DOI: 10.15514/ISPRAS-2023-35(1)-15

Сравнение графовых векторных представлений исходного кода с текстовыми моделями на основе архитектур CNN и CodeBERT

В.А. Романов, ORCID: 0000-0003-3772-0039 <v.romanov@innopolis.ru>
В.В. Иванов, ORCID: 0000-0003-3289-8188 <v.ivanov@innopolis.ru>

Университет Иннополис,
420500, Россия, г. Иннополис, ул. Университетская, д. 1.

Аннотация. Одним из возможных способов уменьшения ошибок в исходном коде является создание интеллектуальных инструментов, облегчающих процесс разработки. Такие инструменты часто используют векторные представления исходного кода и методы машинного обучения, заимствованные из области обработки естественного языка. Однако такие подходы не учитывают специфику исходного кода и его структуру. Данная работа посвящена исследованию методов предварительного обучения графовых векторных представлений исходного кода, где граф представляет структуру программы. Результаты показывают, что графовые векторные представления позволяют достичь точности классификации типов переменных программ, написанных на языке Python, сравнимой с векторными представлениями CodeBERT. Более того, одновременное использование текстовых и графовых векторных представлений в составе гибридной модели позволяет повысить точность классификации типов более чем на 10%.

Ключевые слова: исходный код; классификация типов переменных; Python; графовые нейронные сети; CodeBERT

Для цитирования: Романов В.А., Иванов В.В. Сравнение графовых векторных представлений исходного кода с текстовыми моделями на основе архитектур CNN и CodeBERT. Труды ИСП РАН, том 35, вып. 1, 2023 г., стр. 237-264. DOI: 10.15514/ISPRAS–2023–35(1)–15

Благодарности: Исследование выполнено при поддержке гранта Российского научного фонда (проект № 22-21-00493, https://rscf.ru/project/22-21-00493/).

Comparison of Graph Embeddings for Source Code with Text Models Based on CNN and CodeBERT Architectures

V.A. Romanov, ORCID: 0000-0003-3772-0039 <v.romanov@innopolis.ru>
V.V. Ivanov, ORCID: 0000-0003-3289-8188 <v.ivanov@innopolis.ru>

Innopolis University,
1, Universitetskaya Str., Innopolis, 420500, Russia

Abstract. One possible way to reduce bugs in source code is to create intelligent tools that make the development process easier. Such tools often use vector representations of the source code and machine learning techniques borrowed from the field of natural language processing. However, such approaches do not take into account the specifics of the source code and its structure. This work studies methods for pretraining graph vector representations for source code, where the graph represents the structure of the program. The results show that graph embeddings allow to achieve an accuracy of classifying variable types in Python programs that is comparable to CodeBERT embeddings. Moreover, the simultaneous use of text and graph embeddings as part of a hybrid model can improve the accuracy of type classification by more than 10%.

Keywords: source code; variable type prediction; Python; graph neural networks; CodeBERT

For citation: Romanov V.A., Ivanov V.V. Comparison of Graph Embeddings for Source Code with Text Models Based on CNN and CodeBERT Architectures. Trudy ISP RAN/Proc. ISP RAS, vol. 35, issue 1, 2023. pp. 237-264 (in Russian). DOI: 10.15514/ISPRAS-2023-35(1)-15

Acknowledgements: The study was supported by a grant from the Russian Science Foundation (project no. 22-21-00493, https://rscf.ru/project/22-21-00493/).

1. Введение

В настоящий момент эталонной нейро-архитектурой для обработки исходного кода является трансформер [1]. Большинство моделей данной архитектуры принимают на вход исходный код в виде последовательности токенов. При этом специфические свойства исходного кода, такие как нелинейность исполнения, не учитываются. Примерами моделей, реализованных с использованием данной архитектуры, являются CuBERT и CodeBERT, показавшие свою эффективность для решения целевых задач [2, 3].

Альтернативой таким подходам являются предварительно обученные модели на основе графов, построенных из исходного кода. Такие модели могут учитывать зависимости в программе, даже если исходный код разделён на несколько файлов. Использование представления исходного кода в виде графа перспективно и исследовалось в нескольких работах, в том числе посвященных предварительному обучению [4, 5]. Тем не менее подходы, основанные на графах, новы и недостаточно изучены. В данной работе исследуется два подхода создания предварительно обученных графовых векторных представлений: на основе методов тренировки реляционных векторных представлений и на основе графовых нейронных сетей. Качество полученных векторных представлений оценивается на задаче классификации типов переменных.

Одна из проблем существующих подходов анализа исходного кода с помощью методов обработки текста – ограниченность обрабатываемого контекста. Модели машинного обучения, которые принимают исходный код в виде последовательности токенов (модели на архитектуре трансформер), могут обрабатывать за раз ограниченное количество токенов. При этом релевантные для решаемой задачи части кода могут быть размещены в нескольких файлах. По этой причине актуальными становятся методы создания предварительно обученных моделей на основе графов, построенных из исходного кода.

Идея создания предварительной обученных моделей для исходного кода, основанных на графовых нейро-сетевых моделях (GNN) не нова [5-9]. Большинство таких подходов тренируются путём решения задачи предсказания наличия связей между различными элементами программы и их типов. Модели, использующие графовые нейронные сети, уже показали свою эффективность при решении целевых задач. Однако в существующих исследованиях не проведено сравнение предварительно обученных моделей, реализованных с помощью графовых нейронных сетей и с помощью архитектуры трансформер. Проведение такого сравнения является целью данной работы.

Представления исходного кода в виде последовательности токенов и в виде графа могут дополнять друг друга, например, при создании статического анализатора типов в языках программирования с динамической типизацией (JavaScript, Python). Как правило, статические анализаторы типов полагаются на наличие подсказок в коде (type hints). Однако формальные методы не всегда способны предоставить однозначный ответ. Эту проблему можно частично решить, если механизм вывода типов сможет ранжировать кандидатов на основе дополнительных данных, таких как имена переменных или характер их использования [10]. Для решения задачи классификации типов переменных ранее уже демонстрировалась эффективность отдельно представлений исходного кода в виде последовательности токенов и в виде графа. В данной работе показано, что точность

классификации типов можно улучшить путём применения гибридной модели, которая использует одновременно два вида представлений исходного кода.

Новизна данной работы заключается в следующем.

1) Разработан метод преобразования исходного кода на языке Python в граф, содержащий глобальные связи, такие как вызовы функций и импортирования модулей, и отображает структуру программы.
2) Предложен подход для предварительной тренировки реляционных векторных представлений для исходного кода на основе графа с добавлением k-hop рёбер.
3) Предложен подход предварительной тренировки графовой нейросетевой модели для исходного кода, целью которой является создание векторных представлений. В качестве задач предварительного обучения используются предсказание имён, предсказание и классификация связей в графе, классификация типов узлов в графе.
4) Проведено исследование применимости предварительно обученных векторных представлений для решения задачи классификации типов переменных для программ, написанных на языке Python. Проведено сравнение графовых векторных представлений с векторными представлениями CodeBERT.

Исходный код для получения результатов опубликован на GitHub[1].

2. Обзор литературы

2.1 Предварительно обученные модели для исходного кода

Предварительно обученные модели позволяют сократить время тренировки. Они используются для инициализации моделей машинного обучения при решении самых разных задач. С появлением архитектуры трансформер многие работы исследовали её применение к исходному коду [2, 11, 12]. Один из самых базовых подходов для предварительного обучения – маскирующая модель (MLM). В некоторых работах используются дополнительные задачи предварительного обучения, разработанные специально для исходного кода. К ним относятся перевод между языками программирования, генерация текстового описания для кода и генерация кода из текстового описания [13]. Существуют модификации, использующие информацию из графа программы, например, графа потока данных [4].

В последние годы появляется всё больше работ, исследующих использование модальности исходного кода в виде дерева или графа [4, 14, 15], а также графовые нейронные сети для обучения векторных представлений для исходного кода [5, 13]. Тем не менее сравнение предварительно обученных моделей на основе архитектуры трансформер и графовых нейронных сетей всё ещё не проведено. Одна из причин – сложность оценки качества предварительно обученных моделей.

2.2 Методы оценки предварительно обученных моделей для исходного кода

В последнее время предварительно обученные модели для исходного кода всё чаще оцениваются путем решения таких целевых задач, как обнаружение неправильно используемых переменных [2, 11], предсказание имён переменных и функций [5, 11], поиск исходного кода [4], а также перевод между языками программирования [4, 13]. Иногда задачи направлены на то, чтобы понять, какие свойства исходного кода можно извлечь из предварительно обученных векторных представлений. Примерами таких задач могут быть классификация узлов абстрактного синтаксического дерева программы, оценка цикломатической сложности, оценка длины кода и обнаружение неправильных типов [16].

[1] https://github.com/VitalyRomanov/method-embedding

2.3 Методы решения задачи классификации типов переменных

Использование предварительно обученных моделей часто позволяет сократить время тренировки и количество требуемых данных при решении целевых задач. На настоящий момент не существует успешной предварительно обученной модели для исходного кода, использующей графовые нейронные сети. Тем не менее существует множество работ, в которых такие нейронные сети применяются для решения целевых задач. Одним из примеров является задача классификации типов переменных в программах, написанных с использованием языков программирования с динамической типизацией (Python, JavaScript).

При решении задачи классификации типов часто исходный код, подаваемый на вход модели машинного обучения, представляют в виде последовательности токенов [10, 17, 18]. Такая задача имеет смысл для динамических языков программирования, для которых формальный статический анализатор не всегда может предложить однозначный ответ. При решении этой задачи с помощью машинного обучения, для классификации типа зачастую используется не только информация о структуре исходного кода, но также документация и имена переменных [19, 20]. Результат классификации может затем передаваться в качестве рекомендаций статическому анализатору. В одной из работ был представлен подход под названием TypeWriter, основанный на рекуррентных нейронных сетях (RNN) [21]. Он сочетает в себе вероятностную оценку возможных типов и дальнейшую верификацию предложенных кандидатов. Благодаря второму шагу, такой подход может гарантировать корректность полученного типа.

Самая ранняя работа, посвящённая задаче классификации типов, исследовала применение машинного обучения для классификации типов переменных программ, написанных на языке JavaScript. При этом исходный код был представлен в виде графа потока управления [22]. Недавно был предложен подход под названием Typilus [23]. Авторы этого подхода использовали представление исходного кода в виде графа потока управления и данных. В отличие от предыдущих работ, где тип мог принимать одно из заранее выбранных значений, в этом подходе новые типы могут быть добавлены даже после обучения. В последние годы чаще можно найти работы, в которых основной моделью для классификации типов является графовая нейронная сеть [24-26].

2.4 Существующие подходы преобразования исходного кода в граф

Целью данной работы является исследование применения графовых векторных представлений для решения целевых задач исходного кода. Формат графа может иметь существенный вклад в качество финальных векторных представлений. Далее рассмотрены несколько форматов: в виде абстрактного синтаксического дерева (AST), графа потока управления и графа потока данных, и межпроцедурного графа. Эти виды представлений могут быть комбинированы в разных сочетаниях.

Представление в виде AST получается непосредственно из исходного кода программы. Узлы обозначают элементы исходного кода, а ребра – зависимости. Такое представление получить проще всего, однако оно обладает рядом недостатков: наличие узлов с повторяющейся функциональностью (например узлы циклов for и while), зависимость от языка программирования, большое количество узлов в дереве. Несмотря на это, представление исходного кода с помощью AST на сегодняшний день широко применяется в исследовательских работах [27-29].

При использовании графов потока управления узлы обозначают выражения, а ребра – передачу управления между выражениями. Часто процедуры при таком представлении имеют входные и выходные узлы. Графы потока управления содержат меньше узлов и могут обеспечить представление программы более независимое от языка программирования. Существует множество работ, использующих граф потока управления для анализа исходного

кода с помощью машинного обучения [30-35]. Иногда, вместо использования полноценного графа потока управления, представление AST дополняется рёбрами потока управления.

Графы потока данных получаются путём извлечения зависимостей между переменными. Такие представления могут не содержать условных выражений и, как следствие, операторов управления. Представление в виде графа потока данных получить труднее всего, но его польза для решения задач машинного обучения была не раз продемонстрирована [31, 32, 34-36].

При обработке исходного кода графы потока управления или данных обычно строятся для одной процедуры или функции. Межпроцедурные графы соединяют несколько отдельных графов в один через входные и выходные узлы, которые определены для каждой процедуры. Существующие инструменты позволяют получить такое представление только для узкого круга языков программирования. Один из способов построения такого графа использует информацию от компилятора программы [37].

Формат представления в виде графа полезен прежде всего потому, что он позволяет запечатлеть структурные зависимости в исходном коде. Однако не все зависимости могут быть использованы существующими методами машинного обучения. Особенно это справедливо для узлов в графе, расположенных на большом удалении друг от друга. Чтобы сократить расстояние в графе можно использовать дополнительные ребра, которые явным образом представляют полезные связи между удалёнными узлами. Так, представление AST часто дополняется вспомогательными рёбрами. Они могут быть из числа рёбер потока управления или данных, а также выполнять сугубо вспомогательные функции. Встречающиеся типы дополнительных ребер можно разбить на несколько групп:

- рёбра, обозначающие тип данных (Type, Inherits) [25, 30, 38];
- рёбра упоминания (LastUse, LastWrite) [38];
- рёбра вызова функций (FunctionCall) [6, 39];
- рёбра зависимости данных (NextUse) [23, 28];
- рёбра следования (NextExpression, NextArgument и NextToken);
- рёбра возврата (ReturnTo) [29, 31, 38];
- рёбра соседних узлов (Sibling) [38];
- рёбра аттрибутов (NodeName и NodeType) [28, 40];
- обратные рёбра [41].

При выборе представления графа возникает естественный вопрос: какие типы рёбер в графе важны для достижения высокого качества моделей машинного обучения на выбранной задаче? К сожалению, удалось найти только две работы, затрагивающие данный вопрос [23, 41], в которых проводилась оценка вклада различных типов рёбер при решении задач идентификации неправильно используемых переменных, классификация имён переменных и классификация типов переменных.

2.5 Векторные представления для графов

Современные графовые нейронные сети масштабируется до миллионов и миллиардов узлов [42], что открывает возможности для их применения к широкому кругу задач. В одной из работ был разработан метод под названием M-GNN, предназначенный для создания иерархических векторных представлений для графов [43]. Иерархия создаётся путём упрощения графа за счёт объединения узлов в суперузлы. Чем выше уровень иерархии, тем меньше узлов в графе.

Архитектура графовой нейросети под названием R-GCN была разработана для обработки гетерогенных графов [44]. Она может обрабатывать графы с несколькими типами рёбер.

Данная модель является автокодировщиком, цель которого восстановить информацию о рёбрах в графе. В одной из работ авторы решили использовать декодировщик с меньшим количеством тренируемых параметров [45]. В качестве критерия существования ребра они использовали целевую функцию RotatE. В другой работе R-GCN-подобная архитектура была модифицирована для работы с механизмом внимания [46].

Графовые векторные представления всё чаще находят своё применение при анализе исходного кода с помощью машинного обучения. Далее приводится описание используемого в данной работе подхода преобразования исходного кода в граф и методов тренировки графовых векторных представлений.

3. Методология

Далее описываются применяемые в данной работе методы преобразования исходного кода в граф, тренировки графовых векторных представлений и тестирования полученных векторных представлений на задаче классификации типов переменных.

3.1 Преобразование исходного кода в граф

В рамках данной работы к графовому представлению исходного кода выдвигаются следующие требования: 1) переменные с одинаковым именем, встречающиеся в теле одной функции, должны интерпретироваться как один узел в графе, что позволит более эффективно извлекать информацию о позициях в коде, где используется данная переменная; 2) для выражений, определённых в теле условного оператора if, циклов for и while, блока обработки исключений try, в графе должны присутствовать связи с упомянутыми выше операторами (например, для выражения в блоке оператора if должна присутствовать связь с узлом, соответствующим данному оператору), что позволит увеличить степень вершин графа и сократить его диаметр; 3) в графе должен отражаться порядок исполнения выражений; 4) связи с импортируемыми модулями, вызываемыми функциями, и наследуемыми классами должны однозначно разрешаться; 5) значения констант в исходном коде (чисел и строк) должны не отображаться в графе для сокращения числа уникальных узлов; 6) для уменьшения количества уникальных имён, имена функций и переменных должны быть токенизированы.

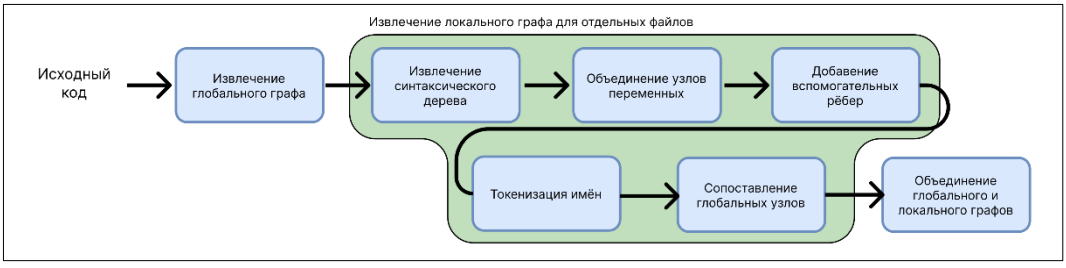

Рис. 1. Процедура преобразования исходного кода в граф
Fig. 1. The procedure for converting the source code into a graph

Стоит отметить, что для некоторых задач исходного кода сформулированное выше представление может быть избыточным. В частности, в разделе 4.8 проводятся эксперименты по проверке необходимости использования глобальных связей, сабтокенизации имён и наличия информации о типах связей между узлами при решении задачи классификации типов переменных. Так как цель работы заключается в проверке качества предварительно обученной модели, для изначальных экспериментов, производится построение графа в соответствии с упомянутыми выше требованиями. Построение графа осуществляется путём использования процедуры, приведённой на рис. 1.

На первом шаге осуществляется извлечение глобальных взаимосвязей с помощью утилиты Sourcetrail[2]. В процессе индексирования Sourcetrail создает базу данных глобальных связей. Примерами могут быть связи с вызываемыми в коде функциями, импортированными модулями, а также отношения наследования. Граф $g_{global} = (N_{global}, E_{global})$ представляет собой множество глобальных узлов и взаимосвязей между ними. Утилита Sourcetrail позволяет извлекать зависимости, даже когда происходит импортирование из сторонних пакетов. В результате для часто используемых библиотек собирается информация о том, как именно они используются. В дополнение, Sourcetrail сохраняет соответствие между исходным кодом и узлами в графе, что позволяет позднее для решения задачи классификации типов одновременно использовать совместное представление исходного кода в виде последовательности токенов и в виде графа.

Далее, обработка исходного кода в кодовой базе осуществляется на уровне отдельных файлов. Для извлечения синтаксического дерева программы используется модуль ast (Python 3.8). Далее происходит формирование локального графа исходного кода $g_{local} = (N_{local}, E_{local})$, который создаётся для отдельно взятого файла. На шаге объединения имён переменных происходит преобразование синтаксического дерева программы в граф. Переменные с одинаковым именем внутри тела функции объединяются в один узел. Затем происходит добавление вспомогательных рёбер. Для выражений, определённых в теле условного оператора if, циклов for и while, блока обработки исключений try, добавляются связи с упомянутыми выше операторами (например, для выражения в блоке оператора if добавляется связь с узлом, соответствующим данному оператору). Для отображения порядка исполнения программы в графе используются дополнительные рёбра next и prev.

Последний шаг в процессе создания локального графа – токенизация имён. На этом шаге в граф добавляются узлы и рёбра типа subword, которые обозначают токены имён. Узлы, представляющие токены, являются общими для всех файлов в кодовой базе. Без токенизации количество уникальных имён растет за счет неологизмов по мере добавления нового кода в кодовую базу [47]. Одним из самых популярных инструментов для токенизации является sentencepiece [48]. Он основан на алгоритмах сжатия, находит наиболее часто встречающиеся подстроки в кодовой базе и использует их в качестве токенов.

На последнем этапе обработки файла в кодовой базе происходит сопоставление глобальных узлов. К локальному графу добавляются ребра типа global_mention, связывающие узлы локального графа с соответствующими им узлами глобального графа. Эти ребра можно представить графом $g_{global\_mention} = \left(N_{local} \cup N_{global}, E_{global\_mention}\right)$. Результатом сопоставления является граф $g = g_{local} \cup g_{global} \cup g_{global\_mention}$. После этого происходит объединение графа g для отдельного файла с общим графом G, который объединяет в себе все обрабатываемые пакеты.

Пример графа, построенного из исходного кода, показан на рис. 2. Исходный код содержит определения двух функций и вызов функции. В ходе извлечения абстрактного синтаксического дерева создаются узлы определений функций (FunctionDef) и аргументов (arg), выражений return, if и (Call). Константы, такие как числа и строки, заменяются специальным узлом Constant. Далее происходит объединение узлов переменных. Так, оба упоминания переменной *c* в функции condition обозначаются одним узлом.

[2] https://github.com/CoatiSoftware/Sourcetrail

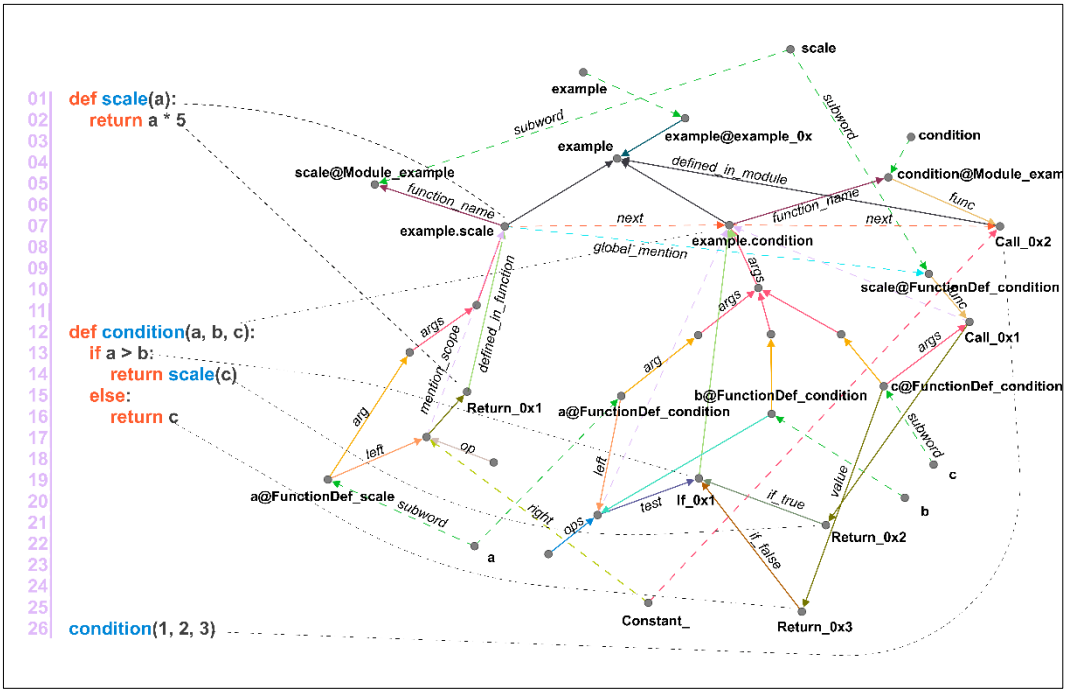

Рис. 2. Пример графа исходного кода, построенного для двух функций. Исходный код показан слева. Пунктирные черные линии ведут от операторов в коде к узлам графа, которые представляют эти операторы. Пунктирные цветные линии представляют собой дополнительные ребра, добавленные при аугментации. Имена некоторых узлов и некоторых ребер, а также всех обратных ребер опущены для облегчения понимания графа

Fig. 2. An example of a source code graph built for two functions. The source code is shown on the left. Dashed black lines lead from the statements in the code to the graph nodes that represent those statements. The dotted colored lines represent additional edges added during the augmentation. The names of some nodes and some edges, as well as all back edges, have been omitted to make the graph easier to understand

Рёбра, представленные сплошными линиями, взяты непосредственно из абстрактного синтаксического дерева. Ребра, представленные пунктирными линиями, получены на этапе добавления вспомогательных рёбер. Например, тип ребра mention_scope в функции scale соединяет узел, представляющий операцию умножения, с узлом, представляющим определение функции. Другие вспомогательные ребра используются для соединения следующих друг за другом выражений в коде (next, prev), контекста условного оператора (if_true, if_false). На этапе токенизации имён создаются узлы и рёбра типа subword. На этапе сопоставления глобальных узлов добавляются рёбра типа global_mention. Все ребра, кроме subword и op, соединяющих операторы, имеют парные обратные рёбра, которые не показаны в данном примере.

Узлы глобального графа можно отделить от общего графа. Глобальный узел представляет собой конкретный класс или функцию, которая может появляться как в различных файлах, так и в различных пакетах. Глобальные узлы связаны с любыми их упоминаниями в локальном графе с помощью отношений типа global_mention. Таким образом, локальный и глобальный графы можно отделить друг от друга, если удалить связи этого типа.

Этот факт можно использовать для предварительной тренировки. Например, в будущем, в качестве одной из целевых функций для предварительной тренировки может быть исследовано предсказание глобальных отношений на основе локального графа.

Атрибуты узлов имеют важную роль при предварительном обучении. В предложенном формате графа каждый узел ассоциируется с именем и типом.

Для большинства узлов, полученных из абстрактного синтаксического дерева, имя определяется типом узла. Для узлов, обозначающих имена переменных, функций, или аргументов, имя берётся из исходного кода. Имена для глобальных узлов извлекаются утилитой Sourcetrail и представляют собой полный путь от пакета до имени элемента (например, для метода __init__ класса string пакета builtins имя будет иметь вид builtins.string.__init__).

3.2 Предварительное обучение графовых векторных представлений

Для создания предварительно обученных векторных представлений предложено два подхода. Первый подход использует методы создания реляционных векторных представлений, разработанных для графов знания. Как и граф знания, представление исходного кода в виде графа содержит сущности (элементы программы) и отношения между ними. Такой способ не требует большого количества вычислительных ресурсов, и поэтому рассматривается в данной работе. Второй подход использует графовые нейронные сети.

3.3 Метод обучения реляционных векторных представлений

Метод обучения реляционных векторных представлений заключается в следующем. Цель тренировки – восстановление троек отношений (h, r, t), где h и t – головной и хвостовой объекты, а r – тип отношения. Цель тренировки заключается в том, чтобы максимизировать правдоподобие для правильных троек и минимизировать – для неправильных. В данной работе рассматривается несколько стандартных подходов для тренировки:

- TransR [49];
- DistMult [50];
- RESCAL [51];
- ComplEx [52];
- RotatE [53].

Одной из особенностей графа исходного кода является низкая связность узлов в графе, которая затрудняет обучение реляционных векторных представлений. Для того чтобы улучшить процесс тренировки, предлагается метод модификации графа, состоящий из следующих шагов:

1) убрать обратные рёбра, так как они нужны только для процесса обмена сообщениями;
2) создать k-hop ребра $k = 1..3$, которые соединяют узлы на расстоянии k и повышают степень связности графа;
3) добавить узлы, обозначающие типы, что позволяет сделать узлы одного типа ближе друг к другу независимо от используемой целевой функции для тренировки векторных представлений.

3.4 Метод обучения векторных представлений с помощью графовых нейронных сетей

В данной работе используется реляционная графовая свёрточная сеть (R-GCN), которая использовалась в недавних работах [5, 54]. Вначале все узлы инициализируются с помощью векторных представлений имён узлов. Затем осуществляется несколько итераций передачи сообщений. Агрегирование сообщений производится после каждой итерации путём усреднения векторных представлений, полученных от соседей. Рекуррентное уравнение для обновления состояния узла, которое часто встречается в литературе, можно записать в виде

$$\boldsymbol{h_i^{n+1}} = \boldsymbol{\sigma}\left(\boldsymbol{h_i^n} + \sum_{\boldsymbol{r \in R(i)}} \sum_{\boldsymbol{j \in N(i,r)}} \boldsymbol{f_r}\left(\boldsymbol{h_j^n}\right) \right), \quad (\mathbf{1})$$

где h_i^n – векторное представление узла i после слоя n; $R(i)$ возвращает типы отношений, в которых участвует узел i; $N(r,i)$ – множество соседей узла i связанных через отношение r; f_r – функция преобразования, зависящая от типа отношения, параметризованная нейронной сетью; σ – функция активации.

В данной работе в качестве целевых задач предварительного обучения исследуются задачи, основанные на: 1) предсказании имён переменных и функций (GNN-NamePred); 2) предсказании связей между узлами графа (GNN-EdgePred); 3) классификации типов связей в графе (GNN-TransR); 4) классификации типов узлов в графе (GNN-NodeClf). Разметку для обучения можно сгенерировать автоматически с помощью простых правил.

Задача предварительного обучения GNN-NamePred использует две составляющие: векторные представления узла и имени. Для вычисления представления узла используется рекурсивная формула (1). Для того чтобы увеличить утилизацию параметров, осуществлена токенизация имён. Каждый токен имени представлен своим уникальным вектором. Токены, участвующие в графе, и токены, полученные из имён, представлены разными векторами. Векторное представление имени вычисляется с помощью выражения

$$v_{name} = \sum_{s \in S_{name}} v_s,$$

где S_{name} – набор токенов для данного имени, а v_s — векторное представление токена S.

В качестве целевой функции используется MarginLoss, ранее применяемая в [42]. В качестве негативных примеров используются имена, присутствующие в кодовой базе, на которой построен граф. Целевая функция имеет вид

$$loss(v_h, v_t, y) = \begin{cases} 1 - \cos(v_h, v_t), & if \;\; y = 1 \\ \max(0, \cos(v_h, v_t)) - margin, & if \;\; y = -1 \end{cases},$$

где y – метка положительного или отрицательного примера.

Задача предварительного обучения GNN-EdgePred типична при обучении автокодировщика на основе графовой нейронной сети. Используя векторное представление узла, необходимо определить, с какими другими узлами в графе он связан. Подобная целевая функция ранее использовалась в других работах посвященных тренировке векторных представлений для исходного кода [5]. В задаче GNN-EdgePred может быть несколько правильных кандидатов. В процессе тренировки позитивный пример выбирается согласно равномерному распределению из списка соседей $N(i)$ узла i. Негативные примеры выбираются случайным образом. При тренировке используется та же целевая функция, что и для задачи GNN-NamePred.

Задача предварительного обучения GNN-TransR заимствована из методов тренировки реляционных векторных представлений. Сами векторные представления вычисляются с помощью рекурсивной формулы (1), а в качестве целевой функции используется TransR ($-|M_r v_h + v_r - M_r v_t|$, где v_h и v_t – векторные представления узлов, соединенных отношением r, v_r – векторное представление отношения, M_r – дополнительная матрица параметров). Этот подход предложен в данной работе и ранее не исследовался для создания предварительно обученных векторных представлений для исходного кода.

Задача предварительного обучения GNN-NodeClf тренируется предсказывать типы узлов в графе. Целевая функция для данной задачи определена выражением

$$loss(v_i, y) = -\sum_{c \in C} y \log(f_c(v_i)),$$

где v_i это векторное представление узла, y – тип узла, C – множество возможных типов узлов, $f_c(v_i)$ – функция оценки правдоподобия принадлежности узла i к классу c. В большинстве случаев тип узла возможно определить по типам связей с соседними узлами. Целевая функция GNN-NodeClf подходит для обучения векторных представлений, которые кодируют локальную структуру в графа.

3.5 Классификация типов переменных

Задачу классификации типов можно формализовать следующим образом. Дан набор токенизированных функций F с разметкой типов переменных L. Типы определены для переменных, обозначенных в сигнатуре функции. Для одной переменной разметка представлена в виде тройки $(start, end, type)$, где $start$ это стартовый токен, end – конечный токен, а $type$ – тип переменной. Набор типов определён в момент тренировки и не может быть расширен. Для каждого токена в соответствие поставлен идентификатор узла в графе, к которому он относится. Цель задачи – определить правильный тип переменной из числа заранее определённых типов.

Для решения этой задачи предложено два подхода. Первый предполагает использование ранее обученных векторных представлений узлов, ассоциируемых с переменными, для классификации типов этих переменных. Второй подход, предполагает одновременное использование представления исходного кода в виде последовательности токенов и в виде графа, и соответствующих векторных представлений. В данной работе сравниваются эти два похода.

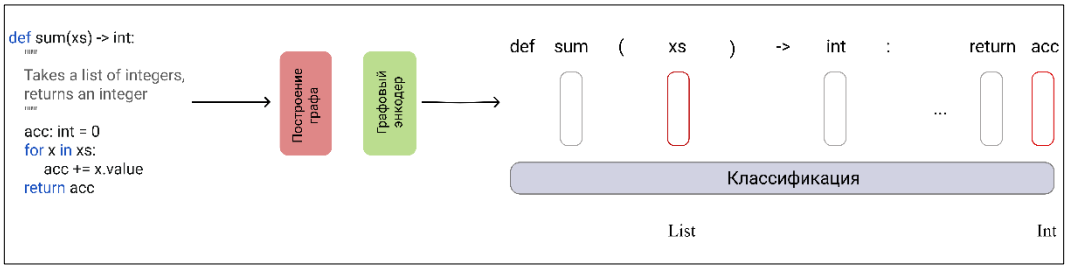

Рис. 3. Архитектура модели TypeClf-Graph для классификации типов переменных. Исходный код представлен в виде графа. Векторные представления переменных (обозначены красной рамкой) передаются на вход классификатора типов

Fig. 3. Architecture of the TypeClf-Graph model for classifying variable types. Source code is represented in the form of a graph. Vector representations of variables (indicated by a red frame) are passed to the input of the type classifier

Первый подход, названный TypeClf-Graph, заключается в классификации типов на основе предварительно обученных векторных представлений. Для классификации типа переменной используется лишь векторное представление соответствующего узла. Выбор узла для классификации является простой задачей, так как всегда известно какие именно узлы в графе соответствуют переменным. Векторное представление подаётся на вход простого классификатора, реализованного при помощи полносвязной нейронной сети. При этом сами векторные представления на этапе тренировки классификатора не обновляются. Архитектура такой модели показана на рис. 3.

Второй подход, названный TypeClf-Hybrid, заключается в классификации типов на основе гибридной модели. Такая модель объединяет в себе представление исходного кода в виде последовательности и в виде графа, что позволяет использовать существующие модели из области обработки естественного языка. Схема работы предложенной гибридной модели представлена на . . В основе лежит текстовый кодировщик, на вход которого исходный код поступает в виде последовательности токенов. Каждому токену в соответствие может быть

поставлен узел из графа. Токены, не несущие семантической нагрузки, могут не иметь соответствующих им узлов в графе (например скобки). В качестве модели обработки токенов могут использоваться любые модели для обработки текста. В данной работе проведены эксперименты со свёрточной моделью TextCNN, основанной на работе [55], и с предварительно обученной моделью CodeBERT, в том числе качестве базовых моделей.

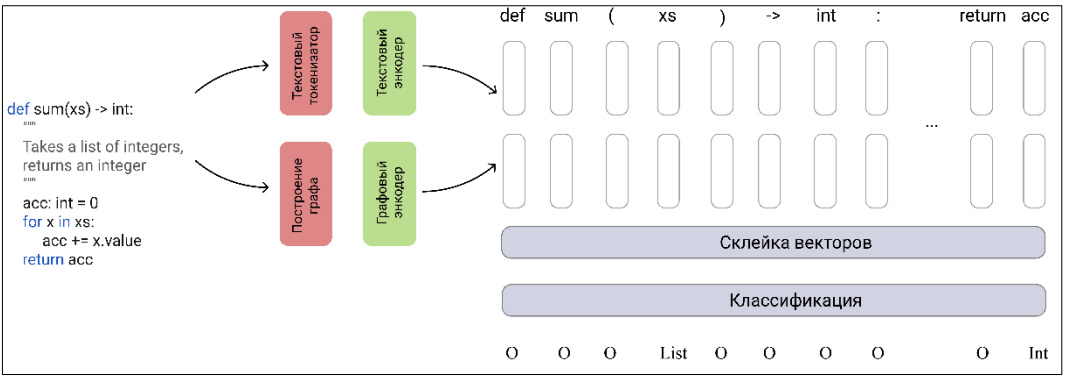

Рис. 4. Архитектура гибридной модели TypeClf-Hybrid. Исходный код представлен в виде последовательности токенов и в виде графа. Два вида векторных представлений конкатенируются и передаются на вход классификатора типов

Fig. 4. Architecture of the TypeClf-Hybrid model. Source code is represented in the form of a sequence of tokens and in the form of a graph. Two kinds of vector representations are concatenated and passed to the input of the type classifier

4. Результаты экспериментов

4.1 Описание наборов данных

В рамках данной работы были созданы два набора данных. Процедура получения наборов данных реализована в составе программного комплекса SourceCodeTools. Новые наборы данных создавались потому, что существующие не содержат информации о глобальных связях в исходном коде. Первый – составлен на основе популярных пакетов Python (далее обозначен PopularPackages, процедура сбора описана в [56]). Второй – основан на наборе данных CodeSearchNet[3] (далее обозначен CSN-Graph). Оба набора данных опубликованы в свободном доступе[4]. Статистика наборов данных приведена в табл. 1. Предварительная тренировка занимает существенное количество времени. Для увеличения количества поведённых экспериментов в основном используется набор данных PopularPackages, который значительно меньше по размеру.

Табл. 1. Статистика наборов данных
Table 1. Dataset statistics

Параметр	Значение
Набор данных PopularPackages	
Количество пакетов в обучающей выборке	142
Средняя степень вершины	6,45
Общее число узлов	2 652 787
Общее число рёбер	10 587 447
Средняя глубина AST	8,38

[3] https://github.com/github/CodeSearchNet
[4] https://disk.yandex.ru/d/GUxvRSPvFxop7g

Параметр	Значение
Набор данных CSN-Graph	
Количество пакетов в обучающей выборке	9 625
Средняя степень вершины	-
Общее число узлов	46 479 185
Общее число рёбер	216 697 812
Средняя глубина AST	16,76

4.2 Набор данных для классификации типов переменных

Для тренировки модели классификации типов переменных был подготовлен ещё один набор данных, основанный на наборе данных PopularPackages. Это решение обусловлено тем, что в числе прочих проводятся эксперименты с реляционными векторными представлениями, которые не имеют возможности обобщаться на новые данные. Чтобы предотвратить утечку информации, все аннотации типов и значения по умолчанию исключены из кодовой базы, на основе которой строится граф. В экспериментах используется две версии набора данных. Первая содержит все возможные типы переменных, а вторая – только 20 самых частых типов, которые составляют 86% всего набора данных.

Финальный набор данных для классификации типов переменных содержит 2938 примеров аннотаций типов. Количество уникальных типов велико из-за того, что в Python существует возможность определения параметрических типов. Чтобы увеличить среднее количество примеров для каждого уникального типа, решено в качестве целевого класса использовать имя основного типа без учёта параметров. Например, тип List[int] упрощается до List. В результате получен набор данных, содержащий 2767 примеров и 89 уникальных меток классов. Аналогичный подход для упрощения типов был применен в [23]. Метрикой, используемой для оценки качества, является HITS@1, показывающая насколько часто правильный тип является первым среди списка предлагаемых типов. Данная метрика эквивалентна точности классификации.

4.3 Используемые вычислительные ресурсы

Для тренировки моделей машинного обучения использовался компьютер с процессором Intel Core i7-7700K, 32Гб оперативной памяти и видеокартой NVIDIA 1080ti (12 Гб). Время предварительной тренировки модели графовой нейронной сети составляет около 1 часа для набора данных PP и 10 дней для набора данных CodeSearchNet-Graph. Время тренировки одной модели реляционных векторных представлений на наборе данных PP размерностью 500 – 8 дней. Время тренировки гибридной модели (TypeClf-Hybrid) без дообучения (300 эпох) – 4 часа.

4.4 Оценка графовых векторных представлений с помощью модели TypeClf-Graph

В данном эксперименте проводится оценка полезности графовых векторных представлений для решения задачи классификации типов. Нейро-классификатор TypeClf-Graph используется для того, чтобы определить, содержат ли векторные представления узлов информацию, ассоциируемую с типами переменных. Классификатор типов представляет собой простую полносвязную нейронную сеть с двумя скрытыми слоями размером 30 и 15. В качестве функции активации используется ReLU. Результат обучения модели классификатора TypeClf-Graph с использованием различных графовых векторных представлений показан в табл. 2. Максимальная точность, достигаемая классификатором реляционных векторных представлений, составляет 52,85 (RotatE). Максимальная точность,

достигаемая классификатором векторных представлений GNN, составляет 59,01 (GNN-NamePred). В качестве базовых моделей используются случайно сгенерированные векторные представления, а также векторные представления FastText [57] (натренированы в рамках данной работы на наборе данных CodeSearchNet со стандартными параметрами, используя библиотеку Gensim), рассчитанные для имён классифицируемых переменных. Все предварительно обученные векторные представления фиксированы и не обновляются в процессе тренировки.

Табл. 2. Точность классификации типов (Hits@k) с помощью модели TypeClf-Graph. Эксперименты повторялись 5 раз. В качестве базовых моделей используются случайно инициализированные вектора (без обучения), а также векторные представления FastText. Размерность всех векторных представлений равна 100

Table 2. Accuracy of type classification (Hits@k) using the TypeClf-Graph model. The experiments were repeated 5 times. Randomly initialized vectors (without training) as well as FastText vector representations are used as base models. The dimension of all vector representations is 100

Метод предварительного обучения	**Hits@1**	**Hits@3**	**Hits@5**
Все типы			
Случайные вектора	12.95±1.72	26.14±3.89	36.14±5.55
FastText	**61.29±0.71**	**79.28±0.95**	**85.86±0.42**
DistMult	45.72±1.99	67.02±2.48	76.98±2.02
RotatE	52.22±1.13	72.11±0.96	80.40±1.09
ComplEx	47.50±2.20	69.58±1.94	77.95±2.06
DistMult $k-hop$	50.51±1.58	72.11±1.46	79.99±2.30
RotatE $k-hop$	**52.85±0.59**	**73.34±1.55**	**80.82±0.90**
ComplEx $k-hop$	49.77±0.94	72.15±2.74	80.40±0.83
GNN-NamePred	**59.01±1.15**	**74.36±0.89**	**80.31±0.95**
GNN-EdgePred	56.81±0.83	73.85±1.08	79.88±1.21
GNN-TransR	46.18±2.34	64.40±2.42	73.58±2.12
GNN-NodeClf	49.05±1.94	68.58±2.51	75.98±1.49
Частые типы			
Случайные вектора	23.88±3.53	44.47±5.13	55.59±7.11
FastText	**65.99±0.97**	**84.25±1.16**	**90.11±1.05**
DistMult	52.33±1.07	74.74±2.13	84.72±1.47
RotatE	59.12±1.58	79.77±1.92	88.78±1.88
ComplEx	53.84±1.12	77.04±1.25	86.41±0.77
DistMult $k-hop$	57.99±1.02	79.34±1.24	88.50±0.63
RotatE $k-hop$	**60.21±1.36**	80.17±2.29	88.58±1.51
ComplEx $k-hop$	56.78±1.41	**80.42±1.95**	**89.02±1.42**
GNN-NamePred	**64.86±1.91**	78.35±1.11	83.83±0.68
GNN-EdgePred	60.79±1.26	**78.45±1.25**	**84.21±1.50**
GNN-TransR	52.92±1.63	71.56±2.55	81.45±2.06
GNN-NodeClf	54.47±1.83	74.84±1.96	82.15±0.86

Из результатов видно, что реляционные векторные представления справляются с задачей классификации типов хуже, чем векторные представления GNN. Можно предположить, что одной из причин для этого является лежащая в основе GNN парадигма передачи сообщений. Переменные, используемые в схожих контекстах, получают сообщения от похожих узлов.

Как следствие, переменные с одинаковым шаблоном использования, имеют схожие векторные представления. Векторные представления FastText, подсчитанные для имён классифицируемых переменных, позволяют достичь самых лучших показателей классификации типов.

4.5 Анализ точности классификации типов

Первая часть анализа заключается в определении зависимости между точностью классификации типа и его частотой (см. рис. 5). Наблюдается три кластера типов. Первый содержит типы вроде object, List, float, Dict, str. Для данного кластера наблюдается увеличение качества классификации с увеличением частотности типа. Второй кластер содержит типы Union, Optional, Any, bytes, bool. Для этих типов характерна низкая точность классификации несмотря на высокую частотность. Третий кластер содержит типы Resolver, Writer, CodeWrite и др. Эти типы являются редкими, но для них наблюдается идеальная точность классификации.

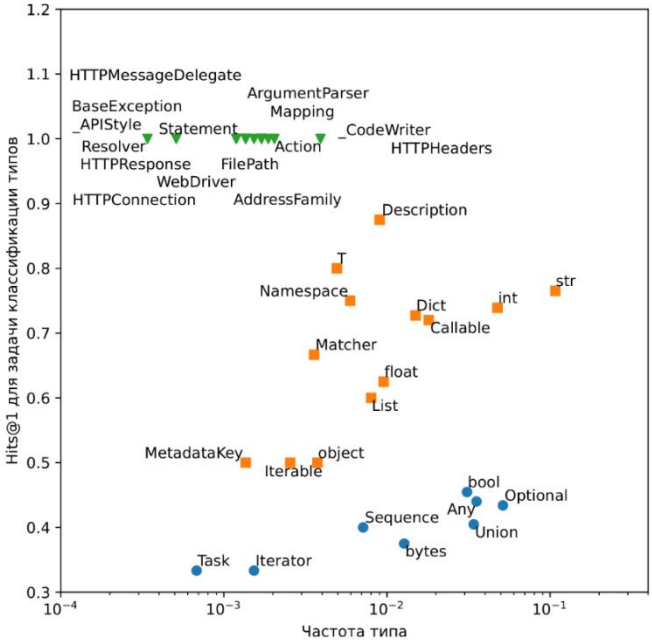

Рис. 5. Точность классификации типов в зависимости от частоты типа. Наблюдаются три кластера: 1) редкие типы, для которых точность классификации высока (FilePath, WebDriver); 2) частые типы, для которых точность классификации низкая (Union, bytes); 3) типы, для которых наблюдается увеличение точности с увеличением частоты (List, int)

Fig. 5. Type classification accuracy depending on type frequency. Three clusters are observed: 1) rare types for which the classification accuracy is high (FilePath, WebDriver); 2) frequent types for which the classification accuracy is low (Union, bytes); 3) types for which there is an increase in accuracy with increasing frequency (List, int)

Вторая часть анализа заключается в анализе матрицы расхождения. Ошибки часто совершаются в пользу частых типов, например, str (22% от всех аннотированных примеров), а также в пользу неоднозначных типов, таких как Optional (10%), Any (7%), Union (6%), Sequence (1.4%), T (0.9%), object (0.7%). Было решено протестировать модель классификации типов после исключения неоднозначных типов. Результат такого теста показан в табл. 3. В результате исключения неоднозначных типов, для векторных представлений GNN-NamePred точность классификации выросла на 35%, а для векторных представлений FastText на 32%.

Табл. 3. Метрика Hits@k классификации частых типов за исключением неоднозначных типов
Table 3. Hits@k metric for classifying common types except for ambiguous types

Метод предварительного обучения	Hits@1	Hits@3	Hits@5
GNN-NamePred	79.93	88.48	91.11
FastText	**81.24**	**93.22**	**96.44**
GNN-NamePred, размерность 500	78.88	87.56	91.77
FastText, размерность 500	82.17	92.96	95.46
CodeBERT, размерность 786	**84.53**	**94.21**	**96.25**

4.6 Анализ влияния размерности на точность классификации типов

В рамках данного эксперимента сравнивается точность классификации переменных при использовании графовых векторных представлений разной размерности. Увеличение количества параметров модели может приводить к улучшению качества работы. В качестве базовой модели используются векторные представления CodeBERT, имеющие размерность 768. Модель CodeBERT доступна из репозитория библиотеки transformer. Результаты данного эксперимента приведены на рис. 6.

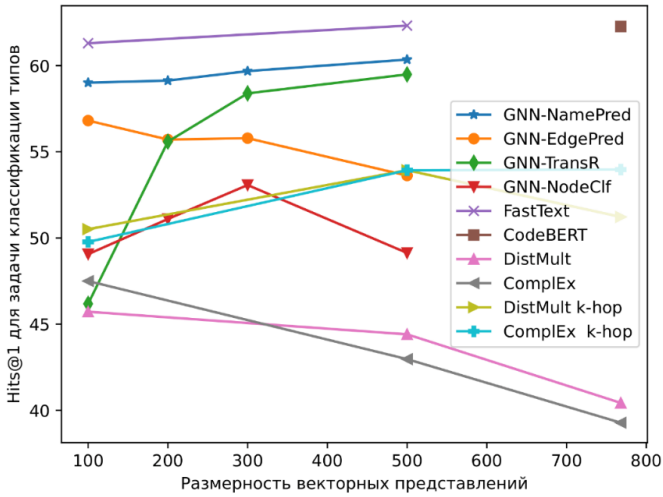

Рис. 6. Точность классификации типов в зависимости от размерности векторных представлений (оценка на всех типах)
Fig. 6. Accuracy of type classification depending on the dimension of vector representations (assessment on all types)

Результаты экспериментов показывают, что качество решения задачи классификации типов улучшается при увеличении размерности векторных представлений GNN-NamePred и GNN-TransR. Для векторных представлений GNN-EdgePred и GNN-NodeClf стабильного улучшения качества классификации при увеличении размерности не наблюдается. Для реляционных векторных представлений при увеличении размерности наблюдается улучшение качества при использовании графа с k-hop рёбрами, и ухудшение – при использовании стандартного графа.

В табл. 4 приведено значение метрик классификации Hits@k для всех типов и для частых типов при использовании векторных представлений размерностью 500. Среди реляционных векторных представлений лучшее качество классификации достигается при обучении на графе, содержащим k-hop рёбра (метод тренировки RotatE не рассматривался ввиду длительного времени тренировки). Среди векторных представлений GNN лучшие показатели

всегда достигаются при использовании GNN-NamePred. Тем не менее векторные представления FastText и CodeBERT позволяют достичь наилучших результатов классификации типов.

Табл. 4. Точность классификации типов Hits@k для графовых векторных представлений (размерность 500), FastText (размерность 500) и векторных представлений CodeBERT (размерность 768)

Table 4. Classification accuracy of Hits@k types for graph vector representations (dimension 500), FastText (dimension 500) and CodeBERT vector representations (dimension 768)

Метод предварительного обучения	Hits@1	Hits@3	Hits@5
Все типы			
DistMult 500	42.45±0.67	60.81±2.75	69.99±2.41
ComplEx 500	41.85±1.55	60.48±1.93	68.77±1.4
DistMult $k-hop$ 500	53.93±1.62	**74.27±1.98**	**81.32±1.78**
ComplEx $k-hop$ 500	**54.19±1.35**	73.75±1.93	80.66±2.00
GNN-NamePred 500	**60.35±0.95**	**76.18±1.12**	**81.49±0.93**
GNN-EdgePred 500	53.62±1.65	69.40±1.74	76.45±0.99
GNN-TransR 500	59.48±0.59	75.70±0.79	81.41±1.00
GNN-NodeClf 500	49.13±2.85	69.21±2.33	76.88±1.19
FastText 500	62.31±0.54	79.52±1.18	**86.33±0.63**
CodeBERT	**65.66±0.76**	**79.60±0.89**	85.00±1.10
Частые типы			
DistMult 500	50.58±2.23	72.01±2.16	80.73±1.35
ComplEx 500	50.19±1.01	71.57±2.17	81.59±2.13
DistMult $k-hop$ 500	61.29±0.98	**81.99±0.90**	**89.19±1.21**
ComplEx $k-hop$ 500	**61.42±0.92**	81.68±1.05	88.75±1.02
GNN-NamePred 500	**65.94±0.31**	**80.88±0.97**	**86.55±0.48**
GNN-EdgePred 500	57.79±1.49	75.36±1.76	82.34±1.54
GNN-TransR 500	62.01±1.77	78.26±1.33	84.49±1.50
GNN-NodeClf 500	55.40±4.12	75.40±1.63	83.37±1.69
FastText 500	65.14±0.72	82.33±2.67	88.13±1.33
CodeBERT	**68.29±1.54**	**84.44±1.13**	**90.63±1.12**

4.7 Оценка влияния длительности тренировки на качество классификации типов

В данном эксперименте выявляется как длительность предварительной тренировки влияет на качество классификации типов. Вначале такой анализ проводится для реляционных векторных представлений. Зависимость точности классификации типов от длительности предварительной тренировки представлена на рис. 7. С увеличением времени тренировки точность классификации типов улучшается, что говорит об улучшении качества векторных представлений. Векторные представления, предварительно обученные методами TransR и RESCAL, демонстрируют самую низкую точность классификации. Самые лучшие результаты получены методом тренировки RotatE при использовании графа с k-hop рёбрами. Однако тренировка такой модели требует значительных ресурсов. По этой причине модель RotatE редко использовалась в других экспериментах. Можно заметить, что точность классификации типов, как правило, выше при использовании графов с k-hop рёбрами

(исключение составляют векторные представления RotatE, натренированные в течение 100 эпох).

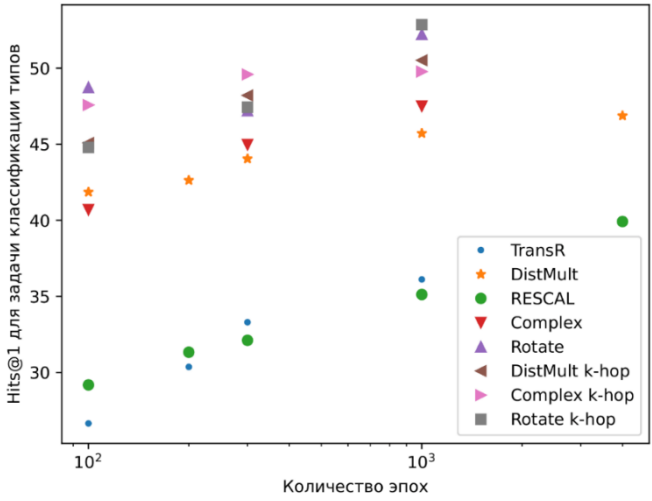

Рис. 7. Зависимость точности классификации типов с помощью реляционных векторных представлений от длительности предварительной тренировки. С увеличением времени тренировки точность классификации улучшается. Точность классификации, как правило, лучше при использовании k-hop векторов (исключение – вектора RotatE, натренированные в течение 100 эпох)
Fig. 7. Relationship between the accuracy of type classification using relational vector representations and the duration of the pretraining. As the pretraining time increases, the classification accuracy improves. Classification accuracy is generally better when using k-hop vectors (the exception is RotatE vectors trained for 100 epochs)

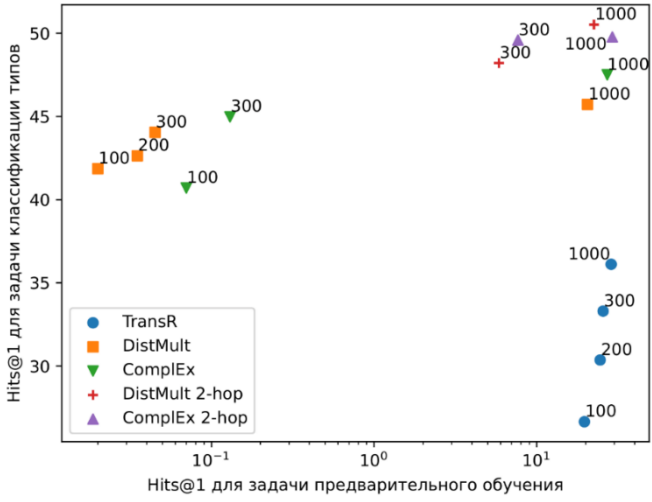

Рис. 8. Влияние длительности предварительного обучения реляционных векторных представлений на точность решения задачи предварительного обучения и задачи классификации типов. Число возле маркера обозначает количество эпох, использованное при тренировке
Fig. 8. Relationship between the duration of pretraining of relational vector representations and the accuracy of solving the pretraining task and the task of type classification. The number next to the marker indicates the number of epochs used during pretraining

Для проверки того, являются ли используемые задачи предварительного обучения полезными для тренировки векторных представлений для исходного кода, проведена проверка зависимости точности решения задачи предварительного обучения и точности классификации типов с использованием полученных векторных представлений. Данная зависимость для реляционных векторных представлений представлена на рис. 8. Наблюдается тенденция, при которой увеличение длительности тренировки (количество эпох) ведёт к увеличению точности решения задачи предварительной тренировки и точности классификации типов.

Далее, подобный анализ был проведён для векторных представлений GNN-NamePred, GNN-EdgePred и GNN-NodeClf (см. рис. 9). Исследование не проводилось для векторных представлений GNN-TransR ввиду ограниченных вычислительных ресурсов.

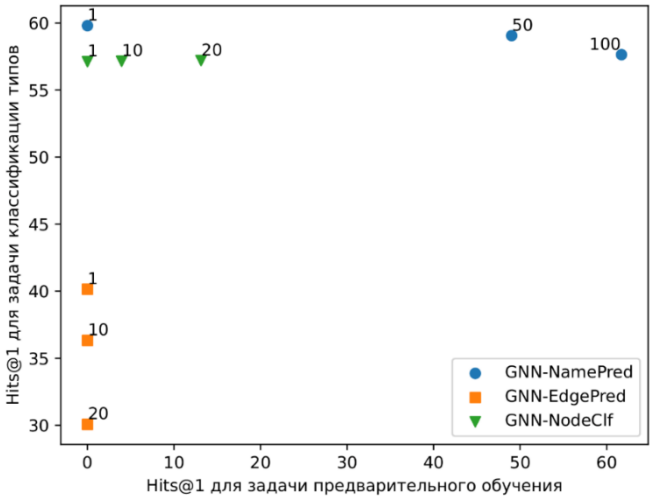

Рис. 9. Влияние длительности предварительного обучения GNN векторных представлений на точность решения задачи предварительного обучения и задачи классификации типов. Число возле маркера обозначает количество эпох, использованное при тренировке

Fig. 9. Relationship between the duration of pre-training of GNN vector representations and the accuracy of solving the pretraining task and the task of type classification. The number next to the marker indicates the number of epochs used during pretraining

Для векторных представлений GNN-NamePred наблюдается незначительный тренд, при котором качество решения задачи предварительного обучения улучшается, а задачи классификации типов незначительно ухудшается или остаётся неизменным.

Для векторных представлений GNN-NodeClf не наблюдается значительного улучшения точности классификации типов при улучшении точности решения задачи предварительно тренировки. В данном эксперименте в процессе предварительной тренировки использовалась скорость обучения, отличная от той, что использовалась для получения значений в табл. 2. С учётом значений из этих таблиц, при достижении точности решения задачи предварительного обучения близкой к единице (0,96), качество решения задачи классификации типов падает до 49,05.

Для векторных представлений GNN-EdgePred точность решения задачи предварительного обучения улучшается незначительно, и в то же время точность классификации типов падает. Данный результат указывает на интересный аспект векторных представлений GNN. Целевая задача GNN-EdgePred похожа на целевые задачи, используемые для тренировки реляционных векторных представлений, тем, что цель тренировки – научиться предсказывать наличие связей в графе на основе векторных представлений узлов. Одно из отличий заключается в том, что методы тренировки реляционных векторных представлений хранят

векторные представления узлов в явном виде, в то время как при использовании графовых нейронных сетей, векторные представления вычисляются на основе окружения узла. С учётом того, что GNN модель не справляется с решением задачи предварительного обучения, можно выдвинуть гипотезу о том, используемая графовая нейронная модель демонстрирует высокое смещение и низкую дисперсию.

4.8 Влияние составляющих графа на качество классификации типов

Граф, используемый для тренировки графовых векторных представлений, содержит большое количество различных типов рёбер. Не все эти рёбра имеют одинаковое влияние на качество финальных векторных представлений. В данной части исследования проводится изучение того, как исключение составляющих графа влияет на качество решения задачи классификации типов. Эксперименты проводились только для векторных представлений GNN. После удаления части графа, предварительная тренировка осуществлялась заново.

Результаты экспериментов представлены в табл. 5. В рамках первого эксперимента из графа были исключены метки типов рёбер. Таким образом, модель графовой нейронной сети R-GCN стала больше похожа на классическую графовую свёрточную нейронную сеть. При этом не наблюдается значительное изменение качества решения задачи классификации типов переменных. Данный факт может сигнализировать о том, что информация о типах рёбер в данный момент недостаточно утилизируется, или типы рёбер не так важны при решении задачи классификации типов. Данные результаты могут отличаться при проверке на других целевых задачах.

Табл. 5. Влияние исключения составляющих графа перед тренировкой графовых представлений GNN на качество решения задачи классификации типов
Table 5. Influence of excluding graph components before training GNN graph representations on the quality of solving the task of type classification

Модификация графа	Hits@1	Hits@3	Hits@5
Стандартный граф	59.01±1.15	74.36±0.89	80.31±0.95
Без глобальных связей	57.12±0.96	74.76±0.83	**80.82±1.32**
Без сабтокенов	58.26±0.46	74.44±0.54	80.43±1.4
Без типов рёбер	**59.33±0.45**	**74.8±1.08**	79.96±0.81

В рамках второго эксперимента из графа были исключены сабтокены. Без сабтокенов, в графе отсутствует информация об именах переменных. Несмотря на это, точность решения задачи классификации типов изменилась незначительно. Данный факт указывает на то, что классификация типов переменных осуществляется в основном за счёт использования структурных признаков, полученных из графа.

В третьем эксперименте из графа были исключены глобальные связи. Изначально предполагалось, что глобальные связи добавляют ценную информацию об использовании частых функций и методов, и могут значительно улучшить качество решения целевых задач. Результаты эксперимента показывают, что исключение глобальных рёбер из графа привело к снижению точности классификации типов, однако данное снижение мало для того, чтобы считать его значительным.

4.9 Влияние набора данных

В предыдущих экспериментах для предварительного обучения использовалась та же кодовая база, что и для классификации типов. Чтобы проверить, могут ли GNN модели для исходного кода обобщаться на новые данные, был подготовлен набор данных CSN-Graph. В данном эксперименте, из-за длительного времени, необходимого для обучения, было проведено сравнение только для моделей, обученных на задаче Name Prediction.

После предварительного обучения был проведён эксперимент по классификации типов с помощью подхода TypeClf-Graph. Результат Hits@1 для всех типов составил 58.85±0.71, а для частых типов – 63.20±0.67. Эти результаты практически совпадают с результатами, полученными при совместном использовании кодовой базы для предварительного обучения и классификации типов (см. табл. 6).

Табл. 6. Оценка метрики Hits@1 на наборе данных для классификации типов, используя векторные представления, предварительно обученные на наборах данных PP и CSN-Graph. Размерность векторных представлений равна 100
Table 6. Estimating the Hits@1 metric on the type classification dataset using vector representations pretrained on the PP and CSN-Graph datasets. The dimension of vector representations is 100

Метод предварительного обучения	Все типы	Частые типы
GNN-NamePred, Набор данных PP	**59.01±1.72**	**64.86±1.91**
GNN-NamePred, Набор данных CSN-Graph	58.85±0.71	63.20±0.67

4.10 Классификация типов с использованием гибридной модели

В данном эксперименте оценивается качество работы гибридной модели TypeClf-Hybrid для классификации типов. Гибридная модель использует одновременно текстовый кодировщик и графовые векторные представления. Были проведены эксперименты с двумя текстовыми кодировщиками: CNN и CodeBERT. На вход CNN модели подаются токены, их префиксы и суффиксы. В качестве векторных представлений для токенов используются вектора FastText, обладающие размерностью 100, предварительно обученные на Python программах из набора данных CodeSearchNet. Вектора для суффиксов и префиксов имеют размерность 50. Они обучаются во время тренировки модели. CNN модель состоит из трёх свёрточных слоёв, обладающих размерностью 40. Допустимое количество токенов в одной последовательности, подаваемой на вход текстового кодировщика, равно 512. Ширина окна CNN равна 10. Векторные представления, полученные на выходе текстового кодировщика, склеиваются с графовыми векторными представлениями. Были проведены эксперименты и с моделями CNN, обладающими большим количеством параметров. Однако увеличение числа параметров приводит к переобучению.

Качество работы гибридной модели TypeClf-Hybrid рассматривается на двух задачах: классификация типов и локализация + классификация. В первой задаче известно местоположение переменных, которые нужно классифицировать. Во второй задаче нужно сначала определить, какие токены должны быть классифицированы.

Ожидается, что гибридная модель с графовыми векторными представлениями должна работать не хуже, чем модель TypeClf-Graph. Кроме того, точность решения задачи локализация + классификация должна быть ниже, чем при решении только задачи классификации.

Табл. 7. Эффективность классификации типов с помощью гибридной модели и реляционных векторных представлений. Реляционные векторные представления имеют размерность 500
Table 7. Type classification using a hybrid model and relational vector representations. Relational vector representations have a dimension of 500

Модель классификации типов	Hits@1 Все типы	Hits@1 Частые типы
CNN, C	58.01±2.1	66.61±0.9
CNN, CL	55.01±1.7	63.33±2.5
CNN + DistMult, C	48.13±2.2	57.26±4.3
CNN + DistMult $k-hop$, C	**52.67±2.0**	**59.89±1.3**
CNN + ComplEx, C	51.27±6.2	55.97±3.9
CNN + ComplEx $k-hop$, C	52.12±2.2	58.98±1.4

Модель классификации типов	**Hits@1 Все типы**	**Hits@1 Частые типы**
CNN + DistMult, CL	-	-
CNN + DistMult $k-hop$, CL	-	-
CNN + ComplEx, CL	-	-
CNN + ComplEx $k-hop$, CL	-	-
CodeBERT, C	62.27±0.2	72.76±0.6
CodeBERT, CL	56.75±0.5	67.48±1.1
CodeBERT + DistMult, C	66.44±0.2	71.54±0.5
CodeBERT + DistMult $k-hop$, C	**66.78±0.4**	71.68±0.4
CodeBERT + ComplEx, C	66.30±0.7	**71.81±0.3**
CodeBERT + ComplEx $k-hop$, C	66.40±0.5	71.64±0.4
CodeBERT + DistMult, CL	55.92±0.3	60.99±0.9
CodeBERT + DistMult $k-hop$, CL	**56.08±0.6**	61.46±1.1
CodeBERT + ComplEx, CL	56.29±0.1	**61.70±0.6**
CodeBERT + ComplEx $k-hop$, CL	55.90±0.5	61.13±0.3

В табл. 7 приведены результаты оценки гибридной модели, использующей реляционные векторные представления (размерность векторных представлений 500). В качестве базовой модели используются текстовая модель CNN и CodeBERT. Рассматривались только модели графовых векторных представлений DistMult и ComplEx, так как они показали лучшие результаты при оценке модели TypeClf-Graph. Векторные представления RotatE с размерностью 500 не проверялись, так как их тренировка занимает слишком много времени.

Результаты показывают, что при использовании реляционных векторных представлений в связке с моделью CNN, лучшая точность классификации достигается при использовании модели DistMult, натренированной на графе с k-hop рёбрами. Не удалось натренировать гибридную модель, использующую CNN кодировщик и графовые векторные представления для решения задачи локализация + классификация, из-за нестабильностей в процессе тренировки. Модель, использующая только CNN кодировщик достигла более высокой точности классификации типов. При использовании реляционных векторных представлений в связке с кодировщиком CodeBERT, результаты классификации типов похожи для разных подходов тренировки векторных представлений. Точность классификации выше, чем при использовании только CodeBERT кодировщика. Точность локализации + классификации не меняется при добавлении реляционных векторных представлений.

Табл. 8. Эффективность классификации типов с помощью гибридной модели и векторных представлений GNN. Графовые векторные представления имеют размерность 500
Table 8. Type classification using hybrid model and GNN vector representations. Graph vector representations have a dimension of 500

Модель классификации типов	**Hits@1 Все типы**	**Hits@1 Частые типы**
CNN, C	58.01±2.1	66.61±0.9
CNN, CL	55.01±1.7	63.33±2.5
CNN + GNN-NamePred, C	65.48±0.8	71.89±0.7
CNN + GNN-EdgePred, C	65.25±0.9	69.49±0.9
CNN + GNN-TransR, C	64.91±1.3	68.63±1.6
CNN + GNN-NodeClf, C	61.60±0.7	68.32±1.8

Модель классификации типов	Hits@1 Все типы	Hits@1 Частые типы
CNN + FastText, C	**67.78±1.2**	**77.06±0.0**
CNN + GNN-NamePred, CL	64.23±0.6	68.78±1.1
CNN + GNN-EdgePred, CL	62.47±0.9	67.04±1.9
CNN + GNN-TransR, CL	62.03±1.4	68.14±1.3
CNN + GNN-NodeClf, CL	58.35±1.0	66.27±0.7
CNN + FastText, CL	**65.60±3.9**	**72.80±0.6**
CodeBERT, C	62.27±0.2	72.76±0.6
CodeBERT, CL	56.75±0.5	67.48±1.1
CodeBERT + GNN-NamePred, C	68.50±0.2	74.53±0.2
CodeBERT + GNN-EdgePred, C	67.83±0.4	74.65±0.2
CodeBERT + GNN-TransR, C	65.66±0.4	73.97±0.3
CodeBERT + GNN-NodeClf, C	65.28±0.5	74.65±0.5
CodeBERT + FastText, C	**70.76±0.2**	**77.06±0.0**
CodeBERT + GNN-NamePred, CL	63.36±0.3	69.39±1.2
CodeBERT + GNN-EdgePred, CL	63.41±0.7	70.38±0.6
CodeBERT + GNN-TransR, CL	60.83±0.5	68.70±0.5
CodeBERT + GNN-NodeClf, CL	61.58±0.6	69.82±0.7
CodeBERT + FastText, CL	**66.41±1.1**	**71.56±0.6**

В табл. 8 приведены результаты экспериментов по оценке качества работы гибридной модели, использующей векторные представления GNN. Гибридная CNN модель достигает точности классификации, схожей с простым классификатором TypeClf-Graph. Более того, точность классификации на частых типах стабильно выше и сравнима с CodeBERT. При классификации всех типов, точность классификации гибридной модели выше, чем модели, использующей только CNN кодировщик или только CodeBERT. На частых типах разница в точности классификации гораздо меньше. Как и ожидалось, задача локализация + классификация является более сложной и приводит к более низким значениям метрики Hits@1. Добавление графовых векторных представлений дает незначительное улучшение при использовании CodeBERT в качестве текстового кодировщика. В целом, точность классификации гибридной моделью, использующей векторные представления GNN выше, чем при использовании реляционных векторных представлений. Однако при использовании векторных представлений FastText можно получить ещё более высокие результаты.

4.11 Влияние предварительного обучения на скорость тренировки

Рис. 10 показывает выигрыш от использования векторных представлений GNN для каждой эпохи. Было проведено сравнение всех гибридных моделей, представленных в табл. 8. Чтобы оценить влияние на динамику обучения, сравнились показатели Hits@1 для каждой эпохи. Для лаконичности приведены только данные о динамике обучения для векторных представлений GNN-NamePred. Для векторных представлений GNN, обученных другими целевыми функциями, динамика аналогична.

Можно заметить, что модели, которые используют графовые векторные представления обучаются быстрее. Для CNN модели, использование дополнительных векторов ускоряет тренировку на десятки эпох. После некоторого момента разница в качество работы перестает меняться, а модели, использующие GNN вектора, сходятся к более высокому значению точности классификации типов. Результаты CodeBERT улучшились более чем на 10%.

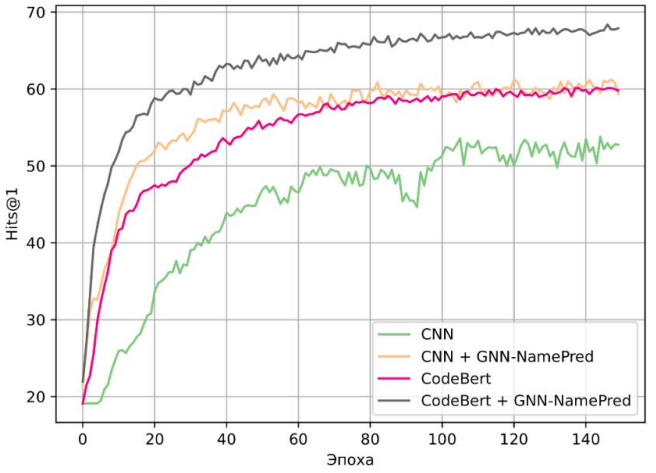

Рис. 10. Точность классификации типов моделью TypeClf-Hybrid в зависимости от эпохи для различных подходов тренировки векторных представлений GNN. Наблюдается улучшение от использования графовых векторных представлений вместе с текстовым кодировщиком по эпохам. При использовании векторов GNN модели обучаются быстрее, финальная точность выше
Fig. 10. Accuracy of type classification by the TypeClf-Hybrid model depending on the epoch for various approaches to training GNN vector representations. There is an improvement from using graphical vector representations along with a text epoch encoder. When using GNN vectors, models are trained faster, the final accuracy is higher

5. Заключение

Разработка предварительно обученных моделей является важным шагом на пути создания интеллектуальных приложений для анализа исходного кода. Большинство существующих предобученных моделей использует методы, созданные для обработки естественного языка. Графовые модели могу служить альтернативой используемым на данный момент подходам. Однако их свойства недостаточно изучены.

В данной статье проведено исследование применения предварительно обученных графовых векторных представлений для решения целевых задач, в частности классификации типов переменных в программах, написанных на языке Python. Было рассмотрено два типа векторных представлений: реляционных и обученных с помощью графовой нейронной сети.

Установлено, что при предварительном обучении графовых векторных представлений для исходного кода следует использовать графовые нейронные сети, так как они позволяют обобщаться на новые данные и показывают лучший результат по сравнению с реляционными векторными представлениями. При увеличении размерности, графовые векторные представления позволяют достичь точности классификации типов схожей с CodeBERT. Более того, совместное использование CodeBERT и графовых векторных представлений позволяет улучшить точность классификации.

Помимо CodeBERT, можно выделить и другие предварительно обученные модели для исходного кода, такие как GraphCodeBERT, UniXCoder и CodeT5. В отличие от CodeBERT, эти подходы в том или ином виде используют информацию из графа программы, которая может позволить сократить разрыв, наблюдаемый между CodeBERT и CodeBERT + GNN-NamePred. В данной работе проведено сравнение векторных представлений, полученных исключительно графовым и исключительно текстовым кодировщиками.

В дальнейшем следует оценить качество графовых векторных представлений на более широком круге задач, таких как поиск ошибок, поиск исходного кода, и генерация текстового описания.

Список литературы / References

[1] Vaswani A. Shazeer N. et al. Attention is all you need. In Proc. of the 31st Conference on Neural Information Processing Systems (NIPS), 2017, 11 p.

[2] Kanade A. Maniatis P. et al. Learning and evaluating contextual embedding of source code. In Proc. of the 37th International Conference on Machine Learning, 2020, pp. 5110–5121.

[3] Feng Z., Guo D. et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proc. of the Conference on Empirical Methods in Natural Language Processing, 2020, pp. 1536-.1547.

[4] Guo D., Ren S. et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. In Proc. of the Ninth International Conference on Learning Representations, 2021, 18 p.

[5] Liu L., Nguyen H. et al. Universal Representation for Code. Lecture Notes in Computer Science, vol. 12714, 2021, pp. 16-28.

[6] Nguyen A.T., Nguyen T.N. Graph-Based Statistical Language Model for Code. In Proc. of the 37th IEEE International Conference on Software Engineering, 2015, pp. 858-868.

[7] Alon U., Sadaka R. et al. Structural language models of code. In Proc. of the 37th International Conference on Machine Learning, 2020, pp. 245-256.

[8] Yang Y., Chen X., Sun J. Improve Language Modelling for Code Completion by Tree Language Model with Tree Encoding of Context. In Proc. of the 31st International Conference on Software Engineering and Knowledge Engineering, 2019, pp. 675–680.

[9] Hellendoorn V.J., Sutton C. et al. Global Relational Models of Source Code. In Proc. of the Eighth International Conference on Learning Representations, 2020, 10 p.

[10] Pandi V., Barr E.T. et al. OptTyper: Probabilistic Type Inference by Optimising Logical and Natural Constraints. arXiv preprint arXiv:2004.00348, 2020, 29 p.

[11] Chirkova N., Troshin S. Empirical study of transformers for source code. In Proc. of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2021, pp. 703-715.

[12] Buratti L., Pujar S. et al. Exploring Software Naturalness through Neural Language Models. arXiv preprint arXiv:2006.12641, 2020, 12 p.

[13] Ahmad W.U., Chakraborty S. et al. Unified pre-training for program understanding and generation. In Proc. of the Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2021, pp. 2655–2668.

[14] Wang Y., Wang et al. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In Proc. of the Conference on Empirical Methods in Natural Language Processing, 2021, pp. 8696-8708.

[15] Guo D., Lu S. et al. UniXcoder: Unified Cross-Modal Pre-training for Code Representation. In Proc. of the 60th Annual Meeting of the Association for Computational Linguistics, vol. 1: Long Papers, 2022, pp. 7212-7225.

[16] Karmakar A., Robbes R. What do pre-trained code models know about code? In Proc. of the 36th IEEE/ACM International Conference on Automated Software Engineering, 2021, pp. 1332-1336.

[17] Cui S., Zhao G. et al. PYInfer: Deep Learning Semantic Type Inference for Python Variables. arXiv preprint arXiv:2106.14316, 2021, 12 p.

[18] Hellendoorn V.J., Bird C. et al. Deep learning type inference. In Proc. of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 152-162.

[19] Malik R.S., Patra J., Pradel M. NL2Type: Inferring JavaScript Function Types from Natural Language Information, In Proc. of the IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 304-315.

[20] Boone C., de Bruin N. et al. DLTPy: Deep Learning Type Inference of Python Function Signatures using Natural Language Context. arXiv preprint arXiv:1912.00680, 2019, 10 p.

[21] Pradel M., Gousios G. et al. Typewriter: Neural type prediction with search-based validation. In Proc. of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2020, pp. 209-220.

[22] Raychev V., Vechev M., Krause A. Predicting program properties from "Big Code". In Proc. of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2015, pp. 111-124.

[23] Allamanis M., Barr E.T. et al. Typilus: Neural type hints. Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), 2020, pp. 91-105.

[24] Peng Y., Gao C. et al. Static inference meets deep learning: a hybrid type inference approach for Python. In Proc. of the 44th International Conference on Software Engineering, 2022, pp. 2019-2030.
[25] Wei J., Goyal M. et al. LambdaNet: Probabilistic type inference using graph neural networks. In Proc. of the Eighth International Conference on Learning Representations, 2020, 11 p.
[26] Ye F., Zhao J., Sarkar V. Advanced Graph-Based Deep Learning for Probabilistic Type Inference. arXiv preprint arXiv:2009.05949, 2020, 25 p.
[27] Fernandes P., Allamanis M., Brockschmidt M. Structured Neural Summarization. In Proc. of the Seventh International Conference on Learning Representations, 2019, 18 p..
[28] Cvitkovic M., Singh B., Anandkumar A. Deep Learning On Code with an Unbounded Vocabulary. In Proc. of the Machine Learning for Programming (ML4P) Workshop at Federated Logic Conference (FLoC), 2018, 11 p..
[29] Dinella E., Dai H. et al. Hoppity: Learning Graph Transformations To Detect and Fix Bugs in Programs. In Proc. of the Eighth International Conference on Learning Representations, 2020, 17 p.
[30] Wang Y., Gao F. et al. Learning a static bug finder from data. arXiv preprint arXiv:1907.05579, 2019, 12 p.
[31] Zhou Y., Liu S. et al. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In Proc. of the 33rd International Conference on Neural Information Processing Systems (NIPS), 2019, pp. 10197-10207.
[32] Brauckmann, A. Goens, S. Ertel and J. Castrillon. Compiler-based graph representations for deep learning models of code. In Proc. of the 29th International Conference on Compiler Construction, 2020, pp. 201-211.
[33] Wan Y., Shu J. et al. Multi-modal attention network learning for semantic source code retrieval. In Proc. of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2019, pp. 13-25.
[34] Wang W., Li G. et al. Detecting Code Clones with Graph Neural Networkand Flow-Augmented Abstract Syntax Tree. In Proc. of the IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), 2020, pp. 261-271.
[35] Li Y., Wang S. et al. Improving Bug Detection via Context-Based Code Representation Learning and Attention-Based Neural Networks. Proceedings of the ACM on Programming Languages, vol. 3, issue OOPSLA, 2019, article no. 162, 30 p.
[36] Ben-Nun T., Jakobovits A.S., Hoefler T. Neural code comprehension: A learnable representation of code semantics. In Proc. of the 32nd International Conference on Neural Information Processing Systems (NIPS), 2018, pp. 3589-3601.
[37] DeFreez D., Thakur A.V., C. Rubio-Gonzalez A.V..Path-based function embedding and its application to error-handling specification mining, In Proc. of the 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, 2018, pp. 423-433, 2018.
[38] Brockschmidt M., Allamanis M. et al. Generative Code Modeling with Graphs. In Proc. of the Seventh International Conference on Learning Representations, 2019, 24 p.
[39] Lu D. Tan N. et al. Program classification using gated graph attention neural network for online programming service. arXiv preprint arXiv:1903.03804, 2019, 12 p.
[40] Zhang J., Wang X. et al. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In Proc. of the IEEE/ACM 41st International Conference on Software Engineering (ICSE), 2019, pp. 783-794.
[41] Allamanis M., Brockschmidt M., Khademi M. Learning to Represent Programs with Graphs. In Proc. of the 6th International Conference on Learning Representations (ICLR), 2018, 17 p.
[42] Hamilton W.L., Ying R., Leskovec J. Inductive representation learning on large graphs. In Proc. of the 31st International Conference on Neural Information Processing Systems (NIPS), 2017, pp. 1025-1035.
[43] Wang Z., Ren Z. et al. Robust embedding with multi-level structures for link prediction. In Proc.of the Twenty-Eighth International Joint Conference on Artificial Intelligence, 2019, pp. 5240–5246.
[44] Schlichtkrull T.N., Kipf P. et al. Modeling Relational Data with Graph Convolutional Networks. Lecture Notes in Computer Science, vol. 10843, 2018, pp. 593-607.
[45] Cai, L. Yan B, et al. TransGCN: Coupling transformation assumptions with graph convolutional networks for link prediction. In Proc. of the 10th International Conference on Knowledge Capture (K-CAP), 2019, pp. 131-138.
[46] Liu X., Tan H. et al. RAGAT: Relation Aware Graph Attention Network for Knowledge Graph Completion. IEEE Access, vol. 9, 2021, pp. 20840–20849.

[47] Allamaras M., Chanthirasegaran P. et al. Learning continuous semantic representations of symbolic expressions. In Proc. of the 34th International Conference on Machine Learning, vol. 70, 2017, pp. 80–88.
[48] Kudo T., Richardson J. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In. Proc. of the Conference on Empirical Methods in Natural Language Processing: System Demonstrations, 2018, pp. 66-71.
[49] Lin Y., Liu Z. et al. Learning entity and relation embeddings for knowledge graph completion. In. Proc. of the Twenty-Ninth AAAI Conference on Artificial Intelligence, 2015, pp. 2181–2187.
[50] Yang B., Yih W. et al. Embedding entities and relations for learning and inference in knowledge bases. arXiv preprint arXiv:1412.6575, 2014, 12 p.
[51] Nickel M., Tresp V., Kriegel H.-P. A three-way model for collective learning on multi-relational data. In Proc. of the 28th International Conference on International Conference on Machine Learning, 2011, pp. 809-816.
[52] Trouillon T., Welbl J. et al. Complex embeddings for simple link prediction. In Proc. of the 33rd International Conference on International Conference on Machine Learning, 2016, pp. 2071-2080.
[53] Sun Z., Deng Z.-H. et al. Rotate: Knowledge graph embedding by relational rotation in complex space. In Proc. of the Seventh International Conference on Learning Representations, 2019, 18 p.
[54] Ling X., Wu L. et al. Deep graph matching and searching for semantic code retrieval. ACM Transactions on Knowledge Discovery from Data (TKDD), vol. 15, issue 5, 2021, article no. 88, 21 p.
[55] Collobert R., Weston J. et al. Natural language processing (almost) from scratch. Journal of Machine Learning Research, vol. 12, 2011, pp. 2493-2537.
[56] Romanov V., Ivanov V., Succi G. Representing Programs with Dependency and Function Call Graphs for Learning Hierarchical Embeddings. In Proc. of the 22nd International Conference on Enterprise Information Systems (ICEIS), vol. 2, 2020, pp. 360-366.
[57] Bojanowski P., Grave E. et al. Enriching word vectors with subword information. Transactions of the Association for Computational Linguistics, vol. 5, 2017, pp. 135-146.

Информация об авторах / Information about authors

Владимир Владимирович ИВАНОВ – кандидат физико-математических наук, доцент. Научные интересы: анализ данных, машинное обучение, разработка программного обеспечения, компьютерная лингвистика, обработка естественного языка, извлечение информации, анализ текста, надежность программного обеспечения и метрики программного обеспечения.

Vladimir Vladimirovich IVANOV – Candidate of Physical and Mathematical Sciences, Associate Professor. Research interests: data analysis, machine learning, software development, computer linguistics, natural language processing, information extraction, text analysis, software reliability and software metrics.

Виталий Анатольевич РОМАНОВ – аспирант. Работал исследователем и инструктором в университете Иннополис с 2016 года. Научные интересы: обработка естественного языка, компьютерная лингвистика, глубокое обучение, большие данные и генеративные модели.

Vitaly Anatolyevich ROMANOV – PhD student. Worked as a researcher and instructor at Innopolis University since 2016. Research interests: natural language processing, computational linguistics, deep learning, big data and generative models.