ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук

Сурков Петр Анатольевич

РАЗРАБОТКА АЛГОРИТМОВ АВТОМАТИЧЕСКОГО ВЫЯВЛЕНИЯ ОШИБОК В СГЕНЕРИРОВАННЫХ АВТОДОПОЛНЕНИЯХ КОДА

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
Ломшаков Вадим Михайлович

Руководитель
Кандидат физико-математических наук
Москвин Денис Николаевич

Санкт-Петербург 2023

Оглавление

Аннотация

Автодополнение кода является важной функцией современных сред разработки программ. Оно помогает программистам писать код эффективнее. Однако ошибки в автодополнениях кода негативно сказываются на компиляции программ и нарушают рабочий процесс программиста. Эта проблема особенно актуальна с растущим использованием технологий машинного обучения для автодополнений кода, предоставляющими не всегда корректные автодополнения. В данном исследовании были разработаны методы обнаружения ошибок в автодополнениях кода в среде разработки PyCharm, использующие инспекции и построение абстрактных синтаксических деревьев. Также была разработана простейшая классификация критичности находимых ошибок. Затем разработанные алгоритмы были интегрированы в инструмент машинного автодополнения Full Line Code Completion. Были собраны данные о производительности и количестве генерируемых ошибочных вариантов с пользователей. Разработанные методы оказались эффективными и производительными.

Ключевые слова: автодополнение кода, среда разработки программ, машинное обучение.

Code completion is an essential feature of modern integrated development environments that aims to assist programmers in writing code more efficiently. However, incorrect auto-completion negatively impact program compilation and disrupt the workflow of programmers. This problem is especially prevalent with the increasing use of machine learning technologies for code completion, which may not always provide accurate suggestions. In this study, we develop methods for verifying auto-completion in PyCharm integrated development environment, which use inspections and construction of abstract syntax trees for verification. Additionally, a basic classification of the criticality of detected errors was developed. Then, the developed algorithms were integrated into the Full Line Code Completion machine autocompletion tool. Data on performance and the number of erroneous suggestions generated were collected from users. The developed methods proved to be effective and efficient.

Keywords: code completion, integrated development environment, machine learning.

Введение

Среда разработки – это программное обеспечение, существенно упрощающее процесс разработки программ. Среда разработки значительно облегчает рабочую деятельность программистов и ускоряет написание кода, предоставляя множество инструментов для разработки, отладки и тестирования программного кода [4]. Автодополнение кода — это инструмент среды разработки, предлагающий автоматические варианты продолжения фрагментов кода. На рисунке (рис. 1) автодополнение предлагает дописать конец ключевого слова return, ориентируясь на уже введённое начало. Автодополнение является одним из ключевых инструментов среды разработки [12].

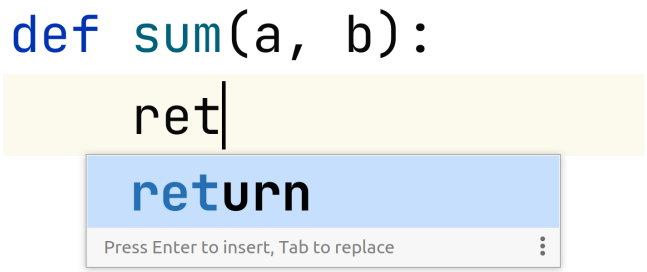

Рис. 1: Пример автодополнения в языке Python

В последние годы активно развиваются подходы с использованием машинного обучения в качестве поставщика автодополнений кода [2, 3, 5, 14]. И хотя такие подходы позволяют генерировать большие фрагменты автодополнений, зачастую они не являются полностью корректными. Нередка ситуация, что автодополнение на основе машинного обучения предлагает вызов функции, однако этой функции нет ни в проекте пользователя, ни в подключенных им библиотеках. Модель машинного обучения просто придумывает имя функции, которая нигде не существует. Затем программист выбирает такое автодополнение, оно вставляется в код и образует ошибку компиляции проекта. Это крайне мешает программистам [2]. Одна из важнейших функций автодополнений — это проверка существования той или иной функции, которую программист намеревается написать [8]. Однако из-за ложных автодополнений у программиста создаётся впечатление, что отсутствующая функция существует и он с уверенностью выбирает её. Таким образом, неверные автодополнения кода обманывают ожидания и привычное поведение программистов.

Стоит понимать, что проблема некорректных автодополнений кода не решится сама собой с развитием моделей машинного обучения. Дело в том, что модели не могут обработать все исходные файлы проекта пользователя, то есть они просто не знают, что происходит в большей части проекта, которую они не видят. Даже самые мощные существующие на сегодняшний день модели, такие как GPT-4, могут обрабатывать порядка 50 страниц текста [10]. И этого совершенно недостаточно, чтобы вместить весь проект пользователя, который может состоять из миллионов строк

кода. Кроме того, даже если представить возможность создания настолько мощных моделей машинного обучения, которые, ко всему прочему, должны быть достаточно быстрые и не слишком дорогие для использования, модель всё равно, вероятно, не сможет идеально выучить всё содержимое проекта: понять все функции, переменные, квалификаторы доступа, наследования, типы и остальные конструкции, чтобы не совершать никаких ошибок при генерации автодополнений кода. Поэтому всегда будет необходимо выполнять пост-обработку сгенерированных автодополнений: проверять их на предмет наличия различных ошибок.

Данная работа уже сейчас активно используется в инструменте под названием Full Line Code Completion [16]. Это плагин среды разработки PyCharm, генерирующий автодополнения на основе машинного обучения для языка программирования Python. Специфика этого плагина заключается в том, что для генерации используются маленькие модели машинного обучения, запускаемые локально на компьютерах пользователей. Поэтому данный плагин специализируется на генерации достаточно коротких автодополнений кода — порядка одной строчки кода. Эта специфика будет играть роль в данной работе с точки зрения необходимой производительности разрабатываемого решения. Кроме того, инструмент Full Line Code Completion нуждался в обнаружении ошибок, так как пользователи регулярно жаловались на генерацию ошибочных автодополнений кода. Существовало 8 публичных [15] и 10 внутренних жалоб.

При этом сами среды разработки, такие как PyCharm, обладают богатым функционалом, который может быть полезен для нахождения ошибок в автодополнениях. Этот функционал можно использовать, разрабатывая плагины к PyCharm [6]. Кроме того, желательно, чтобы поиск ошибок в автодополнениях обнаруживал тот же набор ошибок в автодополнениях, что находит сам PyCharm в коде программ. Этого можно добиться, используя и адаптируя подходы PyCharm к поиску ошибок. В противном случае будет присутствовать несогласованность между ошибками, обнаруживаемые в автодополнениях, и средой разработки PyCharm после вставки этих автодополнений в код.

Стоит понимать, что автодополнения кода должны генерироваться достаточно быстро, чтобы пользователи использовали автодополнения, а не печатали код самостоятельно. Инструмент Full Line Code Completion генерирует автодополнения порядка 200 миллисекунд, так что в данной работе хотелось разработать методы с временем работы не более 40 миллисекунд. Это позволит занимать не более пятой части времени на обнаружение ошибок от всего времени генерации автодополнений кода. В таком случае, о производительности разработанных решений можно будет не беспокоиться, поскольку увеличение времени работы автодополнения кода на пятую часть не является существенно заметным для пользователей.

Наконец, не все ошибки воспринимаются одинаково негативно пользователями.

Одни ошибки в автодополнениях гарантированно являются плохими и никак не могут быть исправлены программистом, кроме как переписыванием автодополнения. Другие же ошибки совершенно незначительны и не являются существенным аргументом для скрытия автодополнения из выдачи пользователю. Поэтому в данной работе хотелось также разработать классификацию обнаруживаемых ошибок, позволяющую отличать практически гарантированно плохие ошибки от менее критичных. Это позволит инструментам, генерирующим автодополнения, таким как Full Line Code Completion, по-разному относиться к автодополнениям с разными ошибками. Например, автодополнения с более критичными ошибками скрывать от пользователя, а менее критичные — игнорировать.

Цель и задачи

Таким образом, цель данной работы заключается в обнаружении ошибок в автодополнениях кода на языке Python средствами среды разработки PyCharm.

Для этого необходимо решить следующие задачи:

- Разработать методы поиска ошибок в автодополнениях на основе PyCharm.
- Классифицировать критичность находимых ошибок.
- Интегрировать разработанные методы в инструмент Full Line Code Completion. Добиться закрытия жалоб пользователей и производительности в не более 40 миллисекунд на обнаружение ошибок.

Таким образом, общая схема решения выглядит следующим образом (рис. 2). Сначала генератор автодополнений генерируют набор автодополнения кода. В них ищутся ошибки и определяется критичность средствами PyCharm, что является основной темой данной работы. Затем, с найденными ошибками работает сам инструмент, предлагающий автодополнения. Он каким-либо образом обрабатывает найденные ошибки, например, скрывает автодополнения с критичными ошибками. В итоге пользователю представляется более лучший набор автодополнений, чем был изначально сгенерирован.

Определение ключевых терминов

Кареткой в среде разработки PyCharm принято называть положение курсора в окне редактирования файла. Многие инструменты среды разработки привязаны к положению каретки. Например, задача автодополнения заключается в продолжении текста после каретки, анализируя текст до неё.

Абстрактное синтаксическое дерево — это структура данных, используемая в компиляторах, интерпретаторах и средах разработки для представления синтаксической

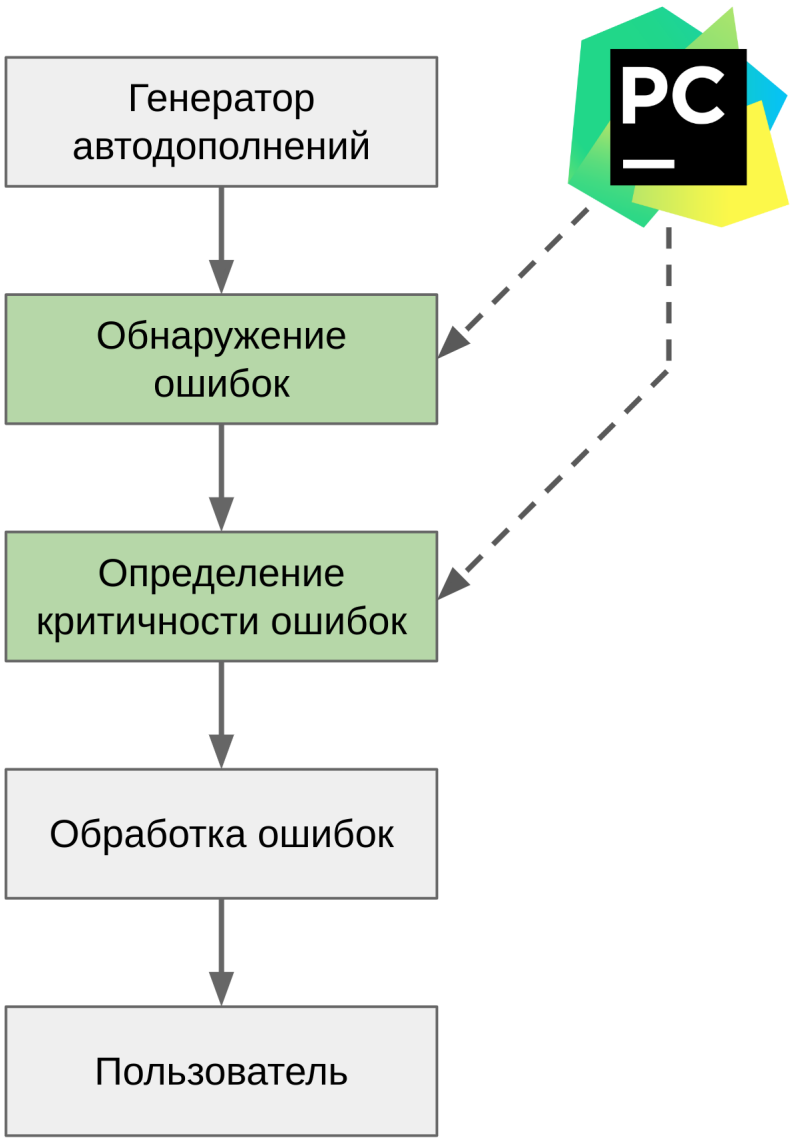

Рис. 2: Схема решения

структуры программы. Оно представляет собой дерево, в котором каждый узел соответствует конструкции языка программирования (например, оператору или выражению), а листья соответствуют элементам, таким как идентификаторы, литералы и так далее [11].

Парсер языка программирования — инструмент, выполняющий построение абстрактного синтаксического дерева по файлу исходного кода программы [11].

Инспекции среды разработки PyCharm — это функциональность среды разработки PyCharm, которая позволяет автоматически проверять код на наличие потенциальных ошибок [6]. Инспекции выполняют статический анализ кода, то есть анализируют код без его выполнения и помогают выявлять ошибки еще на стадии написания кода. Инспекции выполняются автоматически в фоновом режиме, но также могут быть запущены вручную.

Структура работы

В главе 1 представлен обзор существующих решений в области обнаружения ошибок в автодополнениях кода.

В главе 2 представлены алгоритмы обнаружения ошибок в автодополнениях с помощью PyCharm, описаны методы тестирования и представлены оценки производительности разработанных алгоритмов.

В главе 3 представлена классификация критичности находимых видов ошибок и описан алгоритм классификации с технической точки зрения.

В главе 4 описан процесс интеграции в плагин Full Line Code Completion и анализ практического применения разработанных алгоритмов: измерение количества находимых ошибок и производительности на практике.

1. Обзор литературы

1.1. Обнаружение ошибок инструментом IntelliCode Compose

Одним из распространённых инструментов машинного автодополнения является инструмент под названием “IntelliCode Compose”. В статье, посвещённой обзору этой технологии, авторы упоминают, что их инструмент гарантирует генерацию только синтаксически корректных автодополнений кода [5]. Синтаксические ошибки, которые данный инструмент умеет обнаруживать, это определённый класс ошибок, представляющий собой, например, ошибки с непарным использованием скобок в коде программы или, например, с неправильной расстановкой ключевых слов. Появление этих ошибок не зависит от других файлов проекта пользователя и подключенных им модулей, так как такие ошибки определяются синтаксической корректностью одного конкретного файла. Поэтому данный класс ошибок не включает в себя, например, ошибку генерации отсутствующего имени функции в библиотеке. Тем не менее он всё равно заслуживает внимания. Как отмечают авторы статьи, в 7% случаев генерируемые IntelliCode Compose автодополнения не являются синтаксически корректными и данная проверка позволяет обнаружить их и скрыть от пользователя.

Рассмотрим подход, который используется разработчиками IntelliCode Compose для обнаружения синтаксически некорректных автодополнений. Для проведения анализа производится вставка автодополнения в код на место каретки и запускается парсинг получившегося файла исходного кода. Если парсинг завершился успешно, без каких-либо ошибок в области вставленного автодополнения, то автодополнение признаётся синтаксически корректным. Однако с таким подходом существует некоторая ключевая трудность, которая будет подробно разобрана в основной части данной работы. Стоит понимать, что для преодоления этой трудности IntelliCode Compose используют свою собственную модификацию парсера [5].

Хотя данное решение и позволяет находить синтаксические ошибки, его главный недостаток заключается в необходимости модификации парсера. Во-первых, это требует изучение спецификации конкретного парсера для его корректной модификации, что осложняет разработку и перенос решения на другие языки программирования. Во-вторых, такие среды разработки как PyCharm предоставляют свой собственный парсер, на основе которого работает вся среда разработки в целом. Этот парсер не представляется возможным модифицировать. Наконец, инструмент IntelliCode Compose публикуется с закрытым исходным кодом. В результате в данной работе было принято решение разработать свой собственный метод анализа синтаксической корректности автодополнений, работающий поверх стандартного парсера PyCharm, а не переопределяющий его.

1.2. Обнаружение ошибок в автодополнениях у Google Research

В статье от Google Research [9] исследователи рассказывают о нахождении следующих видов ошибок в автодополнениях кода:

- Использование отсутствующих символов в коде (например, когда автодополнение генерирует обращение к несуществующим переменным, вызовы отсутствующих функций или обращение к несуществующим классам).
- Генерация неверного количества аргументов при генерации вызова функций. Например, когда автодополнение предлагает написать вызов функции с меньшим или большим числом аргументов, чем требует функция.
- Неверное использование типов в коде. Например, неверный тип передаваемого выражения при вызове функции или присваивании.

Для обнаружения данных ошибок исследователи Google Research используют языковой сервер. Языковой сервер — это набор инструментов, позволяющих искать обозначенные выше ошибки в файле кода программы. Автодополнения вставляются в исходный файл программы и анализируются с использованием языкового сервера. При отсутствии обозначенных выше ошибок автодополнения признаются корректными и отображаются пользователю. Исследователи отмечают, что после скрытия ошибочных автодополнений частота выбора вариантов, генерируемых их инструментом автодополнения, увеличилась почти вдвое. Это говорит о значительном улучшении качества автодополнения. В случае со средой разработки PyCharm нет необходимости использовать сторонний языковой сервер, так как данная среда разработки сама по себе обладает встроенными аналогичными инструментами обнаружения ошибок в коде.

Хочется отметить существенное упущение данной работы — отсутствие классификации найденных ошибок по уровням критичности. Все находимые ошибки исследователи в представленной работе просто скрывают от пользователей. Тем не менее несложно представить ситуацию, когда программист сначала пишет вызов функции и только потом декларирует её. В этом случае, хотя языковой сервер и проанализирует автодополнение вызова отсутствующей функции как ошибочное, оно является не ошибочным, а желательным для пользователя, но будет скрыто от него.

1.3. Выводы

По результатам анализа работ в области обнаружения ошибок в автодополнениях кода были сделаны следующие выводы:

- Использование парсера языка программирования позволяет находить синтаксические ошибки в автодополнениях. Конкретный реализованный алгоритм в

IntelliCode Compose не является публичным и требует модификации парсера. Поэтому в данной работе будет разработан альтернативный алгоритм поиска синтаксических ошибок.

- Использование языкового сервера позволяет находить другие, более сложные классы ошибок, данный подход показал свою эффективность в отмеченном исследовании. Аналогичные функции языкового сервера в PyCharm выполняют инспекции кода.
- Классификация найденных ошибок в автодополнениях по степени критичности отсутствует в представленных исследованиях, что является существенным недостатком работ. Одна из задач данной работы — разработать и реализовать эту саму классификацию.

2. Обнаружение ошибок в автодополнениях с помощью PyCharm

2.1. Обнаружение ошибок парсером PyCharm

PyCharm обладает функционалом обнаружения синтаксических ошибок в файлах исходного кода. На рисунке (рис. 2.1) представлен фрагмент кода на языке Python, содержащий синтаксическую ошибку. В коде присутствует закрывающая скобка, в то время как парная ей предшествующая открывающаяся скобка отсутствует. PyCharm обнаруживает данную ошибку и выделяет её красной волнистой линией. Поэтому, используя парсер PyCharm возможно находить синтаксические ошибки в автодополнениях кода.

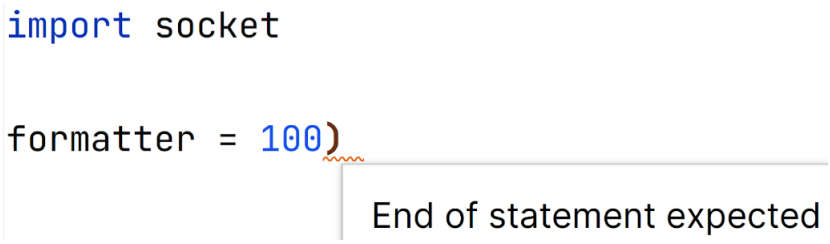

Рис. 2.1: Обнаружение синтаксической ошибки парсером PyCharm

На рисунке (рис. 2.2) представлена жалоба пользователя плагина Full Line Code Completion на синтаксически некорректное автодополнение кода. Данный вариант автодополнения кода предлагает вставить в файл программы закрывающую круглую скобку. Но тогда, как и в случае, показанном на предыдущем рисунке, парная открывающая ей скобка будет отсутствовать.

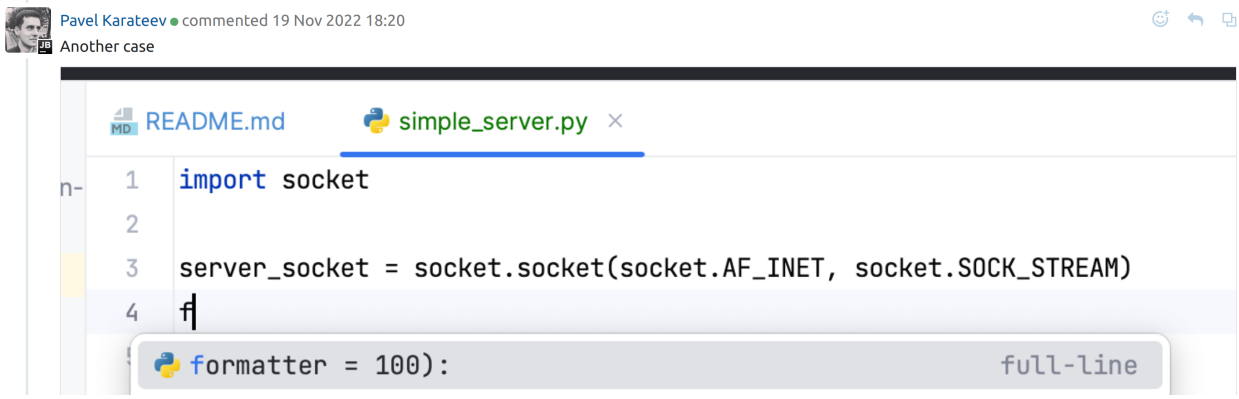

Рис. 2.2: Жалоба пользователя на синтаксически неверное автодополнение

Для обнаружения такой ошибки достаточно произвести вставку автодополнения в код на место каретки пользователя, запустить парсер PyCharm и собрать ошибки на фрагменте автодополнения.

Однако данный алгоритм имеет ложноположительные срабатывания, иногда находя в корректных автодополнениях синтаксические ошибки. Дело в том, что автодополнение кода не всегда является законченным выражением с точки зрения языка программирования. Например, автодополнение может быть лишь началом цикла for (рис. 2.3). Или, например, при написании какого-либо большого выражения — вызова функции с несколькими аргументами, автодополнение может сгенерировать лишь один из аргументов, а не все аргументы целиком. В такой ситуации возникает проблема с описанным выше подходом нахождения синтаксических ошибок. Хотя частичный цикл for (рис. 2.3) и анализируется некорректным с точки зрения парсера языка программирования Python, поскольку такое незаконченное выражение отсутствует в языке программирования. Однако с точки зрения пользователя подобное автодополнение не является ошибочным, поскольку после вставки его в код каретка пользователя переместится в конец вставленного фрагмента, и программист сам завершит выражение до существующего в языке программирования, дописав аргумент функции range, закрыв круглую скобку и поставив двоеточие. Чтобы избежать ложноположительное обнаружение ошибок в таких случаях, в IntelliSense Compose применяют модификацию парсера [5]. В данной же работе был разработан иной алгоритм, работающий с уже существующим парсером PyCharm.

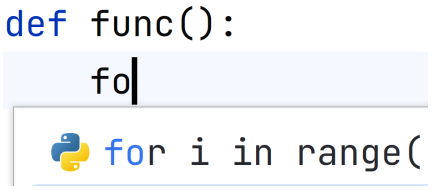

Рис. 2.3: Частичный цикл for предлагаемый автодополнением

Алгоритм состоит из трёх шагов. На первом шаге алгоритма берётся файл исходного кода пользователя, в котором было вызвано автодополнение. Автодополнение вставляется в код на место каретки пользователя, и запускается парсер PyCharm для языка программирования Python. Парсер выдаёт абстрактное синтаксическое дерево. Например, с частичным циклом в примитивном файле исходного кода (рис. 2.3), содержащего лишь единственную функцию func и начало цикла for, абстрактное синтаксическое дерево для файла со вставленным автодополнением будет выглядеть следующим образом (рис. 2.4).

Второй шаг алгоритма является ключевым с точки зрения решения проблем с незаконченными автодополнениями. Он состоит из рекурсивного подъёма по абстрактному синтаксическому дереву, начиная от положения каретки и до корневого элемента дерева. На каждом этапе подъёма анализируются текущие элементы дерева и автодополнение дописывается одним из специальных правил — правил завершения. Каждое правило состоит из стартового элемента, условий на дочерние элементы и

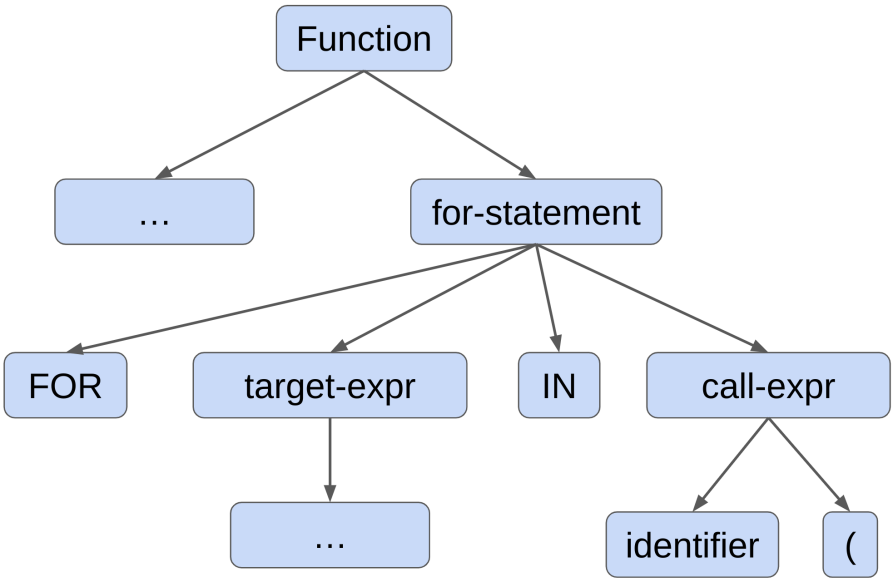

Рис. 2.4: Абстрактное синтаксическое дерево для частичного цикла for

вывода. Стартовый элемент — это каким должен быть текущий рассматриваемый элемент в дереве. Условия — какие элементы должны присутствовать или отсутствовать в качестве дочерних вершин текущего рассматриваемого элемента. Вывод — это какой текст необходимо дописать к автодополнению в случае выполнения условий соответствующего правила. На рисунке (рис. 2.5) представлен набор правил завершения для выражения вызова функции, а на рисунке (рис. 2.6) для цикла for. На изображениях стартовый элемент отмечен синим цветом, условия оранжевым, а вывод зелёным.

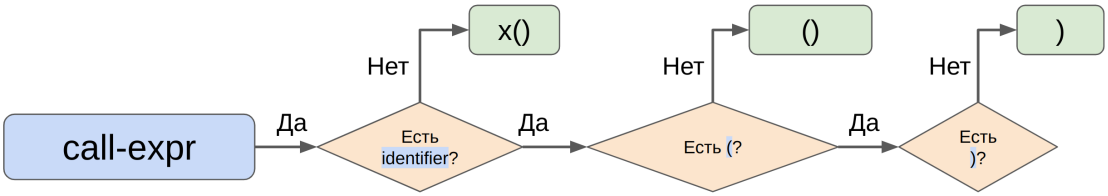

Рис. 2.5: Правила завершения для цикла call-expr

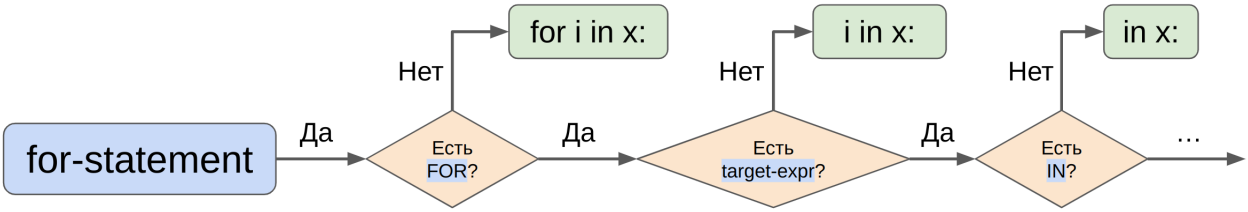

Рис. 2.6: Правила завершения для цикла for

Разберём подробно правила для стартового элемента call-expr. Данные правила применяются в случае, если текущий рассматриваемый элемент дерева имеет тип

call-expr. Затем

1. Если у рассматриваемого элемента отсутствует дочерний элемент типа identifier, то выводом является выражение x(). Этот текст дописывается к автодополнению, и на этом применение правил для элемента call-expr заканчивается.

2. Иначе проверяется наличие дочернего элемента OPEN_BRACE, представляющего собой открывающую круглую скобку. При отсутствии такого элемента выводом является пара круглых скобочек. На этом применение правил в рамках данного элемента заканчивается.

3. Наконец, проверяется отсутствие дочернего элемента CLOSE_BRACE, представляющего собой закрывающую круглую скобку. При его отсутствии к автодополнению дописывается закрывающая круглая скобка.

Всего в ходе данной работы всего было реализовано 81 правило для 36 различных стартовых элементов. Большинство стартовых элементов довольно простые и требуют всего два или три правила, как в случае с call-expr. Но некоторые комплексные конструкции языка программирования Python, такие как циклы for или генераторы списков, могут требовать пять-шесть правил.

Таким образом, в примере с частичным циклом for (рис. 2.3) на втором шаге происходит следующее:

1. Обход начинается с вершины call-expr абстрактного синтаксического дерева, так как это самый низкий элемент в дереве, в котором располагается каретка.

2. Первые два правила для стартового элемента call-expr не имеют вывода, поскольку у рассматриваемого элемента call-expr существует как идентификатор range, так и открывающая круглая скобка.

3. Последнее правило для call-expr применяется с выводом ")", так как ранее в автодополнении отсутствовала закрывающая круглая скобка.

4. Обход для вершины call-expr закончился и поднимается в вершину for-statement.

5. У данной вершины есть множество дочерних элементов, так что все правила кроме самого последнего не применяются. Последнее правило дописывает недостающее двоеточие в выражение.

6. В результате вместе с выводами правил автодополнение выглядит следующим образом: for i in range():.

Таким образом, к концу применений правил мы получаем законченную конструкцию языка программирования, которая может быть успешно проанализирована парсером PyCharm.

На третьем шаге алгоритма, как и на первом шаге, автодополнение вставляется в код пользователя на место каретки, но к нему также добавляется и завершенная на втором шаге часть. После этого запускается парсер PyCharm и при отсутствии ошибок в области вставленного автодополнения оно признаётся синтаксически корректным.

Осталось лишь заметить, что при применении данных правил, корректные автодополнения кода завершаются до полноценных конструкций языка, которые анализируются парсером без ошибок. Если же автодополнение является синтаксически некорректным, например, имеет закрывающую фигурную скобку без парной открывающей, то применение правил не отразится на его синтаксической корректности, поскольку ошибка содержится ещё в изначальном фрагменте автодополнения.

Для тестирования данного алгоритма, во-первых, были разработаны юнит-тесты. Каждый тест представляет собой файл исходного кода с положением каретки в нём, автодополнение кода и флаг корректности автодополнения — должно ли автодополнение проанализироваться как корректное или как некорректное в этом тесте. Всего в ходе данной работы было реализовано более 30 тестов для тестирования данного алгоритма. Во-вторых, для тестирования используется так называемая проверка корректным кодом. Эта проверка позволяет исключить ложноположительные срабатывания алгоритма. Для этого берётся набор корректных файлов из настоящих проектов, в случайное место случайного файла помещается каретка и якобы предлагается автодополнение, являющееся, на самом деле, настоящим продолжением файла. Затем это автодополнение анализируется разработанным методом и проверяется отсутствие ошибок. Так как эти файлы взяты из настоящих и корректных проектов, то ошибки в предлагаемых автодополнениях должны отсутствовать. В данный момент такая проверка производилась на 10 файлах, что помогло исключить ложноположительные срабатывания разработанного алгоритма.

Касательно производительности стоит отметить, что построение абстрактного синтаксического дерева парсером PyCharm происходит ленивым образом: полноценно парсер анализирует только посещаемые элементы в дереве. Поскольку представленный алгоритм задействует не всё абстрактное синтаксическое дерево, а лишь элемент, содержащие каретку, его непосредственных предков и их дочерние элементы, то не стоит ожидать проблем с точки зрения производительности. Действительно, при локальном тестировании время работы данного алгоритма занимает, как правило, от одной до двух миллисекунд, что легко укладывается в лимит 40 миллисекунд. Данные измерения производительности производились на одном компьютере и являются лишь примерной оценкой времени работы. Данные о производительности с настоящих компьютеров пользователей будут представлены в последней главе данной работы.

2.2. Обнаружение ошибок инспекциями PyCharm

На рисунке (рис. 2.7) представлена жалоба пользователя Full Line Code Completion на ошибку в автодополнениях кода. Все автодополнения, предложенные плагином, обращаются к несуществующим переменным модуля socket стандартной библиотеки Python.

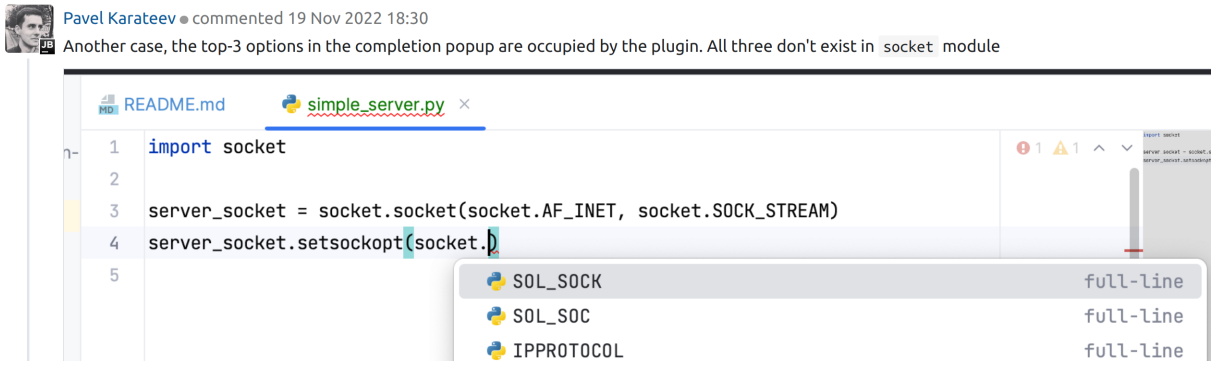

Рис. 2.7: Жалоба пользователя на отсутвующие переменные в автодополнении

Как уже было упомянуто в обзоре литературы, исследователи из Google Research используют языковой сервер для обнаружения подобных ошибок. В PyCharm же для их обнаружения можно воспользоваться инспекциями. Стоит отметить, что инспекции были разработаны в первую очередь для нахождения ошибок в полноценном файле программы, а не в автодополнениях кода. Тем не менее их можно адаптировать к обнаружению ошибок в автодополнениях.

Для этого первым делом автодополнение вставляется в код программы на место каретки таким образом, как оно бы выглядело после вставки программистом в код. Затем, на вставленном фрагменте кода запускается набор из нескольких инспекций. В данный момент используются следующие инспекции среды разработки PyCharm:

- PyUnresolvedReferencesInspection находит ошибки, связанные с использованием отсутствующих переменных, классов, функций и операторов.
- PyCallingNonCallableInspection находит ошибки, связанные с использованием круглых скобок — оператора вызова с объектами, к которым такая операция не применима: константы, переменные, строковые литералы.
- PyArgumentListInspection находит ошибки, связанные с неверно переданными параметрами при вызове функции: неверное число аргументов, неверное использование именованных аргументов и так далее.
- PyRedeclarationInspection находит ошибки, связанные с повторной декларацией, уже декларированной функцией, переменной или класса.

Основная трудность в добавлении новых инспекций заключается в необходимости их специальной адаптации для использования в автодополнениях. Например, рассмотрим инспекцию PyArgumentListInspection. На рисунке (рис. 2.8) представлен фрагмент кода, содержащий функцию start трёх аргументов. Пользователь набирает вызов этой функции, автодополнение предлагает также указать два аргумента. Так как данная функция принимает ровно три аргумента, то инспекция PyArgumentListInspection найдет ошибку в представленном фрагменте автодополнения. Однако очевидно, что автодополнению просто не хватило длины варианта, чтобы сгенерировать все три аргумента, и даже два аргумента являются полезными для пользователя: после их вставки пользователь может самостоятельно дописать третий аргумент и закрыть недостающую скобку. Поэтому было необходимо игнорировать ошибки, обнаруживаемые данной инспекции, если они вызваны недостатком аргументов. При этом, если автодополнение сгенерировало аргументов больше, чем на самом деле есть у функции, то такие ошибки, конечно, необходимо учитывать.

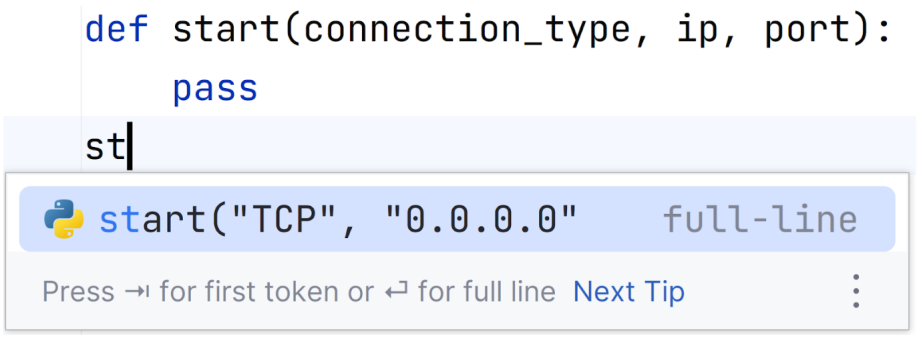

Рис. 2.8: Генерация только части аргументов автодополнением

Другая сложность связана с инспекцией PyUnresolvedReferencesInspection. Среди прочего, она проверяет наличие установленного модуля при его импортировании. Однако зачастую программисты сначала импортируют ещё не установленный модуль, а затем, используя среду разработки PyCharm, устанавливают его по всплывающей подсказке (рис. 2.9).

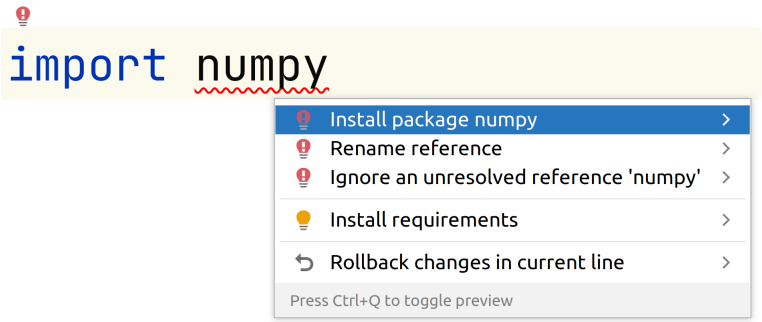

Рис. 2.9: Подсказка установки отсутствующего модуля

Поэтому в данный момент эта инспекция не применяется при генерации автодополнений импортов. Наилучшее решение обозначенной проблемы заключается в сборе списка часто устанавливаемых модулей с помощью подсказки PyCharm. Затем необходимо игнорировать ошибки в автодополнениях, обнаруженные этой инспекцией при написании таких модулей, в независимости от их наличия локально на компьютере пользователя.

Для тестирования корректности обнаружения ошибок инспекциями были разработаны юнит-тесты. Каждый тест представляет собой файл, положение каретки в нём и автодополнение, в котором должна или не должна найтись ошибка. Всего было разработано 140 таких тестов.

Часто автодополнения обращаются к тем же переменным, функциям или классам, которые уже так или иначе упоминались в файле пользователя. Соответственно их определения уже найдены средой разработкой PyCharm, а типы — выведены. Так что инспекции просто используют эти знания для проверки нескольких условий и генерации ошибки при необходимости. Таким образом, инспекции, как правило, требуют считанные миллисекунды для обнаружения ошибок. Локальные измерения показывают, что на выполнение всех инспекций тратится, как правило, всего пять миллисекунд, хотя в некоторых сложных случаях анализ может занимать десятки миллисекунд. Данные о производительности с настоящих компьютеров пользователей будут представлены в последней главе данной работы.

2.3. Выводы и результаты по главе

В данной главе были представлены методы обнаружения ошибок в автодополнениях кода на языке Python, используя методы среды разработки PyCharm.

- Для обнаружения синтаксических ошибок был разработан специальный алгоритм, использующий парсер PyCharm. Представленное решение находит те же ошибки, что и решение от IntelliSense Compose, приведенное в обзоре литературы. Однако разработанное в данной работе решение не требует внутреннюю модификацию парсера, а работает поверх него. Это позволяет, во-первых, интегрировать разработанное решение в среды разработки от JetBrains, обладающие своим собственным фиксированным парсером, а во-вторых, использовать произвольный парсер для обнаружения синтаксических ошибок. Что, в свою очередь, позволяет адаптировать поиск синтаксических ошибок в автодополнениях к другим языкам программирования без необходимости выбирать конкретный парсер и разбираться в его спецификации.

- Для обнаружения более комплексных ошибок в автодополнениях были адаптированы инспекции среды разработки PyCharm. Всего было адаптировано 4

инспекции, приоритет выбора инспекций отдавался согласно количеству жалоб пользователей Full Line Code Completion. Хотя практически с каждой инспекцией связаны различные трудности, можно продолжать адаптировать инспекции PyCharm к использованию в автодополнениях кода. В данной работе обнаруживается два из трёх видов ошибок, находимые в Google Research. Оставшийся ещё неподдержанный вид ошибок, находимый в автодополнениях от Google Research, не имеет публичных жалоб от пользователей, поэтому не являлся приоритетным с точки зрения поддержки.

- Разработанные алгоритмы используют те же самые методы обнаружения ошибок в автодополнениях, что и сам PyCharm для поиска ошибок в файлах исходного кода. Это обеспечивает согласованность: в автодополнениях обнаруживаются те же ошибки, что и средой разработки после вставки автодополнений в код.

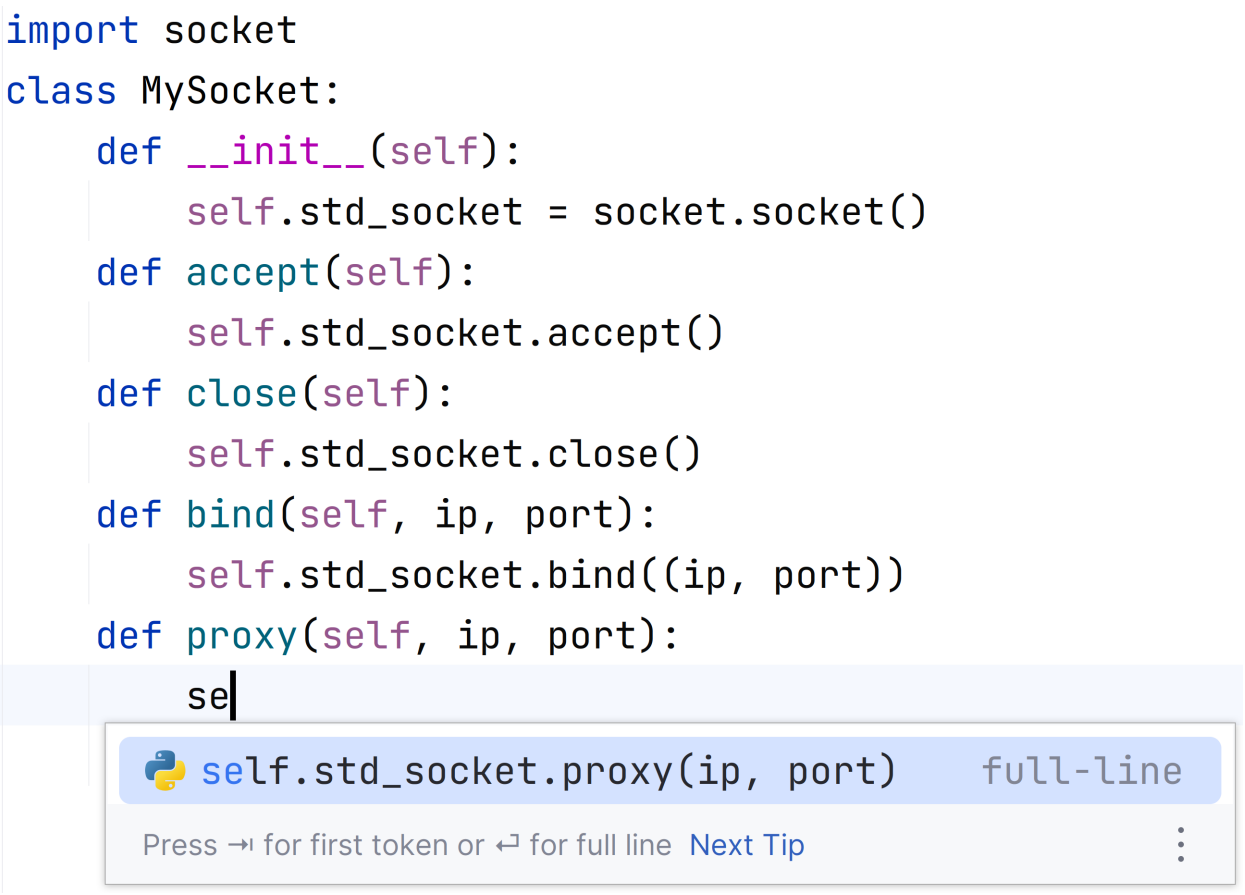

Рис. 3.1: Автодополнение предлагает функцию, отсутствующую в библиотеке

3. Классификация критичности ошибок

3.1. Мотивация и показательные случаи

В предыдущей главе был представлен алгоритм нахождения ошибок в автодополнениях, при этом все найденные ошибки по результатам предыдущей главы считаются равнозначными, так как никакой дополнительной классификации ошибок не вводилось. Однако разные ошибки имеют разный ущерб с точки зрения пользователей. Кроме того, некоторые ошибки в определённых ситуациях и вовсе являются незначительными и могут быть проигнорированы. Конечно, стратегии обработки ошибок определяются инструментом, который собирается использовать алгоритмы, разработанные в ходе данной работы. Некоторые инструменты хотят быть более консервативными и серьёзно относится ко всем ошибкам, другие, наоборот, хотят показывать потенциально ошибочные варианты автодополнений пользователям, предлагая программистам самим выбрать наиболее подходящий вариант автодополнения. Тем не менее существует как минимум два принципиально различных класса ошибок. О них и пойдёт речь в этой главе.

В данном примере (рис. 3.1) имеется класс MySocket на языке Python, представля-

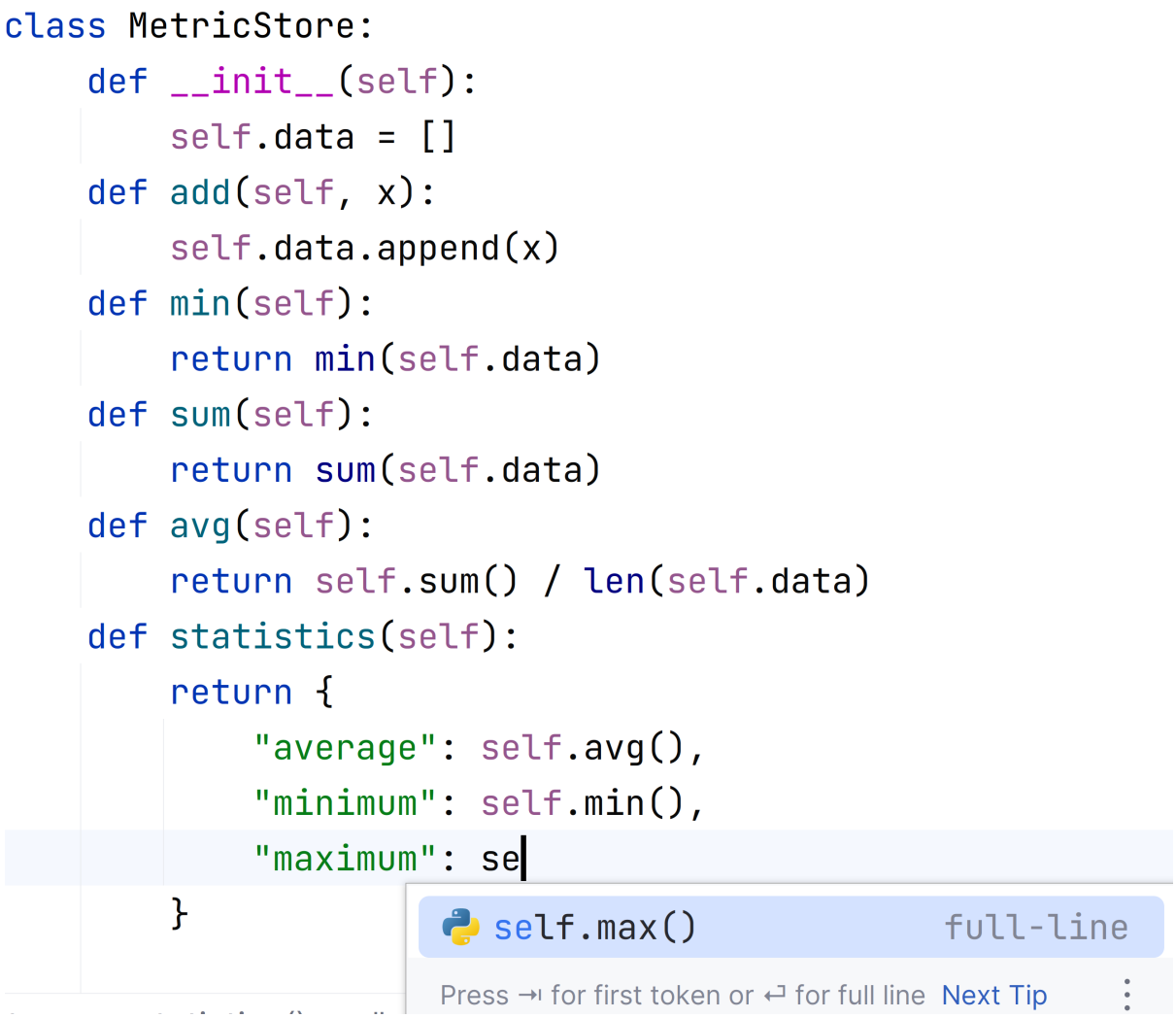

Рис. 3.2: Автодополнение предлагает функцию, отсутствующую в текущем классе

ющий собой обёртку над классом socket из стандартной библиотеки Python. В классе MySocket уже реализованы некоторые методы посредством делегирования к классу socket. Программист хочет написать тело метода proxy и автодополнение предлагает ему вызвать соответствующую функцию из класса socket стандартной библиотеки. Однако такая функция в стандартной библиотеке отсутствует и данное автодополнение является ошибочным. Оно обманывает программиста, создавая впечатление, что функция proxy в стандартной библиотеке существует. В данном случае один из вариантов решения проблемы — это сокрытие ошибочного автодополнения.

Рассмотрим другой пример (рис. 3.2). В класс MetricStore можно добавлять числовые значения (функция add), а также считать некоторые статистики, такие как сумму или среднее значение по добавленным значениям. Программист пишет тело метода statistics, метод должен возвращать словарь с набором различных статистик. Автодополнение предлагает программисту как раз то, что он хочет — записать результат функции максимум под ключом maximum. Программист выбирает данное

автодополнение и обнаруживает, что функцию self.max он ещё не успел реализовать. После чего он реализует функцию self.max в текущем классе, принимая, например, предложение PyCharm создать недостающую функцию (рис. 3.3).

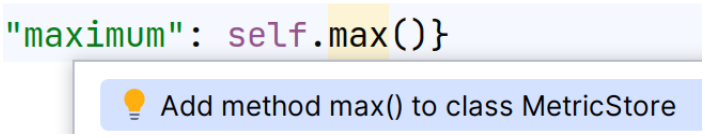

Рис. 3.3: Подсказка PyCharm создать недостающую функцию

Хотя в обоих примерах автодополнения содержат ошибки, в первом случае автодополнение нежелательно для пользователя, но во втором, напротив, именно предложенный вариант с ошибкой пользователь хочет написать. Действительно, иногда программисты сначала пишут вызовы функций, обращаются к переменным или классам и только затем декларируют их. Тем не менее иногда создать недостающую функцию крайне легко, как во втором примере, когда необходимо лишь дописать её в текущем классе. В ситуациях как в первом примере, предлагаемая отсутствующая функция должна появиться в библиотеке, чтобы предлагаемый автодополнением код стал корректным. Данное отличие позволяет выделить как минимум два класса во множестве всех находимых ошибок.

3.2. Алгоритм классификации

В данной работе была разработана следующая классификация критичности ошибок:

- Критичные ошибки — это такие ошибки, что при вставке их в код пользователя полученную ошибку нельзя будет никак устранить в текущем файле, кроме как переписав вставленное автодополнение.
- Приемлемые ошибки — это все оставшиеся ошибки, то есть такие, что при вставке их в код возникнет некоторая ошибка, однако в текущем файле возможно доопределить, например, недостающую функцию или переменную, чтобы устранить ошибку.

Так как пользователь зачастую активно редактирует только один файл исходного кода, то данная классификация разделяет ошибки, которые могут быть исправлены в уже редактируемом в данный момент файле, и те, для исправления которых без изменения автодополнения придётся переходить в другие файлы.

Таким образом, все синтаксические ошибки, находимые парсером, являются критичными. Касательно ошибок, находимых инспекциями, для определения их критичности необходимо выполнить нахождение декларации той или иной функции, переменной или класса, у которой отсутствующая функция вызывается. Например, в

случае с недостающей функцией max (рис. 3.2) средствами PyCharm происходит разрешение квалификатора self, ведущего в текущий файл, так что данная ошибка считается приемлемой. В случае с недостающей функцией proxy (рис. 3.1) квалификатор self.std_proxy приводит к стандартной библиотеке, так что данная ошибка определяется как критичная.

Стоит отметить, что в языке программирования Python возможно динамическое присваивание функций и переменных классам. Распространённый пример использования динамических переменных — это класс ArgumentParser, позволяющий обращаться к аргументам командой строки как к переменным класса. В таком случае находить ошибки отсутствия переменных было бы неправильно, поскольку отсутствующая переменная может быть динамически создана из аргументов командной строки.

Данная проблема присутствует не только на этапе анализа автодополнений, но и при анализе файла целиком, так что она уже была решена в PyCharm. PyCharm обладает детектором динамических переменных и функций, который на основе документации, аннотаций и использований функций распознаёт классы с динамическими переменными и функциями и выключает обнаружение ошибок в таких случаях. Этот же детектор был использован и в данной работе, чтобы не находить ошибки в классах с динамическими переменными и функциями и работать согласованно с PyCharm.

Для тестирования разработанного алгоритма классификации используются те же самые тесты, что были представлены в предыдущей главе. Однако теперь у каждого теста, проверяющего наличие той или иной ошибки в автодополнении, появился специальный параметр с уровнем необходимой критичности, который должен быть присвоен ошибке для успешного прохождения теста.

3.3. Выводы и результаты по главе

В данной главе была представлена классификация ошибок по уровню критичности. Это минимальная классификация, разделяющая легко исправимые и неисправимые в текущем файле ошибки в автодополнениях.

Разработанная классификация позволяет по-разному относиться к ошибкам инструментам, использующим результаты данной работы. Например, автодополнения, содержащие критичные ошибки, можно скрывать, а автодополнения с приемлемыми ошибками показывать пользователю, но помечать как потенциально нежелательные.

Стоит отметить, что это лишь один из вариантов работы с ошибками, который инструмент автодополнения может использовать. Кроме других вариантов обработки ошибок могут быть добавлены новые уровни критичности ошибок, чтобы добиться требуемых от инструмента автодополнения качеств. Например, инструменты автодополнений могут скрывать все автодополнения с любыми ошибками или, например,

скрывать автодополнения, содержащие только критичные ошибки. Конкретное применение классификации ошибок в инструменте Full Line Code Completion будет представлено в следующей главе данной работы.

4. Интеграция в плагин Full Line Code Completion

4.1. Интеграция разработанных методов

В плагине Full Line Code Completion присутствует конвейер пост-обработки сгенерированных вариантов автодополнений кода. В него включены различные примитивные фильтрации автодополнений. Например, отсеивание вариантов автодополнений, содержащих служебные символы, ругательные слова и тому подобное. Затем отсеиваются варианты, похожие друг на друга, и уже затем получившиеся варианты отображаются пользователю. Поскольку нахождение ошибок в автодополнениях является самой тяжеловесной операцией по сравнению со всеми существующими пост-обработками, то при интеграции было принято решение расположить данный вид анализа в самом конце конвейера обработки вариантов — прямо перед показом автодополнений пользователю.

Следующий этап интеграции заключался в разработке стратегий по обработке находимых ошибок в автодополнениях. Существует несколько различных возможных стратегий:

- Игнорирование — автодополнение отображается программисту в исходном виде, вне зависимости от наличия ошибок.
- Маркировка — автодополнение отображается программисту, однако каким-либо образом маркируется как некорректное.
- Исправление — перед показом автодополнения ошибки в нём исправляются.
- Скрытие — автодополнение с ошибками не отображается пользователю.

Так как в данной работе была разработана классификация критичности автодополнений, то для разных уровней критичности можно выбрать разные стратегии обработки. Исправление автодополнений требует отдельных усилий по разработке соответствующих алгоритмов, не проводившихся в рамках данной работы. Данный подход сразу был исключен из рассмотрения. Вариант с игнорированием уступает варианту с маркировкой как минимум на первых этапах разработки, поскольку лучше показать пользователю найденную ошибку, чем не показать её вовсе. Таким образом, были выбраны стратегии маркировки и скрытия для приемлемых и критичных автодополнений соответственно. Действительно, исправить критичные ошибки в автодополнениях либо невозможно совсем, либо требует дополнительных усилий от программиста в посторонних файлах проекта. Поэтому скрытие автодополнений с такими ошибками является разумной стратегией обработки. Касательно приемлемых ошибок, иногда автодополнения с ними бывают полезными, как было показано в предыдущей главе

данной работы. Подход с маркировкой показывает такие автодополнения пользователям и позволяет им самим принять решение о выборе данного автодополнения, ориентируясь на отображаемую маркировку.

Для отображения ошибок в редакторе кода PyCharm классический используемый способ заключается в подчёркивании места ошибки красной волнистой линией (рис. 4.1). Поэтому для маркировки ошибок в автодополнениях было принято решение разработать аналогичный интерфейс. Изначально среда разработки PyCharm не поддерживала возможность маркировки фрагмента автодополнения, можно было помечать только всё автодополнение целиком волнистой линией. Поэтому были сделаны соответствующие изменения в платформу среды разработки PyCharm, открывающие данную возможность. Затем, используя разработанный интерфейс отображения, было поддержано выделение красной волнистой линией ошибочных фрагментов в автодополнениях (рис. 4.2).

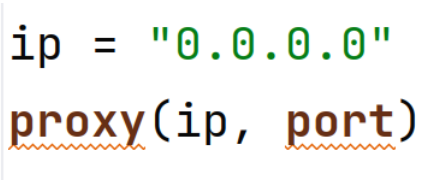

Рис. 4.1: Маркировка ошибок PyCHarm в исходном файле программы

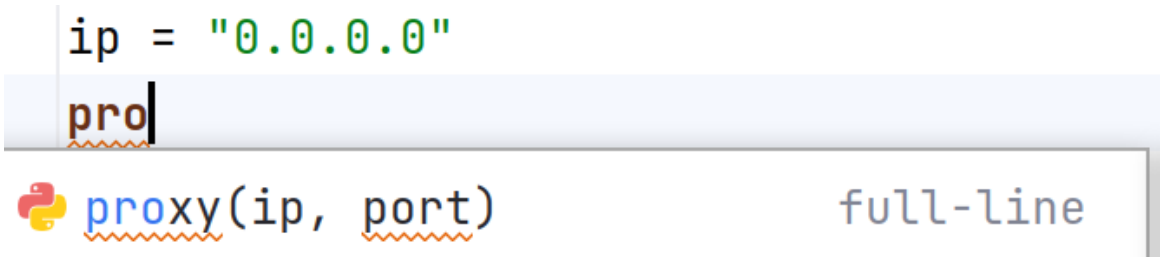

Рис. 4.2: Разработанная маркировка ошибок в автодополнениях

Плагин Full Line Code Completion имеет пользовательские и внутренние настройки. Пользовательские настройки доступны сразу после установки плагина и содержат небольшое количество настраиваемых опций. Внутренние настройки доступны только разработчикам плагина и позволяют конфигурировать плагин более гибко. Для пользовательских настроек в рамках данной работы была добавлена опция включения и выключения анализа некорректных автодополнений кода (рис. 4.3). Данная опция включена по умолчанию, однако при желании пользователи могут выключить её. В расширенные настройки также добавлен переключатель стратегий обработки некорректных автодополнений (рис. 4.4).

Кроме того, в плагине Full Line Code Completion имеется специальное окно диагностики, предназначенное для разработчиков. Данное окно служит для анализа разработчиками сгенерированных вариантов автодополнений и этапов их пост-обработки.

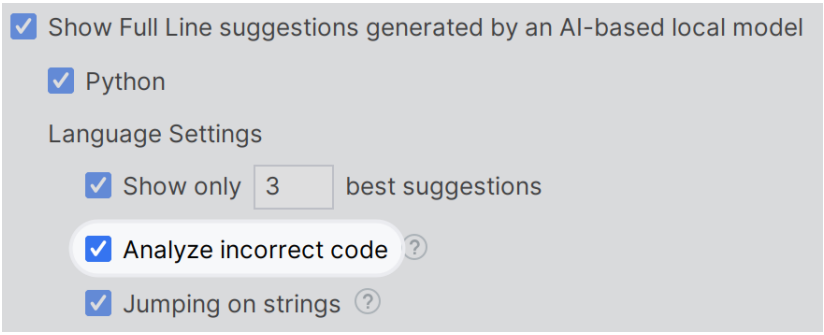

Рис. 4.3: Пользовательские настройки Full Line Code Completion

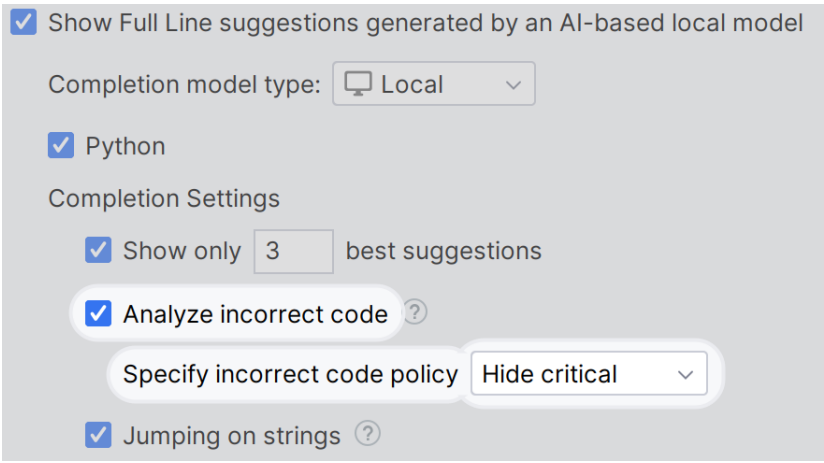

Рис. 4.4: Расширенные настройки Full Line Code Completion

В ходе данной работы была добавлена специальная вкладка в окно диагностики, просвещённая анализу ошибок в автодополнениях (рис. 4.5). Красным цветов в этом окне обозначаются области критичных ошибок, жёлтым — приемлемых, а зелёным — фрагменты автодополнений, не содержащие никаких ошибок.

Во время работы у пользователя плагин Full Line Code Completion записывает различную информацию в логи [1]. Например, данные о том, был ли выбран вариант автодополнения, сгенерированный инструментом Full Line Code Completion. Сколько времени потребовалось на генерацию автодополнений и различная другая информация. Подобные логи собираются со всех пользователей, установивших плагин в программу раннего доступа среды разработки PyCharm [7].

Существует несколько ограничений по информации, которую можно собирать с пользователей в логи. Во-первых, все логируемые данные должны удовлетворять ограничением GDPR [13] об анонимности собираемой информации. Во-вторых, в данный момент есть возможность собирать информацию только о тех вариантах автодополнения, что были показаны пользователю, а не скрыты в процессе их постобработки.

В ходе данной работы было реализовано логирование различной информации касательно обнаружения ошибок в автодополнениях кода от Full Line Code Completion:

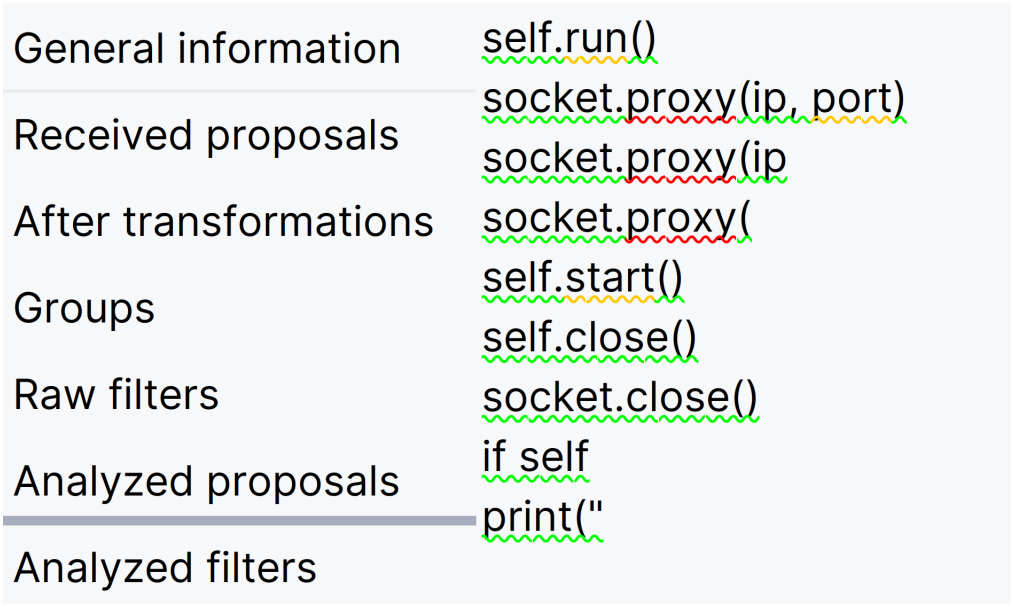

Рис. 4.5: Диагностика анализа ошибок автодополнений Full Line Code Completion

- checks_time — время в миллисекундах, затраченное на обнаружение ошибок в варианте автодополнения.
- syntax_state — были ли обнаружены ошибки во время анализа варианта автодополнения парсером PyCharm. Принимает одно из значений:
  - CORRECT — в варианте автодополнения нет ошибок, обнаруженных парсером.
  - INCORRECT — вариант автодополнения содержит хотя бы одну ошибку, обнаруженную парсером.
  - UNKNOWN — вариант автодополнения не был проанализирован парсером на ошибки (например, из-за отключения анализа в настройках).
- semantic_state — были ли обнаружены ошибки в варианте автодополнения инспекциями PyCharm. Принимает одно из значений:
  - CORRECT — в варианте автодополнения нет ошибок, обнаруженных инспекциями.
  - INCORRECT_CRITICAL — вариант автодополнения содержит хотя бы одну критичную ошибку, обнаруженную инспекциями.
  - INCORRECT_ACCEPTABLE — вариант автодополнения содержит хотя бы одну ошибку, обнаруженную инспекциями, но не содержит критичных ошибок, обнаруженных инспекциями.
  - UNKNOWN — вариант автодополнения не был проанализирован инспекциями на ошибки.

С марта по начало апреля 2023 года проводилась очередная программа раннего доступа. В ней части пользователям PyCharm в обязательном порядке устанавливался плагин Full Line Code Completion. Это помогло собрать множество данных касательно работы реализованных алгоритмов у пользователей. Данные будут представлены в следующем разделе данной главы.

4.2. Анализ практического применения

В результате интеграции разработанных алгоритмов обнаружения ошибок в автодополнениях кода были закрыты множество жалоб пользователей Full Line Code Completion.

- Обнаружение ошибок в автодополнениях с помощью парсера PyCharm привело к закрытию всех трёх жалоб пользователей на синтаксически некорректные автодополнения, причём в каждой из жалоб пользователи прилагали несколько примеров синтаксически некорректных автодополнений. Все они были исправлены с внедрением разработанных в рамках данной работы алгоритмов. Новых жалоб пользователей на этот тип ошибок не поступало.
- Обнаружение ошибок в автодополнениях с помощью инспекций привело к закрытию 11 жалоб пользователей.
- 4 жалобы пользователей остались незакрытыми
  - Две из них касаются ошибочных автодополнений при импортировании модуля, эта проблема и её решение было описано во второй главе данной работы.
  - Оставшиеся две жалобы касаются ошибок, которые могут быть решены с адаптацией ещё нескольких инспекций.

С 29 марта по 4 апреля проходила последняя программа раннего доступа весеннего сезона 2023 года. В её рамках пользователям были доступны все алгоритмы, разработанные в ходе этой работы. Данные на графиках (рис. 4.6 и 4.7) получены от 350 пользователей, которые участвовали в программе раннего доступа в течение указанной недели.

На рисунке (рис. 4.6) представлено распределение времени работы разработанных алгоритмов в миллисекундах у пользователей. В более чем 70% случаев время работы не превышает 10 миллисекунд. При этом желаемое ограничение было 40 миллисекунд. Время работы не превосходит 40 миллисекунд в 93% случаях. Всего было получено 847 таких измерений.

На рисунке (рис. 4.7) представлена вероятность варианта содержать ошибки. Розовым цветом отмечена вероятность автодополнения содержать только некритичные

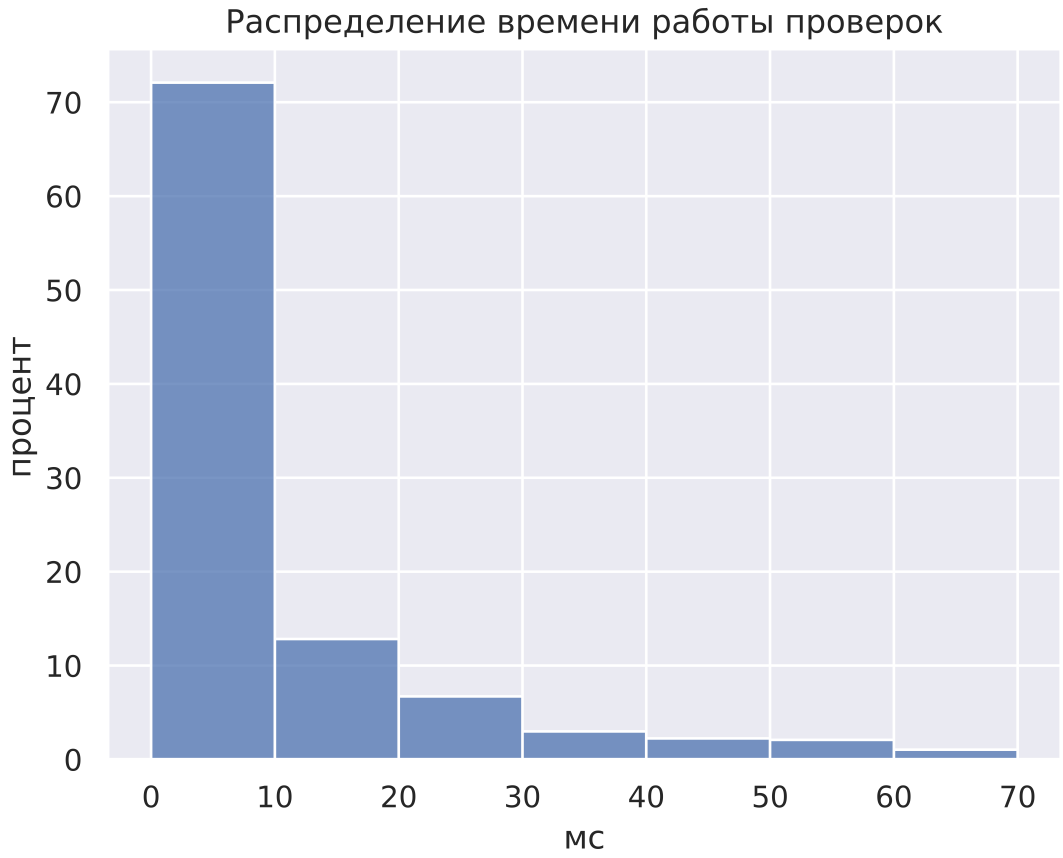

Рис. 4.6: Распределение времени работы проверок у пользователей

ошибки, а тёмно-красным — хотя бы одну критичную ошибку. На рисунке представлено два столбика.

- Левый столбик получен в результате автоматического анализа ошибок в автодополнениях в различных файлах исходного кода на языке программирования Python. Было выбрано 20 случайных файлов из тренировочного датасета модели машинного обучения, используемой в Full Line Code Completion. Затем, в произвольное место выбранных файлов ставилась каретка в автоматическом режиме и вызывалось автодополнение. Генерируемые автодополнения от Full Line Code Completion проверялись на ошибки разработанными методами. Оказалось, что в более, чем 4% процентах случаях автодополнение Full Line Code Completion содержит хотя бы одну приемлемую ошибку и также в порядка 4% процентах содержится хотя бы одна критическая ошибка.

- Правый столбик представляет аналогичную информацию, однако посчитанную не по автодополнениям из случайных файлов, а из статистики по пользователям за указанный промежуток времени. Так как текущий использованный механизм записи логов не позволяет записать информацию касательно скрытых автодо-

полнений кода, то о проценте критичных ошибок в варианте в правом столбике информации в данный момент нет. Тем не менее приемлемых ошибок в вариантах автодополнений генерируется даже больше, чем при анализе случайных файлов. Вероятно, это объясняется тем, что пользователи чаще пишут и редактируют содержательные и нетривиальные фрагменты файлов, в то время как вызовы автодополнений в случайных позициях вызывают автодополнения и в очевидных местах, где оно не допускает ошибки.

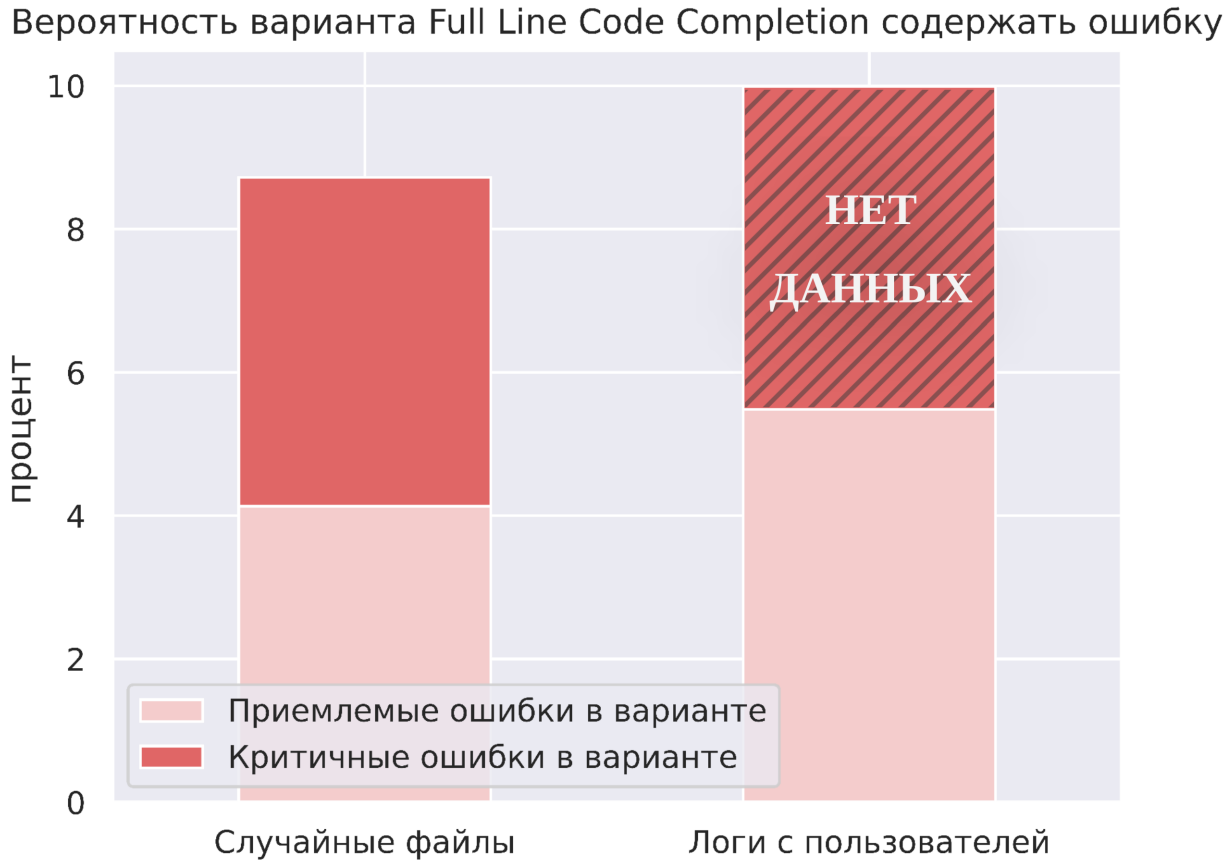

Рис. 4.7: Вероятность варианта Full Line Code Completion содержать ошибку

4.3. Выводы и результаты по главе

- Разработанные алгоритмы были интегрированы в плагин Full Line Code Completion. Были добавлены настройки, диагностика и логирование, касаемые обнаружения ошибок в автодополнениях кода.
- 14 из 18 жалоб пользователей были закрыты, дальнейшая разработка позволит закрыть оставшиеся жалобы.
- Время работы разработанных алгоритмов соответствует желаемому в 93% случаев, что является отличным результатом. Существуют различные возможно-

сти по оптимизации данного времени, однако в данный момент они не являются приоритетными задачами.

- Полученные логи подтвердили работоспособность алгоритмов у пользователей, однако следует собирать больше информации для проведения более подробной аналитики.

Заключение

В ходе данной работы были разработаны методы обнаружения ошибок в автодополнениях с помощью среды разработки PyCharm. В отличие от решения в IntelliCode Compose, разработанное решение реализовано поверх парсера языка программирование и не требует его модификацию. Это позволяет не фиксировать конкретный парсер при разработке алгоритма и легче переносить решение на другие языки программирования. Данное решение закрыло все жалобы пользователей касательно синтаксически некорректных автодополнений кода и успешно показало себя с точки зрения производительности.

Использование инспекций также помогло обнаруживать ошибки в автодополнениях в большом количестве и делать это эффективно. Всего было адаптировано 4 инспекции к поиску ошибок в автодополнениях. При этом приоритет выбора инспекций для адаптации отдавался в первую очередь тем инспекциям, что их интеграция поможет закрыть наибольшее число жалоб пользователей. Тем не менее следует адаптировать и не рассмотренные в рамках данной работы инспекции среды разработки для нахождения ещё большего числа ошибок в автодополнениях.

Также в ходе данной работы была разработана классификация ошибок, находимых в автодополнениях кода. Предложенная классификация позволяет выделять два класса критичности ошибок в автодополнениях: приемлемые и критичные. Это позволяет инструментам автодополнений, таким как Full Line Code Completion, по-разному относиться к разным обнаруживаемым ошибкам. Кроме того, разработанные решения поддерживают возможность добавления большего числа выделяемых классов.

Наконец, решение было интегрировано в плагин Full Line Code Completion в среде разработки PyCharm для языка программирования Python. Для этого были выбраны две простейшие стратегии обработки находимых ошибок. В результате 14 из 18 жалоб пользователей касательно некорректных автодополнений кода были закрыты, а оставшиеся проблемы также могут быть решены с адаптацией не рассмотренных в ходе данной работы инспекций. Был реализован сбор информации с пользователей о производительности разработанных решений. Оказалось, что время работы представленных алгоритмов у пользователей в 93% случаев соответствует желаемому ограничению.

Разработанные алгоритмы успешно используются в инструменте Full Line Code Completion, но могут быть интегрированы и в другие плагины генерации автодополнений кода среды разработки PyCharm. Дальнейшее развитие представленных подходов будет заключаться в поддержке большего числа инспекций, закрытия большего числа жалоб пользователей и адаптацией разработанных алгоритмов для других языков программирования. В ближайшем будущем планируется реализовать аналогичные решения для языка программирования Kotlin.

Действительно, разработанные решения могут быть применены к поиску ошибок в автодополнениях кода не только на языке Python, но и на других языках программирования. Для этого будет необходимо реализовать аналогичные правила завершения для корректного обнаружения синтаксических ошибок в автодополнениях. Затем будет необходимо выбрать и адаптировать инспекции соответствующей среды разработки. Стоит отметить, что набор инспекций в средах разработки JetBrains довольно схож для разных языков программирования, поскольку потенциальные проблемы в языках программирования зачастую похожи. Таким образом, адаптировать разработанные решения в другие среды разработки от JetBrains для других языков программирования потребует значительно меньше усилий, чем первичная разработка, проведённая в рамках данной работы для языка Python.

Список литературы

[1] Bibaev Vitaliy, Kalina Alexey, Lomshakov Vadim et al. All You Need Is Logs: Improving Code Completion by Learning from Anonymous IDE Usage Logs. — 2022. — 2205.10692.

[2] Aye Gareth Ari, Kaiser Gail E. Sequence Model Design for Code Completion in the Modern IDE. — 2020. — 2004.05249.

[3] Chen Mark, Tworek Jerry, Jun Heewoo et al. Evaluating Large Language Models Trained on Code. — 2021. — 2107.03374.

[4] Hou Daqing, Wang Yuejiao. An Empirical Analysis of the Evolution of User-Visible Features in an Integrated Development Environment // Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research. — CASCON '09. — USA : IBM Corp., 2009. — P. 122–135. — URL: https://doi.org/10.1145/1723028.1723044.

[5] Svyatkovskiy Alexey, Deng Shao Kun, Fu Shengyu, Sundaresan Neel. IntelliCode Compose: Code Generation Using Transformer. — 2020. — 2005.08025.

[6] The IntelliJ Platform: a Framework for Building Plugins and Mining Software Data / Zarina Kurbatova, Yaroslav Golubev, Vladimir Kovalenko, Timofey Bryksin // CoRR. — 2021. — Vol. abs/2110.00141. — arXiv : 2110.00141.

[7] JetBrains. Early Access Programs (EAP) - JetBrains // The Early Access Program. — 2023. — URL: https://www.jetbrains.com/resources/eap/ (дата обращения: 13.05.2023).

[8] Marasoiu Mariana, Church Luke, Blackwell Alan F. An empirical investigation of code completion usage by professional software developers // Annual Workshop of the Psychology of Programming Interest Group. — 2015.

[9] Maxim Tabachnyk Stoyan Nikolov. ML-Enhanced Code Completion Improves Developer Productivity // Google research blog. — 2022. — URL: https://ai.googleblog.com/2022/07/ml-enhanced-code-completion-improves.html (дата обращения: 01.05.2023).

[10] OpenAI. GPT-4 // GPT-4. — 2023. — URL: https://openai.com/research/gpt-4 (дата обращения: 01.05.2023).

[11] Spirin Egor, Bogomolov Egor, Kovalenko Vladimir, Bryksin Timofey. PSIMiner: A Tool for Mining Rich Abstract Syntax Trees from Code. — 2021. — 2103.12778.

[12] A Study of Visual Studio Usage in Practice / Sven Amann, Sebastian Proksch, Sarah Nadi, Mira Mezini // 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). — Vol. 1. — 2016. — P. 124–134.

[13] Union European. General Data Protection Regulation // General Data Protection Regulation. — 2018. — URL: https://gdpr.eu/ (дата обращения: 06.05.2023).

[14] Xu Frank F., Vasilescu Bogdan, Neubig Graham. In-IDE Code Generation from Natural Language: Promise and Challenges. — 2021. — 2101.11149.

[15] Youtrack. project: IntelliJ IDEA Subsystem: Editor. Code Completion. Full line tag: redcode // Full Line Code Completion – incorrect code generation. — 2023. — URL: https://youtrack.jetbrains.com/issues/IDEA?q=project:%20%7BIntelliJ%20IDEA%7D%20Subsystem:%20%7BEditor.%20Code%20Completion.%20Full%20line%7D%20tag:%20redcode%20 (дата обращения: 01.05.2023).

[16] s.r.o JetBrains. Full Line Code Completion // Full Line Code Completion - IntelliJ IDEs Plugin | Marketplace. — 2023. — URL: https://plugins.jetbrains.com/plugin/14823-full-line-code-completion (дата обращения: 01.05.2023).