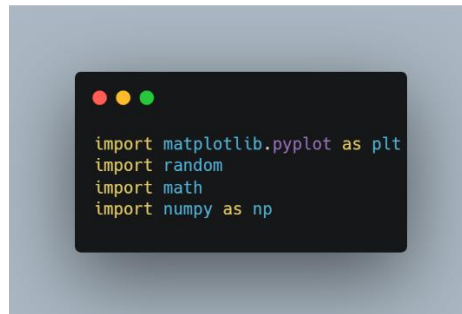


به نام خدا

بهاره کاوسی نژاد – 99431217

پروژه سوم درس هوش مصنوعی – برنامه نویسی ژنتیک

ابتدا کتابخانه های لازم را import می کنیم.



```
import matplotlib.pyplot as plt
import random
import math
import numpy as np
```

کلاس Node را تعریف می کنیم؛ از این کلاس برای نمایش عبارات ریاضی به صورت درختی استفاده می شود. این کلاس از بخش های زیر تشکیل شده است:

- در constructor این کلاس، سه attribute داریم: value و children
- در تابع evaluate مقدار گره کنونی و فرزندانش را تخمین می زنیم؛ بدین صورت که
 - اگر value گره کنونی برابر با None بود، مقدار 0 را برمیگرداند
 - اگر مقدار value گره کنونی یک رشته بود، بدین معنا است که این گره یک عملگر یا یک تابع است؛ این تابع به صورت بازگشتی و با تخمین زدن فرزندانش، مقدار عملگر یا تابع مربوطه را به دست می آورد. به عنوان مثال اگر مقدار value، '+' باشد، فرزند اول و دوم را با هم جمع می کند.
 - اگر مقدار value گره کنونی یک عدد (int یا float) باشد، همان مقدار عددی را بازمیگرداند.
 - اگر مقدار value یک دیکشنری (یعنی متغیر) باشد، نام متغیر را از دیکشنری گرفته و چک می کند که در لیست available_variables قرار دارد یا خیر. اگر نام متغیر در این لیست موجود بود، value مربوطه را از لیست input_values بازمیگرداند.
 - برای value های unknown مقدار 0 را برمیگرداند.
- تابع print_function، یک رشته حاوی تابع نمایش داده شده توسط گره را برمیگرداند.
 - اگر value، None باشد، به این معنا است که به آن گره مقداری assign نشده است، پس رشته '0' را برمیگرداند.
 - اگر value، یک رشته (یعنی عملگر یا تابع) باشد، به صورت بازگشتی print_function را روی گره فرزند صدا می کند تا آنها را به صورت رشته به همراه مقدار value گره کنونی نمایش دهد.
 - اگر مقدار value گره کنونی یک عدد (int یا float) باشد، آن را به رشته تبدیل کرده و بازمیگرداند.

- اگر مقدار value یک دیکشنری (یعنی متغیر) باشد، نام متغیر را از دیکشنری گرفته و آن را به شکل یک رشته بازمیگرداند.
- برای valueهای unknown مقدار '0' را برمیگرداند.

```
class Node:
    def __init__(self, type=None, value=None):
        self.type = type
        self.value = value
        self.children = []

    def evaluate(self, input_values, available_variables):
        if self.value is None:
            return 0
        elif isinstance(self.value, str): # Handle operators and functions
            if self.value == '+':
                return self.children[0].evaluate(input_values, available_variables) +
self.children[1].evaluate(input_values, available_variables)
            elif self.value == '-':
                return self.children[0].evaluate(input_values, available_variables) -
self.children[1].evaluate(input_values, available_variables)
            elif self.value == '*':
                return self.children[0].evaluate(input_values, available_variables) *
self.children[1].evaluate(input_values, available_variables)
            elif self.value == '/':
                divisor = self.children[1].evaluate(input_values, available_variables)
                if divisor != 0:
                    return self.children[0].evaluate(input_values, available_variables) / divisor
                else:
                    return 0
            elif self.value == '^':
                base = self.children[0].evaluate(input_values, available_variables)
                exponent = self.children[1].evaluate(input_values, available_variables)
                if base != 0 or (exponent >= 0 and exponent.imag == 0):
                    return base ** exponent
                else:
                    return 0
            elif self.value == 'sin':
                return math.sin(self.children[0].evaluate(input_values, available_variables))
            elif self.value == 'cos':
                return math.cos(self.children[0].evaluate(input_values, available_variables))
            else:
                return 0 # Return 0 for unknown operators/functions
        elif isinstance(self.value, (int, float)): # Handle numeric constants
            return float(self.value)
        elif isinstance(self.value, dict): # Handle variables
            variable_name = self.value['symbol']
            if variable_name in available_variables:
                index = available_variables.index(variable_name)
                return input_values[index]
            else:
                return 0 # Return 0 for unknown value types

    def print_function(self):
        if self.value is None:
            return '0'
        elif isinstance(self.value, str):
            if self.value in ('+', '-', '*', '/', '^'):
                return f"({self.children[0].print_function()} {self.value}
self.children[1].print_function())"
            elif self.value in ('sin', 'cos'):
                return f"{self.value}({self.children[0].print_function()})"
            else:
                return '0' # Return '0' for unknown operators/functions
        elif isinstance(self.value, (int, float)):
            return str(self.value)
        elif isinstance(self.value, dict):
            return self.value['symbol']
        else:
            return '0' # Return '0' for unknown value types
```

- تابع `create_random_individual` برای تولید `random individuals` با عمق های متفاوت برای برنامه نویسی ژنتیک استفاده می شود.

- در ابتدا چک می کند که `max_depth` ورودی، کمتر یا مساوی با 0 است یا یک عدد تصادفی بین 0 تا 1 از 0.1 کوچکتر است یا خیر. اگر هر کدام از این شروط درست باشند، بدین معنا است که یک گره پایانی (terminal node) باید تولید شود (constant or variable). در این صورت، به صورت تصادفی انتخاب می شود که گره پایانی constant باشد یا variable. اگر constant باشد، یک عدد float بین -10 تا 10 تولید می شود. در غیر این صورت (انتخاب variable)، یک symbol به صورت تصادفی از لیست `available_variables` انتخاب می شود و یک دیکشنری با کلید 'symbol' و مقدار symbol انتخاب شده ساخته می شود. در نهایت، یک گره با value تعیین شده تشکیل می شود و این گره بازگردانده می شود.

- اگر شرط ساختن گره پایانی برقرار نشود، بخش بعدی کد اجرا می شود مشخص می کند یک گره تابع (عملگر) باید تولید شود. این قسمت به صورت تصادفی یک عملگر از لیست `available_operators` انتخاب می کند. یک گره جدید با value این عملگر انتخاب شده تولید می شود. فرزندان این گره به صورت تصادفی توسط تابع `create_random_individual` تولید می شوند که مقدار `max_depth` در آنها کاهش یافته اما همان `available_operators` و `available_variables` را داریم. تعداد فرزندان با مقدار `arity` عملگر انتخاب شده تعیین می شود.

```
def create_random_individual(max_depth, available_operators, available_variables):
    if max_depth <= 0 or random.random() < 0.1: # Terminal node (constant or variable)
        if random.random() < 0.5: # Generate constant
            value = random.uniform(-10, 10)
        else: # Generate variable
            symbol = random.choice(available_variables)
            value = {'symbol': symbol}
        return Node(value=value)
    else: # Function node (operator)
        operator = random.choice(available_operators)
        individual = Node(value=operator)
        individual.children = [create_random_individual(max_depth - 1, available_operators,
            available_variables) for _ in range(operator['arity'])]
        return individual
```

- تابع crossover، در برنامه نویسی ژنتیک برای ایجاد فرزند استفاده می شود. در ابتدا مقدار value فرزند برابر با value والد اول (عملگر یا تابع) قرار داده می شود.
- با استفاده از تابع zip یک پیمایش روی فرزندان والد اول و دوم انجام می شود و به صورت بازگشتی تابع crossover صدا زده می شود تا گره های فرزند ایجاد شده و در یک لیست ذخیره شوند.
- در انتها نیز گره فرزند ایجاد شده بازگردانده می شود.

```
def crossover(parent1, parent2):
    child = Node(value=parent1.value)
    child.children = [crossover(child1, child2) for child1, child2 in zip(parent1.children,
parent2.children)]
    return child
```

- تابع mutation در برنامه نویسی ژنتیک برای ارائه تغییرات تصادفی در یک جمعیت استفاده می شود.
- این تابع در ابتدا چک می کند که max_depth از 0 بزرگتر باشد. در این صورت می توان جهش بیشتری روی یک گره اعمال کرد. در این صورت، با استفاده از enumerate روی تمامی گره های فرزند پیمایش می کند و در هر گره فرزند به صورت بازگشتی تابع mutation را صدا می زند و max_depth را برای آن یک واحد کمتر در نظر میگیرد.
- اگر نتوان جهش بیشتری روی یک گره اعمال کرد، بدین معنا است که یک برگ باید جهش یابد. در ابتدا چک می کند که مقدار value یک رشته است یا خیر. اگر رشته بود، چک می کند که آیا در میان عملگرها و متغیرهای موجود می توان این رشته را پیدا کرد یا خیر. در صورت پیدا شدن، به صورت تصادفی از میان گزینه های موجود یک مقدار دیگر برای آن انتخاب می کند و در غیر این صورت، یک عدد تصادفی بین -10 تا 10 را به value آن assign می کند.
- اگر مقدار value رشته نبود، مستقیماً یک عدد تصادفی بین -10 تا 10 را به عنوان value در نظر میگیرد.

```
def mutation(individual, max_depth, available_operators, available_variables):
    if max_depth > 0:
        for idx, child in enumerate(individual.children):
            individual.children[idx] = mutation(child, max_depth - 1, available_operators,
available_variables)
    else: # Mutate leaf node
        if isinstance(individual.value, str):
            if individual.value in available_operators:
                individual.value = random.choice(available_operators)
            elif individual.value in available_variables:
                individual.value = random.choice(available_variables)
            else:
                individual.value = random.uniform(-10, 10)
        else:
            individual.value = random.uniform(-10, 10)
    return individual
```

- تابع `evaluate_fitness` برای ارزیابی یک `individual` در برنامه نویسی ژنتیک استفاده می شود. این ارزیابی بر اساس `mean squared error` بین خروجی های تولید شده توسط `individual` و خروجی های هدف انجام می شود. هرچه `MSE` کمتر باشد، به معنای `fitness` و خروجی های بهتر است.
- در ابتدا مقدار اولیه `total_error` صفر قرار داده می شود.
- با استفاده از تابع `zip` روی `input_values` و `target_output` پیمایش انجام می شود. لیست های `input_data` و `target_outputs` به صورت جفت جفت پیمایش می شوند.
- در مرحله بعدی ارزیابی انجام می شود و تابع `evaluate` برای هر گره صدا زده می شود و در `output` ذخیره می شود.
- اگر `output`، `None` نباشد (ارزیابی با موفقیت انجام شده باشد)، `error` به صورت اختلاف بین `output` و `target_output` محاسبه می شود. توان دوم این مقدار به `total_error` اضافه می شود.
- `mse` به صورت `total_error` تقسیم بر طول `input_data` محاسبه و بازگردانده می شود.

```
def evaluate_fitness(individual, input_data, target_outputs,
                    available_variables):
    for input_values, target_output in zip(input_data, target_outputs):
        output = individual.evaluate(input_values, available_variables)
        if output is not None:
            error = output - target_output
            total_error += error ** 2
    mse = total_error / len(input_data)
    return mse
```

- تابع select_parents برای انتخاب والدها از یک جمعیت برای operation ژنتیکی استفاده می شود.
- این تابع با استفاده از list comprehension برای محاسبه fitness score برای هر individual در جمعیت محاسبه می شود. این مقدار در یک لیست با نام fitness_scores ذخیره می شود.
- با استفاده از تابع sum مقدار total_fitness را محاسبه می کنیم.
- احتمال انتخاب هر individual به عنوان والد در probabilities به صورت fitness آن individual تقسیم بر total_fitness به دست می آید.
- با استفاده از تابع random.choices دو والد از جمعیت انتخاب می کنیم و در لیست parents ذخیره می کنیم و آن را برمیگردانیم.

```
def select_parents(population, input_data, target_outputs, available_variables):
    fitness_scores = [evaluate_fitness(individual, input_data, target_outputs, available_variables) for
    individual in population]
    total_fitness = sum(fitness_scores)
    probabilities = [fitness / total_fitness for fitness in fitness_scores]
    parents = random.choices(population, probabilities, k=2)
    return parents
```

- این تابع برنامه نویسی ژنتیک را پیاده سازی می کند:
- با استفاده از تابع create_random_individual جمعیت اولیه را می سازیم و در population ذخیره می کنیم.
- متغیرهای best_fitness، best_individual، unchanged_iterations، best_fitness_history و mse_history را مقداردهی اولیه می کنیم.
- در یک حلقه for تا max_generations پیش می رویم:
 - یک لیست به نام new_population برای ذخیره offspring یا فرزندان نسل کنونی می سازیم.
 - در یک حلقه while، فرزندان را می سازیم. این حلقه تا زمانی ادامه میابد که اندازه جمعیت جدید با جمعیت اصلی برابر شود. با استفاده از تابع select_parents دو والد انتخاب می کند و بعد از crossover و mutation، فرزند را به لیست new_population اضافه می کند.
 - در مرحله بعد نسل جدید را جایگزین نسل قبلی می کنیم.
 - در یک حلقه for، fitness هر individual در جمعیت را با استفاده از تابع evaluate_fitness تخمین می زند. مقدار fitness هر individual با مقدار best_fitness کنونی مقایسه شده و اگر از آن بهتر (کمتر) بود، individual فعلی به new best individual جدید تبدیل می شود و fitness آن به new best fitness تبدیل می شود. همچنین شمارنده unchanged_iterations به 0 ریست می شود. اگر

fitness یک individual از best_fitness بهتر نبود، یک واحد به unchanged_iterations اضافه می شود.

- مقدار (MSE) best_fitness و best_fitness کنونی در دو لیست best_fitness_history و mse_history ذخیره می شوند تا بتوانیم تغییرات و پیشرفت fitness در طی نسل ها را بررسی کنیم.
- اگر مقدار best_fitness صفر بود، بدین معنا است که تابع هدف به بهترین صورت ممکن حدس زده شده است؛ پس از حلقه خارج می شویم.
- اگر شمارنده unchanged_iterations به unchanged_threshold مشخص شده رسیده باشد، جهش ژنتیکی به best_individual کنونی اعمال می شود تا variation ایجاد شود و احتمالاً باعث بهبود fitness می شود. شمارنده unchanged_iterations به صفر ریست می شود.
- تابع plot_fitness_history برای نمایش (MSE) fitness در طول نسل ها صدا زده می شود و best_individual بازگردانده می شود.

```
def genetic_programming(input_data, target_outputs, available_operators, available_variables,
                        population_size=100, max_generations=100, max_depth=5, unchanged_threshold=10):
    population = [create_random_individual(max_depth, available_operators, available_variables) for _ in
                  range(population_size)]
    best_individual = None
    best_fitness = float('inf')
    unchanged_iterations = 0
    best_fitness_history = [] # Store the best fitness value for each generation
    mse_history = []
    for generation in range(max_generations):
        new_population = []

        while len(new_population) < population_size:
            parents = select_parents(population, input_data, target_outputs, available_variables)
            offspring = crossover(parents[0], parents[1])
            offspring = mutation(offspring, max_depth, available_operators, available_variables)
            new_population.append(offspring)

        population = new_population

        for individual in population:
            fitness = evaluate_fitness(individual, input_data, target_outputs, available_variables)
            if fitness < best_fitness:
                best_individual = individual
                best_fitness = fitness
                unchanged_iterations = 0
            else:
                unchanged_iterations += 1

        best_fitness_history.append(best_fitness) # Store the best MSE value for the current generation
        mse_history.append(best_fitness) # Store the MSE value for the current generation

        if best_fitness == 0:
            break

        if unchanged_iterations >= unchanged_threshold:
            best_individual = mutation(best_individual, max_depth, available_operators,
                                      available_variables)
            unchanged_iterations = 0

    plot_fitness_history(mse_history)

    return best_individual
```


- تابع `plot_fitness_history` برای رسم `mse` ها استفاده می شود.

```
def plot_fitness_history(mse_history):  
    plt.plot(mse_history)  
    plt.xlabel('Generation')  
    plt.ylabel('Mean Squared Error')  
    plt.title('Genetic Programming  
Progress')  
    plt.show()
```

- از دو تابع `generate_inputs` و `generate2D_inputs` برای تولید ورودی ها بین 10- تا 10 استفاده می شود.

```
def generate_inputs(num_inputs):  
    inputs = []  
    for _ in range(num_inputs):  
        x = random.uniform(-10, 10)  
        inputs.append((x,))  
    return inputs  
  
def generate2D_inputs(num_inputs):  
    inputs = []  
    for _ in range(num_inputs):  
        x = random.uniform(-10, 10)  
        y = random.uniform(-10, 10)  
        inputs.append((x, y))  
    return inputs
```


- در این قسمت ورودی های هر بخش پروژه مشخص شده است.

```
# Part 1
input_data = generate_inputs(100) # Generate 100 input data points
target_outputs = [math.atan(x[0]) + 5 for x in input_data] # Define target outputs for the inputs
available_variables = ['x']

# Part 2
input_data = generate_inputs(500) # Generate 500 input data points
target_outputs = [(x[0]/5 + 1) if x[0] > 0 else (x[0] ** 3 + 5) for x in input_data] # Define target outputs for the inputs
available_variables = ['x']

# Part 4
input_data = generate2D_inputs(500) # Generate 500 input data points
target_outputs = [2 * x[0] + 3 * x[1] for x in input_data] # Define target outputs for the inputs
available_variables = ['x', 'y']
```

- در available_operators عملگرهای ممکن با arity آنها را ذخیره می کنیم.

```
available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '*', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]
```

```
best_function = genetic_programming(input_data, target_outputs, available_operators,
print(b"Best function found: ", best_function.print_function())
```

نتائج:

قسمت اول:

• تابع $3x+1$

```
# Example usage:
# input_data = [(1,), (2,), (3,), (4,)]
# target_outputs = [3, 5, 7, 9]
input_data = generate_inputs(100) # Generate 100 input data points
target_outputs = [3 * x[0] + 1 for x in input_data] # Define target outputs for the inputs

available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '*', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]
available_variables = ['x']

best_function = genetic_programming(input_data, target_outputs, available_operators, available_variables)
print("Best function:", best_function.print_function())

Best function: x
```

• تابع $x/4+5$

```
# Example usage:
# input_data = [(1,), (2,), (3,), (4,)]
# target_outputs = [3, 5, 7, 9]
input_data = generate_inputs(100) # Generate 100 input data points
target_outputs = [x[0]/4 + 5 for x in input_data] # Define target outputs for the inputs

available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '*', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]
available_variables = ['x']

best_function = genetic_programming(input_data, target_outputs, available_operators, available_variables)
print("Best function:", best_function.print_function())

Best function: x
```

• تابع x^2+5

```
# Example usage:
# input_data = [(1,), (2,), (3,), (4,)]
# target_outputs = [3, 5, 7, 9]
input_data = generate_inputs(100) # Generate 100 input data points
target_outputs = [(x[0]**2) + 5 for x in input_data] # Define target outputs for the inputs

available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '*', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]
available_variables = ['x']

best_function = genetic_programming(input_data, target_outputs, available_operators, available_variables)
print("Best function:", best_function.print_function())

Best function: 6.3800066317248785
```

• تابع $\arctan(x)$

```

358 # Example usage:
# input_data = [(1,), (2,), (3,), (4,)]
# target_outputs = [3, 5, 7, 9]
input_data = generate_inputs(100) # Generate 100 input data points
target_outputs = [math.atan(x[0]) + 5 for x in input_data] # Define target outputs for the inputs

available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '*', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]

available_variables = ['x']

best_function = genetic_programming(input_data, target_outputs, available_operators, available_variables)
print("Best function:", best_function.print_function())

```

Best function: 5.223365747544291

قسمت دوم: ایجاد نقاط گسستگی

• تابع $2x+1$ برای x های کوچکتر از 0 و x^3+5 برای سایر نقاط

```

358 # Example usage:
# input_data = [(1,), (2,), (3,), (4,)]
# target_outputs = [3, 5, 7, 9]
input_data = generate_inputs(100) # Generate 100 input data points

# Part 1
# target_outputs = [math.atan(x[0]) + 5 for x in input_data] # Define target outputs for the inputs

# Part 2
target_outputs = [(2 * x[0] + 1) if x[0] < 0 else (x[0] ** 3 + 5) for x in input_data] # Define target outputs for the inputs

available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '*', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]

available_variables = ['x']

best_function = genetic_programming(input_data, target_outputs, available_operators, available_variables)
print("Best function:", best_function.print_function())

```

Best function: x

- تابع $x/5+1$ برای x های بزرگتر از 0 و x^3+5 برای سایر نقاط

```
# Example usage:
# input_data = [(1,), (2,), (3,), (4,)]
# target_outputs = [3, 5, 7, 9]
input_data = generate_inputs(500) # Generate 100 input data points

# Part 1
# target_outputs = [math.atan(x[0]) + 5 for x in input_data] # Define target outputs for the inputs

# Part 2
target_outputs = [(x[0]/5 + 1) if x[0] > 0 else (x[0] ** 3 + 5) for x in input_data] # Define target outputs for the inputs

available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '*', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]

available_variables = ['x']

best_function = genetic_programming(input_data, target_outputs, available_operators, available_variables)
print("Best function:", best_function.print_function())
```

Best function: x

قسمت چهارم: توابع با بیشتر از یک بعد

- تابع $2x + 3y$

```
# Part 3
input_data = generate2D_inputs(500) # Generate 500 input data points
target_outputs = [2 * x[0] + 3 * x[1] for x in input_data] # Define target outputs for the inputs
available_variables = ['x', 'y']

available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '*', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]

best_function = genetic_programming(input_data, target_outputs, available_operators, available_variables)
print("Best function:", best_function.print_function())
```

Best function: x

قسمت پنجم: بررسی تاثیر جهش

- تابع دو بعدی: $2x+3y$

```
# Part 4
input_data = generate2D_inputs(500) # Generate 500 input data points
target_outputs = [2 * x[0] + 3 * x[1] for x in input_data] # Define target outputs for the inputs
available_variables = ['x', 'y']

available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '*', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]

best_function = genetic_programming(input_data, target_outputs, available_operators, available_variables)
print("Best function:", best_function.print_function())
```

Best function: y

- تابع با گسستگی: $x/5+1$ برای x های بزرگتر از 0 و x^3+5 برای سایر نقاط

```
# input_data = generate_inputs(500) # Generate 500 input data points
# target_outputs = [math.atan(x[0]) + 5 for x in input_data] # Define target outputs for the inputs
# available_variables = ['x']

# Part 2
input_data = generate_inputs(500) # Generate 500 input data points
target_outputs = [(x[0]/5 + 1) if x[0] > 0 else (x[0]**3 + 5) for x in input_data] # Define target outputs for the inputs
available_variables = ['x']

# Part 4
input_data = generate2D_inputs(500) # Generate 500 input data points
target_outputs = [2 * x[0] + 3 * x[1] for x in input_data] # Define target outputs for the inputs
available_variables = ['x', 'y']

available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '*', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]

best_function = genetic_programming(input_data, target_outputs, available_operators, available_variables)
print("Best function:", best_function.print_function())
```

Best function: x

• تابع $\arctan(x) + 5$

```
338 ✓ ▶ input_data = generate_inputs(100) # Generate 100 input data points
target_outputs = [math.atan(x[0]) + 5 for x in input_data] # Define target outputs for the inputs
available_variables = ['x']

# Part 2
# input_data = generate_inputs(500) # Generate 500 input data points
# target_outputs = [(x[0]/5 + 1) if x[0] > 0 else (x[0] ** 3 + 5) for x in input_data] # Define target outputs for the inputs
# available_variables = ['x']

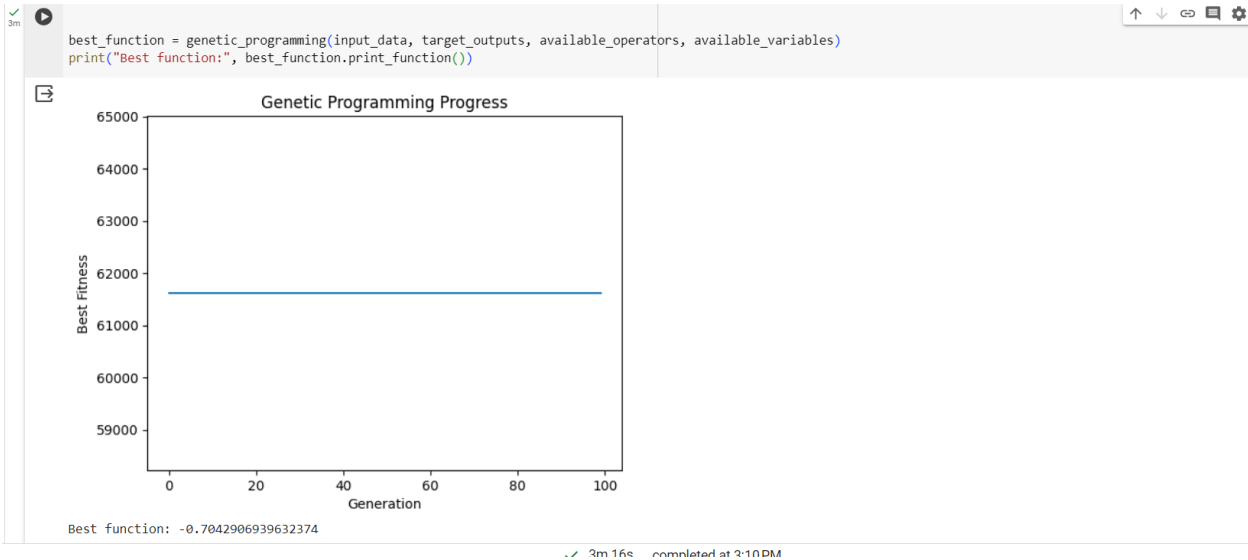
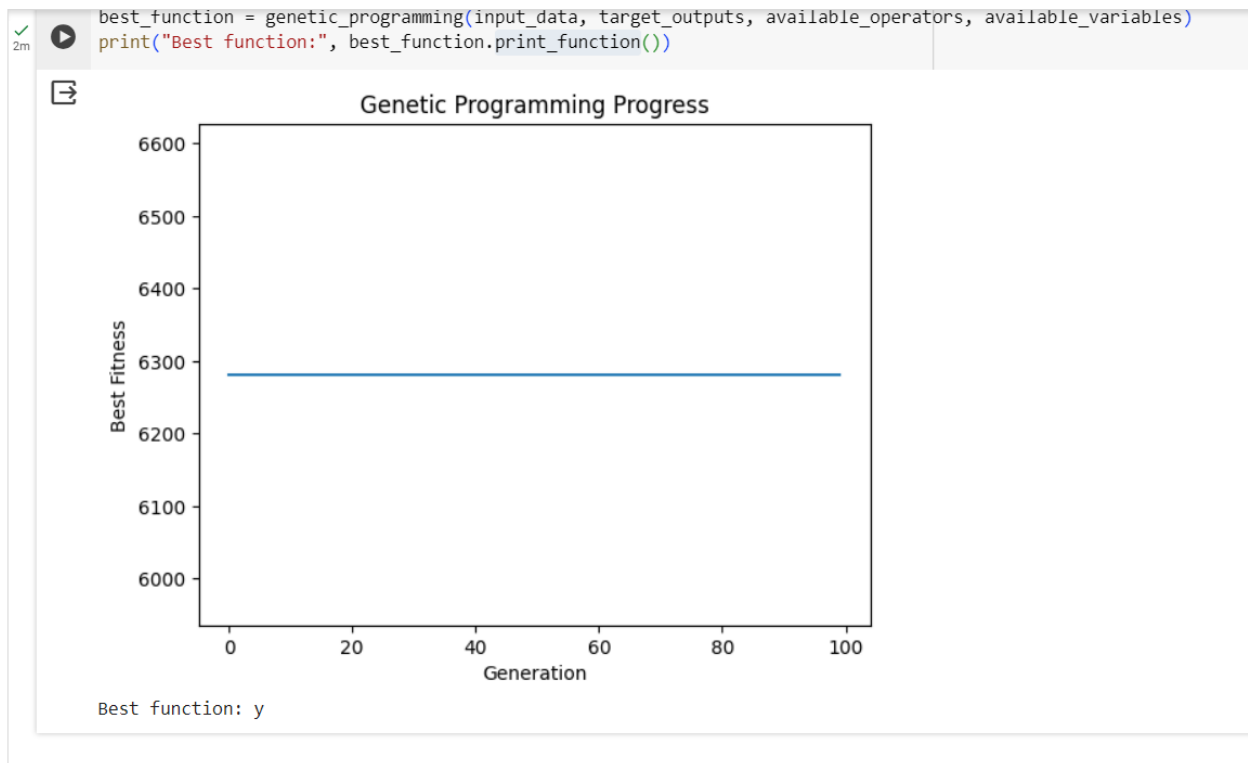
# Part 4
# input_data = generate2D_inputs(500) # Generate 500 input data points
# target_outputs = [2 * x[0] + 3 * x[1] for x in input_data] # Define target outputs for the inputs
# available_variables = ['x', 'y']

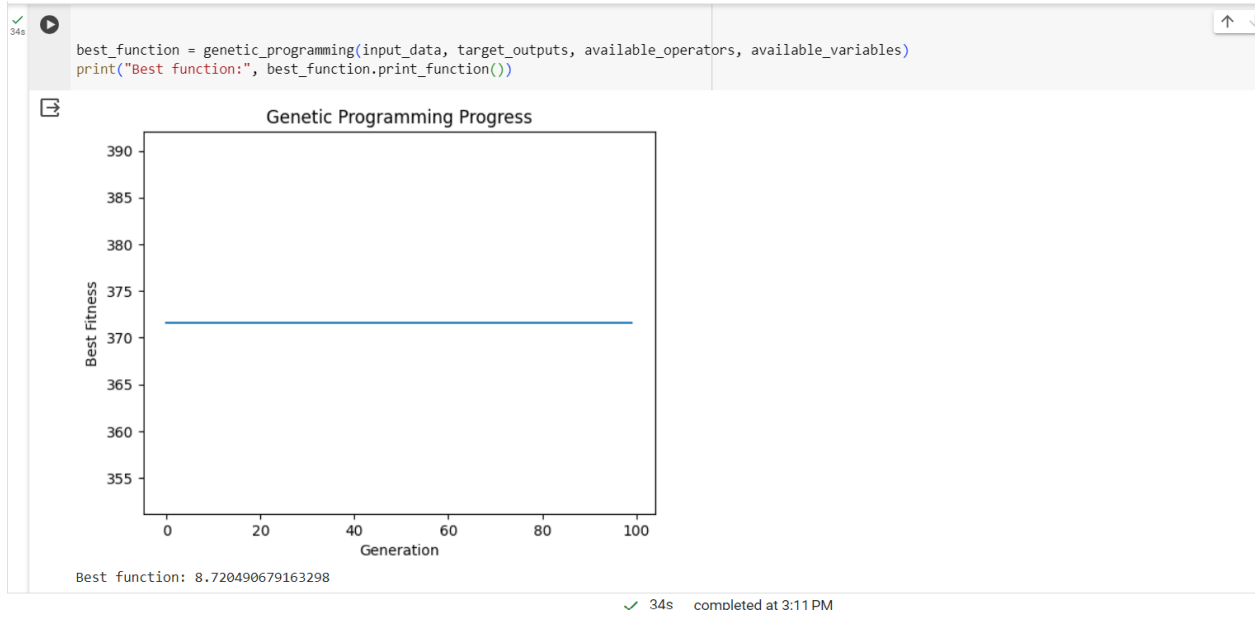
available_operators = [
    {'symbol': '+', 'arity': 2},
    {'symbol': '-', 'arity': 2},
    {'symbol': '**', 'arity': 2},
    {'symbol': '/', 'arity': 2},
    {'symbol': '^', 'arity': 2},
    {'symbol': 'sin', 'arity': 1},
    {'symbol': 'cos', 'arity': 1}
]

best_function = genetic_programming(input_data, target_outputs, available_operators, available_variables)
print("Best function:", best_function.print_function())
```

Best function: 5.100644831721521

قسمت نهم: روند پیشرفت الگوریتم برای ورودی های مختلف





:MSE

