Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Тучина Анастасия Игоревна

Система оценки качества инструментов автодополнения кода в IntelliJ IDEA

Выпускная квалификационная работа бакалавра

Научный руководитель:
доцент, к. т. н. Т. А. Брыксин

Рецензент:
программист ООО “Интеллиджей Лабс” В. И. Бибаев

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Software and administration of information systems
Software Engineering

Anastasia Tuchina

Quality benchmark for code completion tools in IntelliJ IDEA

Bachelor's Thesis

Scientific supervisor:
Candidate of Engineering Sciences Timofey Bryksin

Reviewer:
software engineer, IntelliJ Labs Co. Ltd. Vitaly Bibaev

Saint-Petersburg
2020

Оглавление

Введение

Автодополнение кода — один из наиболее важных и широко используемых разработчиками инструментов, включенный в большинство современных сред разработки. Анализ поведения пользователей внутри интегрированных сред разработки (IDE) показывает, что автодополнение кода находится в числе самых часто используемых функций, предоставляемых средой [19]. Качественное автодополнение повышает продуктивность работы, помогает избежать опечаток и части других ошибок в процессе написания кода и может быть полезным при исследовании методов внешних API [20], поэтому создание все более совершенных инструментов было и остается актуальной исследовательской задачей.

Обычно инструменты автодополнения предоставляют контекстную помощь, показывая пользователю во всплывающем окне ранжированный список элементов, которыми можно дополнить текущий фрагмент кода. Функция автодополнения может вызываться по команде пользователя или срабатывать автоматически, например, при частичном вводе токена.

Несмотря на разнообразие современных инструментов и алгоритмов автодополнения кода, в большинстве случаев первоначальная оценка их эффективности производится на синтетически сгенерированных данных [2, 9, 11], то есть посредством удаления случайного элемента из кода и дальнейшими попытками корректно предсказать его с учётом контекста. Тем не менее, как показывают последние исследования, сгенерированные таким образом синтетические данные не всегда отражают нужды пользователей [24] и текущий рабочий контекст [8]. А это уже будет оказывать значительное влияние на реальную эффективность инструмента. Вследствие этого возникает необходимость проводить оценку инструментов автодополнения кода на реальных данных или данных, соответствующих характеристикам реальных. Кроме того, полезной была бы возможность оценивать качество автодополнения непосредственно в IDE, в которой инструмент будет использоваться.

Постановка задачи

Целью данной работы является разработка системы для интеграции инструментов автодополнения кода с IDE на основе IntelliJ Platform [23] и оценки их качества. Для достижения поставленной цели были сформулированы следующие задачи:

- проанализировать существующие способы оценки качества алгоритмов автодополнения кода;
- разработать инструментарий, позволяющий интегрировать инструменты автодополнения кода с IDE на основе IntelliJ Platform;
- собрать датасет на основе данных об использовании инструментов автодополнения;
- разработать веб-сервис, позволяющий оценивать качество инструментов автодополнения;
- провести апробацию разработанных инструментов на нескольких существующих моделях автодополнения кода.

1. Обзор

1.1. Роль пользовательских данных в разработке программных инструментов

За последние годы был опубликован ряд исследований, целями которых являются сбор и анализ данных об использовании функций интегрированных сред разработки [19], в том числе и автодополнения кода [8, 10], а также поиск возможных способов повышения эффективности этих инструментов на основе полученных данных [16, 24].

1.1.1. Анализ пользовательского поведения в IDE

Часть исследований фокусируются на анализе поведения пользователей в интегрированных средах разработки, сборе и интерпретации данных об использовании функций IDE [19, 7, 10].

Общеизвестным способом анализа статистики использования программных продуктов является сбор логов их использования, но таких данных не всегда достаточно для исследовательских целей. Поэтому авторы статьи [19] разработали инструмент, отслеживающий действия пользователей в IDE и собрали датасет, содержащий записанные данные событий внутри среды (редактирование кода, использование предоставленных средой функций, тестирование и т. д.). Их исследование также направлено и на понимание того, каким образом разработчики используют IDE. В частности, например, автодополнение кода указано как наиболее часто используемая функция IDE наряду с инструментами сборки и автоматическими исправлениями.

Развитие этой идеи описано в работе [7]. Кроме самих событий IDE сохраняются сопутствующие им состояния открытого в редакторе файла, представленные в виде синтаксических деревьев, обеспечивающие воспроизводимость записанных событий. Так как существует возможность воспроизвести события в датасете, он может быть переиспользован, поэтому авторы описывают несколько сценариев, при которых он может быть полезен. Например, при создании анализаторов кода или

проведения экспериментов для оценки качества некоторых программных инструментов.

Работа [10] описывает проведенное на этом датасете исследование применения автодополнения кода, в котором выделены несколько характеристик, присущих только реальным данным, но влияющих на то, будет ли полезен инструмент пользователю. В частности, это исследование показывает, что на применение пользователем автодополнения влияет не только непосредственно наличие в списке выдачи нужного элемента, но и длина самого списка.

Такие исследования не только помогают понять, как разработчики в действительности пользуются средами разработки, но и выделяют полезные характеристики собранных данных, которые нужно учитывать при создании новых программных инструментов.

1.1.2. Улучшение и оценка существующих инструментов на основе пользовательских данных

В статье [16] показано, что использование пользовательской истории изменений кода помогает улучшить существующие алгоритмы автодополнения. В частности, пользователи более склонны обращаться к идентификаторам из недавно измененных фрагментов кода.

Авторы одной из недавних работ [24] провели обширное исследование на реальных пользователях, проанализировав статистику применения встроенного инструмента автодополнения в процессе написания кода в IDE, и сравнили эти данные с синтетическими датасетами для оценки качества автодополнения. Они запустили генерацию синтетических бенчмарков на крупном датасете на C# [5] и собрали статистику по типам полученных данных, получив соотношения, указанные в таблице 1.

Тогда как в пользовательских данных представлены в основном только автодополнения имен идентификаторов, в синтетическом датасете они составляют только примерно треть от общего количества данных. Другой значимый недостаток представленных в искусственном датасете автодополнений состоит в том, что невостребованные пользова-

Тип	Процент
Пунктуация	57,1%
Идентификатор	32.1%
Ключевое слово	10.8%
Числовой литерал	0.5%

Таблица 1: Распределение типов токенов, заимствовано из работы [24]

телями дополнения в виде числовых литералов и знаков пунктуации составляют более половины. Более того, автодополнение токенов таких типов даже не присутствует в современных IDE.

Затем авторы выбрали и обучили несколько существующих моделей автодополнения кода на этом же датасете, оценивая их эффективность на пользовательских данных. Полученные значения top-n метрик не превышают 64%, тогда как при оценке на синтетических данных заявленные метрики в основном составляют 80% и более.

Проведенные в этой работе эксперименты показывают, что стандартные синтетические датасеты зачастую не отражают нужды пользователей и то, что демонстрирующий хорошие результаты на синтетических данных инструмент не всегда может быть действительно эффективен.

1.1.3. Выводы

На основе рассмотренных выше работ можно сделать вывод, что часто при создании программных инструментов и оценке разработанных алгоритмов не учитываются важные характеристики пользовательских данных (кода), что приводит к снижению точности алгоритма при использовании его в реальной жизни или неверной оценке качества инструмента. В то время, как авторы некоторых инструментов имеют возможность собирать пользовательскую статистику и оценивать с ее помощью реальную эффективность своих алгоритмов [15], проблема первичной оценки точности многих инструментов остается открытой.

1.2. Современные методы автодополнения кода

Исследования в области автодополнения кода ведутся с момента возникновения интегрированных сред разработки, причем используются как стандартные языковые модели, так и разрабатываются новые алгоритмы, предназначенные только для моделирования программного кода.

Разные алгоритмы предлагают автодополнения для фрагментов кода различного размера, например, в виде строк кода в TabNine[1] или отдельных программных инструкций [4]. Но так как большая часть алгоритмов автодополнения предназначена для генерации рекомендаций следующего элемента в коде (вызова метода, использования переменной, ключевых слов и т. д.), в этой работе будем рассматривать именно такие подходы.

1.2.1. Стандартные языковые модели

При решении задачи автодополнения кода могут использоваться стандартные языковые модели, применяемые также и при моделировании естественного языка. Это становится возможным по причине схожих характеристик кода и естественного языка, например, наличия повторяющихся конструкций и некоторой предсказуемости в текущем контексте [12]. Такие модели рассматривают код как последовательность токенов и при автодополнении предсказывают следующий токен/набор токенов в последовательности. В частности, активно применяются n-граммы [22, 9] и различные модификации рекуррентных нейронных сетей [21, 11].

Программный код, несмотря на некоторую его схожесть с текстами на естественном языке, имеет и свои отличительные особенности. Одна из них это локализованность [22], то есть наличие в текущем контексте (открытом проекте или файле) специфичной только для него информации, например, имен локальных переменных и методов. В работах [22, 9] предлагается использовать заранее обученную на большом

[1]https://www.tabnine.com

корпусе классическую n-грамм модель в комбинации с локальными n-грамм моделями, адаптируя тем самым модель под текущий рабочий контекст. Для этого алгоритм строит отдельную модель для каждого файла в заданном проекте и сохраняет информацию о встреченных в них последовательностях токенов, формируя таким образом кэш и моделируя локальный контекст.

Авторы [21] впервые предложили использовать рекуррентные нейронные сети для моделирования кода таким же образом, как это делается в задачах моделирования естественного языка. Выбор такого подхода обоснован предоставленной архитектурой этих сетей возможностью обрабатывать последовательности переменной длины и некоторой схожестью кода с текстами на естественном языке.

Тем не менее, при использовании моделей глубокого обучения часто возникает проблема ограниченности словаря идентификаторов, которыми может оперировать данная модель. Для ее минимизации были разработаны несколько решений. В работе [2] было предложено использование нейронных сетей указателей (pointer networks); такая сеть в случае невозможности выбрать элемент из словаря может указать элемент из локального рабочего контекста, которым можно дополнить текущий фрагмент кода. Авторы статьи [11] предложили другой способ решения этой проблемы и разработали модель с открытым словарем, храня в нем не идентификаторы, а фрагменты токенов, которые модель будет предсказывать и составлять из них идентификаторы для выдачи. Такой подход не только значительно расширяет набор доступных модели для предсказания идентификаторов, словарь которых в программном коде, в отличие от естественного языка, не ограничен, но и уменьшает размер сохраняемого словаря.

1.2.2. Моделирование программного кода для задач автодополнения

Программный код помимо самого текста программы содержит также и дополнительную семантическую и синтаксическую информацию, которую упускают стандартные языковые модели, но которая может

помочь при проектировании эффективных алгоритмов автодополнения. Такая информация представлена, например, историей изменений кода, типами переменных и параметров функций, типами возвращаемых функциями значений, окружением в абстрактном синтаксическом дереве кода (AST) и т. д.

На статистике изменений AST кода основан алгоритм автодополнения вызовов API, представленный в статье [16]. При помощи разработанного ими ранее инструмента SpyWare [17] авторы собрали датасет, содержащий события изменения кода реальными пользователями и записанные AST изменений. Они предлагают использовать историю изменений AST программного кода, сравнивая записанные версии дерева. Записанные состояния AST последовательно сравниваются с текущим, а затем на основе результатов этих сравнений составляется список возможных автодополнений. Таким образом, в ранжированном списке выдачи выше оказываются идентификаторы, соответствующие недавно модифицированным программным сущностям и использованные при модификации имена.

Идею фильтрации и ранжирования уже существующего составленного IDE стандартного списка возможных автодополнений предлагают авторы работы [1]. Основной подход к разработке алгоритмов автодополнения, описанный в статье, заключается в обучении моделей на большом количестве репозиториев, которые разбиваются на небольшие фрагменты кода, представляющие локальный контекст, и извлечении из них информации об использованных вызовах методов.

На основе предложенного подхода были разработаны три модели, интерпретирующих эту информацию разными способами.

- **Частота использования методов.** Данный алгоритм учитывает, как часто встречается метод в похожем контексте в использованных во время обучения данных. Чем чаще встречался определенный метод в обучающих данных, тем выше он окажется в финальном списке выдачи.

- **Использование ассоциативных правил.** Для поиска метода,

использованного в контексте, похожем на текущий, применяются ассоциативные правила – поиск логических закономерностей и зависимостей в наборе данных. В данном случае это значит, что если, например, вызов метода достаточно часто встречался после вызова какого-то набора методов, вероятно, что он попадет и в выдачу автодополнения при использовании такого же набора методов.

- **Метод k-ближайших соседей.** Для поиска метода, использованного в контексте, похожем на текущий, используется модификация метода k-ближайших соседей (Best Matching Neighbour). Контекст представляется в виде бинарной матрицы, элементы в которой отражают наличие или отсутствие конкретного признака. При вызове автодополнения алгоритм составляет такую матрицу и ищет похожие использования через расстояние между текущим вектором контекста и элементами в матрицах, использованных во время обучения. Встреченные во время обучения контекстные векторы ранжируются по расстоянию между ними и текущим вектором контекста. Чем ближе текущий вектор контекста к одному из встреченных во время время обучения контекстных векторов, тем выше соотвествующий этому вектору элемент окажется в финальном списке выдачи.

Идею алгоритма Best Matching Neighbour и представления локального контекста в виде бинарных матриц затем обобщили до байесовских сетей [14], получив сравнимую точность, но уменьшив размер модели и увеличив скорость работы алгоритма.

Для разработки алгоритмов автодополнения кода используются также различные модификации и расширения стандартных языковых моделей с помощью дополнительной информации, доступной для него. Так, в статье [18] предлагается расширение стандартной n-грамм модели с помощью семантической информации. Каждый элемент в такой модели вместо текстового представления токена содержит лексему, описывающую его вид, и дополнительную информацию, например,

о типе переменной или возвращаемого значений для функции. Описанный подход позволяет не только уменьшить размер модели за счет сокращения словаря, но и обнаруживать зависимости в коде, не привязывая их напрямую к идентификаторам.

Расширение IntelliCode[2] для Visual Studio Code, предоставляющее автодополнение кода на разных языках, также использует алгоритмы машинного обучения. Частью этого расширения является система Pythia [15] — инструмент автодополнения для Python. В ней задача нахождения зависимостей в коде решается использованием LSTM сети, которая обучается на последовательностях узлов AST, составленных по фрагментам кода из датасета. Фрагменты кода, включающие в себя вызовы методов и обращения к членам классов, представляются в виде последовательного набора N токенов и узлов AST с дополнительной метаинформацией и доступной информацией об их типах, которые отображаются в численные векторы и объединяются в матрицу. По этим представлениям обучаются векторные представления меньшей размерности с помощью Word2Vec [6]. На основе полученной матрицы делаются предсказания кода.

Идею обучения нейронных сетей на AST представлениях вместо стандартной токенизации кода применяют и авторы работы [3]. Transformer превосходит RNN в задаче предсказания следующего токена в коде, поэтому, исходя из предположения, что использование дополнительной информации из кода увеличит точность предсказаний, авторы построили несколько моделей на основе GPT-2 — нейронной сети, используемой изначально для генерации текстов:

- SrcSeq – стандартная модель GPT-2, принимающая на вход последовательность извлеченных из кода токенов;
- DFS – модель, использующая в качестве входной последовательности сериализованные в порядке обхода в глубину AST вместо обычных последовательностей токенов;

[2]https://visualstudio.microsoft.com/ru/services/intellicode/

- DFSud – модификация модели DFS, дополняющая входную последовательность узлов кратчайшими расстояниями между узлами в дереве.

1.2.3. Выводы

Прямое сравнение нескольких алгоритмов автодополнения в том виде, в каком они представлены в статьях, не всегда возможно, так как в этих исследованиях обычно применяются различающиеся наборы метрик и датасеты, на которых проводятся эксперименты. Тем не менее, большинство инструментов предоставляют автодополнение именно для следующего токена в коде, поэтому для них можно использовать один общий набор метрик и, если они предназначены для одного языка, один датасет.

1.3. Методы оценки качества автодополнения кода

Несмотря на многообразие алгоритмов автодополнения кода, существует всего два основных подхода к оценке их качества:

- оценка на синтетически сгенерированных запросах [2, 9, 11, 3];
- оценка на пользовательской истории применения автодополнений [24, 16].

1.3.1. Стандартные методы оценки качества автодополнения кода

Современные алгоритмы автодополнения кода в основном оцениваются на синтетических данных [2, 9, 11, 3], которые могут генерироваться с учетом специфики инструмента (например, типа токенов, которые может дополнить конкретный инструмент). Такие датасеты обычно создаются с помощью генераторов, удаляющих случайный токен (частично или полностью), соответствующий требованиям, из кода, который затем инструмент пытается предсказать.

В некоторых работах [16, 8] эксперименты предлагается проводить на данных о записанных и воспроизводимых действиях пользователей в IDE, но выборки людей в таких экспериментах достаточно малы (40-70 человек).

Как правило, в каждой работе предлагаются свои метрики, датасеты и сценарии экспериментов для оценки качества разработанного алгоритма, что делает невозможным прямое сравнение алгоритмов друг с другом или вызывает проблемы с воспроизводимостью результатов. Это создает необходимость иметь систему для унифицированной оценки инструментов автодополнения с фиксированным датасетом и набором экспериментов [13].

Стандартный метод с использованием синтетических запросов является менее точным по сравнению с идеей применять для этих целей пользовательские истории, так как часть запросов, которые генерируются таким образом, никогда не применяются пользователями [8, 24]. С другой стороны, составить достаточно большой датасет из реальных данных непросто. Собранные ранее датасеты [19, 16, 8], содержащие пользовательские данные о выборе и применении автодополнений, записаны с помощью отдельно разработанных инструментов, отслеживающих действия разработчика в IDE, и предназначены скорее для исследовательских целей по причине небольшой выборки людей и малых объемов данных. Собрать сравнительно большое количество подобных данных таким же образом, как это, например, делается при сборе пользовательской статистики, невозможно ввиду регламентов защиты персональных данных пользователей. Остаётся возможность подобрать набор данных, приближенный к реальным по метрикам и присущим этим данным характеристикам. Такой подход также применяется в нескольких опубликованных работах [24, 10].

1.3.2. Метрики для оценки качества автодополнения кода

Для оценки качества автодополнения часто применяют метрики, учитывающие каким-либо образом позицию нужного элемента в выдаче. Обычно используют следующие метрики.

- Top-n. Метрика Top-n отражает долю запросов, в которых нужный элемент находится не ниже, чем на n позиции в списке выдачи. Чем меньше значение метрики, тем меньшее количество запросов содержит требуемый элемент на верхних позициях.

- Средняя позиция. Эта метрика отражает среднюю позицию требуемого элемента в выдаче, если он был найден. Чем больше значение метрики, тем ниже обычно оказывается элемент в выдаче.

- Recall. Значение метрики показывает, какая доля от всех запросов содержит нужный элемент.

- MRR (среднеобратный ранг).

1.3.3. Оценка качества автодополнения кода в IntelliJ IDEA

Алгоритмы автодополнения кода в IntelliJ IDEA использует эвристики и контекстную информацию для генерации дополнений, а также алгоритмы машинного обучения для ранжирования сгенерированных элементов выдачи.

Основные компоненты IntelliJ Platform, обеспечивающие генерацию и ранжирование выдачи автодополнений, следующие:

- *CompletionContributor* содержит информацию о стратегиях автодополнения, контекстах, в которых автодополнение вызывается и о зарегистрированных экземплярах класса *CompletionProvider*.

- *CompletionProvider* добавляет свои варианты автодополнения в формируемый список выдачи.

- *CompletionSorter* выполняет сортировку сортировку полученных вариантов автодополнения с приоритетами, формируемыми в соответствующем классе *CompletionWeighter*.

- *CompletionFinalSorter* предназначен для финальной сортировки элементов выдачи, полученных из разных экземпляров *CompletionSorter*. Фабрика, инстанцирующая элементы такого типа, должна быть

зарегистрирована в качестве расширения (*extension*), например, в xml-дескрипторе плагина. Это применяется для расширения существующей функциональности платформы компонентами определенных типов, описываемых в соответствующих точках расширения (*extension points*).

Для первичной оценки качества автодополнения в IntelliJ IDEA используется отдельный плагин. Данные генерируются с помощью различных стратегий и эвристик, моделирующие пользовательское поведение. Плагин работает следующим образом.

- Для выбранных файлов строится AST.
- На основе построенного AST и выбранной стратегии генерируются "действия" — шаги, которые требуется выполнить IDE (например, печать текста, вызов автодополнения). Стратегия определяется длиной префикса (количество введенных символов в дополняемом элементе до того, как будет вызвано автодополнение), контекстом (учитывается только предшествующий код или код из всего файла) и типом дополняемых токенов.
- Сгенерированные действия выполняются (интерпретируются) внутри IDE, формируя сессии — данные о вызове автодополнения. Каждая сессия представляется следующей информацией:
  - введенные перед вызовом автодополнения символы (префикс);
  - список выдачи автодополнения;
  - длительность сессии;
  - ожидаемый результат;
  - уникальный идентификатор;
  - дополнительная информация в виде положения дополняемого элемента в тексте файла, типа элемента и т. д.
- по полученным сессиям считаются метрики (top-1, top-5, полнота, средняя позиция ожидаемого элемента) и генерируется HTML-отчет.

Платформа IntelliJ также имеет модуль сбора пользовательской статистики, по полученным данным из которой можно подсчитать в том числе и метрики качества реализованного в IDE автодополнения. Используя полученные метрики, можно собрать датасет, наиболее близкий к реальным данным по значениям этих метрик, а на основе собранного датасета затем построить систему, включающую набор экспериментов для оценки качества на составленном датасете.

1.3.4. Выводы

Для оценки качества большинства алгоритмов автодополнения кода применяются синтетические запросы, которые генерируются случайно из больших корпусов кода. Основанный на таких данных эксперимент может привести к неверной оценке точности инструмента, потому что эти данные обычно генерируются, не опираясь на историю действий пользователя, и потому могут содержать и невостребованные в действительности автодополнения. Тем не менее, составить объемный датасет из реальных данных тоже не представляется возможным по причине регламентов защиты персональных данных. Кроме того, возникает проблема сравнения алгоритмов автодополнения кода, так как при разработке каждого инструмента используется собственный набор экспериментов и метрик качества. В таком случае, применяя данные, полученные из пользовательской статистики, можно подобрать датасет на основе большого количества синтетически сгенерированных данных, подходящий к реальным по метрикам и характеристикам.

Таким образом, можно разработать единую систему с фиксированным датасетом и набором экспериментов, метрик и сценариев. Для того, чтобы унифицировать контекст использования, в котором инструмент применяется, его можно встроить в IDE, внутри которой будут проводиться эксперименты.

2. Предлагаемое решение

В рамках данной работы разрабатывается система, позволяющая интегрировать инструменты автодополнения кода, изменяя ранжирование предложенных IntelliJ IDEA автодополнений. Система также содержит данные, позволяющие оценить реальную эффективность инструментов. Для того, чтобы оценить работу какого-то инструмента, его нужно представить в виде плагина для IntelliJ IDEA.

2.1. Интеграция инструментов автодополнения кода

Существующие инструменты автодополнения кода представлены множеством алгоритмов и моделей, поэтому интегрировать их в IDE не всегда простая задача. По этой причине было принято решение разработать инструментарий, предоставляющий единый набор интерфейсов, которые бы упростили интегрирацию инструментов, изменяющих ранжирование в выдаче автодополнений, предоставленной средой разработки. Интеграция с IDE также накладывает определенные ограничения на генеративные модели автодополнения. Тогда как механизм автодополнения в IntelliJ предполагает доступ к элементу выдачи в проекте или сторонних библиотеках, генеративные модели обычно это не учитывают, поэтому выдача таких моделей фильтруется по наличию в неранжированном списке автодополнений IDE. На Рисунке 1 представлена разработанная архитектура инструментария для интеграции инструментов автодополнения кода в платформу IntelliJ.

Интерфейс Model представляет сам инструмент (модель) автодополнения кода, Vocabulary — словарь элементов, используемых инструментом, если инструмент требует его наличия. Классы AbstractModelWrapper и AbstractVocabularyWrapper предоставляют API для взаимодействия с соответствующими объектами. Классы-наследники AbstractModelRunner служат для отправки запроса модели в момент вызова автодополнения и получения результатов, на основе которых изменяется ранжирование. Каждый из них должен быть зарегистрирован в качестве расширения в xml-файле – дескрипторе плагина.

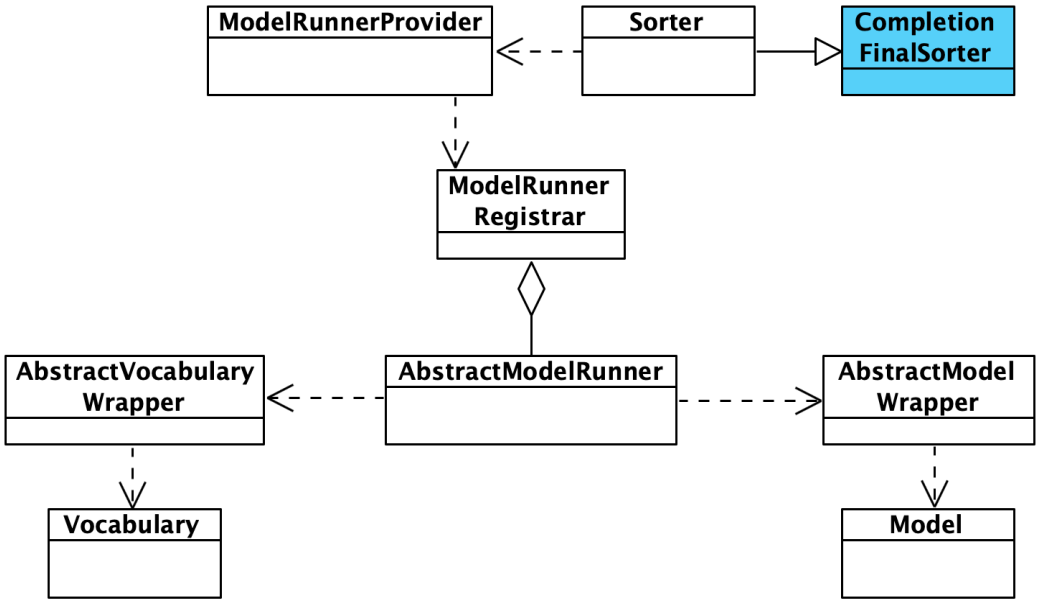

Рис. 1: Архитектура инструментария (синим цветом выделены классы IntelliJ Platform SDK)

При запросе автодополнения в открытом проекте для получения финального ранжирования выдачи IDE обращается к Sorter – наследнику класса CompletionFinalSorter, отвечающему за ранжирование. Он имеет доступ ко всем зарегистрированным наследникам AbstractModelRunner класса, которые использует для получения предсказаний модели. При вызове автодополнения в первый раз нужные классы инстанцируются с помощью сервиса ModelRunnerRegistrar, который выполняет поиск соответствующих расширений в дескрипторе плагина и проверяет единственность расширения такого типа для получения однозначного ранжирования. Sorter обращается к найденному AbstractModelRunner для генерации автодополнений. AbstractModelRunner делает запрос к словарю, если нужно, для получения представления информации в виде, требующемся модели, и отправляет полученные данные в класс модели, которая ранжирует существующие элементы или генерирует свои автодополнения в зависимости от ее типа. Полученные от модели данные в результате запроса отправляются в Sorter, который фильтрует полученные данные по наличию в неранжированном списке выдачи и генерирует финальное ранжирование.

Такое применение инструментов автодополнения напрямую использует возможности IDE, поэтому инструменты должны быть представлены в виде плагинов. Плагины для IntelliJ IDEA поставляются в виде zip-архивов.

Код разработанного инструментария доступен на GitHub[3].

2.2. Веб-сервис

Система тестирования также включает в себя веб-сервис, упрощающий взаимодействие пользователя с инструментом оценки качества, позволяя загрузить zip-файл с плагином и получить отчет о качестве реализованного в плагине алгоритма автодополнения. Сервис использует локальный экземпляр IntelliJ IDEA, куда он устанавливает загруженный плагин для запуска оценки качества автодополнения. Генерация сессий автодополнения на основе загруженного инструмента, дальнейшая оценка и создание отчета производится с помощью плагина для оценки качества автодополнения в IntelliJ. Схема взаимодействия с сервисом показана рисунке 2.

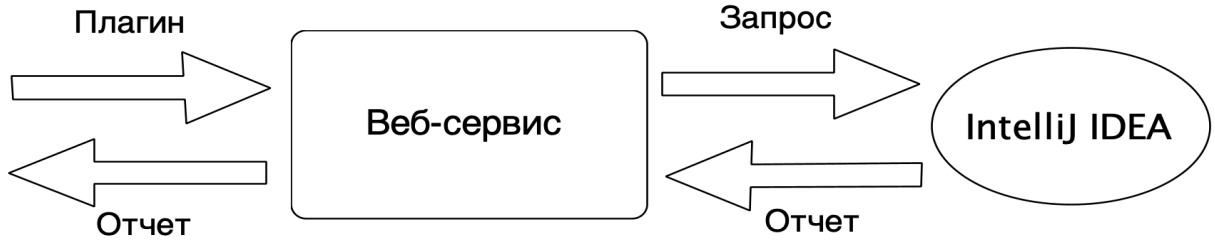

Рис. 2: Схема взаимодействия с веб-сервисом

Для оценки загруженных моделей веб-сервис использует фиксированный набор данных, собранный на основе полученных из пользовательской статистики IntelliJ IDEA метрик качества автодополнения. Подробнее этот набор данных описывается в следующей главе.

[3]https://github.com/JetBrains-Research/code-completion-benchmark-toolkit

2.3. Датасет

В качестве исходных данных для генерации датасета были выбраны 4 больших наиболее популярных репозиториев на GitHub из различных областей, содержащих код на Java. Статистика по выбранным репозиториям приведена в таблице 2. На основе выбранных репозиториев был

Имя репозитория	Описание	Количество строк кода на Java	Количество звезд
intellij-community	IDE для Java и Kotlin	179620948	9877
jenkins	CI инструмент	10158027	15547
languagetool	Инструмент для проверки грамматики	6586085	3685
deeplearning4j	Библиотека для машинного обучения	34706214	11645

Таблица 2: Выбранные репозитории

сгенерирован большой набор сессий автодополнения с помощью плагина для оценки качества автодополнения IntelliJ IDEA. Процесс генерации происходит следующим образом:

- Берется 2% от всех возможных сессий из каждого репозитория (этих данных достаточно, так как каждый репозиторий содержит около 1 млн сессий и более).
- Автодополнение вызывается с использованием полного контекста, то есть всего кода в файле. Это означает, что удаляется только токен, который нужно дополнить, а весь остальной код остается неизменным. Полный контекст выбран в связи с тем, что, как было показано исследованиями, большую часть времени разработчики проводят, редактируя уже написанный код, что и отражает этот контекст.
- Автодополнение вызывается на всех элементах, для которых IDE может это предоставить.
- Используются префиксы автодополнения от 0 до 3, как наиболее часто встречающиеся в пользовательских данных.

2.3.1. Составление датасета

Теперь из множества всех сгенерированных сессий требуется выбрать наиболее близкое по нескольким метрикам подмножество сессий. Как наиболее распространенные и часто используемые при оценке качества автодополнения, были выбраны следующие метрики:

- top-1 — доля сессий автодополнения, в которых нужный элемент находится на первой позиции;
- top-5 — доля сессий автодополнения, в которых нужный элемент находится не ниже, чем на пятой позиции;
- средняя позиция нужного элемента в выдаче.

С помощью статистики использования функций автодополнения в IntelliJ IDEA были получены метрики для двух алгоритмов — стандартного и применяющего методы машинного обучения для финального ранжирования списка выдачи соответственно. Результаты представлены в таблицах 3 и 4 соответственно. По полученным данным видно, что чем длиннее введенный префикс, тем выше top-n метрики и меньше средняя позиция; это означает, что в большем количестве случаев нужный элемент оказывается на верхних позициях в списка выдачи автодополнения.

Префикс / Метрика	0 или 1	2	3
Top-1	0.609	0.762	0.782
Top-5	0.831	0.910	0.910
Средняя позиция	3.694	1.695	1.399
Количество сессий	199012	32645	10666

Таблица 3: Значения метрик для стандартного автодополнения в IntelliJ IDEA

Так как сессии представляют результат выполненного в IDE определенного набора действий, которые приводят к вызову автодополнения, по выбранному подмножеству сессий можно восстановить набор

Метрика \ Префикс	0 или 1	2	3
Top-1	0.652	0.778	0.804
Top-5	0.868	0.926	0.931
Средняя позиция	2.871	1.092	1.050
Количество сессий	120730	18863	5640

Таблица 4: Значения метрик для автодополнения с ранжированием на основе машинного обучения в IntelliJ IDEA

соответствующих действий. Действия являются воспроизводимыми, то есть могут быть переиспользованы для генерации сессий автодополнения внутри IDE с помощью разных инструментов. Они указывают, в каком файле и в каком месте вызывать автодополнение, генерируя таким образом сессии, по которым производится оценка качества.

2.3.2. Обработка данных

Для более удобной обработки сгенерированных сессий автодополнения и подсчета метрик для них добавлена генерация метаинформации, содержащей сведения о длине введенного префикса, позиции ожидаемого элемента в списке выдачи и идентификататоре соответствующей сессии.

Подбор необходимого подмножества сессий, наиболее близкого по полученным метрикам, производится с помощью hyperopt[4]. Таким образом определяется доля сессий, которую нужно взять из каждого репозитория.

Запустив генерацию сессий в описанной выше конфигурации с применением алгоритма автодополнения IntelliJ на основе машинного обучения, а затем использовав hyperopt для выбора подмножеств сессий из каждого репозитория, получили количество требующихся сессий из каждого репозитория для каждого префикса, указанных в таблице 5.

[4]`https://github.com/hyperopt/hyperopt`

Репозиторий \ Префикс	0	1	2	3
intellij-community	3774	1273	2040	639
jenkins	170	3604	183	2386
languagetool	1467	1340	2894	364
deeplearning4j	3837	3442	650	4296
Всего	11245	9659	5767	7685

Таблица 5: Количество сессий, взятых из каждого репозитория, в датасете

Распределения значений метрик для подмножеств данных такого размера и 95%-доверительные интервалы для них показаны на рис. 3, 4 и 5.

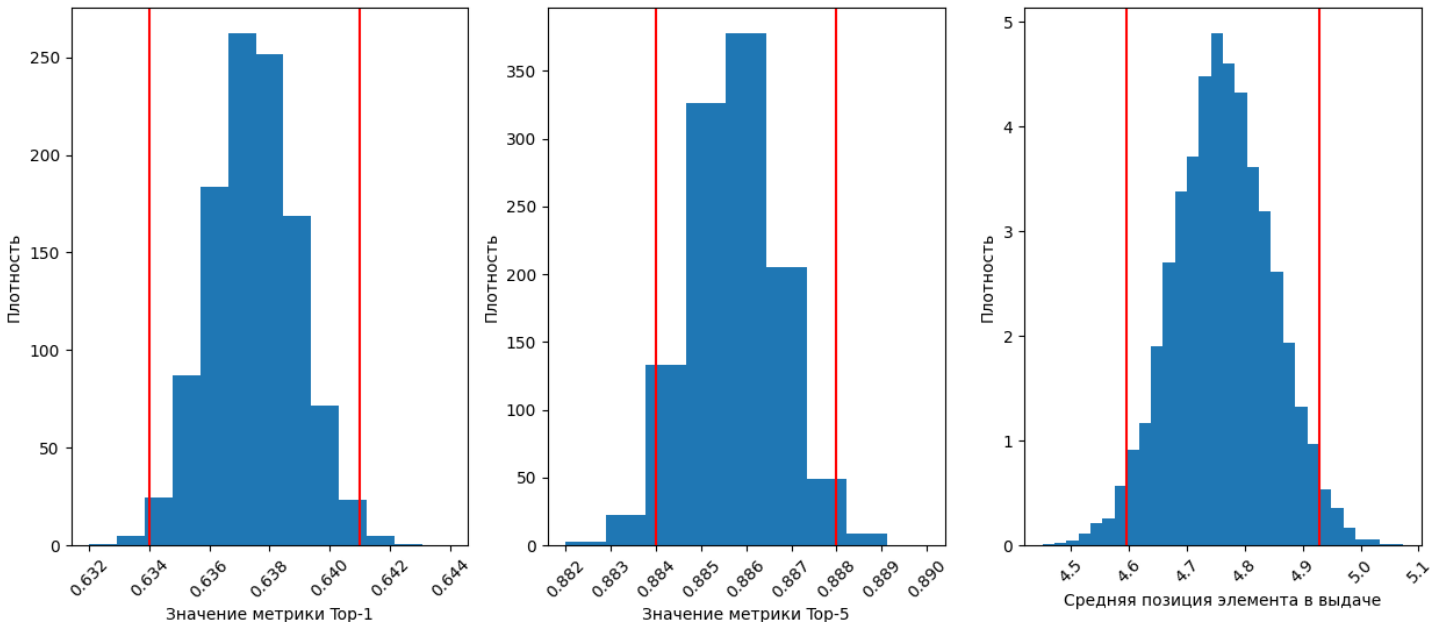

Рис. 3: Значения метрик для наборов сессий с префиксами 0 и 1 (красным выделены границы доверительного интервала)

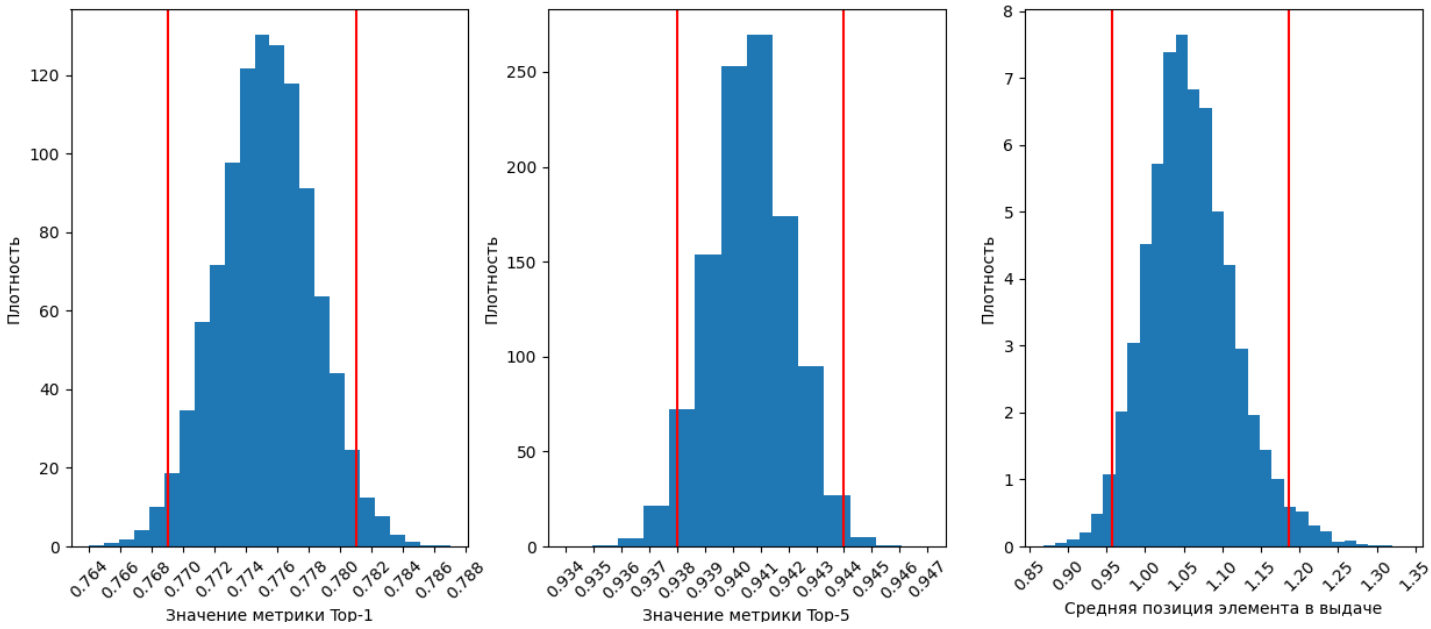

Рис. 4: Значения метрик для наборов сессий с префиксом 2 (красным выделены границы доверительного интервала)

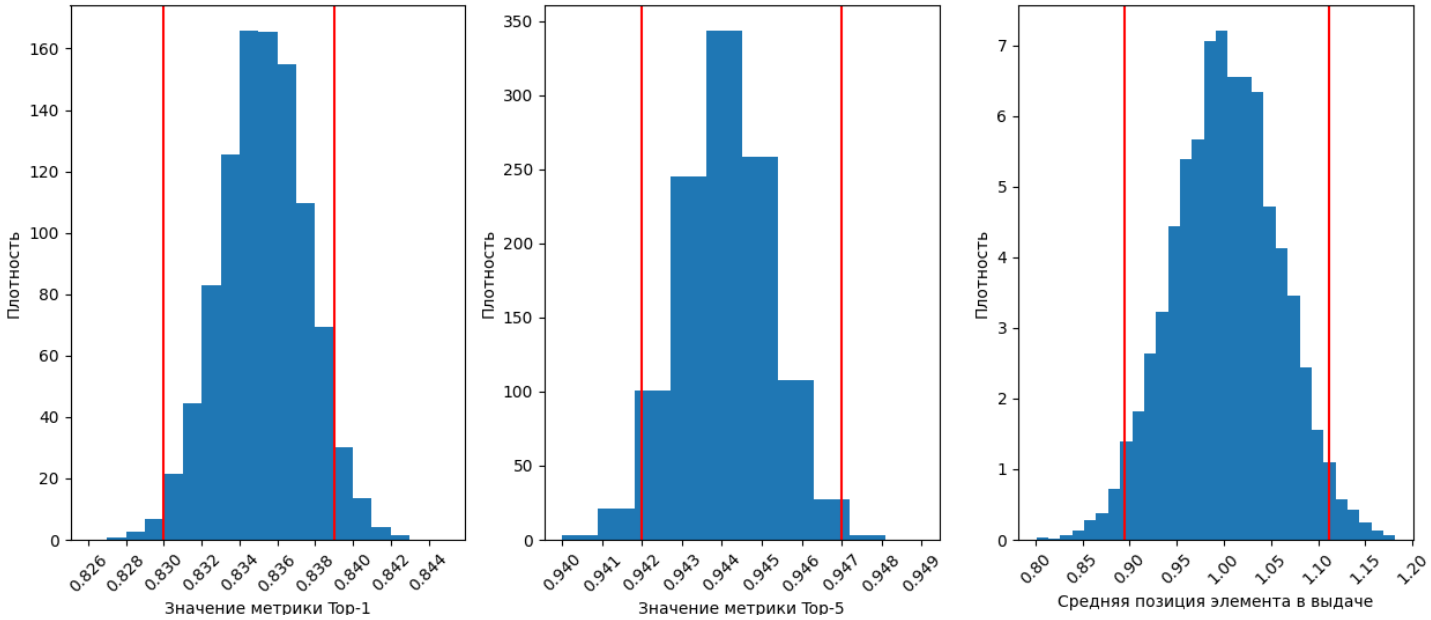

Рис. 5: Значения метрик для наборов сессий с префиксом 3 (красным выделены границы доверительного интервала)

Учитывая приведенные на графиках данные, можно считать, что с вероятностью 95% значения метрик на подмножествах сессий выбранного размера оказываются внутри достаточно узкого интервала, а за его пределами разброс значений также небольшой. Это позволяет считать набор данных такого размера достаточным.

3. Апробация

Для проверки собранного датасета на полученных в результате действиях была запущена генерация сессий с использованием стандартного алгоритма автодополнения в IntelliJ IDEA. Полученные метрики для него в сравнении с реальными указаны в таблице 6.

Метрика \ Префикс		0 и 1	2	3
Top 1	Ожидаемое значение	0.609	0.762	0.782
	Полученное значение	0.624	0.769	0.799
Top 5	Ожидаемое значение	0.762	0.91	0.91
	Полученное значение	0.77	0.907	0.921
Средняя позиция	Ожидаемое значение	3.694	1.695	1.399
	Полученное значение	3.973	1.8	1.431

Таблица 6: Значения метрик для стандартного алгоритма автодополнения IntelliJ IDEA на собранном датасете

По указанным выше данным видно, что отклонения от реальных метрик незначительно, а значит, датасет можно использовать и для оценки других инструментов автодополнения.

Заключение

В ходе данной работы были достигнуты следующие результаты:

- разработан инструментарий для интеграции инструментов автодополнения кода с IntelliJ Platform;
- разработан прототип веб-сервиса, позволяющий оценивать качество инструментов;
- собран датасет на основе значений нескольких полученных из пользовательской статистики IntelliJ IDEA метрик;
- собранный датасет был протестирован на других типах автодополнения IntelliJ IDEA.

Список литературы

[1] Bruch Marcel, Monperrus Martin, Mezini Mira. Learning from Examples to Improve Code Completion Systems // Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering. — ESEC/FSE '09. — 2009. — P. 213–222.

[2] Code Completion with Neural Attention and Pointer Networks / Jian Li, Yue Wang, Michael R. Lyu, Irwin King // Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18. — International Joint Conferences on Artificial Intelligence Organization, 2018. — 7. — P. 4159–4165.

[3] Code Prediction by Feeding Trees to Transformers / Seohyun Kim, Jinman Zhao, Yuchi Tian, Satish Chandra // arXiv preprint arXiv:2003.13848. — 2020.

[4] Combining Program Analysis and Statistical Language Model for Code Statement Completion / S. Nguyen, T. Nguyen, Y. Li, S. Wang // 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE). — 2019. — P. 710–721.

[5] A Dataset of Simplified Syntax Trees for C# / Sebastian Proksch, Sven Amann, Sarah Nadi, Mira Mezini // Proceedings of the 13th International Conference on Mining Software Repositories. — MSR '16. — New York, NY, USA : Association for Computing Machinery, 2016. — P. 476–479.

[6] Distributed Representations of Words and Phrases and Their Compositionality / Tomas Mikolov, Ilya Sutskever, Kai Chen et al. // Proceedings of the 26th International Conference on Neural Information Processing Systems - Volume 2. — NIPS'13. — Red Hook, NY, USA : Curran Associates Inc., 2013. — P. 3111–3119.

[7] Enriching in-IDE process information with fine-grained source code

history / S. Proksch, S. Nadi, S. Amann, M. Mezini // 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). — 2017. — P. 250–260.

[8] Evaluating the Evaluations of Code Recommender Systems: A Reality Check / Sebastian Proksch, Sven Amann, Sarah Nadi, Mira Mezini // Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. — ASE 2016. — New York, NY, USA : ACM, 2016. — P. 111–121.

[9] Hellendoorn Vincent J., Devanbu Premkumar. Are Deep Neural Networks the Best Choice for Modeling Source Code? // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. — ESEC/FSE 2017. — New York, NY, USA : ACM, 2017. — P. 763–773.

[10] Jin X., Servant F. The Hidden Cost of Code Completion: Understanding the Impact of the Recommendation-List Length on its Efficiency // 2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR). — 2018. — P. 70–73.

[11] Karampatsis Rafael-Michael, Sutton Charles. Maybe Deep Neural Networks are the Best Choice for Modeling Source Code // arXiv preprint arXiv:1903.05734. — 2019.

[12] On the Naturalness of Software / Abram Hindle, Earl T. Barr, Zhendong Su et al. // Proceedings of the 34th International Conference on Software Engineering. — ICSE '12. — Zurich, Switzerland : IEEE Press, 2012. — P. 837–847.

[13] Proksch Sebastian, Amann Sven, Mezini Mira. Towards Standardized Evaluation of Developer-Assistance Tools // Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering. — RSSE 2014. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 14–18.

[14] Proksch Sebastian, Lerch Johannes, Mezini Mira. Intelligent Code

Completion with Bayesian Networks // ACM Trans. Softw. Eng. Methodol. — 2015. — Dec. — Vol. 25, no. 1.

[15] Pythia: AI-Assisted Code Completion System / Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, Neel Sundaresan // Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery Data Mining. — KDD '19. — New York, NY, USA : Association for Computing Machinery, 2019. — P. 2727–2735.

[16] Robbes R., Lanza M. How Program History Can Improve Code Completion // Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering. — ASE '08. — USA : IEEE Computer Society, 2008. — P. 317–326.

[17] Robbes Romain, Lanza Michele. SpyWare: A Change-Aware Development Toolset // Proceedings of the 30th International Conference on Software Engineering. — ICSE '08. — Leipzig, Germany : Association for Computing Machinery, 2008. — P. 847–850.

[18] A Statistical Semantic Language Model for Source Code / Tung Thanh Nguyen, Anh Tuan Nguyen, Hoan Anh Nguyen, Tien N. Nguyen // Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. — ESEC/FSE 2013. — New York, NY, USA : Association for Computing Machinery, 2013. — P. 532–542.

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

[20] Stylos Jeffrey. Usability implications of requiring parameters in objects' constructors // In ICSE '07: Proceedings of the 29th International Conference on Software Engineering. — IEEE Computer Society, 2007. — P. 529–539.

[21] Toward Deep Learning Software Repositories / Martin White, Christopher Vendome, Mario Linares-Vásquez, Denys Poshyvanyk //

Proceedings of the 12th Working Conference on Mining Software Repositories. — MSR '15. — Florence, Italy : IEEE Press, 2015. — P. 334–345.

[22] Tu Zhaopeng, Su Zhendong, Devanbu Premkumar. On the Localness of Software // Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. — FSE 2014. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 269–280.

[23] JetBrains. — What is the IntelliJ Platform?, 2020. — URL: `https://www.jetbrains.org/intellij/sdk/docs/intro/intellij_platform.html` (дата обращения: 2020-05-05).

[24] When Code Completion Fails: A Case Study on Real-world Completions / Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, Alberto Bacchelli // Proceedings of the 41st International Conference on Software Engineering. — ICSE '19. — IEEE Press, 2019. — P. 960–970.