Četrta domača naloga

Tadej Petrič

May 21, 2023

1 Prva

Za odkrivanje enačbe sem uporabil pristop z genetskimi algoritmi, z orodjem PySR. To orodje sem izbral, ker nisem vedel dovolj natančno kakšno strukturo ima enačba (da bi lahko efektivno uporabljal gramatike ali linearno regresijo), še vedno pa mi obogoča vpeljavo nekaj predznanja (preko izbire operatorjev) - orodje je zelo fleksibilno.

Nastavitve

- niterations: več kot je, bolje. 3000 je meja, pred katero se ponavadi enačba že ustali
- binary operators: omogočam vse osnovne operacije razen potenc, ki ponavadi niso prisotne v fizikalnih problemih z realnimi števili. Še posebaj pomemben je $-$, saj je toplotni tok odvisnen od razlike.
- unary operators: dodamo cos in sin zaradi kota vetra (iz domenskega znanja pričakujemo sin, vendar bi lahko prišel tudi cos). Dodamo osnovne potence (kvadrat in kub), saj je razmerje med $T_w - T_a$ in toplotnim tokom nelinearno. Dodamo x^{-1}, če bi to potreboval izolacijski indeks. Čeprav so zadnje tri dosegljive že z binarnimi operacijami jih je zelo smiselno dodati kot unarne operacije, da jim povišamo verjetnost pojavitve.
- denoise: podatki imajo malo šuma, torej jih razšumimo. V praksi se ne pozna velika sprememba
- complexity of operators: bolj kompleksnim operacijam zmanjšamo verjetnost pojavljanja.
- weight randomize: povišamo, da povečamo iskalni prostor
- turbo: pohitri program okoli 30%
- maxdepth: omejimo kompleksnost enačbe na bolj enostavne, saj je to enostaven fizikalen pojav

- procs, multithreading, random state, deterministic: potrebno za deterministično izvajanje

Omenim še, da je smiselno, da je kompleksnost kvadriranja kar 2, množenja pa le 0.3 (kljub temu, da je kvadriranje le bolj enostavno množenje), saj v praksi kvadriramo mnogo členov, množimo pa le enega. Tako ima $(a+b+c)\cdot(d+e+f)$ veliko višjo kompleksnost kot $a \cdot d$ (kompleksnost narašča kvadratno, medtem ko pri kvadriranju le linearno glede na gnezdene člene)

Iskanje je implementirano v `pysrpy.py`, preostanek uporabe pa v zvezku `resitev.ipynb`. Najboljše enačbe genetskega najdemo v `hall_of_fame.csv`.

Dobim enačbo

$$(((2.5343432 - \eta)(T_w - T_a)0.027452817))^2 \sin(\theta)$$

Vidimo, da ta enačba ustreza vsem pogojem domenskega predznanja. Theta je v sinusu (ki je ena za pravokotne θ), tok je odvisen od nelinearne razlike $\sim (T_w - T_a)^2$, višji izolacijski indeks pa povzroči nižje toplotne tokove (kjer upoštevamo, da je indeks vedno manjši od 1).

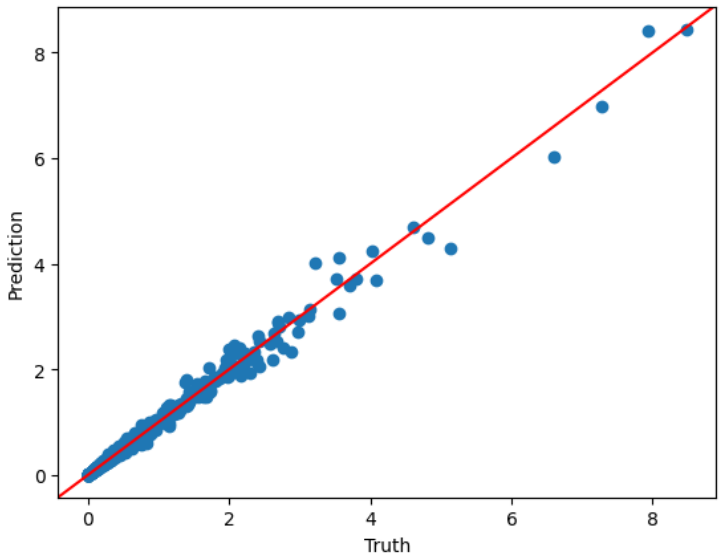

Figure 1: Rezultat genetskega algoritma

Sedaj, ko vemo približno obliko enačbe, uporabimo še linearno regresijo za sanity check: enačbo razstavimo v polinom, parametriziramo konstante in jih poiščemo z linearno regresijo (in regularizirano linearno regresijo). Vidimo, da v vsakem primeru dobimo podobno kot izhod genetskega algoritma ter vsi pristopi dobro opisujejo podatke (kot vidimo iz grafov, kjer je popolno prileganje premica $y = x$). Lahko bi (morali, za objavljen članek. Še posebaj za regularizirane metode lasso in ridge) še dodatno uporabili train-test split, ampak nimamo

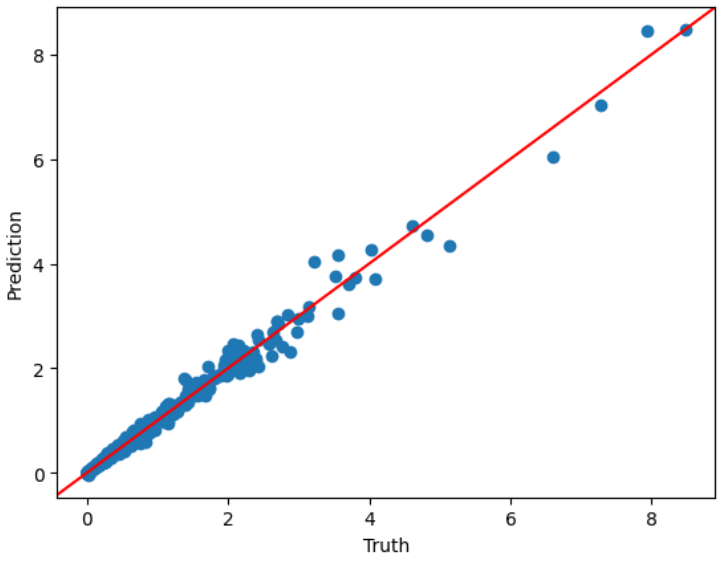

Figure 2: Rezultat linearne regresije

veliko podatkov in rezultati pridejo podobni. Zaradi omenjenih razlogov (več pristopov vodi k isti enačbi, enačba ustreza kriterijem predznanja, ki niso bila uporabljena za pripravo problema, dobro prileganje rešitvi, ...) enačbi še kar dobro zaupam. Verjetno se pa konstante malo razlikujejo od originala (ugibam, da je $2.5343432 = 2.5$ in $0.027452817 = \sqrt{0.00075}$).

2 Druga

Opis datotek

V datoteki `make_data.py` sestavim podatkovne strukture in pripravim podatke. Spišem tudi svojo funkcijo za generiranje podatkov, ker sem slep in sem prepozno prebral, da je že implementirana v domači nalogi (je pa zato zelo enostavno delati nove probleme). Razred `problem` sprejme poljubno funkcijo $\mathbb{R}^n \to \mathbb{R}$, na kateri smo uporabili dekorator `@pandise` (ki funkcijo pretvori v funkcijo pandasovih tabel) in inicializira verjetnosti (in spremenljivke) glede na število parametrov, ki jih sprejme funkcija. Ko pokličemo metodo `make_data` se inicializirajo še podatki. Verjetnosti lahko prosto posodabljamo s spreminjanjem parametrov (npr `problem.E[0]` opisuje verjetnost izbire operatorja +). Gramatiko trenutnih generiramo z `gen_grammar` (kjer pazimo, da izpisujemo verjetnosti v visoki natančnosti v ne-znanstveni notaciji zaradi proGEDa. Bilo bi lepo, če bi lahko izpustil eno vrednost v gramatiki in se ta sama nastavi na $1 - \text{total}$). Metoda `parse` pretvori kodo drevesa v slovar, ki pove kolikokrat je bilo kakšno pravilo uporabljeno (enostaven stack machine in case analysis). V

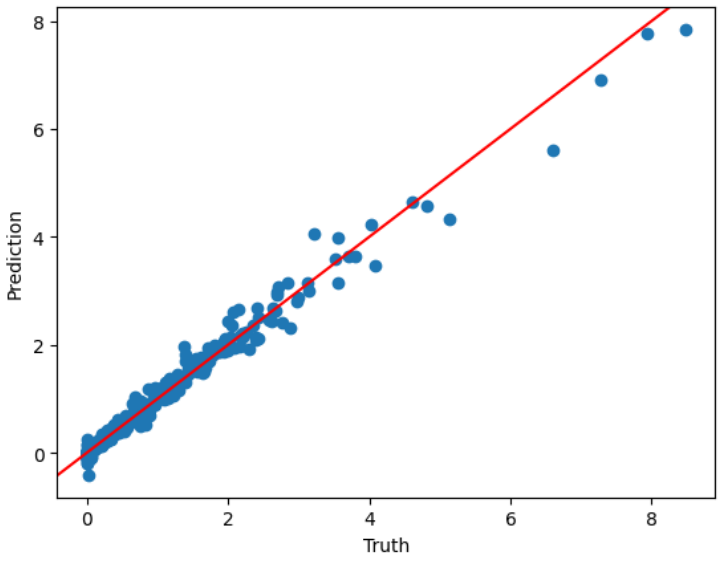

Figure 3: Rezultat ridge regresije, $\lambda = 100000$

datoteki je tudi razred `leaderboard`, ki shranjuje najboljših n različnih (glede na drevo izpeljave) izrazov urejenih po napaki.

V datoteki `generation.py` je implementiran ključni del algoritma. Funkcija `evolve` sprejme objekt `problem` in na njem izvaja `gens` mnogo iteracij funkcije `one_generation`. Vsaka generacija popravi verjetnosti na nekaj boljšega. Za primerjavo je tukaj še funkcija `dummy`, ki uteži ne popravlja in jih pusti enako porazdeljene kot prej (v mojem primeru enakomerno).

V datoteki `save_average.py` inicializiram podatke ter 10x poženem moj algoritem in shranim rezultate. Shranim rezultate tudi za neodvisno vzorčenje. To traja več ur (hitreje bi delalo, če bi lahko le posodobil gramatiko in ohranil modele, saj gre verjetno večino časa za fittanje podatkov).

Opis algoritma

Algoritem deluje iterativno z izboljšavo glede na elito.

Ko inicializiramo podatke so verjetnosti v gramatiki enakomerno porazdeljene. Potem najdemo najboljših n rezultatov in jih shranimo v objekt `leaderboard`.

Nato se sprehodimo po teh najboljših rezultatih. Za vsak rezultat shranimo napako ter nastavimo $s = f(e)$ kjer je e napaka, f hitrost učenja in s hitrost. To nam omogoča, da spremenimo hitrost učenja glede na napako - manjše napake (boljše enačbe) bodo bolj vplivale na verjetnosti kot večje napake in dale večji popravek. Več podrobnosti o izbiri funkcije f kasneje.

Ko imamo hitrost učenja, izračunamo potreben popravek. Preštejemo kolikokrat smo videli različne simbole s `parse` ter sestavimo to v vektor (če ne

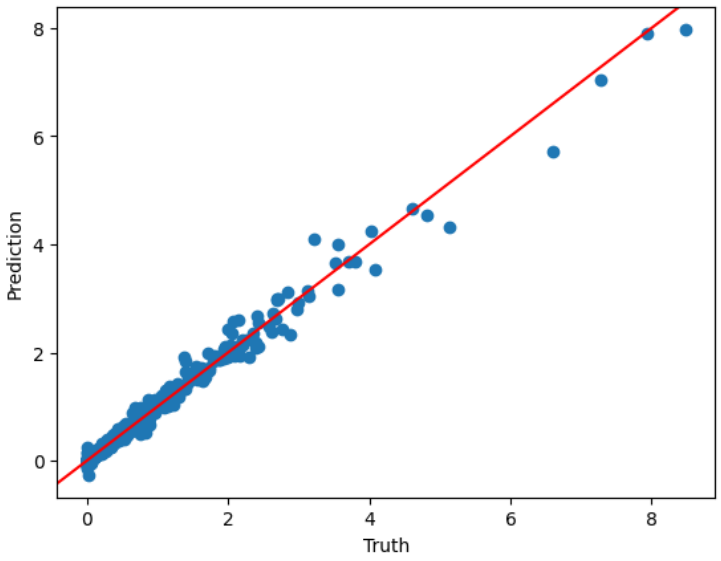

Figure 4: Rezultat lasso regresije, $\lambda = 500$

bi bilo rekurzivnih pravil bi dobili nekakšen MLE estimator, sedaj je pa bolj hevristično). Torej če smo videli v naši dobri enačbi + trikrat, − dvakrat in F petkrat naredimo vektor

$$E_2 = (3 \cdot 2, 2 \cdot 2)/(3 \cdot 2 + 2 \cdot 2 + 5).$$

Simbol za + in − obtežimo z 2, saj vsaka enačba vedno enkrat uporabi pravilo za F. Tako poudarimo, da je v izrazu $x+y$ ključni del vsota (in šele *za* generiranjem vsote se splača generirati F - nastavimo "priority") ter je potrebno povišati verjetnos za ta konstruktor. Podobno za ostala pravila.

Nato popravimo verjetnosti s konveksno kombinacijo $E = E(1-s) + E_2(s)$, kjer je E_2 vektor iz prejšnjega odstavka (podobno naredimo za F, T, V). Konveksna kombinacija dveh elementov z 1-normo 1 nam poskrbi, da dobimo nov element velikosti 1. Prav tako povečamo verjetnost za pravila, ki smo jih večkrat srečali. Geometrično gledano se premaknemo v smer uspešnih verjetnosti za s. Večji kot je s, več se premaknemo. Tukaj omenim še izbiro funkcije hitrosti f. Želimo funkcijo, ki ima vrednost blizu 1 za napake okoli 0 in vrednost blizu 0 za slabe enačbe. Točni parametri so odvisni od enačbe, jaz sem implementiral $(\tanh(-\log(e)/k) + 1)/2$ (kjer je k parameter konvergence. Višji kot je, hitreje konvergira. Lahko ga nastavimo, da se spreminja glede na generacijo, kar nam omogoča, da prvih nekaj generacij bolj spremeni uteži), $1/(x+1)^k$ in konstantno hitrost. Za prvi in drugi problem je boljša prva funkcija, za tretji problem je boljša zadnja funkcija (saj so napake pri tretjem problemu zelo majhne, okoli 0.5 za slabe rezultate, prva funkcija bo pa to razumela kot dober rezultat ne glede na parametre).

To zaključi eno iteracijo algoritma. Ta isti postopek ponavljamo do konvergence, kjer pa upoštevamo, da ne izboljšujemo po najboljših enačbah v generaciji ampak po najboljši videni enačbi na sploh (gledano genetsko, poskrbimo, da najboljših n enačb vedno preživi generacijo).

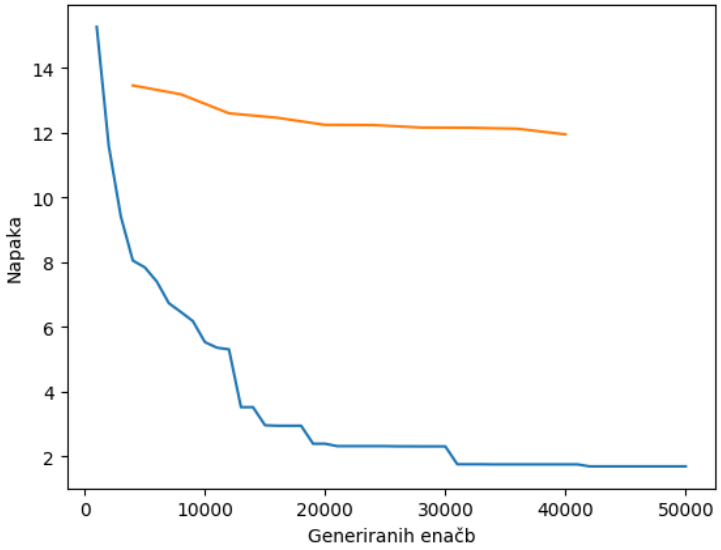

Figure 5: Primerjava za $y = x_1 - 3x_2 - x_3 - x_5$

Na grafih je vedno oranžna barva enakomerna gramatika, modra pa moja izboljšava. Zaradi hitrosti sem kdaj uporabil večji batch size za neodvisno vzorčenje (po definiciji neodvisnosti to ne vpliva na končni rezultat, le na gladkost krivulje) (zato izgleda, da se kdaj krivulje začnejo kasneje). Kdaj sem jim tudi dal malo manj vzorcev, ker generiranje traja več ur ampak to ne spremeni zaključka (zato pri prvi izgleda, da se oranžne krivulje končajo prej). Preostanek grafov govori sam zase, mojemu algoritmu napaka pada veliko hitreje (skriva pa obnašanje v neskončnosti. Ampak nimam neskončno časa torej bo ta graf tudi ostal skrit).

Diskusija

Algoritem veliko izboljša delovanje programa v primerjavi z enakomerno porazdeljeno gramatiko. Slabost pa je, da je bolj odvisen od začetnih hiperparametrov (sploh v tretji enačbi). Za razliko od enakomerno porazdeljenega Monte-Carlo algoritma nam konvergenca ni zagotovljena z verjetnostjo 1, saj je algoritem občutljiv na lokalne minimume. Če bo dolgo časa najboljša enačba imela samo $'x'_0$, bo nastavil verjetnost generiranja $'x'_0$ na ≈ 1 in ne bomo nikoli generirali $'x'_1$. To lahko preprečimo z manj agresivno hitrostjo nastavljanja napake ali pa s povečanjem elite (če hranimo najboljšo enačbo je zelo verjetno, da

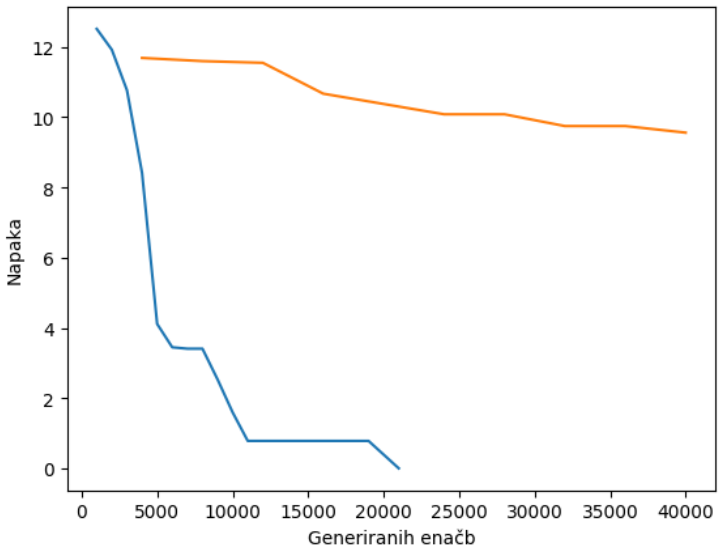

Figure 6: Primerjava za $y = x_1^5 x_2^3$

dobimo nekaj neugodnega. Če hranimo najboljših 5 je manj verjetno, da so *vsi* neugodni). Lahko bi tudi nastavili minimalno verjetnost konstruktorjev (torej, da verjetnost pravila ne more pasti pod npr. 5%).

Povezano z lokalnimi minimumi je tudi to, da enačbe niso čisto zvezne glede na pravila. Lahko se zgodi, da iščemo npr. enačbo oblike $f(g(x))$ ampak in $f(x)$ in $g(x)$ sama sta veliko slabša kot samo x. To lahko povzroči, da ju algoritem izloči prehitro in se ju ne nauči. To je tudi problem pri 3. enačbi, če podatkovje generiramo na napačen način (saj lahko prevladujejo $\sin(x_0), x_0, x_0 \sin(x_0)$, odvisno kolikšen del x osi vzamemo za vzorec in če vzorčimo enakomerno). Temu se lahko izognemo tako, da generiramo več enačb na generacijo. Slabost tega je, da je algoritem potem počasnejši.

Kljub naštetim pomankljivostim algoritem dela dobro. V mojih poskusih namreč klasično vzorčenje z enakomerno porazdelitvijo nikoli ne konvergira do rezultata, medtem ko moj algoritem ponavadi konvergira (v prvi nalogi v 70% primerov, v drugi 100%, v tretji pa v 40% primerov (in dodatno en primer zgreši le za 0.09). Nekateri primeri bi še lahko konvergirali ampak sem omejil število korakov v algoritmu). Težave ima le pri tretji, saj pogosto eliminira $'x_1'$ prehitro ampak to bi lahko popravil z bolj skrbno hiperparametrov. Kljub temu dela bolje kot privzeta metoda.

Algoritem bi lahko izboljšali na več načinov. Lahko bi drugače računali napako (npr. upoštevamo še odvod funkcije), lahko bi bolje izbrali funkcijo hitrosti učenja. Popravljanje je podobno stohastičnemu gradientnemu spustu, torej bi lahko izbrali veliko popravkov iz tam. Najbolj zanimiva opcija, pa bi bila prek reinforcement učenja (npr. deep Q learning). Cilj bi bil, da naredimo nevron-

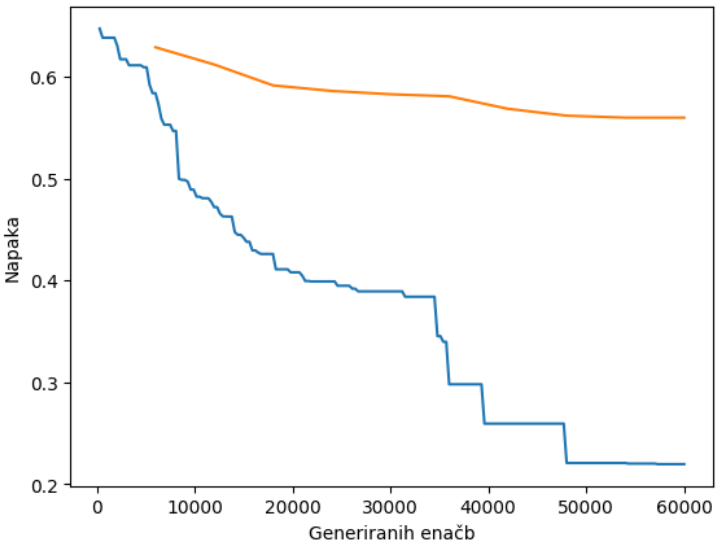

Figure 7: Primerjava za $y = \sin(x_1) + \sin(x_2/x_1^2)$

sko mrežo katere izhodi so uteži gramatike, funkcija napake pa bi bila kako dobro se top n generiranih funkcij prilega danim podatkom. Ta način (kombiniranja nevronskih in gramatik) učenja ima kar nekaj prednosti, saj je gramatika spremenila problem v strukturiranega (fiksno število parametrov, verjetnosti v gramatiki, za izhod) in gramatika preprečuje preprileganje (zaradi težnje k bolj enostavnim izrazom in stohastične narave), moč in fleksibilnost nevronskih mrež pa omogočata bolj inteligentno nastavljanje parametrov.

Sledijo še možne tehnične izboljšave: multithreading bi bilo enostavno dodati v program, še bolj kot to pa bi pomagalo, če bi lahko le spreminjal gramatiko in ohranil modele. Ker se pravila gramatike ne spreminjajo, bi tudi rezultat fittanja ostal enak (kar je najbolj časovno potraten del programa). Fittati bi bilo potrebno le novo-odkrite enačbe (glede na unikatno odkrite sympy izraze). Sicer bi se spremenil `model.p` tekom evolucije ampak to ne vpliva na rezultat. Predvidevam, da bi to skrajšalo čas za 90%.