

بسمه تعالی

گزارش فاز دوم پروژه ب

درس الگوریتم های بیوانفورماتیک

تهیه و تنظیم : الهه بدلی (۹۸۲۰۹۰۷۲)

گزارش پیش رو شامل نحوه ی پیاده سازی و نحوه استفاده از کد می باشد.

فاز دوم پروژه شامل سه بخش است که این سه بخش طی توابع زیر انجام شده است. کد در محیط colab نوشته شده است.

کد از تابع های زیر تشکیل شده است.

phase2_main
read_files
split_genome
create_blocks
initialize_bitarray
Count_Kmers

Filter
Kmer_Count_and_Filtering

که به ترتیب به توضیح هر یک میپردازیم.

تابع Phase2_main:

تابع phase2_main تابع اصلی است که تمامی توابع دیگر را فراخوانی می کند.

```
1 def phase2_main():
2     print("> Read Genomes ...")
3     genomes = read_files(folder_path = 'data mitochondrial genome')
4     print("Genomes Readed!")
5     print("> Split Genomes to Blocks ...")
6     create_blocks(genomes)
7     print("Blocks Created!")
8     All_blocks = read_files(folder_path = 'blocks')
9     print("> Start Count and Filter kmers ...")
10    All_Counts ,All_Presence_Absence = Kmer_Count_and_Filter(All_blocks)
11    print("Kmer Count and Filtering: Done!")
12    return All_Counts ,All_Presence_Absence
```

```
1 All_Counts ,All_Presence_Absence = phase2_main()
```

```
> Read Genomes ...
Genomes Readed!
> Split Genomes to Blocks ...
Blocks Created!
> Start Count and Filter kmers ...
Kmer Count and Filtering: Done!
```

مراحل اصلی انجام کار (هر کدام با جزئیات در بخش تابع مربوطه توضیح داده خواهد شد).

۱. خواندن فایل ژنوم ها

۲. شکستن هر ژنوم به پلاک های مربوطه (مطابق با میزان Overlap)

۳. شمردن kmer ها و فیلتر کردن بر حسب بودن یا نبودن kmer (با توجه به شماره دانشجویی، من که پایان

۷۲ دارد گزینه 0 به عنوان بخش سوم انجام شده است.)

خروجی، این تابع دو دیکشنری است: (برای مثال: خروجی های ژنوم (k3_AF010406.1)

All_Count: که به ازای هر ژنوم و به ازای هر بلاکش تعداد kmer در مقابل هر kmer ش نوشته شده است.

```
1 All Counts['k3 AF010406.1']
```

```
[{'AAA': 54,
  'AAC': 26,
  'AAG': 30,
  'AAT': 29,
  'ACA': 29,
  'ACC': 22,
  'ACG': 12,
  'ACT': 18,
  'AGA': 15,
  'AGC': 26,
  'AGG': 17,
  'AGT': 17,
```

و All_Presence_Absence: که شامل اطلاعات بودن یا نبودن kmer به ازای هر بلاک هر ژنوم است.

```
1 All Presence Absence['k3_AF010406.1']
```

```
[bitarray('1111111111111111111111111111111111111111111111111111111'),  
bitarray('1111111111111111111111111111111111111111111111111111111'),  
bitarray('1111111111111111111111111111111111111111111111111111111'),  
bitarray('1111111111111111111111111111111111111111111111111111111'),  
bitarray('111111111111111111111111111111111111111111111111011101110'),  
bitarray('1111111111111111111111111111111111111111111111111111111'),  
bitarray('11111111111111111111111111111111111111111111111101111111'),  
bitarray('1111111111111111111111111111111111111111111111111111111'),  
bitarray('1111111111111111111111111111111111111111111111111111111'),  
bitarray('1111111111111111111111111111111111111111111111111111111'),  
bitarray('1111111111111111111111111111111111111111111111111111111'),  
bitarray('111111111111111111111111111111111111111111111111011111111'),  
bitarray('1111111111111111111111111111111111111111111111111111110')]
```

```

1 def read_files(folder_path):
2     ''' reading files in given path
3
4     Parameters
5     -----
6     - folder_path: path of files (type: str)
7
8     Return
9     -----
10    - output: a dictionary that its keys are records id(genome name) and
11              values are a string(genome or block) (type:dict)
12    '''
13
14    output = {}
15    fasta_files = glob.glob(os.path.join(folder_path, '*.fasta'))
16    for fasta_file in fasta_files:
17        for seq_record in SeqIO.parse(fasta_file, "fasta"):
18            output.setdefault(seq_record.id, []).append(str(seq_record.seq))
19
20    return output

```

این تابع یک مسیر را دریافت می کند و فایل های فاستای موجود در آن را می خواند و در دیکشنری ذخیره میکند. `id` هر فاستا فایل که نام ژنوم هست به عنوان `key` در دیکشنری و محتوای خود ژنوم به عنوان `value` در نظر گرفته می شود. البته در مراحل بعد از این تابع برای خواندن بلاک ها هم استفاده می شود. بنابراین در آن قسمت نام هر ژنوم به عنوان `key` و بلاک هایش به عنوان `value` در یک لیست ذخیره می شود. در نهایت خروجی را بر میگرداند.

```

1 def split_genome(genome, nt_num, overlap = 0):
2     """ Split genome to blocks (phase 2.1)
3
4     Parameters
5     -----
6     - genome: a genome (type: str)
7     - nt_num: number of nucleotide in a block (type: int)
8     - overlap: blocks overlap amount (type: int)
9
10    Return
11    -----
12    - blocks (type: list)
13    """
14
15    blocks = []
16    start_index = 0
17    while start_index < len(genome):
18        block = genome[start_index : start_index + nt_num]
19        start_index = start_index + nt_num - overlap
20        blocks.append(block)
21    return blocks

```

این تابع بخش اول فاز دوم را پیاده سازی می کند. یک ژنوم و تعداد نوکلئوتاید هایش (که 1000 است) و میزان Overlap که بطور پیش فرض 0 است را دریافت میکند. سپس ژنوم را به بلاک هایش می شکند و لیست بلاک ها را به عنوان خروجی می دهد.

همان طور که در حلقه while مشخص است این تابع از نقطه شروع تا 1000 نوکلئوتاید بعد را به عنوان یک بلاک در نظر میگیرد. حالا با توجه به میزان هم پوشانی نقطه ی شروع بلاک بعدی را محاسبه میکند. مثالش را در ادامه آورده ام.

```
def create_blocks(genomes):
    ''' create all blocks of all given genomes
    Parameters
    -----
    - genomes: all genomes (type: dict)

    Return
    -----
    write created blocks with specifict overlap(k) in "blocks" folder
    file name format: k + number of overlap + _ + genome name + .fasta (example: k0_AF010406.1.fasta)
    '''
    for genome in genomes:
        for k in [0,3,5,7,9,11,24,32]:
            f_out = open('blocks/k'+str(k)+"_"+str(str(genome))+".fasta", 'w')

            blocks = split_genome(genomes[genome][0] , nt_num = 1000 , overlap = k)

            f_out.write("; Blocks Count: " + str(len(blocks)) + "\n\n")

            block_num = 1
            for block in blocks:
                f_out.write(">k" + str(k)+ "_" + str(genome)+ "\n" + block + "\n")
                block_num += 1
            f_out.close()
```

این تابع کل ژنوم های خوانده شده در مرحله read_files را دریافت میکند و تک به تک (به ازای هم پوشانی های خواسته شده) وارد تابع split_genome می کند بلاک هایشان را دریافت و در فایل مربوطه ذخیره میکند.

نحوه ی ذخیره به این صورت است که ما در کل ۴۱ فایل ژنوم داریم و باید به ازای ۸ میزان همپوشانی مختلف (0,3,5,7,9,11,24,32) بلاک هایشان را استخراج کنیم بنابراین در فایل خروجی باید $41 * 8$ فایل داشته باشیم. نحوه نام گذاری فایل بلاک ها با توجه به میزان همپوشانی استفاده شده برای درآوردن آن بلاک ها است.

برای مثال ژنوم AF010406.1 خوانده شده است. ۸ فایل خروجی مربوط به k های مختلف ب شرح زیر است:

k0_AF010406.1.fasta

k3_AF010406.1.fasta

k5_AF010406.1.fasta

k7_AF010406.1.fasta

k9_AF010406.1.fasta

k11_AF010406.1.fasta

k24_AF010406.1.fasta

k32_AF010406.1.fasta

ابتدای نام فایل میزان همپوشانی و سپس نام ژنوم مورد نظر آورده شده است. محتوای هر فایل هم که بلاک های آن ژنوم با میزان همپوشانی مورد نظر است: برای مثال در فایل k9_AF010406.1.fasta که بلاک های ژنوم AF010406.1 با میزان همپوشانی 9 است داریم: (بخشی از تصویر را آورده ام).

(به دلیل مراحل بعد نام ژنوم را به عنوان id هر بلاک آورده ام).

```
; Blocks Count:17
```

```
>k9_AF010406.1
GTTAATGTAGCTTAAACTTAAAGCAAGGCACTGAAAATGCCTAGATGAGT
>k9_AF010406.1
AAGGTAAGCATACTGGAAAGTGTGCTTGGATAAACCAAGATATAGCTTAA
>k9_AF010406.1
GCCCAGTGAATAACGTTAAACGGCCGCGGTATTCTGACCGTGCAAAGGT.
>k9_AF010406.1
TAGCACTAACCTAGCCTTAACTATATGAATCCCCCTACCCATACCCTAT
>k9_AF010406.1
TCGTTATGATTAGCACCCACTGATTGCTCATCTGAATTGGATTTGAAATA
```

که نقاط شروع این ۱۷ بلاک به شرح زیر است:

بلاک اول از 0 تا ۹۹۹ را شامل میشود. با توجه به overlap ۹ تایی بلاک دوم باید از index ۹۹۱ شروع شود تا ۱۰۰۰ تا بعد که ۱۹۹۱ است. حالا نقطه ی شروع بلاک سوم باید از ۹-۱۹۹۱ که ۱۹۸۲ است شروع شود. با توجه به تصویر زیر می بینیم این روند در برنامه صحیح اجرا می شود.

```
length of this genome: 16616
index: 991
index: 1982
index: 2973
index: 3964
index: 4955
index: 5946
index: 6937
index: 7928
index: 8919
index: 9910
index: 10901
index: 11892
index: 12883
index: 13874
index: 14865
index: 15856
index: 16847
Blocks Created!
```

```
def Count_Kmers(blocks , k):
    ''' Count kmers of blocks of given genome (phase 2.2)
    Parameters
    -----
    - blocks: all blocks given genome (type: list)
    - k: k(type: int)
    Return
    -----
    - Kmer_Count: counts of kmers in each block separately (type: list of dicts)
    '''
    Kmer_Count = []
    All_kmers = []
    for block in blocks:
        kmers = []
        for i in range(len(block) - k + 1):
            kmer = block[i:i+k]
            kmers.append(kmer)
        All_kmers.append(set(kmers))
    |
    kmer_count = {}
    for kmer in kmers:
        kmer_count[kmer] = kmers.count(kmer)
    Kmer_Count.append(kmer_count)
    return All_kmers , Kmer_Count
```

این تابع بلاک های یک ژنوم را در قالب لیست دریافت می کند و به ازای هر بلاک ابتدا kmer هایش را استخراج می کند. این کار باعث شمارش ساده تر می شود چون حالت های kmer های تکراری حتی با همپوشانی را هم استخراج میکنیم بعد ساده تر میتوان آن ها را شمرد.

در For دوم کار شمارش را انجام میدهیم. ابتدا هر kmer را از لیست kmerها برمیدارد و می شمارد و در دیکشنری قرار می دهد. با توجه به اینکه در دیکشنری key ها یونیک هستند مشکلی برای kmer های تکراری نداریم چون یک بار شمرده می شوند و در key مربوطه آورده می شوند. یک مثال ساده برای توضیح مطلب فوق:

```
1 kmers = ['a', 'a', 'a' , 'b']
2 kmer_count = {}
3 for kmer in kmers:
4     kmer_count[kmer] = kmers.count(kmer)
5 print(kmer_count)

{'a': 3, 'b': 1}
```

در نهایت تمام kmer ها و تعدادشان را بر میگرداند.

تابع initialize_bitarray:

```
def initialize_bitarray(k):
    bit_array = bitarray.bitarray(4**k)
    bit_array.setall(False)
    return bit_array
```

این تابع در اصل presence/absence table برای هر بلاک در هر ژنوم یک bit array می سازیم. هر جا که kmer های آن بلاک حضور داشته باشد (که نحوه پیدا کردن ایندکس هر kmer را در تابع Filter خواهیم گفت) را True میکنیم و ایندکس هایی که False مانده آن kmer هایی هستند که ندیده ایم.

تابع Filter:

```
def Filter(k, All_kmers):
    ''' Filter kmers by turn the index of presence kmers in related bit-array to 1 (phase 2.3)
    Parameters
    -----
    - k: kmers length (type: int)
    - All_kmers: kmers of blocks (type: list)
    Return
    -----
    - Presence_Absence: turn the index of present kmers in related bit-array (type: list of arrays)
    '''
    bm = { 'A' : 0, 'C' : 1, 'T' : 2, 'G' : 3 }
    Presence_Absence = []

    for block_kmers in All_kmers:
        ba = initialize_bitarray(k)
        for kmer in block_kmers:
            idx = 0
            for j in range(k-1, -1, -1):
                if kmer[j] not in ['A', 'C', 'T', 'G']:
                    kmer = kmer.replace(kmer[j], 'A')
                idx += 4**(k-j-1) * bm[kmer[j]]
            ba[idx] = True
        Presence_Absence.append(ba)
    return Presence_Absence
```

این تابع قسمت سوم این فاز را پیاده سازی می کند. در این تابع به ازای kmer های استخراج شده برای هر بلاک که در مرحله قبل انجام شده ابتدا یک bit-array تعریف میکنیم.

برای هر حرف یک کد نظر میگیریم.

```
bm = { 'A' : 0, 'C' : 1, 'T' : 2, 'G' : 3 }
```

در این صورت هر kmer را در مبنای ۴ نمایش داده ایم. عدد متناظر هر kmer در حالت decimal، ایندکس متناظرش در آرایه bit-array می شود. بنابراین ایندکس موردنظر را true میکنیم. باقی ایندکس هایی که false مانده اند kmer هایی هستند که ندیده ایم.

فقط یک نکته دیگر باقی می ماند آن هم دلیل آن if داخل فور آخر است. یکی از ژنوم ها چند حرف K داخل بلاک هایش بود!

```

3CAATTGACCCAATAATTTGATCAATGGAACAAGTTACCCCAGGGATAA(
SACAAACCGAACCCCTTCGACCKCATAGAAGGCGAGTCAGAACTAGTA(
CCCCATCTCAATTATATGTCAAATCCACCCATCAATCAACACCTACGCC(
ICAGCCATTTTACCTTCCTCCACCCATGTTTATTAATCGTTGACTCTTT(

```

برای حل این مشکل هر حرف غیر از حروف معمول را A در نظر گرفتیم با توجه به اینکه کد A را 0 در نظر گرفتیم در محاسبات بهتر بود.

تابع Kmer_Count_and_Filter :

```

def Kmer_Count_and_Filter(All_blocks):
    All_Counts = {}
    All_Presence_Absence = {}

    for genome in All_blocks:
        k = int(genome[genome.index('k')+1:genome.index('_')]) # به دست آوردن k از نام فایل
        if k != 0:
            all_kmers , kmer_counts = Count_Kmers(All_blocks[genome] , k)
            if k!= 24 and k!= 32:
                presence_absence_this_genome = Filter(k , all_kmers)
                All_Presence_Absence[genome] = presence_absence_this_genome
            All_Counts[genome] = kmer_counts
    return All_Counts , All_Presence_Absence

```

این تابع وظیفه فراخوانی دو تابع Count_Kmers و Filter به ازای همه ی ژنوم ها را دارد.

با توجه به اینکه برای $k=0$ شمارش kmer بی معنی بود صرفاً برای k های دیگر شمارش انجام شده است. از طرفی با توجه به نیازمندی حافظه زیاد برای $k = 24$ و $k = 32$ برای این دو مورد صرفاً count ها را محاسبه کردم.

من کد را در محیط colab نوشته ام. برای همین نیاز بود روی گوگل درایو mount کنم تا بتوانم به فایل دیتاست دسترسی داشته باشم.

```
[1] 1 %cd "/content/drive/My Drive/phase2Algo"
```

```
↳ /content/drive/My Drive/phase2Algo
```

```
[2] 1 mkdir 'blocks' #this folder create in the same path and contain blocks files
```

```
[3] 1 pip install biopython # for read fasta files
```

```
↳ Requirement already satisfied: biopython in /usr/local/lib/python3.6/dist-packages (1.77)
Requirement already satisfied: numpy in /usr/local/lib/python3.6/dist-packages (from biopython) (1.18.5)
```

```
[4] 1 pip install bitarray # for creating presence/absence table
```











```
↳ Requirement already satisfied: bitarray in /usr/local/lib/python3.6/dist-packages (1.4.2)
```

```
[5] 1 from Bio import SeqIO
2 import os, glob
3 import itertools
4 import bitarray
```

در خط اول وارد مسیر پروژه میشود.

فایل دیتاست را در همین مسیر پروژه قرار داده بودم

خط دوم فولدر blocks را می سازد که فایل های فاستای مربوط ب بلاک ها آن جا قرار است ذخیره شوند. که به این صورت است.

My Drive > phase2Algo > blocks ▾				⌵ ⓘ
Name ↑	Owner	Last modified	File size	
 k0_AF010406.1.fasta	me	3:41 PM me	17 KB	
 k0_AF303110.1.fasta	me	3:41 PM me	17 KB	
 k0_AF303111.1.fasta	me	3:41 PM me	17 KB	
 k0_AF533441.1.fasta	me	3:41 PM me	17 KB	
 k0_AJ001562.1.fasta	me	3:41 PM me	16 KB	
 k0_AJ001588.1.fasta	me	3:41 PM me	17 KB	
 k0_AJ002189.1.fasta	me	3:41 PM me	17 KB	
 k0_AJ238588.1.fasta	me	3:41 PM me	16 KB	
 k0_AY488491.1.fasta	me	3:41 PM me	16 KB	
 k0_AY863426.1.fasta	me	3:41 PM me	16 KB	

همچنین دستور هایی برای نصب پکیج های biopython و bitarray را قرار داده ام که در صورتی که نصب نداشته باشید
میتوانید اجرا کنید.

در نهایت پکیج های مورد استفاده را import کردم.

می توانید در محیط google drive یک فولدر ایجاد کنید که این فایل کد و دیتاست میتوکندری در آن قرار دارد. سپس خط
اول cd را به مسیر آن فولدر تغییر دهید و خطوط بعدی را اجرا کنید. ☺