Course 22110
Python and Unix for Bioinformaticians

Project

Resistance to Antibiotics

Alignment of NGS data to a list of genes encoding the resistance to antibiotics. Identification of the genes which are present in the sequenced sample

Group members:
Katarzyna Wisniewska
s203395

Vlad Rosca
s202809

29.11.2021

Introduction

Next generation sequencing is a high-throughput method of determining the order of nucleotides in a genetic sequence. [1] It is crucial in many life science fields, as it provides the opportunity of revealing sequences of not only gene regions of interest, but even whole genomes, therefore allowing confirmation of successful cloning or identification of phenotypic traits, i.a. likelihood of disease or, as in the case of this project, presence of antibiotic resistance within microbial populations.

Such data is very important, as multiple studies report a growing tendency of acquiring antibiotic resistance by microbes, therefore rendering antibiotic treatments inefficient against many pathogenic microorganisms [2][3].

As the current capacity of NGS allows reading of relatively short sequences at a time, the output of this process for bigger sample sizes can be a FASTA file with up to thousands or even millions of sequence reads in random order. This brings the necessity of developing efficient methods of data treatment, sequence alignment and filtering our data of interest out of this vast genetic pool.

Contribution

During the course of this project, all members have contributed equally to both creating the gene detection program, as well as writing the report. Therefore a 50/50 contribution can be considered.

Code

Theory behind the code

The purpose of this project was to develop a Python program that could identify genes responsible for antibiotic resistance in a given metagenomic sample. The input data consisted of NGS data and a database file of possible resistance genes that we could find within it (approx. 2100 genes).

As imposed by the project guidelines, we have divided the reads into k-mers of 19 nucleotides each - a sequence long enough to be assumed unique. The k-mers of each read were treated as a whole, and needed to “align” along a given gene all together.

Two important terms are defined for this project: depth of sequencing and coverage of the gene. Depth is the number of times the NGS reads, combined, match the sequence of the whole gene. Coverage is the fraction of nucleotides in the gene that are considered to be found in the sample. In this project, to ensure that the read match is not a result of missequencing, the minimal requirement for depth was set to 10, meaning that at least 10 NGS reads have to match any specific part of the gene. The requirement for the coverage was set to 95%, i.e. 95% of nucleotides in the gene sequence have to have depth at least 10.

Since NGS often contains errors, it is very likely to have single-nucleotide polymorphism in the read due to sequencing errors. Therefore, one SNP difference was allowed to still consider the sequence of the read a match provided all the other nucleotides in the sequence are identical in the read and in the gene.

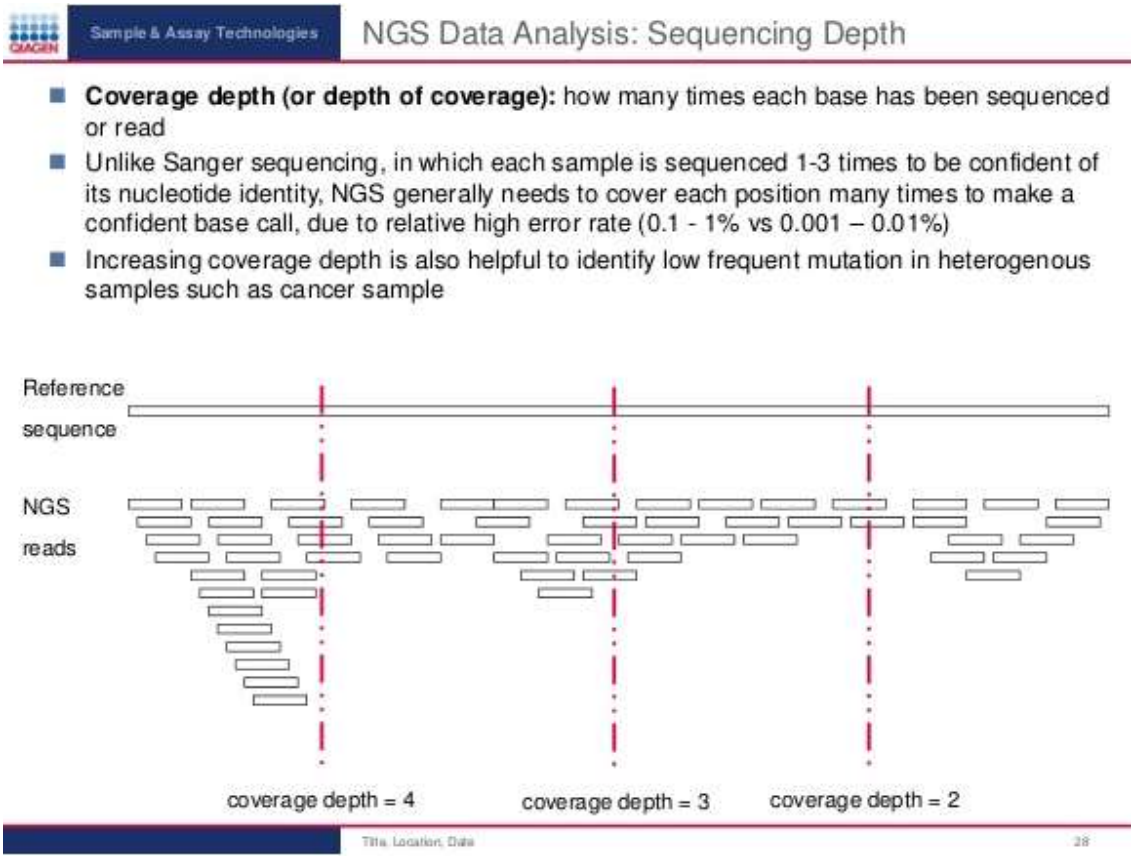

Figure 1. Aligning NGS reads to the reference sequence (the gene)

Each alignment of an NGS read to a part of a gene increased the coverage depth of the nucleotides in this region of the gene by 1.

Finally, when aligning the NGS data, even though k-mers are used to filter out the reads and the genes, it is the whole reads that are aligned, so single k-mer matches do not contribute to the final depth values.

Algorithm design

The main idea behind our code was to find the most efficient way of processing the NGS reads, and then - if present in any gene - establishing their positions on the gene and saving them as depths for the involved nucleotides.

We had an NGS sample separated into two files of paired-end reads, each having around 3.5 million reads. As we could not be sure if a given read was sequenced from

a forward or a reverse strand, a reverse complement strand was created for each read, and the new reads were included in the matching process as well, bringing the targeted read number to around 14 million. It would be insane to save millions of sequences in a variable. Therefore, we decided to work on the lines of the two files while iterating through them.

The resistance gene sequences, however, were saved in a list of strings because we needed to work with them continuously and reading the file over and over would be extremely inefficient. Moreover, all possible k-mers were extracted from the gene sequences in one large common dictionary structure.

Due to a relatively large datasize, time efficiency was crucial for the design of our code. Our main strategy to speed up the processing time was to narrow down the read database by preliminary elimination of reads that do not seem to match any resistance gene, as well as determine which genes are a probable match for the specific read, and thus only analyze the selected genes for any chosen read.

```
Gene_start = begining of the read
Gene_end = end of the read
If gene_start or gene_end in gene_dict:
   Save gene number for analysis
For gene in saved_genes:
   Align ngs_read with gene
```

Preliminary elimination of the reads was done by getting the beginning and the end of each read and checking if any of them is present in the gene dictionary of k-mers. If either the beginning or the end were present, the genes with corresponding k-mers were saved. Afterwards the given read was only analyzed with the selected genes.
The following alignment cases have been considered:

a. If beginning and end of the read are present in the gene – the read may align to any part of the gene, but has to be taken entirely
b. If only begining of the read is present – the read has to align to the end of the gene
c. If only end of the read is present – the read has to align to the begining of the gene

In each case the positions of the corresponding sequences in the gene were found and compared to the reads. If the read (or the part of the read) was identical to the part of the gene or only had one single-nucleotide polymorphism difference (one SNP difference, the read was considered aligned and the data about it was saved. Otherwise, the read was discarded for the given gene.

Program design

In order to make the main code more clear, functions were created for certain repetitive or code-heavy tasks:

- read_filename() - for capturing names of the files either from sys.argv or, in case they were not provided, for demanding the filenames as input from the user.
- complementary_strand(read) - for creating a reverse complement strand to a currently processed NGS read line.
- get_ngsread_kmer_list(dna, k) - to divide a given read line in the input file into 82 k-mers (k = 19), each shifted one nucleotide further than the previous one, and saving these values in a list.
- get_genes(filename) - for identifying and extracting resistance gene sequences, as well as their names (title lines in the file), and then saving them in two separate lists.
- get_gene_kmer(genes, k) - for creating resistance gene k-mers and saving them in the form of a dictionary, where keys are k-mer sequences, while values are lists of gene positions where these k-kmers are present.
- align_ngs_gene(read, gene, start_check, end_check, gene_num, k) – for identifying the position in the gene where the read can be successfully aligned. This is the biggest function which receives all available data on the specific read and the specific gene, and tries to align them. It takes into account all three alignment cases described above as well as the SNP difference counter. If the alignment is not possible, None is returned.

The bigger data structures used in the program:

- gene_list – a list of strings, each containing a sequence of a gene
- gene_names – a list of the identifying lines of each gene
- gene_dict – dictionary of dictionaries, where the *k-mer as the key* leads to the *dictionary* of gene numbers as keys and to the positions of the k-mer in each gene as the values. The most strategically important data structure in the program as it allows to access the position of a k-mer in a selected gene momentarily.
- depth – a list of lists, each containing depth the corresponding nucleotide of the given gene
- coverage – a list of floats containing coverage values for every gene
- avg_depth – a list of floats containing average depths for every gene

In order to filter out the irrelevant reads, the first and the last k-mers of each read were checked on presence in any of the genes. To do so, we needed to be able to assess quickly which k-mers are present in which genes without making additional searching loops. The solution for this was a dictionary having k-mer sequences as the keys and the gene numbers as the values. This was further developed into dictionary of dictionaries, where the *k-mer as the key* led to the *dictionary* of gene numbers as keys and to the positions of the k-mer in each gene as the values.

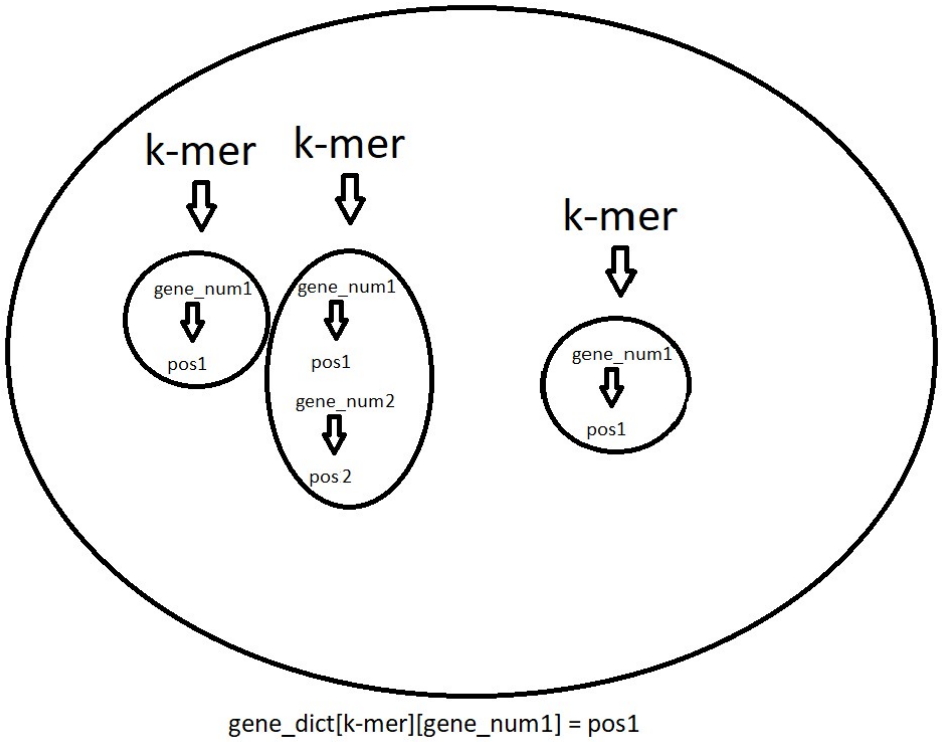

Figure 2. 'gene_dict' variable -- dictionary of dictionaries

All the functions were grouped at the beginning of the code for easy access.
Two libraries: sys and gzip were imported for reading filenames and unzipping the files, respectively.
The code is compartmentalised into sections and subsections with use of comments, as can be seen below:

```
'''MAIN CODE'''

k = 19 #setting the kmer length

###processing resistance genes

#Obtaining lists of resistance gene sequences and names
[gene_list, gene_names] = get_genes('resistance_genes.fsa')

#Obtaining a dict of res_gene kmers with their respective locations as values
gene_dict = get_gene_kmer(gene_list, k)
```

In the presented piece of code, the k-mer size is set to 19, and the available functions are used to create a dictionary of resistance gene k-mers, as well as lists of resistance genes and gene names. A small extract of the resistance gene dictionary looks as follows:

{'ATATATGAGAGAATTCGTT': [[2125, 480]],
'TATATGAGAGAATTCGTTA': [[2125, 481]],
'ATATGAGAGAATTCGTTAA': [[2125, 482]], ...}

Next, a list of initial depths is created with use of a simple for loop shown below. It is a list of lists, where each internal list consists of the amount of zeros equal to the length of a given gene.

Visualisation of the initial depth list is shown below:

[0, 0, 0, …0], [0, 0, 0, 0, 0, 0, …], [0, 0, 0, 0,…]

The main structure of the code is based on multiple for loops, starting with a loop iterating through a list of two input files called “filenames”. This allows implementation of the same steps on both of the files, starting with opening and unpacking them with use of the gzip command :

```
for file in filenames:  #to iterate through both files
    with gzip.open(file, "r") as infile:
```

Next, the program goes through all the lines of a given file, decoding them from ASCII

```
        for b_line in infile:
            line = b_line.decode('ASCII')
```

The read lines in the input files are found with use of stateful parsing, as can be seen below:

```
        if line[0] == '@':
            flag = True
        elif flag:

            if line[0] == '+':
                flag = False
```

Once we are within a flag (a proper line is identified), a complementary strand is created for the sequence of the current read, and both of them are saved as strings in a list called “both_reads”. Another for loop is initialized to iterate through both the forward and reverse strand.

```
            else:
                read_count += 1
                # get the dna seq
                dna_read = line
                #get the reverse complement for the read
                rev_read = complementary_strand(dna_read)
                both_reads = [dna_read, rev_read]
                for read in both_reads:
```

Within this new loop, a list of k-mers is created for each strand, and the first and last k-mer of each read is searched for in the dictionary of resistance genes.

```
for read in both_reads:
    #create kmer lists for both
    read_kmer = get_ngsread_kmer_list(read, k)
    #checking if extremities of the read fit the dict (initial read elimination)
    start_check = gene_dict.get(read_kmer[0])
    end_check = gene_dict.get(read_kmer[-1])
```

If either the first or the last k-mer of the read is present in the dict as a key, the read is considered as a potential match and is further processed. In order to avoid multiple hits per one read, every 19th k-mer is searched for in the dict with a ‘get command. Upon finding the key, the command returns a value assigned to it, in our case being a list of coordinates leading to the positions of a given k-mer in specific genes. With use of these coordinates, the depth can be updated by 1 at nucleotide positions that are covered by the matching k-mer.

Finally, coverage for each gene can be calculated based on the values saved in the depth list. Every nucleotide with depth higher than 10 counts in the coverage, and coverage of 95% and above is accepted to confirm presence of the gene in the

metagenomic sample. The coverage values are saved in a dictionary, which is then sorted in order to get only genes with the highest coverage values.

```
coverageCount = 0
coverage = dict()
for gene_num in range(len(depth)):
    hitCount = 0
    for nt in depth[gene_num]:
        if nt >= 10:
            hitCount += 1
    cov = hitCount/len(depth[gene_num])
    if a > 0.95:
        coverageCount += 1
    coverage[gene_num] = cov

sorted_gene_num = sorted(coverage.keys(), key=coverage.get, reverse=True)
```

Program manual

In order to use the program, open the Ubuntu console and type in:

```
./python.py <filename1> <filename2>
```

Where filename1 and filename2 are names of the input NGS data files. You can also type in

```
./python.py
```

You will then be asked to enter the file names.

While the program is running, it shows percentage of the NGS reads that have been processed. (adjusted to the number of reads in our sample, may work wrong for files with a different size)

The output of the program is the list of genes which obtain 95% of coverage and minimum depth of 10. This means that 95% of nucleotides in the gene has depth of at least 10. The coverage and average depth is displayed for every gene name.

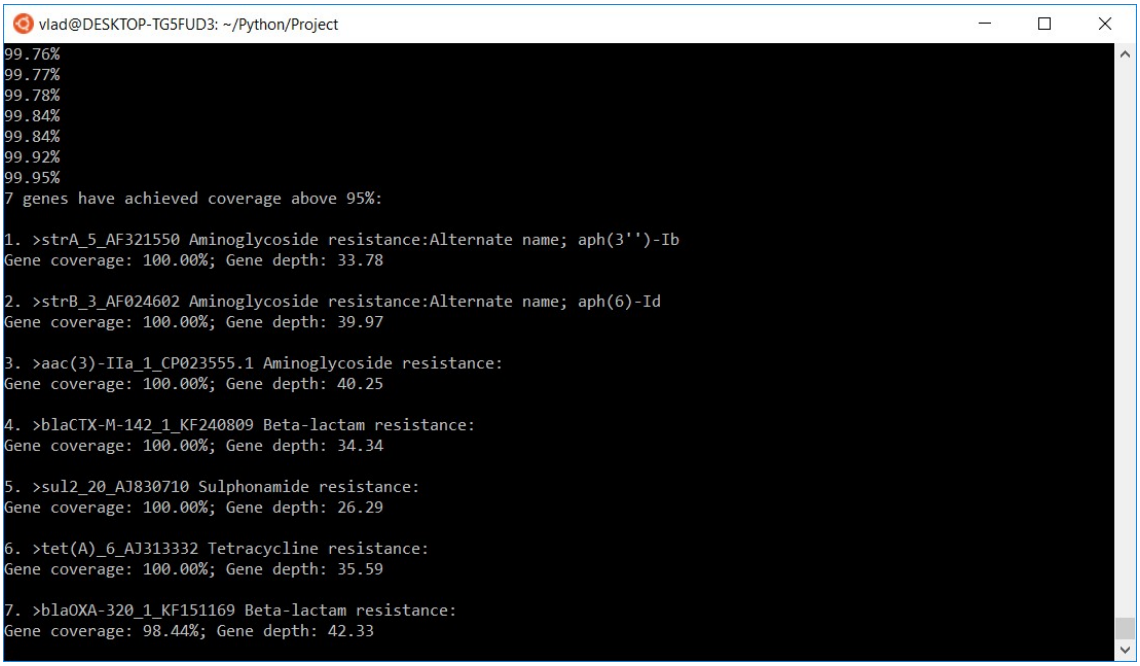

Figure 3. Output of the program with the identifying lines of genes, their coverages and depths.

- Theory - explanation of more theoretical aspects
- Algorithm design - explanation of the central algorithm
- Program design - how the program is built
- Program manual - how to use the program(s), input format, expected output, example runs

Conclusions

We identified 7 genes which have been sequenced and identified in the sample with over 95% of coverage for the minimal depth of 10. Out of those, 6 genes got 100% coverage. The running time of the program is 4.5 minutes for about 14 million NGS reads which is considered a good result.

The program takes into account SNP differences, but the solution is not perfect and can be improved.

References

[1] https://emea.illumina.com/science/technology/next-generation-sequencing.html
[2] Martínez-Martínez L, Calvo J. [The growing problem of antibiotic resistance in clinically relevant Gram-negative bacteria: current situation]. Enfermedades Infecciosas y Microbiologia Clinica. 2010 Sep;28 Suppl 2:25-31. DOI: 10.1016/s0213-005x(10)70027-6. PMID: 21130927
[3] Nathan C, Cars O. Antibiotic resistance--problems, progress, and prospects. N Engl J Med. 2014 Nov 6;371(19):1761-3. doi: 10.1056/NEJMp1408040. Epub 2014 Oct 1. PMID: 25271470.