Автономная некоммерческая организация высшего образования
«Университет Иннополис»

АННОТАЦИЯ
НА ВЫПУСКНУЮ КВАЛИФИКАЦИОННУЮ РАБОТУ
(БАКАЛАВРСКУЮ РАБОТУ)
ПО НАПРАВЛЕНИЮ ПОДГОТОВКИ
09.03.01 – «ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

НАПРАВЛЕННОСТЬ (ПРОФИЛЬ) ОБРАЗОВАТЕЛЬНОЙ ПРОГРАММЫ
«ИНФОРМАТИКА И ВЫЧИСЛИТЕЛЬНАЯ ТЕХНИКА»

Тема **Предсказание типа переменных в коде Python**

Выполнил **Хузин Айдар Илдусович**

подпись

Иннополис, Innopolis, 2023

Оглавление

Аннотация

Основным преимуществом динамически типизированных языков программирования, например Python, является быстрое прототипирование, тогда как статическая типизация делает код понятным, облегчает поддержку и позволяет обнаруживать ошибки раньше. Стандарт PEP484 позволяет опционально типизировать переменные и функции в Python. Чтобы облегчить типизирование, исследователи разработали различные методы для предсказания типов на основе машинного и глубокого обучения. В TypeScript передовой моделью для предсказания типов является TypeBert, основанная на модели BERT, и воспринимающая задачу предсказания типов как аннотацию последовательности. Однако производительность BERT-подобных моделей в задаче вывода типов в Python-коде пока не исследовалась. Для заполнения этого пробела в исследованиях я настроил модель CodeBERT для предсказания типов в Python-коде. Модель была обучена с использованием глубоко обучения по сходству с целью разрешения проблемы ограниченного словаря, которая присутствует в TypeBert. Таким образом, эта модель способна предсказывать новые типы без повторного обучения. В работе я изучил применимость обучения по сходству для CodeBERT и влияние стратегии выбора триплетов и значения маржи на производительность. Полученная модель не превзошла передовые решения, однако демонстрирует сравнимую производительность с общим снижением точности на 5,4% по сравнению с Type4Py.

Глава 1

Введение

Динамическая типизация набирает популярность в последние годы, согласно IEEE Spectrum [1] Python признан самым широко используемым языком программирования в 2022 году. Одним из основных преимуществ динамических языков является возможность быстрого прототипирования, что упрощает разработку [2].

Однако динамическая типизация может привести к проблемам с поддержкой кода и обнаружением ошибок. В этом случае статическая типизация может помочь. Явное указание типов позволяет создавать более понятный и поддерживаемый код, а также помогает обнаруживать ошибки раньше [3].

Чтобы дополнить Python преимуществами статической типизации, сообщество добавило возможность опциональной типизации переменных и функций со стандартом PEP484 [4]. Чтобы облегчить типизацию были разработаны инструменты для статического анализа типов в Python, такие как Pytype [5], Pyre [6], Pysonar2 [7] и PyRight [8]. Статические анализаторы предсказывают типы на основе заранее определенных правил и огра-

ничений. Такой подход уже используется в некоторых статически типизированных языках, например, Scala и Kotlin [9], [10]. Однако у статических анализаторов присутствует проблема переаппроксимации [11].

Для преодоления этих ограничений недавние исследования изучили использование машинного обучения и глубокого обучения на основе условных случайных полей (CRF), вероятностных графических моделей (PGM), LSTM, RNN и GNN. Эти модели показали лучшую производительность по сравнению со статическими анализаторами [12].

На данный момент ведущей моделью для вывода типов в Python является HiTyper [13]. HiTyper представляет собой новый подход, который объединяет статический анализ с нейронными сетями для вывода типов. Для TypeScript ведущей моделью является TypeBert [14], которая рассматривает вывод типов как задачу аннотации последовательности. Однако, несмотря на прогресс в исследованиях вывода типов, недостаточно изучена производительность BERT-подобных моделей в задаче вывода типов в коде Python.

Для заполнения этой пробела в исследованиях я обучил модель CodeBERT [15] для предсказания типов на наборе данных ManyTypes4Py [16]. Кроме того, я решил проблему закрытого словаря, присутствующую в TypeBert, с помощью глубокого обучения по сходству (DSL). В данной работе исследуются следующие вопросы.

1: Какова производительность CodeBERT, обученной с помощью DSL, по сравнению с другими моделями вывода типов для Python?

2: Как выбор стратегии подбора триплетов влияет на производительность?

3: Как выбор значения маржи влияет на производительность?

Глава 2

Обзор литературы

Существует ряд утилит для предсказания типов переменных в Python, основанных на статическом анализе, таких как Pytype [5], Pyre [6], Pysonar2 [7], PyRight [8]. Статические подходы выводят типы на основе набора правил и ограничений. Однако эти методы склонны быть неточными [17] из-за динамической природы Python. Кроме того, статические методы переаппроксимируют [11] поведение программы.

JSNice, представленная Райчевым и др. [18], является моделью на основе обучения, которая предсказывает имена переменных и типы в коде JavaScript с использованием условных случайных полей (CRFs). Она улавливает зависимости между элементами программы и изучает статистические корреляции.

Ху и др. [12] представили подход на основе вероятностных графических моделей (PGM) для вывода типов переменных в Python. Их модель рассматривает поток данных, доступы к атрибутам, имена переменных и явные проверки типов в качестве указаний типов, превосходя производительность PySonar2.

Хелендурн и др. [19] рассматривали вывод типов как задачу аннотации последовательности и предложили модель DeepTyper, которая улавливает широкий контекст, но может назначать разные аннотации типов для одной и той же переменной.

Малик и др. [20] разработали модель NL2Type, основанную на LSTM, которая предсказывает аргументы функций и возвращаемые типы, извлекая информацию на естественном языке из названий функций и комментариев. NL2Type превосходит JSNice и DeepTyper.

Прадел и др. [21] представили модель TypeWriter, которая объединяет естественный язык и контекст кода для вывода типов. TypeWriter кодирует указания типов с помощью рекуррентных нейронных сетей (RNN) и проверяет полученные типы с использованием внешнего инструмента, постепенно проверяющего типы. Она превосходит NL2Type и DeepTyper.

Веи и др. [22] представили вероятностный алгоритм вывода типов LambdaNet для TypeScript. LambdaNet строит граф зависимостей типов и использует графовые нейронные сети (GNN) для вычисления векторных представлений для каждой переменной. Она превосходит DeepTyper и предоставляет предсказания на уровне переменных.

Панди и др. [23] представили OptTyper, который объединяет логические и естественные ограничения для предсказания типов переменных в TypeScript. OptTyper использует логические ограничения из компилятора TypeScript и кодирует естественные ограничения с помощью LSTM на уровне символов. Она достигает более высокой точности, чем LambdaNet.

Алламенис и др. [24] представили Typilus, модель на основе графовых нейронных сетей (GNN) для предсказания типов в Python. Typilus строит графовое представление кода, используя синтаксические шаблоны, поток

данных и имена токенов, и получает типы переменных с использованием алгоритма k-ближайших соседей (KNN).

Джесс и др. [14] применили BERT для вывода типов в TypeScript и представили TypeBert, превосходящую LambdaNet и OptTyper. Эта модель воспринимает задачу предсказания типов как аннотацию последовательности и делает предсказания для фиксированного словаря типов. По предположению авторов, TypeBert способен изучать семантику кода, без создания явных представлений в виде графов.

Мир и др. [25] разработали Type4Py, модель вывода типов в Python, которая извлекает информацию на естественном языке, контекст кода и видимые указатели типов. Type4Py превосходит Typilus и TypeWriter и решает проблему ограниченного словаря типов с помощью обучения по сходству.

Пенг и др. [13] представили HiTyper, передовой подход для предсказания типов в Python. HiTyper объединяет модуль статического вывода типов с нейронным предсказанием типов. Авторы не разрабатывали свою нейронную модель, а использовали Typilus и Type4Py, и улучшили производительность обоих моделей.

Кроме того, в 2023 году Веи и др. объявили о модели TypeT5 [26], которая превосходит HiTyper. OpenAI также выпустила ChatGPT [27], большую языковую модель, способную предсказывать типы переменных в коде Python. Эта работа была по большей части выполнена до выпуска этих моделей, поэтому я не привожу сравнение с ними.

Глава 3

Методология

Несмотря на то, что программный код имеет четкую структуру и существуют известные способы представления его в виде графов, графовые модели для языков Python и TypeScript уступают другим моделям. TypeBert [14] - модель предсказания типов для TypeScript, являющаяся лучшей на данный момент, основана на BERT [28] и рассматривает предсказания типов как задачу аннотации последовательности.

Для понимания того, какую производительность может продемонстрировать модель на основе BERT в Python, модель CodeBERT [15] была адоптирована для задачи предсказания типов. Кроме того, модель была обучена с использованием DSL для решения проблемы закрытого словаря.

3.1 Обучение по сходству

Для того чтобы модель отображала переменные с одинаковыми типами ближе друг к другу и как можно дальше от других переменных, я использовал функцию потери, называемую *Triplet Loss* [29]. При данных якоре a, положительном примере с тем же типом p, отрицательном при-

мере с другим типом n, положительной скалярной марже m и функции расстояния d значение *Triplet Loss* вычисляется следующим образом:

$$L(a, p, n) = max(0, m + d(a, p) - d(a, n)) \tag{3.1}$$

Целью функции L является создание представлений таким образом, чтобы представление якоря было ближе на величину маржи к положительному примеру, чем к отрицательному.

При использовании описанной функции потери необходимо каким-то образом выбирать триплеты. Существует два основных способа делать это: либо формировать триплеты перед обучением, так называемая оффлайн стратегия, либо выбирать их для каждого мини-пакета, другими словами, онлайн стратегия.

Оффлайн стратегия приведет к формированию списка троек *(a, p, n)*. В этом случае модель должна обрабатывать весь код, в котором встречалась переменная, чтобы получить ее представление, поскольку представления контекстуализированы. Оффлайн стратегия сделает обучение неэффективным, поэтому я решил сосредоточиться на онлайн стратегиях, которые описаны далее.

Интуитивным способом выбора троек является обработка всех возможных триплетов, так называемая стратегия *Batch All*. Однако важно выбирать тройки таким образом, чтобы они создавали полезные градиенты для модели. Если отрицательный пример уже находится дальше от положительного примера на величину маржи, то потеря будет равна нулю. Стратегия *Batch All* склонна генерировать большое количество примеров с нулевой потерей, что может привести к размытию потерь от других триплетов.

Стратегия *Batch Hard* (BH) - это еще один способ формирования троек; в функцию потерь передаются только самые сложные положительные и отрицательные примеры для каждого якоря. С данными векторными представлениями X и функцией C, которая определяет правильный класс для векторного представления, потеря в случае *Batch Hard* вычисляется следующим образом:

$$L_{bh}(X) = \overbrace{\sum_{a}}^{\text{все якоря}} L(a, \overbrace{\underset{p}{\operatorname{argmax}}(d(a,p))}^{\text{самый сложный положительный}}, \overbrace{\underset{n}{\operatorname{argmin}}(d(a,n))}^{\text{самый сложный отрицательный}}) \tag{3.2}$$

где

$$a \in X \qquad p : \underset{\substack{p \neq a \\ C(p)=C(a)}}{\overset{p \in X}{}} \qquad n : \underset{C(n) \neq C(a)}{\overset{n \in X}{}}$$

Нужно заметить, что данная стратегия может привести к обучению только на дивиантных примерах. В то же время, триплеты формируются только внутри мини-пакетов, поэтому примеры выбираются из небольшого подмножества всего набора данных, что делает такие триплеты *умеренными*.

Стратегия *Batch Semi-Hard* (BSH) показала лучшую производительность в соответствии с [29], [30].

$$L_{bsh}(X) = \overbrace{\sum_{a}}^{\text{все якоря}} L(a, \overbrace{\underset{p}{\operatorname{argmin}}(d(a,p))}^{\text{самый легкий положительный}}, \overbrace{\underset{n}{\operatorname{argmin}}(d(a,n))}^{\text{полусложный отрицательный}}) \tag{3.3}$$

где

$$a \in X \qquad p : \underset{\substack{p \neq a \\ C(p)=C(a)}}{\overset{p \in X}{}} \qquad n : \underset{\substack{C(n) \neq C(a) \\ d(a,p)<d(a,n)}}{\overset{n \in X}{}}$$

Основная идея этого подхода заключается в том, что необязательно,

чтобы все примеры некоторого класса сходились к одной точке в пространстве, так как предсказание обычно осуществляется при помощи поиска ближайших соседей (KNN); поэтому важны только ближайшие соседи.

3.2 Обучение и предсказание типов

Я обучил модель, используя стратегии *Batch Hard* и *Batch Semi-Hard*, комбинируя их или используя каждую отдельно. Результаты будут рассмотрены в главе 4.

После обучения модель способна отобразить токены в многомерное пространство. Однако она не может определить типы токенов, поэтому создается *пространство типов*.

Тренировочные данные содержат токены с помеченными типами. Для каждого из этих токенов модель вычисляет проекцию в *пространстве типов*. Проекции помечаются соответствующими типами. На этом этапе сохраняются представления для всех типов, встречающихся в тренировочных данных.

Словарь модели можно постепенно расширять во время использования. Если модель встречает переменную с новым типом, она может вычислить ее представление и добавить его в *пространство типов* с соответствующей меткой типа. Нет необходимости повторно обучать или настраивать саму модель.

Для предсказания типов модель вычисляет представление для данной переменной и находит ближайшего соседа *пространстве типов*. Тип этого соседа является предсказанным типом переменной.

3.3 Набор данных

В данной работе я использую набор данных ManyTypes4Py [16]. Мир и др. загрузили проекты на языке Python с зависимостью *mypy* из репозиториев GitHub. Они загрузили 5,382 проекта, удалили дубликаты и сгенерировали *нормализованные представления seq2seq* исходных файлов с использованием библиотеки LibSA4Py [31].

Набор данных имеет некоторые ограничения из-за использования *нормализованного представления seq2seq*, которое вводит потерю информации. Во-первых, отсутствуют табуляции. Табуляции являются важной частью кода Python, и невозможно полностью восстановить семантику без них. Во-вторых, документация и комментарии заменяются токеном "[comment]". Во многих случаях комментарии могут содержать полезную информацию о типе и значении переменных.

3.4 Предварительная обработка типов

Аналогично предыдущим работам [21], [24], [25], я провел несколько предварительных этапов обработки типов:

1. Такие функции, как __str__ и __len__, тривиальны для аннотации. __str__ всегда возвращает string, а __len__ всегда возвращает int. Предсказывание типов, возвращаемых этими функциями, не демонстрирует способность модели выводить типы и размывает результаты, поэтому они были исключены из данных.

2. Типы Any и None были исключены, так как их предсказание не представляет ценности.

3. Параметрические типы с вложенностью больше двух были заменены на тип Any. Например, Dict[str, List[Union[str, int]]] переписывается как Dict[str, List[Any]]. Мир и др. утверждают: "Это позволяет исключить очень редкие и дивиантные типы".

3.5 Оценка производительности

Я следовал установке для оценки производительности, предложенной Мир и соавторами. Модель оценивается на *всех*, *повсеместных*, *частых* и *редких* типах. *Повсеместные* типы определены как {str, int, list, bool, float}. Если частота типа в тренировочном наборе больше 100, то он считается *частым*, иначе - *редким*. *Повсеместные* типы исключаются из множества *частых* типов.

Для измерения производительности предсказания типов я использовал два критерия, предложенных в работе [24].

1. *Точное совпадение*: предсказанный моделью тип t_p и целевой тип t_t совпадают.

2. *Совпадение до параметрического типа*[1]: параметры типа игнорируются, и учитывается только базовый тип. Например, если t_p - List[int], а t_t - List[str], то критерий считается выполненным.

Производительность оценивалась с использованием следующих метрик.

[1] *Повсеместные* типы не имеют параметров типа, поэтому метрики не измеряются отдельно для этого набора типов по критерию *совпадение до параметрического типа*. Однако *повсеместные* типы учитываются при измерении метрик для *всех* типов.

1. *Попадание среди топ-k* (*Hit rate top-k*): указывает, есть ли правильное предсказание среди топ-k предсказаний модели.

2. *Средний обратный ранг (MRR)*: данная метрика оценивает позицию правильного предсказания. В частности, я буду использовать метрику *MRR@10*. В отличие от обычной *MRR*, она учитывает только первые 10 предсказаний модели. И если среди топ-10 предсказаний нет соответствующего предсказания, то ранг считается равным нулю.

$$MRR = \frac{1}{\mid Q \mid} \sum_{q} \frac{1}{\text{rank}_q} \tag{3.4}$$

где:

Q - множество запросов

$q \in Q$

rank_q - позиция первого правильного предсказания для запроса q

Глава 4

Результаты и анализ

Реализация доступна на GitHub.[1]

4.1 Анализ стратегий выбора триплетов

Таблица I предоставляет оценку производительности CodeBERT, обученного с использованием различных стратегий выбора триплетов. После анализа этой таблицы можно сказать, что CodeBERT, обученный с использованием стратегии Batch Hard, имеет лучшую производительность, чем модель натренированная стратегией Batch Semi-Hard, во всех показателях оценки. Например, показатель попадания среди топ-1 лучше на 3,7% и 3,1% по критерию точного совпадения и совпадения до параметрического типа соответственно.

Эти результаты противоречат наблюдениям Шроффа и др. [29]. Авторы не предоставили прямого сравнения этих стратегий, но заключили, что выбор самых сложных примеров может привести к преждевременной сходимости к плохим локальным минимумам. Возможной причиной, поче-

[1] https://github.com/gfx73/CodeBERT-DSL

му это не произошло в моих экспериментах, может быть количество примеров в мини-пакете. Шрофф и др. проводили эксперименты с размером мини-пакета около 1800, где каждый пример в пакете соответствует одному классу. В моем случае каждый пакет содержит около 108 типов. Кроме того, это наблюдение подтверждается исследованием Хуана и др. [30]. Они отметили, что большое количество примеров одного класса в пакете увеличивает вероятность того, что якорь и самый сложный положительный пример будут очень различными. Следовательно, в случае меньшего количества примеров в пакете эта проблема обычно не возникает. Кроме того, в работе [32] также были получены обнадеживающие результаты с использованием стратегии Batch Hard.

Таблица I: Оценка производительности при использовании различных стратегий выбора триплетов

Hit rate	Стратегия	% Точное совпадение				% До параметрического		
		Все	Повсеместные	Частые	Редкие	Все	Частые	Редкие
Топ-1	BSH	69.0	91.0	67.1	14.1	75.2	76.8	31.6
	BH	**71.7**	**92.6**	**70.3**	**18.6**	**77.6**	**78.9**	**36.1**
	BSH+BH	**72.9**	**92.9**	**73.7**	**19.3**	**78.6**	**81.3**	**37.1**
Топ-5	BSH	74.6	95.6	76.3	16.5	80.9	84.8	36.6
	BH	**77.5**	**96.5**	**81.2**	**22.0**	**83.5**	**88.0**	**42.5**
	BSH+BH	**78.2**	**96.3**	**83.3**	**23.4**	**83.9**	**89.3**	**43.7**
MRR@10	BSH	71.4	93.0	71.1	15.2	77.7	80.2	33.8
	BH	**74.1**	**94.3**	**74.9**	**20.0**	**80.1**	**82.8**	**38.9**
	BSH+BH	**75.2**	**94.4**	**77.7**	**21.0**	**80.9**	**84.6**	**39.9**

Также я провел другой эксперимент, применяя стратегию Batch Hard после стратегии Batch Semi-Hard. Этот подход приводит к лучшим резуль-

татам по сравнению с использованием только стратегии Batch Hard.

4.2 Анализ значений маржи

Таблица II: Оценка производительности при различных значениях маржи и стратегии BH

Hit rate	Маржа	% Точное совпадение				% До параметрического		
		Все	Повсеместные	Частые	Редкие	Все	Частые	Редкие
топ-1	0.5	**71.8**	**92.7**	**70.9**	18.3	**77.8**	**79.5**	36.0
	1	71.7	**92.6**	70.3	**18.6**	77.6	78.9	**36.1**
	2	**71.8**	92.2	**71.1**	**19.1**	**77.7**	**79.6**	**37.0**
Топ-5	0.5	**77.7**	**96.8**	**81.4**	**22.0**	**83.7**	**88.5**	42.4
	1	**77.5**	**96.5**	81.2	**22.0**	**83.5**	88.0	**42.5**
	2	**77.5**	96.1	**81.5**	**22.6**	**83.5**	**88.3**	**43.2**
MRR@10	0.5	**74.3**	**94.4**	**75.4**	19.9	**80.3**	**83.3**	38.7
	1	74.1	**94.3**	74.9	**20.0**	80.1	82.8	**38.9**
	2	**74.2**	93.9	**75.5**	**20.6**	**80.2**	**83.3**	**39.7**

Таблица II показывает оценку производительности при тренировке с различными значениями маржи и стратегией Batch Hard. Исходя из этих данных, влияние значения маржи на общую производительность минимально. В частности, как показывают попадание среди топ-1 и MRR@10, изменения составляют всего 0,2% для *всех* типов. Однако можно заметить, что при увеличении значения маржи производительность для *повсеместных* типов имеет тенденцию к снижению, в то время как производительность для *редких* типов склонна улучшаться.

4.3 Оценка производительности финальной модели

Во время предыдущих экспериментов я обнаружил, что модель, обученная сначала стратегией Batch Semi-Hard, а затем стратегией Batch Hard, показывает лучшую производительность на валидационном наборе данных. После этого я оценил производительность этой модели на тестовом наборе данных. Я следовал экспериментальной настройке Мир и др. Таким образом, я сравнил производительность CodeBERT, измеренную мной, с метриками других моделей, представленными Mir и др. [25, Таб. 5]. Результаты сравнения представлены в таблице III.

Таблица III: Оценка производительности разных моделей

Hit rate	Модель	% Точное совпадение				% До параметрического		
		Все	Повс-ые	Частые	Редкие	Все	Частые	Редкие
Топ-1	CodeBERT+DSL	**71.7**	91.4	73.1	**19.5**	**78.0**	**82.1**	**37.8**
	Type4Py	**75.8**	**100.0**	**82.3**	19.2	**80.6**	**85.2**	36.0
	Typilus	66.1	**92.5**	**73.4**	**21.6**	74.2	81.6	**41.7**
Топ-5	CodeBERT+DSL	**77.4**	96.0	82.3	22.9	**83.6**	89.4	43.7
	Type4Py	**78.7**	**100.0**	**88.6**	**24.5**	**84.7**	**92.1**	**45.5**
	Typilus	72.7	**96.7**	**85.1**	**28.2**	80.9	**90.1**	**51.0**
MRR@10	CodeBERT+DSL	**74.1**	93.4	77.0	21.0	**80.4**	**85.2**	**40.3**
	Type4Py	**77.1**	**100.0**	**85.1**	**21.4**	**74.1**	**79.9**	29.4
	Typilus	69.0	**94.4**	**78.5**	**24.4**	67.4	75.8	**32.8**

Можно сказать, что Type4Py имеет самую высокую общую производительность. CodeBERT, натренированная с использованием обучения по

сходству, не достигла передовой производительности, в задаче вывода типов для Python, в отличие от TypeBert в контексте TypeScript. Однако она показала сопоставимую производительность с общим снижением показателя попадания среди топ-1 на 5.4% по сравнению с Type4Py. Возможной причиной этого может быть то, что существующие модели для Python более продвинуты по сравнению с созданными для TypeScript.

Кроме того, производительность CodeBERT может быть улучшена. В наборе данных ManyTypes4Py комментарии и табуляции удалены из кода. Комментарии могут содержать полезную информацию о значении и типе переменных. Кроме того, невозможно полностью восстановить семантику Python кода без табуляций. Таким образом, использование набора данных, не имеющего этих ограничений, вероятно, улучшит производительность модели.

Глава 5

Заключение

В данном исследовании была изучена производительность модели CodeBERT для вывода типов в языке программирования Python. Путем тренировки с использованием обучения по сходству была решена проблема закрытого словаря. Однако результаты показывают, что модель типа BERT, примененная к Python, не достигает ведущей производительности в области вывода типов, в отличие от модели TypeBert в контексте TypeScript. Возможной причиной этого различия в производительности является, я предполагаю, то, что модели для Python более совершенны по сравнению, с созданными для TypeScript. Кроме того я изучил насколько хорошо обучение по сходству может быть применено к модели CodeBERT и как выбор стратегии подбора триплетов и значения маржи влияют на производительность. Для достижения более высокой производительности, необходимы дополнительные исследования.

Список использованной литературы

[1] S. Cass, *Top programming languages 2022*, нояб. 2022. url: https://spectrum.ieee.org/top-programming-languages-2022.

[2] A. Stuchlik и S. Hanenberg, «Static vs. dynamic type systems: An empirical study about the relationship between type casts and development time,» в *Proceedings of the 7th symposium on Dynamic languages*, 2011, с. 97—106.

[3] Z. Gao, C. Bird и E. T. Barr, «To type or not to type: quantifying detectable bugs in JavaScript,» в *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, с. 758—769.

[4] G. V. Rossum, J. Lehtosalo и L. Langa, «PEP 484 – Type Hints,» PEP 484, 2014. url: https://peps.python.org/pep-0484/.

[5] Google, *Pytype*, https://github.com/google/pytype, 2022.

[6] Facebook, *Pyre*, https://github.com/facebook/pyre-check, 2022.

[7] yinwang0, *Pysonar2*, https://github.com/yinwang0/pysonar2, 2022.

[8] Microsoft, *PyRight*, https://github.com/microsoft/pyright, 2022.

[9] Scala, *Scala*, вер. 2.13.10, 13 окт. 2022. url: https://www.scala.com/.

[10] JetBrains, *Kotlin Programming Language*, вер. 1.7.21, 8 нояб. 2022. url: https://kotlinlang.org/.

[11] M. Madsen, «Static analysis of dynamic languages,» *Available: http://pure.au.dk/ws/files/85299449/Thesis.pdf*, 2015.

[12] Z. Xu, X. Zhang, L. Chen, K. Pei и B. Xu, «Python probabilistic type inference with natural language support,» в *Proceedings of the 2016 24th ACM SIGSOFT international symposium on foundations of software engineering*, 2016, с. 607—618.

[13] Y. Peng, C. Gao, Z. Li и др., «Static inference meets deep learning: a hybrid type inference approach for python,» в *Proceedings of the 44th International Conference on Software Engineering*, 2022, с. 2019—2030.

[14] K. Jesse, P. T. Devanbu и T. Ahmed, «Learning type annotation: is big data enough?» В *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, с. 1483—1486.

[15] Z. Feng, D. Guo, D. Tang и др., «Codebert: A pre-trained model for programming and natural languages,» *arXiv preprint arXiv:2002.08155*, 2020.

[16] A. M. Mir, E. Latoškinas и G. Gousios, «Manytypes4py: A benchmark python dataset for machine learning-based type inference,» в *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*, IEEE, 2021, с. 585—589.

[17] Z. Pavlinovic, «Leveraging Program Analysis for Type Inference,» дис. ... док., New York University, 2019.

[18] V. Raychev, M. Vechev и A. Krause, «Predicting program properties from"big code",» *ACM SIGPLAN Notices*, т. 50, № 1, с. 111—124, 2015.

[19] V. J. Hellendoorn, C. Bird, E. T. Barr и M. Allamanis, «Deep learning type inference,» в *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, 2018, с. 152—162.

[20] R. S. Malik, J. Patra и M. Pradel, «NL2Type: inferring JavaScript function types from natural language information,» в *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, IEEE, 2019, с. 304—315.

[21] M. Pradel, G. Gousios, J. Liu и S. Chandra, «Typewriter: Neural type prediction with search-based validation,» в *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2020, с. 209—220.

[22] J. Wei, M. Goyal, G. Durrett и I. Dillig, «Lambdanet: Probabilistic type inference using graph neural networks,» *arXiv preprint arXiv:2005.02161*, 2020.

[23] I. V. Pandi, E. T. Barr, A. D. Gordon и C. Sutton, «Opttyper: Probabilistic type inference by optimising logical and natural constraints,» *arXiv preprint arXiv:2004.00348*, 2020.

[24] M. Allamanis, E. T. Barr, S. Ducousso и Z. Gao, «Typilus: Neural type hints,» в *Proceedings of the 41st acm sigplan conference on programming language design and implementation*, 2020, с. 91—105.

[25] A. M. Mir, E. Latoskinas, S. Proksch и G. Gousios, «Type4py: Deep similarity learning-based type inference for python,» *arXiv preprint arXiv:2101.04470*, 2021.

[26] J. Wei, G. Durrett и I. Dillig, «TypeT5: Seq2seq Type Inference using Static Analysis,» *arXiv preprint arXiv:2303.09564*, 2023.

[27] OpenAI, *ChatGPT: Conversational AI Model*, https://openai.com/research/chatgpt, Accessed: Month Day, Year, 2021.

[28] J. Devlin, M.-W. Chang, K. Lee и K. Toutanova, «Bert: Pre-training of deep bidirectional transformers for language understanding,» *arXiv preprint arXiv:1810.04805*, 2018.

[29] F. Schroff, D. Kalenichenko и J. Philbin, «Facenet: A unified embedding for face recognition and clustering,» в *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2015, с. 815—823.

[30] H. Xuan, A. Stylianou и R. Pless, «Improved embeddings with easy positive triplet mining,» в *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, 2020, с. 2474—2482.

[31] A. Mir, *LibSA4Py*, вер. 0.3.0, 8 апр. 2023. url: https://github.com/saltudelft/libsa4py.

[32] A. Hermans, L. Beyer и B. Leibe, «In defense of the triplet loss for person re-identification,» *arXiv preprint arXiv:1703.07737*, 2017.