Karlsruhe Institute of Technology

Error-Profile-Aware Correction of Next Generation Sequencing Reads

Master Thesis of

Sarah Lutteropp

At the Department of Informatics
Institute of Theoretical Computer Science

Reviewer:	Prof. Dr. Alexandros Stamatakis
	Prof. Dr. Peter Sanders
Advisor:	Dr. Tomas Flouri

Time Period: 01st October 2016 – 31st March 2017

Statement of Authorship

I hereby declare that this document has been composed by myself and describes my own work, unless otherwise acknowledged in the text.

Heidelberg, 31st March 2017

Abstract

In this thesis we develop an error-correction-toolkit for de novo (without a reference genome) DNA sequencing which takes into account technology-specific error profile and coverage bias information. By splitting error profile learning, G/C coverage bias correction and k-mer classification into separate modules, the error correction algorithm itself can remain technology-agnostic. We develop multiple variants for each of these modules.

We evaluate the performance of each module separately by isolating it from the remaining framework modules. The separate isolation shows that especially the coverage bias estimation, the k-mer classification, the context-free error profile estimation (which infers only overall error probabilities) and the sequence-specific error profile estimation (which infers the influence of short sequences up to 6 bases surrounding an erroneous position in a read) already show satisfying results. However, our machine learning variant for estimating the full-context-specific error profile (which takes the complete information into account, this is, the current base in the read, its quality score, the read length, and surrounding bases) still requires further improvements.

We implement an example error correction algorithm for showing how to combine the modules provided by our framework. We briefly compare its performance with current state-of-the-art error correction tools on genome re-sequencing datasets from Illumina and PacBio sequencers, using both simulated and empirical data. Our comparison shows that our example error correction algorithm still demands further work before being able to compete with current state-of-the-art error correction approaches.

Deutsche Zusammenfassung

Diese Masterarbeit handelt von der Korrektur von de novo (d.h., ohne ein bekanntes Referenzgenom) DNA-Sequenzdaten unter Berücksichtigung von technologiespezifischen Fehlerprofilen. Durch die Aufteilung des dazu nötigen Programms in separate, untereinander austauschbare Module wird u. a. ermöglicht, dass der Fehlerkorrektur-Algorithmus leichter benutzbar und erweiterbar ist. Wir entwickeln mehrere Varianten für die jeweiligen Module unseres Frameworks und entwickeln einen beispielhaften Korrekturalgorithmus, um das Zusammenspiel der einzelnen Frameworkmodule zu zeigen. Wir evaluieren die einzelnen Module des resultierenden Fehlerkorrekturframeworks, wobei wir sowohl reale Datensätze von Illumina und Pacific Biosciences als auch simulierte Datensätze betrachten.

Unsere Experimente zeigen, dass die meisten Frameworkmodule zufriedenstellende Resultate zeigen. Jedoch benötigen insbesondere der hier entwickelte Fehlerkorrektur-Algorithmus sowie der hier entwickelte Maschinelles-Lernen-Ansatz zur Erfassung des gesamten technologiespezifischen Fehlerprofils weiteren Entwicklungsaufwand, um konkurrenzfähig zu sein.

Acknowledgements

I want to thank Prof. Dr. Alexandros Stamatakis for giving me the freedom to define a Master thesis topic on my own and try out very risky approaches. And for providing constant motivation and encouragement whenever I thought this project came to a dead end. I also want to thank him a lot for proof-reading and correcting my thesis during his paternity leave.

I am grateful to Prof. Nick Goldman who invited me to visit his group at EBI Hinxton for one month. I also want to thank the Klaus Tschira Foundation who made this visit possible.

I want to thank Prof. Sanders for being the second reviewer of my Master thesis.

I further want to thank all these people (in no particular order) for great and helpful discussions throughout my work on the thesis:

Pierre Barbera, Johannes Bechberger, Vladimir Benes, Jane Charlesworth, William Coleman-Smith, Lucas Czech, Diego Darriba, Mat Davis, Tomas Fitzgerald, Tomas Flouri, Paschalia Kapli, Alexey Kozlov, Khouloud Madhbouh, John Marioni, Tim Massingham, Iain Moal, Benoit Morel, Tobias Rausch, Eric Rivals, Jossy Sayir, Dora Serdari, Asif Tamuri, Kevin Zerr.

I want to thank my parents for their huge financial and emotional support throughout my studies. I also want to thank Moritz von Looz. Thank you very much for being an excellent burnout-protection. ;-) And for proof-reading some of the chapters.

Contents

1. Introduction

1.1. Motivation

Deoxyribonucleic acid (DNA) is the blueprint of life. Knowing the genetic sequence of a living being has many benefits and applications. Example applications include forensics such as combating the illegal trade in African elephant ivory [WJCD+08], building the tree of life [Sta06], categorizing living things into different species based on their DNA [KLZ+17] and improved understanding of genetic diseases [KC91]. Only since the late 1970's, people are able to obtain the DNA sequence of an organism by a process called *DNA sequencing*.

Genome sizes of different species vary significantly. An organism with one of the smallest known genomes is a RNA virus called *bacteriophage MS2* which infects bacteria. Its genome consists of only 3,569 base pairs and it was the first completely sequenced genome in 1976 [FCD+76]. The bacterium *Escherichia coli K-12* has a genome size of 4,639,221 base pairs [BPB+97]. In contrast, the human genome has 3,234,830,000 base pairs per haploid genome [VAM+01]. Note that genome size is not directly linked to complexity. An organism with one of the largest known genome sizes is *Paris japonica*, a flower with a genome of about 150,000,000,000 base pairs [PFL10] which is nearly 50 times larger than the human genome.

Current sequencing methods cannot directly infer the DNA sequence of a full-length genome. Instead, the DNA is split into many shorter pieces and each of these pieces is processed separately. By sequencing these shorter pieces of DNA, we obtain so-called DNA sequencing *reads*. Depending on the sequencing technology used, a read can contain 150 bases (Illumina NextSeq) [Ill17a] or up to 100,000 bases (Oxford Nanopore MinION) [LHO+15]. In order to reconstruct the genome, millions of reads are stitched together in a process called *genome assembly*. Ideally, one would assemble a genome by creating a multiple sequence alignment of all reads. Unfortunately, the task of creating an optimal multiple sequence alignment is known to be $\mathcal{NP}$-hard for most optimality criteria [WJ94] [Eli06]. Thus, genome assemblers rely on heuristic methods. As the process of assembling a genome is already very hard, errors in reads yield obtaining a good assembly even more difficult.

Back in 2001, the cost of sequencing a human genome was estimated to be around $100,000,000. Due to advances in the field of Genomics, the cost for sequencing a human genome dropped to slightly above $1,000 in 2015 [Ins17]. This progress has been achieved by introducing new sequencing technologies (Next Generation Sequencing and Third Generation Sequencing) which have a higher throughput due to their massive sequencing

parallelism, leading to a smaller per-base sequencing cost. However, they also produce reads with a higher error rate, thus increasing the need for appropriate error correction methods.

Different sequencing technologies have different error characteristics. To complicate the matter further, there are notable differences even between two sequencing runs using the same machine. For instance, companies that produce sequencing machines frequently modify the chemistry and usage protocols. Also, many library preparation (a step necessary before sequencing) processes are not standardized. This leads to variations in data quality not only between different laboratories, but also between different lab members who prepare the specimen. This effect is known as the *batch effect* [LSB+10]. In general, this means that it does not suffice to use a single one-solution-fits-all error correction method, but that adaptive solutions are needed.

1.2. Objectives of this Thesis

A recent survey paper by Laehnemann *et al.* [LBM16] states that there is a need for a good modular error correction toolkit. The authors state that such a toolkit should have the following properties:

- optionally consider insertions and deletions
- optionally account for coverage bias (i.e., the influence of the percentage of G or C bases in the genome on how many reads map to a genomic region)
- have pre-defined platform-specific error models or learn them from the dataset
- be open source and adapt a modular approach
- infer haplotypes and repeats from the data
- flexibly combine ideas from different error correction approaches

In this thesis we want to accomplish a first step in this direction. We develop an error-correction toolkit which aims at fulfilling the requirements of Laehnemann *et al.* We mainly focus on de novo whole-genome sequencing datasets of haploid genomes (e.g. bacterial genomes or human mitochondrial genomes). Moreover, to show how our modules can interact, we develop an example error correction method that is based on counting the occurrences of DNA-substrings, so-called k-mers, in the read dataset. Figure 1.1 depicts some of the most common errors that occur in DNA sequencing reads. We aim to correct substitution, insertion, and deletion errors.

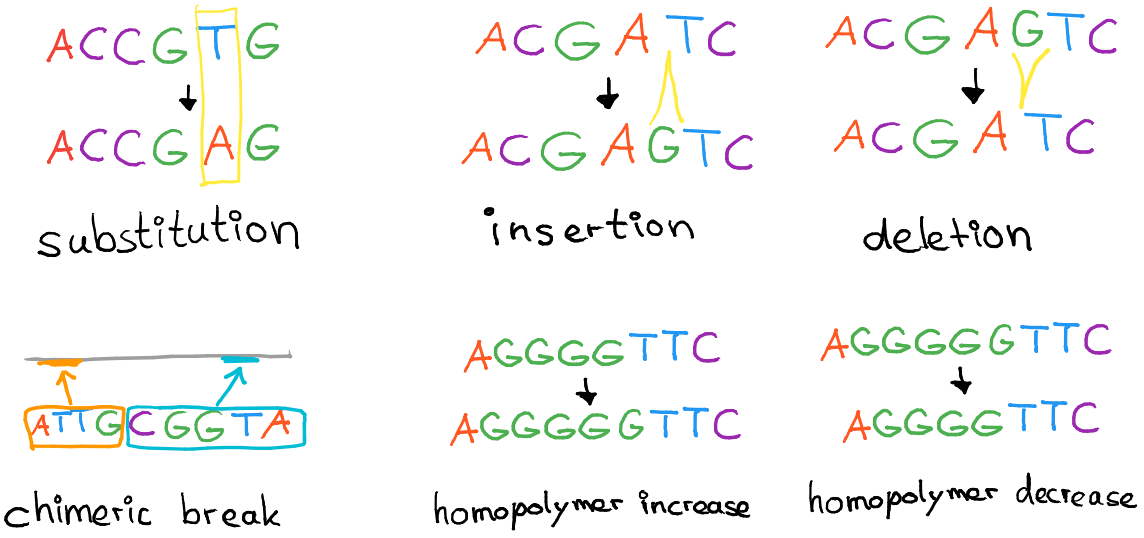


Figure 1.1.: Common types of DNA sequencing errors.

The crucial steps in any sequencing error correction method are to retrieve the error profile, to detect erroneous regions in a read, and to correct the read. In order to allow for a flexible choice of the approach used for each of these steps and improve future extendability, we split the framework into several modules (see Figure 1.2).

Error profile retrieval

We infer the general error profile of a sequencer given a set of reads and a well-known reference genome from a genome re-sequencing dataset (e.g., *E.coli K-12*) by training a machine learning classifier. In order to obtain the run-dependent error profile of a de-novo sequencing run (i.e., a sequencing run where the genome is not previously known), we apply an Empirical Bayes-like approach [Rob64]. In a warm-up phase, we take the general error profile as the initial prior and correct the reads using this profile. Then, we re-train the error profile by counting the error frequencies detected in this warm-up step and discard all corrections afterwards.

Detecting erroneous regions

Based on its bias-corrected observed count in the read dataset and its expected count in an ideal setting, we classify a k-mer as being either erroneous, unique, or belonging to a repetitive region in the genome. This allows us to work with variable-length k-mers, always selecting the smallest value for k such that the k-mer does not belong to a repetitive region.

Correct reads using a novel approach

Using locally adaptive k-mer sizes and a ranking of error probabilities for each position in a read, we determine the best options for correcting a k-mer. So far, we only correct substitution, insertion, and single-base deletion errors.

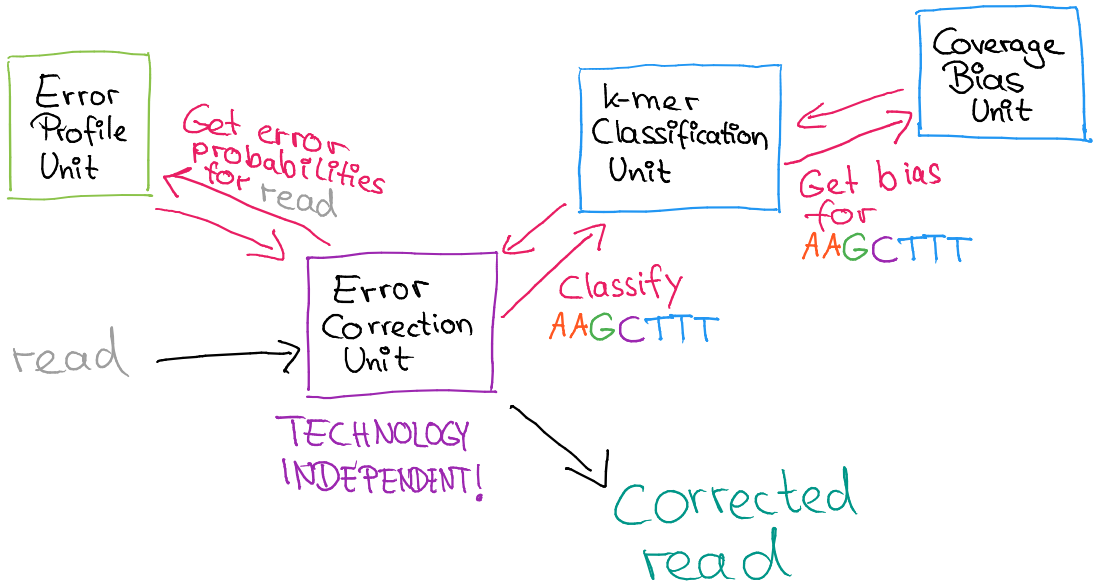


Figure 1.2.: Structure of the system. By encapsulating the technology-specific influences (e.g. error profile and coverage bias) into separate modules, the error correction and k-mer classification algorithms can remain technology-agnostic.

Scientific Contribution

While most of the ideas we implement in this thesis are not entirely new, we combine, extend and modify them. There does not exist a framework yet that integrates the approaches from different methods in a unified system. Current error correction methods only take into account a single aspect of the error profile (such as quality scores or position within a read), but ignore others (such as surrounding motif information).

Thus, we present in this thesis the first comprehensive framework for read error correction. It shortly will be available as open source at `https://github.com/algomaus/PAEC`.

1.3. Structure of the Thesis

The remainder of this thesis is structured as follows. In Chapter 2, we give an overview of current sequencing technologies and their error characteristics. We also summarize the current state-of-the-art error correction approaches with their underlying principles, advantages, and disadvantages. Moreover, we provide a short introduction to the classification problem in machine learning and present some well-known classification methods. We explain how to deal with imbalanced datasets, that is, datasets where some classes are over- or under-represented. We also briefly explain two metrics that are often used for evaluating the performance of a classifier on a dataset, namely the accuracy and the average F-score.

The main algorithmic contribution of this thesis is contained in Chapters 3 to 6.

In Chapter 3, we explain the method we use for estimating the median G/C-coverage bias using k-mers. It combines the approaches used in the EDAR [ZPB$^+$10] and the Fiona software [SWH$^+$14] paper. However, since we want to use different k-mer sizes, some extensions are needed.

Chapter 4 deals with classifying k-mers. By comparing bias-corrected observed and expected counts of a k-mer, in an ideal setting, we can classify it as either untrusted (erroneous), trusted (unique in the genome) or belonging to a repetitive region.

In Chapter 5, we describe how we learn the technology-specific error profile of a read dataset. This yields a black box that takes in a read and returns a ranking of error probabilities for each base in the read.

In Chapter 6, we develop a novel k-mer based method for read error correction to show how the other modules in our framework can be combined. The method covers a read with k-mers of variable size and then applies the most likely corrections to each k-mer until it becomes a trusted k-mer. We correct substitutions, insertions, and single base deletions. We additionally present a first idea for resolving deletions of multiple bases.

Chapter 7 contains some implementation aspects of this framework. We explain our solution for automatically training some classification methods on a given dataset and select the one with the highest average F-score. Then, we provide some noteworthy implementational details for each of the main modules. We conclude this Chapter by explaining how to use the framework.

In Chapter 8, we provide experimental results and their discussion. First, we evaluate each component of the framework independently. Then, we briefly compare the performance of our entire error correction framework with other state-of-the-art error correction methods, using read datasets from Illumina and Pacific Biosciences sequencing machines. We use both empirical bacterial datasets and simulated datasets for evaluation. Our comparison shows that while the remaining modules already provide satisfying results, our error correction algorithm can not compete with current state-of-the-art error correction tools. We further

investigate the reasons and discover that this is mainly caused by our error correction algorithm itself, not by the other modules within our framework.

Finally, the thesis ends with a conclusion and aspects of future work in Chapter 9.

2. Related Work and Background

This chapter provides some background knowledge related to sequencing, error profiles, error correction, and machine learning. Section 2.1 describes the DNA sequencing process. After explaining the very first ever sequencing method used to sequence a genome in more detail, we briefly describe currently used sequencing methods. We summarize well-known factors that influence the sequencing error profile in Section 2.2. In Section 2.3, we provide an overview of already existing approaches for error correction, by classifying them into k-mer based methods, multiple sequence alignment (MSA)-based methods, and suffix tree based methods. We conclude this chapter with a short introduction to the classification problem in machine learning as well as some widely used classification and evaluation methods in Section 2.4.

2.1. Sequencing Technologies and their Error Characteristics

Genomes tend to be much larger than the read lengths of current sequencing technologies. Thus, in order to sequence a genome, one has to first create multiple copies of it and then randomly break these copies into smaller fragments. These fragments are then sequenced and later re-assembled in order to obtain the DNA sequence of the whole genome. This approach is known as *Shotgun Sequencing* [Sta79][And81].

2.1.1. First Generation Sequencing (since 1977)

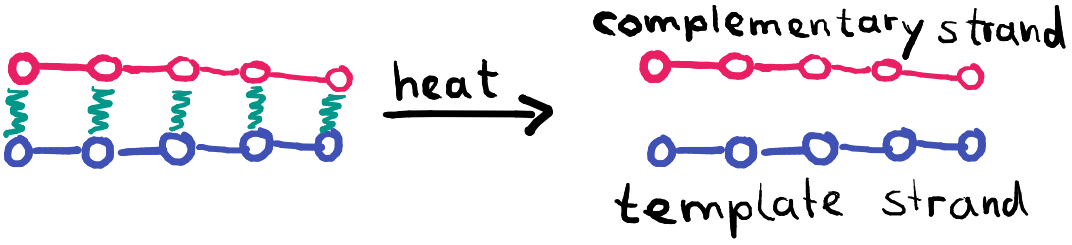
The first method to sequence DNA, Sanger Sequencing, was published in 1977 [SNC77]. Sanger Sequencing provides long (up to 1,000 bases) and highly accurate (per-base accurracy up to 99.999%) reads [SJ08]. However, it has a low throughput and thus, a high per-base sequencing cost. The most dominant error type in Sanger sequencing reads are substitution errors. Errors in Sanger sequencing reads are more likely to occur at positions toward the end of the read.

Sanger Sequencing uses the so-called Chain Termination Method. An easy way to understand the Chain Termination method is to think of two people, Alice and Bob, playing a game (idea taken from Olwen Reina [Rei17]). Alice thinks of a DNA-sequence. Bob's task is to learn the DNA sequence from Alice. The game works as follows: Alice keeps repeating the sequence in her head, silently reciting each of its bases. Bob randomly interrupts Alice. When interrupted, Alice tells Bob the position and base she was currently thinking of. For example, Alice could say "The base at position 42 is an A". Then, Alice restarts at the first

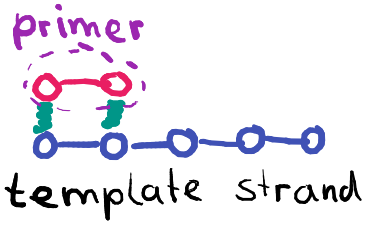
position in the sequence, again waiting for Bob to interrupt her. By repeating this step until all bases are known, Bob can recover the whole DNA sequence imagined by Alice.

The real steps of the Chain Termination Method are:

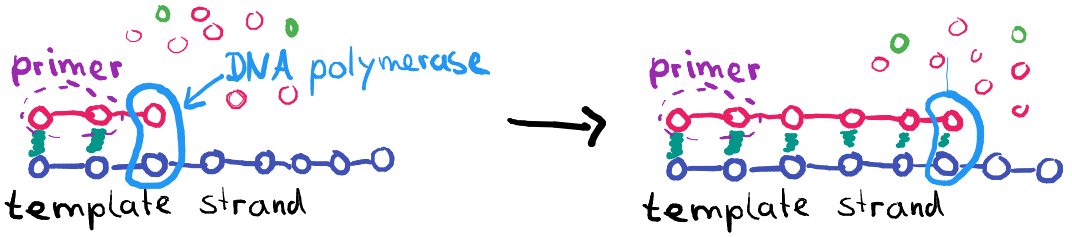
1. Heat up DNA to separate its strands.



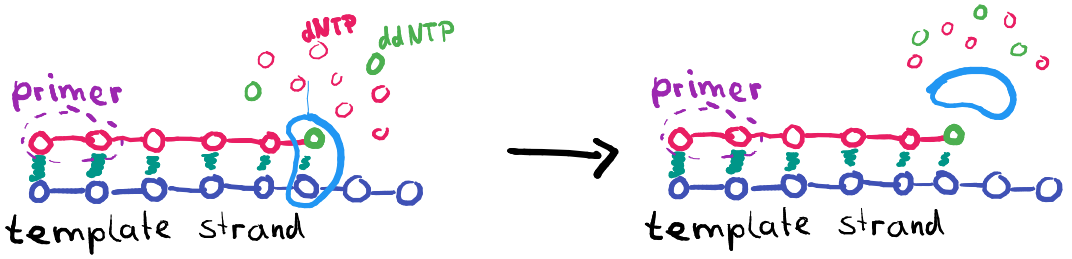
2. Lower the temperature to allow the primer sequence to bind to its complementary sequence in the template DNA strand.



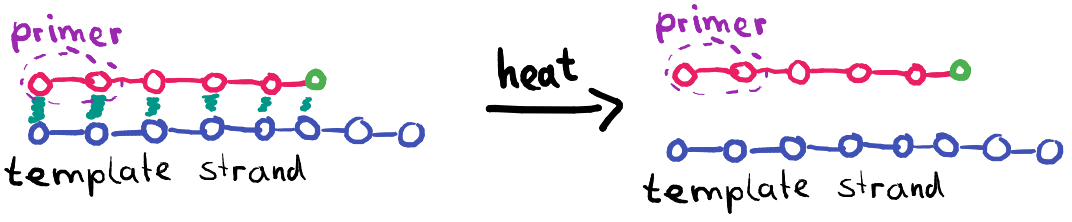
3. Raise temperature such that the DNA Polymerase attaches itself to the primer and creates a new strand of DNA (Polymerase Chain Reaction).



4. Deoxinucleotides (dNTPS; A, C, G, T) are added until a dideoxynucleotide (ddNTP; chemically altered and colored version of A, C, G, T) is added which stops synthesis (Chain Termination). This produces DNA fragments of different lengths.



5. Heat it again to separate the template strand from the complementary strand.



6. Perform a capillary gel electrophoresis. In the electrophoresis, smaller DNA fragments move faster than longer ones in the gel. The last base A, C, G, or T of a fragment can be determined by its color (see Figure 2.1).

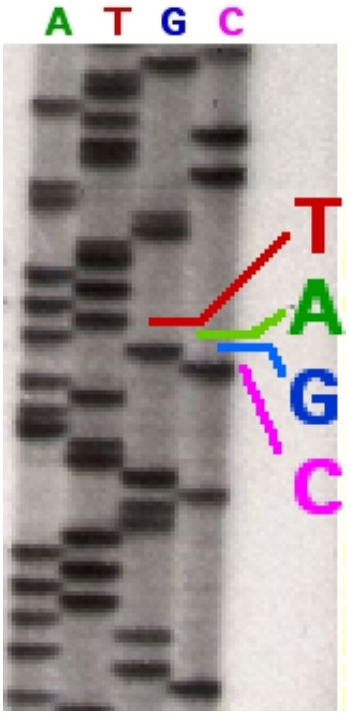


Figure 2.1.: Capillary gel electrophoresis used in the Chain termination method for Sanger sequencing. Smaller DNA fragments move faster in the gel than longer ones.
Image taken from https://upload.wikimedia.org/wikipedia/commons/c/cb/Sequencing.jpg

2.1.2. Next-Generation Sequencing (since 2005)

Next-Generation Sequencing approaches use so-called cyclic-array methods [SJ08]. After fragmenting the DNA into smaller pieces, dedicated artificial DNA sequences, so-called *adapters*, are added to both ends of the DNA fragments in order to create a *sequencing library*. When creating copies of those library DNA molecules, copies that originate from the same library molecule are spatially clustered on a planar substrate. Cyclic-array sequencing consists of many iterations. In iteration i, the ith bases of all fragments are recovered simultaneously. This explains the high throughput of Next-Generation sequencing methods.

While Next-Generation Sequencing methods are faster and cheaper than Sanger sequencing, they unfortunately produce shorter reads (up to a few hundred bases) which are also less accurate. Several companies, such as Illumina [Ill17b] and Ion Torrent [Tor17], offer Next-Generation sequencers. The most common error in Illumina sequencing are substitution errors which are more likely to occur toward the end of reads and after a preceding "GG" motif [SDI+16]. Moreover, Illumina sequencing suffers from a G/C-coverage bias – less reads cover regions of the genome with a high or low G/C-content [RRC+13]. The estimated error rate of Illumina sequencers is between 1% and 2.5% [ARDB16]. As the sequenced reads still contain adapter sequences, a post-processing step is needed to remove these adapter sequences from the reads.

However, not only the sequencing step introduces errors [ARDB16]. The library preparation step can induce errors, as well. Some library preparation methods rely on a Polymerase Chain Reaction (PCR), but PCR-free library preparation approaches also exist. Library

preparation methods that require a PCR can also increase the G/C coverage bias. DNA consists of the bases A,C,G, and T. The G/C-content of a DNA sequence s is defined as

$$gc(s) = \frac{\text{Number of G or C bases in } s}{\text{Total number of bases in } s}.$$

Genomic regions with an extreme G/C-content suffer from non-uniform coverage issues when being sequenced, especially if a PCR is used for library preparation. This means that some regions in the genome are covered by less reads than other regions, leading to difficulties in genome assembly [CLY+13].

In Illumina sequencing, errors that occur in one iteration also affect the later iterations. This is why errors are more likely to occur toward the end of Illumina reads [ARDB16].

2.1.3. Third-Generation Sequencing (since 2010)

Third-Generation sequencing methods use single-molecule sequencing. In single-molecule sequencing, each DNA molecule is sequenced directly, without the need to create multiple copies of it [HC16]. The two main suppliers of Third Generation sequencing methods are Pacific Biosciences [Bio17] (Single-Molecule Real-Time Sequencing) and Oxford Nanopore [Nan17] (Nanopore Sequencing). PacBio uses the so-called Real-Time Sequencing technology. Here, the bases of the DNA are sequenced at the same time as when they are added by the DNA Polymerase. The error rate of PacBio sequencers lies between 15% and 20% [ARDB16].

Oxford Nanopore uses pores on the nanometer scale, so-called *nanopores* to isolate and identify bases. Only one molecule at a time can pass through such a nanopore. While the molecule passes through the nanopore, the electrical current emitted by the nanopore changes. This change is used to determine which molecule (A,C,G,T) passed through the nanopore [ARDB16]. Insertions are the most frequent error type in Oxford Nanopore sequencers. The average error rate in these sequencers is as high as 25% – 40%. However, the technology is still under active development [ARDB16].

Third-Generation Sequencing technologies provide longer (a few thousand bases) and cheaper reads, while being less susceptible to G/C coverage bias. Longer reads are advantageous for the genome assembly process, as they facilitate handling repetitive regions in the genome. The main error types in Third Generation sequencing are insertion and deletion errors. Insertion and deletion errors are often summarized as *indel* errors.

2.2. Additional Sources of Sequencing Errors

There exist a plethora of additional factors which influence the sequencing error profile:

- **Homopolymer bias** Some technologies (e.g. Ion Torrent [BSB+13]) fail to exactly determine the length of homopolymeric regions, thereby introducing new nucleotides or missing out some of them. A *homopolymer* is a DNA sequence $s = s_1 s_2 \dots s_n$ of size $n \geq 3$ such that $s_1 = s_2 = \dots = s_n$.
- **Strand Specificity** A DNA molecule consists of a leading strand and a lagging strand (see Figure 2.2). Some errors are more likely to occur on the leading strand of the genome while others appear more often on the lagging strand. For example, in Ion Torrent sequencing machines, it depends on the specific type of the DNA strand whether errors occur mostly at homopolymeric regions or not [BSB+13].

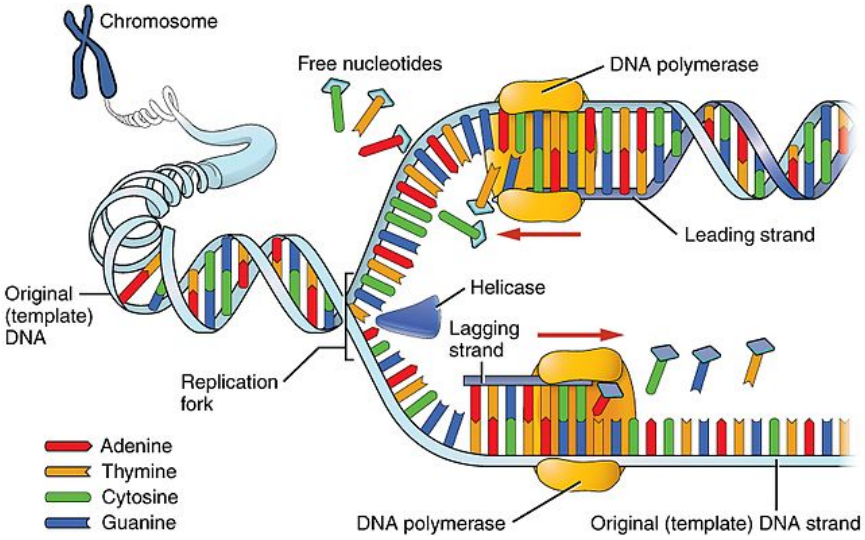


Figure 2.2.: The leading strand and the lagging strand of a DNA molecule. The DNA polymerase on the leading strand can continuously replicate the DNA. On the lagging strand, the DNA Polymerase can only synthesize small pieces at once as the replication fork grows in the opposite direction. Image taken from https://upload.wikimedia.org/wikipedia/commons/thumb/5/50/0323_DNA_Replication.jpg/640px-0323_DNA_Replication.jpg

- **Motif bias (Sequence-specific Errors)** Some errors are often surrounded by a specific, short DNA substring, like, for instance, substitutions after GGC in Illumina sequencing [SID+15].
- **Systematic Error** There are positions/regions in the genome where more errors occur than at other sites [MBD+11].
- **Chimeric Reads** It can happen that a read consists of a sequence at one place of the genome concatenated with another sequence from some other part of the genome.
- **Sequencing Machine** Different sequencing technologies have different error rates. They also vary depending on the specific model. In Illumina sequencers, error rates may even vary between the different sequencing lanes [SSB13].

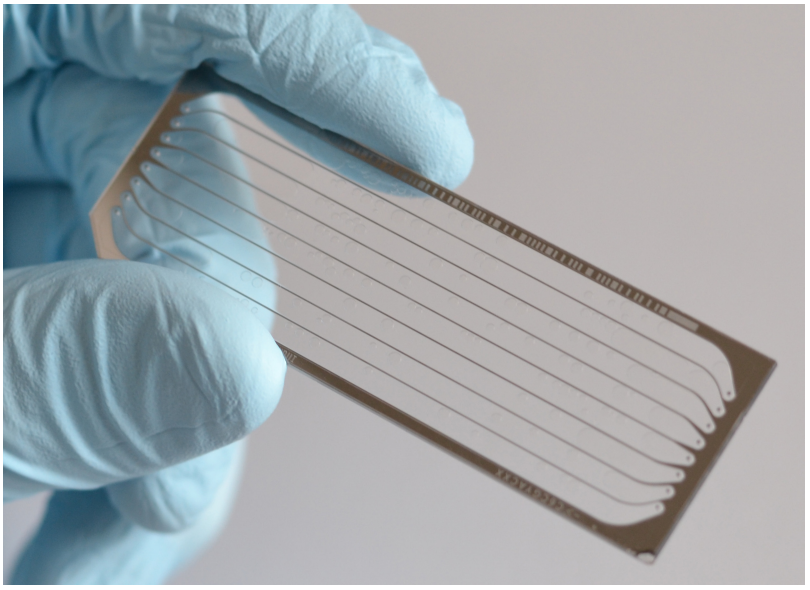


Figure 2.3.: An Illumina flowcell with eight sequencing lanes. Samples are loaded onto each of these lanes for simultaneous sequencing [Ill17c]. Image taken from https://upload.wikimedia.org/wikipedia/commons/0/03/Next_generation_sequencing_slide.jpg

- **Involved Nucleotides** Different substitutions/insertions/deletions occur with different probabilities. This can be due to the technology used to identify different bases. Identifying a base is known as *basecalling*. The Phred quality scores [EHWG98] indicate how confident the basecaller was that a nucleotide was correctly read. Low quality scores may indicate substitution errors. The Phred Quality Score Q is defined as
$$Q = -10 * \log_{10}(P), \text{ where } P \text{ is the base-calling error probability.}$$

- **Uncalled Bases** Sometimes the basecaller fails in identifying whether a base in a read is an A, C, G, or T. This is denoted by the letter N.

- **Polyploidy** Substitution errors need to be distinguished from true biological variability (i.e., heterozygous sites), and single nucleotide polymorphisms (in case multiple genomes within a population are being sequenced).

- **Repetitive Regions** Complex genomes can have many repetitive regions, making them difficult to assemble. These repetitive regions can also be problematic in error correction if reads from different parts of the genome are identified as coming from the same part.

- **Inverted Repeats** An inverted repeat is a DNA sequence followed by its reverse complement. There may be other nucleotides in between (see Figure 2.4).

  The *reverse complement* of a DNA sequence $s = s_1 s_2 \dots s_n \in \{\text{A}, \text{C}, \text{G}, \text{T}\}^n$ is defined as $\hat{s} := c(s_n) \dots c(s_2) c(s_1)$ with $c(\text{A}) := \text{T}$, $c(\text{C}) := \text{G}$, $c(\text{G}) := \text{C}$, and $c(\text{T}) := \text{A}$.

  A long inverted repeat is more likely to cause errors in Illumina sequencing [NOM+11].

  Figure 2.4.: An inverted repeat of the DNA sequence TTACG. The 'N' bases represent arbitrary bases.

- **Human Error (Batch Effect)** As library preparation protocols are not always standardized, library preparation might even differ among members of the same lab. This effect is known as the *batch effect* [LSB+10].

- **Run-Dependency** Error profiles can vary between different runs of the same sequencer [MLT12].

Error Profiles in Literature

Error profiles of various sequencers have been published [NOM+11] [BSB+13] [LHC+15]. There is no standardized way of describing an error profile. Most of these papers provide a multitude of graphs and tables, showing various aspects related to errors. Unfortunately, only a small fraction of this information can directly be used by de novo error correction algorithms. Current sequencing error correction methods only take partial aspects of the error profile into account, such as different substitution rates and quality score information. Many error correction tools ignore sequence-specific errors. Moreover, instead of taking run-specific error profile characteristics into consideration, most tools rely on a static, pre-configured general error profile.

Potential Impact of improved Error Profiles

Many Bioinformatics applications, such as approaches for estimating the genome size from the raw read dataset of an unknown genome [HVB15], could profit (i.e., in terms of improved accuracy) from taking into account the error profile information of a sequencer.

However, to the best of my knowledge, there is currently no stand-alone module available that encapsulates the technology-specific error profile. This would allow other applications to use error profile information without having to deal with the details.

2.3. An overview of Error Correction Approaches

Error correction methods can be classified into three main approaches. We will use the same classification as used in the survey papers by Yang *et al.* [YCA13], Laehnemann *et al.* [LBM16], and Alic *et al.* [ARDB16]. They classify them into k-mer based methods, methods which use suffix trees, and methods that are based on multiple sequence alignment (MSA). Most error correction tools target reads from a single sequencing technology. Some tools however, such as Jabba [MHD$^+$16] and LoRDEC [SR14], aim to correct long reads (obtained by Third Generation Sequencing) with the help of Illumina reads (which exhibit a lower error rate).

2.3.1. K-mer based methods

K-mer based error correction methods use k-mer frequencies. A k-mer is a subsequence of a fixed size k within a read. The *frequency* (or *coverage*) of a k-mer is the number of occurrences of the k-mer in the whole read dataset. Depending on the number of occurrences of a k-mer in the read dataset, the methods decide whether the k-mer is correct or not. For doing so, an – often global – cuttoff value is chosen such that all k-mers with a frequency smaller than this value are considered as being erroneous (see Figure 2.5). Pal and Aluru [PA14] improve this k-mer classification approach by also taking quality scores and counts of highly similar k-mers into account.

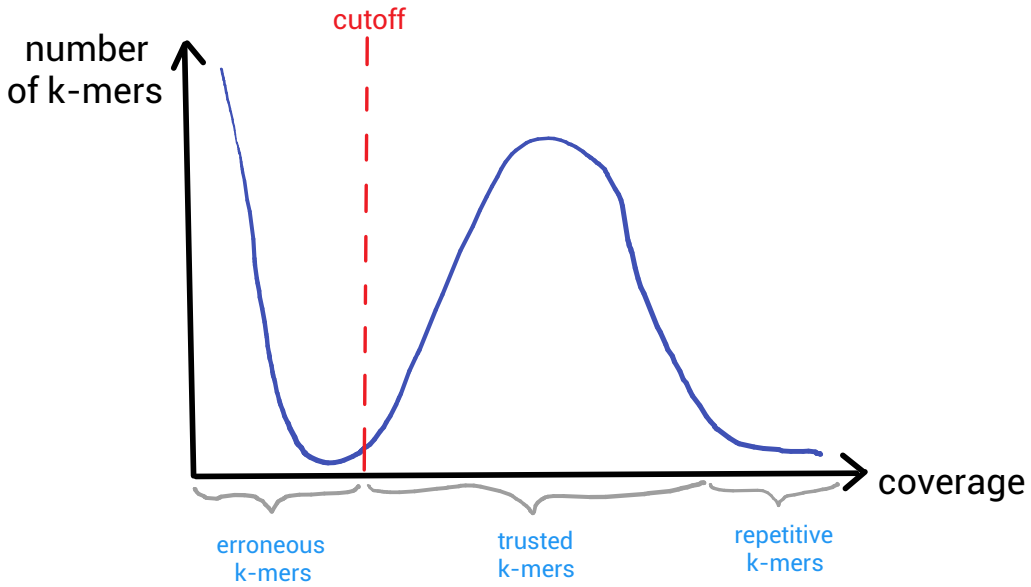


Figure 2.5.: Depending on the k-mer coverage distribution, k-mer based error correction methods determine a cutoff value in order to classify k-mers as being either erroneous, trusted or repetitive.

Most k-mer based error correction methods try to correct an erroneous k-mer by transforming it into a trusted k-mer, using a pre-specified maximum number of nucleotide changes. Some k-mer based methods use the entire k-mer spectrum [YCA13]. In a k-mer *spectrum*, instead of storing only the number of occurrences of each k-mer, the actual positions of the occurrences are also stored.

Examples for k-mer based error correctors are EULER-SR [CP08], Lighter [SFL14], BayesHammer [NKA13] (which clusters similar k-mers instead of using a global cut-off value, and also considers quality scores), and Blue [GDPB14] (which re-infers the offset for each read).

Advantages

- Counting k-mers is faster than building an MSA. There are heuristic approaches which use dedicated probabilistic data structures, so-called *Bloom Filters*, for approximate counting and memory-efficient storage of the approximate k-mer frequencies.
- K-mer based error correction methods perform well on datasets with a low error rate that mostly exhibit substitution errors.

Disadvantages

- K-mer based error correction methods use a **fixed size** k. If k is chosen too small, many k-mers belong to repetitive regions in the genome, thus leading to mis-corrections. If k is chosen too large, correcting an erroneous k-mer becomes more difficult as a larger k-mer is more likely to contain multiple errors.
- Many older k-mer based methods ignore the coverage bias altogether and use a single global cutoff value. However, this means that k-mers with an extreme G/C-content are under-represented in the read dataset. Thus, these k-mers will be mis-classified as being erroneous, even if they are correct.

  Newer k-mer based classification methods often use a local cutoff value instead of a global one in order to bypass coverage bias issues. However, they do not explicitly model the coverage bias itself.
- K-mer based error correction methods rely on the assumption that errors are rare and random. Thus, these methods do not perform well if the sequencing error rate is very high, such as 20 % in Third Generation sequencing.
- Most k-mer based error correction methods only correct substitution errors, but not indels.

2.3.2. Suffix-Tree based methods

Suffix-tree based error correction methods store all reads in a suffix tree (or more efficient variants thereof, such as suffix array [MM93] or FM index [FM00]) [LBM16]. This way, they can efficiently compute k-mer frequencies for different sizes k.

In a suffix-tree, prefixes of reads are grouped together in a node until they show different suffixes (see Figure 2.6). Splits in the suffix tree which lead to branches having a substantially lower number of reads attached to them than their sibling branches, indicate sequencing errors [ARDB16].

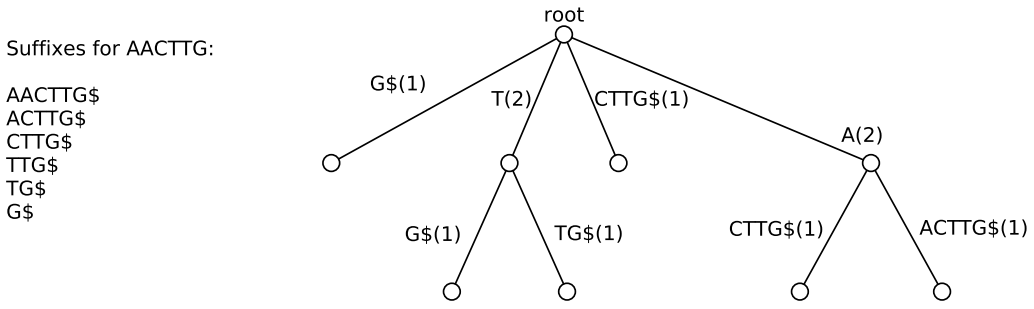


Figure 2.6.: The suffix tree for a single read AACTTG. Suffix-tree based error correction methods build a large suffix tree for all reads.

Examples for suffix-tree based error correction methods are HiTEC [IFI11] (which directly corrects a detected error based on k-mer frequencies) and Fiona [SWH+14] (which applies a statistical error model, assuming uniform coverage and uniform error rates).

Advantages

- Suffix-tree based error correction methods use k-mer of variable sizes.
- Using highly-optimized variants of suffix trees, suffix-tree based error correction approaches are well-suited for high-throughput sequencing data [LBM16].

Disadvantages

- Suffix-tree based error correction methods are slower than simple k-mer based error correction methods.

2.3.3. MSA-based methods

MSA-based error correction methods build an MSA of all reads. Given this MSA, errors are then either corrected by performing a majority vote or by applying more complex approaches (e.g., taking different substitution penalties into account).

Since building an MSA is $\mathcal{NP}$-hard for most optimality criteria [WJ94] [Eli06], MSA-based error correction methods apply heuristic approaches. Older MSA-based methods target small datasets from Sanger sequencing which contain very long reads. These methods compute the MSA by refining pair-wise alignments of all reads [LBM16]. More recent MSA-based methods, such as Coral [SS11] and MuffinEc [ATMB16], pre-align the reads by using k-mer seeds. Hence these methods instantly cluster reads which share a common k-mer and then try to extend the alignment.

Advantages

- Most MSA-based error correction methods aim to correct not only substitutions, but also insertions and deletions.
- Because of the alignment, MSA-based methods can better handle repetitive regions than k-mer based methods.
- MSA-based error correction methods are well-suited for Third Generation Sequencing datasets [LBM16].

Disadvantages

- Building an MSA is a time-intensive step.

2.4. Machine Learning Classifiers

The classification problem in machine learning studies how to automatically make predictions on new observations based on old observations. Given some labeled training data, a machine learning algorithm is trained to determine a classification rule. Each data point is represented by a vector of *features* $x := (x_1, x_2, \ldots, x_n)$. For example, if we want to classify a fruit as being either an apple, a pineapple, or a banana, we could look at its size, color, and shape. Thus, size, color, and shape would constitute the dimensions of the three-dimensional feature space in this fruit-classification problem.

id	size	color	shape	class
0	small	red	round	apple
1	small	yellow	long	banana
2	large	brown	round	pineapple
3	small	green	round	apple
4	large	green	long	banana
5	small	green	round	pineapple
6	small	brown	long	banana
7	large	red	round	**?**

Table 2.1.: The training data consists of entries with $0 \leq \text{id} \leq 6$, as their class label is already known. Given the feature vector $x = (\text{large, red, round})$ for entry number 7, we want to classify it as being either an apple, a pineapple, or a banana. In order to do so, we automatically learn a classification rule from the training data.

k-Nearest Neighbors Classifier

The k-Nearest Neighbors classifier [Alt92] assigns an unlabeled data point to a class by performing a majority vote using its k nearest neighbors in feature space. The parameter k is given as an input to the classifier. We can try different values for k and then decide for the value that performs best on a test data set.

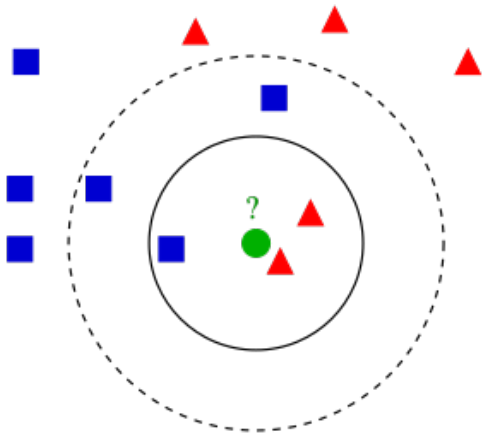


Figure 2.7.: Example of a k-Nearest Neighbor classification. The classes are red triangles and blue squares. If $k = 2$, the green unknown data point is classified as a red triangle. If $k = 5$, it is classified as blue square instead. Image taken from `https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm#/media/File:KnnClassification.svg`

Naïve Bayes Classifier

The Naïve Bayes classifier [LCS+06] assumes strong statistical independence between the features, even if the features are known to be correlated in reality. It then uses the Bayes Theorem to compute class probabilities:

$$\mathbb{P}(\text{class}|\text{x}) = \frac{\mathbb{P}(\text{class}) * \mathbb{P}(\text{x}|\text{class})}{\mathbb{P}(\text{x})}$$

The probabilities are approximated by the relative frequencies in the training dataset. In the example from Table 2.1,

$$\mathbb{P}(\text{banana}) \approx \frac{3}{7} \quad \text{and}$$

$$\begin{aligned}\mathbb{P}((\text{large, red, round})|\text{banana}) &= \mathbb{P}(\text{large}|\text{banana}) * \mathbb{P}(\text{red}|\text{banana}) * \mathbb{P}(\text{round}|\text{banana}) \\ &\approx \frac{1}{3} * \frac{0}{3} * \frac{0}{3} \\ &= 0.\end{aligned}$$

Thus, the final probability $\mathbb{P}(\text{banana}|(\text{large, red, round}))$ is 0.

Logistic Regression Classifier

In its original form, the Logistic Regression classifier [LCS+06] can be used to classify a data point into the classes 1 (true) or 0 (false). It returns a probability that the data point to be classified belongs to class 1. In order to compute this probability, it fits a logistic regression function to the training data set. A logistic function has the form

$$f(t) = \frac{1}{1 + e^{-t}}$$

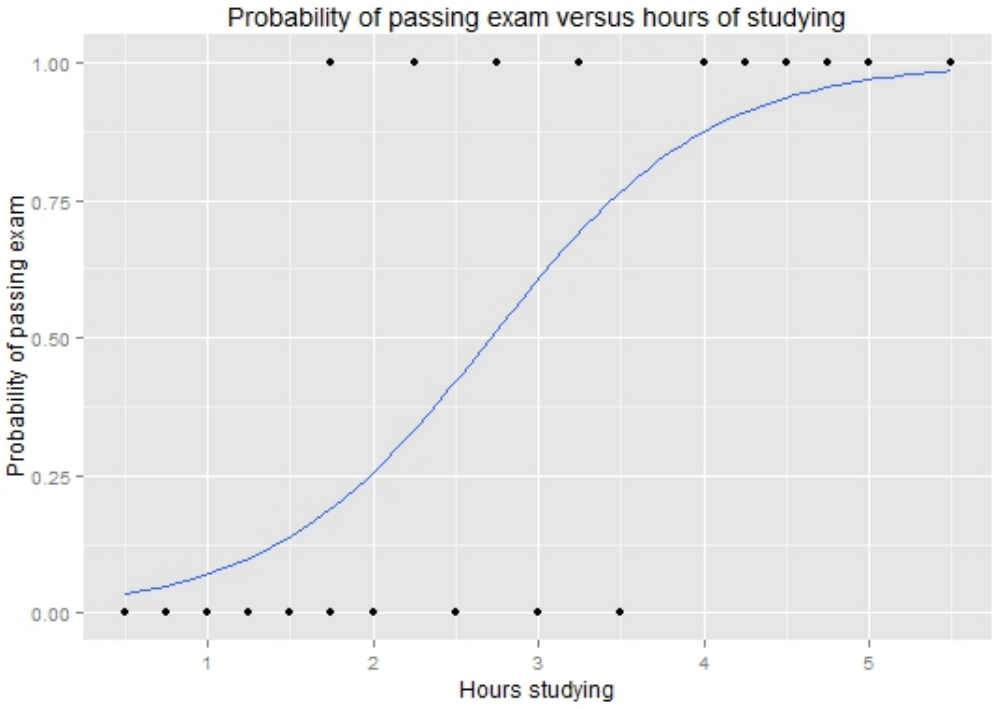


Figure 2.8.: Example for a Logistic Regression classifier that gives the probability of passing an exam given the number of hours spent studying. Image taken from `https://en.wikipedia.org/wiki/Logistic_regression#/media/File:Exam_pass_logistic_curve.jpeg`

In order to perform the classification for more than two classes, multiple logistic regression classifiers are trained and subsequently combined. In our example from Table 2.1, we wanted to classify a fruit as either being an apple, a banana, or a pineapple. In order to approach this classification problem via Logistic Regression, three Logistic Regression Classifiers need to be trained:

1. Is the fruit with features x a banana or not? This classifier estimates $\mathbb{P}(\text{banana}|x)$.
2. Is the fruit with features x an apple or not? This classifier estimates $\mathbb{P}(\text{apple}|x)$.
3. Is the fruit with features x a pineapple or not? This classifier estimates $\mathbb{P}(\text{pineapple}|x)$.

To make the final prediction for the fruit with features x, we pick the class C with the highest probability $\mathbb{P}(C|x)$. All multi-class classification problems can be transformed into several two-class classification problems using this *one-versus-all* method [Aly05] described above.

Decision Tree Classifier

A Decision Tree classifier [SL91] partitions the feature space, assigning each of the partitions to a class. It does so by recursively splitting the feature space into two parts.

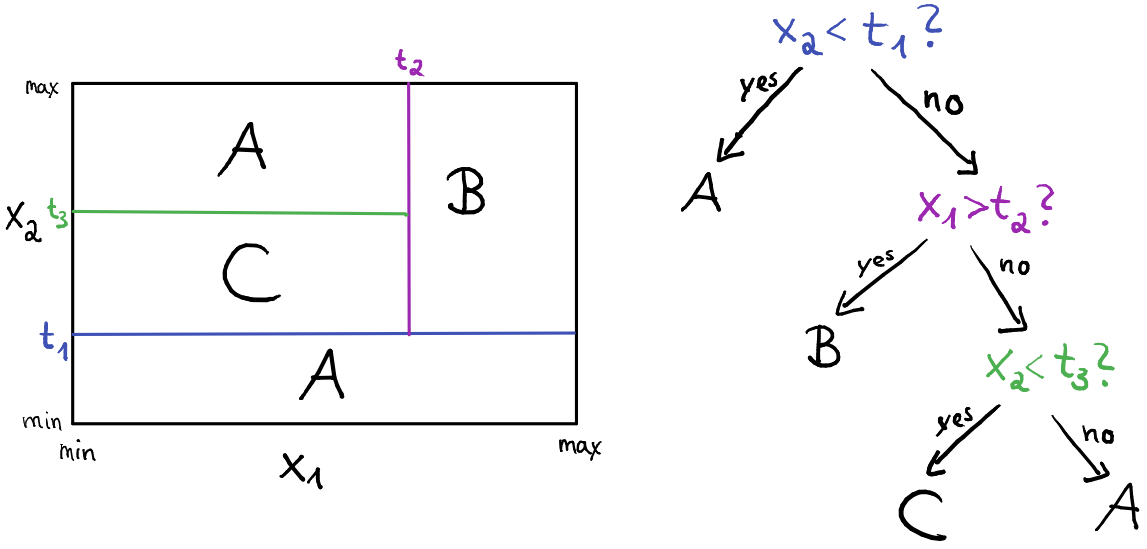


Figure 2.9.: An example for a decision tree classifier for a two-dimensional feature space and three classes A, B and C.

Random Forest Classifier

The Random Forest classifier [Bre01] trains multiple Decision Tree classifiers. Each Decision Tree classifier is trained with a random subset of the training data. Then, the classification of a data point to be classified is done by performing a majority vote using the classifications obtained from the decision tree classifiers. Fernández-Delgado *et al.* [FDCBA14] show that Random Forest classifiers often perform exceptionally well in real-world classification problems.

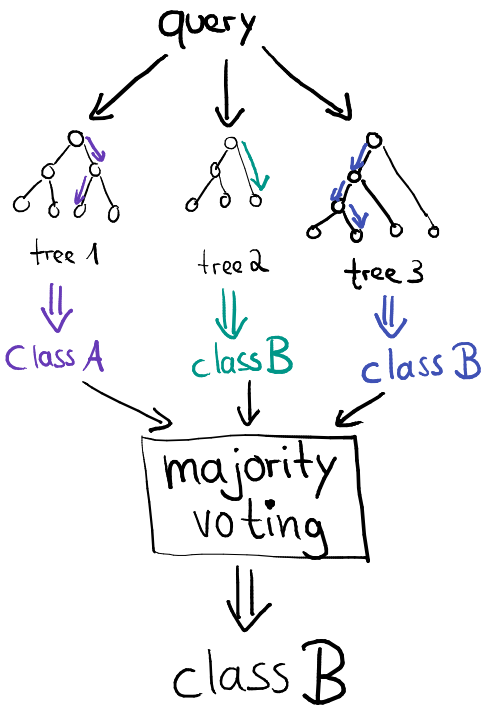


Figure 2.10.: An example of the classification process in a Random Forest classifier. The classification is done by a majority vote on the classifications from various Decision Trees.

AdaBoost Classifier

Adaptive Boosting (AdaBoost [FS95]) is a meta machine-learning algorithm. It takes a weak base learner (i.e., one that does not have perfect, yet suitable classification accuracy, but is fast to train) and then iteratively trains multiple instances of the weak learner. In the first iteration, only one learner is trained. In each of the following iterations, mislabeled data points from the previous classifier are weighted stronger than correctly labeled data points. The final classifier is a linear combination of the weak classifiers trained by this method.

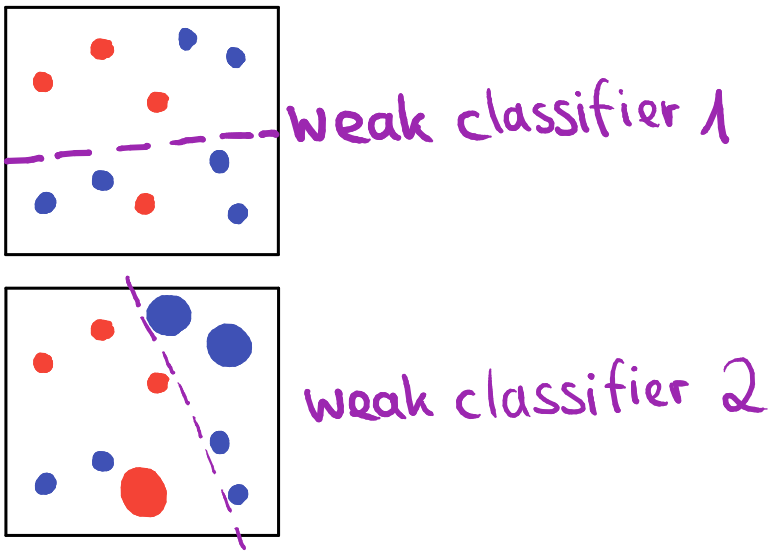


Figure 2.11.: The first two iterations of training an AdaBoost classifier. Larger circles represent data points with an increased weight.

2.4.1. Evaluating a Classifier

In order to be able to evaluate a classifier, not all labeled data is used for training. Instead, a certain percentage of the labeled data (e.g., one third) is retained in order to assess the classifier's performance on this data. The simplest measure for evaluating a classifier is *accuracy* [Bro17a]. The accuracy of a classifier is the percentage of data points that have been correctly assigned to their respective class.

However, the overall accuracy does not always constitute an informative measure. Imagine the following situation: Someone wants to classify brain scans of people that belong either to healthy individuals or serial killers. Since more than 99 percent of people are not serial killers, let us assume that our training dataset also inherits this class distribution. A classifier that would simply label all brain scans as healthy would thus already exhibit an accuracy of more than 99%, while being totally useless for its intended application. To fix this issue, we can look at the *confusion matrix* (see Figure 2.12). The confusion matrix consists of four entries, representing the true positives (TP), true negatives (TN), false positives (FP) and false negatives (FN). All evaluation metrics for classifying use these entries. For instance, the accuracy measure defined above can be written as:

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

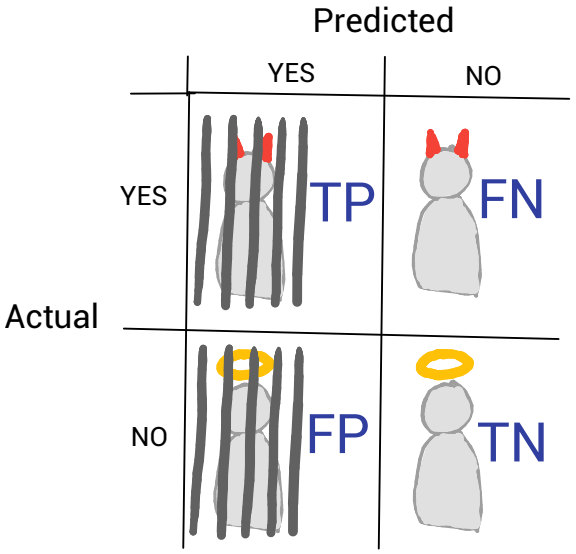


Figure 2.12.: An example for a confusion matrix for the binary classification problem "Is this person a serial killer?". The bars in the picture mean that the person has been classified as serial killer and therefore is put into prison.

F-Score

Another widely used evaluation metric for classifying is the *F-Score* [Rij79]. The F-Score is the harmonic mean of *precision* and *recall* (see Figure 2.13). If more than two classes exist, one can compute the *average F-score* after transforming the multi-class classification problem into several binary-class classification problems.

$$\text{precision} = \frac{TP}{TP + FP}$$

$$\text{recall} = \frac{TP}{TP + FN}$$

$$\text{F-score} = 2 * \frac{\text{precision} * \text{recall}}{\text{precision} + \text{recall}} = \frac{2 * TP}{2 * TP + FP + FN}$$

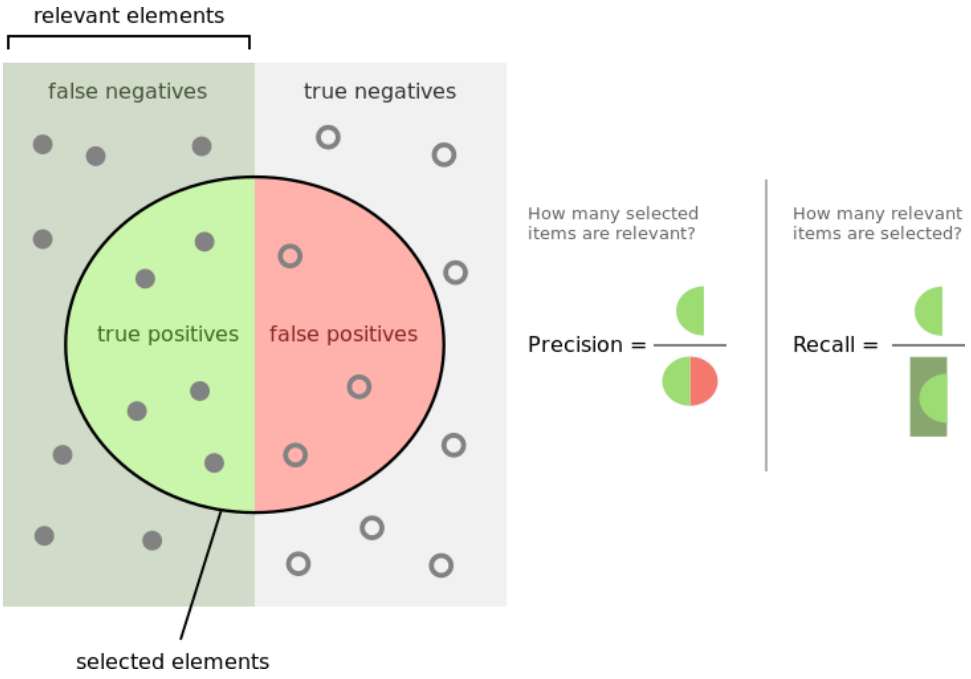


Figure 2.13.: Visualization of precision and recall. Image taken from https://upload.wikimedia.org/wikipedia/commons/2/26/Precisionrecall.svg

2.4.2. Imbalanced Datasets

In real life applications, the classes are often not equally distributed in the dataset. While some basic classifiers such as the k-Nearest-Neighbors classifier can naturally deal with imbalanced datasets, others can not. Let us assume that we have a dataset with classes A and B. Further assume that 95% of the data points belong to class A, and only 5% belong to class B. Using this dataset as-it-is, many classifiers would over-fit on class A and ignore class B completely, leading to an accuracy of 95%.

One way to resolve this issue is to use a different evaluation metric for training, for instance, the average F-score. However, classifiers are often used as black boxes within machine learning frameworks. This means that the user of such a framework has no easy access to the algorithm used for training the classifier. Thus, another way to handle an imbalanced dataset consists in modifying the dataset, using one of the following methods [Bro17b]:

- Artificially increase the number of data points that belong to class B by copying them. This is known as *over-sampling*. Another way of doing this is by assigning higher weights to classes that occur less often in the datasets.

  Instead of just copying existing data, it is also possible to generate new samples of B by combining the characteristics from the data points belonging to B.

- Artificially decrease the number of data points that belong to class A. This is known as *under-sampling*. Alternatively, one can reduce the weights of data points belonging to class A.

In our context of error correction, the error types are not equally likely to occur in the sequencing data. Thus our framework needs to be able to handle imbalanced datasets.

3. Coverage Bias Unit

Own Contributions

- Introduce the term *Perfect Uniform Sequencing Model* to describe an idealized sequencing setting.
- Derive an *exact* formula for the expected count and standard deviation of a k-mer under the Perfect Uniform Sequencing Model, distinguishing between circular and linear genomes.
- Infer median coverage bias values for different G/C-contents, using the relative difference between expected and observed count of a k-mer.
- Answer coverage bias queries by linear interpolation.

Some sequencing technologies, especially Illumina sequencing in combination with a library preparation method that uses PCR (Polymerase Chain Reaction, a method to produce multiple copies of DNA), show a bias in genome coverage depending on the G/C content. If there was no bias, we would expect that all bases within the sequenced genome are covered by approximately the same number of reads. However, this is rarely the case in real sequencing runs. In literature, the term coverage is used in two different meanings (see Figure 3.1). It can either describe the number of reads that map to a single position in the genome or it can describe the number of reads that map to a unique k-mer in the genome. In this thesis, we will always use the latter meaning.

In de novo error correction applications, the genome that has been sequenced has not been sequenced before. Hence, there is no reference sequence to compare to. Thus, when quantifying coverage for de novo correction, we need to start from a read's perspective and answer the following question:

When looking at a k-mer in a read, how often should this k-mer appear in the entire read dataset if it would occur only once in the genome?

We model coverage-bias-correction by using a correction factor that is multiplied with the observed k-mer coverage. This factor allows us to obtain a revised k-mer count without

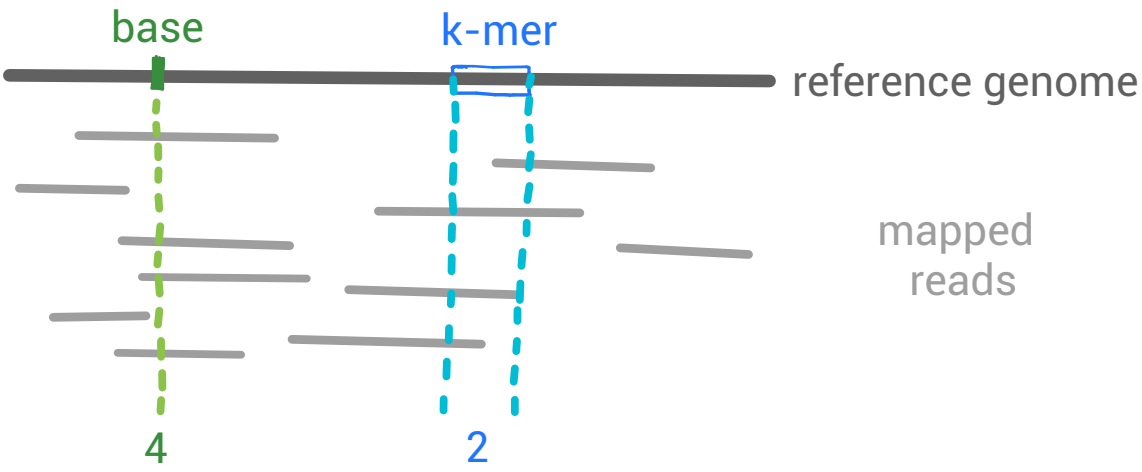


Figure 3.1.: The two meanings of "coverage" in DNA sequencing: Coverage of a k-mer versus coverage of a base. The base highlighted in green is covered by 4 reads and the k-mer highlighted in blue is covered by 2 reads.

coverage bias. Hence, the main purpose of the Coverage Bias Unit is to return a real number $bias(k\text{-mer})$ such that we can obtain a bias-corrected observed k-mer count:

$$\text{cov}_{\text{bias-corrected}}(k\text{-mer}) := \frac{1}{bias(k\text{-mer})} * \text{cov}_{\text{observed}}(k\text{-mer})$$

3.1. Perfect Uniform Sequencing Model

Given a k-mer that is unique in the genome, we compute its expected coverage and standard deviation in an idealized setting. In this thesis, we call this setting the *Perfect Uniform Sequencing Model* (PUSM). In the Perfect Uniform Sequencing Model, we make the following assumptions:

- There is no coverage bias, that is, it is equally likely for a read to map to any region of the genome.
- The reads contain no errors and no adapter sequences.
- No read is longer than the estimated genome size (i.e., no position in the genome occurs more than once in the same read).
- No read is shorter than the k-mer in question.

Since the expected coverage is used for estimating coverage bias in the case that the sequenced genome is unknown, we further assume that the location of the specific k-mer within the genome is unknown and that each location is equally likely.

Let $(l_i, n_i)_{i=1,\dots,m}$ be the read length distribution for m read lengths. The number of reads with length l_i is n_i. In the Fiona software paper by Schulz *et al.* [SWH+14], the expected coverage of a k-mer is approximated by

$$\text{cov}_{\text{expected,pusm}}(k) \approx \sum_{i=1}^{m} n_i * \frac{l_i - k + 1}{N - l_i + 1}$$

where N is the estimated genome size. However, their approximation only works under the assumption that the read lengths are much smaller than the estimated genome size. This is not necessarily the case if long reads are obtained through single-molecule sequencing of a small bacterial genome, for instance. Thus, we need to derive an exact formula for the expected coverage and standard deviation of a k-mer here.

We have to distinguish between two cases. Either the genome is circular or the genome is linear. Since the number of possible positions a read can start at differs between these two cases, the expected coverage of a k-mer is also influenced by this.

3.1.1. Circular Genome

If the genome in question is circular, the formula for expected coverage of the k-mer is very similar to the formula from the Fiona software paper. In a circular genome, a read can start at any position in the genome (see Figure 3.2). Thus, we have N possibilities for mapping a read to the genome. For a read of length l and a k-mer of length k, there are $l - k + 1$ possible mappings of the read to the genome such that the read covers the k-mer. Notice that this holds regardless of the position of the k-mer in the genome. Since we assume that the read is equally likely to be mapped to any position in the genome, the probability that the read covers the k-mer is

$$\mathbb{P}(\text{read of length } l \text{ covers } k\text{-mer}) = \frac{l-k+1}{N}, \text{ where } N \text{ is the estimated genome size.}$$

Figure 3.2.: The first base of a read can be mapped to any position of the circular genome.

As the reads are independent of each other, the expected coverage of a k-mer given n_i reads of length l_i follows a binomial distribution $\mathcal{B}(n_i, p_i)$ with probability

$$p_i = \frac{l_i - k + 1}{N}$$

and variance

$$\text{var}_i = n_i * p_i * (1 - p_i).$$

Since reads of different length are also independent of each other, we obtain the following formulas for the expected coverage and standard deviation of a k-mer in the read dataset:

$$\text{cov}_{\text{expected,pusm}}(k) = \sum_{i=1}^{m} n_i * p_i$$

$$\sigma(k) = \sqrt{\sum_{i=1}^{m} \text{var}_i(k)}$$

3.1.2. Linear Genome

Computing the expected coverage of a k-mer in a linear genome is slightly more complicated. Since the genome is of finite length, the expected number of reads which cover the k-mer depends on the position of the k-mer in the genome.

A linear genome of size N has positions $0, \ldots, N-1$. There are thus $N-l+1$ possibilities to map a read of length l to the genome. There are also $N-k+1$ possibilities to choose a k-mer in the genome. In order to count the number of possibilities that a read covers a k-mer, we introduce the following notation. Let $\mathrm{k_{start}}$ ($\mathrm{r_{start}}$) and $\mathrm{k_{end}}$ ($\mathrm{r_{end}}$) be the position of the k-mer's (read's) first and last base in the genome, respectively. For the read to cover the k-mer, it must hold that

$$(\mathrm{r_{start}} \leq \mathrm{k_{start}}) \text{ and } (\mathrm{r_{end}} \geq \mathrm{k_{end}})$$

For a read of length l, it holds that $\mathrm{r_{end}} = \mathrm{r_{start}} + l - 1$. It further holds that $\mathrm{k_{end}} = \mathrm{k_{start}} + k - 1$. Thus, we can rewrite the above formula as

$$(r_{\mathrm{start}} \leq k_{\mathrm{start}}) \text{ and } (r_{\mathrm{start}} \geq k_{\mathrm{start}} + k - l)$$

Since the genome is linear, it must further hold that $0 \leq \mathrm{r_{start}} \leq N - l$ (see Figure 3.3). This gives us

$$\mathrm{r_{start}} \in [\max\{\mathrm{k_{start}} + k - l, 0\}, \min\{\mathrm{k_{start}}, N - l\}]$$

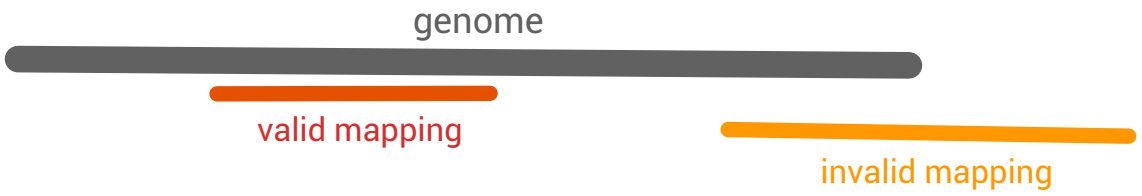


Figure 3.3.: The first base of a read can not be mapped to any position in the linear genome.

Thus, we have $\min\{\mathrm{k_{start}}, N-l\} - \max\{\mathrm{k_{start}} + k - l, 0\}$ possible mappings of a read with length l that cover the k-mer. As the k-mer is equally likely to start at any of the $N-k+1$ possible positions in the genome, we obtain

$$\mathbb{P}(\text{read of length } l \text{ covers } k\text{-mer}) = \frac{1}{N-k+1} * \sum_{k_{\mathrm{start}}=0}^{N-k} \frac{\min\{k_{\mathrm{start}}, N-l\} - \max\{k_{\mathrm{start}} + k - l, 0\}}{N-l+1}$$

It holds that

$$\max\{k_{\mathrm{start}} + k - l, 0\} = \begin{cases} k_{\mathrm{start}} + k - l & , \text{ if } k_{\mathrm{start}} \geq l - k \\ 0 & , \text{ else} \end{cases}$$

and

$$\min\{k_{\mathrm{start}}, N-l\} = \begin{cases} k_{\mathrm{start}} & , \text{ if } k_{\mathrm{start}} \leq N - l \\ N - l & , \text{ else} \end{cases}$$

We can thus rewrite the right side of the equation depending on whether $l - k \leq N - l$ or not. To simplify the notation, we define $\alpha := \frac{1}{(N-k+1)*(N-l+1)}$.

- **Case 1:** $l - k \leq N - l$

  The probability of a read of length l to cover the k-mer is

$$\alpha * \left(\sum_{k_{\text{start}}=0}^{l-k} k_{\text{start}} + \sum_{k_{\text{start}}=l-k+1}^{N-l} (l-k) + \sum_{k_{\text{start}}=N-l+1}^{N-k} (N - k_{\text{start}} - k) \right)$$

  which can be rewritten as

$$\alpha * \left(\sum_{i=1}^{l-k} i + \sum_{i=l-k+1}^{N-l} (l-k) + \sum_{i=N-l+1}^{N-k} (N-k) - \sum_{i=1}^{l-k} (i + N - l) \right).$$

  This can be further transformed to

$$\alpha * \left(\sum_{i=1}^{l-k} i + \sum_{i=l-k+1}^{N-l} (l-k) + \sum_{i=1}^{l-k} (N-k) - \sum_{i=1}^{l-k} (N-l) - \sum_{i=1}^{l-k} i \right).$$

  By applying the Gaussian sum formula $\sum_{i=1}^{n} i = \frac{n^2+n}{2}$ as well as the sum formula $\sum_{i=a}^{b} c = (b - a + 1) * c$, we can further simplify this expression to

$$\alpha * (l-k) * (N-k)$$

- **Case 2:** $l - k > N - l$

  The probability of a read of length l covering the k-mer is

$$\alpha * \left(\sum_{k_{\text{start}}=0}^{N-l} k_{\text{start}} + \sum_{k_{\text{start}}=N-l+1}^{l-k} (N-l) + \sum_{k_{\text{start}}=l-k+1}^{N-k} (N - k_{\text{start}} - k) \right)$$

  which can be rewritten as

$$\alpha * \left(\sum_{i=1}^{N-l} i + \sum_{i=N-l+1}^{l-k} (N-l) + \sum_{i=l-k+1}^{N-k} (N-k) - \sum_{i=1}^{N-l} (i - l + k) \right).$$

  This can be further transformed to

$$\alpha * \left(\sum_{i=1}^{N-l} i + \sum_{i=N-l+1}^{l-k} (N-l) + \sum_{i=l-k+1}^{N-k} (N-k) - \sum_{i=1}^{N-l} (k-l) - \sum_{i=1}^{N-l} i \right).$$

  By applying the Gaussian sum formula $\sum_{i=1}^{n} i = \frac{n^2+n}{2}$ as well as the sum formula $\sum_{i=a}^{b} c = (b - a + 1) * c$, we can further simplify this expression to

$$\alpha * (l-k) * (N-k)$$

As we obtain the same formula for both cases, this gives us

$$\mathbb{P}(\text{read of length } l \text{ covers } k\text{-mer}) = \frac{1}{(N-k+1)(N-l+1)} * (l-k) * (N-l).$$

As the reads are independent of each other, the expected coverage of a k-mer given n_i reads of length l_i follows a binomial distribution $\mathcal{B}(n_i, p_i)$ with probability

$$p_i = \frac{1}{(N-k+1)(N-l_i+1)} * (l_i - k) * (N - l_i)$$

and variance

$$\text{var}_i = n_i * p_i * (1 - p_i).$$

Since reads of different length are also independent of each other, we obtain the following formulas for the expected coverage and standard deviation of a k-mer in the read dataset:

$$\text{cov}_{\text{expected,pusm}}(k) = \sum_{i=1}^{m} n_i * p_i$$

$$\sigma(k) = \sqrt{\sum_{i=1}^{m} \text{var}_i(k)}$$

3.2. The observed Coverage of a k-mer

There are three ways to count $\text{cov}_{\text{observed}}(k\text{-mer})$:

1. The number of reads whose mappings to the genome cover the k-mer.
2. The number of exact matches of the k-mer in the read dataset.
3. The number of approximate matches of the k-mer in the read dataset, taking into account potential errors by using the a statistical error model, that is, the error profile.

We implemented variants 1 and 2 (for details, see Section 7.2). Each variant has its advantages and disadvantages. Variant 1 requires a reference genome. Thus, we can not use this variant for de novo sequencing datasets. Moreover, the quality of variant 1 depends on the quality of the read mapping tool used for mapping the reads to the genome. An advantage of variant 1 is that, given a perfect mapping of the reads to the genome, it is the most accurate way to count the coverage of a k-mer in the read dataset.

While variant 2 is fast to compute, the counts obtained by this variant ignore occurrences of sequences that are highly similar to the k-mer in question. Since reads contain errors, some potential matches of the k-mer are thus not counted. Variant 2 works best if the reads have a very low error rate.

Variant 3 relies on an accurate error profile. As the inexact matches have also to be counted, this method has a higher computational cost than variant 2. The advantage of variant 3 is that it does take into account errors in reads without requiring a reference genome.

3.3. Estimating the Coverage Bias

Let $\text{cov}_{\text{genome}}(k\text{-mer})$ be the number of occurrences of a given k-mer in the genome. Let $\text{cov}_{\text{expected,pusm}}(k\text{-mer})$ be the expected coverage of the k-mer in the PUSM. Since the PUSM assumes that the k-mer occurs only once in the genome, we multiply the expected coverage from the PUSM by $\text{cov}_{\text{genome}}(k\text{-mer})$ to obtain a new expected value

$$\text{cov}_{\text{expected}}(k\text{-mer}) = \text{cov}_{\text{expected,pusm}}(k\text{-mer}) * \text{cov}_{\text{genome}}(k\text{-mer}).$$

Let $\text{cov}_{\text{observed}}(k\text{-mer})$ be the observed coverage of the k-mer in the read dataset. We define the bias of a k-mer as

$$bias(k\text{-mer}) = \frac{\text{cov}_{\text{observed}}(k\text{-mer})}{\text{cov}_{\text{expected}}(k\text{-mer})}.$$

For example, if we count 20 occurrences of the k-mer in the read dataset while we would only expect 10 occurrences, the bias of the k-mer is 2 as it occurs twice as often.

It is known that the main factor leading to coverage bias is the G/C-content gc(k-mer) of a k-mer. It is defined as

$$\text{gc}(k\text{-mer}) := \frac{\text{Number of G or C bases in the } k\text{-mer}}{\text{Total number of bases in the } k\text{-mer}}$$

We assume throughout this thesis that the coverage bias factor only depends on the G/C-content of a specific k-mer, that is

$$bias(k\text{-mer}) := bias(\text{gc}(k\text{-mer})).$$

Using a similar approach as in the EDAR paper by Zhao *et al.* [ZPB+10], we obtain these coverage bias factors by calculating the median coverage bias over all k-mers of a given length and G/C-content (see Algorithm 3.1).

In read error correction, we want k to be large enough to ensure that a k-mer is likely to appear only once in the genome. We follow the suggestion by Kelley *et al.* [KSS10] for choosing the minimum value for k:

$$k_{\min} \approx \frac{\log(200 * N)}{\log(4)}, \text{ where } N \text{ is the estimated genome size.}$$

We further ensure that $k_{\min}$ is an odd number, because in this case the reverse-complement of a k-mer can not be the same as the k-mer itself. This is also beneficial for read error correction as it simplifies counting the number of occurrences of the k-mer. As $k_{\min}$ is the size of most k-mers we expect to use during error correction, we use $k_{\min}$ for learning the coverage bias factors.

This way, we obtain median coverage bias factors $bias(gc)$ for $gc = i * \frac{1}{k_{\min}}, i = 0, \ldots, k_{\min}$. In case we do not encounter any $k_{\min}$-mer with G/C-content $j * k_{\min}$ for a $j \in \{0, 1, \ldots, k_{\min}\}$, we interpolate its value by using the remaining inferred coverage bias factors.

Algorithm 3.1: medianCoverageBias

Input: A number $k_{\min}$
Output: Median coverage bias factors based on GC-content

```
allBiases ← vector of size k_min + 1 of vectors holding double values
gcStep ← 1 / k_min
for each k-mer of size k_min do
    if cov_observed(k-mer) > 0 and cov_genome(k-mer) > 0 then
        cov_expected(k-mer) ← cov_expected,pusm(k-mer) * cov_genome(k-mer)
        gc ← GC-content of the k-mer
        bias ← cov_observed(k-mer) / cov_expected(k-mer)
        allBiases[gc].add(bias)
for i = 0, ... k_min do
    gc ← i * gcStep
    bias(gc) ← computeMedian(allBiases[gc])
```

3.3.1. Run-dependent Coverage Bias

Given a genome re-sequencing dataset, we can determine $\text{cov}_{\text{genome}}(k\text{-mer})$ by counting the k-mer in the genome. A *genome re-sequencing dataset* is a dataset where a known reference genome has been sequenced again.

While we exactly know $\text{cov}_{\text{genome}}(k\text{-mer})$ in a genome re-sequencing dataset, we have to guess this value in case the genome is not known a-priori. Remember that we need $\text{cov}_{\text{genome}}(k\text{-mer})$ in order to compute $\text{cov}_{\text{expected}}(k\text{-mer})$. Since we have chosen $k := k_{\text{min}}$ such that the k-mer is likely to occur exactly once in the genome, we can assume $\text{cov}_{\text{genome}}(k\text{-mer}) = 1$. This gives us

$$\text{cov}_{\text{expected}} = \text{cov}_{\text{expected, pusm}}(k\text{-mer}) * \text{cov}_{\text{genome}}(k\text{-mer}) = \text{cov}_{\text{expected, pusm}}(k\text{-mer}).$$

Since we do not know the reference genome in a *de novo* sequencing dataset, we can not exactly determine whether $\text{cov}_{\text{genome}}(k\text{-mer}) > 0$ or not, this is, whether the k-mer is correct or not. Unfortunately, erroneous k-mers decrease the estimated median coverage bias values as their observed coverage is very low. To minimize this effect, we change line 4 in Algorithm 3.1 to only consider k-mers with an observed coverage $\text{cov}_{\text{observed}}(k\text{-mer}) \geq 0.2 * \text{cov}_{\text{expected, pusm}}(k\text{-mer})$. As a consequence, we can not detect a median coverage bias factor lower than 0.2 by this approach.

Since the coverage bias factors influence the k-mer classification (a k-mer can either be `UNTRUSTED`, `TRUSTED` or `REPETITIVE`, as discussed in Chapter 4) and thus also the error correction, we need to update the k-mer classification and error correction after re-inferring the coverage bias, accordingly.

3.3.2. Answering Coverage Bias Queries

If we want to estimate the coverage bias for a k-mer with a different value of k than used in the estimation, we might have a different gc value for its GC-content. Thus, we use linear interpolation to estimate the coverage bias factor for a GC-content gc. Let gc_{min} be the largest GC-content $\leq$ gc for which we know $bias(\text{gc}_{\text{min}})$. Let gc_{max} be the smallest GC-content $\geq$ gc for which we know $bias(\text{gc}_{\text{max}})$. We then define $bias(\text{gc})$ as

$$bias(\text{gc}) := bias(\text{gc}_{\text{min}}) + \frac{bias(\text{gc}_{\text{max}}) - bias(\text{gc}_{\text{min}})}{\text{gc}_{\text{max}} - \text{gc}_{\text{min}}} * (\text{gc} - \text{gc}_{\text{min}})$$

4. K-mer Classification Unit

Own Contributions

- Classify k-mers based on their expected count under the Perfect Uniform Sequencing Model and their bias-corrected observed count in the read dataset.
- Classify k-mers based on their bias-corrected Z-Scores, by either applying a statistical test or training a machine learning classifier.

In this chapter, we want to develop a method for classifying a k-mer as being either erroneous (`UNTRUSTED`), unique in the genome (`TRUSTED`), or occurring more than once in the genome (`REPETITIVE`). The purpose of this classification is to only correct untrusted k-mers. A k-mer classified as `REPETITIVE` indicates that the value k needs to be increased in order to avoid mis-corrections.

Given a reference genome, we can simply count the number of occurrences $\text{cov}_{\text{genome}}(k\text{-mer})$ of a given k-mer. Then, the k-mer type is

$$type(k\text{-mer}) = \begin{Bmatrix} \texttt{UNTRUSTED} & , \text{ if } \text{cov}_{\text{genome}}(k\text{-mer}) = 0 \\ \texttt{TRUSTED} & , \text{ if } \text{cov}_{\text{genome}}(k\text{-mer}) = 1 \\ \texttt{REPETITIVE} & , \text{ if } \text{cov}_{\text{genome}}(k\text{-mer}) > 1 \end{Bmatrix}.$$

In de novo sequencing, the reference genome is unknown. Thus we need to adapt a different approach to classify a k-mer. Using the Coverage Bias Unit (Chapter 3), we obtain a bias-corrected observed count for every k-mer in the read dataset:

$$\text{cov}_{\text{bias-corrected}}(\text{k-mer}) = \frac{1}{bias(k\text{-mer})} * \text{cov}_{\text{observed}}(\text{k-mer})$$

By the Perfect Uniform Sequencing Model (Section 3.1), we further obtain the expected coverage $\text{cov}_{\text{expected, pusm}}(k\text{-mer})$ and standard deviation σ of a k-mer in an ideal setting, assuming that the k-mer is unique in the genome.

We develop two approaches that use $\text{cov}_{\text{bias-corrected}}(k\text{-mer})$ and $\text{cov}_{\text{expected,pusm}}(k\text{-mer})$ to estimate the k-mer type for each k-mer. The first approach compares the bias-corrected, observed coverage with the expected coverage. The second approach also takes the standard deviation σ into account, using the *Z-Score*.

4.1. Naïve Classification

For easier reading, we define $\mu := \text{cov}_{\text{expected, pusm}}(k\text{-mer})$ and $x := \text{cov}_{\text{bias-corrected}}(k\text{-mer})$. In the naïve approach, we deploy the same idea for estimating $\text{cov}_{\text{genome}}(k\text{-mer})$ as in Section 3.3.1. This yields the following rule for classifying a k-mer (see Figure 4.1):

$$type(k\text{-mer}) = \begin{cases} \texttt{UNTRUSTED} & , \text{ if } x < 0.5 * \mu \\ \texttt{TRUSTED} & , \text{ if } 0.5 * \mu \leq x \leq 1.5 * \mu \\ \texttt{REPETITIVE} & , \text{ if } x > 1.5 * \mu \end{cases}$$

The idea behind this rule is:

If $\text{cov}_{\text{genome}}(k\text{-mer}) = 0$, we would expect $\text{cov}_{\text{bias-corrected}}(k\text{-mer}) = 0$. If $\text{cov}_{\text{genome}}(k\text{-mer}) = 2$, we would expect $\text{cov}_{\text{bias-corrected}}(k\text{-mer}) = 2\mu$. The value $\frac{\mu}{2}$ lies exactly between 0 and μ and the value $\frac{3\mu}{2}$ lies exactly between μ and 2μ.

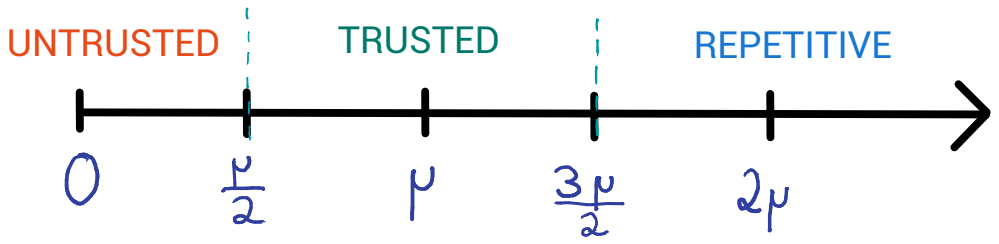


Figure 4.1.: Naïve classification of a k-mer, using only the expected count $\mu = \text{cov}_{\text{expected, pusm}}(k\text{-mer})$ and the bias-corrected observed count of a k-mer.

4.2. Z-Score based Classification

The *Z-Score* (also called *standard score*) represents the distance of a random variable X from its expected value $\mu = E(X)$ in units of its standard deviation σ. It is defined as:

$$Z = \frac{X - \mu}{\sigma}$$

The Z-Score is mainly used for standardizing random variables that follow a normal distribution. For a rationale for assuming a normal distribution, see below.

4.2.1. The Z-Score of a k-mer

We compute the Z-score of a k-mer by using $\text{cov}_{\text{bias-corrected}}(k\text{-mer})$ as well as its expected coverage $\text{cov}_{\text{expected, pusm}}(k\text{-mer})$ and standard deviation σ under the Perfect Uniform Sequencing Model (PUSM) from Section 3.1. Recall that, the PUSM assumes that the k-mer occurs only once in the genome. As the PUSM relies on binomial distribution to compute the expected coverage of a k-mer, this motivates our use of Z-Scores.

We define the (bias-corrected) Z-score of a k-mer as

$$Z(k\text{-mer}) = \frac{\text{cov}_{\text{bias-corrected}}(k\text{-mer}) - \text{cov}_{\text{expected,pusm}}(k\text{-mer})}{\sigma}.$$

We develop two different variants for classifying a k-mer using its Z-score. Variant 1 uses a statistical test to classify a k-mer. Variant 2 trains a machine learning classifier.

4.2.2. Variant 1: Statistical Testing

We conduct a hypothesis test with

$$\mathcal{H}_0 : \text{cov}_{\text{genome}}(k\text{-mer}) = 1$$

$$\mathcal{H}_1 : \text{cov}_{\text{genome}}(k\text{-mer}) \neq 1$$

We classify a k-mer as `TRUSTED`, *if and only if*, $\mathcal{H}_0$ is accepted. If the hypothesis $\mathcal{H}_0$ is true, it holds that $\text{cov}_{\text{expected}}(k\text{-mer}) = \text{cov}_{\text{expected, pusm}}(k\text{-mer})$. As the PUSM uses a binomial distribution which can be approximated by a normal distribution, 95.4% of the values are expected to lie within $\pm 2\sigma$ around the mean of the distribution (see Figure 4.2). If the coverage bias factor used for inferring the bias-corrected observed count was perfect, we would be able to correctly classify more than 95% of the trusted k-mers.

To classify a k-mer as being either `UNTRUSTED`, `TRUSTED`, or `REPETITIVE`, we thus apply the following rule:

$$type(k\text{-mer}) = \begin{Bmatrix} \texttt{UNTRUSTED}, & \text{if } Z(k\text{-mer}) < -2 \\ \texttt{TRUSTED}, & \text{if } -2 \leq Z(k\text{-mer}) \leq 2 \\ \texttt{REPETITIVE}, & \text{if } Z(k\text{-mer}) > 2 \end{Bmatrix}$$

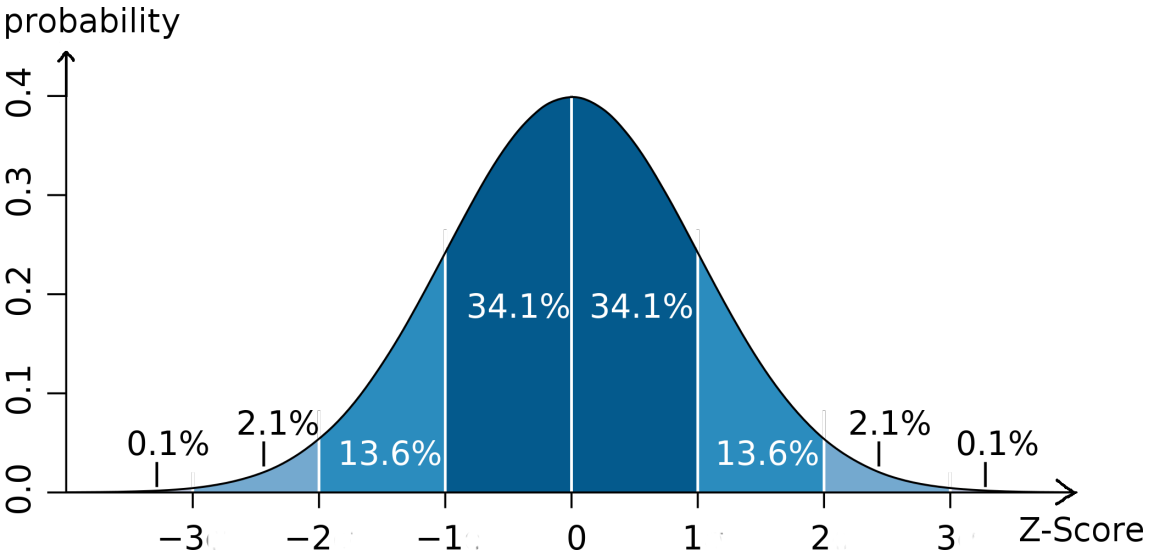


Figure 4.2.: Z-Score of a standard normal distribution. 95.4% of the data points lie within $\pm 2\sigma$ around the mean and thus have a Z-score $-2 \leq Z \leq 2$. Image taken from https://upload.wikimedia.org/wikipedia/commons/thumb/8/8c/Standard_deviation_diagram.svg/2000px-Standard_deviation_diagram.svg.png, axis labels manually added.

4.2.3. Variant 2: Machine Learning

Using the Python script for automatic classifier selection from Section 7.1, we calculate a machine learning based k-mer classification. For this chapter, it suffices to know that the Python script automatically chooses among different machine learning classifiers and selects the classifier which performs best in terms of average F-Score (see Section 2.4.1). We train the classifiers in the script by using a genome re-sequencing dataset (this is a dataset where the reference genome is known and thus also $\text{cov}_{\text{genome}}(k\text{-mer})$).

We use the following features for classification:

- Z-score of the k-mer
- GC-content of the k-mer
- Size of the k-mer
- Observed count $\text{cov}_{\text{observed}}(k\text{-mer})$
- Bias-corrected observed $\text{cov}_{\text{bias-corrected}}(k\text{-mer})$
- Expected count $\text{cov}_{\text{expected,pusm}}(k\text{-mer})$

5. Error Profile Unit

Own contributions

- Infer a technology-specific error profile from genome re-sequencing data or from a set of corrected reads.
- For a presumably erroneous position in a read, compute two rankings of specific errors that can occur at (or directly after, in case of deleted bases) this position:
  - One ranking for the error being an insertion of one base, a substitution of an A, a substitution of a C, a substitution of a G, or a substitution of a T.
  - One ranking for the error being a deletion of an A, a deletion of a C, a deletion of a G, a deletion of a T, or a deletion of multiple bases.
- Devise three variants for inferring the error profile, which differ by taking into account a different parts of the context of a specific error.

The purpose of computing a technology-specific error profile is to rank the possible errors that can occur at a position i in a read by their likelihood of occurrence. This way, we can use these rankings during error correction, and choose among multiple correction candidates.

For computing this ranking, we assume that the position i in the read is erroneous. A position can be erroneous either because the base at this position is an inserted base, a substitution of an A, a substitution of a C, a substitution of a G, or a substitution of a T. A position can also be erroneous due to a deletion error (a deletion of an A, a deletion of a C, a deletion of a G, a deletion of a T, or a deletion of multiple bases) occurring on the right of the base at position i. This is, there may be missing bases between position i and position $i + 1$ in the read (see Figure 5.1).

Since an erroneous position i can be caused by an erroneous base as well as by one or more missing bases, or by both, we compute two rankings for the position: One ranking concerning the base at position i (*base-errors*) and one ranking concerning an artificial "gap" between positions i and $i + 1$ in the read (*gap-errors*).

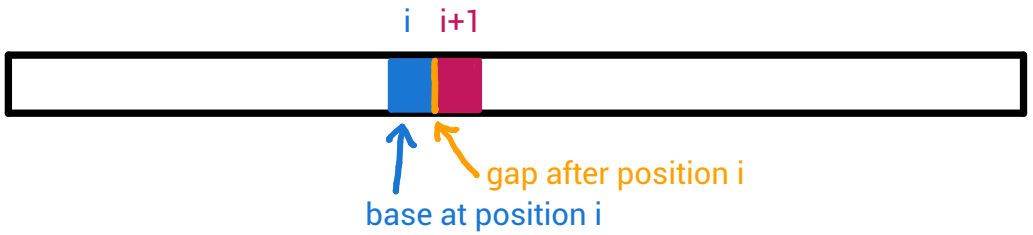


Figure 5.1.: A deletion error may have occurred between the positions i and $i+1$ in the read. As a deletion results in missing bases, they are not present in the read. Thus, we associate them with an artificial "gap" on the right of the base at position i.

In order to be able to compute these rankings, we infer the technology-specific error profile by counting the frequency of specific errors in a pre-corrected training dataset. For obtaining this pre-corrected dataset, we can either use a genome re-sequencing dataset (see Section 7.3) or a previous correction run (see Chapter 6). Each specific error has a context (see Figure 5.2).

The *context* of an error consists of the position i in a read where it occurred, the base at position i, the quality score (from the sequencing machine) of the base at position i, the read length, and various motifs. A *motif* is a short ($\approx$ 3 to 6 bases) sequence of bases surrounding the erroneous position in the read.

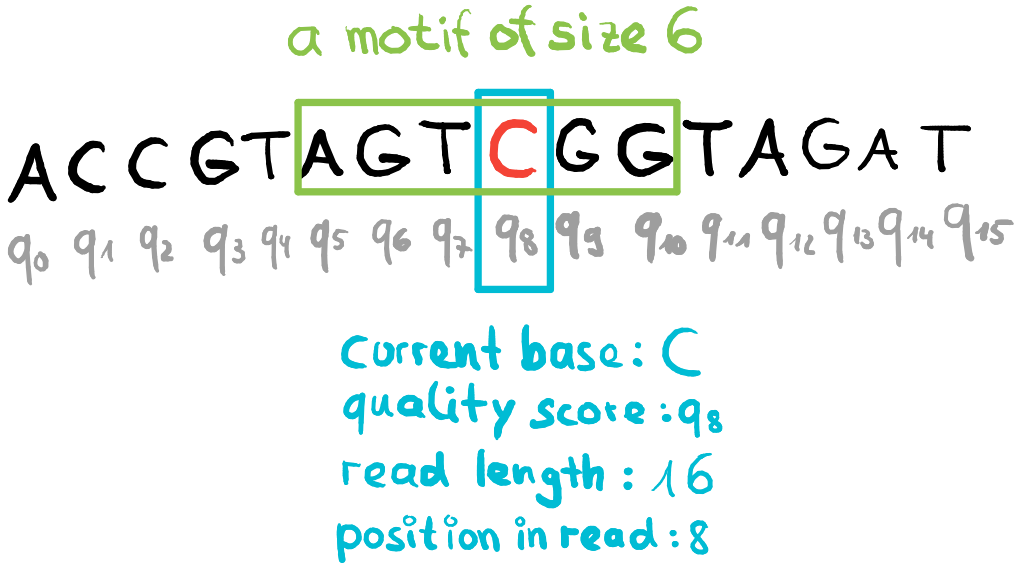


Figure 5.2.: An insertion error at position $i = 8$ and its context. For easier presentation, only one of the motifs surrounding the erroneous position is shown.

We devise three approaches for inferring the error profile, taking various aspects of the error context (which may increase or decrease the likelihood of a specific error) into account:

- **Context-Free Error Profile:** Only consider the base occurring at the erroneous position. Compute an overall error rate as well as the relative frequencies of specific error types in the training dataset.
- **Sequence-Specific Error Profile:** Only consider the motifs, this is, the bases surrounding an erroneous position in a read.
- **Full-Context-Specific Error Profile:** Consider the entire context of an erroneous position.

5.1. Context-Free Error Profile

Using the relative frequencies of specific error types in the training dataset, we estimate overall error probabilities. We only take the current base of an erroneous position into account, ignoring the rest of the error context.

Inferring the Context-Free Error Profile

We estimate the probability of a specific error type by computing its relative frequency in the training dataset:

$$\mathbb{P}(\text{error type}) \approx \frac{\text{number of errors of this error type in the training data}}{\text{total number of bases in the training data}}$$

Regarding substitution errors, we infer a substitution rate matrix by computing the relative frequencies of the substitutions A→C, A→G, A→T, A→N, C→A, C→G, C→T, C→N, G→A, G→C, G→T, G→N, T→A, T→C, T→G, and T→N in the training data. For example,

$$\mathbb{P}(\text{A} \to \text{C}) \approx \frac{\text{number of A} \to \text{C substitution errors in the training data}}{\text{total number of bases in the training data}}$$

We additionally store overall base-related and overall deletion-related error rates. These error rates can be used to infer the probability of a base being correct and the probability of an artificial gap to contain no deleted bases.

$$\text{base-error rate} := \frac{\text{number of insertion and substitution errors in the training data}}{\text{total number of bases in the training data}}$$

$$\text{deletion-error rate} := \frac{\text{number of deletion (of one or more bases) errors in the training data}}{\text{total number of bases in the training data}}$$

Computing error-specific Rankings

The context-free error profile computes the error-specific rankings in the following way: Given a (presumably erroneous) position i in a read and the read itself, we rank the specific error types by their frequency of occurrence in the training data.

If, for example, the base in the read at position i is a C, we compute the base-error-ranking by sorting the estimated probabilities $\mathbb{P}(\text{insertion})$, $\mathbb{P}(\text{substitution A}\to\text{C})$, $\mathbb{P}(\text{substitution G}\to\text{C})$, and $\mathbb{P}(\text{substitution T}\to\text{C})$. We compute the gap-error ranking by sorting the estimated probabilities $\mathbb{P}(\text{deletion of an A})$, $\mathbb{P}(\text{deletion of a C})$, $\mathbb{P}(\text{deletion of a G})$, $\mathbb{P}(\text{deletion of an T})$, and $\mathbb{P}(\text{deletion of multiple bases})$.

5.2. Sequence-Specific Error Profile

In the sequence-specific error profile, we want to detect motifs that influence the likelihood of a specific error type at a presumably erroneous position i in a read. For example, it is known that a substitution error is more likely to occur after a preceding GGC motif in an Illumina read [SID+15]. This is, directly to the left of the erroneous position, we have the bases GGC in the read.

For detecting those motifs influencing the likelihood of a specific error type, we compute Z-Scores for all motifs surrounding a specific error type in the training dataset. We say that a motif surrounds an error type, *if and only if*, there exists an error in the training dataset of the given type that is surrounded by this motif.

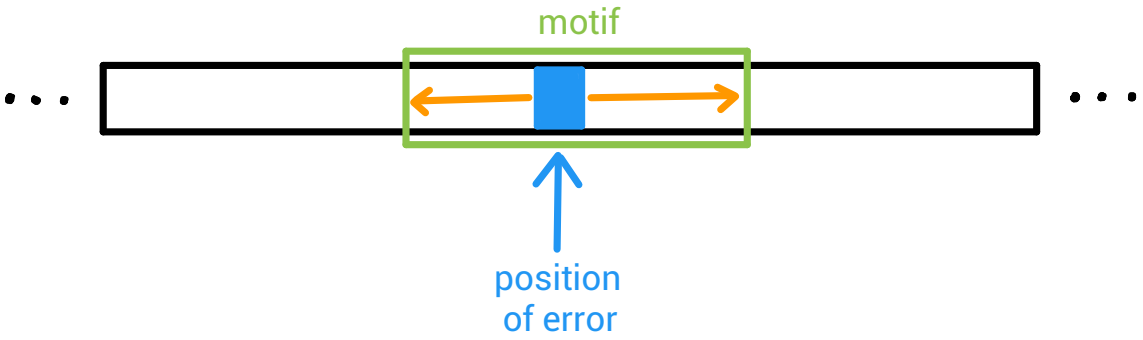


Figure 5.3.: A motif is a characteristic, short ($\approx$ 3 to 6 bases) subsequence in a read which surrounds an erroneous position.

A motif with a Z-Score of 0 for a specific error type indicates that the motif does not influence the likelihood of this error type to occur. A negative Z-Score indicates that the error type is unlikely to occur within the motif. Similarly, a positive Z-Score indicates that the error type is likely to occur within the motif. Thus, given multiple motifs surrounding a presumably erroneous position, the most relevant motif is the one with the highest absolute Z-Score.

Inferring the Sequence-Specific Error Profile

I used the approach from Shin and Park [SP16] to detect surrounding bases of position i that increase or decrease the likelihood of an error to occur. Shin and Park only consider motifs $w_{i-j}...w_i...w_{i+j}$ where the error occurs at position i, that is, exactly in the middle of the motif. We discard this constraint and allow the error to occur at *any* position within the motif. This allows us to also detect relevant preceding or following bases that lead to an increased or decreased likelihood of the error to occur. By using Z-Scores, we can detect the smallest motifs that explain biases towards specific error types.

The Z-Score of a motif $w_1w_2\dots w_m$ is defined as in [SP16]:

$$Z(w_1w_2\dots w_m) = \frac{N(w_1w_2\dots w_m) * E(w_1w_2\dots w_m)}{\sqrt{\text{var}(w_1w_2\dots w_m)}}$$

where $N(w_1w_2\dots w_m)$ denotes how often the motif $w_1w_2\dots w_m$ surrounds a specific error type in the read dataset and the expected count of the motif is

$$E(w_1w_2\dots w_m) = \frac{N(w_1w_2\dots w_{m-1}) * N(w_2w_3\dots w_m)}{N(w_2w_3\dots w_{m-1})}.$$

For easier reading, we define $left := w_1w_2\dots w_{m-1}$, $inner := w_2w_3\dots w_{m-1}$, and $right := w_2w_3\dots w_m$. Shin and Park approximate $\text{var}(w_1w_2\dots w_m)$ by

$$\text{var}(w_1w_2\dots w_m) \approx E(w_1w_2\dots w_m) * \frac{(N(inner) - N(left)) * (N(inner) - N(right))}{(N(inner))^2}.$$

A detailed derivation of the above formulas can be found in the paper by Schbath *et al.* [SPdT95].

It has to be emphasized that $N(\cdot)$ from the above formulas does **not** always refer to the observed coverage $\text{cov}_{\text{observed}}$ of a motif in the read dataset. Instead, we have to distinguish between three cases. Note that this case distinction is not required in the original approach by Shin and Park as they only consider Case 1.

Let $[E]$ be an error type. $[E]$ can be an insertion of a base, one of the 16 substitution types (A→C, A→G, A→T, A→N, C→A, C→G, C→T, C→N, G→A, G→C, G→T, G→N, T→A, T→C, T→G, or T→N), a deletion of an A, a deletion of a C, a deletion of a G, a deletion of a T, or a deletion of multiple bases. Let $w_1 w_2 \dots w_m$ be a motif surrounding the error type $[E]$. If $[E]$ occurs at position p in the motif, we write $[E]_{w_p}$ instead of w_p.

- **Case 1:** The error occurs at position $2 \leq p \leq m-1$.

  In order to compute $N(w_1 w_2 \dots w_m)$, we have to count how often we encounter $w_1 \dots w_{p-1}[E]_{w_p} w_{p+1} \dots w_m$ in the read dataset. This means that for each occurrence of the error $[E]$ at a base w_p, we have to check whether it is surrounded by a prefix $w_1 \dots w_{p-1}$ and a suffix $w_{p+1} \dots w_m$. The argument is analogous for $N(left)$, $N(inner)$, and $N(right)$.

- **Case 2:** The error occurs at position $p = 1$.

  We compute $N(left)$ as in case 1. However, since we have the situation that the motif contains no bases to the left of the error, i.e., $[E]_{w_1} w_2 \dots w_m$, it holds that $N(inner) = \text{cov}_{\text{observed}}(inner)$ and $N(right) = \text{cov}_{\text{observed}}(right)$.

- **Case 2:** The error occurs at position $p = m$.

  We compute $N(right)$ as in case 1. However, since we have the situation that the motif contains no bases to the right of the error, i.e., $w_1 \dots w_{m-1}[E]_{w_m}$, it holds that $N(left) = \text{cov}_{\text{observed}}(left)$ and $N(inner) = \text{cov}_{\text{observed}}(inner)$.

Compute the error-specific Rankings

Given a (presumably erroneous) position i in a read and the read itself, we rank the specific error types by the Z-scores of their most relevant motifs, this is, the motifs with the highest absolute Z-scores for the error types.

For example, given the read ACGGTAGC**G**TAGGCATTAG and the position $i = 8$ (this is base G, marked in bold), we compute the most relevant motif for an insertion error by comparing the Z-scores of all motifs surrounding this base in the read. We consider motifs of sizes 3 up to 6. For example, the motifs of size 3 surrounding the base at position 8 of the read are CG**G**, C**G**T, and **G**TA. In total, we have to compare $3 + 4 + 5 + 6 = 18$ Z-scores for this position in the read and this error type.

All in all, we compute the base-error-ranking in this example by sorting the Z-Scores Z(insertion), Z(substitution A→G), Z(substitution C→G), and Z(substitution T→G) for the respective most relevant motifs surrounding the **G** base at position 8 in the read.

Analogously, we compute the gap-error ranking by sorting the Z-Scores Z(deletion of an A), Z(deletion of a C), Z(deletion of a G), Z(deletion of an T), and Z(deletion of multiple bases) for the respective most relevant motifs surrounding the **G** base at position 8 in the read.

5.3. Full-Context-Specific Error Profile

In the full-context-specific error profile, we consider the entire context of an erroneous position. This is, the position i in a read where the error occurred, the base at position i, the quality score of the base at position i, the read length, and various motifs surrounding the erroneous position in the read.

Unfortunately, the exact configuration (error type, position, quality score, read length, motifs) occurs very rarely in the training dataset (probably only once). Thus, we cannot

estimate its distribution by counting frequencies. Instead, using the Python script for automatic classifier selection from Section 7.1, we infer a machine learning based error profile. For this chapter, it suffices to know that the Python script automatically chooses among different machine learning classifiers and selects the classifier which performs best in terms of average F-Score (see Section 2.4.1).

We train classifiers, one for base-errors (this is, an insertion or a substitution) and one for gap-errors (this is, deletions of one or more bases). We use the error context as our features. To keep the feature vector at a reasonable size, we only take the most relevant motif Z-Score (this is, the Z-Score with highest absolute value) for each specific error type into account.

In order to compute the base-error and the gap-error ranking for a presumably erroneous position i in a read, we sort the class probabilities obtained by the respective classifiers.

Classifier for Base-Errors

We use the following features:

- the erroneous position i
- base at the erroneous position (A, C, G, T, or an uncalled base N)
- quality score of the erroneous position
- length of the read
- motif Z-scores with highest absolute value for each base-error type: Z(insertion), Z(substitution of an A), Z(substitution of a C), Z(substitution of a G), and Z(substitution of a T).

We use the following classes:

- insertion
- substitution of an A
- substitution of a C
- substitution of a G
- substitution of a T

Classifier for Gap-Errors (Deletions)

We use the following features:

- the erroneous position i
- base at the erroneous position (A, C, G, T, or an uncalled base N)
- quality score of the erroneous position
- length of the read
- motif Z-scores with highest absolute value for each base-error type: Z(deletion of an A), Z(deletion of an C), Z(deletion of a G), Z(deletion of a T), and Z(deletion of multiple bases).

We use the following classes:

- deletion of an A
- deletion of a C

- deletion of a G
- deletion of a T
- deletion of multiple bases

We obtain the specific motif Z-Scores (which are needed for extracting the feature vector) by the sequence-specific error profile.

6. Error Correction Unit

Own contributions

- Devise a novel error correction approach which is based on k-mers with locally adaptive sizes k. Correct substitutions, insertions, and single base deletions.
- Present an idea for correcting deletions of multiple bases, using only k-mer counts.
- Adapt to run-specific error profile characteristics by re-inferring the error profile after a first error correction run. Then, correct the reads again, discarding the previous corrections and using the updated error profile.

Our error correction procedure consists of the following steps:

1. Load pre-determined error profiles (see Chapter 5) for each sequencing technology used in the given dataset.

2. Correct the reads using these error profiles. Our current approach does only correct substitutions, insertions, and single base deletions.

3. Update/Retrain the error profiles with the read corrections. This way, we can adapt the error profile to account for the special error characteristics of the specific sequencing run at hand.

4. Discard all corrections, start again with the uncorrected reads.

5. Correct the reads again, this time using the updated error profiles.

We can recount the k-mers after an initial error correction step for obtaining improved k-mer classification results in later error correction steps.

Figure 6 (repeated from Chapter 1) shows how our components interact with each other.

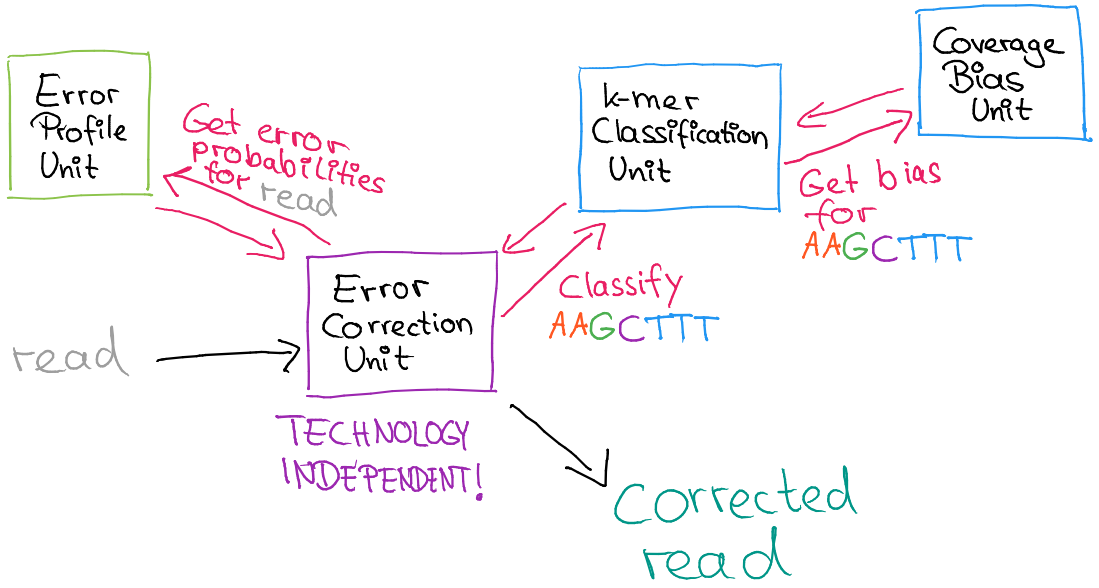
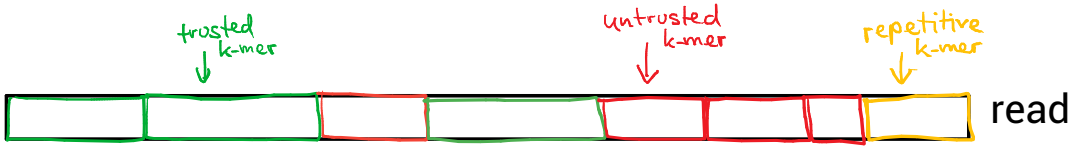


Figure 6.1.: Structure of the system. By encapsulating the technology-specific influences (e.g. error profile and coverage bias) into separate modules, the error correction and k-mer classification algorithms can remain technology-agnostic.

6.1. Correcting a Read

Our variable-length k-mer based method consists of three steps:

1. Cover the read with k-mers (locally adapted choice of k). For each of these k-mers, use the smallest size k such that the k-mer is classified as `TRUSTED` or `UNTRUSTED` (as in Chapter 4).



2. Add overlapping k-mers in order to also detect previously unidentified deletion errors.

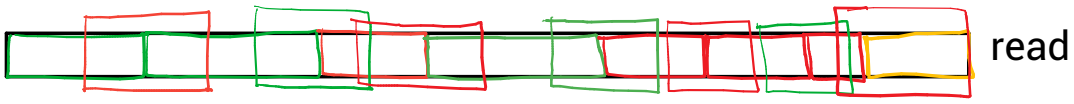


Figure 6.2.: Redundant covering of the read with k-mers of variable size.

3. Fix `UNTRUSTED` k-mers by transforming them into `TRUSTED` k-mers. Use the error profile to select correction candidates with a higher likelihood in the error profile.

In Step 1, we want k to be large enough to ensure that a k-mer is likely to appear only once in the genome.

As in Chapter 3, We follow the suggestion by Kelley *et al.* [KSS10] for choosing the minimum value for k:

$$k_{\min} \approx \frac{\log(200 * N)}{\log(4)}, \text{ where } N \text{ is the estimated genome size.}$$

We further ensure that $k_{\min}$ is an odd number, because in this case the reverse-complement of a k-mer can not be the same as the k-mer itself. This simplifies counting the number of

occurrences of the k-mer in the read dataset. If one of the k-mers in Step 1 gets classified as `REPETITIVE` by the K-mer Classification Unit 4, we extend the k-mer. We extend the k-mer by iteratively adding the two bases to the right of the k-mer in the read, until the resulting k'-mer gets classified as `TRUSTED` or `UNTRUSTED`. We can do this for all but the rightmost k-mer in the read.

A deletion can remain unidentified after Step 1, if it occurs between two (possibly extended) k-mers. In order to also detect previously unidentified deletion errors, we add additional (possibly extended) overlapping k-mers to our previous covering of the read in Step 2.

In Step 3, we aim to transform each of the `UNTRUSTED` (possibly extended) k-mers into `TRUSTED` ones, allowing for only one error per k-mer to occur. Using the Error Profile Unit 5, we obtain a ranking of error type probabilities for each position in the k-mer. We only use this ranking for deciding which error type to try first when correcting a k-mer.

6.2. Approach for Resolving Deletions of Multiple Bases

In literature, the two main approaches for resolving deletions of multiple bases consist in either ignoring them and only resolving single base deletions (i.e., a deletion of m bases can only be resolved by iteratively applying the error correction on the entire dataset m times) or building a multiple sequence alignment of reads which are believed to originate from the same genomic region as the erroneous read. Unfortunately, both approaches have a high computational cost. Thus, we do not follow any of these approaches. Instead, we introduce the following approach.

In a first step, we only detect deletions of multiple bases without trying to correct them. In these pre-corrected reads, we only mark the deletion of multiple bases by inserting a single "_" character in the sequence. In a next step, we use a dedicated algorithm that intends to resolve these multiple deletions, that is, determine their length and recover the deleted bases. If this algorithm fails to resolve a deletion of multiple bases in a read, it breaks up the read into smaller parts.

Given a pre-corrected read with a deletion of multiple bases marked by the character "_", we resolve the deletion of multiple bases by iteratively extending k-mers from both sides of it. We *extend* a k-mer by adding a base to the k-mer, thus obtaining a $k+1$-mer.

For each of the extended k-mers that is classified as `TRUSTED` or `REPETITIVE`, we continue extending them until the middle k-mer region (spanning the position of the deletion of multiple bases, see Figure 6.3) is `TRUSTED`, too. Resolving a deletion of multiple bases is successful if all the left, middle and right k-mer regions are `TRUSTED`. If we can only partially resolve a deletion of multiple bases, we break the read at the position of the deletion of multiple bases. However, we keep the bases we extended within the left and right region, in case they are classified as `TRUSTED` k-mers.

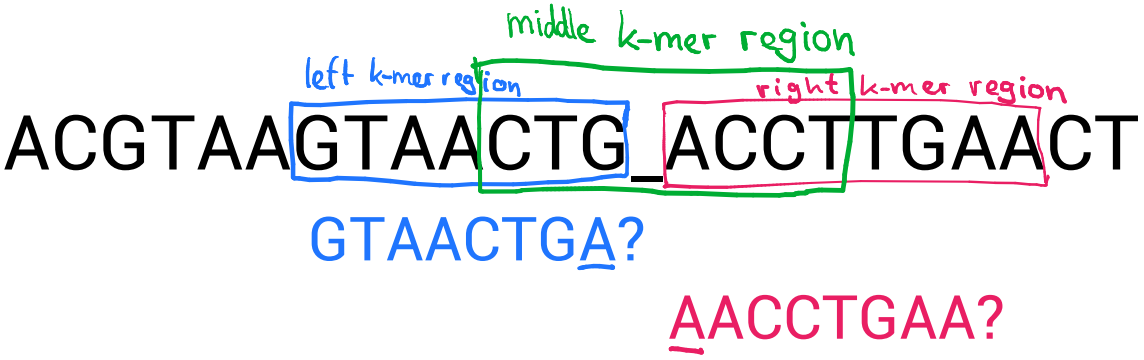


Figure 6.3.: Resolving a deletion of multiple bases in a pre-corrected read. We try to extend both the left and the right k-mer region until the middle k-mer region is classified as `TRUSTED`, as well. If this is not possible, we split the read at the position of the deletion of multiple bases, but keep possibly added bases as long as their corresponding k-mer regions are classified as `TRUSTED`.

The advantage of using this approach is that it is solely based on k-mers and requires no compute-intensive multiple sequence alignment step. Also, instead of simply trying all possible DNA sequences that could have been deleted from the read, we limit our search and hence accelerate it by only continuing k-mer extension if the previous extension already was successful.

Unfortunately, this approach was not successful in preliminary experiments. The inserted corrections were too long and did not correspond to the deleted bases. It remains further work to evaluate and improve this approach.

7. Implementation

Our framework is implemented in `C++ 14` and `Python 2.7`. For parallelizing crucial steps in the framework, such as extracting errors and correcting reads, we implemented a templated Producer-Consumer pattern which uses `std::thread`. The framework uses the `cereal` [GV17] library for serialization. For calling Python code from inside the C++ code, we use the `Python/C API` (`https://docs.python.org/2/c-api/`). We use the `samtools` [LHW+09] toolkit as well as the `SeqAn` [DWRR08] framework for reading and modifying files in the `FASTA`, `FASTQ`, and `SAM` format. For plotting the median coverage bias factors, we use the `gnuplot-iostream` interface by Stahlke *et al.* [SMM+17], which unfortunately adds a dependency to the boost C++ libraries.

As by the 30.03.2017, our framework consists of approximately 6,000 lines of code. However, this number will frequently change in the future as we consider our current implementation to be only a prototype and a proof-of-concept.

7.1. Classifier Selection

In order to choose between various machine learning classification methods, we implemented a Python script that performs the following steps:

1. Take a labeled dataset as input.
2. Split it into 2/3 training and 1/3 testing data, taking into account class imbalances.
3. Train and evaluate the following classifiers: Naïve Bayes, Logistic Regression, Decision Tree, Random Forest, AdaBoost (with DecisionTree base classifier) (see Section 2.4). Due to its high computational cost required for training, the AdaBoost classifier is disabled by default.
4. Decide for the classifier that achieved the highest average F-score (see Section 2.4.1).

This script is used by the machine learning variants of the K-mer Classification Unit (Chapter 4) and the Error Profile Unit (Chapter 5).

We train and evaluate the classifiers using the `scikit-learn` [PVG+11] framework.

In our opinion, machine learning approaches constitute only a last resort, similar to using a meta-heuristic. It would be desirable to develop a proper mathematical model instead, if possible. However, real world problems are often too complex and consist of many dependent features. In our case, as we aim to provide an easy-to-extend technology-independent framework, we can not safely assume to which extent the features are correlated or not.

7.2. Counting a k-mer

We implemented two variants for determining $\text{cov}_{\text{observed}}(k\text{-mer})$:

1. The number of reads whose mappings to the genome cover the k-mer.
2. The number of exact matches of the k-mer in the read dataset.

While variant 1 is only used by the one variant of the Coverage Bias Unit (Chapter 3), variant 2 is used by all modules within our framework.

7.2.1. Observed k-mer Coverage by Read Mapping

If the reference genome is known, we can determine $\text{cov}_{\text{observed}}(k\text{-mer})$ of a k-mer by mapping all reads to the reference genome. IWe use the `BWA-MEM` alignment tool by Li [Li13] for mapping the reads. We discard the unmapped reads as well as reads that map to multiple regions in the genome. Each of the mappings we keep induces an interval $[\text{i.start}, \text{i.stop}]$.

We extend the functionality of the `IntervalTree` class by Garrison [Gar17]. We add a method `findCovering(start,stop)` that returns all intervals covering a given interval $[\text{start}, \text{stop}]$ (see Figure 7.1).

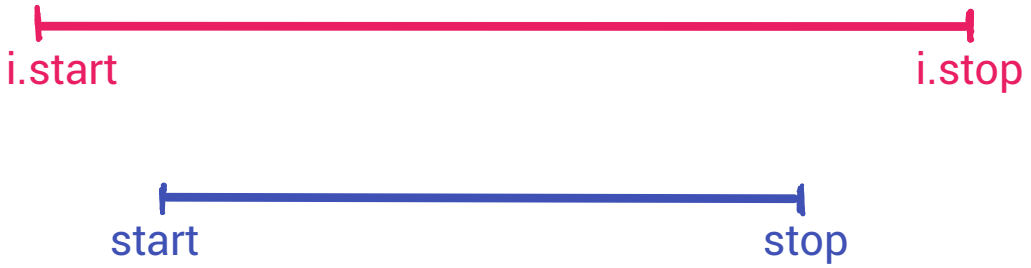


Figure 7.1.: An interval $[\text{i.start}, \text{i.stop}]$ covers the interval $[\text{start}, \text{stop}]$, if and only if $\text{i.start} \leq \text{start}$ and $\text{i.stop} \geq \text{stop}$.

In an preprocessing step, we iterate over all reads and add the aforementioned intervals to the interval tree. Since we can not use this definition of $\text{cov}_{\text{observed}}(k\text{-mer})$ in the context of de novo error correction, we can assume that we are interested in the coverage of a k-mer originating **from the reference genome** (instead of a k-mer originating from a read). However, if the k-mer would originate from a read, we could first map the read to the reference genome and then deduce its position within the genome.

Algorithm 7.1 shows how to compute $\text{cov}_{\text{observed}}(k\text{-mer})$ after the preprocessing has been done.

Algorithm 7.1: Observed Coverage by mapping

Input: An interval $[\text{start}, \text{stop}]$ representing a k-mer in the reference genome
Output: $\text{cov}_{\text{observed}}(k\text{-mer})$ by mapping

```
1 cov ← 0
2 for each interval i = [i.start, i.stop] covering [start, stop] do
3 |   cov ← cov + 1
4 return cov
```

7.2.2. Observed k-mer Coverage by Exact Matches

For counting the number of occurrences of a given k-mer in the read dataset, we need to count both the k-mer itself as well as its reverse-complement. This is because we do not know the orientation of the reads before assembly or mapping. A read can originate from either the leading strand or the lagging strand (which is the reverse-complement of the leading strand) of the genome.

Recall that the reverse-complement of a DNA sequence $s = s_1 s_2 \dots s_n \in \{\mathrm{A}, \mathrm{C}, \mathrm{G}, \mathrm{T}\}^n$ is defined as $\hat{s} := c(s_n) \dots c(s_2) c(s_1)$ with $c(\mathrm{A}) := \mathrm{T}$, $c(\mathrm{C}) := \mathrm{G}$, $c(\mathrm{G}) := \mathrm{C}$, and $c(\mathrm{T}) := \mathrm{A}$. If the k-mer also contains uncalled bases, marked by the letter "N", we say $c(N) := N$.

We use the `FMIndex` implementation by Gog *et al.* [GBMP14] for counting the number of occurrences of a string in the reads file. Algorithm 7.2 shows how to compute $\mathrm{cov}_{\mathrm{observed}}(k\text{-mer})$ using its definition by exact matches. This approach under-estimates the coverage of a k-mer as some occurrences of the k-mer may be masked by errors.

Algorithm 7.2: Observed Coverage by exact matches

Input: A k-mer
Output: $\mathrm{cov}_{\mathrm{observed}}(k\text{-mer})$ by exact matches

```
1 cov ← FMIndex.countString(k-mer)
2 cov ← cov + FMIndex.countString(reverseComplement(k-mer))
3 return cov
```

7.3. Detecting Errors in a Genome Re-Sequencing Dataset

In order to infer the error profile for a genome re-sequencing dataset (see Chapter 5), we need to extract the "true" errors within the reads.

We use the `BWA-MEM` alignment tool [Li13] for mapping the reads to the reference genome. As in Section 7.2.1, we ignore unmapped reads and reads which map to multiple genomic regions. Then, we use the `samtools` [LHW+09] toolkit to sort the BAM file returned by `BWA-MEM` and convert it into a SAM file. The SAM format is a compressed version of the BAM format. Given a BAM or SAM record (see below) and the reference genome, we provide a C++ function that parses the record and extracts the errors.

7.3.1. BAM/SAM Format

An record in the BAM or SAM format (`http://samtools.github.io/hts-specs/SAMv1.pdf`) consists of:

- The name of the read
- The read sequence
- The quality scores of the read
- The position of the first aligned base in the read
- A boolean flag denoting whether the read is unmapped or not
- A boolean flag denoting whether the read is reverse-complemented or not
- Several other flags (which are irrelevant to us)
- A CIGAR string (see below)

7.3.2. CIGAR String

A CIGAR string consists of multiple entries. Each entry consists of a number followed by a CIGAR operation. A CIGAR operation can be one of:

- M: Alignment match (sequence match or mismatch, may contain substitution errors)
- I: Insertion
- D: Deletion
- S: Soft Clipping (unaligned bases which are present in the read sequence).
- H: Hard Clipping (unaligned bases which have been cropped from read sequence). In `BWA-MEM`, hard clipped bases are used for indicating chimeric reads. Since chimeric reads map to multiple genomic regions by definition, this does not occur in the mappings we keep.
- P: Padding: Silent deletion from the reference sequence. This does not occur in mappings obtained by `BWA-MEM`.

For example, if a read gets aligned the following way;

Position in genome:	1	2	3	4	5	6	7		8	9	10	11	12	13	14	15	16	17
Genome:	C	C	A	T	A	C	T		G	A	A	C	T	G	A	C	T	A
Read:					A	C	T	A	G	A	A		T	G	G	C		

The resulting CIGAR string is 3M1I3M1D4M.

7.4. How to use the Framework

As we consider our current implementation to be still a prototype and only a proof-of-concept, we do not recommend using it in its current state.

In order to use our framework, the user only needs to know the classes `Dataset` and `ComponentSetup`.

The `Dataset` class stores the file paths to the reads file. If the reads originate from a genome-resequencing experiment, the user can also specify the path to the SAM file containing the read alignments as well as the path to the reference genome.

The `ComponentSetup` class acts as an interface between the user and the framework. For example, the user can call the framework as follows:

Algorithm 7.3: Example usage of our framework

```
1 Dataset ds(readsFilePath)
2 ComponentSetup cs(ds, CoverageBiasType::MEDIAN_BIAS_READS_ONLY,
    KmerClassificationType::CLASSIFICATION_NAIVE,
    ErrorProfileType::OVERALL_STATS_ONLY,
    ErrorCorrectionType::KMER_BASED)
3 cs.learnCoverageBias()
4 cs.loadErrorProfile(errorProfilePath)
5 cs.correctReads()
```

As our implementation is still under active development, this may change in the future. We aim to provide a complete user manual as well as a graphical user interface in later releases of our framework.

8. Experimental Results and Analysis

In this chapter, we test our framework modules on both empirical and simulated data from Illumina and PacBio sequencers. We perform all experiments on a Thinkpad T540p laptop with an Intel(R) Core(TM) i7-4710MQ CPU @ 2.50GHz and 16 Gigabyte DDR3 RAM.

8.1. Experimental Setup

8.1.1. Empirical Datasets

We use *Escherichia coli K-12 MG1655* reads from Illumina and PacBio. The bacterium *Escherichia coli* (see Figure 8.1) has a circular genome (`https://www.ncbi.nlm.nih.gov/nuccore/556503834`). For Illumina, we use datasets with the accession numbers SRR396536 and SRR396537 from the NCBI database (`https://www.ncbi.nlm.nih.gov/`). These are two different runs from the same sequencing machine. For PacBio, we use the dataset with the accession number SRR1284073. In order to reduce the computational cost, we perform our experiments on a random sample of 10% of the PacBio reads.

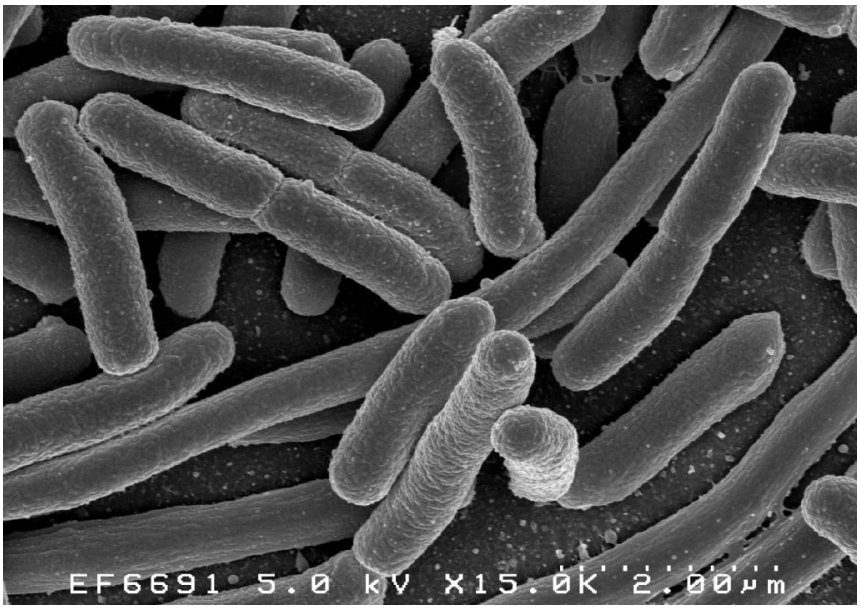


Figure 8.1.: *Escherichia coli*. Image taken from `https://upload.wikimedia.org/wikipedia/commons/f/f8/Coli3.jpg`.

8.1.2. Simulated Datasets

We simulate Illumina reads using the `pIrs` (Profile-based Illumina pair-end reads simulator [HYS+12]) tool. We use the `SimLoRD` (Simulation of Long Read Data [SKR16]) tool to simulate PacBio reads. In empirical datasets, a biological sample that has been sequenced may have diverged from the original reference sample due to mutational changes. In simulated datasets however, we can be sure that the reads originate from a given reference genome.

We call the simulators using the following commands:

```
./pirs simulate -x 70 reference.fasta
simlord -read-reference reference.fasta -n 10000 myreads
```

We simulate reads for the *Escherichia coli K-12 MG1655* genome as well as for the *Ebola* virus sequence (`https://www.ncbi.nlm.nih.gov/nuccore/1050758566?report=fasta`). The *Ebola* virus has a linear genome.

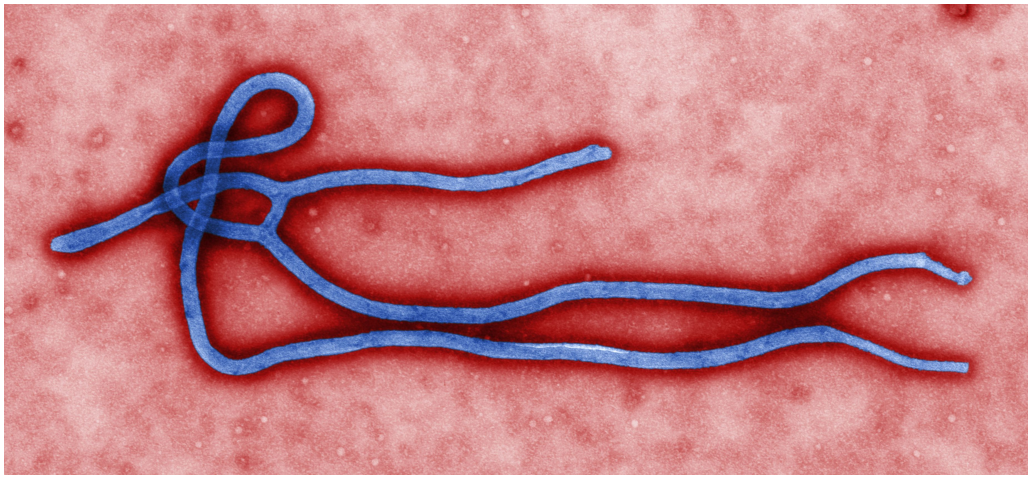


Figure 8.2.: The *Ebola* virus. Image taken from `https://upload.wikimedia.org/wikipedia/commons/e/e6/Ebola_virus_virion.jpg`.

8.2. Coverage Bias Unit

We compare our three variants for computing the median coverage bias factors, as introduced in Chapter 3.

	Variant 1 (reads only)	Variant 2 (alignment)	Variant 3 (reference)
$\text{cov}_{\text{observed}}(k\text{-mer})$	exact matches	alignment	exact matches
use reference genome?	no	yes	yes

For *Escherichia coli*, we use a minimum k-mer size $k_{\min} = 15$. For *Ebola*, we use a minimum k-mer size $k_{\min} = 11$.

Figures 8.3, 8.4, and 8.5 show the resulting median coverage bias factors for the SRR396536, SRR396537 and SRR1284073 dataset, respectively. As the results are similar for the other datasets, we omit them in this Section. They can be found in Appendix A.

Our resulting coverage bias curves fit the shapes discussed in literature [CLY+13]. While the Illumina datasets express low coverage for high G/C contents, the coverage in the

PacBio datasets shows no influence by G/C-content. The simulated datasets express similar coverage biases as the empirical datasets.

Figures 8.3 and 8.4 demonstrate that different runs of the same sequencer express different coverage biases. In the SR396536 dataset, the effect of G/C-content on coverage is higher than in the SR396537 dataset. This motivates the use of run-dependent coverage biases for error correction.

Our experiments show that all our three variants lead to similar results. The median coverage bias factors inferred by variant 2 (aligning the reads to a reference genome) are slightly higher than the median coverage bias factors inferred by our other variants. This is expected behavior as inexact matches of a k-mer are also considered by this variant.

In Figure 8.5, the median bias factors inferred by variant 1 (not using a reference genome) are higher than the median bias factors inferred by variant 3 (using exact matches in a reference genome). This behavior results from discarding coverage bias values lower than $0.2 * \text{cov}_{\text{expected, pusm}}(k\text{-mer})$ in this variant. As all median coverage bias values inferred for the SRR1284073 dataset are lower than 0.5, we conclude that the reads do not cover the whole reference genome (probably due to our artificial down-sampling of the dataset). This causes the Perfect Uniform Sequencing Model to estimate an expected coverage that is too high, leading to correction-factors much smaller than 1.

All in all, our experiments for the Coverage Bias Unit show that we do not need to take into account inexact matches as we already obtain satisfying results by using variant 1.

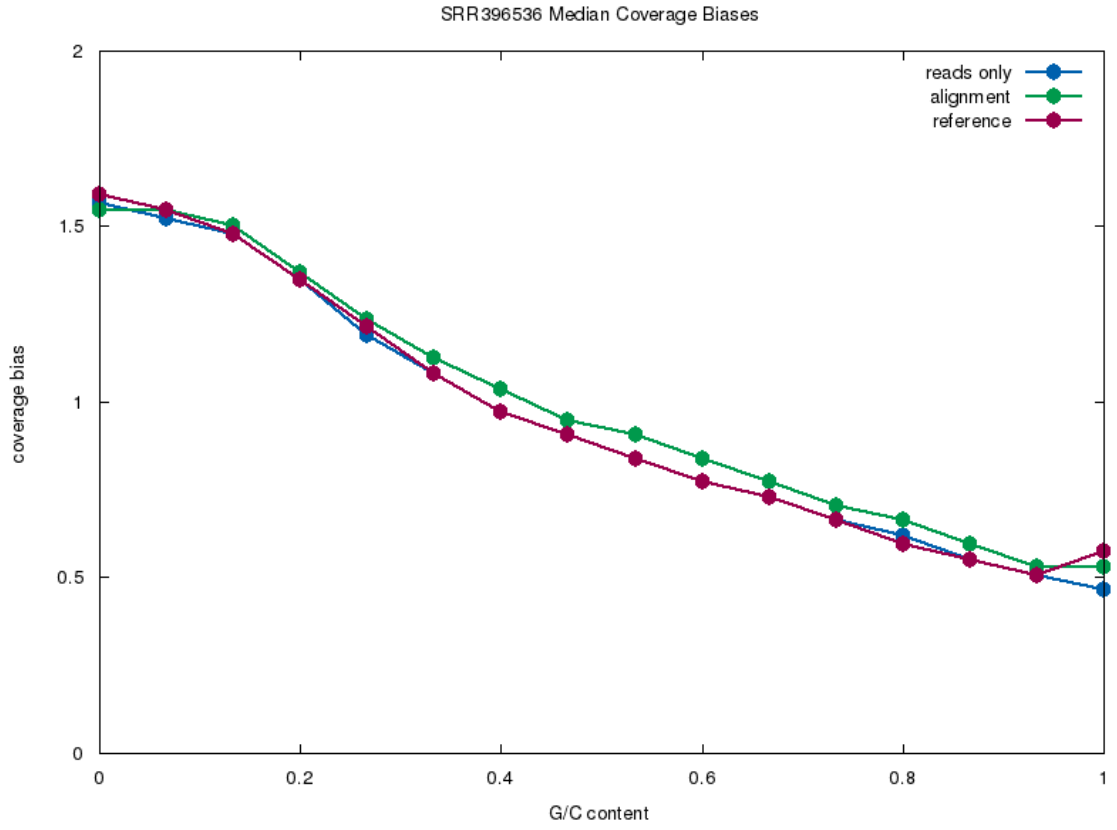


Figure 8.3.: Estimation of Median Coverage Bias for SRR396536, using $k_{\text{min}} = 15$.

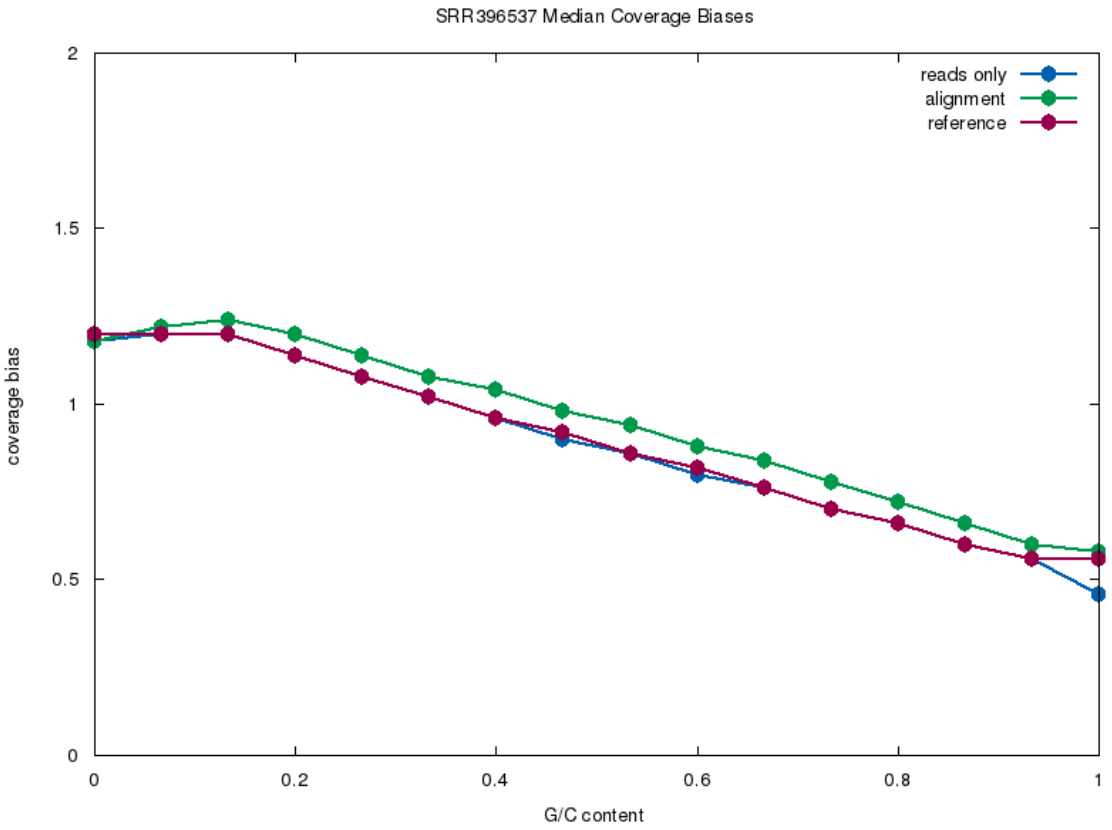


Figure 8.4.: Estimation of Median Coverage Bias for SRR396537, using $k_{\min} = 15$.

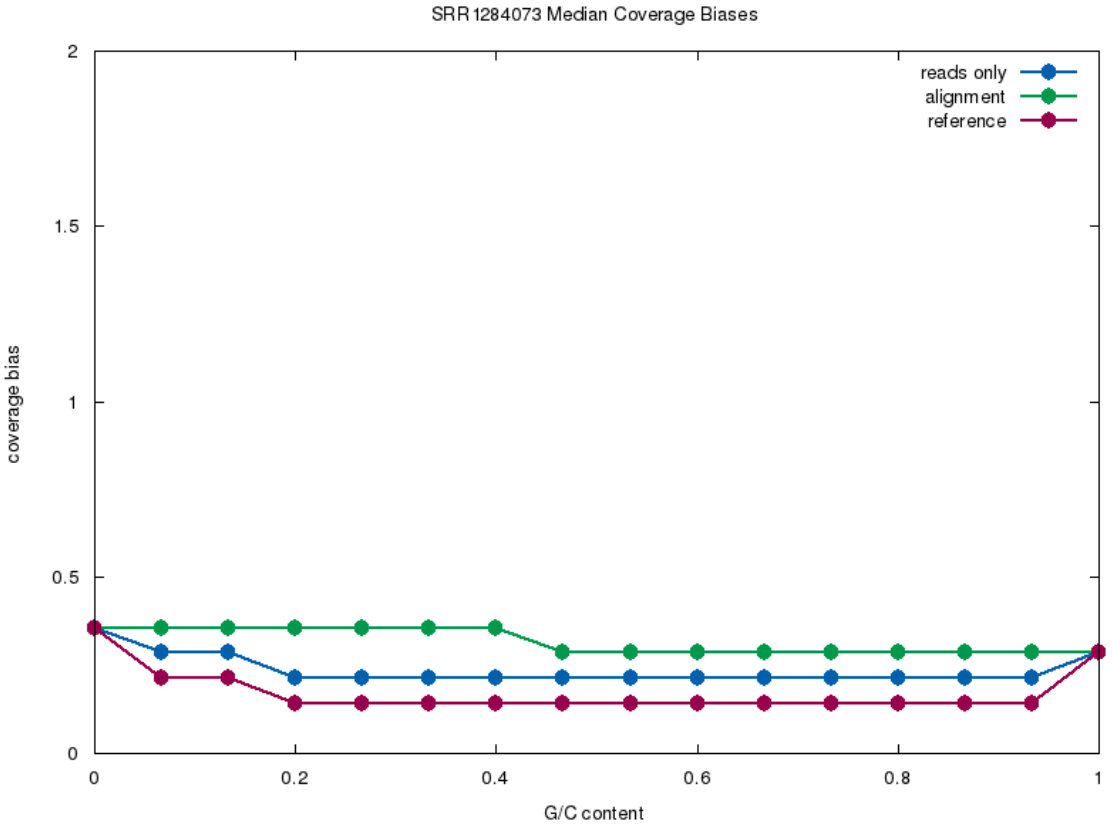


Figure 8.5.: Estimation of Median Coverage Bias for SRR1284073, using $k_{\min} = 15$.

8.3. K-mer Classification Unit

We omit the K-Nearest-Neighbor and the AdaBoost classifier in our experimental results since preliminary experiments have shown that despite higher computational costs for training these classifiers, our overall conclusion does not change. We compare our naïve and statistical approaches, as well as our machine learning approach for classifying a k-mer as either `REPETITIVE`, `TRUSTED`, or `UNTRUSTED` (see Chapter 4). As our experimental results for the datasets express a similar behavior, we only include the results for the SRR396537 dataset in this Section (see Table 8.1). The results for the other datasets can be found in Appendix B. Due to time constraints, we can not compare our k-mer classification methods with other state-of-the-art approaches as they are not available as stand-alone modules.

Our results show that our methods perform especially well in distinguishing `UNTRUSTED` k-mers from the other classes. Our statistical method (using the Z-score of a k-mer) performs worse than our naïve method (comparing $\text{cov}_{\text{bias-corrected}}(k\text{-mer})$ and $\text{cov}_{\text{expected}}(k\text{-mer})$). When using our Python script for automatically choosing a classifier 7.1, the Random Forest classifier always performs best in our experiments. A good performance of Random Forest classifiers on machine learning problems has also be observed by Fernández-Delgado *et al.* [FDCBA14].

As the Random Forest classifier performs better in distinguishing `REPETITIVE` k-mers from `TRUSTED` k-mers than our naïve method, we suspect that taking other factors, such as the size of the k-mer, also into account slightly improves k-mer classification results. However, the increase in average F-Score by doing so is only minimal. Thus, we can also apply the naïve classification method which does not require machine learning at all.

Method	F-Score `REPETITIVE`	F-Score `TRUSTED`	F-Score `UNTRUSTED`	Average F-Score
Naïve	0.89	0.75	0.94	0.86
Statistical	0.82	0.62	0.93	0.79
Naive Bayes	0.86	0.74	0.91	0.84
Decision Tree	0.93	0.78	0.94	0.88
Random Forest	0.93	0.78	0.94	**0.89**
Logistic Regression	0.94	0.70	0.93	0.86

Table 8.1.: F-Scores for SRR396537

8.4. Error Profile Unit

We computed the context-free error profile, the sequence-specific error profile, and the full-context-specific error profile (see Chapter 5) for all datasets. The context-free error profile only computes overall error probabilities based on their frequency of occurrence. The sequence-specific error profile identifies sequence motifs (surrounding a position in a read) that influence the likelihood of an error type to occur. The full-context-specific error profile uses machine learning to estimate error probabilities based on the current base in the read, surrounding motif, quality score, position in the read, and read length.

8.4.1. Context-Free Error Profile

The context-free error profiles follow well-known overall error probabilities for Illumina and PacBio datasets. This is, many substitution errors in the Illumina datasets and many insertion errors in the PacBio datasets. The simulated datasets show similar overall error probabilities as the empirical datasets. Our full experimental results for the context-free error profile can be found in Appendix C.1.

Error Type	Count	log(ℙ(type))
MULTIDEL	292	-13.6911
DEL_OF_A	892	-12.5744
DEL_OF_C	557	-13.0453
DEL_OF_G	512	-13.1296
DEL_OF_T	959	-12.502
INSERTION	2870	-11.4058
A→C	214176	-7.09333
A→G	222468	-7.05534
A→T	64696	-8.29042
A→N	619264	-6.03159
C→A	91428	-7.94457
C→G	116277	-7.70415
C→T	163862	-7.3611
C→N	693737	-5.91803
G→A	91428	-7.30208
G→C	115251	-7.71301
G→T	163862	-7.93652
G→N	693737	-6.0302
T→A	64446	-8.2943
T→C	217570	-7.0776
T→G	213103	-7.09835
T→N	640459	-5.99794

Table 8.2.: Context-Free Error Profile for the Illumina dataset SRR396536

Error Type	Count	log(ℙ(type))
MULTIDEL	61453	-5.86783
DEL_OF_A	58493	-5.91719
DEL_OF_C	90054	-5.48569
DEL_OF_G	88847	-5.49918
DEL_OF_T	61282	-5.87061
INSERTION	1180635	-2.91229
A→C	334768	-4.17266
A→G	320906	-4.21495
A→T	325744	-4.19999
A→N	147622	-4.99144
C→A	361245	-4.09654
C→G	356193	-4.11063
C→T	343356	-4.14733
C→N	151344	-4.96654
G→A	361245	-4.18113
G→C	369493	-4.07397
G→T	343356	-4.13609
G→N	151344	-4.96635
T→A	327012	-4.1961
T→C	324445	-4.20398
T→G	326394	-4.19799
T→N	148789	-4.98357

Table 8.3.: Context-Free Error Profile for the PacBio dataset SRR1284073

8.4.2. Sequence-Specific Error Profile

When listing the motifs with the highest and lowest Z-scores, we omit motifs containing an uncalled 'N' base as they naturally have a very high Z-score since each 'N' in a read is identified as a substitution error. Our full experimental results for the sequence-specific error profile can be found in Appendix C.2.

Our experimental results show that interestingly, in different runs of the same sequencer, nearly identical motifs express a strong influence on error probability (as can be seen in Tables 8.4 and 8.5).

However, since the PacBio dataset SRR1284073 (see Table 8.6) expresses similar error-prone motifs as the Illumina datasets SRR396536 and SRR396537, we conclude that sequence-specific error are linked more to the underlying reference genome than to the sequencing technology which has been used. This conclusion is further supported y the fact that we obtain different relevant motifs for the simulated reads originating from the *Ebola* genome (see Table 8.7).

Z-Score	Motif	Error Type
268.119	GG**G**	SUB_FROM_A
266.19	GG**G**	SUB_FROM_T
165.627	**C**TG	SUB_FROM_G
141.773	C**T**G	SUB_FROM_A
131.915	GG**C**	SUB_FROM_A
129.91	GG**C**	SUB_FROM_T
117.812	**C**AG	SUB_FROM_G
115.038	C**A**G	SUB_FROM_T
102.898	CT**G**	SUB_FROM_C
91.4345	GG**A**	SUB_FROM_A
79.6875	CA**G**	SUB_FROM_C
78.2799	TT**T**	SUB_FROM_G
77.8018	TT**T**	SUB_FROM_C
73.8515	GGG**G**	SUB_FROM_A
72.015	GGG**G**	SUB_FROM_T
68.691	GGCGG**G**	SUB_FROM_A
66.5363	GGCGG**G**	SUB_FROM_T
65.6276	GG**G**	SUB_FROM_G
65.5733	GT**T**	SUB_FROM_A
65.3116	A**C**C	SUB_FROM_G

Table 8.4.: The 20 highest Z-scores for SRR396536

Z-Score	Motif	Error Type
261.364	GG**G**	SUB_FROM_T
259.359	GG**G**	SUB_FROM_A
195.73	**C**TG	SUB_FROM_G
175.425	C**T**G	SUB_FROM_A
152.701	C**A**G	SUB_FROM_T
140.123	**C**AG	SUB_FROM_G
135.101	CT**G**	SUB_FROM_C
122.23	GG**C**	SUB_FROM_A
120.089	GG**C**	SUB_FROM_T
119.964	CA**G**	SUB_FROM_C
88.5459	**A**CC	SUB_FROM_T
84.6021	A**C**C	SUB_FROM_G
81.6422	GG**A**	SUB_FROM_A
80.2634	GT**T**	SUB_FROM_A
74.4338	**G**GT	SUB_FROM_C
74.0543	TT**T**	SUB_FROM_G
73.8033	TT**T**	SUB_FROM_C
72.8299	A**T**C	SUB_FROM_A
72.3074	GT**A**	SUB_FROM_T
69.6961	GGCGG**G**	SUB_FROM_A

Table 8.5.: The 20 highest Z-scores for SRR396537

Z-Score	Motif	Error Type
100.715	**C**TG	SUB_FROM_C
94.9593	TT**T**	INSERTION
90.5478	**C**TG	SUB_FROM_G
85.8336	C**T**G	SUB_FROM_G
85.092	CT**G**	SUB_FROM_G
84.1992	C**T**G	SUB_FROM_C
83.8466	**C**TG	SUB_FROM_A
80.8449	C**T**G	SUB_FROM_A
80.148	**C**TG	SUB_FROM_T
79.0162	**C**AG	SUB_FROM_C
78.5114	C**A**G	SUB_FROM_G
78.3624	CA**G**	SUB_FROM_G
75.8677	C**A**G	SUB_FROM_T
75.1843	C**T**G	SUB_FROM_T
75.0326	C**A**G	SUB_FROM_C
72.1783	**C**AG	SUB_FROM_G
71.0943	C**A**G	SUB_FROM_A
70.5656	CT**G**	SUB_FROM_C
69.5391	**C**AG	SUB_FROM_T
66.3146	CA**G**	SUB_FROM_C

Table 8.6.: The 20 highest Z-scores for SRR1284073

Z-Score	Motif	Error Type
4.88954	**T**GACG	SUB_FROM_A
4.76725	ACTTC**C**	SUB_FROM_A
4.60939	**A**GATGT	SUB_FROM_A
4.34847	**A**TAACG	SUB_FROM_A
4.0	T**C**ATTA	SUB_FROM_T
3.98949	CGTT**C**	SUB_FROM_G
3.97199	T**C**TTG	SUB_FROM_A
3.95368	CG**A**CG	SUB_FROM_G
3.8243	CGTAC**G**	SUB_FROM_G
3.78517	**A**AGACG	SUB_FROM_C
3.74166	CTT**G**AG	SUB_FROM_A
3.71059	T**C**GC	SUB_FROM_T
3.68951	CT**A**C	SUB_FROM_A
3.67945	**G**GATA	SUB_FROM_T
3.61401	CGATA**G**	SUB_FROM_G
3.61401	CGATA**A**	SUB_FROM_A
3.60555	CGT**G**T	SUB_FROM_T
3.60555	C**C**TTTC	SUB_FROM_A
3.60555	AAT**C**AC	SUB_FROM_A
3.56126	**A**CATCG	SUB_FROM_G

Table 8.7.: The 20 highest Z-scores for ebola illumina simulated

8.4.3. Full-Context-Specific Error Profile

We omit the K-Nearest-Neighbor and AdaBoost classifiers in our experimental results since preliminary experiments have shown that despite higher computational costs for training these classifiers, our overall conclusion does not change.

Since precision and F-Score and ill-defined if no data point belongs to a class, we set them to zero in these cases. Our full experimental results for the full-context-specific error profile can be found in Appendix C.3.

As shown in in Tables 8.8 and 8.9, using a Random Forest classifier resulted in higher average F-Scores than using a Naïve Bayes classifier, a Logistic Regression classifier, or a Decision tree in most cases. The low F-Score for insertions in the Illumina datasets was caused by a lack of insertion errors in the dataset. As can be seen in Table 8.10, we obtain a better F-Score for insertions if there is more training data.

Unfortunately, our experimental results for the full-context-specific error profile show that the inference method requires further improvements. It is likely that there is no one-fits-all solution for inferring the full-context-specific error profile of an arbitrary technology.

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score INSERTION	0.02	0.04	0.04	0.01
F-Score SUB_FROM_A	0.74	0.75	0.76	0.73
F-Score SUB_FROM_C	0.77	0.79	0.79	0.61
F-Score SUB_FROM_G	0.52	0.77	0.77	0.43
F-Score SUB_FROM_T	0.64	0.74	0.75	0.64
Average F-Score	0.54	**0.62**	**0.62**	0.48

Table 8.8.: Base-Error Classifiers for the Illumina dataset SRR396537

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score DEL_OF_A	0.65	0.58	0.64	0.66
F-Score DEL_OF_C	0.5	0.46	0.55	0.52
F-Score DEL_OF_G	0.25	0.42	0.48	0.46
F-Score DEL_OF_T	0.59	0.57	0.66	0.62
F-Score MULTIDEL	0.12	0.38	0.47	0.34
Average F-Score	0.42	0.48	**0.56**	0.52

Table 8.9.: Gap-Error Classifiers for the Illumina dataset SRR396537

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score INSERTION	0.37	0.44	0.39	0.35
F-Score SUB_FROM_A	0.11	0.24	0.23	0.18
F-Score SUB_FROM_C	0.3	0.24	0.23	0.25
F-Score SUB_FROM_G	0.11	0.24	0.22	0.17
F-Score SUB_FROM_T	0.12	0.23	0.2	0.13
Average F-Score	0.2	**0.28**	0.25	0.22

Table 8.10.: Base-Error Classifiers for the PacBio dataset SRR1284073

8.5. Error Correction Unit

Unfortunately, calling the machine learning classifiers is too time-consuming for practical use in our current implementation. This is due to the performance loss caused by calling Python code from within C++ code using the Python/C API (`https://docs.python.org/2/c-api/`). Thus, we only perform experiments using our naïve k-mer classification method (see Chapter 4). As the sequence-specific error profile is only meant to be used as a part of the full-context-specific error profile, we only use the context-free error profile (see Chapter 5) in our experiments. To still show the effect of using the full-context-specific error profile on error correction performance, we additionally include experimental results for one small simulated Illumina dataset, using the full-context-specific error profile.

We use the context-free error profile obtained by extracting errors from the underlying genome re-sequencing dataset (see Section 7.3). This might be considered as cheating since this information is not given in a real de novo sequencing dataset. However, since the overall error rates are not expected to differ much between different runs of a sequencer, we expect the effect of doing this to be minimal. Since we already call the error correction with a run-specific error profile, we omit the second step of our error correction method which consists of re-inferring the error profile and re-correcting using the updated error profile. It remains further work to perform more elaborate tests of the Error Correction Unit.

We use the coverage bias values as obtained by the read dataset only (see Chapter 3).

As our results in this section will show that our error correction approach does not compete well with current state-of-the-art sequencing error correction methods, we additionally conduct error correction experiments with perfect k-mer classifications obtained by counting the number of occurrences of a k-mer in the reference genome. This allows us to identify whether the source of our bad error correction performance lies in a bad k-mer classification algorithm or a bad error correction algorithm.

As in the paper by Alic *et al.* [ARDB16], we use the specificity (SP) and sensitivity (SE) metrics for evaluation. The metrics use the following values:

- TP (true positives): errors that have been corrected
- FP (false positives): correct bases/gaps that have been mis-corrected
- TN (true negatives): correct bases/gaps that have remained unaltered
- FN (false negatives): errors that have not been corrected

$$SE := \frac{TP}{TP + FN} \quad \text{and } SP := \frac{TN}{TN + FP}$$

The *sensitivity* measures the percentage of true positives that are correctly identified and the *specificity* measures the percentage of false negatives that are correctly identified.

In addition to these widely-used evaluation metrics, we also compute F-Scores for all error types present in the dataset.

For *Escherichia coli*, we use a minimum k-mer size $k_{\min} = 15$. For *Ebola*, we use a minimum k-mer size $k_{\min} = 11$.

8.5.1. Results and Discussion

We only show experimental results for the simulated Illumina dataset of the *Ebola* viral genome (see Tables 8.11, 8.12, and 8.13). The other experimental results can be found in Appendix D.

Our experimental results express a high specificity (≈ 0.99 in Illumina datasets, ≈ 0.94 in PacBio datasets), but a low sensitivity (≈ 0.2 in Illumina datasets, ≈ 0.02 in PacBio datasets). We conclude that the main disadvantage of our error correction method lies in ignoring too many errors. This is likely due to our restriction of only allowing for a single error to occur within a k-mer. The worse results for PacBio datasets are likely due to an increased error rate in PacBio data and the fact that our current error correction method does not correct deletions of multiple bases.

Our experimental results further show that by using the full-context-specific error profile, our error correction results show just a minor improvement. This is likely due to the way we use the error profile in our error correction approach.

	Total	Substitutions	Insertions	Deletions
True Positives	865	864	1	0
False Positives	3586	3103	340	143
True Negatives	1875850	3770376	942594	3733024
False Negatives	2691	2683	0	8
Sensitivity	0.243251	0.243586	1	0
Specificity	0.998092	0.999178	0.999639	0.999962
F-Score	0.216061	0.229971	0.00584795	0

Table 8.11.: Error correction statistics for ebola illumina simulated, using context-free error profile and naïve k-mer classification

	Total	Substitutions	Insertions	Deletions
True Positives	902	901	1	0
False Positives	3734	2567	637	530
True Negatives	1875062	3768792	942198	3731456
False Negatives	2654	2646	0	8
Sensitivity	0.253656	0.254017	1	0
Specificity	0.998013	0.999319	0.999324	0.999858
F-Score	0.220215	0.256878	0.00312989	0

Table 8.12.: Error correction statistics for ebola illumina simulated, using full-context-specific error profile and naïve k-mer classification

	Total	Substitutions	Insertions	Deletions
True Positives	873	872	1	0
False Positives	2757	2642	91	24
True Negatives	1877277	3773240	943310	3735868
False Negatives	2683	2675	0	8
Sensitivity	0.245501	0.245842	1	0
Specificity	0.998534	0.9993	0.999904	0.999994
F-Score	0.242972	0.246991	0.0215054	0

Table 8.13.: Error correction statistics for ebola illumina simulated, using context-free error profile and perfect k-mer classification

8.5.2. Comparison with other Error Correctors

Alic *et al.* [ARDB16] show that on datasets similar than SRR396536 and SRR396537 from Illumina, the error correction algorithms Coral [SS11], Hybrid SHREC [Sal10], Fiona [SWH+14], and MuffinEC [ATMB16] show sensitivity and specificity values close to 1. Unfortunately, while the specificity of our error correction method is close to 1, too, the sensitivity of our method lies around 0.1 in both datasets. Thus, our error correction approach can not compete with the current state-of-the-art approaches for Illumina datasets.

As Alic *et al.* [ARDB16] also perform experiments on the SRR1284073 dataset from PacBio, we can directly compare the total sensitivity and specificity of our method with the values Alic *et al.* obtain using other methods:

	Sensitivity	Specificity
Our approach	0.0205	0.9415
Our approach (using perfect k-mer classification)	0.0270	0.9513
Hybrid SHREC	0.0004	0.99
Fiona	0.001	0.99
MuffinEC	0.08	0.99

Again, this comparison shows that our error correction method does not compete well with state-of-the-art error correction approaches.

However, it has to be noted that Fiona, HSHREC, and Coral are suffix-tree-based approaches that compute a partial multiple sequence alignment of the reads. As our approach is solely based on k-mers, this might be an unfair comparison. More experiments are needed for future releases of our framework.

9. Conclusion and Future Work

We developed and evaluated a prototype of a modular, open-source, multi-purpose sequencing error correction framework. We devised methods for estimating coverage biases, classifying k-mers, and automatically inferring technology-specific error profiles. For giving an example of how these components can interact with each other, we further implemented an example error correction approach which is based on variable k-mer sizes.

For estimating coverage biases, we devised a statistical model which estimates the expected coverage of a k-mer under an idealized sequencing setting. This model takes the estimated genome size as well as the read length distribution, and computes estimated k-mer counts in the read dataset for both circular and linear genomes. Then, it computes median coverage biases by comparing these expected coverages with the actually observed coverages of the k-mers in the read dataset. Our experimental results have shown that we can infer useful run-dependent coverage biases directly from the dataset, without using additional information and by only taking into account *exact* matches of a k-mer in the dataset.

We used the median coverage bias factors to improve k-mer classification. We devised multiple variants for classifying a k-mer. Each of them takes the bias-corrected observed coverage into account. Each k-mer gets classified as being either `REPETITIVE`, `TRUSTED`, or `UNTRUSTED`. We tried to improve our classification results by using various machine learning classification methods. Our experimental results have shown that by using a Random Forest classifier, we can slightly improve classification results compared to naïve classification. However, our current implementation is too slow due to the performance loss caused by the Python/C API.

As part of our framework, we implemented inference of a context-free error profile which estimates overall error probabilities by counting errors in a genome re-sequencing dataset or a previous correction run. Additionally, our framework infers a sequence-specific error profile which identifies common sequence motifs influencing the probability of specific error types to occur. For taking arbitrary motifs into account, we extended the approach by Shin and Park [SP16]. Moreover, our framework also provides full-context-specific error profiles which take into account not only overall error probabilities, but also quality scores, surrounding motifs and the position of an error in a read. Since the full-context-specific error profile consists of too many (possibly correlated) features, we used machine learning instead of a full mathematical model. Again, our continuous use of the Python/C API for each classification slows down the framework.

To show how our framework modules can interact with each other, we developed a sequencing error correction algorithm which is based on k-mers of varying size. Our error correction algorithm corrects substitutions, insertions, and single base deletions by covering a read with `TRUSTED` or `UNTRUSTED` k-mers. Then, it transforms the `UNTRUSTED` k-mers into `TRUSTED` ones, using the error profile for identifying the most likely error types and allowing one error per k-mer. Unfortunately, our experimental results have shown that our error correction algorithm does not compete well with current state-of-the-art sequencing error correction approaches.

All in all, we did a very first step towards a comprehensive modular open source error correction framework. However, continuous ongoing development is needed before this goal will be fully reached.

Future work

It remains further work to improve the methods we devised in this thesis and to extend the functionality of our framework. We devised an error correction approach which is based on k-mers. Since our experiments have shown that our sequencing error correction approach provides unsatisfying results, we will need to improve our error correction algorithm. For example, similar to the approach of Pal and Aluru [PA14], we could also take counts of highly similar k-mers into account when classifying a k-mer. If a k-mer would be classified as `TRUSTED` but other k-mers within a Hamming distance of 1 would, too, classify all these k-mers as `NINJA` instead. A `NINJA` classification means that an occurrence of the k-mer in a read still has a high chance of being erroneous, despite its count being `TRUSTED`.

It also remains future work to include other current approaches and to devise error correction approaches using multiple sequence alignments. More extensive comparisons of the single modules from our framework with the standard algorithms used in other error correction tools are needed.

In its current form, our k-mer classification approach assumes that the reads originate from a single haploid genome. It remains future work to also support non-haploid genomes and datasets originating from meta-genome studies.

To the best of our knowledge, there is no error correction tool available that fully supports so-called *hybrid read correction* by using read data from multiple runs of distinct sequencing technologies[1]. In order to fully support hybrid correction within our framework, we will need to adapt our k-mer classification unit to work with multiple coverage bias units and combine the read length distributions from different sequencing runs before computing the expected count of a k-mer.

The main factors slowing down our implementation are a non-optimized way of counting k-mers in the read dataset and the extensive use of the Python/C API whenever we want to call a machine learning method. This needs to be changed in future releases. After improving our implementation and supporting non-haploid genomes, scalability of our approach can be tested by correcting reads from NA12878, a human genome sequence that is widely-used for benchmarking.

For increased usability, it remains future work to automatically detect the used sequencing technology from the read names and make parameter suggestions for the user. We could also provide a graphical user interface and show a warning if a run-specific error profile differs a lot from the expected error profile for the used sequencing technology.

According to Eric Rivals [Riv], it might make sense to apply our error profile learning methods to ChIP-Seq and RIBO-Seq data.

[1] There are only tools like Jabba [MHD+16] and LoRDEC [SR14] which aim to correct PacBio SMRT reads with the help of Illumina reads.

Bibliography

[Alt92] Naomi S Altman. An introduction to kernel and nearest-neighbor nonparametric regression. *The American Statistician*, 46(3):175–185, 1992.

[Aly05] Mohamed Aly. Survey on multiclass classification methods. *Neural Netw*, pages 1–9, 2005.

[And81] Stephen Anderson. Shotgun dna sequencing using cloned dnase i-generated fragments. *Nucleic Acids Research*, 9(13):3015, 1981.

[ARDB16] Andy S Alic, David Ruzafa, Joaquin Dopazo, and Ignacio Blanquer. Objective review of de novo stand-alone error correction methods for ngs data. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 2016.

[ATMB16] Andy S Alic, Andres Tomas, Ignacio Medina, and Ignacio Blanquer. Muffinec: Error correction for de novo assembly via greedy partitioning and sequence alignment. *Information Sciences*, 329:206–219, 2016.

[Bio17] Pacific Biosciences. *Pacific BioSciences*, 2017 (accessed March 22, 2017). `http://www.pacb.com/`.

[BPB+97] Frederick R Blattner, Guy Plunkett, Craig A Bloch, Nicole T Perna, Valerie Burland, Monica Riley, Julio Collado-Vides, Jeremy D Glasner, Christopher K Rode, George F Mayhew, et al. The complete genome sequence of escherichia coli k-12. *Science*, 277(5331):1453–1462, 1997.

[Bre01] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[Bro17a] Jason Brownlee. Classification accuracy is not enough: more performance measures you can use. `http://machinelearningmastery.com/classification-accuracy-is-not-enough-more-performance-measures-you-can-use/`, 2014 (accessed March 28, 2017).

[Bro17b] Jason Brownlee. 8 tactics to combat imbalanced classes in your machine learning dataset. `http://machinelearningmastery.com/tactics-to-combat-imbalanced-classes-in-your-machine-learning-dataset/`, 2015 (accessed March 22, 2017).

[BSB+13] Lauren M Bragg, Glenn Stone, Margaret K Butler, Philip Hugenholtz, and Gene W Tyson. Shining a light on dark sequencing: characterising errors in ion torrent pgm data. *PLoS Comput Biol*, 9(4):e1003031, 2013.

[CLY+13] Yen-Chun Chen, Tsunglin Liu, Chun-Hui Yu, Tzen-Yuh Chiang, and Chi-Chuan Hwang. Effects of gc bias in next-generation-sequencing data on de novo genome assembly. *PloS one*, 8(4):e62856, 2013.

[CP08] Mark J Chaisson and Pavel A Pevzner. Short read fragment assembly of bacterial genomes. *Genome research*, 18(2):324–330, 2008.

[DWRR08] Andreas Döring, David Weese, Tobias Rausch, and Knut Reinert. Seqan an efficient, generic c++ library for sequence analysis. *BMC bioinformatics*, 9(1):11, 2008.

[EHWG98] Brent Ewing, LaDeana Hillier, Michael C Wendl, and Phil Green. Base-calling of automated sequencer traces using Phred. i. accuracy assessment. *Genome research*, 8(3):175–185, 1998.

[Eli06] Isaac Elias. Settling the intractability of multiple alignment. *Journal of Computational Biology*, 13(7):1323–1339, 2006.

[FCD+76] Walter Fiers, Roland Contreras, Fred Duerinck, Guy Haegeman, Dirk Iserentant, Jozef Merregaert, Willy Min Jou, Francis Molemans, Alex Raeymaekers, A Van den Berghe, et al. Complete nucleotide sequence of bacteriophage ms2 rna: primary and secondary structure of the replicase gene. *Nature*, 260(5551):500–507, 1976.

[FDCBA14] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems. *J. Mach. Learn. Res*, 15(1):3133–3181, 2014.

[FM00] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[FS95] Yoav Freund and Robert E Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *European conference on computational learning theory*, pages 23–37. Springer, 1995.

[Gar17] Erik Garrison. *A minimal C++ interval tree implementation*, 2017 (accessed March 30, 2017). `https://github.com/ekg/intervaltree`.

[GBMP14] Simon Gog, Timo Beller, Alistair Moffat, and Matthias Petri. From theory to practice: Plug and play with succinct data structures. In *13th International Symposium on Experimental Algorithms, (SEA 2014)*, pages 326–337, 2014.

[GDPB14] Paul Greenfield, Konsta Duesing, Alexie Papanicolaou, and Denis C Bauer. Blue: correcting sequencing errors using consensus and context. *Bioinformatics*, 30(19):2723–2732, 2014.

[GV17] Shane Grant and Randolph Voorhies. *Cereal - A C++11 library for serialization*, 2017 (accessed March 28, 2017). `http://uscilab.github.io/cereal/`.

[HC16] James M Heather and Benjamin Chain. The sequence of sequencers: The history of sequencing dna. *Genomics*, 107(1):1–8, 2016.

[HVB15] Michal Hozza, Tomáš Vinař, and Broňa Brejová. How big is that genome? estimating genome size and coverage from k-mer abundance spectra. In *International Symposium on String Processing and Information Retrieval*, pages 199–209. Springer, 2015.

[HYS+12] Xuesong Hu, Jianying Yuan, Yujian Shi, Jianliang Lu, Binghang Liu, Zhenyu Li, Yanxiang Chen, Desheng Mu, Hao Zhang, Nan Li, et al. pirs: Profile-based illumina pair-end reads simulator. *Bioinformatics*, 28(11):1533–1535, 2012.

[IFI11] Lucian Ilie, Farideh Fazayeli, and Silvana Ilie. Hitec: accurate error correction in high-throughput sequencing data. *Bioinformatics*, 27(3):295–302, 2011.

[Ill17a] Illumina. *Illumina sequencing platforms*, 2017 (accessed March 12, 2017). `https://www.illumina.com/systems/sequencing-platforms.html`.

[Ill17b] Illumina. *Illumina | Sequencing and array-based solutions for genomic research*, 2017 (accessed March 14, 2017). `https://www.illumina.com/`.

[Ill17c] Illumina. *Technology Spotlight: Illumina Sequencing*, 2017 (accessed March 27, 2017). `https://www.illumina.com/documents/products/techspotlights/techspotlight_sequencing.pdf`.

[Ins17] National Human Genome Research Institute. *The Cost of Sequencing a Human Genome*, 2016 (accessed March 12, 2017). `https://www.genome.gov/sequencingcosts`.

[KC91] Michael Krawczak and David N Cooper. Gene deletions causing human genetic disease: mechanisms of mutagenesis and the role of the local dna sequence environment. *Human genetics*, 86(5):425–441, 1991.

[KLZ+17] Paschalia Kapli, Sarah Lutteropp, Jiajie Zhang, Kassian Kobert, Pavlos Pavlidis, Alexandros Stamatakis, and Tomas Flouri. Multi-rate poisson tree processes for single-locus species delimitation under maximum likelihood and markov chain monte carlo. *BMC bioinformatics*, 2017.

[KSS10] David R Kelley, Michael C Schatz, and Steven L Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome biology*, 11(11):R116, 2010.

[LBM16] David Laehnemann, Arndt Borkhardt, and Alice Carolyn McHardy. Denoising dna deep sequencing data—high-throughput sequencing errors and their correction. *Briefings in bioinformatics*, 17(1):154–179, 2016.

[LCS+06] Pedro Larranaga, Borja Calvo, Roberto Santana, Concha Bielza, Josu Galdiano, Iñaki Inza, José A Lozano, Rubén Armañanzas, Guzmán Santafé, Aritz Pérez, et al. Machine learning in bioinformatics. *Briefings in bioinformatics*, 7(1):86–112, 2006.

[LHC+15] Richard M Leggett, Darren Heavens, Mario Caccamo, Matthew D Clark, and Robert P Davey. Nanook: multi-reference alignment analysis of nanopore sequencing data, quality and error profiles. *Bioinformatics*, page btv540, 2015.

[LHO+15] T Laver, J Harrison, PA O'neill, K Moore, A Farbos, K Paszkiewicz, and David J Studholme. Assessing the performance of the oxford nanopore technologies minION. *Biomolecular detection and quantification*, 3:1–8, 2015.

[LHW+09] Heng Li, Bob Handsaker, Alec Wysoker, Tim Fennell, Jue Ruan, Nils Homer, Gabor Marth, Goncalo Abecasis, Richard Durbin, et al. The sequence alignment/map format and SAMtools. *Bioinformatics*, 25(16):2078–2079, 2009.

[Li13] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv preprint arXiv:1303.3997*, 2013.

[LSB+10] Jeffrey T Leek, Robert B Scharpf, Héctor Corrada Bravo, David Simcha, Benjamin Langmead, W Evan Johnson, Donald Geman, Keith Baggerly, and Rafael A Irizarry. Tackling the widespread and critical impact of batch effects in high-throughput data. *Nature Reviews Genetics*, 11(10):733–739, 2010.

[MBD+11] Frazer Meacham, Dario Boffelli, Joseph Dhahbi, David IK Martin, Meromit Singer, and Lior Pachter. Identification and correction of systematic error in high-throughput sequence data. *BMC bioinformatics*, 12(1):451, 2011.

[MHD+16] Giles Miclotte, Mahdi Heydari, Piet Demeester, Stephane Rombauts, Yves Van de Peer, Pieter Audenaert, and Jan Fostier. Jabba: hybrid error correction for long sequencing reads. *Algorithms for Molecular Biology*, 11(1):10, 2016.

[MLT12] Kerensa E McElroy, Fabio Luciani, and Torsten Thomas. Gemsim: general, error-model based simulator of next-generation sequencing data. *BMC genomics*, 13(1):74, 2012.

[MM93] Udi Manber and Gene Myers. Suffix arrays: a new method for on-line string searches. *siam Journal on Computing*, 22(5):935–948, 1993.

[Nan17] Oxford Nanopore. *Oxford Nanopore Technologies*, 2017 (accessed March 22, 2017). `https://nanoporetech.com/`.

[NKA13] Sergey I Nikolenko, Anton I Korobeynikov, and Max A Alekseyev. Bayeshammer: Bayesian clustering for error correction in single-cell sequencing. *BMC genomics*, 14(1):S7, 2013.

[NOM+11] Kensuke Nakamura, Taku Oshima, Takuya Morimoto, Shun Ikeda, Hirofumi Yoshikawa, Yuh Shiwa, Shu Ishikawa, Margaret C Linak, Aki Hirai, Hiroki Takahashi, et al. Sequence-specific error profile of illumina sequencers. *Nucleic acids research*, page gkr344, 2011.

[PA14] Soumitra Pal and Srinivas Aluru. In search of perfect reads. In *Computational Advances in Bio and Medical Sciences (ICCABS), 2014 IEEE 4th International Conference on*, pages 1–2. IEEE, 2014.

[PFL10] Jaume Pellicer, Michael F Fay, and Ilia J Leitch. The largest eukaryotic genome of them all? *Botanical Journal of the Linnean Society*, 164(1):10–15, 2010.

[PVG+11] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. Scikit-learn: Machine learning in python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011.

[Rei17] Olwen Reina. *A Beginner's Guide to Next Generation Sequencing (NGS) Technology*, 2014 (accessed March 13, 2017). `http://bitesizebio.com/21193/a-beginners-guide-to-next-generation-sequencing-ngs-technology`.

[Rij79] C. J. Van Rijsbergen. *Information Retrieval*. Butterworth-Heinemann, Newton, MA, USA, 2nd edition, 1979.

[Riv] Eric Rivals. personal communication.

[Rob64] Herbert Robbins. The empirical bayes approach to statistical decision problems. *The Annals of Mathematical Statistics*, 35(1):1–20, 1964.

[RRC+13] Michael G Ross, Carsten Russ, Maura Costello, Andrew Hollinger, Niall J Lennon, Ryan Hegarty, Chad Nusbaum, and David B Jaffe. Characterizing and measuring bias in sequence data. *Genome biology*, 14(5):R51, 2013.

[Sal10] Leena Salmela. Correction of sequencing errors in a mixed set of reads. *Bioinformatics*, 26(10):1284–1290, 2010.

[SDI+16] Melanie Schirmer, Rosalinda D'Amore, Umer Z Ijaz, Neil Hall, and Christopher Quince. Illumina error profiles: resolving fine-scale variation in metagenomic sequencing data. *BMC bioinformatics*, 17(1):125, 2016.

[SFL14] Li Song, Liliana Florea, and Ben Langmead. Lighter: fast and memory-efficient sequencing error correction without counting. *Genome biology*, 15(11):509, 2014.

[SID+15] Melanie Schirmer, Umer Z Ijaz, Rosalinda D'Amore, Neil Hall, William T Sloan, and Christopher Quince. Insight into biases and sequencing errors for amplicon sequencing with the illumina miseq platform. *Nucleic acids research*, page gku1341, 2015.

[SJ08] Jay Shendure and Hanlee Ji. Next-generation dna sequencing. *Nature biotechnology*, 26(10):1135–1145, 2008.

[SKR16] Bianca Stöcker, Johannes Köster, and Sven Rahmann. Simlord–simulation of long read data. *Bioinformatics*, 32(17):2704–2706, 2016.

[SL91] S Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[SMM+17] Dan Stahlke, Jens Mueller, Robbie Morrison, Daniel Di Marco, and Sylwester Arabas. *C++ interface to gnuplot*, 2015 (accessed March 30, 2017). `https://github.com/dstahlke/gnuplot-iostream`.

[SNC77] Frederick Sanger, Steven Nicklen, and Alan R Coulson. Dna sequencing with chain-terminating inhibitors. *Proceedings of the national academy of sciences*, 74(12):5463–5467, 1977.

[SP16] Sunguk Shin and Joonhong Park. Characterization of sequence-specific errors in various next-generation sequencing systems. *Molecular BioSystems*, 12(3):914–922, 2016.

[SPdT95] Sophie Schbath, Bernard Prum, and Elisabeth de Turckheim. Exceptional motifs in different markov chain models for a statistical analysis of dna sequences. *Journal of Computational Biology*, 2(3):417–437, 1995.

[SR14] Leena Salmela and Eric Rivals. Lordec: accurate and efficient long read error correction. *Bioinformatics*, page btu538, 2014.

[SS11] Leena Salmela and Jan Schröder. Correcting errors in short reads by multiple alignments. *Bioinformatics*, 27(11):1455–1461, 2011.

[SSB13] Julie A Sleep, Andreas W Schreiber, and Ute Baumann. Sequencing error correction without a reference genome. *BMC bioinformatics*, 14(1):367, 2013.

[Sta79] R. Staden. A strategy of dna sequencing employing computer programs. *Nucleic Acids Research*, 6(7):2601, 1979.

[Sta06] Alexandros Stamatakis. Raxml-vi-hpc: maximum likelihood-based phylogenetic analyses with thousands of taxa and mixed models. *Bioinformatics*, 22(21):2688–2690, 2006.

[SWH+14] Marcel H Schulz, David Weese, Manuel Holtgrewe, Viktoria Dimitrova, Sijia Niu, Knut Reinert, and Hugues Richard. Fiona: a parallel and automatic strategy for read error correction. *Bioinformatics*, 30(17):i356–i363, 2014.

[Tor17] Ion Torrent. *Ion Torrent*, 2017 (accessed March 27, 2017). `https://www.thermofisher.com/de/de/home/brands/ion-torrent.html`.

[VAM+01] J Craig Venter, Mark D Adams, Eugene W Myers, Peter W Li, Richard J Mural, Granger G Sutton, Hamilton O Smith, Mark Yandell, Cheryl A Evans, Robert A Holt, et al. The sequence of the human genome. *science*, 291(5507):1304–1351, 2001.

[WJ94] Lusheng Wang and Tao Jiang. On the complexity of multiple sequence alignment. *Journal of computational biology*, 1(4):337–348, 1994.

[WJCD+08] Samuel K Wasser, William Joseph Clark, Ofir Drori, Emily Stephen Kisamo, Celia Mailand, Benezeth Mutayoba, and Matthew Stephens. Combating the illegal trade in african elephant ivory with dna forensics. *Conservation Biology*, 22(4):1065–1071, 2008.

[YCA13] Xiao Yang, Sriram P Chockalingam, and Srinivas Aluru. A survey of error-correction methods for next-generation sequencing. *Briefings in bioinformatics*, 14(1):56–66, 2013.

[ZPB+10] Xiaohong Zhao, Lance E Palmer, Randall Bolanos, Cristian Mircean, Dan Fasulo, and Gayle M Wittenberg. Edar: an efficient error detection and removal algorithm for next generation sequencing data. *Journal of computational biology*, 17(11):1549–1560, 2010.

Appendix

A. Coverage Bias Unit Experiments

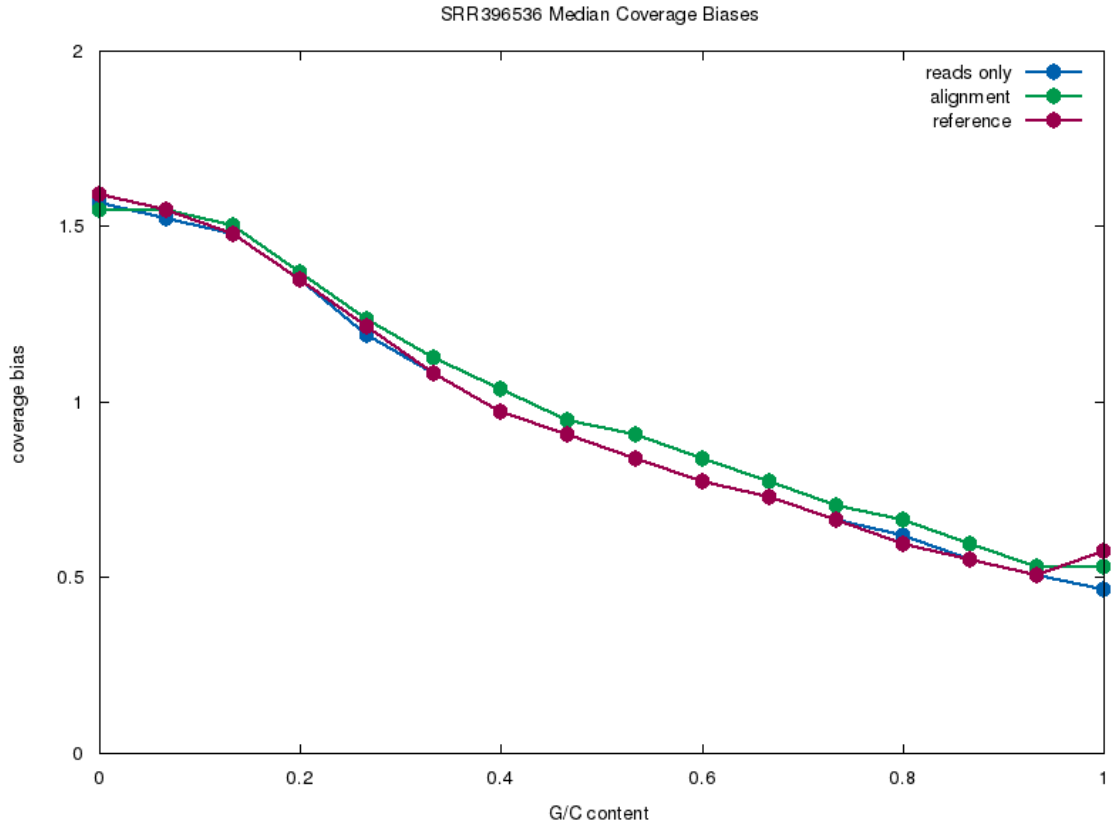


Figure A.1.: Estimation of Median Coverage Bias for SRR396536, using $k_{\min} = 15$.

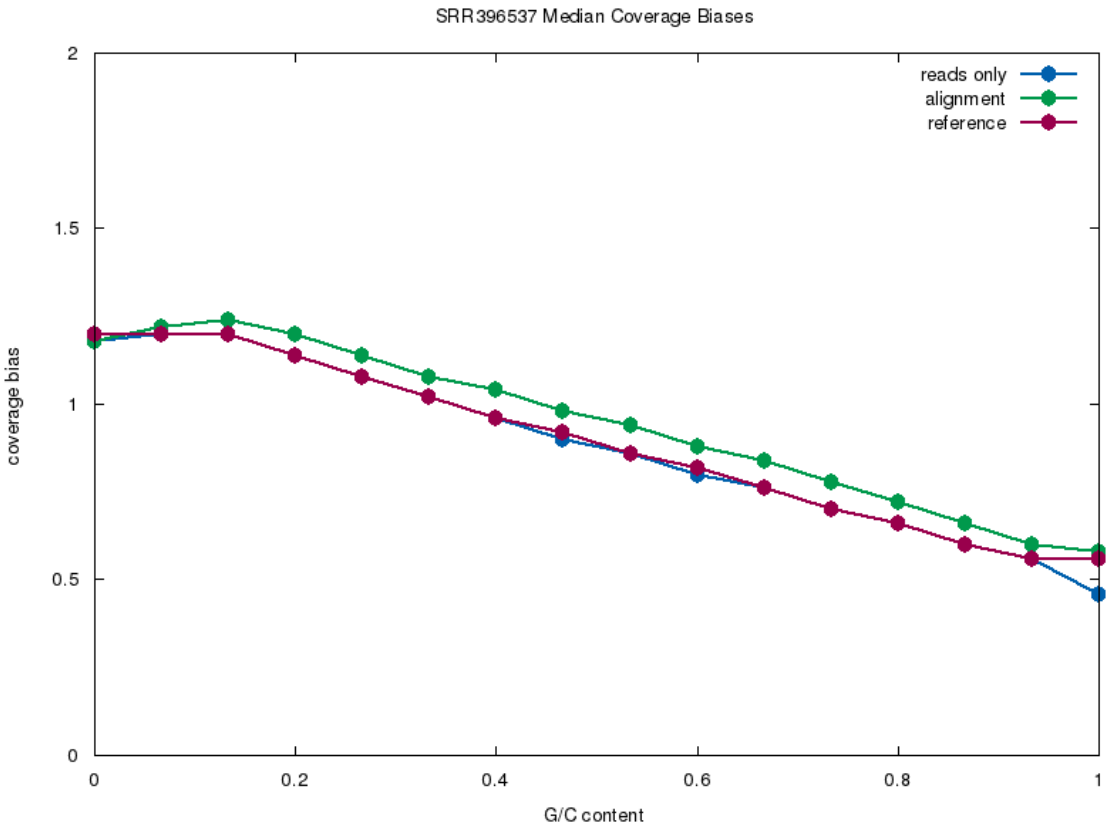


Figure A.2.: Estimation of Median Coverage Bias for SRR396537, using $k_{\min} = 15$.

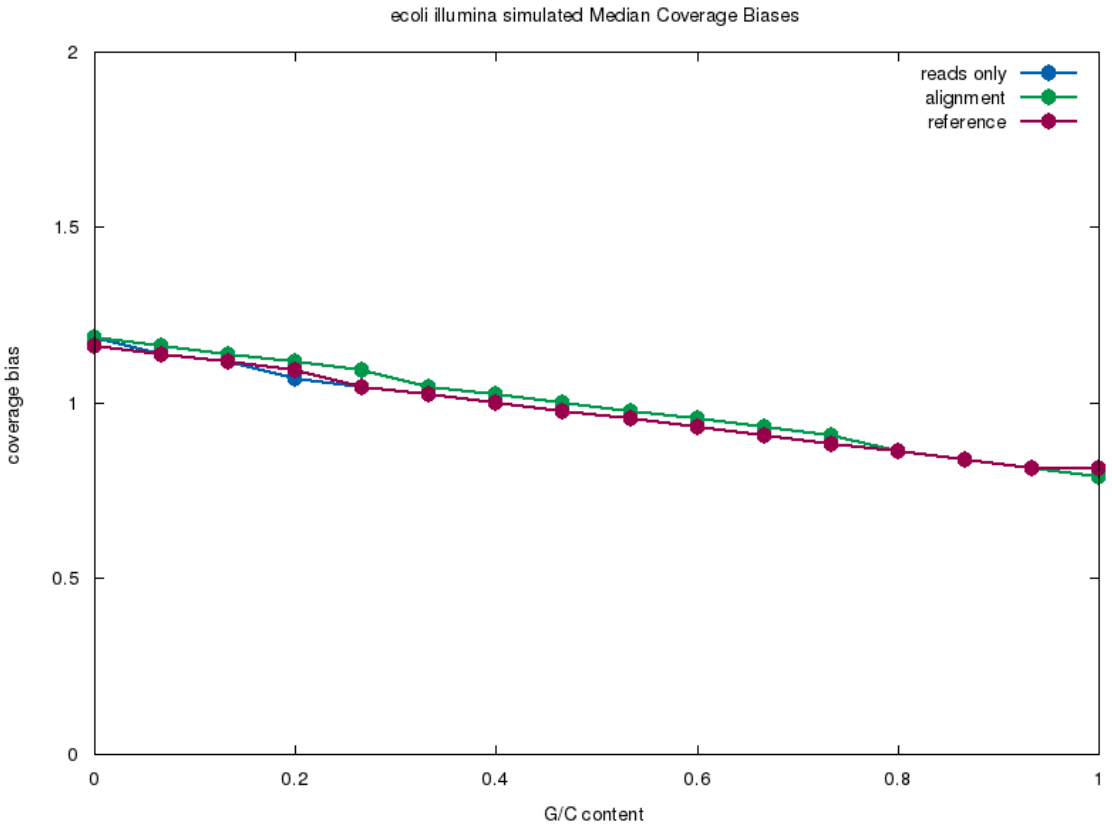


Figure A.3.: Estimation of Median Coverage Bias for the simulated *E.coli* Illumina dataset, using $k_{\min} = 15$.

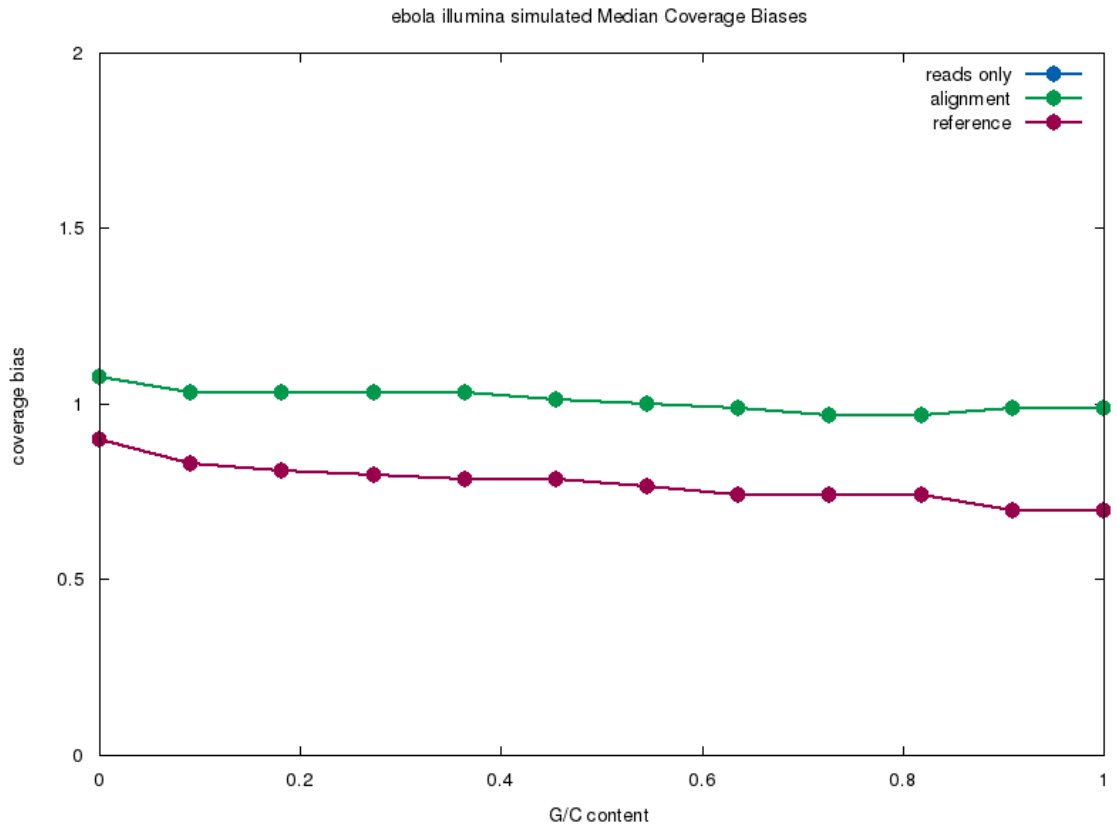


Figure A.4.: Estimation of Median Coverage Bias for the simulated *Ebola* Illumina dataset, using $k_{\min} = 11$.

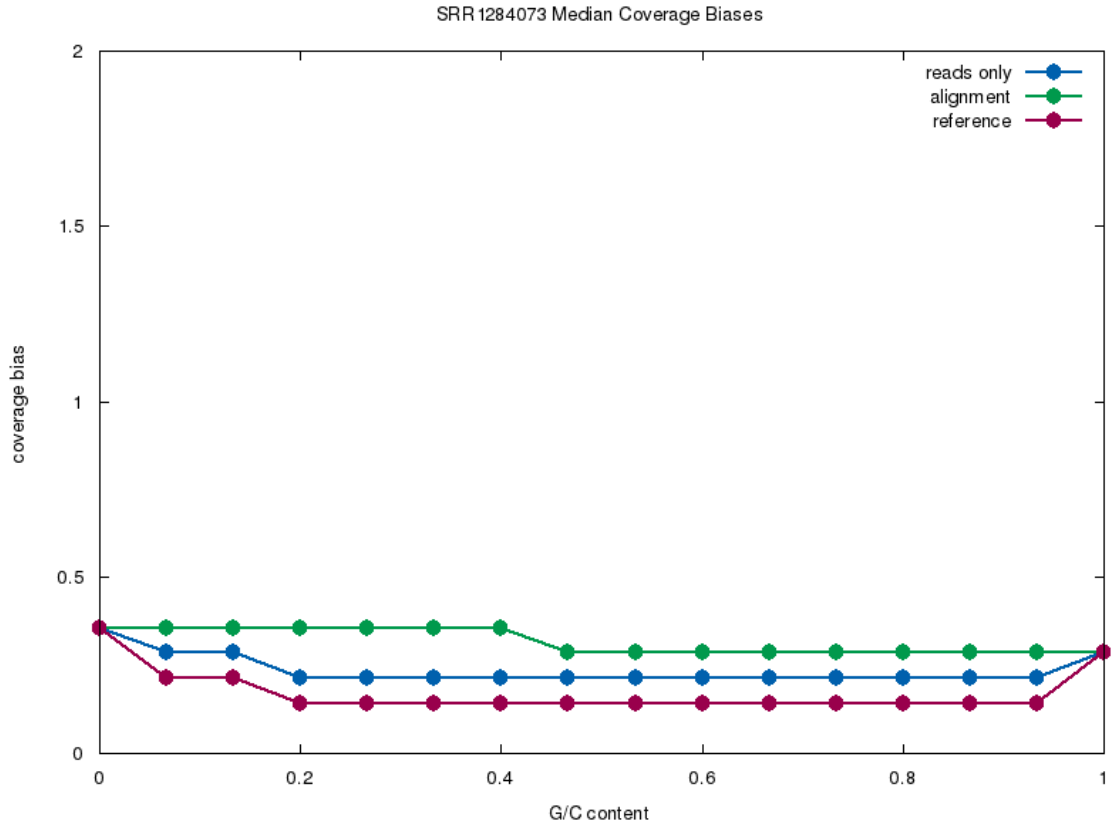


Figure A.5.: Estimation of Median Coverage Bias for SRR1284073, using $k_{\min} = 15$.

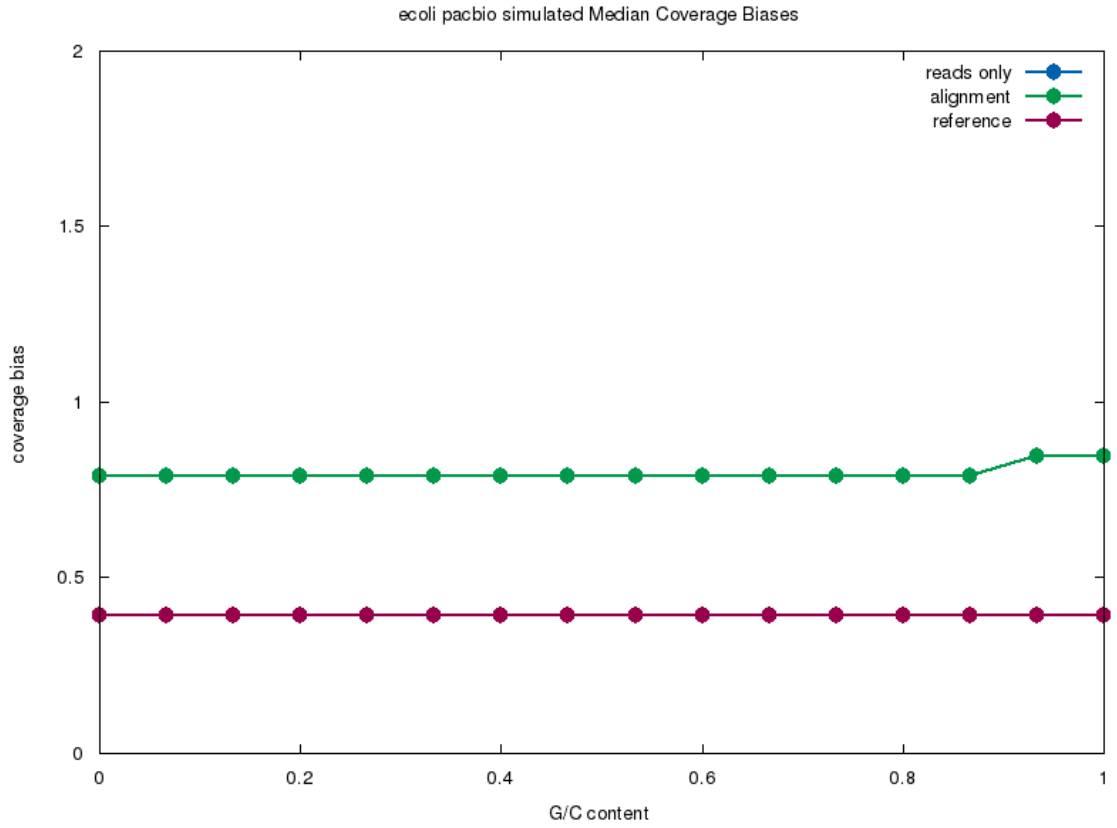


Figure A.6.: Estimation of Median Coverage Bias for the simulated *E.coli* PacBio dataset, using $k_{\min} = 15$.

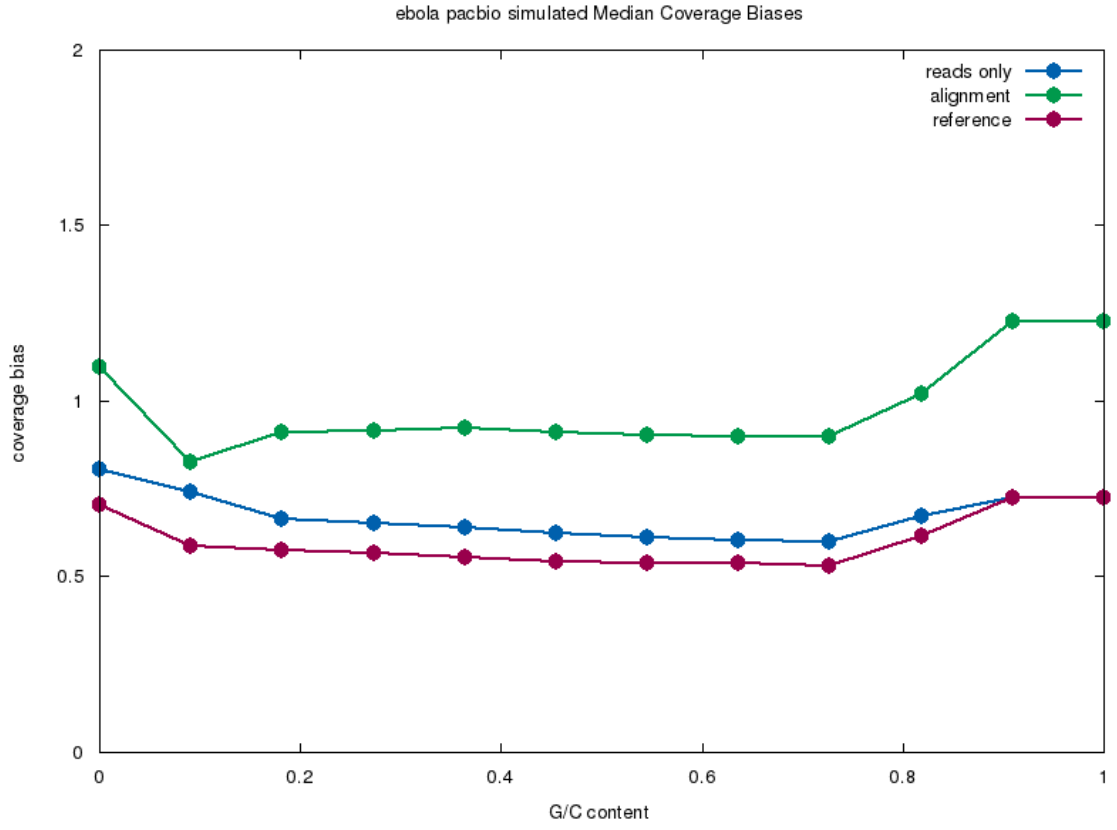


Figure A.7.: Estimation of Median Coverage Bias for the simulated *Ebola* PacBio dataset, using $k_{\min} = 11$.

B. K-mer Classification Unit Experiments

Method	F-Score REPETITIVE	F-Score TRUSTED	F-Score UNTRUSTED	Average F-Score
Naive	0.86	0.71	0.92	0.83
Statistical	0.80	0.59	0.92	0.77
Naive Bayes	0.89	0.57	0.90	0.79
Decision Tree	0.93	0.77	0.93	0.87
Random Forest	0.93	0.77	0.93	**0.88**
Logistic Regression	0.94	0.64	0.91	0.83

Table B.1.: F-Scores for SRR396536

Method	F-Score REPETITIVE	F-Score TRUSTED	F-Score UNTRUSTED	Average F-Score
Naïve	0.89	0.75	0.94	0.86
Statistical	0.82	0.62	0.93	0.79
Naïve Bayes	0.86	0.74	0.91	0.84
Decision Tree	0.93	0.78	0.94	0.88
Random Forest	0.93	0.78	0.94	**0.89**
Logistic Regression	0.94	0.70	0.93	0.86

Table B.2.: F-Scores for SRR396537

Method	F-Score REPETITIVE	F-Score TRUSTED	F-Score UNTRUSTED	Average F-Score
Naïve	0.94	0.75	0.88	**0.86**
Statistical	0.89	0.72	0.88	0.83
Naïve Bayes	0.93	0.75	0.88	**0.86**
Decision Tree	0.93	0.75	0.88	**0.86**
Random Forest	0.94	0.76	0.88	**0.86**
Logistic Regression	0.94	0.65	0.87	0.82

Table B.3.: F-Scores for ecoli_illumina_simulated

Method	F-Score REPETITIVE	F-Score TRUSTED	F-Score UNTRUSTED	Average F-Score
Naïve	0.82	0.77	0.83	0.81
Statistical	0.74	0.74	0.83	0.77
Naïve Bayes	0.75	0.75	0.86	0.79
Decision Tree	0.80	0.78	0.83	0.80
Random Forest	0.81	0.78	0.83	**0.81**
Logistic Regression	0.80	0.76	0.85	0.80

Table B.4.: F-Scores for ebola_illumina_simulated

Method	F-Score REPETITIVE	F-Score TRUSTED	F-Score UNTRUSTED	Average F-Score
Naïve	0.87	0.42	0.92	0.74
Statistical	0.87	0.42	0.91	0.73
Naïve Bayes	0.85	0.25	0.92	0.67
Decision Tree	0.92	0.65	0.93	**0.84**
Random Forest	0.92	0.65	0.93	**0.84**
Logistic Regression	0.92	0.44	0.94	0.77

Table B.5.: F-Scores for SRR1284073

Method	F-Score REPETITIVE	F-Score TRUSTED	F-Score UNTRUSTED	Average F-Score
Naïve	0.86	0.68	0.97	0.84
Statistical	0.86	0.68	0.97	0.84
Naïve Bayes	0.87	0.36	0.93	0.72
Decision Tree	0.94	0.82	0.97	**0.91**
Random Forest	0.94	0.82	0.97	**0.91**
Logistic Regression	0.94	0.75	0.96	0.89

Table B.6.: F-Scores for ecoli_pacbio_simulated

Method	F-Score REPETITIVE	F-Score TRUSTED	F-Score UNTRUSTED	Average F-Score
Naïve	0.72	0.71	0.99	0.81
Statistical	0.52	0.05	0.98	0.52
Naïve Bayes	0.71	0.87	1.00	0.86
Decision Tree	0.73	0.85	0.99	0.86
Random Forest	0.77	0.87	1.00	**0.88**
Logistic Regression	0.70	0.75	0.99	0.81

Table B.7.: F-Scores for ebola_pacbio_simulated

C. Error Profile Unit Experiments

We omit the K-Nearest-Neighbor and the AdaBoost classifier in our experimental results since preliminary experiments have shown that despite higher computational costs for training these classifiers, our overall conclusion does not change.

When printing the motifs with the highest and lowest Z-scores, we omit motifs containing an uncalled 'N' base as they naturally have a very high Z-score since each 'N' in a read is identified as a substitution error.

Since precision and F-Score and ill-defined if no data point belongs to a class, we set it to zero in these cases.

C.1. Context-Free Error Profile

Error Type	Count	log(ℙ(type))
MULTIDEL	292	-13.6911
DEL_OF_A	892	-12.5744
DEL_OF_C	557	-13.0453
DEL_OF_G	512	-13.1296
DEL_OF_T	959	-12.502
INSERTION	2870	-11.4058
A→C	214176	-7.09333
A→G	222468	-7.05534
A→T	64696	-8.29042
A→N	619264	-6.03159
C→A	91428	-7.94457
C→G	116277	-7.70415
C→T	163862	-7.3611
C→N	693737	-5.91803
G→A	91428	-7.30208
G→C	115251	-7.71301
G→T	163862	-7.93652
G→N	693737	-6.0302
T→A	64446	-8.2943
T→C	217570	-7.0776
T→G	213103	-7.09835
T→N	640459	-5.99794

Table C.8.: SRR396536

Error Type	Count	log(ℙ(type))
MULTIDEL	300	-13.7584
DEL_OF_A	826	-12.7455
DEL_OF_C	537	-13.1761
DEL_OF_G	591	-13.0803
DEL_OF_T	938	-12.6184
INSERTION	2822	-11.5169
A→C	219755	-7.16187
A→G	219777	-7.16177
A→T	68584	-8.32633
A→N	1046689	-5.601
C→A	107870	-7.87346
C→G	125573	-7.7215
C→T	124583	-7.72941
C→N	1162449	-5.4961
G→A	107870	-7.66233
G→C	128908	-7.69529
G→T	124583	-7.85686
G→N	1162449	-5.56697
T→A	69202	-8.31736
T→C	219109	-7.16482
T→G	217648	-7.17151
T→N	1081249	-5.56851

Table C.9.: SRR396537

Error Type	Count	log(ℙ(type))
MULTIDEL	97	-14.6879
DEL_OF_A	600	-12.8657
DEL_OF_C	628	-12.8201
DEL_OF_G	609	-12.8508
DEL_OF_T	673	-12.7509
INSERTION	1201	-12.1717
A→C	100619	-7.74351
A→G	68188	-8.13258
A→T	53386	-8.3773
A→N	13707	-9.73694
C→A	68278	-8.13126
C→G	37443	-8.73203
C→T	45486	-8.53744
C→N	13325	-9.76521
G→A	68278	-8.45895
G→C	43562	-8.58066
G→T	45486	-7.48166
G→N	13325	-9.74404
T→A	39752	-8.67219
T→C	61850	-8.23014
T→G	122000	-7.55083
T→N	13508	-9.75157

Table C.10.: ecoli illumina simulated

Error Type	Count	log(ℙ(type))
MULTIDEL	0	-inf
DEL_OF_A	3	-12.6634
DEL_OF_C	1	-13.762
DEL_OF_G	2	-13.0689
DEL_OF_T	2	-13.0689
INSERTION	1	-13.762
A→C	530	-7.48913
A→G	368	-7.85392
A→T	270	-8.16358
A→N	78	-9.4053
C→A	246	-8.25667
C→G	139	-8.82753
C→T	164	-8.66214
C→N	67	-9.55731
G→A	246	-8.63804
G→C	132	-8.8792
G→T	164	-7.80876
G→N	67	-10.3947
T→A	147	-8.77157
T→C	255	-8.22074
T→G	507	-7.53349
T→N	62	-9.63487

Table C.11.: ebola illumina simulated

Error Type	Count	log(ℙ(type))
MULTIDEL	61453	-5.86783
DEL_OF_A	58493	-5.91719
DEL_OF_C	90054	-5.48569
DEL_OF_G	88847	-5.49918
DEL_OF_T	61282	-5.87061
INSERTION	1180635	-2.91229
A→C	334768	-4.17266
A→G	320906	-4.21495
A→T	325744	-4.19999
A→N	147622	-4.99144
C→A	361245	-4.09654
C→G	356193	-4.11063
C→T	343356	-4.14733
C→N	151344	-4.96654
G→A	361245	-4.18113
G→C	369493	-4.07397
G→T	343356	-4.13609
G→N	151344	-4.96635
T→A	327012	-4.1961
T→C	324445	-4.20398
T→G	326394	-4.19799
T→N	148789	-4.98357

Table C.12.: SRR1284073

Error Type	Count	log(ℙ(type))
MULTIDEL	37008	-7.52088
DEL_OF_A	121971	-6.32823
DEL_OF_C	129546	-6.26798
DEL_OF_G	128971	-6.27243
DEL_OF_T	121677	-6.33065
INSERTION	3009034	-3.12264
A→C	665973	-4.63077
A→G	658330	-4.64231
A→T	631828	-4.6834
A→N	275909	-5.51195
C→A	662875	-4.63543
C→G	705176	-4.57357
C→T	657736	-4.64321
C→N	284423	-5.48155
G→A	662875	-4.66107
G→C	717390	-4.5564
G→T	657736	-4.63353
G→N	284423	-5.48447
T→A	649437	-4.65591
T→C	640881	-4.66917
T→G	659847	-4.64001
T→N	271712	-5.52727

Table C.13.: ecoli pacbio simulated

Error Type	Count	log(ℙ(type))
MULTIDEL	37748	-7.50761
DEL_OF_A	143650	-6.17116
DEL_OF_C	104903	-6.48551
DEL_OF_G	104313	-6.49115
DEL_OF_T	143575	-6.17169
INSERTION	3012199	-3.12812
A→C	737730	-4.53497
A→G	661628	-4.64384
A→T	861822	-4.37949
A→N	378494	-5.20234
C→A	707634	-4.57662
C→G	448182	-5.03334
C→T	590170	-4.75813
C→N	245966	-5.63335
G→A	707634	-4.67606
G→C	436294	-5.06023
G→T	590170	-4.84877
G→N	245966	-5.70525
T→A	861528	-4.37984
T→C	592295	-4.75454
T→G	545051	-4.83766
T→N	333185	-5.32985

Table C.14.: ebola pacbio simulated

C.2. Sequence-Specific Error Profile

Z-Score	Motif	Error Type
268.119	GG**G**	SUB_FROM_A
266.19	GG**G**	SUB_FROM_T
165.627	**C**TG	SUB_FROM_G
141.773	C**T**G	SUB_FROM_A
131.915	GG**C**	SUB_FROM_A
129.91	GG**C**	SUB_FROM_T
117.812	**C**AG	SUB_FROM_G
115.038	C**A**G	SUB_FROM_T
102.898	CT**G**	SUB_FROM_C
91.4345	GG**A**	SUB_FROM_A
79.6875	CA**G**	SUB_FROM_C
78.2799	TT**T**	SUB_FROM_G
77.8018	TT**T**	SUB_FROM_C
73.8515	GGG**G**	SUB_FROM_A
72.015	GGG**G**	SUB_FROM_T
68.691	GGCGG**G**	SUB_FROM_A
66.5363	GGCGG**G**	SUB_FROM_T
65.6276	GG**G**	SUB_FROM_G
65.5733	GT**T**	SUB_FROM_A
65.3116	A**C**C	SUB_FROM_G

Table C.15.: The 20 highest Z-scores for SRR396536

Z-Score	Motif	Error Type
-124.703	TG**G**	SUB_FROM_A
-124.051	TG**G**	SUB_FROM_T
-87.549	TG**C**	SUB_FROM_A
-86.443	TG**C**	SUB_FROM_T
-85.4619	TT**G**	SUB_FROM_C
-84.1717	C**T**A	SUB_FROM_A
-78.9295	T**A**G	SUB_FROM_T
-73.4905	CT**A**	SUB_FROM_T
-67.829	AA**T**	SUB_FROM_A
-65.7881	TA**G**	SUB_FROM_C
-63.8715	**C**AA	SUB_FROM_G
-62.7838	AG**G**	SUB_FROM_T
-62.3403	AG**G**	SUB_FROM_A
-60.1977	**C**TA	SUB_FROM_G
-60.1559	CG**A**	SUB_FROM_T
-59.6404	**T**AG	SUB_FROM_A
-59.4657	TG**A**	SUB_FROM_A
-58.2913	TG**T**	SUB_FROM_A
-56.9548	CG**G**	SUB_FROM_A
-55.4038	CG**G**	SUB_FROM_T

Table C.16.: The 20 lowest Z-scores for SRR396536

Z-Score	Motif	Error Type
261.364	GG**G**	SUB_FROM_T
259.359	GG**G**	SUB_FROM_A
195.73	**C**TG	SUB_FROM_G
175.425	C**T**G	SUB_FROM_A
152.701	C**A**G	SUB_FROM_T
140.123	**C**AG	SUB_FROM_G
135.101	CT**G**	SUB_FROM_C
122.23	GG**C**	SUB_FROM_A
120.089	GG**C**	SUB_FROM_T
119.964	CA**G**	SUB_FROM_C
88.5459	**A**CC	SUB_FROM_T
84.6021	A**C**C	SUB_FROM_G
81.6422	GG**A**	SUB_FROM_A
80.2634	GT**T**	SUB_FROM_A
74.4338	**G**GT	SUB_FROM_C
74.0543	TT**T**	SUB_FROM_G
73.8033	TT**T**	SUB_FROM_C
72.8299	A**T**C	SUB_FROM_A
72.3074	GT**A**	SUB_FROM_T
69.6961	GGCGG**G**	SUB_FROM_A

Table C.17.: The 20 highest Z-scores for SRR396537

Z-Score	Motif	Error Type
-118.654	TG**G**	SUB_FROM_T
-117.598	TG**G**	SUB_FROM_A
-104.103	TT**G**	SUB_FROM_C
-103.058	C**T**A	SUB_FROM_A
-99.7532	T**A**G	SUB_FROM_T
-90.6144	CT**A**	SUB_FROM_T
-85.6052	**T**AG	SUB_FROM_A
-83.6985	TA**G**	SUB_FROM_C
-80.1133	TG**C**	SUB_FROM_A
-79.2389	TG**C**	SUB_FROM_T
-76.3415	AA**T**	SUB_FROM_A
-75.6735	**C**AA	SUB_FROM_G
-74.9665	**C**TA	SUB_FROM_G
-72.9099	TG**T**	SUB_FROM_A
-69.0232	**C**TC	SUB_FROM_G
-68.2281	T**T**G	SUB_FROM_A
-67.9784	G**G**CC	SUB_FROM_C
-67.6827	C**T**C	SUB_FROM_A
-66.2892	GG**C**C	SUB_FROM_G
-65.8541	TC**G**	SUB_FROM_C

Table C.18.: The 20 lowest Z-scores for SRR396537

Z-Score	Motif	Error Type
48.7967	C**C**G	SUB_FROM_T
44.0529	**A**TG	SUB_FROM_G
42.7733	**A**AG	SUB_FROM_G
41.7931	C**G**G	SUB_FROM_A
40.6459	C**C**G	SUB_FROM_A
37.703	C**G**G	SUB_FROM_T
31.8024	C**A**G	SUB_FROM_A
25.3793	**G**TG	SUB_FROM_G
25.2881	C**A**G	SUB_FROM_T
23.4332	C**T**G	SUB_FROM_T
22.5813	**G**AG	SUB_FROM_G
21.5309	**A**TG	SUB_FROM_C
21.2494	**T**AG	SUB_FROM_G
21.2272	AC**A**	SUB_FROM_G
20.6903	**T**TG	SUB_FROM_G
20.2057	A**A**C	SUB_FROM_G
20.0898	C**T**G	SUB_FROM_A
19.2762	**A**AG	SUB_FROM_C
18.8335	CA**C**	SUB_FROM_C
16.7921	CT**C**	SUB_FROM_C

Table C.19.: The 20 highest Z-scores for ecoli illumina simulated

Z-Score	Motif	Error Type
-24.1054	T**C**G	SUB_FROM_T
-23.4401	**A**TA	SUB_FROM_G
-23.1838	GG**A**C	SUB_FROM_G
-21.1792	C**G**A	SUB_FROM_A
-20.6087	**A**AA	SUB_FROM_G
-20.0608	CT**C**	SUB_FROM_T
-20.0137	C**C**A	SUB_FROM_T
-19.5167	T**G**G	SUB_FROM_A
-19.0324	C**A**A	SUB_FROM_A
-18.9817	C**C**A	SUB_FROM_A
-18.9714	CA**C**	SUB_FROM_T
-18.4096	T**C**G	SUB_FROM_A
-18.043	T**G**G	SUB_FROM_T
-17.4642	**A**CC	SUB_FROM_G
-17.4345	C**A**AA	SUB_FROM_G
-17.1975	GGC**A**	SUB_FROM_G
-16.8798	**A**TC	SUB_FROM_G
-16.4031	C**A**C	SUB_FROM_G
-16.1393	G**C**G	SUB_FROM_T
-15.9863	CA**C**	SUB_FROM_A

Table C.20.: The 20 lowest Z-scores for ecoli illumina simulated

Z-Score	Motif	Error Type
4.88954	**T**GACG	SUB_FROM_A
4.76725	ACTTC**C**	SUB_FROM_A
4.60939	**A**GATGT	SUB_FROM_A
4.34847	**A**TAACG	SUB_FROM_A
4.0	T**C**ATTA	SUB_FROM_T
3.98949	CGTT**C**	SUB_FROM_G
3.97199	T**C**TTG	SUB_FROM_A
3.95368	CG**A**CG	SUB_FROM_G
3.8243	CGTAC**G**	SUB_FROM_G
3.78517	**A**AGACG	SUB_FROM_C
3.74166	CTT**G**AG	SUB_FROM_A
3.71059	T**C**GC	SUB_FROM_T
3.68951	CT**A**C	SUB_FROM_A
3.67945	**G**GATA	SUB_FROM_T
3.61401	CGATA**G**	SUB_FROM_G
3.61401	CGATA**A**	SUB_FROM_A
3.60555	CGT**G**T	SUB_FROM_T
3.60555	C**C**TTTC	SUB_FROM_A
3.60555	AAT**C**AC	SUB_FROM_A
3.56126	**A**CATCG	SUB_FROM_G

Table C.21.: The 20 highest Z-scores for ebola illumina simulated

Z-Score	Motif	Error Type
-3.60083	AC**G**C	SUB_FROM_A
-3.03974	TG**G**AA	SUB_FROM_T
-2.96702	**G**GA	SUB_FROM_C
-2.95014	A**C**GTT	SUB_FROM_T
-2.89035	TC**G**G	SUB_FROM_A
-2.88194	T**T**GG	SUB_FROM_C
-2.82843	C**G**CTGA	SUB_FROM_A
-2.82843	GGT**G**AA	SUB_FROM_A
-2.82843	T**A**TTGG	SUB_FROM_G
-2.72352	ATC**C**A	SUB_FROM_T
-2.72253	AG**C**TG	SUB_FROM_A
-2.70017	C**C**G	SUB_FROM_T
-2.6968	TG**C**TAA	SUB_FROM_A
-2.68742	AA**A**TT	SUB_FROM_C
-2.64575	GGG**C**TT	SUB_FROM_A
-2.64575	TCC**G**GC	SUB_FROM_A
-2.64575	TGTA**G**C	SUB_FROM_A
-2.64575	ATAC**A**T	SUB_FROM_G
-2.64575	TG**A**CTA	SUB_FROM_G
-2.64575	G**T**GGC	SUB_FROM_T

Table C.22.: The 20 lowest Z-scores for ebola illumina simulated

Z-Score	Motif	Error Type
100.715	**C**TG	SUB_FROM_C
94.9593	TT**T**	INSERTION
90.5478	**C**TG	SUB_FROM_G
85.8336	C**T**G	SUB_FROM_G
85.092	CT**G**	SUB_FROM_G
84.1992	C**T**G	SUB_FROM_C
83.8466	**C**TG	SUB_FROM_A
80.8449	C**T**G	SUB_FROM_A
80.148	**C**TG	SUB_FROM_T
79.0162	**C**AG	SUB_FROM_C
78.5114	C**A**G	SUB_FROM_G
78.3624	CA**G**	SUB_FROM_G
75.8677	C**A**G	SUB_FROM_T
75.1843	C**T**G	SUB_FROM_T
75.0326	C**A**G	SUB_FROM_C
72.1783	**C**AG	SUB_FROM_G
71.0943	C**A**G	SUB_FROM_A
70.5656	CT**G**	SUB_FROM_C
69.5391	**C**AG	SUB_FROM_T
66.3146	CA**G**	SUB_FROM_C

Table C.23.: The 20 highest Z-scores for SRR1284073

Z-Score	Motif	Error Type
-78.402	A**A**A	INSERTION
-52.6709	TT**G**	SUB_FROM_G
-52.5762	TT**G**	SUB_FROM_C
-51.5559	C**C**C	INSERTION
-51.5072	T**G**C	INSERTION
-48.2423	TT**G**	SUB_FROM_T
-46.4335	TA**G**	SUB_FROM_C
-44.9597	C**A**T	INSERTION
-44.8185	TA**G**	SUB_FROM_G
-44.6733	T**T**G	SUB_FROM_G
-43.801	C**T**A	SUB_FROM_G
-43.5637	T**T**G	SUB_FROM_C
-43.2604	T**T**G	SUB_FROM_A
-43.0627	TA**G**	SUB_FROM_T
-43.0538	C**T**A	SUB_FROM_A
-42.4221	T**A**G	SUB_FROM_G
-42.4021	**C**TA	SUB_FROM_C
-41.8836	C**T**A	SUB_FROM_T
-41.7728	C**T**A	SUB_FROM_C
-41.6571	T**A**G	SUB_FROM_C

Table C.24.: The 20 lowest Z-scores for SRR1284073

Z-Score	Motif	Error Type
128.772	CT**G**	SUB_FROM_G
124.859	CA**G**	SUB_FROM_G
116.032	C**T**G	SUB_FROM_G
112.005	C**A**G	SUB_FROM_G
101.282	C**A**G	SUB_FROM_A
100.696	C**T**G	SUB_FROM_C
100.265	**C**TG	SUB_FROM_G
99.8871	C**T**G	SUB_FROM_T
96.9638	C**T**G	SUB_FROM_A
96.2567	C**A**G	SUB_FROM_T
95.886	C**A**G	SUB_FROM_C
95.2675	**C**AG	SUB_FROM_G
93.0508	**C**TG	SUB_FROM_A
91.7736	**C**TG	SUB_FROM_C
88.0609	CT**G**	SUB_FROM_A
87.2608	**C**TG	SUB_FROM_T
86.1968	**C**AG	SUB_FROM_A
84.7839	**C**AG	SUB_FROM_T
82.4405	CA**G**	SUB_FROM_T
79.6529	**C**AG	SUB_FROM_C

Table C.25.: The 20 highest Z-scores for ecoli pacbio simulated

Z-Score	Motif	Error Type
-124.423	C**C**C	INSERTION
-122.74	G**G**G	INSERTION
-75.6867	A**A**A	INSERTION
-75.4499	T**T**T	INSERTION
-60.6689	T**T**G	SUB_FROM_G
-59.8876	T**A**G	SUB_FROM_A
-59.8748	CT**A**	SUB_FROM_A
-59.5421	C**T**A	SUB_FROM_G
-58.0337	TA**G**	SUB_FROM_G
-57.4163	TT**G**	SUB_FROM_G
-55.8463	T**A**G	SUB_FROM_G
-54.9923	C**T**A	SUB_FROM_A
-54.9782	C**A**A	SUB_FROM_G
-54.74	TA**G**	SUB_FROM_T
-54.0227	T**A**G	SUB_FROM_T
-53.2812	**C**AA	SUB_FROM_T
-52.4482	T**T**G	SUB_FROM_A
-51.2628	C**A**A	SUB_FROM_A
-50.6428	C**T**A	SUB_FROM_C
-50.5877	C**T**A	SUB_FROM_T

Table C.26.: The 20 lowest Z-scores for ecoli pacbio simulated

Z-Score	Motif	Error Type
91.1103	C**A**A	INSERTION
87.3485	T**T**G	INSERTION
84.0828	TT**G**	SUB_FROM_A
70.6093	TT**T**	INSERTION
70.5553	TA**A**	SUB_FROM_A
66.5267	GT**T**	SUB_FROM_C
65.5597	A**T**T	INSERTION
61.223	GT**C**	SUB_FROM_C
59.3146	**T**TG	SUB_FROM_A
57.6055	**T**GT	SUB_FROM_T
56.7442	AA**A**	INSERTION
55.5016	C**C**A	INSERTION
55.231	**A**AT	SUB_FROM_T
53.6438	GT**A**	SUB_FROM_C
52.959	TC**A**	SUB_FROM_A
52.3411	C**T**T	INSERTION
52.097	GG**G**	INSERTION
50.219	GA**T**	SUB_FROM_C
49.6681	CC**C**	INSERTION
49.0244	TG**G**	SUB_FROM_A

Table C.27.: The 20 highest Z-scores for ebola pacbio simulated

Z-Score	Motif	Error Type
-128.335	T**T**T	INSERTION
-101.978	A**A**A	INSERTION
-83.9685	G**G**G	INSERTION
-76.6816	C**C**C	INSERTION
-66.447	AT**G**	SUB_FROM_A
-60.1482	C**T**G	INSERTION
-59.2879	**A**AA	SUB_FROM_T
-56.2268	CT**A**	SUB_FROM_A
-56.0337	T**C**A	INSERTION
-51.6865	AT**G**	SUB_FROM_G
-51.1787	C**A**G	INSERTION
-49.6691	T**G**A	INSERTION
-49.6457	CA**T**	SUB_FROM_T
-47.0453	**T**TT	SUB_FROM_A
-46.7477	**C**TA	SUB_FROM_C
-46.7088	AT**G**	SUB_FROM_C
-45.7461	CT**A**	SUB_FROM_C
-44.7176	**C**TA	SUB_FROM_G
-44.649	CT**A**	SUB_FROM_T
-42.977	CAA**A**	SUB_FROM_A

Table C.28.: The 20 lowest Z-scores for ebola pacbio simulated

C.3. Full-Context-Specific Error Profile

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score INSERTION	0.0	0.03	0.03	0.01
F-Score SUB_FROM_A	0.65	0.69	0.7	0.65
F-Score SUB_FROM_C	0.67	0.71	0.72	0.51
F-Score SUB_FROM_G	0.54	0.72	0.72	0.36
F-Score SUB_FROM_T	0.54	0.68	0.69	0.56
Average F-Score	0.48	**0.57**	**0.57**	0.42

Table C.29.: Base-Error Classifiers for SRR396536

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score DEL_OF_A	0.7	0.63	0.7	0.73
F-Score DEL_OF_C	0.22	0.48	0.56	0.54
F-Score DEL_OF_G	0.46	0.42	0.42	0.45
F-Score DEL_OF_T	0.67	0.65	0.68	0.69
F-Score MULTIDEL	0.33	0.35	0.38	0.35
Average F-Score	0.47	0.51	**0.55**	**0.55**

Table C.30.: Gap-Error Classifiers for SRR396536

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score INSERTION	0.02	0.04	0.04	0.01
F-Score SUB_FROM_A	0.74	0.75	0.76	0.73
F-Score SUB_FROM_C	0.77	0.79	0.79	0.61
F-Score SUB_FROM_G	0.52	0.77	0.77	0.43
F-Score SUB_FROM_T	0.64	0.74	0.75	0.64
Average F-Score	0.54	**0.62**	**0.62**	0.48

Table C.31.: Base-Error Classifiers for SRR396537

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score DEL_OF_A	0.65	0.58	0.64	0.66
F-Score DEL_OF_C	0.5	0.46	0.55	0.52
F-Score DEL_OF_G	0.25	0.42	0.48	0.46
F-Score DEL_OF_T	0.59	0.57	0.66	0.62
F-Score MULTIDEL	0.12	0.38	0.47	0.34
Average F-Score	0.42	0.48	**0.56**	0.52

Table C.32.: Gap-Error Classifiers for SRR396537

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score INSERTION	0.02	0.06	0.06	0.03
F-Score SUB_FROM_A	0.29	0.38	0.4	0.35
F-Score SUB_FROM_C	0.17	0.31	0.32	0.23
F-Score SUB_FROM_G	0.43	0.44	0.48	0.35
F-Score SUB_FROM_T	0.44	0.41	0.44	0.33
Average F-Score	0.27	0.32	**0.34**	0.26

Table C.33.: Base-Error Classifiers for ecoli illumina simulated

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score DEL_OF_A	0.58	0.5	0.55	0.55
F-Score DEL_OF_C	0.46	0.53	0.57	0.59
F-Score DEL_OF_G	0.43	0.52	0.58	0.57
F-Score DEL_OF_T	0.58	0.52	0.55	0.59
F-Score MULTIDEL	0.42	0.21	0.14	0.35
Average F-Score	0.49	0.46	0.48	**0.53**

Table C.34.: Gap-Error Classifiers for ecoli illumina simulated

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score INSERTION	0	0	0	0
F-Score SUB_FROM_A	0.61	0.54	0.65	0.60
F-Score SUB_FROM_C	0.44	0.44	0.55	0.53
F-Score SUB_FROM_G	0.42	0.51	0.58	0.56
F-Score SUB_FROM_T	0.53	0.50	0.57	0.56
Average F-Score	0.50	0.40	**0.59**	0.45

Table C.35.: Base-Error Classifiers for ebola illumina simulated

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score DEL_OF_A	0.5	1	1	1
F-Score DEL_OF_C	0	0	0	0
F-Score DEL_OF_G	0	0	0	0
F-Score DEL_OF_T	0	1	1	1
F-Score MULTIDEL	0	0	0	0
Average F-Score	0.17	**0.50**	**0.50**	**0.50**

Table C.36.: Gap-Error Classifiers for ebola illumina simulated

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score INSERTION	0.37	0.44	0.39	0.35
F-Score SUB_FROM_A	0.11	0.24	0.23	0.18
F-Score SUB_FROM_C	0.3	0.24	0.23	0.25
F-Score SUB_FROM_G	0.11	0.24	0.22	0.17
F-Score SUB_FROM_T	0.12	0.23	0.2	0.13
Average F-Score	0.2	**0.28**	0.25	0.22

Table C.37.: Base-Error Classifiers for SRR1284073

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score DEL_OF_A	0.19	0.22	0.27	0.29
F-Score DEL_OF_C	0.43	0.32	0.38	0.32
F-Score DEL_OF_G	0.34	0.33	0.38	0.31
F-Score DEL_OF_T	0.28	0.23	0.26	0.3
F-Score MULTIDEL	0.14	0.19	0.16	0.14
Average F-Score	0.27	0.26	**0.29**	0.27

Table C.38.: Gap-Error Classifiers for SRR1284073

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score INSERTION	0.41	0.52	0.51	0.42
F-Score SUB_FROM_A	0.04	0.27	0.26	0.17
F-Score SUB_FROM_C	0.24	0.28	0.26	0.17
F-Score SUB_FROM_G	0.19	0.28	0.26	0.21
F-Score SUB_FROM_T	0.02	0.27	0.24	0.17
Average F-Score	0.18	**0.32**	0.31	0.23

Table C.39.: Base-Error Classifiers for ecoli pacbio simulated

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score DEL_OF_A	0.32	0.29	0.33	0.33
F-Score DEL_OF_C	0.23	0.31	0.34	0.22
F-Score DEL_OF_G	0.11	0.3	0.33	0.2
F-Score DEL_OF_T	0.41	0.29	0.3	0.36
F-Score MULTIDEL	0.08	0.1	0.07	0.15
Average F-Score	0.23	0.26	**0.27**	0.25

Table C.40.: Gap-Error Classifiers for ecoli pacbio simulated

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score INSERTION	0.43	0.48	0.52	0.43
F-Score SUB_FROM_A	0.31	0.29	0.3	0.27
F-Score SUB_FROM_C	0.02	0.23	0.21	0.13
F-Score SUB_FROM_G	0.0	0.21	0.19	0.09
F-Score SUB_FROM_T	0.18	0.26	0.24	0.22
Average F-Score	0.19	**0.29**	**0.29**	0.23

Table C.41.: Base-Error Classifiers for ebola pacbio simulated

	Naïve Bayes	DecisionTree	RandomForest	LogisticRegression
F-Score DEL_OF_A	0.4	0.47	0.53	0.4
F-Score DEL_OF_C	0.16	0.41	0.46	0.26
F-Score DEL_OF_G	0.21	0.41	0.45	0.28
F-Score DEL_OF_T	0.49	0.49	0.54	0.47
F-Score MULTIDEL	0.07	0.14	0.1	0.14
Average F-Score	0.26	0.38	**0.42**	0.31

Table C.42.: Gap-Error Classifiers for ebola pacbio simulated

D. Error Correction Unit Experiments

	Total	Substitutions	Insertions	Deletions
True Positives	465144	464977	162	5
False Positives	2227238	1670405	360492	196341
True Negatives	494019850	994410064	248602516	981669336
False Negatives	3853283	3847381	2705	3197
Sensitivity	0.107711	0.107824	0.0565051	0.00156152
Specificity	0.995512	0.998323	0.998552	0.9998
F-Score	0.132693	0.144229	0.000891283	5.01133e-05

Table D.43.: Error correction statistics for SRR396536

	Total	Substitutions	Insertions	Deletions
True Positives	522856	522727	127	2
False Positives	1516101	1353927	135277	26897
True Negatives	495336635	997034568	249258642	984311972
False Negatives	3795571	3789631	2740	3200
Sensitivity	0.121076	0.121216	0.0442972	0.00062461
Specificity	0.996949	0.998644	0.999458	0.999973
F-Score	0.164488	0.168921	0.00183697	0.000132886

Table D.44.: Error correction statistics for SRR396536, using perfect k-mer classification

	Total	Substitutions	Insertions	Deletions
True Positives	499545	499372	168	5
False Positives	2196005	1667271	319575	209159
True Negatives	528806492	1064327108	266081777	1050898860
False Negatives	5519008	5513276	2615	3117
Sensitivity	0.0830008	0.0830536	0.0603665	0.00160154
Specificity	0.995864	0.998436	0.9988	0.999801
F-Score	0.114652	0.122106	0.00104178	4.71063e-05

Table D.45.: Error correction statistics for SRR396537

	Total	Substitutions	Insertions	Deletions
True Positives	530810	530681	126	3
False Positives	1756163	1509832	203432	42899
True Negatives	529576483	1065850332	266462583	1052455600
False Negatives	5487743	5481967	2657	3119
Sensitivity	0.0881956	0.0882608	0.0452749	0.000960922
Specificity	0.996695	0.998585	0.999237	0.999959
F-Score	0.127821	0.131794	0.00122128	0.000130367

Table D.46.: Error correction statistics for SRR396537, using perfect k-mer classification

	Total	Substitutions	Insertions	Deletions
True Positives	181325	180974	348	3
False Positives	755481	690495	46740	18246
True Negatives	459411318	923400900	230850225	914244372
False Negatives	697134	693677	853	2604
Sensitivity	0.206413	0.20691	0.289759	0.00115075
Specificity	0.998358	0.999253	0.999798	0.99998
F-Score	0.199778	0.207287	0.0144132	0.000287687

Table D.47.: Error correction statistics for ecoli illumina simulated

	Total	Substitutions	Insertions	Deletions
True Positives	181799	181454	342	3
False Positives	717795	665267	38358	14170
True Negatives	459479013	923536736	230884184	914379316
False Negatives	696660	693197	859	2604
Sensitivity	0.206952	0.207459	0.284763	0.00115075
Specificity	0.99844	0.99928	0.999834	0.999985
F-Score	0.204492	0.210825	0.0171424	0.000357569

Table D.48.: Error correction statistics for ecoli illumina simulated, using perfect k-mer classification

	Total	Substitutions	Insertions	Deletions
True Positives	865	864	1	0
False Positives	3586	3103	340	143
True Negatives	1875850	3770376	942594	3733024
False Negatives	2691	2683	0	8
Sensitivity	0.243251	0.243586	1	0
Specificity	0.998092	0.999178	0.999639	0.999962
F-Score	0.216061	0.229971	0.00584795	0

Table D.49.: Error correction statistics for ebola illumina simulated, using context-free error profile and naïve k-mer classification

	Total	Substitutions	Insertions	Deletions
True Positives	902	901	1	0
False Positives	3734	2567	637	530
True Negatives	1875062	3768792	942198	3731456
False Negatives	2654	2646	0	8
Sensitivity	0.253656	0.254017	1	0
Specificity	0.998013	0.999319	0.999324	0.999858
F-Score	0.220215	0.256878	0.00312989	0

Table D.50.: Error correction statistics for ebola illumina simulated, using full-context-specific error profile and naïve k-mer classification

	Total	Substitutions	Insertions	Deletions
True Positives	873	872	1	0
False Positives	2757	2642	91	24
True Negatives	1877277	3773240	943310	3735868
False Negatives	2683	2675	0	8
Sensitivity	0.245501	0.245842	1	0
Specificity	0.998534	0.9993	0.999904	0.999994
F-Score	0.242972	0.246991	0.0215054	0

Table D.51.: Error correction statistics for ebola illumina simulated, using context-free error profile and perfect k-mer classification

	Total	Substitutions	Insertions	Deletions
True Positives	127583	7086	120477	20
False Positives	1394934	98829	1289184	6921
True Negatives	22460530	44925324	11231331	44916796
False Negatives	6081047	4660780	1060158	360109
Sensitivity	0.0205493	0.00151804	0.102044	5.55357e-05
Specificity	0.941526	0.997805	0.897034	0.999846
F-Score	0.0330049	0.00296872	0.0930218	0.000108971

Table D.52.: Error correction statistics for SRR1284073

	Total	Substitutions	Insertions	Deletions
True Positives	167727	20706	147002	19
False Positives	1165636	286929	875916	2791
True Negatives	22789140	45582488	11395622	45574072
False Negatives	6040903	4647160	1033633	360110
Sensitivity	0.0270151	0.00443586	0.124511	5.27589e-05
Specificity	0.95134	0.993745	0.928622	0.999939
F-Score	0.0444782	0.00832318	0.133423	0.000104701

Table D.53.: Error correction statistics for SRR1284073, using perfect k-mer classification

	Total	Substitutions	Insertions	Deletions
True Positives	645519	37854	607569	96
False Positives	2954637	689694	2247343	17600
True Negatives	104007330	208031576	52007894	207997744
False Negatives	11978034	9037492	2401465	539077
Sensitivity	0.0511361	0.00417108	0.201915	0.00017805
Specificity	0.972377	0.996696	0.958578	0.999915
F-Score	0.0795772	0.00772303	0.207222	0.000344785

Table D.54.: Error correction statistics for ecoli pacbio simulated

	Total	Substitutions	Insertions	Deletions
True Positives	733242	35979	697131	132
False Positives	2801704	614137	2178640	8927
True Negatives	104244137	208505124	52126281	208471424
False Negatives	11890311	9039367	2311903	539041
Sensitivity	0.0580852	0.00396448	0.231679	0.000244819
Specificity	0.973827	0.997063	0.959881	0.999957
F-Score	0.0907562	0.00739893	0.236926	0.000481548

Table D.55.: Error correction statistics for ecoli pacbio simulated, using perfect k-mer classification

	Total	Substitutions	Insertions	Deletions
True Positives	928921	37978	890766	177
False Positives	2906379	604621	2293238	8520
True Negatives	105425107	210867044	52716761	210833384
False Negatives	11426023	8770578	2121433	534012
Sensitivity	0.0751862	0.00431149	0.29572	0.000331343
Specificity	0.973171	0.997141	0.958312	0.99996
F-Score	0.114751	0.00803669	0.28752	0.000652071

Table D.56.: Error correction statistics for ebola pacbio simulated

	Total	Substitutions	Insertions	Deletions
True Positives	965268	39464	925593	211
False Positives	2894705	576638	2307199	10868
True Negatives	105524063	211065072	52766268	211031180
False Negatives	11389676	8769092	2086606	533978
Sensitivity	0.0781281	0.00448019	0.307281	0.000394991
Specificity	0.973301	0.997275	0.958107	0.999949
F-Score	0.119059	0.00837463	0.296427	0.000773931

Table D.57.: Error correction statistics for ebola pacbio simulated, using perfect k-mer classification