УНИВЕРЗИТЕТ У НОВОМ САДУ
**ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА У НОВОМ САДУ**

Марко Ердељи

# Транспилација из *Python*-а у *Rust*: Студија случаја

ДИПЛОМСКИ РАД
- Основне академске студије -

Нови Сад, 2024

Образац **Q2.НА.15-04** - Издање 1

## Садржај

## 1 Увод

Транспилација (енгл. *transpilation*)[1] представља значајан корак у оптимизацији и унапређењу перформанси софтвера. У савременом софтверском инжењерству, употреба трансформација изворног кода у изворни (енгл. *source-to-source transformation*)[2] показала се као кључна за постигнуће ефикасније извршне верзије програма. Ова техника отвара пут ка дубљем разумевању структуре и перформанси софтвера, омогућавајући програмерима да директно утичу на трансформацију изворног кода према специфичним захтевима извршења [1].

Први транспајлери су развијени у касним 70-им и раним 80-им годинама прошлог века. Године 1978, *Intel* је предложио аутоматски преводилац кода за конверзију 8-битних програма у њихове еквивалентне 16-битне програме [2]. *XLT86*™ је предложен 1981. године као преводилац асемблерског језика са 8080 на 8086, са циљем да аутоматски трансформише фајлове *ASM* типа у фајлове *A86* типа [3].

Овај рад фокусира се на транспилацију између два програмска језика високог нивоа (енгл. *high-level programming languages*)[3] , *Python*-а и *Rust*-а, истражујући процесе, изазове и резултате овог задатка. Кроз анализу алата и техника које су коришћене, циљ је да се истражи како транспилација може да унапреди перформансе и одрживост софтверских решења у контексту машинског учења.

У уводном делу рада биће детаљно истражена техника транспилације, са посебним фокусом на процес транспилације *Python*-а у *Rust*. Циљ истраживања је дубље разумевање техника транспилације, као кључне за оптимизацију и унапређење перформанси софтвера. Биће анализирани аспекти *Python*-а и *Rust*-а, са фокусом на њихове предности и ограничења у контексту перформанси, безбедности и скалабилности.

Даље, истраживаће се постојећи транспајлери (енгл. *transpilers*)[4], који се користе за *Python* и *Rust* транспилацију, са анализом алата и техника. Фокус ће бити на документацији процеса транспилације, укључујући кораке за постављање и коришћење транспајлера, као и анализу кода пре и после транспилације. Посебна пажња биће посвећена студији случаја кроз примере мањих пројеката у области машинског учења,

[1] Процес превођења између програмских језика високог нивоа.

[2] Техника која укључује конверзију између различитих језика високог нивоа без међупроцеса са машинским кодом.

[3] Програмски језици високог нивоа су програмски језици који су дизајнирани да буду разумљиви људима и омогућавају апстракцију сложених операција.

[4] Алати за транспилацију кода из једног програмског језика у други.

што ће омогућити дубље разумевање изазова, решења и потенцијалних унапређења.

На крају, у евалуацији ће се сумирати главни налази истраживања, дискутовати предности и мане транспилације, као и дати предлози за будућа истраживања која би могла унапредити ефикасност и примену транспајлера у практичним софтверским решењима.

# 2 Теоријске основе

## 2.1 Дефиниција и значај транспајлера

Хирцел и коаутори [4] дефинишу процес транспилације као ону у којој се софтвер пише на изворном језику, али се компајлира и извршава у различитом програмском језику. Чабер и коаутори [5] објашњавају да је транспилација метод генерисања кода у коме дође до превода из једног програмског језика високог нивоа у други програмски језик ниског нивоа. Разлози зашто је транспилација значајна:

- **Миграција (енгл. *Migration)*:** Омогућава пренос legacy кода на модерније програмске језике, чиме се унапређује одржавање и распространање апликација [6].
- **Компатибилност:** Генерисање кода који је компатибилан са старијим верзијама програмских језика, док развојници користе нове функционалности доступне у модернијим верзијама [7].
- **Преусмеравање вештина програмирања:** Омогућава адаптацију програмера или тимских преференција транспилацијом кода у језик који је погоднији за разумевање или вештине тима [8].
- **Побољшање перформанси:** Унапређење перформанси апликација транспилацијом делова кода у језик који има бољи компајлер или је боље оптимизован за циљну платформу [6].

## 2.2 Структура и фазе транспилације

Архитектура транспајлера може се поделити на два дела – фронтенд (енгл. *frontend*) и бекенд (енгл. *backend*). Фронтенд преводи изворни језик у међурепрезентацију. Бекенд ради са интерном репрезентацијом да би произвео код на излазном језику [9].

### 2.2.1 Фронтенд транспајлера

Фронтенд транспајлера укључује неколико фаза које се секвенцијално извршавају. Структура је приказана на слици 2.1 [10].

- **Лексичка анализа (енгл. *lexical analysis*):** Анализира ниску карактера која је представљена и дели је на токене који су легални чланови вокабулара језика у којем је програм написан.

- **Синтаксна анализа (парсирање) (енгл. *syntactic analysis/parsing*):** Процесира секвенцу токена и производи међурепрезентацију, као што су стабло парсирања или секвенцијални међурепрезентацијски код.

- **Семантичка провера (енгл. *semantic checking*):** Врши проверу програма за статичку-семантичку исправност, утврђујући да ли програм задовољава статичке-семантичке особине захтеване од изворног језика.

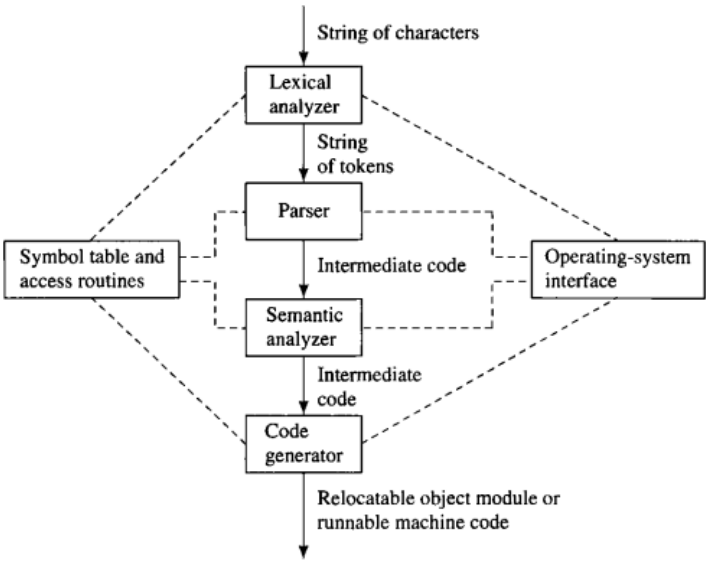
- **Генерисање кода (енгл. *code generation*):** Трансформише међурепрезентацију у еквивалентни код у циљном језику или другом облику прилагодљивом за циљну платформу.

Транспајлери могу бити имплементирани као једнопролазни или вишепролазни системи, слично као и класични компајлери. Једнопролазни транспајлери пружају бржу компилацију али може бити теже постићи високу ефикасност генерисаног кода. Вишепролазни приступи омогућавају бољу оптимизацију и квалитетнији генерисани код, али уз трошак дужег времена компилације.

Као и код компајлера, након што се транспилација заврши, програми или њихови делови обично морају бити повезани (линковани) како би се међусобно повезали и са потребним библиотечким рутинама, те учитани и премештени у меморију ради извршавања.

За многе језике високог нивоа, четири фазе могу бити комбиноване у један пролаз кроз изворни програм како би се произвео брз једнопролазни компајлер (енгл. *compiler*) [5]. Такав компајлер може бити сасвим одговарајућ за повремене кориснике, где је циљ обезбедити брз циклус измена, компилације и дебаговања. Међутим, такав компајлер генерално није у могућности да произведе веома ефикасан код. Алтернативно, фазе лексичке и синтаксне анализе могу бити комбиноване у пролаз који производи табелу симбола и неки облик међурепрезентације.

[5] Компајлер је програм који може да прочита програм написан на једном језику и преведе га у еквивалентни програм на другом језику. За разлику од транспајлера, који преводи програм са једног језика на други у истом нивоу апстракције [11].




Slika 2.1: Структура *frontend*-а традиционалног транспајлера [10]

### 2.2.2 Бекенд транспајлера

Бекенд транспајлера се бави оптимизацијом кода. Ево како неке од оптимизационих техника које се користе у бекенду транспајлера могу да се примене [10]:

- **Замена скаларних референци на низове:** Транспајлери могу трансформисати приступе низовима у скаларним променљивама[6] где је то могуће, како би се смањило оптерећење приступа меморији и побољшала перформанса у циљном језику.
- **Интеграција процедура:** Спајање више процедура или функција у једну целину може смањити трошкове повезане са позивом функција (енгл. *overhead*)[7].
- **Оптимизација репних позива, укључујући елиминацију репне рекурзије:** Оптимизација рекурзивних позива функција ради избегавања прекорачења стека и побољшања перформанси је од суштинског значаја приликом транспилације, поготово приликом конверзије језика као што су Python у Rust.

[6] Скаларна променљива је појединачна променљива која чува једну вредност, за разлику од низова који могу садржати више елемената.

[7] *Overhead* позива функција је додатни ресурс (меморија и процесорско време)

- **Замена скаларних агрегата:** Слично замени скаларних референци на низове, ова оптимизација се фокусира на оптимизацију приступа агрегатним структурама података[8] (као што су структуре или класе), разбијајући их на појединачне компоненте ради побољшања перформанси у циљном језику.

- **Ретка константна пропагација услова:** Пропагација константи кроз условне изразе ради поједностављења израза и потенцијалног елиминације непотребних грана може унапредити ефикасност генерисаног кода током транспилације.

- **Интерпроцедурална пропагација константи:** Проширивање пропагације константи преко граница функција помаже у инлајнирању и оптимизацији кода преко граница модула[9], унапређујући перформансе у циљном језику.

- **Специјализација и клонирање процедура:** Прилагођавање функција или процедура на основу њихових контекста употребе ради оптимизације специфичних случајева који се јављају током транспилације, чиме се побољшава укупна ефикасност кода.

- **Елиминација мртвог кода:** Уклањање недоступних или непотребних сегмената кода током транспилације осигурава да генерисани код буде компактан и ефикасан у циљном језику.

Ове технике се прилагођавају карактеристикама и ограничењима како изворног, тако и циљног језика. Док неке ниске оптимизације као што су машински идиоми или планирање инструкција могу бити мање применљиве због високог нивоа транспилације, технике које се фокусирају на семантичко очување и унапређење перформанси су кључне за ефикасно функционисање транспајлера.

## 2.3 *Python*

*Python* је опште-наменски програмски језик који постоји већ дуго: Гвидо ван Росум, креатор *Python*-а, почео је да развија овај језик још 1990. године. Језик је стаби-

[8] Агрегатне структуре података су комплексни подаци који садрже више различитих елемената или поља.

[9] Модул је организациона јединица у софтверу која садржи повезане функције, податке или друге ресурсе. Проширивање пропагације константи преко граница модула значи да се константе могу пропагирати и оптимизовати преко функција које припадају различитим модулима, што може побољшати перформансе и ефикасност генерисаног кода током транспилације.

лан и веома високог нивоа, динамичан (енгл. *dynamic*)[10] , објектно-оријентисан (енгл. *object-oriented*)[11] и портабилан. *Python* ради на свим главним хардверским платформама и оперативним системима [12].

*Python* нуди високу продуктивност у свим фазама животног циклуса софтвера: анализа, дизајн, прототиповање, кодирање, тестирање, отклањање грешака, подешавање, документација, имплементација и одржавање. Популарност *Python*-а је константно расла током година. Данас је познавање *Python*-а предност за сваког програмера, јер је *Python* присутан у свим нишама и има корисне улоге у било ком софтверском решењу.

*Python* пружа мешавину једноставности, практичности и моћи. Продуктивност кодирања са *Python*-ом расте захваљујући његовој конзистентности и регуларности, богатој стандардној библиотеци и алатима који су лако доступни за њега.

*Python* је језик веома високог нивоа. То значи да *Python* користи виши ниво апстракције, концептуално даљи од основне машине, него класични компајлирани језици као што су *C*, *C++* и *Fortran*, који се традиционално називају језици високог нивоа. *Python* је једноставнији, бржи за обраду (и за људске мозгове и за програмске алате) и регуларнији од класичних језика високог нивоа. Ово омогућава високу продуктивност програмера и чини *Python* атрактивним алатом за развој. Добри компајлери за класичне компајлиране језике могу генерисати бинарни машински код који ради брже од *Python* кода [12].

### 2.3.1 Зашто користити *Python*?

Постоји неколико разлога зашто користити Python [13]:

- **Портабилност:** *Python* ради на готово свим оперативним системима, укључујући *Linux/Unix*, *Windows*, *Mac*, *OS2* и друге.
- **Интеграција:** *Python* се може интегрисати са *COM*, *.NET* и *CORBA* објектима. Постоји *Jython* имплементација која омогућава коришћење *Python*-а на било којој *Java* платформи. *IronPython* је имплементација која даје *Python* програмери-

[10] Језик који подржава динамичко типизирање, што значи да се типови података могу одредити и изменити током извршавања програма.

[11] Објектно- оријентисан програмски језик омогућава организацију кода у објекте, који су инстанце класа. Ови објекти могу садржати податке и функције.

ма приступ *.NET* библиотекама. *Python* такође може садржати обавијен *C* или *C++* код.

- **Лакоћа коришћења:** Врло је лако упознати се са *Python*-ом и почети писати програме. Јасна и читљива синтакса чини апликације једноставним за креирање и отклањање грешака.

- **Флексибилност:** Стално се развијају нови проширења за *Python*, као што су приступ базама података, аудио/видео уређивање, графички кориснички интерфејс, веб развој и тако даље.*Python* је један од најфлексибилнијих језика. Лако је бити креативан са кодом и решавати дизајнерске и развојне проблеме.

- **Отвореност кода (енгл. *open source*)**[12] **:** *Python* је језик отвореног кода, што значи да се може слободно користити и дистрибуирати.

### 2.3.2 Мане *Python*-а

*Python* има неколико недостатака [14]:

- **Увећана потрошња меморије:** Типови података у *Python*-у троше значајно више меморије у поређењу са сличним типовима у другим језицима као што је *C*. На пример, цео број у *Python*-у троши 28 бајтова, док у *C*-у троши само 4 бајта. Ово је углавном због метаподатака које *Python* одржава за сваки објекат, укључујући бројаче референци и динамичке информације о типовима.

- **Одлагање сакупљања отпада (енгл. *garbage collection*)**[13]**:** *Python* користи сакупљача отпада, што може одложити ослобађање меморије и повећати потрошњу меморије у поређењу са језицима који користе другачије управљање меморијом.

- **Прекомерна потрошња ресурса:** *Python*-ова висока потрошња меморије и перформанси могу бити проблематични, нарочито у критичним деловима кода који су захтевни за меморију и перформансе.

[12] Односи се на софтвер чији изворни код је доступан јавности за коришћење, измену и дистрибуцију. Корисници могу слободно да прегледају, модификују и деле софтвер, што промовише транспарентност и сарадњу.

[13] Метод управљања меморијом који аутоматски ослобађа меморију коју више није могуће користити.

- **Недостатак детаљних информација:** Постојећи алати за профилисање (енгл. *profiling*)[14] *Python*-а можда не пружају детаљне информације о перформансама, јер не решавају специфичне изазове окружења извршења *Python*-а.

- **Оптимизација преко нативних библиотеках (енгл. *native libraries*)[15]:** Због трошкова перформанси и меморије у *Python*-у, често је неопходно користити пакете високих перформанси са нативним имплементацијама (нпр. `numpy`, `SciKit-Learn`, `TensorFlow`) за оптимизацију кода. Ово значи да чист *Python* код можда није довољан за апликације које су критичне за перформансе.

## 2.4 *Rust*

*Rust* је модерни системски програмски језик који је развила *Mozilla Research*, а његов иницијални креатор је Грејдон Хор. Развој *Rust*-а је почео 2010. године, са циљем да обезбеди високу безбедност и перформансе у системском програмирању. *Rust* је постао званично стабилан са издавањем верзије 1.0 у мају 2015. године [15].

*Rust* је познат по својој комбинацији брзине и безбедности. Он подржава парадигму власништва (енгл. *ownership*)[16], што омогућава програмерима да пишу сигуран и високо ефикасан код. Језик је истовремено статички типизирани (енгл. *statically-typed*)[17] и подржава више парадигми програмирања, укључујући објектно-оријентисано и функционално програмирање [16].

### 2.4.1 Зашто користити *Rust*?

Постоји неколико разлога зашто користити Rust [17]:

- **Безбедност:** *Rust* је дизајниран са фокусом на сигурност меморије. Његов систем власништва спречава уобичајене грешке као што су употреба после ослобађања меморије и конкурентне трке података. Ове грешке су најчешће узрок безбедносних пропуста у другим програмским језицима.

[14] Процес праћења и анализе перформанси програма.

[15] Библиотеке написане у другим језицима као што су C или C++ које могу побољшати перформансе.

[16] Власништво је концепт који обезбеђује безбедно управљање меморијом без потребе за аутоматским сакупљањем отпада, што елиминише читаву класу грешака везаних за меморију као што су сегментационе грешке и цурење меморије.

[17] Језик у којем се типови свих израза одређују у време компајлирања.

- **Перформансе:** *Rust* пружа перформансе на нивоу са *C* и *C++* захваљујући концепту апстракције са нултим трошковима, што значи да омогућава писање висококвалитетног кода без компромиса у брзини.

- **Екосистем:** *Rust* има богат екосистем алата и библиотека, укључујући `Cargo`, који управља пројектима, грађењем, тестирањем и дистрибуцијом пакета. Уз `Cargo`, *Rust* заједница развија и подржава велики број корисних библиотека за различите домене.

- **Конкурентност (енгл. Concurrency)**[18]**:** *Rust* омогућава писање безбедног конкурентног кода кроз свој систем за позајмљивање (енгл. *borrowing*)[19] и правила за дељење података.

- **Ефикасност:** *Rust* избегава потребу за сакупљањем смећа захваљујући статичкој анализи времена живота објеката, што побољшава ефикасност рада програма.

- **Популарност:** *Rust* је стално на врху листе најомиљенијих програмских језика према *Stack Overflow* истраживању, што указује на задовољство програмера његовом употребом.

### 2.4.2 Мане *Rust*-а

*Rust* има неколико недостатака [19]:

- **Недостатак зрелости:** *Rust* је релативно млад језик. Ово значи да језик можда није довољно тестиран у стварним сценаријима и да још увек пролази кроз значајне промене.

- **Мања понуда програмера:** За разлику од *C* и *C++*, за које је лакше пронаћи и запослити програмере, постоји мања понуда *Rust* програмера на тржишту, што може отежати проналажење талентованих инжењера.

[18] Конкурентност се односи на способност извршавања више задатака или операција истовремено унутар истог програма или система [18].

[19] Позајмљивање је концепт у *Rust*-у који омогућава променљивим да позајмљују референце на податке без промене власништва над тим подацима.

- **Мањи број алата и библиотека:** Иако *Rust* има значајан екосистем алата и библиотека, још увек није на нивоу старијих језика као што су *C* и *Python*. Ово може ограничити брзину развоја и доступност решења за специфичне проблеме.

- **Комплексност система власништва и позајмљивања:** Систем власништва и позајмљивања у *Rust*-у, иако обезбеђује сигурност, представља нове концепте што може захтевати дуже време учења.

- **Недостатак ручног управљања меморијом:** Иако *Rust* избегава многе грешке које се јављају због управљања меморијом, у ретким случајевима када је ручно управљање меморијом потребно, програмери се морају ослонити на небезбедне блокове (енгл. *unsafe blocks*)[20], што може бити изазовно и ризично.

## 2.5 Зашто транспајловати из *Python*-а у *Rust*?

На основу претходно разматраних предности и мане *Python*-а и *Rust*-а, као и њихових карактеристика, могу се идентификовати разлози зашто би транспајловање кода из *Python*-а у *Rust* могло бити корисно [12, 14, 16, 17]. Ево неколико разлога који се темеље на анализираним информацијама:

- **Побољшање перформанси:** *Python*, иако пружа високу продуктивност у фазама развоја, често се суочава са проблемима перформанси због своје динамичке природе и вишег нивоа апстракције. Прелазак на *Rust* може значајно побољшати брзину извршавања апликација захваљујући статичком типизирању и контроли ресурса без сакупљања смећа.

- **Побољшање безбедности кода:** Систем власништва у *Rust*-у омогућава сигурно управљање меморијом, спречавајући многе врсте грешака које су присутне у *Python*-у, који користи аутоматско сакупљање смећа. Транспајловањем кода из *Python*-а у *Rust* може се побољшати безбедност апликације.

- **Контрола над ресурсима:** *Rust* омогућава пуну контролу над ресурсима, што је посебно корисно у окружењима са ограниченим ресурсима. Користећи *Rust* за

[20]Небезбедни блокови у *Rust*-у омогућавају програмерима да деактивирају неке од безбедносних провера *Rust*-а, као што су провере позајмљивања. То омогућава директно управљање меморијом, позивање небезбедних функција или метода, приступ и измену глобалних променљивих, имплементацију небезбедних особина и приступ пољима у унијама.

део система, може се побољшати ефикасност у односу на *Python*-ову употребу ресурса.

# 3 Преглед литературе

## 3.1 *Pyrs*

*Pyrs* је алат за транспајлацију који омогућава претварање *Python* кода у *Rust*. Развијен од стране Јулијана Кончунаса, *Pyrs* пружа решење за оне који желе да мигрирају своје *Python* пројекте у *Rust*, користећи предности *Rust*-ове безбедности и перформанси [20].

У овом делу рада ће се анализирати најновија верзија *PyRs*-а која је интегрисана у *py2many* пројекат [21].

Основни приступ *PyRs*-а подразумева конвертовање *Python*-ових конструкција високог нивоа у њихове еквиваленте у програмском језику *Rust*. *PyRs* је базиран на алату *Clike* [21], који је дизајниран за превођење и трансформацију кода између различитих програмских језика, што омогућава извршавање сложених конверзија са једне синтаксе на другу. Овај процес почиње парсирањем *Python* кода како би се генерисало апстрактно синтаксно стабло (енгл. *abstract syntax tree*) (*AST*). Апстрактно синтаксно стабло служи као посредничка репрезентација изворног кода, обухватајући синтаксну структуру *Python* програма на начин који је независан од оригиналног језика. *AST* комбинује елементе лексичке и синтаксне анализе, обезбеђујући структуру која омогућава даљу трансформацију и генерисање кода.

Када се створи *AST*, *PyRs* наставља са серијом трансформација како би прилагодио *Python*-ове конструкције високог нивоа у синтаксу и семантику *Rust*-а. Ово укључује мапирање *Python*-ових специфичних функција, као што су динамичко типизирање и одређене уграђене функције, у *Rust*-ову статички типизирану и перформансно оријентисану парадигму. Трансформације осигуравају да се динамично понашање *Python*-а или очува кроз *Rust*-ове могућности или прилагоди како би одговарало ограничењима *Rust*-а.

Завршна фаза процеса транспилације укључује генерисање *Rust* кода из трансформисаног *AST*-а. Генерисани код настоји да задржи оригиналну функционалност и логику *Python* програма, истовремено користећи предности *Rust*-а. *PyRs* такође обрађује специфичне оптимизације и идиоматске шаблоне кода у *Rust*-у, осигуравајући да резултујући *Rust* код буде не само функционалан, већ и ефикасан и лак за одржавање.

## 3.2 Преглед сродних истраживања и радова

У раду *„Transpiling Python to Rust for Optimized Performance”* истраживане су две различите употребе како би се испитала тежина и резултати транспајлације и њена изводљивост за уграђене имплементације [22].

Први случај је једноставан *Black–Scholes* модел за одређивање цене опција на финансијским тржиштима. Аутори су имплементирали модел у *Python*-у користећи `numpy` на основу онлајн извора и додали тестове за валидацију. Транспајлација је обављена аутоматским конвертовањем синтаксе и ручним мапирањем библиотека.

Други случај употребе је алгоритам за планирање кретања мобилних манипулатора у роботици. Овај алгоритам омогућава роботу да пронађе пут кроз простор уз поштовање кинематичких ограничења и избегавање судара. Оригинална имплементација користи *BLAS (Basic Linear Algebra Subprograms)* [21] спецификацију за оптимизацију преко `numpy`-а . Транспајлирана имплементација мапира `numpy` функције на *Rust* библиотеку `ndarray`.

Метода транспајлације идентификовала је бројне конверзије које се могу аутоматизовати, иако су неке мануелне интервенције биле неопходне. Разлике у синтакси између *Python*-а и *Rust*-а су углавном биле аутоматски прилагодљиве, са неким изузецима који су захтевали мануелна прилагођавања. Семантика власништва и опсег променљивих у *Rust*-у захтевали су пажљиво руковање током транспајлације.

Транспајловани изворни код у *Rust*-у се придржава семантике и система типова *Rust*-а, омогућавајући високоперформантне имплементације које одговарају нативним моделима извршавања циљаних платформи. Строги систем типова *Rust*-а и правила управљања меморијом омогућавају оптимизације и гарантују безбедност у паралелном извршавању.

Све у свему, процес транспајлације је показао да се *Python* код може ефикасно транспајловати у *Rust*, постижући побољшања у перформансама и одржавајући функционалну еквивалентност на различитим платформама.

[21] BLAS је спецификација за скуп основних рутина за линеарну алгебру које су критичне за многе рачунске операције у научном рачунарству. Ове рутине се користе за основне операције као што су дот производ и производ матрице и вектора, што омогућава унапређење перформанси и преносивост софтвера [23].

# 4 Студија случаја

## 4.1 Постављање пројекта

У овом делу, детаљно је описано како поставити *py2many pyrs* транспајлер за транспилацију *Python* кода у *Rust*. Ово су кораци:

**Предуслови** Потребне су следеће верзије:

- *Python 3.8* или новији
- *Rust 1.50* или новији
- `pip` за управљање *Python* пакетима

**Инсталација:**

**Инсталација *Python* библиотека:**

```
pip install py2many-- user  #         $HOME/.local
```

Или, системски:

```
sudo pip install py2many
```

**Инсталација *Rust* алата:**

```
curl --proto '=https' --tlsv1.2  -sSf https://sh.rustup.
   rs | sh
```

**Конфигурација:** Након инсталације, потребно је додати *py2many* скрипту у `PATH`. На пример, потребно је додати следећи ред у `.bashrc` или `.zshrc` фајл:

```
export PATH=$HOME/.local/bin:$PATH
```

**Покретање транспајлера:**

```
py2many --rust=1 path/to/your_script.py
```

Пример за тестни фајл `fib.py`:

```
py2many --rust=1 tests/cases/fib.py
```

**Компилација резултујућег *Rust* кода:**

```
rustup run nightly cargo build --manifest -path tests/
   expected/fib.rs
```

**Форматирање:** Многи транспајлери ослањају се на језичке специфичне форматере за парсирање и форматирање излаза. За *Rust*, то је `rustfmt`. Овако се инсталира `rustfmt`:

```
rustup component add rustfmt
```

## 4.2 Линеарна регресија

Линеарна регресија је основна статистичка метода која се користи за моделовање односа између једне зависне варијабле и једне или више независних варијабли. Циљ линеарне регресије је да пронађе најбољу праву линију која минимизира разлику између предвиђених и стварних вредности зависне варијабле. Ова метода омогућава предвиђање вредности зависне варијабле на основу вредности независних варијабли, као и процену јачине и природе односа између њих [24].

У овом примеру из студије се проверава како транспајлер ради на најлакшем могућем примеру, без коришћења екстерних библиотека и са коришћењем `typing` библиотеке која омогућава статичко типизирање за *Python* да би се код лакше транспајлирао у *Rust*.

### 4.2.1 *Python* implementacija

*Python* имплементација линеарне регресије обухвата цео процес обуке и предвиђања вредности на основу улазних података. Прво, параметри моделирања као што су тежине и пристрасност иницијализују се на нуле, што омогућава да модел има почетне вредности од којих ће кренути процес учења. Тежине представљају значај сваке карактеристике у подацима, док пристрасност служи као компензациони фактор.

Током процеса обуке, модел користи улазне податке да би израчунао предвиђене вредности. Ове вредности се добијају као збир производа карактеристика и тежина, уз додатак пристрасности. Разлика између предвиђених вредности и стварних вредности представља грешку коју модел настоји да минимизира. Да би се то постигло, израчунавају се градијенти који показују у ком смеру и колико треба прилагодити тежине и пристрасност. Градијенти се рачунају на основу просека разлике између предвиђених и стварних вредности помножене са улазним карактеристикама.

Процес ажурирања параметара укључује смањење тежина и пристрасности за одређени износ који је пропорционалан градијентима и стопи учења. Стопа учења одређује колико брзо или споро модел учи; већа стопа учења значи брже али потенцијално нестабилно учење, док мања стопа учења обезбеђује стабилнији али спорији напредак.

Овај процес обуке понавља се више пута током одређеног броја епоха. Сваком итерацијом кроз епохе, модел постепено побољшава своје параметре како би што боље предвидео вредности на основу тренинг података. По завршетку обуке, оптимизовани параметри модела користе се за предвиђање вредности на новим улазним подацима. Целокупна имплементација обухвата иницијализацију параметара, израчунавање предвиђања, рачунање градијената, ажурирање параметара и на крају, примену обученог модела за предвиђање нових вредности. Овај процес омогућава моделирање односа између улазних карактеристика и циљних вредности.

### 4.2.2 Анализа транспајлираног *Rust* кода

Транспајлирани код линеарне регресије захтевао је неколико значајних измена како би се обезбедила његова исправност и ефикасност. Прво, било је неопходно прилагодити типове променљивих да би били усклађени са стандардима *Rust* језика. Типови променљивих који представљају бројеве карактеристика и узорака су промењени да

користе одговарајуће типове за индексирање у *Rust*-у, што је унапредило стабилност и перформансе кода.

**Транспајлирани код**

```
pub fn initialize_parameters(n_features: i32) {
    let weights = (vec![0.0] * n_features);
    let bias: f64 = 0.0;
    return (weights, bias);
}
```

```
pub fn predict(X: &Vec<Vec<f64>>, weights: &Vec<f64>, bias: f64)
   -> Vec<f64> {
    let mut predictions: List = vec![];
    for i in (0..X.len() as i32) {
        let prediction: f64 = (((0..X[0 as usize].len() as i32)
            .map(|j| ((X[i as usize][j] as f64) * weights[j as
               usize]))
            .collect::<Vec<_>>()
            .iter()
            .sum() as f64)
            + bias);
        predictions.push(prediction);
    }
    return predictions as Vec<f64>;
}
```

**Ревидирани код**

```
pub fn initialize_parameters(n_features: usize) -> (Vec<f64>, f64)
    {
    let weights = vec![0.0; n_features];
    let bias: f64 = 0.0;
    (weights, bias)
}
```

```
pub fn predict(X: &Vec<Vec<f64>>, weights: &Vec<f64>, bias: f64)
   -> Vec<f64> {
    let mut predictions: Vec<f64> = Vec::new();
    for i in 0..X.len() {
        let prediction: f64 = X[i].iter().zip(weights).map(|(x, w)
           | x * w).sum::<f64>() + bias;
        predictions.push(prediction);
    }
    predictions
}
```

Додатно, процес иницијализације вектора тежина и градијената је унапређен да би био у складу са уобичајеним праксама у *Rust*-у. Ово је укључивало коришћење стандардне синтаксе за креирање вектора са одређеним вредностима, што обезбеђује правилно и ефикасно управљање меморијом.

Код за израчунавање предикција је оптимизован да би био читљивији и ефикаснији. Коришћене су методе које омогућавају лакше и брже обрађивање података, чиме је побољшана укупна читљивост и перформансе програма. Такође, параметри су редефинисани тако да омогућавају директне измене без потребе за додатним корацима.

Процес ажурирања параметара такође је поједностављен. Уклоњене су непотребне операције које су раније компликовале код, чиме је постигнута боља читљивост. Поред тога, излазни резултати су кориговани да би били правилно приказани, користећи синтаксу за форматисан приказ у *Rust*-у, што обезбеђује тачне и јасне информације о резултатима програма.

На крају, уместо коришћења сложених типова за број итерација и параметара, изабрани су једноставнији и прикладнији типови. Ове промене су омогућиле да код буде ефикаснији и лакши за одржавање.

## 4.3 Логистичка регресија

Логистичка регресија је статистички метод који се користи за моделовање вероватноће одређеног догађаја. Овај модел је посебно користан када је зависна променљива бинарна, што значи да може имати само две могуће вредности, као што су „0“

и „1“ [25].

У овом примеру, користе се подаци из скупа података *„Diabetics prediction using logistic regression”*, који је доступан на *Kaggle*-у [26].

Ови подаци садрже информације о здравственом стању пацијената у вези са дијабетесом. У табели 4.1 је приказан исечак скупа података који се користи:

| Pregnancies | Glucose | BloodPressure | SkinThickness | Insulin | BMI | DiabetesPedigreeFunction | Age | Outcome |
|---|---|---|---|---|---|---|---|---|
| 6 | 148 | 72 | 35 | 0 | 33.6 | 0.627 | 50 | 1 |
| 1 | 85 | 66 | 29 | 0 | 26.6 | 0.351 | 31 | 0 |
| 8 | 183 | 64 | 0 | 0 | 23.3 | 0.672 | 32 | 1 |

Tabela 4.1: Исечак скупа података о дијабетесу.

Користи се логистичка регресија да анализира утицај различитих фактора на вероватноћу развоја дијабетеса. У овом примеру се не користи `typing` из *Python*-а као у претходном примеру, већ се транспајлира са динамичким типовима ради провере рада транспајлера у таквом окружењу. Такође, користи се `numpy` библиотека за рачунање и тестирање како транспајлер рукује са библиотекама.

### 4.3.1 *Python* implementacija

Python имплементација логистичке регресије почиње учитавањем података из *CSV* (енгл. *comma-seperated values*)[22] датотеке и њиховом конверзијом у бројеве са покретном запетом. Овај корак је неопходан да би се подаци могли даље обрађивати. Затим следи стандардизација карактеристика, где се за сваку карактеристику уклања просек и дели са стандардном девијацијом, што осигурава да све карактеристике буду у сличном распону вредности.

Подаци се потом деле на тренинг и тест сетове. У следећем кораку, примењује се сигмоидна функција, која трансформише линеарне комбинације карактеристика у вредности између 0 и 1, омогућавајући бинарну класификацију. Цена модела се израчунава на основу тренутних параметара.

Градијенти цене се рачунају како би се одредио правац и величина промена у параметрима модела. Ти градијенти се користе у процесу градијентног спуста, који оптимизује параметре модела и постепено побољшава његову прецизност. Када се параметри оптимизују, модел је спреман за прављење предвиђања на основу нових карактеристика.

[22]CSV је текстуални формат за представљање табеларних података где су вредности одвојене зарезима.

На крају, целокупан процес обуке и тестирања модела се изводи у главној функцији. Ова функција координира све кораке, од учитавања и обраде података до оптимизације параметара и процене тачности модела на тренинг и тест сетовима. На крају, резултати се приказују, пружајући јасан увид у успех модела у предвиђањима.

### 4.3.2 Анализа транспајлираног *Rust* кода

Промене у транспајлираном коду логистичке регресије из *Python* у *Rust* биле су неопходне да би се осигурала исправна функционалност и побољшале перформансе. Основне измене су укључивале прелазак на одговарајуће библиотеке и прилагођавање синтаксе за рад са подацима. Ове измене су укратко описане у наставку:

- **Замена библиотеке `numpy` са `ndarray`**: У *Python*-у, библиотека `numpy` пружа широк спектар функција за рад са низовима и математичке операције. У *Rust*-у, библиотека `ndarray` нуди сличне функционалности, али је прилагођена синтакси и типовима података у *Rust* окружењу. Ово укључује методе за израчунавање статистичких показатеља, као што су просек и стандардна девијација, као и методе за рад са матрицама и векторима.

- **Корекција метода за конверзију типова и математичке операције**: У *Python*-у су коришћене методе које нису компатибилне са *Rust*-ом. На пример, методе из `numpy` као што су `np.mean` и `np.std` замењене су одговарајућим методама из библиотеке `ndarray`, као што су `mean_axis` и `std_axis`. Ово укључује и исправне методе за рад са матрицама и векторима у *Rust*-у, као што су `slice` и `s!`, што омогућава правилно раздвајање података на обуке и тестове.

- **Прилагођавање математичких функција**: Методи као што су `np.clip` и `np.log`, који нису компатибилни са *Rust*-ом, замењени су одговарајућим методама из библиотеке `ndarray`. На пример, методе `mapv` и `clamp` су коришћене за рад са подацима типа `Array1<f64>` и `Array2<f64>`, што осигурава исправност математичких операција.

- **Прилагођавање типова података и функција**: Методе попут `np.zeros` и `np.hstack` које нису доступне у *Rust*-у, замењене су са методама библиотеке `ndarray` као што су `Array::zeros` и `concatenate`. Ово осигурава да се гради-

јентни спуст и остале операције извршавају правилно користећи типове података у *Rust*-у, као што су `Array1<f64>` и `Array2<f64>`.

**Транспајлирани код**

```
pub fn sigmoid<T0>(z: T0) -> f64 {
    z = np.clip(z, -500, 500);
    return ((1 as f64) / ((1 + (np.exp(-(z)) as i32)) as f64)
       ) as f64;
}
```

**Ревидирани код**

```
pub fn sigmoid(z: Array1<f64>) -> Array1<f64> {
    let z = z.mapv(|a| a.clamp(-500.0, 500.0));
    (1.0 / (1.0 + (-z).mapv(f64::exp)))
}
```

- **Коришћење модула `BufRead` и `BufReader`**: Уместо метода из библиотеке `pylib`, који нису оптимални за *Rust* окружење, употребљени су модули `BufRead` и `BufReader`. Ови модули пружају напредне могућности за читање великих количина података из фајлова, што побољшава ефикасност и перформансе програма у *Rust*-у.

**Транспајлирани код**

```
pub fn load_data<T0, RT>(file_path: T0) -> RT {
    ({let file = OpenOptions::new().read(true).open(file_path
       )?;
        let lines = file.readlines();});
    let header = lines[0].strip().split(",");
    let mut data = lines[1..]
        .iter()
        .map(|line| line.strip().split(","))
        .collect::<Vec<_>>();
    data = np.array(data, str);
    return data;
}
```

**Ревидирани код**

```
pub fn load_data(file_path: &str) -> Result<Array2<f64>> {
    let file = OpenOptions::new().read(true).open(file_path)
       ?;
    let reader = BufReader::new(file);
    let lines: Vec<_> = reader.lines().collect::<Result<_, _
       >>()?;
    let header = lines[0].trim().split(',').collect::<Vec<_
       >>();
    let data: Array2<f64> = Array::from_shape_vec(
        (lines.len() - 1, header.len()),
        lines[1..]
            .iter()
            .flat_map(|line| {
                line.trim()
                    .split(',')
                    .map(|entry| entry.parse().unwrap())
                    .collect::<Vec<_>>()
            })
            .collect(),
    )?;
    Ok(data)
}
```

## 4.4 *KNN*

*K-Nearest Neighbours* (*KNN*) је не-параметарска класификациона метода која је једноставна али ефикасна у многим случајевима. Да би се класификовао податкак, добијају се његових k најближих суседа, који чине његову околину. Одлучивање о класификацији се обично врши гласањем већине међу подацима у околини, уз или без разматрања тежина базираних на растојању. Међутим, за примену *KNN* методе потребно је изабрати одговарајућу вредност за k, и успех класификације у великој мери зависи од те вредности. У одређеном смислу, *KNN* метода је пристрасна према k. Постоји много начина да се изабере вредност k, али један једноставан метод је да

се алгоритам изврши више пута са различитим вредностима k и изабере она која даје најбоље резултате [27].

У овом примеру, користе се подаци из скупа података *„KNN Data "*, који је доступан на *Kaggle*-у [28].

Ови подаци садрже податке за тестирање класификације коришћењем *KNN* алгоритма. У табели 4.2 је приказан исечак скупа података који се користи:

| XVPM | GWYH | TRAT | TLLZ | IGGA | HYKR | EDFS | GUUB | MGJM | JHZC | TARGET CLASS |
|---|---|---|---|---|---|---|---|---|---|---|
| 1636.67 | 817.99 | 2566.00 | 358.35 | 550.42 | 1618.87 | 2147.64 | 330.73 | 1494.88 | 845.14 | 0 |
| 1013.40 | 577.59 | 2644.14 | 280.43 | 1161.87 | 2084.11 | 853.40 | 447.16 | 1193.03 | 861.08 | 1 |
| 1300.04 | 820.52 | 2025.85 | 525.56 | 922.21 | 2552.36 | 818.68 | 845.49 | 1968.37 | 1647.19 | 1 |
| 1059.35 | 1066.87 | 612.00 | 480.83 | 419.47 | 685.67 | 852.87 | 341.66 | 1154.39 | 1450.94 | 0 |

Tabela 4.2: Исечак скупа података за тестиранје *KNN* алгоритма.

У овом делу испитује се разлике у перформансама између *Python*-а і *Rust*-а са непараметарским класификационим методама као и могућност транспајлера да транспајлује структуре попут класа. Као и са логистичком регресијом, транспајлира се са динамичким типовима и користи се библиотека `numpy` за математичке операције и учитавање података из фајлова.

### 4.4.1 *Python* implementacija

*Python* имплементација *K-Nearest Neighbors* (*KNN*) алгоритма почиње креирањем и конфигурисањем класе `KNN` која укључује дефиницију параметра `k`, који представља број најближих суседа које алгоритам разматра при доношењу одлуке. У следећем кораку, тренинг подаци се учитавају и складиште у класи како би били доступни током фазе предвиђања.

За обављање предвиђања, користи се метод који израчунава растојања између нових примера и свих примера у тренинг скупу. Овај метод се ослања на еуклидско растојање (енгл. *euclidean distance*) које омогућава идентификацију најближих. Користећи библиотеку као што је `numpy`, растојања се брзо израчунавају у матрицама.

Након што су растојања израчуната, алгоритам идентификује `k` најближих суседа за сваки тестни пример. Ознаке ових суседа се анализирају, и најчешће се бира као крајњи резултат предвиђања. Ова процедура укључује коришћење `Counter` из библиотеке `collections` за бројање појављивања сваке ознаке и одређивање најчешће појављујуће ознаке.

Подаци се учитавају из *CSV* датотеке користећи једноставну функцију за читање и парсирање, што укључује конверзију текста у бројеве и раздвајање карактеристика од ознака.

Када се модел обучи на тренинг подацима, предвиђања се праве на основу тестних података. Тачност модела се процењује упоређивањем предвиђених ознака са стварним ознакама тестног скупа. Резултати се затим анализирају и приказују, што омогућава оцену успеха модела у класификацији и укупне ефикасности алгоритма.

Целокупан процес укључује учитавање података, обуку модела, предвиђање и процену тачности, уз коришћење библиотека као што су `numpy` за математичке операције и `collections` за анализу ознака.

### 4.4.2 Анализа транспајлираног *Rust* кода

Транспилација кода *KNN* алгоритма из *Python* у *Rust* укључивала је неке специфичне проблеме, посебно у вези са класом и форматирањем кода. Иако су неки од проблема били слични онима које смо већ сусрели у логистичкој регресији, као што су замена библиотека и корекција математичких операција, постојали су и нови изазови специфични за *KNN*. Основне измене и решења су описани у наставку:

- **Проблеми са типовима у класи**: У *Rust*-у, типови морају бити јасно дефинисани, што је представљало изазов за класу `KNN`. У оригиналном *Python* коду, коришћени су `const` чланови без конкретних типова, што је довело до грешака у *Rust*-у. Исправно решење је било да се дефинишу конкретни типови за све чланове класе, како би се избегле компилационе грешке.

  **Транспајлирани код**

```
pub fn _predict<T0, RT>(&self, x: T0) -> RT {
    pub const distances = self._compute_distances(x);
    pub const k_indices = np.argsort(distances)[..self.k
       ];
    pub const k_nearest_labels = self.y_train[k_indices];
    pub const label_counts = Counter(k_nearest_labels);
    return label_counts.most_common(1)[0][0];
}
```

**Ревидирани код**

```
fn _predict(&self, x: Array1<f64>) -> f64 {
    let distances = self._compute_distances(&x);
    let mut k_indices: Vec<usize> = (0..distances.len()).
       collect();
    k_indices.sort_by(|&a, &b| distances[a].partial_cmp(&
       distances[b]).unwrap());
    k_indices.truncate(self.k);

    let mut label_counts = HashMap::new();
    for &index in &k_indices {
        let label = self.y_train[index].to_string();
        *label_counts.entry(label).or_insert(0) += 1;
    }

    let (most_common_label, _) = label_counts.into_iter()
       .max_by_key(|&(_, count)| count).unwrap();
    most_common_label.parse::<f64>().unwrap()
}
```

- **Проблеми са методом `__init__`**: Метод `__init__` у *Python*-у служи као конструктор класе и користи се за иницијализацију атрибута када се нови објекат створи. Међутим, у *Rust*-у, концепт конструктора је другачији. Уместо метода `__init__`, *Rust* користи функције унутар `impl` блокова за иницијализацију структура, као што је `new()`. Ово значи да је било потребно преуредити функцију `__init__` и адаптирати је у складу са *Rust* синтаксом и структуралним принципима.

**Транспајлирани код**

```
pub fn __init__<T0>(&self, k: T0) {
    self.k = k;
}
```

**Ревидирани код**

```
pub fn new(k: usize) -> Self {
    KNN {
        k,
        X_train: Array2::zeros((0, 0)),
        y_train: Array1::zeros(0),
    }
}
```

- **Форматирање кода**: Поред техничких проблема са типовима, форматирање кода представљало је значајан изазов. Првобитно транспилирани код није био правилно форматиран у *Rust*-у, што је захтевало ручну интервенцију. Ово укључивало корекцију индентирања, размакâ и осталих елемената синтаксе како би код био у складу са стандардима *Rust*-а. Форматер кода није увек успешно решавао проблеме у вези са комплетним синтаксним структурама, што је захтевало детаљну анализу и ручно усавршавање.

- **Замена библиотеке `numpy` са `ndarray`**: Као што је било са логистичком регресијом, и у KNN коду је било потребно заменити функције из `numpy` са одговарајућим методама из `ndarray`. Ово је укључивало прилагођавање метода за израчунавање статистичких показатеља и рад са низовима и матрицама.

- **Коришћење модула `BufRead` и `BufReader`**: Слично као и у логистичкој регресији, у KNN коду је било потребно користити модули `BufRead` и `BufReader` ради побољшања перформанси у управљању великим количинама података.

- **Корекција метода за конверзију типова и математичке операције**: Проблеми са методама и математичким операцијама из `numpy` били су слични онима у логистичкој регресији, укључујући потребу за заменом методе и типова са одговарајућим у `ndarray`.

## 4.5 Наивни бајес

Наивни Бајесов (енгл. *Naive Bayes*) алгоритам је једноставан алгоритам учења који користи Бајесово правило заједно са снажним претпоставком да су атрибути условно независни унутар класе. Иако се ова претпоставка независности често крши у пракси, наивни Бајесов алгоритам и даље често пружа конкурентну тачност класификације. Уз његову рачунарску ефикасност и многе друге пожељне карактеристике, наивни Бајесов алгоритам је широко примењив у пракси.

У овом примеру, користе се исти подаци као и за *KNN* алгоритам, што је сет података из *„KNN Data”* доступан на *Kaggle*-у [28].

Циљ овог примера је да се фокусира на перформансе транспајлираног кода, а не на оцену самог транспајлера. Као и у случају *KNN*, транспајлира се код са динамичким типовима, користећи библиотеку `ndarray` за математичке операције и учитавање података из фајлова.

### 4.5.1 *Python* имплементација

Python имплементација алгоритма наивног бајеса почиње учитавањем података и припремом за обуку модела. Ова имплементација је прилагођена бинарној класификацији, што значи да алгоритам разматра две класе у анализи.

У фази обуке, алгоритам израчунава статистичке параметре за сваку од две класе. То укључује просеке и варијансе карактеристика унутар сваке класе. Ови параметри се затим користе за процену вероватноће нових података.

Током фазе предвиђања, алгоритам користи израчунате статистике да би одредио вероватноћу припадности нових података свакој класи. Пошто се наивни Бајесов модел ослања на хипотезу о независности карактеристика, свака карактеристика се обрађује независно, што омогућава брзо израчунавање вероватноћа. Коначна одлука о класификацији се доноси на основу највише вероватноће.

Подаци се учитавају из *CSV* датотеке, обрађују се за обуку и предвиђање. Након што су предвиђања завршена, тачност модела се процењује упоређивањем предвиђених ознака са стварним ознакама у тестном скупу.

### 4.5.2 Анализа транспајлираног *Rust* кода

Током иницијалне транспилације, појавио се проблем због непостојеће имплементације у алатки `py2many`, конкретно због недостатка подршке за `dict comprehension`. Потребно је било модификовати иницијалну имплементацију да не садржи `dict comprehension` иначе транспајлер не функционише.

Транспилација кода *Naive Bayes* алгоритма из *Python* у *Rust* нису довеле до нових проблема у поређењу са претходним случајем логистичке регресије. Основне измене су биле сличне: замена библиотека, као што је `numpy` са `ndarray`, и корекција метода за рад са подацима. Примењени су исти модули као што су `BufRead` и `BufReader` за оптимизацију читања података и адаптирали математичке функције у складу са синтаксом *Rust*-а. Коришћени су одговарајући методи библиотеке `ndarray`, што је осигурало исправност обраде и израчунавања. Такође било је потешкоћа са типовима јер су класе биле гледане као `i32` док је `ndarray` радио са `f64` vrednostima.

## 4.6 Метода потпорних вектора (*SVM*)

Метода потпорних вектора (енгл. *Support Vector Machines (SVM)*) је техника машинског учења која се користи за класификацију и регресију. Основна идеја *SVM*-а је да пронађе хиперраван која најбоље раздваја податке у различите класе у случају класификације, или да пронађе функцију која најбоље предвиђа вредности у случају регресије [30][31].

Код *SVM*-а, хиперраван која максимизује маргину (удаљеност) између најближих примера из различитих класа се изабира као оптимална. Овај принцип максималне маргине помаже у побољшању генерализације модела на новим подацима.

Циљ овог примера је такође, као и код наивног бајеса да се фокусира на перформансе транспајлираног кода у односу на оригинални код. Као и код логистичке регресије, *KNN*-а и наивног бајеса, транспајлира се са динамичким типовима и користе се библиотеке.

У овој имплементацији користи се *Soft margin SVM* за решавање проблема бинарне класификације, користе се исти подаци као и за *KNN* и за наивни бајес [28]. *Soft margin SVM* је верзија *SVM*-а која дозвољава неке грешке у класификацији ради побољшања способности модела да се носи са шумом у подацима и са подацима који

нису савршено одвојиви. Ова варијанта укључује концепт ”меканих маргина”, где се уводе корекције за податке који су погрешно класификовани или који лежу унутар маргине [30].

### 4.6.1 *Python* имплементација

*Python* имплементација *Support Vector Machine* (SVM) алгоритма са *soft margin* почиње учитавањем података из *CSV* датотеке. Подаци се обрађују тако што се карактеристике и ознаке razdвајају, при чему су ознаке претворене у две класе: `-1` и `1`.

Након учитавања података, извршава се обука модела. Основни принцип обуке *SVM* алгоритма са *soft margin* је да се оптимизују параметри модела тако што се минимизује функција губитка уз примену редукције утицаја грешке. Ова техника омогућава моделу да буде флексибилнији у односу на податке који могу бити неформално распоређени, што је кључно за постизање добрих резултата у стварним сценаријима где подаци могу бити шумни.

Након обуке, модел се користи за прављење предвиђања на основу нових података. Клacifikacija се врши на основу линеарне комбинације карактеристика и оптимизованих параметара модела.

На крају, предвиђене ознаке се упоређују са стварним ознакама ради процене тачности модела.

### 4.6.2 Анализа транспајлираног *Rust* кода

Транспилација *SVM* алгоритма из *Python* у *Rust* није довела до нових проблема у односу на претходне примере са логистичком регресијом, *KNN* и наивним бајесом. Процес је укључивао сличне промене као и раније: замена библиотека, у овом случају `numpy` са `ndarray`, као и прилагођавање метода за рад са подацима. Такође су коришћени исти модули за читање података, као што су `BufRead` и `BufReader`, уз адаптацију математичких функција у складу са синтаксом *Rust*-а.

# 5 Евалуација

У овој секцији биће разматране две кључне аспекте: евалуација процеса транспајлирања и упоредна евалуација перформанси кода написаног у *Python*-у и *Rust*-у.

## 5.1 Евалуација транспајлера

Током процеса транспилације из *Python* у *Rust*, анализирани су различити аспекти рада транспајлера у контексту специфичних примера. Процена ефикасности транспајлера обухватила је неколико различитих случајева употребе.

У случају линеарне регресије са статичким типовима, коришћење `typing` библиотеке у *Python* олакшало је дефинисање типова података и функција у *Rust*, што је било од велике користи за транспајлер. Основни типови и функције су успешно транспајлирани без потребе за додатним прилагођавањем. Међутим, без употребе напредних библиотеки као што је `numpy`, напредне функције као што су статистички прорачуни и рад са матрицама морале су бити имплементиране ручно у *Rust*. Ово је омогућило процену како транспајлер обрађује основне типове и структуре података, али је истовремено открило изазове у управљању позајмљивањем вредности. У неким случајевима, транспајлер није успео да правилно изабере одговарајући тип вредности.

Логистичка регресија, коришћена са динамичким типовима и `numpy` библиотеком у *Python*, донела је изазов транспилацији преласка са `numpy` на `ndarray` библиотеку у *Rust*-у и рад са фајловима. Док `numpy` пружа широк спектар функција за манипулацију низовима и статистичке прорачуне, `ndarray` у *Rust* има другачију синтаксу и приступ. Ово је захтевало прилагођавање метода како би се осигурала тачност и ефикасност у новом окружењу. Такође, управљање логичким вредностима у *Rust*-у разликује се од *Python* приступа, где *Python* аутоматски обрађује логичке вредности, док *Rust* захтева експлицитне методе за трансформацију података. Употреба `ndarray` библиотеке омогућила је коришћење метода као што су `mapv` и `clamp` за математичке операције. Учитавање података из фајлова је такође било успешно транспајлирано коришћењем модула `pylib`, али за ефикаснији рад са фајловима коришћени су `BufRead` и `BufReader` из *Rust*-ове стандардне библиотеке.

Транспилација KNN алгоритма из *Python* у *Rust* представљала је изазов због управљања типовима унутар класа у *Rust*-у. У *Python*-у, типови унутар класа нису строго дефинисани, али у *Rust*-у је потребно јасно дефинисати све типове. Метод `__init__` у *Python*-у, који служи као конструктор класе, замењен је функцијом `new()` у *Rust*-у, што је укључивало прилагођавање синтаксе и начина иницијализације објеката у складу са принципима *Rust*-а. Функције из `numpy` за израчунавање растојања и манипулацију подацима замењене су методама из `ndarray` библиотеке у *Rust*, што је захтевало адаптацију метода како би се осигурала тачност резултата. Поред тога, ручно форматирање кода било је неопходно, јер `rustfmt` није могао самостално решити све проблеме са форматисањем.

Евалуација транспајлера показала је различите аспекте његове ефикасности и ограничења. У свим анализираним примерима, транспајлер је успео да пренесе основну функционалност кода из *Python* у *Rust*, али су се појавили одређени изазови у области управљања типовима, форматисања и коришћења различитих библиотека. У случајевима као што су линеарна и логистичка регресија, транспајлер је показао добру способност у обради основних типова и структура података, али су напредне функције захтевале додатно ручно прилагођавање. Прелазак са `numpy` на `ndarray` у *Rust* показао је потребу за дубљим разумевањем различитих синтакса и приступа у раду са подацима.

Главни изазов у управљању типовима и форматирању кода за *KNN* алгоритам био је захтев за ручним прилагођавањем да би код био у складу са строгим типским системом и принципима форматисања у *Rust*-у.

Укупно гледано, транспајлер је показао способност да успешно преведе код из *Python* у *Rust*, али захтева додатне напоре у погледу прилагођавања типова и ручног уређивања. Даљи рад на усавршавању транспајлера могао би побољшати његову способност у обради напредних функција и поједноставити процес форматисања кода.

## 5.2 Евалуација перформанси

У овој секцији се упоређују перформанси кода написаног у *Python*-у и *Rust*-у, анализа се фокусира на два аспекта: време извршавања и коришћење меморије.

- **Време извршавања**: За мерење времена извршавања кода у *Python*-у користи се `time` модул, који омогућава праћење времена потребног за извршавање ра-

зличитих делова кода. У Rust-у, време извршавања се мери помоћу `Instant` типа из стандардне библиотеке.

- **Коришћење меморије**: У Rust-у, коришћење меморије анализира се коришћењем *Valgrind*-а са опцијом `--tool=massif`. Овај алат пружа детаљан увид у алокацију меморије током извршавања програма [32]. У *Python*-у, `memory_profiler` пакет се користи за мерење меморијских ресурса које сваку функцију у програму користи.

За линеарну регресију, иницијални подаци су били ограничени на мали скуп вредности. Међутим, за тестирање перформанси, број узорака је повећан на 10,000 са 10 карактеристика по узорку. Свака вредност у вектору је 1. Време извршавања није укључивало време потребно за конструисање ових вектора, већ је мерење времена почело након конструкције.

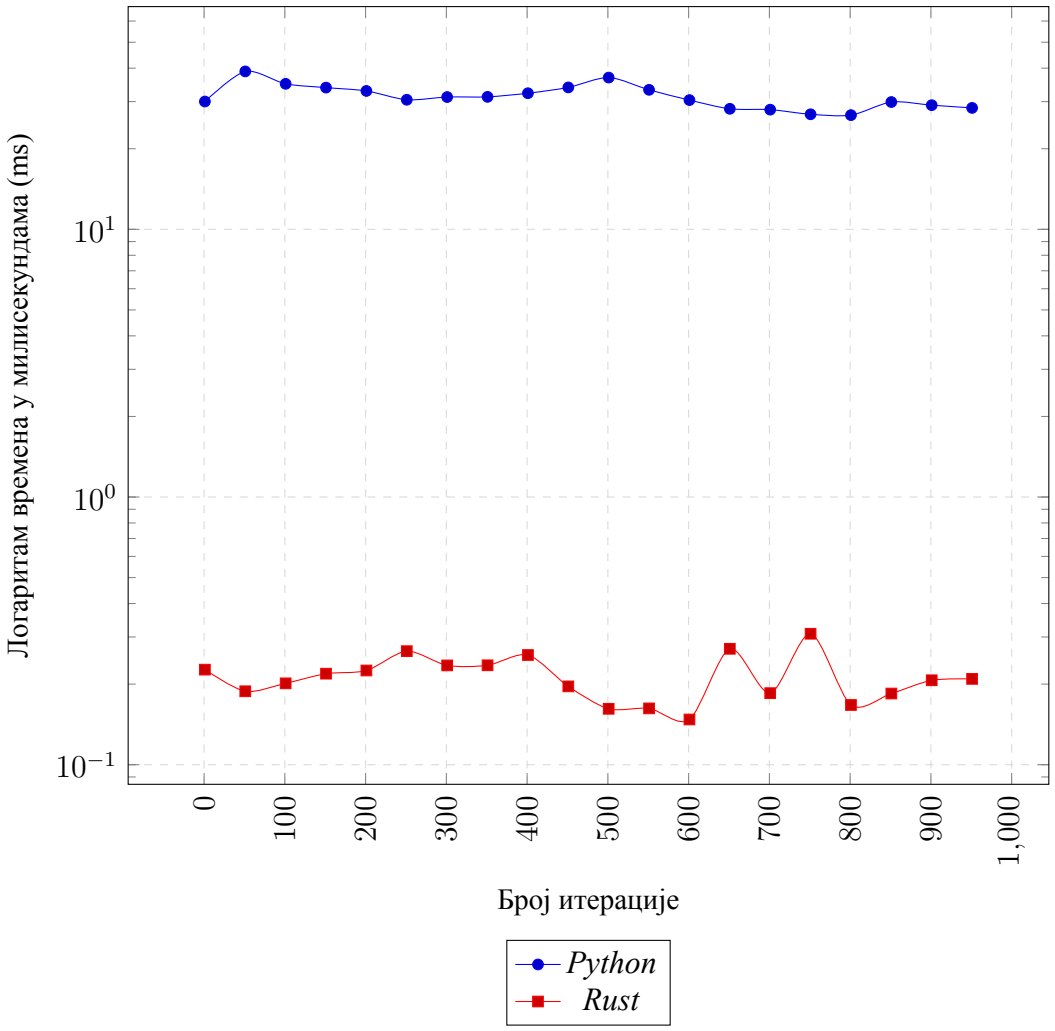
За логистичку регресију, почетни сет података је имао 789 узорака. Овај сет података је умножен како би се повећала величина скупа података за боље тестирање перформанси. Узорци су копирани 6 пута како би се симулирао већи скуп података, што је омогућило детаљнију анализу и поређење између *Python*-а и *Rust*-а.

За *KNN*, наивни бајес и *SVM* алгоритме, коришћен је почетни сет података са 1,000 узорака. Како би се повећала величина скупа података за тестирање перформанси као и код логистичке регресије, узорци су умножени 6 пута.

Тестирање перформанси је спроведено на *Windows* машини за све *Python* и *Rust* програме. Време извршавања и меморија коју користе *Python* програми су измерени у овом окружењу. За анализу коришћења меморије у *Rust* програмима, анализа је изведена у *WSL* (*Windows Subsystem for Linux*) окружењу уз коришћење алата *Valgrind*.

### 5.2.1 Време извршавања

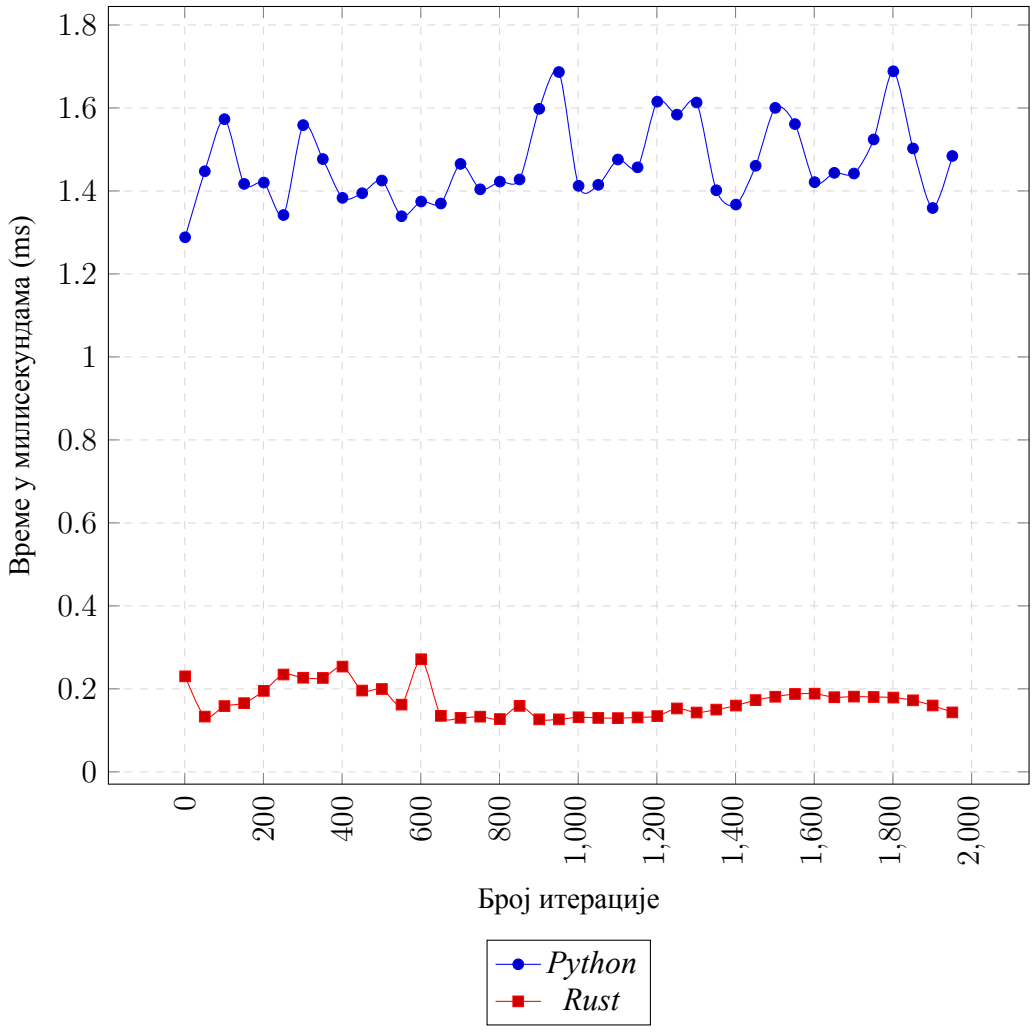
Подаци о времену извршавања за линеарну регресију су сакупљени кроз 1000 итерација за обе имплементације. Уочено је да *Rust* има значајно брже време извршавања у поређењу са *Python*-ом. За прву итерацију, време извршавања у *Python*-у је 30.03 ms, док је у *Rust*-у само 0.23 ms. Ово значајно смањење у времену извршавања показује ефикасност *Rust*-а. Логаритмоване вредности времена извршавања линеарне регресије по итерацији могу се видети на графикону 5.1.




Графикон 5.1: Логаритам времена извршавања линеарне регресије по итерацији.

Током 1000 итерација, времена извршавања у *Python*-у показују варијабилност, са вредностима између 26.71 ms и 38.90 ms. Насупрот томе, времена извршавања у *Rust*-у су конзистентнија и крећу се између 0.15 ms и 0.31 ms, што указује на стабилност *Rust* имплементације.

Подаци о времену извршавања за логистичку регресију су сакупљени кроз 2000 итерација за обе имплементације. Времена извршавања у *Python*-у су опет знатно већа у поређењу са *Rust*-ом. Пример времена извршавања за прву итерацију у *Python*-у је 1.29 ms, док је у *Rust*-у 0.23 ms. Времена извршавања за логистичку регресију по итерацији су приказана на графикону 5.2.

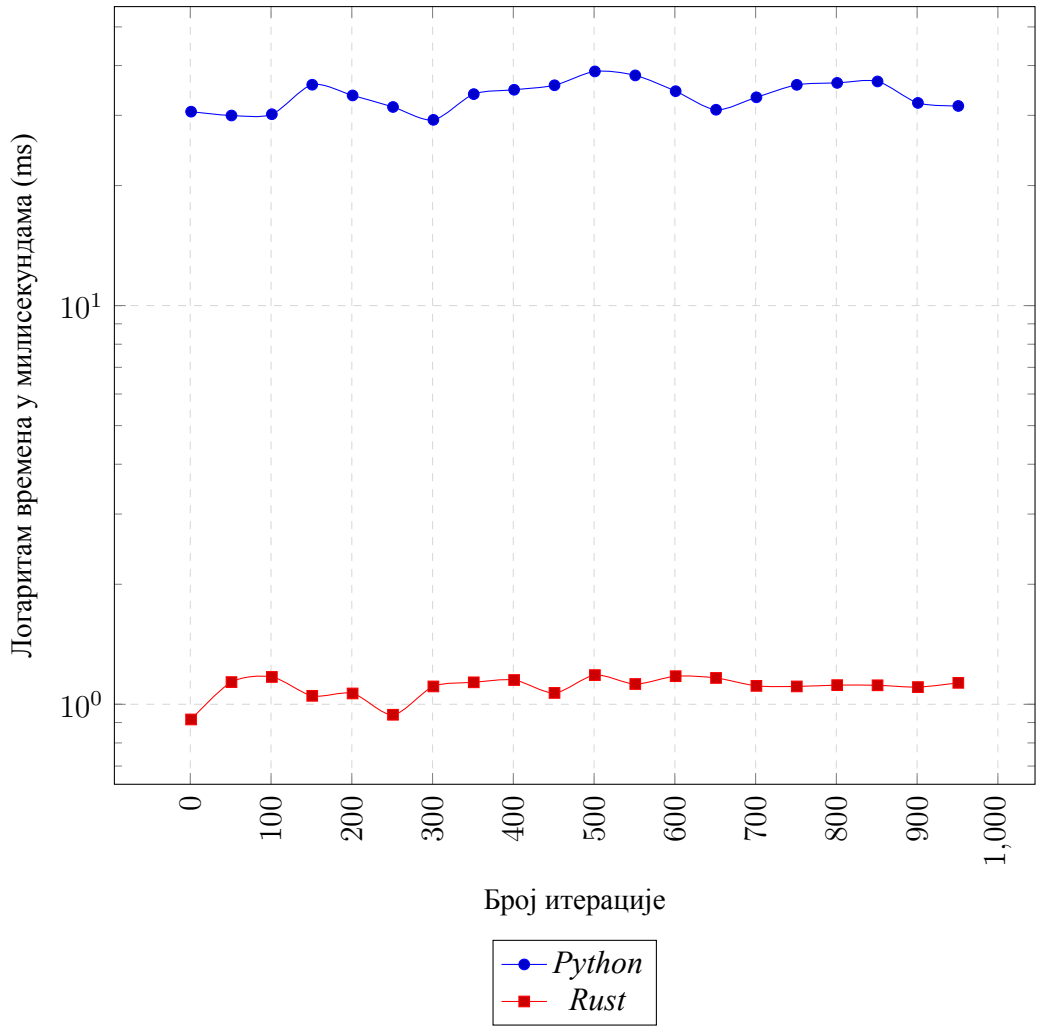



Графикон 5.2: Време извршавања логистичке регресије по итерацији.

За логистичку регресију, времена извршавања у *Python*-у су показала варијацију између 1.29 ms и 1.69 ms, док су времена извршавања у *Rust*-у конзистентно испод 0.2 ms.

Подаци о времену извршавања за *SVM* су сакупљени кроз 1000 итерација за обе имплементације. Логаритмоване вредности времена извршавања *SVM* алгоритма по итерацији могу се видети на графикону 5.3.

У *Rust*-у, времена извршавања по итерацији за *SVM* су варирала од 0.91 ms до 1.18 ms. Ова времена су показала мању варијацију у поређењу са *Python*-ом. Са друге стране, у *Python*-у, времена извршавања су била значајно већа, крећући се од 29.23 ms до 38.67 ms.




Графикон 5.3: Логаритам времена извршавања *SVM*-а по итерацији.

Табела 5.1 упоређује време извршавања за методе из студије у *Python*-у и *Rust*-у. За линеарну регресију, *Rust* је био приближно 179.3 пута бржи од *Python*-а. За логистичку регресију, *Rust* је био приближно 10.1 пута бржи од *Python*-а. За *KNN*, *Rust* је био око 13.8 пута бржи од *Python*-а. Када се упореди време извршавања за Наивни Бајес класификатор, *Rust* је био око 56.8 пута бржи од *Python*-а. За *SVM (Soft Margin)*, *Rust* је био око 30.4 пута бржи од *Python*-а. Напомена је да су времена извршавања у *Python*-у показала значајну варијацију при поновном покретању експеримената на *Windows* оперативном систему, што указује на могуће утицаје које имају фактори попут управљања меморијом и оптимизације извршавања у различитим покретањима.

| Метод | ***Python*** | ***Rust*** |
|---|---|---|
| Линеарна регресија | 32.7982s | 0.1833s |
| Логистичка регресија | 3.5071s | 0.3460s |
| *KNN* | 0.3631s | 0.02627s |
| Наивни Бајес | 0.4084s | 0.00717s |
| *SVM* | 33.8809s | 1.1165s |

Tabela 5.1: Време извршавања за методе из студије у *Python*-у и *Rust*-у.

### 5.2.2 Коришћење меморије

Табела 5.2 показује меморијску употребу за линеарну и логистичку регресију, као и *KNN*, наивни бајес и *SVM* у *Python*-у и *Rust*-у. У свим случајевима, *Rust* показује значајно смањење потрошње меморије у поређењу са *Python*-ом. За линеарну регресију, меморијска употреба у *Rust*-у је смањена за око 58% у поређењу са *Python*-ом. За логистичку регресију, разлика је још израженија, са смањењем од око 81%. За *KNN*, *Rust* је користио око 63% мање меморије у поређењу са *Python*-ом. Када се упореди меморијска употреба за Наивни Бајес класификатор, *Rust* је користио око 78% мање меморије у односу на *Python*-а. За *SVM*, разлика је такође значајна, са смањењем од око 69%.

| Метод | ***Python*** | ***Rust*** |
|---|---|---|
| Линеарна регресија | 3.03 MB | 1.27 MB |
| Логистичка регресија | 3.73 MB | 0.71 MB |
| *KNN* | 3.66 MB | 1.34 MB |
| Наивни Бајес | 3.01 MB | 0.64 MB |
| *SVM* | 2.44 MB | 0.74 MB |

Tabela 5.2: Меморијска употреба метода из студије у *Python*-у и *Rust*-у.

## 6 Закључак

Транспајлирање *Python* кода у *Rust* довело је до побољшања у временским перформансама и коришћењу меморије. *Rust* показује јасне предности у оба аспекта у односу на *Python*.

Време извршавања у *Rust*-у је краће, што је резултат статичке типизације и компајлирања које омогућавају компајлеру да изврши детаљне оптимизације. *Python*, као динамички типизовани језик, не може постићи исте оптимизације због метаподатака који утичу на време извршавања и додатно оптерећују ресурсе. Иако *Python* нуди `numpy` библиотеку за рад са великим количинама података и матричним операцијама, *Rust* има `ndarray` библиотеку која пружа сличне перформансе и функционалности. Библиотека `numpy` је веома ефикасна за рад са бројчано интензивним подацима због своје способности да оптимизује операције на матрицама и великим низовима [33]. Ова ефикасност проистиче из чињенице да `numpy` користи низ функција ниског нивоа написаних у *C* језику, што омогућава брзе операције и минимизира *overhead* који би иначе произашао из *Python*-овог динамичког типизовања и интерпретирања кода. Поред тога, `numpy` библиотека користи контигентне блокове меморије који побољшавају локалност података и омогућавају ефикасније кеширање, што даље побољшава перформансе [33]. С друге стране, `ndarray` у *Rust*-у нуди сличне предности у манипулацији подацима као и `numpy`, али са бољом интеграцијом у *Rust* екосистем.

У погледу коришћења меморије, *Rust* показује значајне предности са значајним смањењем потрошње меморије у поређењу са *Python*-ом. Ово је могуће због статичке типизације и ручног управљања меморијом у *Rust*-у, што елиминише потребу за сакупљачем отпада и смањује *overhead* који *Python* има због динамичког типизовања и управљања меморијом.

Транспајлер се суочава са изазовима који утичу на његове перформансе. Један од главних проблема је управљање позајмљеним вредностима, где транспајлер понекад не успева да тачно одреди типове из *Python*-а и користи шаблоне који нису увек функционални. Ово захтева ручну интервенцију и прилагођавање у *Rust*-у. Такође, потребно је побољшати аутоматизацију процеса транспајлирања, посебно у контексту мапирања из `numpy` библиотеке на `ndarray`. Ова побољшања би могла унапредити ефикасност и тачност транспајлера, чинећи процес преласка кода из *Python*-а у *Rust*

лакшим и ефикаснијим.

Транспајлирање *Python* кода у *Rust* за алгоритме из студије потврђује предности *Rust*-а у погледу брзине и ефикасности, чинећи га одличним избором за интензивне рачунске задатке. Иако постоје области које захтевају додатна побољшања, као што су управљање типовима и аутоматизација процеса, са ручним адаптирањем транспајлираног кода долази до видне предности перформанси у области машинског учења.

У будућности, побољшана верзија транспајлера може играти улогу у подршци развоју великих и сложених неуронских мрежа. Због значајних предности у перформансама које *Rust* нуди, транспајлирање кода из *Python*-а у *Rust* ће постати још привлачније како се транспајлер унапређује да подржи додатне библиотеке и функције. Овакав приступ ће омогућити бржи и ефикаснији тренинг великих неуронских мрежа, чиме ће се побољшати укупна продуктивност и уштеда ресурса у примени напредних техника машинског учења и дубоког учења.

## 7 Литература

[1] D. B. Loveman, “Program Improvement by Source-to-Source Transformation,” J. ACM, vol. 24, no. 1, pp. 121–145, Jan. 1977, doi: 10.1145/321992.322000.

[2] Intel. MCS-86 Assembly Language Converter Operating Instructions for ISIS-II Users. Technical Report. 1978. [Online]. Available: http://www.bitsavers.org/pdf/intel/ISIS_II/9800642A_MCS-86_Assembly_Language_Converter_Operating_Instructions_for_ISIS-II_Users_Mar79.pdf (приступљено 18. јула 2024).

[3] Research, D. XLT86 8080 to 8086 Assembly Language Translator, User’s Guide. Technical Report. 1981. [Online]. Available: http://www.s100computers.com/Software%20Folder/Assembler%20Collection/Digital%20Research%20XLT86%20Manual.pdf (приступљено 18. јула 2024).

[4] M. Hirzel and H. Klaeren, “Code coverage for any kind of test in any kind of transcompiled cross-platform applications,” in Proceedings of the 2nd International Workshop on User Interface Test Automation, in INTUITEST 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1–10. doi: 10.1145/2945404.2945405.

[5] P. Chaber and M. Ławryńczuk, ”Effectiveness of PID and DMC control algorithms automatic code generation for microcontrollers: Application to a thermal process,”2016 3rd Conference on Control and Fault-Tolerant Systems (SysTol), Barcelona, Spain, 2016, pp. 618-623, doi: 10.1109/SYSTOL.2016.7739817.

[6] M. Bysiek, A. Drozd, and S. Matsuoka, “Migrating Legacy Fortran to Python While Retaining Fortran-Level Performance through Transpilation and Type Hints,” 2016 6th Workshop on Python for High-Performance and Scientific Computing (PyHPC), pp. 9–18, 2016.

[7] A. Pilsch, “Translating the Future: Transpilers and the New Temporalities of Programming in JavaScript,” 2018.

[8] P. Misse-Chanabier, V. Aranega, G. Polito, and S. Ducasse, “Illicium A modular transpilation toolchain from Pharo to C,” in IWST19 - International Workshop on Small-

talk Technologies, Köln, Germany, Aug. 2019. [Online]. Available: https://hal.science/hal-02297860 (приступљено 19. јула 2024).

[9] E. Ilyushin and D. Namiot, "On source-to-source compilers," International Journal of Open Information Technologies, vol. 4, Apr. 2016.

[10] S. Muchnick, Advanced compiler design implementation. Morgan kaufmann, 1997.

[11] V. A. Alfred, S. L. Monica, and D. U. Jeffrey, Compilers principles, techniques & tools. pearson Education, 2007.

[12] A. Martelli, A. M. Ravenscroft, S. Holden, and P. McGuire, Python in a Nutshell. O'Reilly Media, Inc., 2023.

[13] B. Dayley, Python phrasebook: essential code and commands. Sams Pub., 2007.

[14] E. D. Berger, S. Stern, and J. A. Pizzorno, "Triangulating python performance issues with SCALENE," in 17th USENIX Symposium on Operating Systems Design and Implementation (OSDI 23), 2023, pp. 51–64.

[15] S. Klabnik, "The History of Rust," Association for Computing Machinery (ACM), on YouTube, June, vol. 22, 2016.

[16] S. Klabnik and C. Nichols, The Rust programming language. No Starch Press, 2023.

[17] W. Bugden and A. Alahmar, "Rust: The programming language for safety and performance," arXiv preprint arXiv:2206.05503, 2022.

[18] M. J. Sottile, T. G. Mattson, and C. E. Rasmussen, Introduction to concurrency in programming languages. CRC Press, 2009.

[19] K. R. Fulton, A. Chan, D. Votipka, M. Hicks, and M. L. Mazurek, "Benefits and drawbacks of adopting a secure programming language: Rust as a case study," in Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021), 2021, pp. 597–616.

[20] J. Konchunas, "Transpiling Python to Rust. An experimental project for converting… | by Julian Konchunas | Medium," Transpiling Python to Rust. [Online]. Available: https://medium.com/@konchunas/transpiling-python-to-rust-766459b6ab8f (приступљено 21. јула 2024).

[21] py2many, "py2many" GitHub repository. GitHub [Online]. Available: `https://github.com/py2many/py2many` (приступљено 21. јула 2024).

[22] H. Lunnikivi, K. Jylkkä, and T. Hämäläinen, "Transpiling python to rust for optimized performance," in International Conference on Embedded Computer Systems, Springer, 2020, pp. 127–138.

[23] L. S. Blackford et al., "An updated set of basic linear algebra subprograms (BLAS)," ACM Transactions on Mathematical Software, vol. 28, no. 2, pp. 135–151, 2002.

[24] S. Weisberg, Applied linear regression, vol. 528. John Wiley & Sons, 2005.

[25] D. W. Hosmer Jr, S. Lemeshow, and R. X. Sturdivant, Applied logistic regression. John Wiley & Sons, 2013.

[26] Jagadish, K. (2019). Diabetics prediction using logistic regression, Version 1. [Online]. Available: `https://www.kaggle.com/datasets/kandij/diabetes-dataset` (приступљено 21. јула 2024).

[27] G. Guo, H. Wang, D. Bell, Y. Bi, and K. Greer, "KNN model-based approach in classification," in On The Move to Meaningful Internet Systems 2003: CoopIS, DOA, and ODBASE: OTM Confederated International Conferences, CoopIS, DOA, and ODBASE 2003, Catania, Sicily, Italy, November 3-7, 2003. Proceedings, Springer, 2003, pp. 986–996.

[28] Saurabh (2018). KNN Data, Version 1. [Online]. Available: `https://www.kaggle.com/datasets/sp241930/knn-data1/` (приступљено 25. јула 2024).

[29] G. I. Webb, E. Keogh, and R. Miikkulainen, "Naïve Bayes.," Encyclopedia of machine learning, vol. 15, no. 1, pp. 713–714, 2010.

[30] C. Cortes and V. Vapnik, "Support-vector networks," Machine learning, vol. 20, pp. 273–297, 1995.

[31] C. J. Burges, "A tutorial on support vector machines for pattern recognition," Data mining and knowledge discovery, vol. 2, no. 2, pp. 121–167, 1998.

[32] N. Nethercote and J. Seward, "Valgrind: a framework for heavyweight dynamic binary instrumentation," ACM Sigplan notices, vol. 42, no. 6, pp. 89–100, 2007.

[33] C. R. Harris et al., “Array programming with NumPy,” Nature, vol. 585, no. 7825, pp. 357–362, 2020.

## 8 Биографија

Марко Ердељи је рођен 14. септембра 2001. године у Новом Саду. Завршио је Основну школу „Свети Сава“ у Житишту, где је носилац Вукове дипломе и звања Ђака генерације. Средњу школу завршио је у Зрењанину, на ЕГШ „Никола Тесла“, такође као носилац Вукове дипломе. Године 2020. уписао је Факултет техничких наука Универзитета у Новом Саду, где полаже све испите предвиђене планом и програмом са просечном оценом 9,36. Током студија учествовао је на такмичењима из машинског учења и сајбер сигурности, са оствареним запаженим резултатима.