Санкт-Петербургский государственный университет

Кафедра системного программирования

Калина Алексей Игоревич

Анализ качества автодополнения кода в интегрированных средах разработки

Магистерская диссертация

Научный руководитель:
канд. физ.-мат. наук, ст. преп. Луцив Д. В.

Рецензент:
программист, ООО "Интеллиджей Лабс" Бибаев В. И.

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software Engineering

Kalina Alexey

Code completion quality analysis in integrated development environments

Master's Thesis

Scientific supervisor:
Candidate of Physics and Mathematics D.V. Lutsiv

Reviewer:
Programmer, IntelliJ Labs Co. Ltd. V.I. Bibaev

Saint-Petersburg
2020

Оглавление

Введение **4**

1. Обзор **8**

2. Плагин оценки качества автодополнения **13**

2.1. Архитектура 13

2.1.1. Формирование действий 13

2.1.2. Исполнение действий 15

2.1.3. Создание отчетов 16

2.2. Стратегии формирования запросов автодополнения . . . 18

2.2.1. Контексты 18

2.2.2. Префиксы 19

2.2.3. Фильтры 20

2.3. Метрики 21

3. Автоматизация оценки качества автодополнения **22**

3.1. Оценка качества автодополнения 23

3.2. Сравнение алгоритмов автодополнения 23

3.3. Построение моделей машинного обучения на основе искусственных запросов 24

4. Моделирование поведения пользователя **25**

4.1. Вероятностная модель поведения пользователя 25

4.2. Эксперименты 26

Заключение **29**

Список литературы **30**

Введение

Текущие реалии таковы, что редкий разработчик программных продуктов пишет код в обычных текстовых редакторах. Большинство программистов используют интегрированные среды разработки (IDE) или редакторы кода, которые поддерживают такие функции как: подсветка синтаксиса, автодополнение, автоматическое форматирование и другие. Благодаря этим возможностям повышается скорость и удобство разработки программного обеспечения.

Автодополнение кода — важная функция, предоставляемая средами разработки. Она может вызываться по команде либо отрабатывать автоматически следующим образом: при частичном вводе токена во всплывающем окне отображается выдача с подсказками (предложениями) о том, как дополнить этот токен. Автодополнение кода решает две основные задачи:

1. Сокращение времени набора. Вместо печати всего идентификатора достаточно ввести несколько символов и выбрать нужный в выдаче автодополнения.

2. Изучение внешнего интерфейса. Программистам часто приходится работать со сторонними библиотеками. Одним из способов изучения возможностей этих библиотек и содержащихся в них сущностях (наряду с изучением документации) является использование автодополнение кода.

Задача автодополнения решается различными способами. Наивный подход заключается в сортировке предложений в алфавитном порядке. В более сложном варианте принимается во внимание степень соответствия предложения уже введенным символам и информация о текущем контексте. Существуют подходы, которые учитывают предпочтения конкретного пользователя в использовании автодополнения, историю изменения проекта или популярность отдельных идентификаторов. Для сравнения различных реализаций автодополнения нужен инструмент оценки качества.

В этой работе качество автодополнения рассматривается в контексте сокращения времени набора кода. Необходимость в оценке качества автодополнения существует по ряду причин:

- определение того, как изменение в реализации алгоритма автодополнения влияет на его качество;
- выявление ситуаций, когда автодополнение работает плохо;
- сравнение нескольких алгоритмов автодополнения.

Существуют разные подходы к оценке качества автодополнения кода. Тем не менее, их можно разделить на две основные группы:

- оценка на основе пользовательской истории [1];
- оценка на основе искусственных запросов [2, 3];

В первом подходе используются логи с информацией об использовании автодополнения кода реальными пользователями. Такой способ дает максимально точную информацию о том, как хорошо работает алгоритм. Тем не менее, у него есть свои недостатки. Сбор актуальных логов может быть трудоемкой задачей, в особенности если обновления в алгоритме происходят регулярно и необходима постоянная оценка его качества.

Лишен этой проблемы второй подход, в котором не используются данные о действиях пользователей. Он основан на формировании искусственных запросов автодополнения в некотором готовом исходном коде. Стандартная реализация такого подхода заключается в удалении некоторых токенов из кода, вызова на их месте автодополнения и сравнении выдачи с исходным токеном.

Подход с использованием искусственных запросов является менее точным по сравнению с первым вариантом. Это связано с тем, что формируемые запросы не всегда соответствуют реальности [4]. Другими словами, возможны варианты когда, пользователь не стал бы вызывать автодополнение кода в той ситуации, которая была промоделирована

в искусственном запросе. Несмотря на этот недостаток, второй способ (в отличие от первого) позволяет быстрее решать все те задачи, которые были поставлены в контексте оценки качества автодополнения. Поэтому именно он является основой этой работы.

С проблемой отличия искусственных запросов от пользовательских можно бороться разными способами. Используя информацию о реальных сессиях пользователей, можно фильтровать токены, на которых вызывать автодополнение. Другой способ заключается в моделировании пользователя в процессе искусственной сессии автодополнения. Это также возможно при наличии логов о поведении пользователя. В этой работе были применены разные стратегии для минимизации описанной проблемы.

Одна из целей оценки качества автодополнения заключается в проверке изменений в алгоритме автодополнения. Такая проверка должна быть регулярной, чтобы было возможно вовремя обнаружить проблему, и она несильно сказалась на пользователях. В условиях распределенной разработки сложно обеспечить такую проверку в ручном формате запуска оценки качества. Для автоматизации этого процесса мы используем TeamCity — сервер непрерывной интеграции. С его помощью можно запускать оценку качества автоматически и регулярно, не пропустив ломающих изменений.

Наша работа сосредоточена на разработке инструмента, с помощью которого можно автоматически оценивать качество автодополнения.

Постановка задачи

Целью данной дипломной работы является создание инструмента для оценки качества функции автодополнения кода в средах разработки (IDEs) компании JetBrains. Для достижения этой цели были сформулированы следующие задачи.

1. Изучение существующих подходов к оценке качества автодополнения в IDE, основанных на искусственных запросах.

2. Реализация инструмента оценки качества автодополнения кода для различных языков программирования и сред разработки компании JetBrains.

3. Создание конфигураций для автоматической оценки качества текущей реализации автодополнения на сервере непрерывной интеграции TeamCity.

4. Формирование запросов автодополнения, приближенных к пользовательским запросам, с помощью моделирования пользователя.

1. Обзор

Автодополнение кода — одна из важнейших возможностей интерактивных сред разработки, поэтому исследования в этой области ведутся с момента возникновения инструментов для программистов и по сегодняшний день достаточно активно. Существуют различные подходы к реализации автодополнения. Рассмотрим некоторые из них.

Авторы статьи [5] предлагают алгоритм автодополнения для программного интерфейса приложения (API), основанный на статистике изменений абстрактного синтаксического дерева (AST) программы. Для построения статистической модели они используют большое число проектов и информацию из системы управления версий. Каждое изменение из системы управления версий разбивается на атомарные изменения путем сравнения AST двух версий кода. В результате вызова автодополнения формируются предложения, при использовании которых будет получено атомарное изменение, наиболее похожее на уже произведенные.

В работе [6] был введен язык частичных выражений для формирования запросов автодополнения. Проблема, которую решают авторы, заключается в использовании незнакомого API разработчиками. В такой ситуации программист не знает точного названия метода, но знает какие аргументы должен принимать метод, решающий нужную задачу. В данной системе программист может написать символ *?* вместо неизвестного названия и нужные аргументы. Среди предложений автодополнения будут только подходящие методы.

Авторы [7] основываются на предположении, что программы, которые пишут люди, в большинстве своем не очень сложные и используют повторяющиеся конструкции. Базируясь на этом, они вводят в процесс автодополнения статистические языковые модели. Для предсказания следующего токена модель использует n-граммы, то есть n предыдущих токенов, и определяет наиболее вероятные варианты продолжения на основе корпуса с примерами.

Статья [8] является продолжением этой идеи. В ней используются

семантические языковые модели. На замену обычным n-граммам пришли семантические: вместо конкретных имен методов и переменных в них используются зарезервированные лексемы, описывающие вид токена, а также сопутствующую информацию, например тип возвращаемого значения и типы аргументов для метода. Благодаря такому подходу, модель способна обнаруживать зависимости в коде, не основываясь на конкретных идентификаторах. К тому же это позволяет хранить меньшие объемы данных.

Группа авторов из Дармштадского Технического Университета в своей статье [2] решали задачу ранжирования существующих предложений автодополнения, а не формирования новых. Для этого они использовали объемную кодовую базу и информацию о вызовах различных методов в этой базе. В результате были получены три алгоритма. Первый из них учитывает частоту использования каждого из методов в выдаче автодополнения (более популярные занимают более высокие позиции). Второй алгоритм использует ассоциативные правила для поиска похожих контекстов на текущий. Последний подход является основным предметом описываемой статьи и базируется на алгоритме k-ближайших соседей. Примеры из кодовой базы представляются в виде бинарной матрицы контекста. Вызов автодополнения в такой системе приводит к поиску похожих использований в такой матрице за счет вычисления расстояний.

Схожую задачу решают и в команде интерактивной среды разработки Intellij IDEA. В качестве основного алгоритма в IDE используется автодополнение, основанное на различных эвристиках и информации о текущем контексте. В результате использования такого подхода необходимый элемент находится среди предложений в подавляющем большинстве случаев. Однако, этот алгоритм далеко не всегда предоставляет интересующий вариант в первых позициях выдачи. Поэтому для Intellij IDEA разрабатывается алгоритм финального ранжирования предложений автодополнения, основанный на методах машинного обучения. Необходимость автоматизированной оценки качества предложений данного алгоритма послужила одной из мотиваций для создания

инструмента, описывающегося в этой работе.

Любой метод оценки качества автодополнения основывается на вычислении метрик по выполненным запросам автодополнения, для которых известен верный результат [9, 10]. Серия запросов к одному токену называется *сессией* автодополнения. Рассмотрим некоторые из таких метрик.

Точность (precision) определяет отношение релевантных предложений к общему числу предложений.

$$Precision = \frac{recommendations\ made\ \cap\ relevant}{recommendations\ made}$$

Полнота (recall) определяет отношение релевантных предложений к общему числу предложений.

$$Recall = \frac{recommendations\ made\ \cap\ relevant}{recommendations\ requested}$$

F-мера (F-measure) — метрика, объеденяющая две предыдущие. Она определяется как взвешенное гармоническое среднее точности и полноты.

$$F\text{-}measure = \frac{2\ \cdot\ Pecision\ \cdot\ Recall}{Precision\ +\ Recall}$$

Среднеобратный ранг (Mean Reciprocal Rank) — метрика, учитывающая позицию (ранг) верного предложения в выдаче [11]. Определяется как среднее обратных рангов по всем запросам автодополнения (Q).

$$MRR = \frac{1}{|Q|}\sum_{i=1}^{|Q|}\frac{1}{rank_i}$$

E-saved — метрика, которая учитывает тот факт, что для каждого токена производится последовательная серия вызовов автодополнения с увеличением префикса. Эта метрика соответствует нормализованному числу символов, которых пользователю не придется вводить вручную

благодаря автодополнению кода [12]. Формула для запроса q:

$$E\text{-}saved(q) = \sum_{i=1}^{|q|} (1 - \frac{i}{|q|}) \sum_{j} P(S_{ij} = 1)$$

В этой формуле S_{ij} является бинарной переменной, которая соответствует тому, удовлетворила ли пользователя j-ое предложение на i-м префиксе.

Несмотря на то, что существует большое число подходов для реализации автодополнения кода, основных методов оценки качества таких алгоритмов всего два. Отдельных исследований по этому вопросу также немного [10, 1]. Тем не менее любой новый предложенный метод автодополнения предполагает проведение экспериментов и оценку качества этого алгоритма. Поэтому рассмотрим способы определения качества на примере представленных выше алгоритмов.

Первый подход к оценке качества основывается на пользовательских логах. В этом случае в качестве сессий используются информация о реальных вызовах автодополнения разработчиков, использующих алгоритм. Такой способ максимально приближен к проверке изначальной цели создания автодополнения — упрощения и сокращения времени разработки программистов. При этом правильным элементом в выдаче является тот, который был выбран пользователем. Недостаток этого метода в том, что он возможен, только если алгоритмом пользуется достаточное число независимых пользователей. По этой причине он не является популярным в научной среде. Кроме того, получение оценки качества таким способом занимает достаточно продолжительное время.

Другой подход использует автоматически сгенерированные запросы автодополнения путем удаления (полностью или частично) токена и вызова на его месте автодополнения. Правильным элементом для этой сессии является этот самый токен. Этот способ оценки качества занимает немного времени и не требует привлечения пользователей, поэтому в подавляющем большинстве статей о новых алгоритмах автодополнения используется именно он.

Главная проблема оценки качества с помощью искусственных за-

просов заключается в том, что такие запросы часто не соответствуют тому, как пользуются автодополнением пользователи. К этому выводу пришли в исследовании [1]. Авторы этой работы создали плагин для среды разработки Visual Studio и собирали информацию о реальных пользовательских запросах, а также формировали собственные искусственные запросы, используя разные существующие стратегии. В результате они получили статистически значимые различия в качестве автодополнения, вычисленного двумя этими способами. Одна из задач нашей работы заключается в проверке близости формируемых запросов с запросами пользователей.

Для достижения близости запросов автодополнения нашего инструмента и пользователей, мы используем моделирование пользовательского поведения. Эта техника популярна в среде поисковых систем. Например, в статье [13] предлагается использовать кликовые модели для моделирования действий пользователей. Идея в том, чтобы представить поведение человека, использующего поисковый движок, в виде графа состояний, где каждое ребро графа является действием пользователя (в данном случае кликом). Каждому ребру соответствует вероятность данного события. Такие вероятности вычисляются на основе имеющихся пользовательских логов. Для непосредственного моделирования производится серия случайных событий, по которым строится путь в графе. Этот путь представляет собой поисковую сессию пользователя. Похожая идея была применена и в нашей работе.

2. Плагин оценки качества автодополнения

В качестве инструмента для оценки качества автодополнения был разработан плагин для интерактивной среды разработки Intellij IDEA. Для запуска оценки качества необходимо выбрать файлы и директории, которые будут использованы для оценки, указать настройки формирования искусственных запросов и запустить выполнение.

2.1. Архитектура

Выполнение плагина разделяется на три этапа:

1. формирование действий;
2. исполнение действий;
3. создание отчетов.

Все конечные и промежуточные результаты выполнения сохраняются в рабочем пространстве — директории, генерируемой автоматически в начале выполнения. Каждый из этапов может быть запущен отдельно на существующем рабочем пространстве, при условии, что в нем содержатся результаты выполнения предыдущего этапа.

2.1.1. Формирование действий

Задача первого этапа состоит в том, чтобы по списку файлов и настройкам выполнения сгенерировать действия, которые будут исполнены интегрированной средой разработки на следующем этапе. Список используемых действий с необходимыми параметрами представлен в таблице 1.

Действие	Параметры
OPEN_FILE	filePath
MOVE_CARET	offset
DELETE_RANGE	begin, end
PRINT_TEXT	text
CALL_COMPLETION	completionType

Таблица 1: Используемые действия

Формально, этап состоит из двух обходов для выбранных файлов. В первом обходе представление файла с исходным кодом, используемое в платформе Intellij, конвертируется в унифицированное абстрактное синтаксическое дерево (UAST). Это вариант AST, в котором используются узлы, общие для различных языков программирования. Благодаря такому преобразованию, дальнейшее выполнение абстрагировано от языка программирования, поэтому поддержка новой IDE не является трудоемкой задачей.

Для простоты дальнейшего изложения введем понятие дополняемых токенов. Все токены программы можно разбить на два класса: токены, для которых пользователь, как правило, применяет автодополнение, и для которых нет. Примерами первого класса могут являться идентификаторы, которые уже встречались в программе, и ключевые слова. Среди токенов второго класса можно упомянуть комментарии, определение новых сущностей и т.п.. Итак, дополняемыми токенами будем называть токены, принадлежащие первому классу.

Для выполнения обхода используется паттерн программирования Посетитель. В плагине определены две базовые реализации посетителя, использование каждой из которых приводит к формированию одной из форм UAST:

- Дерево, включающее базовые определения (классы, методы), вызовы функций и обращения к переменным. Такое представление позволяет использовать фильтрацию токенов, к которым необхо-

димо применить автодополнение. Однако, оно включает не все дополняемые токены в файле. Например, в таком представлении отсутствуют ключевые слова и некоторые специфичные для языков программирования сущности.

- Список токенов. В этом случае конечное представление не будет содержать информацию о типе токена, поэтому не будет возможности фильтровать их при выполнении. Преимуществом же такого представления является полнота (включает все дополняемые токены) и простота реализации.

Таким образом для полной поддержки плагином языка программирования достаточно реализовать два посетителя: для преобразования файла в полноценный UAST и в список дополняемых токенов.

В результате второго обхода генерируются действия по полученному AST. В алгоритме 1 высокоуровнево представлены операции, которые выполняет IDE по сформированным действиям.

Algorithm 1 Алгоритм интерпретации действий

```
1:  procedure Interpret actions
2:      for file ∈ files do
3:          Open(file)
4:          uast ← getUast(file)
5:          for token ∈ uast do
6:              Delete(token)
7:              prefix ← getPrefix(token)
8:              Print(prefix)
9:              result ← CallCompletion(completionType)
10:             if token ∈ result then
11:                 Select(token)
12:             else
13:                 Print(token.without(prefix))
```

2.1.2. Исполнение действий

Второй этап включает исполнение (интерпретацию) сгенерированных действий в интегрированной среде разработки. Результатом вы-

полнения этапа являются сессии автодополнения. Каждая сессия соответствует некоторому токену и может содержать несколько запросов автодополнения. В запросе содержится информация о том, на каком префиксе было вызвано автодополнение, сколько времени продлилось выполнение запроса и какие предложение были предоставлены в результате.

В процессе исполнения действий также сохраняются логи автодополнения. Это подробная информация о месте вызова автодополнения и о каждом элементе в выдаче. Такие логи, собранные на пользовательских сессиях, используются для обучения модели автодополнения, основанного на машинном обучении (ML автодополнение). Благодаря возможности сбора логов с помощью плагина, разработчики ML автодополнения могут значительно сократить время получения достаточного числа логов для проверки различных гипотез.

Так как этап исполнения действий отделен от этапа их генерации, существует возможность использовать одни и те же действия для разных запусков интерпретации. Поэтому для того, чтобы применить плагин для нескольких алгоритмов автодополнения, можно единожды сформировать действия и несколько раз выполнить их интерпретацию.

2.1.3. Создание отчетов

На последнем этапе создаются отчеты качества. Для этого используются сессии, полученные в результате выполнения действий. По ним вычисляются метрики, которые представляются в основном отчете. Такой отчет содержит список файлов, использовавшихся для выполнения и значения метрик для каждого из них (рис. 1). Кроме того, отчет включает ссылки на отчеты для каждого из файлов. В таком отчете представлен исходный код файла, а каждый из токенов, на котором было вызвано автодополнение, выделен цветом (рис. 2). Конкретный цвет зависит от того, на какой позиции в выдаче находился исходный токен. Первой позиции соответствует зеленый цвет, со второй по пятую — желтый, ниже пятой — оранжевый и красный, если исходного токена не оказалось в выдаче. Саму выдачу можно увидеть по клику на токен.

3577 file(s) successfully processed

After | Metrics visibility | Redraw table | Show empty rows

File Report	Found@1 BASIC	Found@5 BASIC	Mean Rank BASIC	Recall BASIC	Sessions BASIC
Summary	**0.681**	**0.878**	**2.417**	**1.000**	**49213**
AfterSuiteEvent.java	1.000	1.000	0.000	1.000	1
GitPushAfterCommitD…	1.000	1.000	0.000	1.000	4
AfterTestEvent.java	0.800	1.000	0.467	1.000	15
ShowDiffAfterWithLo…	0.500	0.500	7.500	1.000	2
UpdateBreakpointsAft…	0.467	0.667	15.667	1.000	15

Page Size 25 | First | Prev | 1 | Next | Last

Рис. 1: Основной отчет о качестве автодополнения

```
9  export function parseCookieValue(cookieStr: string, name: string): string|null {
10   name = encodeURIComponent(name);
11   for (const cookie of cookieStr.split(';')) {
12     const eqIndex = cookie.indexOf('=');
13     const [cookieName, cookieValue]: string[] =
14         eqIndex == -1 ? [cookie, ''] : [cookie.slice(0, eq                    ex + 1)];
15     if (cookieName.trim() === name) {
16       return decodeURIComponent(cookieValue);
17     }
18   }
19   return null;
20 }
21
```

prefix: " st "; latency: 288

string

ast

Рис. 2: Отчет о качестве автодополнения для файла

Этап создания отчетов можно запустить отдельно, передав несколько рабочих пространств, для которых была проведена интерпретация. Например, это могут быть результаты выполнения одних и тех же действий для разных алгоритмов автодополнения. В результате этого, будет сформирован отчет сравнения. Этот отчет содержит метрики для каждого из типов автодополнения (рис. 3), так же как и отчеты по файлам содержат выдачи для каждого из алгоритмов. Каждый токен разделен сиволом $\int$, левая часть соответствует одному алгоритму автодополнения, правая — другому.

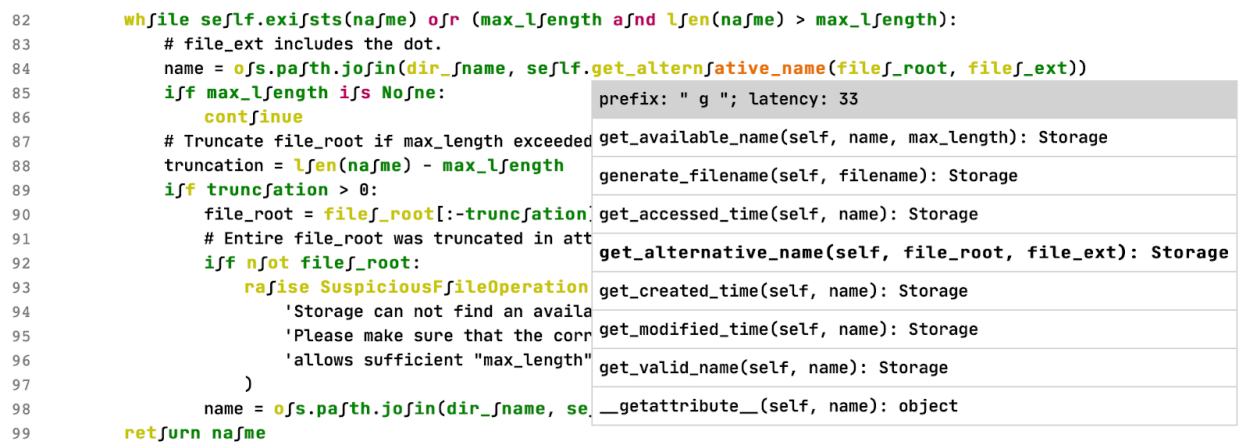

Рис. 3: Сравнительный отчет о качестве автодополнения для файла

2.2. Стратегии формирования запросов автодополнения

Для того, чтобы оценить качество с точки зрения различных моделей поведения пользователя, в плагине есть возможность варьировать некоторые характеристики формирования запросов автодополнения. На рисунке 4 представлено диалоговое окно с настройками выполнения. Рассмотрим параметры, которые оно содержит.

2.2.1. Контексты

Контекст дополняемого токена — одна из характеристик, которую можно изменять в выполнении плагина. Она определяет состояние, в котором находится файл перед вызовом автодополнения. В плагине представлены две реализации этого параметра:

- Предыдущий. Этот контекст содержит все тело метода, предшествующее дополняемому токену, но последующий код метода удаляется. Такой вариант моделирует ситуацию последовательной разработки программ.

- Полный. В этом контексте перед вызовом автодополнения удаляется только токен, участвующий в сессии. Такой подход соответствует работе с кодом, когда первоначальная версия програм-

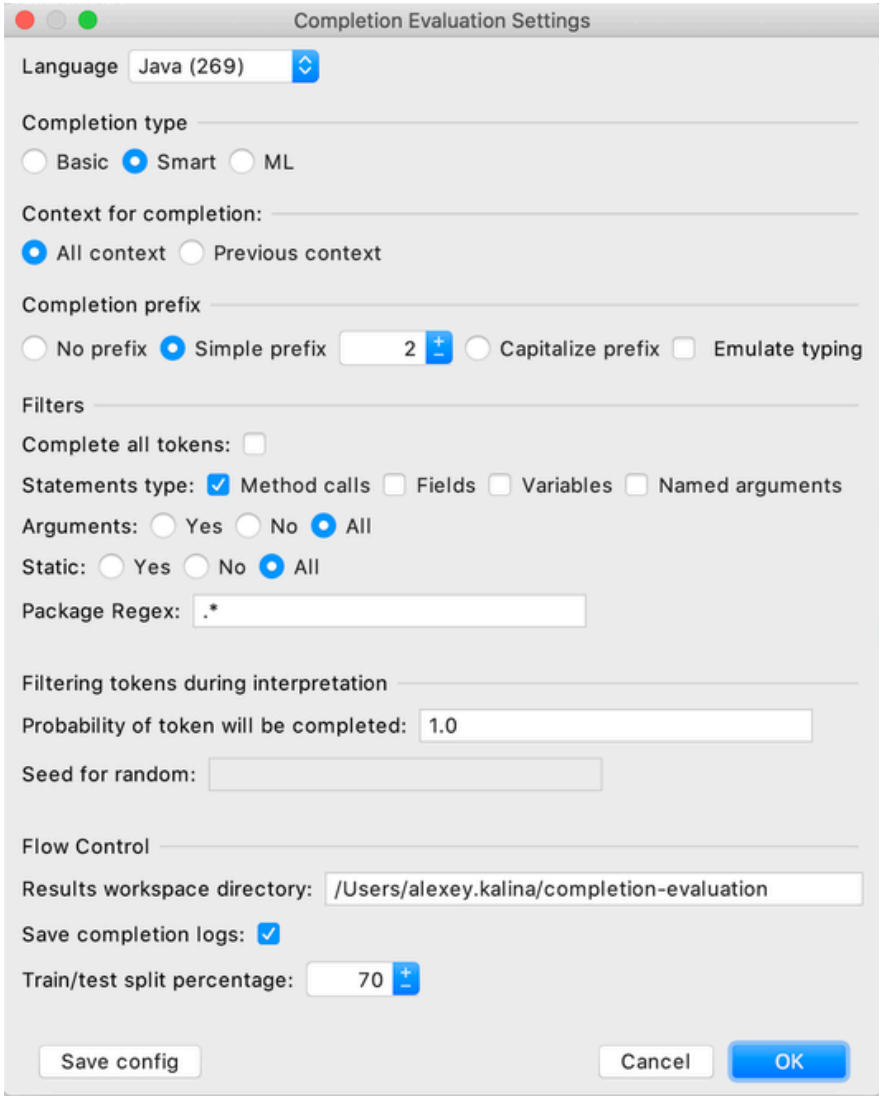

Рис. 4: Диалоговое окно с настройками выполнения

мы уже написана. Существуют исследования, показывающие, что большую часть работы программисты проводят именно в таком режиме.

2.2.2. Префиксы

Перед вызовом автодополнения разработчик может напечатать часть токена, чтобы сократить число возможных предложений. В плагине этому поведению соответствует настройка префиксов. В стратегии формирования запросов используется один из трех вариантов префиксов:

- Простой. Префиксом является подстрока токена фиксированной

длины n.

- Пустой. Частный случай простого префикса с n = 0.
- Капитализированный. Такой префикс включает первый символ и все заглавные буквы токена. Использование капитализированного префикса моделирует использование автодополнения для идентификаторов, состоящих из нескольких слов, написанных в верлюжьем или паскалевском начертании.

Также в плагине есть возможность моделировать написание префикса. В таком случае автодополнение вызывается последовательно с добавлением каждого из символов префикса. При этом появление верного предложения в выдаче приводит к выбору этого варианта и прекращению сессии. Эта опция доступна для простого и капитализированного префиксов. В разделе *Моделирование пользовательского поведения* описан еще один подход к формированию нескольких запросов автодополнения в рамках одной сессии.

2.2.3. Фильтры

Фильтрация токенов доступна для тех языков программирования, для которых используется представление файлов в формате полноценного унифицированного AST. На данный момент такими языками являются Java, Kotlin и Python. Фильтрация позволяет оценивать качество автодополнения только для конкретных групп токенов. В плагине поддержаны следующие виды фильтров и их значения:

- Тип токена: вызов метода, именованный параметр, обращение к переменной или полю.
- Статический член: да или нет.
- Аргумент метода: да или нет.
- Регулярное выражение для имени пакета, которому принадлежит данный член.

Таким образом, можно формировать сложные группы токенов, например: статические методы из стандартной библиотеки. С помощью фильтрации возможно решение проблемы поиска проблемных мест в алгоритме автодополнения. Общий подход заключается в сравнении результатов выполнения для одного и того же проекта с разными фильтрами.

Использование фильтрации доступно как на этапе исполнения действий, так и на этапе генерации отчетов. Применение фильтров во время исполнения действий позволяет сократить время выполнения. Другим способом уменьшения времени выполнения через сокращение числа используемых токенов является опция Вероятность использования токена. На этапе исполнения действий решение о том, использовать или нет следующий токен, принимается с учетом данной опции. В свою очередь фильтрация на этапе формирования отчетов позволяет единожды провести исполнение всех действий и строить отчеты качества для разных подгрупп токенов.

2.3. Метрики

В плагине используются метрики, наглядно демонстрирующие разницу между разными алгоритмами автодополнения. Важным критерием выбора метрик была их интерпретируемость. Перечислим метрики, которые вычисляются в выполнении:

- *Top-1*, *Top-k* — доля сессий, для которых правильный элемент находился в первой/k-ой позиции в выдаче.
- *Полнота* — доля сессий, для которых правильный элемент был в выдаче.
- *Средний ранг* — средняя позиция правильного элемента по всем сессиям, для которых правильный элемент был в выдаче.
- *Средняя и максимальная задержка* — статистики для времени выполнения запроса автодополнения по всем сессиям.

3. Автоматизация оценки качества автодополнения

Есть несколько причин, из-за которых автоматизация процесса оценки качества автодополнения крайне актуальна. Во-первых, изменения в алгоритмах автодополнения производятся регулярно разработчиками из разных команд. Формирование рабочего процесса, при котором все эти программисты должны запускать инструмент оценки качества, привело бы к значительному снижению эффективности в командах. Вторая причина заключается в том, что процесс оценки качества может быть достаточно длительным и занимать несколько часов. Регулярное проведение таких оценок на локальных устройствах программистов, также отрицательно скажется на их продуктивности. Для решения этих проблем было решено воспользоваться сервером непрерывной интеграции.

В качества сервера непрерывной интеграции используется Team City. Это продукт компании JetBrains, позволяющий запускать вычислительные процессы на удаленных агентах. Единицей запуска в Team City является конфигурация. Конфигурация представляет из себя последовательность шагов, каждый из которых является отдельным процессом. Он может быть описан как в виде команды в командной строке, так и в виде команды конкретной системы сборки. В свою очередь конфигурации объединяются в проекты по общему назначению. Важной особенностью работы с конфигурациями Team City является то, что результаты выполнения одной конфигурации могут быть использованы в других.

Для автоматизации использования плагина оценки качества были созданы проекты для каждого из поддерживаемых языков с конфигурациями нескольких типов:

- для оценки качества автодополнения;
- для сравнения нескольких алгоритмов автодополнения;

- для построения ML моделей на основе искусственных логов.

3.1. Оценка качества автодополнения

Первая группа конфигураций используется для оценки качества конкретного алгоритма автодополнения. Результатом работы конфигурации являются отчеты качества и список сессий автодополнения. Высокоуровневый список шагов конфигурации выглядит так:

1. Скачать последнюю версию соответствующей языку интерактивной среды разработки и установить в нее плагин оценки качества автодополнения.

2. Скачать проект с открытым исходным кодом, написанный на соответствующем языке программирования.

3. Обновить параметр в подготовленном конфигурационном файле, отвечающий за тип автодополнения, используемый в выполнении.

4. Запустить выполнение плагина с обновленным конфигурационным файлом, включающее все три этапа.

5. Опубликовать полученный отчет качества и сессии автодополнения.

3.2. Сравнение алгоритмов автодополнения

Вторая группа конфигураций предназначена для формирования отчетов сравнения для нескольких алгоритмов автодополнения. В ней используются результаты выполнения других конфигураций, а именно сформированные сессии. По этим сессиям строится и публикуется отчет. Список шагов конфигурации:

1. Скачать последнюю версию соответствующей языку интерактивной среды разработки и установить в нее плагин оценки качества автодополнения.

2. Скачать результаты выполнения других конфигураций.

3. Запустить выполнение плагина, включающее построение отчета на основе нескольких рабочих пространств.

4. Опубликовать полученный отчет качества.

3.3. Построение моделей машинного обучения на основе искусственных запросов

Последняя группа конфигураций используется для построения моделей ML автодополнения. Такие конфигурации генерируют логи автодополнения в процессе выполнения плагина. Эти логи публикуются как результат работы конфигурации. Они используются конфигурациями проекта, отвечающего за ML автодополнение. Шаги выполнения конфигурации:

1. Скачать последнюю версию соответствующей языку интерактивной среды разработки и установить в нее плагин оценки качества автодополнения.

2. Скачать проект с открытым исходным кодом, написанный на соответствующем языке программирования.

3. Обновить параметр в подготовленном конфигурационном файле, отвечающий за тип автодополнения, используемый в выполнении.

4. Включить формирование логов в конфигурационном файле.

5. Запустить выполнение плагина с обновленным конфигурационным файлом, включающее генерацию и выполнение действий.

6. Опубликовать логи автодополнения.

4. Моделирование поведения пользователя

Сбор логов автодополнения в процессе выполнения плагина позволяет создавать модели ML автодополнения, обученные на таких искусственных логах. Это позволяет разработчикам ML автодополнения значительно сократить цикл получения модели, включающей новую информацию об автодополнении. Однако, эксперименты показали, что такие модели уступают по качеству моделям, обученным на реальных пользовательских логах. Это связано с тем, что выполнение в плагине ведется по одному шаблону, что мало соответствует реальной разработке. По этой причине было решено добавить в плагин возможность моделирования пользователя. Это также должно повысить правдоподобность получаемых оценок качества.

4.1. Вероятностная модель поведения пользователя

В качестве модели пользовательского поведения было решено использовать аналог кликовых моделей, применяющихся в области поисковых систем. Идея состоит в том, чтобы принимать решение о следующем действии в выполнении плагина на основе вероятностей. Эти вероятности рассчитываются по логам, собранным с пользователей IDE, тем самым искусственное выполнение становится более соответствующим реальной разработке.

По логам пользователей были рассчитаны три группы вероятностей:

- Вероятность начать сессию с i-го префикса ($P_{prefix}(i)$). Первый параметр, который отличается в поведении пользователя и плагина, это начальный префикс сессии. Статистика показывает, что сессии значительно реже начинаются с нулевого префикса нежели с единичного. Также присутствуют и более длинные начальные префиксы. Это означает, что пользователи успевают напечатать часть токена до того, как отработает первый запрос автодополнения.

- Вероятность выбрать k-ый элемент на j-ом запросе ($P_{select}(k,\ j)$). В базовом варианте выполнения правильный элемент выбирается всегда, если он содержится в выдаче. Пользовательские логи же показывают, что в реальности элемент крайне редко бывает выбран, если он находится ниже 5-ой позиции в выдаче. Это говорит о том, что разработчик предпочитает продолжить печать токена, чтобы сократить число результатов, нежели искать нужный элемент в нижней части выдачи. Кроме того, вероятность выбора элемента на некоторой позиции зависит от того, какой это по счету запрос в сессии. Например, после нескольких запросов вероятность выбора первого элемента выдачи повышается.

- Вероятность завершить сессию на j-ом запросе ($P_{cancel}(j)$). В реальной разработке всегда есть вероятность того, что пользователь предпочтет самостоятельно напечатать токен, особенно если в верхней части выдачи нет необходимых элементов. С увеличением числа запросов такая вероятность повышается.

Процедура моделирования пользователя с использованием описанных вероятностей представлена в алгоритме 2. Поясним предложенный псевдокод. Описываемая функция получает на вход токен, печать которого необходимо промоделировать. Вызов метода $choosePrefix$ выбирает некоторый префикс токена с учетом вероятности P_{prefix}. Этот префикс печатается в IDE. Далее повторяем для каждого символа оставшейся части токена следующие действия: вызываем автодополнение и пробуем выбрать исходный токен в выдаче с учетом вероятности P_{select}. В случае успеха заканчиваем алгоритм. Иначе пробуем завершить сессию с учетом вероятности P_{cancel}. Если сессия не завершилась, печатаем символ и повторяем цикл.

4.2. Эксперименты

Для проверки похожести моделей, полученных с помощью пользовательских и искусственных логов была проведена следующая процедура:

Algorithm 2 Алгоритм моделирования пользователя

```
procedure User modeling(token)
    prefix ← choosePrefix(P_prefix, token)
    Print(prefix)
    queryOrder ← 0
    for char ∈ token.without(prefix) do
        result ← CallCompletion()
        if token ∈ result then
            position ← result.indexOf(token)
            if try(P_select(position, queryOrder)) then
                selectElement(token)
                break
        if try(P_cancel(queryOrder)) then
            cancelSession()
            break
        Print(char)
        queryOrder += 1
```

1. Пользовательские логи разделяются на две группы.

2. На первой группе логов обучается эталонная модель ML автодополнения.

3. На второй группе логов рассчитываются метрики с использованием полученной модели.

4. Производится выполнение плагина с формированием искусственных логов.

5. Обучается модель ML автодополнения на полученных логах.

6. Рассчитываются метрики для полученной модели на второй группе пользовательских логов.

7. Если значения метрики Топ-1 для новой и эталонной моделей отличаются более чем на пороговое значение, то внести изменения в реализацию или параметры выполнения и перейти к шагу 4. Порогом для разницы между значениями метрик было решено взять 0.01.

	user	default	no user emulation	user emulation
Топ-1, %	66.0	63.7	65.2	65.9
Топ-5, %	86.9	83.9	86.0	86.2
Средний ранг	2.780	2.780	2.740	2.781

Таблица 2: Результаты экспериментов

Данная процедура привела к трем разным моделям, обученным на искусственных логах. Значения метрик представлены в таблице 2. Модель *user* обучена на реальных пользовательских логах. В первом выполнении использовались запросы только к методам, полям и переменным. При этом моделирования пользователя не проводилось. В таблице соответствующая модель называется *default*. При втором выполнении были использованы запросы ко всем дополняемым токенам, но все еще без эмуляции пользователя. В таблице этому выполнению соответствует столбец *no user emulation*. Рост значений метрик говорит о том, что нельзя пренебрегать ключевыми словами и другие элементами, которые не использовались в первом выполнении. В последнем выполнении, которому в таблице соответствует значение *user emulation*, было применено моделирование пользователя. Полученные значения метрик говорят о том, что наиболее похожие на пользовательские запросы и наиболее точные оценки автодополнения получаются с использованием описанной техники моделирования пользователя.

Заключение

В ходе данной работы были получены следующие результаты:

- Изучены способы оценки качества автодополнения в статьях, исследующих эту область.
- Реализован плагин оценки качества автодополнения для сред разработки компании JetBrains.
- На данный момент плагин может работать с проектами, написанными на Java, Python, Kotlin, Ruby, Scala, PHP, JavaScript, TypeScript, Go и C++. Большая часть работы плагина абстрагирована от языка программирования, поэтому поддержка новых языков не является сложной задачей.
- На сервере TeamCity созданы конфигурации для каждого из языков программирования с оценкой качества разных алгоритмов автодополнения и сравнения их между собой.
- Реализовано моделирование пользователя при формировании искусственных запросов автодополнения.

Список литературы

[1] Sebastian Proksch, Sven Amann, Sarah Nadi, and Mira Mezini. Evaluating the evaluations of code recommender systems: A reality check. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 111–121. ACM, 2016.

[2] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[3] Veselin Raychev, Martin Vechev, and Eran Yahav. Code completion with statistical language models. In *Acm Sigplan Notices*, volume 49, pages 419–428. ACM, 2014.

[4] Vincent J Hellendoorn, Sebastian Proksch, Harald C Gall, Alberto Bacchelli, C Bird, ET Barr, and M Allamanis. When code completion fails: a case study on real-world completions. 2019.

[5] Romain Robbes and Michele Lanza. Improving code completion with program history. *Automated Software Engineering*, 17(2):181–212, 2010.

[6] Daniel Perelman, Sumit Gulwani, Thomas Ball, and Dan Grossman. Type-directed completion of partial expressions. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, pages 275–286, 2012.

[7] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *2012 34th International Conference on Software Engineering (ICSE)*, pages 837–847. IEEE, 2012.

[8] Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. A statistical semantic language model for source

code. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 532–542, 2013.

[9] Christopher D Manning, Prabhakar Raghavan, and H Schütze. Chapter 8: Evaluation in information retrieval. *Introduction to information retrieval*, pages 151–175, 2008.

[10] Fei Cai, Maarten De Rijke, et al. A survey of query auto completion in information retrieval. *Foundations and Trends® in Information Retrieval*, 10(4):273–363, 2016.

[11] Nick Craswell. Mean reciprocal rank. *Encyclopedia of Database Systems*, pages 1703–1703, 2009.

[12] Eugene Kharitonov, Craig Macdonald, Pavel Serdyukov, and Iadh Ounis. User model-based metrics for offline query suggestion evaluation. In *Proceedings of the 36th international ACM SIGIR conference on Research and development in information retrieval*, pages 633–642. ACM, 2013.

[13] Ahmed Hassan, Rosie Jones, and Kristina Lisa Klinkner. Beyond dcg: user behavior as a predictor of a successful search. In *Proceedings of the third ACM international conference on Web search and data mining*, pages 221–230, 2010.