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 ASM тип фајлове у A86 тип фајлове [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 инжењерског задатка. Кроз детаљну 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 детаљном анализом алата и техника. Фокус ће бити на документацији процеса транспилације, укључујући кораке за постављање и коришћење транспајлера, као и приказивање примера кода пре и после транспилације. Посебна пажња биће посвећена студији случаја кроз примере мањих пројеката у области машинског учења, што ће омогућити дубље разумевање изазова, решења и потенцијалних унапређења.

На крају, у дискусији 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.

[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.

2 Теоријска Позадина

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 интермедијалну репрезентацију. 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 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.

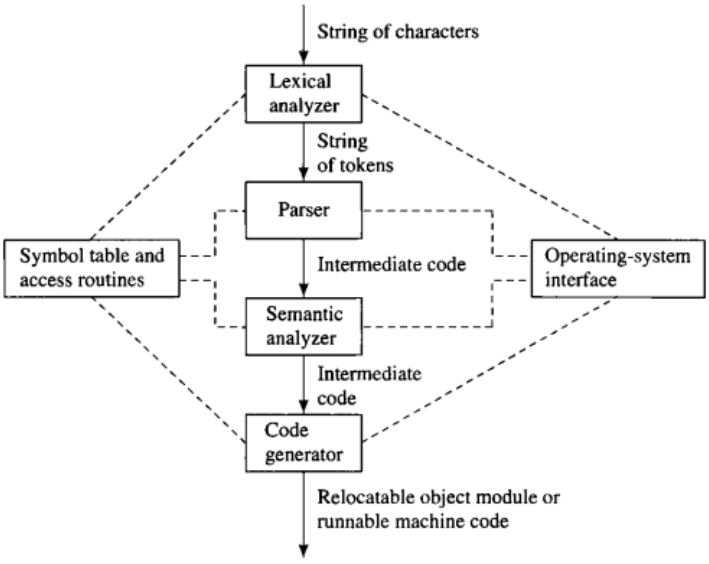
- **Семантичка провера (енгл. *semantic checking*):** Врши проверу програма за статичку - семантичку исправност, утврђујући да ли програм задовољава статичке - семантичке особине захтеване од изворног језика.

- **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 високонивне језике, 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.



Slika 1: Структура *frontend* - а традиционалног транспајлера

[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].

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 јединицу може смањити *overhead* позива функција[7], посебно у језицима где се семантика позива функција значајно разликује.
- **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.
- **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 инлинирању 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.

[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) потребан за позивање функција.

[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.

- **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 стабилан 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.е.

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 компилиране 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.

[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.

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

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

- **Моћ:** Стално се развијају нови проширења за *Python*, као што су приступ базама података, аудио/видео уређивање, графички кориснички интерфејс, веб развој и тако даље.

- **Динамичност:** *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.

- **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.

[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.

[14] Процес праћења и анализе перформанси програма.

- **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`, *Sci-Kit - 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 главни креатор је Грејдон Хоар. 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 јединственој 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.

- **Перформансе:** *Rust* пружа перформансе на нивоу са *C* и *C++* захваљујући концепту апстракције са нултим трошковима, што значи да можете писати висококвалитетни код 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.

[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.

[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.

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

- **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, може бити сложен и захтевати веће време учења за нове програмере.

- **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 [12] [14] [16] [17], можемо идентификовати кључне разлоге зашто би транспајловање кода из *Python* - а у *Rust* могло бити корисно. 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

[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.

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 детаљну контролу над ресурсима, што је посебно корисно у окружењима са ограниченим ресурсима. Користећи *Rust* за 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, детаљно ћемо анализирати најновију верзију *PyRs* - а која је интегрисана у *py2many* пројекат [21].

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 тип - система *Rust* - а, омогућавајући високоперфомантне имплементације које одговарају нативним моделима извршавања циљаних платформи. Строги тип - систем *Rust* - а и семантика алијасинга показивача 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

Функција `initialize_parameters` Ова функција иницијализује параметре моделирања, односно тежине и пристрасност. Тежине се иницијализују на нуле, а пристрасност се такође иницијализује на нулу. Функција враћа листу тежина и пристрасност као tuple.

```
def initialize_parameters(n_features: int)  - > Tuple[List[float],
    float]:
    weights = [0.0] * n_features
    bias = 0.0
    return weights, bias
```

Функција `predict` Ова функција врши предвиђање на основу улазних података X, тежина и пристрасности. За сваки узорак у X, израчунава се предвиђена вредност као збир производа карактеристика и тежина плус пристрасност.

```
def predict(X: List[List[float]], weights: List[float], bias:
   float)  - > List[float]:
    predictions = []
```

```
    for i in range(len(X)):
        prediction = sum(X[i][j] * weights[j] for j in range(len(X
            [0]))) + bias
        predictions.append(prediction)
    return predictions
```

Функција `compute_gradients` Ова функција израчунава градијенте тежина и пристрасноста. Градијенти су израчунати на основу разлике између предвиђених вредности y_pred и стварних вредности y. Функција враћа градијенте за тежине и пристрасност као tuple.

```
def compute_gradients(X: List[List[float]], y: List[float], y_pred
    : List[float], n_samples: int)  - > Tuple[List[float], float]:
    n_features = len(X[0])
    dw = [0.0] * n_features
    db = 0.0

    for i in range(n_samples):
        error = y_pred[i] - y[i]
        for j in range(n_features):
            dw[j] += (2 / n_samples) * X[i][j] * error
        db += (2 / n_samples) * error

    return dw, db
```

Функција `update_parameters` Ова функција ажурира тежине и пристрасност на основу израчунатих градијената и стопе учења. Тежине и пристрасност се коригују смањењем за вредност градијента помножену са стопом учења.

```
def update_parameters(weights: List[float], bias: float, dw: List[
    float], db: float, learning_rate: float)  - > Tuple[List[float
    ], float]:
    for j in range(len(weights)):
        weights[j]  - = learning_rate * dw[j]
    bias  - = learning_rate * db
    return weights, bias
```

Функција `linear_regression` Ова функција извршава процес обучавања модела линеарне регресије. Почетно иницијализује тежине и пристрасност, а затим у сваком од **`epochs`** итерација рачуна предвиђања, градијенте и ажурира параметре. На крају, враћа оптимизоване тежине и пристрасност.

```
def linear_regression(X: List[List[float]], y: List[float],
   learning_rate: float = 0.01, epochs: int = 1000)  - > Tuple[
   List[float], float]:
    n_samples, n_features = len(X), len(X[0])
    weights, bias = initialize_parameters(n_features)

    for _ in range(epochs):
        y_pred = predict(X, weights, bias)
        dw, db = compute_gradients(X, y, y_pred, n_samples)
        weights, bias = update_parameters(weights, bias, dw, db,
           learning_rate)

    return weights, bias
```

Функција `main` У овој секцији, подаци за обуку се дефинишу, функција `linear_regression` се позива за обуку модела, а затим се предвиђања израчунавају помоћу обучених параметара.

```
if __name__ == "__main__":
    X: List[List[float]] = [[1], [2], [3], [4], [5]]
    y: List[float] = [1, 2, 3, 4, 5]

    learning_rate: float = 0.01
    epochs: int = 1000
    weights, bias = linear_regression(X, y, learning_rate, epochs)

    predictions = predict(X, weights, bias)

    print("Predictions:", predictions)
```

4.2.2 Анализа транспајлираног Rust 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);
}
```

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)
}
```

У ревидираном коду, тип **n_features** је промењен из **i32** у **usize**, и исправно је коришћена синтакса **vec![0.0; n_features]** за креирање вектора тежина.

Транспајлирани код

```
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 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
}
```

У ревидираном коду, коришћена је једноставнија и оптимизована синтакса за израчунавање предикција помоћу **iter().zip()** и **map()** метода, што побољшава читљивост и перформансе кода.

Транспајлирани код

```
pub fn compute_gradients(X: &Vec<Vec<f64>>, y: &Vec<f64>, y_pred:
   &Vec<f64>, n_samples: i32) {
let n_features = X[0 as usize].len() as i32;
let mut dw: &mut Vec<f64> = &mut (vec![0.0] * n_features);
let mut db: f64 = 0.0;
```

```
for i in (0..n_samples) {
let error: f64 = (y_pred[i as usize] - y[i as usize]);
for j in (0..n_features) {
dw[j as usize] +=
((((2 as f64) / (n_samples as f64)) * (X[i as usize][j] as f64)) *
    error);
}
db += (((2 as f64) / (n_samples as f64)) * error);
}
return (dw, db);
}
```

Ревидирани код

```
pub fn compute_gradients(
X: &Vec<Vec<f64>>,
y: &Vec<f64>,
y_pred: &Vec<f64>,
n_samples: usize
)  - > (Vec<f64>, f64) {
let n_features = X[0].len();
let mut dw: Vec<f64> = vec![0.0; n_features];
let mut db: f64 = 0.0;
for i in 0..n_samples {
    let error = y_pred[i] - y[i];
    for j in 0..n_features {
        dw[j] += (2.0 / n_samples as f64) * X[i][j] * error;
    }
    db += (2.0 / n_samples as f64) * error;
}
(dw, db)
}
```

У ревидираном коду, тип **`n_samples`** је промењен из **`i32`** у **`usize`**, а вектор **`dw`** је исправно иницијализован са **`vec![0.0; n_features]`** синтаксом која је у складу са *Rust* језиком.

Транспајлирани код

```
pub fn update_parameters(
weights: &Vec<f64>,
bias: f64,
dw: &Vec<f64>,
db: f64,
learning_rate: f64,
) {
for j in (0..weights.len() as i32) {
```

```
weights[j as usize]  - = (learning_rate * dw[j as usize]);
}
bias  - = (learning_rate * db);
return (weights, bias);
}
```

Ревидирани код

```
pub fn update_parameters(
mut weights: Vec<f64>,
mut bias: f64,
dw: &Vec<f64>,
db: f64,
learning_rate: f64
)  - > (Vec<f64>, f64) {
for j in 0..weights.len() {
weights[j]  - = learning_rate * dw[j];
}
bias  - = learning_rate * db;
(weights, bias)
}
```

У ревидираном коду, параметри **`weights`** и **`bias`** су дефинисани као **`mut`**, што омогућава њихову измену без потребе за враћањем нових вредности као резултат. Петља је поједностављена уклањањем кастовања типа, што чини код лакшим за читање. Такође, уклоњен је непотребан **`return`** и функција сада директно враћа резултате.

Транспајлирани код

```
pub fn linear_regression(X: &Vec<Vec<f64>>, y: &Vec<f64>,
   learning_rate: f64, epochs: i32) {
let (n_samples, n_features) = (X.len() as i32, X[0 as usize].len()
    as i32);
let (weights, bias): &_ = &initialize_parameters(n_features);
for _ in (0..epochs) {
let y_pred: &Vec<f64> = &predict(X, &weights, bias);
let (dw, db): &_ = &compute_gradients(X, y, y_pred, n_samples);
let (weights, bias): &_ = &update_parameters(&weights, bias, &dw,
   db, learning_rate);
}
return (weights, bias);
}
```

Ревидирани код

```
pub fn linear_regression(
X: &Vec<Vec<f64>>,
y: &Vec<f64>,
```

```
learning_rate: f64,
epochs: usize
)  - > (Vec<f64>, f64) {
let n_samples = X.len();
let n_features = X[0].len();
let (mut weights, mut bias) = initialize_parameters(n_features);
for _ in 0..epochs {
    let y_pred = predict(X, &weights, bias);
    let (dw, db) = compute_gradients(X, y, &y_pred, n_samples);
    let (updated_weights, updated_bias) = update_parameters(
       weights, bias, &dw, db, learning_rate);
    weights = updated_weights;
    bias = updated_bias;
}
(weights, bias)
}
```

У ревидираном коду, параметри су једноставнији са типом **usize** за **epochs**, што је боље за рад са величинама. Параметри **weights** и **bias** су дефинисани као **mut** како би могли да се ажурирају без враћања нових вредности. Код је поједностављен уклањањем непотребних референци и **return** изјаве, што побољшава читљивост и лакоћу одржавања.

Транспајлирани код

```
pub fn main()  - > Result<()> {
let X: Vec<Vec<f64>> = vec![vec![1], vec![2], vec![3], vec![4],
   vec![5]];
let y: Vec<f64> = vec![1, 2, 3, 4, 5];
let learning_rate: f64 = 0.01;
let epochs: i32 = 1000;
let (weights, bias): &_ = &linear_regression(X, y, learning_rate,
   epochs);
let predictions: &Vec<f64> = &predict(X, &weights, bias);
println!("{} {}", "Predictions:", predictions);
Ok(())
}
```

Ревидирани код

```
pub fn main()  - > Result<()> {
let X: Vec<Vec<f64>> = vec![vec![1.0], vec![2.0], vec![3.0], vec
   ![4.0], vec![5.0]];
let y: Vec<f64> = vec![1.0, 2.0, 3.0, 4.0, 5.0];
let learning_rate: f64 = 0.01;
let epochs: usize = 1000;
```

```
let (weights, bias) = linear_regression(&X, &y, learning_rate,
   epochs);
let predictions = predict(&X, &weights, bias);
println!("Predictions: {:?}", predictions);

Ok(())
}
```

У ревидираном коду, уместо `i32`, `epochs` је сада `usize`, а `X` и `y` користе `f64` уместо `i32`. Испис резултата је такође исправљен коришћењем `{:?}` за правилан приказ `Vec<f64>`.

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 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 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

Функција `load_data` учитава податке из *CSV* датотеке у `numpy` низ.

```
def load_data(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()
    header = lines[0].strip().split(',')
    data = [line.strip().split(',') for line in lines[1:]]
    data = np.array(data, dtype=str)
    return data
```

Функција `convert_to_float` конвертује све податке у бројеве са покретном запетом.

```
def convert_to_float(data):
    return data.astype(float)
```

Функција `feature_scaling` стандардизује карактеристике тако што уклања просек и дели са стандардном девијацијом.

```
def feature_scaling(X):
    means = np.mean(X, axis=0)
    stds = np.std(X, axis=0)
    return (X - means) / stds
```

Функција `split_data` дели податке на тренинг и тест скуп.

```
def split_data(data, test_ratio=0.2):
    num_samples = data.shape[0]
    num_test_samples = int(num_samples * test_ratio)
    X_train = data[: - num_test_samples, : - 1]
    y_train = data[: - num_test_samples,  - 1]
    X_test = data[ - num_test_samples:, : - 1]
    y_test = data[ - num_test_samples:,  - 1]
    return X_train, y_train, X_test, y_test
```

Функција `sigmoid` рачуна сигмоидну функцију за датим вредностима z.

```
def sigmoid(z):
    z = np.clip(z,  - 500, 500)
    return 1 / (1 + np.exp( - z))
```

Функција `compute_cost` рачуна цену логистичке регресије на основу тренутних параметара θ.

```
def compute_cost(X, y, theta):
    m = len(y)
    h = sigmoid(X.dot(theta))
    h = np.clip(h, 1e - 10, 1 - 1e - 10)
    cost = ( - 1 / m) * (y.dot(np.log(h)) + (1 - y).dot(np.log(1 -
        h)))
    return cost
```

Функција `gradient` израчунава градијент функције цене.

```
def gradient(X, y, theta):
    m = len(y)
    h = sigmoid(X.dot(theta))
    grad = (1 / m) * X.T.dot(h - y)
    return grad
```

Функција `gradient_descent` оптимизује параметре θ коришћењем градијентног спуста.

```
def gradient_descent(X, y, theta, alpha, num_iters):
    m = len(y)
    cost_history = np.zeros(num_iters)
    for i in range(num_iters):
        theta  - = alpha * gradient(X, y, theta)
        cost_history[i] = compute_cost(X, y, theta)
    return theta, cost_history
```

Функција `predict` користи оптимизоване параметре θ за прављење прогнозе на основу карактеристика X.

```
def predict(X, theta):
    return sigmoid(X.dot(theta)) >= 0.5
```

Функција `main` извршава читав процес обуке и тестирања модела логистичке регресије. Она учитава податке, обавља обраду, дели податке на тренинг и тест сетове, и врши оптимизацију параметара помоћу градијентног спуста. На крају, приказује тачност модела на тренинг и тест сетовима.

```
if __name__ == "__main__":
    data = load_data('data.csv')
    data = convert_to_float(data)

    X_train, y_train, X_test, y_test = split_data(data)

    X_train = feature_scaling(X_train)
    X_test = feature_scaling(X_test)

    X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))
    X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test))

    theta = np.zeros(X_train.shape[1])
    alpha = 0.01
    num_iters = 2000
```

```
    theta, cost_history = gradient_descent(X_train, y_train, theta
        , alpha, num_iters)

    print(f'Final cost: {cost_history[ - 1]}')
    print(f'Optimal parameters: {theta}')

    train_predictions = predict(X_train, theta)
    test_predictions = predict(X_test, theta)

    train_accuracy = np.mean(train_predictions == y_train) * 100
    test_accuracy = np.mean(test_predictions == y_test) * 100

    print(f'Training Accuracy: {train_accuracy:.2f}%')
    print(f'Test Accuracy: {test_accuracy:.2f}%')
```

4.3.2 Анализа транспајлираног *Rust* кода

Транспајлирани код

```
extern crate anyhow;
extern crate numpy;
extern crate pylib;
use anyhow::Result;
use pylib::FileReadString;
use std::collections;
use std::fs::File;
use std::fs::OpenOptions;
```

Ревидирани код

```
extern crate anyhow;
extern crate ndarray;
use anyhow::Result;
use ndarray::s;
use ndarray::{Array, Array1, Array2, Axis};
use std::collections;
use std::fs::OpenOptions;
use std::io::BufRead;
use std::io::BufReader;
```

Промене су направљене како би се побољшала подршка за рад са подацима у Rust - у. У транспајлираном коду, библиотеке **numpy** и **pylib** замењене су са `ndarray`, што омогућава напредније операције над низовима и генерацију случајних података. Такође, додати су модули `BufRead` и `BufReader` за боље управљање читањем података из фајлова. Ове измене су учињене како би се код учинио ефикаснијим.

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)
```

Ревизија кода доноси неколико кључних побољшања у односу на транспајлирани код. Ревидирани код користи *Rust* - ов стандард за рад са фајловима и низовима, омогућава ефикасније читање и парсирање података и јасно дефинише повратне типове. Транспајлирани код користи неконпатибилну синтаксу и методе из *Python* - а, што може довести до грешака и проблема са интеграцијом у *Rust* окружењу.

Транспајлирани код

```
pub fn convert_to_float<T0, RT>(data: T0)  - > RT {
    return data.astype(float);
}
```

Ревидирани код

```
pub fn convert_to_float(data: Array2<f64>)  - > Array2<f64> {
    data
}
```

Разлика између ова два дела кода лежи у приступу конверзије типова и синтакси. Оригинални код користи генераички тип и *Python* специфичну методу **astype**, која није компатибилна са *Rust* - ом.

Транспајлирани код

```
pub fn feature_scaling<T0, RT>(X: T0)  - > RT {
    let means = np.mean(X, 0);
    let stds = np.std(X, 0);
    return ((X - means) / stds);
}
```

Ревидирани код

```
pub fn feature_scaling(X: Array2<f64>)  - > Array2<f64> {
    let means = X.mean_axis(Axis(0)).unwrap();
    let stds = X.std_axis(Axis(0), 0.0);
    (X - &means) / &stds
}
```

Транспајлирани код користи numpy специфичне методе **np.mean** и **np.std**, које нису компатибилне са *Rust* - ом. Ревидирани код користи методе **mean_axis** и **std_axis** из *Rust* - ове библиотеке ndarray, што је исправно за рад са типом **Array2<f64>**. Ревидирани код такође користи референце на **means** и **stds**, чиме побољшава перформансе избегавањем копирања података.

Транспајлирани код

```
pub fn split_data<T0, T1, RT>(data: T0, test_ratio: T1)  - > RT {
    let num_samples = data.shape[0];
    let num_test_samples: i32 = (num_samples * test_ratio) as i32;
    let X_train = data[(.. - (num_test_samples), .. - 1)];
    let y_train = data[(.. - (num_test_samples),  - 1)];
    let X_test = data[( - (num_test_samples).., .. - 1)];
    let y_test = data[( - (num_test_samples)..,  - 1)];
    return (X_train, y_train, X_test, y_test);
}
```

Ревидирани код

```
pub fn split_data(
    data: Array2<f64>,
    test_ratio: f64,
)  - > (Array2<f64>, Array1<f64>, Array2<f64>, Array1<f64>) {
    let num_samples = data.nrows();
    let num_test_samples = (num_samples as f64 * test_ratio) as
       usize;
    let X_train = data.slice(s![.. - (num_test_samples as isize),
       .. - 1]).to_owned();
    let y_train = data.slice(s![.. - (num_test_samples as isize),
        - 1]).to_owned();
    let X_test = data.slice(s![ - (num_test_samples as isize)..,
       .. - 1]).to_owned();
    let y_test = data.slice(s![ - (num_test_samples as isize)..,
       - 1]).to_owned();
    (X_train, y_train, X_test, y_test)
}
```

У транспајлираном коду користе се методе које нису компатибилне са *Rust* - ом. Ове методе подразумевају синтаксу и типове података који нису доступни у *Rust* - овом окружењу. Ревидирани код користи *Rust* библиотеку `ndarray`, која пружа исправне методе за рад са подацима у облику `Array2<f64>`. Методе као што су `slice` и `s!` омогућавају правилно раздвајање података на обуке и тестове, при чему су подаци конвертовани у `Array2<f64>` и `Array1<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)))
}
```

Методе `np.clip` и `np.exp` нису компатибилне са *Rust* - ом. Ове методе не одговарају синтакси и типовима података који су доступни у *Rust* окружењу. Ревидирани код користи Rust библиотеку `ndarray` и њене методе `mapv` и `clamp`, што је исправно за рад са подацима типа `Array1<f64>`.

Транспајлирани код

```
pub fn compute_cost<T0, T1, T2>(X: T0, y: T1, theta: T2)  - > i32
   {
    let m = y.len() as i32;
    let mut h: f64 = sigmoid(X.dot(theta));
    h = np.clip(h, 1e - 10, ((1 as f64) - 1e - 10));
    let cost: i32 =
        (( - 1 / (m as i32)) * ((y.dot(np.log(h)) + (1 - y).dot(np
           .log(((1 as f64) - h)))) as i32));
    return cost;
}
```

Ревидирани код

```
pub fn compute_cost(X: &Array2<f64>, y: &Array1<f64>, theta: &
   Array1<f64>)  - > f64 {
    let m = y.len() as f64;
    let h = sigmoid(X.dot(theta));
    let h_clipped = h.mapv(|v| v.clamp(1e - 10, 1.0 - 1e - 10));
    ( - 1.0 / m) * (y.dot(&h_clipped.mapv(f64::ln)) + (1.0 - y).
       dot(&(1.0 - h_clipped).mapv(f64::ln)))
}
```

Методе **`np.clip`** и **`np.log`** нису компатибилне са *Rust* - ом. Ове методе не одговарају синтакси и типовима података који су доступни у *Rust* окружењу. Ревидирани код користи *Rust* библиотеку `ndarray` и њене методе **`mapv`** и **`clamp`**.

Транспајлирани код

```
pub fn gradient<T0, T1, T2>(X: T0, y: T1, theta: T2)  - > i32 {
    let m = y.len() as i32;
    let h: f64 = sigmoid(X.dot(theta));
    let grad: i32 = ((1 / (m as i32)) * (X.T.dot((h - (y as f64)))
        as i32));
    return grad;
}
```

Ревидирани код

```
pub fn gradient(X: &Array2<f64>, y: &Array1<f64>, theta: &Array1<
   f64>)  - > Array1<f64> {
    let m = y.len() as f64;
    let h = sigmoid(X.dot(theta));
    X.t().dot(&(h - y)) / m
}
```

Ревидирани код користи *Rust* библиотеку `ndarray` и њене методе као што су **`t`** и **`dot`**, и рачуна градијент користећи податке типа **`Array2<f64>`** и **`Array1<f64>`**.

Транспајлирани код

```
pub fn gradient_descent<T0, T1, T2, T3, T4, RT>(
    X: T0,
    y: T1,
    theta: T2,
    alpha: T3,
    num_iters: T4,
) -> RT {
    let m = y.len() as i32;
    let mut cost_history = np.zeros(num_iters);
    for i in (0..num_iters) {
        theta -= ((alpha as i32) * gradient(X, y, theta));
        cost_history[i] = compute_cost(X, y, theta);
    }
    return (theta, cost_history);
}
```

Ревидирани код

```
pub fn gradient_descent(
    X: &Array2<f64>,
    y: &Array1<f64>,
    mut theta: Array1<f64>,
    alpha: f64,
    num_iters: usize,
) -> (Array1<f64>, Array1<f64>) {
    let m = y.len() as f64;
    let mut cost_history = Array::zeros(num_iters);
    for i in 0..num_iters {
        theta -= &(alpha * &gradient(X, y, &theta));
        cost_history[i] = compute_cost(X, y, &theta);
    }
    (theta, cost_history)
}
```

Метода **`np.zeros`** није доступна у *Rust* окружењу. Ревидирани код користи *Rust* библиотеку ndarray и њене методе као што су **`Array::zeros`** и коректно рачуна градијентни спуст користећи податке типа **`Array1<f64>`** и **`Array2<f64>`**. Ревидирани код такође користи **`&`** операције за избегавање копирања података.

Транспајлирани код

```
pub fn predict<T0, T1>(X: T0, theta: T1)  - > bool {
    return sigmoid(X.dot(theta)) >= 0.5;
}
```

Ревидирани код

```
pub fn predict(X: &Array2<f64>, theta: &Array1<f64>)  - > Array1<
   bool> {
    let sigmoid_values = sigmoid(X.dot(theta));
    sigmoid_values.mapv(|value| value >= 0.5)
}
```

Метод **`sigmoid`** у транспајлираном коду враћа `bool` тип, што није прикладно за рад са вектором вероватноћа. Ревидирани код користи *Rust* библиотеку `ndarray` за рад са подацима типа **`Array2<f64>`** и **`Array1<f64>`**, враћајући **`Array1<bool>`** који представља резултат примене функције **`sigmoid`** на сваки елемент вектора. Метод **`mapv`** се користи за претварање вредности у `bool` тип на основу задатог прага.

Транспајлирани код

```
pub fn main()  - > Result<()> {
    let mut data = load_data("data.csv");
    data = convert_to_float(data);
    let (X_train, y_train, X_test, y_test) = split_data(data);
    let mut X_train = feature_scaling(X_train);
    let mut X_test = feature_scaling(X_test);
    X_train = np.hstack((np.ones((X_train.shape[0], 1)), X_train))
       ;
    X_test = np.hstack((np.ones((X_test.shape[0], 1)), X_test));
    let mut theta = np.zeros(X_train.shape[1]);
    let alpha: f64 = 0.01;
    let num_iters: i32 = 2000;
    let (theta, cost_history) = gradient_descent(X_train, y_train,
        theta, alpha, num_iters);
    println!(
        "{}",
        vec!["Final cost: ", &cost_history[ - 1].to_string()].join
           ("")
    );
    println!(
        "{}",
        vec!["Optimal parameters: ", &theta.to_string()].join("")
    );
    let train_predictions: bool = predict(X_train, theta);
    let test_predictions: bool = predict(X_test, theta);
```

```
    let train_accuracy: i32 = ((np.mean(train_predictions == (
        y_train as bool)) as i32) * 100);
    let test_accuracy: i32 = ((np.mean(test_predictions == (y_test
        as bool)) as i32) * 100);
    println!(
        "{}",
        vec!["Training Accuracy: ", &train_accuracy.to_string(), "
            %"].join("")
    );
    println!(
        "{}",
        vec!["Test Accuracy: ", &test_accuracy.to_string(), "%"].
            join("")
    );
    Ok(())
}
```

Ревидирани код

```
pub fn main()  - > Result<()> {
    let data = load_data("data.csv")?;
    let data = convert_to_float(data);
    let (X_train, y_train, X_test, y_test) = split_data(data, 0.2)
        ;
    let X_train = feature_scaling(X_train);
    let X_test = feature_scaling(X_test);
    let X_train = ndarray::concatenate![Axis(1), Array::ones((
        X_train.nrows(), 1)), X_train];
    let X_test = ndarray::concatenate![Axis(1), Array::ones((
        X_test.nrows(), 1)), X_test];
    let theta = Array::zeros(X_train.ncols());
    let alpha = 0.01;
    let num_iters = 2000;
    let (theta, cost_history) = gradient_descent(&X_train, &
        y_train, theta, alpha, num_iters);
    println!("Final cost: {}", cost_history[num_iters - 1]);
    println!("Optimal parameters: {:?}", theta);
    let train_predictions = predict(&X_train, &theta);
    let test_predictions = predict(&X_test, &theta);

    let train_correct: Array1<f64> = train_predictions
        .iter()
        .zip(y_train.iter())
        .map(|(&pred, &actual)| (pred == (actual != 0.0)) as u8 as
            f64)
        .collect::<Array1<_>>();
```

```
    let train_accuracy = (train_correct.mean().unwrap() * 100.0)
       as i32;

    let test_correct: Array1<f64> = test_predictions
        .iter()
        .zip(y_test.iter())
        .map(|(&pred, &actual)| (pred == (actual != 0.0)) as u8 as
            f64)
        .collect::<Array1<_>>();
    let test_accuracy = (test_correct.mean().unwrap() * 100.0) as
       i32;

    println!("Training Accuracy: {}%", train_accuracy);
    println!("Test Accuracy: {}%", test_accuracy);
    Ok(())
}
```

Методе као што су **np.hstack** и **np.mean** нису компатибилне са *Rust* окружењем. Ревидирани код користи *Rust* библиотеку ndarray за рад са подацима типа **Array2<f64>** и **Array1<f64>**, као и њене методе као што су **concatenate** и **mean**. Метод **concatenate** се користи за додавање колоне са јединицама, а метод **mean** се користи за израчунавање тачности. Методе као што су **np.mean** и операције попут **train_predictions == (y_train as bool)** нису компатибилне са *Rust* окружењем. Примерно, у транспајлираном коду, **np.mean** се користи на логичким вредностима које су у *Python* - у природне за рад, али у *Rust* окружењу, рад са таквим вредностима захтева другачији приступ.

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

Tokom procesa transpajliranja iz *Python* - a u *Rust*, pojavili su se različiti izazovi vezani za kompatibilnost između okruženja i korišćenih biblioteka.

Код за линеарну регресију је током процеса транспајлирања користио *Python* - ов тип систем из библиотеке `typing`, што је омогућило лакши прелазак у *Rust*. Овај приступ се фокусирао на основне типове података и избегавао употребу екстерних библиотеках. Ова одлука је поједноставила процес транспајлирања, јер је типски систем из *Python* - а омогућио прецизно дефинисање типова података и функција, што је олакшало њихову имплементацију у статички типизованом *Rust* - у.

Међутим, без коришћења специјализованих библиотека, напредне функције као што су статистички прорачуни и рад са матрицама и векторима морале су бити имплементиране ручно у *Rust* - у. Коришћење основних типова и функција омогућило је процену како транспајлер обрађује типске анотације и основне структуре података, што је углавном било успешно. Ипак, транспајлер се суочио са изазовима у управљању позајмљивањем вредности, где је било потребно ручно прилагођавање кода. Такође, понекад није успео да правилно изабере одговарајући тип вредности.

Код за логистичку регресију суочавао се с изазовима при преласку са Python библиотеки као што је `numpy` на одговарајућу *Rust* библиотеку `ndarray`. Док `numpy` пружа широк спектар функција за манипулацију низовима и статистичке прорачуне, *Rust* - ова библиотека `ndarray`, иако нуди сличну функционалност, користи другачију синтаксу и приступ. На пример, функције за додавање колона и израчунавање средњих вредности у `numpy` - у замењене су методама из `ndarray` библиотеке које су прилагођене *Rust* окружењу. Ово је захтевало прилагођавање функционалности и метода како би се осигурала тачност и ефикасност у новом окружењу.

Поред тога, управљање логичким вредностима у *Rust* - у разликује се од *Python* - овог приступа. Док *Python* аутоматски обрађује логичке вредности, Rust захтева експлицитне методе за мапирање и трансформацију података. Употреба `ndarray` библиотеке омогућила је имплементацију функционалности као што су трансформације логичких вредности и основне математичке операције.

Такође, употреба модула као што су `BufRead` и `BufReader` била је неопходна за ефикасно управљање великим количинама података. Првобитно је транспајлер превео код тако да користи `pylib`, *Python* интерфејс, што би било неефикасно у смислу перформанси. Користећи `BufRead` и `BufReader`, осигурана је боља ефикасност у раду са великим подацима.

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 различитих 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 [27]. У *Python* - у, `memory_profiler` пакет се користи за мерење меморијских ресурса које сваку функцију у програму користи.

Кодови за линеарну регресију и логистичку регресију су прилагођени за ове тестове.

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.

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

Табела 2 упоређује време извршавања за линеарну и логистичку регресију у *Python* - у и *Rust* - у. За линеарну регресију са генерисаним вредностима и без коришћења `numpy` библиотеке у *Python* - у, *Rust* је био приближно 179.3 пута бржи, док је за логистичку регресију где се користе `numpy` за *Python* и `ndarray` за *Rust*, *Rust* био приближно 10.1 пута бржи.

Метод	***Python***	***Rust***
Линеарна регресија	32.7982s	0.1833s
Логистичка регресија	3.5071s	0.3460s

Tabela 2: Време извршавања за методе из студије у Python - у и Rust - у.

5.2.2 Коришћење меморије

Табела 3 показује меморијску употребу за линеарну и логистичку регресију у *Python* - у и *Rust* - у. У оба случаја, *Rust* показује значајно смањење потрошње меморије у поређењу са *Python* - ом. За линеарну регресију, меморија 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%.

Метод	***Python***	***Rust***
Линеарна регресија	3.03 MB	1.27 MB
Логистичка регресија	3.73 MB	0.71 MB

Tabela 3: Меморијска употреба метода из студије у *Python* - у и *Rust* - у.

6 Закључак

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

Време извршавања у *Rust* - у је значајно 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 [28]. Ова ефикасност проистиче из чињенице да `numpy` користи низ нискоинивних C функција за обраду података, 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 [28]. 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.

Међутим, 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 значајно унапредити ефикасност и тачност транспајлера, чинећи процес преласка кода из *Python* - а у *Rust* лакшим и ефикаснијим.

У закључку, транспајлирање *Python* кода у *Rust* за алгоритме линеарне и логистичке регресије 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 кључну 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 брже и ефикасније 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 Smalltalk 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] N. Nethercote and J. Seward, “Valgrind: a framework for heavyweight dynamic binary instrumentation,” ACM Sigplan notices, vol. 42, no. 6, pp. 89–100, 2007.

[28] 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 Житиштu, 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 полажу 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.