Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра алгоритмических языков

Жуков Павел Николаевич

Система поиска информации в программных репозиториях

МАГИСТЕРСКАЯ ДИССЕРТАЦИЯ

Научный руководитель:
к.ф.-м.н., доцент Головин Игорь Геннадьевич

Москва, 2021

Аннотация

Современная разработка ПО помимо написания программного кода подразумевает также создание большого количества артефактов, таких как данные системы контроля версий, системы отслеживания ошибок и т.д. Для того, чтобы человек любого уровня подготовки мог на естественном языке узнавать информацию о программном репозитории, необходимо иметь систему поиска этой информации, основанную на использовании исходного кода совместно с артефактами его разработки. В данной работе был впервые предложен подход к построению подобной системы с возможностью возврата кратких ответов на естественном языке. Подход также был реализован в виде программного средства с использованием современных моделей машинного обучения, таких как BERT и FastText.

Содержание

1 Введение

Программное обеспечение стало неотъемлемой частью нашего существования в современном мире, проникнув во все сферы жизни. С увеличением сложности и роли программного обеспечения развиваются и методы его разработки.

Во-первых, за последние десятилетия появилось множество программных средств, облегчающих ведение процесса разработки. Список включает системы управления версиями, позволяющие координировать работу разработчиков и отслеживать изменения в файлах; крупные веб-сервисы для хостинга проектов и совместной разработки; системы отслеживания ошибок, позволяющие учитывать и контролировать возникающие в программах ошибки и неполадки, пожелания пользователей, а также следить за процессом работы над этими ошибками и пожеланиями; различные средства для непрерывной интеграции, доставки и развёртывания, много из которых появилось за последние годы; и многое другое. В результате работы этих систем получаются так называемые артефакты — побочные продукты производства программного обеспечения. Объединив их в единую систему и связав с исходным кодом, можно значительно увеличить эффективность поиска информации в проекте.

Во-вторых, проникновение информационных технологий в большинство сфер жизни привело к появлению различных методологий разработки и, соответственно, специальностей, не требующих глубокого понимания кода: специалисты по ручному и автоматизированному тестированию, функциональные и бизнес-аналитики, менеджеры проектов, DevOps-инженеры и другие.

Существование таких систем и специальностей подразумевает, что должен быть способ для далёких от разработки людей получать знания о технических аспектах проекта, собранные в один источник информации, удобный в использовании. Такой способ также мог бы быть использован разработчиками, которые только присоединились к команде и ещё не знакомы со всеми деталями проекта. Более того, даже опытные разработчики имели бы возможность пользоваться подобной системой при возникновении затруднений, ведь общедоступные ресурсы, такие как Stack Overflow и Google, не содержат информации о внутренних проектах компании и их устройстве, что усложняет процесс поиска.

Однако объединить дополнительную информацию с кодом и предоставить эффективный и удобный поиск в получившейся связке является нетривиальной задачей, которая до этого исследовалась лишь по частям. В данной работе предложен подход, который позволяет организовать поиск на естественном языке по программному репозиторию, объединённому с артефактами системы контроля версий.

2 Постановка задачи

Целью работы является разработка системы поиска информации в программном репозитории, позволяющей задавать вопросы на естественном языке и использовать информацию из артефактов программного обеспечения, в частности системы контроля версий. При возможности краткого ответа, система должна уметь отвечать на естественном языке, иначе — выдавать список релевантных фрагментов кода с соответствующей информацией из системы контроля версий.

3 Обзор существующих решений

Решаемую задачу можно разделить на три разных составляющих:

1. Поиск нужного фрагмента в исходном коде;
2. Объединение исходного кода с информацией, извлекаемой из артефактов, полученных в процессе его разработки;
3. Получение ответа на заданный вопрос на естественном языке.

Рассмотрим существующие решения для каждой из этих задач и определим их преимущества и недостатки в применении к поставленной задаче.

3.1 Поиск фрагментов исходного кода

На данный момент существует четыре основных решения этой задачи. Все они используют нейронные сети как с применением обучения без учителя, так и с применением обучения с учителем. Тем не менее, суть всех решений одна — построить такие векторные представления запроса и кода, чтобы семантически близкие конструкции были близки и в векторном пространстве.

3.1.1 Code-Description Embedding Neural Network (CODEnn)

CODEnn [1] — это первая работа, в которой авторы попытались переосмыслить задачу поиска в коде, используя глубокое обучение вместо стандартных подходов информационного поиска. Их мотивация заключалась в том, что этим методам не хватает глубокого понимания семантики запроса и исходного кода, так как они используют лишь сходство токенов. Это вполне разумно, так как конструкции, встречаемые в исходном коде, из-за их специфики часто могут не соответствовать конкретным словам, указанным в запросе.

Нейронная сеть состоит из трёх модулей (рисунок 1): преобразование кода в векторы, преобразование описаний в векторы и модуль для сопоставления построенных представлений. В первых двух модулях используется несколько рекуррентных нейронных сетей [2], позволяющих учитывать порядок токенов во время построения их векторных представлений. Ключевая особенность подхода авторов заключается в том, что для обучения нейронной сети в качестве описаний методов на естественном языке они используют комментарии javadoc — средства для документации кода на языке Java. CODEnn обучается на кортежах из трёх элементов — фрагменте кода, правильном описании и произвольным неправильном описании. Это позволяет максимизировать

сходство корректных пар и минимизировать сходство некорректных. В итоге при поступлении запроса он преобразуется в вектор тем же способом, что и описания методов, и сравнивается с уже проиндексированным кодом с помощью косинусного расстояния.

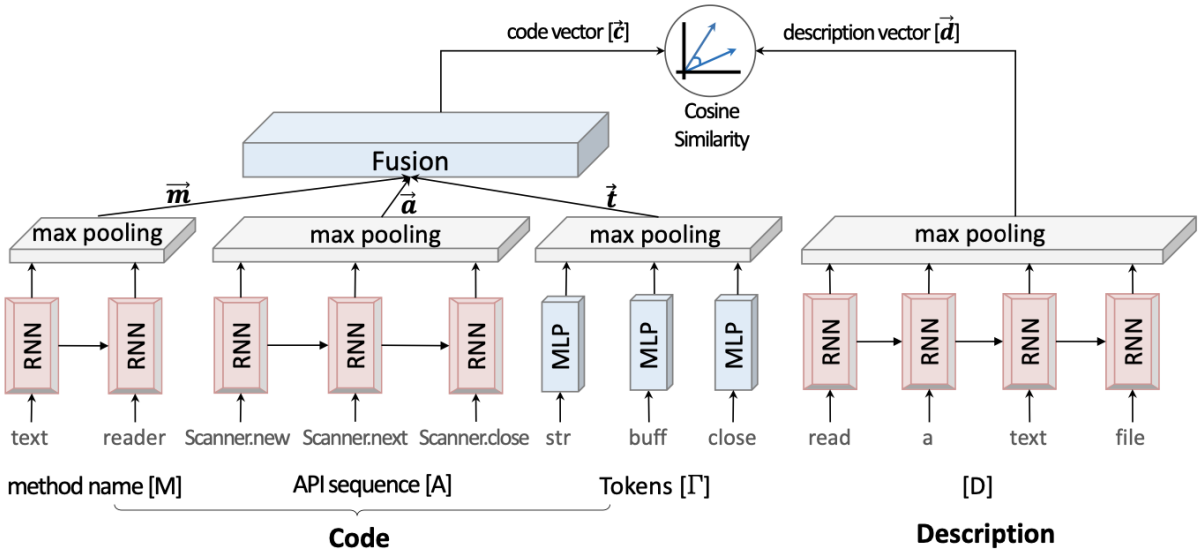

Рис. 1: Архитектура CODEnn [1]

Этот подход стал лидирующим среди остальных решений для поиска в коде на момент его публикации. Глубокое обучение позволило научить нейронную сеть различать запросы с одним и тем же набором токенов, но в разном порядке, и имеющие разный смысл. Тем не менее, этот подход имеет свои недостатки.

Во-первых, его сложно реализовать, так как он включает в себя несколько рекурентных нейронных сетей.

Во-вторых, как отмечают сами авторы, он требует значительного набора данных для обучения. Авторы использовали собранный собственноручно набор из 18 миллионов методов на языке Java. Как следствие, это требует длительного обучения в течение нескольких дней и серьёзных вычислительных мощностей, которых может не быть у желающих воспользоваться подобным методом.

В-третьих, точность работы CODEnn не позволяет выбрать один конкретный фрагмент кода в качестве ответа, что приводит к тому, что правильный фрагмент может находиться на дальних позициях ранжированного списка ответов. Это не значит, что CODEnn работает плохо, так как эта проблема характерна и для других решений в этой области, и связана с тем, что интерпретация правильности того или иного ответа не однозначна. Но этот факт заставляет усомниться в необходимости такого сложного и ресурсоёмкого подхода, если аналоги позволяют добиться похожего набора результатов, но, возможно, с другим ранжированием.

3.1.2 Neural Code Search (NCS)

Модель NCS [3] была представлена примерно в то же время, что и предыдущая модель CODEnn. Главное отличие этого подхода заключается в том, что он использует обучение без учителя. Авторы основывают решение на гипотезе о том, что токены исходного кода содержат достаточно информации для осуществления информационного поиска.

Для извлечения токенов авторы используют названия методов, вызовы методов, так как они дают представление о том, что происходит в коде, перечисляемые типы, строковые литералы и комментарии. Названия переменных не извлекаются умышленно, так как они варьируются у разных разработчиков, а также их семантика обычно покрывается остальными токенами. Далее применяется одна из реализаций модели Word2vec [4] под названием FastText [5]. На предоставленном корпусе она обучается находить векторные представления для каждого токена на основе его соседей в фиксированном окне некоторого размера. Вес каждого документа (метода) представляется суммой векторных представлений его токенов, взвешенных с помощью меры TF-IDF [6], что позволяет уменьшить вес токенов, появляющихся слишком часто, что характерно для конструкций исходного кода. Запрос преобразуется как среднее векторных представлений его слов, причём слова, не встречаемые в корпусе, исключаются.

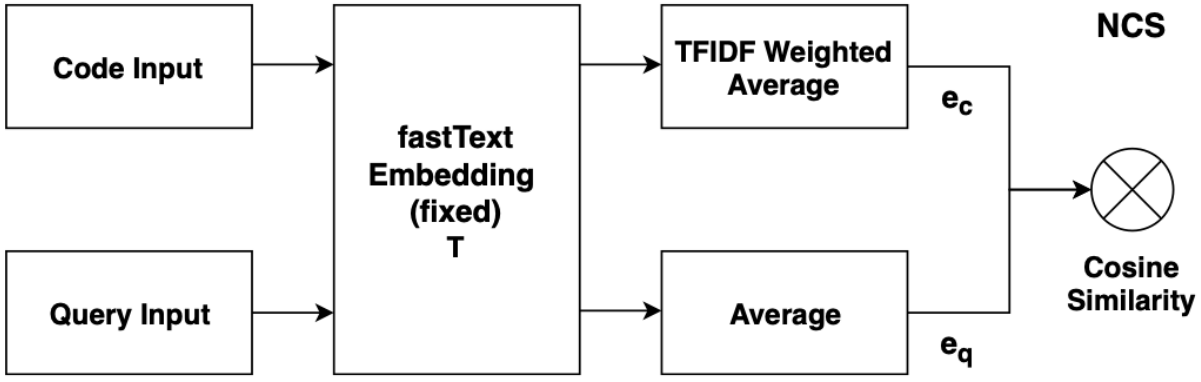

Рис. 2: Архитектура NCS [7]

Для оценки результата авторы собрали набор данных из вопросов со Stack Overflow [8] — портала для вопросов и ответов по тематике разработки программного обеспечения. Набор данных представляет собой заголовок вопроса, фрагмент кода из лучшего ответа и программный репозиторий, в котором можно найти этот фрагмент. Построенная модель смогла подобрать ответ на половину из вопросов. Некоторые запросы удалось переформулировать так, чтобы ответ был найден, но большая часть ошибок произошла по причине полного несовпадения токенов из запроса и из фрагментов кода. Таким образом, гипотеза авторов о достаточности информации в исходном коде не

подтвердилась. Тем не менее, использование обучения без учителя для построения векторных представлений показало лучшие результаты в сравнении с метрикой BM-25 [6], используемой в классическом информационном поиске, хоть и не значительно.

Таким образом, эта работа показала, что обучение без учителя для построения векторных представлений имеет смысл по причине своей простоты и лучшей точности, чем стандартные методы информационного поиска, но работает недостаточно точно, если исходный запрос содержит слова, которых нет в коде. Тем не менее, это не значит, что сложность модели CODEnn оправдана, что показано в следующей работе.

3.1.3 Embedding Unification (UNIF)

UNIF [7] является расширением NCS с использованием обучения с учителем для того, чтобы избежать ошибок при несовпадении слов в запросе и коде. Авторы ставили перед собой задачу сделать это расширение минимальным, чтобы новый подход не стал слишком сложным для применения, как в случае CODEnn.

С помощью метода обратного распространения ошибки [9] на основе некоторого корпуса, состоящего из фрагментов кода и их описаний, из исходных векторных представлений, вычисляемых с помощью FastText в модели NCS, получается два новых представления — для токенов естественного языка и для токенов, получаемых из кода. При взвешивании вместо TF-IDF используются коэффициенты механизма внимания [10]. Взвешенные векторные представления затем суммируются для получения вектора документа (метода). Такой подход широко распространён в машинном переводе, что по смыслу подходит к решаемой проблеме несовпадения токенов.

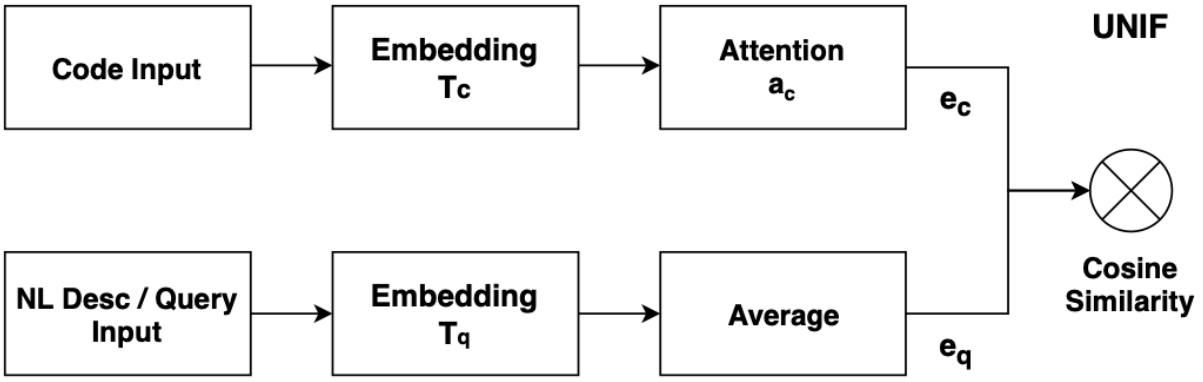

Рис. 3: Архитектура UNIF [7]

Авторы UNIF сравнили этот подход с NCS и CODEnn на одинаковых наборах данных. Сравнение с NCS показало, что в основном UNIF подбирает релевантные результаты для значительно большего количества вопросов, чем NCS, хотя на одном из наборов данных NCS показал немного лучший результат по одному из тестов. При сравнении с

CODEnn оказалось, что UNIF опережает эту архитектуру по всем тестам. Авторы не дают оценку тому, почему это происходит, но предполагают, что использование строгого порядка токенов в CODEnn может приводить к переобучению.

Что наиболее интересно в данной работе, авторы решили сравнить, насколько обучение на основе javadoc комментариев хуже или лучше обучения на основе вопросов со Stack Overflow в связке с лучшими ответами на них. Результат показал, что использование второго подхода улучшает результаты как UNIF, так и CODEnn, причём в случае UNIF количество релевантно подобранных ответов улучшается в 2-5 раз в зависимости от проведённого теста. Это говорит о том, что для задачи поиска в коде целесообразно обучать модель с использованием настоящих вопросов пользователей, а не описаний разработчиков, предоставляемых в самом коде. В то же время это ведёт к дополнительным усилиям по построению подобного набора данных.

3.1.4 Выводы

Из вышеописанных исследований можно сделать вывод о том, что обучение с учителем необходимо в задаче поиска в коде, так как остро стоит проблема несовпадения слов запроса и токенов кода. В то же время, авторы UNIF показали, что такая модель не обязательно должна быть сложной и ресурсоёмкой как CODEnn. В то же время обучение с учителем неизбежно ведёт к необходимости построения параллельного корпуса, что может быть сложной задачей, особенно если речь идёт о специфичной для какой-то области кодовой базе.

В отношении данного исследования стоит отметить, что ни одно из существующих решений не подразумевает объединение кода с артефактами программного обеспечения, например, с артефактами системы контроля версий. Это помогло бы расширить пространство возможных запросов к системе поиска информации, а в некоторых случаях сузить и уточнить поиск. В то же время, не очевидным является вопрос того, целесообразно ли напрямую добавлять данные из артефактов в исходный код для последующего обучения моделей. Например, в случае обучения с учителем ясно, что необходимо будет построить соответствующий большой параллельный корпус, в котором к каждому методу добавлялась бы некоторая дополнительная информация, что является нетривиальной задачей, так как потребует обработки сотен или тысяч репозиториев. К тому же, это может привести систему в нерабочее состояние, так как количество информации, содержащейся в артефактах, обычно намного больше, чем в теле самого метода, что скорее всего будет мешать обучению модели даже в случае обучения без учителя. Также стоит отметить, что NCS и UNIF разработаны в компании Facebook, а следовательно их исходный код не представлен в открытом доступе, что осложняет возможность их расширения.

3.2 Информационные системы для программных репозиториев

3.2.1 Вики-системы

Основным средством, которое потенциально могло бы решить задачу предоставления актуальной технической информации о проекте в понятной форме, являются вики-системы. Обычно это веб-сайты, позволяющие пользователям самостоятельно изменять его содержимое с помощью встроенных инструментов. Как правило, они имеют большинство возможностей, необходимых для комфортной работы: поисковый движок, комментарии, историю версий, средства для представления информации в удобочитаемом виде и другие. Но есть и коренные недостатки самой концепции таких систем.

Так как предполагается, что страницы создаются и заполняются вручную, это неизбежно приводит к проблеме поддержки такой документации. Информация об активно развивающемся программном проекте имеет свойство быстро терять свою актуальность, а время, расходуемое на поддержку внутренней документации, может стоить нескольких невыпущенных технических улучшений самого проекта.

Также, как уже было упомянуто выше, современный процесс разработки часто связан с использованием множества вспомогательных средств, каждое из которых создаёт свои артефакты. Интеграция таких побочных продуктов, например, данных Git-репозитория, в вики-системы обычно не представляется возможной и нецелесообразна с точки зрения потраченных усилий и получаемого результата. Это препятствует созданию единой базы знаний, в которой можно было бы найти актуальную информацию о разрабатываемом программном обеспечении.

3.2.2 Система Hipikat

Наиболее похожим на поставленную тему исследований по предназначению является проект Hipikat [11], разработанный в 2005 г. Его целью было создание «памяти проекта», как называли её авторы, заключающейся в объединении данных нескольких типов артефактов разработки: кода, документации, переписки, отчётов об ошибках и планах по тестированию. Далее Hipikat должен был рекомендовать разработчику релевантные его задаче артефакты.

Несомненно, создателями этой системы была проделана внушительная работа. Разработанный прототип позволял по существующей заявке в системе контроля ошибок или по файлу с кодом получить список релевантных артефактов. Авторы приводят показательный пример, в котором разработчик, решая задачу исправления дефекта в коде, запускает поиск по номеру решаемой заявки, а система рекомендует ему другие заявки об ошибках, наиболее релевантные решаемой. В одной из них похожая проблема затрагивает некоторый файл, и в результате дальнейшего исследования проблемы

выясняется, что ошибка снова находится в этом файле с кодом. Таким образом система Hipikat помогает сузить диапазон поиска ошибки. Хоть приведённый пример использования вряд ли бы действительно помог опытному разработчику, которому такие задачи обычно более целесообразно решать чтением кода, в остальном система действительно может быть полезной, рекомендуя к исследованию старые ошибки в программном коде и обсуждения. Всё это интегрируется в среду разработки и использует удалённый сервер из нескольких модулей для обработки запросов.

Тем не менее, предложенное решение не получилось бы применить к исследуемой задаче, даже если бы оно было в открытом доступе. Во-первых, концепция системы заключается в том, чтобы по одному артефакту найти связанные с ним другие, т.е. пользователю изначально нужно иметь на руках некоторую сущность для возможности поиска, а затем самостоятельно, на основе набора предоставленных источников, искать ответы на свои вопросы. Во-вторых, система Hipikat полностью ориентирована на новых в проекте разработчиков, подразумевая, что у пользователя немалый технический уровень — именно поэтому авторами не ставилась задача создания подобия вопросно-ответной системы. В-третьих, система не пытается осуществлять глубокий анализ имеющихся артефактов для самостоятельного извлечения из них различной информации. То есть для большей части проблем понадобится отдельный, углублённый анализ даже в рамках одного предоставленного в качестве ответа релевантного артефакта. И, наконец, в-четвёртых, эта система ориентирована на использование нескольких конкретных сервисов и программных средств, которые к тому же сильно устарели за прошедшие 16 лет, чего хотелось бы избежать для создания общедоступного решения.

3.2.3 Выводы

Таким образом, ни одно из используемых в жизни или предложенных в прошлом решений не позволяет и не ставит своей целью эффективно решить проблему поиска актуальной информации о проекте с помощью вопросов на естественном языке. Они либо сложны в поддержке, не гибкие и не подходят к задаче, либо требуют высокого технического уровня при использовании, что в рамках поставленной задачи нивелирует их остальные достоинства.

3.3 Получение ответа на вопрос на естественном языке

В результате анализа современных достижений в области вопросно-ответных систем было выявлено, что за последние годы со значительным отрывом от своих предшественников в задачах распознавания естественного языка побеждает нейронная сеть BERT [12] от компании Google. Именно она в данный момент применяется в соответ-

ствующей поисковой системе для улучшения понимания 10% запросов на английском языке. В конце 2018 г. компания Google открыла исходный код BERT и опубликовала несколько моделей [13], предобученных на стенфордском наборе данных для вопросов и ответов [14] (англ. The Stanford Question Answering Dataset, SQuAD), составленном на основе большого количества вопросов к статьям с ресурса Wikipedia. Преимуществом этой нейронной сети перед остальными является то, что она использует ненаправленное обучение для языковых моделей, т.е. позволяет понимать контекст слова на основе окружающих его слов, а не только на основе слов с одной стороны предложения.

Так как это решение является наиболее эффективным для получения ответа на вопросы на естественном языке, имеет смысл попробовать его применить и к решаемой задаче на одном из этапов. Это, в свою очередь, приводит к проблеме преобразования программного кода и артефактов процесса его разработки в некоторое подобие текста на естественном языке.

4 Построение решения задачи

Решаемую задачу можно разделить на несколько стадий, для каждой из которых необходимо построить решение. Следует также помнить, что артефакты разработки программного обеспечения могут поступать из множества источников в различных формах, поэтому решение должно быть легко расширяемым.

4.1 Разбор исходного кода программного репозитория

В качестве целевого языка для исследования поставленной задачи был выбран язык Java [15]. Это обосновано тем, что Java остаётся одним из основных языков для серверной разработки и мобильных приложений, в особенности в корпоративной среде [16][17][18]. Помимо этого язык Java является объектно-ориентированным языком программирования, т.е. в его основе лежит понятие объекта — сущности, объединяющей в себе данные и методы. Это ведёт к понятной и предопределённой структуре исходного кода, а следовательно, к большим возможностям для его анализа.

Исходный код на языке Java, несущий основной пласт информации, находится в классах — шаблонах для создания объектов. Классы включают поля, содержащие некоторые данные, и методы, оперирующие ими и входными параметрами. Также существуют абстрактные сущности, как, например, интерфейсы — шаблоны для создания классов, в основном содержащие в себе перечисление методов, которые должен переопределять любой расширяющий их класс. Тем не менее, интерфейсы почти не несут полезной информации, которую можно было бы использовать для поиска, т.к. не имеют бизнес-логики, поэтому для решаемой задачи они могут быть опущены. То же относится и к абстрактным методам абстрактных классов, т.е классов, которые могут иметь методы без реализации, и экземпляры которого не могут быть созданы.

Таким образом, весь исходный код можно представить как совокупность методов, имеющих реализацию, и уже на их основе строить поисковую систему. Для проверки концепции и простоты было принято решение не анализировать файлы конфигурации, так как они могут иметь самые разнообразные представления, а также вносить шум в исходные данные, хотя это не исключает возможность расширения источников артефактов итоговой поисковой системы конфигурационными файлами.

Для анализа исходного кода было решено использовать абстрактное синтаксическое дерево. Оно представляет собой конечное ориентированное дерево, в котором операнды являются листьями, а операторы — внутренними вершинами. Пример абстрактного синтаксического дерева для метода на рисунке 4 приведён на рисунке 5. Такое представление наиболее удобно для разбора исходного кода, т.к. задача сводится к рекурсивному обходу дерева.

```
        void read() {
            String input = reader.readLine();
            Command command = parser.parse(input);
            if (command != null) {
                commandSender.send(command);
            }
        }
```

Рис. 4: Пример метода для построения абстрактного синтаксического дерева

Для построения поисковой системы было решено извлекать из кода названия методов с названиями классов, в которых они содержатся, а также все вызовы методов, которые находятся внутри их тела. Локальные переменные игнорируются, т.к. их названия имеют свойство сильно варьироваться у разных разработчиков, вплоть до использования названий из одной буквы, таких как «a», «x» и так далее. Также часто бывает, что информация, содержащаяся в названиях локальных переменных уже покрыта названиями методов. Кроме того, из исходного кода дополнительно извлекаются данные, которые затем используются для некоторых алгоритмов на других этапах и для вывода результата, а именно позиция метода в файле, его параметры, их типы и др.

4.2 Извлечение данных из Git

В качестве одного из источников артефактов для построения решения задачи была выбрана система контроля версий, а именно самая распространённая — Git [19].

Система контроля версий является программным обеспечением для сбора и хранения информации об изменяющихся документах. Система Git была создана для контроля версий при разработке ядра операционной системы Linux [20]. Git представляет собой набор утилит, позволяющих:

- добавлять файлы в отслеживаемые или удалять из них;
- сохранять текущее состояние исходного кода как новую версию — так называемый «коммит» (англ. commit);
- создавать ветви (англ. branch) — именованные ссылки на некоторые коммиты, позволяющие впоследствии вести разработку независимо, не затрагивая другие ветви;
- манупулировать этими сущностями, например, объединять ветви (git merge) или отправлять коммиты в удалённый репозиторий (git push).

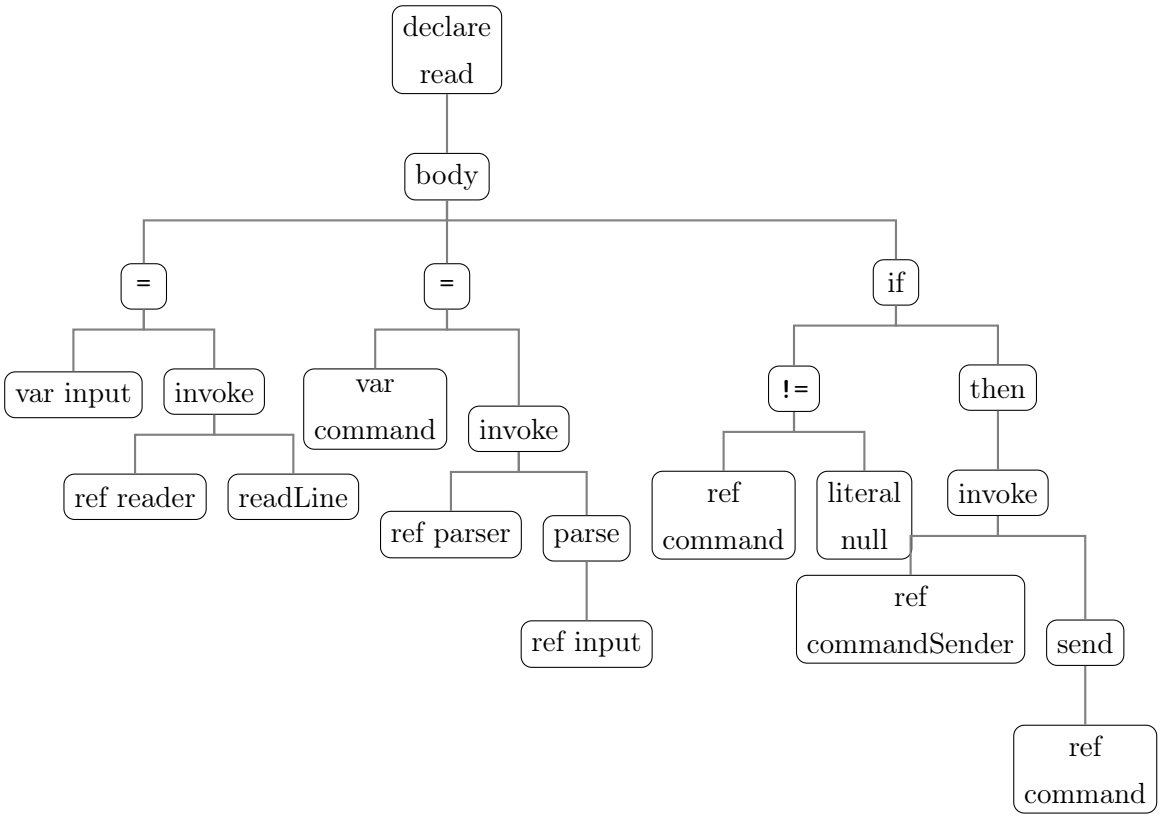

Рис. 5: Абстрактное синтаксическое дерево для метода на рисунке 4

Таким образом, информация из системы контроля версий может быть представлена в виде графа, в котором в узлах находятся коммиты, рёбра соединяют их с «родителями», т.е. с предшествующими коммитами, а ветвление происходит при создании коммитов в новых ветвях Git (рисунок 6). Такой граф принято называть деревом коммитов.

Коммиты несут в себе большое количество полезной информации, которая при определённом подходе может быть использована для построения поисковой системы. Она включает автора коммита; время, в которое он был зафиксирован; адрес электронной почты автора; сообщение, которое автор оставил при фиксации коммита и, что самое важное, список изменённых строк кода.

В качестве примера того, какую информацию можно использовать для поисковой системы, были выбраны атрибуты коммита, в котором метод появился впервые. Таким образом, если пользователь поисковой системы пожелает узнать, кто и когда разработал ту или иную функциональность, у системы будет вся информация для осуществления поиска.

Для того, чтобы извлечь первоначальный коммит для каждого метода, был реализован следующий алгоритм:

1. Выбрать текущую активную ветку репозитория;

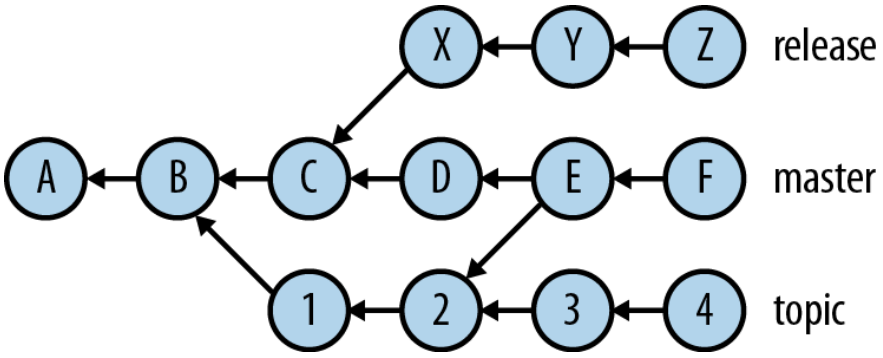

Рис. 6: Пример дерева коммитов, состоящего из трёх ветвей

2. Выбрать текущий коммит;
3. Для каждого изменённого файла извлечь все методы, которые он содержал после текущего коммита;
4. Для каждого изменённого файла извлечь все методы, которые он содержал до текущего коммита;
5. Вычесть из множества методов этапа 3 множество методов этапа 2 — это новые методы;
6. Сохранить информацию текущего коммита, сопоставив её методам, найденным на этапе 5;
7. Если у текущего коммита нет родителя, закончить алгоритм;
8. Иначе перейти к родительскому коммиту, пометить текущим;
9. Перейти к этапу 2.

Как видно, на этапе извлечения коммитов снова разбирается исходный код методом, аналогичным описанному в разделе 4.1. На выходе для каждого когда либо существовавшего метода, будем иметь коммит, в котором он появился впервые.

4.3 Извлечение ответа на заданный вопрос на естественном языке — BERT

Как уже было замечено в обзоре существующих решений, нейронная сеть BERT от компании Google является одним из лучших решений на данный момент в задачах распознавания естественного языка. Так как стоит задача извлечения краткого ответа при возможности, целесообразно попробовать применить BERT при решении данной проблемы.

```
class InputConsole {
    ...
    void readCommand() {
        try {
            String input = reader.readLine();
            Command command = parser.parse(input);
            if (command != null) {
                commandSender.send(command);
            }
        } catch (IOException e) {
            logger.log(Level.SEVERE, "Exception is thrown",
               e);
        }
    }
}
```

Рис. 7: Фрагмент класса `InputConsole` для показа производимых трансформаций

BERT основана на модели «трансформер» [10] и использует двунаправленное обучение (или, как замечают сами разработчики, ненаправленное обучение), позволяя понимать контекст слова на основе соседних слов с обоих сторон — существовавшие до этого модели анализировали последовательность лишь слева направо или справа налево. Достигается это за счёт того, что модель обучают на основе предложений с замаскированными словами. BERT также способна определять, взаимосвязаны ли два предложения между собой.

Единственным препятствием для применения BERT в решаемой задаче является то, что она рассчитана на работу со связным текстом на естественном языке. Из этого вытекает то, что на основе фрагмента кода необходимо сгенерировать некоторый текст.

Рассмотрим решение этой подзадачи на примере фрагмента класса, приведённого на рисунке 7. Как было описано выше, для построения поисковой системы из исходного кода извлекается названия методов и классов. Так как в языке Java принято использовать так называемый «верблюжий регистр» (от англ. CamelCase), все слова пишутся с прописной буквы и слитно, за исключением названий методов и переменных, в которых первое слово пишется со строчной буквы. Таким образом, для преобразования таких токенов в нечто похожее на текст на естественном языке необходимо их разделить, например, «readCommand» будет записано как «read command».

Трансформации всех типов извлекаемых сущностей представлены в таблице 1. Выбор именно таких трансформаций и сущностей обосновывается следующим образом:

Таблица 1: Трансформации исходного кода и коммитов для построения связного текста на естественном языке

Описание	Сущности	Текст
Название метода и его автор	Метод `readCommand()`, автор Yulia	Method "readCommand"was implemented by Yulia.
Цель метода	Метод `readCommand()`, класс InputConsole	Its purpose is read command for input console.
Цель метода, если его имя — main	Метод `main()`, класс InputConsole	Its purpose is running input console.
Сообщение коммита	Коммит	The method was created with message: "Create InputClass.".
Дата коммита	Коммит	The method was created on September 04, 2019.
Логика метода	Тело метода `readCommand()`	The method tokens are: read line parse send log.

- первое предложение необходимо для того, чтобы дать имя описываемому объекту (методу) и не исключить полное совпадение, если пользователь знает, какой именно метод он ищет;
- так называемая цель метода составляется из разделённых имён класса и метода, т.к. в этой форме получается текст, наиболее близкий к тому, как его бы записал человек;
- важно отметить, что название класса крайне необходимо для составления «цели метода», т.к. без него название метода может потерять контекст и смысл, например, в случае, если он называется `run()`;
- для метода `main()` цель описывается иначе, т.к его смысл обычно в запуске кода из класса, в котором он находится;
- токены, извлекаемые из методов, записываются подряд через пробел, имитируя связное предложение, частично описывающее бизнес-логику.

В заключение, следует отметить, что подобный подход легко расширяем и позволяет аналогичным образом добавлять информацию из других артефактов разработки ПО, например, из системы контроля ошибок.

4.4 Извлечение релевантного документа — FastText

Назовём описанные в предыдущем разделе представления методов исходного кода на естественном языке документами, как это принято в информационном поиске. Если применять модель BERT к каждому документу, возникает две существенные проблемы. Во-первых, BERT вернёт ответ для каждого с той или иной вероятностью, но т.к. документы строятся по одному шаблону, эта вероятность будет мало говорить о результате. Например, на вопрос категории «When...», BERT с высокой долей вероятности вернёт для каждого документа содержащуюся в нём дату, но это не будет значить, что документ соответствует запросу. Во-вторых, применение модели BERT к каждому документу, которых может быть значительное количество, было бы довольно ресурсозатратно и долго. Таким образом, перед извлечением ответа на поставленный вопрос необходимо сузить диапазон поиска, т.е. найти некоторое количество релевантных документов.

В обзоре существующих решений были описаны различные подходы к поиску в коде. Тем не менее, как было замечено, эти методы не могут быть напрямую применены к решаемой задаче, т.к. её нельзя свести только лишь к поиску фрагментов — информация из артефактов тоже должна участвовать в индексировании данных. Более того, артефакты разработки программного обеспечения могут иметь разную природу, а их число может расти и постоянно расширять возможности поисковой системы, поэтому обучать нейронную сеть на некотором фиксированном наборе извлекаемых данных было бы нецелесообразно. По этим причинам было принято решение осуществлять поиск по уже преобразованным документам с использованием соответствующих методов информационного поиска.

Существуют классические меры релевантности документа запросу, такие как TF-IDF [6] или BM25 [6], которые надёжно работают, но имеют один важный недостаток — нормализованные токены запроса и документа должны в точности совпадать, что приводит к проблемам, если в запросе используются синонимы слов из документов или присутствуют опечатки. Чтобы снизить влияние этих факторов, было решено использовать FastText [5] — библиотеку от компании Facebook, позволяющую с помощью обучения моделей нейронных сетей получать векторные представления слов таким образом, что близкие по смыслу слова находятся близко и в векторном пространстве. В общем случае такие алгоритмы называются Word2vec [21].

Одна из моделей, которые использует FastText является skip-gram. Она позволяет на основе одного слова предсказать окружающие его слова в некотором окне. Такие модели обычно долго обучаются, но показывают более точные результаты, чем их аналоги. FastText расширяет возможности модели skip-gram, представляя каждое слово как набор N-грамм его букв. Таким образом, сумма векторов N-грамм слова представляет его в векторном пространстве.

Из вышеописанного следует, что получив векторные представления запроса и каждого сгенерированного документа, можно оценить их близость в векторном пространстве с помощью косинусного расстояния:

$$\cos\theta = \frac{AB}{\|A\|_2\|B\|_2}, \tag{1}$$

где

$$\|A\|_2 = \sqrt{\sum_{i=1}^{N} A_i^2} \tag{2}$$

Для представления документа в виде вектора была использована нормализованная сумма векторов его слов:

$$V(doc) = \| \sum_{word \in doc} \|V(word)\|_2\|_2, \tag{3}$$

где V — функция векторного представления.

Перед применением векторизации из запросов и документов удаляются знаки препинания и стоп-слова, вносящие шум. Важно отметить, что FastText уже пытались применять для задачи поиска по фрагментам кода [7], но авторы решили удалять из запросов неизвестные слова. Для решения поставленной задачи было решено этого не делать, т.к. это в некоторых случаях может нивелировать возможности FastText по сопоставлению близких по значению слов.

4.5 Объединение этапов

Объединяя описанные выше этапы, итоговое решение можно высокоуровнево описать следующим алгоритмом:

1. Анализируется текущее состояние программного репозитория, извлекается информация о классах и методах;

2. Анализируется дерево коммитов, посредством разбора исходного кода «до» и «после» извлекается информация о коммитах, в которых методы появились впервые;

3. Результаты пунктов 1 и 2 объединяются посредством сопоставления каждому методу соответствующего коммита;

4. Для каждого метода строится документ, представляющий собой текст на естественном языке, объединяющий информацию из программного кода и артефактов его разработки;

5. Для каждого документа из этапа 4 удаляется пунктуация и стоп-слова, затем строится векторное представление с помощью некоторой модели FastText с использованием символьных N-грамм;

6. Для поступающих запросов строится векторное представление способом, аналогичным пункту 5;

7. Для векторных представлений запроса и документов рассчитывается косинусное сходство;

8. N лучших документов и исходный запрос передаются на вход некоторой модели BERT;

9. Результат работы BERT объединяется со всей дополнительной извлекаемой информацией о методах и возвращается в качестве результатов поиска.

5 Программная реализация

Предложенный подход к организации поиска информации был реализован с помощью сервера с веб-интерфейсом. Наличие сервера помимо прямого назначения позволяет не проходить для каждого запроса все этапы извлечения информации заново, сохраняя данные в памяти, а интерфейс предоставляет удобный способ быстро и в читаемой форме получить все релевантные ответы на заданный вопрос. Вид интерфейса представлен на рисунке 8.

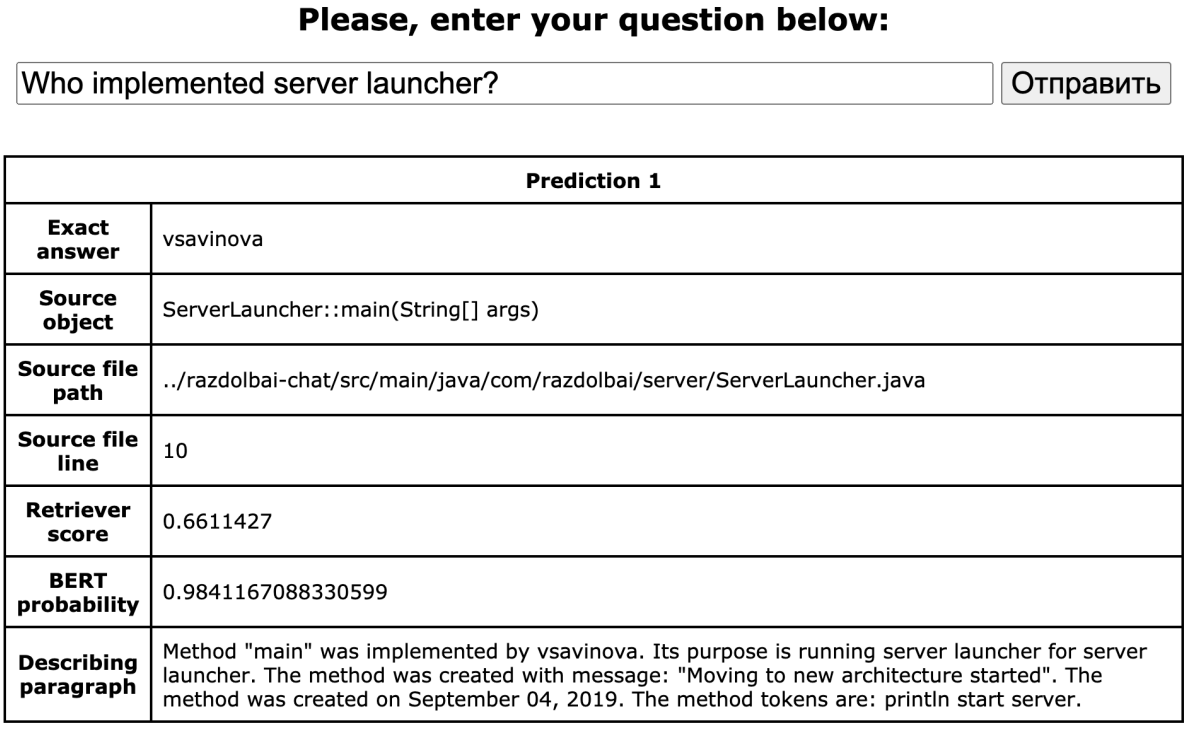

Рис. 8: Интерфейс реализованной системы поиска информации

Программный код состоит из 17 файлов на языке Python 3 [22], а также нескольких файлов для показа веб-интерфейса. Использование Python обусловлено работой с моделями нейронных сетей, которая наиболее распространена именно для этого языка. Опишем некоторые наиболее важные классы:

- `Server` — основная точка входа сервера, содержит всю конфигурацию и обработчики веб-запросов;
- `FasttextPipelineRunner, BM25PipelineRunner` — классы, инкапсулирующие в себе работу всех этапов. Название определяет, какой метод извлечения релевантных документов будет использован;
- `FasttextVectorizer` — класс, позволяющий получить векторное представление любого текста с помощью FastText. Включает в себя предобработку текста.

- `SourceFile, SourceMethod, MainMethod, SourceMethodParam` — классы, описывающие основные обрабатываемые сущности. Содержат в себе различные преобразования, необходимые для построения полного представления о программном репозитории;
- `Miner` — описывает алгоритмы извлечения информации из исходного кода и системы контроля версий;
- `SourceParagraphsTransformer` — описывает преобразования извлекаемой информации в документы, и наоборот;
- `model_util.py` — содержит методы для загрузки необходимых моделей FastText и BERT.

Для построения абстрактного синтаксического дерева был использован пакет javalang [23]. Извлечение данных Git осуществляется с помощью библиотеки pydriller [24].

6 Эксперименты

Эксперименты над реализацией предложенного подхода к построению системы поиска информации проводились с использованием предобученных моделей, предоставляемых авторами соответствующих нейронных сетей. Для извлечения релевантных документов была использована одна из описанных в статье [25] моделей FastText для английского языка, доступная для скачивания по ссылке [26]. Модель была обучена на основе всех статей с онлайн-энциклопедии Wikipedia на 2017 год, а также на основе набора данных, предоставляемых сервисом Common Crawl, который собирает данные веб-страниц и делает их доступными в удобном виде. Модель BERT была обучена на стенфордском наборе данных вопросов и ответов (от англ. The Stanford Question Answering Dataset, SQuAD) [14] версии 1.1. Этот набор данных состоит из более 100000 пар вопросов и ответов, основанных на более чем 500 статьях с Wikipedia. На этом наборе данных использованная модель достигает точности 81.3% по полному совпадению и 88.7% по мере F1, объединяющей в себе точность и полноту.

Для запуска сервера, описанного в предыдущей главе, требуется около 12 ГБ оперативной памяти и около 7 ГБ постоянной памяти вследствие применения нейронных сетей. Тестирование проводилось в облачном сервисе Google Colab [27].

Так как тестирование подобной системы подразумевает ручное написание осмысленных вопросов для каждого из документов, в качестве исследуемого в экспериментах репозитория был выбран небольшой обучающий проект [28]. Он состоит из 26 подходящих файлов на языке Java, содержащих 56 методов, разработанных 10 авторами в 237 коммитах.

В дальнейшем будем оценивать релевантность документов по их вхождению в N лидеров по результатам поиска и по среднеобратному рангу (от англ. mean reciprocal rank, MRR) — статистической мере эффективности информационного поиска при наличии результатов, упорядоченных по вероятности. Среднеобратный ранг высчитывается по формуле:

$$MRR = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{rank_i}, \tag{4}$$

где $rank_i$ — место правильного результата в упорядоченной выдаче, $|Q|$ — количество запросов.

Точность работы BERT будем оценивать бинарно для каждого релевантного запроса, т.е. если BERT правильно извлёк искомую сущность, присваиваем оценку 1, иначе — 0. Усреднение этих результатов позволит оценить среднюю точность.

6.1 Извлечение ответа для вопросов на естественном языке категории «Who»

6.1.1 Схожие токены запроса

Эксперимент заключается в том, чтобы для каждого документа вручную построить вопрос о том, кто создал описываемый в нём метод. Например, «Who implemented server launcher?». При этом токены запроса хоть и не полностью, но будут похожи на те, что присутствуют в документах, чтобы далее можно было заменить их на синонимы и сравнить полученные результаты. Например, для документа «Method "main"was implemented by Юлия Семченкова. Its purpose is running client for client...» запрос будет записан как «Who implemented client runner?».

Таблица 2: Категория вопросов «Who». Результаты поиска по схожим токенам

Топ 1	Топ 3	Топ 5	Топ 10	MRR	Точность BERT
24	20	9	2	0,63601	1

Результаты тестирования приведены в таблице 2. Как видно, большая часть методов вошла в первую тройку лидеров в выдаче. Оценка производилась по прямому совпадению вопроса и документа, на котором он основан, без учёта других релевантных документов. Особенно стоит отметить, что BERT во всех случаях вернул абсолютно правильный ответ с именем создателя метода.

Тем не менее, можно заметить, что существует один документ, который не попал даже в топ 10 — он находится на 12 месте в итоговой выдаче. При внимательном рассмотрении результатов было выявлено, что это произошло по причине перевеса количества схожих токенов в других документах по сравнению с искомым. А именно для метода с именем `send` из класса `CommandSender` был составлен вопрос «Who implemented command sender?», но его документ перевесили другие, в которых слово «command» повторялось несколько раз. Таким образом, можно сделать вывод о том, что, возможно, следует использовать некоторое взвешивание при построении векторных представлений предложений с помощью FastText.

6.1.2 Замена «implemented» на «developed»

Для того, чтобы оценить возможности FastText по построению векторных представлений, а также оценить влияние синонимов на работу BERT, было решено изменить запросы из раздела 6.1.1, начав с замены «implemented» на «developed». Тестирование проводилось аналогичным образом, его результаты приведены в таблице 3.

Таблица 3: Категория вопросов «Who». Результаты поиска по схожим токенам с заменой одного из них синонимом

Топ 1	Топ 3	Топ 5	Топ 10	MRR	Точность BERT
18	28	7	2	0,57236	1

Как видно, список лидеров несколько перераспределился вследствие изменения векторов FastText, и, таким образом, большая доля искомых документов консолидировалась на втором и третьем месте в выдаче, переместившись как сверху вниз, так и снизу вверх. Упомянутый ранее метод `send` переместился с 12 места на 15, что было ожидаемо. Особенно здесь стоит отметить тот факт, что точность ответов BERT осталась максимальной, что говорит о его устойчивости при изменениях контекста.

6.2 Извлечение ответа для вопросов на естественном языке категории «When»

Для тестирования возможности работы с различными сущностями, извлекаемыми из артефактов разработки ПО, было решено провести полностью аналогичные разделу 6.1 эксперименты, но для вопросов о дате создания метода. Таким образом, приведённый ранее в пример вопрос «Who implemented client runner?» трансформируется в «When was client runner implemented?». Результаты экспериментов представлены в таблицах 4 и 5.

Таблица 4: Категория вопросов «When». Результаты поиска по схожим токенам

Топ 1	Топ 3	Топ 5	Топ 10	MRR	Точность BERT
22	23	7	4	0,61209	1

Таблица 5: Категория вопросов «When». Результаты поиска по схожим токенам с заменой одного из них синонимом

Топ 1	Топ 3	Топ 5	Топ 10	MRR	Точность BERT
17	32	4	3	0,57102	1

Как и ожидалось, добавление в запросы двух новых токенов и удаление одного старого почти не изменило векторные представления, а следовательно результаты, хотя небольшие сдвиги в лидерах всё же имеются. Важность этого эксперимента заключается в том, что на нём снова была достигнута максимальная точность работы модели BERT, но на этот раз по извлечению даты из текста.

6.3 Поиск релевантных документов по односложному запросу

Для полного представления о разработанной системе имеет смысл оценить, как будет работать поиск релевантных документов, если запрос состоит из единственного слова, полностью совпадающего с одним из токенов в исходном документе. В данном случае было выбрано название метода. Хотя такие запросы и не являются целью построения данной системы поиска информации, точность поиска документов ограничивает количество запросов, передаваемых в BERT, поэтому важно проанализировать, как он будет работать и на простых примерах.

Так как установить какой-либо порог косинусного сходства можно только с помощью грубой оценки, а случай запроса, состоящего из одного токена вполне можно поддержать алгоритмически, возвращая все документы, в которых этот токен присутствует, для оценки точности было решено считать, что запрос возвращает столько результатов, сколько их есть в полной итоговой упорядоченной выдаче, считая от наиболее вероятных к наименее вероятным до последнего релевантного документа включительно. Полноту в данном предположении считать не имеет смысла, так как она будет равной 100%.

Таблица 6: Результаты поиска по одному токену — имени метода

Топ 1	Топ 3	Топ 5	Топ 10	MRR	$precision_{avg}$
37	12	6	1	0,79153	0,63867

Результаты тестирования приведены в таблице 6. При подсчёте точности под релевантными документами подразумевались те, в которых хоть раз встретилось название искомого метода. Анализируя результаты более досконально, можно заметить, что при поиске методов, состоящих из нескольких слов, точность близка к максимальной. Когда тестируются методы, состоящие из одного слова, общее количество релевантных документов резко увеличивается, так как начинают играть роль извлекаемые из тел методов токены. Методы, содержащие такие вызовы, представляются документами с большим количеством токенов, поэтому в итоговое векторное представление вносятся дополнительные шумы, что ухудшает качество поиска для односложных запросов.

Вхождение результатов запросов в топ лидеров оценивалось по полному совпадению искомого метода с описывающим его документом. Стоит отметить, что в выборке было 9 имён методов, для каждого из которых существовало несколько реализаций в одном или нескольких классах, что повлияло на их вхождение в топ лучших результатов, так как если на первое место попал один из них, то остальные там оказаться не могут. Это ещё раз показывает важность наличия контекста при работе с методам исходного кода. Учитывая этот факт и результаты эксперимента, можно с уверенностью сказать, что си-

стема при необходимости может быть использована и для поиска по ключевым словам, хотя её точность будет ниже, чем в специально разработанной для этого системе.

7 Заключение

В данной работе был впервые предложен подход к организации системы поиска информации на основе исходного кода и артефактов его разработки, позволяющей задавать вопросы и получать ответы на естественном языке со ссылкой на источник. Подход легко расширяем и использует современные и широко используемые модели машинного обучения, такие как BERT и FastText.

На базе этого подхода было реализовано программное средство с веб-интерфейсом, позволяющее организовать систему поиска информации для любого программного Git-репозитория на языке Java.

В ходе экспериментов была показана адекватность реализации предложенной концепции, её способность определять релевантные фрагменты кода и с высокой точностью извлекать краткие ответы на заданные вопросы на естественном языке, и, наконец, её устойчивость к работе с синонимичным контекстом.

Список литературы

[1] Deep Code Search / X. Gu, H. Zhang, S. Kim // 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE). — 2018. — Pp. 933–944.

[2] Recurrent neural network based language model / Tomas Mikolov, Martin Karafiát, Lukas Burget et al. — Vol. 2. — 2010. — 01. — Pp. 1045–1048.

[3] Retrieval on Source Code: A Neural Code Search / Saksham Sachdev, Hongyu Li, Sifei Luan et al. // Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages. — MAPL 2018. — New York, NY, USA: Association for Computing Machinery, 2018. — P. 31–41. https://doi.org/10.1145/3211346.3211353.

[4] Distributed Representations of Words and Phrases and their Compositionality / Tomas Mikolov, Ilya Sutskever, Kai Chen et al. // *CoRR*. — 2013. — Vol. abs/1310.4546. http://arxiv.org/abs/1310.4546.

[5] Enriching Word Vectors with Subword Information / Piotr Bojanowski, Edouard Grave, Armand Joulin, Tomas Mikolov // *CoRR*. — 2016. — Vol. abs/1607.04606. http://arxiv.org/abs/1607.04606.

[6] Neural Models for Information Retrieval / Bhaskar Mitra, Nick Craswell // *CoRR*. — 2017. — Vol. abs/1705.01509. http://arxiv.org/abs/1705.01509.

[7] When Deep Learning Met Code Search / José Cambronero, Hongyu Li, Seohyun Kim et al. // *CoRR*. — 2019. — Vol. abs/1905.03813. http://arxiv.org/abs/1905.03813.

[8] *Atwood, Jeff.* Stack Overflow — Where Developers Learn, Share, & Build Careers. — 2008. [HTML] (https://stackoverflow.com).

[9] *Goodfellow, Ian.* Deep Learning / Ian Goodfellow, Yoshua Bengio, Aaron Courville. — MIT Press, 2016.

[10] Attention Is All You Need / Ashish Vaswani, Noam Shazeer, Niki Parmar et al. // *CoRR*. — 2017. — Vol. abs/1706.03762. http://arxiv.org/abs/1706.03762.

[11] Hipikat: a project memory for software development / D. Cubranic, G. C. Murphy, J. Singer, K. S. Booth // *IEEE Transactions on Software Engineering*. — 2005. — Vol. 31, no. 6. — Pp. 446–465.

[12] BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding / Jacob Devlin, Ming-Wei Chang, Kenton Lee, Kristina Toutanova // *CoRR*. — 2018. — Vol. abs/1810.04805. http://arxiv.org/abs/1810.04805.

[13] *Devlin, Jacob*. Open Sourcing BERT: State-of-the-Art Pre-training for Natural Language Processing. — 2018. [HTML] (https://ai.googleblog.com/2018/11/open-sourcing-bert-state-of-art-pre.html).

[14] SQuAD: 100, 000+ Questions for Machine Comprehension of Text / Pranav Rajpurkar, Jian Zhang, Konstantin Lopyrev, Percy Liang // *CoRR*. — 2016. — Vol. abs/1606.05250. http://arxiv.org/abs/1606.05250.

[15] *Gosling, James*. The Java Language Specification. — 2015. [HTML] (https://docs.oracle.com/javase/specs/jls/se8/html/index.html).

[16] *s.r.o., JetBrains*. The State of Developer Ecosystem 2020. — 2020. [HTML] (https://www.jetbrains.com/lp/devecosystem-2020/).

[17] *Inc, Stack Exchange*. Stack Overflow Developer Survey 2019. — 2019. [HTML] (https://insights.stackoverflow.com/survey/2019).

[18] *Inc, Stack Exchange*. Stack Overflow Developer Survey 2020. — 2020. [HTML] (https://insights.stackoverflow.com/survey/2020).

[19] *Torvalds, Linus*. git –fast-version-control. — 2005. [HTML] (https://git-scm.com).

[20] *Torvalds, Linus*. The Linux Foundation. — 1991. [HTML] (https://www.linuxfoundation.org).

[21] Efficient Estimation of Word Representations in Vector Space / Tomás Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean // 1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings / Ed. by Yoshua Bengio, Yann LeCun. — 2013. http://arxiv.org/abs/1301.3781.

[22] *Rossum, Gvido van*. Welcome to Python.org. — 2008. [HTML] (https://www.python.org).

[23] *Thunes, Chris*. Pure Python Java parser and tools. — 2008. [HTML] (https://github.com/c2nes/javalang).

[24] *Spadini, Davide*. Python Framework to analyse Git repositories. — 2018. [HTML] (https://pydriller.readthedocs.io/en/latest/tutorial.html).

[25] Learning Word Vectors for 157 Languages / Edouard Grave, Piotr Bojanowski, Prakhar Gupta et al. // *CoRR*. — 2018. — Vol. abs/1802.06893. http://arxiv.org/abs/1802.06893.

[26] *Inc., Facebook*. Word vectors for 157 languages. — 2018. [HTML] (https://fasttext.cc/docs/en/crawl-vectors.html).

[27] *Google*. What is Colaboratory? — 2018. [HTML] (https://colab.research.google.com).

[28] *Zhukov, Pavel*. Educational Java project. — 2019. [HTML] (https://github.com/blazzen/razdolbai-chat).