Algorithms for bioinformatics

TDT4287 - Project

Preprocessor for high throughput sequencing reads

Authors:
Rokas Bliudzius
Ola Kristoffer Hoff

1st November 2022

1 Introduction

The goal of this project is to develop a preprocessor for analyzing high throughput DNA sequencing libraries. One of the main goals of the preprocessor would be to identify and remove adapter fragments from the sequences.

The project itself consisted of 5 tasks, where each task consisted of creating a method or an algorithm for solving the task, plotting the results and discussing them.

In task 1 we had a given adapter sequence a and a data-set S of unknown sequences. The goal was to develop an algorithm for identifying all the sequences in S that contain suffixes that perfectly match a prefix of the adapter sequence a. Afterwards there were some questions that needed to be answered.

Task 2 was somewhat divided into two ”main” parts. First, we had to develop an algorithm for identifying all the sequences in S that contain suffixes that match a prefix of a, where this suffix can contain up to a given percentage of mismatches to the length of a. In the second part we had to essentially do the same thing, but now under the assumption that insertions and deletions (indels) are allowed. We created two different algorithms, one for each part.

In task 3 we had to estimate the rate of sequencing errors per sequence and per nucleotide by using the results from tasks 1 and 2. Thereafter, we had to explain and discuss our analyses.

Task 4 consisted of finding the adapter sequence, which in this case was unknown (unlike in task 1 and 2). This meant that we also had a new data-set to work with.

The last exercise, task 5, consisted of 3 smaller tasks. We had a new data-set again, and this one consisted of multiplexed sequences. Our task was to de-multiplex the barcoded library by first identifying the barcodes and use the information to find the number of samples that were multiplexed. Secondly, we had to identify how many sequences that were sequenced from each sample. And finally, we had to identify the sequence length distribution within each sample.

Due to time constraints we were not able to implement all the solutions as we wanted, so some solutions are sub-optimal, but run at good enough speeds.

All the results in this report were calculated and plotted using the same machine/system. See Attachments section 3.5 for technical specifications and practical runtimes for each of the tasks.

2 Methods

2.1 Task 1 - Perfectly matching adapter fragments

To find the perfect prefix-suffix match between a sequence $s \in S$ and a primer $a =$ "$TGGAATTCTCGGGTGCCAAGGAACTCCAGTCACACAGTGATCTCGTATGCCGTCTTCTGCTTG$". We could construct a suffix tree with the two strings. Then traverse the tree with a until we reach the leaf node representing the suffix a_0, which is the entire string a. Then we could check if the parent node p to the leaf node also has a leaf node for the sequence s. If so, the label of p would be the perfect prefix-suffix-match for a and s. However, if p has leaf node from s simply update p to p's own parent node and repeat the check. Do this until a match is found or you reach the root node, in which case there is no perfect match between a and s.

Using this method with Ukkonen's algorithm [1] we can construct the suffix tree in $O(|s| + |a|)$ time and then traverse it in $O(|2a|)$. So it is bound in linear time to the length of the sequence s and primer a.

We then need to do this for all sequences in S so we get a runtime of $O(|S| * |n|), |n| = |s| + |2a|, |S| >> |n| => O(|S|)$, hence the algorithm is linear bound by the number of sequences to process. Given that the condition $|S| >> |n|$ holds true, which is the case for our expected data.

We opted for using a simpler approach with a slightly worse practical runtime, shown in *algorithm 1*. Simply loop over the sequence s; if a matches from point i in s, $s[i]$, and continue until s ends then we have a match from i. If at any point $s[i + j] \neq a[j]$ then we search from $i = i + j$ and $j = 0$.

This will have a similar runtime as the other solution since we are iterating over s once, and over a each time, so it is $O(|s| * |a|)$. And since we need to do this for all $s \in S$ we get $O(|S| * |s| * |a|), |S| >> |s| * |a| => O(|S|)$. This is given the same logic as above.

It might seem that $|s| * |a| >> |s| + |2a|$, but for $|a|$ in the second method to get large, when it is not actually going to lead to a match, is very rare. This is given that we are trying to match some part of the sequence with the primer. To get one match is one in four; two, one in 16; three, one in 64 and so on. So the size of $|a|$ will in practice, for this type of data, be very limited by the statistical improbability of a random matching substring.

Algorithm 1 Perfect matching prefix-suffix

```
a ← adapter
𝕊 ← Sequences
S_0 ← {}, matching sequences
for s ← 𝕊 do
    Match ← false
    Offset ← 0
    for i ← 0; i < |s|; i ← i + 1 do
        if s[i] == a[Offset] then
            Offset ← Offset + 1
            Match ← true
        else
            Offset ← 0
            Match ← false
            Break
        end if
    end for
    if Match == false then
        Continue
    else
        S_0.append(s)
    end if
end for
return S_0
```

2.2 Task 2 - Imperfectly matching adapter fragments

Without indels

Given a number of mismatches based on percent of primer: $k = |a| * K\%$. First we look at matching without indels. This is then equivalent to the second method described above in the previous subsection. This is essentially finding all sequences with a hamming distance [2] lower than the number of mismatches. The only difference is that we do not reset $i = i + j$ and $j = 0$ on a mismatch. We increment the number of mismatches made, and if we have more mismatches then k, then we reset. Resetting is just equivalent to trying to match the primer from the next letter in the sequence. See *algorithm 2*.

Since this is the same method as above the same runtime is to be expected, only that $|a|$ is now allowed to grow more, so it will be slower, but is still bound by $O(|S|)$.

Algorithm 2 Imperfectly matching prefix-suffix, with k%-missmatches, and no indels

$a \leftarrow adapter$
$\mathbb{S} \leftarrow Sequences$
$S_0 \leftarrow \{\},\ matching\ sequences$
$K \leftarrow \% - missmatches$
$k \leftarrow |a| * K$
for $s \leftarrow \mathbb{S}$ **do**
  $Misses \leftarrow 0$
  $Match \leftarrow false$
  $Offset \leftarrow 0$
  for $i \leftarrow 0;\ i < |s|;\ i \leftarrow i + 1$ **do**
    if $s[i] == a[Offset]$ **then**
      $Offset \leftarrow Offset + 1$
      $Match \leftarrow true$
    else
      $Misses \leftarrow Misses + 1$
      if $Misses > k$ **then**
        $Offset \leftarrow 0$
        $Match \leftarrow false$
        $Break$
      end if
    end if
  end for
  if $Match == false$ **then**
    $Continue$
  else
    $S_0.append(s)$
  end if
end for
return S_0

With indels

When we take indels into consideration we cannot do this the same way, then we have to perform and ED (Edit Distance) in the form of DP (Dynamic Programming). We also apply the constraint of not allowing the solution to be in cells k-distance away from the main diagonal, since these will have violated the number of mismatches allowed. This constraint allows us to ignore these cells and gives us less computation. See *algorithm 2*.

To solve the prefix-suffix-problem with k-mismatches as an ED problem we first need to set the initial matrix. This is shown in table 1. First, since we are finding the prefix-suffix-match we must start in column 1 (first G, indicated with green) to get the prefix requirement, hence have a zero cost there, and minus infinity in the others. Since we can match the start to anywhere in the sequence s col 0 is just zeros. However, to fulfill the suffix requirement we must end our path in the last row. We can also set the whole upper-right ”corner-area” to minus infinity, since this is the area that is too far away from the main diagonal to be reached with k-mismatches.

The optimal solution is found if we have matches along the main diagonal or end up as far to the right in the last row as possible (note that if $|a| > |s| + k$ then the upper right cut out will reach the last row before the last column). In this case we must start where the diagonal cut-off intersects the last row (or one to the left to it, to be precise). We start in the bottom right (or left of the diagonal cut-off) and do a

backtrack check. If there is no valid solution, do a backtrack from the cell to the left check. Continue until the row ends or a solution is found.

In the case of table 1 we see that we have matched GGC with AGC to a score of -1, and since $|-1| <= k$ it is a valid solution (path marked in orange).

There is an optimization that can be done in the case where $|a| < |s|$. Since the solution ends in the last row there is a max amount of "height" up the table that can be backtracked. First, if we have a perfectly matching diagonal we gain $|a|-1$ height. Then we can use all of our mismatches to go up k more rows. This gives us that the highest point (lowest row value) a valid solution can start in is at $x = |s| - (|a| - 1) - k$. If $x > 1$ then we don't need to calculate the rows above row x. This means we can set our start row to be equal to $max(1,\ x)$. Then if the start row is $x > 1$, we set the row $x-1$ to hold values of minus infinity, as we did with row zero initially, since we can not have a solution come from that row. This because we know that a solution can never originate from that row or above. This is represented by the red line (with values of minus infinity) in row three in table 1. The highest/earliest start row value in column one is marked blue.

One final optimization we used was the observation that after we have added a new row and/or column, we can check if we need to do any more work. This is because the newly filled in row and/or column contains all the possible values the next rows and columns will be based on. Since these values cannot increase (but can only stay the same), it means that no values in the rest of the matrix will be better than the best value of the current row and column. Therefore, after adding a new row and column, we check if the best value in them is beyond the bound of k-mismatches. If so, the algorithm can stop calculating.

+	¢	G	G	C
¢	0	0	-∞	-∞
C	0	-1	-1	-∞
T	0	-1	-2	-2
C	0	-∞	-∞	-∞
T	0	-1	-2	-3
A	0	-1	-2	-3
G	0	0	-1	-2
C	0	-1	-1	-1

Table 1: Table which illustrates the initial conditions for the prefix-suffix-match with k-mismatches as ED matrix. Here the values are: $a = GGC,\ s = CTCTAGC,\ k = 1$. A cost function of -1 is used for indels and substitutions, and 0 cost for a match.

This solution is bound by $O(|S| * |s| * |a|)$. Moreover, in this case $|a|$ will not have very low values as with other cases, so the runtime will increase quite a bit compared with the earlier solutions with the same bound.

Algorithm 3 Imperfectly matching prefix-suffix, with k%-missmatches, and indels

$a \leftarrow adapter$
$\mathbb{S} \leftarrow Sequences$
$S_0 \leftarrow \{\},\ matching\ sequences$
$K \leftarrow \% - missmatches$
$k \leftarrow |a| * K$
for $s \leftarrow \mathbb{S}$ **do**
    $Matrix, StartColumn \leftarrow ED(a, s, k)$
    $Row \leftarrow Matrix.rows - 1$
    for $Col \leftarrow StartColumn;\ Col > k;\ Col \leftarrow Col - 1$ **do**
        if $Matrix[Row, Col] > k$ **then**
            $Continue$
        else
            $Suffix \leftarrow BackTrack(Matrix, Row, Col, a, s)$
            $S_0.append(Suffix)$
            $Break$
        end if

    end for
    if $Match == false$ **then**
        $Continue$
    else
        $S_0.append(s)$
    end if
end for
return S_0

2.3 Task 3 - Sequencing errors and error distributions

To get an estimate for the rate of sequencing errors per sequence and nucleotide we must base our estimate on something. It would take too much time to check the actual answer, hence why we need to estimate it. We can use the data from task 1 and 2 to answer the questions for the adapter part of the sequence. These numbers should represent the whole set fairly well.

First, we can fit the data that we have into some models and pick one with the best results. This can be done using *Least Squares Method* for over-fitting data to a model. The data can be visualized by using a scatter plot, where the axis represent percentage of mismatches allowed to percentage of matched sequences in the set.

Based on the best model we can estimate how many percent mismatches will lead to the entire set being matched.

Then we can calculate the rate by finding the average of error percentage allowed per part of the set.

$rate = \frac{1}{n} \sum_{i=1}^{n} a_i - a_{i-1}, a_i = \int_0^i model, model(n) = 100\%$

To find the rate and distribution of errors per nucleotides we need to find the hamming distance (number of substitutions) from each sequence to the actual sequence. However, we do not have the actual sequence. What we do have is the adapter a. So we can look at the adapter part of the sequence to estimate the rate and distribution for the whole sequence. We can line up the suffix of the sequence and the adapter, as in task 2, and then check where the errors occur between the suffix and the prefix

of the adapter.

This may be a bit biased when we analyse if errors occur more often at the start and/or the end of sequences rather than the middle. This is because the start of the sequence we analyse, $pre - suf$ (the prefix-suffix-pair), is a connection between the end of the sequence s and the start of the adapter sequence a, not the start of the whole sequence s. However, if we have a uniform distribution, it can be fair to assume that it extends to the whole sequence.

2.4 Task 4 - Finding the adapter sequence

Finding the most probable adapter in a data-set can be done with a keyword tree structure [2]. When constructing the tree, each node will have a counter that is incremented each time the algorithm passes through the node with a new sequence. The counter will then hold the value of the amount of unique sequences that have passed through the node, meaning the amount of sequences in the data-set that have the same suffix up until the node.

Since we are looking for adapter sequences, the trick is to reverse the sequences before filling the keyword tree. This makes it so that the back of the adapter sequence is directly connected to the root. We expect the actual adapter to be more present than mutations of it. Therefore, we can traverse the tree by comparing counters and going to the child node that has the greatest amount of unique sequence paths through it. However, we also need to stop before the algorithm reaches the bottom of the tree and has found the most common sequence in the set. Therefore, we need a threshold value to stop at. The threshold value would represent a ”guess” of lowest percentage of sequences with the same adapter/suffix up until the node. If we assume that the sequences with the same adapter are normally distributed (Gaussian Distribution[3]), a good guess for the threshold would be 95%, since it represents the amount of values within 2 standard deviations in a Normal Distribution.

However, we do not know exactly where the adapter prefix overlaps with the suffix of the sequence. Since we are estimating and finding the most probable adapter, it is prone to having more errors that of random mutation. A neat way of solving this could be to start traversing the tree by choosing the most frequently passed child node. This would give us the most frequent sequence. Then one could add a threshold value, to say, that the most pass child node also must have k% of the passes through the parent node. This controls how dominant the path must be. Then one could plot the solution where k goes from zero to a hundred percent. With a little analysis it should not be hard to spot where the adapter is found.

Our approach was to use trail and error to find a value for k that gave reasonable answers.

Moving on with the traversal, if no child node has the threshold value (or greater) of the unique sequence path counter from the current node, we stop the traversal and return the node’s suffix value (up until that node) as the adapter sequence. It’s expected that when the adapter sequence stops, the values from the actual sequences would be more equally distributed than the adapter sequence.

The algorithm for this can be seen in *algorithm 4*.

Since this is a keyword tree, we can build it in $O(n), n = \sum_{i=1}^{n} |s_i|, s \in S$. Then after the construction, we need to find our solution. This is achieved by a simple traversal of the tree which is done in $O(max(|s|)), s \in S$ or relative to n: $O(log_k(n)), k = |\gamma|$ $(the\ alphabet/number\ of\ characters)$.

Algorithm 4 Finding most probable adapter sequence

$a \leftarrow$ "" $(adapter)$
$\mathbb{S} \leftarrow Sequences$
$k \leftarrow \% - certainty$
$KeywordTree \leftarrow \{\}$
for $s \leftarrow \mathbb{S}$ **do**
  $KeywordTree.append(s.reverse())$
end for

$CurrentNode \leftarrow KeywordTree.Root$
while $CurrentNode\ not\ leaf$ **do**
  $CurrentNode \leftarrow max(CurrentNode.Children.Counter)$ //Find child node with most passes
  if $\frac{CurrentNode.Counter}{CurrenNode.Parent.Counter} \leq k$ **then**
    $Break$
  end if
  $a.append(CurrentNode.Label)$
end while
return $a.reverse()$

2.5 Task 5 - De-multiplex barcoded library

First we must approximate the adapter and remove it. We need to allow for some mismatches because of the filling at the end of the sequence. This is achieved by using the keyword tree from task 4, but only checking the first few depth levels. These levels are somewhat random, but at a certain depth the adapter starts and it becomes very clear which path to follow. One could also check this depth and remove the corresponding length off the back of each sequence, as a rough estimate to remove the filling. Methods to find very ”hot trails” in the keyword tree is also possible, where the trail indicates that the adapter has begun.

Once the adapter is approximated, it can be aligned and removed from the sequences. Based on the observations from earlier data, and the nature of random mutations, we allow a few mismatches in the alignment. We must have a good enough alignment, so that we can get a subset of sequences which have fewer mismatches than the rest.

With the subset of sequences left, we can plot the frequency of the permutation of the last letters, the barcode, and see which occurs the most and from that we have found the barcodes.

With the set of barcodes we can separate the sequences into samples. This is done by simply finding out which barcode the end of the sequence has the closest hamming-distance or edit-distance too.

Once the samples are separated we can remove the barcode from them, then we can, as done earlier, plot the length distribution.

Finally, to find the most frequently occurring sequence within each of the samples,

build a keyword tree with the sample set and follow the most ”walked” path, as done earlier.

Algorithm 5 De-multiplexing sequence set into most probable barcodes

$\mathbb{S} \leftarrow Sequences$
$k_0 \leftarrow \% - certainty\ of\ adapter$
$k_1 \leftarrow \% - certainty\ matching\ adapter$
$a \leftarrow Algorithm4(\mathbb{S}, k_0)$
$S \leftarrow Algortihm2(\mathbb{S}, a, k_1)$
$BarcodeFreqDist \leftarrow \{\}$
$LenghtOfBarcode \leftarrow 8$
for $s \leftarrow S$ **do**
    $Barcode \leftarrow s[|s| - LengthOfBarcode,\ |s|]$
    $BarcodeFreqDist[Barcode] \leftarrow BarcodeFreqDist[Barcode] + 1$
end for
$Plot(BarcodeFreqDist)$
$CutOff \leftarrow UserInput$ //User value after analysing the barcode frequency distribution
$BarcodeFreqDist.reverse\_sort()$
$Samples \leftarrow \{\}$
for $Barcode \leftarrow BarcodeFreqDist.pop\_back();\ Barcode.Freq >= CutOff;\ Barcode = BarcodeFreqDist.pop\_back()$ **do**
    $Sample \leftarrow \{\}$
    for $s \leftarrow S$ **do**
        if $HammingDistance(Barcode,\ s[|s| - LengthOfBarcode,\ |s|]) <= 2$ **then**
            $Sample.append(s[0, |s| - LengthOfBarcode])$
        end if
    end for
    $Samples.append(Sample)$
end for
return $Samples$

3 Results and Discussion

3.1 Task 1 - Perfectly matching adapter fragments

From the one million sequences in set S (*s_3_sequence_1M.txt*) we found that there were 538691 matches, see figure 1, however this data is not clean. As we see it has a spike in the left and right end. The left spike represents all the excess primers that have not been bound to a sequence. This is the case since they have a perfect match, given that the sequence has a length of zero after removing the match. We have therefore removed them from our results. Regarding the right most spike in occurrences, this too is not representable for the data. When we have a sequence that ends in one of four possible letters, and the primer starts with one of these four, it is by random a one in four chance that we get a prefix-suffix-match of length one. Similarly, for matching two gives a random chance of a match to one in 16. Therefore we have also removed all matches of length three or less. After these measures we have the data as shown in figure 2. Now we have 191760 matches, almost 20% of the set S.

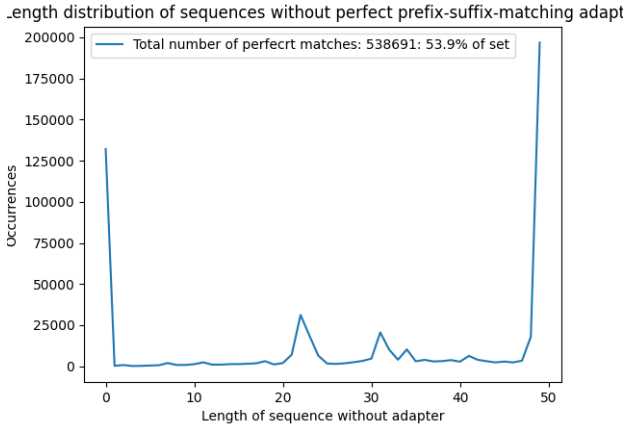

Figure 1: Length distribution of sequences with perfect prefix-suffix-match, after removal of adapter part.

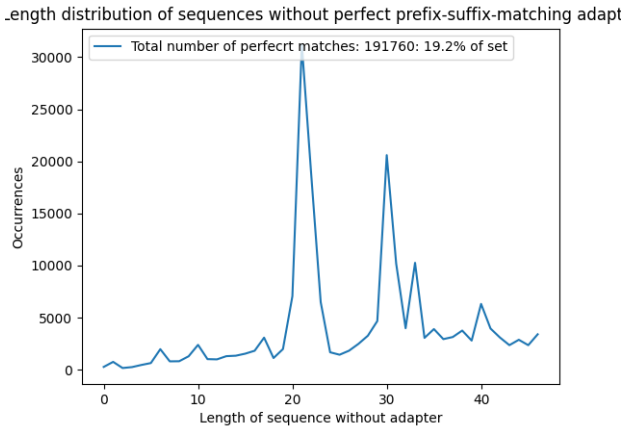

Figure 2: Length distribution of sequences with perfect prefix-suffix-match, after removal of adapter part. Here the perfect overlaps has been removed, and also matches of length three or less.

3.2 Task 2 - Imperfectly matching adapter fragments

Without indels

We can see the unfiltered results for 10% and 25% respectively in figure 3 and 5. The filtered/edited data is shown in figure 4 and 6.

As we can see we have the same bipolar spike problem with the un-edited data as in the previous subsection. However, for slightly different reasons this time. As we can see, in figure 3 and 5, they state that with k-mismatches (6 and 15) we have found imperfect prefix-suffix-matches for 100% of the sequences in S. This is not exactly true.

The left spike is the same as described earlier, and is dealt with in the same manner.

What happened in the right spike is that if we try to match a prefix of a with a suffix of s, and the overlapping sequence is less than the number of allowed mismatches, then we can simply have substitutions for the whole overlap. That is why we have 100% of the set.

To fix this, we only count a match if it has at least three actual matches in the overlap. The result of the filtering is shown in figure 4 and 6.

Now we can see that for 10% and 25% we have respectively matched 552023 and 852139 sequences, or 55% and 85% of all the sequences in S.

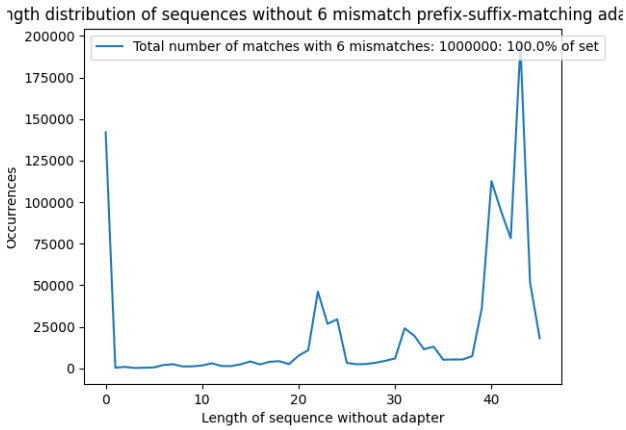

Figure 3: Length distribution of sequences with prefix-suffix-match, with 10% of primer a mismatches, after removal of adapter part.

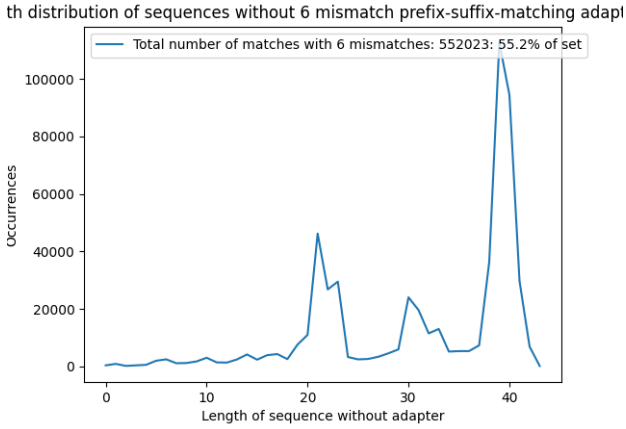

Figure 4: Length distribution of sequences with prefix-suffix-match, with 10% of primer a mismatches, after removal of adapter part. Here the perfect overlaps have been removed, and also matches of length three or less exact matches.

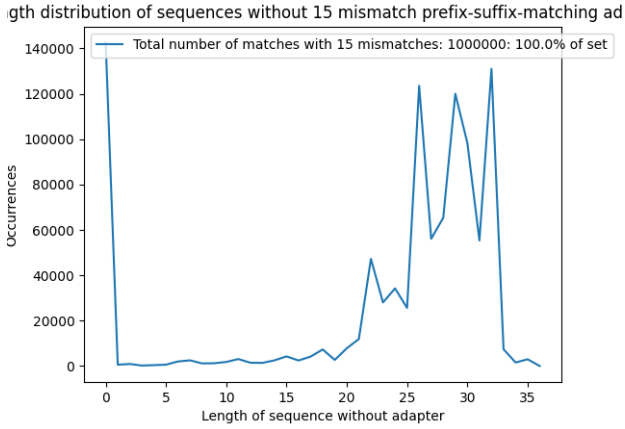

Figure 5: Length distribution of sequences with prefix-suffix-match, with 25% of primer a mismatches, after removal of adapter part.

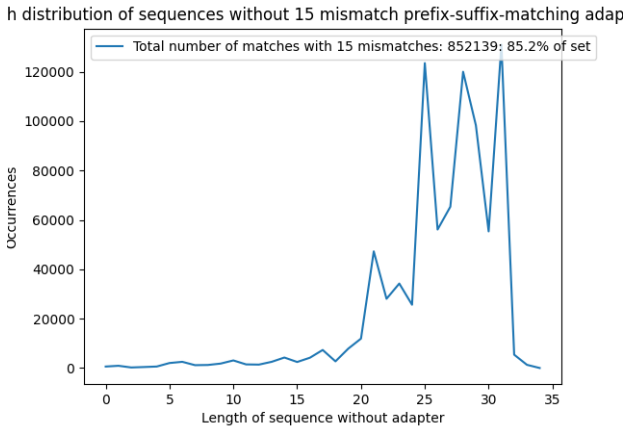

Figure 6: Length distribution of sequences with prefix-suffix-match, with 25% of primer a mismatches, after removal of adapter part. Here the perfect overlaps have been removed, and also matches of length three or less exact matches.

With indels

Not surprisingly we had the same data issues as before, so we cleaned the data as we did above. The clean data results can be seen in figure 7 and 8. 10% and 25% mismatches now give us matches for 780359 (78%) and 853574 (85%) of the total set, S, respectively. In contrast to the results without indels, the 10%-mismatch results has gone up quite a bit, but the 25%-mismatch results are essentially the same.

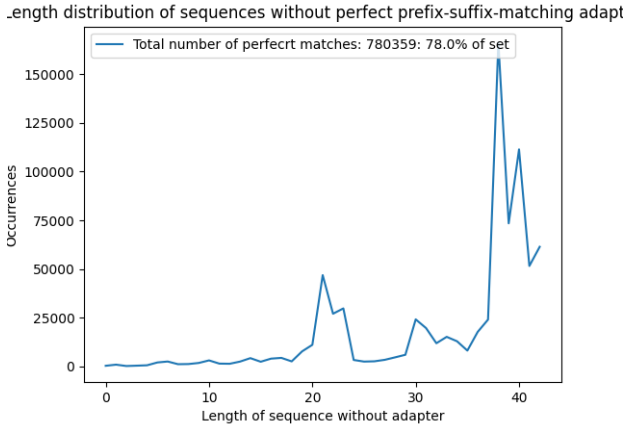

Figure 7: Length distribution of sequences with prefix-suffix-match, with 10% of primer a mismatches, and indels, after removal of adapter part. Here the perfect overlaps have been removed, and also matches of length three or less exact matches.

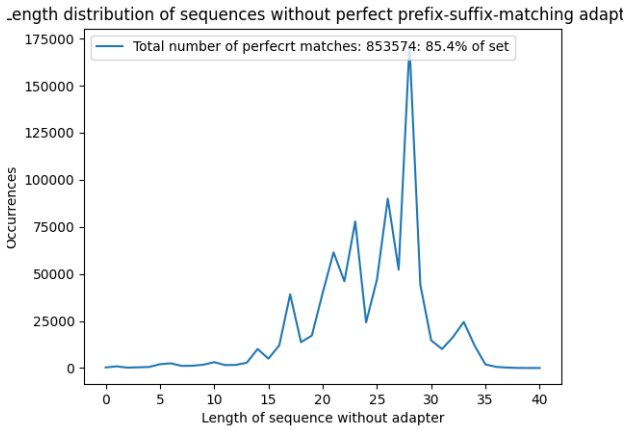

Figure 8: Length distribution of sequences with prefix-suffix-match, with 25% of primer a mismatches, and indels, after removal of adapter part. Here the perfect overlaps have been removed, and also matches of length three or less exact matches.

3.3 Task 3 - Sequencing errors and error distributions

After looking at the scatter plot of the data from task 1 and 2 we tried fitting the data to a linear and a logarithmic model, as shown in figure 9 and 10.

It is clear by the RMSE (Root Mean Square Error)[3] value that the logarithmic model is better. We can then also note that the rate calculated here was $\sim$20%, which means every fifth sequence has no errors. This makes a lot of sense given that it corresponds to the number of perfect matching prefix-suffix-pairs in the set (seen in results from task 1). So the sequencing error rate per sequence is $\sim$80%.

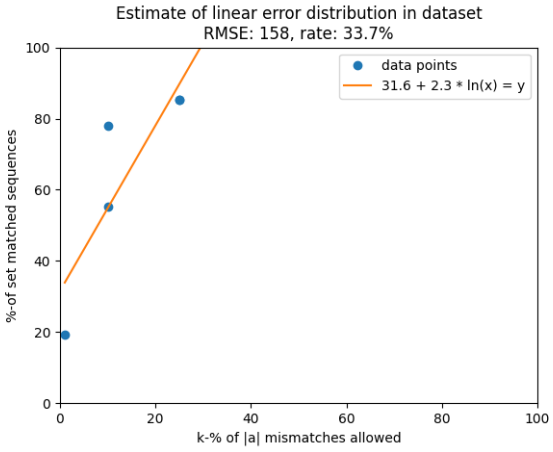

Figure 9: The data from task 1 and 2 fitted to a linear model: $a + bx = y$.

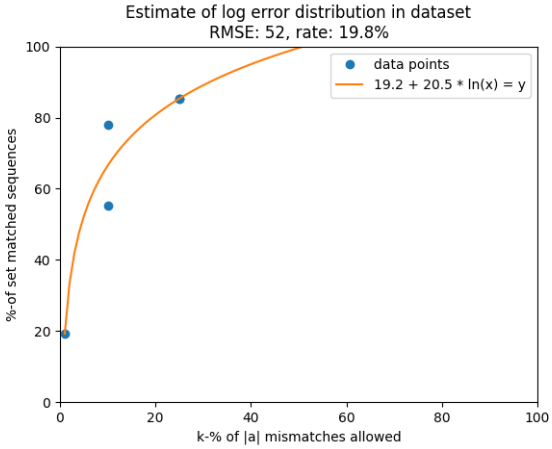

Figure 10: The data from task 1 and 2 fitted to a logarithmic model: $a+b*ln(x*c) = y$.

In figure 11 and 12 we can see the distribution of sequencing error per nucleotide. The rate of errors was 30.7% and 43.9% respectively.

One can observe that the distributions are not uniform. However, we now (as in all previous cases) run into the same problem with barely matching sequences. We have illustrated the cut-off we have used earlier with an orange vertical line in the plots. As one can see, the distributions are fairly uniform on the right hand side of the vertical line. The line represents that all sequences on the right must have at least three actual matches with the adapter. However, still every $\frac{1}{4^4}$ (one in 256) sequences will by chance have a match, and with one million sequences we could increase this threshold to $\frac{1}{4^x} = \frac{1}{1.000.000} = 4^x = 1.000.000 => x = log_4(1.000.000) \approx 9,97 => 10$. Setting a cut-off here means that there by chance will be one sequence in the whole set that can match the sequence by random. This line is the green vertical line. This line gives us an even better idea that it is uniformly distributed. However, we can see in the 25% plot, figure 12, that we still have a little more on the left side. This could be due to the larger k value. As we see the 25% plot is pushed more towards the middle whilst the 10% plot stays in about the first quarter.

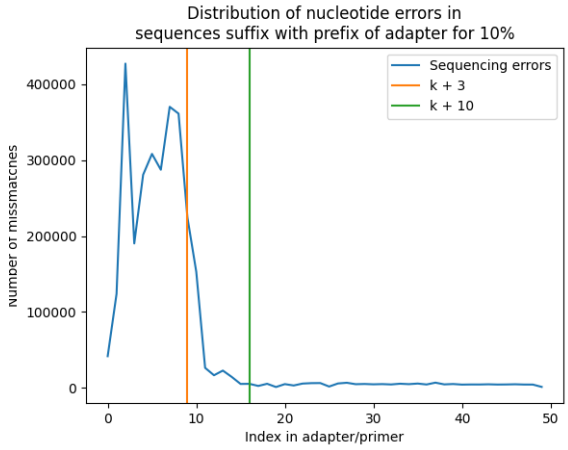

Figure 11: Distribution of nucleotide errors in the sequences. Based on the prefix-suffix-match with the adapter at 10%-mismatch rate.

Figure 12: Distribution of nucleotide errors in the sequences. Based on the prefix-suffix-match with the adapter at 25%-mismatch rate.

3.4 Task 4 – Finding the adapter sequence

With a boundary constraint of $k = 0.7$, meaning that to traverse further down the keyword tree a child node must have at least 70% of the paths through the current node. This constraint got that the most probable adapter sequence was: *"AGATCGGAAGAGCACACGTCTGAACTCCAGTCACGTAGAGATCTCGTATGCCGTCTTCTGCTTGAA"*.

In figure 13 we can see the length distribution of the sequences after removing the adapter.

As we can tell by the spike in the middle of the graph, there is a sequence length of 21 that is drastically more frequent that the others. Note also the smaller spike at length 32.

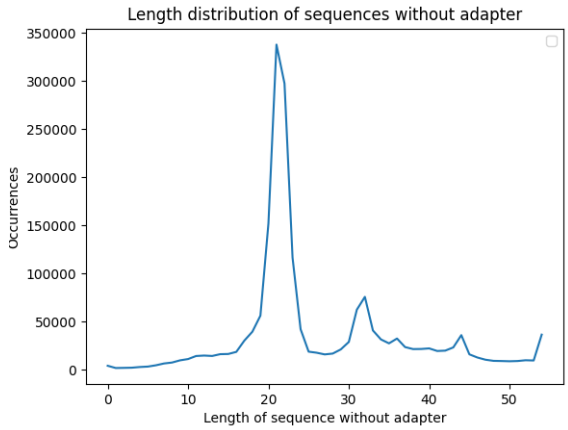

Figure 13: Length distribution of sequences in set: *"tdt4287-unknown-adapter.txt.gz"* after removing most probable adapter.

In figure 14 we can see the frequency distribution of the unique sequences after removing the probable adapter. There was no strong evidence of any common suffix between these. Once the parameter for certainty got low enough for a suffix to be chosen, then the suffix was a whole sequence. Probably the most common sequence,

as we can see in the figure there is one that has ∼35.000 occurrences.

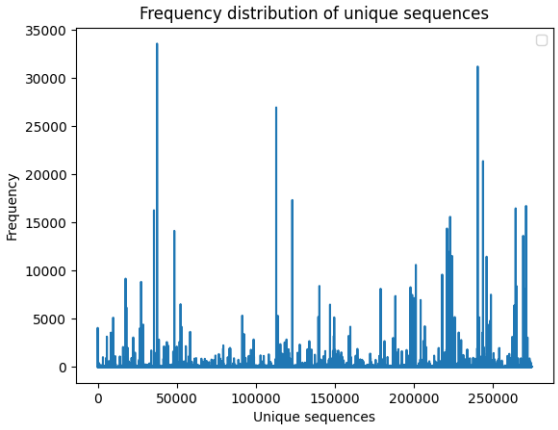

Figure 14: Frequency distribution of unique sequences in set: *"tdt4287-unknown-adapter.txt.gz"* after removing most probable adapter.

There was no statistical proof for finding something interesting in cases for *"s_3_sequence_1M.txt"* and *"seqset3.txt"* data-sets. We thought that we would find something resembling the adapter that we know from task 1 and 2. However, there was too much variety in the data, so no path was obvious from the root. Upon closer inspection we saw that the actual adapter only had about 38% of the total paths from the root, whilst the second best had 23%. Now this could be followed with some other rules for the stop criteria. However, it is very difficult to allow for such small differences given that we have some sequences that occur more often than others, and that could have the same results. This would then merge with our stop criteria which leads to the a whole sequence being presented as the best adapter. Which is what we observed.

Both the size of the data-sets, and the ratio between sequence length and adapter length can have effects on the results here.

3.5 Task 5 - De-multiplex barcoded library

In the table, 2, below we can see that we identified 6 samples. These barcodes was identified from the barcode frequency distribution in figure 15. Furthermore, the table shows the barcodes identified and their frequency of occurrences, and the percent of the original sequence set they connect to. These map ∼57% of all the sequences, which considering the rate of errors is quite similar to the results of task 2, where with 10% mismatches allowed (without indels) it matched 55% of the sequence set. This rate is of course not exactly the same, but it gives this answer some confidence that it is in the same ballpark.

In the fifth column it is shown the estimated number of sequences belonging to each sample. These are found by scaling the percent of sequences from S they map (column four) by the factor of which they all represent the total set. For the first barcode this is: $\frac{12.13\%}{57\%} = 21.28\%$. Hence, this gives us the last column (six), the estimate number of sequences per sample, which is the percent times the total sequence count (10.687.775). Note that the second and the fifth barcode can probably be ruled out due to their insignificant size, compared with the others.

Sample	Barcode	Occurrences	% of sequence in S	Scaled % of total sequences	Estimate sequence count
0	GTATGCCG	1.182.049	12.13%	21.28%	2.274.359
1	ATCGTATG	454.951	11.00%	19.30%	2.062.741
2	TTCGTATG	416.631	2.08%	3.65%	390.104
3	GTTTATCA	333.602	25.09%	44.02%	4.704.759
4	TGATATCA	250.996	6.84%	12.00%	1.282.533
5	CTCGTATG	229.797	0.27%	0.47%	50.233
Total:		2.868.026	57.41%	100%	10.764.729

Table 2: Table which shows the 6 barcodes identified, with their number of occurrences in the frequency distribution and the percentage of the original sequence set mapped, with a hamming distance of maximum 3. The last column shows the estimated number of sequences per sample.

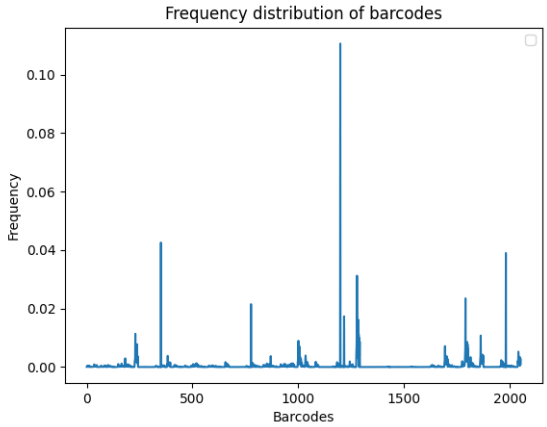

Figure 15: Frequency distribution of barcodes in the set.

Below, in figure 16 to 21, we can see the length distribution within each sample and the most frequent sequence in the given sample.

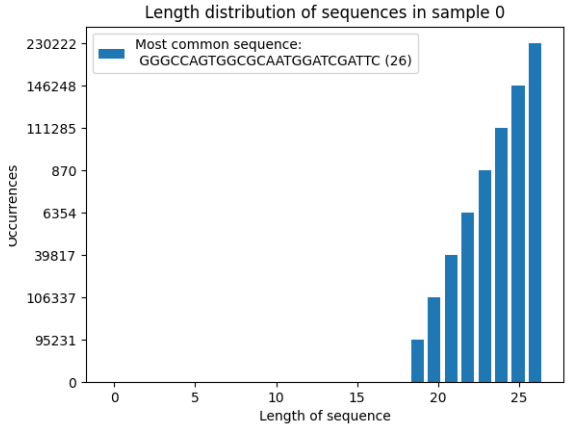

Figure 16: Length distribution and most frequent sequence in sample 0.

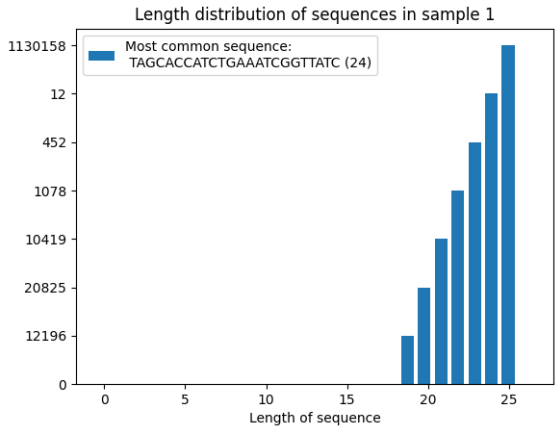

Figure 17: Length distribution and most frequent sequence in sample 1.

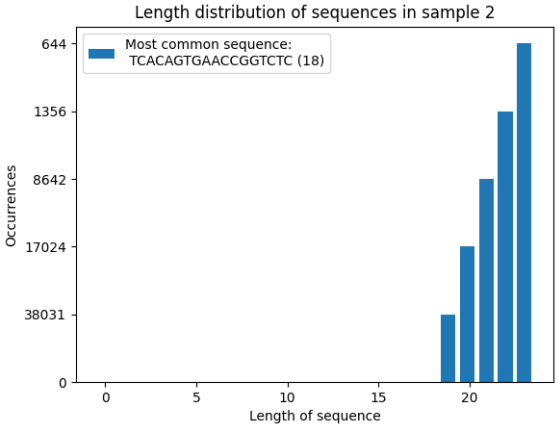

Figure 18: Length distribution and most frequent sequence in sample 2.

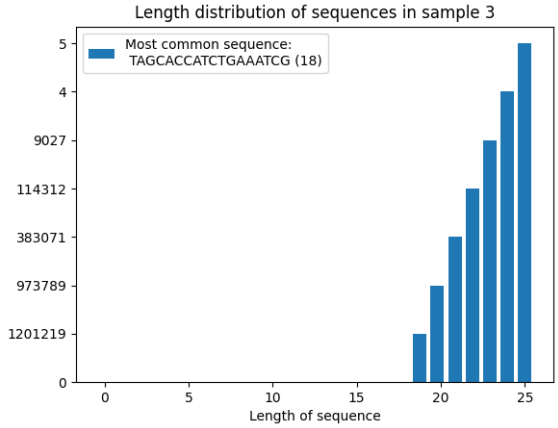

Figure 19: Length distribution and most frequent sequence in sample 3.

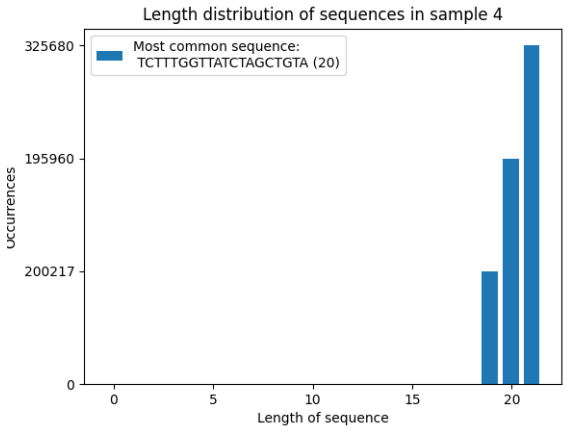

Figure 20: Length distribution and most frequent sequence in sample 4.

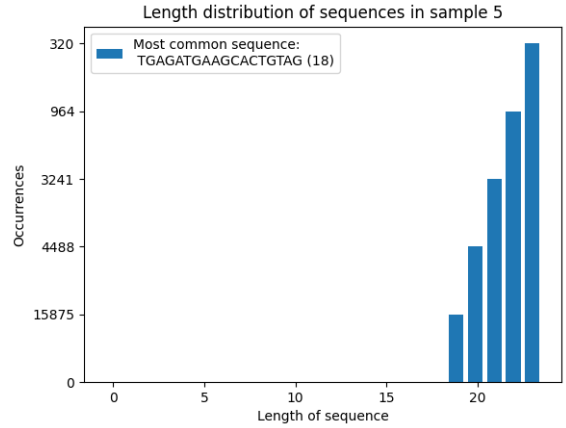

Figure 21: Length distribution and most frequent sequence in sample 5.

Referanser

[1] E. Ukkonen, 'Algorithms for approximate string matching', *Information and Control*, vol. 64, no. 1, pp. 100–118, 1985, International Conference on Foundations of Computation Theory, issn: 0019-9958. doi: https://doi.org/10.1016/S0019-9958(85)80046-2. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0019995885800462.

[2] N. C. Jones and P. A. Pevzner, *An Introduction to Bioinformatics Algorithms*. MIT Press, 2004, isbn: 9780262101066.

[3] G. G. Løvås, *Statistikk for universiteter og høgskoler*, 4th ed. Universitetsforlaget, 2018, isbn: 9788215031040.

Attachments

Task	Language	Runtime
1	C++	0.76s
2 (no indel) (10%/20%)	Python	66.10s/106.38s
2 (w/indel) (10%/20%)	C++	7.50s/11.66s
3	Human analysis	Days
4	C++	17.05s
5	C++	57.34s

Table 3: The different runtimes of the tasks. All running has been done on the same system, for consistency: MacBook Pro (13-inch, 2019, Four Thunderbolt 3 ports); Processor: 2,4 GHz Quad-Core Intel Core i5; Memory: 8 GB 2133 MHz LPDDR3; Graphics: Intel Iris Plus Graphics 655 1536 MB.