DeepAPI#: Синтез цепочки вызовов API CLR/C# по текстовому запросу

Александр Чебыкин
Санкт-Петербургский
Государственный Университет
Email: a.e.chebykin@gmail.com

Михаил Кита
Санкт-Петербургский
Государственный Университет
Email: mihail.kita@gmail.com

Яков Кириленко
Санкт-Петербургский
Государственный Университет
Email: jake.kirilenko@gmail.com

Аннотация—Разработчики часто ищут способы реализовать стандартную функциональность с помощью библиотечных функций (например, создать кнопку в UI или получить данные из формата JSON). Обычно источником такой информации является Интернет. Альтернатива - различные статистические инструменты, которые, обучившись на большом объеме кода, по текстовому запросу могут предоставлять пользователю набор вызовов функций, решающих задачу.

Мы рассматриваем один из таких инструментов — DeepAPI. Этот современный алгоритм основывается на глубоком обучении, и, согласно его авторам, работает лучше аналогов. Мы пытаемся воспроизвести этот результат, взяв целевым языком не Java, как оригинальный DeepAPI, а C#. В данной статье мы рассказываем о возникающих проблемах сбора данных для обучения, сложностях в построения и обучения модели, а также обсуждаем возможные модификации алгоритма.

I. Введение

Последние несколько десятков лет предпринимаются попытки анализировать исходный код для разных целей: генерация имени метода по его телу [1], анализ качества кода [2], генерация кода по текстовому запросу [3], [4]. Последнее особенно интересно, поскольку позволяет разработчикам избежать долгого изучения программных библиотек в поисках нужной функциональности, что является значимой проблемой [5].

Обычно разработчики ищут способы реализовать стандартную функциональность с помощью поисковых движков в сети Интернет, для чего те не оптимизированы [6]. Исследователями предлагались разные замены им, в том числе инструменты, основывающиеся на статистическом анализе большой кодовой базы. Яркие представители:

- MAPO[4] генерирует по имени функции наиболее релевантные шаблоны её использования;
- UP-Miner [7] — улучшенная версия MAPO, достигающая лучших результатов в генерации благодаря более сложному алгоритму;
- SWIM[3] генерирует код по текстовому запросу.

Одним из самых свежих алгоритмов в этой области является DeepAPI [8] — алгоритм из статьи строит последовательность вызовов функций по текстовому запросу, достигая лучших результатов, чем SWIM, за счёт использования моделей глубокого машинного обучения из области обработки естественных языков. Глубокое обучение успешно используется для перевода между натуральными языками, DeepAPI пользуется результатами последних исследований в этой области. Задача генерации кода по тексту здесь рассматривается как задача перевода с английского языка на язык вызовов функций (слова в таком языке — функции языка программирования, предложения — упорядоченные наборы слов).

Результаты, представленные в статье об алгоритме DeepAPI, привлекли наше внимание, и мы решили повторить этот опыт и расширить его. В результате работы оригинального алгоритма DeepAPI получается линейная последовательность вызовов функций, код с их применением программисту приходится писать самостоятельно. Хотелось бы облегчить ему этот этап, предлагая не список функций, а сниппет кода — то есть отрывок кода на языке программирования, в котором нужно модифицировать минимальное количество деталей для того, чтобы он заработал. Для реализации этого может оказаться полезным алгоритм T2API [9], который так же является достаточно свежим исследованием возможности генерации кода по текстовому запросу. В первой части алгоритма по текстовому запросу выбираются некоторые релевантные элементы API, во второй — из них генерируются граф управления и код. Теоретически, замена первой части алгоритма на DeepAPI позволит сделать итоговые результаты более точными, так как модель, используемая на первом шаге T2API, более старая, а также не упорядочивает элементы API на выходе.

На текущий момент мы пытаемся повторить эксперимент из статьи DeepAPI и реализовать алгоритм из неё на базе другого языка программирования. В оригинальной статье авторы работают с элементами API и кодовой базой языка Java и упоминают, что одна из угроз действительности эксперимента — работа только с одним языком. Мы желаем нивелировать эту угрозу, а потому работаем с языком C#.

Наш текущий вклад в исследования можно описать следующими тезисами:

- собран набор тренировочных данных, реализован

инструментарий для сбора данных;
- проведено исследование использования в качестве тренировочных данных данные, отличные от рассмотренных в оригинальной статье;
- построена аппроксимация алгоритма DeepAPI;
- поставлен частично успешный первичный эксперимент.

II. DeepAPI

Модель DeepAPI основывается на современных техниках глубокого обучения и машинного перевода. Ключевая техника — обучение Sequence-to-Sequence [10], суть которой — по входной строке генерировать выходную. При этом обычно первая строка принадлежит одному естественному языку, вторая — другому. В случае DeepAPI исходный язык — английский, выходной — язык элементов API.

Техника Sequence-to-Sequence включает в себя две рекуррентные нейронные сети (рекуррентная нейронная сеть (RNN) — один из подвидов моделей глубокого обучения). Кодер — первая из этих сетей — поэлементно считывает вход и обновляет свои параметры в текущем скрытом состоянии. После окончания считывания последнее состояние — предполагается, что в нём заключена суть входного предложения — берётся в качестве вектора контекста, и передаётся в качестве входа второй RNN — декодеру. Декодер на каждом шаге, сверяясь с вектором контекста, генерирует очередной элемент выходной последовательности, и обновляет своё состояние. Работа декодера прекращается, когда он генерирует слово, означающее окончание строки — <EOS>.

Пример работы модели представлен на рисунке 2. Входной поток — “generate random number”, выходной — “Random.new Random.nextInt”.

На рисунке для простоты понимания состояния расширены по времени: поскольку нейронная сеть рекуррентная, переход из состояния должен вести в него же. То есть здесь h1, h2, h3 — одно и то же состояние в моменты времени 1, 2, 3.

В модели DeepAPI также используется улучшение техники Sequence-to-Sequence: механизм внимания. Вместо использования лишь последнего состояния кодера в качестве вектора контекста для генерации целого предложения, на каждом шаге работы декодера используется свой вектор контекста, получаемый как взвешенная сумма всех состояний кодера

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

где c_j — вектор контекста для шага j, h_t — исторические состояния кодера, a_{jt} — веса этих состояний для шага j (моделируются при помощи отдельной заранее натренированной нейронной сети).

Для увеличения точности работы алгоритма вводится еще одно улучшения — новая функция потерь, включающая в себя наказание за использование чересчур частых элементов API. Ведь если функция часто используется, то скорее всего, она не связана с реализацией конкретной функциональности, а потому считается шумом. Оценка важности элемента API основывается на tf–idf [11], и выглядит так:

$$w_{idf}(y_t) = log(N/n_{y_t})$$

где N — общее число экземпляров в тренировочном наборе, n_{y_t} — число экземпляров, в которых встречается y_t.

Так как DeepAPI должен переводить запросы на английском языке в цепочки вызовов функций, она должна тренироваться на примерах того, какие английские предложения каким цепочкам вызовов функций соответствуют. Такие примеры авторы оригинальной статьи извлекают из открытого исходного кода, а именно из методов с комментариями. Так как целевым языком программирования выбрана Java, имеющая зафиксированную структуру комментария для документации Javadoc, авторы пользуются этим, и берут первое предложение комментария в качестве описания метода на английском языке (по определению Javadoc, именно это написано в первом предложении). Чтобы получить список вызовов функций, код метода анализируется компилятором Eclipse JDT. Выводятся классы каждой переменной, вызовы методов записываются вместе с именем класса, которому этот метод принадлежит. Пример вычленения информации из кода представлен на рисунке 1.

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

Список вызовов API: InputStream.read -> OutputStream.write
Описание: copies bytes from a large inputstream to an outputstream

Рис. 1: Пример получения данных из кода

После обучения, во время генерации результата по запросу, авторами используется лучевой поиск — вместо того, чтобы генерировать самое вероятное слово на каждом шаге, поддерживается список из n текущих самых вероятных слов, и генерация продолжается для

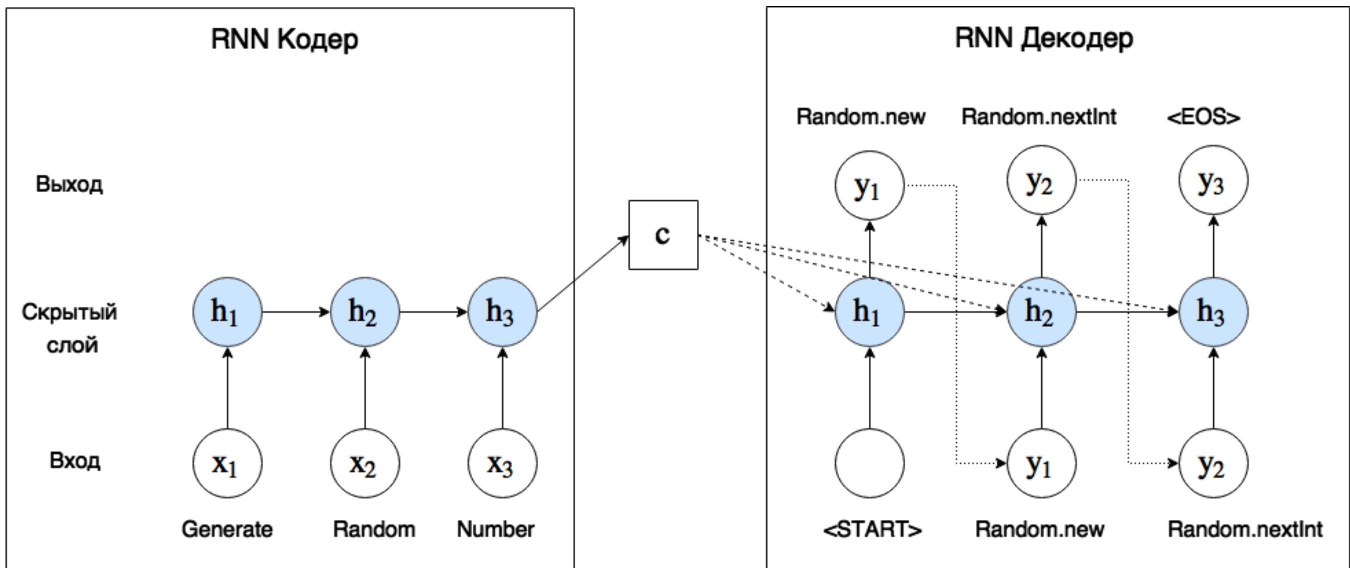

Рис. 2: Пример работы RNN Кодер-Декодера

каждого из них. Другие результаты отсекаются. Цель подобной практики — предотвратить случаи, когда наиболее точная цепочка отбрасывается из-за того, что её первое слово оказалось не самым подходящим. В итоге для каждого запроса получаем список из нескольких возможных ответов.

Для человеконезависимой оценки качества работы алгоритма используется метрика BLEU. BLEU часто применяется для оценки точности машинного перевода, она измеряет, насколько сгенерированная последовательность близка к той, что была написана человеком и использовалась во время обучения. Шкала оценок идёт от 0 до 100.

По представленным в статье оценкам, DeepAPI превосходит предыдущие техники. Поскольку не у всех из них была подсчитана оценка BLEU, авторы DeepAPI реализовали и оценили прошлые алгоритмы самостоятельно. В итоге UP-Miner получил оценку 11.97, SWIM — 19.90, DeepAPI — 54.42.

Имеется также рабочий прототип [12], относительно правдоподобно отвечающий на запросы.

III. Получение данных

Отдельное внимание стоит уделить процессу сбора данных для исследования и тренировки модели. Для получения большего количества данных, мы использовали не только скачивание кода открытых проектов с Github, но и обработку скомпилированных сборок проектов, полученных из репозитория NuGet. В следующих двух секциях мы рассматриваем эти способы подробнее.

A. NuGet

Первый подход заключался в извлечении нужных данных из скомпилированных библиотек, которые в большом количестве доступны в репозитории NuGet [13]. Для улучшения качества данных пакеты рассматривались в порядке убывания популярности: мы исходили из предположения, что наиболее популярные пакеты содержат более качественный код. Пакеты скачивались в автоматическом режиме, после чего содержащиеся в них библиотеки собирались в одну папку, чтобы обеспечить разрешение зависимостей. В случае возникновения коллизии имён предпочтение отдавалось более поздней версии библиотеки. Помимо самих библиотек также собирались и все доступные xml-файлы, содержащие комментарии к методам.

Далее следовал процесс обработки собранных файлов: для каждого извлечённого метода генерировалось синтаксическое дерево, а затем производился его разбор. Описанные действия позволили получить исчерпывающую информацию о каждом вызове метода в коде, включая тип объекта, имя метода и параметры. Эта информация использовалась для построения последовательности вызовов, которая затем сохранялась на диск для дальнейшего использования.

Помимо последовательностей вызовов необходимо было также собрать комментарии к методам. На этом этапе мы столкнулись с проблемой: количество методов с комментариями в скачанных нами пакетах составляет лишь 11.21% от числа всех методов. Были собраны все доступные комментарии из xml-файлов, а для оставшихся методов мы генерировали описания, используя имя данного метода и имя содержащего его класса.

Как показала практика, описанный метод сбора данных оказался достаточно эффективным. Было скачано 4,100 пакетов, содержащих в общей сложности 4,278 библиотек. Итоговое количество собранных методов составило 611,945, из них 68,585 — с комментариями.

B. Github

GitHub [14] — популярный хостинг проектов с открытым исходным кодом. Именно отсюда авторы оригинальной статьи скачали 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 у нас 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 48,789 репозиториев мы получаем 870,008 пар.

Помимо количества репозиториев, возникает проблема 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 мы можем обработать. Кроме этого, мы сталкиваемся и с другими сложностями со стороны языка, например, динамический тип данных dynamic, о котором во время компиляции мы ничего не знаем. Но такие проблемы по сравнению с проблемой неявных типов несущественны: мы провели небольшой эксперимент, запустив поиск в исходном коде 470 случайно выбранных репозиториев слова "dynamic"и "var и получив всего 4,856 результатов для первого, и 176,561 результатов для второго. Поэтому пока что мы не исследуем способы решения подобных неприоритетных проблем, оставляя это на будущее.

В качестве компилятора мы используем 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.

Из 100 самых популярных результатов поиска таким ограничениям удовлетворяет 51 проект, в среднем по всем результатам поиска — 11.2%.

Код из подходящих репозиториев мы обрабатываем аналогично оригинальной статье.

IV. Эксперимент

На текущий момент мы не занимались реализацией упомянутой ранее модификации функции потерь, наказывающую генерацию слишком частых функций. Это изменение не было приоритетным, т.к. влияет на оценку BLEU не больше, чем на 2 очка, увеличивая её с 52.49 до 54.42, а для нас более интересно получить первичный результат примерно того же порядка, что и 52.49.

A. Первый эксперимент

В качестве функции потерь в модели DeepAPI берётся условная логарифмическая функция правдоподобия. Модель тренируется максимизировать её с помощью стохастического градиентного спуска [15] в комбинации с оптимизатором AdaDelta [16]. Авторы реализуют модель на базе библиотеки GroundHog. Однако он устарел и более не поддерживается, поэтому мы реализуем нашу первую модель на основе TensorFlow — популярной библиотеки для машинного обучения с открытым кодом.

Надо упомянуть, что обучение рекуррентных нейронных сетей очень ресурсоёмко, а потому ведётся на видеокартах. В то время как авторы для тренировки пользовались видеокартой Nvidia K20, у нас имеется гораздо менее мощная Nvidia GTX 660. В связи с этим нам пришлось уменьшить параметры тренировки до менее оптимальных. Например, авторы обнаружили, что оптимальное количество нейронов в скрытом слое кодера и декодера — 1000. Мы выставляем их количество в 700. Авторы не обсуждают влияние размера пакета (batch) на модель, выставляя его значение в 200. Мы делаем его размер равным 32.

Мы вводим дополнительное улучшение в виде bucketing — разделения исходных предложений на классы по длине, и тренировки внутри этих классов. Дело в том, что для быстрого обучения нам нужны вектора фиксированной длины — а предложения имеют длину разную. Можно их все дополнять до максимальной длины (с помощью специального слова), а можно разделить на категории вида длина < 10, длина < 20, и т.д., и дополнять до максимума в этой категории, тренируя категории отдельно. Чем меньше дополняем пустыми словами, тем меньше шума, и лучше (теоретически) результат.

Мы тренируем модель на 605,146 парах данных в течение 40,000 итераций, ограничив словарь самыми популярными 10,000 слов в исходном и выходном языках.

Итоговая модель умеет верно отвечать на некоторые запросы, например 'generate random number — System.Random.new System.Random.Next' или 'replace part of string with other string — System.String.Replace', но на большинстве входов выдаёт нерелевантный результат.

B. Второй эксперимент

Так как хороших результатов получить на модели с урезанными параметрами не удалось, мы создаём

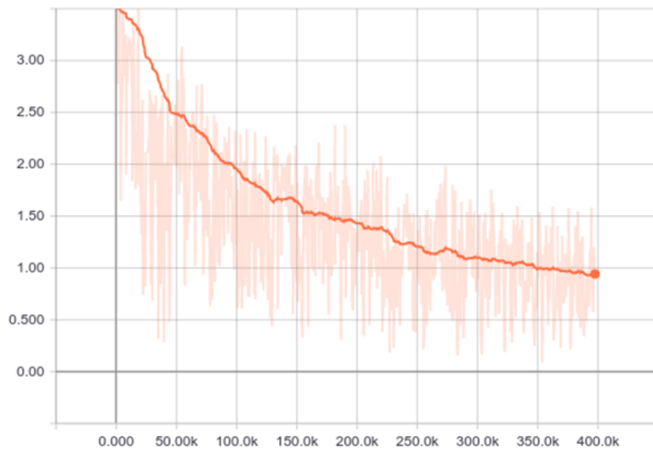

Рис. 3: График изменения функции потерь во время обучения

модель на основе нового фрэймворка от Google - tf-seq2seq, специализирующегося на Sequence-to-Sequence моделях. С его помощью мы обучаем модель на тех же значениях параметров, что и в оригинальной статье, при этом для этого хватает производительности нашей не мощной видеокарты. Также во время декодирования мы используем лучевой поиск - улучшение, предложенное авторами оригинальной статьи и описанное в секции II; в предыдущей модели мы этот механизм не реализовывали.

Модель мы тренируем на 924,593 парах данных в течение 390,000 итераций, ограничивая словарь в исходном и выходном языках самыми популярными 10,000 слов.

Итоговая модель может не только отвечать на одиночные запросы, как старая, но и показывает ненулевые результаты на тестовом множестве из 14,000 пар данных. Надо упомянуть, что пары из тестового множества не используются во время тренировки, поэтому положительные результаты на них означают, 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 уменьшается (см. рисунок 3).

Таким образом, из общей положительной динамики мы можем заключить, что техника имеет право на существование. Мы считаем, что тренировка на большем количестве данных позволит получить сравнимый с оригинальным результат.

V. Дальнейшая работа

Мы планируем продолжать исследование данного подхода. В ближайшей перспективе перед нами стоят следующие цели:

1) собрать набор данных, сопоставимый по размеру с набором данных в оригинальной статье;
2) поэкспериментировать с параметрами обучения, и найти оптимальные;
3) исследовать возможность тренировки на не комментированных методах, используя имена и классы параметров, имя метода;
4) найти способы решения специфичных для целевого языка проблем;
5) реализовать алгоритм T2API;
6) объединить алгоритмы DeepAPI и T2API в один, проанализировать получившийся результат.

VI. Заключение

В этой статье мы описали нашу частично успешную попытку повторить результат статьи DeepAPI. Используя современные методы глубокого обучения и машинного перевода, мы достигли небольших успехов в задаче генерации списка вызовов функций по текстовому запросу.

Мы считаем, что модель DeepAPI действительно может быть реализована на базе другого языка, а также улучшена, что мы и собираемся продемонстрировать по результатам дальнейшей работы.

Список литературы

[1] M. Allamanis, H. Peng, and C. Sutton, “A convolutional attention network for extreme summarization of source code.”
[2] V. Barstad, M. Goodwin, and T. Gjøsæter, “Predicting source code quality with static analysis and machine learning.” in NIK, 2014.
[3] M. Raghothaman, Y. Wei, and Y. Hamadi, “Swim: synthesizing what i mean: code search and idiomatic snippet synthesis,” in Proceedings of the 38th International Conference on Software Engineering. ACM, 2016, pp. 357–367.
[4] T. Xie and J. Pei, “Mapo: Mining api usages from open source repositories,” in Proceedings of the 2006 international workshop on Mining software repositories. ACM, 2006, pp. 54–57.
[5] M. P. Robillard and R. Deline, “A field study of api learning obstacles,” Empirical Software Engineering, vol. 16, no. 6, pp. 703–732, 2011.
[6] J. Stylos and B. A. Myers, “Mica: A web-search tool for finding api components and examples,” in Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on. IEEE, 2006, pp. 195–202.
[7] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, “Mining succinct and high-coverage api usage patterns from source code,” in Proceedings of the 10th Working Conference on Mining Software Repositories. IEEE Press, 2013, pp. 319–328.
[8] X. Gu, H. Zhang, D. Zhang, and S. Kim, “Deep api learning,” in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2016, pp. 631–642.
[9] T. Nguyen, P. C. Rigby, A. T. Nguyen, M. Karanfil, and T. N. Nguyen, “T2api: synthesizing api code usage templates from english texts with statistical translation,” in Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, 2016, pp. 1013–1017.
[10] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to sequence learning with neural networks,” in Advances in neural information processing systems, 2014, pp. 3104–3112.
[11] G. Salton and M. J. McGill, “Introduction to modern information retrieval,” 1986.

[12] H. K. U. of Science and Technology. (2017) Deepapi. [Online]. Available: http://www.cse.ust.hk/~xguaa/deepapi/
[13] Nuget gallery. [Online]. Available: http://www.nuget.org
[14] Github. [Online]. Available: https://github.com
[15] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in Proceedings of COMPSTAT'2010. Springer, 2010, pp. 177–186.
[16] M. D. Zeiler, "Adadelta: an adaptive learning rate method," arXiv preprint arXiv:1212.5701, 2012.

DeepAPI#: CLR/C# call sequence synthesis from text query

Alexander Chebykin, Mikhail Kita, Iakov Kirilenko

Developers often search for an implementation of typical features via libraries (for example, how to create a UI button control, extract data from a JSON-formatted file, etc.). The Internet is the usual source of the information on the topic. However, various statistical tools provide an alternative: after processing large amounts of source code and learning common patterns, they can convert a user request to a set of relevant function calls.

We examine one of those tools — DeepAPI. This fresh deep learning based algorithm outperforms all others (according to its authors). We attempt to reproduce this result using different target programming language — C# — instead of Java used in the original DeepAPI. In this paper we report arising problems in the data gathering for training, difficulties in the model construction and training, and finally discuss possible modifications of the algorithm.