درس: مبانی هوش محاسباتی

موضوع: گزارش پروژه نهایی(فریم ورک الکوریتم ژنتیک)

استاد: مهندس تورانی

پگاه نخ ساز و آرش آقایی

۹۴۰۱۲۲۶۹۰۱۵ - ۹۴۰۱۲۲۶۱۰۱۱

به نام خدا

موضوع پروژه ما طراحی فریم ورک خلاصه ای برای پیاده سازی الگوریتم های ژنتیک در زبان پایتون می باشد که با استفاده از توابع موجود در این پروژه بتوان به پیاده سازی بهتر و راحت تر الگوریتم های ژنتیک در پایتون پرداخت.

نمونه هایی از این مدل فریم ورک وجود دارد که می توان به پر کاربرد ترین آن یعنی DEAP در پایتون اشاره کرد.

از آن جایی که هدف از پروژه درس هوش محاسباتی، یادگیری هر چند بهتر مباحث این درس و پیاده سازی آموخته ها بود؛ ما تصمیم گرفتیم که موضوع پروژه پیاده سازی الگوریتم های ژنتیک باشد که به جرأت می توان گفت یکی از اساسی ترین و مفهومی ترین بخش های این درس است.

در ادامه با توضیحات پروژه همراه شما خواهیم بود:

```
from __future__ import division
import math
import random
from itertools import repeat
import numpy as np
from functools import partial
from operator import attrgetter
from functools import partial
```

در بخش import های پروژه اکثرا از library های پایتون استفاده شده است مانند :division وmath و random و repeat ؛ تنها library که خارج از پایتون باید import شود numpy است که می دانیم برای کار با اعداد و کار های علمی است.

```
try:
    from collections.abc import Sequence
except ImportError:
    from collections import Sequence
```

در این قسمت از کد مخصوص import ازtry استفاده کرده ایم؛ علت استفاده از این مفهوم تست این منظور است که آیا زیر پکیج abc در دسترس می باشد؟ در صورت عدم دسترسی از پکیج collection که والد پکیج collection.abc است استفاده کند.

از sequence نیز برای داشتن توالی و دنباله در collection ها یا مجموعه ها استفاده می کنیم.

در کل در الگوریتم ژنتیک نیاز به مراحل زیر برای پیاده سازی داریم:

- initialization
- selection
- crossover
- mutation

در این پروژه سعی شده است که برای هریک از مراحل بالا چند مدل مختلف تعریف و ارایه کنیم.

اولین مرحله initialization یا همان مقداردهی اولیه است که در این مرحله ما دو مدل repeat initialization و iteration initialization را به عنوان تابع پیاده سازی کردیم .

```
def repeatInitialization(container, func, n):
    arr = []
    for i in range(0, n):
        arr.append(func())
    return container(arr)
```

در قسمت اول یکسری function به آن داده میشود و از آن n بلوک میسازیم و از بلوک ها یک لیست ایجاد می کنیم سپس آن را در container ریخته و به خروجی انتقال میدهیم.

```
def iterationInitialization(container, generator):
    return container(generator())
```

در مدل دوم مثل مدل قبل عمل میکنیم با این تفاوت که فقط یک بار ورودی ها را میگیریم و در container میریزیم و به خروجی داده و return می کنیم.

برای بخش بعدی یعنیselection تابع مربوط به ۳ مدل از مدل های انتخاب را تعریف کردیم؛ این مدل ها عبارتند از:

- random selection
- tournament selection
- roulette selection

```
def randomSelection(individuals, k):
    arr = []
    for i in range(0, k):
        arr.append(random.choice(individuals))
    return arr
```

در این مدل به صورت رندم و تصادفی اشخاص را از جمعیت انتخاب کرده و آن ها را return کرده و به خروجی می دهیم.

```
def tournamentSelection(individuals, k, tournsize, fit_attr="fitness"):

    return [max(randomSelection(individuals, tournsize), key=attrgetter(fit_attr)) for i in xrange(k
)]
```

در قسمت بعدی انتخاب مدل مسابقه ای را تعریف می کنیم که میزان برازش از اهمیت برخوردار است.

در این مدل نیز مانند مدل اول از رندم استفاده میکنیم با این تفاوت که در این مدل بهترین جمعیت را از نظر fitness انتخاب کرده برخی از آن ها را جنریت کرده و برای خروجی در تابع return می کنیم.

```
def rouletteSelection(individuals, k, fit_attr="fitness"):
    s_inds = individuals.sort(key=attrgetter(fit_attr), reverse=True)
    arr = []
    for ind in individuals:
        arr.append(getattr(ind, fit_attr).values[0])
    fitnessSummation = sum(arr)
    selections = []
    for i in range(0, k):
        summation = 0
        for ind in s_inds:
            summation += getattr(ind, fit_attr).values[0]
            if summation > random.random() * fitnessSummation:
                selections.append(ind)
                break
    return selections
```

مدل سوم نیز با استفاده از چرخ رووولت نوشته و define شده است؛ در این مدل انتخاب بر اساس احتمال است. جمعیت را بر اساس برازش هایشان sort کرده و سپس به خروجی ارایه می دهیم.

قسمت بعدی پروژه مربوط به crossover است.

در این قسمت از پروژه از سه مدل زیر استفاده کرده ایم:

- One point crossover
- Two point crossover
- Uniform crossover

```
def onePointCrossOver(ind1, ind2):
    crossBreakPoint = random.randint(1, min(len(ind1), len(ind2)) - 1)
    ind1[crossBreakPoint:] = ind2[crossBreakPoint:]
    ind2[crossBreakPoint:] = ind1[crossBreakPoint:]
    return ind1, ind2
```

اولین تابع تعریف شده مدل onepoint است که الگوریتم آن مانند مثال اسلاید به دست آمده است، به این صورت که دو کروموزوم را به عنوان والدین در نظر گرفته و یک نقطه را در آن ها انتخاب می کنیم؛ سپس بین آن دو کروموزوم کراس اوور را انجام میدهیم به این صورت که قسمت اول کروموزوم دوم و قسمت دوم کروموزوم اول با هم ترکیب شده و کروموزوم جدیدی را می سازد. برای کروموزوم دوم نیز بر عکس عمل فوق اتفاق می افتد.

```
def twoPointCrossOver(ind1, ind2):
    crossBreakPoint1 = random.randint(1, min(len(ind1), len(ind2)))
    crossBreakPoint2 = random.randint(1, min(len(ind1), len(ind2)) - 1)
    crossBreakPoint2 += 1 if crossBreakPoint2 >= crossBreakPoint1 else 0
    if crossBreakPoint2 < crossBreakPoint1:
        crossBreakPoint1, crossBreakPoint2 = crossBreakPoint2, crossBreakPoint1
    else:
        crossBreakPoint1, crossBreakPoint2 = crossBreakPoint1, crossBreakPoint2
    ind1[crossBreakPoint1:crossBreakPoint2], ind2[crossBreakPoint1:crossBreakPoint2] = \
    ind2[crossBreakPoint1:crossBreakPoint2], ind1[crossBreakPoint1:crossBreakPoint2]
    return ind1, ind2
def twoPointsCrossOver(ind1, ind2):
    return twoPointCrossOver(ind1, ind2)
```

برای مدل twopoint نیز مانند مدل onepoint عمل میکنیم با این تفاوت که در این مدل با دو نقطه روی کروموزوم های والد، آن ها را به سه قسمت تقسیم می کنیم و بعد عمل crossover را انجام می دهیم.

```
def uniformCrossOver(ind1, ind2, indpb):
    for i in xrange(min(len(ind1), len(ind2))):
        if random.random() < indpb:
            ind1[i], ind2[i] = ind2[i], ind1[i]
    return ind1, ind2
```

برای مدل سوم که همان uniform crossover است از یک آستانه استفاده می کنیم که در این کد آستانه را indpb نام گذاری کردیم. با استفاده از رندم و مقایسه دو کرومزم و کرومزوم های ساخته شده به پیاده سازی crossover می پردازیم.

بالاخره از یکی از تابع های بالا استفاده کرده و crossover را انجام خواهیم داد مرحله ی بعدی مربوط به مرحله ی جهش می باشد.

در mutation نیز از gaussian و polynomial و flip bit و shuffle index استفاده شده است که در مدل های guassian و polynomial جهش با استفاده از برخی متغیر های پیچیده و فرمول های ریاضی به صورت هوشمندانه تری انجام شده و در مدل های shuffle index و flip bit جهش را به صورت ساده و همانند مفهوم آن در درس پیاده سازی می کنیم.

```
def gaussianMutation(individual, mu, sigma, indpb):
    size = len(individual)
    if not isinstance(mu, Sequence):
        mu = repeat(mu, size)
    if not isinstance(sigma, Sequence):
        sigma = repeat(sigma, size)
    for i, m, s in zip(range(0, size), mu, sigma):
        if random.random() < indpb:
            individual[i] += random.gauss(m, s)
    return individual,
```

همان طور که در بالا گفته شد یکی از تابع ها gaussian می باشد، از این تابع برای جهش می توان استفاده کرد.

همان طور که در کد بالا مشخص است الگوریتم مربوط به آن توسط فرمول های gaussian نوشته می شود و در آن متغیر هایی مثل mu و sigma تعریف کرده و مقدار آن ها را محاسبه می کنیم؛ سپس مقدار رندم انتخاب شده را در نظر گرفته اگر از میزان آستانه کمتر باشد عملیات انجام شده و مقدار individual را return می کنیم.

```
def polynomialBoundedMutation(individual, eta, low, up, indpb):
    size = len(individual)
    if not isinstance(low, Sequence):
        low = repeat(low, size)
    if not isinstance(up, Sequence):
        up = repeat(up, size)
    for i, xl, xu in zip(xrange(size), low, up):
        if random.random() <= indpb:
            x = individual[i]
```

```
            delta_1 = (x - xl) / (xu - xl)
            delta_2 = (xu - x) / (xu - xl)
            rand = random.random()
            mut_pow = 1.0 / (eta + 1.)
            if rand < 0.5:
                xy = 1.0 - delta_1
                val = 2.0 * rand + (1.0 - 2.0 * rand) * xy ** (eta + 1)
                if val < 0:
                    delta_q = -((-val) ** (mut_pow)) - 1.0
                else:
                    delta_q = val ** (mut_pow) - 1.0
            else:
                xy = 1.0 - delta_2
                val = 2.0 * (1.0 - rand) + 2.0 * (rand - 0.5) * xy ** (eta + 1)
                if val < 0:
                    delta_q = 1.0 - (-((-val) ** (mut_pow)))
                else:
                    delta_q = 1.0 - val ** mut_pow
            x = x + delta_q * (xu - xl)
            x = min(max(x, xl), xu)
            individual[i] = x
    return individual,
```

در قطعه کد بعدی که مربوط به polynomial است نیز مثل gaussian با استفاده از فرمول های مربوط به آن الگوریتم جهش را تعریف کرده و مثل قبل مقدار آستانه را با مقدار رندم مقایسه کرده اگر مقدارش بیشتر از مقدار رندم باشد فرمول را اعمال کرده و مقدار individual را بر می گرداند.

```
def shuffleIndexesMutation(individual, indpb):
    for i in range(0, len(individual)):
        if random.random() < indpb:
            swap_indx = random.randint(0, len(individual) - 2)
            if swap_indx >= i:
                swap_indx += 1
            individual[i], individual[swap_indx] = \
                individual[swap_indx], individual[i]
    return individual,
```

در مدل shuffle index mutation نیز اگر مقدار رندم انتخاب شده از مقدار آستانه کمتر باشد swap را انجام داده و کروموزوم با استفاده از swapping جهش میکند.

```
def flipBitMutation(individual, indpb):
    for i in range(0, len(individual)):
        if random.random() < indpb:
            individual[i] = type(individual[i])(not individual[i])

    return individual,
```

مدل دیگر که ساده ترین مدل ما است مدل flip bit میباشد که مخصوص کروموزوم های باینری یا دودویی است که در این مدل نیز مقدار آستانه با مقدار رندم مقایسه شده سپس جهش به صورتی اتفاق می افتد که بیت ۰ تبدیل به ۱ یا بر عکس می شود.

```
l = 10
gen_idx = partial(random.sample, range(l), l)
pop = repeatInitialization(list, gen_idx, 10)

pop = randomSelection(pop, 2)
ind = twoPointsCrossOver(pop[0], pop[1])
ind = polynomialBoundedMutation(ind[0], low=0.0, up=1.0, eta=20.0, indpb=1.0/5)

print(ind)
```

در انتهای کد نیز مثالی از ادعای خود بنا بر اینکه با استفاده از پلتفرم ساخته شده تنها با چند خط کد می توان یک الگوریتم ژنتیک کامل برای تولید ژن جدید ساخت ارایه کرده ایم؛ در این مثال کرومزوم ها به صورت رندم ساخته می شوند. همان طور که از قطعه کد بالا معلوم است repeat initialization برای قسمت مقداردهی اولیه استفاده شده است همچنین از مدل انتخابی random استفاده کرده ایم و برای مدل crossover هم از two point و برای جهش هم از polynomial استفاده شده است.

مثال:

[[۸, ۷, ۹, ۶, ۳, ۴, ۵, ۰, ۱, ۲][۶, ۴, ۳, ۵, ۹, ۸, ۲, ۱, ۰, ۷]]

و از هر مدل یک نوع را می توان انتخاب کرده و پس از انجام مراحل خروجی زیر را می توان دریافت نمود.

([۶, ۴, ۴, ۶, ۳, ۰, ۵, ۱, ۳, ۷)]

این frame work توانایی گسترش بالایی دارد و در صورت ادامه دادن کار بر روی آن می توانیم بسیار کامل تر و با مدل های بیشتر برای پیاده سازی طیف های گسترده تر این الگوریتم بپردازیم.

در انتها جا دارد از استاد گرانقدر جناب تورانی بابت تدریس و اخلاق خوبشان که در درک مفاهیم هوش محاسباتی به ما کمک شایانی کردند تشکر به عمل آوریم.

آرش آقایی و پگاه نخ ساز