Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Jurij Šteblaj

Optimizacija statističnih modelov za izvajanje na heterogenih sistemih

magistrsko delo

magistrski študijski program druge stopnje
računalništvo in informatika

Mentor: izr. prof. dr. Uroš Lotrič
Somentor: asist. Rok Češnovar

Ljubljana, 2020

Zahvala

Zahvaljujem se mentorju izr. prof. dr. Urošu Lotriču za pomoč, nasvete in pregled dela, somentorju asist. Roku Češnovarju za nasvete glede orodja Stan in motivacijo dela ter družini za podporo in spodbudo.

Jurij Šteblaj, 2020

Kazalo

Povzetek

Abstract

1 Uvod **1**

1.1 Motivacija 1

1.2 Sorodna dela 2

1.3 Prispevki 4

1.4 Struktura dela 4

2 Problem večrokega bandita **5**

3 Odločitvene metode **7**

3.1 LinUCB z neodvisnimi modeli 7

3.2 Thompsonovo vzorčenje 8

4 Programska arhitektura **15**

4.1 Velikost primerka 16

4.2 Značilnice primerka 16

4.3 Odločitvene metode 17

4.4 Funkcije z izbiro 18

4.5 Uporaba v statističnem modelu 20

4.6 Primer postopka uporabe 21

5 Rezultati **23**

5.1 Računski problemi 24

5.2 Statistični model 29

6 Zaključki **33**

A Funkciji getChoiceFunc in getSimpleChoiceFunc **35**

B Razlike v kodi dveh modelov **37**

Seznam uporabljenih kratic

kratica	angleško	slovensko
CPE	central processing unit	centralna procesna enota
GPE	graphics processing unit	grafična procesna enota
UCB	upper confidence bound	zgornja meja zaupanja

Povzetek

Naslov: Optimizacija statističnih modelov za izvajanje na heterogenih sistemih

Statistični model v jeziku Stan lahko prevedemo tako, da se izvaja ali na centralni procesni enoti ali na grafični procesni enoti. Napačna izbira naprave lahko močno podaljša čas obdelave. V našem pristopu napravo izbiramo sproti in ob tem učimo odločitveno metodo. Pripravili in preizkusili smo tri take odločitvene metode. Metode se prilagodijo na strojno opremo in odločajo glede na velikost primerka računskega problema. V delu predstavimo programsko arhitekturo, ki omogoča uvedbo funkcij z vgrajenim odločanjem z majhnimi spremembami obstoječe kode. Odločitvene metode smo preizkusili z merjenjem časa izvajanja matematičnih operacij in realnega primera statističnega modela. Odločitvena metoda LinUCB je v vsakem preizkusu dosegla primerljiv ali krajši čas izvajanja, kot na vnaprej izbrani napravi. S predhodnim učenjem odločitvene metode, četudi na manjših primerkih, smo čase izvajanja še skrajšali.

Ključne besede

heterogeni sistemi, Bayesova statistika, GPE, vzorčenje, večroki bandit, optimizacija

Abstract

Title: Optimization of Statistical Models for Execution on Heterogeneous Systems

Statistical models made in Stan can execute on a central processing unit or a graphical processing unit. Incorrect choice of the device can significantly extend execution time. Our approach chooses the executing device and trains the decision method during program execution. We implemented and tested three such decision methods. The methods adjust to present hardware and make decisions based on the size of the problem instances. We offer a programming architecture, which allows for easy construction of functions with built-in decision methods. We tested the methods by measuring execution times of selected mathematical operations and a realistic statistical model. In every test case, the LinUCB decision method achieved a similar or shorter execution time than the methods with a device selected in advance. We further reduced execution time by training the decision method ahead of time, despite training instances being smaller than those used for testing.

Keywords

heterogeneous systems, Bayesian statistics, GPU, sampling, multi-armed bandit, optimization

Poglavje 1

Uvod

1.1 Motivacija

Z metodami Bayesove statistike določamo stopnjo verjetja v neki dogodek. Sodobne metode vključujejo veliko vzorčenja, med katerim se deli programa, kot je matrično množenje, večkrat ponovijo. Število parametrov statističnega modela in vzorcev je lahko zelo veliko, zato so za izvajanje primerne grafične procesne enote (GPE). Te lahko vzporedno obdelajo veliko število enot podatkov, vendar pa je celoten čas obdelave zaradi prenosa podatkov v njihov ločen spomin lahko veliko daljši od samega procesiranja [1]. Visoka vzporednost zato včasih ne odtehta dodatnega časa prenosa. Poleg tega posamezna nit na GPE ni tako zmogljiva kot tista na centralni procesni enoti (CPE). Izvajanje na počasni napravi, pa naj bo to CPE ali GPE, lahko traja zelo dolgo časa, zato se temu skušamo izogniti in učinkovito izkoristiti vire v uporabljenem heterogenem sistemu.

Stan je programski jezik za opisovanje statističnih modelov [2]. Model se da z orodjem Stanc prevesti v razred v programskem jeziku C++, s katerim se računa verjetnost parametrov modela glede na konstante in podatke. Verjetnost se uporabi v metodah iz družine Hamiltonskih metod Monte Carlo, ki vzorčijo iz porazdelitve parametrov modela. V izračunih se uporabljajo operacije, ki se na nekaterih GPE izvedejo hitreje kot na CPE. Primera takih

operacij sta matrično množenje in razcep Choleskega. Orodje Stanc lahko program v jeziku Stan prevede v program v jeziku C++, ki uporablja izvedbe teh metod s knjižnico OpenCL. OpenCL uporabi eno izmed naprav na računalniku. To je navadno GPE, če je na voljo. Ta pristop ima več omejitev. Na hitrost izvedbe vplivajo tip računskega problema, izbrana izvedba algoritma – v kodi C++ ali z OpenCL, velikost primerka problema in lastnosti specifične strojne opreme. Velikost primerka je ločena dimenzija od računskega problema, saj se isti računski problem lahko večkrat pojavi v istem modelu z različnimi velikostmi vhodnih podatkov. Če se moramo odločiti, da bodo znotraj modela vedno uporabljane C++ ali OpenCL izvedbe, nam to ne omogoča, da bi bila izbira izvedbe prilagojena na računski problem ali velikost primerka. Poleg tega znanje o izbiri med C++ in OpenCL izvedbami ni prenosljivo zaradi razlik v strojni opremi uporabnikov. V idealnem primeru bi program sam, glede na računski problem, velikost primerka in lastnosti strojne opreme, izbral izvedbo, ki ima najkrajši čas izvajanja.

1.2 Sorodna dela

Veliko dela je že bilo opravljenega na numerični optimizaciji parametrov jedra (angl. kernel) – funkcije, ki se lahko izvede na GPE ali drugem čipu za vzporedno procesiranje [3]. Optimalne konfiguracije se med napravami razlikujejo, kar dopušča možnost, da izvedba dela programa na neki GPE deluje slabše kot pričakovano. Takrat bi lahko bila izvedba na CPE hitrejša od tiste na GPE. Razlike med napravami tudi zahtevajo prilagajanje izbire izvedbe na napravo uporabnika programa. Van Werkhoven [4] prav tako raziskuje problematiko konfiguracije jedra na heterogenih sistemih. Rezultat je orodje, s katerim lahko uporabnik z različnimi numeričnimi strategijami preišče prostor konfiguracij. Uporabnik lahko vnese kodo, ki se izvaja na CPE ali GPE in sam izbere parametre, ki jih želi nastaviti. Mittal in Vetter [5] tehnike razporejanja opravil v heterogenih sistemih razdelita glede na to, ali

je razporejanje dinamično ali statično, in glede na kriterij za razporejanje opravil. Jezik PetaBricks [6] je programski jezik, ki omogoča programerju, da našteje več izvedb iste metode, prevajalnik pa izbere najboljšo. Ansel et al. [7] in Pacula et al. [8] so raziskovali tudi dodajanje sprotnega učenja z metodami za reševanje problema večrokega bandita, a so se omejili na ogrodje jezika PetaBricks. To omejuje kompatibilnost z drugimi projekti. Poleg tega moramo pri metodi s hkratnim poganjanjem več izvedb in prekinitvijo počasnejše [7] med učenjem razpoloviti računska sredstva. Wen et al. [9] so se ukvarjali z razvrščanjem opravil OpenCL na CPE ali GPE glede na statične in dinamične lastnosti opravil. Statične lastnosti so nizkonivojske lastnosti opravila, na primer število različnih tipov operacij, in so znane pred samim zagonom. Dinamične lastnosti, na primer velikost vhodnih in izhodnih podatkov ter število vzporednih niti v opravilu, so znane šele med poganjanjem. Model podpornih vektorjev (SVM) razvršča opravila med tista, ki imajo na GPE visoko pohitritev, in ostala. Ker je naučen vnaprej, se slabše prilagaja na uporabnikovo strojno opremo.

Optimizacija programa glede na profil izvajanja (angl. profile-directed optimization) je pogosta funkcionalnost prevajalnikov za jezika C in C++, kot je GCC [10]. Program se prvič prevede tako, da med izvajanjem shranjuje podatke o samem izvajanju. Nato se program še enkrat prevede, prevajalnik pa uporabi prej shranjene podatke. Vendar optimizacije, ki jih delajo ti prevajalniki, ne izbirajo med več izvedbami algoritmov, ampak prevedejo izvorno kodo v čim bolj učinkovito strojno kodo [11]. Enak problem ima ponovno prevajanje pri jezikih, ki se iz vmesne kode prevedejo v strojno tik pred poganjanjem (angl. just-in-time compilation), nato pa spremljajo delovanje in se ponovno prevedejo z drugačnimi optimizacijami. Tu je prednost ta, da se lahko prevajanje zgodi večkrat, tudi med izvajanjem [12].

1.3 Prispevki

V nalogi bomo razvili sistem za izbiranje izvedbe dela programa, ki skrajša čas obdelave statističnega modela. Sistem bo sproti meril čas izvedbe in se iz meritev učil. Razvili ga bomo tako, da bo lahko uporabljal rezultate preteklega učenja, tudi če se bodo enako označeni odseki uporabili v drugih statističnih modelih. Naučen sistem bo prilagojen strojni opremi, na kateri se izvaja statistični model.

Drugi prispevek bo uporaba sistema v programski knjižnici Stan [2]. Statistični modeli knjižnice Stan so dober primer za uporabo, ker izvajanje poteka v veliko iteracijah in je sestavljeno iz omejenega nabora računsko intenzivnih funkcij.

1.4 Struktura dela

V naslednjem poglavju bomo pokazali, kako lahko problem izbire naprave, na kateri se bo del programa izvedel najhitreje, obravnavamo kot problem večrokega bandita. V tretjem poglavju bomo predstavili več odločitvenih metod, s katerimi lahko rešujemo ta problem, v četrtem poglavju pa programsko arhitekturo, ki omogoča enostavne spremembe delovanja odločitvenega sistema in nadgradnjo s programerjevim znanjem. V petem poglavju bomo opisali postopek in rezultate eksperimentalnega preizkusa odločitvenega sistema. V zadnjem poglavju bomo našteli sklepne ugotovitve dela.

Poglavje 2

Problem večrokega bandita

Problem izbire naprave za izvajanje funkcije lahko preslikamo v problem kontekstnega večrokega bandita. Problem se odvija v korakih $i = 1, 2, 3, \ldots I$. V vsakem koraku ima algoritem na izbiro diskretne možnosti $\mathcal{A}_i$. Pred izbiro vidi tudi vektor značilnic $\mathbf{x}_{i,a}$ za vsako izmed možnosti $a \in \mathcal{A}_i$, ki so na voljo v tem koraku. Glede na ta vektor in zgodovino prejšnjih opažanj izbere $a_i \in \mathcal{A}_i$ in prejme nagrado r_{i,a_i}. Opažanje $(\mathbf{x}_{i,a_i}, a_i, r_{i,a_i})$ nato uporabi za odločitvene metode izbire v sledečih korakih. Pomembna podrobnost je, da nikoli ne izvemo, kakšno nagrado bi dobili, če bi se odločili za katero izmed drugih možnosti. Cilj je minimizirati obžalovanje algoritma, kar definiramo kot

$$R(I) = \mathbb{E}\left[\sum_{i=1}^{I} r_{i,a_i^*}\right] - \mathbb{E}\left[\sum_{i=1}^{I} r_{i,a_i}\right] \quad , \tag{2.1}$$

kjer a_i^* predstavlja izbiro, s katero bi v koraku i dobili največjo nagrado. Gre za problem, kjer je treba uravnotežiti raziskovanje vseh možnosti in izrabljanje možnosti, za katero trenutno verjamemo, da ima najvišjo pričakovano nagrado. Zaradi tega ne izberemo vedno možnosti z najvišjo pričakovano vrednostjo, saj po enem vzorcu nagrade za vsako od možnosti še ne moremo zanesljivo določiti pričakovane vrednosti nagrade [13]. Zato metode, kot je LinUCB, optimistično izberejo možnost z višjo zgornjo mejo intervala zaupanja, metode, ki temeljijo na Thompsonovem vzorčenju, pa preizkušajo možnosti sorazmerno z verjetjem, da so optimalne [14].

V našem primeru sta diskretni možnosti vedno $\mathcal{A}_i = \{\text{CPE}, \text{GPE}\}$, a se da večino naših implementacij algoritmov z malo truda posplošiti na poljubno število možnosti. Namesto nagrad r_{i,a_i} imamo kazen t_{i,a_i}, ki je čas izvedbe na napravi a_i. Da zagotovimo pravilnost metod brez sprememb v implementaciji, čas negiramo, s čimer dobimo negativno nagrado

$$r_{i,a_i} = -t_{i,a_i} \quad . \tag{2.2}$$

Obžalovanje je v našem primeru

$$R(I) = \mathbb{E}\left[\sum_{i=1}^{I} t_{i,a_i}\right] - \mathbb{E}\left[\sum_{i=1}^{I} t_{i,a_i^*}\right] \quad . \tag{2.3}$$

Tako tudi metode, ki delujejo po principu napovedovanja zgornje meje intervala zaupanja delujejo nespremenjeno, saj neraziskanim možnostim napovejo manj negativno nagrado kot bolj optimistično. Podoben rezultat bi lahko dosegli, če bi namesto zgornje uporabili spodnjo mejo intervala zaupanja, a lahko tako prilagajanje pri bolj zapletenih metodah privede do napak v implementaciji.

Poglavje 3

Odločitvene metode

Zaradi omejitev problema in naših ciljev smo se osredotočili na nekatere metode, druge pa izpustili. Ker nikoli ne izvemo, koliko časa bi trajala izvedba na napravi, ki je nismo izbrali, ne moremo neposredno meriti in modelirati verjetnosti, da bo ena izmed možnosti imela krajši čas obdelave kot druga $\Pr(t_{i,a_x} < t_{i,a_y}), \{a_x, a_y\} \in \mathcal{A}_i$. Zaradi tega se nismo odločili za modeliranje z Bernoullijevo porazdelitvijo, ampak problem rešujemo tako, da skušamo napovedati čas obdelave na vsaki napravi.

Čeprav je ekstrapolacija lahko zelo nezanesljiva, želimo imeti dobro a priori znanje tudi za primerke, ki so večji od učnih. Če metoda domeno vektorja značilnic $\mathbf{x}_{i,a}$ razdeli na podprostore, kjer je napovedani čas obdelave konstantna vmesna vrednost časov v učni množici, na primer povprečje ali mediana, potem ta metoda ne bo nikoli napovedala časa obdelave, ki je večji od največje učne vrednosti v tem podprostoru. Med take metode spadajo tiste, ki temeljijo na navadnih odločitvenih drevesih ali histogramih. Take metode preliminarno ocenimo kot manj primerne za doseganje naših ciljev.

3.1 LinUCB z neodvisnimi modeli

Metodo LinUCB z neodvisnimi modeli so prvič predstavili Li et al. [13] in uporablja linearno regresijo, da določi zgornjo mejo intervala zaupanja za

pričakovano vrednost nagrade za neko možnost v koraku večrokega bandita. To razmerje se da opisati z neenačbo

$$|\mathbf{x}_{i,a}^{\intercal}\hat{\theta}_a - \mathbb{E}\left[r_{i,a}|\mathbf{x}_{i,a}\right]| \leq \alpha\sqrt{\mathbf{x}_{i,a}^{\intercal}\mathbf{A}_a^{-1}\mathbf{x}_{i,a}} \tag{3.1}$$

in enačbo

$$\mathbf{A}_a = \mathbf{X}_a^{\intercal}\mathbf{X}_a + \mathbf{I}_{d\times d} \quad , \tag{3.2}$$

kjer $\mathbf{x}_{i,a}$ predstavlja vektor značilnic za korak i in možnost a, $\hat{\theta}_a$ predstavlja naučene linearne koeficiente za možnost a, njun skalarni produkt ocenjeno pričakovano vrednost nagrade, $\mathbf{X}_a$ pa predstavlja matriko, katere vrstice so pretekli vektorji značilnic primerkov, ko je bila izbrana možnost a. Če velja $\alpha = 1 + \sqrt{\ln(2/\delta)/2}$, je verjetnost, da neenačba (3.1) drži, $1 - \delta$. To, da so modeli neodvisni, pomeni, da se za vsako možnost uči in uporablja povsem ločen model. Avtorji so poleg te različice metode razvili tudi tako, pri kateri je naučeno znanje med možnostmi deljeno [13]. To omogoča tudi obravnavo prej nevidenih možnosti. Zaradi konstantnega in majhnega števila možnosti smo se odločili za enostavnejšo metodo z neodvisnimi modeli. Algoritem lahko opišemo s psevdokodo 1.

Zgornja meja intervala zaupanja je namenoma optimistična, kar pomeni, da je večja od ocenjene pričakovane vrednosti. S tem metoda spodbudi raziskovanje možnosti.

3.2 Thompsonovo vzorčenje

Thompsonovo vzorčenje izbira možnosti z verjetnostjo, ki je enaka izračunanemu verjetju, da je ta možnost optimalna [15]. Možnost je optimalna, ko je njena pričakovana vrednost višja ali enaka od ostalih možnosti. Tako lahko zapišemo, da metoda izbere možnost a z verjetnostjo

$$\int \mathbb{I}\left[\mathbb{E}\left[r|a,\mathbf{x},\theta\right] = \max_{a'}\mathbb{E}\left[r|a',\mathbf{x},\theta\right]\right]\Pr(\theta|D)d\theta \quad , \tag{3.3}$$

kjer D predstavlja množico preteklih opažanj $(\mathbf{x}, a, r)$. Praktične implementacije namesto računanja integrala vzorčijo iz porazdelitve $\Pr(\theta|D)$, glede na

Psevdokoda 1 LinUCB z disjunktnimi modeli

Require: $\alpha \in \mathbb{R}_+$
for $i = 1..I$ **do**
  for all $a \in \mathcal{A}_i$ **do**
    $\mathbf{x}_{i,a} \leftarrow$ izračunaj vektor značilnic
    if a še ni bil viden **then**
      $\mathbf{A}_a \leftarrow \mathbf{I}_{d\times d}$
      $\mathbf{b}_a \leftarrow 0_{d\times 1}$
    end if
    $\hat{\theta}_a \leftarrow \mathbf{A}_a^{-1}\mathbf{b}_a$
    $p_{i,a} \leftarrow \hat{\theta}_a^\intercal \mathbf{x}_{i,a} + \alpha\sqrt{\mathbf{x}_{i,a}^\intercal \mathbf{A}_a^{-1}\mathbf{x}_{i,a}}$
  end for
  $a_i \leftarrow \arg\max_{a\in\mathcal{A}_i} p_{i,a}$
  $r_i \leftarrow$ prejmi nagrado za a_i
  $\mathbf{A}_{a_i} \leftarrow \mathbf{A}_{a_i} + \mathbf{x}_{i,a_i}\mathbf{x}_{i,a_i}^\intercal$
  $\mathbf{b}_{a_i} \leftarrow \mathbf{b}_{a_i} + r_i\mathbf{x}_{i,a_i}$
end for

to izračunajo pričakovano vrednost in izberejo možnost z najvišjo pričakovano vrednostjo. Postopek opisuje psevdokoda 2.

Psevdokoda 2 Thompsonovo vzorčenje

$D \leftarrow \emptyset$
for all $i \in 1..I$ **do**
  $\mathbf{x}_{i,a} \leftarrow$ izračunaj vektor značilnic
  $\theta_i \leftarrow$ vzorec iz $\Pr(\theta|D)$
  $a_i \leftarrow \arg\max_a \mathbb{E}\left[r|\mathbf{x}_i, a, \theta_i\right]$
  $r_i \leftarrow$ prejmi nagrado za a_i
  $D \leftarrow D \cup \{(\mathbf{x}_i, a_i, r_i)\}$
end for

3.2.1 Ločeni modeli za velikosti primerkov

S pregledom časov izvajanja smo se odločili pričakovano vrednost modelirati z normalno porazdelitvijo. Pri tem pristopu za vsak vektor značilnic $\mathbf{x}$ učimo ločen model. Kot a priori znanje uporabimo vnaprej naučene re-

gresijske funkcije. Za vsako možnost imamo dve funkciji, eno za pričakovano vrednost in eno za varianco. To je za naš primer smiselno, saj se v realnih primerih programa v jeziku Stan pojavi zelo malo različnih velikosti primerka. Nagrado tako modeliramo z normalno porazdelitvijo z znano varianco [16], kjer so podatki porazdeljeni z verjetnostjo

$$r|\mu, \sigma^2 \sim \mathcal{N}(\mu, \sigma^2) \quad , \tag{3.4}$$

a priori verjetnost pričakovane vrednosti je

$$\mu|\sigma^2 \sim \mathcal{N}(\mu_0, \sigma_0^2) \quad , \tag{3.5}$$

posteriorna verjetnost pričakovane vrednosti pa

$$\mu|D, \sigma^2 \sim \mathcal{N}\left(\left(\mu_0 \frac{1}{\sigma_0^2} + \overline{\mathbf{r}} \frac{n}{\sigma_{\mathbf{r}}^2}\right)\left(\frac{1}{\sigma_0^2} + \frac{n}{\sigma_{\mathbf{r}}^2}\right)^{-1}, \left(\frac{1}{\sigma_0^2} + \frac{n}{\sigma_{\mathbf{r}}^2}\right)^{-1}\right) \quad , \tag{3.6}$$

kjer je μ_0 a priori pričakovana vrednost nagrade, σ_0^2 a priori varianca nagrade, n število preteklih korakov, v katerih smo izbrali to možnost. $\overline{\mathbf{r}}$ je povprečje nagrad v teh korakih, $\sigma_{\mathbf{r}}^2$ pa varianca nagrad v teh korakih. Pričakovana vrednost nagrade je za neko možnost v nekem koraku

$$\mathbb{E}\left[r|a, \mathbf{x}, \theta\right] = \mu \quad . \tag{3.7}$$

Ob vsakem koraku za vsako možnost izračunamo a priori pričakovano vrednost

$$\mu_0 = \text{priorMeanFunc}_a(\mathbf{x}) \tag{3.8}$$

in a priori varianco

$$\sigma_0^2 = \text{priorVarFunc}_a(\mathbf{x}) \quad . \tag{3.9}$$

priorMeanFunc_a in priorVarFunc_a sta poljubni funkciji za določanje a priori parametrov za možnost a. Z njima uporabimo predhodno znanje o porazdelitvah nagrad. Vsebujeta lahko tudi napovedni model. Če ne želimo uporabiti nobenega predhodnega znanja, definiramo priorVarFunc_a tako, da vedno vrača zelo visoko vrednost, s čimer naredimo a priori porazdelitev neinformativno. Pri našem testiranju smo za vsako možnost a na učni množici

primerkov naučili linearni regresijski model za napovedovanje pričakovane a priori vrednosti μ_0 glede na $\mathbf{x}$. Na napakah tega modela smo naučili tudi model za napovedovanje a priori variance σ_0^2 glede na $\mathbf{x}$.

Naj bo $\mathbf{r} \in \mathbb{R}^n$ vektor nagrad v zgodovini D, kjer so bile odločitve enake a, vektor značilnic pa $\mathbf{x}_i$:

$$\forall r : (\mathbf{x}_i, a, r) \in D \Rightarrow r \in \mathbf{r} \quad . \tag{3.10}$$

Te nagrade smatramo kot vzorec iz porazdelitve nagrad in izračunamo povprečno vrednost vzorca

$$\overline{\mathbf{r}} = \frac{\sum_{r \in \mathbf{r}} r}{n} \tag{3.11}$$

in varianco vzorca

$$\sigma_{\mathbf{r}}^2 = \frac{\sum_{r \in \mathbf{r}} (r - \overline{\mathbf{r}})}{n-1} \quad . \tag{3.12}$$

Pričakovano vrednost nagrade za možnost a v koraku i potem določimo z vzorcem iz porazdelitve v enačbi (3.6). Izbrana možnost v tem koraku je tista, za katero je ta vzorec največji. Celoten postopek je zapisan s psevdokodo 3.

Psevdokoda 3 Thompsonovo vzorčenje z ločenimi modeli

Require: $\forall i \in 1..I, \forall a \in \mathcal{A}_i : \text{priorMeanFunc}_a, \text{priorVarFunc}_a$

$D \leftarrow \emptyset$
for all $i \in 1..I$ **do**
  for all $a \in \mathcal{A}_i$ **do**
    $\mathbf{x}_{i,a} \leftarrow$ izračunaj vektor značilnic
    $\mu_0 \leftarrow \text{priorMeanFunc}_a(\mathbf{x})$
    $\sigma_0^2 \leftarrow \text{priorVarFunc}_a(\mathbf{x})$
    $\overline{\mathbf{r}} \leftarrow \frac{\sum_{r \in \mathbf{r}} r}{n}$
    $\sigma_{\mathbf{r}}^2 \leftarrow \frac{\sum_{r \in \mathbf{r}} (r - \overline{\mathbf{r}})}{n-1}$
    $\mu_a \leftarrow$ vzorec iz $\mathcal{N}\left(\left(\mu_0 \frac{1}{\sigma_0^2} + \overline{\mathbf{r}} \frac{n}{\sigma_{\mathbf{r}}^2}\right)\left(\frac{1}{\sigma_0^2} + \frac{n}{\sigma_{\mathbf{r}}^2}\right)^{-1}, \left(\frac{1}{\sigma_0^2} + \frac{n}{\sigma_{\mathbf{r}}^2}\right)^{-1}\right)$
  end for
  $a_i \leftarrow \arg\max_a \mu_a$
  $r_i \leftarrow$ prejmi nagrado za a_i
  $D \leftarrow D \cup \{(\mathbf{x}_i, a_i, r_i)\}$
end for

3.2.2 Pristop z Bayesovo regresijo

Porazdelitev nagrade za vsako možnost lahko modeliramo z Bayesovo regresijo [16]. To nam omogoča, da ves čas uporabljamo enako učenje, brez ločenega modela za predhodno učenje. Podatki so porazdeljeni z verjetnostjo

$$r|\mathbf{b}, \sigma^2 \sim \mathcal{N}(\mathbf{b}^\intercal \mathbf{x}, \sigma^2) \quad , \tag{3.13}$$

a priori pričakovana vrednost je

$$\mathbf{b} \sim \mathcal{N}(\mu_0, \mathbf{\Lambda}_0) \quad , \tag{3.14}$$

a priori varianca pa

$$\sigma^2 \sim \mathcal{IG}(\alpha_0, \beta_0) \quad . \tag{3.15}$$

Matrika $\mathbf{\Lambda}$ predstavlja matriko točnosti naključnih spremenljivk, ki sestavljajo vektor značilnic, in je definirana kot inverz kovariančne matrike

$$\mathbf{\Lambda} = \mathbf{\Sigma}^{-1} = (\mathbb{E}\left[\mathbf{C}\mathbf{C}^\intercal\right] - \mu_\mathbf{C}\mu_\mathbf{C}^\intercal)^{-1} \quad , \tag{3.16}$$

$$\mathbf{C} = (X_1, \ldots, X_m)^\intercal \quad , \tag{3.17}$$

$$\mu_\mathbf{C} = (\mathbb{E}\left[X_1\right], \ldots, \mathbb{E}\left[X_m\right])^\intercal \quad , \tag{3.18}$$

kjer X_j predstavlja porazdelitev j-tega elementa vektorja značilnic $\mathbf{x}_{i,a}$ za vse korake i, kjer je bila izbrana možnost a. $\mathbf{\Lambda}_0$ je a priori vrednost matrike $\mathbf{\Lambda}$. $\mathcal{IG}$ predstavlja inverzno gama porazdelitev, parametra α in β pa obliko in merilo te porazdelitve. Parametra α_0 in β_0 sta njuni a priori vrednosti teh parametrov. Posteriorna pričakovana vrednost je porazdeljena po

$$\mathbf{b}|\mathbf{r}, \mathbf{X}_a, \sigma^2 \sim \mathcal{N}(\mu, \sigma^2\mathbf{\Lambda}^{-1}) \quad , \tag{3.19}$$

posteriorna varianca pa je porazdeljena po

$$\sigma^2|\mathbf{r}, \mathbf{X}_a \sim \mathcal{IG}(\alpha, \beta) \quad . \tag{3.20}$$

Vrednosti parametrov v enačbah (3.19) in (3.20) lahko izračunamo po enačbah

$$\mathbf{\Lambda} = \mathbf{X}_a^\intercal \mathbf{X}_a + \mathbf{\Lambda}_0 \quad , \tag{3.21}$$

$$\mu = \mathbf{\Lambda}^{-1}(\mathbf{\Lambda}_0 \mu_0 + \mathbf{X}_a^\intercal \mathbf{r}) \quad , \tag{3.22}$$

$$\alpha = \alpha_0 + \frac{|\mathbf{r}|}{2} \quad , \tag{3.23}$$

$$\beta = \beta_0 + \frac{1}{2}(\mathbf{r}^\intercal \mathbf{r} + \mu_0^\intercal \mathbf{\Lambda}_0 \mu_0 - \mu^\intercal \mathbf{\Lambda} \mu) \quad . \tag{3.24}$$

Pričakovana vrednost nagrade je za neko možnost v nekem koraku

$$\mathbb{E}\left[r|a, \mathbf{x}, \theta\right] = \mathbf{b}^\intercal \mathbf{x} \quad . \tag{3.25}$$

Pričakovano vrednost dobimo tako, da najprej izračunamo parametre porazdelitev po enačbah od (3.21) do (3.24), potem vzorčimo najprej po (3.20) in nato po (3.19), potem pa še izračunamo pričakovano vrednost po enačbi (3.25). Celoten postopek odločanja je zapisan v psevdokodi 4.

Psevdokoda 4 Thompsonovo vzorčenje z Bayesovo regresijo

Require: $\mu_0, \mathbf{\Lambda}_0, \alpha_0, \beta_0$

$D \leftarrow \emptyset$
for all $i \in 1..I$ **do**
  for all $a \in \mathcal{A}_i$ **do**
    $\mathbf{x}_{i,a} \leftarrow$ izračunaj vektor značilnic
    $\mathbf{\Lambda} \leftarrow \mathbf{X}_a^\intercal \mathbf{X}_a + \mathbf{\Lambda}_0$
    $\mu \leftarrow \mathbf{\Lambda}^{-1}(\mathbf{\Lambda}_0 \mu_0 + \mathbf{X}_a^\intercal \mathbf{r})$
    $\alpha \leftarrow \alpha_0 + \frac{|\mathbf{r}|}{2}$
    $\beta \leftarrow \beta_0 + \frac{1}{2}(\mathbf{r}^\intercal \mathbf{r} + \mu_0^\intercal \mathbf{\Lambda}_0 \mu_0 - \mu^\intercal \mathbf{\Lambda} \mu)$
    $\sigma^2 \leftarrow$ vzorec iz $\mathcal{IG}(\alpha, \beta)$
    $\mathbf{b} \leftarrow$ vzorec iz $\mathcal{N}(\mu, \sigma^2 \mathbf{\Lambda}^{-1})$
    $\mathbb{E}_a \leftarrow \mathbf{b}^\intercal \mathbf{x}$
  end for
  $a_i \leftarrow \arg\max_a \mathbb{E}_a$
  $r_i \leftarrow$ prejmi nagrado za a_i
  $D \leftarrow D \cup \{(\mathbf{x}_i, a_i, r_i)\}$
end for

Poglavje 4

Programska arhitektura

Pri načrtovanju programske arhitekture sledimo načelu, da s privzetimi metodami zagotovimo zadovoljivo delovanje, hkrati pa damo programerju veliko možnosti, da s svojim znanjem delovanje dodatno izboljša. To dosežemo s programskim vzorcem strategij, kjer za neko funkcionalnost obstaja več izvedb. To omogoča, da se izvedbo algoritma v strategiji po potrebi zamenja [17]. Strategije so lahko funkcije ali pa objekti, če je potrebno kompleksnejše delovanje. Primer tega je večkratna uporaba s hranjenjem stanja. Programer lahko strategije poda v parametrih konstruktorjev in funkcij. Ta koncept je v objektno usmerjenem programiranju pogosto uporabljen skupaj s skupnim objektom, ki vsem odjemalcem priskrbi storitve, ki so bile prej registrirane [18]. V funkcijskem programiranju je soroden koncept funkcija višjega reda. Taka funkcija preko svojih parametrov sprejme neko drugo funkcijo ali pa vrne neko drugo funkcijo kot rezultat.

Primerek nekega problema je ena kombinacija vhodnih podatkov v domeni tega računskega problema. Velikost primerka je njegova lastnost, neodvisna od odločitvenih metod, značilnice primerka pa so prilagoditve tega za neko odločitveno metodo. Ni povsem nujno, da ločimo funkciji za izračun velikosti in značilnic. Funkcija za velikost je lahko tudi identiteta, njeno vlogo pa povsem prevzame funkcija za izračun značilnic. A ločitev teh dveh tipov funkcij nam omogoča, da bolje opredelimo, kaj je pomembno za opis

primerka in kaj so prilagoditve za neko odločitveno metodo. Če imamo več odločitvenih metod, se s tem lahko tudi izognemo ponavljanju kode ali večkratnem računanju.

4.1 Velikost primerka

Velikost primerka je določena iz vhodnih podatkov računskega problema in predstavlja povzetek, ki omogoča čim boljšo napoved časa izvajanja. Lahko je katerikoli podatkovni tip, ki ga bodo sprejele funkcije za izračun značilnic problema. Obliko funkcije za določanje velikosti primerka za neki računski problem lahko zapišemo kot:

$$\text{instanceSizeFunc}\colon \text{Args} \to \text{InstanceSize} \quad . \tag{4.1}$$

Privzeta izvedba deluje tako, da vrne vektor celih števil, ki vsebuje vse dimenzije matrik in vsa cela števila v parametrih klica. Tako v primeru funkcije matričnega množenja

$$\text{multiply} : \mathbb{R}^{m\times n} \times \mathbb{R}^{n\times p} \to \mathbb{R}^{m\times p} \tag{4.2}$$

privzeto uporabimo funkcijo $(\mathbf{A} \in \mathbb{R}^{m\times n}, \mathbf{B} \in \mathbb{R}^{n\times p}) \mapsto (m, n, n, p)$. Programerju damo priložnost, da vnese svojo funkcijo za izračun velikosti primerka. Če ve, da imata matriki eno dimenzijo enako, lahko namesto privzete funkcije uporabi $(\mathbf{A}, \mathbf{B}) \mapsto (m, n, p)$ in s tem poenostavi učenje. Lahko vnese celo izračunano asimptotično časovno kompleksnost algoritma, doda gostoto matrik ali pa katerokoli drugo vrednost, ki jo izpelje iz vhodov.

4.2 Značilnice primerka

Dodatno lahko velikost primerka bolj prilagodimo izbrani odločitveni metodi s spremembo v značilnice primerka. Namen je prilagoditi podatkovne tipe za odločitveno metodo in olajšati njene neželene lastnosti, kot je linearnost ali težave s prevelikimi razlikami v standardnih odklonih različnih parametrov.

Tudi značilnice primerka nimajo določene oblike, ampak jim obliko določajo izbrane odločitvene metode. Veliko metodam ustreza vektor realnih števil.

Primer funkcije za izračun značilnic primerka je multivariatni polinom, kjer se v vsakem členu spremenljivka pojavi v največ d-ti stopnji. To lahko formaliziramo s funkcijo v psevdokodi 5.

Psevdokoda 5 Značilnice kot polinom velikosti primerka

Require: instanceSize $\in \mathbb{R}^n, d \in \mathbb{Z}^+$
1: result $\leftarrow$ [1]
2: **for all** $s \in$ instanceSize **do**
3: **for all** $r \in$ result **do**
4: **for all** degree $\in 1..d$ **do**
5: result.append$(r \cdot s^{\text{degree}})$
6: **end for**
7: **end for**
8: **end for**

4.3 Odločitvene metode

Vse odločitvene metode so izvedene kot razredi, ki imajo skupen osnovni razred. V konstruktorju so lahko dodatni parametri in funkcije za izračun značilnic primerka. Osnovni razred vseh odločitvenih metod ima dve javni metodi. Prva je metoda za izbiro, ki izbere, na kateri napravi naj se izvede primerek. To metodo lahko opišemo kot

$$\text{strategy.choose}\colon \text{InstanceSize} \to \text{Device} \quad . \tag{4.3}$$

Metoda ni čista funkcija, saj lahko uporablja zgodovino, ki je shranjena v poljih objekta. Poleg tega lahko celo spreminja notranje stanje, na primer z jemanjem vzorcev iz generatorja psevdonaključnih števil, ki je shranjen v poljih te odločitvene metode. S tem se spremeni notranje stanje generatorja psevdonaključnih števil.

Druga metoda je namenjena učenju napovednega modela in je oblike

$$\text{strategy.update}\colon (\text{InstanceSize} \times \text{Device} \times \text{Duration}) \to \{\} \quad . \tag{4.4}$$

Namenjena je temu, da v njej programer odločitvene metode določi, kako se posodobi stanje modela z novimi podatki. Primer take posodobitve je dodajanje podatkov v seznam, ki se pregleda ob klicu strategy.choose, ali sprememba koeficientov regresije. Programer lahko implementacijo te metode tudi izpusti, če modela ne želi učiti sproti. Pričakovano je, da izvedbe te metode ne bodo čiste funkcije, saj bodo spremenile notranje stanje objekta, ki predstavlja odločitveno metodo.

4.4 Funkcije z izbiro

Osnovne funkcije za naprave zberemo v podatkovno strukturo, ki nam omogoča iskanje po napravi:

$$\text{funcs}\colon \text{Device} \to (\text{Args} \to \text{Result}) \quad . \tag{4.5}$$

V naši izvedbi za naprave določimo vrstni red, ki se uporabi tudi za seznam osnovnih funkcij. Osnovne funkcije lahko nato združimo v funkcijo z izbiro, ki glede na odločitveno metodo in velikost problema določi, katera funkcija se bo izvedla, nato pa jo izvede s podanimi argumenti. Funkcijo z izbiro lahko formaliziramo kot

$$\text{choiceFunc}\colon (\text{Strategy} \times \text{InstanceSize} \times \text{Args}) \to \text{Result} \quad , \tag{4.6}$$

ki deluje tako, da najprej uporabi odločitveno metodo, da izbere osnovno funkcijo, ki se bo izvedla. Nato jo izvede s podanimi argumenti in izmeri čas izvajanja. Metodo za odločanje posodobi z znanjem, ki vsebuje čas izvajanja, podano velikost primerka in možnost, ki jo je metoda za odločanje izbrala. Nato vrne rezultat izvedbe izbrane funkcije. Delovanje prikazuje tudi psevdokoda 6, izvedbo v jeziku C++ pa dodatek A.

S pomočjo funkcije

$$\text{getChoiceFunc}\colon \text{funcs} \to \text{choiceFunc} \tag{4.7}$$

ustvarimo funkcijo z izbiro choiceFunc, ki vključuje funkcije v zbirki osnovnih funkcij funcs. Osnovne funkcije funcs ostanejo ob vsakem klicu funk-

cije z izbiro choiceFunc enake. Za to smo se odločili, ker je v našem primeru, pri katerem vsaka osnovna funkcija predstavlja izvajanje na eni od naprav računalnika, scenarij, kjer bi se zaloga funkcij spreminjala med izvedbo programa, zelo malo verjeten. Poleg tega je za zagotavljanje pravilnosti odločitvenih metod potrebno ob vsaki spremembi osnovnih funkcij spremeniti ali zamenjati tudi odločitveno metodo, saj naučeni modeli znotraj odločitvene metode mogoče niso več pravilni zaradi drugačnih časov izvajanja.

Primer uporabe za funkcijo matričnega množenja, kjer sta matriki **A** in **B** faktorja, matrika **C** pa produkt, je viden v psevdokodi 7.

Psevdokoda 6 Delovanje funkcije z izbiro

Require: funcs : Device → Args → Result

```
1:  function choiceFunc(strategy, instanceSize, args)
2:      choice ← strategy.choose(instanceSize)
3:      t0 ← trenutni čas
4:      result ← funcs[choice](args)
5:      t1 ← trenutni čas
6:      duration ← t1 − t0
7:      reward ← −duration
8:      strategy.update(instanceSize, choice, reward)
9:      return result
10: end function
```

Psevdokoda 7 Uporaba funkcije z izbiro

Require: funcs, strategy, instanceSizeFunc, **A**, **B**

```
1: choiceFunc ← getChoiceFunc(funcs)
2: C ← choiceFunc(strategy, instanceSizeFunc(A, B), A, B)
```

Če želimo med potekom programa uporabljati isto odločitveno metodo in določati velikost primerka problema na enak način, lahko dobimo poenostavljeno funkcijo z izbiro:

$$\text{simpleChoiceFunc} : \text{Args} \to \text{Result} \quad , \tag{4.8}$$

ki deluje, kot prikazuje psevdokoda 8. Pomagamo si lahko s funkcijo

$$\text{getSimpleChoiceFunc}\colon (\text{choiceFunc} \times \text{Strategy} \times \text{instanceSizeFunc}) \rightarrow \text{simpleChoiceFunc} \quad , \quad (4.9)$$

ki ustvari novo funkcijo simpleChoiceFunc, ki vključuje spremenljivke choiceFunc, Strategy in instanceSizeFunc. Funkcija tako uporablja rezultat funkcije getChoiceFunc. Ločitev postopka na getChoiceFunc in getSimpleChoiceFunc naredi uporabo bolj modularno, saj je funkcija z izbiro lahko zaradi tega sestavljena tudi drugače, kot z metodo getChoiceFunc. Rezultat getSimpleChoiceFunc je funkcija, ki ima enake tipe vhodov in izhodov, kot jih imajo osnovne funkcije funcs. Njeno uporabo za primer matričnega množenja, kjer sta matriki **A** in **B** faktorja, matrika **C** pa produkt, prikazuje psevdokoda 9. Implementacija getSimpleChoiceFunc v jeziku C++ je v dodatku A.

Psevdokoda 8 Delovanje poenostavljene funkcije z izbiro

Require: choiceFunc, strategy, instanceSizeFunc

```
1: function simpleChoiceFunc(args)
2:     return choiceFunc(strategy, instanceSizeFunc(args), args)
3: end function
```

Psevdokoda 9 Uporaba poenostavljene funkcije z izbiro

Require: choiceFunc, strategy, instanceSizeFunc, **A**, **B**

```
1: simpleChoiceFunc ← getSimpleChoiceFunc(choiceFunc, strategy, instanceSizeFunc)
2: C ← simpleChoiceFunc(A, B)
```

4.5 Uporaba v statističnem modelu

Poenostavljena funkcija z izbiro lahko enostavno nadomesti funkcijo, ki je uporabljena v kodi, ki jo proizvede orodje Stanc. To lahko dosežemo tudi brez spremembe klicev funkcij tako, da definiramo funkcijo z enakim imenom v kontekstu s prednostjo pred tistim, v katerem je bila definirana prej

uporabljana funkcija. Če definiramo nove funkcije kot polja objekta, ki predstavlja statistični model, ali v formalnih parametrih klicev drugih funkcij ali metod, potem nova funkcija maskira staro. S tako uporabo funkcij višjega reda omogočimo možnost sprememb delovanja, kar ima tudi širšo uporabnost. Staro funkcijo lahko definiramo kot privzeto vrednost formalnega parametra in tako ohranimo enako obnašanje starih programov, ki ne podajo svojih implementacij teh funkcij. Razlike v kodi modela, ko v njem uporabimo poenostavljene funkcije z izbiro, so vidne v dodatku B.

4.6 Primer postopka uporabe

Celoten primer uporabe za množenje matrik je prikazan v psevdokodi 10. Najprej različne postopke za množenje matrik definiramo tako, da imajo obliko funkcij z enakimi tipi vhodnih in izhodnih podatkov (vrstice 1-10). Te funkcije združimo v skupno podatkovno strukturo, seznam (vrstica 14). Nato lahko iz seznama s funkcijo getChoiceFunc dobimo funkcijo z izbiro, ki vsebuje te osnovne funkcije in ob svoji izvedbi izvede eno izmed njih (vrstica 15). Odločitveno metodo sestavimo s konstruktorjem njenega razreda, ki pa je nekoliko drugačen za vsako odločitveno metodo (vrstica 16). Če nam privzeta funkcija za izračun velikosti primerka ne ustreza, lahko definiramo novo, ki pa mora imeti enake vhodne tipe kot osnovne funkcije (vrstice 11-13). Funkcijo z izbiro, odločitveno metodo in, če želimo, funkcijo za izračun velikosti uporabimo v funkciji getSimpleChoiceFunc, da dobimo poenostavljeno funkcijo z izbiro (vrstica 17). To lahko nato uporabimo, kot bi uporabili katerokoli izmed osnovnih funkcij za množenje matrik (vrstica 18). Vsakič, ko jo kličemo, ta z odločitveno metodo izbere eno od osnovnih funkcij, jo izvede in vrne njen rezultat. Poenostavljeno funkcijo z izbiro lahko uporabimo tudi večkrat, kar hkrati uči odločitveno metodo.

Psevdokoda 10 Primer postopka uporabe za matrično množenje

Require: $\mathbf{A} \in \mathbb{R}^{m \times n}, \mathbf{B} \in \mathbb{R}^{n \times p}$

function multiplyCpu($\mathbf{M1} \in \mathbb{R}^{m \times n}, \mathbf{M2} \in \mathbb{R}^{n \times p}$)
    return $\mathbf{M1} \cdot \mathbf{M2}$
end function

function multiplyGpu($\mathbf{M1} \in \mathbb{R}^{m \times n}, \mathbf{M2} \in \mathbb{R}^{n \times p}$)
    $\mathbf{M1}_{\text{gpu}} \leftarrow \mathbf{M1}$ prekopiramo v pomnilnik GPE
    $\mathbf{M2}_{\text{gpu}} \leftarrow \mathbf{M2}$ prekopiramo v pomnilnik GPE
    $\mathbf{Y}_{\text{gpu}} \leftarrow \mathbf{M1}_{\text{gpu}} \cdot \mathbf{M2}_{\text{gpu}}$ // Zmnožimo na GPE.
    $\mathbf{Y} \leftarrow \mathbf{Y}_{\text{gpu}}$ prekopiramo nazaj v glavni pomnilnik
    return $\mathbf{Y}$
end function

function instanceSizeFunc($\mathbf{M1} \in \mathbb{R}^{m \times n}, \mathbf{M2} \in \mathbb{R}^{n \times p}$)
    return (m, n, p)
end function

funcs $\leftarrow$ [multiplyCpu, multiplyGpu]
choiceFunc $\leftarrow$ getChoiceFunc(funcs)
strategy $\leftarrow$ sestavimo nov objekt želene odločitvene metode
simpleChoiceFunc $\leftarrow$ getSimpleChoiceFunc(choiceFunc, strategy, instanceSizeFunc)
// simpleChoiceFunc lahko uporabimo večkrat.
$\mathbf{C} \leftarrow$ simpleChoiceFunc($\mathbf{A}, \mathbf{B}$)

Poglavje 5

Rezultati

V tem poglavju bomo preizkusili, kako se odločitvene metode obnašajo za reševanje samostojnih računskih problemov in kako vplivajo na vzorčenje s praktičnim primerom statističnega modela.

Strategija je izvedba odločitvene metode. V tem poglavju preizkušamo sledeče strategije:

- strategijo CPE, ki vedno izbere CPE,
- strategijo GPE, ki vedno izbere GPE,
- strategijo LinUCB, ki uporablja odločitveno metodo LinUCB,
- strategijo LinUCB*, ki uporablja odločitveno metodo LinUCB in je bila učena že pred preizkusom,
- strategijo TL, ki uporablja Thompsonovo vzorčenje z ločenimi modeli,
- strategijo TL*, ki uporablja Thompsonovo vzorčenje z ločenimi modeli in je bila učena že pred preizkusom,
- strategijo TR, ki uporablja Thompsonovo vzorčenje z Bayesovo regresijo,
- strategijo TR*, ki uporablja Thompsonovo vzorčenje z Bayesovo regresijo in je bila učena že pred preizkusom.

Za vsako strategijo smo vedno uporabili enake parametre, da jih ne bi s tem preveč prilagodili posameznemu preizkusu. Izjema je strategija TL*, kjer se a priori vrednosti določijo kot del predhodnega učenja. Za strategiji LinUCB in LinUCB* smo uporabili parameter $\alpha = 10^{-5}$. Za strategijo TL smo uporabili a priori vrednosti $\mu = 0, \sigma^2 = 100$. Za strategiji TR in TR* smo uporabili a priori vrednosti $\Lambda_0 = \mathbf{I}_{d\times d} * 1000, \mu_0 = 0_d, \alpha_0 = 10^{-2}, \beta_0 = 10^{-2}$, kjer je d velikost vektorja značilnic primerka.

5.1 Računski problemi

Minimizirati želimo obžalovanje, kot je definirano v enačbi (2.3). Ker je čas obdelave močno odvisen od velikosti primerka, ga naredimo primerljivega med primerki tako, da ga delimo z optimalnim časom trajanja za tisti primerek. Času izvajanja prištejemo tudi čas odločanja in čas posodabljanja modela. Optimalni čas izvajanja je čas izvajanja na najhitrejši napravi in nima dodanega časa izbire ali učenja modela. Pričakovano vrednost razmerja v koraku I računamo tako, da generiramo 20 naključnih zaporedij primerkov. Vzorci za korak I so tako relativna obžalovanja v koraku I v več zaporedjih. Ker vzorčimo tako, da generiramo več naključnih zaporedij, ne vzorčimo tudi na nivoju samih primerkov. Zato kot pričakovano vrednost vsote časa uporabimo kar zaporedje izmerjenih časov. Naša metrika za korak I je

$$\mathbb{E}\left[R_{\text{rel}}(I)\right] = \mathbb{E}\left[\frac{\sum_{i=1}^{I}(t_{i,a_i} + t_{\text{choose}\,i} + t_{\text{update}\,i})}{\sum_{i=1}^{I} t_{i,a_i^*}}\right] - 1 \quad . \tag{5.1}$$

Postopek priprave podatkov je dodatno opisan v psevdokodi 11. Porazdelitve primerkov v preizkusih so bile kompromis med željo, da analiziramo velik problemski prostor, in omejitvijo, ki je hitro rastoči čas reševanja nekaterih problemov.

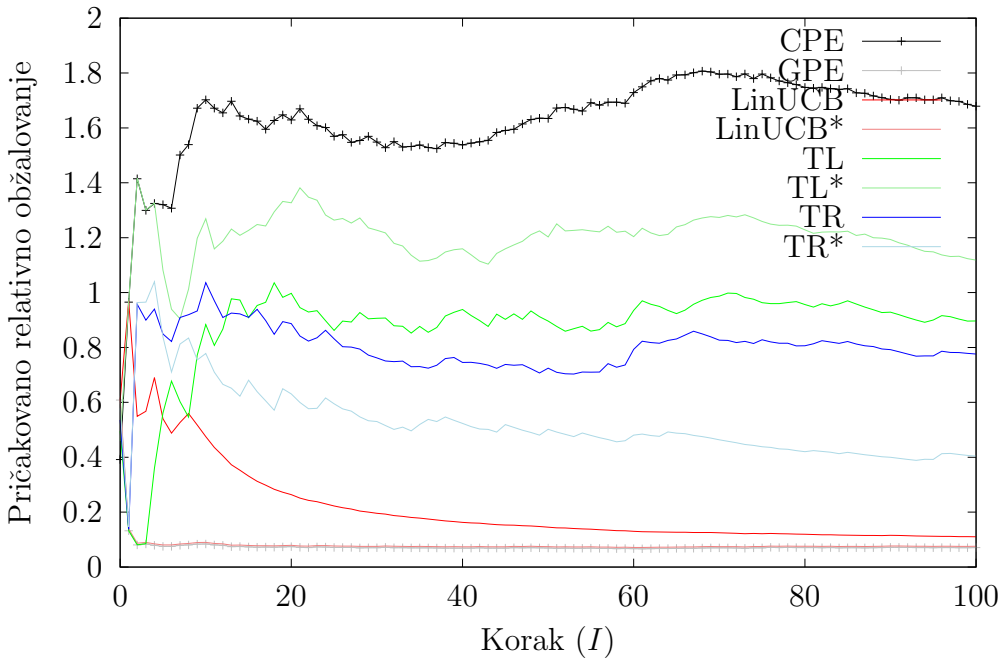
5.1.1 Množenje matrik

Slika 5.1 prikazuje pričakovano relativno obžalovanje po I korakih za problem množenja matrik. Vsak primerek je par kvadratnih matrik, kjer je dimen-

Psevdokoda 11 Izračun relativnega obžalovanja

Require: strategy $\in$ Strategy, series $\in \mathbb{N}_+, I \in \mathbb{N}_+, t\colon (\mathbb{N}_+ \times \mathbb{N}_+ \times \text{Device}) \to \mathbb{R}$

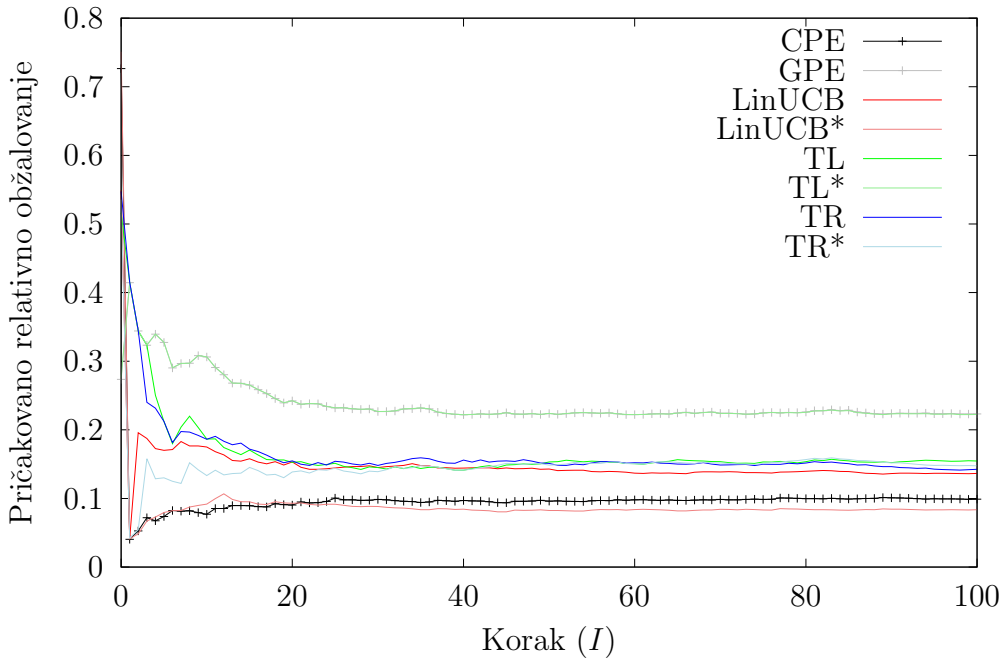
$\text{sum}_t \leftarrow 0^{\text{Strategy} \times \text{series} \times I}$
for all strategy $\in$ Strategy **do**
  for all serie $\in$ [1..series] **do**
    // Ponastavimo učenje odločitvene metode, da so rezultati zaporedij
    // primerkov neodvisni.
    reset(strategy)
    for all $i \in [1..I]$ **do**
      t0 $\leftarrow$ trenutni čas
      choice $\leftarrow$ strategy.choose(instanceSize)
      strategy.update(instanceSize, choice, t(serie, i, choice))
      t1 $\leftarrow$ trenutni čas
      $\Delta t \leftarrow t(\text{serie}, i, \text{choice}) + \text{t1} - \text{t0}$
      if $i = 1$ **then**
        $\text{sum}_t[\text{strategy}, \text{serie}, i] \leftarrow \Delta t$
      else
        $\text{sum}_t[\text{strategy}, \text{serie}, i] \leftarrow \Delta t + \text{sum}_t[\text{strategy}, \text{serie}, i-1]$
      end if
    end for
  end for
end for
$\text{sum}_t^* \leftarrow 0^{\text{series} \times I}$
for all serie $\in$ [1..series] **do**
  for all $i \in [1..I]$ **do**
    $\Delta t \leftarrow \min_{d \in \text{Device}} t(\text{serie}, i, d)$
    if $i = 1$ **then**
      $\text{sum}_t^*[\text{serie}, i] \leftarrow \Delta t$
    else
      $\text{sum}_t^*[\text{serie}, i] \leftarrow \Delta t + \text{sum}_t^*[\text{serie}, i-1]$
    end if
  end for
end for
$R_{\text{rel}} \leftarrow 0^{\text{Strategy} \times I}$
for all strategy $\in$ Strategy **do**
  for all $i \in [1..I]$ **do**
    $R_{\text{rel}}[\text{strategy}, i] \leftarrow \text{mean}_{\text{serie} \in [1..\text{series}]} \left(\frac{\text{sum}_t[\text{strategy}, \text{serie}, i]}{\text{sum}_t^*[\text{serie}, i]} - 1 \right)$
  end for
end for



Slika 5.1: Relativno obžalovanje pri množenju matrik.

zija vzorec iz diskretne enakomerne porazdelitve v intervalu $[150, 250]$. Učni primerki so uporabljali interval $[1, 200]$.

Strategija GPE ima v vseh korakih, razen v prvem, nižje pričakovano relativno obžalovanje kot strategija CPE. Skoraj enako pričakovano relativno obžalovanje ima strategija LinUCB*. Strategija LinUCB ima v prvih korakih nekoliko višje pričakovano relativno obžalovanje, potem pa ta pada. Strategija TL ima pričakovano relativno obžalovanje vmes med vrednostma strategij CPE in GPE. Strategija TL* ima še nekoliko slabši rezultat. Nekoliko boljšega ima strategija TR, ampak dolgoročni trend ni razviden. Strategija TR* ima nižjo pričakovano relativno obžalovanje kot strategija TR. To se s koraki niža, a počasneje kot pri strategiji LinUCB in veliko počasneje kot pri strategiji LinUCB*.

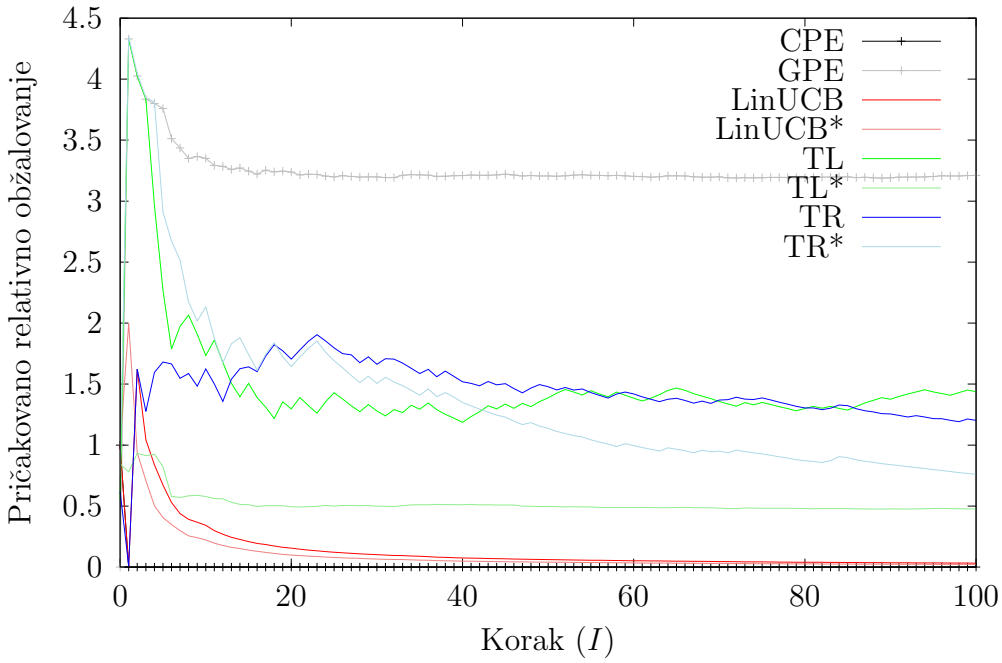


Slika 5.2: Relativno obžalovanje pri razcepu Choleskega.

5.1.2 Razcep Choleskega

Slika 5.2 prikazuje pričakovano relativno obžalovanje po I korakih za problem razcepa Choleskega. Vsak primerek je pozitivno definitna matrika, kjer je dimenzija vzorec iz diskretne enakomerne porazdelitve v intervalu $[275, 700]$. Učni primerki so uporabljali interval $[1, 200]$.

Strategija CPE ima v vseh korakih, razen v prvem, nižje pričakovano relativno obžalovanje kot strategija GPE. Strategija LinUCB* ima še nekoliko nižjo vrednost in je v tem primeru najboljša izmed strategij. Strategije LinUCB, TL, TR, TR* imajo podobno pričakovano relativno obžalovanje, ki pa je le vmesna vrednost med vrednostmi za strategiji CPE in GPE. Strategija TL* ima enako pričakovano relativno obžalovanje kot strategija GPE. Te strategiji sta najpočasnejši za ta primer.



Slika 5.3: Relativno obžalovanje pri seštevanju vektorjev.

5.1.3 Seštevanje vektorjev

Slika 5.3 prikazuje pričakovano relativno obžalovanje po I korakih za problem seštevanja vektorjev. Vsak primerek je vektor, kjer je dimenzija vzorec iz diskretne enakomerne porazdelitve v intervalu $[275, 1000000]$. Učni primerki so uporabljali interval $[1, 200]$.

Strategija CPE ima v vseh korakih, razen v prvem, pričakovano vrednost 0. Strategija GPE ima od približno dvajsetega koraka naprej pričakovano vrednost nekoliko nad 3. Pričakovani vrednosti LinUCB in LinUCB* zelo hitro padeta pod pričakovano vrednost 0,5, potem pa še naprej padata v vsakem koraku. Strategija LinUCB* ima še nekoliko nižjo pričakovano relativno obžalovanje kot LinUCB. Pričakovana vrednost strategije TL* po nekaj korakih pade na 0,5, potem pa se ne spreminja več. Strategija TL ima pričakovano vrednost 1,5. Strategiji TR in TR* izboljšujeta svoje pričakovano relativno obžalovanje. Strategija TR* se izboljšuje precej hitreje kot TR, a po 100 korakih še vedno ni tako dobra kot strategija TL*.

5.2 Statistični model

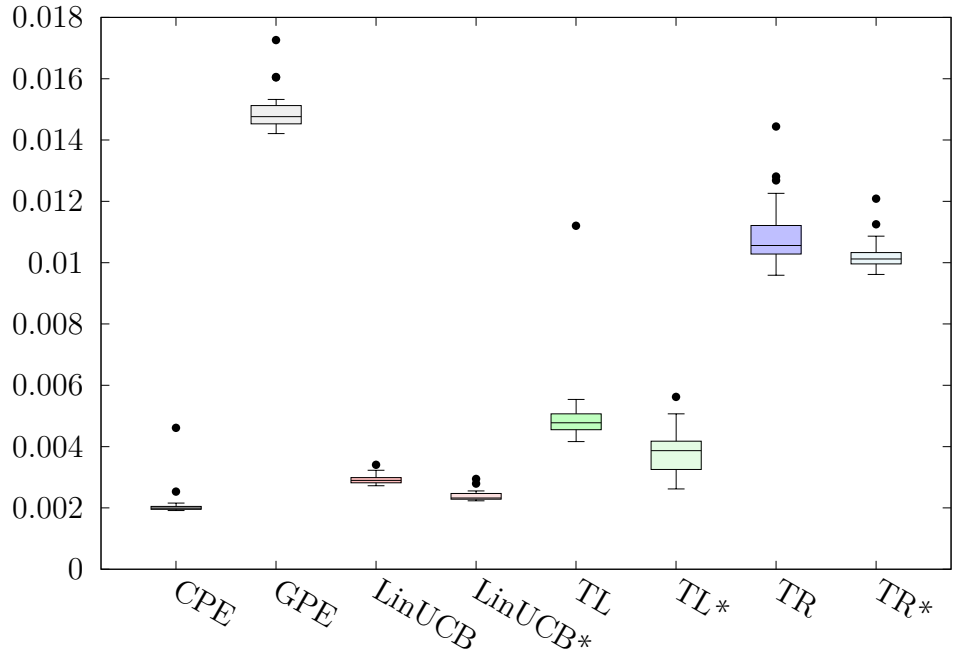
Uporabimo model in podatke, kot so jih Češnovar et al. [19] za regresijo z Gaussovim procesom. V modelu smo zamenjali funkciji za množenje matrik in razcep Choleskega s funkcijama z izbiro. Izberemo več velikosti vhodov in za vsako velikost generiramo 20 serij vzorcev. Za vsako strategijo izmerimo, koliko časa traja, da dobimo 20 vzorcev.

Za vsako velikost vzorcev naredimo svoj graf s škatlami z brki [20]. Vsaka škatla predstavlja eno strategijo. Spodnji rob škatle predstavlja prvi kvartil, zgornji pa tretjega. Črta znotraj škatle predstavlja drugi kvartil oziroma mediano. Brki predstavljajo najmanjšo in največjo vrednost, ki ju še ne štejemo med izjeme. Med izjeme štejemo vse podatke, ki niso znotraj intervala $\left[Q_1 - \frac{3}{2}(Q_3 - Q_1), Q_3 + \frac{3}{2}(Q_3 - Q_1)\right]$.

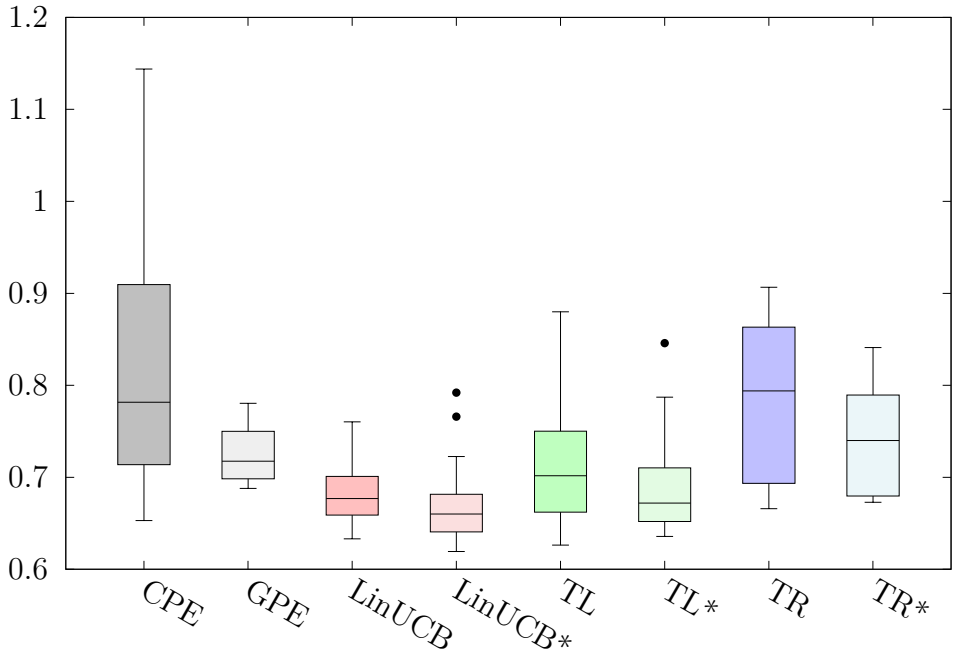
Slika 5.4 prikazuje skupni čas vzorčenja, ko so vhodni podatki zelo majhni. Vektorji v vhodnih podatkih so dolžine 16. V tem primeru je čas vzorčenja na CPE veliko krajši od časa na GPE. Strategiji LinUCB in LinUCB* se dobro prilagodita na to in imata le rahlo slabši čas vzorčenja, kot če bi procesirali le na CPE. Strategiji TL in TL* potrebujeta nekaj več časa, a se tudi približata času, ki ga ima obdelava izključno na CPE. Strategiji TR in TR* se ne naučita dovolj hitro, da čas vzorčenja ne bi bil le povprečje vzorčenja na CPE in na GPE.

Slika 5.5 prikazuje skupni čas vzorčenja, ko so vhodni podatki tako veliki, da vzorčenje na CPE in na GPE traja približno enako časa. To je v našem primeru takrat, ko so vektorji vhodnih podatkov dolžine 256. Vidimo lahko, da strategije LinUCB, LinUCB*, TL in TL* dosegajo dober rezultat. S temi strategijami vzorčimo enako hitro, včasih pa celo hitreje, kot strategija, ki vedno izbere eno izmed naprav. Med temi sta še posebno učinkoviti strategiji, ki sta imeli predhodno učenje. Strategiji TR in TR* se tudi v tem primeru učita prepočasi, da bi imeli dober čas vzorčenja.

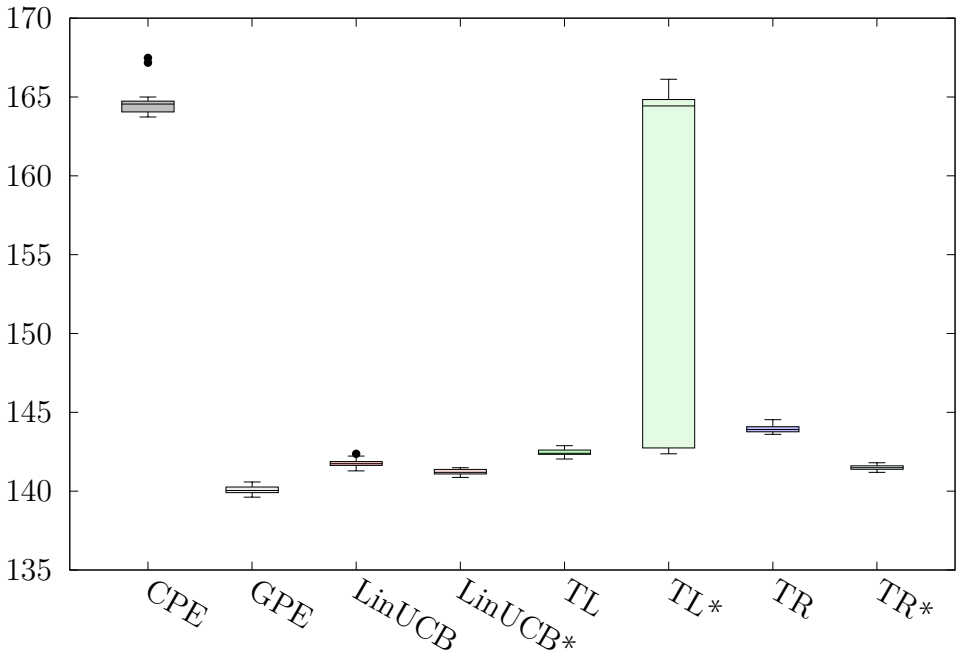
Slika 5.6 prikazuje skupni čas vzorčenja, ko so vhodni podatki zelo veliki Vektorji v vhodnih podatkih so dolžine 3072. V tem primeru je čas vzorčenja na GPE krajši od tega na CPE. Obe strategiji, ki uporabljata LinUCB, se



Slika 5.4: Skupni čas vzorčenja pri vhodnih vektorjih dolžine 16.



Slika 5.5: Skupni čas vzorčenja pri vhodnih vektorjih dolžine 256.



Slika 5.6: Skupni čas vzorčenja pri vhodnih vektorjih dolžine 3072.

dobro prilagodita in imata le rahlo slabši čas vzorčenja, kot če bi procesirali le na GPE. Strategija TL se tudi približa času na CPE. Strategija TL* pa zaradi slabo naučene funkcije za določanje a priori pričakovane vrednosti časa izvajanja velikokrat izbere izvajanje na CPE, kar podaljša čas izvajanja. Rezultat strategije TR je nekoliko slabši od strategije TL, a še vedno precej bližje najboljšemu času izvajanja strategije GPE kot najslabšemu strategije GPE. Čas strategije TR* je med boljšimi, saj je med časoma strategije LinUCB* in LinUCB.

Poglavje 6

Zaključki

V delu smo raziskali in implementirali tri odločitvene metode za reševanje problema večrokega bandita. Sestavili smo programsko arhitekturo, kjer izbiramo med več izvedbami algoritmov za reševanje računskih problemov. Programska arhitektura nam omogoča, da združimo več osnovnih funkcij v take, ki so enake oblike kot osnovne, ampak imajo vgrajeno izbiro. Zaradi tega lahko v obstoječi kodi osnovne funkcije zamenjamo s funkcijami z izbiro z nekaj majhnimi spremembami. To smo pokazali na primeru programske kode, v katero se prevede model v jeziku Stan. Programska arhitektura poleg tega omogoča, da poznavalec računskega problema vnese veliko svojega znanja, hkrati pa lahko nastavimo privzete vrednosti, ki bodo zaradi sprotnega učenja odločitvenih metod dosegale sprejemljive rezultate.

Funkcije z izbiro smo eksperimentalno preizkusili na raznih računskih problemih in porazdelitvah velikosti primerkov ter na praktičnem primeru statističnega modela, kot ga lahko definiramo v jeziku Stan. Za primerjavo smo dodali dve odločitveni metodi, ki v vsakem koraku izbereta ali CPE ali GPE, ker sta to trenutni možni izbiri v prevajalniku Stanc. Da bi se izognili izbiri a priori vrednosti, ki bi delovale le za posamezen preizkus, smo enake a priori vrednosti in postopke učenja uporabili za vse velikosti primerkov, ne glede na preizkus. Preizkusi so obravnavali različne računske probleme in velikosti primerkov. Trenutna možnost, ki za vsak računski problem in vsak

primerek izbere CPE, se je slabo odrezala pri velikih primerkih problema množenja matrik in razcepa Choleskega. Možnost, ki vedno izbere GPE, pa se je slabo odrezala pri vseh majhnih primerkih in pri seštevanju vektorjev. Metoda LinUCB je v vsakem preizkusu dosegla rezultat, ki je bil najboljši ali primerljiv z najboljšim. V vsakem preizkusu je bil rezultat metode LinUCB izboljšan, če smo metodo predhodno učili. Učenje je imelo vpliv tudi takrat, ko smo reševali večje primerke problemov, čeprav smo učili le na manjših. Ostale odločitvene metode, kot tudi LinUCB, bi se verjetno dalo dodatno izboljšati s prilagajanjem a priori vrednosti in učnih primerkov. Za nadaljnje delo je vsa koda na voljo na javno dostopnem repozitoriju[1].

[1]https://github.com/jurijsteblaj/device-multiarmed-bandit.git

Dodatek A

Funkciji getChoiceFunc in getSimpleChoiceFunc

Sledi kode funkcije getChoiceFunc v jeziku C++. Kot parameter prejme seznam osnovnih funkcij in vrne funkcijo choiceFunc, kot je opisana v psevdokodi 6.

```
template <typename R, typename... Args>
std::function<R(Strategy&, InstanceSize, Args...)> get_choice_func(
        std::array<std::function<R(Args...)>, DEVICE_COUNT> funcs) {

    return [funcs](Strategy& strategy, InstanceSize instance_size, Args... args) {
        auto choice = strategy.choose(instance_size);

        auto start = Clock::now();
        auto result = funcs[choice](args...);
        auto end = Clock::now();

        strategy.update(instance_size, choice, end - start);
        return result;
    };
}
```

Sledi koda funkcije getSimpleChoiceFunc v jeziku C++. Kot parametre prejme funkcijo z izbiro, odločitveno metodo in funkcijo za izračun velikosti primerka. Vrne funkcijo simpleChoiceFunc, kot je opisana v psevdokodi 8.

```
template <typename R, typename... Args>
std::function<R(Args...)> get_simple_choice_func(
        std::function<R(Strategy&, InstanceSize, Args...)> choice_func,
        Strategy& strategy,
        std::function<InstanceSize(Args...)> instance_size_func
            = default_instance_size_func<Args...>) {

    return [choice_func, &strategy, instance_size_func](const Args&&... args) {
        return choice_func(strategy, instance_size_func(args...), args...);
    };
}
```

Dodatek B

Razlike v kodi dveh modelov

Sledi izpis razlik med kodo v jeziku C++, kot jo proizvede prevajalnik Stanc, in kodo, kjer lahko nastavimo funkcije za reševanje računskih problemov. V prvem odseku sta parametrom funkcije gp_pred_rng dodani funkciji za množenje matrik in razcep Choleskega. V drugem sta temu razredu statističnega modela dodani polji za enaki funkciji. V tretjem sta taki funkciji dodani parametrom konstruktorja razreda. Konstruktor prekopira te funkciji v polja v drugem odseku. Kot privzeti vrednosti sta nastavljeni funkciji, ki se izvedeta na CPE, kar mogoča enako uporabo razreda kot pred spremembo. V četrtem odseku sta funkciji iz drugega odseka uporabljeni v klicu funkcije gp_pred_rng.

```
--- models/GP.hpp 2020-06-11 21:14:31.107120527 +0200
+++ models/GP_altered.hpp 2020-10-22 22:37:15.118000000 +0200
@@ -135,5 +135,10 @@
             const std::vector<T2__>& x1, const T3__& alpha, const T4__& rho,
             const T5__& sigma, const T6__& delta, RNG& base_rng__,
-            std::ostream* pstream__) {
+            std::ostream* pstream__,
+            std::function<stan::math::matrix_d(
+              const stan::math::matrix_d&,
+              const stan::math::matrix_d&)> multiply,
+            std::function<stan::math::matrix_d(
+              const stan::math::matrix_d&)> cholesky_decompose) {
   using local_scalar_t__ = typename boost::math::tools::promote_args<T0__,
           T1__,
@@ -351,4 +356,8 @@
```

```
   int N_predict;
   std::vector<double> x_predict;
+  std::function<stan::math::matrix_d(
+    const stan::math::matrix_d&,
+    const stan::math::matrix_d&)> multiply;
+  std::function<stan::math::matrix_d(const stan::math::matrix_d&)> cholesky;

  public:
@@ -357,6 +366,14 @@
   std::string model_name() const { return "GP_model"; }

-  GP_model(stan::io::var_context& context__, unsigned int random_seed__ = 0,
-           std::ostream* pstream__ = nullptr) : model_base_crtp(0) {
+  GP_model(stan::io::var_context& context__,
+           std::function<stan::math::matrix_d(
+             const stan::math::matrix_d&,
+             const stan::math::matrix_d&)> multiply_func
+               = [](const auto& a, const auto& b){ return stan::math::multiply(a, b); },
+           std::function<stan::math::matrix_d(const stan::math::matrix_d&)> cholesky_func
+            = [](const auto& a){ return stan::math::cholesky_decompose(a); },
+           unsigned int random_seed__ = 0,
+           std::ostream* pstream__ = nullptr)
+           : model_base_crtp(0), multiply(multiply_func), cholesky(cholesky_func) {
     typedef double local_scalar_t__;
     boost::ecuyer1988 base_rng__ =
@@ -612,5 +629,6 @@
       assign(f_predict, nil_index_list(),
         gp_pred_rng(x_predict, y, x, alpha, rho, sigma,
-          1e-10, base_rng__, pstream__), "assigning variable f_predict");
+          1e-10, base_rng__, pstream__, multiply, cholesky),
+          "assigning variable f_predict");
       current_statement__ = 5;
       validate_non_negative_index("y_predict", "N_predict", N_predict);
```

Literatura

[1] C. Gregg, K. Hazelwood, Where is the data? Why you cannot debate CPU vs. GPU performance without the answer, in: (IEEE ISPASS) IEEE International Symposium on Performance Analysis of Systems and Software, IEEE, 2011, pp. 134–144.

[2] B. Carpenter, A. Gelman, M. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, A. Riddell, Stan: A probabilistic programming language, Journal of Statistical Software, Articles 76 (1) (2017) 1–32. doi:10.18637/jss.v076.i01.
URL https://www.jstatsoft.org/v076/i01

[3] C. Nugteren, V. Codreanu, Cltune: A generic auto-tuner for opencl kernels, 2015 IEEE 9th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (Sep 2015). doi:10.1109/mcsoc.2015.10.
URL http://dx.doi.org/10.1109/MCSoC.2015.10

[4] B. van Werkhoven, Kernel tuner: A search-optimizing GPU code auto-tuner, Future Generation Computer Systems 90 (2019) 347 – 358. doi:https://doi.org/10.1016/j.future.2018.08.004.
URL http://www.sciencedirect.com/science/article/pii/S0167739X18313359

[5] S. Mittal, J. S. Vetter, A survey of CPU-GPU heterogeneous computing techniques, ACM Comput. Surv. 47 (4) (2015) 69:1–69:35. doi:10.

1145/2788396.
URL http://doi.acm.org/10.1145/2788396

[6] J. Ansel, C. Chan, Y. L. Wong, M. Olszewski, Q. Zhao, A. Edelman, S. Amarasinghe, Petabricks: A language and compiler for algorithmic choice, in: ACM SIGPLAN Conference on Programming Language Design and Implementation, Dublin, Ireland, 2009.
URL http://groups.csail.mit.edu/commit/papers/2009/ansel-pldi09.pdf

[7] J. Ansel, M. Pacula, Y. L. Wong, C. Chan, M. Olszewski, U.-M. O'Reilly, S. Amarasinghe, Siblingrivalry: Online autotuning through local competitions, in: International Conference on Compilers Architecture and Synthesis for Embedded Systems, Tampere, Finland, 2012.
URL http://groups.csail.mit.edu/commit/papers/2012/ansel-cases12-siblingrivalry.pdf

[8] M. Pacula, J. Ansel, S. Amarasinghe, U.-M. O'Reilly, Hyperparameter tuning in bandit-based adaptive operator selection, in: European Conference on the Applications of Evolutionary Computation, Malaga, Spain, 2012.
URL http://groups.csail.mit.edu/commit/papers/2012/pacula-evoapps12-banditext.pdf

[9] Y. Wen, Z. Wang, M. F. O'boyle, Smart multi-task scheduling for OpenCL programs on CPU/GPU heterogeneous platforms, in: 2014 21st International Conference on High Performance Computing (HiPC), IEEE, 2014, pp. 1–10.

[10] R. M. Stallman, GNU compiler collection internals, Free Software Foundation (2002).

[11] M. Godbolt, Optimizations in C++ compilers, Queue 17 (5) (2019) 69–100. doi:10.1145/3371595.3372264.
URL https://doi.org/10.1145/3371595.3372264

[12] B. Goetz, Java theory and practice: Dynamic compilation and performance measurement, IBM Developer Works (2004-2007) (2004) 1–8.

[13] L. Li, W. Chu, J. Langford, R. E. Schapire, A contextual-bandit approach to personalized news article recommendation, CoRR abs/1003.0146 (2010). arXiv:1003.0146.
URL http://arxiv.org/abs/1003.0146

[14] S. L. Scott, A modern bayesian look at the multi-armed bandit, Applied Stochastic Models in Business and Industry 26 (6) (2010) 639–658. arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/asmb.874, doi:10.1002/asmb.874.
URL https://onlinelibrary.wiley.com/doi/abs/10.1002/asmb.874

[15] O. Chapelle, L. Li, An empirical evaluation of thompson sampling, in: Advances in neural information processing systems, 2011, pp. 2249–2257.

[16] A. Gelman, J. B. Carlin, H. S. Stern, D. B. Dunson, A. Vehtari, D. B. Rubin, Bayesian data analysis, CRC press, 2013.

[17] E. Gamma, Design patterns: elements of reusable object-oriented software, Pearson Education India, 1995.

[18] D. R. Prasanna, Dependency injection, Manning Publications, 2009.

[19] R. Češnovar, S. Bronder, D. Sluga, J. Demšar, T. Ciglarič, S. Talts, E. Štrumbelj, GPU-based parallel computation support for stan, CoRR abs/1907.01063 (2019). arXiv:1907.01063.
URL http://arxiv.org/abs/1907.01063

[20] T. Williams, C. Kelley, H. Broker, R. John Campbell, D. Cunningham, G. Denholm, R. Elber, C. Fearick, L. Grammes, L. Hart, et al., Gnuplot 5.2. 2: An interactive plotting program (2017).