مجید فریدفر

۸۱۰۱۹۹۵۶۹

گزارشکار تمرین کامپیوتری دوم هوش مصنوعی (ترم بهار ۱۴۰۲)

بخش اول: ژنتیک

در ابتدا، دو تا کلاس تعریف می‌کنیم:

1. Stock

```
class Stock:
    _name: str
    _return: float
    _risk: float
```

در اینجا _name نام سهم، _return مقدار برگشتی (سود) سهم و _risk مقدار ریسک سرمایه گذاری روی این سهم است.

2. Chromosome

```
class Chromososme:
    genes: list[float]
```

هر کروموزم (که معادل یک جواب برای مسئله است)، تشکیل شده از تعدادی ژن. که این ژن‌ها نشاندهده‌ی ضریب آن سهم در سرمایه‌ی ما (مقداری بین صفر و یک) است. پس تعداد ژن‌ها، برابر تعداد سهم‌ها و مجموع مقادیر آن‌ها، برابر یک است.

در ادامه اعمال mutate و crossover را پیاده‌سازی می‌کنیم:

1. Mutate

به عنوان یک method در کلاس کروموزم تعریف می‌کنیم.

```
    def mutate(self, probabilty):
        for i in range(len(self.genes)):
            if random.uniform(0, 1) < probabilty:
                self.genes[i] = random.random()

        self.set_sum_to_one()
```

در این جا، هر ژنی با توجه به احتمال میوتیشن، شانس تغییر دارد. در نهایت با استفاده از متود set_sum_to_one با تقسیم مقادیر تمام ژن‌ها، به جمع کل، کاری می‌کنیم که جمع سهم‌ها یک بشود:

```
    def set_sum_to_one(self):
        total = sum(self.genes)
        self.genes = [a/total for a in self.genes]
```

2. Crossover

```
def uniform_cross_over(parents: list[Chromososme], probability):
    num_of_genes = len(parents[0].genes)

    childs = [Chromososme([0]*num_of_genes),
              Chromososme([0]*num_of_genes)]

    for i in range(num_of_genes):
        for_0 = 1 if random.uniform(0, 1) < probability else 0

        childs[0].genes[i] = parents[for_0].genes[i]
        childs[1].genes[i] = parents[1 - for_0].genes[i]

    childs[0].set_sum_to_one()
    childs[1].set_sum_to_one()

    return childs
```

همان طور که مشاهده می‌شود در این این تابع با دریافت دو کروموزوم (در آرایه‌ی parents)، و با توجه احتمال کراس‌اور، تمام ژن‌ها می‌توانند باهم قاطی بشوند. در نهایت دو کروموزم تولید شده (در آرایه‌ی childs) را بعد از نرمالایز کردن برمی‌گردانیم.

حالا تعریف fitness function را ارائه می‌دهیم:

```
    def fitness_function(self, stocks: list[Stock]):
        fitness = 0

        for i in range(len(self.genes)):
            fitness += self.genes[i] * stocks[i]._return * RETURN_WEIGHT
            fitness -= self.genes[i] * stocks[i]._risk * RISK_WEIGHT

        fitness += self.get_num_of_purhased_stocks() * NUM_OF_PURCHASED_STOCKS_WEIGHT

        return fitness
```

در این تابع، برای هر کرموزوم، از سه ویژگی زیر استفاده کرده‌ایم:

۱. مقدار کل سود:

برابر جمع مقدار سود هر سهم (ضربِ ضریب ارائه شده در مقدار سود سهم).

۲. مقدار کل ریسک:

برابر جمع مقادیر ریسک هر سهم (ضرب مقدار ضریب ارائه شده در مقدار ریسک سهم).

۳. تعداد سهم‌های خریداری شده:

که با استفاده از متود get_num_of_purchased_stocks محاسبه شده است.

هم چنین بر هر کدام، یک وزن نسبت می‌دهیم که مشخص می‌کند تاثیر آن در fitness function چقدر خواهد بود.

```
RETURN_WEIGHT = 7
RISK_WEIGHT = 1
NUM_OF_PURCHASED_STOCKS_WEIGHT = 0.01
```

همان طور که مشاهده می‌شود بیشترین تاثیر را مقدار return و کم‌ترین تاثیر را تعداد سهم‌های خریداری شده دارا هستند.

در ادامه به توضیح کلاس اصلی مسئله می‌پردازیم:

```
class PortfolioManagement:
    stocks_information: list[Stock]
    population: list[Chromososme]
    population_size: int
    fitness_scores: list
    num_of_elites: int
    generation_number: int
    mating_pool: list[Chromososme]
    cross_over_rate: int
    cross_over_probability: int
    mutation_rate: int
    mutation_probability: int
```

که دارای فیلدهای زیر است:

- لیستی از اطلاعات مربوط به سهم‌ها در stocks_information.
- لیستی از کروموزوم‌ها مربوط به نسل کنونی در population.
- تعداد کروموزم‌ها در population_size.
- لیستی مرتب شده از کرومزوم‌های حاضر در نسل به ترتیبِ fitness score مربوطه در fitness_scores. که به این شکل محاسبه می‌شود:

```
    def rank_by_fitness_scores(self):
        self.fitness_scores = []

        for chromosome in self.population:
            self.fitness_scores.append((chromosome,
chromosome.fitness_function(self.stocks_information)))

        self.fitness_scores.sort(key = lambda x: x[1])
```

- تعداد کروموزوم‌های نخبه در num_of_elites. یعنی کروموزوم‌هایی که بین جمعیت کنونی از بقیه بهترند. آن‌ها مستقیما به نسل بعدی می‌روند و وارد mating pool نخواهند شد.
- شماره‌ی نسل در generation_number.
- Mating_pool لیستی از کروموزوم‌های غیر نخبه است که برای اعمال کراس‌اور و میوتیشن انتخاب شده‌اند. که به این شکل محاسبه می‌شود که به هر کوروموزوم احتمالی نسبت میدهیم. حالا با توجه به این احتمال، کروموزوم در این لیست قرار خواهد گرفت. در نهایت لیست را بُر میزنیم (roulette wheel):

```
    def select_parents_for_mating_pool(self):
        total = ((len(self.population))*(len(self.population)+1))/2

        self.mating_pool = []
        for i in range(self.population_size - self.num_of_elites):
            random_number = int(random.random()*total)
            s_index = 0

            for j in range(len(self.population)):
                s_index += j+1

                if s_index >= random_number:
                    self.mating_pool.append(self.fitness_scores[j][0])
                    break

        random.shuffle(self.mating_pool)
```

- Cross_over_probability: احتمال جابه‌جایی در دو ژن در هنگام عمل کراس اور.
- Mutaiton_probability: احتمال تغییر هر ژن در هنگام میوتیشن.
- Cross_over_rate: احتمال انجام کراس اور روی دو کروموزوم.
- Mutation_rate: احتمال انجام میوتیشن روی هر کروموزوم.

در ابتدا فایل مربوط به سهم‌ها را می‌خوانیم و اطلاعات آن‌ها را در stocks_information ذخیر می‌کنیم:

```
    def read_input(self, filename):
        with open(filename, 'r') as file:
            next(file)

            for line in file:
                _number, _name, _risk, _return = line.split(',')
                self.stocks_information.append(Stock(_name, float(_return),
float(_risk)))
```

در ادامه نسل اول کروموزوم‌ها را به صورت تصادفی تولید می‌کنیم. به این صورت که به تعداد سهم‌ها، ژن تصادفی تولید می‌کنیم. سپس آن را نرمالایز کرده و به یک کروموزم استاندارد مسئله می‌رسیم. این کار به تعداد جمعیت نسل ادامه می‌دهیم:

```
    def generate_first_generation(self):
```

```
        num_of_genes = len(self.stocks_information)
        self.population: list[Chromososme] = [None] * self.population_size

        for i in range(self.population_size):
            genes = []
            for j in range(num_of_genes):
                genes.append(random.random())

            self.population[i] = Chromososme(genes)
            self.population[i].set_sum_to_one()

        self.generation_number = 1
```

همچنین یک متود داریم که بررسی می‌کند آیا به جواب بهینه رسیده ایم یا نه؟ (زمانی به جواب بهینه میرسیم که مقدار سودی که میخواستیم را با حداکثر ریسک مورد نظرمان به دست آورده باشیم. همین طور حداقل به تعداد لازممان سهم خریده باشیم). اگر رسیده بودیم آن را چاپ میکنیم:

```
    def is_goal(self, chromosome: Chromososme):
        total_risk = total_return = 0

        for i in range(len(self.stocks_information)):
            total_risk += chromosome.genes[i] * self.stocks_information[i]._risk
            total_return += chromosome.genes[i] * self.stocks_information[i]._return

        if total_return >= NEEDED_RETURN and total_risk <= HIGHEST_RISK and
chromosome.get_num_of_purhased_stocks() >= LEAST_STOCKS:
            print("Answer:")
            print("Return:", total_return)
            print("Risk:", total_risk)
            print("Num of Purchased Stocks:",
chromosome.get_num_of_purhased_stocks())

            return True
```

با استفاده از متود generate_next_generation، نسل بعدی را تولید میکنیم.

```
    def generate_next_generation(self):
        self.rank_by_fitness_scores()

        self._print()

        for i in range(-1, -10, -1):
            if self.is_goal(self.fitness_scores[i][0]):
                return self.fitness_scores[i][0]

        self.select_parents_for_mating_pool()
```

```
        next_population = []

        for i in range(self.population_size-1, self.population_size-
self.num_of_elites-1, -1):
            next_population.append(self.fitness_scores[i][0])

        for i in range(0, len(self.mating_pool)-1, 2):
            childs = [self.mating_pool[i], self.mating_pool[i+1]]

            if random.uniform(0, 1) < self.cross_over_rate:
                childs = uniform_cross_over([childs[0], childs[1]],
self.cross_over_probability)

            if random.uniform(0, 1) < self.mutation_rate:
                childs[0].mutate(self.mutation_probability)

            if random.uniform(0, 1) < self.mutation_rate:
                childs[1].mutate(self.mutation_probability)

            next_population.append(childs[0])
            next_population.append(childs[1])

        self.population = next_population
        self.generation_number += 1

        return None
```

همان طور که مشاهده میشود، ابتدا کروموزم ها را بر اساس رنکشان مرتب میکنیم. سپس بررسی میکنیم که آیا هیچ کدام از کروموزوم های نسل کنونی جواب بهینه هستند یا نه؟ اگر بودند نسل بعدی را نمیسازیم. (برای این کار به چک کردن ۱۰ تای اول کفایت کرده ایم). سپس اقدام به ساختن نسل بعدی میکنیم. در ابتدا تمام کروموزم های نخبه را به نسل بعدی منتقل میکنیم. همچنین mating_pool را هم درست میکنیم. سپس اعمال کراس اور و میوتیشن را روی این کروموزوم ها انجام میدهیم و کروموزم های تولید شده را به نسل بعدی مدهیم. حالا نسل جدید را جایگزین نسل قبلی میکنیم. و شماره ی نسل را افزایش میدهیم.

در متود زیر هم مسئله را حل میکنیم. به این شکل که تا به یک کروموزوم بهینه برسیم نسل بعدی را میسازیم:

```
    def solve(self):
        self.generate_first_generation()

        while True:
            answer = self.generate_next_generation()

            if answer != None:
                return answer
```

نتایج:

```
Generation Number: 86
Best Fitness Score: 73.53214624565803
Worst Fitness Score: 9.405530216163577
Average of Fitness Scores: 51.499409853508624
-----------------------------------------------------
Answer:
Return: 10.00800773198044
Risk: 0.523907878205121
Num of Purchased Stocks: 400
Coeffs: [0.00016542563854289282, 0.0002762477810344184, 0.0012047633203781285, 0.000
```

یعنی بعد از تولید ۸۶ نسل به جواب رسیدیم و ضرایب هم داده شده است. فایل اصلی تست هم به صورت ipynb. در codes/tests موجود است.

سوال‌ها:

۱. اگر جمعیت اولیه بزرگ باشد، اجرای الگوریتم زمان بسیار زیادی خواهد گرفت.
همچنین اگر کم باشد، چون تنوع در نسل کمتر است، ممکن است به جواب بهینه‌ای نرسیم.

۲. این کار تاثیر منفی روی سرعت الگوریتم دارد. چون مرحله به مرحله پردازش بیشتری لازم است که هم باعث کند شدن الگوریتم و هم باعث افزایش فضای مورد نیاز برنامه برای اجرا می‌شود.
اما تاثیر مثبتی روی دقت الگوریتم خواهد داشت. چون با اینکار باعث افزایش تنوع در نسل خواهیم شد که ما را به حواب بهینه نزدیک تر میکند. البته این تنوع زیاد همیشه هم خوب نیست. مثلا اگر این تنوع از نوع نامناسبش بیشتر باشد، تاثیر منفی ای روی رشد جمعیت و تولید نسل های بعدی خواهد داشت.

۳. کراس اور: این عمل، باعث میشود با ترکیب صفات مختلف دو کروموزوم متفاوت، کروموزوم های جدیدتری بسازیم و امید داشته باشیم که این کروموزوم های جدید از قبلی ها بهترند.
میوتیشن: با اعمال تغییر روی ژنهای یک کروموزوم، کروموزوم جدیدی میسازیم. به این ترتیب میتوانیم به کروموزوم بهتری هم برسیم و تنوع جمعیت را بالاتر ببریم و از گیر افتادن در یک اکسترمم نسبی جلوگیری کنیم.
استفاده‌ی تنها از کراس اور، ممکن است به علت کمبود تنوع، ما را در اکسترمم نسبی گیر بیندازد و استفاده‌ی تنها از میوتیشن، شانس به جواب بهینه رسیدن الگوریتم کاهش میدهد. پس بهتر است از هر دو عمل استفاده کنیم.

۴. تعریف تابع فیتنس تاثیر به سزایی در سرعت دارد. همینطور نرخ میوتیشن و کراس اور هم روی سرعت به جواب رسیدن الگوریتم موثر است.

۵. اینجا، جایی است که در یک اکسترمم نسبی (نه واقعی) گیر افتاده ایم. که دلیل اصلی آن احتمالا کمبود تنوع بوده. در این شرایط بهتر است نرخ میوتیشن را بالا ببریم. همچنین میتوانیم چند نسل مختلف را عنوان نسل آغازین در نظر گرفت.

۶. میتواین یک سقف برای تعداد نسل ها در نظر گرفت. در این صورت میتوانیم بگوییم تا این جا، بهترین جواب این بوده و به همان اکتفا کنیم.