Санкт-Петербургский государственный университет

Кафедра системного программирования

Смирнов Олег Евгеньевич

Совершенствование методов классификации типов ошибок в решениях задач онлайн-курсов по программированию

Курсовая работа

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

Консультант:
исследователь в лаборатории JetBrains Research Лобанов А. В.

Санкт-Петербург
2020

Оглавление

Введение

В эпоху глобализации и интернета онлайн-образование ввиду его доступности обретает невероятную популярность, и открытые онлайн-курсы составляют наиболее важную его часть. Однако существует ряд проблем, из-за которых качество этих курсов пока не может приблизиться к университетским, одной из них является масштабируемость получения обратной связи от преподавателя. В частности, эта проблема возникает в массовых открытых онлайн-курсах по программированию, когда получение своевременных и релевантных подсказок по решению практических задач играет значимую роль в освоении материала. Число студентов там порой достигает нескольких тысяч, и у авторов курса, очевидно, нет возможности дать ответ или подсказку каждому. Автоматическая система проверки же в большинстве случаев запускает присланный код решения на наборе тестов и сообщает пользователю вердикт, таким образом, (в случае неудачи) не возвращая искомой полезной информации о том, как исправить ошибку.

На данный момент существует множество подходов к проблеме генерации подсказок в таких случаях. Некоторые из них сводятся к поиску ближайшего к неправильному (по некоторой метрике) правильного решения, а затем предлагают видоизмененный сценарий редактирования в качестве подсказки [23]. Есть и подходы, где предлагается выделять некоторые шаблоны из неудачного решения для последующего сопоставления с существующими, которые уже были размечены экспертами [3]. Проблема настолько актуальна, что существует даже статья, в которой авторы пытаются обобщить этот процесс и систематизировать все подходы, рассматривая концепцию сужений и трансформаций существующих в базе подсказок по критерию их релевантности в конкретных случаях [16].

В рамках исследовательской группы лаборатории JetBrains Research была предложена работа, в которой предлагается алгоритм генерации рукописных подсказок на основе анализа сценариев редактирования, т.е. последовательностей изменений абстрактного синтаксического де-

рева кода (Abstract Syntax Tree, AST) [15]. Подход предусматривает кластеризацию неправильных решений, последующую ручную разметку получившихся кластеров ошибок искомыми подсказками и классификацию новых решений по принадлежности одному из кластеров. На апробации алгоритм показал достойные результаты, и было решено продолжить работу над его улучшением.

Постановка задачи

Целью данной работы является совершенствование существующего подхода для определения и классификации типичных ошибок в коде решений задач онлайн-курсов по программированию.

Для достижения данной цели в рамках работы были поставлены следующие задачи:

1. Провести обзор предметной области.
2. Проанализировать подходы машинного обучения для классификации изменений в исходном коде.
3. Агрегировать наиболее подходящие и разработать улучшенный алгоритм классификации.
4. Исследовать возможность ускорения отдельных частей существующего подхода.
5. Провести сравнительный анализ алгоритмов посредством экспериментов с решениями конкретных учебных задач.
6. Провести апробацию лучшего подхода с использованием имеющихся данных сразу по нескольким задачам.

1. Обзор

1.1. Исходный алгоритм генерации подсказок

Авторы исходного исследования [15] разделяют подход к решению задачи генерации подсказок на два основных этапа:

1. подготовка данных, включающая в себе кластеризацию и разметку полученных кластеров экспертами;

2. построение на основе размеченных данных классификатора ошибок и его применение.

На первом этапе предлагается кластеризовать не сами неправильные решения (иначе в один кластер могут попасть схожие структурно решения, содержащие разные ошибки), а последовательности элементарных изменений AST, которые преобразуют неверное решение к ближайшему (по некоторой метрике) верному. Такие последовательности называются сценариями редактирования. Далее, после применения алгоритма иерархической агломеративной кластеризации (Hierarchical Agglomerative Clustering [17], HAC) несколько самых больших кластеров размечаются экспертами. Утверждается, что в качестве метки кластеру можно присвоить некоторый набор инструкций по исправлению ошибки — искомую подсказку.

Таким образом, к концу первого этапа получается размеченный набор данных, где каждому неправильному решению из тренировочной выборки сопоставлено правильное, и исправление, преобразующее одно к другому, имеет некоторую метку. Чтобы закодировать эти исправления, на втором этапе применяются разные подходы, один из них называется «мешок слов» (Bag-of-Words, BoW). Любой сценарий редактирования представляется в виде вектора, i-тая компонента которого равна количеству i-тых атомарных изменений AST [12] в текущем исправлении. После этого в качестве классификатора используется метод k ближайших соседей: для любого ранее неизвестного неправильного решения ищется ближайшее правильное среди известных, из сце-

нария редактирования составляется искомый BoW-вектор, и для него по метрике косинусного расстояния в размеченных данных находятся ближайшие. С помощью взвешенного голосования предсказываются вероятности того, в каком кластере лежит данное исправление.

Часть результатов исследования представлена в таблице 1. Оказалось, что подход, использующий нечеткую модификацию расстояния Жаккара (подробнее в статье) в качестве метрики между векторами в среднем лучше других, но основанный на BoW-векторах метод практически ему не уступает.

Метрика	Классификатор	PR-AUC
fuz_jac	k-nearest-15	0.713
fuz_jac	k-nearest-10	0.718
fuz_jac	k-nearest-5	0.716
Bag-of-Words	k-nearest-5	0.707

Таблица 1: Результаты апробации классификаторов

В заключении этой статьи указано, что в будущем можно попробовать использовать другие представления исходных данных, а также не только метрические, но и линейные методы классификации, деревья решений и ансамбли моделей.

1.2. Методы поиска ближайших соседей

Как уже было заявлено выше, одной из частей исходного подхода является поиск ближайших соседей. Он применяется как для классификации сценариев редактирования, так и для подбора наиболее похожего правильного решения для неправильных. Современные методы поиска k ближайших соседей в метрическом пространстве включают в себя множество алгоритмов, обзор которых приведен в статье [4]. Одними из самых популярных структур данных, используемых для увеличения скорости поиска, являются метрические деревья (Ball trees, [18]) и KD-деревья [19]. Однако во время поиска соседей в пространстве решений явных векторов просто нет — есть только исходный код и

функция, определяющая расстояние между двумя объектами (предполагаемая метрика). В таком случае можно использовать подходы векторного представления кода на основе дистрибутивной семантики (code2vec, [24]), но получающиеся вектора имеют довольно большую размерность, и производительность описанных выше структур данных падает.

Оригинальную эвристику поиска ближайших соседей предлагают авторы статьи [21]: если использовать указанную модификацию алгоритма кластеризации, то для набора из n правильных решений можно построить $\Omega(\sqrt{n})$ кластеров по $\mathcal{O}(\sqrt{n})$ элементов каждый. Это значит, что теперь вместо поиска ближайшего среди абсолютно всех правильных решений за $\mathcal{O}(n)$ можно искать соседа только среди некоторых репрезентативных элементов каждого кластера, сократив асимптотику поиска до $\Omega(\sqrt{n})$.

1.3. Вероятностные подходы классификации изменений в коде

Задача классификации изменений кода с целью выявления общих ошибок довольно специфична. Большинство исследований в этой области так или иначе связаны с классификацией типов коммитов в системах контроля версий, а также генерацией сообщений для них. Существует и статья [22], в которой авторы пытаются вручную задать правила определения рефакторингов кода в конкретных изменениях, но в нашем случае они неприменимы, так как число рефакторингов там ограничено, а в учебной задаче студенты могут допускать неограниченное число ошибок.

1.3.1. Методы классического машинного обучения для классификации коммитов

В статьях [2, 13] авторы исследуют возможность определить характер изменения, которое вносит коммит, по его содержанию. В их моделях присутствуют изменения нескольких типов: коррекционные

(исправляющие ошибки); адаптирующие какие-либо внешние ресурсы; улучшающие код; добавляющие новую функциональность; и оставшиеся, нерелевантные. И если в первой статье для определения искомого типа рефакторинга предлагается рассматривать только распределение ключевых слов из сообщения коммита, то во второй используются еще и тип изменения исходного кода. С помощью различных методов машинного обучения и их ансамблей, а также градиентного бустинга [6], авторы добились неплохих показателей доли правильных ответов классификатора (а также легко интерпретируемых результатов за счет деревьев решений), но такие подходы неприменимы в поставленной задаче в силу значительной ориентированности на мета-данные коммита.

В исследовании [10] для классификации ошибочных изменений уже используются глобальные метрики кода, такие как максимальная вложенность блоков и комментарии, а также Bag-of-Words из токенов добавленных и удаленных строк; затем применяется метод классификации с помощью опорных векторов. В учебных задачах объём кода зачастую небольшой, а учитывая то, как сильно могут различаться подходы студентов, найти общие метрики для выявления в них ошибок не представляется возможным.

1.3.2. Применение рекуррентных нейронных сетей

Существует также немало исследований, посвященных смежной задаче генерации описаний и комментариев для коммитов. Там можно найти множество идей по представлению данных, полезных при классификации изменений в коде. Почти все они основаны на глубоком обучении, и, в частности, на рекуррентных нейронных сетях.

Рекуррентные нейронные сети (Recurrent Neural Networks, RNN) — особой вид архитектуры нейронных сетей, основанный на последовательном распространении информации за счет передачи скрытых состояний (рис. 1). При их использовании появляется возможность трансформации последовательностей переменной длины с сохранением некоторого смысла контекста, из-за этого, в частности, они получили широкое распространение в области обработки естественных языков.

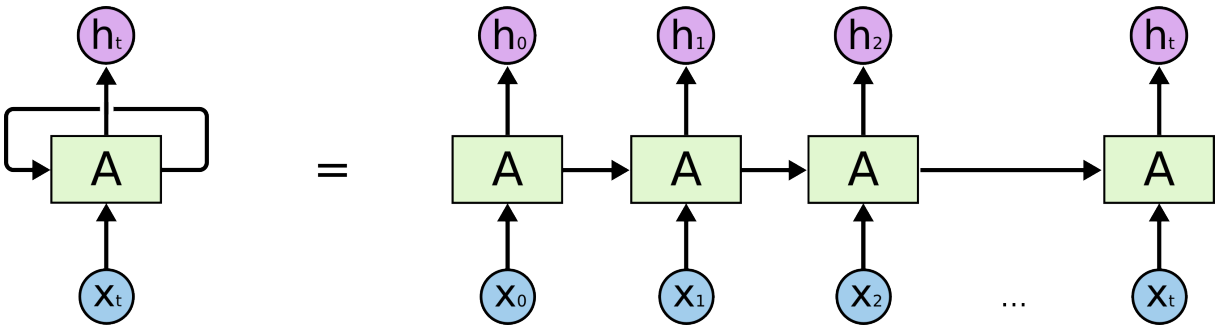

Рис 1: Рекуррентная нейронная сеть слева и ее развернутое представление справа. $\{x_i\}_{i=0}^{t}$ — входящая последовательность, блок A — искомый рекуррентный слой, $\{h_i\}_{i=0}^{t}$ — последовательность скрытых состояний слоя.

На практике, однако, RNN в чистом виде встречается достаточно редко [9], используются же в основном модели Long Short-Term Memory [7], Gated Recurrent Unit [5], и другие.

В задачах генерации описания коммитов [8, 1] авторы чаще всего применяют seq2seq-модели [20]: сначала часть нейронной сети, называемая энкодером, сворачивает набор токенов коммита в некое векторное представление, а затем часть, именуемая декодером, преобразует его в искомое описание на естественном языке (рис. 2).

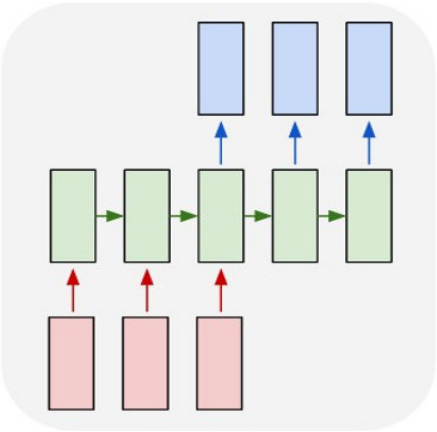

Рис 2: Seq2seq модель нейронной сети. Красным цветом обозначена входящая последовательность, зеленым — развернутое представление рекуррентного слоя, синим — итоговая последовательность.

Для улучшение качества работы сетей используются методы обработки токенов, которых нет в словаре, а также механизм внимания (Attention), который может быть полезен в контексте задачи классификации. Механизм внимания — это отдельный полносвязный слой нейронной сети, обучающий некоторый набор весов так, что в контексте данной задачи более «важные» компоненты тензора данных будут иметь больший вес, чем менее «важные». Таким образом, появляется возможность увеличивать смысловую нагрузку векторного представления последовательности токенов, что может пригодиться при векторизации изменений в искомой задаче классификации.

2. Разработанное решение

Работа с исходным проектом [14] преследовала две основных цели: увеличение точности предсказаний алгоритма и уменьшение времени работы самых медленных его частей. Некоторые из реализованных в ходе данной работы улучшений оригинального подхода описаны ниже.

2.1. Аугментация данных

Набор данных из размеченных сценариев редактирования, полученный на первом этапе описанного в исходной работе подхода, сравнительно мал, так как метки получали только достаточно большие кластеры, а решений, не укладывающихся в рамки какого-либо подхода, тоже довольно много. В связи с этим был применен следующий метод аугментации данных: для каждого неправильного решения из тренировочной выборки теперь ищется не одно, а три ближайших по метрике правильных решения (рис. 3), где константа $k = 3$ была подобрана экспериментальным путем. Тем самым количество данных увеличивается в три раза. Более того, теперь вектора, соответствующие сценариям редактирования, становятся больше похожими на те вектора, для которых позже производится классификация, ведь если для тренировочной выборки точно существовала пара верное-неверное решение из одной сессии одного студента, то в случае классификации на практике это не так, правильного решения из текущей сессии пока что не существует, но нужна подсказка, и приходится искать ближайшее верное среди тренировочных.

2.2. Поиск соседей в пространстве решений

С помощью сэмплирующего профилировщика Java Flight Recorder было подтверждено, что поиск ближайшего элемента в пространстве решений является самой длительной частью исходного подхода. Учитывая, что искать пары правильное-неправильное решение нужно не только при обучении моделей, но и при ответе на запрос подсказки

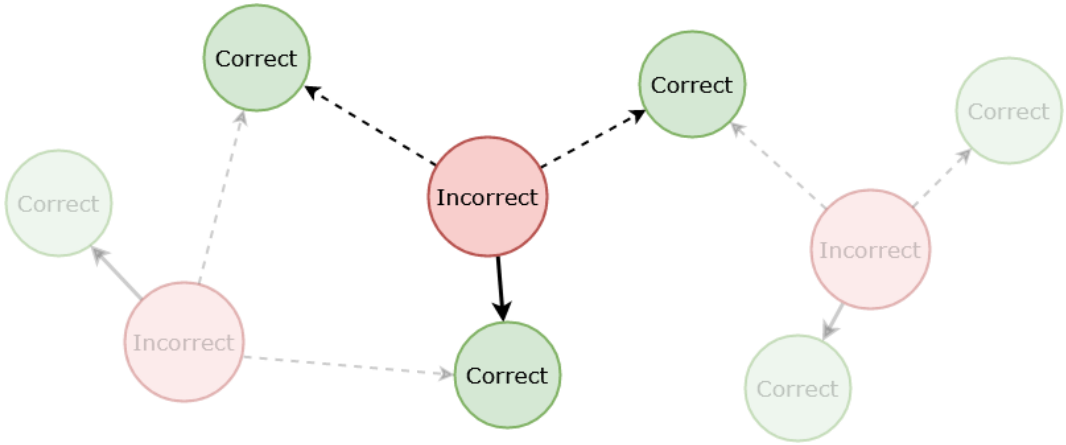

Рис 3: Схема аугментации данных в пространстве решений. Для неправильных решений (вершины красного цвета, Incorrect) ищется три ближайших правильных (зеленых, Correct). Стрелка между вершинами соответствует сценарию редактирования — пунктирная, если решения из разных пользовательских сессий, и полужирная, если из одной (однако не всегда верное решение из текущей сессии является ближайшим).

в промышленном использовании, оптимизация этого фрагмента представлялась наиболее востребованной.

Предлагаемая эвристика основана на подходе, используемом в статье [21] — его асимптотические преимущества уже были описаны в обзоре. Чтобы для n корректных решений построить искомые $\Omega(\sqrt{n})$ кластеров, были внесены некоторые изменения в алгоритм HAC: кластеры объединялись, если и только если суммарный размер кластера, получаемого при слиянии, был меньше, чем $\sqrt{n}$. Таким образом, при правильно подобранном пороге кластеризации (гиперпараметр), получались $\Omega(\sqrt{n})$ кластеров по $\mathcal{O}(\sqrt{n})$ решений каждый. Далее, для каждого кластера выбирался репрезентативный элемент — наименее удаленное (по метрике редакционного расстояния) от остальных правильное решение. В итоге, чтобы найти ближайшее корректное решение для текущего некорректного (затем получить сценарий редактирования, и работать уже с ним), предлагается выбрать ближайший репрезентативный элемент за $\Omega(\sqrt{n})$ и искать искомого соседа среди элементов соответствующего кластера. Как показали результаты апробации, качество классификации при таком подходе падает незначительно, хотя

скрипты редактирования в среднем и увеличиваются на 10.23%, тогда как время работы алгоритма становится заметно лучше (таблица 4).

2.3. Последовательности токенов

В исходном подходе каждое атомарное изменение AST (действия с вершинами дерева четырёх типов: вставка, удаление, модификация, перемещение) из сценария редактирования кодировалось одним из шести способов, которые в разной степени описывали это изменение. Представление могло быть как максимально сжатым и содержать только информацию об изменяемой вершине, так и максимально подробным, т.е. содержать ещё информацию о её родителе и прародителе — контексте в дереве. Из полученных кодов составлялся словарь, а далее, с его помощью, BoW-вектора, описанные в обзоре.

Текущий подход предлагает работать не с BoW-векторами, а с последовательностями токенов, получившимися при развертке сценариев редактирования алгоритмом, который подробно кодирует тип и метки изменяемой вершины дерева, её родителя, а также родителя её родителя. Токеном предлагается называть наименьшую единицу, несущую полезную информацию о какой-либо части атомарного изменения — например, тип удаляемой вершины, метку содержимого родителя перемещаемой вершины в новом контексте, тип текущего атомарного изменения, и т.д.

С таким подходом у классификатора появляется возможность не только узнать информацию о количестве i-тых атомарных изменений в сценарии редактирования, но и обнаружить некоторые паттерны в расположении токенов в последовательности, а также работать с неизвестными ранее изменениями за счет универсальности представления набором искомых токенов.

2.4. Методы глубого обучения

Учитывая небольшой объём тренировочной выборки в начале исследования, применять методы классического машинного, а не глубокого

обучения (в силу их легкой переобучаемости) представлялось логичным. Однако после аугментации данных решено было попробовать использовать нейросетевой подход, архитектура которого представлена на рис. 4.

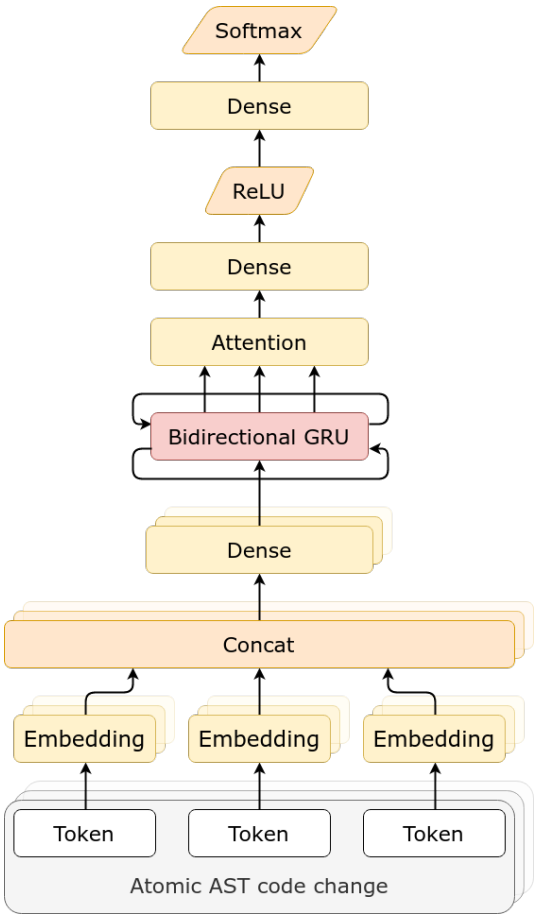

Рис 4: Предлагаемая архитектура нейронной сети.

Она предусматривает следующую последовательность действий при обучении:

1. с помощью Embedding-слоя получить векторные представления каждого токена из заданного набора;
2. разбить представления (embeddings) на группы, соответствующие атомарным изменениям;
3. объединить embeddings из одной группы конкатенацией;

4. провести их через полносвязный слой, получив уже векторные представления для атомарных изменений;

5. текущий набор представлений подать слою Bidirectional GRU (Gated Recurrent Unit), получить последовательность скрытых состояний этого слоя;

6. трансформировать их в одно представление с помощью механизма внимания;

7. подать его полносвязному слою с функцией активации ReLU;

8. провести через еще один полносвязный слой для получения вектора нужной размерности;

9. применить функцию Softmax для преобразования значений в вероятности классов.

Обучение нейронной сети происходило с помощью метода ранней остановки, в качестве функции потерь использована кросс-энтропия, метод стохастического градиентного спуска — Adam [11]. Во избежание переобучения была использована регуляризация: механизм сокращения весов (weight-decay), а также метод исключения (dropout) случайных 30% нейронов из полносвязного и Embedding-слоев.

2.5. Детали реализации

Алгоритмы выделения токенов, аугментации, поиска и кэширования соседей в пространстве решений, а так же модифицированной кластеризации и выбора репрезентативных элементов были написаны на языке Java в рамках исходного проекта [14]. Для реализации классификатора в силу гибкости и обилия библиотек для машинного обучения был выбран Python, а для прототипирования нейронных сетей — популярный фреймворк PyTorch. Обучение моделей происходило на платформе GPU. Итогом работы стали два открытых пулл-реквеста в публичный репозиторий проекта.

3. Апробация

Разработанные алгоритмы и модели были переданы авторам оригинального решения [14], которые провели эксперименты над данными из их статьи [15], предоставленными платформой Stepik. Методики экспериментов были разработаны автором данной курсовой работы, разметка и прочие операции с данными — сотрудниками JetBrains Research. Автору были предоставлены лишь анонимизированные результаты работы алгоритмов.

3.1. Набор данных

К четырем задачам, отличающимся тематикой и средним размером решения, которые были размечены и обработаны еще в исходном проекте, для чистоты эксперимента было добавлено две новых избранных задачи (*filter* и *integral*), разметка которых была проведена с нуля. Подробнее о том, как происходила разметка задач (назначение парам правильное-неправильное решение метки, являющейся искомой подсказкой для исправление ошибки) и их кластеризация, описано в [15]. В таблице 2 для каждой из шести задач указаны параметры тренировочной, валидационной и тестовой выборки решений: N – общее количество сессий студентов, LOC – средний размер решения в строчках, $N_{correct}$ – количество правильных решений (только для тренировочной, так как для тестовых правильное решение ищется среди уже известных), N_{wrong} – количество неправильных решений. Задача *deserialization* подразумевала десериализацию массива элементов, *factorial* – вычисление факториала с помощью BigInteger, *loggers* – работу с логгированием, *reflection* – определение имени метода, *filter* – работу с Stream API и Collections, *integral* – численное интегрирование с заданной точностью.

3.2. Независимое тестирование классификаторов

Для сравнения классификаторов, как и в исходной статье, была применена несколько видоизменённая метрика качества PR AUC (площадь

Problem	N	LOC	Train		Validate	Test
			$N_{correct}$	N_{wrong}	N_{wrong}	N_{wrong}
deserialization	2624	19	2294	1643	60	140
factorial	8883	14	8330	5357	60	140
loggers	3403	15	3137	2067	60	140
reflection	4217	11	3827	2742	60	140
filter	2673	24	2549	1747	60	140
integral	4952	12	4539	2859	60	140

Таблица 2: Информация о выборках решений по задачам.

под precision-recall кривой). Подробнее, если TP – количество правильных ответов, в которых классификатор уверен, FP – количество неверных ответов, в которых классификатор также уверен, а N – общий размер выборки, то под *precision* и *recall* понимались выражения 1 и 2, а кривая строилась по точкам с разными порогами уверенности классификатора. Чем ближе значение метрики к единице, тем лучше качество тестируемой модели.

$$precision = \frac{TP}{TP + FP} \tag{1}$$

$$recall = \frac{TP}{N} \tag{2}$$

Сравнение классификаторов проводилось с одинаковыми исходными данными: решения уже были кластеризованы алгоритмом HAC на основе BoW-векторов с общим размером словаря 20000 (константа была подобрана еще в исходном исследовании), а для всех неправильных решений из тестовой выборки уже был найден ближайший сосед — правильное решение из тренировочной выборки. В таблице 3 приведены результаты сравнения по метрике PR AUC следующих алгоритмов:

1. Нескольких версий классификатора KNN (алгоритм K ближайших соседей) с различными K над BoW-векторами длиной 20000 с метрикой косинусного расстояния.

2. Предлагаемой нейросетевой модели AttentionalBidirectionalGRU, архитектура которой была описана в разделе с реализацией, а гипер-параметры подобраны на валидационной выборке отдельно для каждой из задач.

3. KNN над представлениями (embeddings), полученными на предпоследнем слое нейронной сети. Эвристически они должны обладать дистрибутивной семантикой и содержать в себе некоторую информацию о сути сценария редактирования, который кодируют.

Model	*deserialization*	*factorial*	*loggers*	*reflection*	*filter*	*integral*
KNN-1	0.645	0.681	0.728	0.688	0.357	0.578
KNN-5	0.679	0.726	0.755	0.765	0.388	0.603
KNN-15	0.708	**0.736**	0.783	0.772	0.400	0,613
AttBiGRU	**0.731**	0.717	0.794	**0.780**	0.562	0.624
KNN-embed	0.693	0.666	**0.795**	0.774	**0.588**	**0.634**

Таблица 3: Сравнение качества работы классификаторов на тестовой выборке по значению метрики PR AUC.

Таким образом, предлагаемый нейросетевой подход на абсолютном большинстве задач показывает лучшие результаты, чем классический поиск ближайших соседей в пространстве BoW-векторов. Более того, можно заметить, что метрика качества возрастает наиболее значительно в задачах бóльшим средним размеров решений — в частности, *filter* и *deserialization*.

3.3. Измерение времени поиска ближайших соседей

Для исследования эвристики поиска соседей в пространстве решений среди репрезентативных элементов кластеров необходимо было сравнивать два «ортогональных» параметра: время работы алгоритма и качество классификации на полученных данных (поскольку оно менялось, ведь изменялись и скрипты). Из-за этого результаты классификации фиксировались только у одной модели AttentionalBidirectionalGRU

(как показавшей наиболее достойные результаты на апробации) при разных входных данных: тестовой выборке из скриптов редактирования, полученных поиском во всём пространстве решений, и тестовой выборке, полученной поиском только по репрезентативным элементам кластеров. Замеры времени проводились стандартными средствами Java на ЭВМ со следующими характеристиками: CPU – Intel Core i3-6006U, 2.0 GHz, 2 cores (4 threads), 3 Mb cache; RAM – 8Gb, DDR3; OS – Ubuntu 18.04 LTS.

Один замер определял время составления отложенной выборки на 200 решений (валидационной и тестовой вместе), то есть суммарное время поиска ближайшего правильного решения для каждого из двухсот неправильных в текущем наборе данных, тем самым сглаживая выбросы по поиску для каждого в отдельности. Теоретически асимптотика обычного поиска равнялась $\mathcal{O}(kn)$, а эвристического — $\Omega(k\sqrt{n})$, где n — количество правильных решений в тренировочной выборке, а $k = 200$ — количество неправильных решений в отложенной. Первый замер исключался как потенциальный выброс (феномен «холодного старта»), а для оставшихся десяти вычислялось среднее значение и доверительный интервал в 95%, который указан в таблице 4 для каждой из задач.

Problem	Complete search		Heuristic search	
	PR AUC	Time (s)	PR AUC	Time (s)
deserialization	0.731	453,6±4,9	0.715	101,2±3,8
factorial	0.717	209,1±7,7	0.694	42,9±0,5
loggers	0.794	208,5±4,3	0.753	84,7±5,5
reflection	0.780	157,1±1,9	0.766	54,3±2,5
filter	0.562	2029,7±19,5	0.541	172,8±1,2
integral	0.624	1097,1±13,8	0.571	121,2±1,6

Таблица 4: Сравнение качества классификации (PR AUC) на тестовой выборке и времени обработки отложенной выборки по каждой задаче.

Следовательно, при совершенно незначительных флуктуациях метрики качества, предложенная эвристика поиска даёт в среднем четырёхкратное уменьшение времени работы подхода. Можно полагать, что используемый подход замены кластеров на их метрические центры мо-

жет найти применения и в других областях машинного обучения на исходном коде.

3.4. Определение глобальных ошибок посредством анализа большого набора данных

После экспериментов над шестью избранными задачами по отдельности логичным представлялось попробовать кластеризовать сценарии редактирования сразу по многим задачам. Гипотеза была такова, что они должны образовывать кластеры с наиболее частыми и общими ошибками студентов в учебных задачах по программированию. Для этих целей из предоставленного набора данных было выделено 38 задач, содержащих наибольшее количество решений, получены сценарии редактирования для неправильных (поиском правильного среди решений той же конкретной задачи) и запущен алгоритм иерархической агломеративной кластеризации на разреженных BoW-векторах (с общим размером словаря токенов равным 255 631). Из получившихся кластеров 100 наибольших были размечены вручную: оказалось, что только 40 из них содержали в себе скрипты редактирования сразу из нескольких задач — в остальных кластерах ошибки, соответствующие находящимся там исправлениям, были слишком специфичны и не поддавались обобщению.

В выделенных сорока кластерах наиболее часто встречаемыми ошибками были синтаксические: забыты импорты пакетов и выражения return, опечатки, неверные модификаторы доступа и названия методов/классов. Однако встречались и некоторые семантические ошибки, например, отладочный вывод, неинициализированные переменные, неверное преобразование типов, или неправильные константы (модуля/точности/...). Получается, что хоть результаты кластеризации и можно считать удовлетворительными, потенциальная полезность предлагаемых классификатором исправлений для других задач будет маленькой, ведь выделенные ошибки обнаруживаются еще на стадии компиляции программы. Возможно, такой подход стоит развивать в дальнейшем как некото-

рый базовый алгоритм, умеющий предсказывать большой, но конечный набор исправлений, который можно дообучать «видеть» специфичные ошибки на каждой новой задаче в отдельности.

4. Заключение

В ходе курсовой работы были достигнуты следующие результаты:

1. Проведен обзор предметной области, изучены подходы и методы машинного обучения для кластеризации и классификации изменений в коде.

2. Разработан нейросетевой алгоритм классификации сценариев редактирования кода в учебных задачах.

3. Реализован эвристический алгоритм поиска в пространстве решений, значительно превосходящий по скорости исходный аналог. За время работы над курсовой были сделаны два пулл-реквеста в публичный репозиторий проекта [14].

4. Проведена апробация нового подхода и сравнительный анализ алгоритмов на реальных данных по шести учебным задачам с платформы Stepik.

Таким образом, из полученных результатов апробации можно сделать вывод, что исходный подход был оптимизирован как по качеству классификации исправлений, так и по общему времени работы алгоритма. В будущем планируется исследовать возможности обобщения текущего подхода работы с исправлениями в коде на глобальный уровень, например, предсказывать вероятные ошибки при коммите/пулл-реквесте в репозиторий публичного проекта на платформе GitHub.

Список литературы

[1] Automatic Generation of Pull Request Descriptions / Zhongxin Liu, Xin Xia, Christoph Treude et al. // arXiv preprint arXiv:1909.06987. — 2019.

[2] Automatic classication of large changes into maintenance categories / Abram Hindle, Daniel M German, Michael W Godfrey, Richard C Holt // 2009 IEEE 17th International Conference on Program Comprehension / IEEE. — 2009. — P. 30–39.

[3] Autonomously generating hints by inferring problem solving policies / Chris Piech, Mehran Sahami, Jonathan Huang, Leonidas Guibas // Proceedings of the Second (2015) ACM Conference on Learning@ Scale / ACM. — 2015. — P. 195–204.

[4] Bhatia Nitin et al. Survey of nearest neighbor techniques // arXiv preprint arXiv:1007.0085. — 2010.

[5] Empirical evaluation of gated recurrent neural networks on sequence modeling / Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, Yoshua Bengio // arXiv preprint arXiv:1412.3555. — 2014.

[6] Friedman Jerome H. Greedy function approximation: a gradient boosting machine // Annals of statistics. — 2001. — P. 1189–1232.

[7] Hochreiter Sepp, Schmidhuber Jürgen. Long short-term memory // Neural computation. — 1997. — Vol. 9, no. 8. — P. 1735–1780.

[8] Jiang Siyuan, Armaly Ameer, McMillan Collin. Automatically generating commit messages from diffs using neural machine translation // Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering / IEEE Press. — 2017. — P. 135–146.

[9] Jozefowicz Rafal, Zaremba Wojciech, Sutskever Ilya. An empirical

exploration of recurrent network architectures // International Conference on Machine Learning. — 2015. — P. 2342–2350.

[10] Kim Sunghun, Whitehead Jr E James, Zhang Yi. Classifying software changes: Clean or buggy? // IEEE Transactions on Software Engineering. — 2008. — Vol. 34, no. 2. — P. 181–196.

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

[12] Lehnert Steffen, Riebisch Matthias et al. A taxonomy of change types and its application in software evolution // 2012 IEEE 19th International Conference and Workshops on Engineering of Computer-Based Systems / IEEE. — 2012. — P. 98–107.

[13] Levin Stanislav, Yehudai Amiram. Boosting automatic commit classification into maintenance activities by utilizing source code changes // Proceedings of the 13th International Conference on Predictive Models and Data Analytics in Software Engineering / ACM. — 2017. — P. 97–106.

[14] Lobanov Artyom. Bugs Classification. — `https://github.com/JetBrains-Research/bugs-classification`. — 2019.

[15] Lobanov Artyom, Bryksin Timofey, Shpilman Alexey. Automatic Classification of Error Types in Solutions to Programming Assignments at Online Learning Platform // International Conference on Artificial Intelligence in Education / Springer. — 2019. — P. 174–178.

[16] McBroom Jessica, Koprinska Irena, Yacef Kalina. A Survey of Automated Programming Hint Generation–The HINTS Framework // arXiv preprint arXiv:1908.11566. — 2019.

[17] Müllner Daniel. Modern hierarchical, agglomerative clustering algorithms // arXiv preprint arXiv:1109.2378. — 2011.

[18] Omohundro Stephen M. Five balltree construction algorithms. — International Computer Science Institute Berkeley, 1989.

[19] Sproull Robert F. Refinements to nearest-neighbor searching ink-dimensional trees // Algorithmica. — 1991. — Vol. 6, no. 1-6. — P. 579–589.

[20] Sutskever I, Vinyals O, Le QV. Sequence to sequence learning with neural networks // Advances in NIPS. — 2014.

[21] TipsC: tips and corrections for programming MOOCs / Saksham Sharma, Pallav Agarwal, Parv Mor, Amey Karkare // International Conference on Artificial Intelligence in Education / Springer. — 2018. — P. 322–326.

[22] Weissgerber Peter, Diehl Stephan. Identifying refactorings from source-code changes // 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06) / IEEE. — 2006. — P. 231–240.

[23] Writing reusable code feedback at scale with mixed-initiative program synthesis / Andrew Head, Elena Glassman, Gustavo Soares et al. // Proceedings of the Fourth (2017) ACM Conference on Learning@ Scale / ACM. — 2017. — P. 89–98.

[24] code2vec: Learning distributed representations of code / Uri Alon, Meital Zilberstein, Omer Levy, Eran Yahav // Proceedings of the ACM on Programming Languages. — 2019. — Vol. 3, no. POPL. — P. 1–29.