Санкт-Петербургский государственный университет

Кафедра системного программирования

Чебыкин Александр Евгеньевич

Применение машинного обучения для анализа использования API

Курсовая работа

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

Санкт-Петербург
2017

Оглавление

Введение **3**

1. Обзор литературы **5**

1.1. Deep API Learning 5

1.2. Другие подходы 7

2. Получение данных **9**

2.1. Получение списка большого количества репозиториев . . 9

2.1.1. Количество проектов 10

2.1.2. Количество ответов на поисковый запрос 10

2.1.3. Ограничение количества запросов 10

2.2. Скачивание репозиториев 11

2.3. Извлечение данных 11

3. Построение модели **13**

4. Оценка работы модели **15**

Заключение **16**

Список литературы **17**

Введение

Анализ программного кода для различных целей ведётся уже не одно десятилетие. Исследователи занимаются проблемами анализа качества кода [2], генерации подходящего имени метода по его телу [1], генерации отрывков кода по текстовому запросу [8, 10]. Последнее особенно интересно тем, что экономит время программистов, потенциально давая им возможность не изучать большое количество программных библиотек в целях реализации некоей стандартной функциональности; необходимость заниматься этим сейчас — значимая проблема [9].

В рамках данной курсовой работы рассматривается предлагающая решение этой проблемы статья Deep API Learning [5]. Алгоритм генерации цепочек вызовов функций по текстовому запросу, представленный в ней, основывается на технологиях глубокого машинного обучения, используемых для обработки естественных языков. Подробное описание алгоритма и всех сопутствующих понятий приводится в разделе 1.

Целями данной работы является исследование релевантности алгоритма, а также возможности его переноса на другой язык программирования по сравнению с использовавшимся авторами оригинальной статьи.

Для достижения данной цели ставятся следующие задачи:

- собрать набор данных для обучения модели алгоритма;
- решить проблемы сбора данных, связанные со сменой целевого языка;
- построить модель;
- обучить модель и сравнить полученный результат с оригинальным.

Предполагается, что достижение поставленных задач поспособствует исследованиям в данной области, как в теоретическом плане, так и в практическом: наличие собранного набора данных, а также инструментов, предназначенных для этого, сделает будущие исследования проще.

Релевантность задачи и полученных в ходе этой работы результатов подтверждается принятием статей о ней на студеческие конференции SEIM-2017 и "Современные технологии в теории и практике программирования — 2017"; статья с последней конференции опубликована в её сборнике, статья с первой — будет опубликована в ближайшее время.

1. Обзор литературы

1.1. Deep API Learning

Основой данной работы является уже упомянутый алгоритм из статьи Deep API Learning. Его ключевая идея — представить генерацию кода как перевод с английского на язык вызова функций (в таком языке слова — имена функций целевого языка программирования, предложения — упорядоченные наборы слов). В связи с этим, алгоритм основывается на существующих до него передовых техниках глубоко машинного обучения, нацеленных на перевод между естественными языками.

На высоком уровне используемые техники классифицируются как Sequence-to-Sequence (последовательность-к-последовательности): модель должна определенным образом по входной последовательности слов сгененрировать выходную. Примером для перевода с английского языка на русский может быть пара последовательностей (”generate random nuber” — ”сгенерировать случайное число”), а для перевода с английского языка на язык вызовов функций — пара (”generate random number” — ”Random.new Random.nextInt”).

Реализуется Sequence-to-Sequence модель RNN Кодер-Декодером — специализированной парой рекуррентных нейронных сетей. Рекуррентная нейронная сеть (Recurrent Neural Network, RNN) — вид модели глубокого машинного обучения, обладающей долговременной памятью. Кодер — первая сеть — поэлементно считывает входную последовательность, обновляя собственные параметры в скрытом состоянии. После окончания считывания считается, что в полученном скрытом состоянии кодера заключена суть исходного предложения, поэтому оно передаётся в качестве вектора контекста декодеру — второй сети. Декодер на основе вектора контекста и последнего сгенерированного собой слова предсказывает каждое следующее слово, пока не предскажет окончание строки — специальный символ <EOS>.

Пример работы Кодер-Декодера представлен на рисунке 1. Входная последовательность — “generate random number”, выходная —

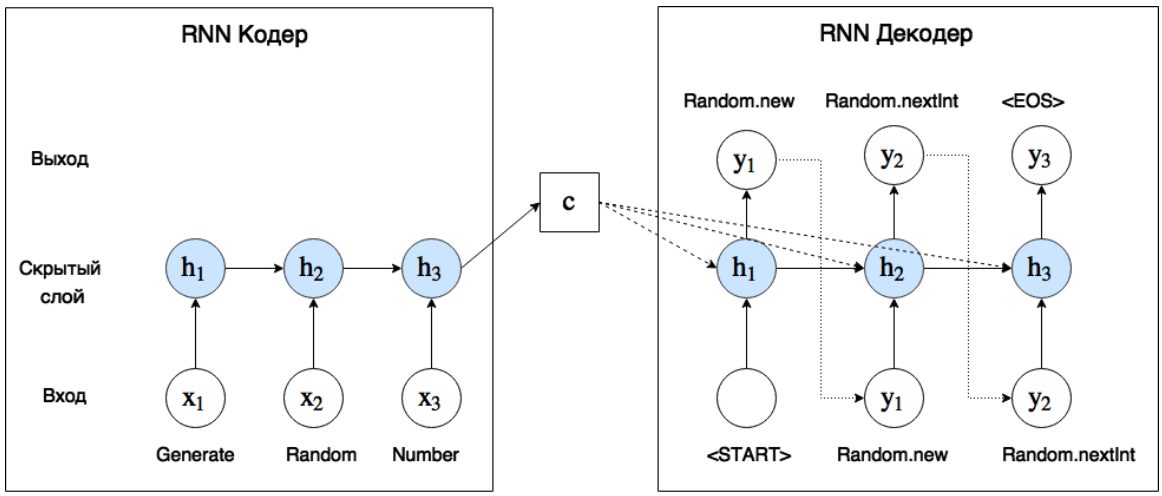

Рис. 1: Пример работы RNN Кодер-Декодера

“Random.new Random.nextInt”.

Для простоты понимания, на иллюстрации состояния нейронных сетей расширены по времени: поскольку нейронная сеть рекуррентная, переход из состояния должен вести в него же (таким образом и достигается наличие долговременной памяти). Таким образом h1, h2, h3 — одно и то же состояние в моменты времени 1, 2, 3.

В целях улучшения качества работы модели часто используется модель внимания: вместо использования только лишь финального состояния кодера в качестве вектора контекста, на каждом шаге генерации используется взвешенная сумма всех исторических состояний кодера — таким образом, разные слова входной последовательности оказывают неравное влияние на разные слова выходной последовательности.

$$c_j = \sum_{t=1}^{T} a_{jt} h_t$$

Здесь c_j — вектор контекста для шага j, h_t — исторические состояния кодера, a_{jt} — веса этих состояний для шага j.

Коэффициенты a_{jt} взвешенных сумм предсказываются отдельной специально натренированной нейронной сетью.

В алгоритме используется еще одно улучшение: лучевой поиск. Дело в том, что хотелось бы генерировать по входу наиболее вероятную выходную последовательность. Однако при стандартном алгоритме ге-

нерации на каждом шаге выбирается самое вероятное из текущих возможных слов. При этом мы можем не получить наилучший выход из-за того, что первое слово в нём оказалось неоптимальным. При использовании лучевого поиска на каждом шаге отслеживается n наиболее вероятных выходных цепочек.

1.2. Другие подходы

До Deep API Learning были попытки решить схожие по направлению проблемы с помощью статистических методов, не связанных с машинным обучением. Наиболее выдающиеся из этих подходов:

- SWIM[8] генерирует код по текстовому запросу, основываясь на закрытом наборе данных — статистике запросов в поисковой системы Bing;
- MAPO[10] по названию функции генерирует наиболее релевантные шаблоны её использования;
- UP-Miner [7] — улучшенная версия MAPO, достигающая лучших результатов в генерации благодаря более сложному алгоритму.

В соответствующий статьях авторы называют свои алгоритмы многообещающими, но, насколько известно автору данной курсовой работы, алгоритмы эти так и не были внедрены. У реализации алгоритма Deep API, напротив, есть публичный рабочий прототип [4], выдающий похожие на правду ответы.

Также создатели Deep API оценивают свои результаты и результаты предыдущих алгоритмов по человеконезависимой метрике BLEU. Это известная метрика, часто применяющаяся для оценки качества точности машинного перевода. Её результат (по шкале от 0 до 100) показывает, насколько сгенерированная последовательность близка к написанной человеком.

По представленным в статье оценкам, Deep API превосходит предыдущие техники. Поскольку не у всех из них была подсчитана оцен-

ка BLEU, авторы Deep API реализовали и оценили прошлые алгоритмы самостоятельно. В итоге UP-Miner получил оценку 11.97, SWIM — 19.90, DeepAPI — 54.42.

Рис. 2: Схема работы приложений

2. Получение данных

Отдельное внимание стоит уделить процессу сбора данных для исследования и тренировки модели.

GitHub [6] — популярный хостинг проектов с открытым исходным кодом.

Работа с сайтом делится на 3 больших этапа, которым соответствуют 3 независимых приложения, совместная работа которых позволяет достичь поставленной задачи получения данных из исходного кода.

Код всех приложений находится в открытом доступе с лицензией MIT по адресу `https://www.github.com/AwesomeLemon/api-extraction`

Общая архитектура приведена на рисунке 2

2.1. Получение списка большого количества репозиториев

Список репозиториев получается поиском по GitHub с ограничением на язык репозитория. Для этого используется открытое API GitHub. Но оказалось, что в API на языке C# (реализуемым библиотекой Octokit.NET) нет механизма paging, позволяющего получать доступ дальше, чем к первой странице результатов поиска.

В связи с этим, хотя большая часть представленных в данной работе инструментов написана на C#, эта часть проекта реализуется отдель-

ным скриптом на Ruby — в API на этом языке (реализуемом библиотекой Octokit.rb) нужная функциональность есть.

На этом этапе возникают и другие проблемы.

2.1.1. Количество проектов

Авторы оригинальной статьи работали с 442,928 проектов на Java. Для фильтрации проектов авторы выставили условие, что у каждого из них должна быть хотя бы одна звезда.

По подобному запросу для языка C# выдаётся 121,672 репозитория (`https://github.com/search?utf8=%E2%9C%93&q=language%3AC%23+stars%3A%3E0&type=Repositories&ref=searchresults`).

Таким образом, по сравнению с Java, репозиториев, с которыми можно потенциально работать, примерно в 4 раза меньше. Однако судя по имеющимся данным, кажется возможным собрать количество данных того же порядка. Авторы в итоге получили 7,519,907 пар "описание на естественном языке — список вызовов API". После обработки 49,679 репозиториев в нашем распоряжении оказываются 1,650,445 пар.

2.1.2. Количество ответов на поисковый запрос

Также количество поисковых результатов с Github ограничено 1000 ссылок. Для обхода этого ограничения вместо одного глобального запроса используется множество маленьких, уточняющие дату создания репозитория, например "1 — 10 марта 2014". В ходе небольшого эксперимента оказалось, что промежуток в 10 дней достаточно мал, чтобы в любой год количество созданных за такой промежуток времени репозиториев не превосходило 1000.

2.1.3. Ограничение количества запросов

Также GitHub ограничивает количество запросов в минуту тридцатью. Из-за предыдущей проблемы необходимо делать много запросов, и этот предел быстро достигается, после чего доступ закрывается. Поэтому вводится искусственная задержка в 20 секунд между каждыми

десятью запросами — таким образом, мы делаем около 30 запросов минуту, но при этом строго меньше 30.

2.2. Скачивание репозиториев

Для работы с системой контроля версий git на C# существует специальная библиотека LibGit2Sharp, позволяющая взаимодействовать с репозиториями. Здесь она используется для скачивания репозиториев по списку, полученному на предыдущем этапе.

2.3. Извлечение данных

На этом этапе возникают пролблемы из-за выбранного целевого языка. В Java из исходного кода можно легко получить тип (или надтип) переменной из её объявления. Однако в языке C#, начиная с версии 3.0, переменные могут иметь неявный тип var, поэтому для выведения явного типа таких переменных требуется компиляция всего проекта. Это ограничивает количество репозиториев, которые могут быть обработаны.

Кроме этого, обнаруживаются и другие сложностями со стороны языка, например, динамический тип данных dynamic, о котором во время компиляции ничего неизвестно. Но такие проблемы по сравнению с проблемой неявных типов несущественны: был проведен небольшой эксперимент, в виде запуска поиска в исходном коде 470 случайно выбранных репозиториев слова ”dynamic” и ”var”. Было получено всего 4,856 результатов для первого, и 176,561 результатов для второго. Поэтому пока что способы решения подобных неприоритетных проблем не исследуются.

В качестве компилятора используется Roslyn — компилятор C# с открытым исходным кодом, разработанный Microsoft. Чтобы собирать проекты в автоматическом режиме, необходимо, чтобы:

1. не надо было предпринимать никаких дополнительных действий (например, запускать индивидуальные для каждого проекта

скрипты);

2. проект содержал файл с расширением .sln — именно с ним может работать Roslyn.

Этим ограничениям удовлетворяют около 36.3% обработанных нами проектов.

Обработка кода проходит следующим образом:

1. проводится поиск методов, у которых есть структурированный XML-комментарий документации и непустое тело;

2. для каждого из таких методов из комментария извлекается секцию “summary” — в ней по определению должна содержаться кратко описанная суть метода. Она чистится от содержимого в скобках, тэгов и других ненужных символов; в итоге получается словарь пар “метод — комментарий метода”;

3. каждом из методов запускается специально реализованный обходчик синтаксического дерева. Тот обходит узлы дерева и сохраняет последовательность вызовов функций в том порядке, в котором они были бы сделаны при обычном исполнении программы. Таким образом, мы получаем список вызова функций, комментарии у нас уже есть, и в итоге мы имеем описание одной и той же функциональности и на английском языке, и на языке вызовов функций;

Пример обработки метода — на рисунке 3

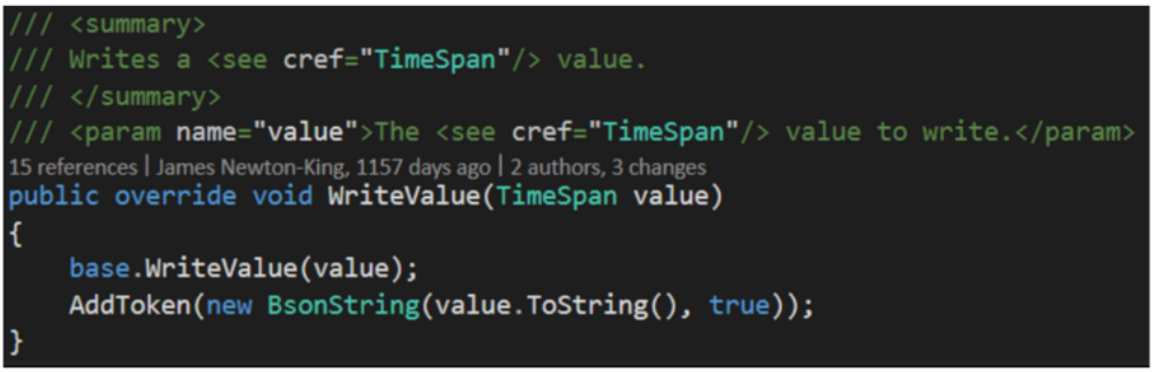

Рис. 3: Пример извлечения данных из комментированного метода

3. Построение модели

Модель из оригинальной статьи была реализована со всеми улучшениями, крому одного: специализированной функции потерь, уменьшающую вероятность появления в выходе слишком часто встречаемых в исходном коде функций. Это улучшение не приоритетно, т.к. чрезвычайно мало влияет на точность работы.

В настоящее время большая часть моделей машинного обучения создаётся на основе специальных фрэймворков. Сначала я попробовал реализовать модель на основе популярной библиотеки TensorFlow от Google. Итоговая модель оказалась довольно ресурсоёмкой. Надо упомянуть, что обучение рекуррентных нейронных сетей — в принципе чрезвычайно ресурсоёмкий процесс, который ведётся на видеокартах. У авторов Deep API имелась для этих целей видеокарта Nvidia K20, я пользовался довольно старой и менее мощной Nvidia GTX 660. В итоге памяти видеокарты не хватило для запуска модели с параметрами, указанными в оригинальной статье. Обучение с урезанными параметрами привело к околонулевым результатам; алгоритм работал верно на очень небольшом количестве искусственных примеров.

В апреле 2017 года Google выпустил новый фрэймворк для машинного обучения, основывающийся на TensorFlow, но оптимизирован-

ный специально для работы с рекуррентными нейронными сетями: tf-seq2seq. С его помощью удалось запустить модели с равными оригинальным параметрами.

Стоит упомянуть, что авторы оригинальной статьи пользовались библиотекой GroundHog, на данный момент устаревшей, и более не поддерживающейся.

В качестве функции потерь в модели Deep API берётся условная логарифмическая функция правдоподобия. Модель тренируется максимизировать её с помощью стохастического градиентного спуска [3] и его оптимизатора AdaDelta [11].

Модель тренируется на 924,593 парах данных в течение 390,000 итераций, её словарь ограничен самыми популярными 10,000 слов в исходном и выходном языках.

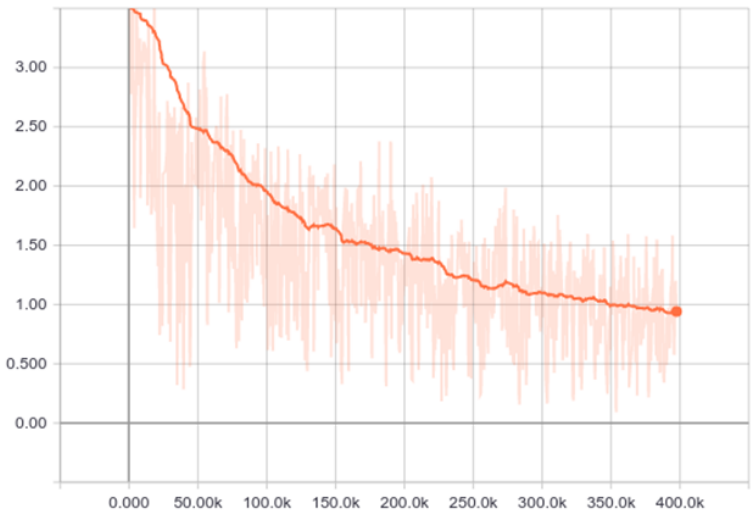

Рис. 4: График изменения функции потерь во время обучения

4. Оценка работы модели

После тренировки модель показывает ненулевые результаты на тестовом множестве из 14,000 пар данных. Стоит отметить, что тестовые примеры не используются во время тренировки, поэтому из положительных результатов на них следует, что модель не просто запомнила данные ей в тренировочном множестве примеры, а действительно научилась обобщать знания. Оценка модели по метрике BLEU растёт от 0.48 после 158,000 итераций до 1.95 после 390,000 итераций. Помимо этого, искусственная функция потерь, которую модель минимизирует, стабильно убывает (см. рисунок 4).

Полученный результат гораздо хуже оригинального (54.42 по шкале BLEU). Однако поскольку эксперимент был проведен на меньшем количестве данных и процесс обучения был не таким долгим, а также поскольку прослеживается общая положительная динамика, то можно сделать вывод, что подход идейно перспективен, и исследования в этом направлении стоит продолжать.

Заключение

В результате выполнения данной курсовой работы была исследована релевантность одного из передовых алгоритмов генерации кода, а также возможность смены его целевого языка. Для проведения эксперимента были реализованы несколько программных инструментов по сбору данных из открытых источников; с их помощью был собран массивный набор данных, примененный для обучения модели. Хотя эксперимент не дал результатов, сходных с оригинальными, подход показал себя достаточно перспективным; планируется продолжать исследования по более качественной его реализации.

Список литературы

[1] Allamanis Miltiadis, Peng Hao, Sutton Charles. A Convolutional Attention Network for Extreme Summarization of Source Code. — arXiv : cs.LG/1602.03001v2.

[2] Barstad Vera, Goodwin Morten, Gjøsæter Terje. Predicting Source Code Quality with Static Analysis and Machine Learning. // NIK. — 2014.

[3] Bottou Léon. Large-scale machine learning with stochastic gradient descent // Proceedings of COMPSTAT'2010. — Springer, 2010. — P. 177–186.

[4] Deep API Learning. — 2017. — URL: `http://www.cse.ust.hk/~xguaa/deepapi/` (online; accessed: 19.03.2017).

[5] Deep API learning / Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, Sunghun Kim // Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering / ACM. — 2016. — P. 631–642.

[6] GitHub. — URL: `https://github.com` (online; accessed: 24.04.2017).

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

[8] Raghothaman Mukund, Wei Yi, Hamadi Youssef. SWIM: synthesizing what I mean: code search and idiomatic snippet synthesis // Proceedings of the 38th International Conference on Software Engineering / ACM. — 2016. — P. 357–367.

[9] Robillard Martin P, Deline Robert. A field study of API learning obstacles // Empirical Software Engineering. — 2011. — Vol. 16, no. 6. — P. 703–732.

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

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