بسمه تعالی

گزارش پروژه ۱ هوش مصنوعی

نحوه دقیق نگاشت مساله به درخت و محدودیت های احتمالی و تولید جمعیت اولیه:

به منظور نگاشت مساله به درخت ابتدا عناصر موجود در درخت را به دو نوع عملگر و عملوند تقسیم بندی می کنیم. یک ارتفاع بیشینه و یک مقدار معین که از هر ارتفاع چقدر داشته باشیم نیز در نظر می گیریم و سپس به ازای هر ارتفاع از درخت یعنی ۱، ۲، ۳، ...، h که h مقدار ماکسیمم است درخت های تصادفی تولید می کنیم. این امر به گونه ای است که عملگر های مورد نظر ابتدا در یک لیست قرار می گیرند و اندیسشان در درون گره های درخت می نشیند. عملوند ها نیز برای x عدد ویژه ۳- و برای دو عدد e و π نیز به ترتیب اعداد ۱- و ۲- را در نظر می گیریم برای باقی اعداد نیز همان مقدارشان در نظر گرفته می شود. حال شروع به ساختن درخت می کنیم و از کتابخانه **binarytree** نیز کمک می گیریم و آرایه درخت را می سازیم. در هنگام ساخت آرایه درخت توجه داریم که از اندیس $i > \left\lfloor \frac{n-1}{2} \right\rfloor$ به بعد همه ی عناصر برگ می شوند و باید عملوند باشند. اما برای عناصر قبل از آن مقدار، ما ابتدا عملگر ها و عملوند ها (ترجیحا x) را به گونه ی تصادفی انتخاب می کنیم و در جای مورد نظر قرار می دهیم توجه داریم که سعی می کنیم مقادیر عملگر با احتمال بیشتری انتخاب شود. در حین انتخاب نیز خانه های شامل $2 \times i + 1$ که عملگر سینوس و کسینوس در خانه i قرار دارد را **None** می گزاریم تا خاصیت تک عملوندیش حفظ شود برای عملوند ها نیز خانه های $2 \times i + 1$ و $2 \times i + 2$ را چنین می کنیم. بدین ترتیب درخت ها ساخته می شوند و به عنوان جمعیت اولیه در یک لیست قرار می گیرند.

```
before_1 = np.array([-4, -5, -6, -7, -8, -9, -10, -11, -12, -13])
after_1 = (-1) * before_1 - 1
before_2 = np.array([2, 3, 5, 6, 34, 234, 13, 3, 2123, 42, 1, 3, 21, 32, 13, 1])
after_2 = np.power(before_2, 2)
initial_points = (np.hstack((before_1, before_2)), np.hstack((after_1, after_2)))
population_1 = set()
list_of_keys = np.array(['+', '-', '*', '/', '^', "Sin", "Cos"])
dict_of_const = {-1: 'exp', -2: 'pi'}
list_of_const = np.array(list(dict_of_const.keys()))
max_height = 10
number_per_each = 400

for i in range(1, max_height):
    list_length = np.power(2, i + 1) - 1
    left = int(np.ceil((list_length - 1) / 2))
    right = list_length - left
    variable = [-3] * (list_length - 1)
    for k in range(number_per_each):
        tree_list = produce_random_tree(list_length, left, variable)
        population_1.add(bt.build(list(tree_list)))
```

```
def produce_random_tree(list_length, left, variable):
    random_ops = np.random.choice(np.arange(7), left)
    random_operands = np.round(np.random.normal(np.mean(initial_points[1]), np.std(initial_points[1]), list_length - 1),
                               4)
    random_const = np.random.choice(list_of_const, list_length - 1)
    chooser_rand = [0, 0, 0, 0]
    chooser = [random_ops, variable, random_operands, random_const]
    tree_list = np.array([None] * list_length)
    array = chooser[0]
    selected = float(array[chooser_rand[0]])
    tree_list[0] = selected
    chooser_rand[0] += 1
    tree_list[1] = float("inf")
    if selected == 5 or selected == 6:
        if 2 < list_length:
            tree_list[2] = None
    else:
        tree_list[2] = float("inf")
    for j in range(1, left):
        if tree_list[j] == float("inf"):
            rand = np.random.choice([0, 1, 2, 3], 1, p=[0.55, 0.43, 0.01, 0.01])[0]
            array = chooser[rand]
            selected = array[chooser_rand[rand]]
            chooser_rand[rand] += 1
            tree_list[j] = float(selected)
            if rand == 0:
                tree_list[2 * j + 1] = float("inf")
                if selected == 6 or selected == 5:
                    if 2 * j + 2 < list_length:
                        tree_list[2 * j + 2] = None
```

```
    selected = float(array[chooser_rand[0]])
    tree_list[0] = selected
    chooser_rand[0] += 1
    tree_list[1] = float("inf")
    if selected == 5 or selected == 6:
        if 2 < list_length:
            tree_list[2] = None
    else:
        tree_list[2] = float("inf")
    for j in range(1, left):
        if tree_list[j] == float("inf"):
            rand = np.random.choice([0, 1, 2, 3], 1, p=[0.55, 0.43, 0.01, 0.01])[0]
            array = chooser[rand]
            selected = array[chooser_rand[rand]]
            chooser_rand[rand] += 1
            tree_list[j] = float(selected)
            if rand == 0:
                tree_list[2 * j + 1] = float("inf")
                if selected == 6 or selected == 5:
                    if 2 * j + 2 < list_length:
                        tree_list[2 * j + 2] = None
                else:
                    tree_list[2 * j + 2] = float("inf")
    for j in range(left, list_length):
        if tree_list[j] == float("inf"):
            rand = np.random.choice([1, 2, 3], 1, p=[0.5, 0.4, 0.1])[0]
            array = chooser[rand]
            selected = array[chooser_rand[rand]]
            chooser_rand[rand] += 1
            tree_list[j] = float(selected)
    return tree_list
```

تابع شایستگی:

در مرحله بعدی با استفاده از تابع تعریف شده **weighted_by()** به ازای هر درخت شایستگی یا فیتنسش را در یک دیکشنری ذخیره می کنیم. یعنی ابتدا برای یک درخت مورد نظر تمامی نقاط را خروجی می گیریم و سپس طبق رابطه ی زیر که همان وارون جذر میانگین خطای مربعات هست. شایستگی به ازای هر درخت ذخیره می کنیم.

$$fitness = \frac{1}{\sqrt{\frac{\sum_{k=1}^{N}(\widehat{y_k} - y_k)^2}{N}}}$$

```python
def err_computation(tree, points):
    list_of_values = []
    for x, y in zip(list(points[0]), list(points[1])):
        y_bar = np.around(solve(tree, x), 2)
        if not np.isfinite(y_bar):
            continue
        square_inverse = np.around(np.power(np.subtract(y_bar, y), 2), 5)
        if not np.isfinite(square_inverse):
            continue
        list_of_values.append(square_inverse)
    return np.around(1 / np.sqrt(np.mean(np.array(list_of_values))), 5)


def weighted_by(population, points):
    fitness_respect_to_tree = dict()
    for tree in population:
        fitness = err_computation(tree, points)
        if not np.isfinite(fitness):
            continue
        fitness_respect_to_tree[tree] = fitness
    return fitness_respect_to_tree
```

```python
def solve(root, x):
    if root is not None:
        if root.val == float(-3):
            return x
        elif root.val == float(-1):
            return np.around(np.e, 5)
        elif root.val == float(-2):
            return np.round(np.pi, 5)
        elif root.val not in list(map(float, range(7))):
            return np.around(root.val, 5)
        else:
            A = solve(root.left, x)
            B = solve(root.right, x)
            if root.val == float(0):
                return np.around(np.add(A, B), 5)
            elif root.val == float(1):
                return np.around(np.subtract(A, B), 5)
            elif root.val == float(2):
                return np.around(np.multiply(A, B), 5)
            elif root.val == float(3):
                if A == 0 and B == 0:
                    return float(1)
                if A != 0 and B == 0:
                    return float("inf")
                return np.around(np.divide(A, B), 5)
            elif root.val == float(4):
                if B < 0:
                    B = np.multiply(-1, B)
                    A = 1 / A
                return np.around(np.power(A, B), 5)
            elif root.val == float(5):
```

```
            return np.round(np.pi, 5)
        elif root.val not in list(map(float, range(7))):
            return np.around(root.val, 5)
        else:
            A = solve(root.left, x)
            B = solve(root.right, x)
            if root.val == float(0):
                return np.around(np.add(A, B), 5)
            elif root.val == float(1):
                return np.around(np.subtract(A, B), 5)
            elif root.val == float(2):
                return np.around(np.multiply(A, B), 5)
            elif root.val == float(3):
                if A == 0 and B == 0:
                    return float(1)
                if A != 0 and B == 0:
                    return float("inf")
                return np.around(np.divide(A, B), 5)
            elif root.val == float(4):
                if B < 0:
                    B = np.multiply(-1, B)
                    A = 1 / A
                return np.around(np.power(A, B), 5)
            elif root.val == float(5):
                return np.around(np.sin(np.multiply(A, np.pi / 180)), 5)
            else:
                return np.around(np.cos(np.multiply(A, np.pi / 180)), 5)
    else:
        return float(0)
```

نحوه انتخاب والدین :

به منظور انتخاب والدین از تابع **weighted_random_choice()** کمک می گیریم که در آن بر اساس شایستگی موجود بین درخت ها ، درختی را با احتمال زیر انتخاب می کند.

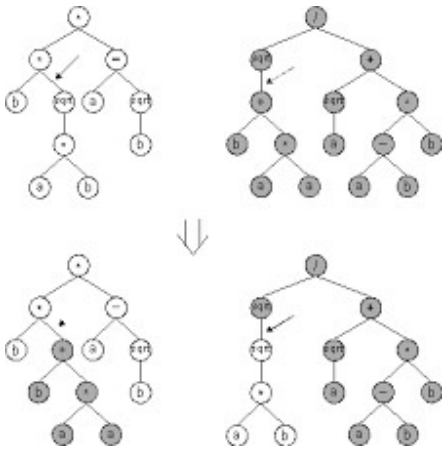
$$Tree_i = \frac{fitneess_i}{\sum_{all\_trees} fitness}$$

دو نمونه از مجموعه درخت ها با احتمال بالا گرفته می شود. و به عنوان والدین فرزندان انتخاب می شوند.

```python
def weighted_random_choice(weights_local, j):
    all_fitness = np.array(list(weights_local.values()))
    finite_trees = list(weights_local.keys())
    all_fitness_rounded = np.around(all_fitness, 2)
    fitness_probability = np.divide(all_fitness_rounded, np.sum(all_fitness_rounded))
    # while not np.isfinite(fitness_probability).all():
    #     print("I'm in loop")
    #     tree_indexes_that_is_finite = list(np.where(np.any(np.isfinite(fitness_probability))))
    #     finite_trees = []
    #     for i in range(len(weights_local.keys())):
    #         if i in tree_indexes_that_is_finite:
    #             finite_trees.append(tree_list[i])
    #     all_fitness = []
    #     for element in list(weights_local.keys()):
    #         if element in finite_trees:
    #             all_fitness.append(weights_local[element])
    #     all_fitness_rounded = np.round(all_fitness, 2)
    #     fitness_probability = np.divide(all_fitness_rounded, np.sum(all_fitness_rounded))
    indexes = np.random.choice(np.arange(len(all_fitness)),  2, replace=True, p=fitness_probability)
    return (finite_trees[indexes[0]], finite_trees[indexes[1]])
```

نحوه تولید نسل بعد و ترکیب متقاطع :

با استفاده از تابع **reproduce()** فرزند نسل بعد را تولید می کنیم این به این صورت است که یک گره به شکل کاملا تصادفی از هر دو درخت انتخاب می شود و سپس به شکل تصادفی از راست یا چپ آن یک برش زده می شود تا ۴ تا زیر درخت داشته باشیم حال این ها را به شکل کاملا ضربدری پیوند می زنیم.



حال به شکل تصادفی یکی از این فرزندان را انتخاب می کنیم تا نسل بعدی را تشکیل دهد.

```
def reproduce(tree_1, tree_2):
    set_result_1 = list(set(tree_1.levelorder).difference(set(tree_1.leaves)))
    chosen_node_tree_1_offset = np.random.choice(np.arange(len(set_result_1)), 1)[0]
    chosen_node_tree_1 = set_result_1[chosen_node_tree_1_offset]
    way_1 = True
    if chosen_node_tree_1.right is not None:
        way_1 = bool(np.random.randint(0, 2))
        if way_1:
            temp_node_1 = chosen_node_tree_1.left
        else:
            temp_node_1 = chosen_node_tree_1.right
    else:
        temp_node_1 = chosen_node_tree_1.left
    set_result_2 = list(set(tree_2.levelorder).difference(set(tree_2.leaves)))
    chosen_node_tree_2_offset = np.random.choice(np.arange(len(set_result_2)), 1)[0]
    chosen_node_tree_2 = set_result_2[chosen_node_tree_2_offset]
    way_2 = True
    if chosen_node_tree_2.right is not None:
        way_2 = bool(np.random.randint(0, 2))
        if way_2:
            temp_node_2 = chosen_node_tree_2.left
        else:
            temp_node_2 = chosen_node_tree_2.right
    else:
        temp_node_2 = chosen_node_tree_2.left
    if way_2:
        chosen_node_tree_2.left = temp_node_1
    else:
        chosen_node_tree_2.right = temp_node_1
    if way_1:
        chosen_node_tree_1.left = temp_node_2
    else:
```

```
            temp_node_1 = chosen_node_tree_1.left
        else:
            temp_node_1 = chosen_node_tree_1.right
    else:
        temp_node_1 = chosen_node_tree_1.left
    set_result_2 = list(set(tree_2.levelorder).difference(set(tree_2.leaves)))
    chosen_node_tree_2_offset = np.random.choice(np.arange(len(set_result_2)), 1)[0]
    chosen_node_tree_2 = set_result_2[chosen_node_tree_2_offset]
    way_2 = True
    if chosen_node_tree_2.right is not None:
        way_2 = bool(np.random.randint(0, 2))
        if way_2:
            temp_node_2 = chosen_node_tree_2.left
        else:
            temp_node_2 = chosen_node_tree_2.right
    else:
        temp_node_2 = chosen_node_tree_2.left
    if way_2:
        chosen_node_tree_2.left = temp_node_1
    else:
        chosen_node_tree_2.right = temp_node_1
    if way_1:
        chosen_node_tree_1.left = temp_node_2
    else:
        chosen_node_tree_1.right = temp_node_2
    randomness = bool(np.random.randint(0, 2))
    if randomness:
        return tree_1
    else:
        return tree_2
```

جهش و شرط خاتمه الگوریتم:

به منظور انجام عمل جهش ما با نرخ $\frac{1}{n}$ که در آن n اندازه جمعیت می باشد. هر گره یا راس تشکیل دهنده را جهش می دهیم یعنی با احتمال فوق جهش میابد. فرآیند جهش نیز بدین صورت است که یک درخت تصادفی در گره مورد نظر تولید می شود و جایگیزین آن گره می گردد تولید این درخت تصادفی مشابه همان قسمت اول است. شرط خاتمه الگوریتم نیز از آنجایی که جمعیت رو به سمت کاهش می رود تا زمانی است که جمعیت تهی نشده باشد.

```
j = 0
the_best_trees = []
begin = datetime.datetime.now()
while len(population_1) != 0:
    weights = weighted_by(population_1, initial_points)
    if len(weights.keys()) != 0:
        the_max_value = max(list(weights.values()))
        for element in list(weights.keys()):
            if weights[element] == the_max_value:
                the_best_trees.append((element, the_max_value))
    else:
        break
    population_2 = list()
    list_of_kies = list(weights.keys())
    for i in range(len(list_of_kies)):
        parent_tuple = weighted_random_choice(weights, j)
        parent_1 = parent_tuple[0]
        parent_2 = parent_tuple[1]
        child = reproduce(parent_1, parent_2)
        child_new = mutate(child, np.divide(1, len(population_1)), max_height)
        population_2.append(child_new)
    population_1 = population_2
    j += 1
end = datetime.datetime.now()
for i in range(len(the_best_trees)):
    print(the_best_trees[i][0], the_best_trees[i][1], j,  end - begin, sep="\n\n")
    print("Expression : " + print_formula(the_best_trees[i][0], ''))
```

```
def mutate(child_1_old, mutation_rate, max_height):
    probability = [1 - mutation_rate, mutation_rate]
    respect_child_1 = []
    for node in child_1_old.levelorder:
        Flag = bool(np.random.choice([0, 1], 1, p=probability)[0])
        if Flag:
            i = np.random.randint(1, max_height, 1)[0]
            list_length = np.power(2, i + 1) - 1
            left = int(np.ceil((list_length - 1) / 2))
            variable = [-3] * (list_length - 1)
            random_tree = produce_random_tree(list_length, left, variable)
            respect_child_1.append((node, bt.build(list(random_tree))))
    for old_node, mutated_node in respect_child_1:
        old_node.left = mutated_node
    return child_1_old
```

چالش های مواجه شده و روش حل آنها:

- در روند حل مساله محاسبات اعشاری اعداد ممیز شناور، بسیاری از عبارات در طی محاسبات به **Nan** و **inf** تبدیل می شدند که به دلیل محدودیت های تابعی یا محدودیت های ریاضی درخت مورد نظر اتفاق می افتاد. سعی شد که مقادیری از نقاط داده شده که این چنین خطاهایی را بوجود می آوردند از دامنه اعداد محاسبه شده برای فیتنس خارج شوند.
- سعی شد که ترکیب معقولی از عبارات مورد نظر در هر درخت وجود داشته باشد به گونه ای که درخت تصادفی تشکیل شده شامل عبارت ثابت (تابع ثابت) نباشد.
- چالشی که هنوز بر قوت خود باقی است؛ محاسبه احتمالات در نسل های بالایی است که برنامه را با خطا مواجه می کند و آن را متوقف می کند.(همان مقادیر **Nan** را می گیرد)
- چالشی بعدی نزولی بودن روند جمعیت است که هنوز حل نشده است.

گزارشی از عملکرد الگوریتم و انجام ۳ آزمایش ورودی:

برای تابع $y = \sin x$ یک تقریب زده شد که نتیجه اش در زیر آمده است:

```
[]

   __5.0
  /
-3.0


353.55339

25

0:00:13.371255
[(Node(5.0), 353.55339), (Node(5.0), 353.55339), (Node(5.0), 353.55339), (Node(5.0), 353.
Expression : Sin(x)

Process finished with exit code 0
```

ورودی در خود برنامه به شکل زیر آمده است:

```
initial_point = (np.arange(40), np.sin(np.arange(40) * np.pi / 180))
```

برای تابع $y = \tan x$ یک تقریب زده شد که نتیجه اش در زیر آمده است:

```
   __2.0__________
  /               \
-3.0             _0.0_
                /     \
            __5.0     -3.0
           /
         -3.0


316.22777

Generation: 61

0:00:22.909724
Expression : (x*(Sin(x)+x))
```

ورودی به شکل روبرو می باشد:

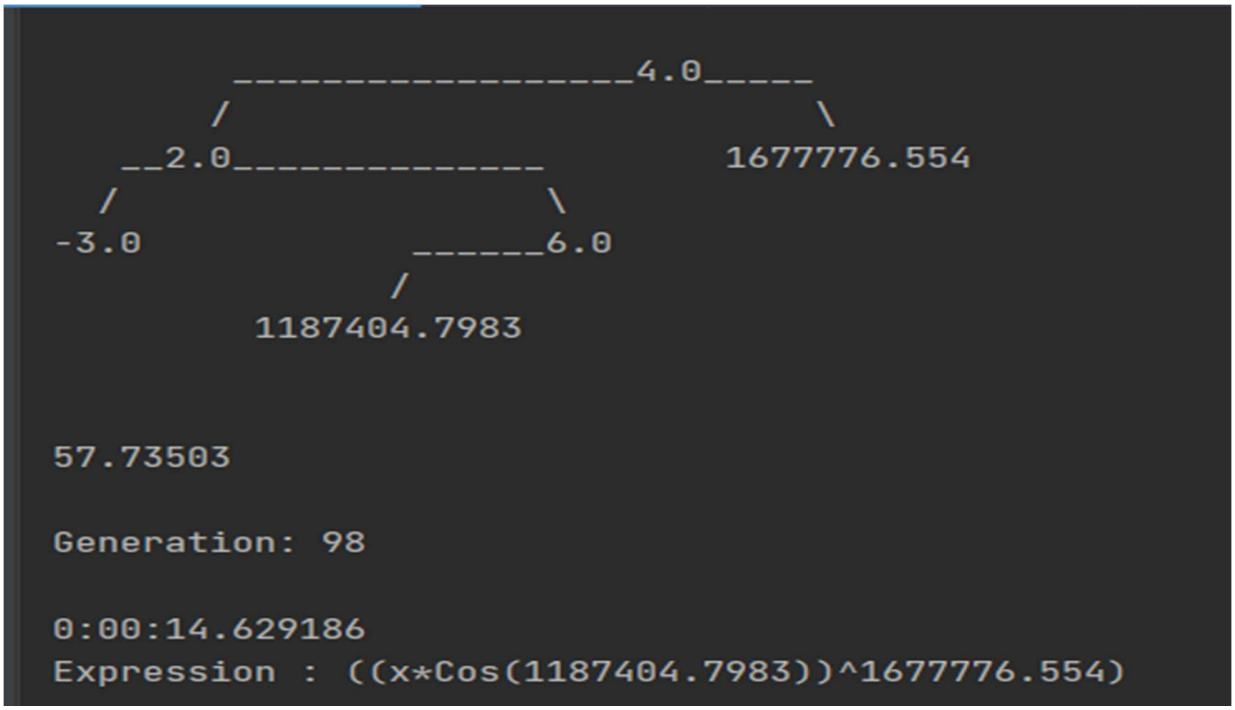
```
initial_points = (np.hstack((before_1, before_2)), np.hstack((np.tan(before_1 * np.pi / 180), np.tan(before_2 * np.pi / 180)
```

برای یک تابع کاملا نامشخص تعدادی نقطه تصادفی تولید شد و سپس آزمایش گردید.

```
                      ______2.0____________________                    -3.0
                     /                             \
          ___________2.0_                  ___________3.0______
         /            \                   /                    \
     ____0.0____       -3.0          __4.0____              __2.0_
    /           \                    /         \            /      \
993.4499     1885.8904             -3.0     1149.3903     -3.0     -3.0


0.0042

Generation: 24

0:00:29.429553
Expression : ((((993.4499+1885.8904)*x)*((x^1149.3903)/(x*x)))-x)
```

```
initial_points = (np.hstack((before_1, before_2)), np.hstack((np.random.randint(0, 3000, 10), np.random.randint(0, 3000, 10))
```

برای نقاط گسستگی نیز مثال تبصره ۳ از داکیومنت پروژه را در نظر گرفتیم:



که ورودی اش به شکل پایین است:

```
before_1 = np.array([-4, -5, -6, -7, -8, -9, -10, -11, -12, -13])
after_1 = (-1) * before_1 - 1
before_2 = np.array([2, 3, 5, 6, 34, 234, 13, 3, 2123, 42, 1, 3, 21, 32, 13, 1])
after_2 = np.power(before_2, 2)
initial_points = (np.hstack((before_1, before_2)), np.hstack((after_1, after_2)))
```

علی رغم وجود باگ در کد و داشتن پتانسیل بهبود، این الگوریتم از کارایی لازم برای مسائل عملی برخورددار است وبسیاری از مسائل دنیای امروز که در برگیرنده جست و جوی محلی و عناصر طبیعی هستند را می توان مدل نمود.