

УНИВЕРЗИТЕТ У БЕОГРАДУ
МАТЕМАТИЧКИ ФАКУЛТЕТ



Стефан Миленковић

**ПОБОЉШАЊЕ СТАТИЧКОГ
ПРОФАЈЛЕРА *GRAALMHC*
КОРИШЋЕЊЕМ СЕМАНТИКЕ ИЗ
ИМЕНА МЕТОДА**

мастер рад

Београд, 2026.

Ментор:

проф. др Милена ВУЛОШЕВИЋ ЈАНИЧИЋ, редовни професор
Универзитет у Београду, Математички факултет

Чланови комисије:

доц. др Мирко СПАСИЋ, доцент
Универзитет у Београду, Математички факултет

др Милан ЧУГУРОВИЋ, асистент
Универзитет у Београду, Математички факултет

Датум одбране: _____

Наслов мастер рада: Побољшање статичког профајлера *GraalMHC* коришћењем семантике из имена метода

Резиме: Оптимизација вођена профилом извршавања програма (енгл. *Profile-Guided Optimization*, скраћено *PGO*) је техника оптимизације превођења програма која користи информације о понашању програма током његовог извршавања чиме остварује значајна побољшања перформанси програма. Ипак, њену ширу примену ограничава динамичко прикупљање профила, које захтева обезбеђивање репрезентативних улаза и додатна извршавања програма. Статички профајлери превазилазе те недостатке предвиђањем профила без извршавања програма, при чему се савремена решења све чешће заснивају на методама машинског учења. Међутим, ни најсавременији статички профајлери, попут профајлера *GraalMHC* који класификује методе према учесталости позива ради смањења величине извршивог програма, не користе семантику из имена метода, иако она може навестити понашање методе током извршавања.

Основни допринос овог рада је проширење скупа атрибута класификационог модела профајлера *GraalMHC* кодирањем потпуно квалификованих имена метода помоћу уграђивања речи (енгл. *word embedding*). Размотрена су два уграђивања речи, општенаменско уграђивање *GloVe* и наменски натренирано уграђивање, и спроведена је исцрпна претрага за најбољом конфигурацијом кодирања. Решење је имплементирано у оквиру компајлера *GraalVM Native Image*. У односу на верзију профајлера која не користи имена метода, проширени профајлер на нивоу оптимизација *O2* смањује величину извршивог програма за 4,7% до 6,4% уз регресију у времену извршавања мању од 2,4%, док на нивоу *O3* постиже смањење од 8,4% до 10,8% уз регресију до 4,4%. Наменски натренирано уграђивање даје боље резултате од уграђивања *GloVe* уз вишеструко мање просторно заузеће (2,77 MB према 163 MB). Добијени резултати потврђују да имена метода носе значајне семантичке информације које могу унапредити статичко предвиђање профила извршавања програма.

Кључне речи: компајлери, статички профајлери, машинско учење, ансамбли, систем *GraalVM*

Садржај

1	Увод	1
2	Систем <i>GraalVM</i>	4
2.1	Компајлер <i>GraalVM</i> Native Image	5
2.2	Оптимизација вођена профилем извршавања програма	5
2.3	Статичко предвиђање профила	8
3	Машинско учење	13
3.1	Скуп података	13
3.2	Надгледано учење	14
3.3	Токенизација	20
3.4	Уграђивања речи	20
4	Имплементација и евалуација	24
4.1	Проширење скупа атрибута	24
4.2	Конфигурације за кодирање имена метода	25
4.3	Скупови података за обучавање, валидацију и евалуацију модела	29
4.4	Одабир модела	32
4.5	Евалуација одабраних модела	38
5	Закључак	43
	Литература	45

Глава 1

Увод

Оптимизација вођена профилем извршавања програма (енгл. *Profile-Guided Optimization*, скраћено *PGO*) је техника оптимизације превођења која користи информације о понашању програма током његовог извршавања. Те информације, обједињене у профил извршавања, обухватају податке као што су учесталост извршавања појединих инструкција, грана и метода, као и информације о токовима управљања и позивима метода. На основу ових података компајлер доноси боље одлуке приликом примене различитих оптимизација, што доводи до побољшања перформанси програма.

Иако *PGO* може да обезбеди значајна унапређења перформанси, његова примена у пракси није увек једноставна. Да би се добио профил извршавања, неопходно је програм најпре превести са инструментованим кодом, затим га извршити над репрезентативним скупом улазних података и прикупљени профил искористити приликом поновног превођења. Овакав поступак захтева додатно време и ресурсе, а његова успешност у великој мери зависи од квалитета изабраних улазних података. Уколико они не представљају стварне начине употребе програма, добијени профил може бити нерепрезентативан, што умањује ефекте оптимизација, а понекад их може учинити и неефикасним.

Један од начина да се ови недостаци превазиђу јесте статичко предвиђање профила извршавања. Код овог приступа профил се не добија извршавањем програма, већ се процењује анализом изворног кода или међукода. На тај начин избегава се потреба за инструментовањем програма и његовим додатним извршавањем, што омогућава примену оптимизација вођених профилем и у ситуацијама када прикупљање стварног профила није могуће или није

практично. Међутим, цена оваквог приступа је мања прецизност процењеног профила, јер се он заснива на моделима и претпоставкама, а не на стварном понашању програма током извршавања.

Савремени статички профайлери најчешће процењују профил извршавања применом скупа хеуристичких правила [8, 44]. Последњих година све већу пажњу привлаче приступи засновани на методама машинског учења, који настоје да науче зависности између карактеристика програма и његовог понашања током извршавања, чиме се постиже већа тачност предвиђања профила [36, 32, 23, 11, 22, 12].

Статички профайлери користе различите податке за предвиђање профила. То могу бити информације које се извлаче из међурепрезентације програма или графа тока података. Међутим, ни најсавременији статички профайлери не користе семантику из имена метода програма упркос томе што она може садржати корисне информације о томе како треба приступити оптимизацијама. На пример, очекивано је да ће се метода са именом `handleError` извршавати само у случају да дође до грешке, па се може претпоставити да се она неће често позивати. Са друге стране, за методу са називом `getValue` се може очекивати да ће бити метода за добављање вредности (енгл. *getter*) и као таква ће се често позивати у програму. Ово је основна претпоставка рада, односно да имена метода носе значајну семантику за предвиђање профила извршавања програма.

Главни допринос рада и новину представља проширивање статичког профайлера *GraalMHC* [12] семантиком из имена метода. Коришћена су два различита уграђивања речи (енгл. *word embedding*) и спроведена је исцрпна претрага за најбољом конфигурацијом кодирања потпуно квалификованих имена метода. Профили предвиђени проширеним профайлером омогућавају генерисање извршивог програма са 4,7% до 10,8% мањом величином у односу на програме генерисане без семантике из имена метода, док је регресија у перформансама извршавања програма сразмерна оној у постојећим статичким профайлерима.

Рад је организован у пет поглавља. Поглавље 2 описује систем *GraalVM* и његове основне делове. Поред тога, уводи појам статичких профайлера и описује статички профайлер *GraalMHC*. Поглавље 3 уводи основне појмове машинског учења, а посебно описује ансамбле стабала и уграђивања речи. Поглавље 4 описује поступак проширивања профайлера семантиком из имена

метода, описује разматране конфигурације и даје резултате евалуације најбољег модела. Поглавље 5 закључује рад и предлаже правце даљег рада.

Глава 2

Систем *GraalVM*

Систем *GraalVM* [2] је извршно окружење (енгл. *runtime environment*) високих перформанси настало са идејом проширења Јава развојног пакета (енгл. *Java Development Kit*, скраћено *JDK*). Основни циљеви система су побољшање перформанси Јава виртуелне машине (енгл. *Java Virtual Machine*, скраћено *JVM*), смањење времена покретања и меморијског заузећа програма помоћу компајлирања унапред (енгл. *ahead-of-time*, скраћено *AOT*), као и додавање подршке за уграђивање додатних језика у апликације које користе *JVM*.

Главне целине система су:

- Компајлер *Graal* [1] – *JIT* (енгл. *just-in-time*) компајлер који примењује додатне оптимизације и пружа боље перформансе у односу на компајлере *C1* и *C2*, који се користе у оквиру виртуелне машине *HotSpot* [25], подразумеване Јава виртуелне машине у оквиру пројекта *OpenJDK*, који развија компанија *Oracle*.
- Компајлер *GraalVM Native Image* [3] – *AOT* компајлер који смањује време покретања и меморијско заузеће програма (више детаља у одељку 2.1).
- Радни оквир *Truffle* [5] – радни оквир за прављење интерпретера програмских језика за извршавање под системом *GraalVM*.
- *GraalVM Polyglot* [4] – апликативни програмски интерфејс (енгл. *Application Programming Interface*, скраћено *API*) за уграђивање других програмских језика у апликације које се покрећу над системом *GraalVM*.

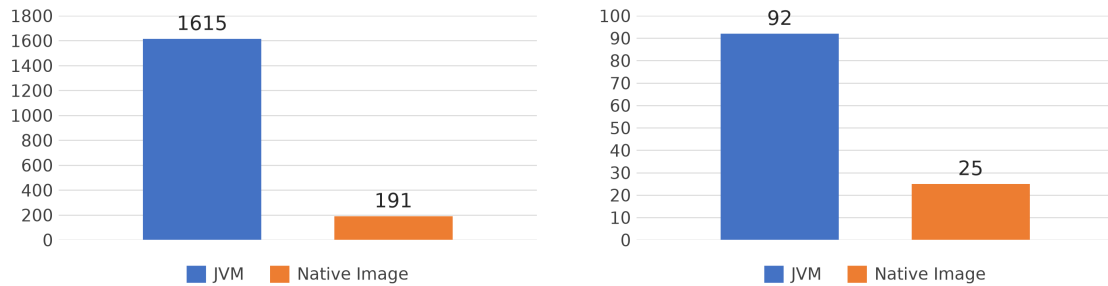
2.1 Компајлер *GraalVM Native Image*

Компајлери традиционално преводе изворни код у фајл извршив на платформи на којој се покрећу. То за последицу има да резултујући фајл не може да се извршава на системима или архитектурама другачијим од оне на којој је настао. Програмски језик Јава [14] решава тај проблем уводећи концепт Јава виртуелне машине која представља посредника између преведеног фајла и циљне физичке машине. Код се преводи до међурепрезентације (Јава бајткода) коју Јава виртуелна машина интерпретира на циљној платформи. Овим се омогућава да се исти изворни код може покретати на произвољној платформи која садржи *JVM* и отвара простор динамичким техникама оптимизације које повећавају ефикасности извршавања програма. Овај приступ одговара *JIT* компајлирању коју програмски језик Јава подразумевано користи.

Са друге стране, Јава виртуелна машина уводи додатну комплексност и оптерећење система, што за последицу има веће меморијско заузеће и лошију временску ефикасност при покретању. Компајлер *GraalVM Native Image* се бави решавањем недостатака Јава виртуелне машине тако што омогућава превођење Јаве унапред. Прецизније, омогућава превођење Јава бајткода на машински код циљне платформе. Тиме се значајно смањује време покретања програма, али долази и до мањег заузећа меморије током извршавања. На слици 2.1 су приказана поређења између времена покретања и меморијског заузећа програма добијених помоћу Јава виртуелне машине и компајлера *GraalVM Native Image*. Времена покретања тестираних програма преведених компајлером *GraalVM Native Image* су 8,5 пута мања него времена покретања добијена Јава виртуелном машином. Осим тога, величине резидентног скупа су 3,7 пута мање. Ово је нарочито значајно у контексту извршавања у облаку (енгл. *cloud computing*) и апликацијама које користе микросервисну архитектуру.

2.2 Оптимизација вођена профилем извршавања програма

Компајлер *GraalVM Native Image* производи извршиве програме чије је време покретања значајно мање у односу на време покретања које нуди Јава виртуелна машина. Ипак, Јава виртуелна машина омогућава боље пер-

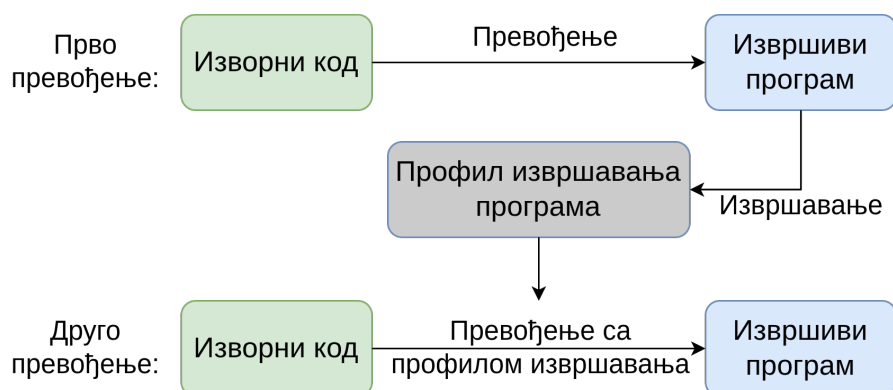


- (1) Време покретања – кашњење приликом покретања програма (ms). (2) Меморијско заузеће – величина резидентног скупа (MB).

Слика 2.1: Поређење перформанси *JVM* и компајлера *GraalVM Native Image* [42]

формансе за програме који се дугорочно извршавају. Наиме, *JVM* прикупља информације током извршавања програма и примењује одговарајуће оптимизације сходно њима. Превођења унапред по својој природи не могу да имају такве информације, међутим, уколико дозволимо додатна извршавања програма, компајлери могу да користе сличан приступ и произведу ефикасније програме.

Оптимизација вођена профилем извршавања програма (енгл. *Profile-Guided Optimization*, скраћено *PGO*) представља технику оптимизације програма у којој се уводи додатно покретање програма ради прикупљања података о његовом понашању током извршавања, а затим се ти подаци користе за усмеравање накнадних оптимизација. На слици 2.2 је дат преглед превођења коришћењем профила извршавања програма.



Слика 2.2: Оптимизација вођена профилем извршавања програма

Током извршавања прве верзије програма, долази до прикупљања података о извршавању програма, тзв. *профила извршавања програма*. Профил представља сумаризоване информације о извршавању програма. Најчешће се прикупљају информације попут броја извршавања тачне гране наредбе гранања, броја итерација петљи, броја позива метода и слично.

Након прикупљања профила, покреће се још једно превођење које користи профил извршавања помоћу којег компајлер усмерава оптимизације. На пример, информација профила да се метода `getValue` позива више пута од методе `handleError` сугерише компајлеру да примени уметање кода¹ (енгл. *inlining*) за методу `getValue`, док методи `handleError` не би посветио много времена јер се њеном оптимизацијом не добија значајно боље укупно време извршавања програма.

Информације о броју извршавања грана које одговарају испуњеним условима наредби гранања, компајлер може искористити тако што би доделио више времена оптимизовању грана које се чешће извршавају. На пример, уколико услов наредбе гранања представља проверу да ли је фајл успешно отворен и током прикупљања профила је тај услов био испуњен 100 пута чешће него што није, компајлер се може фокусирати на грану која одговара испуњеном услову наредбе гранања и не мора се трошити додатно време за грану која одговара неиспуњеном услову. Иако се у пракси може извршити грана која одговара услову који није испуњен, претпоставка је да ће то бити значајно ређе, па се може очекивати да ће доћи до смањења укупног времена извршавања у просечном случају. Осим броја извршавања грана, може се говорити и о вероватноћама њихових извршавања које се добијају као количници броја извршавања одговарајуће гране и укупног броја пролазака кроз наредбу гранања.

Информације о броју извршавања грана које одговарају испуњеним условима гранања, компајлер може искористити тако што би доделио више времена оптимизовању грана које се чешће извршавају. На пример, уколико услов наредбе гранања представља проверу да ли је фајл успешно отворен и током прикупљања профила је тај услов био испуњен 100 пута чешће него што није, компајлер се може фокусирати на грану која одговара испуњеном услову наредбе гранања и не мора се трошити додатно време за грану која одговара

¹Уметање кода је оптимизација која позив функције замењује телом позване функције, чиме се избегава трошак позива и омогућава примена даљих оптимизација на проширеном коду.

неиспуњеном услову. Иако се у пракси може извршити грана која одговара услову који није испуњен, претпоставка је да ће то бити значајно ређе, па се може очекивати да ће доћи до смањења укупног времена извршавања у просечном случају. Осим броја извршавања грана, може се говорити и о вероватноћама њихових извршавања које се добијају као количници броја извршавања одговарајуће гране и укупног броја пролазака кроз наредбу гранања. Слично, информације о броју извршавања петљи у профилу наводе компајлер на петље којима треба посветити додатну пажњу. На пример, за петље од којих се очекује велики број понављања, компајлер може другачије распоредити променљиве у регистре да би оптимизовао њихово често коришћење.

Извршиви програми који настају коришћењем оптимизације вођене профилем извршавања програма су најчешће временски и меморијски значајно ефикаснији од програма насталих без њиховог коришћења. Ипак, квалитет програма значајно зависи од квалитета профила. Уколико се извршавање прве верзије програма (слика 2.2) не одвија над репрезентативним улазима, може се десити да компајлер оптимизује методу која нема велики утицај на укупне перформансе, док у потпуности занемари неку другу методу која је заправо значајна. То за последицу има да оптимизација вођена профилем извршавања програма захтева додатну обавезу од програмера где је, осим писања кода, потребно обезбедити и репрезентативне улазе за прикупљање профила. Поред тога, уводи се додатно превођење и покретање који успоравају целокупни процес превођења.

2.3 Статичко предвиђање профила

Оптимизација вођена профилем извршавања програма може значајно да побољша перформансе и меморијско заузеће резултујућег програма. Ипак, захтевају да програмери припреме улазе који одговарају реалном коришћењу програма и уводе додатну фазу превођења и извршавања. Из тих разлога, упркос јасним побољшањима, оптимизација вођена профилем није опште прихваћена у модерним компајлерима унапред [11].

Једно решење за поменуте недостатке је да додатно покретање програма заменимо статичким предвиђањем профила. Статичко предвиђање профила подразумева генерисање информација које профил садржи на основу изворног кода програма, без извршавања. На пример, за методу која садржи више

петљи него нека друга може се још на нивоу изворног кода, без извршавања програма, претпоставити да ће у извршавању проводити више времена. Статичко предвиђање профила зато нема додатно покретање програма и, важније, не захтева од програмера да обезбеди релевантне улазе. Са друге стране, квалитет предвиђених профила је најчешће лошији од профила насталих извршавањем програма.

Модерни компајлери унапред користе статичко предвиђање профила које је најчешће засновано на хеуристикама [8, 44]. Компајлер *Clang* [17, 29] подразумевано користи ручно писане хеуристике за предвиђање вероватноћа извршавања грана. Компајлер *GCC* [39] такође подразумевано користи сличне хеуристике. Поред хеуристика, у последње време су све заступљеније предикције засноване на машинском учењу, које често превазилазе хеуристичке методе [36, 32, 23, 11, 22, 12]. Такви приступи могу да обухвате шири контекст од ручно писаних метода и имају могућност препознавања суптилних правилности међу подацима.

Предвиђања заснована на машинском учењу примењују се и изван система *GraalVM*, на различитим нивоима представљања програма. Профајлер *VE-SPA* [23] неуронском мрежом предвиђа фреквенције извршавања директно на нивоу бинарног кода и на основу њих оптимизује распоред кода. Систем *Itthermal* [19] дубоком неуронском мрежом предвиђа пропусни опсег (енгл. *throughput*) основних блокова и тако превазилази ручно писане аналитичке моделе у задњем делу компајлера. Оквир *MLGO* [40] уграђује научене моделе у компајлерску инфраструктуру *LLVM* и заменом хеуристике за уметање кода (енгл. *inlining*) смањује величину извршивог програма. Посебно је релевантан приступ у раду *Festival* [45], који предвиђа учесталости циљева виртуелних позива на основу особина изведених из декларација метода, попут дубине пакета и модификатора приступа, при чему их кодира ручно писаним скаларним атрибутима. Тиме *Festival* показује да имена и декларације метода носе корисне информације за предвиђање профила, али их искоришћава само делимично, што директно мотивише прелазак на научена уграђивања речи (енгл. *word embedding*) која се разматрају у овом раду.

Постојећа решења у систему *GraalVM*

Систем *GraalVM* кроз компајлер унапред *GraalVM Native Image* подржава статичко предвиђање профила. Статичко предвиђање профила засновано

на методама машинског учења подржано је кроз профајлере *GraalSP* [11], *GraalNN* [22] и *GraalMHC* [12].

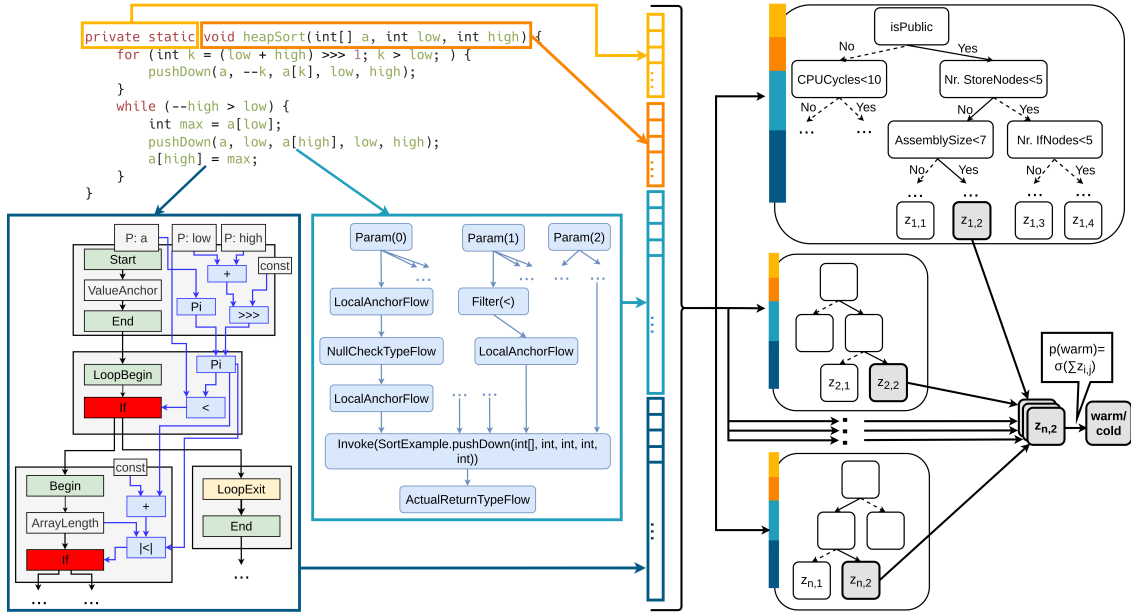
GraalSP предвиђа вероватноће извршавања грана помоћу ансамбла стабала *XGBoost* [10]. Статички профајлер *GraalNN* такође предвиђа вероватноће извршавања грана и представља унапређење профајлера *GraalSP*. Користи графовску неуронску мрежу [34] да би могао да разматра структурне информације директно на основу графа контроле тока. Како мрежа директно ради са графовском структуром, успева да учи неочигледна правила међу чворовима графа контроле тока и даје боље коначне резултате.

GraalMHC је статички профајлер за класификацију метода према учесталости њихових позива са циљем смањења величине извршивог програма. Компајлер *GraalVM Native Image* разликује три типа метода: хладне, регуларне и вруће. Хладне методе се веома ретко извршавају, вруће методе се веома често извршавају, док су регуларне између њих по учесталости извршавања. Применом ансамбла стабала за бинарну класификацију *XGBoost*, профајлер *GraalMHC* разврстава методе на *хладне* и *топле*, при чему друга класа обухвата и регуларне и вруће методе по класификацији компајлера *GraalVM Native Image*. Оптимизације користе информације које профајлер предвиђа и као резултат се добија мањи извршиви фајл. Модел користи атрибуте изведене из:

1. потписа метода
2. модификатора приступа метода
3. међурепрезентације метода
4. графа тока података метода

Атрибути се кодирају у одговарајуће векторе и прослеђују ансамблу стабала који даље класификује методе. Структура система приказана је на слици 2.3.

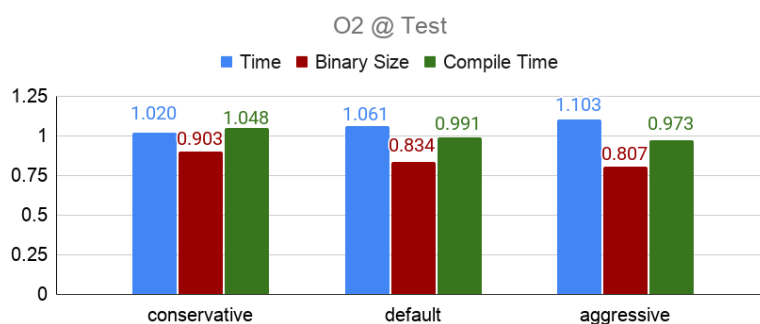
Статички профајлер *GraalMHC* подржава три различита модела *S1*, *S2* и *S3* који се разликују у хиперпараметрима класификационог модела *XGBoost* које користе. Први од њих представља конзервативну опцију где је регресија у времену извршавања најмања, али је исто тако и најмање смањење величине извршивог програма. Са друге стране, трећа опција представља агресивну верзију у којој је највећа уштеда у величини извршивог програма, али је и највећа регресија у перформансама. Друга, подразумевана опција, представља

Слика 2.3: Статички профайлер *GraalMHC* [12]

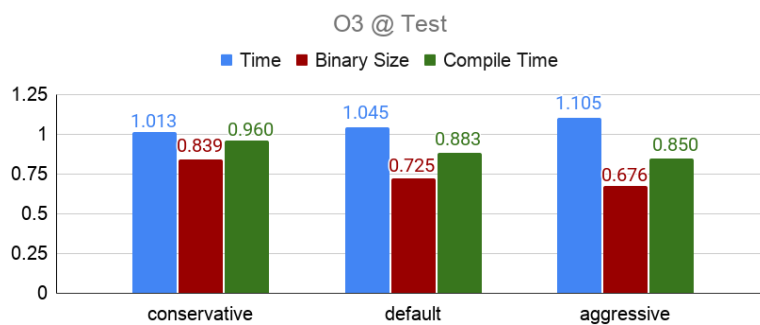
компромис између претходне две пружањем просечног смањења величине за умерену регресију у времену извршавања. Овај приступ је унапређен у верзији профайлера под називом *GraalMHC⁺⁺* који уместо чувања три различита модела, користи један модел са три различите конфигурације: *conservative*, *default* и *aggressive*. Конфигурације се разликују у прагу који класификациони модел у позадини користи за разврставање метода на хладне и топле. Конфигурација *conservative* користи низак праг за класификацију чиме остварује најмању уштеду на величини извршивог програма, али зато има најмањи негативан утицај на повећање време извршавања програма. Конфигурација *aggressive* има висок праг и нуди највећа смањења величине извршивог програма, док конфигурација *default* користи праг између њих који се узима као подразумевани. Конфигурације *conservative*, *default* и *aggressive* редом одговарају моделима *S1*, *S2* и *S3* статичког профайлера *GraalMHC*.

На сликама 2.4 и 2.5 је приказан утицај профайлера *GraalMHC⁺⁺* на генерисане програме у три различите конфигурације и на два нивоа оптимизација. Приказан је утицај на време извршавања, величину извршивог програма и време компајлирања релативно у односу на резултате које производи компајлер *GraalVM Native Image* без профайлера *GraalMHC⁺⁺*. Главни допринос је у смањењу величине извршивог програма која је између 9,7% и 19,3% мања на оптимизационом нивоу *O2*. Са друге стране, време извршавања се повећава

између 2% и 10,3%. На оптимизационом нивоу *O3* су разлике значајније, где је величина извршивог програма између 16,1% и 32,4% мања зарад губитка од 1,3% до 10,5% у перформансама. Овакви резултати су пожељни у ситуацијама где је простор ограничен, а време извршавања није пресудно. Времена компајлирања су директно сразмерна величини извршивог програма [12]. Зато се програми у конфигурацији *conservative*, где је уштеда у величини најмања, преводе најспорије, док су превођења у конфигурацији *aggressive*, која производи програме најмање величине, најбржа.



Слика 2.4: Резултати профайлера *GraalMHC⁺⁺* на нивоу оптимизација *O2*



Слика 2.5: Резултати профайлера *GraalMHC⁺⁺* на нивоу оптимизација *O3*

Различите конфигурације се понашају у складу са очекивањима. Наиме, конфигурација *conservative* има нижи праг за класификацију топлих метода од конфигурације *aggressive*. То значи да ће конфигурација *aggressive* препознавати више хладних метода од конфигурације *conservative*, а тиме се очекује и мања величина извршивог програма. Ипак, неке од њих ће бити погрешно класификоване као хладне иако се често извршавају. Тиме се добијају лошије перформансе што резултати потврђују.

Глава 3

Машинско учење

Машинско учење је област рачунарства која проучава поступке помоћу којих рачунарске системи уче правилности из података и користе их за предвиђање или доношење одлука над новим подацима. Уместо да се сва правила експлицитно задају програмом, модел се обучава на подацима који описују посматрани проблем. У контексту статичког предвиђања профила, подаци су методе програма описане атрибутима изведеним из кода, а циљ је да модел предвиди информацију корисну компајлеру, на пример да ли је метода често позивана.

Поступци машинског учења се најчешће деле на надгледано, ненадгледано и учење поткрепљивањем [33]. У надгледаном учењу сваки инстанца у скупу података има познату циљну вредност коју модел треба да научи да предвиђа. У ненадгледаном учењу циљне вредности нису дате, већ се траже структуре у самим подацима, као што су групе сличних инстанци. Учење поткрепљивањем посматра агента који кроз интеракцију са окружењем бира акције и учи на основу награда или казни које добија. У овом раду се користи надгледано учење, јер се модел обучава на подацима за које је познато да ли методе припадају одговарајућој класи.

3.1 Скуп података

Скуп података је колекција инстанци над којима се модел обучава и проверава. Свака инстанца се представља вектором атрибута. Атрибути су мерљива или изведена својства инстанци, на пример број аргумената методе, модификатор приступа или вредности добијене кодирањем имена методе. Поред

атрибута, у надгледаном учењу је позната и циљна вредност. Она може бити нумеричка вредност, када се решава регресиони проблем, или ознака класе, када се решава класификациони проблем.

Да би процена квалитета модела била поуздана, подаци се деле на три дела: скуп за обучавање, валидациони скуп и тест скуп. Скуп за обучавање се користи за прилагођавање параметара модела. Валидациони скуп се користи током развоја модела, на пример за избор хиперпараметара или избор између више различитих конфигурација. Тест скуп се користи тек након избора модела и служи за завршну процену квалитета над подацима који нису учествовали ни у обучавању ни у избору конфигурације.

Раздвајање валидационог и тест скупа је важно да би се обезбедио непристрасан одабир модела. Ако се модел бира на основу тест скупа, резултат на њему више није независна процена понашања над новим подацима. Због тога се у експериментима најпре користи валидациони скуп за избор модела, док се тест скуп чува за коначну евалуацију.

Поред тога, на поузданост процене утиче и начин на који се подаци расподељују у скупове. Инстанце у скупу података често нису међусобно независне, већ се групишу по заједничком пореклу, тако да инстанце из исте групе деле специфичности које не важе за остале групе. Ако се подела врши насумично, инстанце исте групе доспевају и у скуп за обучавање и у тест скуп, па модел може да научи специфичности те групе и да их искористи приликом предвиђања над тест скупом. Процена је тада превише оптимистична јер тест скуп више не представља заиста нове податке. Зато се подела врши по групама, при чему група треба да одговара целини на коју се модел примењује у пракси. Код статичког предвиђања профила таква група је програм, јер методе истог програма деле конвенције именовања, стил писања и структуру кода, па се подаци деле по програмима или скуповима програма, а не насумично по појединачним методама.

3.2 Надгледано учење

У надгледаном учењу полази се од скупа података

$$\{(x_1, y_1), (x_2, y_2), \dots, (x_m, y_m)\},$$

где је x_i вектор атрибута, а y_i циљна вредност за инстанцу i . Задатак алгоритма учења је да на основу ових података изгради модел који за нови податак x предвиђа вредност y . Током обучавања модел се прилагођава тако да грешка на скупу за обучавање буде мала, али је коначан циљ добра генерализација, односно мала грешка на подацима који нису коришћени за обучавање.

Поред параметара који се уче директно из података, многи алгоритми имају и хиперпараметре. Хиперпараметри су вредности које се задају пре обучавања и одређују облик или сложеност модела. На пример, код ансамбла стабала (енгл. *tree ensemble*) то могу бити број стабала и највећа дубина појединачног стабла. Пошто различите вредности хиперпараметара могу значајно да промене квалитет модела, оне се обично бирају на основу резултата на валидационом скупу.

Регресиони модели

Регресија је задатак надгледаног учења у коме је циљна вредност нумеричка. На пример, модел може да предвиђа време извршавања програма, количину заузете меморије или вероватноћу да ће се нека грана програма извршити. Квалитет регресионог модела се најчешће мери величином грешке између предвиђене и стварне вредности.

Класификациони модели

Класификација је задатак надгледаног учења у коме циљна вредност припада коначном скупу класа међу којима не постоји поредак. Ако постоје само две класе, говори се о бинарној класификацији. У случају статичког профайлера *GraalMHC*, методе се класификују као *топље* или *хладне*, па је реч о бинарној класификацији.

Класификациони модел за сваки улазни податак може да врати само ознаку класе или нумеричку оцену припадности некој класи. На пример, модел може да врати оцену да је метода хладна, а затим се метода класификује као хладна ако је та оцена већа од изабраног прага. Такав праг одлучивања омогућава да се исти модел користи у различитим режимима рада. На пример, конзервативнији праг може да смањи ризик од погрешног проглашавања важне методе хладном, док агресивнији праг доводи до класификације више

метода хладним што, након примена оптимизација, може да узрокује мању величину извршивог програма.

Квалитет класификационог модела се често приказује матрицом конфузије. Код бинарне класификације она садржи број тачно класификованих позитивних и негативних инстанци, као и број лажно позитивних и лажно негативних инстанци. Из ових вредности се могу извести метрике као што су тачност (енгл. *accuracy*), прецизност (енгл. *precision*) и одзив (енгл. *recall*). Матрица конфузије је корисна јер показује не само колико модел грешити, већ и коју врсту грешке прави. У компајлерским оптимизацијама, на пример, погрешна класификација методе која се ретко позива ће незнатно променити укупно време извршавање програма. Са друге стране, погрешна класификација методе која се често позива има израженије последице јер може значајно повећати укупно време извршавања програма.

У бинарној класификацији уобичајено је да се једна класа означаи као позитивна, а друга као негативна. Исходи класификације се тада описују помоћу четири вредности: TP (енгл. *true positives*) је број позитивних инстанци које је модел тачно класификовао као позитивне, TN (енгл. *true negatives*) је број негативних инстанци тачно класификованих као негативне, FP (енгл. *false positives*) је број негативних инстанци погрешно класификованих као позитивне, а FN (енгл. *false negatives*) је број позитивних инстанци погрешно класификованих као негативне. Ове четири вредности чине поља матрице конфузије, а поменуте метрике се преко њих дефинишу на следећи начин:

$$\begin{aligned} accuracy &= \frac{TP + TN}{TP + TN + FP + FN} \\ precision &= \frac{TP}{TP + FP} \\ recall &= \frac{TP}{TP + FN} \end{aligned}$$

Тачност представља удео тачно класификованих инстанци у укупном броју инстанци. Прецизност показује колики је удео стварно позитивних инстанци међу свим инстанцама које је модел прогласио позитивним, док одзив показује колики је удео позитивних инстанци које је модел успео да препозна.

Када су класе у скупу података небалансиране, тачност сама по себи може да буде обмањујућа метрика. У скупу података коришћеном у овом раду хладних метода је вишеструко више него топлих, па би и тривијалан модел

који сваку методу прогласи хладном имао високу тачност, иако не препознаје ниједну топлу методу. Матрица конфузије је у таквим ситуацијама информативнија, јер приказује број тачних и погрешних класификација за сваку класу посебно, док прецизност и одзив мере квалитет предвиђања у односу на изабрану позитивну класу, независно од тога колико је та класа заступљена у подацима. На пример, нека се у скупу од 100 метода њих 85 позива ретко, а 15 често. Класификациони модел који све хладне методе препознаје тачно, а од топлих само једну, имаће метрике приказане у табели 3.1. Тачност је висока и може деловати задовољавајуће, међутим низак одзив открива да модел заправо не врши добру класификацију.

Табела 3.1: Пример матрица конфузије и метрике класификатора над небалансираним подацима

		Предвиђена класа				
		топла	хладна	Тачност	Прецизност	Одзив
Стварна класа	топла	1	14	$\frac{86}{100} = 0,86$	$\frac{1}{1} = 1,00$	$\frac{1}{15} \approx 0,07$
	хладна	0	85			

Стабла одлучивања

Стабло одлучивања је модел који предвиђа вредност тако што низом једноставних услова дели простор података. Унутрашњи чвор стабла садржи услов над једним атрибутом, гране одговарају исходима тог услова, а листови садрже предвиђање. Код класификације лист најчешће садржи класу или расподелу класа међу инстанцама који су стигли до тог листа.

На пример, један чвор може да провери да ли је број позива методе у скупу за обучавање већи од одређеног прага или да ли неки атрибут изведен из имена методе има позитивну вредност. Инстанца се затим прослеђује левој или десној грани, све док не стигне до листа. Оваква структура је интерпретабилна јер се предвиђање може пратити кроз низ услова који су примењени на конкретну инстанцу.

Стабла одлучивања могу да опишу нелинеарне односе међу атрибутима и

не захтевају да атрибути буду на истој скали¹. Зато није потребно претходно стандардизовати податке, односно одузети им просечну вредност и поделити девијацијом. Међутим, једно дубоко стабло може лако да се превише прилагоди скупу за обучавање. Такво прекомерно прилагођавање (енгл. *overfitting*) доводи до добрих резултата на инстанцама за обучавање, али и до слабијих резултата на новим инстанцама. Овај проблем се може ублажити ограничавањем дубине стабла или поткресивањем (енгл. *pruning*), али се у пракси као делотворан и широко коришћен приступ показало и комбиновање више стабала у ансамбл, чиме се побољшава генерализација и смањује дисперзија појединачних модела. Дисперзија модела говори о томе колико су његова предвиђања осетљива на мале промене у скупу за обучавање. Модел велике дисперзије и при малим изменама тог скупа даје знатно другачије стабло одлучивања, због чега слабо генерализује на нове инстанце.

Ансамбли стабала

Ансамбл стабала је модел који предвиђање доноси комбиновањем предвиђања више стабала одлучивања. Појединачно стабло је склоно прекомерном прилагођавању и стога има велику дисперзију. Комбиновањем више стабала појединачне грешке се делимично поништавају, чиме ансамбл постаје отпорнији на прекомерно прилагођавање и боље генерализује од било ког појединачног стабла.

Постоје различити начини да се изгради ансамбл. Један приступ је да се стабла граде независно, свако на другачијем подскупу инстанци или атрибута, а њихова предвиђања се затим усредњавају или се о коначној класи одлучује гласањем. Други приступ је да се стабла граде редом, тако да се свако наредно стабло не обучава на самим циљним вредностима, већ на грешци коју ансамбл изграђен до тог тренутка још увек прави, и учи малу поправку која ту грешку смањује. Зато ниједно појединачно стабло само за себе не даје коначно предвиђање нити је боље од осталих, већ коначно предвиђање настаје тек сабирањем доприноса свих стабала. Класификациони модел коришћен у овом раду припада другој групи, а детаљан опис оваквог поступка изградње дат је у потпоглављу о библиотеци *XGBoost*.

¹На пример, стаблу одлучивања не смета ако су вредности једног атрибута у интервалу $[-10^6, -10^3]$, а неког другог у интервалу $[10^{-8}, 10^{-7}]$.

Поред бољих предвиђања, ансамбли стабала омогућавају и процену значаја (енгл. *feature importance*) појединачних атрибута. Значај атрибута се рачуна на основу тога колико је тај атрибут допринео одлукама у стаблима, на пример колико често је коришћен за поделу података и колико је при томе смањивао грешку. Атрибути са највећим значајем највише утичу на предвиђања модела, па се на основу њих може сагледати на које информације се модел највише ослања.

Библиотека *XGBoost*

XGBoost је библиотека и алгоритам за градијентно појачавање стабала (енгл. *gradient tree boosting*) [10]. Уместо једног стабла, модел се састоји од низа стабала која се граде постепено. Свако наредно стабло покушава да исправи грешке које су направила претходна стабла, а коначно предвиђање добија се комбиновањем њихових резултата. На тај начин се добија модел који је изражајнији од једног стабла, али и даље задржава многе практичне предности стабала одлучивања.

Нова стабла формирају се тако што се у сваком кораку посматра колико тренутна предвиђања ансамбла одступају од тачних вредности на подацима за обучавање. Наредно стабло се затим обучава да предвиди правац и величину поправке којом се то одступање највише смањује, што се формално изводи преко градијента функције губитка, одакле и потиче назив методе. Само стабло се гради похлепно тако што се чворови деле по оним атрибутима и праговима који највише смањују грешку, док се не достигне задата дубина или док даља подела не доноси довољно побољшање. Допринос новог стабла се пре додавања у ансамбл множи бројем из интервала $(0, 1]$ који се назива кораком учења (енгл. *learning rate*) и који одређује колики део предложене поправке ансамбл прихвата у једном кораку. Тиме се ограничава утицај сваког појединачног стабла на коначна предвиђања, јер преостали део грешке исправљају наредна стабла. Мањи корак учења смањује ризик од прекомерног прилагођавања, али захтева више стабала да би се постигао исти квалитет, а сам поступак се понавља док се не изгради задати број стабала.

Код класификације, *XGBoost* може да предвиди оцену припадности класи, а затим се на основу прага одлучивања добија коначна класа. Овај приступ је погодан када се жели више режима рада истог модела. У статичком профилеру се, на пример, различитим прагом могу добити конзервативнија или

агресивнија предвиђања.

Важни хиперпараметри модела *XGBoost* су број стабала и дубина стабала. Већи број стабала и већа дубина могу да повећају способност модела да научи сложене правилности, али истовремено повећавају ризик од пре-прилагођавања и величину самог модела. Због тога се ове вредности бирају експериментално, најчешће на основу валидационог скупа.

3.3 Токенизација

Токенизација је поступак поделе текста на мање јединице које се називају токени². У обради природног језика токен најчешће одговара речи, док у општем случају може бити и мања целина, као што је део речи или појединачни симбол. Токени су основне јединице над којима раде технике за нумеричко представљање текста, попут уграђивања речи, па је токенизација њихов почетни корак.

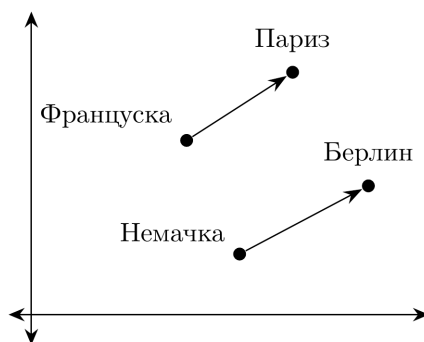
За разлику од природног језика, где су речи обично раздвојене размацима, идентификатори у програмима су сложености које спајају више речи без експлицитних граница. На пример, назив методе `updateUserProfile` записан је у камиљој нотацији (енгл. *camelCase*), у коме велико слово означава почетак нове речи, па се токенизацијом дели на токене `update`, `user` и `profile`. На сличан начин, потпуно квалификовано име методе се најпре раздваја на име пакета, име класе и име методе, а потом се свака од тих целина дели на појединачне токене.

3.4 Уграђивања речи

Уграђивање речи (енгл. *word embedding*) је поступак којим се речи или токени представљају векторима реалних бројева. Циљ је да вектори сачувају део семантичких односа међу речима: речи које се јављају у сличним контекстима треба да имају сличне векторске репрезентације. Такво представљање омогућава да се текстуални подаци користе као нумерички атрибути у моделима машинског учења.

²Иако истог назива, токени у лексичкој анализи представљају другачији појам и означавају унапред дефинисане категорије.

Једноставније представе текста, као што је индикатор да ли се токен појављује или број појављивања токена, сваки токен третирају као засебну целину, па су у њима свака два различита токена подједнако удаљена без обзира на значење. Уграђивања речи омогућавају да се сличност токена изрази нумерички тако што сваком токenu додељују вектор фиксне димензије. На пример, токени који се односе на сличне операције могу имати блиске векторе иако нису исти токен. Илустрација доделе вектора речима се може видети на слици 3.1. Уграђивања речи које означавају државе блиска су у простору уграђивањима речи које означавају њихове главне градове. Поред тога, међусобни односи ових уграђивања су слични што се огледа у томе да је разлика вектора државе и њеног главног града приближно иста за све такве парове.



Слика 3.1: Пример уграђивања речи. Сличне речи су блиске у циљном векторском простору и односи међу речима се чувају.

Постоје различити начини за учење ових вектора, а два најпознатија су *Word2Vec* и *GloVe*. *Word2Vec* је група модела за учење уграђивања речи заснованих на плиткој неуронској мрежи [21]. Вектори се добијају решавањем помоћног задатка предвиђања, при чему постоје две архитектуре: *CBOW* (енгл. *continuous bag of words*) и *Skip-Gram*. Архитектура *CBOW* предвиђа реч на основу речи из њеног контекста, док *Skip-Gram* ради обрнуто и на основу дате речи предвиђа речи из њеног контекста. *GloVe* (енгл. *global vectors*) је модел који уграђивања речи учи из глобалних статистика заједничког појављивања речи у корпусу [26]. Уместо појединачних контекста, *GloVe* полази од броја заједничких појављивања свих парова речи у корпусу и тражи векторе који одражавају те глобалне статистике. Оба приступа полазе од претпоставке да се значење токена може делимично закључити на основу контекста у коме се токен појављује.

Модел *Skip-Gram* обучава се на паровима речи који се формирају клизањем прозора фиксне ширине преко текста. За сваку реч у корпусу, речи које се налазе у њеној непосредној околини, односно унутар прозора, узимају се као њене контекстне речи, чиме се добијају парови облика (*циљна реч, контекстна реч*). Мрежа се затим обучава да за дату циљну реч додели високу вероватноћу њеним контекстним речима, а ниску свим осталим речима из речника. Пошто се речи са сличним значењем појављују у сличним околинама, оне током обучавања добијају сличне векторе, јер морају да предвиђају сличне контексте. Само уграђивање речи чита се из параметара обучене мреже: свакој речи из речника одговара по један научени вектор, а помоћни задатак предвиђања се након обучавања одбацује. У пракси се обучавање убрзава техникама попут негативног узорковања (енгл. *negative sampling*), код кога се уместо целог речника у сваком кораку разматра само мали број насумично изабраних речи које нису у контексту. Важни хиперпараметри модела су ширина прозора и димензија вектора: шири прозор наглашава шире, тематске сличности међу речима, док ужи прозор боље хвата локалне, синтаксичке односе.

За разлику од модела *Word2Vec*, који корпус обрађује пролазећи кроз појединачне контексте, *GloVe* најпре у једном пролазу кроз корпус изгради матрицу заједничког појављивања, у којој свако поље бележи колико пута се пар речи јавио у истом контексту. Вектори речи се затим уче тако да односи међу њима, прецизније њихови скаларни производи, одражавају учесталости из те матрице: паровима речи који се често јављају заједно одговарају вектори који су у том смислу усклађенији. Приликом учења већа тежина даје се паровима са поузданијим статистикама, односно онима који се у корпусу јављају довољно често, док се врло ретки парови мање узимају у обзир. На овај начин *GloVe* комбинује предности глобалних статистика целог корпуса са особиним локалних метода да добијени вектори чувају семантичке односе међу речима.

Уграђивања речи могу бити општа или наменски обучена. Општа уграђивања, као што је *GloVe*, уче се на великим корпусима природног језика и покривају велики број речи. Са друге стране, наменски обучена уграђивања могу боље да одражавају домен у коме се користе, на пример називе метода, класа и пакета у Јава програмима. Она су често мања и могу садржати управо токене који су важни за конкретан задатак. На пример, у раду ће бити

разматрано наменски обучено уграђивање речи величине 2,77 MB које даје упоредиве резултате са општим уграђивањем речи *GloVe* величине 163 MB.

Глава 4

Имплементација и евалуација

Циљ овог рада је уграђивање семантике из имена метода ради побољшања класификације метода према учесталости извршавања. Прецизнија класификација током фазе компајлирања омогућава поузданије препознавање метода које се често извршавају, чиме се смањује величина извршивог програма.

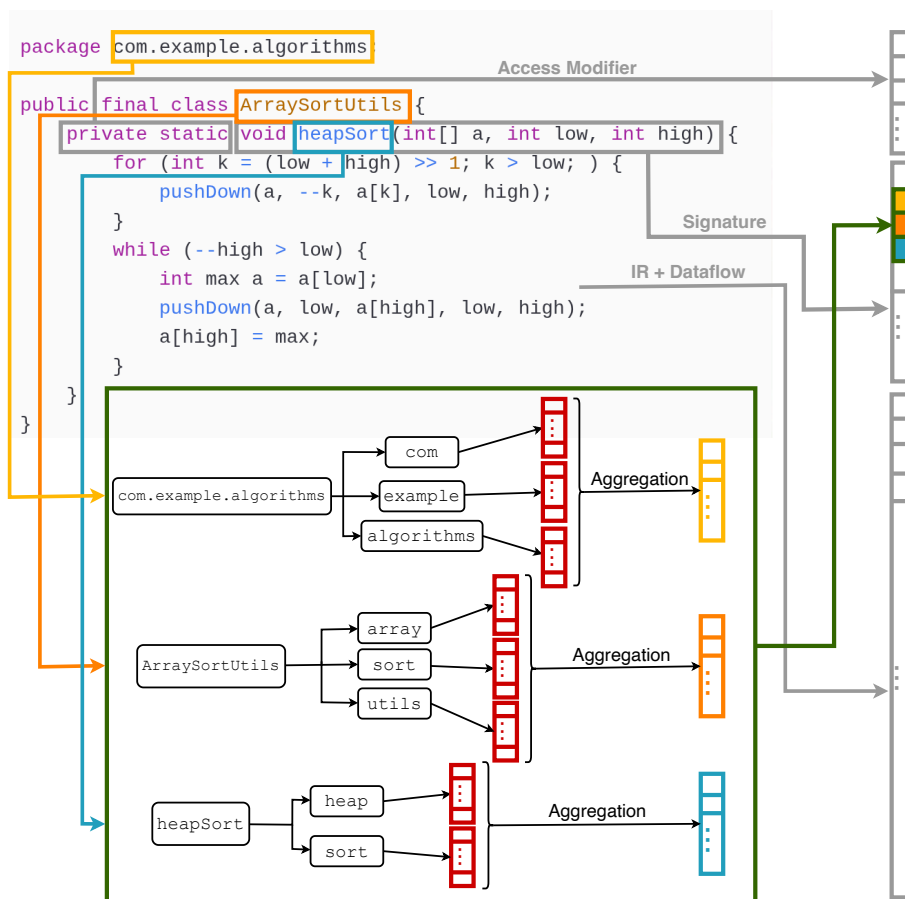
Имплементација је урађена у оквиру система *GraalVM*, конкретно, статичког профајлера *GraalMHC*. Проширен је скуп атрибута класификационог модела ансамбла стабала *XGBoost* кодирањем семантике из имена метода. То је урађено коришћењем уграђивања речи *GloVe* [26], као и коришћењем наменски обученог уграђивања речи за потребе предвиђања профила. За оба уграђивања речи је урађена исцрпна претрага за најбољом конфигурацијом и спроведена одговарајућа претрага хиперпараметара.

Поменуто обучавање и претраге конфигурација се обављају унапред и ван компајлера. У сам компајлер се уграђује искључиво већ обучен модел, односно ансамбл стабала *XGBoost* извезен у формат *ONNX* (енгл. *Open Neural Network Exchange*). Због тога се при превођењу програма обавља само закључивање модела, без икаквог обучавања, чиме се обезбеђује да предложени приступ не успорава значајно превођење програма.

4.1 Проширење скупа атрибута

Скуп атрибута изведених из потписа метода је проширен кодирањем имена метода. За методу се посматра њено потпуно квалификовано име (енгл. *fully qualified name*) из кога се извлаче три дела: име пакета, име класе, као и само име методе. Сваки од њих се токенизује, односно дели на мање јединице

— токене. Затим се на одабране токене примењује уграђивање речи, чиме се кодирају појединачни токени у векторе фиксне димензије. Тако добијени вектори се агрегирају у један вектор који се даље прослеђује моделу који класификује методе. Опис система се може видети на слици 4.1.



Слика 4.1: Проширивање скупа атрибута статичког профайлера *GraalMHC* кодирањем имена метода

4.2 Конфигурације за кодирање имена метода

Због начина кодирања имена метода, да би један модел био потпуно одређен, потребно је одабрати:

1. начин избора токена

2. уграђивање речи за кодирање
3. начин агрегације кодова токена.

За сваки од избора је разматрано неколико различитих опција, чиме је дефинисан скуп конфигурација на основу којих је биран најбољи модел (одељак 4.4).

Начин избора токена. Разматран је избор фиксног броја токена као и коришћење свих доступних токена у случају да функција агрегације производи векторе фиксне дужине независно од величине улаза. У случају фиксног броја токена бирано је n токена, где је $1 \leq n \leq 6$, а границе су одабране емпиријски. Поред тога, издвојени су токени и израчунат је број појављивања сваког од њих у 500 најпопуларнијих пројеката на сервису *GitHub* [13] заснованих на алату *Maven* [7]. Међу n одабраних токена налази се k токена са најмањом фреквенцијом, док се преосталих $n - k$ бира међу токенима са највећом фреквенцијом. Вредност k бирамо тако да је $0 \leq k \leq n$. Уколико је доступно мање од n токена, улаз у функцију агрегације се допуњује одговарајућим бројем нула вектора. На овај начин се разматра укупно

$$|\{(n, k) \mid 1 \leq n \leq 6 \wedge 0 \leq k \leq n\}| = 27$$

различитих начина за одабир токена за кодирање имена метода и класа.

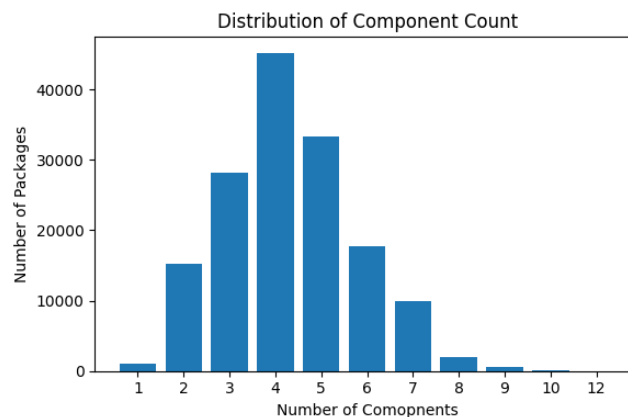
Мотивација за одабир најфреквентнијих токена је претпоставка да токени који се појављују чешће могу имати већи значај од оних који су ретки. Са друге стране, токени који се појављују често могу да буду превише општи, те се зато бирају и они који су најређи са идејом да носе специфичности одговарајуће методе.

На пример, метода са називом `updateUserProfileUsingQuorum` се иницијално дели на речи у камиљој нотацији као што је приказано у табели 4.1. Затим се за сваку реч рачуна њен корен (енгл. *stemming*) и одбацују се речи које се не налазе у речнику. За рачунање корена речи је коришћен Портеров алгоритам (енгл. *Porter stemmer*) [28]. Портеров алгоритам рачуна корене речи поступним уклањањем уобичајених суфикса енглеског језика, па се речи `update`, `updates` и `updated` свODE на исти корен `updat`. За $n = 3$ и $k = 1$, били би одабрани токени `updat`, `us` и `quorum`. Приметимо да коришћење само најфреквентнијих токена не би узело у обзир токен `quorum` који носи значајну семантичку вредност. Такође, узимање само најмање фреквентних токена би изоставило токен `updat` који је подједнако семантички важан.

Табела 4.1: Пример токенизације методе `updateUserProfileUsingQuorum`

Реч	Токен	Број појављивања
<code>update</code>	<code>updat</code>	4 481
<code>user</code>	<code>user</code>	1 950
<code>profile</code>	<code>profil</code>	526
<code>using</code>	<code>us</code> ¹	3 376
<code>quorum</code>	<code>quorum</code>	81

Поред тога, за одабир токена пакета нису коришћене фреквенције речи јер семантички значај токена пакета опада здесна налево. На пример, за пакет са називом `org.apache.commons.math.linear.decomposition` одговарајућа токенизација даје токене: `org`, `apache`, `commons`, `math`, `linear` и `decomposition`². Како токен `decomposition` има веће семантичко значење од токена `org`, то код пакета бирамо n токена здесна налево независно од поменутог претходно израчунатог броја појављивања. Тако би за $n = 3$ били одабрани токени `math`, `linear` и `decomposition`. Расподела броја компоненти пакета из репозиторијума *GraalVM Reachability Metadata* [24] се може видети на слици 4.2. С обзиром да 92% пакета има мање од 7 компоненти, разматране вредности за n су $1 \leq n \leq 6$, односно разматрано је 6 различитих начина за одабир токена приликом кодирања имена пакета.



Слика 4.2: Расподела броја компоненти пакета

Уграђивање речи. Коришћена су два уграђивања речи. Једно је опште-

¹Портеров алгоритам реч `using` своди на основу `us`. Оваква редукција позната је особина алгоритма и представља прихватљив компромис, будући да савршен алгоритам за одређивање основе речи у енглеском језику не постоји.

²За сваки од њих се заправо чувају корени.

наменско уграђивање речи *GloVe* са 400 000 инстанци кодираних у векторе димензије 50. Друго је наменски обучено уграђивање речи за предвиђање профила извршавања програма засновано на *Skip-Gram* архитектури модела *Word2Vec* [21] са 7 093 инстанце кодиране у векторе димензије 100. За обучавање је коришћен скуп од 800 000 инстанци репозиторијума *GraalVM Reachability Metadata* [24]. Кратак преглед коришћених уграђивања речи се може видети у табели 4.2.

Табела 4.2: Коришћена уграђивања речи

Уграђивање речи	Број инстанци	Димензија	Укупна величина
<i>GloVe</i>	400 000	50	163 MB
Наменски обучено	7 093	100	2,77 MB

Укупна величина представља просторно заузеће уграђивања речи, односно колико је бајтова потребно за чување одговарајућег уграђивања. Као општије, уграђивање речи *GloVe* захтева значајно више простора што може да представља проблем приликом испоруке компајлера. Са друге стране, наменски обучено уграђивање речи се фокусира на семантику и токене потребне за предвиђање профила извршавања програма што омогућава знатно компактнију репрезентацију. Уграђивање речи у овом раду је дељено, односно за кодирање сва три дела потпуно квалификованог имена методе (пакета, класе и имена методе) користи се исто уграђивање речи.

Агрегација. Коришћене су две методе агрегације вектора кодова токена: упросечавање и конкатенација. Упросечавање узима произвољан број вектора исте димензије и враћа један вектор полазне димензије добијен рачунањем аритметичке средине улаза. Конкатенација формира вектор надовезивањем вектора улаза чија је димензија једнака збиру димензија улаза. Предност упросечавања је што враћа вектор фиксне димензије независно од броја улаза и класификациони модел је мањи због мањег броја атрибута. Са друге стране, конкатенацијом додајемо информације о појединачним токенима као и редоследу њиховог појављивања у именима, али је зато неопходно да на улазу буде фиксан број вектора. Наиме, модел очекује улаз фиксне димензије, па је у случају конкатенације неопходно одређивање броја улазних вектора унапред.

На пример, метода са називом `handleNewRequest` се дели на токене `handle`, `new` и `request`. Сваком од тих токена се придружују одговарајући вектори e_1 ,

e_2 и e_3 путем одабраног уграђивања речи. Те векторе је потребно објединити у један који би био прослеђен класификатору. Упросечавање их обједињује тако што враћа вектор $\frac{e_1+e_2+e_3}{3}$, док конкатенација враћа вектор $[e_1; e_2; e_3]$, односно вектор добијен надовезивањем сва три.

4.3 Скупови података за обучавање, валидацију и евалуацију модела

Коришћени подаци потичу из шест различитих скупова програма који заједно обухватају 149 апликација: 42 апликације потичу из пет устаљених скупова референтних програма (енгл. *benchmarks*), док преосталих 107 чине савремене апликације отвореног кода из репозиторијума *GraalVM Reachability Metadata*. Улога референтних програма је првенствено евалуација Јава програма на примерима из стварног света. Део програма је посебно направљен за систем *GraalVM*.

Barista. *Barista* [6] је скуп референтних програма отвореног кода посвећен мерењу перформанси Јава микросервиса. Коришћени су микросервиси *Micronaut* [20], *Spring Boot* [38] и *Quarkus* [31]. Конкретно, одабрано је укупно 5 апликација: *Pegasus*, *Similarity* и *Shopcart* из микросервиса *Micronaut*; *Petclinic* из микросервиса *Spring Boot* и *Tika* из микросервиса *Quarkus*. Изабране апликације oponaшају реалистична оптерећења микросервиса, попут обраде долазних захтева и размене података преко мреже, па мерење њихових перформанси добро одражава понашање у продукцијским условима. Поред тога, *Micronaut*, *Spring Boot* и *Quarkus* спадају међу најкоришћеније оквире за развој Јава микросервиса и могу се превести компајлером *GraalVM Native Image*, што их чини посебно погодним за евалуацију статичког профилера.

DaCapo. Скуп референтних програма отвореног кода *DaCapo* [9] обухвата широк спектар програма за евалуацију разних типова Јава апликација. Укључује програме са нетривијалним меморијским захтевима попут статичких анализатора кода [27], система за рендеровање [16] и система за управљање базама података [15]. Реч је о модерним и разноврсним програмима из комерцијалне и слободне употребе, међу којима су и симулација процеса на мрежи *AVR* микроконтролера, креирање *PDF* докумената, као и индексирање и претраживање великих колекција докумената.

DaCapo con Scala. Скуп референтних програма *DaCapo con Scala* [35] представља проширење скупа програма *DaCapo* фокусирањем на процену ефикасности Скала програма на Јава виртуелној машини. Садржи библиотеке за обраду програмских језика укључујући Скала компајлере и алате за форматирање кода. Поред тога, укључује и декодере класа за програмски језик Скала, као и апликације за обраду и повезивање *XML* података, такође написане у Скали.

Renaissance. *Renaissance* [30] је скуп разноликих референтних програма отвореног кода за евалуацију радних оквира (енгл. *framework*) и програма који раде на Јава виртуелној машини. Обухвата референтне програме из различитих парадигми као што су конкурентни, паралелни, функционални и објектно-оријентисани програми. Поред тога, садржи и апликације за обраду великих количина података, мрежну размену порука, обраду токова података (енгл. *stream processing*) и машинско учење. Међу разноврсним примерима су оптимизација генетских алгоритама помоћу библиотеке *Jenetics* [43], симулација интензивног серверског оптерећења помоћу радног оквира *Twitter Finagle* [41], као и превођење Скала програма помоћу компајлера *Dotty* [18].

SpecJVM2008. *SpecJVM2008* [37] је скуп референтних програма конзорцијума *SPEC* (енгл. *Standard Performance Evaluation Corporation*) намењен процени перформанси Јава извршивих окружења (енгл. *Java Runtime Environment*, скраћено *JRE*). Његови референтни програми обухватају стварне апликације и првенствено оптерећују процесор и меморијски подсистем, уз малу зависност од улазно-излазних операција. Садржи референтне програме из различитих области као што су компајлирање, компресија података, обрада *XML* докумената, криптографија, научна израчунавања и рад са базама података.

GraalVM Reachability Metadata. Јавно доступни репозиторијум *GraalVM Reachability Metadata* [24] на сервису *GitHub* садржи метаподатке неопходне за исправно превођење и покретање широког спектра Јава апликација и библиотека помоћу компајлера *GraalVM Native Image*. Реч је о разноврсним савременим апликацијама и библиотекама које се користе у пракси. Репозиторијум обухвата библиотеке за рад са базама података, компресију података, логовање, мрежну комуникацију, HTTP протоколе, веб сервере, апликације за слање и примање е-поште и друге компоненте које одражавају реалистичне сценарије употребе.

Подела скупа података

За обучавање су коришћени програми из репозиторијума *GraalVM Reachability Metadata*. Поред њих, коришћени су и скупови референтних програма *SpecJVM2008* и *DaCapo con Scala*.

За валидацију и тестирање модела коришћени су скупови програма *Barista*, *DaCapo* и *Renaissance*. Валидациони скуп чини 30% насумично изабраних референтних програма међу њима, док је преосталих 70% коришћено за тестирање. У табели 4.3 је приказана подела скупа референтних програма на валидацију и тестирање.

Величине добијених скупова података приказане су у табели 4.4. Скуп за обучавање обухвата 122 програма (107 из репозиторијума *GraalVM Reachability Metadata* и 15 из скупова *SpecJVM2008* и *DaCapo con Scala*) са укупно 5 086 839 метода. Валидациони скуп чини 7 програма са 366 992 методе, док тест скуп садржи 20 програма са укупно 1 048 347 метода.

Важна карактеристика описаних скупова података је изражена небалансираност класа (енгл. *class imbalance*): хладних метода има вишеструко више него топлх, па хладне методе чине приближно између 79,2% и 83,9% свих метода у сваком од скупова за обучавање, валидацију и тестирање (табела 4.4). Оваква расподела је природна за компајлер *GraalVM Native Image*. Наиме, анализа досежности (енгл. *reachability analysis*) током превођења открива све методе које програм потенцијално може да изврши, док се приликом извршавања често позива само мали део њих. Небалансираност класа утиче и на начин процене и коришћења модела. С једне стране, чиста тачност није поу-

Табела 4.3: Подела скупа референтних програма на валидацију и тест

Скуп програма	Валидација	Тестирање
<i>Barista</i>	micronaut-pegasus	quarkus-tika, micronaut-similarity, micronaut-shopcart, spring-petclinic
<i>DaCapo</i>	h2, lusearch	pmd, luindex, xalan, fop, sunflow
<i>Renaissance</i>	finagle-http, future-genetic, scrabble, fj-kmeans	reactors, scala-stm-bench7, par-mnemonics, akka-uct, mnemonics, scala-kmeans, philosophers, rx-scrabble, dotty, scala-doku, finagle-chirper

Табела 4.4: Број програма и метода по скуповима података

Скуп	Број програма	Број метода	Удео хладних метода
Обучавање	122	5 086 839	83,8%
Валидација	7	366 992	79,2%
Тест	20	1 048 347	83,9%

здана мера квалитета модела јер би модел који све методе класификује као хладне имао тачност већу од 80%, иако не препознаје ниједну топлу методу. Из тог разлога се квалитет класификације анализира матрицом конфузије (поглавље 3) која раздваја врсте грешака чије су последице различите: погрешно класификоване хладне методе непотребно повећавају величину извршивог програма, док погрешно класификоване топле методе успоравају његово извршавање.

Описана подела података извршена је по програмима, а не по појединачним методама. Разлози за такав приступ објашњени су у одељку 3.1: када би се методе распоређивале насумично, методе истог програма могле би да доспеју и у скуп за обучавање и у тест скуп, па би модел, коришћењем специфичности тог програма, на тесту деловао боље него што заиста генерализује. Поделом по програмима се то спречава, јер се методе једног програма не појављују истовремено у обучавању и у тесту, чиме се мери способност модела над подацима који нису раније виђени.

4.4 Одабир модела

Као што је описано у одељку 4.2, разматрано је 27 начина за одабир токена за класе и методе, а 6 начина за пакете. Разматрана су два начина агрегације и два различита уграђивања речи. При томе, уколико се користи упросечавање као метод агрегације, постоји још један додатни начин за одабир токена. Зато је за кодирање имена метода и класа разматрано укупно $(27 + (27 + 1)) \cdot 2 = 110$ конфигурација, док је за кодирање имена пакета разматрано $(6 + (6 + 1)) \cdot 2 = 26$ конфигурација. Потпуно квалификовано име методе се зато може кодирати на $26 \cdot 110 \cdot 110 = 314\,600$ начина. Узевши у обзир време потребно за обучавање и евалуацију једног модела, није изводљиво испитати сваку од ових комбинација. Зато је спроведена претрага за најбољим моделом над подскупом свих комбинација. Укупно су истражена 222 модела, по 111 за свако од

два коришћена уграђивања речи. Затим је за најбоље конфигурације извршена претрага хиперпараметара, чиме је број разматраних модела достигао укупно 288. На овај начин се пореде два уграђивања речи над идентичним конфигурацијама, односно поређење различитих уграђивања се одвија у једнаким условима. Начин на који су ови модели одабрани и подела на поменуте групе детаљно су описани у наставку поглавља.

Експериментална поставка

Да би резултати мерења били поновљиви, сва мерења су спроведена под строго контролисаним условима, по узору на методологију коришћену при евалуацији статичких профајлера *GraalSP* [11], *GraalNN* [22] и *GraalMHC* [12]. Иста поставка је коришћена током одабира модела и током евалуације.

Хардвер. Сва мерења су извршена на наменском (енгл. *bare-metal*) серверу са два процесора *Intel Xeon E5-2699 v3* (укупно 36 физичких језгара, односно 72 хардверске нити) и 256 GB *DDR4* радне меморије. Да би се смањила нестабилност мерења између узастопних покретања, мерени процес везује за једно језгро процесора, а машина се током мерења користи искључиво у ту сврху. Додатно, искључени су режим повременог убрзавања процесора (енгл. *turbo boost*) и дубока стања мировања процесора (енгл. *C-states*), радна фреквенција процесора је фиксирана, а привремене датотеке се смештају у радну меморију путем система датотека *tmpfs*, чиме се уклања утицај диска на мерење.

Софтвер. Као оперативни систем коришћен је *Oracle Linux 8.10* са језгром (енгл. *kernel*) верзије 5.15. Извршиви програми се генеришу алатом *Oracle GraalVM Native Image EE*, верзије 26-dev+8.1, уз употребу сакупљача отпада (енгл. *garbage collector*) *G1* и пратећих алата *GCC 11*, *Ant 1.10* и *CMake 3.22*. Модели се обучавају у програмском језику *Python 3.10* помоћу библиотеке *XGBoost 2.1.1*, а затим се преводе у формат *ONNX* (енгл. *Open Neural Network Exchange*) алатима *onnxmltools 1.13* и *skl2onnx 1.15*. Закључивање модела током компајлирања обавља библиотека *ONNX Runtime 1.20*. Сва превођења и извршавања одвијају се унутар изолованих контејнера *Podman*. Обучавање и одабир модела се спроводе унапред, ван компајера, а добијени модел се уграђује у сам компајлер, без потребе за помоћним сервисима или додатним зависностима током извршавања.

Протокол мерења. Време извршавања се мери у изолованим покретањима референтних програма, док се величина извршивог програма и време компајлирања добијају из поновљених узастопних превођења. Свако мерење се понавља најмање 10 пута, а референтни програми чија мерења више осцилују и већи број пута. Сирове вредности се најпре усредњавају аритметичком средином, засебно по сваком референтном програму, па се тек тако добијене вредности нормализују у односу на основну верзију и агрегирају геометријском средином по скупу референтних програма.

Процена квалитета модела

За референтни програм се прати његово време извршавања, величина извршивог програма и време компајлирања на валидационом скупу података коришћењем подразумеваног нивоа оптимизације (*O2*) и подразумеваног модела *S2* статичког профайлера. Модел се оцењује тако што се за сваки референтни програм из валидационог скупа израчунају поменуте три релативне вредности у односу на вредности добијене над одговарајућим референтним програмима без коришћења кодова имена метода. Затим се израчуна геометријска средина релативних вредности над свим референтним програмима и један модел се квантификује са три нумеричке вредности P_{rel} , S_{rel} и C_{rel} , које редом одговарају процени утицаја модела на време извршавања, величину извршивог програма и време компајлирања. Бољим моделима одговарају мање вредности.

Поређење модела је рађено помоћу тежинске суме вредности P_{rel} и S_{rel} . Вредност C_{rel} није коришћена јер постоји јака корелација између величине извршивог програма и времена компајлирања, при чему се мањи програми брже преводе и обрнуто [12]. Моделу $\mathcal{M}$ се додељује тежински збир (енгл. *Simple Additive Weighting*, скраћено *SAW*), дефинисан формулом:

$$SAW(\mathcal{M}) = w_p \cdot P_{rel}(\mathcal{M}) + w_s \cdot S_{rel}(\mathcal{M})$$

где су вредности w_p и w_s тежине. Одабран је однос $w_p : w_s = 6 : 1$ чиме се фаворизује време извршавања у односу на величину програма. Однос је изабран на основу резултата статичког профайлера *GraalMHC* где је постигнуто процентуално приближно 6 пута веће смањење величине извршивог програма у односу на повећање времена извршавања. Процењујемо да је модел $\mathcal{M}_1$ бољи од модела $\mathcal{M}_2$ уколико важи $SAW(\mathcal{M}_1) < SAW(\mathcal{M}_2)$.

Претрага скупа конфигурација

За оба разматрана уграђивања речи, *GloVe* и наменски обучено уграђивање, покренуте су идентичне претраге конфигурација. Скуп конфигурација је инкрементално формиран тако што је започето претрагом конфигурација које кодирају само имена метода. Затим се на основу добијених резултата бирају конфигурације које поред имена метода кодирају и класе. Помоћу нових резултата се бира и последњи скуп конфигурација који кодира имена метода, класа и пакета. На самом крају се за модел са најмањом вредношћу тежинског збира претражује простор различитих хиперпараметара и бира најбољи модел по тежинском збиру.

Конфигурације са именима метода. У овом скупу конфигурација су разматране све поменуте опције из одељка 4.2 за начин избора токена и начин агрегације кодова токена. Укупно је истражено 55 таквих конфигурација: 27 са конкатенацијом и 28 са упросечавањем као начином агрегације³.

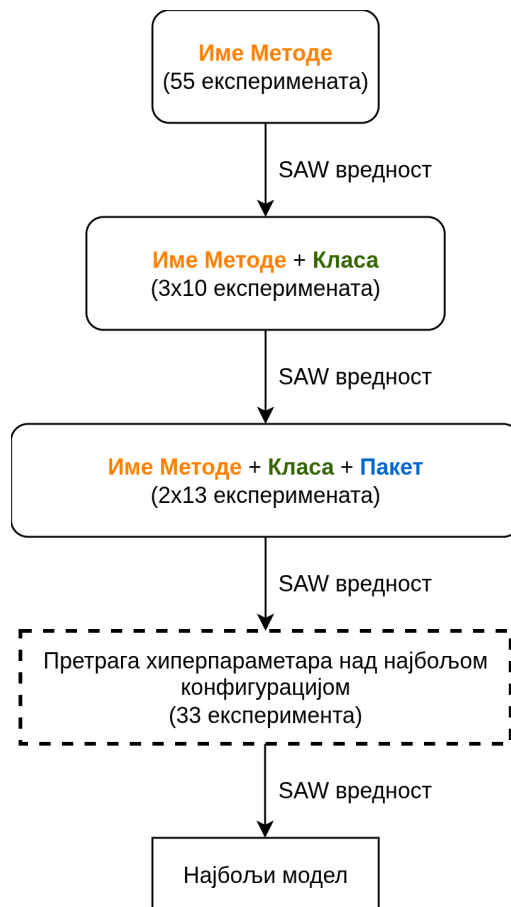
Конфигурације са именима метода и класама. Претходних 55 конфигурација се рангира по тежинским збировима. Кодирање имена метода извршава се неком од три најбоље конфигурације, док се кодирање класа извршава неком од десет најбоље ранжираних конфигурација из претходне групе конфигурација. У овој групи експеримената се укупно разматра $3 \cdot 10 = 30$ различитих конфигурација.

Конфигурације са именима метода, класама и пакетима. Претходних 30 конфигурација се рангира по тежинским збировима и бирају се најбоље две за кодирање имена метода и класа. Као што је поменуто у одељку 4.2, пакети се по својој природи другачије кодирају и за њих разматрамо 6 начина за одабир токена уз коришћење конкатенације и један више уколико се користи упросечавање као метод агрегације. Зато разматрамо $6 + 7 = 13$ нових конфигурација за кодирање пакета. За оба коришћена уграђивања речи се посматрају идентичне конфигурације, тако да се укупно разматра $2 \cdot 13 = 26$ експеримената у овој групи експеримената.

Поступак претраге је описан на слици 4.3. Укупно је испитано $55 + 30 + 26 = 111$ конфигурација и над најбољом од њих извршена претрага за најбољом од 33 комбинације хиперпараметара. Претрага хиперпараметара описана је

³Број 27 одговара свим изборима параметара n и k којих има $|\{(n, k) \mid 1 \leq n \leq 6 \wedge 0 \leq k \leq n\}| = 27$. Упросечавање допушта једну конфигурацију више, јер се њиме могу обухватити и сви доступни токени.

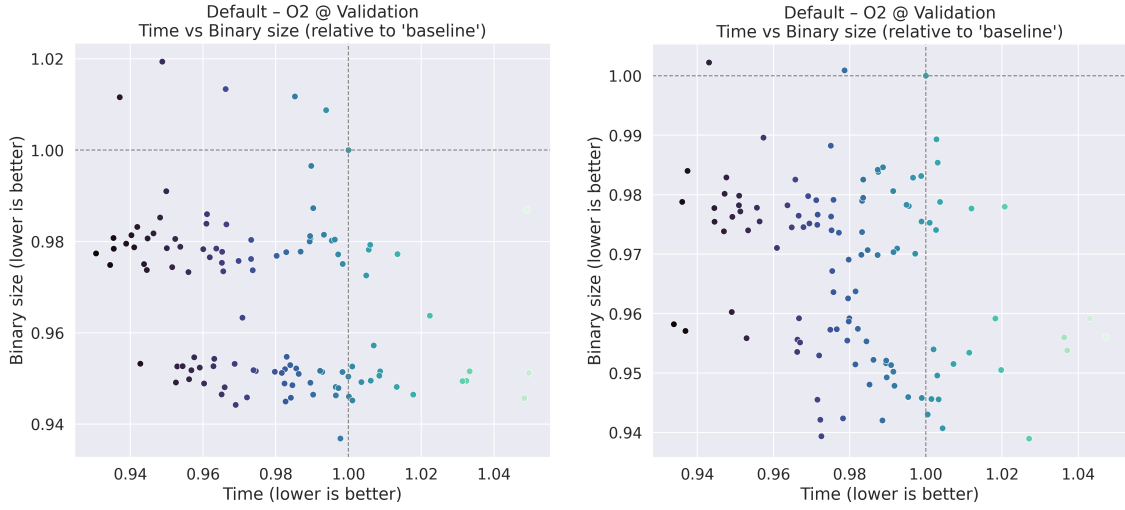
детаљно у наставку.



Слика 4.3: Поступак одабира најбољег модела

Слика 4.4 приказује резултате описаних конфигурација на валидационом скупу података. Хоризонтална оса представља релативне вредности времена извршавања, док вертикална оса представља релативне вредности величине извршивог програма. Мање вредности на обе осе одговарају бољим моделима, а тачка (1, 1) одговара моделу статичког профилера *GraalMHC* који не користи семантику из имена метода. Конфигурације за оба уграђивања речи су рангиране по тежинским збировима и одабране су најбоље конфигурације по коришћеним уграђивањима речи. Како се евалуације различитих конфигурација у великој мери поклапају за оба коришћена уграђивања речи, то су на овом нивоу резултати оба уграђивања речи упоредиви. Тек након претраге хиперпараметара се примећује значајнија разлика у корист наменски обученог уграђивања речи.

Најбоља конфигурација која користи уградњу речи *GloVe* кодира само



(1) *GloVe* уграђивање речи

(2) Наменски обучено уграђивање

Слика 4.4: Резултати 111 различитих модела на валидационом скупу за свако од коришћених уграђивања речи. Тамније боје представљају мање тежинске збирове, односно боље моделе. На обе осе су мање вредности боље.

имена метода тако што их упросечи са вредностима $n = 4$ и $k = 0$. Најбоља конфигурација са наменски обученим уграђивањем речи користи имена метода, класе и пакете. Имена методе се конкатенирају са вредностима $n = 1$ и $k = 1$; класе са вредностима $n = 6$ и $k = 4$, а пакети са $n = 4$.

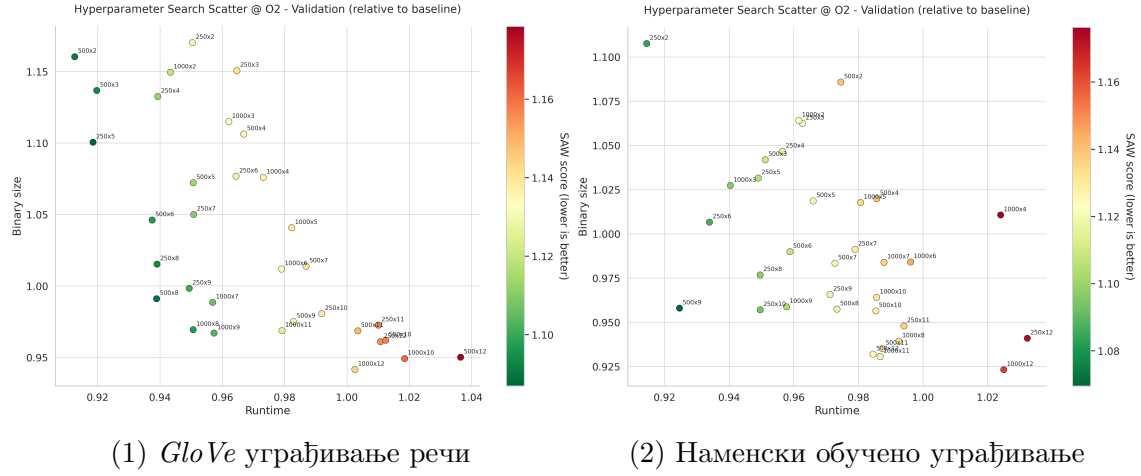
Претрага хиперпараметара

Сви истражени модели су користили исте вредности хиперпараметара класификационог модела *XGBoost* које су се показале као најбоље за статички профилер *GraalMHC*. Како је скуп атрибута промењен, пожељно је спровести претрагу за новим хиперпараметрима.

Конкретно, посматрана су два хиперпараметра: број стабала одлучивања T и дубина стабала одлучивања D . Разматране су вредности

$$T \in \{250, 500, 1000\}, \quad D \in \{2, 3, \dots, 12\},$$

односно по 33 модела за претрагу хиперпараметара. На слици 4.5 приказани су резултати претраге хиперпараметара. Хоризонтална оса приказује релативне вредности времена извршавања, док вертикална приказује релативне вредности величине извршених програма референтних програма на валидационом скупу. Мање вредности на обе осе одговарају бољим моделима.



Слика 4.5: Претрага хиперпараметара. Зелене вредности одговарају бољим моделима, а црвене лошијим.

Са слике 4.5 се уочава тренд да повећање дубине стабала D постепено смањује величину извршивог програма, уз мали утицај на време извршавања. Међутим, након дубине $D = 9$ даље продубљивање стабала доноси све мању додатну уштеду јер величина извршивог програма опада још само незначитно, док регресија у времену извршавања расте и најизраженија је код најдубљих модела ($D = 12$). Због тога оптималан модел није ни најдубљи ни највећи ансамбл, већ унутрашња тачка у простору хиперпараметара. Модели су рангирани према тежинским збировима, па је као најбољи одабран модел са наменски обучено уграђивањем речи и вредностима хиперпараметара $T = 500$ и $D = 9$.

Модел са најмањим тежинским збиром користи наменски обучено уграђивање речи. Поред бољих перформанси, ово уграђивања је значајно мање величине од 2,77 MB у поређењу са *GloVe* општенаменским уграђивањем чија најмања верзија захтева 163 MB. Уштеда у простору је посебно важна приликом испоруке компајлера, због ограничености самог простора за испоруку производа.

4.5 Евалуација одабраних модела

За оба уграђивања речи разматрано је по $111 + 33 = 144$ модела, чиме је укупно испитано $144 \cdot 2 = 288$ модела од којих је одабран је модел са најмањом вредношћу тежинског збира на валидационом скупу. Његова конфигурација

је описана у табели 4.5.

Табела 4.5: Конфигурација најбољег модела на валидационом скупу

Уграђивање речи	Кодирање имена	Кодирање класе	Кодирање пакета	T	D
Наменски обучено	конкатенација; $n = 1, k = 1$	конкатенација; $n = 6, k = 4$	конкатенација; $n = 4$	500	9

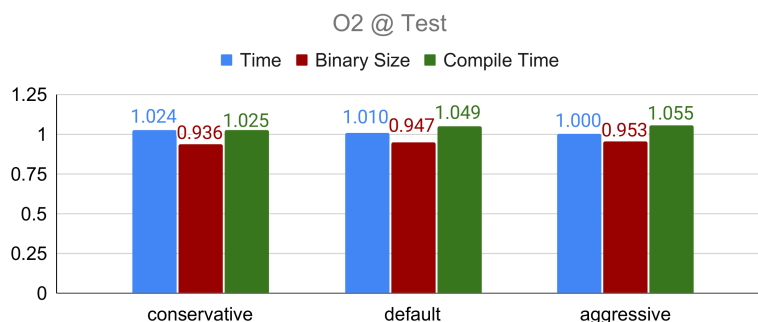
Модел је евалуиран на тест скупу над моделом $S2$ статичког профажлера *GraalMHC*. Поред тога, извршена је евалуација над конфигурацијама *conservative* и *aggressive* статичког профажлера *GraalMHC⁺⁺*. Како је конфигурација *default* статичког профажлера *GraalMHC⁺⁺* идентична моделу $S2$ статичког профажлера *GraalMHC*, сви резултати су приказани у контексту статичког профажлера *GraalMHC⁺⁺*.

Свака конфигурација је евалуирана на два нивоа оптимизација: $O2$ и $O3$. Резултати евалуације су приказани као релативне промене добијене коришћењем статичког профажлера *GraalMHC⁺⁺* проширеног семантиком из имена метода у односу на основну верзију профажлера која их не користи (енгл. *baseline*). Приказане су вредности за време извршавања, величину извршивог програма и време компајлирања на скупу за тестирање добијене по протоколу мерења описаном у одељку 4.4.

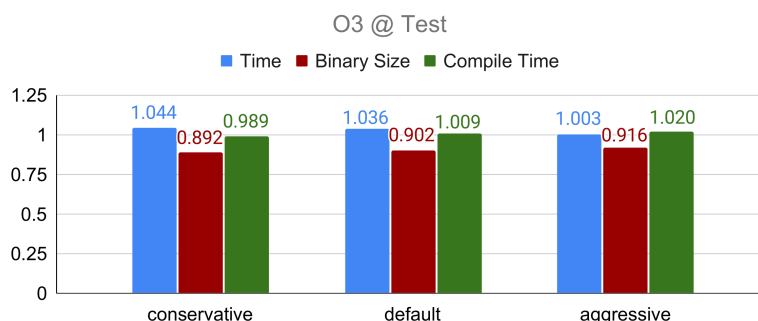
Резултати евалуације на нивоу оптимизације $O2$ су приказани на слици 4.6. Са конфигурацијом *default*, *GraalMHC⁺⁺* проширен именима метода, постиже 5,3% мању величину извршивог програма зарад 1% регресије у времену извршавања. Конфигурације *conservative* и *aggressive* прате сличан тренд где се постиже уштеда од 4,7% до 6,4% на величини извршивог програма са регресијом на перформансама до 2,4%.

Резултати са оптимизационим нивоом $O3$ су приказани на слици 4.7. Најбољи модел очекивано испољава већу уштеду у величини извршивог програма уз нешто већу регресију у времену извршавања у односу на ниво оптимизације $O2$. Величина извршивог програма је мања за 8,4% до 10,8% са регресијама до 4,4%, при чему је регресија најмања у конфигурацији *aggressive* где је 0,3%.

Однос смањења величине извршивог програма и регресије у времену извршавања је приближно исти као у унапређењима која доноси статички профажлер *GraalMHC⁺⁺* без семантике из имена метода. То значи да добијени резултати представљају прихватљив компромис у ситуацијама у којима је ме-



Слика 4.6: Резултати најбољег модела на тест скупу података на нивоу оптимизације *O2*

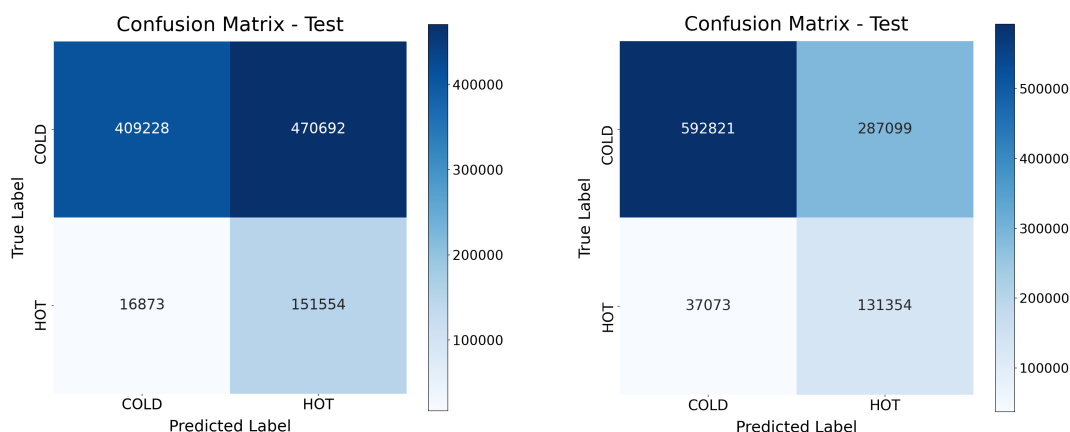


Слика 4.7: Резултати најбољег модела на тест скупу података на нивоу оптимизације *O3*

моријски простор ограничен, а брзина извршавања није кључна. Посебно, под-разумевана конфигурација (*default* са нивоом оптимизација *O2*) нуди значајно смањење величине извршивог програма са малим утицајем на перформансе.

Квалитет модела се може видети и поређењем матрица конфузије на тест скупу са моделом који не користи семантику из имена метода (слика 4.8). Смањење величине извршивог програма долази примарно од тачно класификованих хладних метода. Односно, што је већи број тачно класификованих хладних метода, величина извршивог програма је мања. Наиме, погрешна класификација хладних метода доводи до тога да компајлер посвећује више значаја методама које се не извршавају често што се огледа у непотребно већем меморијском заузећу. Са друге стране, погрешна класификација топлих метода доводи до занемаривања метода које се често извршавају и погоршава време извршавања. Проширени модел (слика 2) тачно класификује 592 821 хладну методу наспрам 409 228 код основног модела (слика 1), чиме одзив за класу хладних метода расте са 46,5% на 67,4%, а укупна тачност са 53,5% на

69,1%. Из тог разлога је величина извршивог програма мања. Са друге стране, број погрешно класификованих топлих метода је већи (37 073 наспрам 16 873) па одзив за класу топлих метода опада са 90,0% на 78,0%, чиме се време извршавања повећава. Прецизност за класу топлих метода је ипак већа код проширеног модела (31,4% наспрам 24,4%), јер основни модел знатно чешће хладне методе погрешно проглашава топлим.



(1) Без семантике из имена метода (2) Са семантиком из имена метода

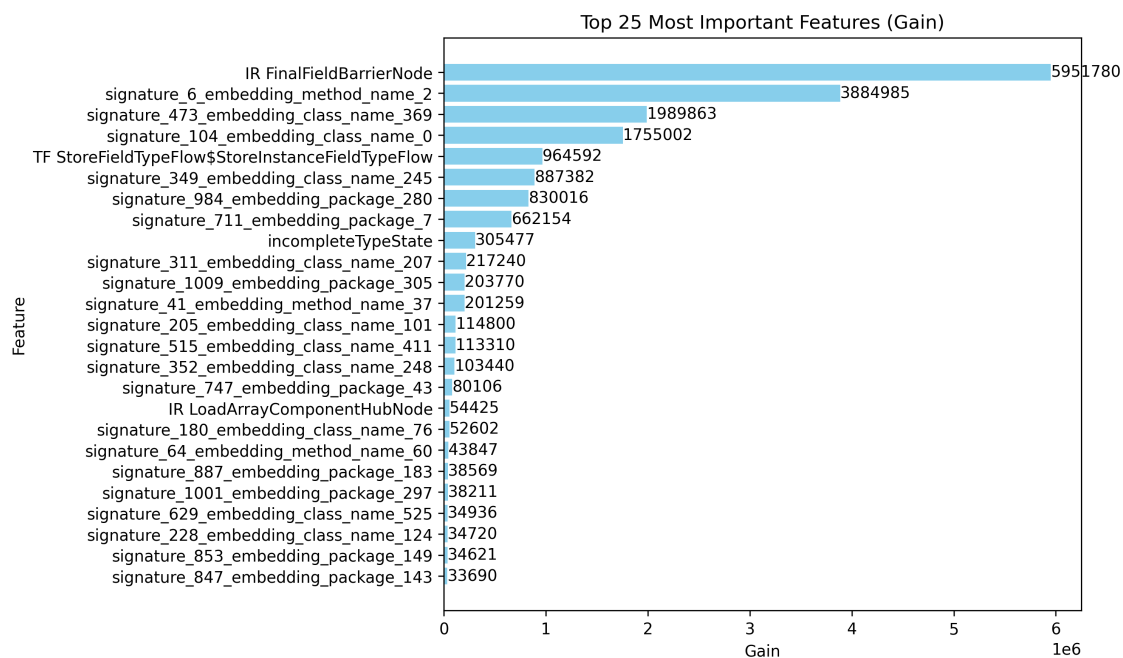
Слика 4.8: Матрице конфузије на тест скупу

Значај семантике имена метода се може видети на слици 4.9 која приказује најутицајније атрибуте обученог ансамбла стабала *XGBoost*. Модел укупно има 1320 атрибута који потичу од (1) потписа метода, (2) модификатора приступа метода, (3) међурепрезентације метода и (4) графа тока података метода. У првој категорији је 1100 нових атрибута у односу на *GraalMHC* који потичу од кодирања потпуно квалификованих имена метода. Међу њима, 100 потиче од имена методе, 600 од имена класе и 400 од пакета. Ти атрибути су на слици 4.9 именовани као `signature_RB_embedding_DE0`, где је RB редни број атрибута, а DE0 може бити `method_name`, `class_name` или `package` у зависности од ког дела потпуно квалификованог имена метода произилази.

Међу 25 најутицајнијих атрибута, 21 потиче од кодирања имена метода. Међу њима, три атрибута су добијена из имена метода, десет из имена класа и осам из пакета. То указује да информације садржане у именима метода носе значајан сигнал за разликовање топлих и хладних метода и да, поред самих назива метода, информације из класа и пакета носе важне семантичке информације. Модел током доношења одлука у великој мери користи семан-

ГЛАВА 4. ИМПЛЕМЕНТАЦИЈА И ЕВАЛУАЦИЈА

тичке информације издвојене из потпуно квалификованих имена метода, што потврђује основну претпоставку овог рада.



Слика 4.9: Утицај атрибута најбољег модела класификационог ансамбла стабала *XGBoost*

Глава 5

Закључак

Најзначајнији допринос овог рада је додавање семантике из имена метода у статичке профајлере што представља новину у односу на постојећа решења која не користе имена метода за предвиђање профила извршавања програма. Идеја је заснована на претпоставци да имена метода могу да носе значајне информације о понашању метода чиме могу да помогну бољем предвиђању профила. Решење је имплементирано у оквиру статичког профајлера *GraalMHC* и извршена је исцрпна претрага за оптималном конфигурацијом за кодирање потпуно квалификованих имена метода. Тиме је унапређен квалитет профајлера бољом класификацијом метода према учесталости позива и добијено значајно смањење величине извршивог програма уз регресију перформанси сразмерну оној која се јавља у постојећим решењима.

У односу на основну верзију која не користи имена метода, проширени статички профајлер *GraalMHC* за модел *S2* обезбеђује 5,3% мањи извршиви програм уз регресију у времену извршавања од 2,4% на нивоу оптимизација *O2*. На нивоу оптимизација *O3*, величина извршивог програма је 9,8% мања уз повећање времена извршавања 3,6%. Статички профајлер *GraalMHC⁺⁺* над конфигурацијама *conservative* и *aggressive*, проширен кодирањем имена метода, даје од 6,4% до 10,8% мање извршиве програме уз регресију перформанси од 2,4% до 4,4%. Поред тога, коришћено је посебно обучено уграђивање речи које је обезбедило боље резултате од општенаменског уграђивања речи *GloVe* уз значајно мање коришћење простора од 2,77 MB у односу на 163 MB.

Осим проширења статичког профајлера *GraalMHC*, сличан приступ додавању семантике из имена метода је имплементиран у оквиру статичког профајлера *GraalNN* [22] који предвиђа вероватноће извршавања грана. Добијено

је смањење до 6,53% величине извршивог програма без статистички значајног губитака на времену извршавања. Тиме се потврђује да имена метода носе довољно важне семантичке информације и да могу да побољшају предвиђања у различитим проблемима.

Даљи рад се може одвијати у два правца. Један је анализа и испитивање утицаја тренутног решења. Конкретно, могу се урадити додатни експерименти са циљем провере утицаја семантике која је додата у модел. То се може урадити анализом података добијеним из поређења примера на којима проширени профајлер успешно класификује методе, а основна верзија не. Поред тога, може се испитати отпорност утицаја маскирања кода (енгл. *code obfuscation*) на проширени профајлер. Маскирање кода представља измену изворног кода програма чиме он постаје тежи за разумевање и праћење са циљем скривања унутрашње логике. Тиме се отежава неовлашћена измена програма и обрнути инжењеринг (енгл. *reverse engineering*).

Други правац је побољшање решења коришћењем сложенијих модела. Напреднији језички модели, попут великих језичких модела (енгл. *large language models*, скраћено *LLM*) [46], имају боље разумевање семантике чиме се очекује боља коначна класификација. Са друге стране, једна од предности овог рада јесте коришћење ансамбала стабала за класификацију који ни временски ни просторно нису захтевни. Зато коришћење сложенијих модела, иако може допринети бољим резултатима, потенцијално може ограничити практичну употребу профајлера значајним повећавањем времена компајлирања.

Литература

- [1] Graal Compiler. on-line at: <https://www.graalvm.org/latest/reference-manual/java/compiler/>.
- [2] GraalVM. on-line at: <https://www.graalvm.org/>.
- [3] GraalVM Native Image. on-line at: <https://www.graalvm.org/latest/reference-manual/native-image/>.
- [4] Polyglot Programming. on-line at: <https://www.graalvm.org/latest/reference-manual/polyglot-programming/>.
- [5] Truffle Language Implementation Framework. on-line at: <https://www.graalvm.org/latest/graalvm-as-a-platform/language-implementation-framework/>.
- [6] Barista: A modern, open, and diversified benchmark suite for the jvm, 2026. Приступљено: 14. 6. 2026.
- [7] Apache Software Foundation. Apache maven, 2026. Приступљено: 7. 6. 2026.
- [8] Thomas Ball and James R. Larus. Branch prediction for free. *SIGPLAN Not.*, 28(6):300–313, June 1993.
- [9] Stephen M Blackburn, Robin Garner, Chris Hoffmann, Asjad M Khang, Kathryn S McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z Guyer, et al. The dacapo benchmarks: Java benchmarking development and analysis. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 169–190, 2006.

- [10] Tianqi Chen and Carlos Guestrin. XGBoost: A scalable tree boosting system. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, page 785–794. ACM, August 2016.
- [11] Milan Čugurović, Milena Vujošević Janičić, Vojin Jovanović, and Thomas Würthinger. GraalSP: Polyglot, efficient, and robust machine learning-based static profiler. *Journal of Systems and Software*, 213:112058, 2024.
- [12] Milan Cugurovic, Aleksandar Prokopec, Boris Spasojevic, Vojin Jovanovic, and Milena Vujošević Janičić. GraalMHC: ML-based method-hotness classification for binary-size reduction in optimizing compilers. In *Proceedings of the 35th ACM SIGPLAN International Conference on Compiler Construction*, CC '26, page 1–13, New York, NY, USA, 2026. Association for Computing Machinery.
- [13] GitHub, Inc. Github, 2026. Приступљено: 7. 6. 2026.
- [14] James Gosling, Bill Joy, Guy Steele, Gilad Bracha, Alex Buckley, Daniel Smith, and Gavin Bierman. The Java Language Specification, Java SE 25 Edition. on-line at: <https://docs.oracle.com/javase/specs/jls/se25/jls25.pdf>.
- [15] HSQLDB Group. H2 database engine, 2025.
- [16] Christopher Kulla. Sunflow Render System Documentation, 2026.
- [17] Chris Lattner. Llvm and clang: Next generation compiler technology. In *The BSD conference*, volume 5, pages 1--20, 2008.
- [18] Guillaume Martres. Interactive development using the dotty compiler (tool paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala*, SCALA 2017, page 41–44, New York, NY, USA, 2017. Association for Computing Machinery.
- [19] Charith Mendis, Alex Renda, Saman Amarasinghe, and Michael Carbin. It-hemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In *International Conference on Machine Learning*, pages 4505--4515. PMLR, 2019.

- [20] Micronaut Team. Micronaut – a modern, jvm-based, full-stack framework for building modular, easily testable microservice and serverless applications, 2026. Приступљено: 14. 6. 2026.
- [21] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.
- [22] Lazar Milikic, Milan Cugurovic, and Vojin Jovanovic. GraalNN: Context-sensitive static profiling with graph neural networks. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 123--136, 2025.
- [23] Angélica Aparecida Moreira, Guilherme Ottoni, and Fernando Magno Quintão Pereira. Vespa: static profiling for binary optimization. *Proceedings of the ACM on Programming Languages*, 5(OOPSLA):1--28, 2021.
- [24] Oracle. GraalVM reachability metadata repository. <https://github.com/oracle/graalvm-reachability-metadata>, 2026. Приступљено: 8. 6. 2026.
- [25] Oracle Corporation. The Java HotSpot performance engine architecture. <https://www.oracle.com/java/technologies/whitepaper.html>, 2019. Приступљено: 9. 7. 2026.
- [26] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532--1543, 2014.
- [27] PMD Open Source Project. PMD Documentation, 2025.
- [28] Martin F. Porter. An algorithm for suffix stripping. *Program*, 14(3):130--137, 1980.
- [29] LLVM Project. LLVM language reference manual, 2026. Приступљено: 26. 3. 2026.
- [30] Aleksandar Prokopec, Andrea Rosà, David Leopoldseder, Gilles Duboscq, Petr Tůma, Martin Studener, Lubomír Bulej, Yudi Zheng, Alex Villazón, Doug Simon, Thomas Würthinger, and Walter Binder. Renaissance: benchmarking suite for parallel applications on the jvm. In *Proceedings of the 40th*

- ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 31–47, New York, NY, USA, 2019. Association for Computing Machinery.
- [31] Quarkus Team. Quarkus: Supersonic subatomic java, 2019. Приступљено: 14. 6. 2026.
 - [32] Nadav Rotem and Chris Cummins. Profile guided optimization without profiles: A machine learning approach. *arXiv preprint arXiv:2112.14679*, 2021.
 - [33] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson, 4th edition, 2020.
 - [34] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.
 - [35] Andreas Sewe, Mira Mezini, Aibek Sarimbekov, and Walter Binder. Da capo con scala: design and analysis of a scala benchmark suite for the java virtual machine. *SIGPLAN Not.*, 46(10):657–676, October 2011.
 - [36] Ching-Yen Shih, Drake Svoboda, Siao-Jie Su, and Wei-Chung Liao. Static branch prediction for LLVM IRs using machine learning. 2021.
 - [37] Kumar Shiv, Kingsum Chow, Yanping Wang, and Dmitry Petrochenko. Specjvm2008 performance characterization. In David Kaeli and Kai Sachs, editors, *Computer Performance Evaluation and Benchmarking*, pages 17–35, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.
 - [38] Spring Team. Spring boot, 2005. Приступљено: 14. 6. 2026.
 - [39] Richard M Stallman et al. Using the GNU compiler collection. *Free Software Foundation*, 4(02), 2003.
 - [40] Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. MLGO: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808*, 2021.
 - [41] Twitter, Inc. Finagle. <https://twitter.github.io/finagle/>, 2026. Приступљено: 14. 6. 2026.

- [42] Zoe Wang. GraalVM Native Image Support in the AWS SDK for Java 2.x. on-line at: <https://aws.amazon.com/blogs/developer/graalvm-native-image-support-in-the-aws-sdk-for-java-2-x/>.
- [43] Franz Wilhelmstötter. Jenetics. <https://jenetics.io>, 2021. Приступљено: 14. 6. 2026.
- [44] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th Annual International Symposium on Microarchitecture, MICRO 27*, page 1–11, New York, NY, USA, 1994. Association for Computing Machinery.
- [45] Cheng Zhang, Hao Xu, Sai Zhang, Jianjun Zhao, and Yuting Chen. Frequency estimation of virtual call targets for object-oriented programs. In *European Conference on Object-Oriented Programming*, pages 510–532. Springer, 2011.
- [46] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 2026.

Биографија аутора

Стефан Миленковић рођен је 18. маја 2001. године у Крушевцу. Математичку гимназију у Београду завршио је 2020. године, а основне студије на Математичком факултету Универзитета у Београду, смер Рачунарство и информатика, 2024. године. Исте године уписао је мастер студије на истом факултету, смер Математика и рачунарство. Од октобра 2024. године ради као сарадник у настави на Математичком факултету, где је држао вежбе из предмета *Преводјење програмских језика*, *Програмске парадигме*, *Лексичка анализа* и *Скрипти програмирање*. Од марта 2025. до маја 2026. године похађао је истраживачку праксу у компанији *Oracle*, а током лета 2025. године истраживачку праксу на универзитету *EPFL* у Лозани, у Швајцарској.