Научная статья

УДК 004.4`2
DOI 10.25205/1818-7900-2024-22-2-57-67

Использование цепей Маркова для автоматического завершения исходного кода программы

Владислав Сергеевич Тимофеев

Новосибирский государственный университет
Новосибирск, Россия

v.timofeev@g.nsu.ru; https://orcid.org/0009-0008-1060-4613

Аннотация

В сфере программирования используются разнообразные инструменты с целью оптимизации процесса разработки. Среди них особое место занимают интегрированные среды разработки (IDE), обеспечивающие широкий спектр сервисов, включая текстовый редактор, отладчик и интеллектуальное завершение кода. Настоящая работа посвящена разработке модели, направленной на предсказание вариантов завершения исходного кода программы. Для улучшения точности модели были использованы комбинации цепей Маркова, основанные на различных методах вычисления текущего контекста программы: линейном и с использованием абстрактного синтаксического дерева (AST). Линейный метод анализа контекста представляет собой анализ токенизированного представления исходного кода, в то время как второй метод использует структуру исходного кода в виде AST. Объединение различных моделей позволяет сохранить больше семантической информации о коде и учитывать при автодополнении индивидуальный стиль написания кода. Разработанная модель демонстрирует высокую точность предсказаний при минимальном объеме вычислительных ресурсов, что делает ее применимой в интегрированных средах разработки.

Ключевые слова

автоматическое завершение кода, цепи Маркова, нейронные сети, интегрированные среды разработки, язык программирования Pascal

Для цитирования

Тимофеев В. С. Использование цепей Маркова для автоматического завершения исходного кода программы // Вестник НГУ. Серия: Информационные технологии. 2024. Т. 22, № 2. С. 57–67. DOI 10.25205/1818-7900-2024-22-2-57-67

Code Completion Using Markov Chains

Vladislav S. Timofeev

Novosibirsk State University, Novosibirsk, Russian Federation

v.timofeev@g.nsu.ru; https://orcid.org/0009-0008-1060-4613

Abstract

Modern software engineers use many tools to speed up the development process. Many of them use integrated development environments (IDEs), which provide services such as text editors, debuggers and even intelligent code completion. This paper is dedicated to the development of a model for predicting variants of program source code termination. To improve the accuracy of the model, we used combinations of Markov chains constructed using different ways of calculating the current context of the program: linear and with AST. The linear way of computing the context is an analysis of the tokenized representation of the source code. The second method, on the other hand, uses a representation of the

source code in the form of an abstract syntax tree. Combining the different models preserves more semantic information about the code, also adding the ability to support custom code writing style features. In order to compare the different models, a new dataset has been created specifically for the Pascal language. A detailed comparison of the working mechanisms as well as the prediction accuracy on the collected data is given. The proposed model showed high enough accuracy of predictions with minimal computation costs, which allows using it in integrated development environments.

Keywords

code completion, Markov chains, neural networks, integrated development environment, Pascal programming language

For citation

Timofeev V. S. Code completion using Markov chains. *Vestnik NSU. Series: Information Technologies*, 2024, vol. 22, no. 2, pp. 57–67 (in Russ.) DOI 10.25205/1818-7900-2024-22-2-57-67

Введение

Важной частью современных интегрированных сред разработки (англ. integrated development environment, или IDE) является поддержка автоматического завершения исходного кода программы [1]. Данная технология в реальном времени предлагает пользователю подсказки с информацией о возможных вариантах завершения уже написанной части исходного кода. Самые простые реализации используют введенный пользователем префикс и предлагают список доступных сущностей языка (названия функций, классов, переменных и т. д.), предотвращая синтаксические ошибки и избавляя разработчиков от необходимости постоянно обращаться к документации за счет подсказок. Однако особенный интерес представляют модели, учитывающие текущий контекст. Такие модели выдают более корректную информацию, значительно ускоряя процесс разработки и помогая предотвратить семантические ошибки.

В данной статье описана разработка алгоритма автоматического завершения исходного кода программы с использованием комбинации из нескольких цепей Маркова, построенных с помощью различных способов вычисления текущего контекста программы. Последовательно проведены краткий обзор и анализ существующих решений, описаны особенности использования цепей Маркова в контексте рассматриваемой предметной области, представлен разработанный автором алгоритм завершения кода, а также проведен анализ результатов работы алгоритма. Проверка алгоритма осуществлялась на специально собранном наборе данных, состоящем из более чем 20000 программ.

Обзор существующих работ

В настоящее время уже существует достаточно много различных решений. Среди наиболее простых подходов стоит упомянуть алгоритм лексикографической сортировки ответов с заданным префиксом. Более продвинутые методы основаны на статистических подходах, включающих подсчет использования каждой языковой сущности и последующую сортировку ответов в соответствии с их частотой. Также существуют алгоритмы, которые вместо префикса используют различные варианты сокращений, например, аббревиатуры [2]. Тем не менее эти алгоритмы не учитывают текущий контекст пользователя, что приводит к снижению точности предсказаний [3].

С ростом популярности нейронных сетей появились модели, показывающие значительный прирост точности. Например, модель, использующая сети долгой краткосрочной памяти (англ. long short-term memory, или LSTM) [4]. В данном подходе сначала векторизуется токенизированный исходный код, затем применяются рекуррентные нейронные сети, учитывающие текущий контекст. Это значительно повышает точность работы модели. В статье [5] приводится подробное сравнение результатов работы различных подходов. Самую высокую точность показывает модель с использованием LSTM. В более актуальной работе [6] удалось повысить точность данной LSTM-модели за счет расширения набора данных для обучения. Однако

представленные в этих статьях решения направлены на предсказание только вызовов методов из популярных Python-библиотек, тогда как ключевая особенность данной работы заключается в разработке алгоритма, способного предсказывать любые языковые сущности, а также учитывающего особенности пользовательского кода. Помимо этого, LSTM-модель требует наличия значительного набора данных и вычислительных мощностей для обучения. Сбор такого количества данных для Pascal является проблематичным, так как Python является намного более популярным языком программирования. Также следует отметить, что из-за своей архитектуры представленные выше решения не расширяемы на другие языки программирования, а также достаточно чувствительны к пользовательскому стилю написания кода.

Разработка алгоритма автоматического завершения кода

В качестве языка программирования, для которого алгоритм будет предлагать варианты завершения кода, был выбран язык Pascal. Данный язык обладает относительно простой грамматикой, на которой значительно проще проверять гипотезы при построении модели. Предполагается, что решение может быть легко использовано для других языков, использующих грамматику Pascal. Например, многие языки для программируемых логических контроллеров: Structured Text (ST) или Structured Control Language (SCL), poST и т. д. Помимо этого, данное решение является частью разрабатываемой на данной момент IDE для Pascal с помощью инструмента Xtext [7].

Использование цепей Маркова

Для того чтобы точнее предсказывать, что сейчас будет писать пользователь, нам важно учесть контекст, проанализировав уже написанный пользователем код. Использование методов глубокого обучения для работы с исходным кодом программ является довольно сложной задачей, показывающей при этом относительно низкий прирост точности [8], поэтому для решения этой задачи предлагается использовать другой подход, а именно построение модели с использованием цепей Маркова. Цепь Маркова – это стохастическая модель, описывающая последовательность возможных событий, в которой вероятность каждого события зависит только от состояния, достигнутого в предыдущем событии [9]. Таким образом, предлагается накапливать предыдущие языковые сущности, чтобы предсказать следующую с помощью этой модели.

В качестве состояния предлагается использовать не сами языковые сущности, а их обезличенные типы: вызов процедуры, вызов функции, название переменной, название константы, название типа, строковый литерал, ключевое слово языка и т. д. Другими словами, задача цепи Маркова – предсказать следующий тип языковой сущности. Далее результат предсказания может быть использован как параметр других алгоритмов автоматического завершения кода. В данной статье рассмотрено следующее применение модели: список всех доступных пользователю языковых сущностей фильтруется по указанному префиксу и сортируется согласно вероятности их типа, полученной с помощью цепи Маркова. Рассмотрим алгоритм работы модели на конкретном примере (рис. 1).

В блоке (0) объявлены различные языковые сущности, для простоты все они имеют общий префикс «ABC1». Пользователь запрашивает варианты завершения для префикса «AB». Действие (1) обозначает сбор всех вариантов ответа, имеющих заданный префикс. Далее происходит вызов цепи Маркова (2), которая выдает типы возможных ответов и их вероятности. Затем объединяются полученные результаты (3): конкретные языковые сущности сортируются по вероятности их типа, и пользователю предлагается отсортированный список возможных

завершений. Стоит отметить, что для запроса предсказаний из собранного контекста формируется несколько ключей различной длины (рис. 2).

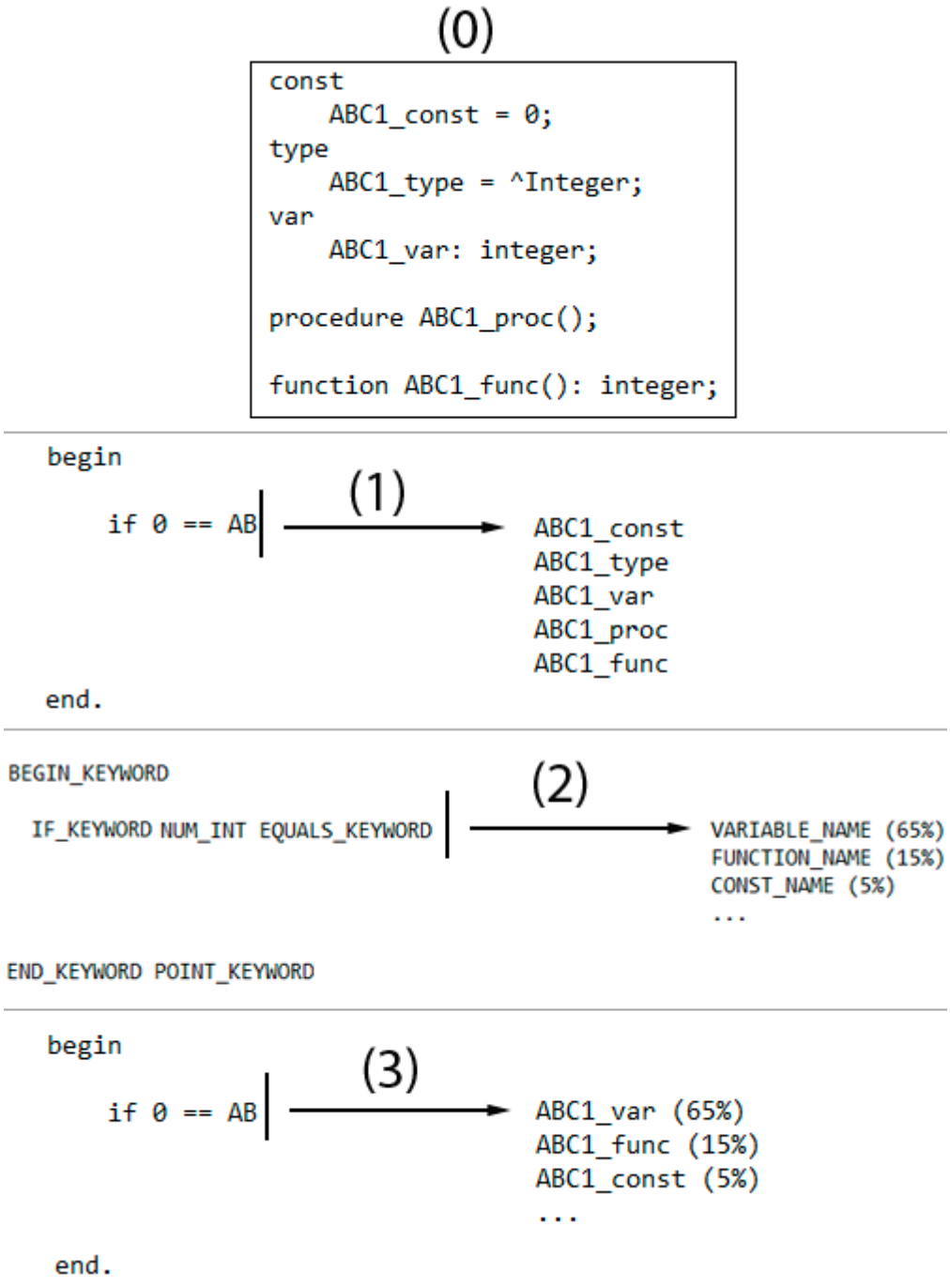

Рис. 1. Пример работы модели
Fig. 1. Example of model operation

```
type Predict:
    String tokenType
    Float probability

Predict[] getPredicts(String fileContent, Int offset, Int maxContextSize):
    String[] inputContext = getContext(fileContent, offset, maxContextSize)
    Predict[] predicts = []
    for (Int i = 0; i < inputContext.size; i++):
        String[] partialContext = subArray(inputContext, i, inputContext.size)
        Predict[] partialPredict = predictByMarkovChain(partialContext)
        predicts.add(partialPredict)
    Predict[] sortedPredicts = sortByProbability(predicts)
    return sortedPredicts
```

Рис. 2. Псевдокод алгоритма получения предсказаний
Fig. 2. Pseudocode of the algorithm for obtaining predictions

Линейный сбор контекста

Под линейным сбором контекста подразумевается, что исходный код представлен в виде последовательности токенов. Далее это представление используется для получения множества цепей событий, формируя непрерывные подмножества токенов различной длины. Рассмотрим упрощенный пример (рис. 3) формирования цепей Маркова с количеством событий равным трем, в котором, для простоты, не учитываются некоторые токены, состоящие из одного символа.

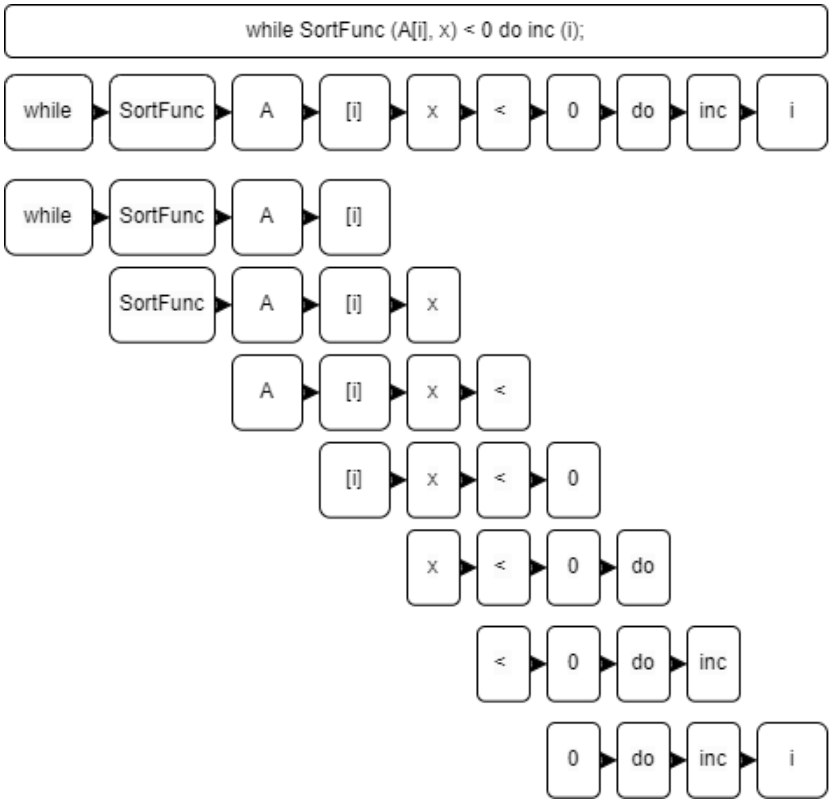

Рис. 3. Упрощенный пример формирования цепей Маркова
Fig. 3. A simplified example of Markov chain formation

```
type Token:
    Int startOffset
    Int endOffset
    String text
    String type

String[] getContext(String fileContent, Int offset, Int maxContextSize):
    Token[] tokens = lexer(fileContent)
    for (Int i = 0; i < tokens.size; i++):
        Token token = tokens[i]
        if (token.startOffset <= offset && token.endOffset <= offset):
            String[] tokenTypes = []
            for (Int j = max(0, i - maxContextSize); j < i; j++):
                tokenTypes.add(tokens[j].type)
            return tokenTypes
    return []
```

Рис. 4. Псевдокод алгоритма линейного сбора контекста
Fig. 4. Pseudocode of the linear context collection algorithm

Составив и обезличив все такие цепочки из набора данных для обучения, мы можем рассчитать для данного набора типов вероятности следующих типов токенов. Все цепочки разбиваются на пары ключ – значение. Ключом выступает набор типов, а значением – список типов, которые могут идти после ключа, с их вероятностями. Когда алгоритму поступает запрос на завершение, достаточно собрать указанное заранее количество токенов слева, сформировать из них ключ и получить список следующих вероятных типов токенов (рис. 4). Стоит отметить, что с использованием хеш-таблиц данное решение работает по времени за O(1) и состоит из набора примитивных операций, что позволяет иметь несколько различных наборов цепочек и агрегировать их результат.

Сбор контекста с использованием AST

Абстрактное синтаксическое дерево (англ. abstract syntax tree, или AST) – конечное помеченное ориентированное дерево, в котором внутренние вершины сопоставлены (помечены) с операторами языка программирования, а листья – с соответствующими операндами.

Если внимательнее изучить пример формирования цепей (рис. 3), то можно заметить, что при линейном сборе контекста довольно быстро (всего лишь за пару токенов) теряется информация о глобальном контексте. Начиная со второй цепи, ключевое слово «while» было полностью утеряно, т. е. модель не понимает, что пользователь, работая на данный момент с условием или телом цикла, хочет получить релевантные варианты завершения этой синтаксической конструкции. Данная проблема приводит к понижению точности предсказаний. Для ее решения предлагается использовать второй набор цепей Маркова, который был собран с использованием AST. Рассмотрим, как выглядит тот же код, но в виде упрощенного синтаксического дерева (рис. 5).

Теперь, когда алгоритму поступает запрос на завершение, он с использованием знания о позициях всех элементов в файле находит ближайшую к позиции курсора вершину, затем ее родительскую вершину, затем ее левого родственника, а после снова родительскую вершину и так далее, пока не наберется необходимое количество вершин (рис. 6). Эта цепочка эквивалентна пути от листа по родителям в LCRS-представлении *n*-арного дерева (англ. left-child right-sibling, или LCRS) [10]. Далее, как и в линейном варианте, алгоритм формирует ключ, получает из хеш-таблицы предсказание цепи Маркова и формирует ответ. Стоит отметить,

что в IDE зачастую уже имеется построенное AST, поэтому данное решение имеет ту же асимптотику временной сложности, что и линейная версия алгоритма.

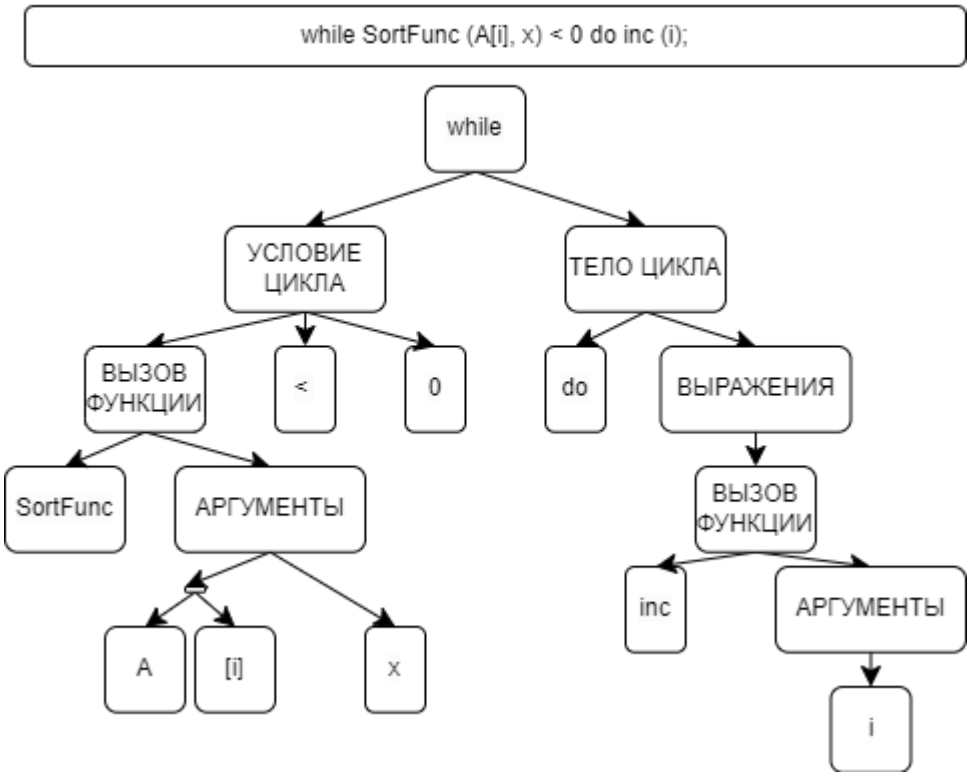

Рис. 5. Упрощенный пример AST
Fig. 5. Simplified AST example

```
type AstNode:
    Int startOffset
    Int endOffset
    String text
    String type
    AstNode parent
    AstNode leftSibling

String[] getContext(String fileContent, Int offset, Int maxContextSize):
    AstNode root = parse(fileContent)
    AstNode node = getDeepestNodeByOffset(root, offset)
    String[] nodeTypes = []
    while(nodeTypes.size < maxContextSize):
        if (node.leftSibling == NULL):
            node = node.parent
            if (node == NULL):
                break
        else:
            node = node.leftSibling
        nodeTypes.add(node.type)
    return nodeTypes
```

Рис. 6. Псевдокод алгоритма сбора контекста с использованием AST
Fig. 6. Pseudocode of the context collection algorithm using AST

Комбинирование моделей

Одна объемлющая синтаксическая конструкция обычно включает в себя несколько инструкций или выражений. Так как линейный контекст содержит больше информации о самих выражениях, то ожидается, что он зачастую будет показывать результат лучше, чем цепь, обученная на контексте из AST. Однако знание о структуре кода может повысить точность предсказаний для некоторых специфичных токенов, завершающих синтаксические конструкции [11]. Поэтому итоговый ответ предлагается формировать, объединив результаты цепи с линейным контекстом и цепи с AST.

Обе цепи были получены путем обучения на специально собранном наборе данных из более чем 20000 программ на языке Pascal. Важно отметить, что в статье подробно рассмотрены лишь два способа построения цепей, однако архитектура системы позволяет легко добавлять новые способы сбора контекста. Так, в модель были добавлены еще две цепи: линейная и с AST. Однако эти цепи обучаются на текущем пользовательском проекте, что позволяет учитывать стиль программирования пользователя. Для тестирования был создан проект, состоящий всего из 2000 строк кода, в котором для эмуляции стиля программирования во всех местах, где это возможно, результаты вызова функций передавались в качестве аргументов другой функции напрямую, без размещения в промежуточных переменных. Рассмотрим простой пример кода (рис. 7), где промежуточный результат присваивается в переменную, и тот же пример кода, но отредактированный под данную особенность стиля написания кода (рис. 8).

```
begin
        maxA = Max(a1, a2);
        maxB = Max(b1, b2);
        max = Max(maxA, maxB);
end.
```

Рис. 7. Пример кода
Fig. 7. Code example

```
begin
        max = Max(Max(a1, a2), Max(b1, b2));
end.
```

Рис. 8. Пример конкретной особенности стиля написания кода
Fig. 8. An example of a specific feature of code writing style

Зачастую в обучающем наборе использовался первый стиль написания, поэтому заранее обученные цепи при тестировании на пользовательском проекте после токена «FUNCTION_NAME» (вызов функции) предлагали самый вероятный следующий токен «VARIABLE_NAME» (имя переменной). Однако цепи, обученные на самом проекте, содержащем вышеописанные особенности стиля, в той же ситуации предлагали «FUNCTION_NAME» как более вероятный токен, что подтверждает гипотезу о применимости подхода для учета пользовательского стиля написания кода. Однако дальнейшее тестирование данной идеи осложняется

отсутствием в наборе данных достаточного количества проектов с ярко выраженными особенностями написания кода.

Метрики оценки решения

Для оценки решения использовались наиболее популярные в смежных исследованиях метрики:

1. Точность топ-1. Количество ответов, когда самый вероятный ответ модели являлся правильным.

2. Точность топ-2. Количество ответов, когда среди двух самых вероятных ответов модели один являлся правильным.

3. Степень уверенности модели. Так как модель предоставляет вероятности для каждого ответа, полученные вероятности для правильного ответа складывались. Если правильный ответ совсем отсутствовал в списке результатов, то его вероятность считалась равной нулю. Данная метрика применима только для оценки работы самих моделей, предсказывающих типы токенов.

Все метрики собирались для каждого вида токенов отдельно, что позволило провести анализ полученных данных и внести коррективы в модель.

Собранный набор данных был разделен на две части: 90 % данных использовалось для обучения модели, оставшиеся 10 % – для тестирования.

Результаты работы алгоритма

Для начала рассмотрим наиболее интересные результаты в разрезе типов токенов (рис. 9). Можно заметить, что модель, построенная на контексте, собранном с помощью AST, работает для ключевых слов языка (keywords) значительно лучше, чем модель с линейным контекстом. Тем не менее она плохо справляется в среднем – из-за низкой точности других типов токенов. Именно поэтому за счет комбинирования цепей удалось значительно увеличить точность предсказаний.

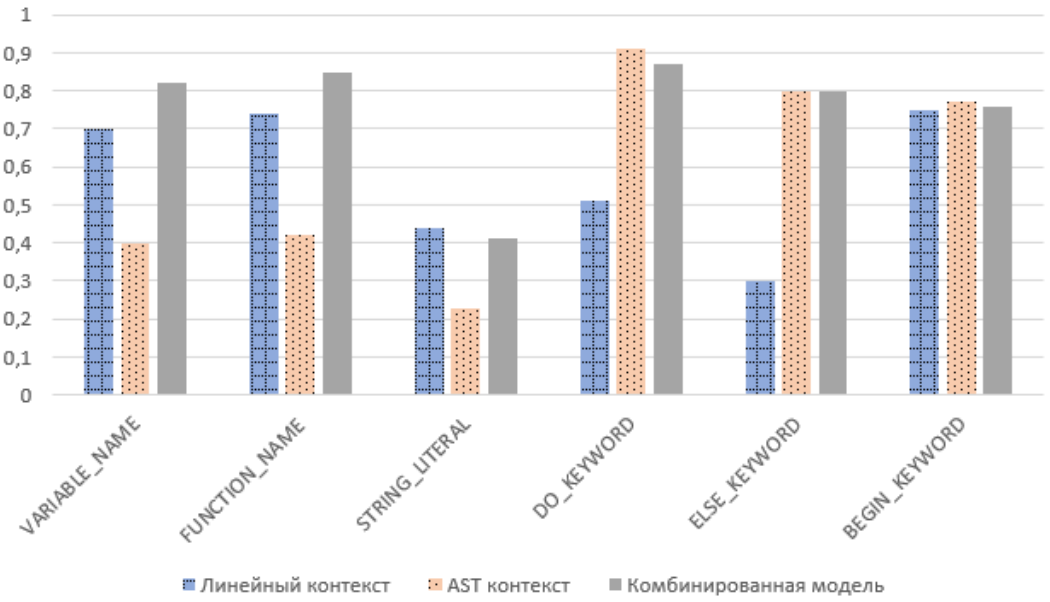

Рис. 9. Точность топ-1 моделей в разрезе типов токенов
Fig. 9. Top-1 model accuracy by token type

В результате тестирования моделей, предсказывающих типы токенов, были получены следующие значения метрик (табл. 1).

Таблица 1

Результаты работы моделей

Table 1

Results of the models

	Линейный контекст	AST-контекст	Комбинированная модель
Точность топ-1	0,49	0,23	0,57
Точность топ-2	0,64	0,37	0,79
Точность топ-5	0,98	0,72	0,98
Степень уверенности	0,84	0,79	0,91

Как и ожидалось, комбинированная модель показывает наилучший результат. Стоит отметить, что в таблице приведена точность предсказания только типа токена. Полученные типы токенов необходимо использовать в качестве дополнительного параметра для повышения точности алгоритма предсказаний самих значений токенов. Рассмотрим влияние работы модели на префиксный алгоритм предсказаний (табл. 2).

Таблица 2

Результаты работы алгоритмов предсказания завершения кода

Table 2

Results of the code completion algorithms

	Префиксный алгоритм	Префиксный алгоритм, с использованием комбинированной модели
Точность топ-1	0,09	0,28
Точность топ-2	0,17	0,41
Точность топ-5	0,43	0,64

Обычный префиксный алгоритм показывает довольно низкую точность топ-5 предсказания самих токенов, равную 0,43. Если отсортировать результаты работы префиксного алгоритма, используя результаты работы комбинированной модели, можно заметить ощутимый прирост точности до 0,64. Данная точность является приемлемой для использования в разрабатываемой IDE для Pascal, поэтому было решено использовать данный алгоритм. Однако предполагается, что, благодаря высокой производительности модели, результаты ее работы могут быть интегрированы в уже существующие более сложные алгоритмы.

Заключение

В статье представлен анализ существующих подходов к решению задачи автоматического завершения исходного кода программы. Разработана архитектура модели, комбинирующая цепи Маркова, обученные на различных контекстах: линейном и с использованием AST. В кратком изложении представлен анализ устойчивости модели к различным пользовательским стилям написания кода за счет включения в модель цепей, обучаемых на текущем пользовательском проекте. В качестве набора данных были собраны более 20000 различных программ на языке Pascal. Представлен сравнительных анализ работы нескольких реализаций моделей. Показано, что комбинированная модель обладает более высокой точностью по сравнению с остальными

моделями. На примере префиксного алгоритма продемонстрировано, что модель позволяет повысить точность предсказаний.

Данное решения является важной частью в разрабатываемой IDE для Pascal и позволит пользователям эффективнее работать со средой программирования за счет более точных подсказок с вариантами завершения кода.

Список литературы / References

1. **Marasoiu M., Church L., Blackwell A.** An empirical investigation of code completion usage by professional software developers. Annual Workshop of the Psychology of Programming Interest Group, 2015.
2. **Han S., Wallace D. R., Miller R. C.** Code Completion from Abbreviated Input. 2009 *IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2009, pp. 332–343. DOI: 10.1109/ase.2009.64
3. **Hou D., Pletcher D. M.** An evaluation of the strategies of sorting, filtering, and grouping API methods for Code Completion. *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, Sep. 2011, pp. 233–242. DOI: 10.1109/icsm.2011.6080790
4. **Ginzberg A., Kostas L., Balakrishnan T.** Automatic code completion. Stanford, Class Project, 2017.
5. **Svyatkovskiy A., Zhao Y., Fu S., Sundaresan N.** Pythia: AI-assisted Code Completion System. *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, Jul. 2019, pp. 2727–2735. DOI: 10.1145/3292500.3330699
6. **Buksbaum D.** Increasing Code Completion Accuracy in Pythia Models for Non-Standard Python Libraries. Doctoral dissertation. Nova Southeastern University, 2023, https://nsuworks.nova.edu/gscis_etd/1188
7. **Eysholdt M., Behrens H.** Xtext: implement your language faster than the quick and dirty way. *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, Oct. 2010, pp. 307–309. DOI: 10.1145/1869542.1869625
8. **Hellendoorn V. J., Devanbu P.** Are deep neural networks the best choice for modeling source code? *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, Aug. 2017, pp. 763–773. DOI: 10.1145/3106237.3106290
9. **Chan K. C., Lenard C. T., Mills T. M.** An introduction to Markov chains. *49th Annual Conference of Mathematical Association of Victoria*, 2012, pp. 40–47. DOI: 10.13140/2.1.1833.8248
10. **Cormen T. H., Leiserson C. E., Rivest R. L., Stein C.** Introduction to Algorithms (2nd ed.). MIT Press and McGraw-Hill, 2001, pp. 214–217.
11. **Wu B., Liang B., Zhang X.** Turn tree into graph: Automatic code review via simplified AST driven graph convolutional network. *Knowledge-Based Systems*, 2022, pp. 109450. DOI: 10.1016/j.knosys.2022.109450

Сведения об авторах

Тимофеев Владислав Сергеевич, магистрант

Information about the Author

Vladislav S. Timofeev, Master's Student

Статья поступила в редакцию 22.04.2024;
одобрена после рецензирования 24.06.2024; принята к публикации 24.06.2024

The article was submitted 22.04.2024;
approved after reviewing 24.06.2024; accepted for publication 24.06.2024