УДК 004.423.4
DOI 10.25205/1818-7900-2020-18-2-62-75

Разработка программных средств для улучшения работы механизма автодополнения кода с использованием алгоритмов машинного обучения в интегрированной среде разработки для языка Python

А. О. Матвеев [1], **А. В. Быстров** [2], **В. И. Бибаев** [3], **Н. И. Поваров** [3]

[1] *Новосибирский государственный университет*
Новосибирск, Россия

[2] *Институт систем информатики им. А. П. Ершова СО РАН*
Новосибирск, Россия

[3] *ООО «Интеллиджей Лабс»*
Санкт-Петербург, Россия

Аннотация

Автоматическое дополнение кода является важной функцией интегрированной среды разработки. Оно позволяет пользователям упростить набор длинных выражений в процессе программирования. Методы автоматического дополнения кода, как рассматриваемые в научных работах, так и реализованные в коммерческих продуктах, варьируются от применения эвристик для конкретных случаев до машинного обучения. При этом такие методы обычно опираются на статистические данные и не учитывают поведение пользователей. В статье предлагается подход к улучшению механизма автоматического дополнения кода для языка Python на основе сбора данных о работе этого механизма у реальных пользователей. Эти данные используются для обучения модели с целью последующего ранжирования вариантов автодополнения с помощью алгоритмов машинного обучения. Для обучения модели используются два типа признаков: контекстные и элементные. Контекстные признаки описывают информацию о коде рядом с позицией курсора в текстовом редакторе. Элементные признаки описывают характеристики предлагаемого варианта дополнения кода, например длину совпадающего префикса или тип варианта. Отмечается зависимость модели от ограничений на время ее срабатывания и размер. В работе также рассматриваются различные подходы к оценке качества полученной модели.

Ключевые слова

Python, PyCharm, автоматическое дополнение кода, машинное обучение, интегрированная среда разработки

Для цитирования

Матвеев А. О., *Быстров А. В.*, *Бибаев В. И.*, *Поваров Н. И.* Разработка программных средств для улучшения работы механизма автодополнения кода с использованием алгоритмов машинного обучения в интегрированной среде разработки для языка Python // Вестник НГУ. Серия: Информационные технологии. 2020. Т. 18, № 2. С. 62–75. DOI 10.25205/1818-7900-2020-18-2-62-75

© А. О. Матвеев, А. В. Быстров, В. И. Бибаев, Н. И. Поваров, 2020

Development of Software Tools to Improve the Work of the Code Completion Mechanism Using Machine Learning Algorithms in an Integrated Development Environment for Python

A. O. Matveev [1], **A. V. Bystrov** [2], **V. I. Bibaev** [3], **N. I. Povarov** [3]

[1] *Novosibirsk State University*
Novosibirsk, Russian Federation

[2] *A. P. Ershov Institute of Informatics Systems SB RAS*
Novosibirsk, Russian Federation

[3] *Intellij Labs LLC*
St. Petersburg, Russian Federation

Annotation

Auto-completion is an essential feature of any popular text editor for some language. It allows users to avoid the process of annoying typing of long expressions in their projects. There are a lot of different works in this direction in scientific research and commercial products. These works are very different and either use some special features and heuristics to improve code completion or use machine learning techniques. Most of these approaches rely on synthetic data and do not take into account the behavior of real users. The article proposes an approach to improve the automatic code completion mechanism for the Python language by collecting information about usage of this mechanism by real users. The obtained data is used to train the model to rank completion variants with machine learning algorithms. To train the model, two types of features are used: contextual and elemental. Contextual features describe information about the code next to the cursor position in a text editor. Elemental features describe the characteristics of the proposed variant, for example, the length of the matching prefix. When building a model, it is important to take into account the limits of the response time of the model and its size. Also, in the paper, various approaches of assessing the quality of the final model are considered.

Keywords

Python, PyCharm, code completion, machine learning, IDE

For citation

Matveev A. O., Bystrov A. V., Bibaev V. I., Povarov N. I. Development of Software Tools to Improve the Work of the Code Completion Mechanism Using Machine Learning Algorithms in an Integrated Development Environment for Python. *Vestnik NSU. Series: Information Technologies*, 2020, vol. 18, no. 2, p. 62–75. (in Russ.) DOI 10.25205/1818-7900-2020-18-2-62-75

Введение

Механизм автоматического дополнения кода в современных интегрированных средах разработки – один из наиболее часто используемых разработчиками, вплоть до нескольких раз в минуту [1]. Основная цель этого механизма – ускорить и упростить процесс написания программы разработчиком. Дополнительной его целью является минимизация количества опечаток и действий пользователя в целом. Автоматическое дополнение кода позволяет избежать утомительного набора длинных выражений (имен переменных, методов) в программах, исключая возможность опечатки. Кроме того, разработчики нередко не помнят точное имя метода или переменной, которую они хотят использовать. В этом случае механизм автоматического дополнения кода может предложить пользователю существующие варианты подходящего имени из списка. Еще одна полезная особенность заключается в том, что автоматическое дополнение кода побуждает разработчиков использовать более длинные и описательные имена методов и переменных, что приводит к более читаемому и понятному коду. Ввод длинных имен вручную может быть неудобен, но автоматическое дополнение кода ускоряет набор, позволяя не печатать все имя переменной или метода до конца, а выбирать из списка необходимый вариант после набора лишь части имени.

Обычно функция автоматического дополнения кода реализована в виде всплывающего окна (рис. 1), появляющегося, когда пользователь начинает вводить текст или же вызывает

эту функцию явно, с помощью комбинаций клавиш клавиатуры. Это всплывающее окно содержит возможные варианты дополнения для текущего положения в коде, например допустимые имена методов, переменных, ключевые слова, имена классов и др. Существуют реализации автоматического дополнения, в которых пользователю предлагается дополнить не одно, а сразу несколько слов до конца текущей строки или же название метода целиком. Выбранный пользователем вариант вставляется в текст. Таким образом, пользователь пишет код, не вводя весь текст.

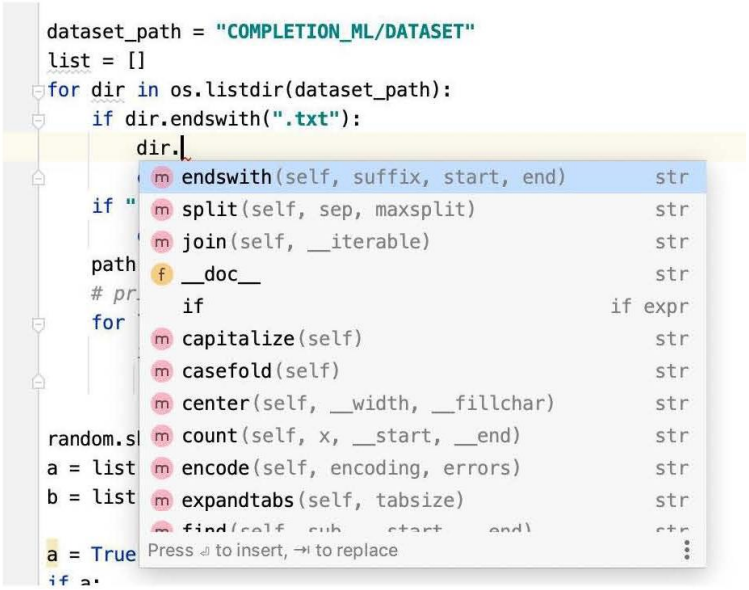

Рис. 1. Автоматическое дополнение кода в PyCharm
Fig. 1. Automatically adding code to PyCharm

Базовые системы автоматического дополнения кода весьма ограничены. В списке предлагаемых ими вариантов часто встречаются ненужные и мало используемые методы (включая методы, унаследованные от суперклассов, находящихся высоко в иерархии наследования). Это особенно часто происходит при работе с развитыми классами с большим количеством функциональных возможностей, которые можно использовать по-разному.

В настоящее время механизмы автоматического дополнения кода для интегрированных сред активно разрабатываются, появляются различные встраиваемые инструменты для популярных сред, такие как TabNine [1], IntelliCode [2], Codota [3], Kite [4] и др. В этих инструментах применяются различные подходы как к интерфейсу пользователя, так и к способам подбора вариантов. Некоторые предлагают варианты дополнения или дополняют существующий список, а есть и те, которые только сортируют уже существующие варианты. Вариантом может быть как одно слово, так и строка целиком или даже весь метод. Подходы могут отличаться и использующимися в них алгоритмами, начиная от простого набора эвристик для конкретных случаев до использования алгоритмов машинного обучения. В последнее время алгоритмы машинного обучения набирают популярность, так как они позволяют решать столь

[1] https://www.tabnine.com/.
[2] https://visualstudio.microsoft.com/ru/services/intellicode/.
[3] https://www.codota.com/.
[4] https://kite.com/.

непростую задачу в автоматическом режиме, используя готовые алгоритмы для оптимизации.

В данной статье предлагается улучшение автоматического дополнения кода для языка Python с использованием алгоритмов машинного обучения для сортировки вариантов, предлагаемых стандартными механизмами интегрированной среды для разработки PyCharm.

Академические работы в области автоматического дополнения кода

Ранние исследования в области автоматического дополнения кода были посвящены улучшению результатов работы механизма для языка Java на основе набора эвристик (Hou & Pletcher (2011) [2], Han et al. (2009) [3]). В качестве эвристик авторы использовали группировку вариантов, сортировку, основанную на иерархии типов и популярности предлагаемых вариантов, фильтрацию предлагаемых вариантов, например приватных методов.

Некоторые исследователи строили статистические модели для ранжирования и предложения вариантов автоматического дополнения кода на основе открытых источников данных. Так, Bruch и соавторы (2009) [4] предлагали искать подходящие варианты на основе существующей кодовой базы, а Robbes & Lanza (2008) [5] – на истории изменений в системах контроля версий.

Существуют также работы, базирующиеся на различных языковых моделях. Raychev и соавторы (2014) [6] используют статистические языковые модели, применяемые при обработке естественного языка, чтобы предлагать следующее слово по текстовой последовательности. Статистические модели широко используются в различных задачах обработки естественного языка. Одной из самых простых таких моделей является n-граммная модель [7], где частоты последовательных n токенов используются для прогнозирования вероятности появления последовательности токенов. Частоты таких n-грамм получены из корпуса текста и сглаживаются различными алгоритмами, такими как сглаживание Кнезера – Нея [8], чтобы учесть разреженность n-грамм. Эти модели могут быть применены и в алгоритмах автоматического дополнения кода, где каждый токен считается словом и выполняется n-граммный анализ на основе программного кода. Однако такие модели не очень хорошо фиксируют большую долю неизвестных токенов из имен переменных и функций [6].

Недавние исследования в этой области посвящены использованию моделей глубокого обучения для предложения и ранжирования вариантов автоматического дополнения кода, например Li и соавторы (2018) [9].

Коммерческие продукты

Помимо академических работ, посвященных задаче автоматического дополнения кода, существуют и коммерческие программные инструменты, предлагающие решение этой задачи. К таким инструментам, которые обычно могут быть установлены в различные интегрированные среды разработки, относятся, например, TabNine, IntelliCode, Codota, Kite. Приведем их краткое описание.

Codota – инструмент, использующий эвристики для автоматического дополнения кода всей строки. При этом в предложенных вариантах находятся в том числе и вызовы конструкторов с параметрами – локальными переменными из текущего метода, если они подходят по типу или по имени. Последовательность вызовов конструкторов или методов определяется по контексту. При этом учитывается, какие вызовы были сделаны до этого. Для языка Java имеется возможность получить информацию по использующимся методам – посмотреть похожие цепочки вызовов на github и stackoverflow.

Kite – встраиваемый в интегрированные среды разработки механизм улучшения автоматического дополнения кода для языка Python. Как и codota, учитывает предыдущие вызовы методов из контекста и предлагает подходящий вариант, основываясь на похожих примерах

из большого количества исходных кодов. Имеет функцию подстановки в аргументы функции локальных переменных (сопоставляет подходящие по имени переменные).

TabNine – инструмент для автоматического дополнения кода, работающий на основе обученной модели GPT-2 [5] (нейронная сеть, предназначенная для решения задач машинного обучения в направлении естественных языков). Модель была обучена на большом объеме исходного кода для разных языков и умеет предлагать некоторые нетривиальные варианты, используя все преимущества GPT-2. Модель позволяет по контексту определить, что нужно использовать антоним к некоторому слову, использованному ранее в контексте (например, yes – no). К недостаткам этого инструмента можно отнести большой разброс в качестве предложенных вариантов. Есть случаи, когда результаты работы весьма впечатляющи, но есть такие, в которых предложенные варианты очень далеки от того, что требуется в контексте. При этом частое появление неподходящих результатов может создавать серьезный дискомфорт для пользователя. Модель также получается слишком громоздкой и ресурсоемкой. Чтобы работать локально на устройстве пользователя, для ее работы необходимо иметь доступ к вычислительному серверу, что требует от компаний, приобретающих этот инструмент, разворачивания дополнительных систем и поддержки их в рабочем состоянии.

Описание работы механизма автоматического дополнения кода в среде PyCharm

В работе механизма автоматического дополнения кода можно выделить два этапа. Сначала в окружающих файлах и стандарте языка необходимо найти подходящие элементы и добавить в список в качестве вариантов дополнения. Затем этот список необходимо отсортировать таким образом, чтобы максимально облегчить пользователю выбор нужного варианта.

Для добавления элементов используется абстракция completion contributor [6]. Наивной реализацией данной абстракции является добавление всех элементов из текущего файла или каталога, которые соответствуют введенному пользователем префиксу. Однако для получения более качественного списка вариантов применяется множество различных completion contributor, использующих более сложные правила, основанные на анализе абстрактного синтаксического дерева, информации о типах и других факторах.

Каждый completion contributor отвечает за добавление определенного набора элементов в список. Есть отдельный completion contributor для ключевых слов, имен методов, локальных переменных. Completion contributor начинает свою работу, как только произойдет вызов автоматического дополнения кода, инициированный пользователем или автоматически. Эргономика интерфейса пользователя требует, чтобы список вариантов появился на экране не позднее, чем через 300 миллисекунд после того, как автоматическое дополнение кода инициировано. При этом может получиться так, что часть contributor-ов еще не успели закончить работу. Тогда первые 5 элементов списка, полученного к этому моменту, фиксируются и показываются пользователю сразу, а новые элементы затем добавляются в фоновом режиме в конец списка. Это необходимо для того, чтобы не получилось, что пользователь выбрал некоторый вариант, а в результате подставился другой, только что добавленный в начало списка. Таким образом, после первого показа пользователю сортировка не должна затрагивать первые 5 вариантов, а должна упорядочивать только последующие. Если же все completion contributor-ы успели отработать до первого показа пользователю вариантов, то можно сортировать весь список.

Простейшее решение для сортировки выбранных вариантов – это сортировка по алфавиту, и иногда оно работает хорошо, но когда вариантов много, пользователю приходится перемещаться по длинному списку упорядоченных по алфавиту элементов или вводить остав-

[5] https://openai.com/blog/gpt-2-1-5b-release/
[6] https://www.jetbrains.org/intellij/sdk/docs/tutorials/custom_language_support/completion_contributor.html

шуюся часть слова вручную. Поэтому нужно сортировать элементы таким образом, чтобы желаемый элемент оказался в первой позиции или как можно выше в списке. Для подбора и сортировки вариантов интегрированные среды разработки обычно используют набор эвристик. Однако эвристики слишком сложны в обслуживании и не всегда работают достаточно хорошо. Помимо этого, они плохо масштабируемы – добавление новых правил может влиять на работу существующих и не учитывать скрытые закономерности.

В среде PyCharm сортировка элементов осуществляется стандартным сортировщиком, работа которого основана на использовании упорядоченного набора так называемых completion weigher-ов. Completion weigher [7] – это абстракция, задача которой – задать вес каждому элементу. Сначала элементы разбиваются на классы эквивалентности в соответствии с весами первого weigher, и эти классы эквивалентности сортируются в соответствии с определяющими их весами. После этого внутри каждого класса эквивалентности независимо происходит подобная сортировка элементов при помощи второго weigher и т. д. Результат такой сортировки напрямую зависит от порядка weigher-ов. При этом может оказаться, что некоторые элементы необходимо будет поднимать вверх по списку при помощи других сортировщиков. Это приводит к тому, что система теряет гибкость. Изменение же порядка weigher-ов может иметь неочевидные последствия. Исправив один сценарий, оно может сломать несколько других.

Таким образом, хотелось бы заменить стандартный сортировщик более гибким, улучшение которого не требовало бы глубокого понимания сложных взаимосвязей в программной системе.

В данной работе описан подход к улучшению сортировки в алгоритме автоматического дополнения кода с помощью алгоритмов машинного обучения. Использование таких алгоритмов позволит избежать необходимости в комбинировании эвристик вручную, оно будет происходить автоматически, необходимо лишь разработать ряд эвристик – признаков для обучения модели. Важным преимуществом данного подхода является то, что для улучшения механизма не нужно модифицировать существующий код, достаточно добавлять новые признаки и таким образом получать более качественные модели.

PSI-интерфейс структуры программы

Механизм автоматического дополнения кода в интегрированной среде разработки на платформе Intellij опирается на интерфейс структуры программы PSI. Этот интерфейс предоставляет доступ к представлению программного кода в виде абстрактного синтаксического дерева и позволяет производить операции с этим представлением, выделять составные части конструкций языка. PSI-структура представляет собой иерархию элементов, таких как psi-файл, psi-класс, psi-метод, psi-выражение и т. п. На рис. 2 показан PSI-файла для простого фрагмента кода.

С PSI-структурой взаимодействуют все основные компоненты платформы, связанные с работой с кодом, в том числе автоматическое дополнение. Реализации completion contributor-ов используют PSI-структуру для того, чтобы предложить варианты автодополнения, которые будут синтаксически корректными в данном контексте. С помощью PSI-структуры получается информация о местоположении каретки в момент вызова механизма и список элементов контекста вызова – текущей функции или класса. Некоторые варианты в списке дополнения могут иметь ссылки на psi-элементы, например элементы классов, переменных имеют отображение в psi-структуру. В то же время некоторые варианты автодополнения, например ключевые слова языка, не имеют отображения в psi-структуру.

[7] https://github.com/JetBrains/intellij-community/blob/837e67159a532cd18ec1c6418774deb8a3e24dca/platform/analysis-api/src/com/intellij/codeInsight/completion/CompletionWeigher.java

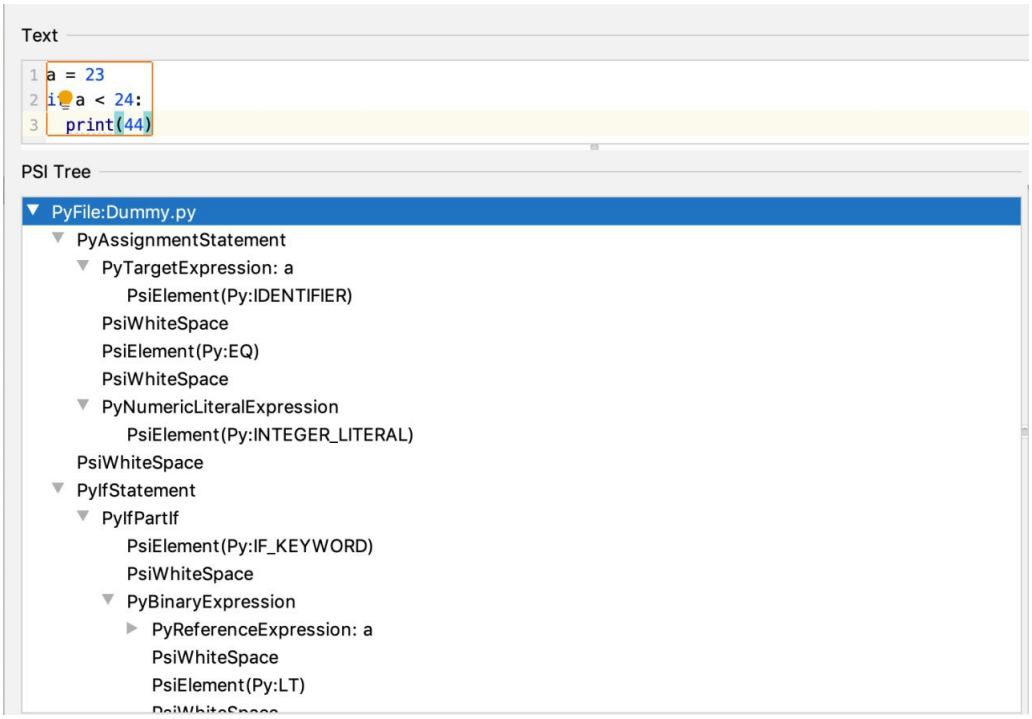

Рис. 2. Пример PSI-структуры для простого примера кода
Fig. 2. Sample PSI structure for a simple code example

Использование телеметрии

Большинство академических работ, посвященных улучшению автоматического дополнения кода, и коммерческих разработок, основаны на использовании статических данных – исходных кодов проектов, или искусственно построенных сценариях. Эти данные служат для построения некоторой модели и проверки ее качеств. С одной стороны, из таких данных действительно можно извлечь много полезной информации, с другой стороны, они не принимают во внимание поведение пользователя, не дают информации о том, где ему действительно необходима хорошая работа механизма автоматического дополнения.

Такую информацию можно получить, анализируя телеметрию – логи взаимодействия пользователя с механизмом автоматического дополнения. В них фиксируется, как именно в каждом конкретном случае был использован механизм:

- был ли он вызван явно, или же окно с вариантами появилось автоматически;
- какой префикс был введен до этого;
- отказался ли пользователь от предложенных вариантов, закрыв окно.

Понятно, что поведение пользователя в этом отношении может зависеть как от типа решаемой им задачи, так и от его индивидуальных особенностей. В идеале механизм автоматического дополнения кода должен адаптироваться к действиям и привычкам пользователя. Предлагаемый нами подход к улучшению механизма автодополнения с помощью моделей, учитывающих поведение пользователя, позволяет двигаться в этом направлении.

Полученная телеметрия может быть полезна еще и для оценки качества автодополнения. В академических работах для оценки качества чаще всего используются синтетические наборы тестов, известные специалистам. Такие наборы позволяют сравнивать работу разных систем и особенно полезны при написании обзорных статей. Но является ли такой подход полезным с точки зрения создания качественного продукта для пользователя – вопрос весьма

дискуссионный. Так, в статье Vincent J. Hellendoorn и соавторов [10] обсуждается, можно ли доверять искусственным наборам тестов и насколько сильно они могут отличаться от реальных сценариев использования механизма автодополнения.

Реализация подхода с использованием сбора логов для языка Python

Процесс поиска новых признаков для улучшения модели безграничен и зависит лишь от фантазии разработчиков. В работе предложен набор признаков, с которого можно начать оценивать работу подхода и качество модели. Выделяются две категории признаков: признаки позиции и признаки элементов. Признаки позиции описывают контекст рядом с кареткой. Этот тип признаков зависит только от положения каретки и не использует информацию о предлагаемом варианте автоматического дополнения. Признаки элемента описывают некоторые характеристики предлагаемого варианта автоматического дополнения, например общий префикс с введенной пользователем частью имени или длина предлагаемого варианта.

В текущей версии механизма для языка Python реализованы следующие признаки позиции:

- находится ли каретка в условии (рис. 3, *а*);
- находится ли каретка в управляющей части цикла (рис. 3, *б*);
- находится ли каретка после оператора if без ветки else (рис. 3, *в*);
- находится ли каретка в контексте аргументов (рис. 3, *г*);
- предыдущие ключевые слова языка в той же строке (рис. 3, *д*);
- предыдущие ключевые слова в том же столбце (рис. 3, *е*).

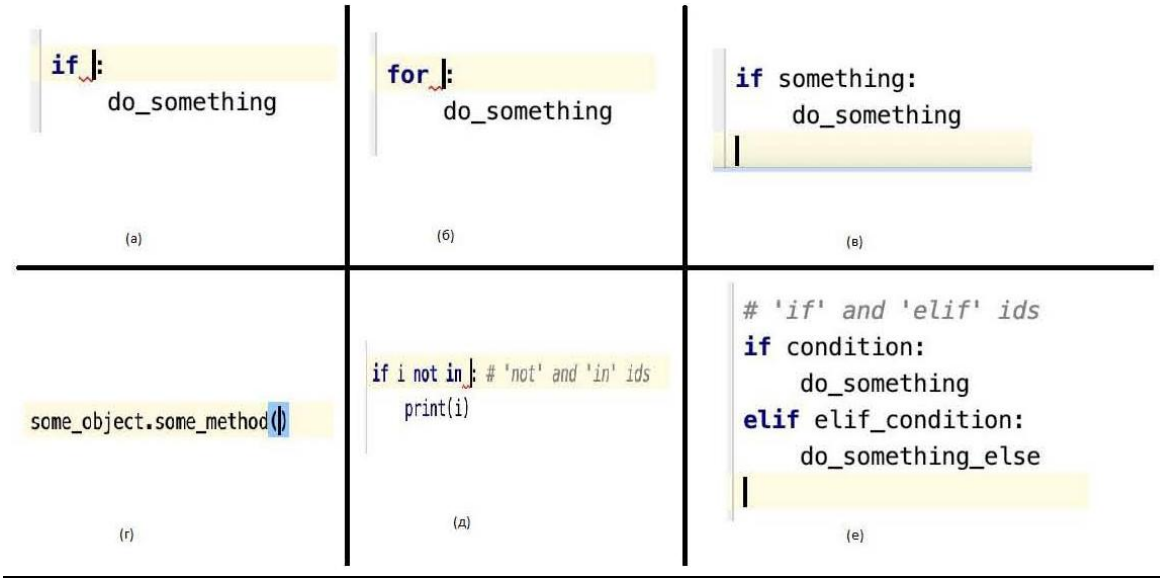

Рис. 3. Контекстные бинарные признаки

Fig. 3. Context binary signs

Для элементов сейчас реализованы следующие признаки.

- Популярность импортируемого модуля – частота его использования в кодах репозитория GitHub. Популярность основных модулей (numpy, tensorflow и т. п.) была подсчитана как количество результатов поискового запроса в GitHub по имени модуля. Значение этого признака со временем может стать неактуальным, поскольку популярности модулей меняются.

Необходимо будет обновлять информацию о популярности и обучать модель заново. Устаревания признака можно избежать, корректируя его с учетом частоты использования модуля реальными пользователями, отраженной в телеметрии.

- Популярность встроенных функций – признак, аналогичный предыдущему, но для встроенных функций Python, таких как max, min, map и др.
- Идентификатор ключевого слова: если вариант автоматического дополнения – это ключевое слово python, то значением функции будет некоторый идентификатор, назначенный этому ключевому слову (ключевые слова могут быть каким-то образом нумерованы).
- Количество элементов с таким же именем в некотором контексте, например внутри метода.
- Тип предложенного варианта, например, что он является ключевым словом, именем метода, класса или локальной переменной или других вариантов.

Некоторые признаки могут быть использованы не только для языка Python, но и для других языков, например, номер строки, номер столбца, длина совпадающего префикса, длина предлагаемого варианта, сколько раз данный вариант встречается в текущем файле и другие подобные признаки. Такие признаки должны базироваться на общих для различных языков свойствах и не должны зависеть от особенностей внутреннего представления конкретного языка. К ним также можно отнести значения, которые выдает *n*-граммная модель. Собирается информация о том, какие слова чаще всего встречаются после цепочки предыдущих слов в данном файле, и на основе этого оцениваются вероятности для элементов быть следующими в текущем контексте.

Сбор признаков и обучение модели

Значения признаков вычисляются обертками (wrappers), которые были реализованы для стандартных механизмов платформы. Эти обертки предоставляют интерфейсы для вычисления контекстных и элементных признаков для конкретных языков.

Для вычисления контекстных признаков был добавлен дополнительный completion contributor, задачей которого является не добавить новые элементы в список, а собрать вычисленные признаки со всех подключенных реализаций специальных интерфейсов для сбора таких признаков. Интерфейс предоставляет данные о позиции и контейнер для сохранения информации о контексте. Для одного языка реализаций этого интерфейса может быть несколько, и все они могут использоваться одновременно. Конкретная реализация, вычисляет и сохраняет признаки и, возможно, еще некоторую дополнительную информацию о контексте, используя информацию о позиции в PSI-структуре.

Для вычисления элементных признаков, таких как, например, длина совпадающего префикса, длина слова, добавлен специальный completion weigher. Он задает один и тот же вес каждому элементу и, таким образом, не влияет на сортировку. Помимо этого weigher-а, имеется так же интерфейс для сбора элементных признаков. Реализаций этого интерфейса может быть несколько, разделенных логически. Например, одна реализация может включать сбор признаков, основанных на имени, другая – на соответствии некоторому psi-элементенту. Помимо этого, имеется возможность использовать информацию о контексте, вычисленную при помощи реализаций сборщиков контекстных признаков. Это позволяет производить вычисление один раз, а затем использовать его результат для вычисления признаков элементов. Например, можно один раз собрать все имена, встречающиеся в функции, в которой производится автодополнение, и для каждого элемента вычислить признак – количество имен, совпавших с этим элементом. Задачей weigher является добавление признаков, собранных всеми реализациями данного интерфейса.

После того, как все признаки вычислены, они кодируются так, чтобы в них не содержалось никакой приватной информации о программе пользователя, и добавляются в логи. Признаки вычисляются в ходе сессии автоматического дополнения кода, т. е. когда пользователю

видно окно с вариантами. Если пользователь выберет некоторый элемент из списка или допечатает некоторый элемент из списка до конца вручную, то сессию можно будет использовать для обучения, построив для каждого набора вариантов элементов для дополнения, видных пользователю, целевой вектор. В этом векторе значение равно 1, если соответствующий элемент был выбран пользователем, и 0 в противном случае.

Используя такой целевой вектор и векторы признаков для каждого элемента, можно обучить модель выдавать для каждого элемента его вес – вероятность, что будет выбран именно этот элемент. Такая задача – обучение ранжированию – стандартная задача машинного обучения. Известны различные ее решения, но в нашем случае механизма автодополнения есть ограничения на размер и производительность модели: модель не должна быть слишком большой, поскольку все вычисления выполняются на клиентском персональном компьютере, и должна работать достаточно быстро, чтобы задержка появления всплывающего окна автоматического дополнения не превышала 300 миллисекунд. С учетом этих ограничений нами выбран алгоритм random forest regressor, который может быть легко адаптирован к этим требованиям. Чтобы избежать переобучения, модель обучается на наборе данных, полученных в течение одной недели, и проверяется на наборе данных, полученных в течение следующей недели.

Тестирование и результаты

Задача оценки качества механизма автоматического дополнения кода весьма нетривиальна. Можно выделить два подхода к ее решению: оценка качества на искусственных примерах и оценка качества на основе телеметрии работы механизма у реальных пользователей. Первый подход хорош своей универсальностью и возможностью сравнивать качество работы разных механизмов. Однако оценки, полученные с его помощью, не всегда адекватны, поскольку искусственные примеры могут существенно отличаться от сценариев работы реальных пользователей. У подхода на основе телеметрии такой проблемы нет, но он требует специальной поддержки со стороны системы.

Рассмотрим три способа проверки, насколько применение машинного обучения улучшает работу механизма автоматического дополнения кода:

- проверка во время обучения модели;
- проверка на искусственных примерах;
- A/B тестирование.

Проверка во время обучения модели

Пользователи были разбиты на две группы. На данных первой группы модель обучалась, а на данных второй проверялось ее качество. В качестве метрик, демонстрирующих качество модели, были выбраны метрики top-1 и top-5 – отношение количества сессий, в которых нужный элемент оказался соответственно на первом и пятом месте в списке, ко всему количеству сессий. Чем больше попаданий в эти метрики, тем выше качество. Результаты проверки представлены в табл. 1. Как можно видеть, с новым механизмом результаты улучшились для обеих метрик. Отметим, что для обучения использовались все варианты списков, видимые пользователю. Например, если он начинает вводить слово и окно с вариантами появляется только после того, как введен первый символ, это будет первый вариант списка, если после это пользователь вводит еще один символ и список вариантов фильтруется по префиксу, это уже другой вариант. Улучшение наблюдалось для всех вариантов.

Проверка на искусственных примерах

Эта проверка организована следующим образом. Выбирается проект, написанный на языке Python и запускается специальный плагин, задачей которого является открывать каждый файл с исходным кодом в проекте и для каждого слова в этом файле стирать его целиком,

вызывать автоматическое дополнение, затем скрывать его, фиксировать результат, затем печатать следующий символ стертого слова, снова фиксировать результат и т. д. Таким образом происходит имитация работы пользователя, который будто бы вызывает механизм автоматического дополнения кода для всех слов программы. Информация обо всех сессиях собирается и может быть использована как для обучения модели, так и для оценки качества работы механизма. В эксперименте для оценки использовались сессии с длинами префикса 0 и 1 (табл. 2).

Таблица 1

Сравнение топ-1 и топ-5, полученных после обучения модели

Table 1

Comparison of top-1 and top-5 metrics obtained after model training

Название метрики	Механизм работы	
	стандартный	новый
Top-1	0.547	0.735
Top-5	0.812	0.928

Таблица 2

Результат на искусственных сессиях

Table 2

Result in artificial sessions

Название метрики	Механизм работы	
	стандартный	новый
Для префикса 0		
Top-1	0.086	0.319
Top-5	0.189	0.586
Средняя позиция	97.96	36.67
Среднее время (мс)	46.9	65.6
Для префикса 1		
Top-1	0.66	0.72
Top-5	0.924	0.936
Средняя позиция	0.79	0.555
Среднее время (мс)	36.4	43.575

Результат проверки демонстрирует значительное увеличение попадания в top-1 и top-5, а также весьма существенное уменьшение средней позиции, под которой понимается усредненное по всем сессиям положение корректного варианта в списке вариантов автоматического дополнения. Чем меньше эта позиция, тем в среднем выше в списке находится необходимый элемент. Временные затраты на одну автоматическую сессию увеличились на 20 миллисекунд, что вполне допустимо. Можно заметить, что результат стандартного механизма в этой проверке гораздо хуже, чем при проверке в ходе обучения модели. Это объясняется тем, что в обучении участвуют все варианты списков, а в данном случае только варианты списков при нулевом введенном префиксе.

Таблица 2 демонстрирует улучшение работы механизма в отношении попадания в топ-1 и уменьшения средней позиции. Заметим, что и результат работы стандартного механизма на искусственных сессиях с префиксом 1 значительно лучше, чем для нулевого префикса, и даже лучше, чем усредненный результат по всем вариантам. В распределении значений при-

знака длины введенного префикса 46 % приходится на префикс 0 и 42 % – на префикс 1. Можно ожидать, что более длинные префиксы результат еще улучшат.

Проверка A/B-тестированием

A/B-тестирование – метод маркетингового исследования, направленного на выявление лучшего варианта некоторого функционала продукта. Суть метода заключается в том, что сначала выбирается некоторый элемент продукта и несколько вариантов реализации этого элемента. Создаются версии продукта с этими различными реализациями элемента и распределяются между пользователями. Данные об использовании продукта собираются, и по некоторым метрикам определяется, какая версия продукта наиболее удачна. На основе этого выбирается наилучшая по этим метрикам реализация элемента. В данной статье элемент – это алгоритм сортировки вариантов, предлагаемых во время автоматического дополнения кода, а реализациями являются стандартный механизм автоматического дополнения кода и механизм с применением машинного обучения.

A/B-тестирование точно так же, как и сбор признаков для обучения, происходит во время EAP (Early access product – версия продукта, которая предоставляется пользователям бесплатно с целью ознакомить их с новой функциональностью и отследить потенциальные проблемы, которые могут возникнуть у пользователей при добавлении этой функциональности). Половина пользователей со стандартным механизмом ранжирования вариантов автоматического дополнения кода попадают в группу A, другая половина – в группу B. На основе телеметрии, полученной от пользователей, можно сравнить две версии. В данном случае для сравнения качества используются такие метрики, как отношение долей сессий, завершившихся выбором варианта из списка, печатью варианта до конца без использования механизма автоматического дополнения (если вариант был в списке) и сессий, отмененных пользователем. Оцениваются также среднее время сессии, попадание нужного элемента в top-1 и top-5 и другие метрики.

В табл. 3 представлены результаты A/B-тестирования, полученные после включения модели. Можно заметить улучшение метрик top-1 и top-5 и средней позиции.

Таблица 3

Результаты A/B-тестирования

Table 3

Results of AB-testing

Название метрики	Механизм работы	
	стандартный	новый
Top-1	0.736	0.793
Top-5	0.943	0.967
Средняя позиция	2.8	1.85
Explicit select	0.245	0.262
Typed select	0.397	0.423
Длина сессии (мс)	1144	1057

В следующих версиях системы работа механизма будет улучшена за счет более глубокого учета контекста вызова механизма автоматического дополнения кода. Будет реализована возможность не просто оценивать качество системы в целом, а подробно изучить случаи, в которых автодополнение вызывается часто и в которых механизм требует доработки. Например, положение каретки внутри условий, циклов, в контексте аргументов функции. Для проблемных мест следует добавить новые признаки, которые улучшат работу модели.

Заключение

В статье предложен комплексный подход к решению задачи по улучшению работы механизма автоматического дополнения кода в интегрированной среде разработки PyCharm на основе машинного обучения. Описан способ сбора данных о работе механизма у реальных пользователей, построение модели с использованием алгоритмов машинного обучения и использующиеся в ней признаки. Предложенный подход реализован для языка Python. Проверка качества работы улучшенного механизма с использованием трех различных методик подтвердила обоснованность предложенного подхода. Новый механизм включен в промышленную версию среды PyCharm в качестве основного.

Гибкость реализованного механизма позволяет развивать его в различных направлениях: автоматизация сбора данных, построение моделей и оценка результатов, добавление новых признаков для обучения модели, использование более мощных библиотек для построения модели.

Список литературы / References

1. **Murphy G. C., Kersten M., Findlater L.** How are Java software developers using the Eclipse IDE? *Software, IEEE*, 2006, vol. 23 (4), p. 76–83.
2. **Hou D., Pletcher D.** Towards a Better Code Completion system by API grouping, filtering, and popularity-based ranking. In: RSSE'10: Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering, 2010, p. 26–30. DOI 10.1145/1808920.1808926
3. **Han S., Wallace D. R., Miller R. C.** Code completion from abbreviated input. In: IEEE / ACM International Conference on Automated Software Engineering, 2009, p. 332–343. DOI 10.1109/ASE.2009.64
4. **Marcel Bruch**, **Martin Monperrus, Mira Mezini.** Learning from Examples to Improve Code Completion Systems. In: Proceedings of the 7th joint meeting of the European Software Engineering Conference and the ACM Symposium on the Foundations of Software Engineering. Amsterdam, Netherlands, 2009, ff10.1145/1595696.1595728ff. ffhal-01575348
5. **Robbes R., Lanza M.** Improving code completion with program history. *Automated Software Engineering*, 2010, no. 17. DOI 10.1007/s10515-010-0064-x
6. **Raychev V., Vechev M., Yahav E.** Code Completion with Statistical Language Models. *ACM SIGPLAN Notices*, 2014, no. 49. DOI 10.1145/2594291.2594321
7. **Fink G**. n-Gram Models. *Automated Software Engineering*, 2014. DOI 10.1007/978-1-4471-6308-4_6
8. **Kneser R., Ney H.** Improved backing-off for m-gram language modeling. In: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing, May 1995, vol. 1.
9. **Li Jian, Wang Yue, Lyu M., King I.** Code Completion with Neural Attention and Pointer Networks. Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18), 2018, p. 4159–4165. DOI 10.24963/ijcai.2018/578
10. **Hellendoorn V., Proksch S., Gall H., Bacchelli A.** When Code Completion Fails: A Case Study on Real-World Completions. In: ICSE 2019, p. 960–970. DOI 10.1109/ICSE.2019.00101

Материал поступил в редколлегию
Received
21.04.2020

Сведения об авторах

Матвеев Андрей Олегович, магистрант факультета информационных технологий Новосибирского государственного университета (Новосибирск, Россия)
andrey.matveev@jetbrains.com

Быстров Александр Васильевич, кандидат физико-математических наук, доцент кафедры систем информатики ФИТ НГУ (Новосибирск, Россия)
avb@iis.nsk.su

Бибаев Виталий Игоревич, программист, ООО «Интеллиджей Лабе» (Санкт-Петербург, Россия)
vitaly.bibaev@jetbrains.com

Поваров Никита Игоревич, руководитель отдела Data Science, ООО «Интеллиджей Лабе» (Санкт-Петербург, Россия)
ninkita.povarov@jetbrains.com

Information about the Authors

Andrey O. Matveev, Master's Student, Faculty of Information Technologies, Novosibirsk State University (Novosibirsk, Russian Federation)
andrey.matveev@jetbrains.com

Alexander V. Bystrov, Candidate of Physical and Mathematical Sciences Faculty of Information Technologies, Novosibirsk State University (Novosibirsk, Russian Federation)
avb@iis.nsk.su

Vitaly I. Bibaev, programmer, Intellij Labs LLC (St. Petersburg, Russian Federation)
vitaly.bibaev@jetbrains.com

Nikita I. Povarov, Data Science department head, Intellij Labs LLC (St. Petersburg, Russian Federation)
ninkita.povarov@jetbrains.com