
21.                 node_code_list.append(node_code)
22.                 node_type_list.append(node.type)
23.
24.             for batch_code, batch_type in zip(batch_processing(node_code_list, batch_size), batch_proc
    essing(node_type_list, batch_size)):
25.                 source_embedding = codebert.get_code_embedding(batch_code)
26.                 # The node representation is the concatenation of label and source embeddings
```

```
27.             source_label = torch.tensor(batch_type).unsqueeze(1).to(torch.device('cuda'))
28.             # print(source_label.shape)
29.             embedding = torch.cat((source_embedding, source_label), dim=1)
30.             embeddings = torch.cat((embeddings, embedding), dim=0)
31.
32.         return embeddings
```

5.3 图池化

图池化的工作是根据图的节点表示获得图表示，并基于图表示对图进行分类。由于源代码中没有实现Readout层，我按照论文的思路对该部分进行了编程实现，相关原理见4.3节。这里我探讨下作者设计卷积池化操作来得到图表示相比于直接将所有图节点的特征进行线性聚合的好处。1d卷积操作是根据卷积核的视野通过滑动窗口的方式对序列中的每个节点聚合邻居特征，因此这里的channel应该指的feature_size（特征向量的大小，即特征数），输入序列长度为max_node（图的节点个数，即序列长度）。卷积可以理解为上下文学习的过程，通过上下文来学习代码执行逻辑。此外，池化会改变特征向量的大小，相当于在聚合的过程中进行了特征压缩。由于论文对Y和Z两个特征表示均进行了两层卷积，为了在图池化过程模型参数得到有效的训练，这两个特征应分别设计相应的Conv参数。图5-1为图池化的框架图。

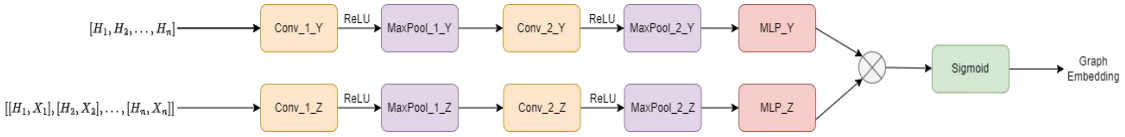

图 5-1 图池化框架图

下面给出关于图池化的代码实现，值得注意的是，经过门控图神经网络输出的张量形状为[batch_size, max_node, feature_size]，为了和nn.Conv1d（输入形状应为[N,C,L]）进行适配，需要交换-2，-1维的位置，即调整为[batch_size, hidden_size, max_node]。

```
1.  class Conv(nn.Module):
2.      def __init__(
3.          self, max_nodes, hidden_size, embed_size
```

```
        ):
            super(Conv, self).__init__()
            self.max_nodes = max_nodes
            self.conv_y1 = nn.Conv1d(in_channels=hidden_size, out_channels=hidden_size, kernel_size=3)
            self.pool_y1 = nn.MaxPool1d(kernel_size=3, stride=2)
            self.conv_y2 = nn.Conv1d(in_channels=hidden_size, out_channels=hidden_size, kernel_size=1)
            self.pool_y2 = nn.MaxPool1d(kernel_size=2, stride=2)
            self.fc_y = nn.Linear(hidden_size, 1)

            self.conv_z1 = nn.Conv1d(in_channels=hidden_size + embed_size, out_channels=hidden_size + embed_size, kernel_size=3)
            self.pool_z1 = nn.MaxPool1d(kernel_size=3, stride=2)
            self.conv_z2 = nn.Conv1d(in_channels=hidden_size + embed_size, out_channels=hidden_size + embed_size, kernel_size=1)
            self.pool_z2 = nn.MaxPool1d(kernel_size=2, stride=2)
            self.fc_z = nn.Linear(hidden_size + embed_size, 1)

            init_weights(self.conv_y1)
            init_weights(self.conv_y2)
            init_weights(self.conv_z1)
            init_weights(self.conv_z2)
            init_weights(self.fc_y)
            init_weights(self.fc_z)

            self.sigmoid = nn.Sigmoid()

        def forward(self, h, x):
            h = h.reshape(-1, self.max_nodes, h.shape[-1])
            x = x.reshape(-1, self.max_nodes, x.shape[-1])
            combined = torch.cat((h, x), dim=-1)
            y1 = self.pool_y1(F.relu(self.conv_y1(h.transpose(-2, -1))))
            y2 = self.pool_y2(F.relu(self.conv_y2(y1))).transpose(-2, -1)
            y = self.fc_y(y2)

            z1 = self.pool_z1(F.relu(self.conv_z1(combined.transpose(-2, -1))))
            z2 = self.pool_z2(F.relu(self.conv_z2(z1))).transpose(-2, -1)
            z = self.fc_z(z2)

            mult = torch.mul(y, z)
            avg = mult.mean(dim=1).squeeze(-1)
            return self.sigmoid(avg)
```

6 实验设计

6.1 实验环境

表 1 为关于实验设备的相关说明。表 2 为关于程序运行所依赖的 conda 环境说明。

表 1： 实验设备说明

操作系统	Ubuntu22.04
显卡	Nvidia RTX 4090(24GB)
Cuda 版本	≤12.4
GPU 驱动	550.90.07

表 2： conda 环境说明

Python 版本	3.9.13
库文件	scikit-learn>=0.22.2 gensim==3.8.1 cpgclientlib==0.11.111 pandas>=1.0.1 transformers==4.41.1
Torch 版本	2.3.0+cu118
torch-geometric	2.6.1
torch-scatter	2.1.2
torch-sparse	0.6.18

6.2 超参数设置

模型训练参数：Learning-rate=5e-6, epoch=100,patience=20,batch_size=256

图嵌入参数：max_node=205

图池化参数：见表 3

表 3： ReadOut Hyper-Parameters

Layer	Y	Z
Conv_1	in_channel=hidden_size,	in_channel=hidden_size+emb_size,

	out_channel=hidden_size, kernal_size=3	out_channel=hidden_size+emb_size, kernal_size=3
Conv_2	in_channel=hidden_size, out_channel=hidden_size, kernal_size=1	in_channel=hidden_size+emb_size, out_channel=hidden_size+emb_size, kernal_size=1
MaxPool_1	kernal_size=3, stride=2	
MaxPool_2	kernal_size=2, stride=2	
MLP	hidden_size-->1	hidden_size+emb_size-->1

6.3 实验设计

我主要针对图嵌入模型、关系图类型及数量、门控图神经网络层数这三个影响因素开展了一系列的消融和对比实验。考虑到不同嵌入模型得到的特征表示大小并不相同，然而无论是消息传播还是图池化，都不擅长处理维度较高的特征向量，因此在进行消息传播之前，会先经过一个 MLP 将原始嵌入向量进行降维为门控神经网络的隐藏层大小。在进行图池化之前，会通过一个 MLP 将原始嵌入向量降维到指定的 emb_size。在这里我们设定：

MLP_Word2Vec：emb_size 101- -> 25 hidden_size:200

MLP_CodeBert: emb_size 769-->100 hidden_size:200

MLP_Code-llama:emb_size 4097-->512 hidden_size:200

（1）num_layers = 6, edge_type=”AST”

	Word2vec	CodeBert	Code-llama
Accuracy	**0.5783**	0.4956	0.5696
Precision	0.5669	0.4956	**0.5974**
Recall	0.6316	**1.0**	0.4035

（2）num_layers = 8, edge_type=”AST”

	Word2vec	CodeBert	Code-llama
Accuracy	**0.5869**	0.4956	0.4826

Precision	**0.5704**	0.4956	0.4545
Recall	0.6754	**1.0**	0.2193

（3）num_layers = 10, edge_type=”AST”

	Word2vec	CodeBert	Code-llama
Accuracy	**0.5826**	0.4956	0.4826
Precision	**0.59**	0.4956	0.4528
Recall	0.5175	**1.0**	0.2105

（4）num_layers = 6, edge_type=”AST”&“CFG”

	Word2vec	CodeBert	Code-llama
Accuracy	0.5609	**0.6174**	0.5783
Precision	0.5822	**0.6757**	0.6049
Recall	0.4035	**0.4386**	0.4298

（5）num_layers = 8, edge_type=”AST”&“CFG”

	Word2vec	CodeBert	Code-llama
Accuracy	0.5739	**0.5783**	0.5696
Precision	0.5909	**0.6393**	0.5684
Recall	**0.4561**	0.4457	**0.4561**

（6）num_layers = 10,edge_type=”AST”&“CFG”

	Word2vec	CodeBert	Code-llama
Accuracy	**0.5696**	0.5609	0.5652
Precision	**0.5905**	0.5684	0.5833
Recall	0.4298	**0.4737**	0.4298

（7）num_layers = 6, edge_type=”AST”&“CFG”&”DFG”

	Word2vec	CodeBert	Code-llama
Accuracy	0.5783	0.5652	**0.5913**
Precision	0.6269	0.5795	**0.6316**
Recall	0.3684	**0.4474**	0.4211

（8）num_layers = 8, edge_type=”AST”&“CFG”&”DFG”

	Word2vec	CodeBert	Code-llama
Accuracy	0.5435	**0.5739**	0.5087
Precision	0.56	**0.5869**	0.5029
Recall	0.3684	0.4737	**0.7719**

（9）num_layers = 10, edge_type=”AST”&“CFG”&”DFG”

	Word2vec	CodeBert	Code-llama
Accuracy	**0.5652**	0.5348	0.5217
Precision	**0.6029**	0.5507	0.5099
Recall	0.3596	0.3333	**0.9035**

6.4 结果分析

（1）更改门控图神经网络的层数对模型性能有显著影响，并且在不同词嵌入模型下表现不一样。对于 Word2Vec 模型，层数适当的增长可以提高分类性能，我认为这是由于层数越深可以捕获更深层次的关系；但对于 CodeBert、Code-llama 模型，层数的增加反而会导致性能下降，这可能是因为噪声累积或者过拟合的原因导致模型训练出现了问题。

（2）补充额外的关系类型在使用 Word2vec 作图嵌入的场景下会对模型性能其负面作用，但是对使用 CodeBert、Codellama 等模型进行图嵌入的场景来说，对提升模型性能起到了正向作用。这可能是因为 word2vec 本身无法充分理解代码语言，因此增加新的关系子图用于特征聚合相当于添加了噪声，所以性能将会下降。

（3）在富关系情境下，使用代码生成模型作为图嵌入模型往往能够更好地发挥其代码理解能力，在单一关系场景下，基于 word2vec 的模型性能更优。此外，关于参数量更大的 code-llama 性能不如 codebert 的原因，我认为可能与 emb_size 和 hidden_size 的设定有关，在对节点嵌入向量进行特征映射的过程中可能会丢失一部分信息。另外，codellama 模型主要用于做生成任务，而图嵌入往往不需要复杂的生成能力，因此在嵌入时可能会引入噪声。

7 总结与展望

本实验让我了解了代码漏洞检测领域相关的背景和研究现状，深刻认识到了代码漏洞检测领域的重要性和挑战性。在本实验中，我根据 Devign 原论文的思路对模型进行了复现。在复现的过程中发现实验提供的源代码存在一些值得改进的地方，我对图嵌入、消息传播、图池化等多个模块进行了一定程度的修改，并进行了多组对比测试，实验结果一定程度上印证了我的假设。

在进行对比测试的时候，有些设定下模型没有得到很好的训练（loss 几乎不下降，很快就早停了），这可能和我的超参数设定有一些关系，例如我发现更改特征向量映射后的 emb_size 大小对模型性能的影响很大。总之 devign 模型的健壮性还有待增强。

8 课程感想

本人的研究方向是推荐系统，在选本门课程之前，我对图神经网络有一定的了解，不过这门课程让我对图神经网络在其他领域的应用有了更丰厚的认识。此外，我之前在做研究的过程中往往只关注了如何利用图的结构信息，对于图信号处理没有过多的接触，在该课程中老师对图滤波、傅里叶变换等知识的细致讲解让我对图神经网络的工作机制理解的更加深刻，并对动态图分析、降噪等技术有了新的认识。

此外，本课程中老师讲解了一些常用的图神经网络与大模型结合的方法，这也是我在做研究的过程中比较关注的点，其中将图信息以代码语言的方式输入给大模型进行训练的思路是我之前所没有思考过的。课堂上老师分享的一些心得也激发了对我自身的科研工作新的思考。

参考文献

[1] Zhou Y, Liu S, Siow J, et al. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks[J]. Advances in neural information processing systems, 2019, 32.

[2] Wei W, Ren X, Tang J, et al. Llmrec: Large language models with graph augmentation for recommendation[C]//Proceedings of the 17th ACM International Conference on Web Search and Data Mining. 2024: 806-815.

[3] Zhang Y, Bao K, Yan M, et al. Text-like Encoding of Collaborative Information in Large Language Models for Recommendation[J]. arXiv preprint arXiv:2406.03210, 2024.

[4] Ren X, Wei W, Xia L, et al. Representation learning with large language models for recommendation[C]//Proceedings of the ACM on Web Conference 2024. 2024: 3464-3475.

[5] He X, Tian Y, Sun Y, et al. G-retriever: Retrieval-augmented generation for textual graph understanding and question answering[J]. arXiv preprint arXiv:2402.07630, 2024.

[6] Guo Z, Xia L, Yu Y, et al. LightRAG: Simple and Fast Retrieval-Augmented Generation[J]. arXiv preprint arXiv:2410.05779, 2024.

附 录：实验结果截图

Num_layer=6 + Word2vec + Ast

```
Accuracy: 0.5782608695652174
Precision: 0.5669291338582677
Recall: 0.631578947368421
F-measure: 0.5975103734439833
Precision-Recall AUC: 0.6054810398290893
AUC: 0.6009528130671505
MCC: 0.1582992408228086
Error: 96.09461023257049
```

Num_layer=8 + Word2vec + Ast

```
TP: 77, FP: 58, TN: 58, FN: 37
Accuracy: 0.5869565217391305
Precision: 0.5703703703703704
Recall: 0.6754385964912281
F-measure: 0.6184738955823292
Precision-Recall AUC: 0.6176921625107454
AUC: 0.6085148215366001
MCC: 0.178146739318454
Error: 95.04002988701367
```

Num_layer=10 + Word2vec + Ast

```
TP: 59, FP: 41, TN: 75, FN: 55
Accuracy: 0.5826086956521739
Precision: 0.59
Recall: 0.5175438596491229
F-measure: 0.5514018691588786
Precision-Recall AUC: 0.6003526569148623
AUC: 0.6080611010284331
MCC: 0.16550329227910723
Error: 96.15670629565548
```

Num_layer=6 + Word2vec + Ast&CFG

```
TP: 46, FP: 33, TN: 83, FN: 68
Accuracy: 0.5608695652173913
Precision: 0.5822784810126582
Recall: 0.40350877192982454
F-measure: 0.47668393782383417
Precision-Recall AUC: 0.5840317091563136
AUC: 0.5753932244404114
MCC: 0.1253202603505768
Error: 95.1872284573335
```

Num_layer=8 + Word2vec + Ast&CFG

```
TP: 52, FP: 36, TN: 80, FN: 62
Accuracy: 0.5739130434782609
Precision: 0.5909090909090909
Recall: 0.45614035087719296
F-measure: 0.5148514851485149
Precision-Recall AUC: 0.5885730668690747
AUC: 0.5854506957047791
MCC: 0.14998233168057806
Error: 96.34604132762531
```

Num_layer=10 + Word2vec + Ast&CFG

```
TP: 49, FP: 34, TN: 82, FN: 65
Accuracy: 0.5695652173913044
Precision: 0.5903614457831325
Recall: 0.4298245614035088
F-measure: 0.4974619289340102
Precision-Recall AUC: 0.593405487275277
AUC: 0.5973986690865094
MCC: 0.14233748644369718
Error: 95.64784830053516
```

Num_layer=6 + Word2vec + Ast&CFG&DFG

```
TP: 42, FP: 25, TN: 91, FN: 72
Accuracy: 0.5782608695652174
Precision: 0.6268656716417911
Recall: 0.3684210526315789
F-measure: 0.4640883977900552
Precision-Recall AUC: 0.5930494441331053
AUC: 0.5769812462189957
MCC: 0.16825521249744374
Error: 95.44254146535388
```

Num_layer=8 + Word2vec + Ast&CFG&DFG

```
TP: 42, FP: 33, TN: 83, FN: 72
Accuracy: 0.5434782608695652
Precision: 0.56
Recall: 0.3684210526315789
F-measure: 0.4444444444444444
Precision-Recall AUC: 0.5795624117464069
AUC: 0.587416817906836
MCC: 0.0895251542003036
Error: 96.32437108391737
```

Num_layer=10 + Word2vec + Ast&CFG&DFG

```
TP: 41, FP: 27, TN: 89, FN: 73
Accuracy: 0.5652173913043478
Precision: 0.6029411764705882
Recall: 0.35964912280701755
F-measure: 0.45054945054945056
Precision-Recall AUC: 0.5943241322927879
AUC: 0.600574712643678
MCC: 0.13902681470252512
Error: 95.93935797507743
```

Num_layer=6 + CodeBert + Ast

```
TP: 114, FP: 116, TN: 0, FN: 0
Accuracy: 0.4956521739130435
Precision: 0.4956521739130435
Recall: 1.0
F-measure: 0.6627906976744187
Precision-Recall AUC: 0.6071441037026897
AUC: 0.6085148215366002
MCC: 0.0
Error: 95.54877902241518
```

Num_layer=8 + CodeBert + Ast

```
TP: 114, FP: 116, TN: 0, FN: 0
Accuracy: 0.4956521739130435
Precision: 0.4956521739130435
Recall: 1.0
F-measure: 0.6627906976744187
Precision-Recall AUC: 0.6090377619007972
AUC: 0.6138082274652148
MCC: 0.0
Error: 95.50661741289167
```

Num_layer=10 + CodeBert + Ast

```
TP: 114, FP: 116, TN: 0, FN: 0
Accuracy: 0.4956521739130435
Precision: 0.4956521739130435
Recall: 1.0
F-measure: 0.6627906976744187
Precision-Recall AUC: 0.6096395138372981
AUC: 0.6079854809437387
MCC: 0.0
Error: 95.6344349083875
```

Num_layer=6 + CodeBert + Ast&Cfg

```
TP: 50, FP: 24, TN: 92, FN: 64
Accuracy: 0.6173913043478261
Precision: 0.6756756756756757
Recall: 0.43859649122807015
F-measure: 0.5319148936170213
Precision-Recall AUC: 0.6236934568110272
AUC: 0.6193284936479129
MCC: 0.24798715237048458
Error: 93.40384474012552
```

Num_layer=8 + CodeBert + Ast&Cfg

```
TP: 39, FP: 22, TN: 94, FN: 75
Accuracy: 0.5782608695652174
Precision: 0.639344262295082
Recall: 0.34210526315789475
F-measure: 0.4457142857142858
Precision-Recall AUC: 0.6128844068493489
AUC: 0.6063974591651542
MCC: 0.17266361831250948
Error: 94.86166821117193
```

Num_layer=10 + CodeBert + Ast&Cfg

```
TP: 54, FP: 41, TN: 75, FN: 60
Accuracy: 0.5608695652173913
Precision: 0.5684210526315789
Recall: 0.47368421052631576
F-measure: 0.5167464114832535
Precision-Recall AUC: 0.6162644225697008
AUC: 0.6238656987295825
MCC: 0.1220919463432508
Error: 97.10438894189221
```

Num_layer=6 + CodeBert + Ast&Cfg&Dfg

```
TP: 51, FP: 37, TN: 79, FN: 63
Accuracy: 0.5652173913043478
Precision: 0.5795454545454546
Recall: 0.4473684210526316
F-measure: 0.504950495049505
Precision-Recall AUC: 0.6064866125291444
AUC: 0.6079098608590442
MCC: 0.1320902485444095
Error: 97.20366682457649
```

Num_layer=8 + CodeBert + Ast&Cfg&Dfg

```
TP: 54, FP: 38, TN: 78, FN: 60
Accuracy: 0.5739130434782609
Precision: 0.5869565217391305
Recall: 0.47368421052631576
F-measure: 0.5242718446601942
Precision-Recall AUC: 0.6131335379533791
AUC: 0.6091197822141561
MCC: 0.14910501299480675
Error: 97.83130979462209
```

Num_layer=10 + CodeBert + Ast&Cfg&Dfg

```
TP: 38, FP: 31, TN: 85, FN: 76
Accuracy: 0.5347826086956522
Precision: 0.5507246376811594
Recall: 0.3333333333333333
F-measure: 0.41530054644808745
Precision-Recall AUC: 0.6086061381257108
AUC: 0.6082123411978222
MCC: 0.07210950743036612
Error: 97.81611664110254
```

Num_layer=6 + Code-llama + Ast

```
TP: 46, FP: 31, TN: 85, FN: 68
Accuracy: 0.5695652173913044
Precision: 0.5974025974025974
Recall: 0.40350877192982454
F-measure: 0.481675392670157
Precision-Recall AUC: 0.6170386897119691
AUC: 0.6119177253478524
MCC: 0.14437177246089025
Error: 88.2516684538802
```

Num_layer=8 + Code-llama + Ast

```
TP: 25, FP: 30, TN: 86, FN: 89
Accuracy: 0.4826086956521739
Precision: 0.45454545454545453
Recall: 0.21929824561403508
F-measure: 0.2958579881656805
Precision-Recall AUC: 0.4838263931595155
AUC: 0.41772534785238963
MCC: -0.04609157559038639
Error: 99.89079596981693
```

Num_layer=10 + Code-llama + Ast

```
TP: 24, FP: 29, TN: 87, FN: 90
Accuracy: 0.4826086956521739
Precision: 0.4528301886792453
Recall: 0.21052631578947367
F-measure: 0.2874251497005988
Precision-Recall AUC: 0.48279647150835503
AUC: 0.4233212341197822
MCC: -0.04686671509602966
Error: 99.90819506853994
```

Num_layer=6 + Code-llama + Ast&Cfg

```
TP: 49, FP: 32, TN: 84, FN: 65
Accuracy: 0.5782608695652174
Precision: 0.6049382716049383
Recall: 0.4298245614035088
F-measure: 0.5025641025641027
Precision-Recall AUC: 0.6184806387474027
AUC: 0.6218239564428313
MCC: 0.1611612806259653
Error: 87.0352080471398
```

Num_layer=8 + Code-llama + Ast&Cfg

```
TP: 52, FP: 37, TN: 79, FN: 62
Accuracy: 0.5695652173913044
Precision: 0.5842696629213483
Recall: 0.45614035087719296
F-measure: 0.5123152709359605
Precision-Recall AUC: 0.6197189223949553
AUC: 0.6221264367816092
MCC: 0.14081577275416407
Error: 84.85841863637677
```

Num_layer=10 + Code-llama + Ast&Cfg

```
TP: 49, FP: 35, TN: 81, FN: 65
Accuracy: 0.5652173913043478
Precision: 0.5833333333333334
Recall: 0.4298245614035088
F-measure: 0.49494949494949503
Precision-Recall AUC: 0.6188910519329061
AUC: 0.6141863278886872
MCC: 0.13301968491744307
Error: 86.05413617042166
```

Num_layer=6 + Code-llama + Ast&Cfg&Dfg

```
TP: 48, FP: 28, TN: 88, FN: 66
Accuracy: 0.591304347826087
Precision: 0.631578947368421
Recall: 0.42105263157894735
F-measure: 0.5052631578947367
Precision-Recall AUC: 0.6129767747382483
AUC: 0.6168330308529947
MCC: 0.19098438453903407
Error: 87.32232008870187
```

Num_layer=8 + Code-llama + Ast&Cfg&Dfg

```
TP: 88, FP: 87, TN: 29, FN: 26
Accuracy: 0.508695652173913
Precision: 0.5028571428571429
Recall: 0.7719298245614035
F-measure: 0.6089965397923875
Precision-Recall AUC: 0.5455760033071122
AUC: 0.5334996975196612
MCC: 0.025704917156177023
Error: 98.1246186704883
```

Num_layer=10 + Code-llama + Ast&Cfg&Dfg

```
TP: 103, FP: 99, TN: 17, FN: 11
Accuracy: 0.5217391304347826
Precision: 0.5099009900990099
Recall: 0.9035087719298246
F-measure: 0.6518987341772152
Precision-Recall AUC: 0.5705625414179831
AUC: 0.5647307924984877
MCC: 0.07654588855455577
Error: 95.62136134882623
```