Санкт-Петербургский государственный университет
Математико-механический факультет

Кафедра системного программирования

Чебыкин Александр Евгеньевич

Синтез программного кода с использованием машинного обучения

Бакалаврская работа

Научный руководитель:
ст. преп. Я. А. Кириленко

Рецензент:
СПбАУ РАН ст. преп А. А. Шпильман

Санкт-Петербург
2018

Saint Petersburg State University
Faculty of Mathematics and Mechanics

System Programming

Aleksandr Chebykin

Source code generation using machine learning techniques

Bachelor's Thesis

Scientific supervisor:
senior lecturer I.A. Kirilenko

Reviewer:
SPbAU RAS senior lecturer A. A. Shpilman

Saint-Petersburg
2018

Содержание

Введение

С течением времени инструменты для разработки программного обеспечения становятся всё изощреннее, упрощая или вовсе ликвидируя некоторые части труда программиста. Текстовые редакторы постепенно заменяются интеллектуальными интегрированными средами разработки (IDEs), которые могут, например, подсказывать имена переменных, находить дублирующиеся секции кода, генерировать шаблонный код конструкторов.

Наивысшая вообразимая степень развития таких инструментов помощи разработчику лежит в полным исключении человеческого труда. Эта мечта давно привлекает внимание исследователей в области компьютерных наук. В течение долгого времени идёт поиск алгоритма, способного, получив на вход описание программы в удобном для человека представлении — на естественном языке или в виде модели предметной области — выдать код на требуемом языке программирования.

Уже существует подходы, способные в очень ограниченных контекстах решать эту задачу. Однако большинство из них сосредоточенно на Domain Specific Languages (DSLs) [1, 2] или на непрактично ограниченных подмножествах языков общего назначения [3].

Вопрос о существовании алгоритма, способного генерировать разнообразный код на языках общего назначения, остаётся открытым. Чрезвычайная сложность исследовательской задачи ведёт к разбиению её на разнообразные множества составных частей, которые рассматриваются отдельно.

Во-первых, изучается проблема рекомендации программисту существующих библиотек и способов работы с этими библиотеками через их Application Programming Interfaces (APIs) [4]. Задача интересна тем, что разработчики в действительности часто сталкиваются с однообразными проблемами, решения для которых в виде библиотек уже были созданы, протестированы и отлажены, а потому в большинстве случаев пригодны для использования. Но поиск нужной библиотеки и правильного способа работы с ней — нетривиальная задача. Чаще всего её решают с помощью поисковых систем в сети Интернет, которые не предназначены для этого, а потому бывают неэффективны [5]. Большое количество открытого исходного кода позволяет предлагать в качестве альтернативы статистические модели, на этом коде обученные.

В этой области исследований, называемой API mining [6], ведутся активные работы, предлагаются разнообразные алгоритмы [6, 7], которые можно примерно охарактеризовать как генерирующие по пользовательскому запросу аппроксимацию кода.

Второй популярный [8, 9, 10, 11] подход к проблеме – рассмотреть задачу генерации компилируемого кода по некоему удобному структурированному входу, например, по названиям используемых классов [9]. Преимущество такой формулировки – в возможности ограничивать вход так, чтобы по нему было возможно генерировать

синтаксически и семантически верный выход. Но это является и ограничением: работать со специфическим для алгоритма форматом входа пользователю не так удобно, как, например, с естественным языком.

Постановка задачи

Цель данной работы — объединить существующие подходы в области генерации кода и создать инструмент, способный по описанию на английском языке генерировать синтаксически и семантически верный Java код для работы с API.

Для достижения этой цели ставятся следующие задачи.

- Изучить существующие подходы и выбрать наиболее перспективные из них.
- Собрать данные для обучения статистической модели.
- Эффективно обучить модель, в том числе исследовать возможную предобработку данных, улучшающую результат.
- Интегрировать модель в существующую среду разработки в виде плагина.
- Выполнить апробацию плагина.

1. Обзор существующих решений

1.1. Терминология

В данной работе речь часто идёт о вызовах API. Вызов API определяется как вызов публичного метода API некоей библиотеки, он состоит из двух частей — имени класса и имени метода этого класса. Например, вызов API «Random.nextInt» состоит из имени класса «Random» и имени метода «nextInt».

Также упоминаются нейронные сети — класс статистических моделей, позволяющий аппроксимировать широкий круг функций. Базовая единица сети — нейрон — вычисляет уникальное для себя несложное преобразование входного вектора в скаляр. Нейроны объединяются в соединенные последовательно слои. В полносвязной нейронной сети выход каждого слоя — то есть скаляры, вычисленные каждым из нейронов в слое и объединенные в вектор — служит входом каждого из нейронов следующего слоя. Нейроны, находящиеся в одном слою, между собой не связаны. В случае, если есть набор данных с соответствием входов и выходов (случай обучения с учителем), нейронные сети обучаются аппроксимировать целевую функцию, изменяя вычисляемые каждым из нейронов преобразования. Цель обучения — как можно лучше предсказать выходные данные по входным.

Рекуррентные нейронный сети (RNNs) — класс нейронных сетей, разрешающий рекуррентные соединения нейронов самих с собой, и таким образом дающий им возможность запоминать информацию.

Кодер-декодер — специальная архитектура нейросетевой модели [12], чаще всего использующаяся для машинного перевода. Кодер — первая нейронная сеть — считывает вход и представляет его в виде вектора в скрытом пространстве. Предполагается, что в таком векторе, называющимся вектором контекста, оказывается суть входа, независимая от его конкретного представления. Соответственно по вектору контекста декодер генерирует представление сути входа в терминах выходного пространства.

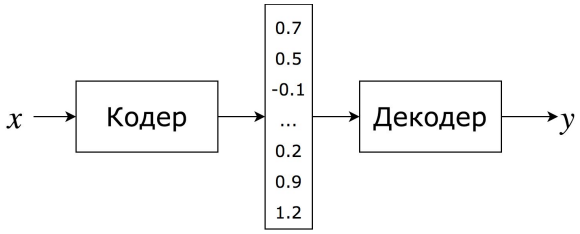

Рис. 1: Обобщенная архитектура кодера-декодера

1.2. Алгоритмы рекомендации API

1.2.1. MAPO, UP-Miner, PAM, DeepAPI

Существует множество алгоритмов, задача которых — извлечь из исходного кода и предоставить пользователю самые популярные способы работы с библиотеками через API (в виде последовательностей вызовов API).

Исторически первым из подобных алгоритмов был MAPO [7]. Он кластеризует последовательности вызовов API, находит наиболее частотные вызовы внутри кластеров, а затем ранжирует их в соответствии с похожестью на контекст кода, из которого был вызван алгоритм.

Алгоритм UP-Miner [6] является улучшенной версией MAPO и предлагает дополнительный шаг кластеризации с использованием n-грамм последовательностей вызовов API в качестве метрики.

Подход PAM [13] значительно превосходит MAPO и UP-Miner в точности и полноте рекомендаций. В PAM используется вероятностная модель, состоящая из совместного распределения находящихся в коде вызовов API и ненаблюдаемых паттернов использования API, задуманных разработчиком. Такая модель аппроксимируется с помощью EM-алгоритма [14].

Наиболее интересен алгоритм DeepAPI [15], основанный на рекуррентных нейронных сетях. Принимая на вход описание требуемой функциональности на английском языке (например, «generate random number»), модель может генерировать последовательность вызовов API языка Java (в данном случае — «Random.new Random.nextInt», что соответствует созданию объекта типа Random и вызова его метода nextInt).

Подход выгодно отличается от уже рассмотренных алгоритмов с двух сторон. Во-первых, другие алгоритмы не позволяют пользователю ясно обозначить свои пожелания (MAPO и UP-Miner принимают на вход имя вызова API, но вызов API может использоваться в разных сценариях, поэтому принимать его в качестве входа — значит оставлять место неопределенности; PAM вообще не требует другого входа, кроме кода, использующего определенную библиотеку, с вызовами из которой он и работает). Из-за этого выход моделей содержит много ненужного пользователю, и нет уверенности, что в нём есть что-то ему полезное. А DeepAPI позволяет пользователю точно определить пожелания на английском языке.

Во-вторых, для использования этих алгоритмов пользователь должен знать, какие вызовы API (в случае MAPO и UP-Miner) или библиотеки (в случае PAM) ему интересны. DeepAPI может рекомендовать как вызовы API, так и специфику их использования.

Принимая во внимание эти соображения, в качестве первого алгоритма комбинированной модели генерации кода предлагается взять DeepAPI.

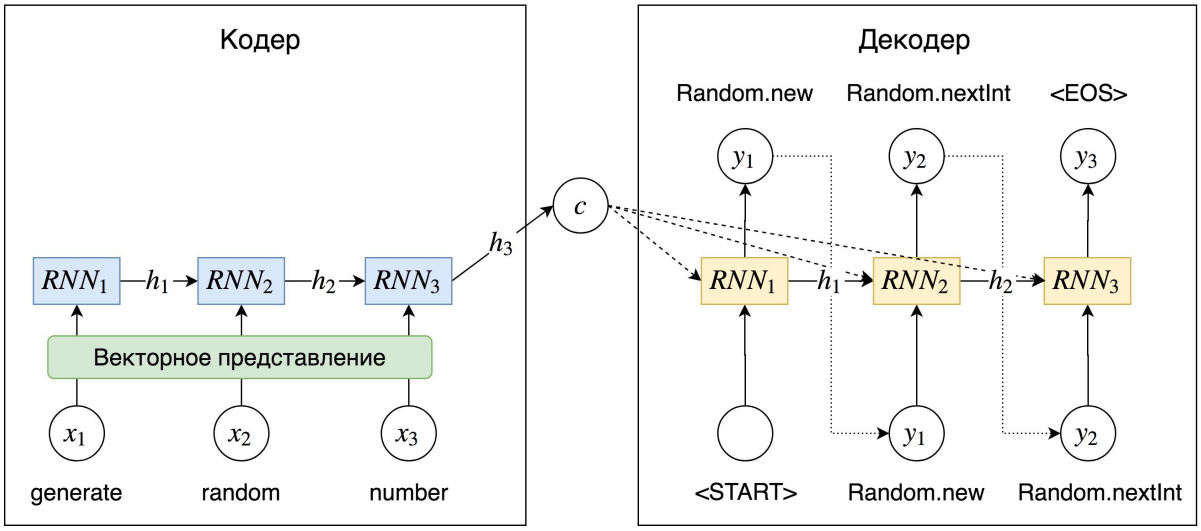

Рис. 2: Пример работы RNN кодера-декодера

1.2.2. Детали работы DeepAPI

Рассмотрим алгоритм DeepAPI подробнее. Его авторами задача рекомендации API формулируется как задача перевода с английского языка на язык вызовов API (в котором слова — вызовы API, предложения — последовательности слов). Поэтому оказывается возможным использовать обширные наработки в области нейронного машинного перевода, а именно взять в качестве модели кодер-декодер [12], где кодер и декодер — рекуррентные нейронные сети.

Кодер считывает вход по одному слову, преобразует каждое в вектор в многомерном пространстве (embedding) и последовательно обновляет своё скрытое состояние. К концу предложения в этом состоянии содержится смысл входного предложения. Затем вектор контекста передаётся декодеру, который генерирует слова последовательно, ориентируясь на вектор контекста и последнее сгенерированное слово. Декодер оканчивает работу, когда генерирует специальный символ, означающий конец предложения.

Пример работы модели представлен на рисунке 2. На рисунке состояния сетей развернуты во времени, поэтому RNN_1, RNN_2, RNN_3 это состояние рекуррентной нейронной сети на шагах $1, 2, 3$. Стоит обратить внимание, что кодер и декодер состоят из разных нейронных сетей и работают в разные промежутки времени: в данном примере сначала рекуррентная сеть-кодер делает 3 шага, а затем рекуррентная сеть-декодер делает 3 шага.

Преимущества такой модели включают в себя:

- успешную работу с синонимами (так как слова, используемые в похожих контекстах, имеют близкое векторное представление);

- хорошие результаты на длинных входах благодаря запоминающей способности рекуррентных сетей;

- зависимость результата от последовательности слов во входе, в отличие от моделей, использующих мешок слов: из-за потери информации о последовательности они не способны, например, отличить запрос «convert string to int» от «convert int to string».

Большой недостаток такой модели — необходимость в большом количестве данных для её обучения. Здесь под данными понимаются пары описаний одной и той же функциональности на английском языке и на языке вызовов API. Пример такой пары: («generate random number», «Random.new Random.nextInt»).

Сбор данных для обучения — необходимая и важная часть нашей работы, подробно описанная в секции 2.

Для базовой модели кодера-декодера, описанной выше, существует несколько улучшений, которые надёжно улучшают результаты её работы. Два таких популярных улучшения используются и в DeepAPI.

- Обычный кодер заменяется двусторонним, который обрабатывает вход как в прямом порядке, так и в обратном. Два получившихся вектора контекста конкатенируются для получения финального вектора контекста [16].

- Механизм внимания [17] позволяет декодеру фокусироваться на разных входных словах при генерировании разных выходных. Реализуется это следующим образом: вместо фиксированного вектора контекста на каждом шаге в его качестве берётся взвешенная сумма всех прошлых состояний кодера. Веса в этой сумме определяются отдельной однослойной нейронной сетью, тренируемой вместе со всей моделью.

1.3. Алгоритмы генерации кода

1.3.1. Генерация кода по описанию на естественном языке

Последние алгоритмы в области генерации кода по текстовому запросу представлены в работах *Ling et al.* [3] и *Yin et al.* [18]. Первые предлагают новую архитектуру нейронных сетей — Latent Predictor Networks — которая даёт возможность лучше копировать важные слова (например, названия переменных и строковые константы) из входа в выход. Вторая из статей представляет новую версию кодера-декодера, специализированную для генерации синтаксических деревьев вместо стандартных линейных последовательностей.

Большой недостаток этих работ — используемые данные. *Ling et al.* работают с тремя наборами данных: два из них очень похожи и представляют собой код на Python

карт из видеоигр HearthStone и Magic The Gathering и описания этих карт; третий — аннотированные разработчиками строчки кода на Python, использующие фрэймворк Django. *Yin et al.* сообщают результаты на 2 из этих 3 наборах данных — HearthStone и Django.

Датасет из карт Hearthstone довольно однороден и ограничен небольшим подмножеством Python, напоминая таким образом DSL больше, чем язык общего назначения. А набор данных Django, хотя и покрывает разнообразные случаи использования языка и фрэймворка, имеет непрактично длинные описания каждой строчки кода. Например строчка "`for i in range(0, len(result)):`" описывается так: "for every i in range of integers from 0 to length of result, not included". Генерация кода по описанию, превосходящему его по длине, кажется непрактичным.

Также авторы этих работ ставят себе целью генерацию обычного кода, включающего, например, сложение с вычитанием, доступ к ячейкам массива и другие базовые элементы кода. Мы, напротив, заинтересованы в коде, использующем API существующих библиотек, то есть работающем на более высоком уровне.

1.3.2. Генерация кода по структурированному входу

Алгоритмы из этой области принимают пользовательский ввод в довольно разных формах. SyPet [11] позволяет генерировать код, использующий Java SE, по сигнатуре метода и тестовым методам. Алгоритм основывается на построении сети Петри [19] API библиотеки, генерации скетча программы с пропусками и их заполнении с помощью SAT-решателей.

Prospector [10] принимает на вход два типа и генерирует код, преобразовывающий первый из этих типов во второй. Для этого составляется граф всех преобразований из одного типа в другой и находится кратчайший путь в нём.

Bayou [9] — наиболее интересная модель в этой области. Принимая на вход так называемые «признаки» (evidences), которые представляют собой имена классов и методов или их части, алгоритм способен генерировать валидный код на Java. Алгоритм позволяет решать более широкий круг задач, чем Prospector, и требует меньше входных данных, чем SyPet. Поэтому предлагается использовать именно его в качестве второго алгоритма нашей комбинированной модели.

Аналогично DeepAPI, в Bayou используются нейронные сети в архитектуре кодер-декодер. Кодер — обычная нейронная сеть с одним скрытым слоем — переводит набор признаков в вектор контекста. По нему декодер — рекуррентная нейронная сеть, генерирующая не линейную последовательность, но дерево узлов — генерирует скетч — аппроксимацию синтаксического дерева, содержащую только основную информацию о структуре и типах. Информация о конкретных переменных добавляется к сгенерированному нейросетевой моделью скетчу с помощью комбинаторной стохастической

конкретизации.

1.4. Плагин к IntelliJ IDEA, реализующий Bayou

Плагин к IntelliJ IDEA, интегрирующий алгоритма Bayou в эту среду разработки, разрабатывается Владиславом Танковым [20]. Так как именно Bayou я рассматриваю как вторую часть предлагаемой мной комбинированной модели, я переиспользую работу, проведенную Владиславом.

Этот плагин получает от пользователя входную информацию (имена классов и методов, код для работы с которыми хочет увидеть пользователь) через особые аннотации или в комментариях на специально разработанном DSL. Затем составляется запрос и направляется в модуль bayou-implementation. Сгенерированный код плагин преобразует в формат Project Structure Interface (PSI) [21], используемый для представления информации о файлах в IntelliJ IDEA, и вставляет в пользовательский файл, откуда был сделан запрос.

Благодаря тому, что модуль bayou-implementation, реализующий алгоритм Bayou, отделен от остального кода, его оказывается возможным переиспользовать в данной работе.

2. Сбор данных

Для обучения модели DeepAPI необходим большой набор пар описаний одной и той же функциональности на двух языках: английском и языке вызовов API.

Источником такой информации могут служить методы с комментариями документации. В Java по стандарту Javadoc первое предложение такого комментария должно кратко описывать его суть[1], а описания на языке вызовов API можно получить из кода метода, лианеризовав его синтаксическое дерево и выбрав из него только вызовы API. Процесс обработки отдельного метода подробно описан в разделе 2.2.

Большое число документированных методов можно извлечь из проектов с открытым исходным кодом, опубликованных в сети Интернет.

GitHub[2] — один из наиболее популярных хостингов таких проектов. Так же, как и в оригинальной статье DeepAPI, набор нужных данных извлекается из репозиториев, опубликованных там.

Дополнительно нами исследовались альтернативные источники данных, в частности, другие сайты с опубликованными проектами с открытым кодом — Codeplex[3] и SourceForge[4]. К сожалению, на этих сайтах оказалось лишь относительно небольшое число проектов, многие из которых постепенно мигрируют на GitHub, или уже сделали это. Также у этих сайтов нет API для поиска проектов, а наличие такого API необходимо для автоматического сбора данных. Таким образом, сбор потенциально небольшого количества дополнительных данных кажется нетривиальной задачей, а потому эти альтернативные источники данных нами не используются.

Мы собираем данные с GitHub в несколько шагов:

1. получение списка интересных нам проектов;
2. скачивание проектов;
3. обработка проектов, представляющая собой поиск документированных методов и извлечение из них первых предложений комментариев и особым образом линеаризованных вызовов API.

Высокоуровневый обзор процесса представлен на рисунке 3. Обсудим каждый шаг подробнее.

2.1. Поиск и скачивание релевантных проектов

Нам интересны репозитории с кодом на Java. Аналогично оригинальной статье DeepAPI рассматриваются только те проекты, у которых есть хотя бы одна звезда

[1] `http://www.oracle.com/technetwork/articles/java/index-137868.html`
[2] `http://github.com`
[3] `https://archive.codeplex.com/`
[4] `https://sourceforge.net/`

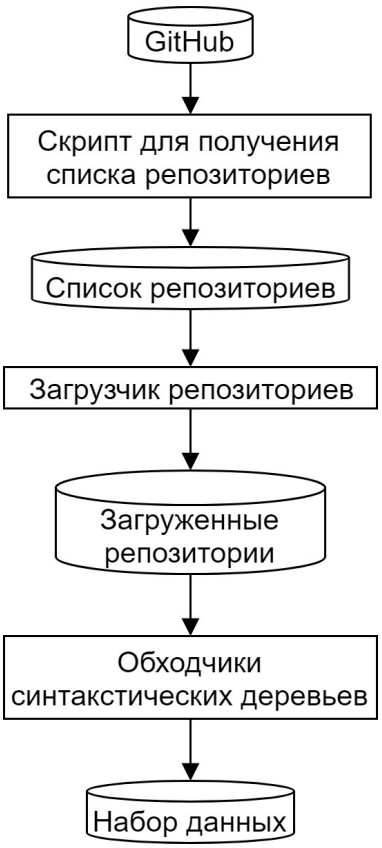

Рис. 3: Процесс сбора данных

(пользователь сайта GitHub может отметить звездой понравившийся ему проект). Такое ограничение позволяет отсеять неиспользуемые или учебные проекты. С помощью передачи нужных значений в качестве параметров в GitHub Search API можно добиться выполнения обоих требований.

Используя это API через официальную библиотеку Octokit.rb[5], было получено 400,475 ссылок на релевантные проекты, созданные с 2012 по 2017 года.

Во время работы с GitHub Search API возникают небольшие технические трудности.

Во-первых, на любой запрос оно возвращает не более 1 000 результатов, нам же необходимо собрать информацию об уже упоминавшихся ранее 400 475 проектов. Чтобы обойти это ограничение, в каждом запросе мы ограничиваем время создания репозитория коротким промежутком времени, например, “2016-01-01 .. 2016-01-03”. Каждый наш запрос покрывает 3 дня: такой отрезок времени достаточно короткий, чтобы в течение него было создано не более 1 000 репозиториев.

Во-вторых, Search API ограничивает количество запросов в минуту тридцатью. Чтобы не зайти за этот предел, наш скрипт простаивает 2 секунды после каждого запроса.

Полученные URL репозитория, его полное название, количество звёзд, наблюда-

[5] `https://github.com/octokit/octokit.rb`

телей и ответвлений сохраняются в локальную базу данных SQLite[6].

После сбора этой информации начинается процесс скачивания репозиториев с помощью средства контроля версий Git. Для его ускорения опция «depth» выставляется равной 1: таким образом не тратится время на скачивание полной истории проекта.

Пути к скачанным репозиториям сохраняются в базе данных.

2.2. Извлечение данных

В первую очередь необходимо найти методы с комментариями документации. Здесь и далее используется библиотека JavaParser[7], предназначенная именно для удобной работы с Java кодом из Java кода.

Сначала JavaParser запускается на файлах с исходным кодом, проводит их синтаксический анализ и строит синтаксические деревья.

После построения синтаксических деревьев файлов на каждом из них запускается специально реализованный нами обходчик синтаксических деревьев, которые находит и сохраняет методы, имеющие Javadoc комментарии. Из каждого комментария выделяется первое предложение, из него убираются скобки со всем содержимым, удаляются знаки препинания, а буквы переводятся в нижний регистр.

Синтаксические деревья передаются другому реализованному нами обходчику, который извлекает последовательность API.

Для правильного выделения вызова API из кода необходимо вывести типы используемых переменных, а также типы возвращаемых методами значений (а для этого надо знать, например, какой из переопределенных методов вызван). Для вывода типа в обоих ситуациях используется интегрированная в JavaParser библиотека JavaSymbolSolver, специализированная на этом.

Надо упомянуть, что эта библиотека неидеальна и содержит некоторое количество ошибок. В течение нашей работы мы обнаружили и отследили до места возникновения 4 из них[8]. Был отправлен пулл-реквест[9] с решением одной из этих ошибок, который был принят; остальные ошибки, более сложные и требующие углубленного знания внутренних процессов работы библиотеки, были оставлены разработчикам, её поддерживающим. Оставшиеся ошибки возникают достаточно редко и потому в ходе данной работы игнорируются.

Последовательность API извлекается аналогично исходной статье. Обходчик идёт по дереву так, как по нему мог бы идти интерпретатор во время исполнения: в обратном порядке (post order), обрабатывая параметры вызова до обработки самого

[6] https://www.sqlite.org

[7] https://github.com/javaparser/javaparser

[8] https://github.com/javaparser/javaparser/issues?utf8=%E2%9C%93&q=+is%3Aissue+author%3AAwesomeLemon

[9] https://github.com/javaparser/javaparser/pull/1458

```java
/**
 * Copies bytes from a large (over 2GB) <code>InputStream</code> to an
 * <code>OutputStream</code>.
 * <p>
 * This method uses the provided buffer, so there is no need to use a
 * <code>BufferedInputStream</code>.
 * <p>
 * ...
 * @since 2.2
 */
public static long copyLarge(final InputStream input,
        final OutputStream output, final byte[] buffer) throws IOException {
    long count = 0;
    int n;
    while (EOF != (n = input.read(buffer))) {
        output.write(buffer, 0, n);
        count += n;
    }
    return count;
}
```

Последовательность API: InputStream.read OutputStream.write
Описание: copies bytes from a large inputstream to an outputstream

Рис. 4: Пример извлечения данных

вызова, и так далее. Встретив вызов конструктора *new C()*, обработчик добавляет к последовательности API *C.new*. Встретив вызов метода *o.m()*, где *o* — объект класса C, обработчик добавляет к последовательности API *C.m*. При обнаружении конструкции *if-else*, обрабатывается условное выражение внутри *if*, затем блок ветки *if* и наконец блок ветки *else*.

Вводится еще один шаг, не встречавшийся в исходной статье: при обработки конструкции *try-with-resources* сохраняется класс *C* создаваемого в узле *try* объекта и после обработки блока *try* к последовательности API добавляется вызов *C.close* (который в таком случае вызывается неявно). Мы считаем, что информация о том, что некоторые цепочки API должны заканчиваться финализацией, полезна, т.к. позволит модели правильно работать с ресурсами, её требующей.

Пример извлечения данных из метода представлен на рисунке 4.

Извлеченные данные сохраняются в уже упомянутую локальную базу данных.

В результате мы получаем 17 631 306 пар английских описаний и последовательностей API. Однако это число не имеет смысла сравнивать с 7 519 907 — количеством данных в исходной статье. В электронном письме авторы объяснили нам, что 7 519 907 это размер набора данных после фильтрации по словарю (когда удаляются вызовы API, не входящие в Java SE) и популярности (когда удаляются английские слова и

вызовы API, не входящие в список 10 000 самых популярных в своей категории). Такая предобработка удаляет некоторые пары целиком, значительно снижая размер набора данных.

Предобработка данных и итоговое их количество обсуждаются в секции 3.1.

Для возможности простого повторения эксперимента мы выкладываем наш код в открытый доступ[10].

[10] `https://github.com/AwesomeLemon/api-extraction-java`

3. Обучение модели DeepAPI

3.1. Предобработка данных

Собранные данные могут быть улучшены перед их использованием для обучения модели. Предложенные шаги предобработки нацелены на облегчение обучения модели и улучшение результатов, что подтверждается экспериментом, описанным в секции 5.1.

В данном разделе мы описываем создание двух наборов данных:

1. dataset-original близок к набору данных из оригинальной статьи;
2. dataset-bayou проходит более тщательную предобработку.

В таблице 1 указано влияние каждого из шагов предобработки на количество данных и финальный размер наборов данных.

3.1.1. Фильтрация по звёздам

Изначально аналогично авторам оригинальной статьи мы собираем только те репозитории, у которых есть хотя бы 1 звезда. Однако такие проекты содержат большое количество некачественных комментариев, поэтому при создании набора данных dataset-bayou мы используем только проекты, имеющие не меньше 2 звёзд.

3.1.2. Фильтрация по языку

Предполагается, что модель принимает на вход предложение на английском языке, однако в действительности многие комментарии написаны не на нём. Поэтому предлагается автоматически определять язык комментария с помощью специальной библиотеки[11] и отбрасывать пары с описанием не на английском языке.

Однако многие предложения на английском распознаются как написанные на других языках. Предположительно, это происходит из-за небольшой длины предложений и из-за использования в них профессионального программистского жаргона. Чтобы не терять эти предложения, меняется тактика фильтрации.

Вместо того, чтобы оставлять только пары описаний с комментариями на английском языке, удаляются пары с комментариями на других языках, которые хорошо распознаются и часто встречаются в данных. После изучения данных вручную было решено удалять комментарии на китайском, корейском, немецком, японском, русском, польском языках (перечисление идёт в порядке убывания популярности). Причина хорошего распознавания этих языков лежит, скорее всего, в использовании ими разных алфавитов, не совпадающих с английским.

[11] `https://github.com/Mimino666/langdetect`

Такая фильтрация оставляет в словаре по большей части исключительно английские слова.

Этот фильтр не применяется при создании набора данных dataset-original.

3.1.3. Фильтрация по словарю

Авторы оригинальной статьи оставляют в последовательностях только стандартные вызовы API из библиотеки Java SE.

Так как наша глобальная цель — интегрировать подход DeepAPI с Bayou, при создании обоих наборов данных для более эффективной работы с Bayou мы оставляем вызовы API только с теми классами и методами, которые может использовать Bayou — это часть методов из Java SE и часть методов из стандартных библиотек для Android.

Также мы делаем наблюдение, что вызовы API из стандартных библиотек встречаются повсеместно, в том числе там, где ключевая функциональность лежит в других, более специфических, вызовах. При этом комментарии скорее всего относятся именно к специфическим вызовам. Отфильтровывая их, мы получаем описание, несоответствующее оставшимся вызовам. Поэтому предлагается удалять такие пары описаний, где было отфильтровано по словарю больше 20% вызовов.

При этом не будут отфильтрованы только пары описаний, где подавляющее большинство вызовов – целевые для нас. Скорее всего комментарии к таким методам описывают именно использование этих вызовов.

Примером пары описаний, в которой из-за фильтрации специфического вызова теряется соответствие между комментарием и вызовами может послужить такая пара: («utility function to set the value stored in a particular Keyframe»,
«java.lang.Object.getClass java.lang.reflect.Method.invoke
com.actionbarsherlock.internal.nineoldandroids.animation.Keyframe.setValue»). Здесь 2 стандартных вызова, но описание относится к третьему, специфическому вызову, который будет отфильтрован. Мы считаем правильным, что этот пример, в котором было отфильтровано 33.(3)% вызовов, будет удалён из набора данных dataset-bayou.

При создании набора данных dataset-original удаления пар, где отфильтровано много вызовов, не производится.

3.1.4. Сокращение повторяющихся вызовов

В некоторых извлеченных последовательностях API вызовы API могут повторяться несколько раз подряд. Такое может произойти, например, в результате линеаризации синтаксического дерева в случае, если вызов делается в разных ветках конструкции *if-else* с разными параметрами. Так как мы не сохраняем значения параметров, такие вызовы в последовательности API будут идентичными.

Нет практического смысла в том, чтобы модель умела генерировать в некоторых ситуациях несколько одинаковых вызовов подряд, а потому такие повторения сокращаются до одного вызова API.

Этот шаг не влияет на количество данных, но улучшает их качество.

Повторяющиеся вызовы не сокращаются при создании набора данных dataset-original.

3.1.5. Фильтрация по уникальности

В данных содержится неожиданно много повторений, уникальных пар — примерно 30% из всех. Заметим, что пары считаются разными даже если английские описания совпадают, а последовательности API — нет, и наоборот.

Были обнаружены несколько источников возникновения повторений:

- сгенерированные автоматически код и комментарии (например библиотеками для создания пользовательского интерфейса, таких как Swing);
- намеренные потерь информации во время сбора данных (например, не сохраняются значения параметров вызовов функций; также на предыдущем шаге были удаляются многие вызовы API, не вошедшие в словарь);
- код библиотек, скопированный в код пользовательского приложения вместо обыкновенного импорта.

Мы применяем этот шаг при создании обоих наборов данных по нескольким причинам. Во-первых, не имеет практического смысла иметь в тренировочном наборе повторения. Во-вторых, велик шанс попадания повторяющихся пар одновременно в тренировочный и тестовый наборы. В таком случае тестирование окажется невалидным, поскольку модель будет оценена на тренировочных данных, что в корне неверно.

Этот шаг значительно уменьшает количество пар (см. таблицу 1).

3.1.6. Фильтрация по популярности слов

Для повторения условий исходной статьи при создании набора данных dataset-original используются словари из 10 000 самых популярных английских слов и 10 000 самых популярных вызовов API, с которыми умеет работать Bayou.

Но в связи с тем, что эти методы не самые распространенные в наших данных, в словарях появляются очень редкие и специфичные слова и вызовы API. Поэтому при создании улучшенного набора данных dataset-bayou мы ограничиваем размер английского словаря 3 000 самых популярных слов, а размер словаря вызовов API — 5 000.

Если после фильтрации не остаётся слов в английском описании или в последовательности API, мы полностью удаляем такую пару.

Влияние каждого из шагов предобработки на количество данных и финальный размер наборов данных представлены в таблице 1.

Таблица 1: Количество данных после каждого из шагов фильтрации

Шаг фильтрации	Количество данных после шага фильтрации	
	dataset-original	dataset-bayou
Звёзды	17 631 306	10 100 706
Язык	—"—	9 510 103
Словарь	7 626 318	1 687 537
Повторяющиеся вызовы	—"—	—"—
Уникальность	2 352 032	555 156
Популярность	2 010 172	512 183

3.2. Технические подробности обучения

Для обучения моделей используется фрэймворк OpenNMT-tf[12], специализированный именно для работы с моделями нейронного машинного перевода. OpenNMT-tf разработан поверх другого популярного фрэймворка — TensorFlow [22], который имеет API для работы из Java кода. OpenNMT-tf поддерживает совместимость с этим API, а поэтому обученные модели можно удобно использовать в Java коде нашего плагина.

Для обучения моделей используется стохастический градиентный спуск [23]. Как и в оригинальной статье, модель обучается минимизации следующей целевой функции:

$$L(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}(-\log p_\theta(y_{it}|x_i))$$

где N — количество тренировочных пар, T — длина i-ой целевой последовательности, $p_\theta(y_{it}|x_i)$ — правдоподобие того, что текущая модель с параметрами θ сгенерирует верное t-ое целевое слово в i-ой целевой последовательности, получив на вход соответствующую i-ую входную последовательность.

Иными словами, минимизируется отрицательная логарифмическая функция правдоподобия на всех тренировочных данных.

Из набора данных dataset-original 10 000 пар данных выделяются для тестирования, из dataset-bayou — 10 000 пар для тестирования и выбора гиперпараметров обучения, а 10 000 пар — для валидации на выбранных гиперпараметрах.

[12] `https://github.com/OpenNMT/OpenNMT-tf`

Мы обучаем 3 модели, данные о которых представлены в таблице 2 и подробно описаны далее.

Таблица 2: Модели

Название	Набор данных	Гиперпараметры
data-orig-params-orig	dataset-original	Исходные
data-bayou-params-orig	dataset-bayou	Исходные
data-bayou-params-optim	dataset-bayou	Оптимизированные

Сначала мы обучаем модель **data-orig-params-orig** на наборе данных dataset-orginal с параметрами, описанными в оригинальной статье. В качестве реализации рекуррентной нейронной сети берётся двухслойное GRU [17] с 1000 скрытых нейронов, размер векторного представления выставляется равным 120, размер батча – 200.

В оригинальной статье указано, что в качестве оптимизатора обучения используется AdaDelta [24], но не указаны её параметры. При обучении со стандартными параметрами из статьи, предлагающей AdaDelta (rho=0.95, epsilon=1e-6), модель перестаёт обучаться после короткого промежутка времени, скорее всего из-за неограниченного роста нормы градиента.

Поэтому мы обучаем заново описанную выше модель с помощью оптимизатора Adam [25] со стандартными значениями и стратегии сокращения темпа обучения под названием «Noam decay» [26], процесс работы которой — линейное увеличение темпа обучения в начале обучения («разогрев»), а затем экспоненциальное его уменьшение. Мы выставляем число шагов разогрева в 2 000, а экспоненциальное уменьшение — каждые 8 шагов после окончания этапа разогрева. Такая стратегия уменьшения темпа обучения позволяет моделям хорошо обучаться и не приводит к экспоненциальному росту нормы градиента.

Сравнение числовых значений функции потерь для моделей, обучаемых с помощью AdaDelta и Adam приведено на рисунках 5 – 6 (обратите внимание, что рисунки представлены в разных масштабах). Видно, что обучаемая с помощью AdaDelta модель перестаёт учиться примерно на шаге 50 700, когда значение её функции потерь равно 4.997. На шаге 218 100 значение функции потерь равно 4.947. При этом норма градиента на некоторых шагах достигает значений, больших 200. Напротив, модель, обучаемая с помощью Adam на шаге 50 700 имеет значение функции потерь 2.596, а на шаге 218 100 — 1.577, а нормы градиентов ни на каком шаге не превышают 30.

Приняв во внимание существенную разницу в производительности, мы все модели обучаем с помощью Adam.

Модель **data-bayou-params-orig** мы обучаем тем же образом и с теми же параметрами, что и предыдущую, но уже на наборе данных dataset-bayou.

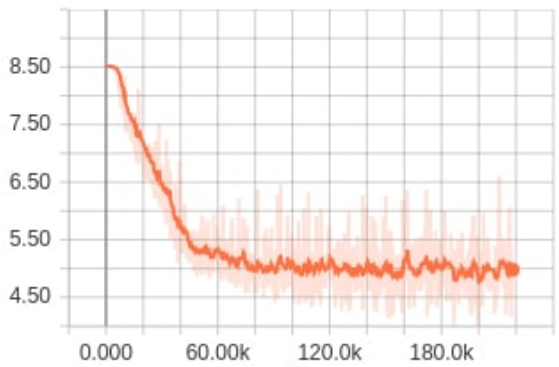

Рис. 5: Функция потерь (AdaDelta)

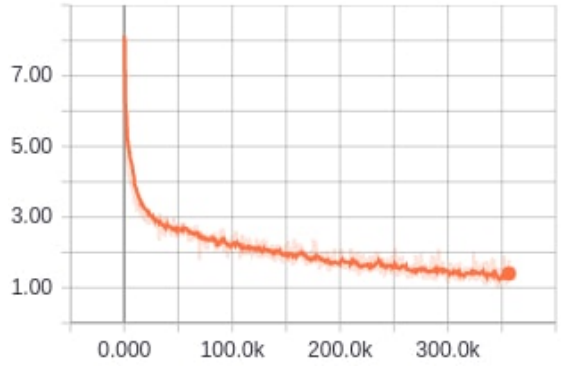

Рис. 6: Функция потерь (Adam)

Затем мы эмпирически подбираем более оптимальные параметры и обучаем финальную модель **data-bayou-params-optim**. В качестве реализации рекуррентной нейронной сети берётся LSTM [27] — более сложная реализация RNN нежели GRU, но чаще использующаяся в контексте машинного перевода. И в кодере, и в декодере 3 слоя с 1000 нейронов в каждом, размер векторного представления — 120, размер батча — 110.

Для регуляризации модели используется рекуррентный dropout [28] с параметром 0.4. Это значит, что к выходу каждого слоя, кроме входного, применяется следующее преобразование: каждое из значений выходного вектора либо обнуляется с вероятностью 0.4, либо не изменяется с вероятностью 0.6.

Результаты обучения моделей обсуждаются в секции 5.1.

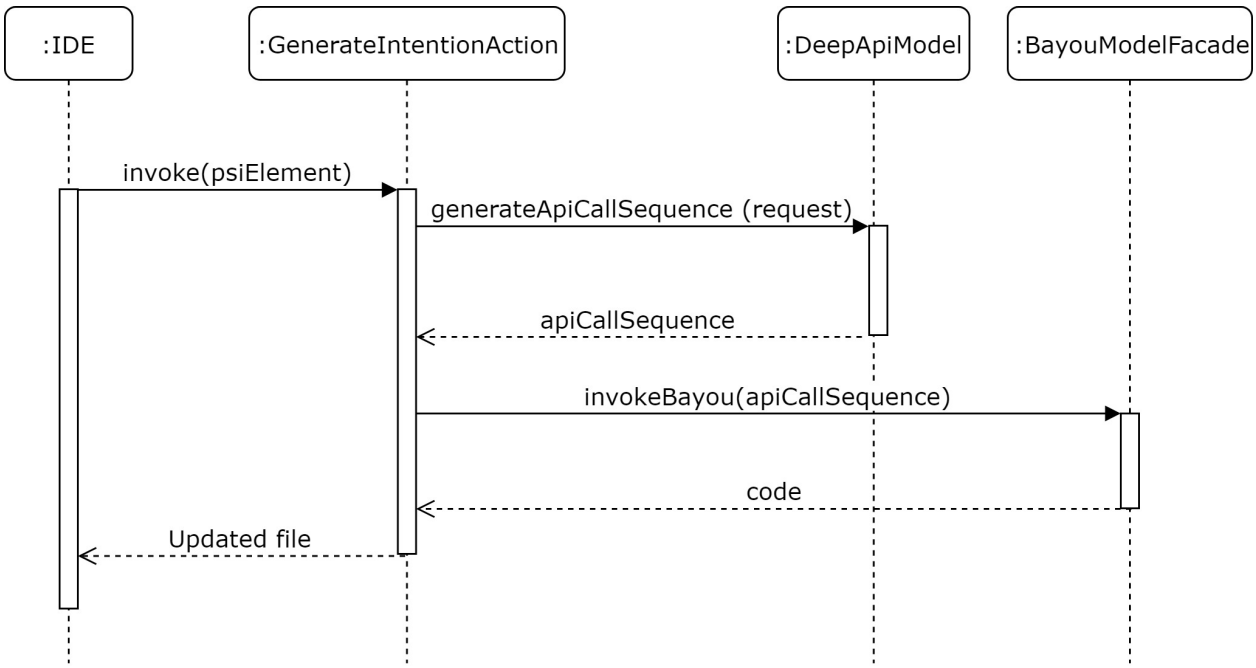

Рис. 7: Процесс работы плагина

4. Интеграция в IDE IntelliJ IDEA

Для удобного использования модели конечными пользователями она интегрируется в среду разработки, в качестве которой была выбрана IntelliJ IDEA — популярная IDE, предоставляющая широкие возможности для создания плагинов[13].

Высокоуровневый процесс работы плагина представлен на диаграмме последовательностей на рисунке 7.

Пользователь набирает запрос внутри комментария в своём коде, после чего возникает подсказка, предлагающая сгенерировать код. Она реализована как Intention Action[14]. Внешний вид представлен на рисунке 13.

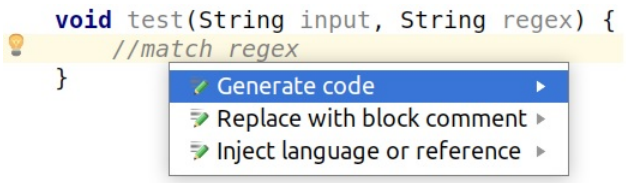

Рис. 8: Пользовательский интерфейс

Эта подсказка реализована классом GenerateIntentionAction, который наследует специальный интерфейс IntentionAction. Когда пользователь запускает подсказку, экземпляр этого класса получает от IDE необходимую информацию о месте запуска.

Текст комментария передаётся классу DeepApiModel, где проводится его обработка, аналогичная обработке при сборе данных: удаляются знаки препинания, символы

[13] https://www.jetbrains.com/help/idea/plugin-development-guidelines.html
[14] https://www.jetbrains.com/help/idea/intention-actions.html

переводятся в нижний регистр, предложение разделяется на слова. Затем вызывается экспортированная TensorFlow модель и возвращается результат её работы — список вызовов API.

Из списка вызовов выделяются отдельно имена типов и имена методов, которые можно передавать Bayou. Однако было замечено, что иногда DeepAPI рекомендует большое количество разных типов и методов, а Bayou не способен генерировать код при слишком большом количестве входных признаков. Поэтому на практике из рекомендаций DeepAPI в Bayou передаётся не более чем 3 имени типа и 3 имени метода.

Как уже упоминалось, в качестве реализации алгоритма Bayou используется библиотека bayou-implementation, разработанная Владиславом Танковым.

Программный код, полученный в результате работы Bayou, встраивается в исходный файл. Результат генерации для примера из рисунка 8 представлен на рисунке 9.

```
void test(String input, String regex) {
    //match regex
    {
        boolean b1;
        Pattern p1;
        Matcher m1;
        p1 = Pattern.compile(regex);
        m1 = p1.matcher(regex);
        b1 = m1.matches();
        return;
    }
}
```

Рис. 9: Результат работы плагина

Исходный код плагина находится в открытом доступе[15].

[15] https://github.com/AwesomeLemon/deep-api-bayou-intellij-plugin

5. Апробация

5.1. Внутренние метрики модели DeepAPI

В оригинальной статье DeepAPI для оценки производительности модели используется BLEU [29] — метрика, предназначенная для оценки моделей машинного перевода. Она измеряет сходство сгенерированного моделью выхода с целевой последовательностью, написанной человеком. Для этого сравнивается процент совпавших n-грамм с поправкой на длину сгенерированной последовательности (т.к. более короткие последовательности имеют больше совпадающих n-грамм).

Результаты обучения моделей представлены в таблице 3.

Таблица 3: Результаты обучения моделей

Модель	BLEU
data-orig-params-orig	3.46
data-bayou-params-orig	8.12
data-bayou-params-optim	12.82

Модель **data-orig-params-orig** отличается от модели **data-bayou-params-orig** только предобработкой, а поэтому улучшение BLEU на 4.66 показывает эффективность предложенных шагов предобработки.

Лучше всего работает модель **data-bayou-params-optim** (именно её мы используем в плагине), но её BLEU 12.82 далёк от BLEU 54.42 из оригинальной статьи. Однако стоит обратить внимание, что авторы оригинальной статьи не проверяли данные на уникальность, а так как уникальны примерно 30% из них, а тестовые данные выбираются случайно, логично предположить, что в тестовой выборке авторов 70% пар совпадали с парами в тренировочной выборке. Поэтому дополнительно мы оцениваем модель **data-bayou-params-optim** на 10 000 пар данных, из которых 30% — из валидационного набора, а 70% — из тренировочного. Полученный BLEU 47.73 довольно близок к оригинальному 54.42, что поддерживает наше предположение о попадании в оригинальной статье тренировочных данных в тестовый набор.

Также в оригинальной модели приводятся примеры работы модели на 30 запросах. Чтобы показать, что наша модель работает на схожем уровне, мы сообщаем её производительность на 25 из этих запросов, убрав 5 запросов, необходимые вызовы для которых не содержатся в списке вызовов, с которыми умеет работать Bayou, а потому и в словаре нашей модели.

Сравнение представлено в таблице 5. Для каждого запроса наша модель генерирует 10 наиболее вероятных рекомендаций. Столбец «Ранг» отмечает, на каком по счёту месте моделью сгенерирован первый верный ответ. Видно, что в целом наша

модель работает на том же уровне, что и оригинальная.

Как уже было описано в разделе 3.2, в процессе обучения модели минимизируют отрицательное логарифмическое правдоподобие. При этом значение функции потерь в течение обучения, как и ожидается, уменьшается на тренировочной выборке, однако на тестовой оно увеличивается. Но так как при этом растёт и значение метрики BLEU на тестовой выборке, мы делаем вывод, что модели не переобучаются.

Чтобы показать, что метрике BLEU в данной ситуации следует доверять больше, чем правдоподобию, рассмотрим конкретный пример.

В тестовых данных английскому описанию «strips the comment after a line if present» соответствует последовательность API из первой строки таблицы 4. На шаге 195 109 обучения модель **data-bayou-params-optim** генерирует по английской аннотации последовательность API из второй строки, значение функции потерь для которой — 0.74, а BLEU — 66.87 . На шаге 471 817 модель генерирует последовательность API из третьей строки, для которой значение функции потерь — 0.78, а BLEU — 75.98.

Таблица 4: Пример для сравнения BLEU и функции потерь

Источник	**Последовательность API**
Тестовые данные	String.indexOf String.substring
Сгенерировано на шаге 195 109	String.startsWith String.substring String.endsWith String.length String.substring
Сгенерировано на шаге 471 817	String.length String.charAt String.substring

Увеличение BLEU объясняется отказом модели от генерации лишних вызовов String.startsWith и String.endsWith. Но вместе с уменьшением вероятности их генерации уменьшается и вероятность генерации верного вызова String.indexOf. Возможное объяснение лежит в том, что модель, обученная в течении большего числа итераций, даёт меньший вес редким вызовам, что позволяет ей генерировать более чистые последовательности API.

Таблица 5: Работа на популярных запросах

Запрос	**DeepAPI**		**data-bayou-params-optim**	
	Ранг	Вызовы API	Ранг	Вызовы API
convert int to string	2	Integer.toString	1	Integer.toString
convert string to int	1	Integer.parseInt String.toCharArray Character.digit	1	Integer.parseInt
append strings	1	StringBuilder.append StringBuilder.toString	2	StringBuilder.new StringBuilder.append StringBuilder.toString
get current time	10	System.currentTimeMillis Timestamp.new	4	Date.new Date.getTime
parse datetime from string	1	SimpleDateFormat.new SimpleDateFormat.parse	1	SimpleDateFormat.new SimpleDateFormat.parse
test file exists	1	File.new File.exists	1	Properties.getProperty File.new File.exists
open a url	1	URL.new URL.openConnection	2	URL.openStream
open file dialog	1	JFileChooser.new JFileChooser.showOpenDialog JFileChooser.getSelectedFile	3	JFileChooser.new FileNameExtensionFilter.new JFileChooser.setFileFilter JFileChooser.showOpenDialog JFileChooser.getSelectedFile File.getAbsolutePath
get files in folder	3	File.new File.list File.new File.isDirectory	1	File.listFiles ArrayList.new File.isDirectory List.add
match regular expressions	1	Pattern.compile Pattern.matcher Matcher.group	1	Pattern.matcher Matcher.matches
generate md5 hash code	1	MessageDigest.getInstance MessageDigest.update MessageDigest.digest	1	MessageDigest.getInstance MessageDigest.update MessageDigest.digest
generate random number	1	Random.new Random.nextInt	2	Random.new Random.nextInt
round a decimal value	1	Math.floor Math.pow Math.round	—	—
execute sql statement	1	Connection.prepareStatement PreparedStatement.execute PreparedStatement.close	1	Connection.prepareStatement PreparedStatement.execute
create file	3	File.exists File.createNewFile	1	File.new File.exists File.createNewFile

Запрос	**DeepAPI**		**data-bayou-params-optim**	
	Ранг	Вызовы API	Ранг	Вызовы API
copy file	2	FileInputStream.new FileOutputStream.new FileInputStream.read FileOutputStream.write FileInputStream.close FileOutputStream.close	1	FileInputStream.new FileOutputStream.new FileInputStream.read FileOutputStream.write FileInputStream.close FileOutputStream.close
copy a file and save it -to your destination path	1	FileInputStream.new FileOutputStream.new FileInputStream.getChannel FileOutputStream.getChannel FileChannel.size FileChannel.transferTo FileInputStream.close FileOutputStream.close FileChannel.close FileChannel.close	1	FileInputStream.new FileOutputStream.new FileInputStream.read FileOutputStream.write FileInputStream.close FileOutputStream.close
delete files and folders -in a directory	1	File.isDirectory File.list File.new File.delete	2	File.isDirectory File.list File.new File.delete
reverse a string	2	StringBuffer.new StringBuffer.reverse	1	StringBuffer.new StringBuffer.reverse StringBuffer.toString
create socket	1	ServerSocket.new ServerSocket.bind	10	Socket.new InetSocketAddress.new Socket.connect
rename a file	1	File.renameTo File.delete	1	File.new File.renameTo
download file from url	1	URL.new URL.openConnection URLConnection.getInputStream BufferedInputStream.new	4	URL.new URL.openStream InputStreamReader.new BufferedReader.new StringBuilder.new BufferedReader.readLine StringBuilder.append StringBuilder.toString
serialize an object	1	ObjectOutputStream.new ObjectOutputStream.writeObject ObjectOutputStream.close	1	ByteArrayOutputStream.new ObjectOutputStream.new ObjectOutputStream.writeObject ByteArrayOutputStream.toByteArray
read binary file	1	DataInputStream.new DataInputStream.readInt DataInputStream.close	3	FileInputStream.new DataInputStream.new InputStreamReader.new BufferedReader.new BufferedReader.readLine DataInputStream.close
parse xml	1	InputSource.new DocumentBuilder.parse	—	—

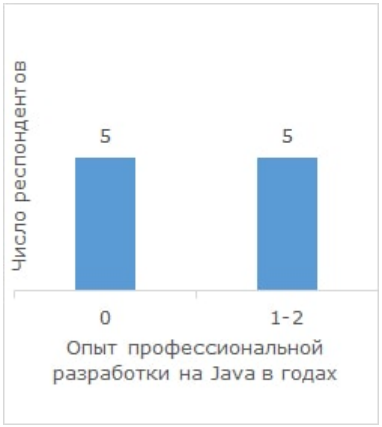

Рис. 10

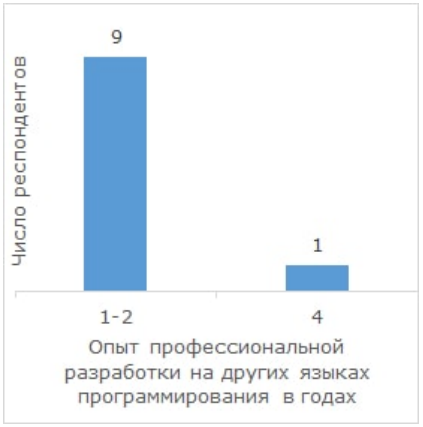

Рис. 11

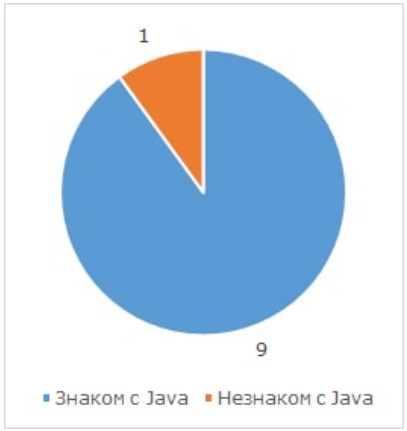

Рис. 12

5.2. Внешняя апробация

5.2.1. Описание эксперимента

Мы попросили 10 начинающих разработчиков выполнить 5 небольших заданий с помощью плагина, а затем оценить его с разных точек зрения.

Респонденты сообщали в годах их опыт профессионального программирования на Java и опыт профессионального программирования на других языках. Также мы просили указать, знакомы ли они в принципе с Java, то есть написали ли хотя бы строчку кода на этом языке.

Распределение ответов на эти вопросы представлено на рисунках 10 – 12.

50% респондентов не имели опыта профессиональной работы с Java, другие 50% имели опыт, не превышающий 2 лет. Также 90% занимались профессиональной разработкой на других языках 2 года и меньше. При этом знакомы с Java 90% респондентов.

Информация о заданиях, предлагаемых пользователям, представлена в таблице 6. Обратим внимание, что задания сформулированы пространно, где это возможно, и на русском языке, чтобы случайно не дать пользователям точную формулировку запроса, а позволить им сформулировать его самостоятельно.

Пользователям предлагалось самостоятельно протестировать написанный с помощью плагина код.

После работы пользователи оценивали плагин по нескольким параметрам:

- удобство пользовательского интерфейса (по шкале от 1 до 5, где 1 — очень плохо, 5 — очень хорошо);
- полезность плагина (по шкале от 1 до 5);
- скорость работы плагина за исключением первого запуска (по шкале от 1 до 5).

Таблица 6: Задания для тестирования

Название	**Описание**	**Тест**
Регулярные выражения	Напишите функцию, которая получает строку, регулярное выражение в виде строки и возвращает, удовлетворяет ли строка этому регулярному выражению.	Написанная функция должна вернуть true, получив на вход строки «123», «\\d+».
Чтение файла	Напишите функцию, которая получает путь к текстовому файлу в виде строки, считывает все строки из него и объединяет их в одну строку.	Передайте путь к любому текстовому файлу на вашем диске, выведите результат работы функции на экран и проверьте, что он совпадает с содержанием файла.
Файлы в папке	Напишите функцию, которая получает путь к папке и печатает названия всех файлов и папок в ней.	Передайте путь к непустой папке на вашем диске (например, “C:\\” на Windows или “/” на Linux) и проверьте, что вывод программы соответствует её содержанию.
Число из строки	Напишите функцию, которая конвертирует строку в целое число	Получив на вход строку «123», функция должна вернуть число 123
Строка из числа	Напишите функцию, которая конвертирует целое число в строку	Получив на вход число 123, функция должна вернуть строку «123»

Также пользователи отмечали, для каких из 5 заданий полезный код был сгенерирован с первого раза, а для каких заданий плагин не смог сгенерировать релевантный код.

5.2.2. Анализ результатов

На рисунках 13— 15 представлены гистограммы оценок респондентами пользовательского интерфейса плагина, полезности плагина, скорости его работы.

Рис. 13: Оценка пользовательского интерфейса

Рис. 14: Оценка полезности плагина

Рис. 15: Оценка скорости работы

Как видно, пользовательский интерфейс посчитали очень удобным: ему дали максимальную оценку 70% человек.

Полезность плагина посчитали неудовлетворительной 20% респондентов, большая часть осталась им довольна. Выборочное среднее оценки полезности равно 3.4.

Респонденты в подавляющем большинстве оценили скорость плагина как удовлетворительную. По нашим наблюдениям, большая часть времени работы плагина уходит на генерацию кода алгоритмом Bayou. В будущем можно рассмотреть способы ускорения этого процесса.

На рисунке 16 показано для каждого задания сколько пользователей смогли получить релевантный код с первого раза, сколько — не с первого, а сколько не смогли получить релевантный код вообще.

Видно, что большинству пользователей плагин помог с первого раза решить задания «Регулярные выражения», «Число из строки», «Строка из числа». Плагин показал себя хуже в заданиях «Чтение файла», «Файлы в папке». Возможная причина лежит в том, что эти задания требовали вывода информации на экран, который в Java реализован через использование системной константы «System.out». Но модель DeepAPI генерирует только имена типов и методов, релевантные запросу, а Bayou — наиболее вероятный код, их использующий. Ни одна из моделей не работает с именованными системными константами, а потому технически не может использовать «System.out». Это, безусловно, является ограничением предложенного подхода.

Также данные на этой диаграмме показывают, что в большинстве случаев если плагин генерирует полезный код, то с первой попытки, что, несомненно, является

Рис. 16: Производительность плагина в тестовых заданиях

важным фактором для перспектив его практического применения.

Таким образом, в целом оценка плагина показала его полезность, а соответственно и релевантность предложенного подхода генерации кода.

Заключение

В рамках данной работы были достигнуты следующие результаты.

- Изучены существующие алгоритмы в области генерации программного кода, выбраны два наиболее перспективных — DeepAPI и Bayou.
- Реализованы программные инструменты для сбора данных из проектов с открытым исходным кодом, позволившие собрать данные для обучения модели DeepAPI.
- Подобраны оптимальные параметры для обучения модели DeepAPI, реализована предварительная обработка данных, включающая в себя фильтрацию по языку, проверку уникальности на уровне пар данных и сокращение повторений на уровне слов.
- Разработан плагин на Java для интеграции комбинированной модели в среду разработки IntelliJ IDEA, в том числе реализовано взаимодействие с уже существующей реализацией алгоритма Bayou.
- Плагин протестирован пользователями, получены по большей части положительные отзывы, указывающие, что плагин позволяет успешно решать большинство рассмотренных заданий, имеет удобный интерфейс, но работает не очень быстро; человеконезависимая метрика BLEU демонстрирует хороший результат обучения модели DeepAPI, подтверждающий эффективность предложенной предобработки и подобранных параметров обучения.

Список литературы

[1] Program synthesis using natural language / Aditya Desai, Sumit Gulwani, Vineet Hingorani [и др.] // Proceedings of the 38th International Conference on Software Engineering / ACM. 2016. С. 345–356.

[2] Gulwani Sumit, Marron Mark. Nlyze: Interactive programming by natural language for spreadsheet data analysis and manipulation // Proceedings of the 2014 ACM SIGMOD international conference on Management of data / ACM. 2014. С. 803–814.

[3] Latent predictor networks for code generation / Wang Ling, Edward Grefenstette, Karl Moritz Hermann [и др.] // arXiv preprint arXiv:1603.06744. 2016.

[4] Bierhoff Kevin. Api protocol compliance in object-oriented software. Ph.D. thesis: Carnegie Mellon University. 2009.

[5] Stylos Jeffrey, Myers Brad A. Mica: A web-search tool for finding api components and examples // Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on / IEEE. 2006. С. 195–202.

[6] Mining succinct and high-coverage API usage patterns from source code / Jue Wang, Yingnong Dang, Hongyu Zhang [и др.] // Proceedings of the 10th Working Conference on Mining Software Repositories / IEEE Press. 2013. С. 319–328.

[7] Xie Tao, Pei Jian. MAPO: Mining API usages from open source repositories // Proceedings of the 2006 international workshop on Mining software repositories / ACM. 2006. С. 54–57.

[8] Uhler Richard, Dave Nirav. Smten: automatic translation of high-level symbolic computations into SMT queries // International Conference on Computer Aided Verification / Springer. 2013. С. 678–683.

[9] Murali Vijayaraghavan, Chaudhuri Swarat, Jermaine Chris. Bayesian Sketch Learning for Program Synthesis // arXiv preprint arXiv:1703.05698. 2017.

[10] Jungloid mining: helping to navigate the API jungle / David Mandelin, Lin Xu, Rastislav Bodík [и др.] // ACM Sigplan Notices / ACM. Т. 40. 2005. С. 48–61.

[11] Component-based synthesis for complex APIs / Yu Feng, Ruben Martins, Yuepeng Wang [и др.] // ACM SIGPLAN Notices. 2017. Т. 52, № 1. С. 599–612.

[12] Learning phrase representations using RNN encoder-decoder for statistical machine translation / Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre [и др.] // arXiv preprint arXiv:1406.1078. 2014.

[13] Fowkes Jaroslav, Sutton Charles. Parameter-free probabilistic API mining across GitHub // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering / ACM. 2016. С. 254–265.

[14] Friedman Nir. The Bayesian structural EM algorithm // Proceedings of the Fourteenth conference on Uncertainty in artificial intelligence / Morgan Kaufmann Publishers Inc. 1998. С. 129–138.

[15] Deep API learning / Xiaodong Gu, Hongyu Zhang, Dongmei Zhang [и др.] // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering / ACM. 2016. С. 631–642.

[16] Schuster Mike, Paliwal Kuldip K. Bidirectional recurrent neural networks // IEEE Transactions on Signal Processing. 1997. Т. 45, № 11. С. 2673–2681.

[17] Bahdanau Dzmitry, Cho Kyunghyun, Bengio Yoshua. Neural machine translation by jointly learning to align and translate // arXiv preprint arXiv:1409.0473. 2014.

[18] Yin Pengcheng, Neubig Graham. A syntactic neural model for general-purpose code generation // arXiv preprint arXiv:1704.01696. 2017.

[19] Hack Michel. Petri net language. 1976.

[20] Bayou integration. `https://github.com/ml-in-programming/bayou-integration`. Accessed: 09.05.2018.

[21] Project Structure Interface. `http://www.jetbrains.org/intellij/sdk/docs/basics/project_structure.html`. Accessed: 09.05.2018.

[22] TensorFlow: A System for Large-Scale Machine Learning. / Martín Abadi, Paul Barham, Jianmin Chen [и др.] // OSDI. Т. 16. 2016. С. 265–283.

[23] Kiefer Jack, Wolfowitz Jacob. Stochastic estimation of the maximum of a regression function // The Annals of Mathematical Statistics. 1952. С. 462–466.

[24] Zeiler Matthew D. ADADELTA: an adaptive learning rate method // arXiv preprint arXiv:1212.5701. 2012.

[25] Kingma Diederik P, Ba Jimmy. Adam: A method for stochastic optimization // arXiv preprint arXiv:1412.6980. 2014.

[26] Attention is all you need / Ashish Vaswani, Noam Shazeer, Niki Parmar [и др.] // Advances in Neural Information Processing Systems. 2017. С. 6000–6010.

[27] Gers Felix A, Schmidhuber Jürgen, Cummins Fred. Learning to forget: Continual prediction with LSTM. 1999.

[28] Dropout: A simple way to prevent neural networks from overfitting / Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky [и др.] // The Journal of Machine Learning Research. 2014. Т. 15, № 1. С. 1929–1958.

[29] BLEU: a method for automatic evaluation of machine translation / Kishore Papineni, Salim Roukos, Todd Ward [и др.] // Proceedings of the 40th annual meeting on association for computational linguistics / Association for Computational Linguistics. 2002. С. 311–318.