Санкт-Петербургский государственный университет

Программная инженерия

Соколов Ярослав Сергеевич

Система автодополнения фрагментами кода

Бакалаврская работа

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

Рецензент:
Специалист по машинному обучению
ООО «ИнтеллиДжей Лабс»
Е. Г. Булычев

Санкт-Петербург
2020

SAINT PETERSBURG STATE UNIVERSITY

Software engineering

Iaroslav Sokolov

Full line code completion system

Bachelor's Thesis

Scientific supervisor:
Associate professor, Ph.D. Timofey Bryksin

Reviewer:
Machine learning engineer
«IntelliJ Labs» Co. Ltd
Egor Bulychev

Saint Petersburg
2020

Оглавление

Введение **4**

1. Обзор **6**

1.1. Системы автодополнения следующего токена 6
1.2. Варианты токенизации кода 7
1.3. Алгоритмы автодополнения следующего токена 9
1.4. Представления кода 14
1.5. Системы автодополнения фрагментами кода 16
1.6. Выводы из обзора 20
1.7. Архитектура плагина 23

2. Реализация алгоритма генерации фрагментов кода **25**

2.1. Подготовка данных для обучения 25
2.2. Обучение нейронных сетей 26
2.3. Реализация лучевого поиска 28

3. Оптимизация компонентов алгоритма **31**

3.1. Модификация парного кодирования байтов 31
3.2. Токенизация неполного контекста 32

4. Апробация реализованной системы **35**

4.1. Данные 35
4.2. Метрика качества 35
4.3. Сравнение с аналогами 36
4.4. Влияние модификаций токенизатора 37

Заключение **40**

Список литературы **42**

Приложение А. Акт о внедрении **48**

Введение

Интегрированные среды разработки (Integrated Development Environment, IDE) сегодня играют важную роль в разработке многих программных продуктов. Они предоставляют разработчику множество инструментов: от удобного представления изменений в коде до предсказания имён методов. Одним же из самых популярных инструментов является функция автодополнения кода [24, 48], которая предлагает возможные варианты текущего недописанного или следующего токена, не только сокращая временные затраты на его написание, но и предотвращая неправильное использование интерфейсов систем [11].

Существует множество научных работ [4, 5, 6, 14, 17, 20, 25, 27, 28, 31], которые ставят перед собой цель улучшить автодополнения кода. В своих исследования авторы часто применяют методы из смежной, быстроразвивающейся области обработки естественных языков. При этом в области обработки естественных языков исследователи добились качественной генерации длинных текстов [15, 18], в то время как все исследования задачи автодополнения кода сфокусированы на улучшении автодополнения лишь одного токена кода.

По этой причине расширение систем автодополнения до генерации целых фрагментов кода представляет особый интерес. Это может помочь разработчикам быстрее писать типовые части программы или может быть полезным, чтобы увидеть вызов некоторого метода с уже подставленными переменными и лучше понять логику его работы.

Таким образом, в рамках данной работы предлагается разработать систему автодополнения фрагментами кода, оптимизировать её для работы в реальных условиях и исследовать её влияние на скорость написания кода.

В качестве первого языка программирования для апробации системы был выбран Python, так как он достаточно популярен и имеет большую кодовую базу, которая может пригодиться для обучения статистических подходов. В случае получения хороших результатов планируется поддержка и других языков программирования.

Постановка задачи

Целью работы является создание системы автодополнения фрагментами кода в виде плагина для PyCharm, IDE для языка программирования Python. Над данным проектом работает команда из нескольких человек. В рамках данной работы были поставлены следующие задачи:

1. провести обзор предметной области;
2. выбрать и реализовать алгоритм генерации фрагментов кода;
3. оптимизировать компоненты алгоритма для его использования на практике;
4. провести апробацию полученного плагина.

1. Обзор

В обзоре будут рассмотрены системы автодополнения следующего токена, существующие системы автодополнения фрагментами кода и то, как такие системы могут быть устроены.

1.1. Системы автодополнения следующего токена

Все существующие сегодня системы автодополнения следующего токена кода можно представить в виде схемы, изображённой на рисунке 1. Они принимают на вход уже введённый программистом код (контекст) в виде текста или абстрактного синтаксического дерева (Abstract Syntax Tree, AST). Затем контекст разбивается на набор токенов с помощью токенизатора, по которому алгоритм автодополнения предлагает следующий токен.

В этой схеме можно выделить три компоненты, которые варьируются в различных работах, посвящённых улучшению автодополнения кода: способ представления кода, способ токенизации кода и алгоритм автодополнения. В данном разделе будет рассмотрено, как именно устроены эти компоненты в существующих на сегодня работах, посвящённых автодополнению.

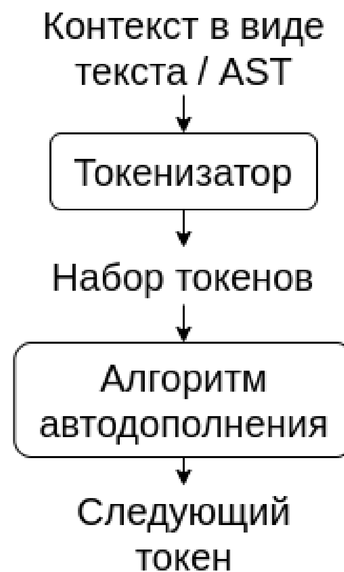


Рис. 1: Схема системы автодополнения следующего токена.

1.2. Варианты токенизации кода

Токенизатор — часть любой системы автодополнения, как показано на рисунке 1. При использовании любого алгоритма автодополнения код необходимо в первую очередь представить в виде чисел. Общепринятой практикой является разбитие кода на части, которые могут являться лексемами языка программирования, вершинами AST, символами и тому подобное. Такие части кода называют *токенами*, а процесс разбиения кода, да и любого другого текста, на токены — *токенизацией*. Затем каждому уникальному токену присваивается свой уникальный числовой идентификатор. После этого код или текст можно представить в виде упорядоченного набора чисел, разбив его на токены и заменив каждую часть на соответствующий ей идентификатор. Множество всех различных токенов называют *словарём*.

Системы с закрытым словарём

Одним из способов токенизации кода является его разбиение на лексемы с помощью синтаксического анализатора. Этот подход широко используется ввиду его простоты и того факта, что в области обработки естественных языков подобный подход показывает себя хорошо [4, 5, 20, 25, 27, 31].

Ключевой проблемой такого подхода выступает размер словаря. Он зачастую оказывается огромным попросту из-за того, что существует множество различных имён переменных, функций, классов и так далее. При этом некоторые из токенов могут встречаться в наборе данных для обучения настолько редко, что статистический алгоритм не способен выучить закономерности, связанные с этими токенами. Также, чем больше размер словаря, тем больше обучаемых параметров у алгоритма, что плохо сказывается на его времени обучения, времени работы и размерах. Поэтому при работе с таким способом токенизации часто прибегают к замене всех редких токенов на один специальный токен, что позволяет сократить размер словаря [4,5,20,27,31]. В другой работе авторы унифицируют все идентификаторы в коде таким образом, что-

бы при работе в реальных условиях их можно было обратно заменить на реальные названия переменных, классов и других объектов [25].

Системы с подобным способом токенизации называют *системами с закрытым словарём*, поскольку они способны сгенерировать только те токены, которые попали в словарь. Остальные же токены называют *неологизмами*, которыми могут являться как редкие идентификаторы, заменённые на этапе сокращения словаря, так и слова, которых не было в тренировочных данных вовсе.

В работе [14] авторы выяснили, что в языках программирования неологизмов гораздо больше, чем в естественных языках. Из-за этого при использовании такого подхода со словарём с большим размером в 74 046 токенов удаётся покрыть всего 85% кода в тренировочном множестве и 80% кода в данных, которых модель ещё не видела. Остальные токены являются неологизмами, и информацию о них модель не может использовать. Поэтому использование такого подхода при работе с кодом может быть менее целесообразно, чем при работе с естественными языками.

Системы с открытым словарём

Системы с открытым словарём способны генерировать любой код или текст. Простейшим примером системы такого типа является система, разбивающая текст на символы. Так как её словарь состоит из символов, она может с их помощью сгенерировать любой текст. Также преимуществом является малый размер словаря. К недостатку можно отнести то, что для генерации одного и того же фрагмента текста модели с таким методом токенизации нужно сгенерировать гораздо больше токенов, чем в случае токенизации на уровне лексем, что сильно снижает скорость работы такой системы.

Чтобы избежать долгой генерации следующего токена кода, авторы работы [14] использовали вместо символов группы символов, n-граммы. Это позволяет генерировать следующий токен в несколько раз быстрее, чем при посимвольной генерации. Как оказалось, такая система превосходит в точности предсказания системы с закрытым словарём [14, 43].

Другим решением обозначенной проблемы выступает парное кодирование байтов (byte pair encoding) [32]. Авторы предложили следующий способ построения словаря: сначала в качестве словаря берутся все символы, встречающиеся в тексте; затем находится самая частая пара токенов из текущего словаря и добавляется в качестве нового токена в словарь; предыдущая операция повторяется n раз. Таким образом, наиболее популярные слова оказываются отдельными токенами в словаре, а непопулярные разбиваются на подслова.

В работе [17] авторы испытали парное кодирование байтов в задаче автодополнения кода. Оказалось, что такой подход показывает лучшее качество генерации не только по сравнению с системами с закрытым словарём, но и по сравнению с системой, работающей на n-граммах.

1.3. Алгоритмы автодополнения следующего токена

Основной компонентной систем автодополнения является алгоритм автодополнения следующего токена. Для построения таких алгоритмов всё чаще используют машинное обучение [14, 27] и нейронные сети [4,6,17,20,23,25,28,31,37]. На текущий момент именно такие подходы показывают наилучшие результаты [43], поэтому далее будут рассмотрены только они.

При построении таких подходов задачу автодополнения кода формализуют в следующий вид: для всех подходящих вариантов следующего токена необходимо вычислить его вероятность при заданном контексте

$$P(next_token|context)\ \forall next_token \in suitable_tokens, \tag{1}$$

где $context$ — контекст в произвольном виде, $suitable_tokens$ — множество подходящих токенов, а $next_token$ — следующий токен, который необходимо подсказать программисту.

Ранжирующие алгоритмы автодополнения кода

Попытки применить машинное обучение в задаче автодополнения кода постепенно усложнялись параллельно с развитием этой области. В одном из первых подходов, использующих машинное обучение [28], подходящие токены, для которых необходимо вычислить вероятность (1), извлекаются из программы путём её статического анализа. В ходе такого анализа могут быть найдены различные поля, методы или атрибуты классов, аргументы функции, объявленные переменные, доступные модули библиотек и так далее. В одной из недавних работ [10] показано, что при такой постановке задачи можно добиться высокого качества при очень высокой скорости работы системы.

Данный подход имеет свои достоинства и недостатки. С одной стороны, такой выбор подходящих токенов означает, что система автодополнения способна предложить только синтаксически корректный код. А с другой — её возможности сильно ограничены возможностями статического анализа кода. Например, такие системы никогда не смогут предложить имя нового метода или значение строковой переменной.

Генеративные алгоритмы автодополнения кода

Затем развитие получили модели, которые способны генерировать варианты автодополнения без помощи статического анализа кода. Это достигается за счёт построения набора всевозможных токенов (словаря) на заранее выбранном тренировочном наборе данных. Затем словарь используется в качестве подходящих токенов для вычисления вероятности следующего токена (1).

Достоинства и недостатки прямо противоположны достоинствам и недостаткам ранжирующих алгоритмов. С одной стороны, теперь ничем не ограничен и может генерировать как имена новых функций, так и комментарии в коде. С другой — он может сгенерировать синтаксически некорректный код.

Нейронные сети в алгоритмах автодополнения кода

На данный момент подходы, использующие нейронные сети как для ранжирования, так и для генерации следующего токена, показывают наилучшие результаты при прочих равных [43]. При этом используются различные архитектуры нейронных сетей. Далее будут рассмотрены две наиболее популярные архитектуры: рекуррентная и Трансформер (Transformer).

Для описания этих архитектур необходимо немного уточнить вид контекста для вычислении вероятности (1). На практике используют два вида контекста: текст и абстрактное синтаксическое дерево. При этом в обоих случаях контекст представляют в виде последовательности токенов, будь то токены языка или узлы синтаксического дерева в порядке обхода в глубину. Различия же появляются в модификации некоторых блоков нейросетей, которые не влияют на их глобальную архитектуру.

Теперь, когда известно, что контекст представляется в виде последовательности токенов, можно уточнить формулировку задачи (1): для каждого возможного следующего токена x_{i+1} необходимо вычислить его вероятность при заданном контексте $x_i...x_1$:

$$P(x_{i+1}|x_i, ..., x_1) \ \forall x_{i+1} \in suitable_tokens, \quad (2)$$

где $x_i, ..., x_1$ — последовательность токенов, образующих контекст, а x_{i+1} — следующий токен.

Рекуррентные нейронные сети

Для подсчёта вероятности (2) была предложена архитектура рекуррентной нейронной сети, которая изображена на рисунке 2. Нейронная сеть в ходе обучения должна настроить параметры W для параметризованной функции $(h_i, y_i) = f_W(x_i, h_{i-1})$ таким образом, чтобы ошибка в предсказании вероятности (2) была минимальной. Такая нейросеть теоретически способна обрабатывать последовательности любой длины, сохраняя необходимую информацию о входной последовательности в

скрытых состояниях: при вычислении вероятности $P(x_{i+1}|x_i, ..., x_1)$ вся необходимая информация о последовательности $x_i, ..., x_1$ будет содержаться в скрытом состоянии h_i.

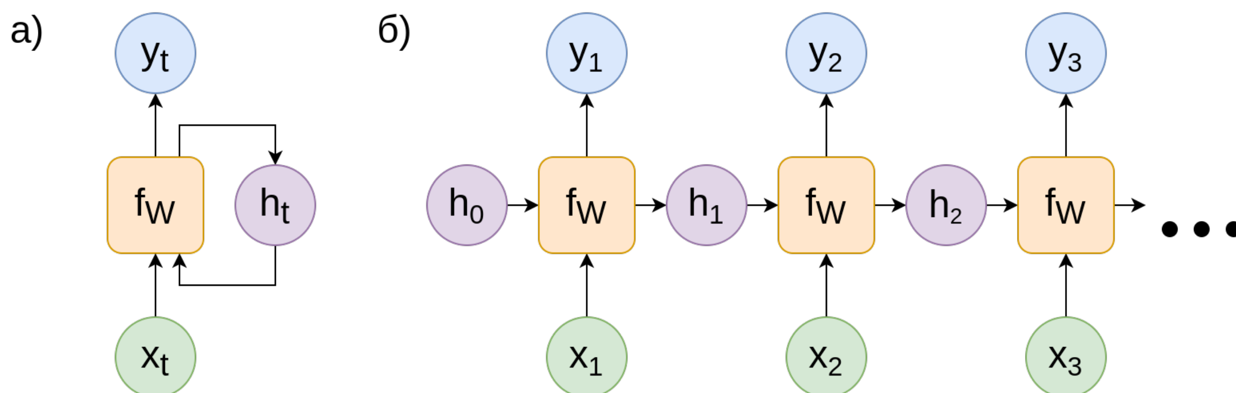


Рис. 2: Архитектура однослойной рекуррентной нейронной сети (а) в общем виде и (б) её развернутое в шагах обработки последовательности представление. Здесь f_W — параметризованная функция, параметры которой настраиваются в ходе обучения, $\{x\}_{t=1}^n$ — входная последовательность, $\{y\}_{t=1}^n$ — выходная последовательность, $\{h\}_{t=0}^n$ — скрытые состояния нейросети на каждом шаге.

Рекуррентная архитектура нейронных сетей довольна популярна в области автодополнения кода. Её используют как при ранжировании вариантов следующего токена [10, 28], так и при его генерации [4, 6, 17, 20, 25, 28]. При этом пытаются применить различные модификации как самой архитектуры нейронной сети [6], так и представлений кода [4, 25], которые она использует.

Одним из недостатков рекуррентной архитектуры нейросети является последовательная обработка данных как во время генерации, так и во время обучения [21]. Этот недостаток невозможно исправить модификациями рекуррентной архитектуры, так как он вызван рекуррентной зависимостью, которая в неё заложена. Действительно, чтобы посчитать вероятность $P(x_{i+1}|x_i, ..., x_1)$, необходимо вычислить $f_W(x_i, h_{i-1})$, где h_{i-1} является результатом вычисления $f_W(x_{i-1}, h_{i-2})$. Таким образом, функции f_W невозможно вычислять параллельно на всех элементах входной последовательности. На практике это приводит к тому, что при обучении аппаратные ресурсы используются лишь

частично, что плохо сказывается на времени обучения, затрудняя использование большего количества данных и параметров W.

Архитектура нейронной сети Трансформер

В качестве решения описанной проблемы рекуррентной архитектуры нейронных сетей авторы работы [1] предложили новую архитектуру — Трансформер, — с помощью которой возможно обрабатывать всю входную последовательность параллельно.

Архитектура Трансформер изображена на рисунке 3. Отличие от рекуррентной архитектуры заключается в том, что функция f_W принимает на вход сразу всю входную последовательность. В результате рекуррентная зависимость отсутствует, так что вычисление функции f_W можно производить для всех $\{x\}_{i=1}^{n}$ параллельно, что позволяет гораздо эффективнее использовать аппаратные ресурсы.

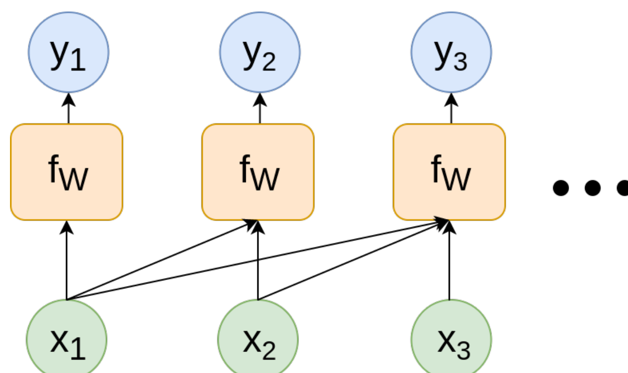


Рис. 3: Архитектура Трансформер с одним слоем. Здесь f_W — параметризованная функция, которая настраивается в ходе обучения, $\{x\}_{t=1}^{n}$ — входная последовательность, $\{y\}_{t=1}^{n}$ — выходная последовательность.

В задаче моделирования вероятности (2) Трансформер обходит рекуррентные аналоги [15], причем в работе [18] показано, что при увеличении количества параметров модели Трансформер и размера тренировочных данных качество можно значительно улучшить, чего нельзя сказать о рекуррентных нейросетях.

Сегодня Трансформер является стандартной архитектурой практически во всех задачах, связанных с обработкой естественного языка,

в связи с чем она активно развивается. На данный момент модификацией, показывающей на данный момент лучшее качество в моделировании языка, является Трансформер с увеличенной длиной контекста (Transformer extra long, Transformer-XL) [40]. Данная модификация способна эффективно использовать в 5.5 раз больше контекста, нежели обычный Трансформер. Эта особенность может быть важна не только для естественных языков, но и для языков программирования.

За 2019 год появился ряд работ, использующих архитектуру Трансформер в задаче автодополнения кода [5,10,16,31,37]. При этом в работах [4,5,31] исследователи показали, что автодополнение одного токена можно улучшить, заменив архитектуру нейронной сети с рекуррентной на Трансформер.

1.4. Представления кода

Другой компонентой системы автодополнения, схема которой изображена на рисунке 1, является представление контекста. Контекст, то есть уже написанный код, можно представлять по-разному: с ним можно работать как с текстом, абстрактным синтаксическим деревом, графом потоков управления, данных и так далее. Однако во всех работах, посвящённых автодополнению кода, его представляют только в виде текста или абстрактных синтаксических деревьев. Другие представления не рассматриваются в литературе из-за того, что на каждом шаге генерации для каждой гипотезы необходимо было бы перестраивать представление с учётом этих гипотез. Это может занимать большое количество времени попросту из-за того, что таких построений нужно производить очень много.

Автодополнение кода на уровне текста

Такие системы рассматривают код как обычный текст, не зная ничего ни о его структуре, ни о типах, которые в нём используются. Код токенизируют и обучают модели на задаче предсказания следующего токена по предыдущим (2). Это довольно популярный подход в области

автодополнения кода [6, 17, 20, 23, 37], так как исследователи свободны почти без модификаций использовать последние достижения в смежной, быстроразвивающейся области — обработке естественных языков.

Автодополнение кода на уровне абстрактных синтаксических деревьев

Данные подходы используют информацию о структуре кода, представленную в виде абстрактного синтаксического дерева. Кроме самой структуры, AST также содержит информацию о типах языка программирования.

Задача автодополнения следующего токена изначально ставится так, что в качестве контекста доступен лишь тот код, который находится до предсказываемого токена, как сформулировано в задаче (2). При этом в тексте программы токены появляются в том же порядке, что и в обходе AST в глубину. Поэтому AST чаще всего представляют в виде последовательности вершин в порядке обхода в глубину и решают задачу предсказывания следующей вершины в этом порядке [4, 25, 31].

Простая смена последовательности токенов из текста программы на узлы в AST уже даёт преимущество, так как в AST также содержится информация о типах. При этом алгоритмы предсказания следующего токена могут быть дополнительно модифицированы, чтобы учитывать древовидную структуру AST, что также даёт существенный прирост в качестве [5].

На данный момент единый подход к использованию информации о структуре AST ещё не устоялся. В некоторых работах [5] авторы добавляют специальные параметры в нейронную сеть, которые позволяют ей выучивать то, на каких предков узла нужно обращать внимание, в то время как в области обработки естественных языков существуют более сложные подходы для учёта древовидной структуры [13, 34, 41, 42].

1.5. Системы автодополнения фрагментами кода

Существующие исследования в области автодополнения кода фокусируются на подсказке программисту одного токена, в то время как задача автодополнения фрагментами кода оставалась неисследованной до совсем недавнего времени.

В 2019 году начали появляться коммерческие системы автодополнения фрагментами кода. Осенью 2019-го года компания Microsoft заявила о поддержке автодополнения фрагментами кода [23]. Статья [16], в которой описывается эта система, вышла только в середине мая 2020 года. К тому же, исходного кода так и не было опубликовано.

Чуть больший интерес представляет инструмент Deep TabNine [37], в который поддержка автодополнения фрагментами кода была добавлена летом 2019-го года. Позже этот инструмент был куплен компанией Codota[1], что подтверждает заинтересованность сообщества в такого рода автодополнении. Deep TabNine доступен в качестве плагина для большого количества IDE, в том числе и для PyCharm. Однако, как и в предыдущем случае, исходного кода или исчерпывающего описания работы в открытом доступе нет.

Судя по доступной информации, обе системы берут за основу языковую модель для естественных языков GPT-2 [18]. Эта система токенизирует текст с помощью парного кодирования байтов, а затем, используя нейронную сеть с архитектурой Трансформер, оценивает вероятности (2). При адаптации этой системы к задаче автодополнения кода, судя по всему, её существенно не изменяли и представляли код в виде текста.

Таким образом, аналоги хоть и существуют, но, ввиду отсутствия подробного описания работы или исходного кода, работать над их улучшением невозможно. В данной же работе предлагается разработка системы автодополнения фрагментами кода с открытым исходным кодом.

[1] Codota приобретает TabNine, URL: https://www.crunchbase.com/acquisition/codota-acquires-tabnine--d166cc3b/

Генерация текстов на естественном языке

В области обработки естественных языков существуют подходы, которые способны качественно генерировать длинные тексты [15,18]. Они основаны на алгоритмах, которые генерируют один токен. Как показано на рисунке 4, алгоритм по входному контексту в виде набора токенов генерирует один следующий токен, который становится частью входного контекста на следующем шаге для генерации очередного токена.

Опыт исследований в области генерации кода по описанию на естественном языке показывает, что та же техника применима и к исходному коду [2,3,19,35,44,46]. Также нет оснований полагать, что в коммерческих системах автодополнения фрагментами кода [23, 37] используется другая техника. Поэтому для решения задачи генерации фрагментов кода можно адаптировать системы генерации одного токена кода с помощью такой схемы генерации последовательности токенов (см. рисунок 4).

Рис. 4: Генерация текста с помощью алгоритма генерации одного токена.

Системы автодополнения следующего токена и языковые модели

для естественных языков решают задачу оценки вероятности следующего слова или токена по контексту предыдущих слов или токенов, как сформулировано в задаче (2). Для генерации длинных текстов необходимо оценить вероятность целой последовательности токенов:

$$P(x_{i+n}...x_{i+1}|x_i...x_1)\ \forall x_{i+n}...x_{i+1} : x_{i+n}, ..., x_{i+1} \in suitable_tokens, \qquad (3)$$

где $x_i...x_1$ — последовательность токенов, образующих контекст, а $x_{i+n}...x_{i+1}$ — последовательность предсказываемых токенов.

Однако если решать эту задачу полным перебором и честно считать для каждой последовательности токенов длины n её вероятность (3), то алгоритм будет работать за время $O(V^n)$, где V — размер словаря, а n — длина последовательности. На практике V почти всегда превосходит 10 000, что делает такое решение неприемлемо долгим.

Лучевой поиск

На самом деле для генерации текста или фрагмента кода необязательно искать вероятность для каждой цепочки токенов из выражения (3), а необходимо найти такую цепочку, которая имеет максимальную вероятность. Для приближенного решения этой задачи был предложен алгоритм лучевого поиска (beam search) [36]. На данный момент алгоритм является самым качественным из тех, что на практике используются для решения этой задачи [8].

Алгоритм лучевого поиска является жадным и не гарантирует нахождения наиболее вероятной последовательности. Однако, количество просматриваемых алгоритмом вариантов можно настраивать параметром B, который называют шириной поиска или луча. Алгоритм имеет сложность $\mathcal{O}(VBn)$. Таким образом, варьируя значение параметра B, можно добиться хорошего соотношения между качеством генерируемых последовательностей и приемлемым временем работы.

Работа алгоритма изображена на рисунке 5. На каждом шаге поддерживается B цепочек токенов, называемых *гипотезами*, с максимальной вероятностью. Для перехода на следующий шаг алгоритм для

каждой гипотезы оценивает варианты её продолжения следующим токеном:

$$P(x_i|hyp)\ \forall hyp \in cur_hyps, \forall x_i \in suitable_tokens, \quad (4)$$

и отбирает B новых самых вероятных цепочек, где cur_hyps — множество текущих гипотез.

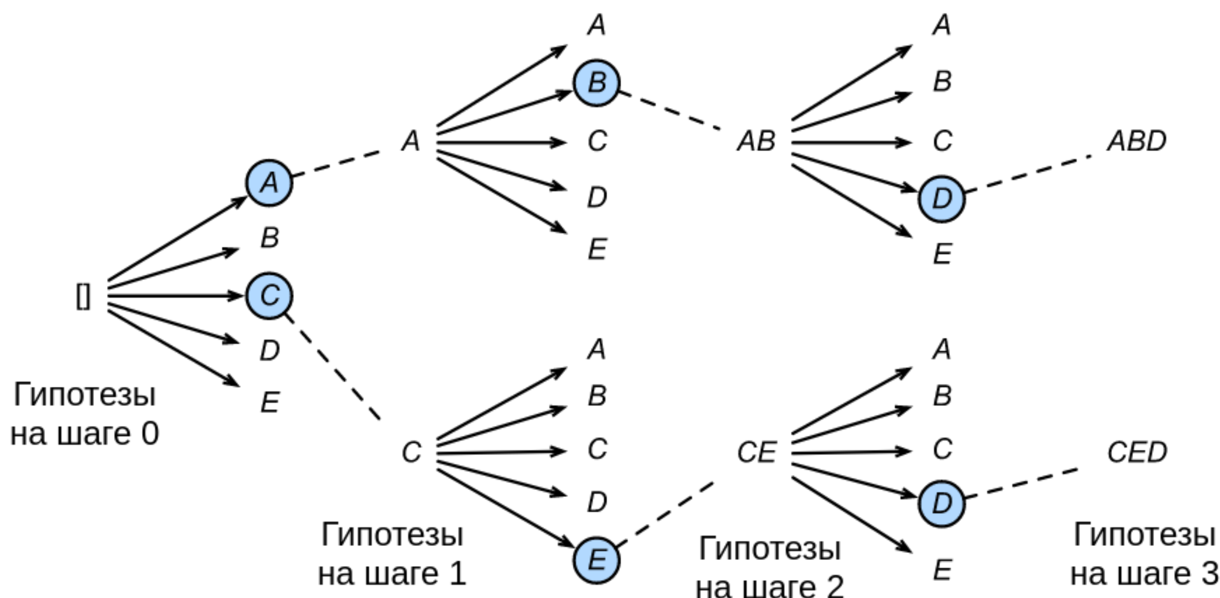


Рис. 5: Работа алгоритма лучевого поиска с параметром $B = 2$ и словарём $\{A, B, C, D, E\}$.[3]

Генерация разнообразных последовательностей

В задаче автодополнения кода важно предлагать несколько различных вариантов, так как модель может ошибаться и предсказывать наибольшую вероятность неверным токенам. Если же модель способна генерировать несколько наиболее вероятных гипотез, то вероятность, что хотя бы одна из них верна, увеличивается. Однако, при генерации нескольких наиболее вероятных цепочек токенов с помощью алгоритма лучевого поиска, эти цепочки получаются очень похожими друга на

[3]Из материалов курса «Dive into Deep Learning»: `https://d2l.ai/chapter_recurrent-modern/beam-search.html`

друга, что приводит к тому, что зачастую они либо все неверны, либо все похожи на верную и можно выбрать любую из них [8].

Существует множество модификаций лучевого поиска, которые направлены на увеличение разнообразия генерируемых цепочек токенов. Авторы работы [8] провели сравнение таких модификаций по двум критериям: качество и разнообразность различных цепочек токенов. Наибольший интерес представляет модификация лучевого поиска с помощью механизма штрафования за похожесть цепочек (Hamming diversity reward) [9], так как она показывает сопоставимое с лучевым поиском качество, значительно увеличивая разнообразность выдачи.

Данная модификация организует одновременную работу нескольких алгоритмов лучевого поиска, которые работают без изменений. Изменения затрагивают вероятности $P(x_i|x_{i-1},...,x_1)$, которые эти алгоритмы используют. Первый алгоритм отрабатывает на неизменённых вероятностях, генерируя наиболее вероятные цепочки. Затем вероятности выбранных им токенов уменьшаются на некоторую константу:

$$\forall tok \in chosen_toks, \forall context : P(tok|context) := P(tok|context) - \alpha,$$

где $chosen_toks$ — выбранные предыдущими алгоритмами лучевого поиска токены, $context$ — цепочка токенов, образующая контекст, α — параметр, отвечающий за величину штрафа. После этого следующий алгоритм лучевого поиска отрабатывает на изменённых вероятностях и меняет их дальше для дальнейших итераций.

1.6. Выводы из обзора

Для достижения цели данной работы необходимо выбрать систему, способную генерировать фрагменты кода наиболее качественно. Для этого, согласно схеме на рисунке 4, необходимо выбрать представление кода, токенизатор, алгоритм генерации следующего токена кода и алгоритм генерации последовательностей.

Выбор представления кода

Последние работы в области автодополнения кода показывают, что использование кода в виде абстрактного синтаксического дерева позволяет добиться лучших результатов, нежели при работе с кодом как с текстом. Однако обычно такие системы строятся на основе уже существующих систем, которые представляют код в виде текста. Научные же работы по автодополнению фрагментами кода начали появляться только в мае 2020 года, из-за чего сразу построить систему, которая бы представлял код в виде абстрактных синтаксических деревьев, может быть сложно. Кроме того, коммерческие системы автодополнения фрагментами кода [23, 37] показывают хорошее качество при текстовом представлении кода. Поэтому, чтобы избежать непредвиденных сложностей при построении первого прототипа генерации фрагментов кода, было решено использовать код в виде текста. Затем можно будет модифицировать систему для другого представления кода.

Выбор метода токенизации кода

Совсем недавно авторы работы [43] собрали набор данных, состоящий из реальных сессий автодополнения кода, и сравнили несколько существующих подходов на этих данных. Оказалось, что результаты, которые приводились ранее на синтетических данных, имеют мало общего с результатами на реальных данных. Авторы сделали выводы о том, что подходы с закрытым словарём работают гораздо хуже аналогов с открытыми словарями. Исходя из этого, было принято решение использовать токенизатор с открытым словарём. В качестве такого токенизатора была выбрана техника парного кодирования байтов. Она позволяет не только быстрее генерировать последовательности, чем при использовании n-грамм или посимвольной токенизации, но и показывает лучшее качество в задаче автодополнения кода [17].

Выбор алгоритма автодополнения следующего токена

В той же статье [43] авторы говорят о том, что подходы на основе рекуррентных сетей с закрытым словарём показывают наихудшее качество, тогда как подходы на основе машинного обучения с открытым словарём сильно их превосходят. В своём анализе авторы говорят о том, что системы, основанные на нейронных сетях, превосходят остальные системы при прочих равных, поэтому они предполагают, что подобные системы с открытым словарём в будущем будут способны показывать наилучшие результаты.

В силу этого в качестве алгоритма автодополнения следующего токена кода было решено использовать нейронные сети. На данный момент архитектура Трансформер показывает лучшие результаты в сравнении с рекуррентными нейронными сетями как в задачах обработки естественных языков [1, 15, 18], так и в задаче автодополнения кода [5, 16, 31]. Однако во всех работах нейронная сеть с архитектурой Трансформер имеет больше настраиваемых параметров, чем рекуррентные нейронные сети, что плохо сказывается на времени работы алгоритма. Это может быть критично, так как функция автодополнения для комфортной работы должна отрабатывать незаметно для пользователя. Поэтому было решено протестировать оба варианта архитектуры нейронной сети для автодополнения следующего токена.

В качестве нейронной сети с Трансформер архитектурой в первую очередь было решено использовать Transformer-XL, поскольку она на данный момент показывает лучшее качество в задаче моделирования естественных языков [40]. В то же время существует гораздо более популярная нейронная сеть с архитектурой Трансформер – GPT-2, которая имеет гораздо б*о*льшую популярность, из-за чего существует множество реализаций различных техник её ускорения, что может пригодиться при встраивании такого алгоритма в расширение к IDE. В качестве же рекуррентной нейронной сети было решено использовать SHA-RNN [22], которая на одной из задач показывает качество, сопоставимое с Transformer-XL, но которая при этом в разы меньше и быстрее.

Выбор алгоритма генерации последовательностей

Для генерации фрагментов кода с помощью уже обученного алгоритма автодополнения следующего токена будет использоваться алгоритм лучевого поиска с механизмом штрафования за похожие цепочки. Данное решение было принято, поскольку именно такой алгоритм генерации последовательностей сегодня показывает наилучшие результаты в задачах обработки естественных языков [8].

1.7. Архитектура плагина

Опыт коммерческих систем [37] и последней работы в области автодополнения фрагментами кода [16] показывают, что для качественной генерации фрагментов кода необходимо использовать тяжеловесные и медленные нейронные сети. При этом для обеспечения приемлемого пользовательского опыта необходимо обеспечить минимальную задержку выдачи вариантов автодополнения. Поэтому, как и в других работах [16, 37], было решено запускать эти нейронные сети на графическом ускорителе отдельного сервера.

На рисунке 6 изображена архитектура всей системы в виде диаграммы компонентов. Система состоит из трёх основных частей: расширение для PyCharm на стороне клиента, серверная инфраструктура и алгоритм автодополнения фрагментами кода. На стороне клиента происходит взаимодействие с пользователем, предобработка данных и формирование запроса на сервер. Серверная инфраструктура обеспечивает бесперебойную работу алгоритма автодополнения путём распределения нагрузки между несколькими процессами. Алгоритм автодополнения отвечает за генерацию фрагментов кода по полученному контексту программы.

Само расширение для PyCharm и инфраструктуру для сервера реализовывал другой студент СПбГУ Кирилл Крылов. В рамках же данной работы будет разработан алгоритм автодополнения фрагментами кода, который состоит из нескольких модулей и управляется из модуля *Connector*.

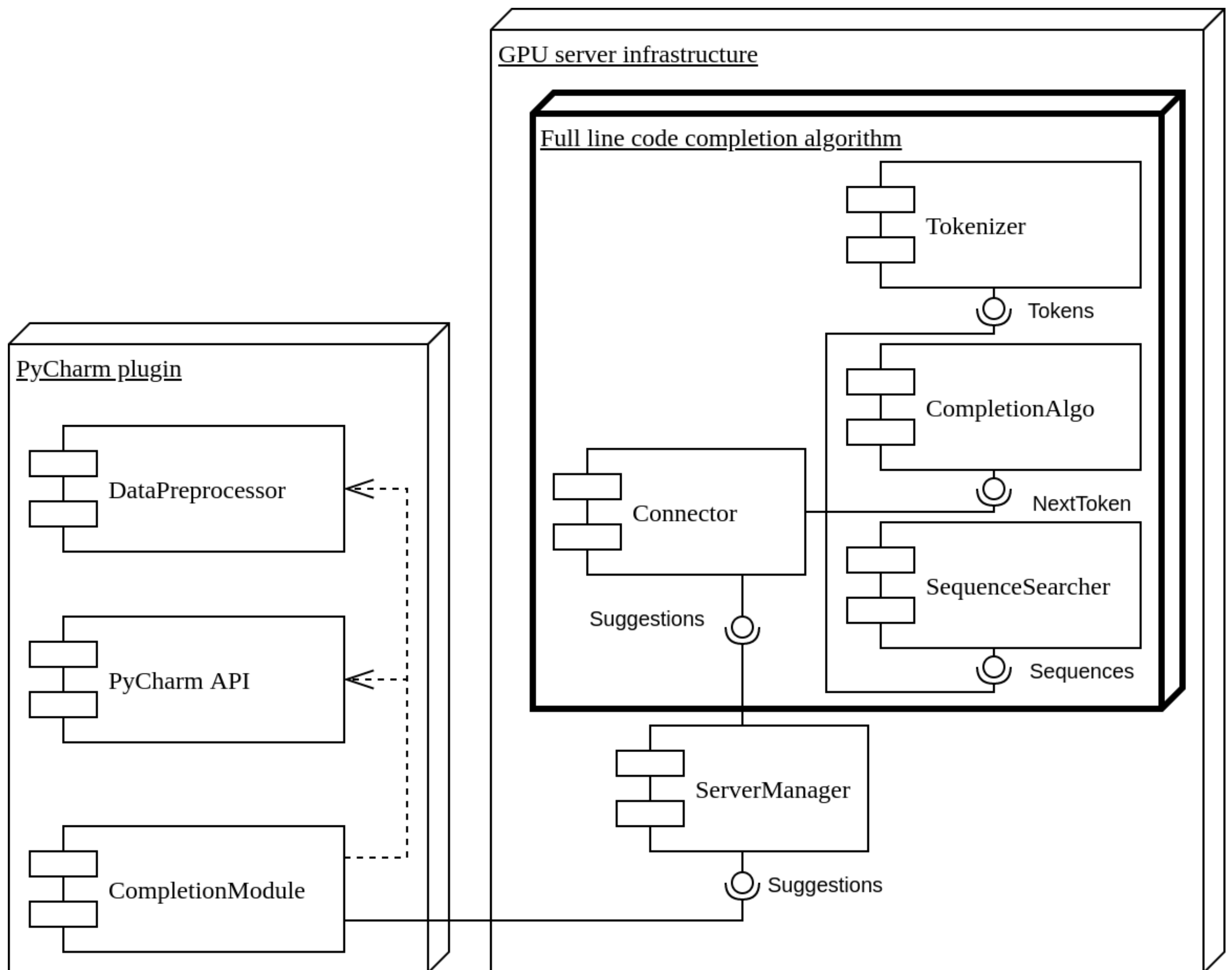


Рис. 6: Клиент-серверная архитектура плагина для PyCharm. Плагин отсылает контекст на сервер и получает фрагменты кода. Инфраструктура сервера обеспечивает работу нескольких процессов и распределяет нагрузку. В рамках данной работы реализована подсистема «Full line code completion algorithm».

2. Реализация алгоритма генерации фрагментов кода

В данной части работы приводится описание разработанной системы, некоторых её особенностей и проблем, которые они решают.

2.1. Подготовка данных для обучения

В качестве данных, на которых будут обучаться нейронные сети, было решено использовать исходный код с платформы GitHub. На этом ресурсе доступно огромное количество проектов, каждый из которых содержит множество файлов с исходным кодом. Качество проектов можно косвенно оценивать с помощью так называемых звёзд, которые пользователи могут ставить, если они считают проекты важными в том или ином смысле. Студентом СПбГУ Кириллом Крыловым были собраны все репозитории, количество звёзд у которых больше либо равно десяти. Объём таких репозиториев в сумме составил один терабайт, чего предположительно хватит для обучения модели любого размера [30]. Затем Кириллом Крыловым эти данные были отфильтрованы от точных дубликатов и тех файлов, которые предположительно были сгенерированы автоматически.

Любой алгоритм машинного обучения требует, чтобы во время обучения данные подавались на вход в случайном порядке. В идеальном сценарии, при каждом проходе по данным они должны быть перемешаны по-разному. Поэтому один раз перемешать данные заранее недостаточно, необходимо делать это прямо во время процесса обучения.

Специфика исходного кода такова, что он хранится в большом количестве относительно небольших файлов. Экспериментально было установлено, что скорость чтения таких данных в случайном порядке очень низкая, гораздо ниже скорости их обработки даже масштабной нейронной сетью.

Эту проблему можно решить, если перемешать данные заранее и сохранить их в нескольких файлах большого размера. Тогда можно счи-

тывать такие файлы в случайном порядке и перемешивать содержимое каждого прямо во время обучения алгоритма.

Для конвертирования множества мелких файлов в описанный выше формат был разработан скрипт на языке программирования Python. Этот язык был выбран, так как скорость выполнения скрипта в основном ограничена скоростью операций ввода и вывода с долговременного накопителя, и весь остальной проект уже был реализован на Python.

Схема работы скрипта изображена на рисунке 7. Изначально доступны пути ко всем файлам с исходным кодом, которые нельзя ни загрузить в память ввиду их объёма, ни случайно перемешать ввиду низкой скорости чтения в этом порядке. Поэтому они разбиваются на блоки по несколько путей, и блоки перемешиваются. Скорость чтения блоков в случайном порядке близка к скорости чтения всех файлов последовательно. Затем блоки считываются в память по несколько за раз, и все файлы в считанных блоках перемешиваются между собой для записи в несколько файлов большего размера в таком порядке.

Стоит отдельно отметить важность разбиения путей на блоки с последующим перемешиванием. Если этого не делать, а просто читать какое-то количество файлов и перемешивать их в памяти, то первые файлы из начальных данных будут всегда содержаться в первых файлах большего размера, то есть случайность будет нарушена, что может привести к ухудшению процесса обучения нейронных сетей.

Таким образом, подобная предобработка данных позволила снизить время обработки 85 гигабайт кода на Python с недели до трёх часов. Это критично, так как в будущем планируется поддержка других языков программирования, общий объём данных которых составляет 1 терабайт.

2.2. Обучение нейронных сетей

Для всех трёх выбранных архитектур нейронных сетей, а именно Transformer-XL, GPT-2 и SHA-RNN, существуют открытые реализации на языке Python, в которых реализованы сами нейронные сети и

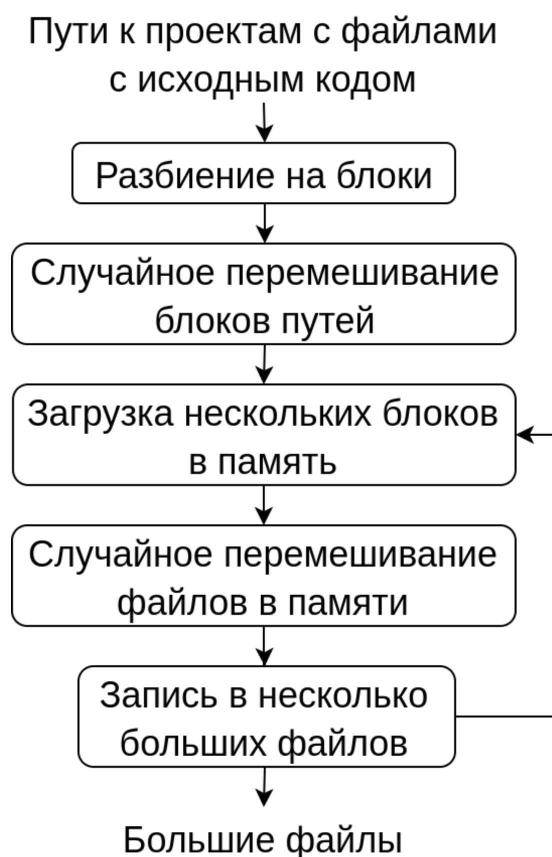


Рис. 7: Схема процесса предобработки данных

алгоритмы их обучения с помощью библиотеки PyTorch.

Для всех этих реализаций в рамках данной работы была поддержана возможность тренировки на исходном коде. Особое внимание стоит уделить работе с реализацией Transformer-XL, в которой было найдено несколько критических ошибок, которые не позволяли обучить алгоритм, и большое количество других мелких ошибок. Работа над этой реализацией проводилась в репозитории на GitHub[4]. Одна из критических ошибок заключалась в неправильном сохранении и восстановлении процесса обучения, которые необходимы на тот случай, если этот процесс непредвиденно прервётся, что довольно вероятно при обучении длительностью в неделю и больше.

Другая проблема в реализации Transformer-XL заключалась в том, что метрики качества периодически сильно падали, из-за чего обучение на первых стадиях замедлялось, а на поздних, когда скорость ро-

[4]Реализация Transformer-XL на GitHub `https://github.com/cybertronai/transformer-xl`

ста качества невелика, оно и вовсе прекращалось. Оказалось, что разбиение данных на примеры в довольно редком случае производилось некорректно. А именно, формирование нескольких примеров из конкатенации всех примеров производилось без учёта того, где эти примеры начинаются и заканчиваются. Таким образом, пример иногда начинался с середины файла, что периодически вызывало резкое ухудшение качества генерации модели.

2.3. Реализация лучевого поиска

Для генерации фрагментов кода необходимо разработать алгоритм поиска последовательностей, который позволит с помощью алгоритма, обученного на задаче оценки вероятности следующего токена, генерировать цепочки токенов, которые и будут образовывать фрагменты кода. Как уже было упомянуто в выводе из обзора, в качестве такого алгоритма было решено использовать лучевой поиск с механизмом штрафов за похожие цепочки.

В ходе поиска готового решения было замечено, что реализации, которые используют определённую эвристику [12], позволяющую ускорить процесс генерации в несколько раз, встречаются только как части какой-либо библиотеки и сильно завязаны на реализацию этих библиотек. Данный факт не позволяет взять готовую для использования реализацию. По этой причине было решено самостоятельно реализовать алгоритм лучевого поиска и его модификацию. Такой алгоритм довольно часто оказывается полезен и в других задачах в области применения машинного обучения в программной инженерии, поэтому при его разработке стоит обеспечить удобную интеграцию с другими алгоритмами предсказания следующего токена.

Архитектура разработанного алгоритма представлена на рисунке 8. Управление поиском цепочек токенов происходит через класс генератора (*Generator*), при создании которого необходимо указать параметры алгоритма: идентификаторы терминальных символов, размер лучевого поиска и параметры, связанные с механизмом штрафования. Затем

можно сгенерировать цепочку токенов, передав контекст для генерации и количество итераций в метод поиска.

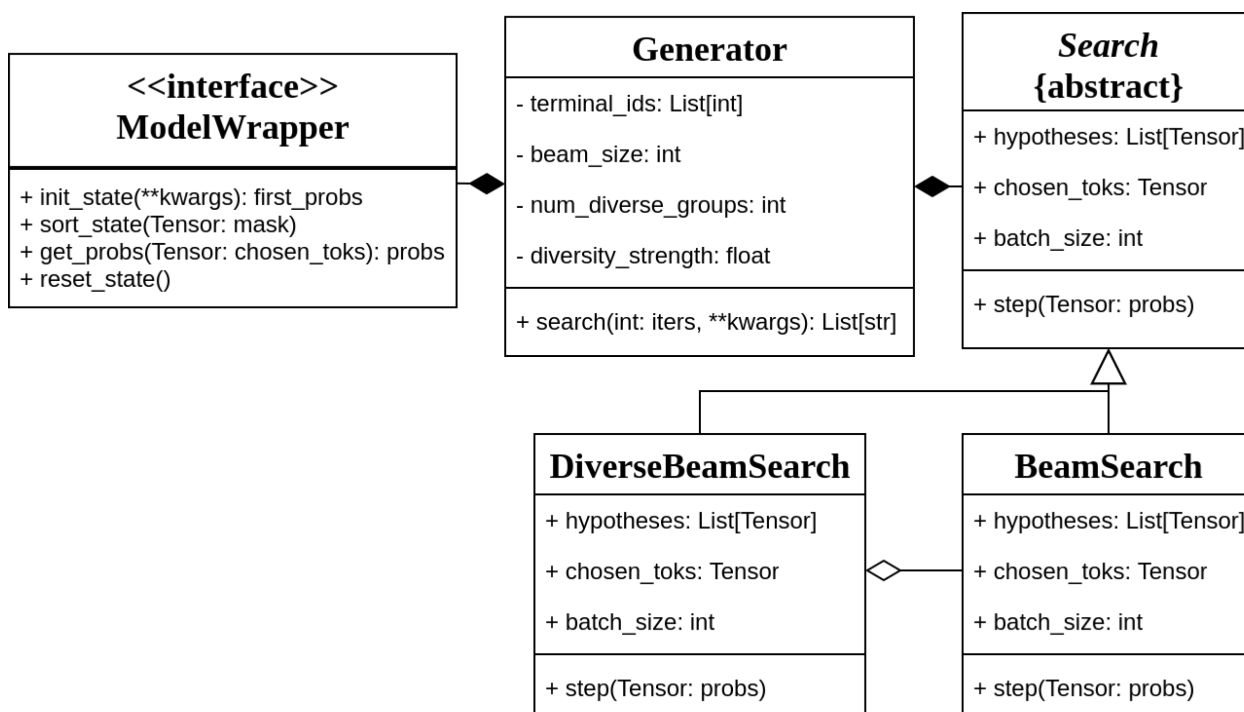


Рис. 8: Архитектура алгоритма лучевого поиска с механизмом штрафования за похожие цепочки.

Также был выделен абстрактный класс поиска (*Search*), наследуя который можно легко реализовывать различные алгоритмы поиска последовательностей токенов и их модификации. Этот класс имеет довольно минималистичный интерфейс: необходимо несколько раз вызвать метод шага алгоритма (*step*), передав в него вероятности появления следующих токенов. После каждого вызова этого метода состояние экземпляра поиска меняется, обновляя текущие наиболее вероятные гипотезы. Эти гипотезы в свою очередь используются для получения следующего вероятностного распределения при помощи алгоритма генерации одного токена.

Для того, чтобы воспользоваться этой реализацией алгоритма поиска последовательности токенов, имея алгоритм предсказания следующего токена, необходимо реализовать интерфейс *ModelWrapper*. При этом инициализировать состояние алгоритма можно как угодно, так

как все необходимые аргументы можно передавать в метод *search* у генератора. Эти параметры далее без изменений будут переданы в метод инициализации (*init_state*).

На текущий момент данной реализацией алгоритма успел воспользоваться в рамках своей выпускной квалификационной работы бакалавра студент НИУ ВШЭ Михаил Правилов в рамках задачи генерации описания к изменению в коде. Для этого он успешно реализовал интерфейс *ModelWrapper* для алгоритма, который генерирует такое описание по одному слову.

3. Оптимизация компонентов алгоритма

Все выбранные компоненты реализованной системы были изначально придуманы и разработаны для решения различных задач в области обработки естественных языков. Чтобы лучше удовлетворять требованиям, которые специфичны для задачи автодополнения фрагментами кода, некоторые из этих компонентов были соответствующим образом модифицированы.

3.1. Модификация парного кодирования байтов

Одним из таких требований к системе автодополнения фрагментами кода является высокая скорость работы. Эта скорость в основном зависит от времени работы генерации фрагментов кода, которое состоит из времени обработки контекста, растущего вместе с количеством токенов в контексте, и времени генерации нескольких следующих токенов.

Ускорить и обработку контекста, и генерацию нескольких токенов можно за счёт сокращения среднего количества токенов, которые требуются для представления произвольного фрагмента кода. Как уже было сказано в обзоре, одним из предназначений техники парного кодирования байтов как раз является сокращение количества токенов. Это достигается путём объединения самых частотных пар токенов в новый токен. При этом алгоритму запрещается объединять токены, которые находятся в разных лексемах языка программирования. Это необходимо для того, чтобы в одном токене не содержалось несколько лексем или их частей, что удобно при генерации одной лексемы языка программирования.

В данной же работе задача заключается в генерации целого фрагмента кода, поэтому описанное выше ограничение становится необязательным и его можно снять. Тогда несколько лексем языка программирования смогут объединяться в единые токены. Таким образом, среднее количество токенов, требуемых для представления одного и того же фрагмента кода, может сократиться.

В таблице 1 приведён пример токенизации одного и того же фраг-

мента кода с помощью техники парного кодирования байтов из предыдущих статей и предлагаемой модификации. С помощью такой модификации можно компактнее представлять последовательности лексем, которые встречаются в коде довольно часто, например сочетания «for i in » и «range(». Тем самым можно сократить как время обработки контекста, так и время генерации цепочки нескольких токенов.

Метод токенизации	Набор токенов	Кол-во токенов
BPE	for/ i/ in/ ran/ge/(/10/)/:	9
Модификация BPE	for i in /range(/10/):	4

Таблица 1: Пример токенизации фрагмента кода «for i in range(10):» с помощью оригинальной техники парного кодирования байтов (BPE) и предлагаемой модификации BPE.

На момент выполнения работы существовало несколько библиотек, которые предназначены для построения и использования токенизатора на основе техники парного кодирования байтов [33, 39, 45, 47]. Однако, как было выяснено, ни одна из этих библиотек не имеет интерфейса, который бы позволял с его помощью модифицировать алгоритм парного кодирования байтов описанным выше способом. Поэтому было решено взять самую быструю библиотеку, YouTokenToMe [45], и модифицировать её исходный код для реализации предложенного подхода. Код получившегося решения доступен на GitHub[5].

3.2. Токенизация неполного контекста

Вторая модификация токенизации контекста является актуальной не только для генерации фрагментов кода, но и для задачи генерации любого текста по некоторому контексту, при которой задействуется техника парного кодирования байтов в качестве механизма токенизации.

В области обработки естественных языков при решении такой задачи контекст сперва токенизируют с помощью парного кодирования байтов, а затем по этим токенам алгоритм делает предсказание. При этом,

[5]Реализованная модификация библиотеки YouTokenToMe: `https://github.com/JetBrains-Research/YouTokenToMe`

в момент токенизации контекста может случиться так, что последний токен окажется «неполным». Например, пользователь ввёл контекст «for i i» и хочет получить строку «for i in range(10):». В этом случае контекст будет представлен в виде токенов «for/ i/ i» и следующий токен, который должен предсказать алгоритм, будет «n». Тогда при генерации всех остальных токенов эта часть контекста будет состоять из токенов «for/ i/ i/n», что довольно необычно с точки зрения алгоритма, так как в словаре есть цельный токен « in», и во время обучения алгоритма все вхождения подстроки « in» всегда представляются в виде одного токена « in». По этой причине качество генерации при таком неполном контексте может ухудшаться.

Для того, чтобы решить эту проблему, предлагается использовать последний токен не в качестве токена в контексте, а в качестве дополнительной информации при генерации следующего токена. То есть алгоритм предсказания будет видеть все токены контекста за исключением последнего и предсказывать распределение вероятностей следующего токена. Это означает, что каждому токену в словаре будет присвоена вероятность того, что этот токен может оказаться следующим. Тогда с помощью последнего «неполного» токена можно отфильтровать токены в словаре так, чтобы остались только те, что начинаются так же, как и этот «неполный» токен. Для примера выше, когда пользователь ввёл контекст «for i i», этот контекст будет представлен в виде токенов «for/ i», а токен « i» будет использован для фильтрации следующего предсказания. По этому контексту алгоритм сгенерирует токен « in», не имея при этом возможности сгенерировать какой-либо токен, который начинается не с « i». Тогда при генерации всех оставшихся токенов контекстом будут являться токены «for/ i/ in», что полностью соответствует тому, что алгоритм видел при обучении.

Нетривиальной частью реализации такой токенизации неполного контекста является фильтрация токенов словаря с помощью некоторого префикса. Сложность состоит в том, что словарь чаще всего состоит из десятков тысяч различных токенов. Поэтому для решения этой задачи было решено использовать структуру данных бор, несколько её

модифицировав для этой задачи. Для этого по словарю, представленному в виде набора строк, строится бор, который в узлах хранит не номера терминирующих строк, а номера всех строк, которые имеют соответствующий пройденному пути префикс. Пример такого бора для словаря {she: 1, his: 2, hers: 3, he: 4} можно увидеть на рисунке 9. С помощью такой структуры данных фильтрацию токенов в словаре можно проводить за $\mathcal{O}(|S|)$, где S — это последний «неполный» токен в словаре.

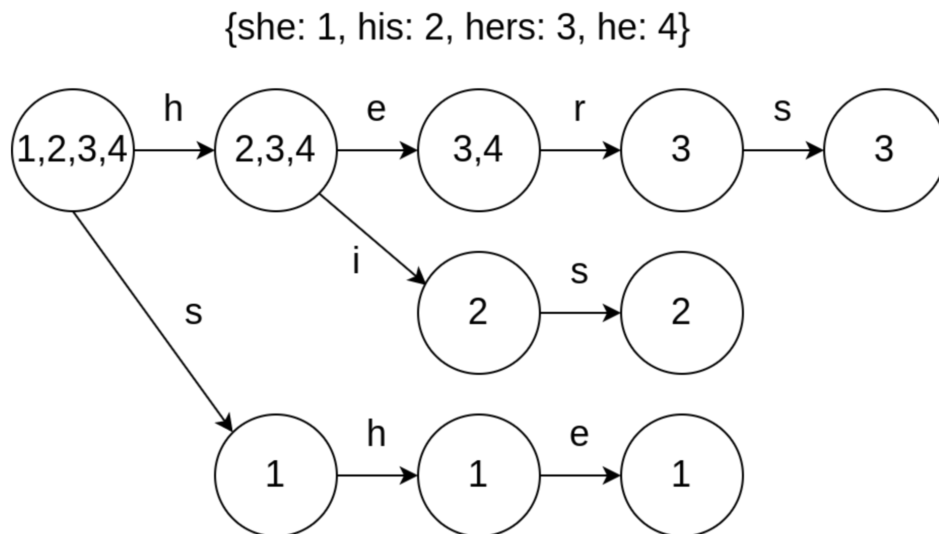


Рис. 9: Пример бора для словаря {she: 1, his: 2, hers: 3, he: 4}. Используется для эффективного поиска всех подходящих токенов в словаре по некоторому префиксу.

4. Апробация реализованной системы

В данной главе описывается сравнение полученной системы с аналогами и влияние модификаций процесса токенизации на качество и скорость генерации фрагментов кода.

4.1. Данные

Перед обучением каких-либо алгоритмов данные были случайно разбиты на три части, одна из которых имеет объём 30 мегабайт и служит для тестирования различных частей системы, а две другие используются для обучения как алгоритмов предсказания следующего токена, так и техники парного кодирования байтов. Тестовая часть состоит из 7 426 файлов с исходным кодом на Python из различных проектов, которых нет в тренировочных данных. Общее число строк в этих файлах равно 708 692.

4.2. Метрика качества

Метрики, которые используются для оценки качества автодополнения одного токена кода, вычисляются исходя из положения правильного токена в выдаче системы автодополнения. В случае же автодополнения фрагментами кода «правильность» предлагаемого варианта не так однозначна, поскольку может быть ситуация, в которой предлагаемый вариант будет отличаться от правильного на один символ. В таком случае реальный пользователь скорее всего предпочтёт выбрать такой вариант и заменит этот символ и, тем самым, сэкономит себе время. Поэтому из-за невозможности однозначно разбить все предлагаемые варианты на правильные и неправильные эти метрики использовать нельзя.

В работе, посвящённой автодополнению фрагментами кода [16], была использована метрика на основе расстояния Левенштейна, которая решает описанную выше проблему. Однако с помощью такой метрики нельзя сравнить между собой системы автодополнения одним токеном

и фрагментами кода.

Для решения этих проблем Кириллом Крыловым была реализована метрика CodeGolf, которая подсчитывает количество пользовательских действий в редакторе IDE, которое необходимо для набора некоторого файла с исходным кодом. При отсутствии какого-либо автодополнения для набора файла длины n необходимо n действий, так как необходимо вручную ввести каждый символ этого файла. Если же есть какая-то система автодополнения, то программе кроме ввода символов доступен выбор вариантов, которые эта система предлагает. Таким образом, если система автодополнения предлагает хорошие варианты, метрика становится меньше. Для учёта длины файла с исходным кодом, на котором проводится измерение метрики, число действий нормируется на длину этого файла:

$$\text{CodeGolf} = \frac{\#\text{действий для написания файла}}{\text{длина файла с исходным кодом}} * 100\%.$$

4.3. Сравнение с аналогами

Полученную систему автодополнения фрагментами кода было решено сравнить с несколькими уже существующими системами. В первую очередь необходимо произвести сравнение с уже встроенной в PyCharm системой автодополнения одного токена, так как именно её функциональность призван заменить разработанный плагин. Затем стоит произвести сравнение с ближайшим аналогом, Deep TabNine, который также доступен пользователям в виде плагина к PyCharm.

Точно неизвестно, на каких данных обучалась система Deep TabNine, из-за чего сравнение метрик на данных с GitHub невозможно. По этой причине набор данных для сравнения систем был составлен из трёх различных проектов, два из которых являются популярными проектами на сервисе GitLab, копий которых нет на GitHub, а третий проект является кодом для инфраструктуры сервера данной системы автодополнения, который публично не доступен на GitHub.

В таблице 2 представлены значения метрики CodeGolf для каждой

из выбранных систем и среднее время отклика системы автодополнения во время вычисления метрики. Результаты сравнения демонстрируют, что полученные в рамках данной работы системы превосходят по качеству как встроенное в PyCharm решение, так и ближайший аналог Deep TabNine. Однако они же являются самыми медленными из представленных. Ускорение нейронных сетей сейчас является активной областью исследований [7, 26, 29, 38], которая успешно применяется в системах автодополнения фрагментами кода [16], поэтому этот недостаток можно исправить в будущем.

Система автодополнения	CodeGolf, % 95% дов. интервал (меньше лучше)	Время отклика, мс
PyCharm ML completion	(74, 80)	**(7, 11)**
Deep TabNine	(43, 57)	(77, 92)*
Эта работа, GPT-2	**(30, 40)**	(279, 316)**
Эта работа, SHA-RNN	(37, 44)	(176, 224)**

Таблица 2: Качество и время отклика различных систем автодополнения. *Вычисления производились на CPU (2,3 GHz 8Core Intel Core i9). **Система имеет клиент-серверную архитектуру (задержка сети 80-100 мс), вычисления производились на GPU (NVIDIA Tesla V100).

4.4. Влияние модификаций токенизатора

В данной части работы исследуется влияние реализованных модификаций техники парного кодирования байтов на скорость и качество работы системы.

Влияние на скорость работы

В первую очередь было решено измерить, как влияет на скорость модификация, которая заключается в снятии ограничения на слияние токенов между лексемами языка. Для этого на всей тестовой выборке была посчитана средняя степень сжатия каждого файла с исходным

кодом. Она может быть вычислена по следующей формуле:

$$\text{Степень сжатия} = \frac{\text{длина фрагмента кода}}{\#\text{токенов после токенизации}}.$$

Стоит заметить, что она напрямую зависит от количества операций слияний токенов при построении словаря. Поэтому для оценки модификации в сравнении с оригинальным алгоритмом на рисунке 10 изображена зависимость степени сжатия от количества слияний. Судя по получившимся результатам, такая модификация позволяет значительно снизить количество токенов, требуемых для токенизации одного и того же фрагмента кода, что ускоряет процесс генерации.

Сейчас в системе используется словарь с 16 384 различными токенами. Для такого количества слияний среднюю длину файла в токенах с помощью модификации удалось уменьшить в 1.47 раз, что в среднем ускорит процесс генерации в те же 1.47 раз.

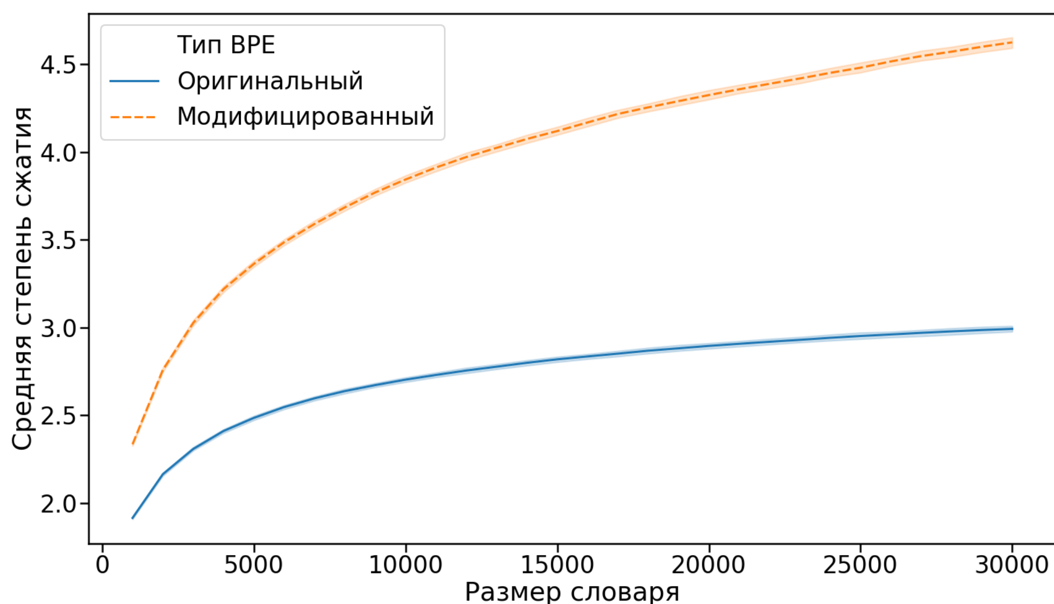


Рис. 10: Степень сжатия кода с помощью техники парного кодирования байтов в зависимости от размера словаря.

Влияние на качество работы

Ввиду относительно низкой скорости работы системы, которая вычисляет метрику CodeGolf, для исследования влияния изменения токенизации неполного контекста были использованы те же данные, что и для сравнения системы с аналогами. Для сравнения были получены значения метрики CodeGolf с применением модификации для токенизации неполного контекста и без него. Результаты представлены в таблице 3. Модификация показывает себя значимо лучше только для одной из архитектур нейронных сетей. Возможно, при использовании большего количества данных результаты могут стать значимыми и для второй архитектуры.

Система автодополнения	CodeGolf, % 95% дов. интервал (меньше лучше)
GPT-2 без токенизации неполного контекста	(37, 46)
GPT-2 с токенизацией неполного контекста	(30, 40)
SHA-RNN без токенизации неполного контекста	(46, 54)
SHA-RNN с токенизацией неполного контекста	(37, 44)

Таблица 3: Качество реализованной системы автодополнения с модификацией токенизации неполного контекста и без неё.

Заключение

В ходе данной работы были получены следующие результаты.

1. Проведён обзор предметной области. Рассмотрены системы автодополнения одного токена кода и процесс генерации текстов применительно к автодополнению фрагментами кода. Выбраны компоненты системы автодополнения фрагментами кода.

2. Выбран и реализован алгоритм генерации фрагментов кода. Реализована предобработка данных объёмом порядка терабайта. Обучено несколько архитектур нейронных сетей, для одной из которых были исправлены критические ошибки. Спроектирован и реализован алгоритм лучевого поиска и его модификаций.

3. Оптимизированы компоненты реализованного решения под специфику поставленной задачи. Предложены и реализованы улучшенная токенизация неполного контекста и модификация парного кодирования байтов для автодополнения фрагментами кода.

4. Проведена апробация полученной системы. Произведено её сравнение с автодополнением PyCharm и Deep TabNine. Реализованная система показывает лучшие результаты по сравнению с аналогами. Исследовано влияние предложенных улучшений процесса токенизации на скорость и качество работы системы. Модификации позволяют ускорить процесс генерации в 1,5 раза и улучшить качество качество предлагаемых фрагментов кода.

В результате данной работы был реализован алгоритм автодополнения фрагментами кода, который был успешно внедрён в плагин для PyCharm (см. акт о внедрении в приложении А). Также была разработана инфраструктура для обучения, которую в дальнейшем планируется переиспользовать для разработки алгоритмов для других языков программирования. Весь код доступен по ссылке на GitHub[6].

[6]Репозиторий с кодом для обучения и использования алгоритма автодополнения фрагментами кода `https://github.com/SokolovYaroslav/FLCC-Model-Training`

Благодарности

Я бы хотел поблагодарить Ярослава Булатова, одного из авторов репозитория «Scalable training of Transformer-XL»[7], который помогал адаптировать Transformer-XL для задачи генерации кода и выделил вычислительные ресурсы для получения результатов с помощью этой нейронной сети.

Также я бы хотел поблагодарить Никиту Поварова и компанию JetBrains за выделение вычислительных ресурсов для всех дальнейших экспериментов.

Отдельная благодарность JetBrains Research за поощрение стипендиями многих студентов, за счёт которых они могут полностью посвятить себя исследовательской деятельности.

Кроме того, я бы хотел выразить свою благодарность Кириллу Крылову, который отлично делал свою работу в этом проекте и помогал ставить эксперименты.

Большое спасибо Егору Булычеву за то, что согласился отрецензировать эту работу и дал ценные замечания.

Отдельное спасибо Ярославу Голубеву за помощь с пониманием всех аспектов русского и английского языков и за ценные замечания по простоте восприятия работы.

И, конечно же, я бы хотел поблагодарить своего научного руководителя, Тимофея Брыксина, который очень качественно и трудолюбиво оказывает помощь своим студентам в проведении исследований и написании дипломных работ.

[7] https://github.com/cybertronai/transformer-xl

Список литературы

[1] Attention is all you need / Ashish Vaswani, Noam Shazeer, Niki Parmar et al. // Advances in neural information processing systems. — 2017. — P. 5998–6008.

[2] Cambronero José P, Rinard Martin C. AL: autogenerating supervised learning programs // Proceedings of the ACM on Programming Languages. — 2019. — Vol. 3, no. OOPSLA. — P. 1–28.

[3] Chakraborty Saikat, Allamanis Miltiadis, Ray Baishakhi. CODIT: Code Editing with Tree-Based NeuralMachine Translation // arXiv preprint arXiv:1810.00314. — 2018.

[4] Code Completion with Neural Attention and Pointer Networks / Jian Li, Yue Wang, Michael R. Lyu, Irwin King // Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18. — International Joint Conferences on Artificial Intelligence Organization, 2018. — 7. — P. 4159–4165. — Access mode: https://doi.org/10.24963/ijcai.2018/578.

[5] Code Prediction by Feeding Trees to Transformers / Seohyun Kim, Jinman Zhao, Yuchi Tian, Satish Chandra // arXiv preprint arXiv:2003.13848. — 2020.

[6] Deep-AutoCoder: Learning to Complete Code Precisely with Induced Code Tokens / X. Hu, R. Men, G. Li, Z. Jin // 2019 IEEE 43rd Annual Computer Software and Applications Conference (COMPSAC). — Vol. 1. — 2019. — July. — P. 159–168.

[7] DistilBERT, a distilled version of BERT: smaller, faster, cheaper and lighter / Victor Sanh, Lysandre Debut, Julien Chaumond, Thomas Wolf // arXiv preprint arXiv:1910.01108. — 2019.

[8] Diverse beam search: Decoding diverse solutions from neural sequence models / Ashwin K Vijayakumar, Michael Cogswell,

Ramprasath R Selvaraju et al. // arXiv preprint arXiv:1610.02424. — 2016.

[9] Diverse beam search for improved description of complex scenes / Ashwin K Vijayakumar, Michael Cogswell, Ramprasaath R Selvaraju et al. // Thirty-Second AAAI Conference on Artificial Intelligence. — 2018.

[10] Fast and Memory-Efficient Neural Code Completion / Alexey Svyatkovskoy, Sebastian Lee, Anna Hadjitofi et al. // arXiv preprint arXiv:2004.13651. — 2020.

[11] Fischer Lars, Hanenberg Stefan. An empirical investigation of the effects of type systems and code completion on api usability using typescript and javascript in ms visual studio // ACM SIGPLAN Notices. — 2015. — Vol. 51, no. 2. — P. 154–167.

[12] Google's neural machine translation system: Bridging the gap between human and machine translation / Yonghui Wu, Mike Schuster, Zhifeng Chen et al. // arXiv preprint arXiv:1609.08144. — 2016.

[13] Harer Jacob, Reale Chris, Chin Peter. Tree-Transformer: A Transformer-Based Method for Correction of Tree-Structured Data // arXiv preprint arXiv:1908.00449. — 2019.

[14] Hellendoorn Vincent J, Devanbu Premkumar. Are deep neural networks the best choice for modeling source code? // Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering / ACM. — 2017. — P. 763–773.

[15] Improving language understanding by generative pre-training / Alec Radford, Karthik Narasimhan, Tim Salimans, Ilya Sutskever // URL https://s3-us-west-2. amazonaws. com/openai-assets/researchcovers/languageunsupervised/language understanding paper. pdf. — 2018.

[16] IntelliCode Compose: Code Generation Using Transformer / Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, Neel Sundaresan // arXiv. — 2020. — P. arXiv–2005.

[17] Karampatsis Rafael-Michael, Sutton Charles. Maybe Deep Neural Networks are the Best Choice for Modeling Source Code // arXiv preprint arXiv:1903.05734. — 2019.

[18] Language models are unsupervised multitask learners / Alec Radford, Jeffrey Wu, Rewon Child et al. // OpenAI Blog. — 2019. — Vol. 1, no. 8.

[19] Latent Predictor Networks for Code Generation / Wang Ling, Phil Blunsom, Edward Grefenstette et al. // Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). — 2016. — P. 599–609.

[20] Learning Python Code Suggestion with a Sparse Pointer Network / Avishkar Bhoopchand, Tim Rocktäschel, Earl T. Barr, Sebastian D. Riedel // ArXiv. — 2017. — Vol. abs/1611.08307.

[21] Learning long-term dependencies with gradient descent is difficult / Yoshua Bengio, Patrice Simard, Paolo Frasconi et al. // IEEE transactions on neural networks. — 1994. — Vol. 5, no. 2. — P. 157–166.

[22] Merity Stephen. Single Headed Attention RNN: Stop Thinking With Your Head // arXiv preprint arXiv:1911.11423. — 2019.

[23] Microsoft AI-assisted tools. — Access mode: https://devblogs.microsoft.com/visualstudio/ai-assisted-developer-tools/ (online; accessed: 19.12.2019).

[24] Murphy Gail C, Kersten Mik, Findlater Leah. How are Java software developers using the Elipse IDE? // IEEE software. — 2006. — Vol. 23, no. 4. — P. 76–83.

[25] Pythia: AI-assisted Code Completion System / Alexey Svyatkovskiy, Ying Zhao, Shengyu Fu, Neel Sundaresan //

Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. — KDD '19. — New York, NY, USA : ACM, 2019. — P. 2727–2735. — Access mode: http://doi.acm.org/10.1145/3292500.3330699.

[26] Q8bert: Quantized 8bit bert / Ofir Zafrir, Guy Boudoukh, Peter Izsak, Moshe Wasserblat // arXiv preprint arXiv:1910.06188. — 2019.

[27] Raychev Veselin, Bielik Pavol, Vechev Martin. Probabilistic Model for Code with Decision Trees // Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications. — OOPSLA 2016. — New York, NY, USA : ACM, 2016. — P. 731–747. — Access mode: http://doi.acm.org/10.1145/2983990.2984041.

[28] Raychev Veselin, Vechev Martin, Yahav Eran. Code Completion with Statistical Language Models // Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation. — PLDI '14. — New York, NY, USA : ACM, 2014. — P. 419–428. — Access mode: http://doi.acm.org/10.1145/2594291.2594321.

[29] Sanh Victor, Wolf Thomas, Rush Alexander M. Movement Pruning: Adaptive Sparsity by Fine-Tuning // arXiv preprint arXiv:2005.07683. — 2020.

[30] Scaling laws for neural language models / Jared Kaplan, Sam McCandlish, Tom Henighan et al. // arXiv preprint arXiv:2001.08361. — 2020.

[31] A Self-Attentional Neural Architecture for Code Completion with Multi-Task Learning / Fang Liu, Ge Li, Bolin Wei et al. // arXiv preprint arXiv:1909.06983. — 2019.

[32] Sennrich Rico, Haddow Barry, Birch Alexandra. Neural Machine Translation of Rare Words with Subword Units // Proceedings of the

54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers). — 2016. — P. 1715–1725.

[33] SentencePiece, библиотека для токенизации с помощью техники парного кодирования байтов. — Access mode: https://github.com/google/sentencepiece (online; accessed: 23.05.2020).

[34] Shiv Vighnesh, Quirk Chris. Novel positional encodings to enable tree-based transformers // Advances in Neural Information Processing Systems. — 2019. — P. 12058–12068.

[35] StructVAE: Tree-structured latent variable models for semi-supervised semantic parsing / Pengcheng Yin, Chunting Zhou, Junxian He, Graham Neubig // arXiv preprint arXiv:1806.07832. — 2018.

[36] Sutskever Ilya, Vinyals Oriol, Le Quoc V. Sequence to Sequence Learning with Neural Networks // Advances in Neural Information Processing Systems 27 / Ed. by Z. Ghahramani, M. Welling, C. Cortes et al. — Curran Associates, Inc., 2014. — P. 3104–3112. — Access mode: http://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf.

[37] TabNine home page. — Access mode: https://tabnine.com/ (online; accessed: 19.12.2019).

[38] Tinybert: Distilling bert for natural language understanding / Xiaoqi Jiao, Yichun Yin, Lifeng Shang et al. // arXiv preprint arXiv:1909.10351. — 2019.

[39] Tokenizers, библиотека для токенизации с помощью техники парного кодирования байтов. — Access mode: https://github.com/huggingface/tokenizers (online; accessed: 23.05.2020).

[40] Transformer-xl: Attentive language models beyond a fixed-length context / Zihang Dai, Zhilin Yang, Yiming Yang et al. // arXiv preprint arXiv:1901.02860. — 2019.

[41] Tree-Structured Attention with Hierarchical Accumulation / Xuan-Phi Nguyen, Shafiq Joty, Steven CH Hoi, Richard Socher // arXiv preprint arXiv:2002.08046. — 2020.

[42] Wang Yau-Shian, Lee Hung-Yi, Chen Yun-Nung. Tree Transformer: Integrating Tree Structures into Self-Attention // arXiv preprint arXiv:1909.06639. — 2019.

[43] When Code Completion Fails: A Case Study on Real-world Completions / Vincent J. Hellendoorn, Sebastian Proksch, Harald C. Gall, Alberto Bacchelli // Proceedings of the 41st International Conference on Software Engineering. — ICSE '19. — Piscataway, NJ, USA : IEEE Press, 2019. — P. 960–970. — Access mode: `https://doi.org/10.1109/ICSE.2019.00101`.

[44] Yin Pengcheng, Neubig Graham. A syntactic neural model for general-purpose code generation // arXiv preprint arXiv:1704.01696. — 2017.

[45] YouTokenToMe, библиотека для токенизации с помощью техники парного кодирования байтов. — Access mode: `https://github.com/VKCOM/YouTokenToMe` (online; accessed: 23.05.2020).

[46] Zavershynskyi Maksym, Skidanov Alex, Polosukhin Illia. NAPS: Natural program synthesis dataset // arXiv preprint arXiv:1807.03168. — 2018.

[47] fastBPE, библиотека для токенизации с помощью техники парного кодирования байтов. — Access mode: `https://github.com/glample/fastBPE` (online; accessed: 23.05.2020).

[48] A study of visual studio usage in practice / Sven Amann, Sebastian Proksch, Sarah Nadi, Mira Mezini // 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER) / IEEE. — Vol. 1. — 2016. — P. 124–134.

A. Акт о внедрении

Акт

о внедрении выпускной квалификационной работы
студента Соколова Ярослава Сергеевича на тему
«Система автодополнения фрагментами кода»

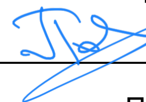
Настоящий акт подтверждает, что результаты выпускной квалификационной работы Соколова Я.С. на тему «Система автодополнения фрагментами кода» успешно внедрены в проект «Full Line Code Completion», реализуемом ООО «ИнтеллиДжей Лабс». Что позволило добавить возможность автодополнения целых строк кода в интегрированной среде разработки «PyCharm».

В представленных материалах проведён обширный обзор предметной области, аргументированно выбрано и затем реализовано наиболее подходящее решение для задачи автодополнения целых строк кода.

По предварительным результатам, такое автодополнение позволяет быстрее писать типовые фрагменты кода на языке программирования Python, сокращая среднее число необходимых действий для написания кода примерно в два раза по сравнению с уже существующей системой автодополнения.

Руководитель команды аналитики

Координатор проектов по машинному обучению
ООО «ИнтеллиДжей Лабс»



____

Поваров Н.И.

26 мая 2020 г.