

Санкт-Петербургский государственный университет

Евдокимов Данил

Выпускная квалификационная работа

Автоматизация и улучшение декомпиляции Kotlin

Уровень образования: бакалавриат

Направление *09.03.04 «Программная инженерия»*

Основная образовательная программа *СВ.5080.2021 «Программная инженерия»*

Научный руководитель:
доцент кафедры Системного Программирования, к.ф.-м.н., Гориховский В. И.

Рецензент:
главный инженер-программист, ООО «ШВАЧЕР», Васенина А. И.

Санкт-Петербург
2025

Saint Petersburg State University

Yevdokimov Danil

Bachelor's Thesis

Automating and improving Kotlin decompilation

Education level: bachelor

Speciality *09.03.04 "Software Engineering"*

Programme *CB.5080.2020 "Software Engineering"*

Scientific supervisor:
docent, C.Sc, Gorikhovskii V. I.

Reviewer:
chief software engineer, LLC «SHVACHER», Vasenina A. I.

Saint Petersburg
2025

Оглавление

Введение	4
1. Обзор	5
1.1. Обзор существующих решений	5
1.2. Обзор технологий для анализа Kotlin-кода	7
1.3. Обзор метрик оценки качества декомпиляции	8
1.4. Обзор математических методов	10
2. Постановка задачи	12
3. Описание решения	13
3.1. Тестовые наборы и оценка существующих подходов . . .	13
3.2. Разработка подхода к декомпиляции Kotlin	25
4. Эксперимент	35
4.1. Условия эксперимента	35
4.2. Исследовательские вопросы	35
4.3. Метрики	35
4.4. Результаты	36
4.5. RQ1	37
4.6. RQ2	37
4.7. Обсуждение результатов	38
4.8. Угрозы нарушения корректности	38
4.9. Воспроизводимость эксперимента	38
Заключение	39
Список литературы	40

Введение

Язык программирования Kotlin набирает популярность у разработчиков¹. Он был разработан компанией JetBrains и официально представлен в 2011 году. Он ориентирован на повышение выразительности и безопасности кода при сохранении совместимости с экосистемой Java. В 2017 году компания Google объявила Kotlin официально поддерживаемым языком для разработки Android-приложений², а в 2019 году — основным языком для Android-разработки³.

Однако в области инструментов реверс-инжиниринга отсутствует полноценный декомпилятор, способный корректно восстанавливать исходный код из байт-кода JVM с учётом специфических особенностей языка Kotlin⁴.

Существующие декомпиляторы, созданные для Java, могут использоваться и для байт-кода, полученного из Kotlin. Однако после декомпиляции в Java-код для получения Kotlin-кода необходимо применять отдельные инструменты-конвертеры, что добавляет этап в процесс декомпиляции. Java-декомпиляторы не восстанавливают конструкции Kotlin, такие как лямбда-выражения, data-классы и другие. Это вызывает трудности при анализе и отладке программ, усложняет процессы реверс-инжиниринга и миграции.

Таким образом, актуальна задача создания декомпилятора для Kotlin, учитывающего идиоматические конструкции языка и повышающего точность восстановления исходного кода.

¹TIOBE Index, URL: <https://www.tiobe.com/tiobe-index/> (дата обращения: 26 мая 2025 г.).

²Google makes Kotlin a first-class language for writing Android apps, URL: <https://techcrunch.com/2017/05/17/google-makes-kotlin-a-first-class-language-for-writing-android-apps/> (дата обращения: 26 мая 2025 г.).

³Kotlin is now Google's preferred language for Android app development, URL: <https://techcrunch.com/2019/05/07/kotlin-is-now-googles-preferred-language-for-android-app-development/> (дата обращения: 26 мая 2025 г.).

⁴Kotlin Programming Language, URL: <https://kotlinlang.org/> (26 мая 2025 г.).

1. Обзор

В данном разделе проводится анализ существующих инструментов и подходов, применяемых для декомпиляции Kotlin. Обзор включает три ключевых аспекта: преобразование байт-кода в Java, преобразование Java в Kotlin, а также существующие подходы, направленные непосредственно на декомпиляцию Kotlin-программ.

Кроме того, рассматриваются технологии, применяемые для анализа синтаксических и структурных характеристик Kotlin-кода. Завершающие разделы обзора посвящены метрикам, позволяющим количественно оценивать качество декомпиляции, и методам для выявления закономерностей в них.

1.1. Обзор существующих решений

На сегодняшний день не существует устоявшихся инструментов и подходов, осуществляющих декомпиляцию Kotlin. На практике можно использовать двухэтапный подход: сначала байт-код преобразуется в Java-код с помощью Java-декомпиляторов, после чего полученный Java-код конвертируется в Kotlin с использованием конвертеров. Альтернативой такому подходу является применение LLM, способных генерировать Kotlin-код напрямую.

1.1.1. Обзор декомпиляторов Java

Существует множество декомпиляторов Java, среди которых Jad, Procyon, Krakatau, JADX и другие. В данном обзоре выбраны те, которые обладают сравнительно высокой популярностью [5, 10], активной поддержкой, а также имеют лицензии, позволяющие их применение в коммерческих и некоммерческих проектах. Такими являются Fernflower⁵, CFR⁶ и JD-GUI⁷. Более подробное сравнение характеристик

⁵Fernflower, URL: <https://github.com/JetBrains/intellij-community/tree/master/plugins/java-decompiler/engine> (дата обращения: 2 марта 2025 г.).

⁶CFR, URL: <https://github.com/leibnitz27/cfr> (дата обращения: 2 марта 2025 г.).

⁷JD-GUI, URL: <https://github.com/java-decompiler/jd-gui> (дата обращения: 2 марта 2025 г.).

декомпиляторов см. табл. 1.

Декомпилятор	Разработчик	Интеграция	Версия Java	Лицензия
Fernflower	JetBrains	Встроен в IntelliJ IDEA	от Java 4	Apache 2.0
CFR	Lee Benfield	Независимый инструмент	от Java 6	MIT
JD-GUI	Emmanuel Dupuy	Независимый инструмент	от Java 4	GPL-3.0

Таблица 1: Обзор декомпиляторов Java.

1.1.2. Обзор инструментов для конвертации Java в Kotlin

Для анализа существующих решений выбраны три представителя разных подходов к преобразованию Java-кода в Kotlin: официальный инструмент от JetBrains, специализированная языковая модель и универсальная языковая модель.

- J2K⁸ — инструмент, встроенный в среду разработки IntelliJ IDEA.
 - *Преимущества*: сохранение структуры исходного кода.
 - *Недостатки*: не использует идиоматические конструкции Kotlin; необходим полный корректный Java-код; отсутствует CLI.
- CodeConvert⁹ — специализированная языковая модель для задач конвертации между языками программирования.
 - *Преимущества*: генерация более идиоматического и читаемого Kotlin-кода с учётом контекста.
 - *Недостатки*: ограничение на объём входного кода; возможны аномалии в преобразовании; отсутствует API.
- ChatGPT (4o)¹⁰ — универсальная языковая модель.
 - *Преимущества*: способность генерировать Kotlin-код без промежуточного Java-представления; поддержка широкого контекста; присутствует API.

⁸Mixing Java and Kotlin in one project – tutorial, URL: <https://kotlinlang.org/docs/mixing-java-kotlin-intellij.html> (дата обращения: 2 марта 2025 г.).

⁹CodeConvert, URL: <https://www.codeconvert.ai> (дата обращения: 2 марта 2025 г.).

¹⁰ChatGPT, URL: <https://chatgpt.com> (дата обращения: 2 марта 2025 г.).

– *Недостатки*: возможны аномалии в преобразовании.

1.1.3. Обзор существующих подходов к декомпиляции Kotlin

На основе рассмотренных инструментов сформированы десять различных комбинаций декомпилятор–конвертер, направленных на восстановление Kotlin-кода из байт-кода: CFR + J2K, CFR + CodeConvert, CFR+ChatGPT, Fernflower + J2K, Fernflower + CodeConvert, Fernflower + ChatGPT, JD-GUI + J2K, JD-GUI + CodeConvert, JD-GUI + ChatGPT, Bytecode + ChatGPT.

Комбинации с участием CodeConvert используются ограниченно из-за лимитов на объём входного текста и малого количества бесплатных обращений.

Кроме того, J2K не имеет CLI¹¹, что затрудняет автоматизацию с использованием этого инструмента. Сторонние решения, такие как clij2k¹², основанные на автоматизации графического интерфейса, позволяют частично обойти это ограничение, но не обеспечивают воспроизводимость и стабильность.

1.2. Обзор технологий для анализа Kotlin-кода

Для оценки синтаксических и семантических характеристик Kotlin-кода использовались инструменты и библиотеки, предназначенные для количественного анализа кода на уровне абстрактного синтаксического дерева.

Для анализа структуры кода и построения AST применялась библиотека `tree-sitter-kotlin`¹³, реализующая грамматику языка Kotlin для парсера Tree-sitter. Она позволяет осуществлять синтаксический разбор и автоматизированный сбор метрик.

Для последующего анализа данных и визуализации результатов применялись библиотеки Python: `pandas`, `matplotlib`, `statistics`, `sklearn`,

¹¹Command line interface for J2K, URL: <https://youtrack.jetbrains.com/issue/KTIJ-14957> (дата обращения: 26 мая 2025 г.).

¹²clij2k, URL: <https://github.com/ermattt/clij2k> (дата обращения: 26 мая 2025 г.).

¹³tree-sitter-kotlin, URL: <https://github.com/fwcd/tree-sitter-kotlin> (дата обращения: 26 мая 2025 г.).

peft, optuna, transformers, TensorFlow, PyTorch.

1.3. Обзор метрик оценки качества декомпиляции

Для количественной оценки качества декомпилированного кода использовались структурные, языковые и энтропийные метрики [4, 7, 9, 14, 15, 17, 18, 19].

- *Program Size*. Число вершин в AST программы.
- *Conditional Statements*. Число `if/if-else`.
- *Conditional Complexity*. Для каждого условного выражения e со множеством узлов $N(e)$:

$$w(e) = \sum_{n \in N(e)} w(n), \quad w(n) = \begin{cases} 1, & \text{булев литерал или переменная,} \\ 0.5, & \text{унарный оператор,} \\ 1, & \text{бинарный } \{\&\&, ||\}. \end{cases}$$

Среднее по всем m условиям: $\text{CondComplexity} = \frac{1}{m} \sum_{e=1}^m w(e)$.

- *Abrupt Control Flow*. Число `break/continue`.
- *Labeled Blocks*. Число помеченных блоков.
- *Local Variables*. Число объявлений локальных переменных.
- *Cyclomatic Complexity*. Для графа потока управления $G = (N, E_G)$ и P компонент связности:

$$M = |E_G| - |N| + 2P.$$

- *Halstead Vocabulary*.

$$\eta = n_1 + n_2,$$

где n_1 — уникальные операторы, n_2 — уникальные операнды.

- *Halstead Length.*

$$N = N_1 + N_2,$$

где N_1 — все операторы, N_2 — все операнды.

- *Halstead Volume.*

$$V = N \log_2(\eta).$$

- *Halstead Difficulty.*

$$D = \frac{n_1}{2} \cdot \frac{N_2}{n_2}.$$

- *Halstead Effort.*

$$E = D \cdot V.$$

- *Halstead Time.*

$$T = \frac{E}{18} \text{ секунд.}$$

- *Halstead Bugs.*

$$B = \frac{V}{3000}.$$

- *Chapin Q.* Пусть P, M, C, T — количества переменных групп `Input`, `Modified`, `Control`, `Temporary`.

$$Q = P + 2M + 3C + 0.5T.$$

- *Pivovarsky Complexity.* Пусть $v^*(G)$ — число управляющих конструкций, P_i — уровень вложенности i -й управляющей конструкции в дереве синтаксического разбора.

$$N(G) = v^*(G) + 1 + \sum_{i=1}^{v^*(G)} P_i.$$

Для оценки идиоматичности и статистических свойств декомпилированного кода использовались вероятностные и информационно-теоретические метрики [8, 11].

- *Conditional Entropy.* Условная энтропия $H(X|Y)$ измеряет сред-

ную неопределённость переменной X при известной Y :

$$H(X|Y) = - \sum_{x,y} p(x,y) \log_2 p(x|y).$$

- *Cross Entropy*. Оценивает расхождение между истинным распределением p и аппроксимирующим распределением $\hat{p}$:

$$H(p, \hat{p}) = - \sum_x p(x) \log \hat{p}(x).$$

- *Perplexity*. Перплексия определяется как экспонента от кросс-энтропии:

$$\text{PPL} = 2^{H(p, \hat{p})}.$$

- *Kullback–Leibler divergence*. Асимметричная мера различия между распределениями p и q :

$$D_{\text{KL}}(p \parallel q) = \sum_x p(x) \log_2 \frac{p(x)}{q(x)}.$$

- *Jensen–Shannon divergence*. Симметричная и сглаженная модификация KL-дивергенции:

$$\text{JSD}(p \parallel q) = \frac{1}{2} D_{\text{KL}}(p \parallel m) + \frac{1}{2} D_{\text{KL}}(q \parallel m), \quad \text{где } m = \frac{1}{2}(p + q).$$

1.4. Обзор математических методов

Для анализа признаков применялись методы, направленные на устранение избыточности и обеспечение корректности последующего анализа [1, 6].

- После нормализации с помощью MinMax-масштабирования исключаются признаки, у которых дисперсия ниже порогового значения θ :

$$\text{Var}(x'_j) < \theta$$

Это позволяет удалить малополезные признаки с ограниченной вариативностью.

- Для устранения мультиколлинеарности рассчитывается матрица корреляций. Признаки, для которых абсолютное значение корреляции превышает порог ρ , образуют пары кандидатов на удаление:

$$|\text{corr}(x_j, x_k)| > \rho$$

Итеративно исключается один признак из каждой сильно коррелирующей пары, выбирая набор признаков, удаление которых обеспечивает наибольшее снижение мультиколлинеарности. Процесс повторяется при убывающих порогах ρ , до тех пор, пока корреляционная матрица оставшихся признаков не становится невырожденной:

$$\text{rank}(\text{corr}(X)) = \dim(X)$$

2. Постановка задачи

Цель данной работы заключается в разработке подхода и инструмента для декомпиляции Kotlin. Для достижения этой цели сформулированы следующие задачи.

1. Анализ существующих решений в области декомпиляции Java, конвертации Java в Kotlin и их комбинаций.
2. Разработка набора тестовых примеров и набора метрик для оценки эффективности подходов декомпиляции.
3. Разработка подхода для декомпиляции байт-кода JVM в Kotlin.
4. Реализация инструмента на основе разработанного подхода.
5. Проведение экспериментальной оценки инструмента с использованием реализованных метрик.

Реализация этих задач позволит создать инструмент для декомпиляции байт-кода JVM в Kotlin, что упростит процессы реверс-инжиниринга и миграции приложений.

3. Описание решения

3.1. Тестовые наборы и оценка существующих подходов

3.1.1. Разработка тестовых примеров

В рамках решения разработаны тестовые примеры, демонстрирующие особенности языка Kotlin, отличные от Java или реализуемые в Kotlin специфическим образом. Тесты охватывают широкий спектр возможностей языка, что позволяет оценить корректность преобразования и выявить потенциальные проблемы при декомпиляции.

1. `HelloKotlin` — вывод строки в консоль.
2. `HigherOrderFunction` — использование функций высшего порядка.
3. `Result` — работа с типом `Result`.
4. `Exception` — обработка исключительных ситуаций с помощью конструкции `try-catch`.
5. `Extension` — расширение существующих типов с методом `greet` для строк.
6. `ScopeFunctions` — использование функций области видимости (`apply`, `let`, `run`, `also`, `with`).
7. `Destructuring` — деструктуризация данных для извлечения значений из объектов.
8. `Recursion` — рекурсивное вычисление факториала.
9. `SafeCast` — безопасное приведение типов с оператором `as?`.
10. `ClassDelegation` — делегирование реализации интерфейсов с помощью ключевого слова `by`.

11. `Reified` — использование `inline`-функций с параметрами `reified` для получения информации о типе.
12. `Lambda` — простое лямбда-выражение для обработки списка чисел.
13. `SmartCast` — работа смарт-приведения после проверки типа переменной.
14. `TailRecursion` — использование ключевого слова `tailrec` для оптимизации рекурсивных вызовов.
15. `OperatorOverloading` — перегрузка операторов для определения собственного поведения оператора `+`.
16. `List` — работа со списками и перебор элементов.
17. `Singleton` — паттерн одиночка с ключевым словом `object`.
18. `Nothing` — функция, которая всегда выбрасывает исключение с типом `Nothing`.
19. `Elvis` — использование оператора Элвиса для обработки возможного `null`.
20. `CompanionObject` — использование `companion object` для создания статических методов.
21. `Loop` — цикл с использованием диапазона.
22. `Generic` — применение обобщений (generics) в Kotlin.
23. `GenericsInOutWhere` — использование ключевых слов `in` и `out` и ограничений `where` для обобщений.
24. `DataClass` — описание классов данных с ключевым словом `data`.
25. `PropertyDelegation` — делегирование свойств с `Delegates.observable`.
26. `SequenceProcessing` — работа с последовательностями.

27. `IfElse` — использование условного оператора `if-else`.
28. `LazyInitialization` — ленивое инициализирование свойств.
29. `Builder` — паттерн билдер с лямбда-расширением для создания списка.
30. `NullValueSafety` — обработка `null` с помощью безопасного вызова.
31. `Person` — создание класса с полями и методом.
32. `LabeledBlocks` — использование меток для управления потоком выполнения.
33. `AbruptControlFlow` — использование операторов прерывания (`break`, `continue`).

3.1.2. Подбор датасетов

В рамках исследования проведён поиск открытых датасетов с фрагментами кода на языке Kotlin. Отбор выполнен с учётом доступности лицензий и репрезентативности данных. Выбраны следующие датасеты:

- `KStack-clean`¹⁴ — подмножество датасета `KStack` с 25 000 Kotlin-фрагментами, отобранными по критерию ценности для изучения алгоритмических концепций.
- `KExercises`¹⁵ — синтетический датасет с 15 000 упражнениями на языке Kotlin, сформированными на основе `HumanEval`-формата и сгенерированными моделью GPT-3.5.

¹⁴`KStack-clean`, URL: <https://huggingface.co/datasets/JetBrains/KStack-clean> (дата обращения: 26 мая 2025 г.).

¹⁵`KExercises`, URL: <https://huggingface.co/datasets/JetBrains/KExercises> (дата обращения: 26 мая 2025 г.).

3.1.3. Применение существующих подходов декомпиляции Kotlin к тестам

Тестовые случаи декомпилированы из байт-кода в Java с использованием CFR, JD-GUI или Fernflower. Полученные Java-файлы затем преобразованы в Kotlin с помощью J2K, CodeConvert или ChatGPT (4o). Также выполнено преобразование байт-кода в Kotlin через ChatGPT.

Поскольку J2K не имеет командного интерфейса и связан с графической оболочкой IDE, для обеспечения воспроизводимости каждый тест выполнялся в двух сценариях:

- С минимальным вмешательством, ограниченным автоматически-исправлениями среды разработки для устранения влияния UI-инструмента J2K.
- Полностью автоматизированный — без вмешательства человека.

Результаты экспериментов представлены в табл. 2.

Сценарий	CFR	JD-GUI	Fernflower	Bytecode
ChatGPT (4o)	93.94%	96.97%	93.94%	100.00%
J2K, с правками	57.58%	54.55%	57.58%	—
J2K, без правок	57.58%	45.45%	27.27%	—

Таблица 2: Процент успешных компиляций полученного кода.

3.1.4. Качественная оценка существующих подходов к декомпиляции Kotlin

Краткое обобщение этих наблюдений представлено в табл. 3.

Комбинация	Преимущества	Недостатки
Байт-код + ChatGPT	Часто восстанавливается структура и логика кода, включая поддержку функций <code>buildList</code> , <code>fail</code> и лямбд. Корректно обрабатывается тип <code>Nothing</code> в некоторых сценариях.	Иногда теряются отдельные конструкции (например, блок <code>else</code>), меняются условия или возникают неоднозначности в типах.
CFR + ChatGPT	Часто восстанавливается структура и логика кода.	Может добавлять временные переменные или изменять условные блоки. Иногда наблюдаются неточности в типах, например, замена проверки типа на прямое приведение. Возможны удаления или изменения отдельных проверок (<code>if</code> , <code>else</code>).
Fernflower + ChatGPT	Часто восстанавливается структура кода с поддержкой типичных паттернов Java/Kotlin (<code>data class</code> , лямбды).	Может генерировать более громоздкую логику, добавлять лишние переменные или комментировать сложные участки. Иногда отсутствуют конструкции (<code>apply</code> , <code>let</code> , <code>run</code> , <code>with</code>) или возникают необычные параметры (<code>\$this\$</code>).

Комбинация	Преимущества	Недостатки
JD-GUI + ChatGPT	Часто восстанавливается структура и логика кода, особенно для небольших проектов. Удобен для быстрой оценки байт-кода и восстановления лямбд.	Может добавлять вспомогательные объекты, изменять порядок объявлений или переименовывать сущности. В некоторых случаях изменяет способ инициализации.
CFR + CodeConvert	Сохраняется читаемость и структура кода.	Возможны дублирования, коллизии имен, дополнительные проверки и некорректная обработка лямбд.
Fernflower + CodeConvert	Сохраняется читаемость кода.	Могут появляться неоптимальные циклы, синтетические методы или переменные, а также потеря логики при делегировании.
JD-GUI + CodeConvert	Сохраняется структура кода.	Иногда теряются расширения коллекций, наблюдаются переименования и дублирование кода.
CFR + J2K	Сохраняется структура кода.	Могут появляться лишние блоки или исчезать части кода, разворачивание лямбд в синтетический код и сложности при обработке вызовов.

Комбинация	Преимущества	Недостатки
Fernflower + J2K	Частично сохраняется структура кода.	Генерация вспомогательных методов (copy, componentX), отсутствие некоторых конструкций Kotlin и оборачивание кода в анонимные объекты.
JD-GUI + J2K	Частично сохраняется структура кода.	Увеличение детализации, переименование сущностей, нарушение порядка объявлений, закомментированный или обёрнутый код и усложнение логики при работе с лямбдами.

Таблица 3: Качественная оценка существующих подходов к декомпиляции Kotlin.

Первичная оценка полученных декомпиляций показывает, что каждая комбинация инструментов имеет свои преимущества и ограничения. Для последующего анализа требуется проведение дополнительных экспериментов и количественной оценки.

Полная сравнительная таблица результатов с примерами доступна на GitHub: <https://github.com/akabynda/KotlinDecompilerSyntheticTests/tree/master/src/testResults>.

3.1.5. Реализация метрик для оценки качества декомпиляции

Для оценки качества декомпилированного кода разработана система сбора метрик на языке Python. Архитектура проекта предполагает использование декораторов и автоматическую регистрацию функций метрик. Метрики основаны на анализе синтаксического дерева, энтропийных свойствах текста и модели языка Kotlin.

Все функции метрик оформляются с помощью декоратора

`@metric(name)` и автоматически добавляются в реестр при импорте модуля. Реализации регистрации метрик и метрики `Program Size` приведены в лист. 1 и 2.

Листинг 1: Регистрация метрик в пакете.

```
registry: Dict[str, Callable] = {}

def metric(name: str):
    def wrapper(fn: Callable):
        registry[name] = fn
        return fn

    return wrapper
```

Листинг 2: Метрика Program Size.

```
@metric("Program Size")
def program_size(root: Node) -> int:
    def _count(node: Node) -> int:
        return 1 + sum(_count(c) for c in node.children)
    return _count(root)
```

Кроме структурных метрик, реализованы сравнительные метрики на основе распределения токенов: кросс-энтропия (CE), дивергенция Кульбака-Лейблера (KL), расстояние Йенсена-Шеннона (JSD), перплексия (PPL), условная энтропия на основе биграмм (Conde).

Реализация расчёта Cross-Entropy приведена в лист. 3.

Листинг 3: Реализация Cross-Entropy.

```
def cross_entropy(self, p_src: str, q_src: str, eps=1e-12):
    p, q = self.dist(p_src), self.dist(q_src)
    return sum(q.get(k, eps) *
               math.log2(1 / max(p.get(k, eps), eps))
               for k in self.merge(p, q))
```

Кроме сравнения распределения оригинала с распределением результата декомпиляции, выполняется сравнение с распределением языка Kotlin. Для построения модели языка использовались KStack-clean и KExercises. Для них собраны частотные распределения униграмм и биграмм. Итоговая модель сохраняется в формате JSON и используется для расчёта:

- LM_CE — кросс-энтропия между моделью и файлом.
- LM_KL, LM_PPL, LM_JSD, LM_CondE — аналогичные вышеописанным, но относительно модели.

Реализация расчёта Cross-Entropy относительно модели языка приведена в лист. 4.

Листинг 4: Cross-Entropy относительно модели языка.

```
def cross_entropy_lang(self,
                       lang_dist: dict[str, float],
                       src: str,
                       eps=1e-12):
    q = self.dist(src)
    return sum(q.get(k, eps) *
               math.log2(1 / max(lang_dist.get(k, eps), eps))
               for k in self.merge(lang_dist, q))
```

Реализация всех метрик и вспомогательных функций доступна в репозитории: <https://github.com/akabynda/KotlinDecompiler/tree/master/metrics>.

3.1.6. Сокращение размерности пространства признаков

Таким образом, для каждого примера вычислено 30 количественных признаков. Такое количество затрудняет интерпретацию и анализ. В связи с этим исключены признаки с дисперсией меньше 0.01 после масштабирования по минимуму и максимуму, а также сильно коррелирующие столбцы, чтобы избежать вырождения корреляционной матрицы [1, 6].

Отбор проведён не только на сравнительно небольшом тестовом наборе, но и на более репрезентативных датасетах `KStack-clean` и `KExercises`, что позволило получить более устойчивый и обобщённый набор из 8 признаков: `Conditional Entropy`, `Conditional Complexity`, `Halstead Distinct Operators`, `JSD`, `KL`, `Cross-Entropy (LM)`, `Conditional Entropy (LM)`, `KL (LM)`.

Данный список может использоваться для последующего анализа, построения моделей и визуализации в сниженной размерности без существенной потери информации.

3.1.7. Количественная оценка существующих подходов к декомпиляции Kotlin

В результате вычисления метрик для набора разработанных тестовых примеров получены усреднённые значения для каждого подхода. Результаты количественной оценки представлены на рис. 1.

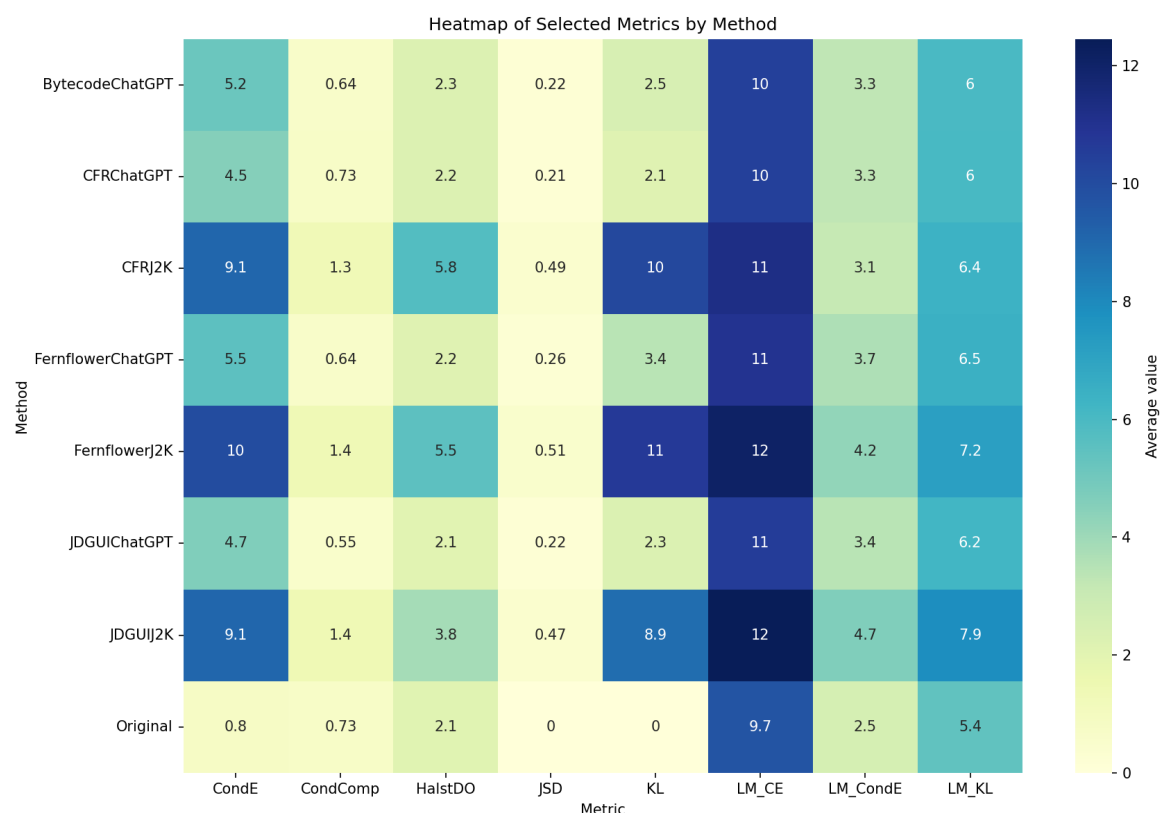


Рис. 1: Средние значения ключевых метрик для существующих подходов.

- **CondE.** Высокие значения метрики наблюдаются у **FernflowerJ2K** и **JDGUIJ2K**, что указывает на значительное различие в распределении признаков между декомпиляцией и оригинальным кодом. Комбинации с **ChatGPT** демонстрируют более низкие значения, указывая на сохранение близости к исходной структуре данных.
- **Conditional Complexity.** Минимальные значения фиксируются у **BytecodeChatGPT** и **FernflowerChatGPT**, что может отражать упрощение логики ветвлений. Наибольшие значения отмечаются у **FernflowerJ2K** и **JDGUIJ2K**, что указывает на увеличение сложности условий после преобразования.
- **Halstead Distinct Operators.** Минимальные значения наблюдаются у **FernflowerChatGPT** и **CFRChatGPT**, что указывает на сохранение разнообразия операторов, близкого к оригиналу. Наибольшие значения отмечаются у **FernflowerJ2K** и **JDGUIJ2K**, что

свидетельствует о добавлении различных операторов.

- JSD. Для всех вариантов с ChatGPT показатель близок к минимальному уровню, что указывает на небольшое отклонение от оригинального кода. Значительное увеличение наблюдается у JDGUIJ2K и FernflowerJ2K, что указывает на значительное изменение структуры кода.
- KL. Минимальные значения фиксируются у BytecodeChatGPT и CFRChatGPT, что указывает на высокую схожесть с оригиналом. Наибольшие отклонения наблюдаются у FernflowerJ2K и JDGUIJ2K, что свидетельствует о значительных структурных различиях.
- LM_CE. Наименьшие значения наблюдаются у BytecodeChatGPT, что указывает на высокое соответствие распределению токенов языковой модели. Показатели, близкие к оригиналу или ниже, могут указывать на более идиоматичный стиль. Наибольшие значения у FernflowerJ2K демонстрируют значительное расхождение с моделью.
- LM_CondE. Минимальные значения характерны для BytecodeChatGPT, что указывает на схожесть с моделью языка. Наибольшие значения у JDGUIJ2K и FernflowerJ2K свидетельствуют о значительном отклонении от идиоматичного синтаксиса.
- LM_KL. Наименьшие значения фиксируются у BytecodeChatGPT, что подтверждает высокую схожесть с моделью языка. Наибольшие отклонения наблюдаются у JDGUIJ2K и FernflowerJ2K, что указывает на значительное расхождение с языковой моделью.

Таким образом, можно заключить, что большинство комбинаций с использованием ChatGPT (4o) демонстрируют показатели метрик, близкие к оригинальному коду, а в некоторых случаях более близкие к идиоматичному стилю. Преобразования с использованием J2K приводят

к увеличению значений ключевых метрик, что указывает на внесение избыточной сложности и ухудшение читаемости кода.

3.2. Разработка подхода к декомпиляции Kotlin

Полученные результаты подтверждают потенциал LLM-подходов для декомпиляции, так как структура и стиль кода остаются близкими к оригиналу. Однако остаются ограничения: ChatGPT является универсальной моделью, не адаптированной для работы с байт-кодом и языком Kotlin. Это приводит к потенциальным ошибкам, нестабильности результатов и отсутствию контроля над архитектурой модели и процессом обучения. Кроме того, API является платным.

Следовательно, необходима разработка специализированной модели для декомпиляции Kotlin, обученной на парах байт-код — исходный код с учётом структурной и идиоматической точности.

3.2.1. Формирование датасета из пар байт-код — исходный код

Исходные данные получены из датасетов KStack-clean и KExercises. Для их загрузки использовалась библиотека datasets, после чего исходные файлы с расширением .kt сохранялись в локальной директории. Каждый пример из KStack-clean привязан к репозиторию с уникальной парой owner-repo и коммитом, что обеспечивает воспроизводимость и точное сопоставление исходного файла с конкретной версией.

Для каждого репозитория создавалась структура:

`{owner}__{repo}__{commit}/<путь_к_файлу>`

где {owner} — владелец, {repo} — название репозитория, {commit} — сокращённый SHA-идентификатор. Такая организация учитывает зависимости между файлами и позволяет выполнять компиляцию в рамках одного проекта. В то же время каждый репозиторий KExercises состоит из одного файла.

Компиляция выполнялась с помощью `kotlinc (-jvm-target 23)`. При ошибках формировался временный проект **Gradle** с автоматически определяемыми зависимостями, извлекаемыми из импортов исходных файлов, с использованием сервиса **Maven Central**.

В процессе подготовки выполнялась автоматическая корректировка импортов и удаление дубликатов. Все ошибки компиляции сохранялись в файл `compile_errors.log`.

Из 15 000 репозиториев датасета **KExercises** успешно скомпилированы 11 654, а из 3 342 репозиториев датасета **KStack-clean** — 786.

Байт-код каждого репозитория сохраняется в отдельной директории. Для сопоставления исходных `.kt`-файлов и соответствующих им `.class`-файлов строится двухуровневый индекс: по пути относительно репозитория и по структуре пакетов, что позволяет корректно сопоставлять даже в случаях различий между структурой исходников и байт-кода. Для каждого найденного сопоставления выполняется преобразование соответствующих `.class`-файлов с помощью `javap -c -p`. Если декомпиляция всех классов из пары пуста, такая пара пропускается.

Итоговые данные сериализуются в формате **JSONL**, где каждая запись содержит:

- `kt_path`: путь к исходному `.kt`-файлу относительно директории `originals`;
- `kt_source`: исходный текст файла;
- `classes`: список декомпилированных классов с ключами `class_path` (относительный путь к файлу класса), `javap` (вывод декомпилятора), `javap_err` (вывод ошибок декомпилятора).

Процесс формирования финального датасета выполняется параллельно для ускорения, с контролем уникальности и отсутствия дублирующихся сопоставлений. Сохраняются только те пары, для которых удалось получить хотя бы один непустой вывод декомпилятора.

Объединённый набор данных, полученный из двух датасетов, разделён на обучающую и тестовую части, при этом 10% отведено для

тестирования.

Датасеты опубликованы по следующим ссылкам:

- <https://huggingface.co/datasets/akabynda/KExercises-KStack-clean-bytecode>
- <https://huggingface.co/datasets/akabynda/KExercises-bytecode>
- <https://huggingface.co/datasets/akabynda/KStack-clean-bytecode>

3.2.2. Подбор моделей для кодогенерации

Объединённый датасет, полученный из KStack-clean и KExercises, имеет ограниченный объём в 13 528 Kotlin-файлов, что не позволяет обучить большую языковую модель с нуля. В связи с этим выбран подход дообучения существующих моделей. Критериями отбора моделей для дообучения являлись открытая лицензия и возможность запуска на графическом процессоре с 8 ГБ видеопамати при условии 4-битной квантизации, чтобы обеспечить доступность на платформах с ограниченными ресурсами.

Были отобраны модели, которые входят в лидирующие позиции в области генерации кода¹⁶ и модели от JetBrains.

- Qwen/Qwen2.5-Coder-0.5B, Qwen2.5-Coder-0.5B-Instruct
- Qwen/Qwen2.5-Coder-1.5B, Qwen2.5-Coder-1.5B-Instruct
- JetBrains/deepseek-coder-1.3B-kexer
- deepseek-ai/deepseek-coder-1.3b-base,
deepseek-ai/deepseek-coder-1.3b-instruct
- Qwen/Qwen2.5-Coder-3B, Qwen2.5-Coder-3B-Instruct

¹⁶Big Code Models Leaderboard, URL: <https://huggingface.co/spaces/bigcode/bigcode-models-leaderboard> (дата обращения: 26 мая 2025 г.).

- JetBrains/Mellum-4b-base
- deepseek-ai/deepseek-coder-6.7b-instruct,
deepseek-ai/deepseek-coder-6.7b-base
- JetBrains/deepseek-coder-6.7B-kexer
- deepseek-ai/deepseek-coder-7b-base-v1.5,
deepseek-ai/deepseek-coder-7b-instruct-v1.5
- Qwen/Qwen2.5-Coder-7B, Qwen2.5-Coder-7B-Instruct
- JetBrains/CodeLlama-7B-Kexer,
JetBrains/CodeLlama-7B-KStack-clean,
JetBrains/CodeLlama-7B-KStack
- codellama/CodeLlama-7b-hf,
codellama/CodeLlama-7b-Instruct-hf

3.2.3. Декомпиляция с использованием выбранных моделей

Для оценки декомпиляции протестированы выбранные языковые модели, каждая из которых использовалась по следующему сценарию. В качестве экспериментальных данных выбраны 100 примеров с наименьшей длиной байт-кода из тестовой части датасета KExercises-KStack-clean-bytecode для более быстрого инференса. Данный набор образует 97 репозиторий и используется для анализа качества декомпиляции.

1. **Загрузка модели:** Каждая модель и токенизатор загружаются с помощью библиотеки `transformers`. По умолчанию применяется 4-битная квантизация для минимизации потребления видеопамати через `BitsAndBytesConfig`. В случае ошибки загрузки выполняется переход к 8-битной квантизации или полной загрузке модели.

2. **Построение промта:** Для каждого примера на основе байт-кода формируется текстовый запрос на декомпиляцию. Промт стандартизирован: *“Convert the following JVM byte-code into Kotlin source. Output Kotlin code only.”* Для моделей с поддержкой chat-интерфейса (например, Qwen) используется диалоговая схема промта, совместимая с их chat-template.
3. **Буферизация и группировка:** Запросы формируются пакетами, размер которых определяется динамически на основе доступной видеопамати и средней длины входных данных для каждой модели (автоматически оценивается перед запуском). Для каждого пакета применяется padding токенов.
4. **Генерация:** Ответы моделей генерируются методом `transformers.generate` без выборки (`do_sample=False`, `num_beams=1`) для обеспечения воспроизводимости результатов. Максимальная длина ответа `max_new_tokens` рассчитывается эвристически:
$$\text{max_new_tokens} = 1.5 \times \text{percentile}(\text{kt_sources_tokens}, 95),$$
где `kt_sources_tokens` — множество длин токенизированных исходных файлов тестового подмножества.
5. **Парсинг и сохранение:** Сгенерированные ответы фильтруются функцией пост-обработки, выделяющей Kotlin-код с помощью регулярных выражений. Каждая запись сериализуется в JSONL-файл с ключами: `kt_path` (относительный путь к примеру) и именем модели.
6. **Очистка памяти:** После генерации и сохранения очередного пакета выполняется очистка GPU-памяти и выгрузка модели из VRAM для оптимизации использования ресурсов.
7. **Восстановление/дозапись:** Для предотвращения потери данных при повторном запуске реализована поддержка дозаписи: для

уже обработанных примеров результаты не пересчитываются.

3.2.4. Количественный анализ моделей для декомпиляции Kotlin

После генерации с использованием выбранных моделей для каждого набора выполняется компиляция по тому же сценарию, что и при подготовке исходного датасета: исходники группируются по репозиториям, автоматически определяются зависимости, после чего выполняется попытка компиляции. Для каждой модели отдельно фиксируется число успешно скомпилированных решений, что позволяет учитывать покрытие при расчёте итоговых метрик.

Далее для каждого набора декомпилированных исходников рассчитываются структурные, языковые и энтропийные метрики, а также показатели успешности компиляции. Все результаты сохраняются в формате **CSV** для дальнейшего анализа и визуализации.

Средние значения ключевых метрик для различных моделей приведены на рис. 2.

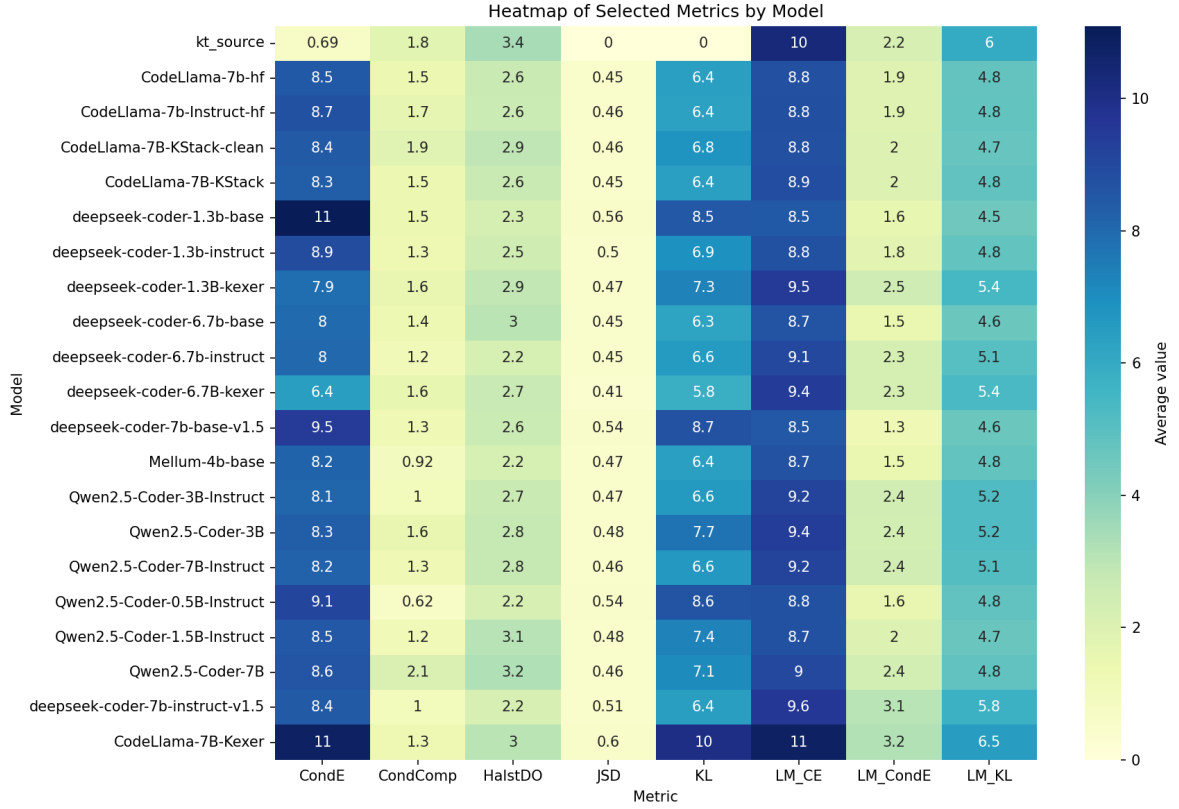


Рис. 2: Средние значения ключевых метрик для различных моделей.

Для комплексной оценки каждой модели рассчитываются расстояния до эталонного вектора исходного кода. Пусть для каждой модели i задан вектор признаков:

$$\mathbf{x}_i = [x_{i1}, x_{i2}, \dots, x_{i8}],$$

а для исходного кода — эталонный вектор:

$$\mathbf{r} = [r_1, r_2, \dots, r_8].$$

При этом для эталона метрики CondE, JSD, KL зануляются, так как они уже являются вычисленными относительно оригинала для каждого $\mathbf{x}_i$.

Вычисляются следующие расстояния [2]:

$$\begin{aligned}
d_{\text{euclidean}}(\mathbf{x}_i, \mathbf{r}) &= \sqrt{\sum_{j=1}^8 (x_{ij} - r_j)^2}, \\
d_{\text{manhattan}}(\mathbf{x}_i, \mathbf{r}) &= \sum_{j=1}^8 |x_{ij} - r_j|, \\
d_{\text{cosine}}(\mathbf{x}_i, \mathbf{r}) &= 1 - \frac{\mathbf{x}_i \cdot \mathbf{r}}{\|\mathbf{x}_i\| \|\mathbf{r}\|}, \\
d_{\text{chebyshev}}(\mathbf{x}_i, \mathbf{r}) &= \max_j |x_{ij} - r_j|.
\end{aligned}$$

Для учёта различий в количестве успешно скомпилированных примеров (n_i) применяется масштабирование с учётом покрытия датасета:

$$c_i = \frac{n_i}{n_{\text{orig}}}.$$

Для каждого расстояния d определяется масштабированное значение:

$$d_i^{\text{cov}} = \frac{d(\mathbf{x}_i, \mathbf{r})}{\max(c_i, \varepsilon)}, \quad \varepsilon = 10^{-3}.$$

Особое внимание уделяется расстоянию Чебышёва ($d_{\text{chebyshev}}$), которое отражает наибольшее расхождение хотя бы по одной метрике. Минимизация этого расстояния позволяет выявить модели, близкие к эталону по всем метрикам, что делает его приоритетным критерием для итоговой оценки. Результаты сохраняются в файл **CSV** для дальнейшего использования и доступны по ссылке: https://github.com/akabynda/KotlinDecompiler/blob/master/best_models.csv.

Таким образом, модель с минимальным значением $d_{\text{chebyshev}}^{\text{cov}}$ определяется как наиболее качественная с точки зрения совокупности всех метрик. Для последующего дообучения выбраны модели с минимальным значением этой метрики среди компактных (до 1.5B параметров), так как они обеспечивают более высокую скорость инференса и доступны для большего числа вычислительных систем с ограниченными ресурсами [3]. Такими моделями являются **Qwen2.5-Coder-0.5B-Instruct** ($d_{\text{chebyshev}}^{\text{cov}} = 10.99$, 84 успешные компиля-

ции из 97) и `deepseek-coder-1.3b-instruct` ($d_{\text{chebyshev}}^{\text{cov}} = 8.81$, 92 успешные компиляции из 97).

3.2.5. Дообучение моделей для декомпиляции

Для последующего этапа дообучения выбраны модели `Qwen2.5-Coder-0.5B-Instruct` и `deepseek-coder-1.3b-instruct`, продемонстрировавшие наилучшее сочетание качества и компактности на этапе тестирования.

Из-за ограниченных вычислительных ресурсов подбор гиперпараметров выполняется на основе значений, использованных при подготовке модели `JetBrains/deepseek-coder-1.3B-kexer`¹⁷. `total_batch_size` уменьшается для ускорения обучения и минимизации риска переобучения.

Процесс дообучения включает следующие этапы:

1. **Загрузка и препроцессинг данных:** Используется тренировочный сплит датасета `KExercises-KStack-clean-bytecode`, загруженный через `datasets`. Каждый пример конвертируется в диалоговый формат: промт включает задание на декомпиляцию и байт-код, а целевым ответом является исходный код на Kotlin, выделенный в Markdown-блоке.
2. **Токенизация:** Последовательности токенизируются с усечением по длине, равной 95-му перцентилю распределения токенов по тренировочному сплиту.
3. **Квантизация:** Для повышения эффективности использования памяти применяется 4-битная квантизация [16] с параметрами:

```
bnb_4bit_quant_type="nf4",  
bnb_4bit_use_double_quant=True,  
bnb_4bit_compute_dtype=torch.float16.
```

¹⁷`deepseek-coder-1.3B-kexer`, URL: <https://huggingface.co/JetBrains/deepseek-coder-1.3B-kexer> (дата обращения: 26 мая 2025 г.).

4. **LoRA-адаптация:** Дообучение проводится методом LoRA [12] с параметрами ($r = 16$, $\alpha = 8$, $\text{dropout} = 0.05$), что позволяет адаптировать модель без полной настройки всех весов.
5. **Обучение:** В качестве оптимизатора используется AdamW [13]. Гиперпараметры:

```
train_epochs = 4,  
grad_accumulation = 32,  
learning_rate = 10-4,  
weight_decay = 0.05,  
warmup_ratio = 0.1,  
max_grad_norm = 0.7.
```
6. **Воспроизводимость:** Фиксируются random seed для всех ключевых библиотек.
7. **Сохранение результатов:** Сохраняются веса LoRA-адаптера и обновлённый токенизатор, чтобы их можно было применять к исходной модели.

Итоговые версии моделей и токенизаторов опубликованы по следующей ссылке: <https://github.com/akabynda/KotlinDecompiler/tree/master/models>.

4. Эксперимент

Данный раздел описывает процесс проверки эффективности предложенного подхода к декомпиляции Kotlin. Эксперимент проводится для количественного сравнения качества различных моделей, а также для анализа влияния дообучения.

4.1. Условия эксперимента

Эксперимент выполняется на устройстве с видеокартой NVIDIA RTX 3070 (8 ГБ видеопамяти), процессором Intel Core i7-12700KF и 32 ГБ оперативной памяти. Используется операционная система Windows 11, Python 3.12 и CUDA 12.1. Все версии библиотек, необходимые для запуска, фиксируются в `requirements.txt` проекта.

В качестве тестового набора используется тестовый сплит созданного датасета `KExercises-KStack-clean-bytecode`. Для ускорения инференса и обеспечения воспроизводимости выбираются 100 примеров с наименьшей длиной байт-кода (всего 97 уникальных репозиториях).

4.2. Исследовательские вопросы

RQ1: Позволяет ли дообучение улучшить качество декомпиляции в сравнении с исходными моделями без дообучения?

RQ2: Способны ли компактные модели после дообучения обеспечить качество, сопоставимое или превышающее качество крупных моделей без дообучения?

4.3. Метрики

Для количественного анализа качества декомпиляции используются следующие метрики:

- **Базовые метрики:** *Conditional Complexity*, *Halstead Distinct Operators*, *JSD*, *KL*, *Conditional Entropy*, *Cross-Entropy (LM)*, *Conditional Entropy (LM)*, *KL (LM)*.

- **Сводные показатели:** евклидово, манхэттенское, косинусное и расстояние Чебышёва между медианными векторами метрик декомпилированных результатов и эталонного кода.

Приоритетной метрикой является масштабированное расстояние Чебышёва, так как оно позволяет объективно оценить наибольшее отклонение: чем меньше это расстояние, тем лучше соответствие по всем метрикам в совокупности.

4.4. Результаты

В результате эксперимента из всех протестированных моделей лучшие результаты по соотношению качества и компактности показывают `Qwen2.5-Coder-0.5B-Instruct` (масштабированное расстояние Чебышёва 10.99, 84 успешные компиляции) и `deepseek-coder-1.3b-instruct` (8.81, 92 успешные компиляции). После дообучения на тренировочном сплите того же датасета эти модели демонстрируют улучшенные результаты:

- `deepseek-coder-1.3b-instruct-Finetuned`: 6.64, 90 успешных компиляций из 97.
- `Qwen2.5-Coder-0.5B-Instruct-Finetuned`: 7.51, 90 успешных компиляций из 97.

Для сравнения: крупная модель `Qwen2.5-Coder-7B-Instruct` без дообучения достигает 8.72 при 92 успешных компиляциях из 97. Это подтверждает, что дообучение позволяет компактным моделям достигать уровня качества, сопоставимого или превышающего результаты крупных моделей без дообучения.

Средние значения ключевых метрик для различных моделей после дообучения представлены на рис. 3.

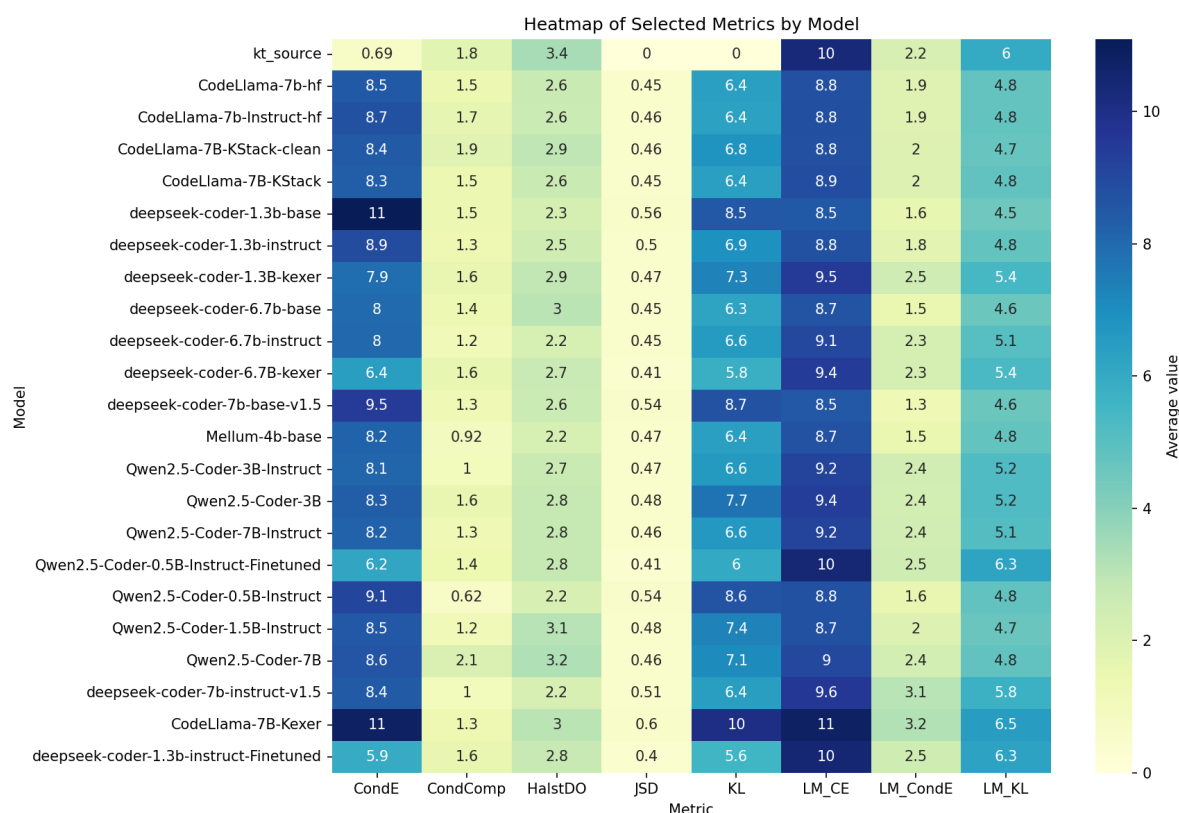


Рис. 3: Средние значения ключевых метрик для различных моделей после дообучения.

4.5. RQ1

Дообученные модели демонстрируют снижение отклонений по всем ключевым метрикам по сравнению с исходными моделями. Это подтверждает, что дообучение позволяет улучшить качество декомпиляции (ответ: *да*).

4.6. RQ2

Компактные модели после дообучения показывают сопоставимые или лучшие результаты по метрикам качества относительно крупных моделей без дообучения, при этом обеспечивая меньшие требования к памяти и большую скорость инференса (ответ: *да*).

4.7. Обсуждение результатов

Результаты показывают, что дообучение позволяет повысить качество декомпиляции байт-кода JVM в Kotlin. Выбор компактных моделей обоснован их доступностью для платформ с ограниченными вычислительными возможностями. Основное преимущество подхода заключается в возможности адаптации моделей к специфике задачи без необходимости обучения с нуля.

4.8. Угрозы нарушения корректности

Потенциальные угрозы достоверности результатов связаны с ограниченным объёмом тестового датасета и возможными особенностями репрезентативности. Чтобы минимизировать эти риски, используется разнообразный набор исходных репозиторий и фиксированные параметры генерации. Повторяемость экспериментов обеспечивается единообразием процедур компиляции и измерения метрик.

4.9. Воспроизводимость эксперимента

Весь исходный код эксперимента, скрипты дообучения и генерации, а также подготовленные датасеты доступны в репозитории. Для запуска эксперимента требуется установить зависимости, указанные в **README** и **requirements.txt**, и воспроизвести шаги дообучения и генерации. Все ключевые параметры и пути к данным явно указаны в конфигурационных файлах, что обеспечивает воспроизводимость результатов.

Заключение

В ходе данной работы достигнуты следующие результаты:

- Проведён анализ существующих решений в области декомпиляции Java и преобразования Java в Kotlin, а также их комбинаций.
- Разработан набор тестовых примеров¹⁸, а также реализован набор метрик для выявления ограничений существующих подходов и обоснования необходимости специализированных решений.
- Предложен новый подход к декомпиляции Kotlin, основанный на дообучении и использовании языковых моделей.
- Сформирован и подготовлен датасет¹⁹, содержащий пары *байт-код — исходный код Kotlin*, основанный на двух источниках: KExercises и KStack-clean.
- Проведён анализ существующих моделей генерации кода от 0.5B до 7B параметров.
- Из выбранных моделей отобраны две наиболее перспективные компактные модели. Эти модели дообучены на специализированном датасете пар байт-код — исходный Kotlin-код²⁰.
- Проведена экспериментальная оценка качества декомпиляции с использованием нового подхода. Результаты демонстрируют улучшение показателей дообученных моделей.

Эти результаты показывают, что предложенный подход на базе LLM является эффективным для декомпиляции Kotlin.

Код проекта доступен на GitHub: <https://github.com/akabynda/KotlinDecompiler>.

¹⁸KotlinDecompilerSyntheticTests, URL: <https://github.com/akabynda/KotlinDecompilerSyntheticTests> (дата обращения: 26 мая 2025 г.).

¹⁹KExercises-KStack-clean-bytecode, URL: <https://huggingface.co/datasets/akabynda/KExercises-KStack-clean-bytecode> (дата обращения: 26 мая 2025 г.).

²⁰Дообученные модели, URL: <https://github.com/akabynda/KotlinDecompiler/tree/master/models> (дата обращения: 26 мая 2025 г.).

Список литературы

- [1] Blum Avrim L., Langley Pat. Selection of relevant features and examples in machine learning // [Artificial Intelligence](#). — 1997. — Vol. 97, no. 1. — P. 245–271. — Relevance. URL: <https://www.sciencedirect.com/science/article/pii/S0004370297000635> (дата обращения: 26 мая 2025 г.).
- [2] Do all roads lead to Rome? Studying distance measures in the context of machine learning / Eva Blanco-Mallo, Laura Morán-Fernández, Beatriz Remeseiro, Verónica Bolón-Canedo // [Pattern Recognition](#). — 2023. — Vol. 141. — P. 109646. — URL: <https://www.sciencedirect.com/science/article/pii/S0031320323003473> (дата обращения: 26 мая 2025 г.).
- [3] Ni Jiliang, Pu Jiachen, Yang Zhongyi et al. From Large to Super-Tiny: End-to-End Optimization for Cost-Efficient LLMs. — 2025. — [2504.13471](#).
- [4] Govil Nikhil. [Applying Halstead Software Science on Different Programming Languages for Analyzing Software Complexity](#) // 2020 4th International Conference on Trends in Electronics and Informatics (ICOEI)(48184). — 2020. — P. 939–943. — URL: <https://ieeexplore.ieee.org/abstract/document/9142911> (дата обращения: 26 мая 2025 г.).
- [5] Gusarovs Konstantins. An Analysis on Java Programming Language Decompiler Capabilities // [Applied Computer Systems](#). — 2018. — Vol. 23. — URL: <http://dx.doi.org/10.2478/acss-2018-0014> (дата обращения: 26 мая 2025 г.).
- [6] Guyon Isabelle, Elisseeff André. An Introduction of Variable and Feature Selection // [J. Machine Learning Research Special Issue on Variable and Feature Selection](#). — 2003. — 01. — Vol. 3. — P. 1157 – 1182.

- [7] Halstead Maurice H. Elements of Software Science (Operating and programming systems series). — USA : Elsevier Science Inc., 1977. — ISBN: [0444002057](#).
- [8] Hu Taojun, Zhou Xiao-Hua. Unveiling LLM Evaluation Focused on Metrics: Challenges and Solutions. — 2024. — [2404.09135](#).
- [9] Wu Wei-Cheng, Yan Yutian, Egilsson Hallgrimur David et al. Is This the Same Code? A Comprehensive Study of Decompilation Techniques for WebAssembly Binaries. — 2024. — URL: <https://arxiv.org/abs/2411.02278> (дата обращения: 26 мая 2025 г.).
- [10] Java decompiler diversity and its application to meta-decompilation / Nicolas Harrand, César Soto-Valero, Martin Monperrus, Benoit Baudry // [Journal of Systems and Software](#). — 2020. — Vol. 168. — URL: <http://dx.doi.org/10.1016/j.jss.2020.110645> (дата обращения: 26 мая 2025 г.).
- [11] Lizier Joseph T. JIDT: An Information-Theoretic Toolkit for Studying the Dynamics of Complex Systems // [Frontiers in Robotics and AI](#). — 2014. — . — Vol. 1. — URL: <http://dx.doi.org/10.3389/frobt.2014.00011> (дата обращения: 26 мая 2025 г.).
- [12] Hu Edward J., Shen Yelong, Wallis Phillip et al. LoRA: Low-Rank Adaptation of Large Language Models. — 2021. — [2106.09685](#).
- [13] Loshchilov Ilya, Hutter Frank. Decoupled Weight Decay Regularization. — 2019. — [1711.05101](#).
- [14] McCabe T. J. A Complexity Measure // [IEEE Trans. Softw. Eng.](#) — 1976. — . — Vol. 2, no. 4. — P. 308–320. — URL: <https://doi.org/10.1109/TSE.1976.233837> (дата обращения: 26 мая 2025 г.).
- [15] Naeem N.A., Batchelder M., Hendren L. [Metrics for Measuring the Effectiveness of Decompilers and Obfuscators](#) // 15th IEEE International Conference on Program Comprehension (ICPC '07). — 2007. —

URL: <http://dx.doi.org/10.1109/ICPC.2007.27> (дата обращения: 26 мая 2025 г.).

- [16] Dettmers Tim, Pagnoni Artidoro, Holtzman Ari, Zettlemoyer Luke. QLoRA: Efficient Finetuning of Quantized LLMs. — 2023. — [2305.14314](#).
- [17] Shudrak Maksim, Zolotarev Vyacheslav. Improving Fuzzing Using Software Complexity Metrics // [Lecture Notes in Computer Science](#). — 2016. — 03. — Vol. 9558. — P. 246–261. — URL: https://www.researchgate.net/publication/312577578_Improving_Fuzzing_Using_Software_Complexity_Metrics#read (дата обращения: 26 мая 2025 г.).
- [18] Tai Kuo-Chung. A program complexity metric based on data flow information in control graphs // Proceedings of the 7th International Conference on Software Engineering. — ICSE '84. — IEEE Press, 1984. — P. 239–248.
- [19] X. Yu D.A. Lamb. [Metrics Applicable To Software Design](#). — WIT Press, 1994. — Vol. 8. — P. 16. — URL: <https://www.witpress.com/Secure/elibrary/papers/SQM94/SQM94024FU.pdf> (дата обращения: 26 мая 2025 г.).