Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Кафедра системного программирования

Тучина Анастасия Игоревна

Поиск ошибок в коде при помощи глубокого обучения в IntelliJ IDEA

Курсовая работа

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

Санкт-Петербург
2019

Оглавление

Введение

Автоматическое обнаружение ошибок в программном коде – актуальная проблема для разработчиков и активно набирающая популярность задача для исследователей. Большинство современных решений, в основном, концентрируются на статическом анализе кода и использовании существующих паттернов ошибок или наборов правил. Обычно детекторы ошибок пишутся вручную, а затем расширяются с помощью всевозможных эвристик и паттернов [1, 4].

Несмотря на достаточную эффективность, распространенные существующие подходы имеют ряд проблем. Большое количество классов ошибок делает реализацию алгоритма, распознающего большую их часть, нетривиальной задачей, поэтому на практике детекторы, использующие статический анализ, покрывают лишь малую часть всех возможных ошибок. Кроме того, инструменты, реализованные с использованием подобных подходов, достаточно трудно расширять на новые классы ошибок. Также важно, что многие анализаторы кода не используют необходимую для понимания текста программы информацию, содержащуюся в нем (например, названия функций и переменных), вследствие чего упускают часть ошибок. Таким образом, многие ошибки, особенно в динамически типизированных языках, остаются незамеченными.

В последние годы появляются детекторы, использующие элементы естественного языка в программном коде и полагающиеся на лексические особенности и лексическую схожесть идентификаторов при статическом анализе [2, 7].

Существуют и другие подходы, основывающиеся на алгоритмах машинного обучения и предлагающие рассматривать задачу обнаружения ошибок в коде как задачу бинарной классификации (например, DeepBugs [8]). DeepBugs предоставляет набор инструментов для анализа кода на JavaScript, использующих семантические особенности элементов естественного языка в тексте программы и позволяющих обучать детекторы для различных классов ошибок автоматически. Для обучения детекторов ошибок требуется большой датасет, из которого

извлекаются фрагменты кода, считающиеся корректными. В дальнейшем они преобразуются с помощью специальных генераторов в примеры некорректных фрагментов кода. Оба полученных набора данных составляют набор обучающих данных и используются для обучения моделей-детекторов.

Различные инструменты для поиска ошибок в коде наиболее актуальны для использования с динамически типизированными языками, где некоторые ошибки могут быть обнаружены уже на стадии выполнения программы ввиду отсутствия статической проверки типов. Поэтому инструмент, подобный DeepBugs, может быть полезен при использовании внутри IDE в качестве плагина.

Постановка задачи

Целью данной работы является интеграция детекторов ошибок DeepBugs с IntelliJ IDEA/WebStorm, а также их адаптация и реализация для Python с последующей интеграцией с IntelliJ IDEA/PyCharm. Для достижения поставленной цели требуется:

- разработать архитектуру плагинов;
- реализовать плагин для IntelliJ IDEA/WebStorm, задействующий предложенные модели;
- адаптировать предложенный подход для Python с использованием IntelliJ Platform SDK [9];
- выбрать или собрать датасет для обучения детекторов ошибок, содержащий файлы на Python;
- реализовать набор инструментов для предобработки данных из датасета на Python, то есть:
  - реализовать возможность представления фрагментов кода в виде кортежей различных признаков и их векторизации;
  - составить словарь из наиболее часто встречающихся элементов в коде, содержащий их векторные представления;
- обучить модель-детектор для каждого из выбранных классов ошибок для Python;
- реализовать плагин для IntelliJ IDEA/PyCharm, поддерживающий инспекции кода, которые задействуют полученные модели.

1. Обзор существующих решений

1.1. Популярные инструменты

Набор существующих и активно используемых инструментов для поиска ошибок в программном коде достаточно разнообразен, но большинство из них предполагает применение различных методов статического анализа кода, а также широкое использование эвристик.

Примеры известных инструментов/подходов.

- Статический анализатор с возможностью расширения с помощью эвристик, использующий для каждого из паттернов ошибок отдельный детектор, работающий в соответствии с определенным набором правил (lgtm[1], Google Error Prone [1], FindBugs [4]).

- Статический анализатор, использующий лексические особенности элементов естественного языка в коде [2].

- Детектор, использующий алгоритмы машинного обучения [6], обучающийся только на положительных примерах кода и поэтому требующий добавления определенных эвристик, так как иначе может распознавать любые необычные фрагменты кода как ошибки.

1.2. DeepBugs

DeepBugs – инструментарий для анализа кода на JavaScript, в основе которого лежит анализ семантических особенностей элементов естественного языка в программном коде, позволяющий обучать детекторы вместо написания их вручную и легко расширяющийся на новые классы ошибок. Подход, используемый DeepBugs, предлагает расценивать задачу обнаружения ошибок в коде как задачу бинарной классификации. Главное отличие DeepBugs от остальных подходов, предполагающих использование машинного обучения, заключается в том, что детекторы обучаются не только на примерах корректных фрагментов кода, но и

[1] `https://lgtm.com`

на сравнимом количестве примеров некорректных фрагментов, что повышает их точность.

DeepBugs предоставляет набор инструментов для извлечения данных из датасета, а также для генерации из извлеченных фрагментов кода данных для обучения модели и их векторизации, из чего можно заключить, что создание детекторов ошибок почти полностью автоматизировано, как показано на рис. 1.

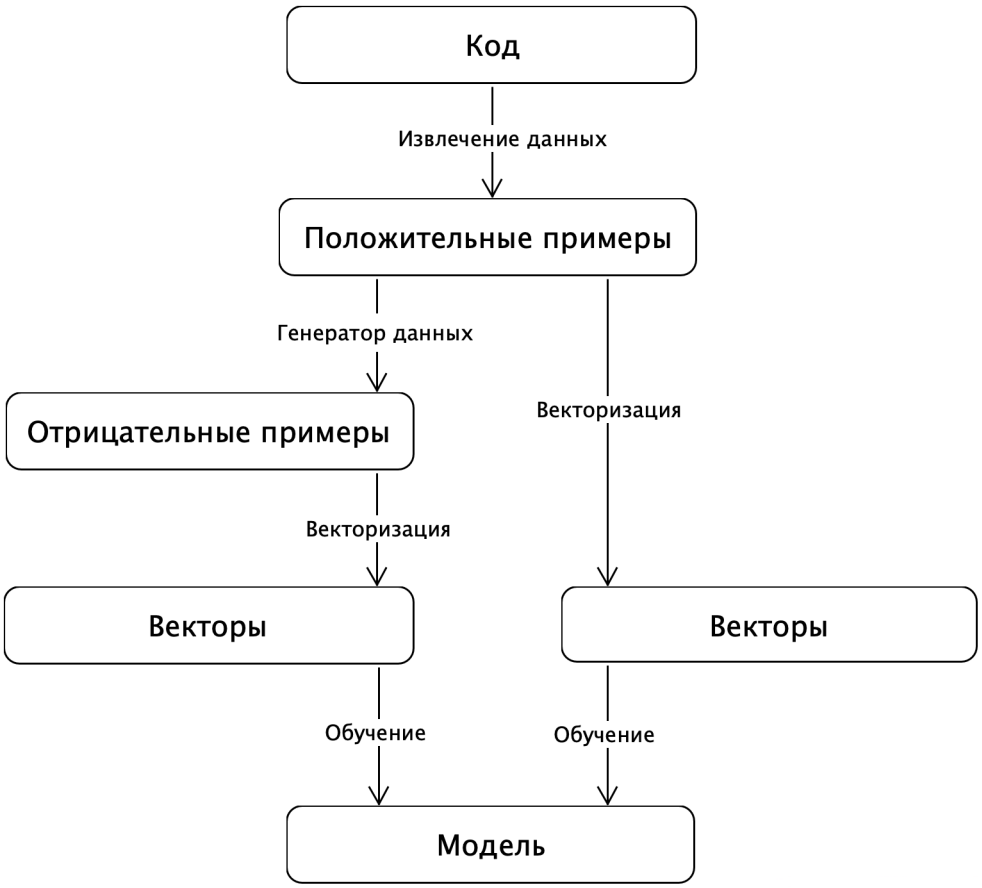

Рис. 1: Создание детектора ошибок с помощью DeepBugs

Для обучения используется объемный корпус кода (около 150000 файлов, содержащих код на JavaScript), из которого с помощью набора специальных реализованных инструментов извлекаются подходящие фрагменты кода, считающиеся корректными и называемые положительными примерами. Полагая, что большинство кода в открытом доступе корректно, DeepBugs использует генераторы обучающих данных для получения некорректных фрагментов кода, называемых отрицательными примерами, из уже существующих положительных, при-

меняя соответствующие конкретному классу ошибок преобразования. Примеры из обоих наборов представляются в виде кортежей своих признаков и векторизуются, составляя набор обучающих данных для модели.

В настоящее время DeepBugs поддерживает детекторы для трех классов ошибок, имеющих достаточно высокую точность (68% истинно-положительных результатов) даже в сравнении с популярными статическими анализаторами.

- Детектор неправильных операторов в бинарных выражениях.

  Пример фрагмента корректного кода:
  `i < length`

  Пример фрагмента некорректного кода:
  `i <= length`

- Детектор неправильных операндов в бинарных выражениях.

  Пример фрагмента корректного кода:
  `width - x`

  Пример фрагмента некорректного кода:
  `width - y`

- Детектор неправильного порядка аргументов в вызовах функций (для функций с двумя параметрами).

  Пример фрагмента корректного кода:
  `setSize(width, height)`

  Пример фрагмента некорректного кода:
  `setSize(height, width)`

1.2.1. Извлечение данных

Для обучения детекторов ошибок авторами статьи был выбран датасет 150k JavaScript Dataset[2], при извлечении данных из файлов кото-

[2] `https://eth-sri.github.io/js150`

рого строится AST с помощью acorn для каждого из них. Затем выбираются нужные узлы, соответствующие требуемым фрагментам кода. Часть их признаков извлекается с помощью следующих функций:

- Извлечение имен узлов AST.

  Пусть name(n) – функция, извлекающая имя узла AST.

  Если выражение, содержащееся в узле n:

  - идентификатор – извлекается его имя;
  - литерал – извлекается его строковое представление;
  - this – извлекается строка ”this”;
  - выражение вида base.prop – извлекается name(prop);
  - выражение вида base[k] – извлекается name(base);
  - вызов функции вида base.callee(..) – извлекается name(callee);
  - выражение вида `x++`/`x--` – извлекается name(x);
  - если тип выражения, представляемого n, не совпадает ни с одним из типов выше, n не рассматривается.

- Извлечение типов узлов AST.

  Если выражение, содержащееся в узле n:

  - число – извлекается ”number”;
  - строка – извлекается ”string” ;
  - логическое выражение – извлекается ”boolean”;
  - регулярное выражение – извлекается ”regex”.
  - null – извлекается ”null” ;
  - this – извлекается ”object” ;
  - тип не определен – извлекается ”undefined”;
  - для остальных выражений тип узла не извлекается и обозначается как ”unknown”.

1.2.2. Представление фрагментов кода в виде кортежей

Фрагменты кода представляются в виде кортежей извлеченных признаков, которые затем векторизуются.

- Бинарная операция:
  $(n_{left}, n_{right}, op, t_{left}, t_{right}, k_{parent}, k_{grandP})$
  - n_{left} – имя левого операнда.
  - n_{right} – имя правого операнда.
  - op – оператор.
  - t_{left} – тип левого операнда.
  - t_{right} – тип правого операнда.
  - k_{parent} – тип родительского узла.
  - k_{grandP} – тип родительского узла для родительского узла.
- Вызов функции (принимающей два аргумента):
  $(n_{base}, n_{callee}, n_{arg_1}, n_{arg_2}, t_{arg_1}, t_{arg_2}, n_{param_1}, n_{param_2})$
  - n_{base} – имя базового класса.
  - n_{callee} – имя вызываемой функции.
  - n_{arg_1} – имя первого аргумента функции.
  - n_{arg_2} – имя второго аргумента функции.
  - t_{arg_1} – тип первого аргумента функции.
  - t_{arg_2} – тип второго аргумента функции.
  - n_{param_1} – имя первого параметра функции.
  - n_{param_2} – имя второго параметра функции.

1.2.3. Векторизация отдельных токенов

Для дальнейшей векторизации наборов признаков фрагментов кода необходимо получить векторные представления имен некоторых токенов, встречающихся в коде. Чтобы повысить эффективность детекторов ошибок, при векторизации токенов требуется получить похожие

векторные представления для семантически схожих имен идентификаторов и литералов. Для этого воспользуемся моделью CBOW для Word2Vec [3], переопределив понятие контекста для применимости подхода к программному коду. Из наиболее часто встречающихся токенов, представляющих имена идентификаторов или литералы, составляется словарь (размер словаря $|V| = 10000$).

1.2.4. Модификация CBOW Word2Vec

Word2Vec – программный инструмент для анализа семантики естественных языков на основе контекста. Основная концепция Word2Vec заключается в том, что значение слова связано с контекстом, в котором оно используется. Модель CBOW позволяет предсказывать слово по его контексту. Контекстом слова называется последовательность слов, стоящих слева и справа от текущего. Количество учитываемых слов называется размером окна и регулируется в зависимости от задачи. Рассмотрим код как последовательность токенов. Тогда контекстом каждого токена будут токены, находящиеся слева и справа от выбранного (установленный размер окна $w = 20$). Необходимые данные извлекаются из кода для каждого токена, затем они используются для обучения CBOW модели. После обучения модель имеет внутренние представления для каждого из токенов, учитывающие их семантику, которые можно использовать как векторные представления этих токенов.

1.2.5. Генерация данных для обучения

Каждому классу ошибок соответствует отдельный генератор, применяющий соответствующие преобразования к извлеченным ранее данным.

- Детектор неправильных операторов в бинарных выражениях.

  Для создания отрицательного примера в том же файле, в котором расположена текущая бинарная операция, находится другой бинарный оператор и подставляется вместо существующего.

  - Положительный пример:
    $(n_{left}, n_{right}, op, t_{left}, t_{right}, k_{parent}, k_{grandP})$
  - Отрицательный пример:
    $(n_{left}, n_{right}, op', t_{left}, t_{right}, k_{parent}, k_{grandP})$
    op' – оператор, выбранный из того же файла, т.ч. $op' \neq op$.

- Детектор неправильных операндов в бинарных выражениях.

  Для создания отрицательного примера в том же файле, в котором расположена текущая бинарная операция, находится другой бинарный операнд и подставляется вместо правого или левого операнда в данном выражении (выбор случаен).

  - Положительный пример:
    $(n_{left}, n_{right}, op, t_{left}, t_{right}, k_{parent}, k_{grandP})$
  - Отрицательный пример:
    $(n'_{left}, n_{right}, op, t'_{left}, t_{right}, k_{parent}, k_{grandP})$
    или
    $(n_{left}, n'_{right}, op, t_{left}, t'_{right}, k_{parent}, k_{grandP})$
    n'_{left} и t'_{left} (n'_{right} и t'_{right}) – имя и тип операнда, выбранного из того же файла, т.ч. $n'_{left} \neq n_{left}$ ($n'_{right} \neq n_{right}$).

- Детектор неправильного порядка аргументов в вызове функции.

  Для создания отрицательного примера в каждом фрагменте данных, соответствующем вызову функции, меняются местами аргументы и их типы.

  - Положительный пример:
    $(n_{base},\ n_{callee}, n_{arg_1}, n_{arg_2}, t_{arg_1}, t_{arg_2}, n_{param_1}, n_{param_2})$
  - Отрицательный пример:
    $(n_{base},\ n_{callee}, n_{arg_2}, n_{arg_1}, t_{arg_2}, t_{arg_1}, n_{param_1}, n_{param_2})$

1.2.6. Векторизация полученных данных

Для генерации данных для обучения детекторов ошибок нужно векторизовать извлеченные из узлов AST кортежи их признаков.

Векторизация для каждого из признаков представляется соответствующим ему образом.

- Каждый из операторов закодирован с помощью one-hot encoding.
- Каждый тип данных закодирован с помощью уникального случайного бинарного вектора длины 5.
- Каждый тип узла AST закодирован с помощью уникального случайного бинарного вектора длины 8.
- Для имен идентификаторов получены векторные представления, содержащиеся в словаре. Пример не рассматривается, если имени одного из идентификаторов, содержащихся в кортеже, нет в словаре. Размер векторного представления – 200.

2. Архитектура плагина для IntelliJ IDEA

В рамках данной работы было принято решение интегрировать модели-детекторы DeepBugs с IntelliJ Platform. Детекторы ошибок реализуются в виде инспекций кода, каждая из которых является частью соответствующего плагина [3].

2.1. Основные компоненты архитектуры

На рис. 2 представлена архитектура инструмента для обнаружения ошибок в коде (голубым цветом выделены классы SDK).

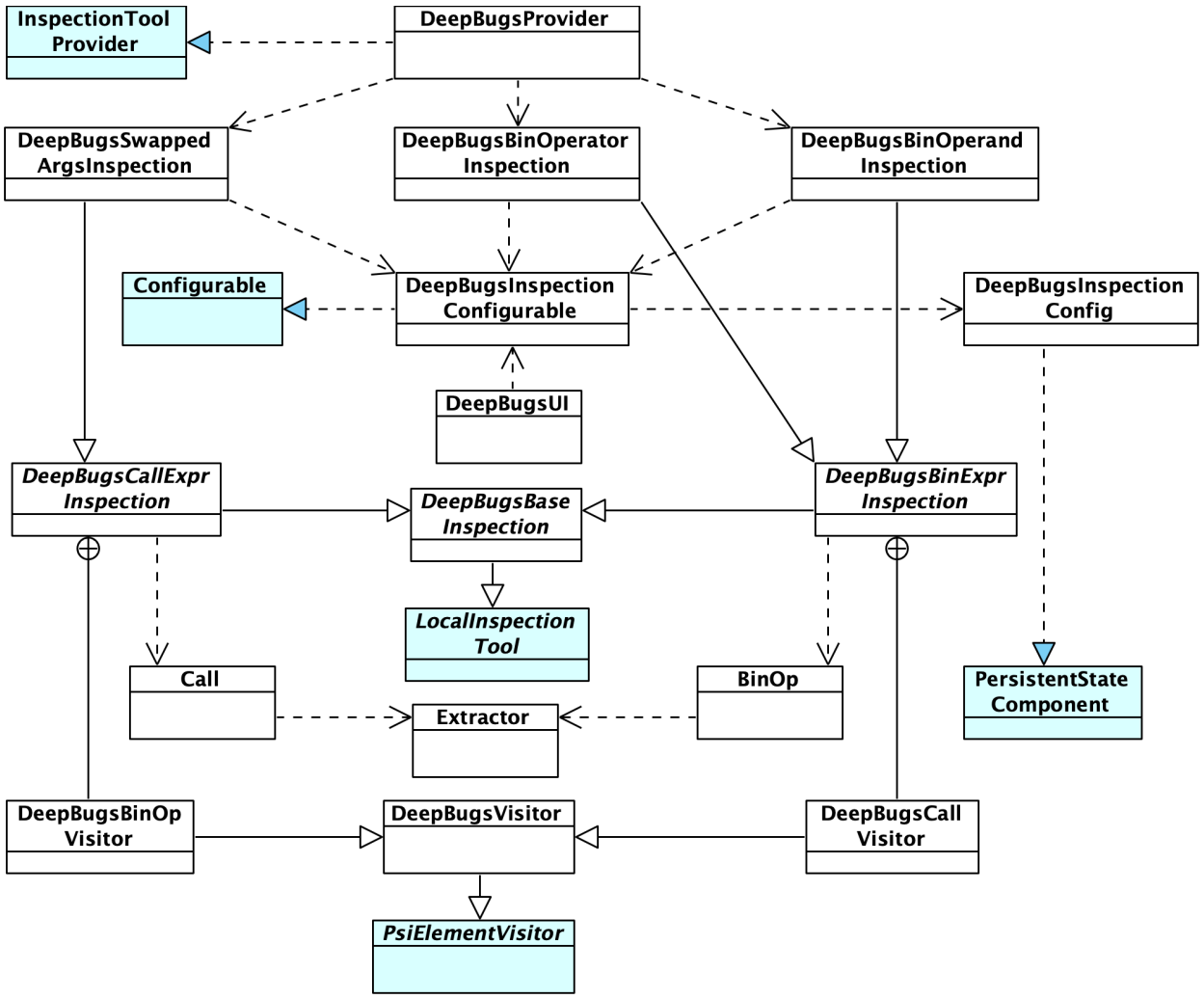

Рис. 2: Архитектура плагина

Класс DeepBugsProvider реализует интерфейс InspectionToolProvider,

[3] https://github.com/ml-in-programming/DeepBugsPlugin

поэтому он будет использован для получения специфицированных в методе getInspectionClasses() инспекций кода (DeepBugsBinOperatorInspection, DeepBugsBinOperandInspection, DeepBugsSwappedArgsInspection). Инспекции преобразуют фрагменты кода (вызовы функций – PsiCallExpression и бинарные выражения – PsiBinaryExpression) с помощью методов класса Extractor в экземпляры data классов Call и BinOp, затем полученные данные векторизуются. Векторизованные данные используются моделями для предсказания вероятности того, что фрагмент кода некорректен. Если вычисленная вероятность больше определенного порогового значения, фрагмент расценивается как возможно некорректный и подсвечивается. Значение порога можно настроить, для пользователя эту возможность предоставляет класс DeepBugsInspectionConfigurable, реализующий интерфейс Configurable. Он использует класс DeepBugsUI, который обеспечивает пользовательский интерфейс для настройки. Хранение установленных пользователем настроек представляется с помощью класса DeepBugsInspectionConfig, который реализует интерфейс PersistentStateComponent, позволяющий сохранять данные в неизменном виде между запусками IDE. Также можно вернуть настройки по умолчанию.

2.2. Другие детали архитектуры

Помимо классов, работающих непосредственно с SDK, предусмотрены еще несколько возможностей.

Детектору ошибок для работы требуются файлы, содержащие векторные представления фрагментов кода и файлы моделей. Распространять их вместе с плагином нежелательно, так как они достаточно объемны (размеры файлов, содержащих векторные представления идентификаторов, могут превышать 100Мб). Следовательно, потребовалось использовать сетевой репозиторий для хранения файлов. В данный репозиторий выложены необходимые файлы. С плагином распространяется его дескриптор, содержащий имена нужных файлов, пути к ним и URL для скачивания. На стороне пользователя создается локальный

репозиторий – директория с файлами и дескриптором локального репозитория. При запуске IDE с подключенным плагином проверяется, присутствует ли каждый из необходимых файлов на стороне пользователя. В случае, если какие-то файлы отсутствуют, они скачиваются с сетевого репозитория. Ввиду большого объема некоторых файлов, их скачивание может занять длительное время, поэтому была добавлена возможность отслеживать текущий прогресс с помощью ProgressIndicator.

Еще одна возможность позволяет при возникновении исключений при работе плагина автоматически создать Issue в GitHub репозитории проекта. При возникновении исключения автоматически формируется сообщение об ошибке, содержащее информацию о месте ее возникновения, ее stacktrace, версии плагина, IDE и т.д., затем пользователю предлагается отправить Issue в репозиторий разработчика, если сообщение сформировано успешно, иначе – перейти в репозиторий и создать Issue самостоятельно.

3. Выбор датасета

Детекторы ошибок в DeepBugs могут быть полезными на практике только в случае наличия достаточного количества данных для обучения используемых ими моделей. Поэтому необходимо подобрать датасет, который предоставляет большое количество данных для обучения детекторов ошибок и при этом:

- не содержит дубликатов файлов;
- не содержит обфусцированный код (детекторы ошибок во многом полагаются на семантику и контекст идентификаторов и литералов, поэтому, например, обфускация имен может значительно повлиять на точность модели на практике в худшую сторону);
- для кода из каждого файл из датасета возможно построить AST с помощью одного из существующих инструментов для Python.

Датасет можно как составить самостоятельно, собрав определенное количество проектов с GitHub, так и использовать уже готовый.

Многие проекты на GitHub могут содержать дубликаты (являться ответвлениями других репозиториев, уже имеющихся в выборке). Также не исключено наличие обфусцированного кода и невозможность правильно обрабатывать содержащиеся файлы из-за частичной несовместимости синтаксических конструкций Python2 и Python3, что не всегда возможно отследить с помощью программных инструментов. Поэтому можно подобрать уже готовый датасет, соответствующий указанным требованиям.

В качестве датасета для данной работы был выбран 150k Python Dataset[4].

3.1. 150k Python Dataset

150k Python Dataset – датасет, собранный специально для создания программных инструментов, использующих машинное обучение и

[4] `https://eth-sri.github.io/py150`

требующих значительного количества обучающих данных для используемых моделей. В частности, данный датасет был использован для обучения моделей, являющихся частью инструмента для автодополнения кода DeepSyn [5]. Данный датасет предоставляет более 8000 тысяч проектов с GitHub (около 320000 файлов). Каждый файл в 150k Python Dataset уникален, а AST, построенное для кода в каждом файле (авторы рекомендуют использовать asttokens для обработки каждого из файлов), содержит не более 30000 узлов, что влияет на скорость обработки данных. Более того, авторы датасета при его составлении использовали только репозитории проектов, имеющих лицензии MIT, BSD или Apache. Большая часть файлов (более 95%) также совместима и с Python2, и с Python3, что обеспечивает универсальность.

4. Предобработка данных

При адаптации подхода DeepBugs к созданию детекторов ошибок для Python достаточно реализовать собственный подход к извлечению признаков узлов AST и составить словарь подходящего размера на основе выбранного датасета, в остальном используя те же способы представления фрагментов кода в виде кортежей, получения векторных представлений для каждого из признаков и фрагментов кода в целом, а также генерации обучающих данных из извлеченных фрагментов. Цель предобработки данных заключается в извлечении подходящих фрагментов кода из датасета и представлении этих фрагментов в виде кортежей, состоящих из набора их признаков (имен использованных идентификаторов, типов данных и т.д.). Предобработка данных состоит из следующих этапов.

1. Извлечение требующихся для обучения данных в нужной форме с использованием AST.

2. Векторизация отдельных токенов, получение словаря.

4.1. Извлечение данных

Для извлечения данных из файлов сначала строится AST с помощью asttokens для каждого из них. Затем выбираются нужные фрагменты кода, представленные узлами этого дерева, и извлекаются требующиеся признаки.

Функции извлечения признаков не могут быть использованы для Python в таком виде, в каком они реализованы в DeepBugs для JavaScript, из-за особенностей синтаксиса языка. Поэтому потребовалось реализовать набор инструментов для извлечения данных из кода на Python самостоятельно.

4.1.1. Извлечение имен узлов AST

Пусть name(n) – функция, извлекающая имя узла AST.

Если выражение, содержащееся в узле n:

- идентификатор – извлекается его имя.
- литерал – извлекается его строковое представление.
- self – извлекается строка ”self”.
- выражение вида base.prop – извлекается name(prop).
- выражение вида base[k] – извлекается name(base).
- вызов функции вида base.callee(..) – извлекается name(callee).
- если тип выражения, представляемого n, не совпадает ни с одним из типов выше, n не рассматривается.

4.1.2. Извлечение типов узлов AST

Если выражение, содержащееся в узле n:

- число – извлекается ”number”.
- строка – извлекается ”string”.
- логическое выражение – извлекается ”boolean”.
- лямбда-выражение – извлекается ”lambda”.
- None – извлекается ”none”.
- self – извлекается ”object”.
- для остальных выражений тип узла не извлекается и обозначается как ”unknown”.

4.2. Составление словаря

Словарь составляется из наиболее часто встречающихся в коде токенов, извлеченных из обучающих данных и покрывающих большую часть датасета. В данном случае около 50000 наиболее распространенных токенов могут покрыть приблизительно 90% набора всех идентификаторов, содержащихся в датасете, как показано на рис. 3.

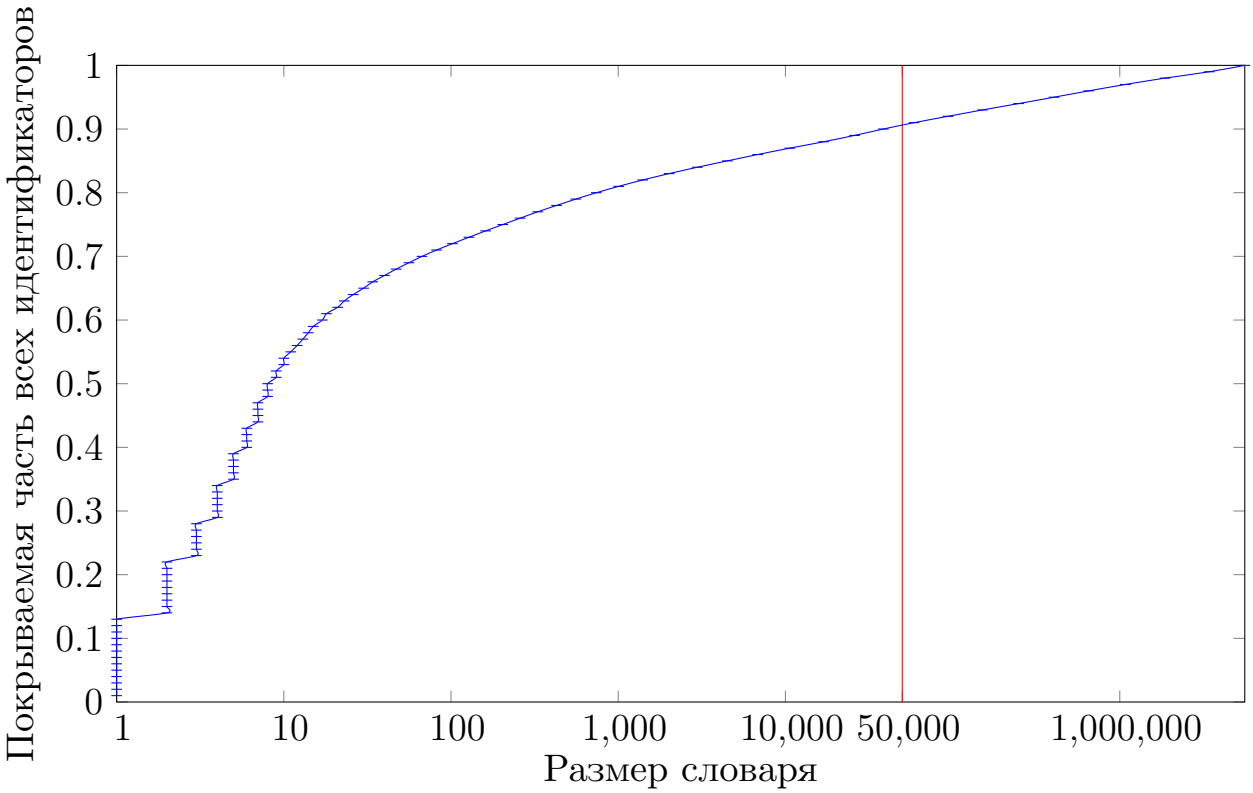

Рис. 3: Покрытие множества всех идентификаторов в датасете

С целью исключить менее распространенные токены, для которых недостаточно обучающих данных ввиду их неоднозначности и редкой встречаемости в коде, а также для экономии памяти и обеспечения относительной эффективности обработки словаря при анализе кода, устанавливаем его размер $|V| = 50000$.

5. Апробация

В ходе данной работы были реализованы и обучены несколько детекторов для Python, используемых для предсказания ошибок. Эксперимент проводился на данных, сгенерированных из фрагментов кода, извлеченных из выбранного датасета с помощью реализованного набора инструментов для предобработки[5].

Датасет 150k Python Dataset:

- 219411 файлов для обучения;
- 109669 файлов для валидации.

В таблице 1 представлено количество использованных примеров.

Детектор	Количество данных	
	Обучение	Валидация
Детектор неправильных операторов в бинарных выражениях	3677456	1834956
Детектор неправильных операндов в бинарных выражения	3675316	1815835
Детектор неправильного порядка аргументов в вызовах функций	1181396	568409

Таблица 1: Количество использованных данных

В результате были получены модели-детекторы, имеющие точности, указанные в таблице 2.

Детектор	Точность
Детектор неправильных операторов в бинарных выражениях	92.08%
Детектор неправильных операндов в бинарных выражениях	85.29%
Детектор неправильного порядка аргументов в вызовах функций	92.06%

Таблица 2: Точность детекторов ошибок

[5] `https://github.com/ml-in-programming/DeepBugs`

Заключение

Результаты

В ходе выполнения данной работы были достигнуты следующие результаты:

- разработана архитектура плагинов;
- выбран датасет для Python (150k Python Dataset);
- реализован набор инструментов для предобработки данных из датасета на Python;
- получены векторные представления отдельных элементов кода;
- обучены модели для каждого из выбранных паттернов ошибок для Python;

Также на основе разработанной архитектуры были реализованы и загружены в публичный репозиторий JetBrains следующие плагины, задействующие модели-детекторы:

- плагин для IntelliJ IDEA/WebStorm[6];
- плагин для IntelliJ IDEA/PyCharm[7].

[6] `https://plugins.jetbrains.com/plugin/12220-deepbugsjavascript`
[7] `https://plugins.jetbrains.com/plugin/12218-deepbugspython`

Список литературы

[1] Building Useful Program Analysis Tools Using an Extensible Java Compiler / E. Aftandilian, R. Sauciuc, S. Priya, S. Krishnan // 2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation(SCAM). — Vol. 00. — 2012. — P. 14–23.

[2] Detecting Argument Selection Defects / Andrew Rice, Edward Aftandilian, Ciera Jaspan et al. // Proc. ACM Program. Lang. — 2017. — Oct. — Vol. 1, no. OOPSLA. — P. 104:1–104:22.

[3] Efficient Estimation of Word Representations in Vector Space / Tomas Mikolov, Kai Chen, Greg Corrado, Jeffrey Dean // CoRR. — 2013. — Vol. abs/1301.3781. — 1301.3781.

[4] Hovemeyer David, Pugh William. Finding Bugs is Easy // SIGPLAN Not. — 2004. — Dec. — Vol. 39, no. 12. — P. 92–106.

[5] Learning Programs from Noisy Data / Veselin Raychev, Pavol Bielik, Martin Vechev, Andreas Krause // SIGPLAN Not. — 2016. — Jan. — Vol. 51, no. 1. — P. 761–774.

[6] Monperrus Martin, Bruch Marcel, Mezini Mira. Detecting Missing Method Calls in Object-Oriented Software // ECOOP 2010 – Object-Oriented Programming. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2010. — P. 2–25.

[7] Nomen Est Omen: Exploring and Exploiting Similarities Between Argument and Parameter Names / Hui Liu, Qiurong Liu, Cristian-Alexandru Staicu et al. // Proceedings of the 38th International Conference on Software Engineering. — ICSE '16. — New York, NY, USA : ACM, 2016. — P. 1063–1073.

[8] Pradel Michael, Sen Koushik. DeepBugs: A Learning Approach to Name-based Bug Detection // Proc. ACM Program. Lang. — 2018. — Oct. — Vol. 2, no. OOPSLA. — P. 147:1–147:25.

[9] JetBrains. — What is the IntelliJ Platform?, 2018. — URL: https://www.jetbrains.org/intellij/sdk/docs/intro/intellij_platform.html (дата обращения: 2018-11-10).