ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

Факультет Санкт-Петербургская школа
физико-математических и компьютерных наук

Кузьмин Сергей Сергеевич

Вывод типов в Python с использованием машинного обучения и его интеграция в PyCharm

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
В.И. Бибаев

Руководитель
канд. тех. наук
Т.А. Брыксин

Консультант
Е.О. Богомолов

Санкт-Петербург 2022

Оглавление

Динамически типизированные языки программирования, такие как Python и JavaScript, увеличивают скорость разработки в обмен на преимущества статической типизации, среди которых поиск ошибок, связанных с типизацией, и более качественная поддержка кода. Для получения преимуществ статической типизации в таких языках появляются опциональные аннотации типов. Поскольку добавление аннотаций является долгим процессом, способным вести к ошибкам, исследователи используют разные методы для его автоматизации. Среди таких методов существуют статические методы, корректно, но редко предсказывающие тип, которые используются в статических анализаторов типов и средах разработки, и методы машинного обучения, предсказывающие тип вероятностно, но в большем числе случаев. На данный момент в работах, посвященным выводу типов с использованием машинного обучения, не рассматривается практическая постановка задачи, а именно помощь разработчикам в процессе их работы в IDE. В частности, большинство работ не фокусируются на потребляемых моделями ресурсах, не учитывают в оценке качества моделей статический анализ, используемый в IDE, а также не предоставляют возможностей для использования представленных моделей в процессе разработки. В данной работе проведено сравнение существующих подходов к выводу типов на языке Python, учитывающее результаты статического вывода типов в IDE PyCharm, а также представлен плагин для IDE PyCharm, позволяющий использовать информацию о типах, полученных с помощью моделей машинного обучения, для добавления аннотаций типов в процессе разработки и улучшения качество алгоритмов автодополнения в IDE PyCharm.

Ключевые слова: вывод типов, Python, PyCharm, автодополнение кода, плагины IntelliJ.

Dynamically typed programming languages, such as Python and JavaScript, provide increased development speed in exchange for advantages of static typing, some of which are embedded type checking and better maintaining of codebases. To mitigate these issues and gain advantages of statically typed languages, dynamically typed languages introduced optional type annotations. Since providing annotations manually is an error-prone and work-consuming process, researchers invented several methods for automatic type inference. Such methods are divided into two groups: methods based on static analysis which always provide sound predictions but can rarely deduce a type, and methods based on machine learning which are probabilistic and not always sound but provide predictions much more often. As of now, most of researches dedicated to type inference based on machine learning do not consider practical applications of this task, namely helping developers during their work with IDE. In particular, researches do not pay attention to resources consumed by their models, do not consider the statyc analysis implememented in modern IDEs and offer no options to apply their models during development in IDE. This work conducts a comparison of existing approaches to type inference in Python which takes in consideration static analysis implemented in PyCharm IDE. This work also introduces a plugin for PyCharm IDE that allows developers to use type predictions provided by machine learning methods to annotate their code during development. The plugin also uses type predictions to improve the quality of code completion algorithms implemented in PyCharm IDE.

Keywords: type inference, Python, PyCharm, code completion, IntelliJ plugins.

Введение

Актуальность и релевантные работы

В течение последних лет динамически типизированные языки, такие как Python или JavaScript, становятся все более популярными. Так, в рейтинге IEEE Spectrum за 2021 год Python занимает первое место в списке языков программирования, используемых для разработки веб-сервисов, настольных и научных приложений [32]. Динамическая типизация позволяет ускорить разработку [9, 29] в обмен на отсутствие ошибок типизации [9] и раннее обнаружение ошибок [12], предоставляемых статически типизированными языками.

Для того чтобы справиться с этими проблемами, в 2014 году было предложено улучшение языка Python PEP-484 [19], добавляющее опциональные аннотации типов в версии Python начиная с 3.5. Такие аннотации не влияют на результат выполнения программы, но помогают программистам разрабатывать и поддерживать код. С другой стороны, добавление и проверка всех аннотаций вручную, как и аннотирование старых баз кода, является затруднительной задачей для разработчиков. В связи с этим, для автоматизации этих процессов было разработано множество инструментов. В основном эти инструменты делятся на две группы: статические анализаторы кода [41, 22, 11], позволяющие и проверять корректность типов, и добавлять новые аннотации (выводить типы автоматически), соответствующие уже существующим, и анализаторы, основанные на методах машинного обучения и использующие косвенную информацию о коде [6, 7, 15, 33, 34, 35], например, имена функций, параметров, комментарии к функциям.

Инструменты, основанные на статическом анализе, позволяют гарантировать корректность полученных аннотаций, поскольку полагаются в своем выводе типов на строгие правила. В свою очередь, методы, основанные на машинном обучении, являются вероятностными и могут использовать большее количество информации, что позволяет им покрывать большее множество аннотаций в сравнении со статическими анализаторами. Это позволяет использовать преимущества обоих методов и проверять предложенные с по-

мощью машинного обучения аннотации на корректность. Такой подход применяется в некоторых работах [34, 35]. В большинстве работ, основанных на машинном обучении, рассматривается академическая постановка задачи – вывести типы неаннотированных элементов на большом корпусе кода, что позволяет оценивать и сравнивать результативность моделей. При этом исследователями редко рассматривается другая постановка задачи вывода типов – вывод типов в процессе разработки в IDE, а именно не исследованы затрачиваемые предложенными решениями ресурсы, не исследованы метрики, которые бы учитывали вывод типов в статических анализаторах или IDE, не исследованы метрики, относящиеся к другим практическим задачам, таким как автодополнение в процессе разработки в IDE.

Цель и задачи

Данная работа ставит целью создание инструмента, позволяющего использовать вывод типов на основе методов машинного обучения в процессе разработки в IDE. Для этого ставятся следующие задачи:

- Проанализировать существующие подходы к выводу типов на основе машинного обучения и проанализировать их применимость с учетом статического вывода типов, используемого в IDE.
- Интегрировать наиболее подходящее решение в IDE в качестве плагина.
- Проанализировать возможность использования данного плагина в реальном времени.
- Проанализировать влияние работы плагина на качество работы IDE в других задачах.

Достигнутые результаты

В рамках данной работы был реализован плагин MLTypes4PyCharm для IDE PyCharm, позволяющий пользоваться моделями машинного обучения

в процессе разработки для задач восстановления аннотаций типов у параметров и возвращаемых значений функций. Плагин позволяет находить неаннотированные функции и параметры в коде, показывать их пользователю и использовать встроенные модели машинного обучения для добавления аннотаций типов в реальном времени. В рамках реализации плагина, по сравнению с оригинальной реализацией одной из моделей, были ускорены в 6.3 и более чем в 100 раз сбор данных о функциях на Python и вызов модели, соответственно. Полученное ускорение позволяет использовать MLTypes4PyCharm в других задачах PyCharm, связанных с анализом кода. Плагин предоставляет информацию о выведенных типах системе автодополнения вызовов методов параметров функций и методов классов. Предоставленная информация позволила повысить метрику полноты для встроенного алгоритма автодополнения в PyCharm на 4.5% и 11% для двух различных интегрированных моделей.

Для поиска наиболее подходящей модели для интеграции был предложен алгоритм сравнения качества моделей, который учитывает статический вывод типов, использующийся в IDE PyCharm. Для этого был реализован плагин без графического интерфейса, способный автоматически открывать проекты на Python в PyCharm в консольном режиме, разрешать внутренние и внешние зависимости в проектах и собирать данные о типах, выведенных PyCharm. С помощью этого плагина было проведено сравнение существующих подходов к выводу типов в Python с использованием машинного обучения, учитывающее статический вывод типов в PyCharm, а также сравнение ресурсов, потребляемых различными моделями. Было показано, что более легковесная модель после учета PyCharm сокращает разрыв с тяжелой и более эффективной моделью на 4.4% в задаче предсказания всех типов, и на 20% в задаче предсказания типов возвращаемых значений, при этом она требует более чем в 150 раз меньше памяти.

Для сбора данных о функциях на Python был реализован плагин без графического интерфейса, позволяющий в автоматическом режиме собирать данные о функциях, необходимые для работы существующих моделей, основанных на машинном обучении. Данные о функциях извлекаются с помощью внутреннего представления PyCharm, что позволяет получать

более подробную информацию о функциях, чем в предшествующих работах и открывает возможность для использования такой информации в будущих исследования, связанных с выводом типов или другого анализа кода на Python.

Структура работы

В главе 1 представлен обзор существующих работ в области вывода типов в динамически типизированных языках.

В главе 2 описаны методика и результаты сравнительного анализа существующих подходов с учетом статического вывода типов, используемого в IDE.

В главе 3 описаны технические подробности реализации инструмента для вывода типов, интегрированного в IDE PyCharm.

В заключении представлены анализ результатов работы и возможные пути дальнейшего развития исследований.

1. Обзор литературы

1.1. Автоматический вывод типов в языках с динамической типизацией

В ранних работах по выводу типов в динамически типизированных языках JavaScript и Python [26, 16] используются только статические методы для вывода типов. Это значит, что такие методы имеют заранее заданное множество правил, позволяющее им определить возможные типы для конкретной переменной. Эти методы предсказывают типы, всегда являющиеся корректными, однако они могут вывести тип только для малого числа примеров, а также не могут учесть структурную информацию о переменных. Тем не менее в редакторах кода и интегрированных средах разработки используется именно статический анализ, так как для их корректной работы важно делать верные выводы о существующих в программе типах.

Для того чтобы выводить типы переменных, которые нельзя точно определить статическими анализаторами, появились методы, основанные на машинном обучении. Такие методы можно разделить на две группы по размеру множества предсказываемых типов: методы с закрытым и открытым словарем. Методы с закрытым словарем решают задачу классификации и предсказывают типы из заранее заданного множества, например, 1000 наиболее часто встречающихся типов в репозиториях с открытым исходным кодом [6]. Методы с открытым словарем не ограничивают размер множества предсказываемых типов, что позволяет им предсказывать неизвестные на этапе обучения пользовательские типы. Большинство ранних подходов к выводу типов с помощью машинного обучения предсказывают типы из закрытого словаря. Раичев и др. создали инструмент для вывода типов на JavaScript JSNice [25]. Авторы используют вероятностные случайные поля (CRF, Conditional Random Fields) для определения типов у параметров функций. Этот инструмент учитывает релевантные для вывода типов отношения между структурными элементами в коде с помощью сетей зависимостей и успешно предсказывает аннотации типов для параметров функций. Однако данный подход часто не может предсказать какой-либо тип для па-

раметра (например, если параметр не используется в функции), при этом предсказания совершаются из закрытого словаря. Эти факторы негативно влияют на точность полученных предсказаний.

В работе Хеллендорна и др. [7] было предложено рассматривать задачу вывода типов как одну из задач обработки естественного языка, а именно задачу определения части речи или определения именованных сущностей, и пытаться вывести тип идентификаторов, основываясь на контексте кода, в котором они находятся. На основе этих идей был разработан инструмент DeepTyper, использующий для предсказаний методы глубокого обучения. Инструмент реализует двустороннюю рекуррентную нейронную сеть, основанную на длинных цепях с краткосрочной памятью (LSTM, Long Short Term Memory) [13], для предсказания типов переменных, функций и параметров. Эта сеть рассматривает код как последовательность отдельных слов и знаков препинания (токенов) и преобразует такую последовательность в последовательность векторов типов. Каждый элемент вектора представляет собой вероятность конкретного токена иметь определенный тип. Такая архитектура не учитывает возможность переменной встречаться в разных местах в программе и предсказывает тип для каждого вхождения независимо. Для обработки таких сценариев было предложено использовать так называемый слой согласованности, усредняющий представления токена по всем его вхождениям, что улучшает результативность вывода типов, но является лишь частичным решением проблемы.

Рассматривать код как последовательность токенов — достаточно распространенный подход [4, 10], позволяющий использовать подходы из области обработки естественного языка и легко переиспользовать модель для разных языков программирования. Минусом подхода является то, что он сильно ограничивает информацию, которую модель может использовать. В качестве другой информации для моделей может использоваться, например, информация на естественном языке [6, 33, 35], информация о взаимосвязях между идентификаторами [15] или информация о потоках данных [35].

Малик и др. предложили использовать в моделях не контекстуальную информацию, а информацию на естественном языке и реализовали модель

NL2Type [17]. В этом подходе также выбрана сеть LSTM для обучения, но в качестве входных данных в ней используются имена функций, параметров и комментарии к ним. Предложенный подход показывает себя лучше, чем рассмотренные подходы JSNice, который использует только взаимосвязи между идентификаторами, и DeepTyper, использующий только последовательности токенов в качестве входных данных, что позволяет сделать вывод о полезности информации на естественном языке для вывода типов. Однако NL2Type не использует контекстуальную информацию, содержащуюся в близких друг к другу элементах в исходном коде, и задействует малое количество информации на естественном языке. Так, 20% функций в датасете NL2Type не содержат комментариев, и в таких случаях модель полагается только на имена функций и параметров.

Дальнейшие исследования, использующие глубокое обучение, сочетают информацию на естественном языке с контекстуальной информацией. Бун и др. [6] предложили использовать возвращаемые выражения функций в качестве признаков в дополнение к комментариям и именам функций и параметров и реализовали модель DLTPy. Авторы выбрали язык Python для вывода типов и сравнивали свой подход с подходом NL2Type, что позволило показать, что модели естественного языка могут быть использованы для различных динамически типизированных языков. В работе выяснилось, что комментарии играют значительную роль в выводе типов в Python, однако в качестве датасета рассматривались только функции, у которых присутствуют комментарии. В то же время в других работах [34] было показано, что 60% функций на Python в проектах с открытым исходным кодом не содержат комментарии.

Авторы модели TypeWriter [34] предложили использовать большее количество информации для вывода типов. Для обучения своей модели они собирают те же признаки, что и в модели DLTPy, и добавляют к ним все вхождения каждого параметра функции в выражения внутри функции. Другим нововведением данной работы было использование для предсказания типов в функции всех доступных типов в файле, внутри которого находится функция. Для того чтобы получить эти типы, модель собирает данные о классах, объявленных в файле, и все импортирования модулей внутри этого

файла. После предсказания типов в работе предложено использовать статический анализ для проверки корректности. Подход заключается в переборе вариантов подстановки типов внутри функции и последующем запуске статического анализатора. Это позволяет не допустить ошибок в процессе вывода типов, но сильно увеличивает время, затрачиваемое на каждое предсказание. Авторы показывают, что данные о доступных типах не играют значительную роль в выводе, хотя и позволяют увеличить точность модели. Это может происходить из-за недостаточно глубокого анализа импортирований и отсутствия обработки переименований типов (например, *from pandas import DataFrame as df*).

Несмотря на показанные результаты, все вышеупомянутые работы рассматривают для вывода типов закрытый словарь. Это налагает на модели серьезные ограничения в предсказательной способности для более редких типов. Так, типы, которые встречаются хотя бы 100 раз, покрывают только 70% всех аннотаций, а остальные 30% аннотаций встречаются менее 100 раз в датасете из проектов на Python, состоящем из 120,000 файлов [34].

Более поздние работы посвящены выводу типов с использованием открытого словаря. Вей и др. [15] предложили использовать для представления кода графовые нейронные сети. В своей работе LAMBDANET они строят граф типовых зависимостей, который представляет собой граф с гиперребрами, где вершинами являются объединение множества типов, доступных в проекте, и множество из 100 наиболее часто встречающихся типов в датасете, а ребрами являются взаимоотношения между ними. Такие ребра делятся на 2 категории – логические ребра, например, ребро $SubType(\alpha, \beta)$, которое показывает, что тип α является подтипом β, и контекстуальные ребра, например, ребро $NameSimilar(\alpha, \beta)$, показывающее, что имена типов α и β пересекаются по под-токенам. Такой подход, с одной стороны, позволяет учесть и информацию на естественном языке, и логическую информацию, и, с другой стороны, позволяет учесть в предсказаниях как часто встречающиеся типы, так и определенные пользователем в каждом конкретном проекте.

Typilus, подход, предложенный Алламанисом и др. [35], использует для обучения метод, называющийся обучение по сходству (DSL, deep similarity

learning). Этот подход используется, например, в задачах компьютерного зрения для распознавания различий между лицами людей [3]. Аналогично этому методу Typilus обучает графовую нейронную сеть для поиска различий и сходств между разными типами и строит пространство типов, представляющее собой d-мерное пространство, где точка представляет собой какой-то тип. Затем, для переданных программой функций и параметров вычисляется соответствующая им точка в этом пространстве, и с помощью алгоритма поиска k ближайших соседей [5] вычисляются наиболее вероятные типы.

Графовые подходы, описанные выше, требуют для своего анализа построения сложных графовых структур, а Typilus также проводит анализ потока данных, что еще больше увеличивает время, требуемое для вывода типов. Помимо этого, исследователи сделали предположение [2], что графовые сети по мере роста числа слоев в них сжимают в представление вершины экспоненциальное количество информации, из-за чего они не могут уловить взаимодействия между элементами на долгих дистанциях. Несмотря на эти недостатки, обе модели могут предсказывать типы из неограниченного словаря, что позволяет улучшить результаты для типов, определенных пользователем.

В более поздней работе Type4Py [33] авторы предлагают совместить использование DSL, открывающее возможность выводить новые типы, с рекуррентными нейронными сетями, что в совокупности дает быстрый анализ кода и высокую точность даже на неизвестных заранее типах. В предложенной модели используются те же признаки, что и в модели TypeWriter, однако производится более глубокий анализ импортирований. Для такого анализа строится граф зависимостей, в который попадают все импортирования и их транзитивные зависимости. Затем с использованием полученных графов строится ограниченный словарь видимых типов. Далее, этот словарь используется для кодирования типа идентификатора (параметра, функции или переменной) в вектор, который будет использоваться для предсказания. Было показано, что видимые типы значительно влияют на предсказательную способность модели.

Модели, основанные на DSL, в большинстве строят в том или ином виде

пространство типов, в котором совершают поиск ближайших соседей для репрезентации текущего типа. Такой подход сильно увеличивает результативность работы метода, но в зависимости от размеров пространства может требовать от сотен мегабайт до нескольких гигабайт дискового места. Такие объемы памяти мало влияют на процесс обучения и подсчета метрик, так как они в основном проводятся в облачных хранилищах, однако в случае использования модели на локальном устройстве они могут создавать серьезные неудобства.

Стоит отметить, что большинство представленных работ рассматривает задачу вывода типов в динамически типизированных языках в следующей постановке: модели на вход дается множество файлов с отсутствующими или частично отсутствующими аннотациями типов, и модель предсказывает типы для неаннотированных идентификаторов. В такой постановке в качестве единственного учитываемого результата берутся метрики, отображающие вероятность правильного типа оказаться среди первых n предсказаний [35, 6, 34], или, в некоторых работах, среднеобратный ранг (MRR, Mean Reciprocal Rank) [33], поощряющий не только нахождение правильного предсказания на первом месте, но и его нахождение среди первых нескольких предложений. Однако для практических применений моделей для вывода типов кроме точности важны и другие показатели, например, время предсказания или объём требуемой памяти. Помимо этого, должна быть реализована инфраструктура, позволяющая использовать предсказания модели в процессе разработки. Так, авторы Type4Py разработали расширение для редактора исходного кода Visual Studio Code [36], предоставляющее веб-сервер для отправки файлов на языке Python, в ответ на которые выдается предсказания типов идентификаторов. Это позволяет решить проблему с объёмом памяти, однако требует постоянного доступа к сети и пересылки файлов, которые могут содержать, например, коммерческую тайну.

1.2. Выводы

По результатам анализа работ в области вывода типов в динамических типизированных языках были сделаны следующие выводы:

- Среди существующих подходов к выводу типов в динамических типизированных языках стоит выделить методы, использующие машинное обучение [7, 33, 34, 6, 35]. Эти методы являются вероятностными, и их предсказания не всегда являются корректными, в отличие от предсказаний статических анализаторов. Однако преимущество таких методов в том, что они могут использовать большее количество информации, например, имена параметров и функций или вхождения параметра в тело функции. Эта информация позволяет делать правильные предсказания во многих случаях, когда статический анализ не может вывести тип.

- Среди методов машинного обучения превалируют архитектуры, основанные на графовых и рекуррентных нейронных сетях. Графовые сети представляют код в виде графов различных отношений между идентификаторами в коде. Это позволяет явно учитывать важные взаимосвязи между идентификаторами, однако такой подход требует более сложных вычислений на стадии обработки данных и при этом демонстрирует меньшие результаты по сравнению с рекуррентными сетями. Рекуррентные сети в свою очередь не страдают от экспоненциального роста количества хранящейся в одной вершине сети информации и показывают лучшие результаты.

- В задаче вывода типов могут использоваться разные типы словарей: открытые и закрытые. В случае закрытого словаря модель предсказывает тип из заранее заданного множества типов, а модели с открытым словарем могут предсказывать неограниченное множество типов. В большинстве моделей, рассматривающих открытый словарь, строится пространство типов, что увеличивает количество затрачиваемых ресурсов и ограничивает варианты применения этих моделей на практике.

- Существующие работы уделяют меньшее внимание вариантам применения предложенных моделей на практике. Так, основной способ применения модели – восстановить пропущенные аннотации внутри целого проекта. Помимо этого, есть и другие практические задачи, в которых может пригодиться вывод типов с помощью моделей машинного обучения, например, улучшение анализа кода в IDE или автодополнение.

- Предыдущий пункт затрагивает процесс разработки в IDE, и лишь немногие работы хоть как-то касаются этого вопроса. При этом в рамках разработки в IDE не всегда важна абсолютная точность модели, и более важной может оказаться метрика, показывающая, как хорошо модель предсказывает типы, которые не может предсказать интегрированный в IDE статический анализатор.

2. Сравнительный анализ существующих подходов с учетом статического вывода типов, используемого в IDE

Для сравнения существующих подходов к выводу типов на Python с учетом анализатора, используемого в IDE, необходимо для каждого проекта в тестовой выборке узнать, какие типы в этом проекте может вывести IDE. Для отдельно открытого в графическом режиме проекта эта задача не представляет трудностей – разработчик может выделить необходимое место для вставки типа идентификатора, и IDE предложит тип, который может вывести, либо не предложит ничего, если о данном идентификаторе недостаточно информации. Однако в случае подсчета результатов вывода типа на большом корпусе проектов возникает проблема автоматизации и ускорения этого процесса. В этой главе описан предложенный подход к автоматизации процесса сбора данных о типах в IDE PyCharm и результаты сравнения существующих подходов к выводу типов с учетом типов, выведенных с помощью PyCharm.

2.1. Анализ вывода типов в IDE

В качестве IDE была выбрана среда PyCharm, разработанная компанией JetBrains. PyCharm является наиболее популярной средой разработки на Python [23]. Данная IDE использует статический анализ для вывода типов и обладает обширной поддержкой плагинов, что позволит в будущем использовать модели машинного обучения для улучшения качества вывода типов. Алгоритм вывода типов в PyCharm полагается на внутренние зависимости внутри проекта и внешние зависимости, которые представляют собой используемые в данном проекте библиотеки (например, *PyTorch* или *pandas*).

В данном разделе представлен алгоритм, позволяющий получить типы, выведенные с помощью PyCharm, в консольном режиме. Схема работы этого алгоритма представлена на рисунке 2.1.

Подключение к проекту интерпретатора Python предполагает загрузку

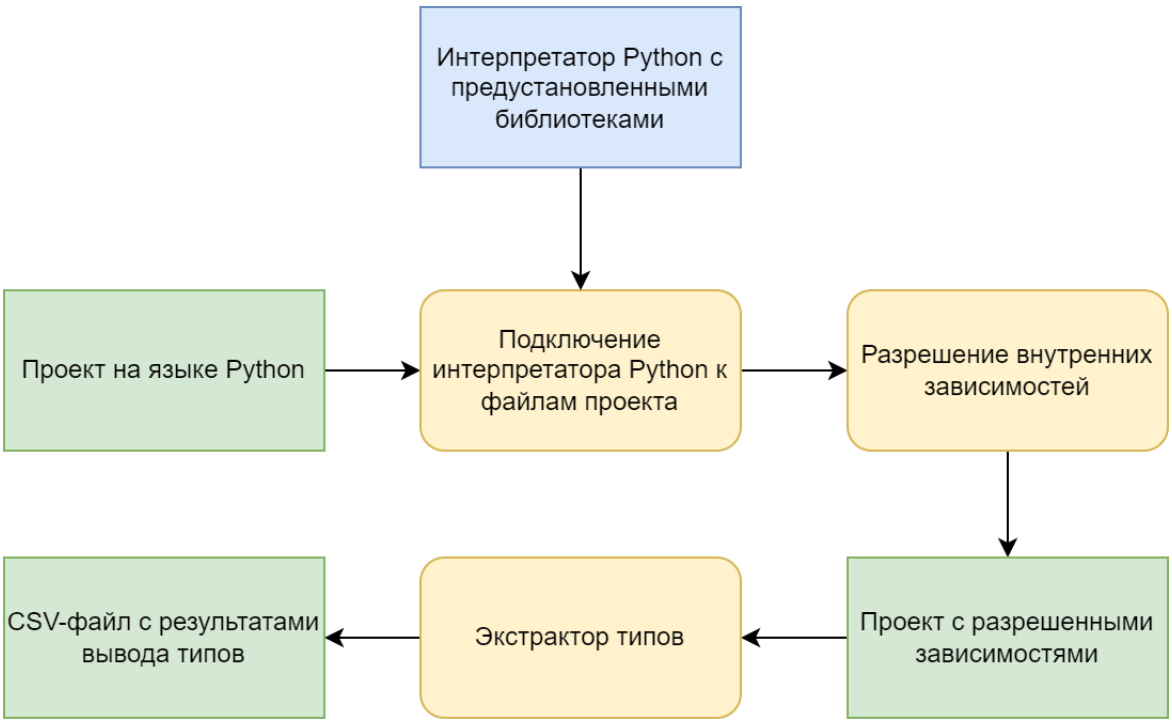


Рис. 2.1: Реализованный алгоритм для вывода типов в консольном режиме в PyCharm.

и установку внешних библиотек для присутствующих в датасете проектов. Для этой задачи существует несколько путей решения. Первый вариант — устанавливать внешние зависимости для каждого проекта в свой независимый интерпретатор Python. Это позволит увеличить точность вывода типов, так как зависимости в разных проектах не будут противоречить друг другу по версиям. Второй вариант — заранее установить ограниченный список библиотек, которые будут использоваться для разрешения зависимостей во всех проектах. Этот подход теряет в точности по сравнению с первым, но является более простым в реализации.

Для выбора между этими альтернативами было замечено следующее: при открытии проекта в первый раз и установке интерпретатора PyCharm необходимо проиндексировать библиотеки, установленные в текущем интерпретаторе, для проведения статического анализа. После этого PyCharm сохраняет результаты индексации во внутренний кэш. Это значит, что при всех последующих открытиях проекта PyCharm не тратит время на индексацию и читает данные из кэша. Последнее верно не только для нескольких открытий одного и того же проекта, но и для открытий разных проектов,

в каждом из которых используется один и тот же интерпретатор. Таким образом, второй подход является не только более простым в реализации, но и гораздо менее времязатратным. Эмпирические наблюдения позволили показать, что время открытия проекта в первый раз и настройка интерпретатора занимает около 3 минут в среднем среди выбранных 100 проектов, а все последующие около 1 минуты. Ввиду того, что размеры датасетов для обучения моделей исчисляются несколькими тысячами проектов, то при выборе первого подхода настройка проектов займет на несколько дней больше, чем при выборе второго подхода.

Вышеописанное позволило сделать выбор в пользу установки ограниченного списка библиотек заранее. Для этого 500 наиболее популярных библиотек на Python, согласно данным Top PyPi Packages [44], были установлены в Conda-окружение.

Разрешение внутренних зависимостей предполагает разметку в проекте корней модулей (source roots). Такие корни модулей показывают PyCharm, начиная с каких папок в проекте необходимо искать импортирования. Эта задача решается разработчиком проекта в графическом интерфейсе, однако в общем случае решить ее трудно. Например, в файле может быть объявлена директива *from utils import Logger*, и заранее неизвестно, где именно находится требуемый модуль *utils*. Обычно структура проекта достаточно проста, и необходимый модуль лежит либо в той же директории (тогда корень модуля указывать не нужно), либо в какой-то директории, лежащей на пути вверх по иерархии от текущей директории к корню.

В итоге для разрешения внутренних зависимостей в рамках одного проекта был предложен следующий алгоритм: для каждой директивы вида *from foo.bar import Logger* или *import foo.bar* в файле производится поиск в иерархии директорий, начиная с этого файла, вверх к корню проекта. Поиск останавливается, когда находится такая директория, от которой существует путь вида foo/bar или $foo/bar.py$. Возможный результат работы такого алгоритма показан на рисунке 2.2.

Для реализации процесса разрешения внутренних зависимостей в проекте был написан консольный плагин без графического интерфейса (headless), который обходит все файлы на языке Python в проекте и для каждого файла

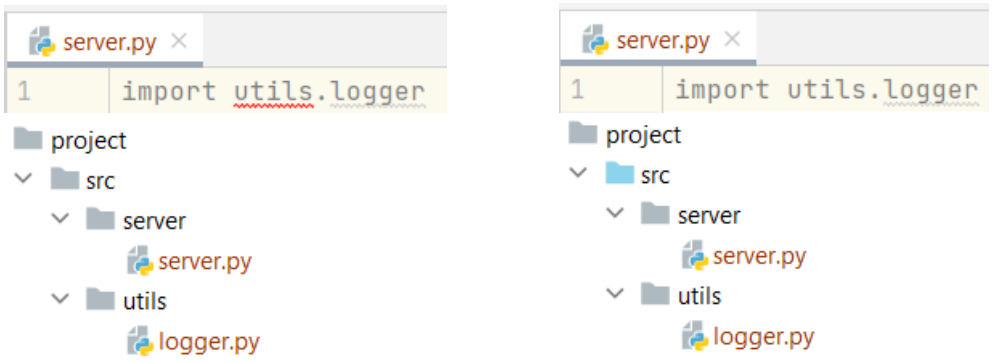


Рис. 2.2: Иерархия проекта до указания корней модулей (слева) и после (справа). Директиву *import utils.logger* невозможно разрешить без указания корней модулей. Директива была разрешена после указания корня модуля *src*. Голубым цветом указаны полученные корни модулей.

выполняет вышеописанный алгоритм. Это, в сочетании с предустановленными внешними библиотеками, позволило разрешить более 85% зависимостей в проектах.

После настройки проекта извлекаются типы, выведенные PyCharm для каждого проекта. Поскольку при наличии аннотаций в файле задача вывода типов не представляет интереса, из каждого проекта предварительно были удалены аннотации типов с помощью библиотеки *strip-hints*[1]. Для последующего получения выведенных типов был реализован аналогичный консольный плагин, который при обходе файла обрабатывает все объявленные в нем функции и для каждой функции выводит тип возвращаемого значения и типы всех аргументов. Полученные данные записываются в CSV-файл для упрощения последующих обработки и анализа, так как большинство реализованных решений для вывода типов получают на вход и отдают на выход файлы этого формата. Для каждого примера (тип возвращаемого значения или аргумента) в этом файле хранятся путь к файлу, имя функции (аргумента), вид примера (аргумент или функция), и сам тип. Аннотации Any и None для функций и Any для параметров не сохраняются, поскольку такие аннотации не представляют ценности для вывода типов.

Отметим, что полученная архитектура плагина позволяет получать результаты вывода типов в PyCharm без запуска IDE, то есть в консольном ре-

[1] `https://pypi.org/project/strip-hints/`

жиме. Для работы плагина требуется только существующий интерпретатор Python. Это позволяет автоматизировать подсчет результатов и значительно упростить выполнение кода, связанного с IDE, на удаленном сервере.

2.2. Сравнение с учетом вывода типов в PyCharm

Для сравнения результатов вывода типов был взят датасет, представленный в статье ManyTypes4Py [18]. Отличительной особенностью данного датасета по сравнению с другими [24] является то, что все проекты в датасете используют статический анализатор mypy [41] для проверки корректности типов. Это, кроме корректности указанных аннотаций, значит то, что эти аннотации присутствуют в файлах датасета. Алламанис и др. [1] показали, что наличие дубликатов файлов в датасете негативно влияет на результаты моделей машинного обучения, поэтому весь датасет дедуплицирован при помощи поиска похожих документов с использованием меры TF-IDF [30] и последующего поиска ближайших соседей для идентификации кандидатов в дубликаты. Датасет содержит 2.1 миллиона функций и 3.9 миллиона аргументов, из которых аннотированы типами 325 тысяч (15.5%) и 480 тысяч (12.2%), соответственно. Датасет разбит на обучающую, валидационную и тестовую выборки в соотношении 70 к 20 к 10, соответственно.

Для сравнения были выбраны две модели, основанные на рекуррентных нейронных сетях — DLTPy [6] и Type4Py [33], и одна модель, основанная на графовых нейронных сетях — Typilus [35]. Выбор Typilus обусловлен наличием открытого исходного кода и тем, что другие рассмотренные графовые модели являются специфичными для JavaScript. Выбор DLTPy и Type4Py обусловлен наличием открытого исходного кода для каждой из моделей, высоким качеством работы Type4Py, заявленным в оригинальной статье, и тем, что DLTPy является более простой и легковесной моделью по сравнению с Type4Py. Сравнение этих моделей с учетом вывода типов в PyCharm позволит показать, какую роль предложенные в Type4Py улучшения будут играть на практике, и насколько они важны в сравнении с усложнением модели.

Поскольку целью данной работы является интеграция моделей машин-

ного обучения для вывода типов в IDE, важно рассмотреть метрику, учитывающую характеристики вывода типов в PyCharm. Из-за того, что предсказания IDE основываются на статическом анализе и всегда корректны (в противном случае будет предложен тип Any), из датасета для тестирования моделей требуется убрать все примеры, для которых IDE выдала предсказание, отличное от Any. Так, была предложена следующая метрика:

$$Acc'(Model) = \frac{Correct_{Model} - Correct_{PyCharm \cap Model}}{N_{Examples} - Correct_{PyCharm}} \quad (1)$$

Здесь $Correct_{Model}$ ($Correct_{PyCharm}$) — это количество примеров, на которых модель (PyCharm) ответила правильно, $Correct_{PyCharm \cap Model}$ — количество примеров, на которых и модель, и PyCharm отработали правильно, $N_{Examples}$ – общее количество примеров, для которых проводилось тестирование. Примерами могут быть как все данные (типы возвращаемых значений функций, аргументов функций), так и одна разновидность примеров, например, типы аргументов.

Результаты вывода типов для рассматриваемых моделей до и после удаления примеров, предсказанных PyCharm, представлены в таблицах 2.1 и 2.2 соответственно. В этих таблицах точность – это доля предсказаний модели, являющихся правильными, точность (returns) и точность (args) – это точности, посчитанные для датасета, содержащего только типы возвращаемых значений и типы параметров соответственно. Аналогично Acc' (returns) и Acc' (args) – это метрика Acc', посчитанная для датасета, содержащего только типы возвращаемых значений и типы параметров соответственно.

Таблица 2.1: Точности выводимых типов для рассматриваемых моделей на всех примерах.

Модель	Точность	Точность (returns)	Точность (args)
PyCharm	0.157	0.256	0.115
DLTPy	0.510	0.404	0.541
Typilus	0.457	0.385	0.487
Type4Py	0.616	0.513	0.687

Полученные результаты показывают, что разрыв в точности вывода типов всех примеров между моделями DLTPy и Type4Py сократился с 20.8%

Таблица 2.2: Точности выводимых типов для рассматриваемых моделей на примерах, оставшихся после учета вывода типов PyCharm.

Модель	Acc'	Acc' (returns)	Acc' (args)
DLTPy	0.482	0.327	0.529
Typilus	0.386	0.261	0.431
Type4Py	0.561	0.350	0.684

до 16.4%, при этом для типов возвращаемых значений этот разрыв сократился с 27% до 7%, а для типов параметров увеличился лишь на 2.3%. Модель Typilus, в свою очередь, после учета вывода типов PyCharm стала еще больше отставать от двух других моделей, увеличив разрыв с DLTPy с 11.6% до 24.9%. Стоит отметить, что даже после учета результатов вывода типов в PyCharm модель Type4Py показывает наилучшие результаты среди выбранных для сравнения подходов.

Помимо сравнения моделей с учетом вывода типов в PyCharm было проведено сравнение объема ресурсов, используемых моделями. Модели Typilus и Type4Py строят пространство типов, в котором затем совершают поиск ближайших соседей для вывода типов конкретного примера, тогда как DLTPy решает задачу классификации и не требует построения лишних структур. Помимо этого модели имеют разные архитектуры и разное число тренируемых параметров, что также влияет на объём требуемой памяти. Эти различия отражены в таблице 2.3.

Таблица 2.3: Память, требуемая для работы моделей.

Модель	Размер модели (МБ)	Размер пространства типов (МБ)
DLTPy	6	-
Typilus	9	1000
Type4Py	67	>1500

По результатам сравнения качества и требуемой памяти была выделена модель DLTPy. Эта модель отличается от других низким объёмом требуемой памяти, что при этом не сказывается негативным образом на её точности даже после удаления типов, выведенных с помощью PyCharm. При этом маленький объём памяти позволяет эффективно интегрировать данную модель во внутреннее устройство PyCharm и не создаст пользователям

дополнительных затруднений в случае её использования, в то время как более результативная модель Type4Py требует для своего использования более 1.5 гигабайт памяти.

2.3. Выводы

В данной главе представлены результаты сравнения современных подходов к выводу типов с использованием машинного обучения на Python с учетом вывода типов в IDE PyCharm. Было показано, что наиболее результативная до учета PyCharm модель Type4Py [33], основанная на рекуррентных сетях, всё ещё является наиболее результативной и после учета PyCharm. Более простая и легковесная модель DLTPy [6], также основанная на рекуррентных сетях, после учета PyCharm уменьшила разрыв с Type4Py. Модель Typilus [35], основанная на графовых сетях, в данном сравнении не смогла конкурировать с другими подходами. Это может происходить как из-за экспоненциального роста числа информации, которую графовые модели пытаются запомнить в одной вершине [2], так и из-за того, что пространство типов в Typilus меньше по размеру, чем аналогичное пространство в модели Type4Py.

Для того чтобы сравнить модели с учетом PyCharm, были посчитаны результаты вывода типов в IDE PyCharm — точность 25.6% для типов возвращаемых значений и 11.5% для типов параметров. PyCharm не удается точно предсказывать типы параметров функций, поскольку о них в коде доступно меньше информации по сравнению с количеством информации о возвращаемом значении функции. Например, зная, что в возвращаемых выражениях функции вызывается метод из какой-то библиотеки, то можно точно вывести тип возвращаемого значения в случае, если эта библиотека содержит аннотации типов.

Для подсчета результатов работы IDE PyCharm был реализован headless плагин, позволяющий в автоматизированном режиме открывать проекты на Python и разрешать в них внешние и внутренние зависимости. Для разрешения внешних зависимостей были предустановлены наиболее популярные библиотеки на Python [44]. Для разрешения внутренних зависимостей

в автоматизированном режиме в проекте расставляются корни модулей, используемые PyCharm для индексации зависимостей внутри проекта. Предложенный алгоритм способен разрешить более 85% импортирований и может в консольном режиме получать результаты вывода типов в PyCharm в формате CSV-файлов.

3. MLTypes4PyCharm: плагин для вывода типов в PyCharm

В главе 2 был проведен сравнительный анализ существующих подходов к выводу типов в Python. Анализ показал, что наиболее подходящей для интеграции в PyCharm моделью является модель DLTPy [6]. Эта модель более легковесная по сравнению с Type4Py [33] и Typilus [35], что позволяет удобно распространять ее либо в виде плагина, либо встроенной в IDE. В то же время, качество DLTPy не сильно отстает от более результативных моделей даже после учета вывода типов в PyCharm.

Авторы модели Type4Py распространяют свою модель в виде плагина для Visual Studio Code [36]. У предложенного ими решения есть ряд недостатков: разработчикам не рекомендуется менять файл после однократного запуска модели, разработчикам предлагается добавлять только сами аннотации типов, модель распространяется в виде веб-сервера, запрос к которому представляет собой исходный файл на Python. Эти факторы сильно ограничивают применимость такого подхода на практике, ведь во время разработки в IDE пользователь может хотеть пользоваться преимуществами IDE, не проставляя вручную аннотации типов, а отправка на веб-сервер файлов из проектов может быть невозможна из-за приватности данных. В связи с вышеописанным, к разрабатываемому инструменту для вывода типов в PyCharm с использованием машинного обучения поставлены следующие требования:

- Позволять разработчикам использовать инструмент во время написания кода, то есть вызывать его в любое время, а не только после окончания работы над всем, кроме аннотаций типов.
- Улучшать работу IDE даже в случае, когда разработчик не добавляет аннотации типов.
- Располагаться на локальном устройстве и не требовать доступа к сети.

В этой главе описывается устройство инструмента MLTypes4PyCharm, реализованного для решения этих задач и распространяемого в виде плагина

для IDE PyCharm.

3.1. Устройство MLTypes4PyCharm

Архитектура работы инструмента представлена на рисунке 3.1.

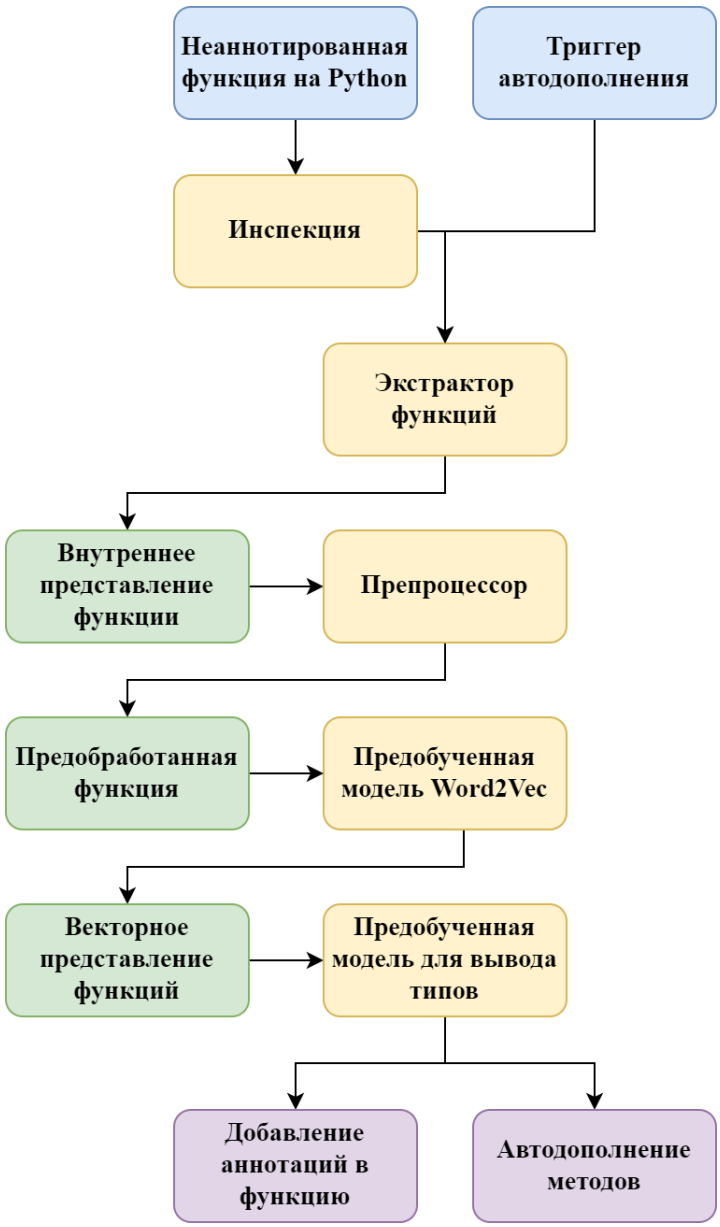


Рис. 3.1: Архитектура плагина для вывода типов в процессе разработки.

Функции без аннотаций типов возвращаемого значения или типов параметров обрабатываются с использованием средств PyCharm и передаются в инструмент для сбора данных. Затем полученные данные в виде внутреннего представления функций передаются в обработчик, использую-

щий библиотеку Stanford CoreNLP [28], для дальнейшей очистки: удаления стоп-слов и токенизации. После этого обработанная функция переводится в представление в виде численного вектора. Для векторизации информации об именах функций используются представления токенов, полученные из предобученной модели Word2Vec [8]. Этот подход используется в предложенных ранее решениях и обуславливается тем, что векторизованные представления должны отражать семантическую информацию, содержащуюся в функциях. На уровне слов это же делает Word2Vec – похожим словам он сопоставляет близкие в пространстве векторы. Векторизованные представления передаются в предобученную модель, которая предсказывает несколько наиболее вероятных типов. Предсказанные типы используются для решения задач добавления аннотаций к функции или могут использоваться в других задачах по анализу кода в PyCharm.

Для вызова моделей машинного обучения существует множество библиотек [21, 31, 27, 40, 42]. Большинство библиотек используются как для обучения модели, так и для получения предсказаний. Такой избыток функциональности предполагает наличие в модели излишней информации, не необходимой на момент вызова, и увеличивает количество ресурсов, потребляемых моделью. Среди библиотек, специализирующихся на запуске моделей, выделяют библиотеки, использующие универсальный формат представления Open Neural Network Exchange (ONNX) [40, 42, 14]. Большинство таких библиотек разработаны на C++, и некоторые из них предоставляют программный интерфейс (API, application programming interface) для разработки на языках Java и Kotlin. Для работы с проектами на этих языках запускается скомпилированная библиотека с помощью Java Native Interface, что может ухудшить производительность. Исходя из описанных факторов, для вызова модели была использована библиотека KInference [14], разработанная на чистом языке Kotlin, целью которой является запуск моделей машинного обучения, включая нейронные сети, на языках Java и Kotlin.

3.2. Сбор данных

В предыдущих работах, рассматривающих Python для вывода типов или других задач, связанных с анализом кода, для сбора данных используются библиотеки [38, 43, 39], представляющие код в виде абстрактных синтаксических деревьев (AST, abstract syntax tree). Этот метод прост в использовании и позволяет получать все необходимые данные, однако такие библиотеки не располагают API для их использования на других языках. PyCharm предоставляет более мощный инструмент для работы с AST – программно-структурный интерфейс (PSI, program structure interface), работать с которым можно на чистом языке Koltin. PSI позволяет работать с AST файлов на Python в похожей манере, при этом обладает всеми преимуществами работы с IDE, например, анализом ссылок и использований функций.

В этой работе был реализован headless плагин, работающий с PSI и собирающий данные о функциях на PyCharm. Эти данные затем используются во встроенной в MLTypes4PyCharm модели, и могут использоваться в других исследованиях по выводу типов или анализу кода на Python. В текущей версии плагин получает из функции следующую информацию:

- имя функции и всех ее аргументов;
- комментарий к функции и отдельно комментарии к возвращаемому значению и всем аргументам функции;
- типы возвращаемого значения функции и всех ее аргументов;
- предсказанные с помощью внутренних алгоритмов PyCharm типы возвращаемого значения функции и всех ее аргументов;

Помимо этого, как было предложено в работе TypeWriter [34], плагин получает из каждого файла доступные в нем типы, а именно:

- все объявленные в файле классы;
- все импортирования в этом файле, в том числе импортирования вида *from module import **, раскрываются и собираются все классы, доступные в импортируемом модуле.

Стоит отметить, что на данный момент более глубокий анализ импортирований не проводится, поскольку в рамках данной работы эта задача не является приоритетной, но такой анализ возможен.

3.3. Работа плагина MLTypes4PyCharm

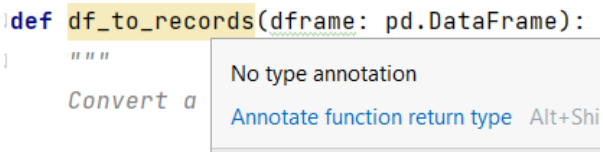
Для использования плагина MLTypes4PyCharm в задаче добавления аннотаций типов в процессе разработки необходимо:

- сообщать разработчикам об отсутствии в их файлах аннотаций типов функций и/или параметров;
- предоставлять разработчикам возможность добавить аннотации типов к функциям и/или параметрам.

Для выполнения этих задач были реализованы две точки расширения PyCharm — инспекция и исправление. Инспекция — это средство, позволяющее обрабатывать произвольную вершину PSI и определять, является ли данная вершина проблемной в рамках данной инспекции (например, есть инспекция, выделяющая неиспользуемые импортирования). В случае наличия каких-то проблем инспекция позволяет выделить в коде область, за которую ответственна соответствующая вершина PSI, и передать эту вершину в специальный обработчик исправлений, при этом давая пользователю возможность выбрать, какой обработчик использовать, и необходимо ли его использовать. Обработчик исправлений в свою очередь получает на вход информацию о проблемной вершине и предлагает какие-либо действия для решения этой проблемы, которые могут затрагивать не только переданную вершину (например, удаление всех неиспользуемых импортирований, даже если на вход было передано только одно).

Для MLTypes4PyCharm была реализована инспекция, выделяющая в коде функции без аннотации типа возвращаемого значения и списки параметров функций, среди которых есть хотя бы один параметр без аннотации типа. Примеры работы инспекции представлены на рисунках 3.2, 3.3.

Обработчик исправлений в MLTypes4PyCharm совершает действия, описанные в разделе 3.1, а именно собирает данные о переданной функции (в

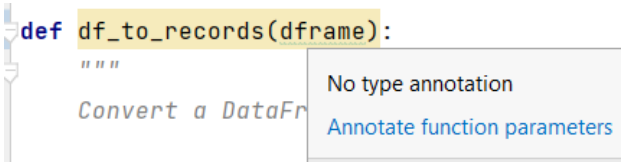


(a) Неаннотированная функция выделяется в коде.

```
def __init__(self, prefix: str):
    self.prefix = prefix
```

(b) Особые методы, в которых не требуется аннотация типа возвращаемого значения, не выделяются.

Рис. 3.2: Пример работы инспекции для функций без аннотаций типа возвращаемого значения.



(a) Список параметров, содержащий хотя бы один неаннотированный параметр, выделяется в коде.

```
def incr(self, key: str) -> None:
    """Increment a counter"""
    raise NotImplementedError()
```

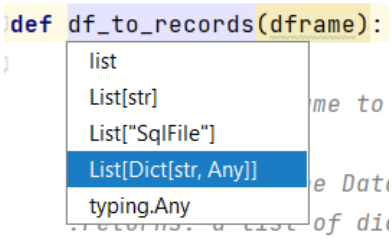
(b) Особые параметры, не требущие аннотации типов, не выделяются.

Рис. 3.3: Пример работы инспекции для параметров без аннотаций типов.

случае, когда передан список параметров – о функции, содержащей этот список), очищает эти данные, переводит в векторизованный вид и вызывает встроенную модель машинного обучения для получения предсказаний. В качестве исправления проблемы обработчик предлагает добавить аннотации типов к функции или списку параметров.

Для неаннотированных функций пользователю предлагается на выбор до 5 типов, предсказанных моделью, среди которых он может выбрать подходящий. Для списка параметров обработчик самостоятельно подставляет наиболее вероятные аннотации типов для каждого неаннотированного параметра. Такой подход обусловлен тем, что при наличии в списке хотя бы 3-х неаннотированных параметров пользователю приходилось бы выбирать среди 8 вариантов в случае 2 наиболее вероятных типов, предсказанных моделью, и среди более 27 вариантов, если учитывать хотя бы 3 предсказания модели.

Примеры работы обработчика исправлений представлены на рисунках 3.4, 3.5.



(a) Список предложеннных пользователю типов для функции с исходным типов *List[Dict[Str, Any]]*.

```
def df_to_records(dframe) -> \
        List[Dict[str, Any]]:
    """
    Convert a DataFrame to a set of
```

(b) Функция с добавленной аннотацией типа возвращаемого значения.

Рис. 3.4: Пример работы обработчика исправлений для функции без аннотации типа возвращаемого значения. Точное предсказание находится на четвертом месте в выдаче. Предсказанный на первом месте тип *list* также подходит, но является более общим, чем исходный.

```
def cum(
    df: DataFrame,
    operator: str,
    columns: Dict[str, str],
) -> DataFrame:
```

(a) Изначальная сигнатура функции.

```
def cum(
    df: pandas.DataFrame,
    operator: str,
    columns: pandas.DataFrame
) -> DataFrame:
```

(b) Предсказанные аннотации типов для параметров после удаления их из исходной сигнатуры.

Рис. 3.5: Пример работы обработчика исправлений для списка параметров. Предсказанный для параметра *columns* тип *pandas.DataFrame* не является правильным.

Для оценки применимости плагина на практике необходимо измерить время работы полученного решения и сравнить его с временем работы оригинальных реализаций моделей. В рамках этой работы было проведено сравнение двух наиболее критичных по времени операций — сбора данных и вызова модели — для следующих версий плагина:

- версия, напрямую вызывающая оригинальную реализацию модели DLTPy [6];
- предложенное в MLTypes4PyCharm решение.

Сравнение было проведено для выбранных случайным образом 10 функций в файлах из датасета ManyTypes4Py [18], содержащих от 20 до 200 строк кода на Python. Средние показатели времени работы представлены в таблице 3.1.

Версия плагина	Сбор данных, с	Вызов модели, с
Оригинальная реализация DLTPy	2.32	2.53
MLTypes4PyCharm	0.37	0.02

Таблица 3.1: Среднее время выполнения операций для функций в различных версиях плагина.

Полученные результаты показывают, что по сравнению с непосредственным вызовом оригинальной реализации модели DLTPy в предложенном плагине получено ускорение в 6.3 раз для сбора данных и в 125 раз для вызова модели. Такое ускорение позволяет использовать MLTypes4PyCharm как для добавления аннотаций типов в процессе разработки кода, так и для улучшения качества работы PyCharm в других задачах, связанных с анализом кода.

3.4. Применение MLTypes4PyCharm для автодополнения кода

Несмотря на то что разработчики не всегда добавляют аннотации типов, разработанный инструмент может улучшать работу IDE в фоновом режиме. Одним из таких улучшений является более точное предсказание

вариантов автодополнения методов и полей у переменных, являющихся параметрами функций, то есть предсказание возможных вариантов автодополнения в тех местах, где разработчиком совершается попытка вызвать метод у какого-либо параметра внутри функции. Для улучшения качества таких предсказаний предложен следующий алгоритм:

1. Получить место в коде, в котором была совершена попытка автодополнения, а именно вершину PSI.
2. С помощью полученной вершины PSI обратиться к параметру, у которого вызывается метод.
3. Собрать данные о функции, содержащей данный параметр, с помощью функциональности, описанной в разделе 3.2.
4. Получить несколько наиболее вероятных типов для данного параметра с помощью интегрированных моделей машинного обучения.
5. Получить все методы, объявленные в выведенных типах, с помощью функциональности PyCharm.
6. Добавить полученные методы в качестве вариантов автодополнения методов для исходного места в коде.

Поскольку наличие аннотаций типов позволяет PyCharm наверняка знать корректный вариант автодополнения, проекты для оценки качества алгоритма не должны содержать аннотации типов, либо иметь их в малом количестве. Поэтому для оценки полученного алгоритма был вручную собран датасет из 7 проектов на GitHub, содержащий более 5,951 позиций для автодополнения, в которых только 18% параметров содержат аннотации типов. Для сравнения были выбраны 3 версии автодополнения:

- PyCharm без дополнительных плагинов — такая конфигурация показывает, как работают алгоритмы, встроенные в IDE в данный момент.
- PyCharm с установленным плагином MLTypes4PyCharm, в который интегрирована модель DLTPy — такая конфигурация показывает, как работает плагин в текущей версии.

- PyCharm с установленным плагином MLTypes4PyCharm, в котором используются предсказания, полученные с помощью локального сервера модели Type4Py — такая конфигурация показывает, насколько хорошо работает автодополнение с более мощной моделью.

Чтобы оценить качество работы, был использован плагин evaluation-completion [2], разработанный компанией JetBrains. Этот плагин собирает места в коде, в которых необходимо вызвать автодополнение метода, и затем с помощью встроенных алгоритмов ранжирования, основанных на машинном обучении, упорядочивает предложенные варианты для автодополнения. В конце плагин подсчитывает метрики, среди которых были выбраны следующие:

- Found@1 — метрика, показывающая, насколько часто предсказание, находящееся на первом месте в выдаче, является правильным.
- Found@5 — метрика, показывающая, насколько часто правильное предсказание находится среди первых пяти позиций в выдаче.
- Recall (полнота) — метрика, показывающая, насколько часто правильное предсказание в принципе находится в выдаче.

Метрики Found@1 и Found@5 отображают возможности внутренних алгоритмов ранжирования правильно размещать корректные предсказания плагина MLTypes4PyCharm на первых местах в выдаче, поскольку разработчики чаще выбирают именно первый вариант для автодополнения, предложенный алгоритмом [20]. Улучшение этих метрик будет обозначать мгновенное улучшение качества автодополнения, видимое пользователям.

Метрика полноты отображает, насколько часто MLTypes4PyCharm предлагает алгоритмам ранжирования корректное предсказание, которое нужно показать пользователю. Улучшение этой метрики будет обозначать увеличение размеров обучающей выборки для алгоритмов ранжирования, что впоследствии может привести к улучшениям других метрик.

Поскольку плагин рассчитывает метрику для одного проекта, для получения итогового результата было взято средневзвешенное значение метрики

[2] https://github.com/JetBrains/intellij-community/tree/master/plugins/evaluation-plugin

по всем проектам. Например, для полноты этот результат рассчитывается по следующей формуле:

$$Recall_{weighed} = \frac{\sum\limits_{project} Recall(project) \times N_{examples}(project)}{\sum\limits_{project} N_{examples}(project)} \quad (2)$$

Здесь $Recall_{weighed}$ — это итоговое значение метрики, $Recall(project)$ и $N_{examples}(project)$ — значение метрики и количество примеров в одном проекте соответственно.

По результатам запуска плагина *evaluation-completion* и усреднения метрик были получены результаты, указанные в таблице 3.2.

Модель	Found@1	Found@5	Recall
PyCharm	0.391	0.580	0.650
MLTypes4PyCharm, DLTPy	0.360	0.572	0.679
MLTypes4PyCharm, Type4Py	0.375	0.591	0.722

Таблица 3.2: Усредненные результаты запуска плагина evaluation-completion для выбранных версий автодополнения.

Для оценки статистической значимости полученных результатов был проведен тест Вилкоксона [37]. Этот тест проверяет нулевую гипотезу о том, принадлежат ли две связанные выборки значений одному и тому же распределению, а именно проверяет, является ли разница поэлементных значений двух выборок симметричной относительно нуля. Для проведения теста были взяты выборки значений метрик для разных версий автодополнения, и посчитано p-значение теста Вилкоксона с помощью библиотеки *scipy.stats*[3]. В частности было получено, что улучшение полноты по сравнению с PyCharm у обеих версий является статистически значимым на уровне 0.01, при этом различия в значениях метрик Found@1 и Found@5 не являются значимыми.

[3] `https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.wilcoxon.html`

3.5. Выводы

В данной главе был представлен инструмент MLTypes4PyCharm — плагин для IDE PyCharm, позволяющий улучшать работу PyCharm с помощью информации о типах, полученной из моделей машинного обучения. Используя внутреннюю функциональность PyCharm, плагин способен выделять неаннотированные места в коде и предоставлять пользователю возможность добавить в эти места аннотации типов, выведенные при помощи моделью машинного обучения, представленной в работе DLTPy [6].

Функциональность сбора данных была реализована в виде headless плагина, который использует для анализа кода PSI — внутреннее представления кода в PyCharm. Этот плагин способен собирать как информацию о функциях на Python, используемую в существующих моделях вывода типов, основанных на машинном обучении, так и более сложную информацию, которую можно извлечь благодаря анализу кода в PyCharm, например, места вызовов функций в коде. Такой инструмент позволяет исследователям в перспективе извлекать из кода на Python большее количество данных и может улучшить как модели для вывода типов, так и модели, используемые для других задач, связанных с анализом кода.

MLTypes4PyCharm достигает ускорения в более чем 6 раз для сбора данных и более чем 100 раз для вызова модели по сравнению с вызовом оригинальной реализации модели DLTPy. Время работы плагина в среднем не превышает 0.4с от начала вызова до добавления аннотаций типов. Низкое время работы позволило использовать информацию, полученную из MLTypes4PyCharm для улучшения качества автодополнения кода в PyCharm. В главе был предложен алгоритм, использующий типы, выведенные с помощью MLTypes4PyCharm, для автодополнения методов и полей у переменных, являющихся параметрами функций. Было проведено сравнение алгоритмов автодополнения, встроенных в PyCharm, и MLTypes4PyCharm, показавшее, что метрика полноты выросла на 4.5% и 11% для интегрированных моделей DLTPy и Type4Py [33] соответственно.

Заключение

Главным результатом данной работы является плагин MLTypes4PyCharm для IDE PyCharm, использующий информацию, получаемую из моделей машинного обучения, для добавления аннотаций типов и для улучшения качества автодополнения в PyCharm. Для интегрированных моделей DLTPy и Type4Py метрика полноты в задаче автодополнения методов выросла на 4.5% и 11% соответственно. В отличие от других инструментов, позволяющих пользоваться моделями машинного обучения для вывода типов, MLTypes4PyCharm не требует подключения к сети и улучшает работу IDE PyCharm в реальном времени. Среднее время работы инструмента с моделью DLTPy не превышает 0.5с, что более чем в 100 раз меньше, чем вызов напрямую оригинальной реализации модели. Также в данной работе были получены следующие результаты:

- Проведено сравнение современных работ по выводу типов, основанных на машинном обучении, с учетом статического вывода типов, реализованного в IDE PyCharm. Было обнаружено, что более простая и легковесная модель DLTPy [6], после учета вывода типов в PyCharm на 20% уменьшила разрыв с более тяжелой и эффективной моделью Type4Py в точности вывода типов возвращаемых значений. Для извлечения типов, которые способен вывести PyCharm, был разработан консольный плагин, позволяющий в автоматизированном режиме совершать настройку проектов на Python.

- Реализован консольный плагин, собирающий данные о функциях на Python с помощью внутреннего представления PyCharm. Этот плагин может извлекать как уже используемую во многих работах информацию, так и еще не исследованные в современных подходах к выводу типов данные. Плагин реализован на языке Kotlin и требует для своей работы только настройки интерпретатора Python.

Дальнейшая работа возможна в следующих направлениях:

- Использование новых данных, получаемых из внутреннего представления PyCharm, в качестве признаков в моделях машинного обучения для вывода типов. Это позволит анализировать влияние большего количества факторов на качество предсказаний моделей.

- Использование полученного улучшения метрики полноты для разработки более совершенных алгоритмов автодополнения кода в PyCharm. Это позволит улучшить качество и скорость разработки в IDE PyCharm.

- Разработка инструмента, позволяющего интегрировать большее число моделей машинного обучения в PyCharm и в автоматическом режиме сравнивать качество этих моделей. Это позволит уменьшить временные затраты на сравнение различных моделей и даст исследователям возможность больше сфокусироваться на изобретении более совершенных моделей, а не на трудоёмком сравнении с другими подходами.

- Реализация аналогичных плагинов для других динамически типизированных языков, поддерживаемых средами разработки на основе IntelliJ Platform, например, для JavaScript.

Список литературы

[1] Allamanis Miltiadis. The adverse effects of code duplication in machine learning models of code // Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software. — 2019. — P. 143–153.

[2] Alon Uri, Yahav Eran. On the Bottleneck of Graph Neural Networks and its Practical Implications // International Conference on Learning Representations. — 2021. — Access mode: `https://openreview.net/forum?id=i80OPhOCVH2`.

[3] Chopra S., Hadsell R., LeCun Y. Learning a similarity metric discriminatively, with application to face verification // 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05). — Vol. 1. — 2005. — P. 539–546 vol. 1.

[4] Codebert: A pre-trained model for programming and natural languages / Zhangyin Feng, Daya Guo, Duyu Tang et al. // arXiv preprint arXiv:2002.08155. — 2020.

[5] Cover Thomas, Hart Peter. Nearest neighbor pattern classification // IEEE transactions on information theory. — 1967. — Vol. 13, no. 1. — P. 21–27.

[6] DLTPy: Deep Learning Type Inference of Python Function Signatures using Natural Language Context / Casper Boone, Niels de Bruin, Arjan Langerak, Fabian Stelmach // arXiv preprint arXiv:1912.00680. — 2019.

[7] Deep learning type inference / Vincent J Hellendoorn, Christian Bird, Earl T Barr, Miltiadis Allamanis // Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering. — 2018. — P. 152–162.

[8] Distributed representations of words and phrases and their compositionality / Tomas Mikolov, Ilya Sutskever, Kai Chen et al. // Advances in neural information processing systems. — 2013. — Vol. 26.

[9] An Empirical Study on the Impact of Static Typing on Software Maintainability / Stefan Hanenberg, Sebastian Kleinschmager, Romain Robbes et al. // Empirical Softw. Engg. — 2014. — oct. — Vol. 19, no. 5. — P. 1335–1382. — Access mode: https://doi.org/10.1007/s10664-013-9289-1.

[10] Evaluating Large Language Models Trained on Code / Mark Chen, Jerry Tworek, Heewoo Jun et al. — 2021. — 2107.03374.

[11] Flow: A Static Type Checker for JavaScript. — https://flow.org/. — 2014. — Accessed: 2022-05-24.

[12] Gao Zheng, Bird Christian, Barr Earl T. To Type or Not to Type: Quantifying Detectable Bugs in JavaScript // 2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE). — 2017. — P. 758–769.

[13] Hochreiter Sepp, Schmidhuber Jürgen. Long Short-term Memory // Neural computation. — 1997. — 12. — Vol. 9. — P. 1735–80.

[14] KInference: Running ONNX models in vanilla Kotlin. — https://github.com/JetBrains-Research/kinference. — Accessed: 2022-05-24.

[15] Lambdanet: Probabilistic type inference using graph neural networks / Jiayi Wei, Maruth Goyal, Greg Durrett, Isil Dillig // arXiv preprint arXiv:2005.02161. — 2020.

[16] Maia Eva, Moreira Nelma, Reis Rogério. A static type inference for python // Proc. of DYLA. — 2012. — Vol. 5, no. 1. — P. 1.

[17] Malik Rabee Sohail, Patra Jibesh, Pradel Michael. NL2Type: Inferring JavaScript Function Types from Natural Language Information // 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). — 2019. — P. 304–315.

[18] Mir A. M., Latoskinas E., Gousios G. ManyTypes4Py: A Benchmark Python Dataset for Machine Learning-Based Type Inference // IEEE/ACM 18th International Conference on Mining Software Repositories (MSR). — IEEE Computer Society, 2021. — May. — P. 585–589.

[19] PEP 484: Type Hints. — https://peps.python.org/pep-0484/. — 2014. — Accessed: 2022-05-24.

[20] Parnin Chris, Orso Alessandro. Are Automated Debugging Techniques Actually Helping Programmers? // Proceedings of the 2011 International Symposium on Software Testing and Analysis. — ISSTA '11. — New York, NY, USA : Association for Computing Machinery, 2011. — P. 199–209. — Access mode: https://doi.org/10.1145/2001420.2001445.

[21] PyTorch: An Imperative Style, High-Performance Deep Learning Library / Adam Paszke, Sam Gross, Francisco Massa et al. // Advances in Neural Information Processing Systems 32 / Ed. by H. Wallach, H. Larochelle, A. Beygelzimer et al. — Curran Associates, Inc., 2019. — P. 8024–8035.

[22] Pyre: A performant type-checker for Python 3. — https://pyre-check.org/. — 2022. — Accessed: 2022-05-24.

[23] Python Developers Survey 2021. — https://www.jetbrains.com/lp/devecosystem-2021/python/. — 2021. — Accessed: 2022-04-12.

[24] Raychev Veselin, Bielik Pavol, Vechev Martin. Probabilistic model for code with decision trees // ACM SIGPLAN Notices. — 2016. — Vol. 51, no. 10. — P. 731–747.

[25] Raychev Veselin, Vechev Martin, Krause Andreas. Predicting Program Properties from "Big Code" // Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '15. — New York, NY, USA : Association for Computing Machinery, 2015. — P. 111–124. — Access mode: https://doi.org/10.1145/2676726.2677009.

[26] Salib Michael. Starkiller: A static type inferencer and compiler for Python : Ph. D. thesis / Michael Salib ; Massachusetts Institute of Technology. — 2004.

[27] Scikit-learn: Machine Learning in Python / F. Pedregosa, G. Varoquaux,

A. Gramfort et al. // Journal of Machine Learning Research. — 2011. — Vol. 12. — P. 2825–2830.

[28] The Stanford CoreNLP natural language processing toolkit / Christopher D Manning, Mihai Surdeanu, John Bauer et al. // Proceedings of 52nd annual meeting of the association for computational linguistics: system demonstrations. — 2014. — P. 55–60.

[29] Stuchlik Andreas, Hanenberg Stefan. Static vs. Dynamic Type Systems: An Empirical Study about the Relationship between Type Casts and Development Time // Proceedings of the 7th Symposium on Dynamic Languages. — DLS '11. — New York, NY, USA : Association for Computing Machinery, 2011. — P. 97–106. — Access mode: https://doi.org/10.1145/2047849.2047861.

[30] TF–IDF // Encyclopedia of Machine Learning / Ed. by Claude Sammut, Geoffrey I. Webb. — Boston, MA : Springer US, 2010. — P. 986–987. — ISBN: 978-0-387-30164-8. — Access mode: https://doi.org/10.1007/978-0-387-30164-8_832.

[31] Abadi Martín, Agarwal Ashish, Barham Paul et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. — 2015. — Software available from tensorflow.org. Access mode: https://www.tensorflow.org/.

[32] Top Programming Languages 2021. — https://spectrum.ieee.org/top-programming-languages/. — Accessed: 2022-05-17.

[33] Type4Py: Deep Similarity Learning-Based Type Inference for Python / Amir M Mir, Evaldas Latoskinas, Sebastian Proksch, Georgios Gousios // arXiv preprint arXiv:2101.04470. — 2021.

[34] Typewriter: Neural type prediction with search-based validation / Michael Pradel, Georgios Gousios, Jason Liu, Satish Chandra // Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. — 2020. — P. 209–220.

[35] Typilus: neural type hints / Miltiadis Allamanis, Earl T Barr, Soline Ducousso, Zheng Gao // Proceedings of the 41st acm sigplan conference on programming language design and implementation. — 2020. — P. 91–105.

[36] Visual Studio Code - Code Editing. Redefined. — https://code.visualstudio.com/. — 2022. — Accessed: 2022-05-24.

[37] Wilcoxon Frank. Individual Comparisons by Ranking Methods // Biometrics Bulletin. — 1945. — Vol. 1, no. 6. — P. 80–83. — Access mode: http://www.jstor.org/stable/3001968 (online; accessed: 2022-05-24).

[38] ast — Abstract Syntax Trees. — https://docs.python.org/3/library/ast.html. — Accessed: 2022-05-24.

[39] astor: Python AST read/write. — https://github.com/berkerpeksag/astor. — Accessed: 2022-05-24.

[40] developers ONNX Runtime. ONNX Runtime. — https://onnxruntime.ai/. — 2021. — Version: x.y.z.

[41] mypy: Optional Static Typing for Python. — http://mypy-lang.org/. — 2014. — Accessed: 2022-05-24.

[42] ncnn: High-performance neural network inference framework optimized for the mobile platform. — https://github.com/Tencent/ncnn. — Accessed: 2022-05-24.

[43] typed-ast: Modified fork of CPython's ast module that parses '# type' comments. — https://github.com/python/typed_ast. — Accessed: 2022-05-24.

[44] van Kemenade Hugo, Si Richard. hugovk/top-pypi-packages: Release 2022.05. — 2022. — May. — Access mode: https://doi.org/10.5281/zenodo.6509621.