Computational Biology I project 20/21

Salmonella Outbreak

MANUCHARIAN Vardan, USMANOVA Anastasiia,
ZACHAREK Roman Edward
MSIAM M2

January 8, 2021

Contents

1 Introduction

Nowadays, humanity faces with an increasing amount of issues connected with infections and bacteria. New mutations occur in bacteria and existing antibiotics sometimes have no effect. To create new types of antibiotics we need to understand what mutations appeared and where.

For the Salmonella bacteria it was found two different strains of the bacteria (resistant to tetracycline and wild type).

We took a call from TAFTAR to develop a tool for finding the difference between two strains and understanding what gene(s) is involved.

In this report we will explain the algorithm that we offer and show the results on a real data.

2 Problem definition

We need to create a tool that has the following specifications:

input: two FASTA sequencing files (one is wild type, second contains mutations, Illumina reads),

output: a list of SNPs.

Moreover, we need to understand the difference between two strains and find mutated gene(s).

The following features of a data has to be taken into account:

- Illumina produces short reads (length of 250 symbols, positions of reads are approximated by a uniform distribution)
- number of reads is 1993167
- in produced data there is $< 1\%$ of sequencing errors (uniformly distributed, which consists in substituting a letter by another one)

3 Proposed solution

We could solve this problem by whole DNA sequencing of two FASTA files and subsequent comparison of them (for example, with the use of de Bruijn graphs [1]), but it is computationally not efficient solution for this task.

We are proposing instead to use hash maps in order to create a database of all possible k length strains(called k-mers) from both files in order to compare them and deduce where the SNPs are located. The method is as follows:

1. build a dictionary of all possible k-mers for both of input files (with taking into account the optimal choice of k – for this we need to know number of occurrences of k-mers in our files)

2. find k-mers which belong to a first FASTA and don't belong to a second and vice versa while taking into account possible errors in reads – for this we will not count k-mers which occur only few times in the whole file, in our case we take a threshold of 5. That is, we ignore k-mers that occur less than 5 times.

3. to compare found k-mers – if a k-mer is present more than 5 times in one file, it means that is was indeed sequenced from the DNA, and if it's not present in the second file, it means that a SNP is present in that k-mer.

4. to find the name of a protein with the use of "blastx" tool.

Our solution has a linear time complexity. Bellow we will describe precisely each of the steps of our method with mathematical justification and empirical proof.

4 Optimal choice of k-mers's length

4.1 Theory

Correct choice of k-mers's length is necessary for building the appropriate dictionary for both of our input files. Small and large k has their own disadvantages.

Disadvantages of large k:

1. The probability to have an error in a k-mer is proportionally increased with the length of the k-mer, so we will have a lot of k-mers which we will meet only 1-2 times and will not use them in final comparison of k-mers of two FASTA files as we will count them as k-mers with errors.

Proof. Error is uniformly distributed with an error rate $p = 0.01$. Number of errors of a k-mer (for each reading) follows $Bin(k, p)$. So the probability that a k-mer will have n errors is

$$\binom{k}{n} p^n (1-p)^{k-n}.$$

Therefore, the probability to have the number of errors ≥ 2 for $k = 30$ is 3.6%, for $k = 60$ is 12%, for $k = 100$ is 26%. $\square$

2. Computationally not efficient. The bigger the length of the k-mer, the bigger the number of unique keys in dictionary, because the number of overlapping k-mers is smaller when the length of the k-mer is bigger. We have more keys in our dictionary, so the computation is much slower, and it also takes much more memory. For example, we compared the time it took to have the code executed with $k = 100$ and $k = 15$, and the execution was 2.2 times longer for the case $k = 100$.

Disadvantages of small k:

If we take small k-mers, the probability that they will occur in several parts of strain is increasing. If a mutation appears, we will not be able to unequivocally restore the gene(s) with mutation as k-mers cover several parts of strain. It is also possible not to find a mutation by our method as k-mers will have such a small length that they will occur in both of dictionaries of input files.

Idea of finding the optimal k:

To find the lower bound of k we will look at k-mers's frequency distributions for real data for different k. If k is too small and some k-mers occur several times, the distribution will be a mixture of Poisson distribution with several peaks. Otherwise it will be Poisson distribution with only one peak.

Proof. Let's say L is the length of genome, l is the length of one read and N is the number of reads.

First, let's fix some k-mer (and it's position in strain) and find the probability p that a random read will contain this k-mer. It's clear that there are $l - k + 1$ subsequences with length l containing this k-mer. As the positions of reads in the genome are well approximated by $U\{1, 2, \dots, L - l + 1\}$, we have

$$p = \frac{l - k + 1}{L - l + 1}.$$

Now, as the reads positions are independent, the number of reads containing our k-mer follows $Bin(N, p)$. We know that N is sufficiently large ($\approx 2M$) and p is sufficiently small ($\approx \frac{l}{L} \approx \frac{250}{5M}$) thus the number of reads containing our k-mer is well approximated by $Pois(\lambda)$ where

$$\lambda = Np.$$

But it's only about some fixed k-mer position. Let's say this k-mer occurs t times in the genome. Then $\xi_1, \dots, \xi_t$ stands for number of reads containing each k-mer where each of those two are independent random variables and $\forall i\ \xi_i \sim Pois(\lambda)$. Then the frequency of that k-mer is

$$\xi_1 + \dots + \xi_t \sim Pois(\lambda) + \dots + Pois(\lambda) \sim Pois(\lambda t).$$

In the end, there are k-mers that occur only once in the genome and their frequencies are distributed as $Pois(\lambda)$. There are k-mers that occur twice and their frequencies are distributed as $Pois(2\lambda)$. There are k-mers that occur 3 times and their frequencies are distributed as $Pois(3\lambda)$ and etc... So the distribution of frequencies of all k-mers is a mixture of $Pois(\lambda), Pois(2\lambda), Pois(3\lambda)$ and etc... In the extreme case, when each k-mer occurs only once in the genome, it will be just a Poisson distribution with parameter λ. □

4.2 Practice

We build the distribution of k-mers's frequency for different k. As we can see, for k = 10 the Poisson distribution has several peaks, so it means that k = 10 is too small value (see Figure 1). For k = 20, there are two peaks, but the second one is very small. Therefore, the lower bound for the length k is around 20. But for us, the optimal choice of k will be 30 as we want to use part of gene to find appropriate protein (blastx requires more than 50 symbols). For blastx, we restore part of gene with the length 60 = 30 + 30.

To start off, for a given FASTA file, and for a given k, we go through each sequence (the 250 long sequence of chars) and add to the dictionary each possible k-mer if the k-mer in question wasn't already in the dictionary, and set its value to zero. If it is already in the dictionary, we increment its value. This way the dictionary holds each possible k-mer and its frequency. This process was written in the function `get_data(fasta_reads, k)` where `fasta_reads` is the FASTA file, and `k` is the length of the k-mer, this function returns the created dictionary with all possible k-mers and their respective counts.

```
def get_data(fasta_reads, k):
    fasta_reads_dict = {}

    l = len(str(fasta_reads[0].seq))

    for j in tqdm(range(len(fasta_reads))):
        i = 0
        seq_ = str(fasta_reads[j].seq)
        while i <= l-k :
            chunk = seq_[i:i+k]
            if chunk in fasta_reads_dict.keys() :
                fasta_reads_dict[chunk] += 1
            else:
                fasta_reads_dict[chunk] = 1
            i += 1
    return fasta_reads_dict
```

Once this function is applied for both files, we get two dictionaries that need to be compared in order to find the mutations. This will be done in the next section.

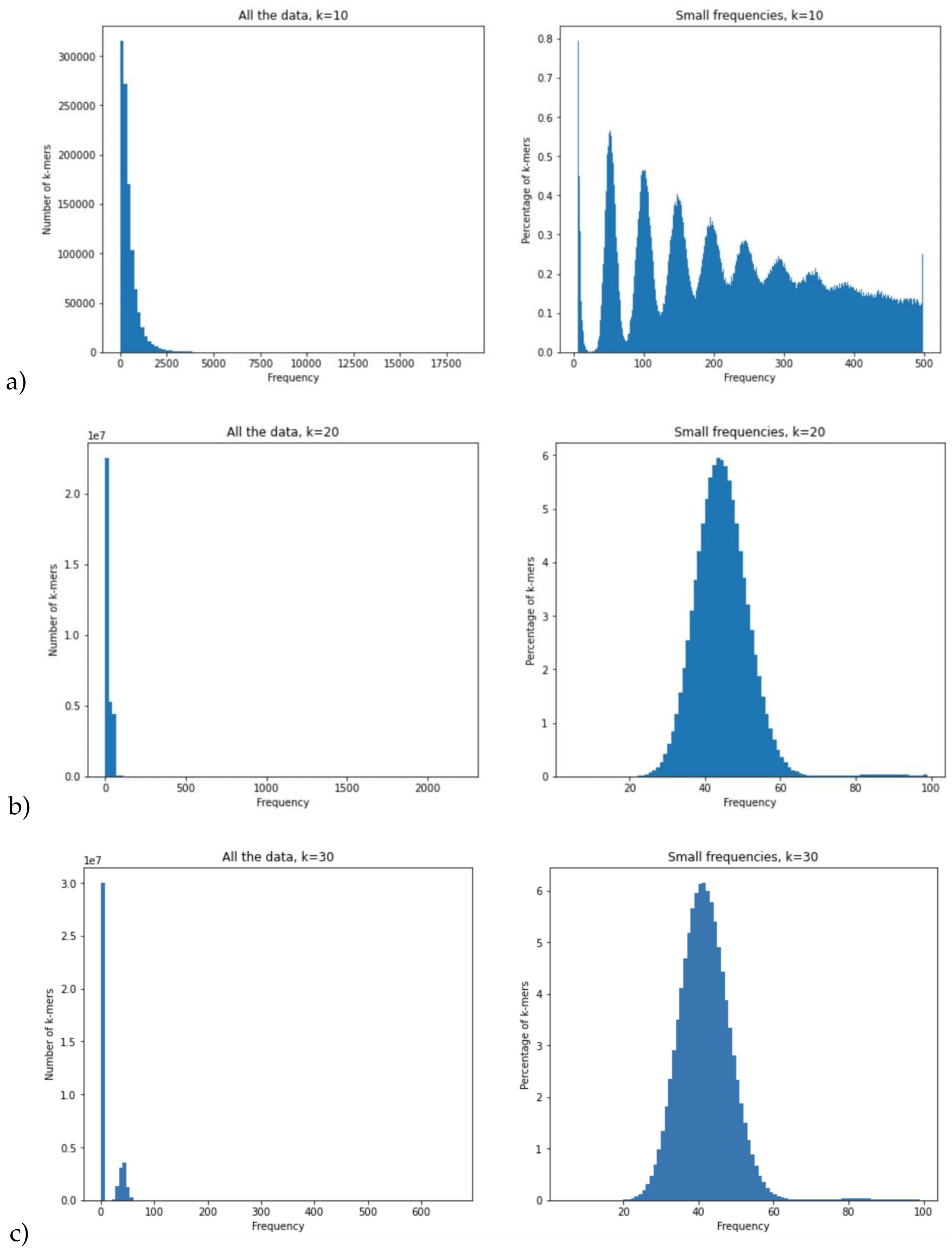


Figure 1: *(a) k = 10 (b) k = 20 (c) k = 30*
Distribution of k-mers's frequency.

5 Comparison of two FASTA files

After we choose an optimal length of k-mers = 30, we need to take into account the errors which occur in FASTA files due to Illumina reads. We put a threshold = 5 on number of k-mers's occurrence. As we can see in a Figure 1 on a plot for k = 30 we have outlier close to frequency = 1 – it is errors. We will count k-mers only if they appear in input files more then 5 times.

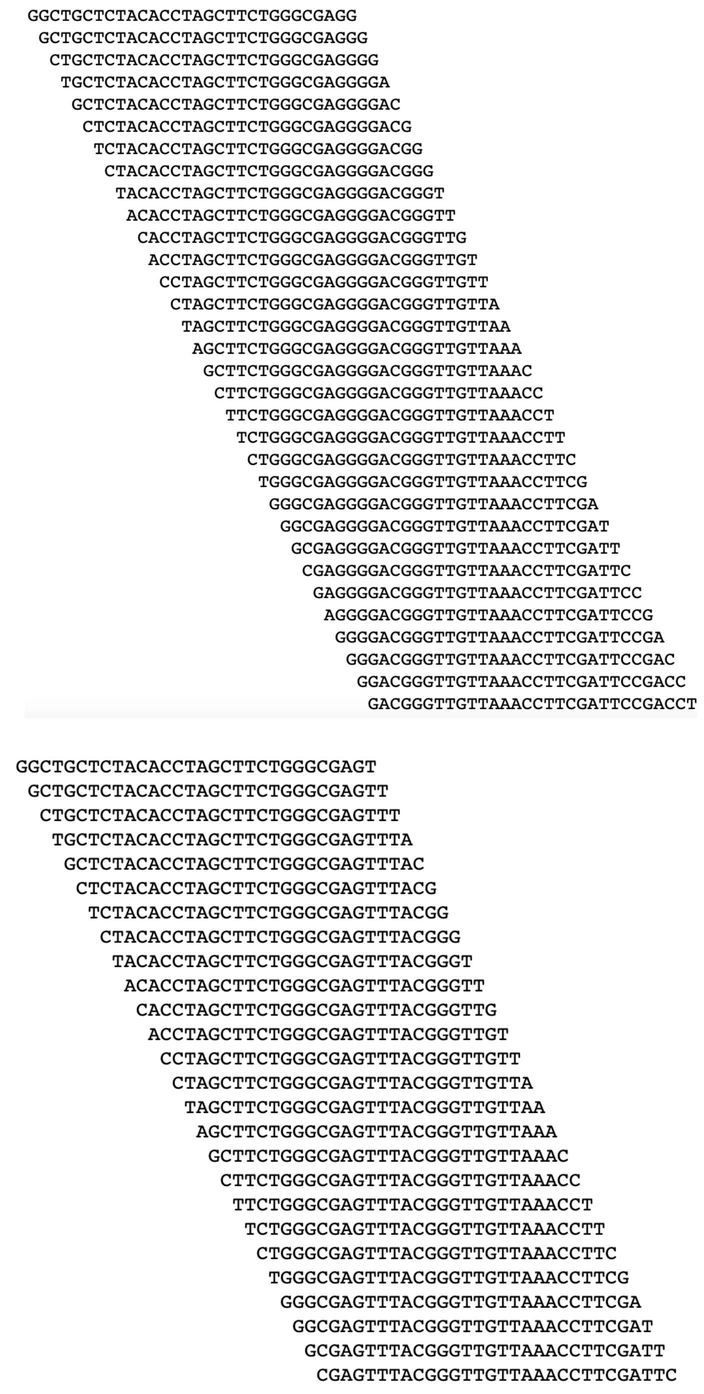
After defining a threshold we can compare dictionaries of two FASTA files. We find k-mers which belong to a first FASTA and don't belong to a second and viceversa while taking into account possible errors in reads. It is done using the function `find_missing_keys_in_dict(from_dict, to_dict, file_name_1, file_name_2)`, the `from_dict` argument is to specify the dictionary through whose keys we will go through and verify if they are in the second dictionary, which is defined by the argument `to_dict`, the arguments `file_name_1` and `file_name_2` are here to specify the names we would like to use when printing in the shell. It will return the k-mers that weren't found in the second dictionary. The python code is below:

```python
def find_missing_keys_in_dict(from_dict, to_dict, file_name_1, file_name_2
                                          ):
    print("Looking at sub-reads from {} file that miss in the {} file".
                                               format(file_name_1, file_name_2))

        chosen_chunks = {}

        for chunk, count in tqdm(from_dict.items()):
            if chunk not in to_dict and count > 5:
                chosen_chunks[chunk] = count

        return chosen_chunks
```

We proceed by using this function for both of our files. Once this is done, we get two dictionaries, and we need to build a sequence that contains the SNPs.We proceed by using this function for both of our files.

6 Finding of mutations

After the comparison of k-mers's dictionaries for two FASTA files we found following list of SNPs:



a)

```
GGCTGCTCTACACCTAGCTTCTGGGCGAGG
 GCTGCTCTACACCTAGCTTCTGGGCGAGGG
  CTGCTCTACACCTAGCTTCTGGGCGAGGGG
   TGCTCTACACCTAGCTTCTGGGCGAGGGGA
    GCTCTACACCTAGCTTCTGGGCGAGGGGAC
     CTCTACACCTAGCTTCTGGGCGAGGGGACG
      TCTACACCTAGCTTCTGGGCGAGGGGACGG
       CTACACCTAGCTTCTGGGCGAGGGGACGGG
        TACACCTAGCTTCTGGGCGAGGGGACGGGT
         ACACCTAGCTTCTGGGCGAGGGGACGGGTT
          CACCTAGCTTCTGGGCGAGGGGACGGGTTG
           ACCTAGCTTCTGGGCGAGGGGACGGGTTGT
            CCTAGCTTCTGGGCGAGGGGACGGGTTGTT
             CTAGCTTCTGGGCGAGGGGACGGGTTGTTA
              TAGCTTCTGGGCGAGGGGACGGGTTGTTAA
               AGCTTCTGGGCGAGGGGACGGGTTGTTAAA
                GCTTCTGGGCGAGGGGACGGGTTGTTAAAC
                 CTTCTGGGCGAGGGGACGGGTTGTTAAACC
                  TTCTGGGCGAGGGGACGGGTTGTTAAACCT
                   TCTGGGCGAGGGGACGGGTTGTTAAACCTT
                    CTGGGCGAGGGGACGGGTTGTTAAACCTTC
                     TGGGCGAGGGGACGGGTTGTTAAACCTTCG
                      GGGCGAGGGGACGGGTTGTTAAACCTTCGA
                       GGCGAGGGGACGGGTTGTTAAACCTTCGAT
                        GCGAGGGGACGGGTTGTTAAACCTTCGATT
                         CGAGGGGACGGGTTGTTAAACCTTCGATTC
                          GAGGGGACGGGTTGTTAAACCTTCGATTCC
                           AGGGGACGGGTTGTTAAACCTTCGATTCCG
                            GGGGACGGGTTGTTAAACCTTCGATTCCGA
                             GGGACGGGTTGTTAAACCTTCGATTCCGAC
                              GGACGGGTTGTTAAACCTTCGATTCCGACC
                               GACGGGTTGTTAAACCTTCGATTCCGACCT
```

b)

```
GGCTGCTCTACACCTAGCTTCTGGGCGAGT
 GCTGCTCTACACCTAGCTTCTGGGCGAGTT
  CTGCTCTACACCTAGCTTCTGGGCGAGTTT
   TGCTCTACACCTAGCTTCTGGGCGAGTTTA
    GCTCTACACCTAGCTTCTGGGCGAGTTTAC
     CTCTACACCTAGCTTCTGGGCGAGTTTACG
      TCTACACCTAGCTTCTGGGCGAGTTTACGG
       CTACACCTAGCTTCTGGGCGAGTTTACGGG
        TACACCTAGCTTCTGGGCGAGTTTACGGGT
         ACACCTAGCTTCTGGGCGAGTTTACGGGTT
          CACCTAGCTTCTGGGCGAGTTTACGGGTTG
           ACCTAGCTTCTGGGCGAGTTTACGGGTTGT
            CCTAGCTTCTGGGCGAGTTTACGGGTTGTT
             CTAGCTTCTGGGCGAGTTTACGGGTTGTTA
              TAGCTTCTGGGCGAGTTTACGGGTTGTTAA
               AGCTTCTGGGCGAGTTTACGGGTTGTTAAA
                GCTTCTGGGCGAGTTTACGGGTTGTTAAAC
                 CTTCTGGGCGAGTTTACGGGTTGTTAAACC
                  TTCTGGGCGAGTTTACGGGTTGTTAAACCT
                   TCTGGGCGAGTTTACGGGTTGTTAAACCTT
                    CTGGGCGAGTTTACGGGTTGTTAAACCTTC
                     TGGGCGAGTTTACGGGTTGTTAAACCTTCG
                      GGGCGAGTTTACGGGTTGTTAAACCTTCGA
                       GGCGAGTTTACGGGTTGTTAAACCTTCGAT
                        GCGAGTTTACGGGTTGTTAAACCTTCGATT
                         CGAGTTTACGGGTTGTTAAACCTTCGATTC
```

Figure 2: (a) *Wild type FASTA file* (b) *Mutated FASTA file*
List of SNPs

Once this is done we need to build a sequence that contains the SNPs. This is done by using the function `combine_found_snps(dict_)` which takes a dictionary `dict_` as an argument and outputs the SNP in the correct form. The code for this is given below:

```
def combine_found_snps(dict_):
    arr = set(dict_.keys())
    to = {}
    from_ = {}

    for el1 in arr:
        for el2 in arr:
            if el1 != el2 and el1[1:] == el2[:-1]:
                assert(el1 not in from_.keys())
                from_[el1] = el2
                to[el2] = el1

    res = []
    for el in set(from_.keys()).difference(set(to.keys())):
        cur = el
        part = el
        while(cur in from_.keys()):
            cur = from_[cur]
            part += cur[-1]
        res.append(part)
    return res
```

Result for the first dictionary: [1]

```
 ['GGCTGCTCTACACCTAGCTTCTGGGCGAGGGGACGGGTTGTTAAACCTTCGATTCCGACCT',
 'AGGTCGGAATCGAAGGTTTAACAACCCGTCCCCTCGCCCAGAAGCTAGGTGTAGAGCAGCC']
```

And for the second:

```
['GGCTGCTCTACACCTAGCTTCTGGGCGAGTTTACGGGTTGTTAAACCTTCGATTCCGACCT',
 'AGGTCGGAATCGAAGGTTTAACAACCCGTAAACTCGCCCAGAAGCTAGGTGTAGAGCAGCC']
```

We need to somehow match the strings from these 2 lists. To do that we will go through each possible pair of strings of these lists (i.e. 4 pairs) and if Levenshtein distance between these subsequences is less than 10 we state that the pair is a SNP.

As we can see, all of our preceding code chunks are linear, thus overall complexity is linear.

As a final step, we will show how we dovetail all of these function in order to get the process working, we do this:

[1] Actually two founded parts of genes are a part of the same gene. We just need to reverse it and substitute nucleotides ($A \leftrightarrow T, C \leftrightarrow G$) on their equivalents.

```python
def list_of_snps(argv1, argv2):
    dict_argv1 = get_data(argv1, 30)
    dict_argv2 = get_data(argv2, 30)

    chunks_missing_in_second_dict = find_missing_keys_in_dict(dict_argv1,
                                             dict_argv2, "first", "second")
    chunks_missing_in_first_dict = find_missing_keys_in_dict(dict_argv2,
                                             dict_argv1, "second", "first")

    found_snps = combine_found_snps(set(chunks_missing_in_second_dict.keys
                                             ()))
    original_without_snps = combine_found_snps(set(
                                             chunks_missing_in_first_dict.keys
                                             ()))

    for original in original_without_snps:
      for found in found_snps:
          if levenshtein_distance(original, found) < 10:
              print("The found snps are: ", end='')
              print(found)
              print("The orignial is:    ", end='')
              print(original)
```

7 Finding a corresponded protein

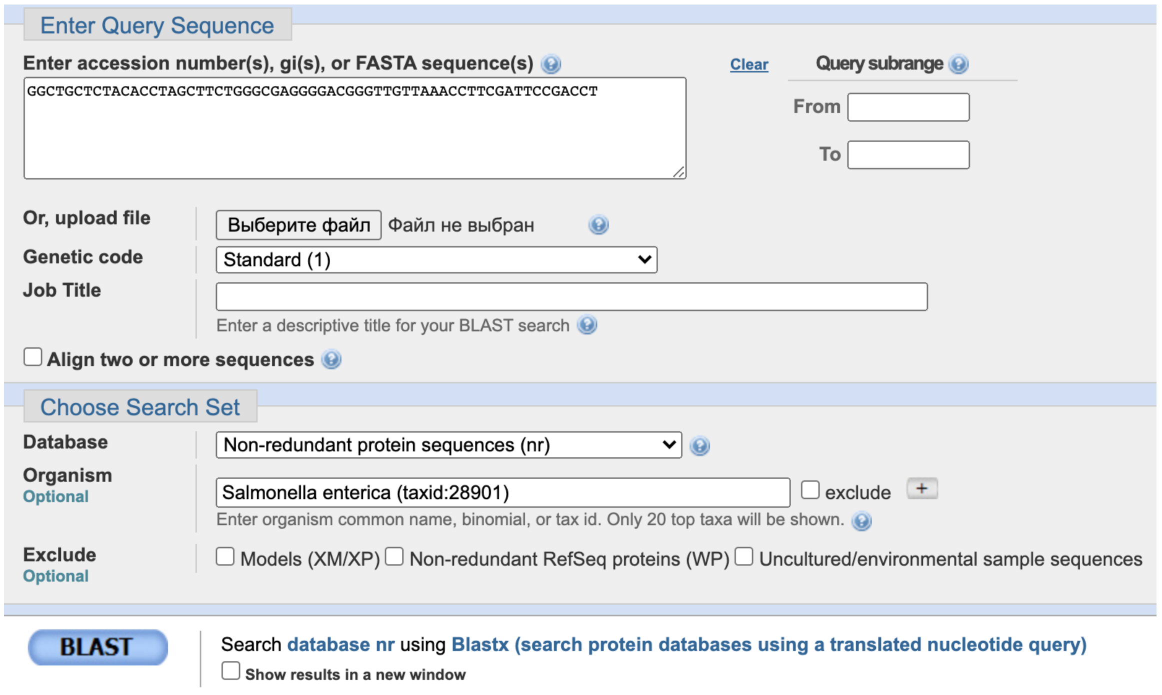
We will use a tool blastx to find a corresponded protein. For this we will copy subsequence with mutation and make a search in a database of proteins for salmonella.

original:

GGCTGCTCTACACCTAGCTTCTGGGCGAG**TTT**ACGGGTTGTTAAACCTTCGATTCCGACCT

mutation:

GGCTGCTCTACACCTAGCTTCTGGGCGAG**GGG**ACGGGTTGTTAAACCTTCGATTCCGACCT



Enter Query Sequence

Enter accession number(s), gi(s), or FASTA sequence(s): GGCTGCTCTACACCTAGCTTCTGGGCGAGGGGACGGGTTGTTAAACCTTCGATTCCGACCT

Clear

Query subrange — From: — To:

Or, upload file: Выберите файл Файл не выбран

Genetic code: Standard (1)

Job Title:

Enter a descriptive title for your BLAST search

☐ Align two or more sequences

Choose Search Set

Database: Non-redundant protein sequences (nr)

Organism (Optional): Salmonella enterica (taxid:28901) ☐ exclude

Enter organism common name, binomial, or tax id. Only 20 top taxa will be shown.

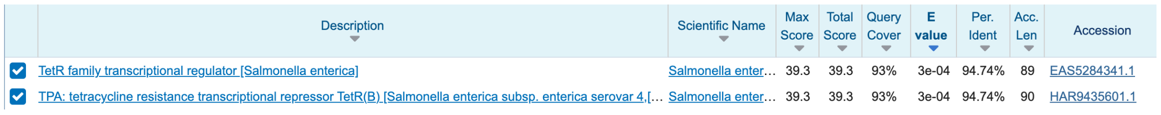
Exclude (Optional): ☐ Models (XM/XP) ☐ Non-redundant RefSeq proteins (WP) ☐ Uncultured/environmental sample sequences

BLAST

Search database nr using Blastx (search protein databases using a translated nucleotide query)

☐ Show results in a new window

a)



	Description	Scientific Name	Max Score	Total Score	Query Cover	E value	Per. Ident	Acc. Len	Accession
✓	TetR family transcriptional regulator [Salmonella enterica]	Salmonella enter…	39.3	39.3	93%	3e-04	94.74%	89	EAS5284341.1
✓	TPA: tetracycline resistance transcriptional repressor TetR(B) [Salmonella enterica subsp. enterica serovar 4,[…	Salmonella enter…	39.3	39.3	93%	3e-04	94.74%	90	HAR9435601.1

b)

Figure 3: *(a) Blastx's input (b) Founded protein*
Finding a protein corresponded to mutations in blastx.

We found a protein from "TetR familly transcriptional regulator" which is resistant to a tetracycline (family of antibiotics to which bacteria have evolved resistance) [2]. "These proteins play an important role in conferring antibiotic resistance to large categories of bacterial species" [2].

8 Conclusion

Our research group faced a problem: Salmonella bacteria mutated and became resistant to tetracycline, which means it is resistant to antibiotics. Our goal was to find gene(s) that were responsible for that.

We developed a tool that finds mutations between resistant and wild type of bacteria, and which does it in linear time and space complexity. It allowed us to find involved protein TetR that are responsible for tetracycline resistance.

9 Library

[1]: Association mapping from sequencing reads using K-mers, 2018 (Atif Rahman, Ingileif Hallgrímsdóttir, Michael Eisen, Lior Pachter)

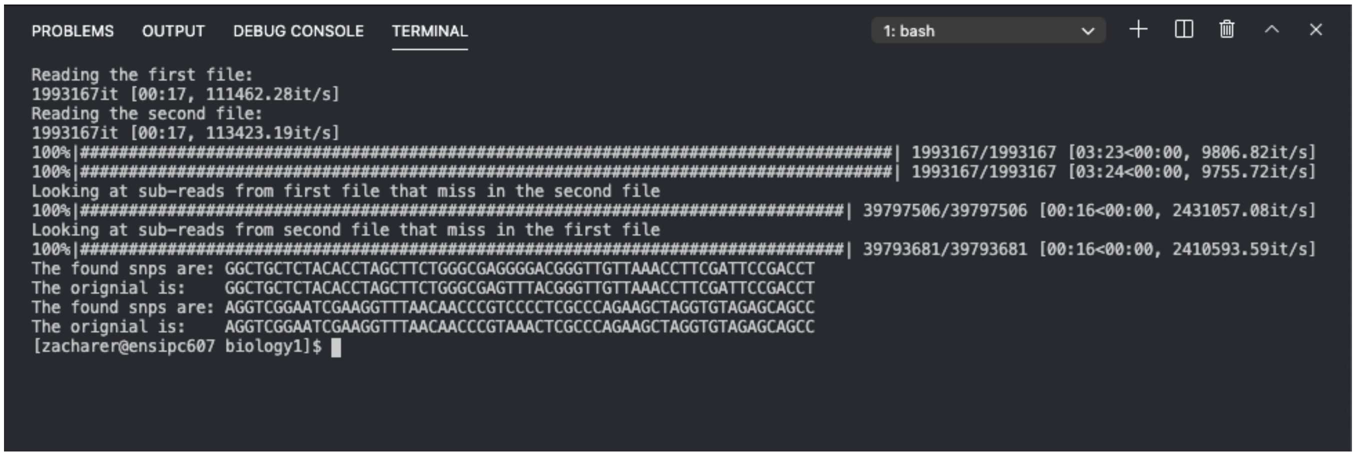
[2]: https://en.wikipedia.org/wiki/TetR

10 Appendix

The tool is written in the `snp_detect.py` file. In order to execute it, one needs to give it two FASTA files as command line arguments. An example of execution would be:

```
python3 snp_detect.py salmonella-enterica-variant.reads.fna
salmonella-enterica.reads.fna
```

The output would look like this:



```
Reading the first file:
1993167it [00:17, 111462.28it/s]
Reading the second file:
1993167it [00:17, 113423.19it/s]
100%|####################################################################################| 1993167/1993167 [03:23<00:00, 9806.82it/s]
100%|####################################################################################| 1993167/1993167 [03:24<00:00, 9755.72it/s]
Looking at sub-reads from first file that miss in the second file
100%|###############################################################################| 39797506/39797506 [00:16<00:00, 2431057.08it/s]
Looking at sub-reads from second file that miss in the first file
100%|###############################################################################| 39793681/39793681 [00:16<00:00, 2410593.59it/s]
The found snps are: GGCTGCTCTACACCTAGCTTCTGGGCGAGGGGACGGGTTGTTAAACCTTCGATTCCGACCT
The orignial is:    GGCTGCTCTACACCTAGCTTCTGGGCGAGTTTACGGGTTGTTAAACCTTCGATTCCGACCT
The found snps are: AGGTCGGAATCGAAGGTTTAACAACCCGTCCCCTCGCCCAGAAGCTAGGTGTAGAGCAGCC
The orignial is:    AGGTCGGAATCGAAGGTTTAACAACCCGTAAACTCGCCCAGAAGCTAGGTGTAGAGCAGCC
[zacharer@ensipc607 biology1]$ 
```