BC205 Algorithms in Bioinformatics Simaiaki Agapi Eleni

Exercise 1

Question 2: Identifying non-mers in a bacterial genome

Non-mers are k-mers that don't have a single instance in a greater corpus (e.g. a genome), that is they do not exist in a genome. Search the genome of E. coli for any given 10-mer and report the 10-mers that do not exist in the genome.

General outline:

The main idea of identifying the non-mers is quite simple as a concept. First, we have to find all the possible 10-mers that can be produced in nature. Then, we proceed with isolating all the existing 10-mers in the reference genome, in our case *Escherichia Coli.* Finally, we compare those and keep all 10-mers that are not found in the reference genome.

In detail:

1. Download the reference genome of *Escherichia Coli* (fasta format). The link is given in the question. Another source for the reference genome would be through the NCBI – NIH database (https://www.ncbi.nlm.nih.gov/data-hub/taxonomy/562/).
2. Reading the reference genome and saving it in a single line string without any spaces or other information that can be found in a fasta format file. The Python code below reads the first line of the fasta file, which contains information on the sequence identification etc., discards it and for each next line keeps only the text found and appends it in the initially empy string “ecolseq. Here we have the whole genome, so we know that only the first line does not contain nucleotide sequence. In another case we would have to adjust the code.

```
file = "./ecoli.fa.txt"
      with open (file) as l:
          l.readline()
          ecolseq =""
          for ln in l:
              lnseq = ln.strip()
              ecolseq += lnseq
```

3. Finding all the possible 10-mers that can be formed. This in known as permutations with replacement and can be calculated easily in Python using the function

itertools.product() from the library itertools. We create a list with all the nucleotides that will be used as the base for the production of all possible permutations. In our care we are looking into 10-mers. It is necessary to use the “”.join() function, so that each combination will appear as one string and nor separate bases.

```
import itertools
    k=10
    bases = ["A","T","G","C"]
    combos = itertools.product(bases, repeat=k)
    strcomb = ("".join(y) for y in combos)
```

4. Find all 10-mers in the reference genome. The expected number of 10-mers is equal to its total length of nucleotides minus ten plus one. The idea here is to keep the first 10-mer and move every time by one nucleotide saving the new 10-mer in a list. At the end we will have a list of all the 10-mers found sequentially in the reference genome. Then we can only keep the unique ones, as there is a possibility that some might exist more than once. That can happen easily easily in Python by using the set() function.

```
kmers=[]
    for i in range(len(ecolseq)-k+1):
        kmers.append(ecolseq[i:i+k])
    kmers = set(kmers)
```

5. Counting the non-mers. Since we now have the the 10-mers found in the reference genome “kmers” and all the possible 10-mers “strcomb” we can easily scan the “strcomb” in the “kmers” and keep only those that are not found in the latter. Here the code provided is not quite simplified as it contains some check points.

```
count=0
    nonkmers =[]
    allk =0
    res =0
    for y in strcomb:
        allk += 1      #control to check if the theoritical 4^10 has been
    produced with the product functional tool
        if y not in kmers:
            count  +=1  #counting  the  10-mers  not  included  is  same  as
    len(nonkemers)/not necessary / check point
```

```
        nonkmers.append(y)
    else:           #not necessary / check point
        res+=1
```

6. Finally, the variable `count` or `len(nonkmers)` will give the number of non-mers. Some of those can be seen bellow. Here the function random.sample() from the library random was used to select randomly non-mers as a representation of some of the non-mers found. Another way would be to present the first e.g., 10 non-mers in the `nonkmers[:10]` list but since in the code there is another solution using solely sets instead of lists (for faster processing I believe) I decided to to do t like that. Additionally, the itertools.product() gives a somehow sorted gnerator, hence, to provide a more diverse representation I went for the random.sample() approach.

```
import random
        print("The number of non 10-mers is:",count)
        print("Some of those non-mers are", random.sample(nonkmers,10))
```

The results of the latter commands are:

```
The number of non 10-mers is: 223321
        Some of those non-mers are ['GCAAGTTTAG', 'ATTATGTGCA', 'CTAGGAGGTC', 'TCGCATAGGT', 'TTCCTCCTGG', 'TCTTAGACGT', 'GCAAGCCGTG', 'GGGTCAATCG', 'TAAATTCTAG', 'CCATAGGCGA']
```

7. In order to cross check my results, I tried to calculate theoritically the expected non-mers. I did so by calculating the all the possible 10-mers that according to permutation with replacement are to be 4^10 = 1048576 and subtracting the number of the unique 10-mers found in the reference genome. Yet, there is a small deviation of 19 non-mers more in the ones found with the described procedure compared to the theoretical expected. I am not quite sure why is that. The same happened with the second way I tried (it can be found in the BC205_Exercise_1_Simaiaki_Agapi.py file).

```
theor = 4**10
        obs = len(kmers)
        exp = theor - obs
        print("According to theory we would expect", exp,"non-mers since all the 10-mers are", theor,"and the observed in E.Coli are", obs)
```

The results printed are:

According to theory we would expect 223302 non-mers since all the 10-mers are 1048576 and the observed in E.Coli are 825274

More information on the procedure can be found in the python file BC205_Exercise_1_Simaiaki_Agapi.py