پروژه سوم درس هوش مصنوعی
استاد :دکتر آش عبدی

مجتبی جانباز ۹۹۵۲۱۱۶۳

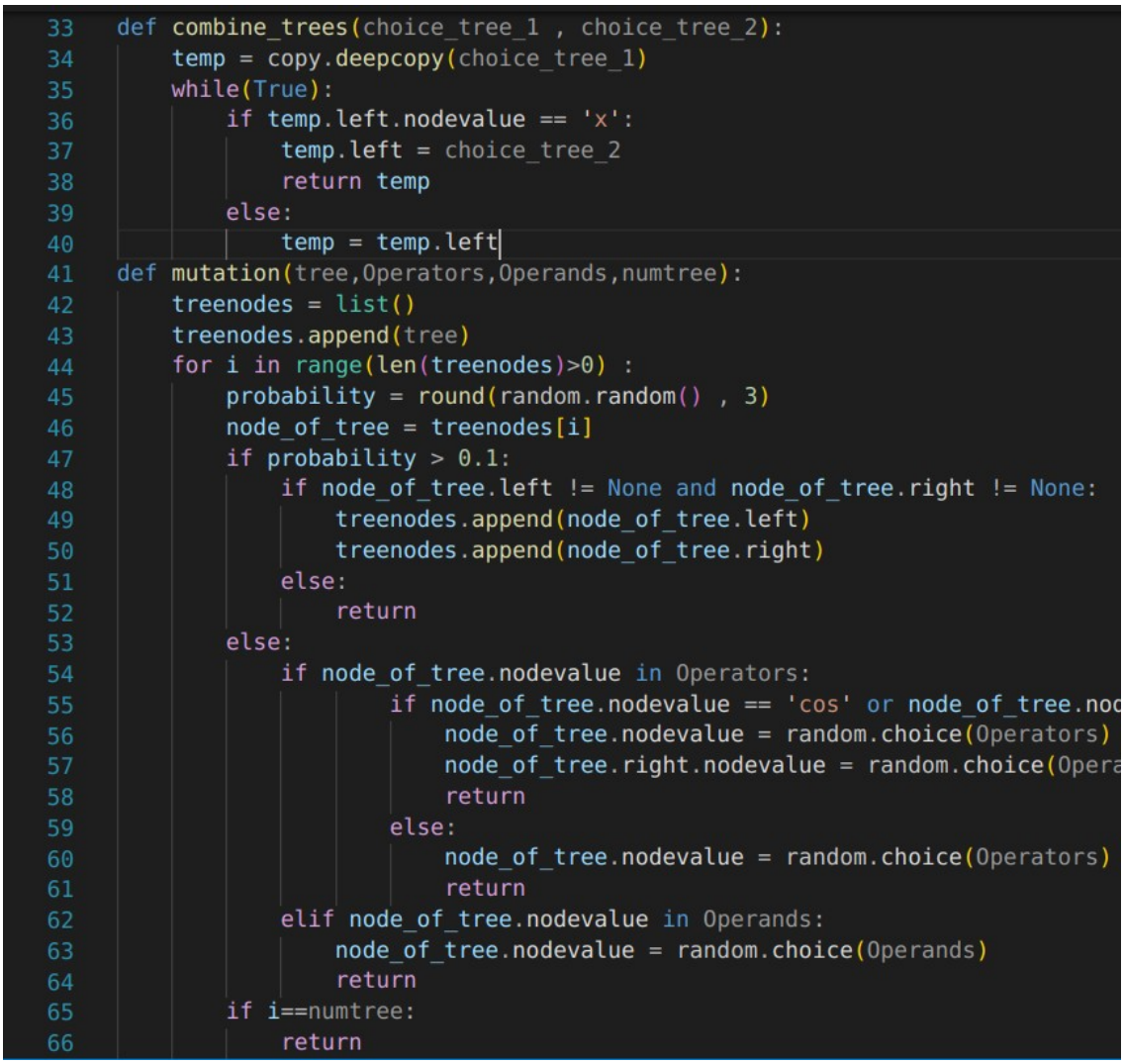
۸/۲/۱۴۰۲

```python
import random
import numpy as np
import copy
import math

class node:
    def __init__(self,nodevalue):
        self.nodevalue = nodevalue
        self.right = None
        self.left = None
        self.MSE = None

    def __lt__(self, other):
        return self.MSE < other.MSE
```

کلاس نود برای ساخت درختی هست که ضابطه احتمالی ما رو میسازد

```python
import random
import numpy as np
import copy
import math

class node:
    def __init__(self,nodevalue):
        self.nodevalue = nodevalue
        self.right = None
        self.left = None
        self.MSE = None

    def __lt__(self, other):
        return self.MSE < other.MSE

def print_tree(root):
    if root is None:
        return
    if root.left is None and root.right is None:
        if root.nodevalue == 'x':
            return 'x'
        else:
            return root.nodevalue

    LeftSum = print_tree(root.left)
    if root.nodevalue in ['+','-','*','/','**']:
        RightSum = print_tree(root.right)
        return '(' + str(LeftSum)+str(root.nodevalue)+str(RightSum)+')'
    else :
        return str(root.nodevalue)+'(' + str(LeftSum) + ')'

```

تابع چاپ درخت که در بالا هست برای چاپ کردن ضابطه به شکل ریاضیاتی هست



```
def combine_trees(choice_tree_1 , choice_tree_2):
    temp = copy.deepcopy(choice_tree_1)
    while(True):
        if temp.left.nodevalue == 'x':
            temp.left = choice_tree_2
            return temp
        else:
            temp = temp.left
def mutation(tree,Operators,Operands,numtree):
    treenodes = list()
    treenodes.append(tree)
    for i in range(len(treenodes)>0) :
        probability = round(random.random() , 3)
        node_of_tree = treenodes[i]
        if probability > 0.1:
            if node_of_tree.left != None and node_of_tree.right != None:
                treenodes.append(node_of_tree.left)
                treenodes.append(node_of_tree.right)
            else:
                return
        else:
            if node_of_tree.nodevalue in Operators:
                    if node_of_tree.nodevalue == 'cos' or node_of_tree.noo
                        node_of_tree.nodevalue = random.choice(Operators)
                        node_of_tree.right.nodevalue = random.choice(Opera
                        return
                    else:
                        node_of_tree.nodevalue = random.choice(Operators)
                        return
            elif node_of_tree.nodevalue in Operands:
                node_of_tree.nodevalue = random.choice(Operands)
                return
        if i==numtree:
            return
```

در این عکس تابع اول برای ترکیب درخت ها استفاده میشه که در الگوریتم ژنتیک صدا زده میشود.

در این تابع دو دخت که در الگوریتم ژنتیک به طور رندوم از لیست درخت های جمعیت اولیه انتخاب میکنیم را با هم ترکیب میکنیم.
و در تابع بعدی که برای جهش از آن استفاده میکنیم به این صورت عمل میکند که یک عدد رندوم تولید میشود ار بین ۰ تا ۱ و اگر این عدد بزرگتر از ۰.۱ باشد جهش صورت نمیگیرد و درخت به همان صورت باقی میماند ولی اگر این مقدار رندوم کمتر از عدد ۰.۱ باشد جهش رخ میدهد که به این معنی هست که به طور رندوم یکی ازنود های درخت با یک نود دیگر جابجا میشود .

```
69  def calculate_tree(tree , x_entr):
70      pridict_y = list()
71      tree_function = str(print_tree(tree))
72      for point_x in x_entr:
73          res=0
74          try:
75              tree_function = tree_function.replace('x' , str(point_x))
76              res = eval(tree_function)
77          except:
78              pass
79          pridict_y.append(res)
80      return pridict_y
81
82  def find_best_trees(tree , x , y):
83      pridict_y = calculate_tree(tree , x)
84      summation = 0
85      for i in range (0,len(y)):
86          difference = y[i] - pridict_y[i]
87          squared_difference = difference**2
88          summation = summation + squared_difference
89      MSE = summation/len(y)
90      return MSE/10
91
```

. در بالا دو تابع داریم که تابع اول مقدارضابطه درخت را بر اساس ورودی هایی که ابتدای برنامه میدهیم محاسبه میکند

و تابع دوم مقادیر خروجی درخت را که در تابع قبل لیست شده بود را با مقادیر خروخی خود تابع اصلی بر اساس ورودی های اول برنامه مقایسه میکند و یک مقداری را به عنوان کل ارزش درخت بر اساس میزان درست بودنش نسبت به خروجی اصلی برمیگرداند.
MSE مقدار

```
92  # x o y dar khod tabe asli
93  x = list()
94  # baze az 1 ta 10 ba fasele 0.2
95  temp = 0.8
96  for i in range(46):
97      temp = temp + 0.2
98      temp = round(temp , 1)
99      x.append(temp)
100 y = list()
101 for i in range(46):
102     # y.append(x[i]+2)
103     # y.append(x[i]*math.sin(x[i]+1))
104     # y.append(2*x[i]**3 + 2)
105     y.append(6*math.sin(x[i]) + 3)
106     # y.append(-1*x[i]+3)
107
```

در اینجا ورودی های تابع ما مشخص میشود و در پایین خروجی بر اساس ضابطه بدست می آید.

```python
#tolid jamiat avalie random
number_population_trees = 50
operators = ['*' , '/' , '+' , '-' , '**' , 'math.sin' , 'math.cos']
operands = [1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , 'x']
first_population_trees = list()
for i in range(number_population_trees):
    operatorTMP = random.choice(operators)
    operandTMP = random.choice(operands)
    # print(operandTMP)
    parent = node(operatorTMP)
    parent.left = node('x')
    if operatorTMP == 'sin' or operatorTMP == 'cos':
        parent.right = node(None)
    else:
        parent.right = node(operandTMP)
    #agar khod tabe bod hazf mishe va yeki dige sakhte mishe
    if operatorTMP == '+' and operandTMP == '2':
        del parent
        i -= 1
    first_population_trees.append(parent)

```

در عکس بالا جمعیت اولیه ما تولید میشود با عملگر ها و عملوند های مشخص شده به این صورت که به طور رندوم از عملگر ها و عملوند ها انتخاب میکنیم و بر اساس نوعش عملگر ها که تک عضوی یا دو عضوی باشند یک درخت با ارتفاع ۱ میسازیم و در اخر با توجه به ضابطه انتخابی اگر درختی دقیقا با همین ضابطه ها ساخته بشه حذف میشه و درخت رندوم دیگری به جمعیت اولیه ما اضافه میشه.

```
#GP
population_befor = list() #nasli ke alan hastim
population_befor = copy.deepcopy(first_population_trees)

population_after = list() #entekhabi haye nasle badi
population_after = copy.deepcopy(first_population_trees)

generation_number = 50
number_trees_combined = 10
numtree_mutation = 1
best_mse_in_generation = list()
iteration=0
for iteration in range(generation_number):
    # population_after = copy.deepcopy(population_befor)
    numtree_mutation = numtree_mutation+2
    delindx = list()
    for j in range(int(number_trees_combined/2)):
        choisen_tree_1 = random.choice(population_befor)
        choisen_tree_2 = random.choice(population_befor)
        population_after.append(combine_trees(choisen_tree_1 , choisen_tree_2))
        population_after.append(combine_trees(choisen_tree_2 , choisen_tree_1))
        if j==int(number_trees_combined/2)-1:# delete derakht moshabeh
            for a in range(len(population_after)):
                for s in range(len(population_after)):
                    if population_after[a].MSE == population_after[s].MSE and a != s:
                        delindx.append(a)
            delindx = [*set(delindx)]
            if len(delindx) < int(number_trees_combined/2) + number_population_trees:
                j = j- int(int(number_trees_combined/2)/3)

```

.در اینجا الگوریتم ژنتیک را برای جمعیت اولیه اججرا میکنیم
الگوریتم ما به این صورت است که دو لیست جمعیت اولیه و بازماندگان جمعیت اولیه وجود دارد و ابتداع دو درخت به طور رندوم از جمعیت اولیه انتخاب میکنیم و با استفاده از تابع ترکیب درخت از این دو دو درخت جدید میسازیم و این دو درخت جدید را در بازمانده ها میریزیم که از قبل دارای درخت های قبلی(درخت های درون جمعیت اولیه) هست و بعد وقتی این چرخه تولید درخت ترکیبی در حال پایان است درخت های با ارزش برابر را یکیشان را .حذف میکنیم که درخت تکراری نداشته باشیم

```
159
160     for indximprtnt in range(len(delindx)):
161         population_befor.append(population_after[delindx[indximprtnt]])
162     for k in range(len(delindx)):
163         population_after.append(population_befor[k])
164
165
166     # population_befor.clear()
167     for tree in population_after:
168         mutation(tree,operators,operands,numtree_mutation)
169
170     for tree in population_after:
171         tree.MSE = round(find_best_trees(tree , x , y) , 3)
172
173     population_befor.clear()
174     population_after.sort()
175     population_befor=population_after[:number_population_trees]
176     population_after.clear()
177     population_after=population_befor[:number_population_trees]
178
179     best_mse_in_generation.append(population_befor[0].MSE)
180     if population_befor[0].MSE < 1:
181         break
182
```

و بعد به ازای تمامی درخت های موجود در بازماندگان تابع جهش رو صدا میزنیم که با احتمال خیلی کمی روی آنها جهش انجام دهد و بعد تابع انتخاب بهتربین درخت رو صدا میزنیم که میاد ارزش تمامی درخت ها را مشخص میکند و به این صورت هست که هرچه نزدیک تر به ۰ باشد بهتر است MSE این مقدار
و درخت ها را با توجه به ارزششان سورت میکنیم در لیست بازماندگان و بعد چنتای اول(بر اساس تعداد جمعیت اولیه) را درون جمعیت اولیه جدید میریزیم و این جمعیت اولیه جدید میشود جمعبت اولبه ما در نسل بعدی.
تعداد جمعیت اولیه که خودمان میسازیم را مشخص میکنیم و بعد همیشه در نسل های متفاوت باید همین تعداد جمعیت اولیه داشته

باشیم و در پایان هر چرخه باید بهترین درخت ها را بر اساس ارزششان انتخاب کنیم که جمعیت اولیه نسل بعد را تشکیل دهند .
و در نهایت وقتی این چرخه بعه پایان میرسد که با به درختی با ارزش بسیار بالا برسیم
MSE<1
و یا نسل های ما تمام شود که این هم خودمان مشخص میکنیم چند نسل طول بکشد .
و در نهایت خروجی به این صورت میشود که بازمانده های اخرین نسل به ترتیب ارزششان سورت میشوند و بهترین ضابطه کشف شده اخرین خروجی ما هست که ارزشش هم مشخص است و همچنین اینکه درخت در کدام نسل به بهترین خروجی خود رسید هم چاپ میشود

```
183         if best_MSE > population_befor[0].MSE:
184             best_tree = population_befor[0]
185             best_MSE = population_befor[0].MSE
186             best_gen = generation
187         if population_befor[0].MSE < 1:
188             break
189 
190     # print("\n\n\nbest function")
191     # print(print_tree(population_befor[0]))
192     # print("MSE = "+str(population_befor[0].MSE))
193     # print("\n")
194     # print("Best tree")
195     # print(print_tree(best_tree))
196     # print("Best mse = "+str(best_MSE))
197     # print("Best generation = "+str(best_gen))
198 
199     population_befor.reverse()
200     for i in range(len(population_befor)):
201         print(print_tree(population_befor[i]))
202         print("MSE = "+str(population_befor[i].MSE))
203     print("\n")
204     print("Best tree")
205     print(print_tree(best_tree))
206     print("Best mse = "+str(best_MSE))
207     print("Best generation = "+str(best_gen))
```

به عنوان یکی از عملوند ها این بود که ممکن x دلیل استفاده از هم در ضابطه باشد x است ۲
ممکن بود در درخت هایی که رندوم تولید شد تقسیم بر ۰ هم بیاید که این مقدار را ۰ قرار دادم و همچنین برای اعدادی که خیلی بزرک میشدند ۱۰۰۰۰۰۰۰ قرار دادم که به مشکل برای محاسبه و مقدار دهی ها در برنامه مواجه نشوم
به علت آنکه در الگوریتم ژنتیک ما توابع رندومی تولید میکنیم و ممکن هست خیلی از جواب دور شویم پس تمامی جمعیت قبلی را با جمعیتی که تولید کردیم با هم مقایسه میکنیم که به نسل های بعدی درخت های مطمین تری منتقل شود
از آنجا که تمامی عملگر ها دو قسمتی نیستند برای ساخت درخت دچار چالش هایی شدم که با استفاده از قانون گذاری برطرفش کردم های اولیه در سمت چپ و تمامی x به این صورت که تمامی عملگر های تک عنصری عنصرشان در سمت راست باشد و در ادامه هم برای این نوع عملگر ها فرزند چپشان را خالی کردم که برای مراحل بعد اشتباهی پیش نیاید

خروجی با تعداد نسل کم

```
101   for i in range(100):
102       y.append(x[i]+2)
103       # y.append(x[i]*math.sin(x[i]+1))
104       # y.append(6*math.sin(x[i]) + 3)
105       # y.append(-1*x[i]+3)
106
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Best tree
((math.sin((math.sin((x-9))**8))+8)+1)
Best mse = 3.333
Best generation = 9
```

```
101   for i in range(100):
102       # y.append(x[i]+2)
103       y.append(x[i]*math.sin(x[i]+1))
104       # y.append(6*math.sin(x[i]) + 3)
105       # y.append(-1*x[i]+3)
106
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Best tree
math.sin(math.sin((x*4)))
Best mse = 1.74781617
Best generation = 3
```

```
101   for i in range(100):
102       # y.append(x[i]+2)
103       # y.append(x[i]*math.sin(x[i]+1))
104       y.append(2*math.sin(x[i]) + i*3)
105       # y.append(-1*x[i]+3)
106
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Best tree
((((math.sin((x**5))*x)/9)-6)-6)
Best mse = 760.83165491
Best generation = 7
```

```
101   for i in range(100):
102       # y.append(x[i]+2)
103       # y.append(x[i]*math.sin(x[i]+1))
104       # y.append(2*math.sin(x[i]) + i*3)
105       y.append(-1*x[i]+3)
106
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

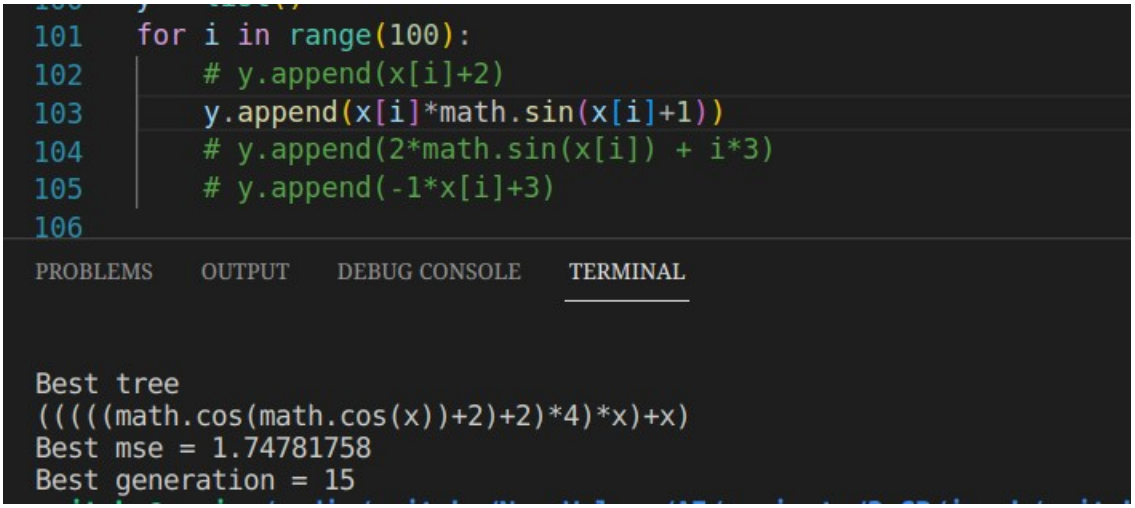
```
Best tree
math.sin(math.cos(math.sin(x)))
Best mse = 3.33350922
Best generation = 3
```

خروجی با تعداد نسل بالا تر

```
101   for i in range(100):
102       y.append(x[i]+2)
103       # y.append(x[i]*math.sin(x[i]+1))
104       # y.append(2*math.sin(x[i]) + i*3)
105       # y.append(-1*x[i]+3)
106
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
Best tree
(((x/2)/x)/1)
Best mse = 3.33307894
Best generation = 4
```



میشود به راحتی فهمید که با داشتن زمان بیشتر که بشود تعداد نسل های بیشتری تولید کرد برای توابع سخت تر و پیچیده تر نتیجه بهتری گرفت و اما برای توابع آسون تر در ایتریشن های کم هم میشود نتیجه خوبی گرفت.

در صورت بالا بودن نسل دقت بیشتر است و سرعت کمتر و برعکس
و اما به طور کلی چون درخت های ما تا حدود زیادی مخصوصا در اوایل به طور رندوم تر ساخته میشوند احتمال جواب خوب گرفتن در نسل های کمتر هم وجود دارد
با تست های انجام شده دیدم که حتی در نسل های خبلب پایین هم برای توابع سخت جواب بسیار مناسبی وجود دارد و خب به علت تصادفی بودن درخت ها میشود در اجرا های متفاوت جواب های به شدت متفاوتی گرفت .

در نهایت این الگوریتم جزو روش های به شدت قدرتمند جست و جوی محلی است و برای تخمین زدن بسیار کاربردی است.
با افزایش تعداد جمعیت اولیه و توابع ترکیبی بسیار در حدس بهترین تابع کمک میکنه و همچنین میتونه از پراکندگی زیاد ما توی نسل های اول جلوگیری کنه .
اضافه کردن جمعیت اولیه به جمعیت اولیه نسل بعد به شدت مفید هست و باعس تخمین زودتر و در هر مرحله نزدیک تر میشود.