SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

SEMINAR

Simbolička regresija

Dino Šantl

Mentor: *Prof. dr. sc. Domagoj Jakobović*

Zagreb, svibanj 2013.

SADRŽAJ

1. Uvod

Simbolička regresija je pronalaženje matematičkog izraza koji najbolje opisuje ulazne primjere. Za razliku od "obične" regresije, simbolička regresija ne pretpostavlja model, odnosno prema [1], simbolička regresija je slučaj regresije u kojem model nije definiran prije postupka pronalaženja tj. osim parametara modela traži se i sam model.

U ovom seminaru opisuju se načini ostvarivanja simboličke regresije na računalu. Kao uvod daje se jednostavan primjer u kojem se pojavljuje osnovna terminologija vezana za temu, a koja se koristi i u daljnjem tekstu. Opisat će se računalne metode koje se danas koriste za ostvarivanje simboličke regresije s naglaskom na genetsko programiranje. Pokazat će se dva pristupa koja mogu poboljšati simboličku regresiju (pretpostavlja se korištenje genetskog programiranja). Ukratko će se dati ideje koje mogu ubrzati simboličku regresiju pomoću paralelizacije algoritma. Na kraju, kao motivacija za daljini rad na ovom području, dati će se primjer korištenja simboličke regresije u praksi.

1.1. Simbolička regresija

Sve do nedavno simboličku regresiju provodili su isključivo ljudi na temelju vlastite intuicije i iskustva. Ako se malo bolje pogleda, pretpostavljanje modela od strane čovjeka zapravo je simbolička regresija. Kao takva, simbolička regresija se ne može implementirati na računalu zbog toga što problem nije matematički dobro definiran. Definicija simboličke regresije ne govori ništa o tome kako se pronalazi matematički izraz i isto tako ne govori o tome što znači da matematički izraz *dobro* opisuje ulazne primjere.

Ulazni primjeri za simboličku regresiju su parovi ulaznih i izlaznih vrijednosti funkcije. Formalno se to može zapisati kao: $\{\mathbf{x_i}, y_i\}$, gdje je *i* indeks pojedinog ulaznog primjera: $i = 1...N$, pri čemu je *N* ukupan broj primjera, a $\mathbf{x_i}$ predstavlja ulazne parametre funkcije zapisane pomoću vektora (ako se radi o funkciji više varijabli).

Za skalarnu funkciju jedne varijable ulazni primjeri mogu biti parovi realnih bro-

jeva: $\{\{0,0\},\{-1,1\},\{1,1\}\}$. Neka se za ove ulazne primjere provodi simbolička regresija. Za ovaj primjer pretpostavlja se korištenje samo elementarnih matematičkih funkcija i operatora $+,-,/,*$. Potrebno je još odrediti na koji način se mogu ocjenjivati dvije matematičke funkcije tj. odrediti koja je bolja. Jedan način za to je korištenjem srednje kvadratne pogreške:

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - h(\mathbf{x_i}))^2 \tag{1.1}$$

gdje je h matematički izraz dobiven u nekom koraku pretraživanja. Pitanje koje se sada može postaviti je: Koja je najbolja matematička funkcija koja opisuje zadane ulazne primjere? Naravno, rješenje ne mora biti jedinstveno, jer ako više funkcija daje istu minimalnu pogrešku, tada su sve te funkcije najbolje (bolje od drugih). Kao što se može naslutiti ovo je veoma težak računarski problem i on se ne može rješavati klasičnim pristupom tj. pretraživanjem svih mogućih matematičkih funkcija koje mogu nastati iz funkcija i operatora koji su pretpostavljeni. Za dani ulazni primjer jedna od mogućih matematičkih funkcija je $h(x) = x \cdot x$. Pogreška za ovu funkciju je 0, a kako pogreška ne može biti manja od 0, zaključujemo da je ta funkcija najbolja. Istu pogrešku daje i funkcija $h(x) = abs(x)$ i prirodno se postavlja pitanje koja je bolja. To se ne može odrediti iz podataka te su ove dvije funkcije jednako dobre. U praksi se često događa da matematički izrazi postaju jako kompliciran (njihov zapis je jako dugačak). Zato je ponekad u pretraživanje potrebno uvesti pristranost kako bi se kraća rješenja mogla preferirati u odnosu na onda dulja. Iako se ovaj problem može riješiti i uvođenjem regularizacije, tj. da se u računanje pogreške uvede i mjera za duljinu matematičkog izraza, ali to neće biti opisano u ovom seminaru.

2. Ostvarivanje simboličke regresije na računalu

Prostor rješenja, tj. prostor svih matematičkih izraza koji se mogu izgraditi iz nekog početnog skupa osnovnih elemenata najčešće je beskonačan. Zbog toga je potrebno koristi neku od heuristika. Evolucijski algoritmi se nameću kao moguće rješenje simboličke regresije. Danas se za simboličku regresiju koriste tri glavna pristupa [2], a to su:

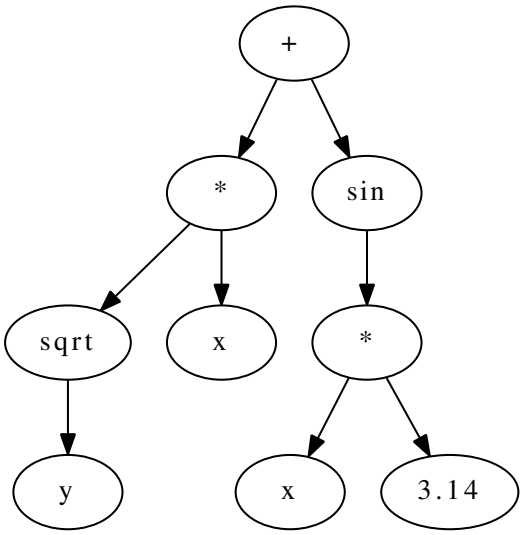
1. Genetsko programiranje
2. Gramatička evolucija
3. Analitičko programiranje

U nastavku će biti opisan svaki od pristupa. *Genetsko programiranje* biti će opisano detaljno, tj. do implementacijske razine. Za *gramatičku evoluciju* i *analitičko programiranje* dati će se samo osnovne informacije koje su potrebne za razumijevanje rada tih algoritama. Zajedničko svima je to da evolucijom dolaze do novih rješenja. Kvaliteta rješenja se tada procjenjuje prema odstupanju od danih ulaznih podataka, tj. računom pogreške (ne mora biti srednja kvadratna pogreška). Glavna razlika je prikaz rješenja u memoriji (tj. prikaz jedinki). Posljedica toga su i drugačije definirani genetski operatori. Za sve postupke se pretpostavlja da su funkcije i operatori definirani prije provođenja samih postupaka, tj. skup funkcija i skup operatora su konačni skupovi.

2.1. Genetsko programiranje

Genetsko programiranje (engl. *Genetic programming*) svrstava se u evolucijske algoritme. Genetsko programiranje koristi isti koncept kao i genetski algoritam, pa će radi potpunosti biti objašnjen cijeli koncept. Jedinka u genetskom programiranju je program. Za potrebe simboličke regresije to je samo matematički izraz tj. funkcija.

Najčešći prikaz jedinke je u obliku sintaksnog stabla (slika 2.1). Jedinka se osim spomenutog načina može prikazati linearno, tj. kao niz znakova. Često se koristi prefiksni prikaz jer je pogodan za izračun izraza (dovoljno je koristiti stog).



Slika 2.1: Primjer sintaksnog stabla koje prikazuje izraz: $\sqrt{y} \cdot x + sin(3.14 \cdot x)$

2.1.1. *Steady-state* algoritam

Pokazat će se *Steady-state* algoritam (slika 2.2). zbog njegove jednostavnosti. Može se koristiti i populacijski algoritam, ali on neće biti ovdje izložen.

Koraci koji se obavljaju pri ovom postupku su:

1. Generiranje slučajne populacije
2. Evaluacija populacije
3. Selekcija
4. Križanje
5. Mutacija
6. Evaluacija nove jedinke
7. Ubacivanje u populaciju
8. Kraj ako je zadovoljen uvjet zaustavljanja, inače skoči na 3. korak

Slika 2.2: Koraci *steady-state* algoritma

2.1.2. Generiranje slučajne populacije

Slučajna se populacija u genetskom programiranju generira tako da se stvara slučajno stablo. Kako je stablo sintaksno, unutarnji čvorovi su operatori ili funkcije a listovi su konstante ili varijable. Pri generiranju stabla potrebno je paziti da stablo ne naraste previše. Zato se kao parametar može dati maksimalna duljina stabla. Ponekad je potrebno generirati i malo dublja stabla pa se kao parametar može dati i minimalna dubina stabla.

Stablo se generira tako da se za korijen stabla slučajno odabere operator ili funkcija. Nakon toga potrebno je generirati onoliko djece koliko operator ima operanada ili funkcija argumenata. Djeca mogu biti opet operator ili funkcija, ali i varijabla i konstanta. Postupak se dalje obavlja rekurzivno sve dok se ne postigne dovoljna dubina stabla.

2.1.3. Evaluacija

Evaluacija je postupak u kojem se jedinke ocjenjuju. Da bi evolucija bila uspješna potrebno je na pravilan način ocijeniti jedinke. Ocjena za simboličku regresiju sama se nameće, a to je odstupanje ulaznih primjera od izlaza dobivenog matematičkog izraza. U ovom koraku potrebno je definirati funkciju dobrote. Najčešće se koristi kvadratno

odstupanje, ali to nije nužno. Ponekad je zbog numeričke pogreške potrebno definirati malo drugačije izraze kao što je korišteno to u [3]:

Srednja logaritamska pogreška:

$$-\frac{1}{N}\cdot\sum_{i=1}^{N} log\left[1+abs(y_i-h_i)\right] \tag{2.1}$$

Dodatno se za funkciju dobrote ne mora koristiti čista greška već funkcija dobrote može skalirati sve vrijednosti prema najboljoj i sl.

2.1.4. Selekcija

Selekcija je postupak odabiranja jedinki. Za različite varijante genetskog programiranja koriste se različite vrste selekcija, tj. ova operacija isto nije čvrsto definirana. Selekcijom se u algoritmu biraju roditelji (koji se kasnije križaju). Selekcija mora biti takva da veću prednost daje boljim jedinkama. Selekcija mora biti simulirana nedeterminističkom metodom jer inače slabije jedinke nikad neće dobiti priliku (gubitak genetskog materijala). Postoji više operatora odabira, jedan od češće korištenih je turnirski odabir. Iz populacije se slučajno (uniformno) odabire k jedinki. Od tih k odabire se najbolja.

2.1.5. Križanje

U genetskom programiranju operator križanja može se ostvariti tako da se odabere podstablo u svakom roditelju, te se ta dva podstabla zamijene. Kako su time dobivene dvije nove jedinke, za novu jedinku mogu se uzeti oboje (neke varijante genetskog programiranja), bolja ili slučajna. Pretpostavka za varijantu koja se opisuje je da se uzima samo jedna jedinka.

2.1.6. Mutacija

Operator mutacije može se ostvariti tako da se odabere slučajno podstablo u jedinki i zamijeni se s novim stablom koje je slučajno generirano. Parametar koji se može zadati je vjerojatnost da li operator mutacije djeluje nad jedinkom ili ne. Jednako tako mogu se zadavati kompliciraniji parametri kao što je distribucija vjerojatnosti po dubini stabla, pa tako čvorovi koji su u sredini mogu biti odabrani najvećom vjerojatnošću, a oni koji su na vrhu ili dnu nekom manjom vjerojatnošću.

2.1.7. Ubacivanje u populaciju

U ovom koraku se isto kao i kod selekcije koristi operator odabira. Razlika je u tome što se mora odabrati loša jedinka koja će biti izbačena, a na njezino mjesto će doći jedinka koja je dobivena križanjem roditelja i mutacijom.

2.2. Gramatička evolucija

Gramatička evolucija (engl. *Grammatical evolution*) inspirirana je biološkim procesom stvaranja proteina iz genetskog materijala organizama [4]. Gramatička evolucija može se ostvariti na sličan način kao i genetsko programiranje, ali je razlika u prikazu jedinke. Zbog toga je potrebno definirati nove operatore za križanje i mutaciju.

2.2.1. Prikaz jedinke

Prikaz jedinke je jednostavniji od prikaza u genetskom programiranju. Jedinka se prikazuje nizom bitova. Program tj. matematički izraz koji se dobiva nema direktnu reprezentaciju već je implicitno opisan jedinkom.

Jedinka je niz bitova fiksne duljine. Duljina niza je broj djeljiv s 8 (ne mora nužno biti 8, ali najjednostavnije za implementaciju je 8). Prevođenje niza u matematički izraz vrši se uzimanjem po 8 bitova iz niza. Tih 8 bitova predstavlja jedan cijeli broj (od 0 do 255). Niz se može promatrati i kao niz cijelih brojeva osam puta kraće duljine od originalnog niza bitova. Iteriranjem kroz niz cijelih brojeva gradi se matematički izraz primjenom gramatičkih pravila. Koje gramatičko pravilo će biti odabrano ovisi o trenutnom broju u nizu.

Za prikaz gramatičkih pravila koristit će se Backus–Naur forma (*BNF*). Gramatika se može prikazati četvorkom $\{N, T, P, S\}$ [4]. N predstavlja nezavršne znakove, T su završni znakovi, P su pravila i S je početni ne završni znak.

Neka je zadana jedinka duljine 56 bita. Niz se može zapisati kao niz cijelih brojeva duljine 7: $[21, 2, 12, 1, 8, 5, 4]$. Za svaki nezavršni znak i produkciju u zagradi s desne strane napisan je broj. Taj broj predstavlja indeks pravila koje će biti korišteno prema trenutnom broju u nizu. Kreće se od početnog nezavršnog znaka ***<izraz>***. Prvi broj u nizu je 21. Kako za taj nezavršni znak nema 21 produkciju (već samo 3) računa se ostatak pri dijeljenju s 3. Rezultat je 0 ($21\ mod\ 3 = 0$). Odabire se produkcija *(0)* i dobiva se desna strana produkcije: ***<izraz><operator><izraz>***. Uvijek se gleda najljeviji

Neka je zadana gramatika:
N = {<izraz>, <operator>, <var>, <func>}
T = {X, 3.14, +, -, *, /, Sin, Log}
S = {<izraz>}

Pravila P:

<izraz>	::=	<izraz> <operator> <izaraz>	(0)
	|	<var>	(1)
	|	<func>(<izraz>)	(2)
<operator>	::=	+	(0)
	|	-	(1)
	|	*	(2)
	|	/	(3)
<var>	::=	X	(0)
	|	3.14	(1)
<func>	::=	Sin	(0)
	|	Log	(1)

nezavršni znak. To je opet ***<izraz>***. Sad je na redu broj 2 i za njega ne treba računati ostatak. Primjenjuje se produkcija *(2)* i rezultat je ***<func>(<izraz>)<operator><izraz>***. Sljedeći je broj 12, a nezavršni znak za kojeg se gledaju produkcije je ***<func>***. U toj skupini postoje dvije produkcije, a kako je ostatak dijeljenja broja 12 s 2 jednak 0, uzima se produkcija koja s desne strane ima završni znak Sin. Ovaj postupak se primjenjuje sve dok svi znakovi ne budu završni. Rezultat je: $Sin(X) - 3.14$.

Ako je gramatika definirana tako da za neki nezavršni znak postoji samo jedna produkcija, tada se ona uzima bez trošenja elementa u nizu. Može se reći da se čita element niza samo u slučaju kad postoji izbor.

Postoje dva scenarija koja se mogu dogoditi:

1. Izraz je poprimio svoj konačni oblik, a nisu pročitani svi brojevi u nizu - brojevi koji nisu pročitani jednostavno se zanemare

2. Izraz nije poprimio konačni oblik (postoje nezavršni znakovi u njemu), a svi brojevi u nizu su iskorišteni - koristi se umotavanje niza (engl. *wrapping*), tj. čitanje niza jednostavno počinje ispočetka. Inspiracija za ovo dolazi iz biologije (npr. neke bakterije koriste sličnu tehniku da bi iz istog genetskog materijala dobile različite gene) [4]. Možda se na prvi pogled čini kao da će opet biti generiran identičan izraz, ali to najčešće ne mora biti slučaj, jer odabir trenutne produkcije ovisi o kontekstu u kojem se trenutno generirani izraz nalazi, tj. o najljevijem nezavršnom znaku, pa tako isti slijed brojeva može generirati sasvim drugačiji izraz u nastavku.

2.2.2. Genetski operatori

Genetski operatori moraju se malo drugačije definirati. Kako je prikaz jedinke niz bitova, a isti prikaz često se koristi u genetskim algoritmima, svi operatori definirani u tom kontekstu su valjani za gramatičku evoluciju. Za primjer može se uzeti križanje s jednom točkom prekida.

Mutacija se može definirati kao vjerojatnost promijene jednog bita u jedinki, a vjerojatnost se daje kao parameter algoritmu.

2.3. Analitičko programiranje

Analitičko programiranje (engl. *analytic programming*) koristi drugačiji način prikaza jedinke od genetskog programiranja i gramatičke evolucije.

2.3.1. Prikaz jedinke

Iako analitičko programiranje koristi različit prikaz jedinke, veoma je sličan gramatičkoj evoluciji tj. u neizravnom prikazu programa (matematičkog izraza). Najbitnija razlika je što analitičko programiranje prikazuje jedinku odmah kao konačan niz cijelih brojeva konstantne duljine. Druga bitna razlika je što se ne koristi gramatika već nešto jednostavniji pristup. Analitičko programiranje ne razlikuje funkcije, operatore, varijable i konstante. Cijeli taj skup koji korisnik zadaje zove se *opći skup funkcija*. Elementi tog skupa postave se u niz i svakome se pridijeli indeks. Slično kao i kod gramatičke evolucije čita se niz. Svaki element u nizu (cijeli broj) određuje koji element iz skupa općih funkcija će se koristiti. Ako se iz skupa odabere operator, sljedeća dva elementa niza (jedinke) određuju operande. U slučaju da se odabere funkcija, tada su argumenti funkcije određeni elementima koji slijede iza trenutnog.

Isto kao i kod gramatičke evolucije moguća su dva scenarija. Slučaj kada je generiranje programa završilo, a jedinka još nije do kraja pročitana, ostali elementi se zanemaruju. Za razliku od gramatičke evolucije, u analitičkom programiranju nikada neće doći do kraja niza (jedinke), a da generirani izraz nije ispravan (npr. nedostaju argumenti funkcije ili nedostaju operandi). Trik koji se koristi je podjela *općeg skupa funkcija* u podskupove, te se u podskupovima definiraju novi indeksi za svaki element. Podskupovi su organizirani tako da su zajedno svrstani svi elementi koji koriste isti ili manji broj parametara (tablica 2.1). Tako npr. sve funkcije dvije varijable, operatori, varijable i konstante su članovi jednog podskupa koji se može označiti s GFS_{arg2}. Najniži podskup je skup u kojem se nalaze varijable i konstante tj. u tom skupu se

nalaze elementi koji ne trebaju parametre. Kada se pročita element niza provjeri se koliko je za taj odabir potrebno sljedećih elemenata niza. Ako je taj broj veći nego je ostalo elemenata niza, ide se u podskup koji koristi jednak ili manji broj parametara, te se prema indeksima tog podskupa odabire novi element u izgradnji izraza.

Za jednostavan primjer može se uzeti jedinka (niz) od 5 brojeva. Neka je zadani niz $[1, 38, 72, 2, 23]$. U tablici 2.1 definirane su korisničke funkcije i numerirane su u svakom podskupu. Izgradnja izraza kreće od broja 1. Može se koristiti GFS_{all} skup jer je najveći broj parametara 2, a kako postoje sljedeća dva člana u nizu (38 i 72) nije potrebno smanjivati skup. Broj 1 predstavlja operaciju množenja. Operandi operacije su definirani brojevima 38 i 72. Kako su oba broja veća od ukupnog broja elemenata skupa računa se ostatak dijeljenja s brojem elemenata u promatranom podskupu i dodaje se 1. Za operande uzimaju se elementi skupa na indeksima 3 odnosno 1 (vrijedi samo za trenutno promatrani skup). Prvi operand je funkcija sin koja treba jedan argument. Uzima se prvi koji nije iskorišten, a to je broj 2 u nizu. U ovom trenutku javlja se problem jer 2 u GFS_{all} skupu označava zbrajanje koje treba dva operanda, a u nizu je ostao samo jedan neobrađeni broj. Zato se ide u podskup koji sadrži elemente s manjim brojem parametra. U podskupu GFS_{arg1} indeks 2 označuje cos funkciju i ona se može iskoristiti, a njezin je parametar broj 23. Broj 23 pomoću računanja ostatka daje indeks 6 što je varijabla X. To je u redu jer su svi elementi niza obrađeni. Ostaje još obraditi broj 72 koji predstavlja drugi operand operacije množenja. Kako više nema elemenata za čitanje, u obzir dolazi samo podskup GFS_{arg0}. Pripadajući indeks za broj 72 u tome skupu je varijabla X. Dakle konačan izgled izraza je: $sin(X) \cdot cos(X)$.

U ovom primjeru može se uočiti da bi se stog mogao iskoristiti u implementaciji analitičkog programiranja (izgradnja izraza obavlja se rekurzivno). Za jedan prolaz kroz n članova niza potrebno je n koraka. Trenutna pozicija u nizu određuje i koliko je još članova ostalo, pa je vremenska složenost pretvaranja jedinke u matematički izraz $O(n)$.

2.3.2. Genetski operatori

Kako autori izvornog rada [2] napominju, analitičko programiranje je samo metoda pretvaranja jedinke u program tj. matematički izraz. Zbog toga se ne definiraju posebni operatori, već se mogu definirati bilo kakvi operatori koji rade s nizom cijelih brojeva stalne duljine. Radi potpunosti opisa predlaže se operator križanja s jednom točkom prekida, ali umjesto da se prekida niz bitova, prekida se niz cijelih brojeva.

Mutacija se može definirati dvama parametrima. Prvi parametar je vjerojatnost

Tablica 2.1: Oganizacija funkcija, operatorau, varijabli i konstanti u podskupove

Oznaka skupa	Elementi skupa	Napomena
GFS_{all}	1. $\cdot$ 2. $+$ 3. sin 4. cos 5. $max(a, b)$ 6. X 7. Y 8. Z 9. 3.14	Opći skup funkcija
GFS_{arg1}	1. sin 2. cos 3. X 4. Y 5. Z 6. 3.14	Korištenje jednog ili manje parametara
GFS_{arg0}	1. X 2. Y 3. Z 4. 3.14	Bez dodatnih parametara

mutacije pojedinog elementa u nizu. Na svaki element koji je odabran za mutaciju dodaje se slučajni broj iz *Gaussove* distribucije. Zato je potreban još jedan parametar - varijacija distribucije (srednja vrijednost je 0). Kako je *Gaussova* distribucija definirana na skupu realnih brojeva, rezultat mutacije na pojedinom elementu niza može se zaokružiti na prvi manji cijeli broj.

3. Nadogradnja algoritma

U ovom poglavlju biti će opisane dvije metode koje unaprjeđuju pronalaženje matematičkih izraza. Metode su preuzete iz članaka [5] i [6]. Metode koje se obrađuju su:

- Intervalna aritmetika
- Linearno skaliranje

Intervalna aritmetika je usko povezana s načinom prikaza jedinke u *genetskom programiranju*. Za razliku od *intervalne aritmetike*, *linearno skaliranje* je univerzalno primjenjivo na bilo koju implementaciju simboličke regresije jer se koristi u računu pogreške (što je zajedničko svim metodama).

3.1. Intervalna aritmetika

3.1.1. Motivacija

Genetsko programiranje pretpostavlja da je svaki matematički izraz ispravan ako se može prikazati pomoću sintaksnog stabla. Ova pretpostavka može dovesti do prenaučenosti modela. To se može pokazati jednostavnim primjerom. Neka je rezultat simboličke regresije funkcija $h(x) = \frac{1}{x}$. Domena ulaznih primjera zadana u intervalu $[-5, 5]$. Iako ta funkcija daje najmanju grešku ona će uzrokovati prenaučenost. Za $x = 0$ funkcija daje na izlazu beskonačnost (∞). Veoma važno je uočiti da je kodomena ulaznih primjera konačna (svi y_i su konačni). Nikakvo mjerenje ne može kao izlaz dati beskonačnost, te ovakav rezultat simboličke regresije nema smisla. Najjednostavniji pristup je korištenje sigurnih operatora. Sigurni operator ima isto matematičko značenje kao i pripadajući *"obični"* operator, uz dodatak koji se brine o nevaljanom rezultatu operacije. Za operator dijeljenja $\frac{x}{y}$ može se definirati da je rezultat dijeljenja jednak 1 ako je $y = 0$. Ovakvim ograničenjem riješen je problem beskonačnosti. Za ostale operacije i funkcije koje nisu ograničene mogu se definirati slična

ograničenja. Iako je ovo veoma elegantno rješenje, ono ponekad može dovesti do prenaučenosti modela baš zbog konstantnih veličina koje ne moraju odgovarati redovima veličina promatranih ulaznih podataka. Intervalna aritmetika rješava ovaj problem bez dodatnih pretpostavki. Cijena koja se plaća je dodatan obilazak stabla.

3.1.2. Intervalna aritmetika u genetskom programiranju

Intervalna aritmetika je metoda za računanje gornje i dojne granice izlaza funkcije. Najčešće se koristi kako bi se procijenila izlazna vrijednost pri provođenju numeričkih postupaka. Ako je poznato da je ulazni parametar funkcije u nekom intervalu $x \in [-10, 10]$, tada se za proizvoljne funkcije ili operatore može odrediti interval u kojem se nalazi njihov izlaz. Ako se kao operacija uzme $2 \cdot x$ intervalna aritmetika će kao izlaz dati interval $[-20, 20]$. Općenito računanje intervala nije ovako trivijalno.

Genetsko programiranje ne koristi intervalnu aritmetiku za računanje izlaznog intervala, već kao metodu koja detektira nedefiniranost funkcije na određenom intervalu. Kako je na početku poznato u kojem opsegu se nalaze ulazne varijable, ta informacija se koristi kako bi se izbjegle funkcije koje su nedefinirane. Intervalna aritmetika provodi se na sintaksnom stablu. Kreće se od listova te računa izlaze sve prema korijenu stabla. Ako neko podstablo ima interval beskonačne duljine, to podstablo se jednostavno obriše.

Stablo se mora dva put obilaziti, jednom da se maknu podstabla koja narušavaju definiranost funkcije u intervalu i drugi put kako bi se izračunao konkretan izlaz (ne interval).

U tablici 3.1 su dani neki operatori i funkcije te način na koji se izračunava interval. Problem može stvarati operator množenja koji množi dvije iste varijable. Matematički gledano $x \cdot x = x^2$, ali intervalna aritmetika neće generirati isti interval za ova dva izraza. Zbog toga se preporučuje korištenje funkcije $sqr(x)$ koja može definirati granice intervala koje nisu negativne (prikazano u tablici 3.1).

Iako se intervalna aritmetika lako ugradi u genetsko programiranje, trivijalna implementacija može dovesti do pogrešnih rezultata zbog numeričke pogreške. Zbog toga se preporučuje korištenje gotovih biblioteka koje vode računa o numeričkim pogreškama - primjerice *Interval Arithmetic Library* biblioteka koja se nalazi u sklopu *Boost* biblioteke za *C++* jezik.

Tablica 3.1: Intervalna aritmetika - definicije funkcija. Oznaka g i oznaka d uz varijablu označavaju gornju i donju granicu intervala

Funkcija ili operator	Donja granica	Gornja granica
$x + y$	x_d+y_d	x_g+y_g
$-x$	$-x_g$	$-x_d$
$x \cdot y$	$min(x_d \cdot y_d, x_d \cdot y_g, x_g \cdot y_d, x_g \cdot y_g)$	$max(x_d \cdot y_d, x_d \cdot y_g, x_g \cdot y_d, x_g \cdot y_g)$
e^x	e^{x_d}	e^{x_g}
$1/x$	$min(1/x_d, 1/x_g)$	$max(1/x_d, 1/x_g)$
$sqr(x)$	x_d^2	x_g^2

3.2. Linearno skaliranje

3.2.1. Motivacija

Matematički izrazi (funkcije) $f(x) = x^2$ i $f(x) = x^2 + 100000$ su veoma slični. Druga je funkcija pomaknuta po y osi. Ako su ulazni primjeri generirani iz zadanih funkcija, pokazuje se da prva funkcija ne predstavlja nikakav problem za osnovnu varijantu simboličke regresije te se rješenje pronalazi u prvim iteracijama. Drugi izraz se teško pronalazi osnovnom varijantom simboličke regresije. Problem je u konstanti. Konstante koje se generiraju prilikom traženja matematičkog izraza ili su slučajno generirane ili je to konačan skup. Jako je mala vjerojatnost da se pogodi konkretna konstanta ili da se generira ekvivalentni izraz koji se da svesti na jednostavniji oblik. Ideja linearnog skaliranja je da pokuša ocijeniti oblik funkcije, a ne njezine prave izlaze. Iz primjera se vidi da je oblik funkcije parabola, samo što je pomaknuta po y osi. Umjesto da se izlaz funkcije računa kao $f(x)$ koristi se izraz $a \cdot f(x) + b$, gdje su a i b vrijednosti koje minimiziraju pogrešku. Funkcija definira oblik, a parametri a i b pokušavaju prilagoditi taj oblik ulaznim primjerima (sažimanjem i izduživanjem oblika i pomakom po y osi).

3.2.2. Provedba linearnog skaliranja

Linearno skaliranje ili linearna regresija je matematička metoda koja aproksimira ulazne podatke pravcem s ciljem minimiziranja srednje kvadratne pogreške. Linearno skaliranje se u simboličkoj regresiji koristi kao metoda za prilagođavanje pronađene funkcije ulaznim podacima. Dobivena funkcija $f(x)$ pokušava se prilagoditi tako da se traže parametri a i b koji će minimizirati srednju kvadratnu pogrešku između izlaza funkcije $a \cdot f(x) + b$ i kodomene ulaznih primjera y_i.

Iako linearno skaliranje može biti provedeno u više dimenzija, najjednostavnije je provesti jednodimenzionalno linearno skaliranje. Razlog tome je što za više dimenzijsko linearno skaliranje potrebno računati inverz matrice, što je računski veoma zahtjevno. Za više dimenzija moglo bi se provesti linearno skaliranje takvo da se traže parametri u linearnoj kombinaciji nekoliko funkcija koja minimizira srednju kvadratnu pogrešku, ali to je za sada nedovoljno proučeno područje pa se u praksi još ne koristi. Za linearno skaliranje u tri dimenzije potrebno je uzeti tri funkcije i tražiti parametre a, b, c, d takve da minimiziraju pogrešku između kodomene ulaznih primjera i funkcije: $a \cdot f(x) + b \cdot g(x) + c \cdot h(x) + d$. U nastavku biti će opisan samo postupak za jednu dimenziju.

Računanje dobrote funkcije temelji se na računu srednje kvadratne pogreške. Cilj je minimizirati funkciju dobrote. Ako generirana funkcija ima dobar oblik, ali je krivo skalirana, to će uzrokovati veliku pogrešku i velika je vjerojatnost da takvo rješenje neće biti kao jedinka u zadnjoj iteraciji. Umjesto da se pogreška računa prema izrazu 1.1, računa se prema sljedećem:

$$E = \frac{1}{N}\sum_{i=1}^{N}(y_i - (a \cdot h(\mathbf{x_i}) + b))^2 \tag{3.1}$$

Kako parametri a i b nisu poznati potrebno ih je izračunati. Oni se računaju minimiziranjem izraza 3.1, što je upravo ekvivalentno linearnom skaliranju gdje je $h(\mathbf{x_i})$ domena, a y_i kodomena. Rezultat traženja minimuma pogreške E dan je izrazima:

$$a = \frac{\sum_{i=1}^{N} h(x_i) \cdot y_i - N \cdot \bar{h}\bar{y}}{\sum_{i=1}^{N} h(x_i)^2 - N \cdot \bar{h}^2} \tag{3.2}$$

$$b = \bar{y} - a \cdot \bar{x} \tag{3.3}$$

Izrazi $\bar{h}$ i $\bar{y}$ predstavljaju srednje vrijednosti nad N ulaznih primjera. Ovako dobiveni parametri sigurno minimiziraju srednju kvadratnu pogrešku, pa vrijedi:

$$\frac{1}{N}\sum_{i=1}^{N}(y_i - (a \cdot h(\mathbf{x_i}) + b))^2 \leqslant \frac{1}{N}\sum_{i=1}^{N}(y_i - \cdot h(\mathbf{x_i}))^2 \tag{3.4}$$

U radu [5] detaljno je analizirana ova metoda, te osim nejednakosti 3.4 dane su druge nejednakosti koje vrijede te je provedena detaljna analiza utjecaja linearnog skaliranja u genetskom programiranju.

Iako matematički čisti postupak, pri traženje parametara a i b pomoću računala treba biti oprezan. Opasna operacije koja se može dogoditi je sumiranje brojeva raz-

ličitih redova veličina, što može dovesti do velike numeričke pogreške. Zato je pri svakom sumiranju potrebno sortirati elemente koji se sumiraju, te ih zbrajati od najmanjeg prema najvećem da se postigne najveća moguća preciznost. Ugrađuje se i dodatni mehanizam koji računa varijaciju izlaza funkcije $h((\mathbf{x_i})$, te ako je ta vrijednost veća od 10^7 ili pak manja od 10^{-7} jedinka se odbacuje - razlog je numerička stabilnost.

Vremenska složenost provedbe linearne regresije je $O(N)$. Iako u veliko O notaciji nema razlike, korištenje linearne regresije usporava računanje funkcije dobrote barem za faktor 4 (izrazi 3.2 i 3.3). Kada bi se pazilo na numeričku pogrešku pri sumiranju tada bi vremenska složenost bila $O(N \cdot logN)$ - zbog sortiranja.

4. Paralelizacija

U ovom poglavlju biti će prikazane ideje za paralelizaciju *intervalne aritmetike* i *linearnog skaliranja*.

4.1. Intervalna aritmetika

Paralelizacija intervalne aritmetike na sintaksnom stablu je trivijalna. Najlakše je problem prevesti u model zadataka i kanala. Sintaksno stablo (koje ima sve bridove usmjerene od korijena prema listovima) može se okrenuti tako da se promjene smjerovi svih bridova. Takvo stablo može se direktno predstaviti kao model zadataka i kanala, gdje je zadatak definiran operatorom ili funkcijom, a brid predstavlja slanje poruke u kojoj se nalazi izračunati izlaz. Ako je n broj unutarnjih čvorova stabla, tada je za neparalelizirani obilazak stabla potrebno $O(n)$ koraka. Ako je na raspolaganju dovoljan broj procesora (ne manji od najvećeg broja čvorova u nekoj razini stabla) složenost paralelnog algoritma je $O(log(n))$. Paralelizacija intervalne aritmetike se isplati samo ako su sintaksna stabla za određeni problem dovoljno velika, što najčešće nije slučaj.

4.2. Linearno skaliranje

Na paralelizaciju linearnog skaliranja može se gledati na dva načina. Prvi pogled je da se paralelizira računanje dobrote jedinki, što je često radi pri paralelizaciji evolucijskih algoritma. Drugi pogled na paralelizaciju je paraleliziranje operacije sumiranja, kojih ima više nego pri samom računanju srednje kvadratne pogreške. Prije svakog sumiranja potrebno je sortirati podatke zbog numeričke pogreške. Bez ulaženja u konkretnu implementaciju sortiranja, paralelno sortiranje se može svesti na složenost $O(\frac{n}{p} \cdot log(\frac{n}{p}))$, gdje je n broj elemenata koji se sortiraju, a p broj procesora. Suma se primjenom algoritma redukcije može svesti na istu složenost - $O(\frac{n}{p} \cdot log(\frac{n}{p}))$.

Ako je potrebno odabrati jedan pogled paralelizacije potrebno je pažljivo analizirati problem. Ako je broj jedinki puno veći od broja ulaznih primjera tada se više

isplati raditi paralelizaciju nad jedinkama. Ako ima relativno malo jedinki, a puno više ulaznih primjera tada je bolje paralelizirati računanje suma.

5. Praktična primjena

U ovom poglavlju daju se primjeri korištenja simboličke regresije u praksi, konkretno opisat će se metoda korištena u [3]. Simbolička regresija može se koristiti u bilo kojem problemu gdje treba izgraditi model, tj. matematički izraz.

5.1. Izvlačenje prirodnih zakona iz podataka

Postoje sustavi za koje je lako definirati matematički model. Problem nastaje kada se želi izlaz sustava prikazati u eksplicitnom obliku. Postoje numeričke metode koje mogu izračunati izlaz za neku ulaznu vrijednost, ali analitičko rješenje problema često ne postoji. Simbolička regresija u tom smislu ima ulogu generirati matematički izraz koji bi najbolje opisivao izlaz sustava. Za čovjeka je takav izlaz prihvatljiviji jer može iz generiranog izraza tumačiti (bolje analizirati) izlaz sustava, što može dovesti do drugačijeg pristupa rješavanju problema.

U [3] opisano je kako iz ulaznih podataka dobiti matematički model koji ih opisuje. Iako se čini da je problem ekvivalentan prethodno opisanom, razlika je u tome što se traže veze između varijabli (parcijalne derivacije), a ne izlaz sustava. Simbolička regresija se koristi kao dio algoritma.

Koraci algoritma su:

1. Prikupljanje podataka iz fizikalnog sustava
2. Numerički izračun parcijalnih derivacija iz prikupljenih podataka
3. Traženje matematičke funkcije koja povezuje parcijalne derivacije
4. Ocjena dobivene funkcije
5. Analiza najboljih funkcija

Nakon što se prikupe podaci, numerički se računaju parcijalne derivacije kao $\frac{\Delta y}{\Delta x}$. Gdje su x i y parovi mjerenih veličina. Traži se funkcija (npr. dvije varijable) $f(x, y)$

koja će najbolje opisati parcijalne derivacije. Funkcija se traži simboličkom regresijom, a ocjena funkcije (dobrota) daje se na temelju parcijalnih derivacija. Za primjer dvije varijable funkcija se ocjenjuje prema razlici:

$$\frac{\Delta y}{\Delta x} - \frac{\frac{\partial f}{\partial x}}{\frac{\partial f}{\partial y}} \tag{5.1}$$

Umjesto da se traži kako dobro je opisan izlaz funkcije, traži se kako dobro funkcija povezuje varijable, tj. informacije koje su skrivene u podacima. Simbolička regresija na kraju daje skup najboljih funkcija koje je pronašla. Analizom rješenja čovjek odabire funkcije koje imaju najviše smisla. U eksperimentima koji su provedeni pokazalo se da postupak može dati izlaze koji su aproksimacije fizikalnih zakona. Ako se zna da je neki sustav opisan s $sin(x)$, a ulazni podaci su bili relativno mali brojevi, tada izlaz ne mora biti $sin(x)$ već može biti i samo x, što je dobra aproksimacija za mali x te se takvi rezultati mogu iskoristiti za lakšu analizu kompleksnih sustava.

6. Zaključak

Za potrebe traženja matematičkih izraza simbolička regresije kao metoda (bez ulaženja u implementaciju) veoma je koristan alat u mnogim područjima. Problemi koji se javljaju prilikom implementacije su i danas aktualni i ovo je područje veoma aktivno. Ipak treba biti oprezan prilikom korištenja simboličke regresije jer ako je za neki problem poznato njegovo ponašanje (model) tada je bolje tražiti parametre modela, npr. linearnom regresijom. Kada je model nepoznat i žele se izvući informacije iz podataka tada je simbolička regresija dobar alat. Osim što omogućava generalizaciju iz opisanih primjera na matematički izraz, sam simbolički prikaz matematičkog izraza čovjeku omogućava lakšu interpretaciju rezultata. Danas se za ostvarivanje simboličke regresije koriste računalne metode kao što su evolucijski algoritmi koji se nadograđuju i prilagođavaju u svrhu dobivanja što boljih rezultata.

7. Literatura

[1] D. Augusto and H. Barbosa, "Symbolic regression via genetic programming," *Neural Networks, 2000. Proceedings. Sixth Brazilian Symposium on*, vol. 1, no. 1, pp. 173–178, 2000.

[2] R. S. R. J. Ivan Zelinka, Donald Davendra and Z. Oplatkova, "Analytical programming - a novel approach for evolutionary synthesis of symbolic structures," in *Evolutionary Algorithms* (E. Kita, ed.), 2011.

[3] M. Schmidt and H. Lipson, "Distilling Free-Form Natural Laws from Experimental Data," *Science*, vol. 324, pp. 81–85, Apr. 2009.

[4] M. O'Neill and C. Ryan, "Grammatical evolution," 2001.

[5] M. Keijzer, "Scaled symbolic regression," *Genetic Programming and Evolvable Machines*, vol. 5, pp. 259–269, Sept. 2004.

[6] M. Keijzer, "Improving symbolic regression with interval arithmetic and linear scaling," in *Genetic Programming, Proceedings of EuroGP'2003* (C. Ryan, T. Soule, M. Keijzer, E. Tsang, R. Poli, and E. Costa, eds.), vol. 2610 of *LNCS*, (Essex), pp. 70–82, Springer-Verlag, 14-16 Apr. 2003.