SymbolicRegression (симболичка регресија)

1.Увод

Ово је пројекат из симболичке регресије у коме смо се потрудили да од нуле саградимо пројекат и да имамо тестерски програм на коме можемо да видимо перформансе.

Овај пројекат је рађен на рачунару следећих конфигурација:

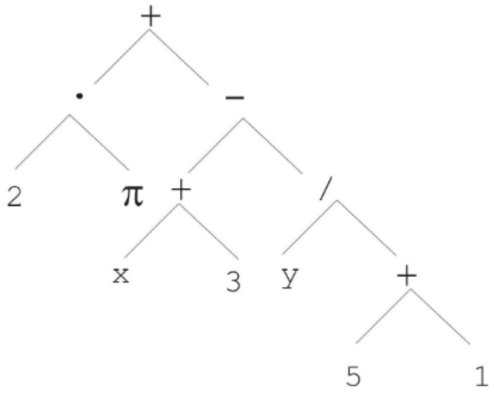
OS	System type	RAM	CPU	Graphic card	Speed:
Windows10	64-bit OS, x64-based processor	16 GB	AMD Ryzen 5 5600H	NVIDIA GrForce RTX 3050	3.30 GHz

2. Проблем симболичке регресије

Симболичка регресија спада у алгоритме генетског програмирања. Да бисмо описали какве особине има генетско програмирање, приказаћемо следећу табелу:

Појмови	Генетско програмирање
Репрезентација	Стабло
Укрштање	Размена подстабла
Мутација	Случајна промена у дрвету
Селекција родитеља	Фитнес сразмера
Селекција преживелих	Генерацијска замена

Као што видимо, ово је проблем *популационих метахеуристика*. Наш програм се не понаша детерминистички, а пошто је репрезентација решења стабеларан, наш пројекат има рекурзивне карактеристике. У случају пројекта, задатак је да правимо **математичке изразе** које могу да апроксимирају неке једноставније функције, нпр. полиноме вишег степена. Стабла требају да изгледају на следећи начин, нпр. на примеру израза $2\pi + ((x + 3) - \frac{y}{5+1})$:

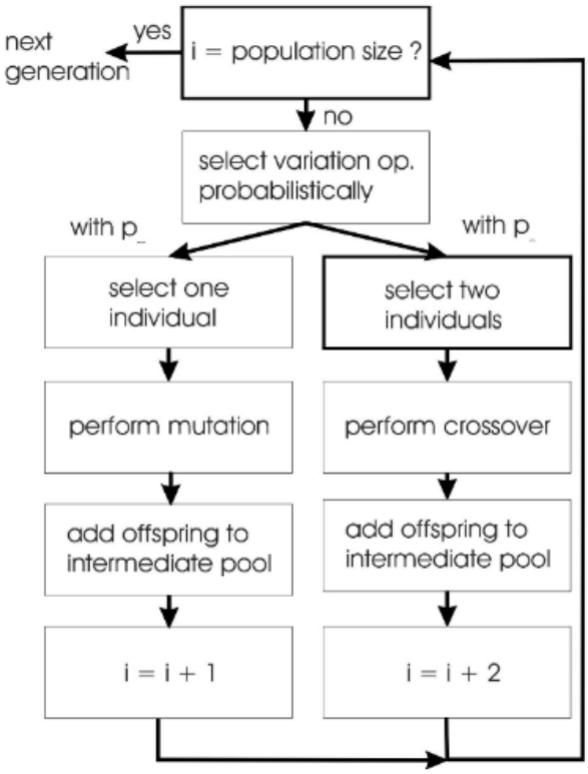


Као што видимо, ова стабла имају 2 различита типова чвора, **термове (Т)** које садрже бројеве и променљиве, и **функције (Ф)** које су облика у горњем примеру (*,-,+,/). За мој пројекат узео сам да постоје 4 типа чвора која су основ мојих симболичких стабала:

1. **Показивач на први чвор**, илити *dummy node* који показује на корен стабла.
2. **Термови**, који могу бити променљиве или конкретни бројеви.
3. **1-арне операције**, које су тригонометријског типа, sin и cos.
4. **2-арне операције**, које могу бити +, -, * и /.

Ова стабла правимо рекурзивно од показивачког чвора па све до листова, који су термови. Иницијално, популација је насумично направљена са максимално фиксираном бројем чворова 5.

Као што знамо, ГП је еволутивни алгоритам, то значи да се кроз генерације мења популација тако што преживљавају најбољи. Да бисмо то урадили, морамо да или **укрштамо два хромозома** или да **мутирамо један хромозон** са неком вероватноћом p који смо пре тога **селектовали** из претходне генерације. Општа шема је следећа:

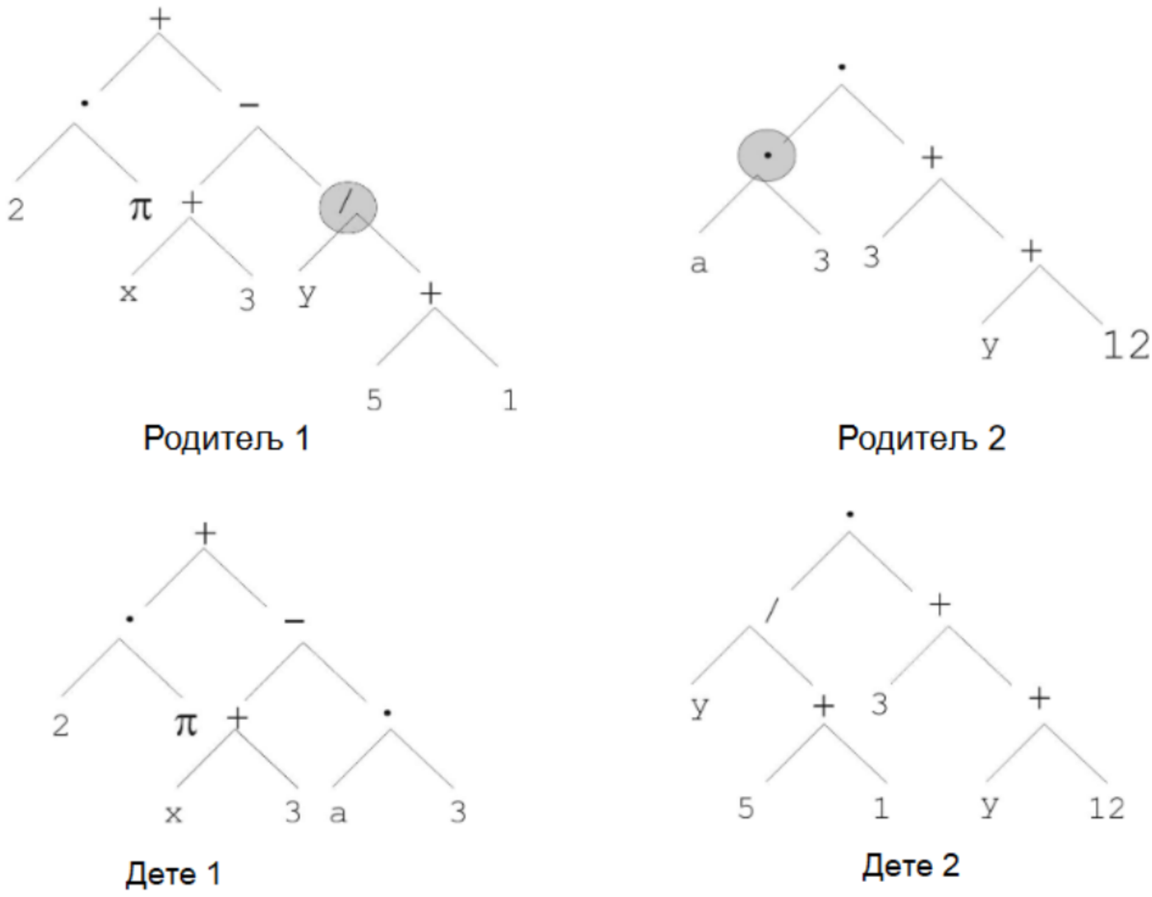


2.1. Мутација

Мутација код ГП-а се ради тако што се узима случајно одабрано подстабло или лист и он се мења са новим случајним генерисаним стаблом или подлистом. То ново подстабло није обично превелико са којим се мења подстабло. *Напомена*: Величина детета може да буде већа од величине родитеља.

2.2. Укрштање

Као што смо рекли, укрштање се ради тако што се чини замена два случајно одабрана подстабла између родитеља. Исто као код мутације, величине детата може бити већа. Пример укрштања је следећи:



2.3. Селекција

Селекција родитеља је фитнес сразмерна. У нашем проблему, ми прво сортирамо нашу популаицију по фитнесу. После тога, директно узимамо проценат првих x% најбољих и прослеђујемо је даљу у следећу генерацију да не би изгубили добре карактеристике. Остатак броје следеће популације селектујемо користећи **турнирску селекцију**. То је врста селектовања где узимамо n хромозона из претходне генерације и враћамо онај са најбољим фитнесом.

3. Експериментални резултати

Да бисмо поредили ваљаност мог пројекта, направио сам и споредни програм који користи `gplearn` и `sklearn` библиотеку да бисмо могли да видимо како нам ради пројекат. Значи, ако су нам функције које користимо у стаблу следеће: **+, -, *, /, cos и sin**, поред тога нам је вредност рачунања фитнеса *mean apsolute error*, тј. апсолутна сума разлике тачне и претпостављене вредности подељена са бројем улазних елемената. Нека нам је функција коју желимо да апроксимирамо има следећу форму:

$$x^4 - 4x^3 + x^2 - 5x + 1$$

Нека имамо 1000 елемената у популацији, 40 генерација, 10 елемената за турнир,300 елемената који елитистички извлачимо, и стопа мутације нека буде 10%.

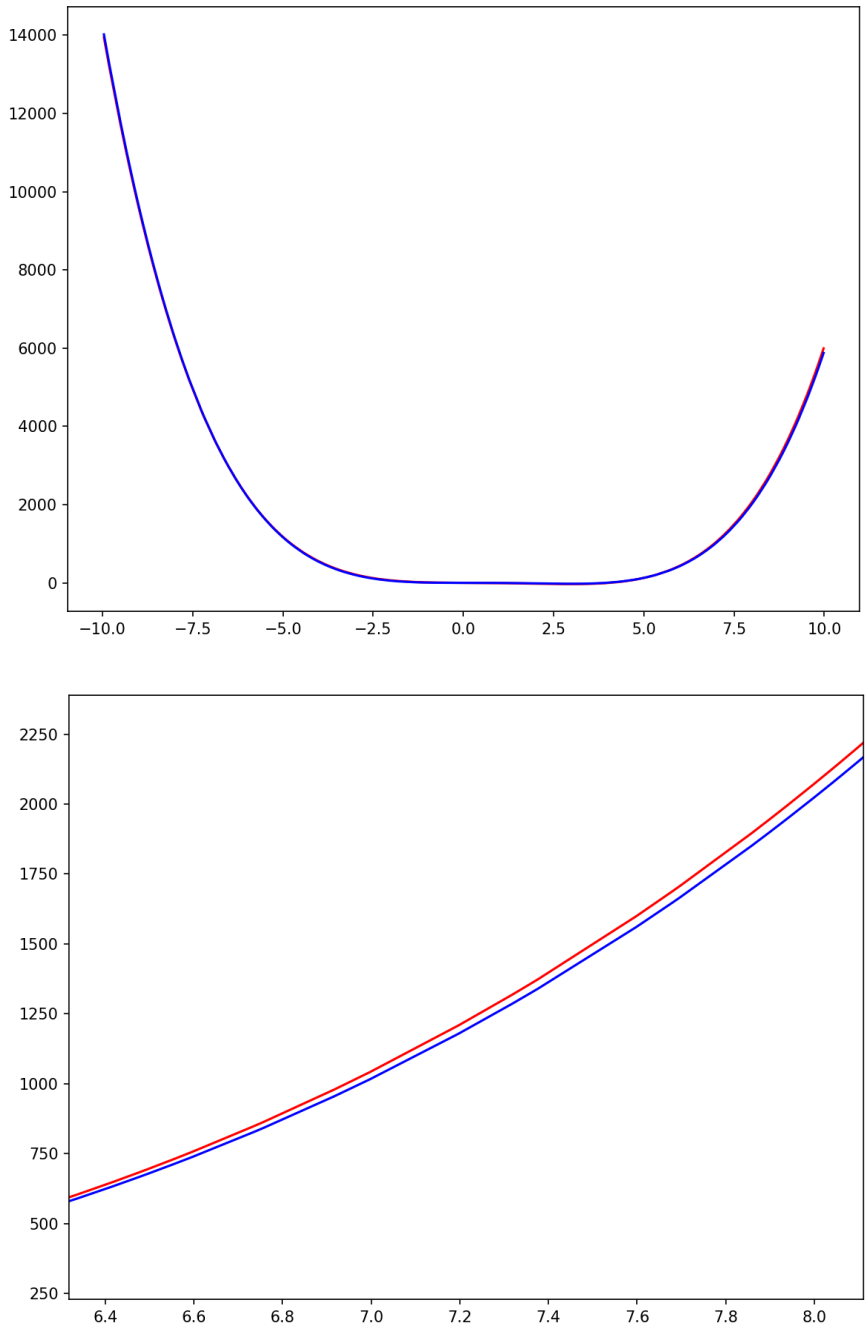
Мој пројекат пуца у раним фазамо, до 6 генерације је дошао дошао до следећих разултата:

```
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
SREDNJA VREDNOST FITNESSA JE: 8432.986278774795
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
NAJBOLJE: Generacija 6 izraz: ( ( cos(19.92) - cos(7.34) ) - x ) = 35.96951422723252
NAJGORE: Generacija 6 izraz: ( ( 4.64 / ( cos(( ( 10.19 + cos(4.38) ) - 19.46 )) * x ) ) * 5.43 ) = 812794.9103309292
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
```

Пошто `gplearn` програм сигурно не пуца, намерно сам ставио да узме да је број генерација 7 да бих га упоредио са мојим пројектом, док је све остало исто. Резултати су следећи:

```
    |   Population Average    |             Best Individual              |
---- ------------------------- ------------------------------------------ ----------
 Gen   Length          Fitness   Length          Fitness      OOB Fitness  Time Left
   0    15.98          2035.34       63          1385.16          1248.18      7.22s
   1    15.66          2248.53       11          1208.85          1277.68      5.90s
   2    27.80          4771.57       35          747.135          602.465      5.84s
   3    50.17          14900.4       36          503.276            517.2      5.60s
   4    51.42          17483.4       38           206.15          231.288      3.72s
   5    45.42            14514       24          34.3894          37.3937      1.77s
   6    40.13          16104.7       32          20.3396          22.1538      0.00s
```

Мој пројекат је имао бољу средњу вредност фитнеса, али зато `gplearn` очекивано, боље предвиђа најбољи елемент.



Слике које приказују како gplearn предвиђа елементе(плавом бојом) упоредо са правим елементима (црвене боје)

4. Документација пројекта

4.1. Main.py

Ово је класа одакле покрећемо пројекат, ту можемо да правимо неки насумичан улаз помоћу неке наше функције коју треба да апроксимира нап симболички регресор.

4.2.SymbolicRegression.py

Ово је фолдер у којима се налазе две класе са којима суштински радимо пројекат и једна помоћна класа. Овде декларишемо да све исписујемо у `output.txt` фајл.

4.2.1. Class Node

Класа помоћу којих можемо да правимо симболичка дрва. Нама су генерално симболичка дрва скуп чворова које могу бити следећег типа (`class Type(Enum)`):

1. `FIRST` - то нам је dummy чвор који служи да показује на први елемент у стаблу.
2. `TERM` - то нам је чвор који представља променљиву или број.
3. `OPERATOR` - то нам је чвор који представља бинарну операцију(+,*,-,%).
4. `TRIGONOMETRY` - то нам је чвор који представља косинусну/синусну операцију.

Класа се конструише на следећи начин: `__init__(self,type,level,char = None,value=None)`.

Методе које имамо су следеће у овој класи:

1. `getSubTreeFromPath(self,path,GP)` - рекурзивна метода која узима путању(`path`) и враћа чвор који је резултат путање од почетног чвора стабла.
2. `putSubTree(self,path,node)` - рекурзивна метода која ставља на путању(`path`) дати чвор(`node`).
3. `getRandomPath(self,GP)` - метода која генерише насумичну путању која се прави до максималне могућности дубљине стабла. Та путања нам се чува у `GP.localPath`.
4. `setChild1(self,node), setChild2(self,node)` - ставља на лево или десно дете нови чвор.
5. `stringNode(self)` - враћа нам ниску која представља наш математички израз.
6. `mutateInPath(self,path,GP,nodeNumForMutation)` - метода која у односу на нашу случајну путању(`path`) мења елемент у стаблу тако да прави на његовом месту насумично подстабло које је дубине `nodeNumForMutation`.
7. `getDepthOfNode(self)` - враћа дубину стабла/подстабла.
8. `getValue(self,xValue)` - враћа вредност стабла ако је променљива x једнако `xValue`.

4.2.2. Class GP(short for Genetic Programming)

Класа у којима се чува популација симболичких дрвета, где се налазе алгоритми конструисања насумичних дрвета, мутација, укрштања, као и сам *еволутивни алгоритам*.

Класа се контруише на следећи начин:
`__init__(self,goals,POPULATION_NUMBER,ITERATION_NUMBER,TOURNAMENT_SIZE,ELITISM_SIZE,MUTATION_RATE)`. Где је:

- `goals` - низ елемената која имају x вредност и f(x) вредност функције коју требамо да апроксимирамо.
- `POPULATION_NUMBER` - број популације.
- `ITERATION_NUMBER` - број итерације.

- `TOURNAMENT_SIZE` - број турнирске селекције.
- `ELITISM_SIZE` - број најбољих елемената која преживљавају следећу генерацију.
- `MUTATION_RATE` - вероватноћа са којом се ради мутација или укрштање.

Методе које се налазе:

1. `GP(self)` - метода која нам служи да узме нашу популацију `population` и да је кроз генерације мења тако да што више може да служи као апроксиматор функције.
2. `tournamentSelection(self)` - метода турнирске селекције, враћа индекс из тренутне генерације за укрштање/мутацију следећег хромозона.
3. `createRandomPopulation(self)` - у атрибуту `population` ствара насумичну популацију.
4. `betterMutation(self,index,GP)` - мутира `index`-ти елеменат популације.
5. `betterCrossover(self,index1,index2)` - метода која узима два хромозона, налази им насумичну путању `localPath` и онда им размењује елементе.
6. `calculateFitness(self,index)` - рачунамо фитнес нашег хромозона на следећи начин:

$$Fitness = \sum_{i=1}^{TrainingSet} |aproximateValue_i - realValue_i|$$

7. `generateRandomNode(self, depth, xValue,isParentTrig = False)` - генеришемо насумичан чвор у једном хромозону по томе која је његова дубина у којом се налази `depth`.
8. `generateSubTree(self,currentNode,depth,nodeNum,xValue,areWeGenerateFullTree = False)` - рекурзивна метода која генерише ново стабло/подстабло. `currentNode` нам означава чвор у коме се метода тренутно налази. `depth` је дубина до које смо генерисали дрво. `nodeNum` помоћни број за генерисање операција. `areWeGenerateFullTree` је индикатор којем наглашавамо јер правимо ново стабло(морамо да генеришемо `FIRST` тип чвора) или не.

Имамо и помоћну класу `DepthOfNode`, она сам служи да бисмо могли добро рекурзивно да имамо појам о томе на којој смо дубини стабла.

4.3.gplearn_test.py

Ово је пајтон програм који служи за упоређивање резултата нашег пројекта. Овде користимо `sklearn` и `gplearn` библиотеке које нам служе да правимо симболички регресор. Укратко:

```
import numpy as np #Biblioteka sa rad sa matematickim problemima
import pandas as pd #Biblioteka za pravljenje DataFrame-ova
import matplotlib.pyplot as plt #Biblioteka ya graficki prikaz podataka

from gplearn.genetic import SymbolicRegressor #Simbolicki regresor
from sklearn.model_selection import train_test_split #metoda za deljenje ulaznih
podataka na trening i test skup
from sympy import * #biblioteka za dobro formatiranje teksta.
```

Параметри која узима `SymbolicRegression:`

```
SymbolicRegressor(population_size=1000,function_set=function_set,
                            generations=7, stopping_criteria=0.01,
                            p_crossover=0.7, p_subtree_mutation=0.1,
                            p_hoist_mutation=0.05, p_point_mutation=0.1,
                            max_samples=0.9, verbose=1,
                            parsimony_coefficient=0.01, random_state=0,
                           feature_names=X_train.columns)
```

5. Закључак

Пројекат који је овде представљен није ни мало савршен. Пуца у раним генерацијама и достиже рану конвергенцију. Генерално, тешко је оджавати добар баланс стабла и због тога ми можда и пуца пројекат. Треба боље да се програмер носи са рекурзијом, рекурзивним позивима и одржавањем баланса стабла да би овај пројекат могао бити бољи, до тада, само тоеријски може пројекат да има смисла.

6. Референце

1. Сајт професора Александра Картеља: http://poincare.matf.bg.ac.rs/~kartelj/
2. Statistical genetic programming for symbolic regression, Maryam Amir Haeri, Mohammad Mehdi Ebadzadeha, Gianluigi Folino: http://poincare.matf.bg.ac.rs/~kartelj//gavrilo/g4.pdf