# Genetic programming for symbolic regression

Petar Pavlović

## 1. Postavka problema:

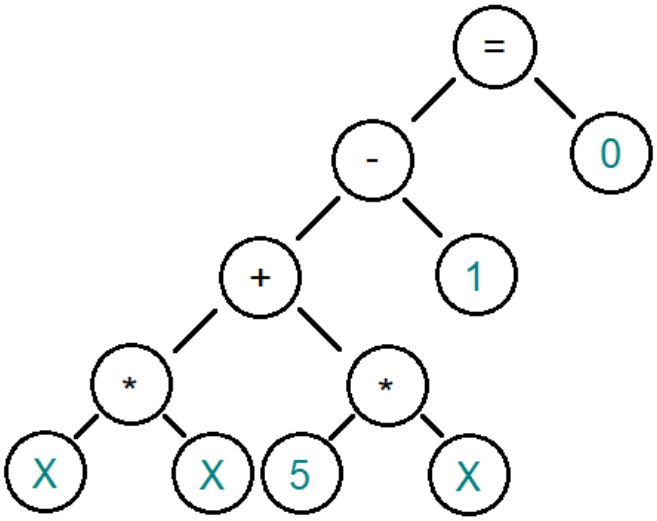
Prvobitan problem je bio zabavne prirode. Ideja je bila da se napravi program koji bi rešavao jednu od igara iz emisije “Slagalica”, a program je bio namenjen za rešavanje specifično igre “Moj broj”. Ovaj program mada simpatičan po svojoj prirodi nije koristan u “naučne” svrhe, odnosno optimizacioni metod koji je korišćen, genetsko programiranje, je malo prekomplikovan pristup rešavanja ovog problema zbog njegove maksimalne veličine drveta koje se gradi.

Zbog prethodno navedenog “problema“ niske kompleksnosti, radi povećanja kompleksnosti i uopštavanja datog problema prelazimo na genetsko programiranje za simboličku regresiju. Ideja jeste da date podatke najbolje predstavimo nekom funkcijom, vršimo simboličku regresiju, koju dobijamo genetskim programiranjem.

Poznat je problem linearne regresije, predstavljanje podataka korišćenjem linearne funkcije sa minimalnom greškom, a zatim korišćenje iste radi predviđanja vrednosti za neke nepoznate podatke. U našem slučaju radimo nešto slično ali korišćenjem raznih funkcija koje imamo u bazi, kao što su sinus, kosinus, logaritam, množenje, deljenje... Zašto je ovakav pristup bolji od pristupa linearne ili logističke regresije? Zašto je dati pristup bolji od korišćenja neke neuronske mreže? Prvo da odgovorimo na drugo pitanje. Idejno duboko učenje, neuronske mreže, su pristrasne u pravljenju rešenja koje imaju glatke funkcije dok u rešenju simboličke regresije možemo dobiti i funkcije koje bolje određuju podatke zbog osobine da budu prekidne, oštrih ivica. Odgovor na prvo pitanje slično kao za drugo, mi ne možemo dobiti rešenja koja nisu neprekidna korišćenjem linearne ili logističke regresije. Princip simboličke regresije je dosta brži od dubokog učenja, sporiji od regresija ali u svakom slučaju može mnogo bolje da predstavi podatke.

## 2. Pristup rešavanju postavljenog problema:

Sad ćemo malo više da opišemo način rešavanja problema i ideje. Još jednom šta je ideja simboličke regresije? – Idejno kombinacijom različitih simbola, odnosno funkcija, promenljivih i konstanti gradimo jednačinu koja najbolje predstavlja date podatke. Kako se bavimo jednačinama jasno je da moramo dato rešenje predstavljati korišćenjem sintaksnog stabla.




Slično definišemo i naša stabla. Sada kada znamo kako predstavljamo naša rešenja na dalje moramo da razumemo pristup genetskog programiranja.

### 2.1. Genetsko programiranje:

Genetski algoritmi oponašaju prirodnu evoluciju populacije, ili bar predlog evolucije koju je dao Darvin. Kod genetskih algoritama ceo proces se izvodi nad **genomima**. Genom ili DNK lanac koji nosi informacije o izgradnji neke jedinke, odnosno kako se formiraju proteini. U genetskim algoritmima imamo genome koji su u obliku odabrane strukture podataka. To su uglavnom nizovi bitova, karaktera, celih brojeva.... a mogu biti i stabla koja se koriste u genetskom programiranju.
Genomi se sastoje iz **gena**, kod nas gen je jedan član niza ili jedan čvor stabla. Kao u evoluciji na dalje se prenose oni geni jedinki koje po svojim osobinama imaju najbolje šanse da prežive. Evolucija se dešava tako što kroz generacije, ili epohe, vršimo dve operacije **ukrštanje** i **mutaciju**.
**Ukrštanje** je simulacija razmnožavanja dve jedinke i nastajanje dve nove jedinke, služi kao eksploatacija prostora na osnovu onih informacija koje već imamo, odnosno rekombinovanje istih. Koje dve jedinke ćemo ukrstiti biramo **turnirom**. Turnir je proces u kom od izabranih **k** jedinki biramo one dve najbolje i te dve vrše ukrštanje. Naravno ukoliko stavimo da je **k = n** (n je veličina populacije) uvek će biti izabrane one dve jedinke koje su najbolje. Testiranje kvaliteta jedinke se predstavlja **funkcijom cilja** (**fitness** funkcija). Fitness funkcija nam govori koliko je data jedinka, rešenje, kvalitetna u smislu koliko nam dobro rešenje daje. Kvalitet rešenja zavisi od problema do problema. Što je vrednost ove funkcije veća to je rešenje bolje.
Nakon ukrštanje vršimo mutaciju. **Mutacija** je operacija koja može, a i ne mora da se izvrši, služi kao eksploracija prostora, odnosno uvodi neke nove osobine koje se do sada nisu pojavile u rešenju.

Kada znamo sve operacije sama petlja koja simulira evoluciju je jednostavna:

**inicijalizujemo populaciju P(n), T(n)**
**while** nije ispunjen uslov zaustavljanja**:**

```
for i = 0; i = i + 2, i < n:
        p1, p2 = turnir(P(n))
        c1, c2 = ukrstanje(p1, p2)
        mutiraj(c1)
        mutiraj(c2)
        T[i] = c1
        T[i+1] = c2

P(n) = T(n)
```

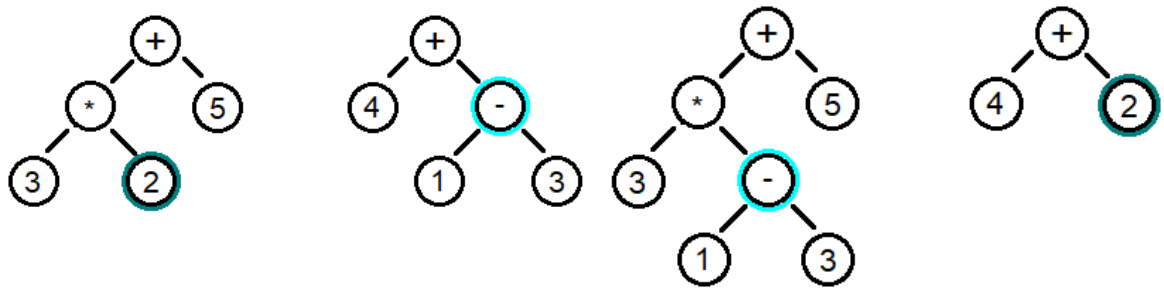
**end**

T je generacija koja je napravljena u ovoj epohi, n predstavlja kolika je populacija. Uslov zaustavljanja može da bude neki predodređeni broj epoha (unapred određen broj iteracija), kvalitet fitness funkcije je dostigao neku vrednost, vreme izvršavanja je isteklo ....

Kada znamo osnovni genetski algoritam razumevanje genetskog programiranja nije toliko teško. Sama petlja izvršavanja je ista, jedina razlika jeste predstavljanje rešenja i logika funkcija ukrštanja i mutacije. U genetskom programiranju rešenja su u obliku drveta, jer se na taj način mogu rekreirati programi. Programe prilikom obrade kompajleri grade sintaksna stabla koja predsatavljaju programe i funkcije koje se dešavaju unutar programa. Slično mi želimo da naš program gradi nova stabla ukrštanjem i izmenom njegovih podstabala, poštovajući sintaksna pravila.

U genetskom programiranju jedan od problema jeste dubina stabla, jer jako brzo mogu postati preduboka i preširoka što usporava rad programa a opet ne doprinosi mnogo samom rešenju. Ovaj problem možemo rešiti uvođenjem provere koja posmatra veličinu stabla i daje lošu fitness vrednost onim rešenjima čija su stabla prevelika. U prvoj verziji ovog problema, “Moj broj” igra, uvek se znalo da će postojati 6 konstantni koje možemo da iskoristimo u kombinaciji sa osnovnim matematičkim operacijama, u cilju pronalaženja jednačine koja daje neki traženi broj. Kako smo znali koliko konstantni se može pojaviti možemo pretpostaviti da ako se koriste binarne operacije recimo sabiranje, u najgorem slučaju ako su sve konstante iskorišćene imaćemo 6 listova pa samim tim ukupno 11 čvorova, odnosno 2n-1 čvorova (n  predstavlja broj listova).

Ukrštanja:



## 2.2. Izrada koda:

Od samog početka ideja je bila da imam tri klase, koje dele posao u ”celine”.

### *2.2.1. Node:*

**Node** klasa  predstavlja drveta. Ideja je da u ovoj klasi imam definisane sve funkcije koje se mogu koristiti, kao i pravila gradnje drveta i ograničenja dozvoljenih funkcija. Pod ograničenja mislim na deljenje sa nulom, postojanje imaginarnih brojeva i sl.

Unutar ove klase definisani su i **value** vrednost koja evaluira samo drvo. U verziji nakon uvećanja problema, prilikom uvođenja promenljivih više nije moguće izračunati rešenja odmah. Uvodim novu funkciju koja računa rešenja za sve vrednosti promenljivih prosleđenih podataka.

Klasa sadrži vrednost veličine stabla, broj čvorova pretstavljena je sa **size**. Čvor je funkcija koja kao decu ima druge čvorove koji mogu biti funkcije ili listovi.

Problem na koji sam naišao prilikom izrade ove klase jeste stepenovanje negativnih brojeva, više korenovanje negativnih brojeva. Ukoliko se stepenuje negativan broj nekom realnom vrednošću ne postoji način da se zaključi unapred da li će taj broj biti compleksan ili ne. Zbog ovoga pojavljivale su se kompleksne vrednosti tamo gde ih ne očekujemo, npr. -1 stepenovan sa 1/3 što je -1 ali kako se 1/3 predstavlja kao 0.33333333... python odmah računa ovo kao imaginarnu vrednost. Problem je delimično „rešen“ uvođenjem klase **Fractures** koja pronalazi, do na grešku, racionalni broj koji najbolje predstavlja dati realan broj. Imajući ovo uzimam delilac i proveravam da li je paran ili neparan na osnovu čega određujem da li postoji koren datog negativnog broja. Ako je neparan stepenujem apsolutnu vrednost datog negativnog broj a onda samo dodam minus ispred.... Vrlo upitno rešenje znam ☺.

### *2.2.2. Genome:*

**Genome** klasa vrši sve one operacije koje smo naveli u poglavlju genetskog programiranja. Dakle, mutacija i ukrštanje su definisani u ovoj klasi. Same funkcije nisu toliko kompleksne jer je njihova funkcionalnost obezbeđena pravilnim radom funkcija unutar node klase.

Node klasa ne sadrži način za građenje novih rešenja, za to je zadužena genome klasa. Na osnovu dobijenih vrednosti promenljivih i evaluira se greška datog rešenja koristeći MSE.

Ova klasa vodi računa o konstantama koje se pojavljuju u stablima kao i o tome da li data vrednost sadrži sve promenljive. Promenljive moraju da se pojave u rešenjima jer u suprotnom funkcija koju dobijemo ne zavisi od podataka, što nama ne treba. Ovakvim rešenjima se daje kazna tako što im se da beskonačno velika greška. Iz prethodnog teksta može se pretpostaviti da je naš fitness u stvari greška, odstupanje podataka od funkcije koju smo pronašli, samim tim mi tražimo minimum fitness ne maksimum.

### *2.2.3 GenAlg:*

**GenAlg** je klasa koja u sebi sadrži osobinu ukrštanja i turnira kao i samu populaciju. Unutar ove klase nakon što je inicijalizujemo se gradi populacija. Posebnom funkcijom **findSolution** započinjemo petlju koja se izvršava dok fitness ne padne ispod neke konstante vrednosti ili dok ne istekne broj epoha koje smo zadali po inicijalizaciji populacije.

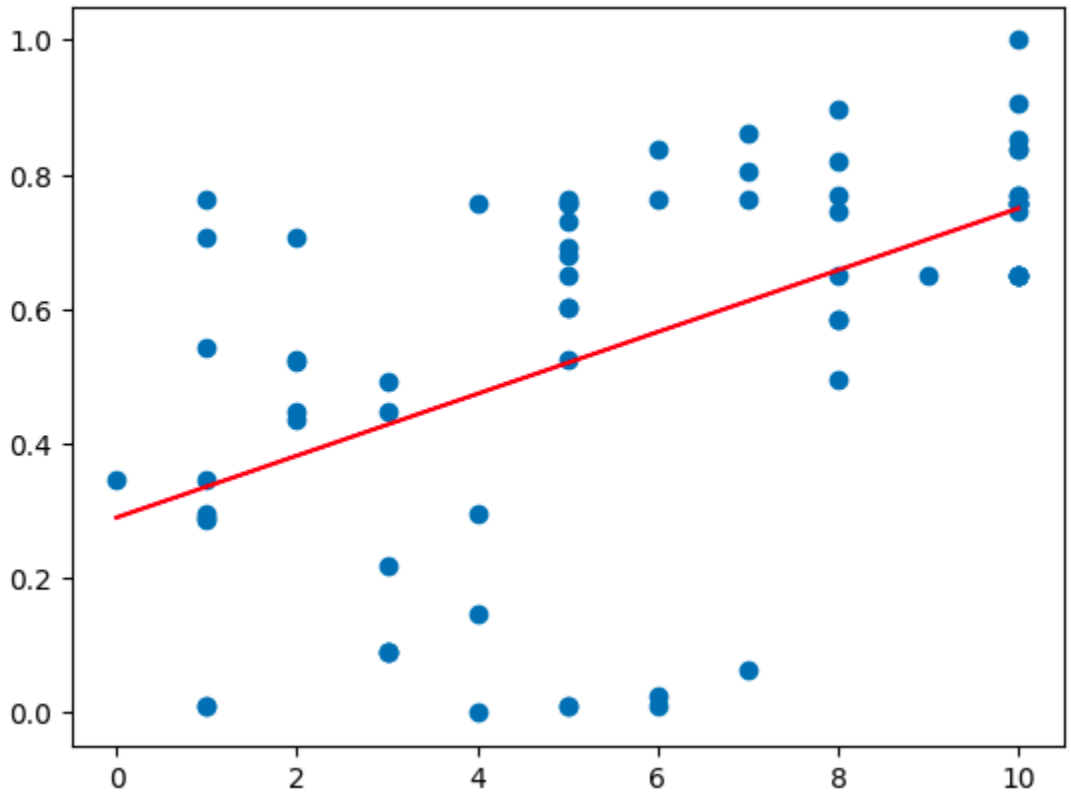
## 3. Eksperimenti:

Prvobitno program je isprobavan nad nekim proizvoljnim funkcijama. Program je vraćao precizne vrednosti za tipične funkcije kao što su x*x, sin(x), cos(x)… Za funkcije kao što su cos(x) + sin(x) ili čak (x + y) i sl. vraćane su funkcije koje dosta liče na traženu kada se plotuju ali umele su da budu dosta komplikovanije od funkcije čije smo tačke prosledili.

Nakon inicijalnih testova isproban je nad nekim manjim podacima i uglavnom uspeva dobro da predstavi podatke...

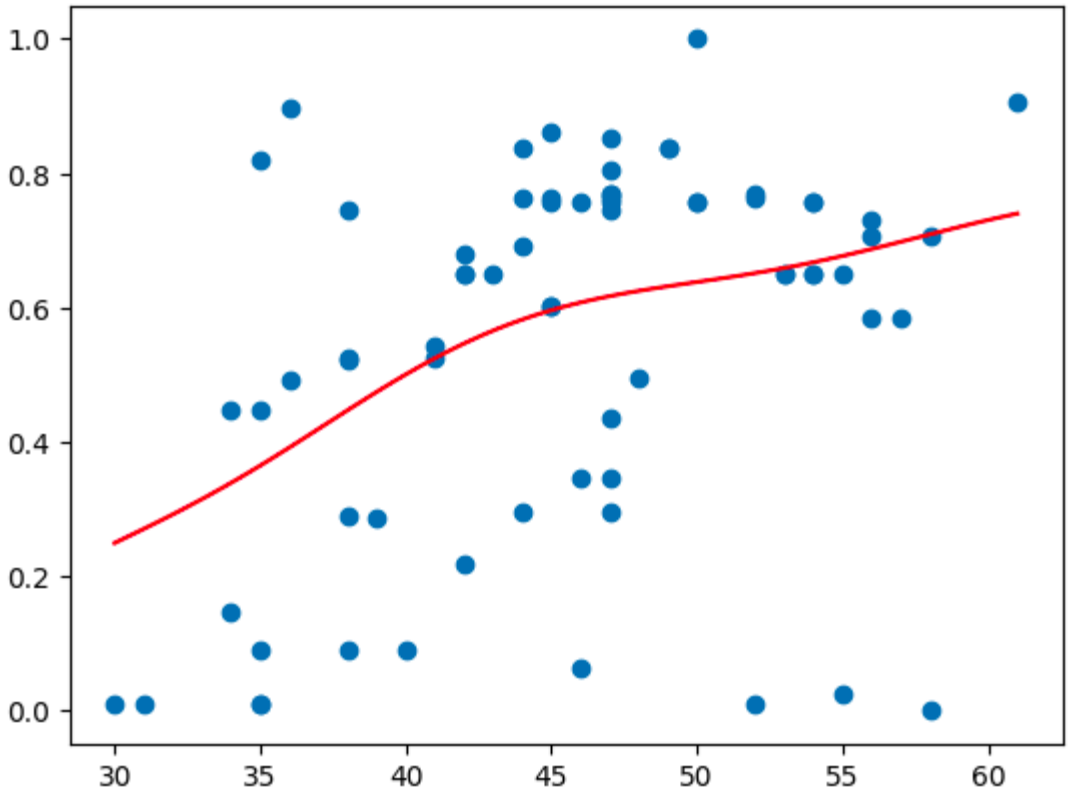
### 3.1. Plaćanje na osnovu godina iskustva:

Jedan od testova bio je nad podacima koji su nosili informacije o zaposlenima i njihovim godinama, plati i slično... Dobijeni graf je sledeći:



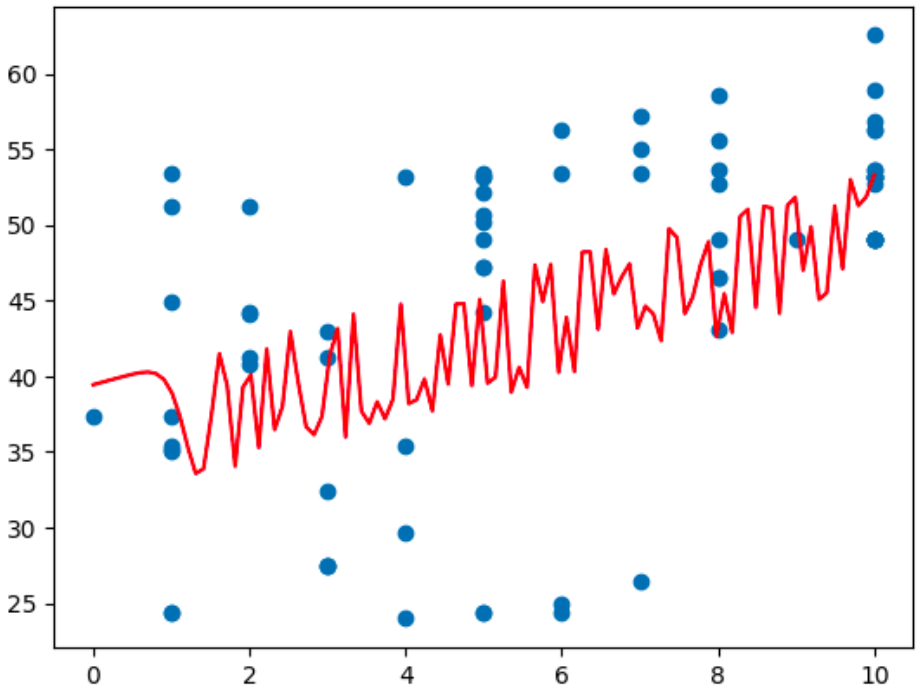

X osa predstavlja godine iskustva dok Y osa predstavlja platu, imajmo u vidu da su podaci normirani radi izbegavanja overflow-a. Mada vidimo na osnovu ovog grafa da plata i ne zavisi **toliko** od godina iskustva... Čini se kao da funkcija može da bude bolja, trenutna izgleda kao ne prati toliko podatke, i dalje nisam siguran zašto ne uspeva na ovako malim podacima ne može da nađe funkciju koja bolje opisuje podatke.

Mada kad pogledamo podatke gde smo za X osu odabrali da bude godine osobe koja se zapošljava:




Sami podaci su dosta razbacani i nisam siguran da uopše može da se napravi neka korelacija....

U jednoj od iteracija dobijena je sledeća funkcija:

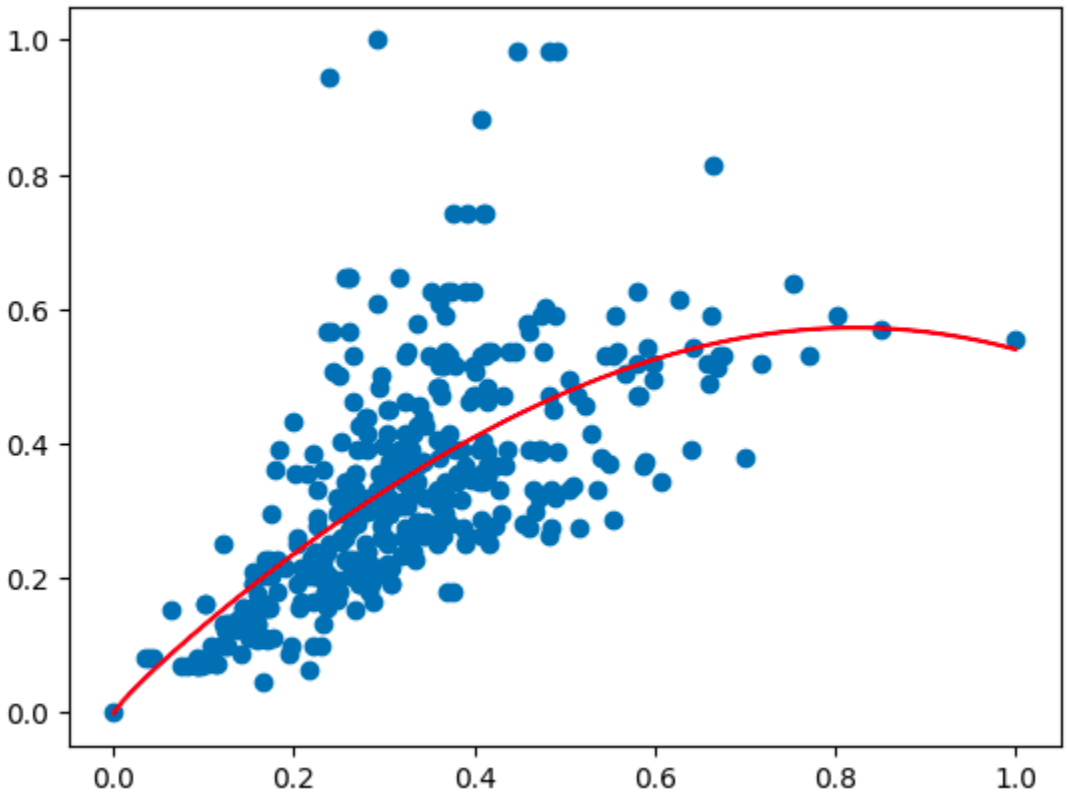


Ovaj graf predstavlja odnos godina iskustva (X osa) i plata (Y osa) ali u hiljadama, dakle ako na Y osi vidimo 45 to je ustvari 45k (valuta nije bitna ☺). Funkcija je blago rečeno poludela, i ovo je jedina iteracija u kojoj smo dobili ovakvu funkciju, mada vidi se da je pokušala da dostigne određene tačke na osnovu naglih skokova u funkciji kod onih tačaka koje se mogu smatrati i kao outlier. Recimo tačka (4, 54) koja je daleko od ostale tri tačke.

Logično po samom grafu vidi se da je greška velika jer su ovo greške od po 10k, a svesni smo da u bilo kojoj valuti ovo su velike razlike.

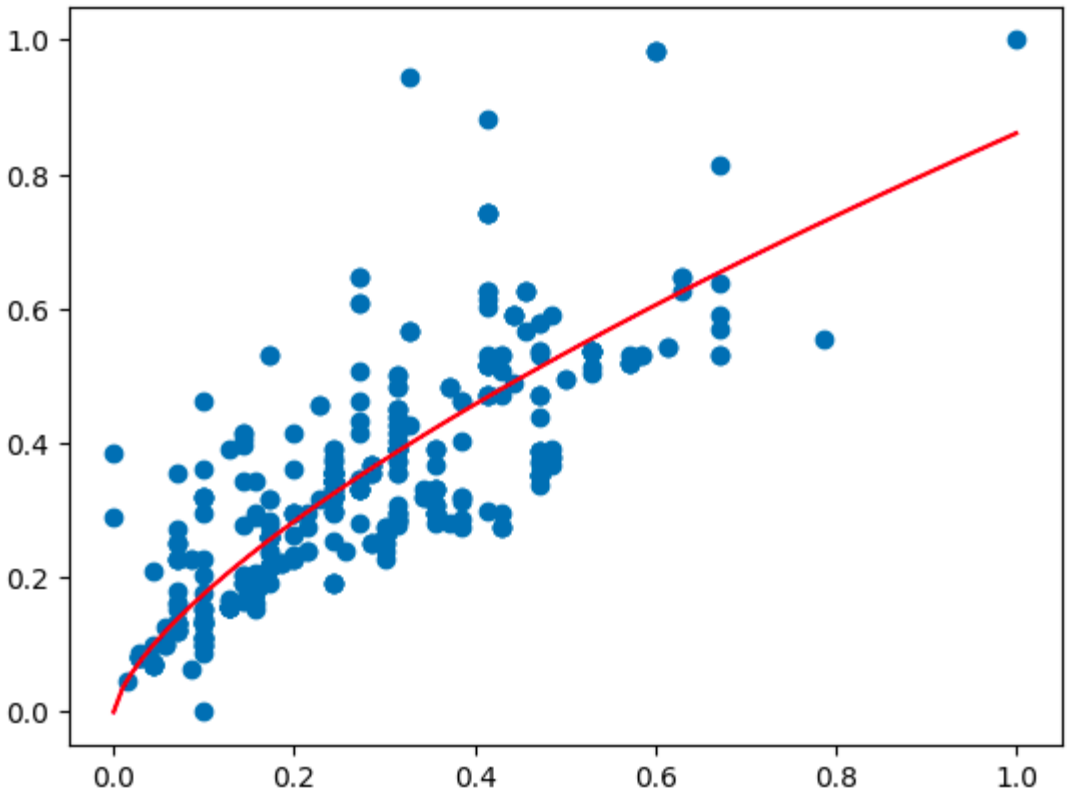
### 3.2. Konjske snage u odnosu na težinu kola:

U ovom primeru posmatrali smo malo veću bazu koja se sastoji o informacijama automobila, njihove snage, dužine, visine, modela, godišta... Nas je zanimalo od čega tačno zavisi snaga automobila, odabrali smo odnos težine kola i snage (konjske snage).




Ovog puta funkcija je čak i dobra, greška nije toliko velika pošto su sami podaci očigledno više zavisni nego u primeru godina iskustva i plate.

Naknadno smo uzeli da proverimo odnos zapremine motora i snage motora i ponovo vidimo da je zavisnost slične prirode kao za prethodni odnos:




Vidimo da je ova zavisnost čak i bolja, što ima smisla kako uglavnom kola sa većom zapreminom imaju mogućnost za postizanje veće energije.

## 4. Zaključak:

Sam algoritam ne daje toliko loše vrednosti, uglavnom su to greške reda 10^-2, mada nije to dovoljno niska greška. Podaci kod ovakvih rešenja i dalje odstupaju dosta od funkcija koje pronađemo.

Sam algoritam radi relativno sporo, dakle trebalo bi videti gde može da se dobije na vremenu... redundantno računanje.

Bilo bi dobro rešiti na pametniji način problem kompleksnih brojeva koji sam naveo u 2.2.1. poglavlju. I proširiti samu klasu Node, dodavanjem drugih elementarnih funkcija.

## 5. Literatura:

[1.] On the current paradigm in artificial intelligence – Nello Cristianini
[2.] Evolutionary algoritms and their applications to engineering problems – Adam Slowik, Halina Kwasnicka
[3.] Genetic programming as a means for programming computers by natural selection – John R. Koza
[4.] Symbolic Regression for Model Discovery in Python and Julia - Cambridge University RSE Seminars