دانشگاه اصفهان

دانشکده مهندسی کامپیوتر

عنوان:

تمرین اول هوش محاسباتی: الگوریتم‌های ژنتیک

Genetic Algorithms

نگارش

دانیال شفیعی

مهدی مهدیه

امیررضا نجفی

استاد راهنما

دکتر کارشناس

اسفند ۱۴۰۳

فهرست مطالب

۰ مقدمه

هدف از این تمرین آشنایی بیشتر با الگوریتم‌های ژنتیک و استفاده‌ی بیشتر از آن‌ها در کاربردهای عملی است.

۱ مبانی و مفاهیم الگوریتم ژنتیک

۲ درک و حل مسائل با الگوریتم ژنتیک

۱. (آ) اگر هیچ گره‌ای نباید دو بار دیده شود، یک کروموزوم باید باید یک دور بین همه‌ی گره‌ها باشد که این دور شامل طی ترتیب طی کردن ۱۰ گره یا معادل طی کردن ۱۰ یال است. پس کروموزوم ما شامل ۱۰ ژن است.

(ب) اینکه بین کدام شهرها ارتباط وجود داشته باشد پیش‌فرض‌های مسئله است اما به طور کلی می‌توان گفت اگر گراف کامل و بدون طوقه باشد، از هر گره‌ای به همه‌ی گره‌های دیگر یال وجود دارد. ما در نظر گرفتیم این یال‌ها جهت‌دار است پس اگر از یک گره به گره‌ی دیگر رفت برگشت نیازی نیست. با این اوصاف تعداد کل ژن‌های ممکن $n \times \frac{n-1}{2}$ یال می‌شود. که اینجا $n = 10$ است پس $10 \times \frac{9}{2} = 45$ ژن وجود دارد.

۲. (آ) ژن‌ها را به تابع fitness می‌بریم:

$$\text{fit}(x_1) = 6 + 5 - 4 - 1 + 3 + 5 - 3 - 2 = 9$$

$$\text{fit}(x_2) = 8 + 7 - 1 - 2 + 6 + 6 - 0 - 1 = 23$$

$$\text{fit}(x_3) = 2 + 3 - 9 - 2 + 1 + 2 - 8 - 5 = -16$$

$$\text{fit}(x_4) = 4 + 1 - 8 - 5 + 2 + 0 - 9 - 4 = -19$$

به ترتیب x_2، x_1، x_3 و x_4 برازنده هستند.

(ب) عملیات ترکیب

- ترکیب نقطه‌ای: در این روش به دو فرزند جدید می‌رسیم.

$$x_{21} = 8712|3532$$

$$x_{21} = 6541|6601$$

- ترکیب دو نقطه‌ای: با استفاده از این روش به دو فرزند جدید می‌رسیم. ما فرض می‌کنیم منظور از نقاط b و f یعنی بعد از این نقاط ترکیب اتفاق می‌افتد

$$x_{131} = 65|9212|35$$

$$x_{313} = 23|4135|85$$

- ترکیب یکنواخت: برای انجام این ترکیب نیازمند به یک ماسک هستیم. این ماسک یک ژن تصادفی با مقادیر دودویی است که نشانگر این است که آن ژن را از کروموزوم اول بگیریم یا دوم. که انتخاب اول یا دوم هم احتمال است. ما با استفاده از

برنامهٔ ۱: تولید ماسک تصادفی

```
import random
mask = ''.join(random.choice('01') for _ in range(8))
print(mask)
```

یک رشته‌ی تصادفی از ۰ و ۱ تولید می‌کنیم. ما فرض می‌کنیم ۰ معادل رشته‌ی اول و ۱ معادل رشته‌ی سوم باشد.

$$\text{mask} = 01001010$$

$$x_{13} = 8|3|12|1|6|8|1$$

$$x_{31} = 2|7|92|6|2|0|5$$

(ج) برازش فرزندان: با استفاده از تکه کد زیر برازندگی هر فرزند را محاسبه می‌کنیم:

برنامهٔ ۲: محاسبه‌ی برازندگی

```
chromosome = input()
a, b, c, d, e, f, g, h = [int(char) for char in chromosome]
fitness = a + b - c - d + e + f - g - h
print(fitness)
```

$$\text{fit}(x_{21}) = 87123532 = 15$$

$$\text{fit}(x_{21}) = 65416601 = 17$$

$$\text{fit}(x_{131}) = 65921235 = -5$$

$$\text{fit}(x_{313}) = 23413585 = -5$$

$$\text{fit}(x_{23}) = 83121681 = 6$$

$$\text{fit}(x_{32}) = 27926205 = 1$$

تعبیر بهتر شدن و بدتر شدن تعبیر نا دقیقی است. ما دو شاخص را برای بهتر شدن و بدتر شدن در نظر می‌گیریم.

۱. بالاترین برازندگی: در والدها بالاترین برازندگی ۲۳ بود که به ۱۷ کاهش یافت یعنی بدتر شده.

۲. میانگین برازندگی: در شرایط قبلی برازندگی معادل -0.75 $= \frac{-3}{4} = \frac{9+23-16-19}{4}$ می‌شود و در فرزندان $\frac{15+17-5-5+6+1}{6} = \frac{29}{6} \approx 4.83$ می‌شود که رشد قابل توجهی است.

(د) برای بیشینه کردن برازندگی، ژن‌های a، b، e و f باید مقدار ۹ داشته باشند و c، d، g و h باید مقدار ۰ را داشته باشند. برازندگی بهینه برابر $4 \times 9 - 0 = 36$ می‌شود.

(ه) ما سعی کردیم بهترین ترکیب را بسیازیم و آن $x_{\text{optimal}} = 87116601$ خواهد بود که برازندگی آن ۲۴ خواهد شد. پس نمی‌توان بدون جهش به نقطه‌ی بهینه رسید و حداقل ۱۲ تا فاصله با نقطه‌ی برازندگی وجود خواهد داشت.

۳. (آ) مقدار برازندگی به ازای هر x:

$$\text{fit}(x_1) = 1 - 4 + 7 = 4$$

$$\text{fit}(x_2) = 8 - 16 + 7 = -1$$

$$\text{fit}(x_3) = 27 - 36 + 7 = -2$$

$$\text{fit}(x_4) = 64 - 64 + 7 = 7$$

(ب) بله. می‌توانیم با اضافه کردن $\forall c : c \geq 2$ همه‌ی مقدارها را نامنفی کنیم. مثلا اگر $c = 3$ در نظر بگیریم رابطه‌ی برازندگی $\text{fit}(x) = x^3 - 4x^2 + 10$ خواهد شد.

(ج) به هر برازندگی مقدار ثابت ۲ اضافه می‌شود پس

$$\text{TotalFitness} = (4+3) \times 2 + (-1+3) \times 3 + (-2+3) \times 3 + (7+3) \times 2$$

$$= 14 + 6 + 3 + 20 = 43$$

(د) مقدار برازندگی نسبی برای هر نمونه‌ی x به صورت زیر خواهد شد:

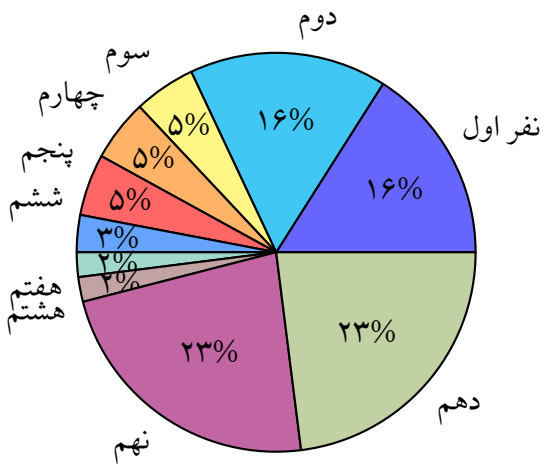
$$P(x=1) = \frac{7}{43} = 0.1628$$

$$P(x=2) = \frac{2}{43} = 0.0465$$

$$P(x=3) = \frac{1}{43} = 0.0233$$

$$P(x=4) = \frac{10}{43} = 0.2326$$

می‌توانیم آن را به صورت یک گردونه هم نشان دهیم



شکل ۱: گردونه‌ی شانس برای این نمونه از جمعیت

(ه) مزیت تابع جدید این است که به ازای هر مقدار x، تابع برازندگی همواره نامنفی است. برای محاسبه‌ی $g(x)$ تمام مقدادیر بدست آمده در بخش آ را به توان ۲ می‌رسانیم.

$$\text{fit}(x_1) = 4^2 = 16$$

$$\text{fit}(x_2) = (-1)^2 = 1$$

$$\text{fit}(x_3) = (-2)^2 = 4$$

$$\text{fit}(x_4) = 7^2 = 49$$

(و)
- **فشار انتخاب**: فشار انتخاب یعنی درجه‌ی اینکه افراد برازنده‌تر چقدر شانس زنده ماندن دارند. برعکس اضافه کردن مقدار ثابت، در «به توان رساندن» فشار انتخاب زیاد می‌شود. البته این تا حدودی بستگی به روش انتخاب هم دارد. مثلا اگر از الگوریتم انتخاب رتبه پایه [۱] استفاده کنیم دیگر این مسئله جدی نیست.
- **همگرایی**: می‌توان گفت با افزایش فشار انتخاب، همگرایی سریع‌تر می‌شود اما خطر گیر کردن در یک نقطه‌ی بهینه‌ی محلی وجود دارد.
- **تنوع:** افزایش فشار انتخاب باعث کاهش تنوع و همگرایی سریع به یک قله‌ی محلی خاص می‌شود که ممکن است بهترین نباشد. با افزایش فشار انتخاب تنوع در انتخاب گونه‌ها را از دست می‌دهیم.

[۱] rank-based selection

۳ پیاده‌سازی، ارزیابی و تجزیه و تحلیل الگوریتم ژنتیک جهت انتخاب بهترین ویژگی برای مسئله‌ی واقعی دسته‌بندی مشتریان

۱. پیش پردازش داده‌ها:

(آ) حذف دادههای پرت:

برای پر کردن دادههای پرت از روش IQR method استفاده میکنیم این روش به این صورت است که IQR را برابر با Q3 - Q1 قرار میدهیم(چارک اول : Q1، چارک سوم: Q3) سپس دادههای کوچکتر از Q1 - 1.5*IQR و بزرگتر از Q3 + 1.5IQR را حذف میکنیم.

برنامهٔ ۳: حذف دادههای پرت

```
df = pd.read_csv('Customer Classification dataset/Train.csv')

numeric_cols = df.select_dtypes(include=[np.number]).columns.tolist()

numeric_cols.remove('ID')

for col in numeric_cols:
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
    IQR = Q3 - Q1
    lower_bound = Q1 - 1.5 * IQR
    upper_bound = Q3 + 1.5 * IQR
    df = df[(df[col] >= lower_bound) & (df[col] <= upper_bound)]
```

(ب) رمزگذاری ویژگیهای دسته ای (categorical):

مهم ترین پارامتر در رمزگذاری (encoding) داده های دسته ای این است که ببینیم که این داده ها داده های کیفی ترتیبی (Ordinal Qualitative) هستند یا کیفی اسمی (Nominal Qualitative) اگر داده ها کیفی ترتیبی باشند نیاز داریم که داده ها را به گونه ای پیش پردازش بکنیم که این ترتیب همچنان حفظ شود و در صورتی که داده ها اسمی باشند نیازی به این کار نداریم و میتوانیم برای هر دسته یک ستون درست کنیم که با مقادیر درست (True) و غلط (False) مشخص کنیم که به این دسته قرار دارد یا خیر.

در داده هایی که در سوال به ما داده شده بود ستون های Gender، Ever_Married، Graduated، Profession و Var_1 ستونهایی بودند که داده های انها به صورت کیفی اسمی بود و برای همین برای رمز گذاری انها از get_dummies استفاده کردیم.

برنامهٔ ۴: رمزگذاری با get_dummies

```
columns_to_encode = ['Gender', 'Ever_Married', 'Graduated', 'Profession',
    'Var_1']

features = pd.get_dummies(
    features,
    columns=columns_to_encode,
```

```
    prefix=columns_to_encode,
    drop_first=True
)
features.head()
```

و ستون Spending_Score را که دارای داده های ترتیبی به نام های Low، Average و High بود را به صورت دیگری رمزگذاری کردیم تا مدل توانایی درک این که این ۳ مقدار دارای ترتیب مشخصی هستند را متوجه شود. اینکار را به این صورت انجام دادیم که این ۳ ستون را به ترتیب از کوچک به بزرگ از ۰ تا ۲ مقداردهی کردیم.

برنامهٔ ۵: رمزگذاری برای ستون spending_score

```
score_mapping = {
    'Low': 0,
    'Average': 1,
    'High': 2
}
features['Spending_Score'] = features['Spending_Score'].map(score_mapping)
features.head()
```

(ج) پر کردن داده های خالی (Nan) :

برای پرکردن داده های خالی از KNNImputer در کتابخانه sikitlearn استفاده کردیم به این صورت که n_neighbors که یکی از پارامترهای این تابع است را ۸ انتخاب کردیم و این به این معناست که هر ردیفی که مقدار خالی داشته باشد میاید و ۸ ردیف نزدیک به ان را پیدا میکند و سپس از مقادیر ستون مورد نظر(یعنی ستونی که مقدار خالی در ان قرار دارد.) در ان ردیفها میانگین گرفته و انرا به عنوان مقدار جدید سلول خالی قرار میدهد.

برنامهٔ ۶: پر کردن سلول‌های خالی

```
from sklearn.impute import KNNImputer
imputer = KNNImputer(n_neighbors=8)
imputed_X = imputer.fit_transform(features)
features = pd.DataFrame(imputed_X, columns=features.columns)
```

۱. پیاده سازی الگوریتم ژنتیک:

الگوریتم ژنتیک یک روش جستجو و بهینه‌سازی مبتنی بر تکامل طبیعی است که در آن مفاهیمی از ژنتیک مانند انتخاب طبیعی، ترکیب و جهش برای یافتن بهترین جواب به کار گرفته می‌شوند. این الگوریتم در مسائل مختلف از جمله بهینه‌سازی، یادگیری ماشین و مسائل ترکیبیاتی مورد استفاده قرار می‌گیرد.

مراحل الگوریتم ژنتیک:

(آ) مقداردهی اولیه (Initialization) :

مجموعه‌ای از کروموزوم‌ها (یا راه‌حل‌های ممکن) به‌صورت تصادفی ایجاد می‌شود.

برنامهٔ ۷: مقداردهی اولیه

```
def initialize(count, column_count):
    zero_array = np.zeros((1, column_count), dtype=int)
    my_set = set()

    while len(my_set) != count:
        random_numbers = random.sample(range(0, 21), column_count)
        copied_zero_array = zero_array.copy()
        for i in range(column_count):
            copied_zero_array[0][i] = random_numbers[i]
        my_set.add(tuple(copied_zero_array[0]))
    my_list = list(my_set)
    return my_set
```

(ب) ارزیابی برازندگی (Fitness Evaluation):

هر کروموزوم بر اساس یک تابع برازندگی ارزیابی می‌شود تا میزان تطابق آن با هدف مشخص شود. در کدی که ما زدیم تابع برازندگی، accuracy_score مدل است.

(ج) انتخاب (Selection):

کروموزوم‌های بهتر شانس بیشتری برای انتخاب شدن و انتقال به نسل بعدی دارند. روش‌های مختلفی برای این کار وجود دارد، از جمله:

چرخ رولت (Roulette Wheel Selection)

انتخاب بر اساس رتبه (Rank-Based Selection)

انتخاب تورنمنت (Tournament Selection)

که ما هر ۳تای انها را پیاده سازی کردیم.

برنامهٔ ۸: roulette-wheel-selection

```
def roulette_wheel_selection(initialize_gen, fitness_list, repeatable,
    count):
    indices = list(range(len(fitness_list)))
    if repeatable:
        #repeatable
        selected_index = random.choices(indices, weights=fitness_list,
    k=count)
    else:
        #non-repeatable
        selected_index = np.random.choice(indices, size=count,
    replace=False, p=fitness_list/sum(fitness_list))
    selected_ones = list()
    initialize_gen = list(initialize_gen)
    for i in selected_index:
        selected_ones.append(initialize_gen[i])
    return selected_ones
```

برنامهٔ ۹: rank-based-selection

```
def rank_based_selection(initialize_gen, fitness_list, repeatable, count):
    fitness_list = get_rank_array(list(fitness_list))

    indices = list(range(len(fitness_list)))
    if repeatable:
        #repeatable
        selected_index = random.choices(indices, weights=fitness_list, k=count)
    else:
        #non-repeatable
        selected_index = np.random.choice(indices, size=count, replace=False, p=fitness_list/sum(fitness_list))
    selected_ones = list()
    initialize_gen = list(initialize_gen)
    for i in selected_index:
        selected_ones.append(initialize_gen[i])
    return selected_ones
```

برنامهٔ ۱۰: tournament-selection

```
def tournament_selection(initial_gen, gen_fitnesss, replacement, count, sample=2):
    selected_population = []

    if replacement:
        for _ in range(count):
            tournament_indices = random.choices(range(len(initial_gen)), k=sample)
            winner_index = max(tournament_indices, key=lambda i: gen_fitnesss[i])
            selected_population.append(initial_gen[winner_index])
    else:
        not_used = list(range(len(initial_gen)))
        thrown = []

        while len(not_used) >= sample and len(selected_population) < count:
            tournament_indices = random.sample(not_used, sample)
            winner_index = max(tournament_indices, key=lambda i: gen_fitnesss[i])
            selected_population.append(initial_gen[winner_index])
            for idx in tournament_indices:
                not_used.remove(idx)
                if idx != winner_index:
                    thrown.append(idx)
```

```
        pool = thrown + not_used
        while len(selected_population) < count and pool:
            if len(pool) >= sample:
                tournament_indices = random.sample(pool, sample)
            else:
                tournament_indices = random.choices(pool, k=sample)
            winner_index = max(tournament_indices, key=lambda i:
    gen_fitnesss[i])
            selected_population.append(initial_gen[winner_index])

    return selected_population
```

(د) ترکیب (Crossover):

کروموزوم‌های انتخاب شده با یکدیگر ترکیب می‌شوند تا فرزندان جدید تولید شود. روش‌های متداول شامل:

ترکیب تک‌نقطه‌ای (Single-Point Crossover)

ترکیب دونقطه‌ای (Two-Point Crossover)

ترکیب یکنواخت (Uniform Crossover)

که ما هر ۳تای اینها را هم پیاده سازی کردیم.

برنامهٔ ۱۱: single-point-crossover

```
def single_point_crossover(parents):
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 < len(parents):
            p1, p2 = parents[i], parents[i + 1]
            cross_point = random.randint(1, len(p1) - 1)
            child1 = p1[:cross_point] + p2[cross_point:]
            child2 = p2[:cross_point] + p1[cross_point:]
            offspring.extend([child1, child2])
    return offspring
```

برنامهٔ ۱۲: two-point-crossover

```
def two_point_crossover(parents):
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 < len(parents):
            p1, p2 = parents[i], parents[i + 1]
            point1 = random.randint(1, len(p1) - 2)
            point2 = random.randint(point1 + 1, len(p1) - 1)
            child1 = p1[:point1] + p2[point1:point2] + p1[point2:]
            child2 = p2[:point1] + p1[point1:point2] + p2[point2:]
            offspring.extend([child1, child2])
```

```
    return offspring
```

برنامهٔ ۱۳: uniform-crossover

```
def uniform_crossover(parents, swap_prob=0.5):
    offspring = []
    for i in range(0, len(parents), 2):
        if i + 1 < len(parents):
            p1, p2 = parents[i], parents[i + 1]
            child1, child2 = [], []
            for gene1, gene2 in zip(p1, p2):
                if random.random() < swap_prob:
                    child1.append(gene2)
                    child2.append(gene1)
                else:
                    child1.append(gene1)
                    child2.append(gene2)
            offspring.extend([''.join(child1), ''.join(child2)])
    return offspring
```

(ه) جهش (Mutation):

برخی از کروموزوم‌ها دچار تغییرات جزئی می‌شوند تا از همگرایی زودرس جلوگیری شود و تنوع حفظ گردد.

برنامهٔ ۱۴: mutate

```
def mutate(individual, max_value):
    index = random.randint(0, len(individual) - 1)
    new_value = random.randint(1, max_value)
    while new_value in individual:
        new_value = random.randint(1, max_value)
    individual[index] = new_value
    return individual
```

(و) تکرار (Iteration):

همه‌ی مراحل الف تا ه را دوباره انجام میدهیم تا به یکی از دو شرط زیر برسیم:

رسیدن به حداکثر تعداد تکرار (iterations)

رسیدن به نقطه‌ی همگرایی(عدم بهبود برای مدت طولانی)