Lossless DNA Compression

Woramanot Yomjinda

Supervisor: Christian Steinruecken

Department of Engineering
University of Cambridge

This dissertation is submitted for the degree of
Master of Philosophy in Machine Learning and Machine Intelligence

Queens' College　　　　August 2020

Declaration

I, Woramanot Yomjinda of Queens' College, being a candidate for the MPhil in Machine Learning and Machine Intelligence, hereby declare that this report and the work described in it are my own work, unaided except as may be specified below, and that the report does not contain material that has already been used to any substantial extent for a comparable purpose. This report makes use of the following software to aid in the production of the results:

1. Arithmetic Coding infrastructure code. The code is available at `https://github.com/q4/ppm-dp/tree/master/java`.
   - The code provides fundamental building block for the models in Section 5.
   - The Java code is modified to be used in `Python` programming environment.
2. TensorFlow machine learning package. The code is available at `https://www.tensorflow.org/api_docs/python/tf/keras/layers`.
   - The package provides deep neural network models for methods in Section 5.
   - The package is used for `LSTM` and bi-directional `LSTM` models.
3. Scikit-learn machine learning package. The code is available at `https://scikit-learn.org/stable/modules/classes.html`.
   - The package provides machine learning models for methods in Section 5.
   - The package is used for Naïve Bayes, Decision Tree, and Random Forest classifiers models.

Word count: $13,898$

Woramanot Yomjinda

August 2020

Abstract

In this dissertation, our goal is to explore lossless DNA compression algorithms, experiment with predictive probabilistic models for DNA sequences, and perform a comprehensive analysis and comparison among different schemes. We omit lossy compressors as well as reference-based and pre-trained models to ensure that the algorithms are general and can also compress non-DNA files. The methods in this dissertation are based on techniques that cleanly separate the compression task into probabilistic models (predictors) and arithmetic coders (compressors). Existing methods based on arithmetic coding have become state-of-the-art in lossless compression in many domains.

As a novel contribution, we implement probabilistic models ranging from traditional machine learning models, such as Random Forests, to deep neural network models which are not conventionally used with arithmetic coding. For each model, we supply a concise and robust mathematical formalisation. As background research, we ran an exploratory data analysis on DNA sequences that are referenced in Forensic Science and DNA compression papers. We ascertain that DNA sequences exhibit particular structure, such as repetitions, codon-dictionaries, and complementary palindromes. We leverage these structural properties with a bi-directional `LSTM` model and a specific hyperparameter-tuning technique.

This dissertation performs a compression ratio comparison on the E.Coli DNA sequence. The experiment shows promising results for our bi-directional `LSTM`, with a state-of-the-art compression ratio, but at a prohibitive runtime cost. We also benchmark against a randomized sequence with the same unigram frequency distribution as a DNA sequence. We find, as expected, that standard compression algorithms do not compress DNA sequences as effectively as specialised algorithms. Our paper also performs an extensive comparison on the multi-species large DNA corpus, investigating the computation speed, compression ratio, and memory usage of several different DNA compression algorithms. This comparison table provides insights into the trade-off between speed and compression ratio for different compression methods.

Acknowledgements

Many thanks to Christian Steinruecken for being a great advisor. I am grateful for his proposal of the interesting topic of this paper and the overwhelming support that he has provided me, as well as for giving me lessons on compression techniques, a great amount of advice, and multiple video-call meetings amidst global pandemic. I am very happy that I have had a chance to further explore the topic of DNA compression from a machine learning perspective and hopefully contribute to a better understanding of this topic for DNA databanks and future researchers.

I also am beyond grateful for my family. I thank my mother, father, and older brother for encouraging and cheering me on through Cambridge and outside of Cambridge.

For the creation of this dissertation, I am grateful for Dr Tanin Bhoopat and Dr Anak Yomjinda for providing their expertise on the topic of DNA analysis in Forensic Science. I am truly grateful for Chenchen Zhang for her constant and immeasurable support.

Contents

1 Introduction **10**

1.1 Motivation . . . 10

1.2 Problem Description . . . 11

1.3 Contributions . . . 12

1.4 Dissertation Structure . . . 13

2 Literature Review **15**

2.1 Lossless Compression Techniques . . . 15

2.1.1 Non-arithmetic Coding Algorithms . . . 15

2.1.2 Arithmetic Coding Algorithm . . . 16

2.2 DNA-specific Compression . . . 17

2.2.1 Reference-free Methods . . . 17

2.2.2 Reference-based Methods . . . 19

2.3 Other Related Topics . . . 20

3 Arithmetic Coding **22**

3.1 Arithmetic Coder . . . 23

3.1.1 Probability Table and Encoding . . . 24

3.1.2 Binary Value Stream and Renormalisation . . . 25

3.2 Predictor Model . . . 28

3.2.1 Context-free/Static Predictors . . . 28

3.2.2 Context-based/Adaptive Predictors . . . 30

4 Data **31**

4.1 Common DNA Files Format . . . 31

4.1.1 FASTA . . . 31

4.1.2 FASTQ . . . 32

4.1.3 SAM . . . 33

4.1.4 BLAST . . . 33
4.2 Datasets . . . 34
4.3 DNA Traits Analysis . . . 35
4.3.1 DNA Traits in Forensic Literature . . . 36
4.3.2 EDA . . . 37

5 Method **40**
5.1 Prediction by Partial Matching (PPM) . . . 40
5.2 Traditional Machine Learning Approaches . . . 43
5.2.1 Naïve Bayes Classifier . . . 43
5.2.2 Decision Tree Classifier . . . 44
5.2.3 Random Forest Classifier . . . 45
5.3 Neural Network Approach . . . 46
5.3.1 RNN . . . 46
5.3.2 LSTM Model . . . 48
5.4 Attention Heads and Transformer . . . 51
5.5 Bi-directional LSTM + CNN + Attention . . . 53
5.6 Context Mixing Model: PAQ Family . . . 56
5.6.1 PAQ1-6: Adaptive Model Mixing . . . 56
5.6.2 PAQ8-ZPAQ: Architecture and Model . . . 57

6 Results **63**
6.1 Performance Comparison . . . 63
6.2 Categorical Accuracy Plot over Training Period . . . 67
6.3 Time vs. Performance on the DNA corpus . . . 69

7 Conclusion and Future Work **72**
7.1 Conclusion . . . 72
7.2 Future Work . . . 74

List of Figures

1 An illustration of arithmetic coding process (Moffat et al., 1998) 23

2 An example of an arithmetic encoding process when encoding a sequence 'CGTA', with a static probability value for each base: p(A)=p(T)=0.2, p(C)=0.5, p(G)=0.1 (Wang et al., 2019) 25

3 Algorithm encoder renormalisation (Moffat et al., 1998) 27

4 Algorithm decoder renormalisation (Moffat et al., 1998) 27

5 An example of static predictors: A fixed frequency table computed from the Brown Corpus to estimate English symbol frequency (Witten et al., 1987) . . 29

6 An example of a FASTA file (Hosseini et al., 2016) 32

7 An example of a FASTQ file (Hosseini et al., 2016) 33

8 An example of a SAM file (Hosseini et al., 2016) 33

9 PPM model after processing the string "abracadabra" using up to the second order model (Knoll and de Freitas, 2012). 42

10 Simple recurrent neural network (Mikolov et al., 2011) 47

11 LSTM memory cell with gating units (Sundermeyer et al., 2012) 49

12 Bi-directional LSTM model in Keras/Tensorflow 53

13 CLDNN Architecture (Sainath et al., 2015) 55

14 PAQ8 architecture (Knoll and de Freitas, 2012) 58

15 PAQ8 model mixer architecture (Knoll and de Freitas, 2012) 59

16 Categorical accuracy for various algorithms plotted over the training period during online training 68

List of Tables

1 SequenceSqueeze Competition Performance 19

2 Hutter Prize Contestants and Winners on enwik8 21

3 A example of static model with symbols {`a`, `e`, `i`, `o`, `u`, `!`}, ... 24

4 Renormalisation Example in 8-bit 28

5 The Large Corpus 34

6 The DNA Corpus 35

7 Top vs Bottom three symbols count for each N-gram in `E.Coli` and an artificial control sequence 39

8 Length-N STR-repeats comparison of `E.Coli` vs. the artificial control sequence 39

9 Count of Length-N single symbol repeats comparison in `E.Coli` and the artificial control sequence 39

10 Compression size table of various algorithms for the artificial control sequence 64

11 Compression size table of various algorithms for `E.Coli` file 66

12 Compression ratio and speed benchmark on the DNA Corpus 71

13 Full table of various compression algorithms for `E.Coli` file (Part 1) 79

14 Full table of various compression algorithms for `E.Coli` file (Part 2) 80

Nomenclature

APM	Adaptive Probability Map
BpC	Bits per Character
BPTT	Backpropagation through time
BWT	Burrows–Wheeler transform
CMX	Context Mixing Model
DNA	Deoxyribonucleic acid
FASTA	FAST-All format
KMER	k-mer partitioning
LSTM	Long Short-Term Memory
LZW	Lempel–Ziv–Welch
MAP	Maximum a posteriori probability
MTF	Move-to-Front
PPM	Prediction by Partial Matching
RLE	Run Length Encoding
SAM	Sequence Alignment/Map format
SNPs	single nucleotide polymorphisms
VINT	variable integers for positions

1 Introduction

1.1 Motivation

With the development of modern gene sequencing technology, we are generating increasingly large amounts of genomics data. These new DNA sequencing technologies, such as next-generation sequencing (NGS) and single-molecule sequencing, have enabled and advanced genomics research and functional genomics to higher levels (Kuruppu et al., 2011). In turn, databases like GenBank may double or triple in size every year (Bakr et al., 2013), and thus, the electronic storage of DNA/nucleotide sequences has become increasingly important. The ability to compress DNA sequences is one of the determining factors of how much DNA data we can store in a data centre, how fast we can transmit DNA sequences across the world, and how quickly we can access DNA data records. Data storage costs contribute a substantial proportion of total cost in the creation and analysis of DNA sequences. DNA sequences are composed of symbols from a 4-letter alphabet (one for each nucleotide), and each simple is typically represented by an 8-bit ASCII symbol. Assembled sequences from contemporary sequencing projects range from terabytes to petabytes in size (Bakr et al., 2013). As of June 2020, GenBank records hold data comprising approximately 427.8 billion bases from 217.1 million reported sequences. For set-based records (WGS/TSA/TLS), the size is on the order of 8.5 trillion bases from 1.7 billion sequences (Gen). While the increase in the DNA sequences is still manageable due to a tremendous increase in the disk storage capacity, more efficient DNA compression could help make better use of this capacity, and store larger collections of genomics libraries. Efficient lossless compression techniques and data structures can help efficiently store, access, transmit, and search these large datasets.

The goal of this DNA sequence compression project is to develop a novel compressor that can losslessly compress DNA sequences better than contemporary general-purpose compression algorithms.

1.2 Problem Description

There are two main branches of compression schemes, lossless compression and lossy compression. A lossless compression algorithm encodes an input file X to an output file Y, such that Y is expected to be smaller than X, and X must be fully recoverable from Y. Lossy compression differs from lossless compression in that it is acceptable to recover an approximate representation of X after decompression (Steinruecken, 2014a).

DNA sequences are composed of four bases: Adenine (`A`), Cytosine (`C`), Thymine (`T`), Guanine (`G`). In some organisms, `G` is swapped for Uracil (`U`). Thus, each base (symbol) of RNA and DNA can be represented by two bits 00, 01, 10, 11. Sequences of such bases are commonly stored in formats such as `FASTA`/`Multi-FASTA`, `FASTQ` and `SAM`/`BAM` (Hosseini et al., 2016).

Popular standard compression algorithms include `compress`, `gzip`, `bzip2`, and algorithms in the `PPM`-family. These algorithms were designed to do well on human text, and they often perform reasonably on other types of inputs, such as spreadsheets or program binaries. However, these algorithms weren't specifically designed for DNA, and are typically unable to compress DNA better than 2 bits per character (BpC) (Bakr et al., 2013).

We evaluate how well we compress the DNA data through the average compression ratio in units of "bits per base", where each character is a nucleotide base. A uniformly random sequence from a 4-symbol alphabet can be compressed with a rate of 2 BpC. However, as DNA sequences are not completely random, we would expect to be able to compress DNA sequences with a rate smaller than 2 BpC, given a sufficiently intelligent context-aware compression algorithm. The basic criteria for success for this dissertation are to create an algorithm that satisfies the following:

1. The program must be able to compress *any* input file, including non-DNA data (for which the algorithm is not specifically designed).

2. The program can decompress any output file to recover the original input file perfectly, i.e. it must be a lossless compression algorithm.

3. The program is practically useful for compressing DNA sequences, i.e. the algorithm must compress and decompress with reasonable usage of memory and CPU time, and must compress better than the 2 BpC baseline.

1.3 Contributions

Existing state-of-the-art DNA compression algorithms are generally format-specific and unable to compress non-DNA files (or files outside of the format). The currently best performing algorithms utilize reference-based approaches, using external DNA sequences as reference files. These algorithms typically cannot compress files outside of prespecified formats. For instance, in SequenceSqueeze's `FASTQ`-compression competition, James Bonfield's `Fqzcomp` dominates most of the contest's categories – memory usage, speed, and ratio – while maintaining total data integrity, but cannot compress generic text files (Holland and Lynch, 2013).

The objective of this dissertation is to design a compression algorithm that embeds a reasonable prior and understanding of DNA characteristics, rather than to build the state-of-the-art search system for sequence matching with reference files. Another main focus is to explore and verify the effectiveness of compression algorithms that have not been conventionally used. The novel contributions of this dissertation are:

Contribution 1: This dissertation scrutinises nucleotide sequences through the lens of the **Forensic, Scientific, and DNA Compression** literature. We examine DNA sequences from these literature, and confirm their characteristics through novel data analysis.

Contribution 2: We bring **DNA-specific traits** to existing algorithms by using designated hyper-parameter tuning, which leads to an increase in compression effectiveness for DNA compression.

Contribution 3: We conduct a study **comparing** different lossless DNA compression methods, including standard algorithms, recent methods, and our own approaches. We investigate and compare the speed, compression ratio, and memory usage of these algorithms.

We also benchmark standard compressors on an artificially created pseudo-DNA sequence, sampled from a uniform distribution over nucleotides.

Contribution 4: This dissertation presents a comparison table made using a **large multi-species DNA corpus**, with DNA ranging from that of viruses to that of humans. Our extensive comparison allows researchers to pick appropriate compressors for different types and lengths of sequences.

Contribution 5: Algorithms adopted in this dissertation include online learning-capable **traditional machine learning** methods (which are seldom used for compression), as well as a neural network model that sacrifices speed for state-of-the-art compression power.

Contribution 6: This project provides a **mathematical formalisation** for every model, providing deeper insight why specific models outperform or underperform on particular types of data.

Contribution 7: We introduce the concept of a **bi-directional context** to DNA compression, which allows the compression algorithm to utilise the palindromic and complementary-palindromic nature of the DNA sequences.

1.4 Dissertation Structure

This dissertation has 7 sections. Following the introduction, Section 2 discusses related topics and common lossless compression techniques. We also include a discussion on state-of-the-art DNA-specific compression. Section 3 explains the concept of arithmetic coding, and how compression can be decomposed into a predictive model and an arithmetic coder. Section 4 provides an in-depth explanation of DNA formats and data sets, and analyzes DNA characteristics from multiple perspectives. Section 5 describes a few different models for predictors, ranging from traditional machine learning methods to neural network online learning models. We then continue with the family of context mixing models. Afterwards, we present the results for comparison among algorithms in Section 6, and conclude the dissertation in section 7. We make suggestions for future work in the same section.

2 Literature Review

2.1 Lossless Compression Techniques

Many data compression algorithms and techniques exist in the lossless compression literature. Different formats of data sometimes have different algorithms dedicated to them; any particular type of data may have various compression algorithms, with possibly different approaches. While we implement DNA compression with an Arithmetic Coder at its core, we first briefly review a few different approaches to data compression.

2.1.1 Non-arithmetic Coding Algorithms

Run Length Encoding Algorithm (RLE): RLE is a rudimentary data transformation technique that is used in some compression algorithm. The transform identifies sequences of a repeating symbol ('runs') and replaces these with a tuple that encodes the symbol and a count (the length of the run). Non-runs are encoded as is. How these RLE-transformed representations are encoded to a sequence of bits depends on the choice of a final coding step, which is often chosen based on convenience. The effectiveness of RLE is entirely based on the repetitiveness and the length of these repetitions. For example, in a sequence "`ATCGGG`", `GGG` is a run and `ATC` is a non-run (Kodituwakku and Amarasinghe, 2010).

Huffman Encoding: Huffman Encoding is a technique for developing a fixed set of code-words for symbols giving a fixed probability distribution over those symbols. The probability distribution is often calculated from symbol frequencies. Code-words for each symbol are assigned by the Huffman algorithm based on the probability distribution. Longer code-words are assigned to symbols with smaller probabilities and shorter code-words are assigned to larger probabilities. In practice, the Huffman algorithm constructs a binary tree, assigning symbols as leaves based on probabilities and taking the binary labels of path from the root to the leaf as code-words. Huffman Encoding calculates the probabilities from the source then constructs a binary tree whose details are saved and transferred along with the compressed file. The tree typically needs to be transmitted, as it is needed for decompression (Kodituwakku and Amarasinghe, 2010).

Dictionary Compression Algorithms: By definition, a dictionary is a structure similar to a table which stores sets of words in a language using indexes instead to represent long or common words. Frequently occurred string patterns are indexed and stored in the dictionary. These index values are then used to represent the corresponding string patterns during the compression process. Adaptive dictionary compress algorithms construct a dictionary as they compress, and, thus, the decompression process does not require an attached dictionary. Examples of a dictionary-based model include the Lempel-Zev algorithm (`LZ`), `LZ77`, and the Lempel-Zev-Welch algorithm (`LZW`) (Kodituwakku and Amarasinghe, 2010).

Many well-known compression algorithms use a combination of the aforementioned algorithms with other algorithms. `Compress` uses `LZW`. `gzip` uses `LZ` and Huffman Coding. `bzip2` uses Burrows–Wheeler transform (`BWT`), Move-to-Front (`MTF`), and Huffman Coding (Bakr et al., 2013).

2.1.2 Arithmetic Coding Algorithm

While `RLE`, `Huffman`, and `LZW` are standard and well-known text compression algorithms, algorithms based on arithmetic coding have typically outperformed other techniques in terms of compression effectiveness, as evidenced by multiple independent benchmarks (Hutter, 2006; Mahoney, 2011). The decoupling of probabilistic prediction and encoding provides substantially more modeling flexibility than algorithms that are based on direct transformations of the inputs. Compression based on arithmetic coding always consists of two parts: a predictor and an arithmetic encoder/decoder. The better the prediction for the next symbol, the better the compression. We provide a full description in a subsequent section. The most important properties of arithmetic coding are as follows (Bell et al., 1989):

1. Arithmetic coding can code a symbol that has occurrence probability p in a number of bits arbitrarily close to $\log \frac{1}{p}$.

2. Arithmetic coding allows the symbol probabilities to be different at each step.
3. Arithmetic coding requires minimal memory, regardless of the number of conditioning classes in the model.
4. Arithmetic coding is fast.
5. Arithmetic coding runs sequentially and cannot be parallelised.

An important advantage is that by using arithmetic coding, each symbol can add a "fractional number of bits" to the output (Bell et al., 1989). In general, arithmetic coding achieves good compression results when combined with a context-aware predictive model (such as PPM), and a context mixing model (such as PAQ).

2.2 DNA-specific Compression

There are numerous compressors specifically designed for DNA formats, e.g. FASTQ/SAM. Two main types of these genomics sequence compressors are reference-based methods and reference-free methods. Many of these DNA-specific compressors have been submitted to the SequenceSqueeze DNA compression competition (Holland and Lynch, 2013).

2.2.1 Reference-free Methods

The basic idea of reference-free DNA sequence compression is the exploitation of structural properties, such as palindromes, as well as statistical properties of the sequences.

For example, Biocompress is similar to the LZ compression technique, but contains some DNA-specific modifications. The algorithm detects repeats and complementary palindromes in the DNA input sequence. Biocompress then encodes these repeats and palindromes using the index of the occurrence along with the type and length of the repeat/palindrome. As an extension, the Biocompress-2 algorithm adds an order-2 contextual frequency learner and arithmetic coder, for cases when no significant repetition or palindrome is present (Hosseini et al., 2016).

However, the top performers in the SequenceSqueeze competition are Quip (Jones et al., 2012), SCALCE (Hach et al., 2012), and Fastzq (Hosseini et al., 2016).

Quip is mainly used for compressing files in the FASTQ format, but it also supports the SAM and BAM file formats. Quip exploits "codons" – the natural triplets trait of DNA sequence, supported with arithmetic coding. Quip utilises order-3 models and high order Markov chains and has two main variants, Quip-r (reference-based compression) and Quip-a (assembly-based compression). For Quip-a, the model assembles a reference data based on the first 2.5 millions characters (Hosseini et al., 2016).

SCALCE is a boosting technique that achieves higher compression effectiveness and speed through symbol reorganisation. The scheme uses the *locally consistent parsing* (LCP) technique to derive long core substrings which are shared between the symbols in the source. The technique then reorganises the file by bucketing these core substrings into buckets and compressing the result with LZ variants in each bucket (Hosseini et al., 2016).

Fastqz scheme is an arithmetic coding-based method. Fastqz tunes context models/predictors to the data format, such as FASTQ, by specifying the context models in ZPAQ format through the libzpaq compression library. ZPAQ is a part of PAQ family, a group of state-of-the-art context mixing models that use an adaptive ensemble of multiple context models in combination with bit-wise predictions. Fastqz then utilises a byte-wise coder with ZPAQ predictors to achieve high compression ratio (Hosseini et al., 2016).

Below, we show a performance comparison of these methods on 2 sampled human DNA sequences SRR027520_1 and SRR065390_1 to gain a better understanding of their relative weaknesses and strengths (Bonfield and Mahoney, 2013).

Table 1: SequenceSqueeze Competition Performance

Program	Performance (fraction of original size)		Memory usage (MB)	
	SRR027520	SRR065390	SRR027520	SRR065390
SCALCE -slow	0.2572	0.1635	5162	5257
DSRC -slow	0.2477	0.1524	1058	1965
Quip -slow	0.2219	0.1584	777	775
Fastzq -slow	**0.2195**	**0.1340**	1459	1527
gzip	0.3535	0.2805	**1**	**1**
bzip2	0.2905	0.2250	7	7

Understanding how each of these models leverages DNA-characteristics is helpful for building a successful reference-less DNA compression model.

2.2.2 Reference-based Methods

Reference-based DNA compression models have a set of reference sequences to accompany both the compression and the decompression process. Therefore, these models specialise in sequence-matching between reference and target sequences, aligning the sequences and encoding any mismatches between the sequences. Example methods in this category include `CRAM` (Fritz et al., 2011) and `Goby` (Bonfield and Mahoney, 2013), which are format-specific to `FASTQ` files. Because of the access to reference sequences, these methods can reach much higher compression ratios than reference-less methods.

When not limited to `FASTQ`, and given access to a large reference genome (4 GB), the `DNAzip` algorithm provides state-of-the-art performance for compressing James Watson's genome data set (Christley et al., 2009). `DNAzip` leverages the high similarity between different human genomes, saving only the differences to the DNA reference sequence. These differences in the data are separated into three types:

1. Substitutions of single nucleotides (`SNPs`): the changed letter is recorded along with the position in the sequence.

2. Insertions of multiple nucleotides: the sequence of nucleotide letters to be inserted is

recorded along with the position of insertion.

3. Deletions of multiple nucleotides: the length of the deletion is recorded along with the position of deletion.

After finding and recording these differences, multiple post-processing techniques are used to achieve a high compression ratio. These techniques include variable integers for positions (VINT), delta positions (DELTA), SNP mapping (DBSNP) and k-mer partitioning (KMER). With 3 GB of DNA sequences as reference and 1.2 GB of SNPs as reference, DNAzip can compress 3 GB of DNA sequences into a 4 MB file (Christley et al., 2009; Hosseini et al., 2016).

2.3 Other Related Topics

The following two areas are closely related to lossless DNA compression: Lossy DNA Compression, and Human Knowledge Compression.

Human Knowledge Compression: The field of human knowledge compression is motivated by the concept that a better compression ratio implies higher intelligence, thus measuring intelligence by the size of compressed files. Lossless data compression requires the compressor to comprehend and predict intrinsic patterns in the files. Thus, for a compressor to outperform another on an unknown input that represents a large collection of human knowledge, it needs to be "more intelligent" than the other compressor. The Hutter Prize is the most well-known competition in this field. Hutter posits that a good snapshot of the human world knowledge is represented by the online encyclopedia Wikipedia. enwik9 is a representative 1 GB extract from Wikipedia. The current state-of-the-art compressor in this area uses a dictionary for preprocessing and a context mixing model (CMX) for compression. Table 2 below, except for durilca, are past winners of the Hutter Prize (Hutter, 2006).

Table 2: Hutter Prize Contestants and Winners on enwik8

Program	Performance		Resource	
	Size	BpC	Time	Memory
phda9	15,242,496	1.225	5h	1048 MB
decomp8	15,932,968	1.278	9h	936 MB
paq8hp12	16,481,655	1.319	9h	936 MB
paq8hp5	16,898,402	1.366	5h	900 MB
durilca0.5h	17,958,687	1.444	30min	1650 MB
paq8f	18,289,559	1.466	5h	854 MB

Lossy DNA Compression: Unlike lossless compression, lossy compression can result in much smaller output files because the original file only needs to be partially recoverable. For example, `LEON` utilises a probabilistic de Bruijn graph/bloom filter. All information that is included in the compressed file (except for the bloom filter) is encoded with arithmetic coding. On 20 samples of human sequences file, `LEON` achieves a compression of 1/11.4 of the original size compared to `gzip`'s 1/2.59 (Benoit et al., 2015) at a 14.37% loss of accuracy. While extremely powerful, `LEON` cannot fully recover the original file without errors. However, DNA sequences display high redundancy across reads, i.e most of their characteristics remain the same even if some symbols are lost or replaced. Lossy compression may, for some purposes, be deemed sufficient (Hosseini et al., 2016).

3 Arithmetic Coding

An arithmetic coding encodes, the target sequence as a sub-interval within the real number interval $[0, 1)$. This sub-interval shrinks in size as the target sequence increases in length, which leads to more number of bits required to represent the location of the interval. Each symbol to be encoded reduces the interval size depending on the probability of the symbol, with the constraint that the probability cannot be 0 (Witten et al., 1987).

In this section, we describe an arithmetic coding algorithm that allows for an almost optimal compression output when given an input sequence and a corresponding sequence of probability distributions. The output sequence will have a length (in bits) that is roughly equal to the input sequence's Shannon information content. Arithmetic coding provides a unique advantage by cleanly separating the data modelling task from the code generation task. This separation allows us to focus on data modelling for our compression methods, and leave the rest to an arithmetic coder (Steinruecken, 2014b).

Overall, the two main components of such a compression process are the arithmetic coder and a predictor model. The model is identical for both compression and decompression. The general structure of an arithmetic coding process is shown in Figure 1:

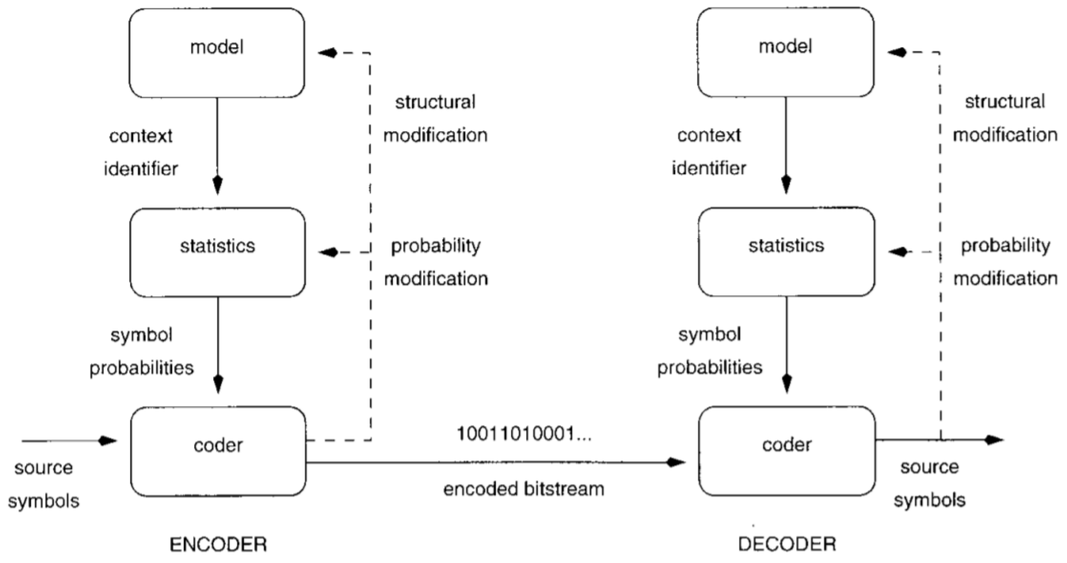


Figure 1: An illustration of arithmetic coding process (Moffat et al., 1998)

3.1 Arithmetic Coder

Both compression and decompression processes rely fundamentally on the cumulative probability range. During encoding, the encoder first calculates cumulative probabilities and generates an array of intervals, one for each symbol in the source alphabet. As the encoder reads symbol-by-symbol from the input sequence, it selects the interval corresponding to that symbol from the cumulative probability range. The coder may update the predictor based on the symbol read. The selected interval is then further split into sub-intervals using the probabilities of each symbol. The encoder then reads the next symbol and repeats the process by choosing the next sub-ranges. Each symbol from the source is read once until the end of the sequence. Any real number that lies within this final interval contains the information of the entire input sequence. The binary representation of such a number is the output sequence of the arithmetic encoder. During decoding, the encoded output can be decoded using the same probability distribution and predictor update model as the encoding process, reproducing the same sequence of internal states, and the original input sequence

(Kodituwakku and Amarasinghe, 2010).

3.1.1 Probability Table and Encoding

Before the encoding or decoding process begins, we initiate the range for encoding and decoding to be the semi-open interval $0 \leq x < 1$ or $[0, 1)$. This range narrows as we parse the target sequence symbol-by-symbol based on the probability allotted for the given symbol. Witten et al. (1987) give an example where the vocabulary only contains {`a`, `e`, `i`, `o`, `u`, `!`}, each with fixed probability as shown in Table 3.

Table 3: A example of static model with symbols {`a`, `e`, `i`, `o`, `u`, `!`},

Symbol	Probability	Range
a	.2	[0.0,0.2)
e	.3	[0.2,0.5)
i	.1	[0.5,0.6)
o	.2	[0.6,0.8)
u	.1	[0.8,0.9)
!	.1	[0.9,1.0)

Witten et al. (1987) illustrates that to transmit `eai`, the encoder and decoder initialise to $[0, 1)$. When encoder reads `e`, it reduces the encoding range to $[0.2, 0.5)$, which is the allocated range for `e` in Table 3. The encoder reads `a` which is allocated $[0, 0.2)$. Thus our new range is the range $[0, 0.2)$ within the range $[0.2, 0.5)$, which is $[0.2, 0.26)$. Next, we parse `i` so we have to zoom into the range $[0.5, 0.6)$ of $[0.2, 0.26)$, which further reduces our range to $[0.23, 0.236)$.

DNA sequences have only 4 symbols: `A`, `T`, `C`, `G`. Wang et al. (2019) provides an example for encoding a sequence '`CGTA`' using a fixed probability model with the following probabilities $p(\texttt{A}) = p(\texttt{T}) = 0.2$, $p(\texttt{C}) = 0.5$, $p(\texttt{G}) = 0.1$. Again, the coder initialises the interval to $[0, 1)$, and by reading the first symbol '`C`', the interval narrows to $[0.2, 0.7)$. Next, reading '`G`' reduces the interval to $[0.55, 0.6)$ and the process continues. Because each subsequent interval is a subset of the preceding interval, the interval becomes narrower after every symbol. After encoding '`A`', the coder reaches the final interval of $[0.59, 0.592)$, within which we can select any value (such as the middle value 0.591) to represent the original input sequence. For the best compression, the output of the target sequence is the binary value stream of the interval. The interval selection procedure of an arithmetic encoder for a sequence ‘CGTA’ is demonstrated in Figure 2 (Wang et al., 2019).

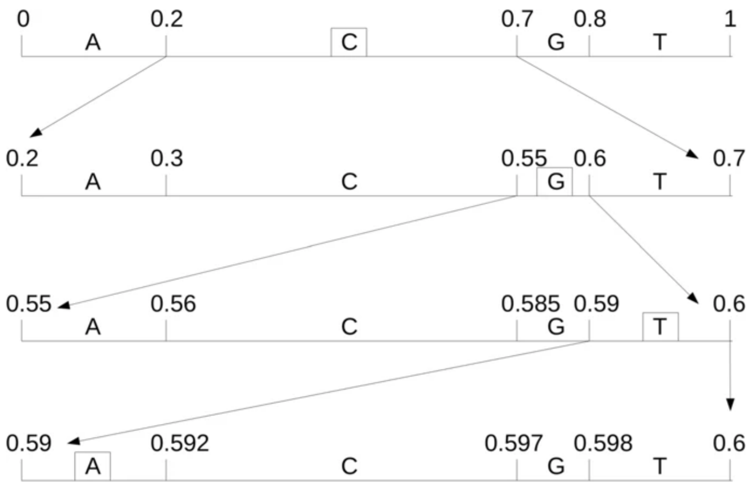


Figure 2: An example of an arithmetic encoding process when encoding a sequence ‘CGTA’, with a static probability value for each base: p(A)=p(T)=0.2, p(C)=0.5, p(G)=0.1 (Wang et al., 2019)

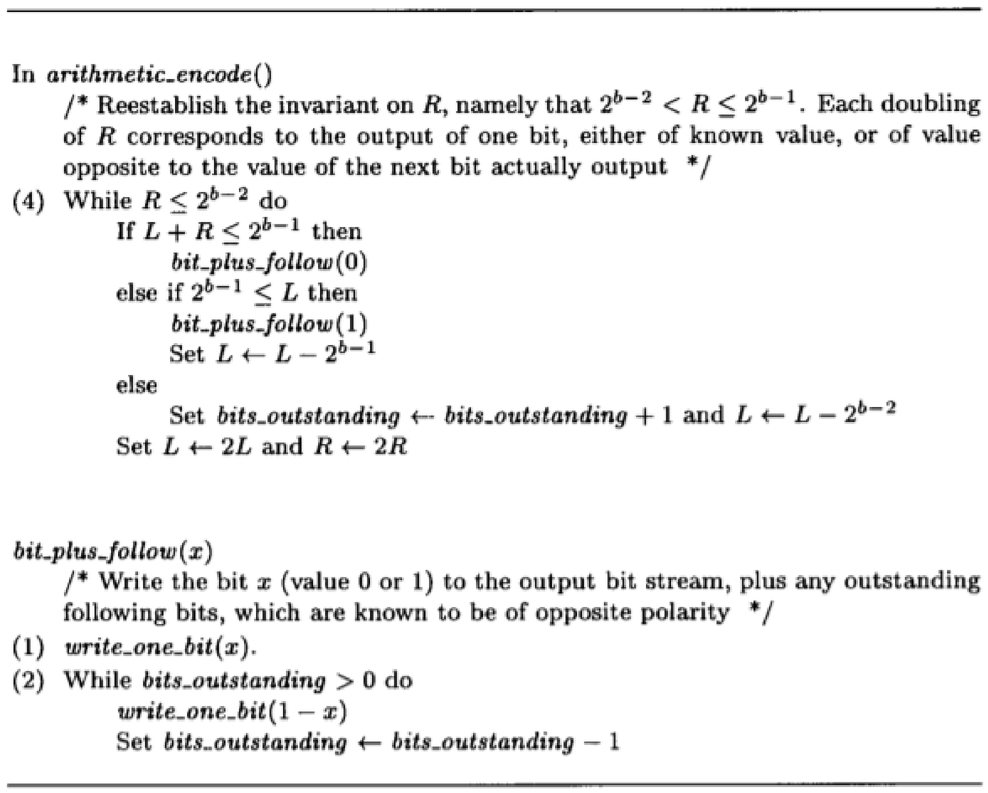
Note that instead of a probability distribution over symbols, we can have a frequency table that records the occurrence counts of each character. We can convert such a frequency table into a probability distribution, for example by summing up the total count and dividing each character’s count by the total. Another way to attain a distribution from a frequency table is to use a Dirichlet process. A dynamically updated frequency table is often used as a rudimentary adaptive predictor model.

3.1.2 Binary Value Stream and Renormalisation

In Section 3.1.1, we determined that our encoded message for the sequence ‘CGTA’ must be within the interval $[0.59, 0.592)$. Within this interval, for instance, we can pick 0.5904, 0.5908, 0.5912, or 0.5916, which are all equal in length. This redundancy implies that the use of decimal digits in this way can lead to inefficiency. To produce a compressed binary output sequence, we represent a point within the final interval with a fixed-point binary number using minimal precision. We could use 0.100101111 ($303/512 = 0.5917$ in decimal) which only costs 9 bits. Note that if there are any zeroes in the tail, they must be specified in the binary form to prevent the message from being ambiguous unless the size of the compressed stream is included.

Our 9 bits output, however, is still larger than the information content of the message, which we can calculate as $\sum -\log_2(p_i) = -\log_2(0.5) - \log_2(0.1) - \log_2(0.2) - \log_2(0.2) = 8.966$ bits, leading to about 0.4% inefficiency. This inefficiency becomes insignificant for longer sequences, and is rarely a concern in practice.

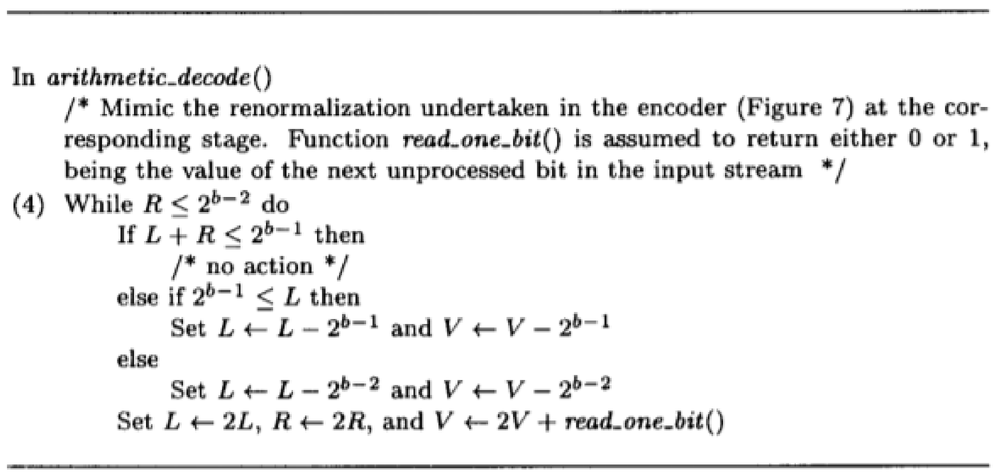
In technical terms, Moffat et al. (1998) assume the coding interval to be reflected through two variables L and R, where L is the lower-bound and R is the interval size. They are represented by unsigned b-bit integers, where b might typically be 32 or 64. Thus, the encoded target can be represented by $[L, L + R)$ at any given state. Because imprecise coding interval division can lead to a loss in compression effectiveness, we maintain R to be as large as possible while keeping R in the interval of $2^{b-2} < R \leq 2^{b-1}$ before each coding step. This is accomplised by periodically renormalising R. We illustrate the renormalisation process of L, R for the encoder in Figure 3 and for the decoder in Figure 4 (Moffat et al., 1998).



```
In arithmetic_encode()
    /* Reestablish the invariant on $R$, namely that $2^{b-2} < R \leq 2^{b-1}$. Each doubling
    of $R$ corresponds to the output of one bit, either of known value, or of value
    opposite to the value of the next bit actually output  */
(4) While $R \leq 2^{b-2}$ do
        If $L + R \leq 2^{b-1}$ then
            bit_plus_follow(0)
        else if $2^{b-1} \leq L$ then
            bit_plus_follow(1)
            Set $L \leftarrow L - 2^{b-1}$
        else
            Set bits_outstanding $\leftarrow$ bits_outstanding $+ 1$ and $L \leftarrow L - 2^{b-2}$
        Set $L \leftarrow 2L$ and $R \leftarrow 2R$


bit_plus_follow($x$)
    /* Write the bit $x$ (value 0 or 1) to the output bit stream, plus any outstanding
    following bits, which are known to be of opposite polarity  */
(1) write_one_bit($x$).
(2) While bits_outstanding $> 0$ do
        write_one_bit($1 - x$)
        Set bits_outstanding $\leftarrow$ bits_outstanding $- 1$
```

Figure 3: Algorithm encoder renormalisation (Moffat et al., 1998)



```
In arithmetic_decode()
    /* Mimic the renormalization undertaken in the encoder (Figure 7) at the cor-
    responding stage. Function read_one_bit() is assumed to return either 0 or 1,
    being the value of the next unprocessed bit in the input stream  */
(4) While $R \leq 2^{b-2}$ do
        If $L + R \leq 2^{b-1}$ then
            /* no action */
        else if $2^{b-1} \leq L$ then
            Set $L \leftarrow L - 2^{b-1}$ and $V \leftarrow V - 2^{b-1}$
        else
            Set $L \leftarrow L - 2^{b-2}$ and $V \leftarrow V - 2^{b-2}$
        Set $L \leftarrow 2L$, $R \leftarrow 2R$, and $V \leftarrow 2V +$ read_one_bit()
```

Figure 4: Algorithm decoder renormalisation (Moffat et al., 1998)

For renormalisation in the encoding process, if both L and $L + R$ are less than or equal to 2^{b-1}, we will output a 0-bit then adjust L and R afterwards. Similarly, if both L and

$L + R$ are larger than or equal to 2^{b-1}, we will output a 1-bit and adjust L and R accordingly. When neither of the above happens and $R \leq 2^{b-2}$, our encoding interval $[L, L + R)$ must straddle 2^{b-1}. In this case, we add 1 to the variable `bits_outstanding` so the next 0 or 1 bit output is followed by one more opposing bit, e.g. 100 or 011 for $bits\_outstanding = 2$ (Moffat et al., 1998).

Below, we provide an illustration of the process for the model dividing the interval $[0, 1)$ into three one-thirds ranges. Each one-third interval is then approximated with 8-bit precision, which, due to known precision, can be encoded as binary ranges for our use (Witten et al., 1987; MacKay, 2003).

Table 4: Renormalisation Example in 8-bit

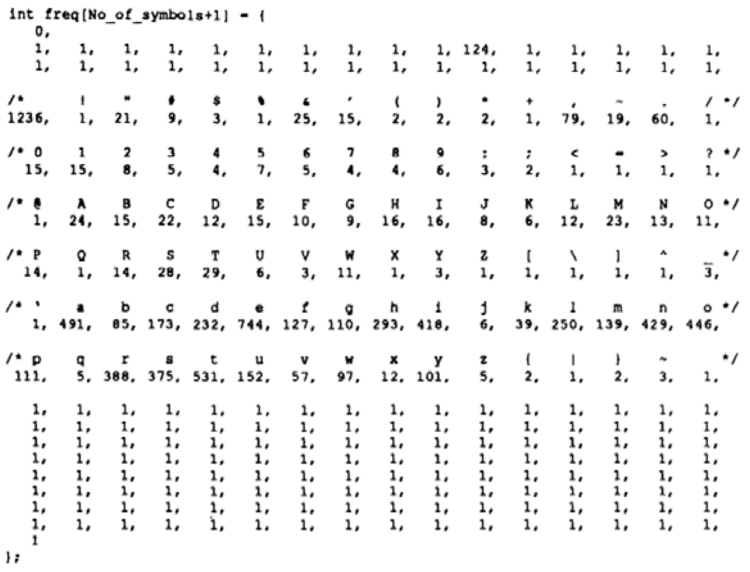
Sym.	Prob.	Interval in 8-bit precision		Range	To Output	Post-renorm. Range
		In fractions	In Binary			
A	1/3	$[0, 85/256)$	[0.00000000,0.01010101)	00000000–01010100	0	00000000–10101001
B	1/3	$[85/256, 171/256)$	[0.01010101,0.10101011)	01010101–10101010	None	01010101–10101010
C	1/3	$[171/256, 1)$	[0.10101011,1.00000000)	10101011–11111111	1	01010110–11111111

3.2 Predictor Model

As shown in section 3.1.1, the compression effectiveness of arithmetic coding depends primarily on the predictive power of its probability table. Predictors are either static or context-aware algorithms that populate the probability table. This provides an arithmetic coding method with high flexibility, as the predictor algorithm can range from the simplest frequency table to a complex deep learning model.

3.2.1 Context-free/Static Predictors

We start with a static predictor model, which is a simple model whose frequencies or probabilities for each symbol are fixed. For instance, one might estimate symbol frequencies for each English character from a corpus, as exemplified in Figure 5.



```
int freq[No_of_symbols+1] = {
    0,
    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,  124,    1,    1,    1,    1,    1,
    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,

/*        !     "     #     $     %     &     '     (     )     *     +     ,     -     .     / */
 1236,    1,   21,    9,    3,    1,   25,   15,    2,    2,    2,    1,   79,   19,   60,    1,

/*  0     1     2     3     4     5     6     7     8     9     :     ;     <     =     >     ? */
   15,   15,    8,    5,    4,    7,    5,    4,    4,    6,    3,    2,    1,    1,    1,    1,

/*  @     A     B     C     D     E     F     G     H     I     J     K     L     M     N     O */
    1,   24,   15,   22,   12,   15,   10,    9,   16,   16,    8,    6,   12,   23,   13,   11,

/*  P     Q     R     S     T     U     V     W     X     Y     Z     [     \     ]     ^     _ */
   14,    1,   14,   28,   29,    6,    3,   11,    1,    3,    1,    1,    1,    1,    1,    3,

/*  `     a     b     c     d     e     f     g     h     i     j     k     l     m     n     o */
    1,  491,   85,  173,  232,  744,  127,  110,  293,  418,    6,   39,  250,  139,  429,  446,

/*  p     q     r     s     t     u     v     w     x     y     z     {     |     }     ~       */
  111,    5,  388,  375,  531,  152,   57,   97,   12,  101,    5,    2,    1,    2,    3,    1,

    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,    1,
    1
};
```

Figure 5: An example of static predictors: A fixed frequency table computed from the Brown Corpus to estimate English symbol frequency (Witten et al., 1987)

Nevertheless, basing symbol frequencies on counts from a corpus means that some symbols will have a zero frequency count. A classic method is to give these symbols a frequency equal to one, so the predictor still works in a file with all possible 256 symbols. For a frequency table, a simplistic implementation might normalise to a certain total number, such as 8,000, for ease of conversion to a probability table (Witten et al., 1987).

However, using a static model for arithmetic coding has glaring disadvantages. The arithmetic coder will only perform well in highly specific circumstances and perform poorly for general-purpose tasks. Especially for English text, the probability for the next character depends heavily on the preceding characters. Using a static predictor will lead to a loss in compression effectiveness (Cleary and Witten, 1984). Under general conditions, Witten et al. (1987) claim that static predictors will not provide a higher compression ratio than adaptive predictors which we describe in the next section.

3.2.2 Context-based/Adaptive Predictors

Unlike a static model, an adaptive predictor model learns the symbol frequencies/probabilities as it processes through the target sequence. A simple adaptive predictor initialise each symbol in the alphabet with a count of 1 to reflect a lack of prior information, and then update the model as each symbol is read, to better reflect the observed frequencies. Given that the encoder and the decoder utilise the same updating algorithm and the same initial values for the frequency table model (or same prior probabilities for a probability table model), their predictors will produce identical predictions at every step. The encoder updates its copy of the predictor after reading and encoding the next symbol. The decoder identifies the next symbol with the current model, and then updates the predictor with the newly decoded symbol. Thus, the two processes are in perfect correspondence, allowing lossless compression of the target sequence and its full recovery via decompression (Witten et al., 1987).

Updating the predictor, however, can be computationally expensive. To use adaptive predictors in a frequency table model, the cumulative total has to be maintained for every update to ascertain that the updated model will not experience an integer overflow. One way to prevent this situation is to rescale all frequencies down when a count exceeds a safe threshold. For the probability table model, we also need to update the entire table anytime there is a context or model update, leading to longer computation time (Witten et al., 1987).

Another important trait is that both arithmetic encoding and decoding are performed in a single pass through the target sequence. Thus, the model updates and statistics can only be collected from the preceding portion of the sequence. Throughout the process, the model is updated continually symbol-by-symbol. Well-known examples of context-based predictors include Langdon and Rissanen's fixed-order Markov models (Rissanen, 1976) and Cleary and Witten's classical `PPM` approach (Cleary and Witten, 1984).

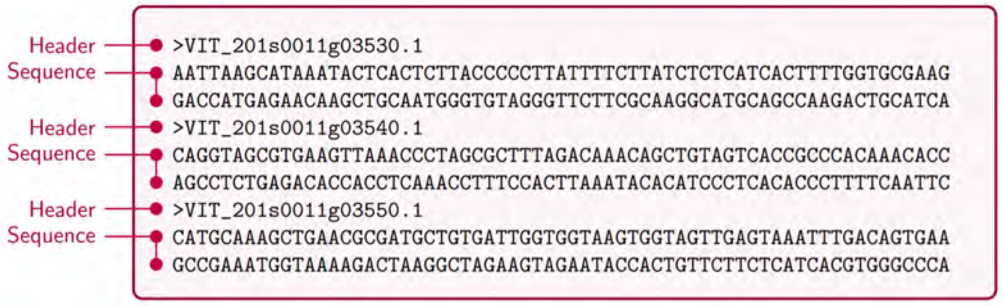
4 Data

4.1 Common DNA Files Format

In bioinformatics, handling a profusion of ambiguous and often poorly defined file formats can be a challenge. Over time, several formats have achieved the status of a *de-facto* standard, most of which are ad-hoc human-readable formats. Bill Pearson defined the `FASTA` sequence file format as an input format for Pearson's `FASTA` toolsets. `FASTA` has over time evolved to become a consensus DNA sequence format. Later on, the `FASTQ` format for DNA emerged as a natural extension for the `FASTA` format and has become another ubiquitous de-facto format for data exchange among differed tools. `FASTQ` extends the `FASTA` format by storing numeric quality scores for each nucleotide in the DNA sequence. However, the `FASTQ` format suffers from the lack of clear definition (Cock et al., 2010). Li et al. (2009) invented the Sequence Alignment Map (`SAM`) as a generic alignment format to store read alignments against given reference sequences. For DNA-specific compression, the state-of-the-art compressors often convert `FASTA` or `FASTQ` files to `SAM` files before performing the actual compression (Bonfield and Mahoney, 2013).

4.1.1 FASTA

The `FASTA` format is a classical text-based representation for DNA sequences or amino acid (protein) sequences. Each amino acid or nucleotide in the sequence is represented by a single-letter code (`A`, `T`, `C`, `G` for DNA). Sequence names and comments may precede the DNA sequence in `FASTA` format. The header line of the file is called the *description line* (`defline`), beginning with '>' and followed by any additional comments. The next line is the *sequence line* as shown in Figure 6. Multiple `FASTA` files of different DNA sequences can be concatenated into a single "`Multi-FASTA`" file. There are also special characters such as `N` which means "either `A`,`C`,`T`,`U`,`G`" or `B` which means not `A` (Pearson and Lipman, 1988).


Header
Sequence
Header
Sequence
Header
Sequence

```
>VIT_201s0011g03530.1
AATTAAGCATAAATACTCACTCTTACCCCCTTATTTTCTTATCTCTCATCACTTTTGGTGCGAAG
GACCATGAGAACAAGCTGCAATGGGTGTAGGGTTCTTCGCAAGGCATGCAGCCAAGACTGCATCA
>VIT_201s0011g03540.1
CAGGTAGCGTGAAGTTAAACCCTAGCGCTTTAGACAAACAGCTGTAGTCACCGCCCACAAACACC
AGCCTCTGAGACACCACCTCAAACCTTTCCACTTAAATACACATCCCTCACACCCTTTTCAATTC
>VIT_201s0011g03550.1
CATGCAAAGCTGAACGCGATGCTGTGATTGGTGGTAAGTGGTAGTTGAGTAAATTTGACAGTGAA
GCCGAAATGGTAAAAGACTAAGGCTAGAAGTAGAATACCACTGTTCTTCTCATCACGTGGGCCCA
```

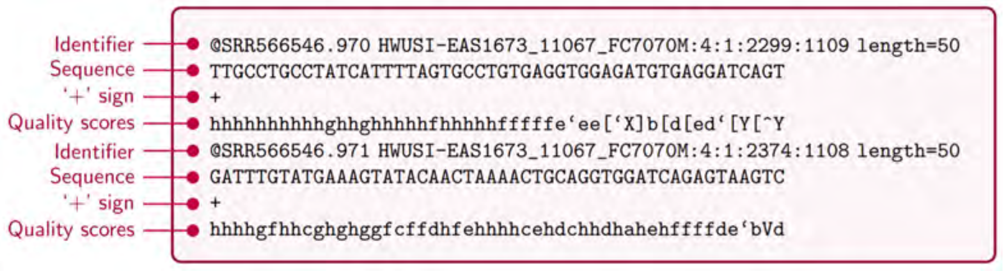
Figure 6: An example of a FASTA file (Hosseini et al., 2016)

4.1.2 FASTQ

Unlike FASTA, FASTQ represents nucleotide sequences with alphabets and includes a quality score, represented by an ASCII character, for each symbol. FASTQ is the de facto standard for output of the high-throughput sequencing instruments. The FASTQ format has four main components:

1. A sequence identifier.
2. The raw sequence letters.
3. A '+' character, optionally followed by the same sequence identifier.
4. The quality scores.

An example of data in FASTQ format is shown in Figure 7. Each ASCII character in the quality values corresponds to a value $Q_{\text{sanger}} = -10 \log_{10} p$ where p is the probability that the corresponding raw symbol is incorrect. Standard compressors, including Gzip and bzip2, treat FASTQ as a text file, resulting in poor compression ratios. FASTQ-specific compressors typically compress different fields (identifier, sequence, +, and quality scores) separately. These algorithms include DSRC, Quip, and Fqzcomp (Hosseini et al., 2016).


Identifier
Sequence
'+' sign
Quality scores
Identifier
Sequence
'+' sign
Quality scores

```
@SRR566546.970 HWUSI-EAS1673_11067_FC7070M:4:1:2299:1109 length=50
TTGCCTGCCTATCATTTTAGTGCCTGTGAGGTGGAGATGTGAGGATCAGT
+
hhhhhhhhhhghhghhhhhfhhhhhfffffe`ee[`X]b[d[ed`[Y[^Y
@SRR566546.971 HWUSI-EAS1673_11067_FC7070M:4:1:2374:1108 length=50
GATTTGTATGAAAGTATACAACTAAAACTGCAGGTGGATCAGAGTAAGTC
+
hhhhgfhhcghghggfcffdhfehhhhcehdchhdhahehffffde`bVd
```

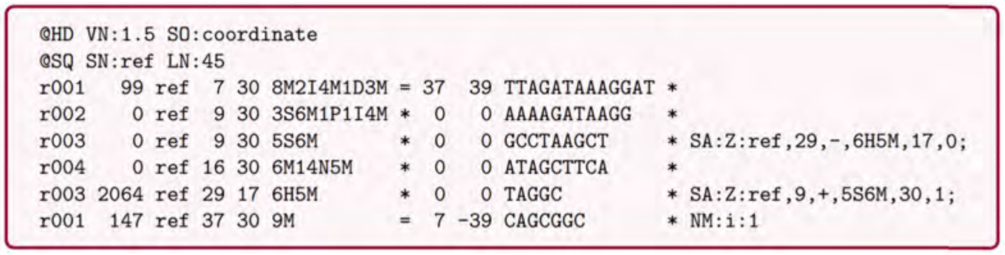
Figure 7: An example of a FASTQ file (Hosseini et al., 2016)

4.1.3 SAM

Sequence Alignment Map (SAM) represents a nucleotide sequence as differences to a known reference sequence in a text-based format. There are two main components for a DNA sequence in SAM format, as shown in Figure 8:

1. A header which starts with an @ character.
2. The alignment section, which contains a total of 11 fields, including QNAME, CIGAR, SEQ and QUAL.

A related format is BAM: a binary version of SAM, compressed using the Blocked GNU Zip Format tool (BGZF) (Hosseini et al., 2016).



```
@HD VN:1.5 SO:coordinate
@SQ SN:ref LN:45
r001   99 ref  7 30 8M2I4M1D3M = 37  39 TTAGATAAAGGAT *
r002    0 ref  9 30 3S6M1P1I4M *  0   0 AAAAGATAAGG   *
r003    0 ref  9 30 5S6M       *  0   0 GCCTAAGCT     * SA:Z:ref,29,-,6H5M,17,0;
r004    0 ref 16 30 6M14N5M    *  0   0 ATAGCTTCA     *
r003 2064 ref 29 17 6H5M       *  0   0 TAGGC         * SA:Z:ref,9,+,5S6M,30,1;
r001  147 ref 37 30 9M         =  7 -39 CAGCGGC       * NM:i:1
```

Figure 8: An example of a SAM file (Hosseini et al., 2016)

4.1.4 BLAST

Another common format is the Basic Local Alignment Search Tool (BLAST). BLAST constructs a lookup table for the query and then scans for the heuristic points for significant local alignments in the database (Chen et al., 2015).

4.2 Datasets

Our analysis is performed on a standard corpus that is commonly used for compression benchmarks. We perform the majority of our analysis on the `E.Coli` file from the Canterbury corpus: it contains the genome of the Escherichia Coli bacterium. The `E.Coli` file can be found here: `http://corpus.canterbury.ac.nz/descriptions/#large`. The corpus collection on this website contains three corpora: the Canterbury corpus, the Calgary corpus, and the Large corpus. The contents are shown in Table 5.

Table 5: The Large Corpus

File	Abbrev	Category	Size
E.coli	E.coli	Complete genome of the E. Coli bacterium	4638690
bible.txt	bible	The King James version of the bible	4047392
world192.txt	world	The CIA world fact book	2473400

Another corpus that is commonly used for compression benchmarks is the DNA corpus: `http://people.unipmn.it/manzini/dnacorpus/`. The DNA corpus contains DNA sequences that were used to test the compression algorithms in *A Simple and Fast DNA compression algorithm* by Manzini and Rastero (2004). The DNA corpus includes a variety of nucleotide sequences from very short mitochondrial DNA sequences to long and highly repetitive sex chromosomes sequences. Unfortunately, the Manzini's and Rastero's DNA corpus is limited to 4 species, all belonging to the eukarya domain, which poorly reflects the diversity of the DNA sequences. Pratas and Pinho (2018) proposed a new DNA sequence corpus with $534,263,017$ bases (509.5 MB, later updated to 685.6 MB) from 15 DNA sequences (updated to 17) varying in sizes, domains and kingdoms. This includes DNA sequences from multiple virus types (phage, virus, mimivirus), archaea, bacteria and eukaryota (protozoan, fungi, amoebozoa, plant, and animalia). The corpus is available at `https://tinyurl.com/DNAcorpus` and the details are shown in Table 6.

Table 6: The DNA Corpus

Name	Species name	Type	Size
OrSa	Oriza sativa	Eukaryota, plant	43,262,523
HoSa	Homo sapiens	Eukaryota, animalia	189,752,667
GaGa	Gallus gallus	Eukaryota, animalia	148,532,294
AnCa	Antilo capra	Eukaryota, animalia	142,189,675
DaRe	Danio rerio	Eukaryota, animalia	62,565,020
DrMe	Drosophila miranda	Eukaryota, animalia	32,181,429
EnIn	Entamoeba invadens	Eukaryota, amoebozoa	26,403,087
WaMe	Wallemia muriae	Eukaryota, fungi	9,144,43
ScPo	Schizosaccharomyces pombe	Eukaryota, fungi	10,652,155
PlFa	Plasmodium falciparum	Eukaryota, protozoan	8,986,712
EsCo	Escherichia coli	Bacteria	4,641,652
HePy	Helicobacter pylori	Bacteria	1,667,825
AeCa	Aeropyrum camini	Archaea	1,591,049
HaHi	Haloarcula hispanica	Archaea	3,890,005
YeMi	Yellowstone lake mimivirus	Virus, mimivirus	73,689
BuEb	Bundibugyo ebolavirus	Virus	18,940
AgPh	Aggregatibacter phage S1249	Virus, phage	43,970

As shown in Table 5 and Table 6, both corpora contain the `E.Coli` file which we will use as the main benchmark. For both corpora, each nucleotide sequence only contains the characters `A`, `T`, `C`, and `G`.

4.3 DNA Traits Analysis

In `FASTA` files, DNA sequences are composed from four possible main characters: `A`, `T`, `C`, and `G`. Within a nucleotide sequence, three-letter sub-strings such as `AGC` and `GCT` are called codons, and in a living cell, each codon identifies the synthesis of one (out of 20 possible) amino acids. There are $4 * 4 * 4 = 64$ possible combinations, so some codons may produce the same amino acid. In each organism, each cell contains the same DNA sequences that make up the organism's genome. These genomes are organised into chromosomes for higher life forms such as humans (Bakr et al., 2013).

DNA sequences are not random sequences. If the DNA sequences are completely randomised with a uniform distribution of nucleotide occurrences, then it would be most efficient and logical to store them with a trivial 2-bit code, as no code could do better than 2 BpC. But DNA sequences guide physical processes, such as the expression of proteins in living organisms, which means it is structured in a non-random way. For example, there are often repeated substrings within DNA sequences, for example `ACACAC`. There can also be approximate repeats, such as `AACGAACC` or `AAATAA`. In a DNA sequence, `A` and `T` are complements of each other, as are `C` and `G`. For the DNA sequence `AAACGT`, the complementary sequence would be `TTTGCA`. There is something special about this example, because when we read `TTTGCA` backwards, we would obtain back the original sequence `ACGTTT`. `ACGTTT` is then denoted as the "palindrome" or reverse complement of `AAACGT`. Some sequences (like `AATT`) are their own reverse complements (Bakr et al., 2013).

Within the same species, collections of DNA sequences are also highly redundant. Consider that only approximately 0.1% of the 3 GB human genome is specific to each human; the remaining 2.997 GB is essentially shared across all humans (Bakr et al., 2013).

4.3.1 DNA Traits in Forensic Literature

Some substrings of a longer DNA sequence are *genes* that encode specific proteins or particular genetic traits of an organism. However, some strings are just repeats of previous substrings and do not appear to have clear genetic benefits. Watson and Crick dub these regions as 'Stutter / Junk DNA' (Gen and Suhaib Ahmed, 2014).

Alex Jeffrey states that these varying number of repeats can form a pattern that is unique to each individual, and appear more frequently in certain chromosomes. These patterns allow scientists to:

1. Figure out paternal/maternal relationships through checking stutter inheritance or lack thereof.

2. Match DNA from a crime scene, whether from hair strain, white blood cell, or a single cell of cheek membrane found in saliva.

3. Research ancestry and heritage of groups of people.

4. Analyse linkage for diagnosis of genetic disorders.

Technically, these stutters are also called *tandem repeats*. Tandem repeats account for approximately 20% of the human genome. When the repeats are 2–6 bp in length, such as '`GATAGATAGATAGATAGATAGATAGATAGATAGATAGATAGATA`,' they are called *micro-satellites* or *short tandem repeats* (`STR`). When the repeat sequences are 7–80 bp in length, they are called *mini-satellites* or *variable number tandem repeats* (`VNTR`) (Gen and Suhaib Ahmed, 2014). These `STR`s tend to be consecutive, rather than spread out. Bacterial DNA and viral RNA have fewer `STR`s, and generally shorter repeats.

4.3.2 Confirmation through Exploratory Data Analysis

We perform an exploratory data analysis (EDA) on the `E.Coli` file, which contains around 4.6 million DNA nucleotides. This analysis allows us to statistically confirm the DNA-specific traits mentioned in section 4.3 and section 4.3.1. We used `Python3` for this task. We also produced an artificial sequence of identical length, composed of randomly sampled and identically distributed nucleotides. This artificial sequence provided us with a counter-factual benchmark to check whether the aforementioned characteristics of DNA occur by chance or are a measurable signal that differs from randomness. The "fake `E.Coli`" sequence also serves as an interesting benchmark for the compression comparison.

First, we examine the `N`-gram occurrence counts of the sequences. Recall that three-letter sub-strings within the DNA sequences are called codons (e.g. `AGC` and `GCT`). Because the distribution of codons is imbalanced by nature, a comparison between real and counterfeit `E.Coli` should show a skew in the distribution for certain `3`-gram strings. To substantiate this claim with evidence, we look at the `N`-gram counts for `N`$=2,3,6$ for both `E.Coli` and the artificial random sequence. We then provide the `N`-gram strings and counts for the top 3 (most common) and bottom 3 (least common) `N`-grams. Details are provided in Table 7, which shows a significant skew in distributions for certain `3`-grams and even more drastically for 6-grams. The 6-gram sequences with highest frequency in real E.Coli is cgccag with a count of 5392, while the highest frequency in the control sequence is gccccg with a count of only 1329. The table confirms our hypothesis that the E.Coli DNA sequence favors certain triplet patterns. This finding leads to the hypothesis that using contextual hyperparameters that are multiples of 3 might lead to better predictions.

Next, we look at short tandem repeats (STRs), where each repeat is 2–6 bp in length. Note that a sequence with more than 6 total repeats (such as GATAGATAGATAGATAGATAGATAGATA) rarely occurs in bacteria and prokaryotic microorganisms. Only human and animal DNA sequences contain STRs that extend over 30 repeats. There are 16 core locations for STRs on the human genome that are used by forensic scientists to perform analyses. Even 13 exact matches of STRs provide more than 99.99% confidence that two samples come from the same person (Gen and Suhaib Ahmed, 2014). For each STRs of length N, we observe the count, the longest number of repeats, and the average number of repeats for E.Coli and the control sequence in Table 8. We can see that there is a significant skew in the number of STRs for lengths 3, 6, and 9. We also find that the E.Coli sequence has an unusually high number of repeats of the length 8.

We also look at single-nucleotide repetitions, such as the length-4 repetition AAAA. We want to see if natural DNA has a significantly higher number of single symbol repetitions. In Table 9, we show that E.Coli does have more single nucleotide repeats than the control sequence, especially for the order of 6, 7, and 8, where the counts are about twice of those of the control sequence.

Finally, we look for the complementary palindrome characteristic in the sequences. AATT and TCGA are examples of complementary palindromes. Taking a sample sequence of the length of $30,000$ from the same region of both files, we count the number of complementary palindromes in each. E.Coli has $10,278$ palindromes and the control sequence has $9,942$. In the same interval, the number of complementary palindrome in E.Coli file is $49,995$ and in the control file is $49,803$. This illustrates that real DNA file has the characteristics of both complementary and generic palindrome, although not to a statistically significant extent.

Table 7: Top vs Bottom three symbols count for each N-gram in `E.Coli` and an artificial control sequence

N-Gram	File	Top Three Sequences						Bottom Three Sequences					
		Top 1	Count	Top 2	Count	Top 3	Count	Bottom 3	Count	Bottom 2	Count	Bottom 1	Count
2 gram	E.Coli	gc	383734	cg	346554	tt	339460	ag	237826	ct	235986	ta	211939
2 gram	Control	cc	299846	gg	299710	cg	298861	aa	280804	at	280228	ta	280190
3 gram	E.Coli	cgc	115631	gcg	114559	ttt	109824	gag	42466	tag	27233	cta	26753
3 gram	Control	ccg	76319	ggg	76252	gcc	76216	ata	68981	tta	68877	tat	68640
6 gram	E.Coli	cgccag	5392	ctggcg	5254	gccagc	4823	tctagg	19	ctagga	17	cctagg	16
6 gram	Control	gccccg	1335	cggggg	1329	cccgcg	1328	tcaaaa	976	ttatat	962	aaaata	956

Table 8: Length-N STR-repeats comparison of `E.Coli` vs. the artificial control sequence

STR Length	Count		Longest Repeats		Average Repeats	
	E.Coli	Control	E.Coli	Control	E.Coli	Control
2	201463	217875	6	6	2.068	2.065
3	77573	54358	5	4	2.032	2.017
4	13781	13661	4	3	2.003	2.004
5	3584	3341	2	3	2.000	2.001
6	2034	834	3	2	2.001	2.000
7	238	222	2	2	2.000	2.000
8	68	49	6	2	2.132	2.000
9	44	17	2	2	2.000	2.000

Table 9: Count of Length-N single symbol repeats comparison in `E.Coli` and the artificial control sequence

Length	Count	
	E.Coli	Control
2	679312	653006
3	163460	163273
4	42951	40983
5	13882	10301
6	4121	2476
7	1002	631
8	215	151

5 Method

5.1 Prediction by Partial Matching (PPM)

PPM is often used as the *benchmark* for adaptive predictors. The PPM algorithm utilises a Markov model, modeling the occurrence probability of a given symbol conditional on a *context* of the symbols immediately preceding it. The PPM model is parameterised by its *order*, which is the number of symbols in the largest context used for the next symbol prediction. PPM is an adaptive model that maintains an up-to-date set of context-dependent symbol counts (often in a tree-like data structure). These counts are used to compute the probability for the next symbol in the sequence.

A basic PPM variant might allocate an initial count of 1 to the possibility of occurrence for each symbol in a context in which the symbol has not occurred in. Denote $c_i(\varphi)$ as the number of times the symbol φ occurs in the context i for each symbol in the alphabet (256 types in ASCII). We use C_i as the number of times that we see context i, i.e. $\Sigma_\varphi c_i(\varphi)$. PPM models the occurrence probability of symbol p in context i as follows:

$$p_i(\varphi) = \frac{c_i(\varphi)}{1+C}, \qquad \text{when } c_i(\varphi) > 0$$

When a character occurrence is novel in a given context or $c_i(\varphi) = 0$, PPM computes an *escape probability* as the remaining probability when accounting for all seen characters:

$$p_i(\text{ESC}) = 1 - \sum_{\varphi \in A, c(\varphi)>0} p_i(\varphi) = \frac{1}{1+C}$$

PPM then "escapes" to the next shorter context, and uses that context's probability distribution: $p_i(\varphi) = p_i(\text{ESC}) * p_{\text{shorter}(i)}(\varphi)$. Given a to be the dictionary size of the coding symbols A (256 for ASCII) and q to be the number of symbols occurred so far in context i, a total of $a - q$ symbols haven't appeared in context i. Each novel symbol is then allocated the overall coding probability

$$p(\varphi) = \frac{1}{1+C} \cdot \frac{1}{a-q}, \quad c(\varphi) = 0$$

Another technique classes a symbol in a given context i as novel unless the symbol has occurred twice in the given context. This technique hypothesises that the first occurrence could be a one-off event due to an error or other anomalies. If a symbol exists twice in the same context, then it is much more likely to exist again in the same context. As such, a probability of an already twice-occurred symbol is modelled as:

$$p_i^*(\varphi) = \frac{c_i^*(\varphi) - 1}{C^*}, \quad \text{when } c_i^*(\varphi) > 1$$

Thus, the escape probability is

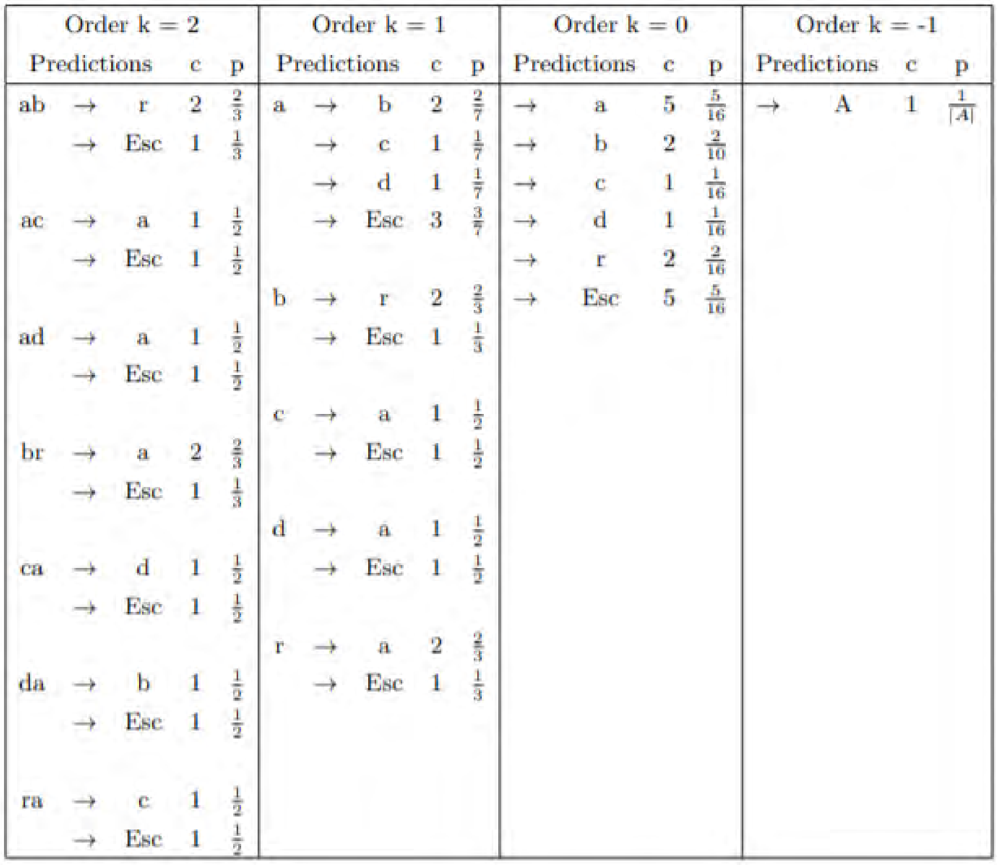
$$1 - \sum_{\varphi \in A, c_i^*(\varphi) > 1} p_i^*(\varphi) = \frac{q^*}{C^*}$$

We then allocate to every novel character the overall coding probability

$$p^*(\varphi) = \frac{q^*}{C^*} \cdot \frac{1}{a^* - q^*}, \quad c^*(\varphi) \leqslant 1$$

(Cleary and Witten, 1984).

To give an example, we consider the input sequence `abracadabra` containing only lower case letters. Given each symbol in the sequence, `PPM` will represent the probability of each symbol occurring by constructing a probability distribution. There is no prior information for the first symbol in the sequence, so a general strategy is to assign a uniform distribution. Given that the first symbol is `a`, we will assign a slightly higher probability for `a` when predicting the second character given we already observed it once in history. After we parse the entire `abracadabra` sequence, we can predict the likelihood for the next symbol by searching the input history for the longest matching sequence to the recent input. For `abracadabra`, the closest match to the last four letters `abra` starting at eighth position is the abra string occurring at the first position. Given the longest match, we can predict that the following character would likely be the character following the matched sequence. For abra, the first sequence precedes the symbol c at the fifth position. Thus, it is reasonable to predict the next symbol in the sequence abracadabra to be a c. This is an example of an order-4 matching. The longer the order, the better the predictions, given a sufficiently long input sequence (Knoll and de Freitas, 2012).



Order k = 2			Order k = 1			Order k = 0			Order k = -1		
Predictions	c	p	Predictions	c	p	Predictions	c	p	Predictions	c	p
ab → r	2	$\frac{2}{3}$	a → b	2	$\frac{2}{7}$	→ a	5	$\frac{5}{16}$	→ A	1	$\frac{1}{\|A\|}$
→ Esc	1	$\frac{1}{3}$	→ c	1	$\frac{1}{7}$	→ b	2	$\frac{2}{10}$			
			→ d	1	$\frac{1}{7}$	→ c	1	$\frac{1}{16}$			
ac → a	1	$\frac{1}{2}$	→ Esc	3	$\frac{3}{7}$	→ d	1	$\frac{1}{16}$			
→ Esc	1	$\frac{1}{2}$				→ r	2	$\frac{2}{16}$			
			b → r	2	$\frac{2}{3}$	→ Esc	5	$\frac{5}{16}$			
ad → a	1	$\frac{1}{2}$	→ Esc	1	$\frac{1}{3}$						
→ Esc	1	$\frac{1}{2}$									
			c → a	1	$\frac{1}{2}$						
br → a	2	$\frac{2}{3}$	→ Esc	1	$\frac{1}{2}$						
→ Esc	1	$\frac{1}{3}$									
			d → a	1	$\frac{1}{2}$						
ca → d	1	$\frac{1}{2}$	→ Esc	1	$\frac{1}{2}$						
→ Esc	1	$\frac{1}{2}$									
			r → a	2	$\frac{2}{3}$						
da → b	1	$\frac{1}{2}$	→ Esc	1	$\frac{1}{3}$						
→ Esc	1	$\frac{1}{2}$									
ra → c	1	$\frac{1}{2}$									
→ Esc	1	$\frac{1}{2}$									

Figure 9: PPM model after processing the string "abracadabra" using up to the second order model (Knoll and de Freitas, 2012).

5.2 Traditional Machine Learning Approaches

5.2.1 Naïve Bayes Classifier

Bayesian classifiers base the likelihood for each class on its feature vector. The Naïve Bayes classifier simplifies the learning process by assuming independence between classes, i.e. $P(\mathbf{X} \mid C) = \prod_{i=1}^{n} P\left(X_i \mid C\right)$ where $\mathbf{X} = (X_1, \cdots, X_n)$ is a feature vector and C is a class. While this independence assumption may be unrealistic in practice, Naïve Bayes classifiers are popular and give a baseline accuracy that sets a benchmark for more complex classifiers. Naïve Bayes classifiers are also highly versatile, with applications in various fields ranging from bioinformatics to natural language processing.

In mathematical terms, we denote a Bayes or Bayes-optimal classifier as $h^*(\mathbf{x})$. Given a feature vector, the Bayes classifier uses the class posterior probabilities as discriminant functions, i.e. $f_i^*(\mathbf{x}) = P(C = i \mid \mathbf{X} = \mathbf{x})$. By applying Bayes rule, we have

$$P(C = i \mid \mathbf{X} = \mathbf{x}) = \frac{P(\mathbf{X} = \mathbf{x} \mid C = i)P(C = i)}{P(\mathbf{X} = \mathbf{x})}$$

where we can ignore $P(\mathbf{X} = \mathbf{x})$ as it is identical for every class, providing us with Bayes discriminant functions

$$f_i^*(\mathbf{x}) = P(\mathbf{X} = \mathbf{x} \mid C = i)P(C = i)$$

where we denote $P(\mathbf{X} = \mathbf{x} \mid C = i)$ as the class-conditional probability distribution (CPD). Hence, the Bayes classifier, given example x, finds the maximum a posteriori probability (MAP) hypothesis.

$$h^*(\mathbf{x}) = \arg\max_i P(\mathbf{X} = \mathbf{x} \mid C = i)P(C = i)$$

Nevertheless, when the feature space is high-dimensional, the direct estimation of $P(\mathbf{X} = \mathbf{x} \mid C = i)$ from the examples can be often difficult. Thus, Naïve Bayes NB($\mathbf{x}$) is a commonly used simplification. Naïve Bayes simplifies the calculation by assuming that, given the class, features are independent. Naïve Bayes is defined by discriminant functions (Rish et al., 2001):

$$f_i^{NB}(\mathbf{x}) = \Pi_{j=1}^{n} P\left(X_j = x_j \mid C = i\right) P(C = i)$$

5.2.2 Decision Tree Classifier

A decision tree classifier is a member of the table look-up rules classifier family which utilises multi-stage decision making approaches. The core idea is to convert a decision table to optimal decision trees through sequential approaches, breaking up several complex decisions into a cascade of simpler decisions. As such, hierarchical approaches are the key emphasis. These approaches permit the rejection of class labels at intermediate stages while constructing multi-stage classifiers.

Safavian and Landgrebe (1991) present a non-parametric method for feature space hierarchical partitioning. The method involves the concept of average mutual information. We denote the average mutual information from observed event X_k at a node k in a tree T about class C_k as

$$I_k\left(C_k; X_k\right) = \sum_{C_k X_k} p\left(C_{ki}, X_{kj}\right) \cdot \log_2 \int \frac{p\left(C_{ki} \mid X_{kj}\right)}{p\left(C_{ki}\right)}$$

Each X_k measures node k's selected feature with either of the two outcomes: higher or lower than the associated threshold of the node k selected feature.

We then express the average mutual information between the partitioning tree T and the class C as

$$I(C; T) = \sum_{k=1}^{L} p_{k'} I_k\left(C_{k'} X_k\right)$$

where L is the number of tree T internal nodes and p_k is class C_k probability. For a decision tree classifier T, the average mutual information $I(C; T)$ directly relates to the probability of misclassification p_e, as follows:

$$I(C;T) \geq -\sum_{j=1}^{m} \left[p\left(C_j\right) \cdot \log_2 p\left(C_j\right)\right] + p_e \log_2 p_e + \left(1-p_e\right)$$
$$\log_2\left(1-p_e\right) + p_{e^*} \log_2(m-1)$$

This equality represents the minimum required $I(C;T)$ for the desired probability of misclassification p_e. As such, at each node k, a decision tree classifier T has to maximise average mutual information gain (AMIG). When $I(C;T)$ increases above the minimum required $I(C;T)$ for the specified probability of error, the optimisation algorithms then reaches termination (Safavian and Landgrebe, 1991).

5.2.3 Random Forest Classifier

A random forest classifier is an ensemble classifier, similar to context mixing methods. The building blocks for random forest classifiers are decision tree classifiers constructed from independently sampled random vectors. As such, the classifier utilises a combination of randomly selected features at each node to construct a decision tree. Each tree then has one vote to cast for the input vector classification. Data sampling is done through a bagging method, which bootstraps training data by randomly sampling with replacement N samples (the size of the data). The success of random forest relies on hyper-parameter selections and the method used for pruning.

Quinlan's Information Gain Ratio criterion and Breiman's Gini Index are popular attribute selection measure for decision tree induction. The Gini Index evaluates an attribute's impurity with respect to the classes and is commonly used to select attributes.

In mathematical terms, the Gini index for assigning a training set T to a class C_i is expressed as

$$\sum_{j \neq i} \left(f\left(C_i, \mathrm{T}\right) / \mid \mathrm{T} \mid\right)\left(f\left(C_j, \mathrm{T}\right) / \mid \mathrm{T} \mid\right)$$

where $f\left(C_i, \mathrm{T}\right) / \mid \mathrm{T} \mid$ is the probability that the selected case is from the class C_i.

Through this method, we use a combination of features to grow a tree up to specified maximum depth without being pruned. This lack of pruning is a stark contrast between random forest classifier and decision tree classifier, whose effectiveness depends heavily on the choice of pruning methods. The random forest effectiveness, however, depends highly on the hyperparameters, though Breiman illustrates that the generalisation error will converge as the number of decision trees increase. This eliminates the need for pruning and also eliminates the overfitting problem due to the Strong Law of Large Numbers. As such, a random forest classifier's success depends heavily on the pre-specified number of decision trees (N) and the number of features (K). The classifier then searches for best split only from selected features and consists only of N trees. For new data classification, the input data pass through each of the N trees, and each tree casts a vote into a ballot for the most likely class (Pal, 2005).

5.3 Neural Network Approach

5.3.1 Recurrent Neural Network

General classification models like Markov-based models operate under the same principle as the N-gram approach, i.e. using a length N context for probabilistic modelling. Instead of taking into account only preceding texts, the recurrent neural network (RNN) generalises the context-based language model further. RNN learns short term memory representation from the data through neurons with recurrent connections. Shallow feed-forward neural networks with 1-hidden layer have been effective at clustering texts. By leveraging a deep architecture, an RNN can cluster similar histories and context together (Mikolov et al., 2011).

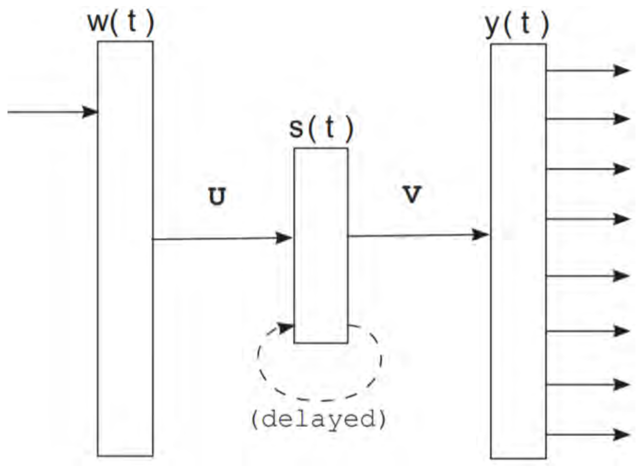


Figure 10: Simple recurrent neural network (Mikolov et al., 2011)

We illustrate the RNN architecture in Figure 10.We concatenate the vector $\mathbf{w}(t)$ and $s(t-1)$ to form $\mathbf{x}(t)$. $\mathbf{w}(t)$ represents the current word with size equivalent to vocabulary size, using 1 of N coding. $s(t-1)$ is the values from the previous time step in the hidden layer. An RNN's deep architecture is structured into input, hidden, and output layers and utilises backpropagation for training. These three layers are computed as follows:

$$\mathbf{x}(t) = \left[\mathbf{w}(t)^T \mathbf{s}(t-1)^T\right]^T$$
$$s_j(t) = f\left(\textstyle\sum_i x_i(t) u_{ji}\right)$$
$$y_k(t) = g\left(\textstyle\sum_j s_j(t) v_{kj}\right)$$

where $f(z)$ is the sigmoid activation function and $g(z)$ is the softmax activation function. The softmax function ensures that the output layer generates a probability distribution output that sums to 1 and each probability is greater than 0.

$$f(z) = \frac{1}{1+e^{-z}}, \quad g(z_m) = \frac{e^{z_m}}{\sum_k e^{z_k}}$$

In the output layer, we collect an error vector from the cross-entropy criterion. Afterwards, the error vector is backpropagated to the hidden layer. For RNNs, we use backpropagation through time (BPTT) as an extension of the backpropagation algorithm. In truncated BPTT, the method propagates the error back in time through recurrent networks for τ numbers of steps. This allows information from multiple steps of hidden layers to be incorporated into the network through BPTT (Mikolov et al., 2011).

5.3.2 Long-short Term Memory (LSTM) and stateful LSTM

LSTMs provide a natural extension of RNNs. During the process of backpropagation, the error function's gradient is propagated through steps of hidden layers of neural networks in time. In practically relevant cases, this gradient is multiplied by a scaling factor larger or smaller than one, which results in either exploding gradient or vanishing gradient due to the exponential compounding effect. The gradient from the previous step then overshadows the current step gradient or gets completely overshadowed. In natural language processing context, the position in the sentence is the time step, so either only the last word matters or last word is completely irrelevant (Hochreiter and Schmidhuber, 1997).

This decaying error backflow causes an extensive problem to the computation time for recurrent backpropagation when storing information over time intervals with extensive length. To solve these gradient-related problems, Hochreiter and Schmidhuber (1997) introduce "Long Short-Term Memory" (LSTM), an efficient and novel gradient-based method for truncating gradient without hurting the learning process.

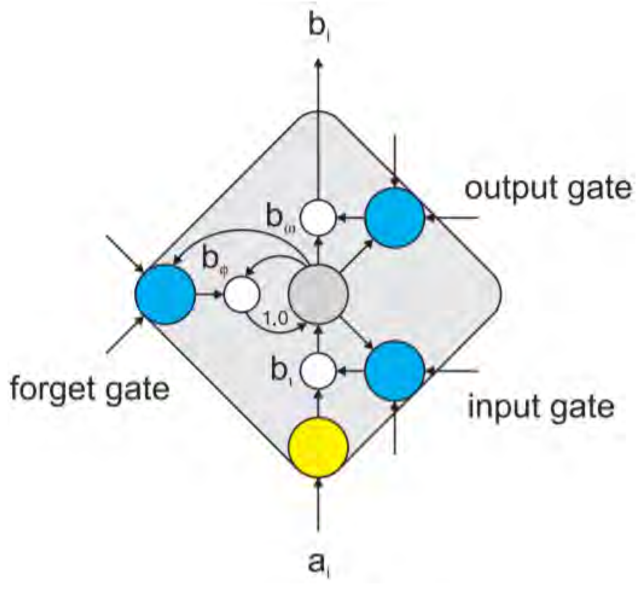


Figure 11: LSTM memory cell with gating units (Sundermeyer et al., 2012)

For standard neural network i, the unit contains exclusively the input activation a_i and the output activation b_i. These input-output activations correlate when a tanh activation function is utilised through

$$b_i = \tanh(a_i) \odot$$

Improving upon the RNN architecture, the LSTM adds multiple intermediate steps. The model multiplies the output of the activation function on a_i by a factor b_i. Because of the recurrent self-connection, the result is then added by the multiplication between the previous time step's inner activation value and $b_\diamond$. The final result is then fed to another activation function after rescaled by b_ω, which returns b_i. Denoted by the small white circles in Figure 11, the factors $b_i, b_\phi, b_\omega \in (0, 1)$ are controlled by the blue circle units depicting input, output, and forget gate in the Figure 11. The previous hidden layer's activations, the current layer's activations from preceding time steps, and the LSTM unit's inner activation are then summed by the gating units. A logistic sigmoid function squashes the gating units' output, which afterwards are set respectively to b_2, b_ϕ, or b_ω, (Sundermeyer et al., 2012).

For this paper, we utilise the LSTM architecture of Bellard (2019). The LSTM architecture contains L layers of LSTM cells with L ranging from 1 up to 7. The output of the corresponding cells from previous layers along with the previous symbol is taken as the input for each cell. The LSTM cell, for each layer $l = 1 \dots L$, is defined as follows:

$$\begin{aligned}
f_{t,l} &= \text{sigm}\left(\text{LayerNorm}\left(W_l^f\left[h_{t-1,l}; h_{t,0}; \dots; h_{t,l-1}\right]\right)\right) \\
i_{t,l} &= \text{sigm}\left(\text{LayerNorm}\left(W_l^i\left[h_{t-1,l}; h_{t,0}; \dots; h_{t,l-1}\right]\right)\right) \\
o_{t,l} &= \text{sigm}\left(\text{LayerNorm}\left(W_l^o\left[h_{t-1,l}; h_{t,0}; \dots; h_{t,l-1}\right]\right)\right) \\
j_{t,l} &= \tanh\left(\text{LayerNorm}\left(W_l^j\left[h_{t-1,l}; h_{t,0}; \dots; h_{t,l-1}\right]\right)\right) \\
c_{t,l} &= f_{t,l} \odot c_{t-1,l} + \min\left(1 - f_{t,l}, i_{t,l}\right) \odot j_{t,l} \\
h_{t,l} &= o_{t,l} \odot c_{t,l}
\end{aligned}$$

where $\odot$ is the element-wise multiplication. The model sets the input of the first layer, such as:

$$h_{t,0} = \text{One}\left(s_{t-1}\right)$$

The model computes probabilities p_t for the symbol at time t as:

$$p_t = \text{softmax}\left(W^e\left[h_{t,1}; \dots; h_{t,L}\right] + b^e\right)$$

$W_i^f, W_i^i, W_i^o, W_i^j, W^e, b^e$ are learned parameters. $c_{t,l}$ and $h_{t,l}$ are vectors of size N_c. We set $h_{-1,l}$ and $c_{-1,1}$ to the zero vector. the concatenation of vectors a_0 to a_n is represented by $[a_0; \dots; a_n]$. $y = \text{One}(k)$ is a N_s-size vector such as $y_i = 0$ for $i \neq k$ and $y_k = 1$

For the above model, the y =LayerNorm(x) is defined as followed:

$$\begin{aligned}
y_i &= \frac{(x_i - \mu)}{\sigma + \epsilon} g_i + b_i \text{ with } i = 0 \ldots N-1 \\
\mu &= \frac{1}{N} \sum_{i=0}^{N-1} x_i \\
\sigma &= \sqrt{\frac{1}{N} \sum_{i=0}^{N-1} (x_i - \mu)^2} \\
\epsilon &= 10^{-5}
\end{aligned}$$

where g and b are per-instance learned parameters. This `LayerNorm` layer normalisation operation provides a significant increase in performance compared to non-normalised `LSTM`.

The output variant from $i_{t,l}$ can be used directly by the cell equation $c_{t,l}$ to ensure that the cell-state is bounded. Since the model only trains with up to 2 epochs per training data (each data seen at most twice), the chance of overfitting is low, and no dropout layer is necessary. For the training method, the model uses regular truncated backpropagation on prespecified numbers of consecutive time steps. For a stateful `LSTM` model, we set $c_{t,l}$ and $h_{t,l}$ (the initial states of the training segment) to the previous segment's last values. We specify the training batch size B. The smaller the batch, the better the performance. However, larger batches make matrix operations more efficient, improving the computation speed but compromising accuracy. No gradient clipping is used, and the optimiser utilised is `Adam` with $\beta_1 = 0, \beta_2 = 0.9999$ and $^2\epsilon = 10^{-5}$ (Bellard, 2019).

5.4 Attention Heads and Transformer

Instead of relying on a recurrent neural network architecture for global input-output dependence, the transformer model opts for an attention-based mechanism. The attention mechanisms provide significantly higher room for optimisation through parallelisation. These mechanisms also disregard the distance in the input/output sequences, which leads to better dependency modelling in transduction and sequence modelling for various tasks (Bellard, 2019).

For our dissertation, we utilise Bellard (2019) transformer model. the model contains a tunable number of L transformer cell layers. Below, we illustrate the overall structure for each layer $l = 1 \dots L$, attention head $i = 1 \dots N_h$, and backward time-step $j = 0 \dots M-1$

$$\begin{aligned}
q_{t,l,i} &= W_{l,i}^{q} h_{t,l-1} \\
k_{t,l,i,j} &= W_{l,i}^{k} h_{t-j,l-1} \\
v_{t,l,i,j} &= W_{l,i}^{v} h_{t-j,l-1} \\
a_{t,l,i,j} &= k_{t,l,i,j}^{\top} \left(q_{t,l,i} + u_i\right) + w_{i,\min(j,d_{\max})}^{\top} \left(q_{t,l,i} + v_i\right) \\
r_{t,l,i} &= \operatorname{softmax}\left(\frac{a_{t,l,i}}{\sqrt{d_{key}}}\right) \\
u_{t,l,i} &= \sum_{k=0}^{M-1} r_{t,l,i,k} \cdot v_{t,l,i,k} \\
o_{t,l} &= \text{ LayerNorm }\left(W_l^{p} \cdot [u_{t,l,1}; \dots ; u_{t,l,N_h}] + h_{t,l-1}\right) \\
e_{t,l} &= W_l^{g} \cdot \operatorname{ReLU}\left(W_l^{f} o_{t,l} + b_l^{f}\right) \\
h_{t,l} &= \text{ LayerNorm }\left(e_{t,l} + o_{t,l}\right)
\end{aligned}$$

The model sets the first layer input:

$$h_{t,0} = W^{ei} \cdot \operatorname{One}\left(s_{t-1}\right)$$

and compute probabilities p_t for the symbol at time t:

$$p_t = \operatorname{softmax}\left(W^{eo} h_{t,L} + b^{eo}\right)$$

$W_{l,i}^{q}, W_{l,i}^{k}, W_{l,i}^{v}, u_i, w_{i,j}, v_i, W_l^{p}, W_l^{f},\ b_l^{f}, W_l^{g}, W^{ei}, W^{eo}, b^{eo}$ are learned parameters. $h_{t,l}$ and $o_{t,l}$ have a dimension of d_{model} . $q_{t,l,i}$ and $k_{t,l,i}$ have a dimension of $d_{key} = \frac{d_{\text{model}}}{N_h} \cdot v_{t,l,i}$ has a dimension of $d_{\text{value}} = d_{\text{key}} \cdot b_l^{f}$ has a dimension of d_{inner} (Bellard, 2019).

5.5 Bi-directional LSTM model with Convolutional Neural Network Layer and Attention Head

Given our model in section 5.3.2 and section 5.4, however, DNA compression is an online-learning task. The transformer is an extremely powerful model but requires a large amount of data to reach full potential. On the other hand, LSTM is one-directional and cannot capture the full palindromic nature of DNA analysed in section 4.3.2.

To fully utilise section 4.3.2, we create a bi-directional LSTM with multiple hyperparameters tuning to suit the DNA characteristics. The model is the same as section 5.3.2, but instead of feeding only $h_{t,l}$ to the LSTM cell, in each layer, we feed both forward $[h_{t,0}; ...; h_{t,l-1}]$ and backward $[h_{t,l-1}; ...; h_{t,0}]$ into separate LSTM and combine the result through concatenation. Like section 5.3.2, we perform layer normalization between layers. An example of 2 layers bi-directional LSTM model implementation in Tensorflow2.0 is shown in Figure 12.

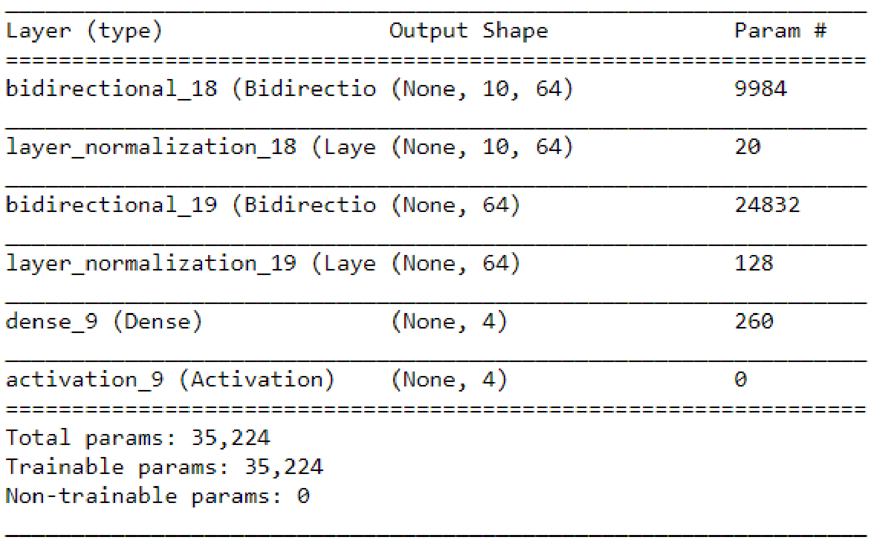

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
bidirectional_18 (Bidirectio (None, 10, 64)            9984      
_________________________________________________________________
layer_normalization_18 (Laye (None, 10, 64)            20        
_________________________________________________________________
bidirectional_19 (Bidirectio (None, 64)                24832     
_________________________________________________________________
layer_normalization_19 (Laye (None, 64)                128       
_________________________________________________________________
dense_9 (Dense)              (None, 4)                 260       
_________________________________________________________________
activation_9 (Activation)    (None, 4)                 0         
=================================================================
Total params: 35,224
Trainable params: 35,224
Non-trainable params: 0
_________________________________________________________________
```

Figure 12: Bi-directional LSTM model in Keras/Tensorflow

We can combine this architecture with the attention head described in section 5.4. We then experiment with up to 4 layers with normalisation between layers and with both stateful and stateless models using various hyperparameters adjusted for speed or closeness to DNA characteristics.

Standard LSTM also has another issue where temporal modelling is performed on the input feature x_t. It is possible to better understand temporal structure between successive time steps by using higher-level modelling of x_t to deconstruct underlying factors of input variation. Sainath et al. (2015) show that CNN can provide a higher-level understanding of discriminatively trained features while removing variation within the input. Sainath et al. propose an architecture where a few fully connected CNNs precede layers of LSTM, resulting in the CLDNN architect, as shown in Figure 13.

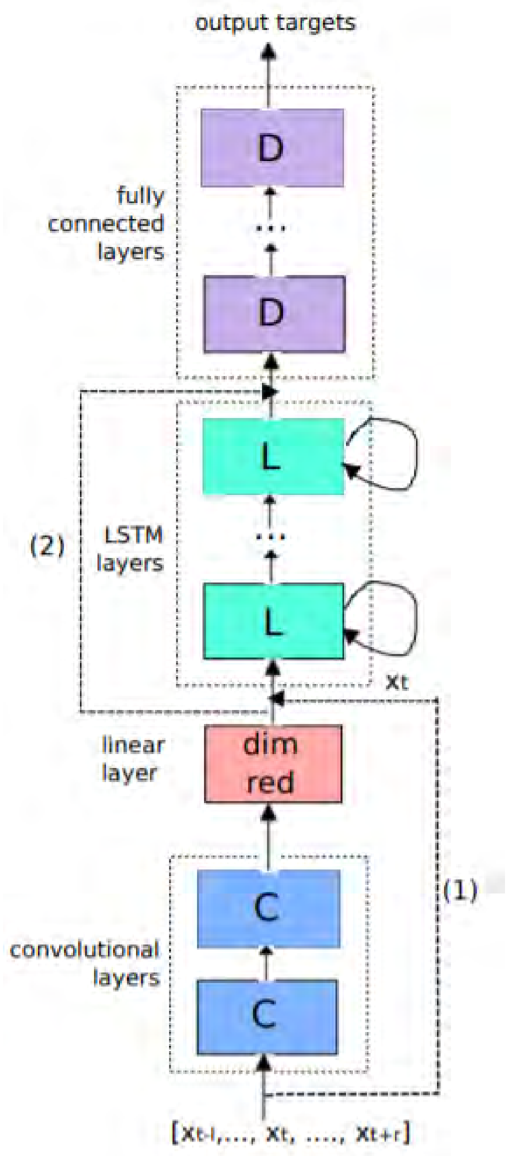


Figure 13: CLDNN Architecture (Sainath et al., 2015)

Because DNA characteristics have 4 symbols A, T, C, G, both 1-D and 2-D CNN seem to be an appropriate choice for preprocessing the sequence for the LSTM architecture.

5.6 Context Mixing Model: PAQ Family

As an evolution of well-known PPM in section 5.1, PAQ is an umbrella term for a family of context mixing algorithm. Compression-ratio wise, PPM data compression methods have been state-of-the-art benchmark up to the 1990s. PAQ family has since dominated compression benchmarks. While in general, compression algorithms are required to balance a trade-off between computation speed, compression ratio, and memory usage. PAQ8, especially, has achieved record-breaking compression rations without trading significant time and memory usage (Knoll and de Freitas, 2012).

PAQ has achieved incredible performance and huge success in the compression community. However, there is an obvious lack of comparison against machine learning methods in both scientific papers and publications. The reason is that the inner-workings of PAQ are rarely explained. Knoll and Freitas Mahoney (2005) claims that aside from their paper, only incomplete high-level descriptions of PAQ1–PAQ6 exist, same with the famous PAQ8. This is due to the available C++ source code, which is optimised to be as close to machine language as possible. Therefore, it is difficult to extract actual algorithms and architecture details of PAQ8 (Mahoney, 2005). Our dissertation aims to provide clarification to both machine learning-based algorithms in section 5 and to PAQ family algorithms for efficient comparison.

5.6.1 PAQ1-6: Adaptive Model Mixing

Matt Mahoney first develops PAQ1 in January 2002, which uses the following contexts:

1. Similar to PPM, the general-purpose model consist of eight contexts of length 0 to 7 bytes. Each context includes 0 to 7 bits of the current byte preceding the predicted bit.

2. PAQ1 uses unigram and bigram models, i.e. two word-oriented contexts of length 0 or 1. These include whole words preceding the predicted word and are case sensitive, with alphabets from `a` to `z`.

3. To model 2-D data like databases or images, two fixed-length record models are utilised.

These two contexts are the column number as well as the above byte. Detecting a consecutive series of a uniform stride of 4 identical byte values determines the record length.

4. One match context to locate the last 8-bytes or longer matched context for predicting the bit following the match.

These models use semi-stationary update, except for match. PAQ represents state (n_0, n_1) in 8-bit value to estimate large counts. The models are mixed, hench the name context mixing model, and the weights are empirically tune where the eight length n general purpose contexts have weights $w = (n+1)^2$ (Mahoney, 2005).

PAQ4 is introduced in October 2003 through the inclusion of an adaptive weighting model for 18 contexts. The 3-bit context containing the previous whole byte's 3 most significant bits select 8 weight sets to be used by the mixer (Mahoney, 2005).

PAQ5 is introduced in December 2003 through the inclusion of a second mixer. The second mixer selects weights by last two bytes' two most significant bits which forms a 4-bit context. For analog data (8 and 16 bit mono and stereo audio, 24 -bit color images and 8 -bit data), 6 new models are added where noisy low order bits contexts are discarded (Mahoney, 2005).

PAQ6 is introduced in December 2003 through the inclusion of non-stationary/run-length models. This provides an update to the semi-stationary models of original PAQ. PAQ6 adds a model to translates relative CALL operands for Intel executable code. It also adds 10 general-purpose contexts with 4 addition long context match models, 7 analog models with including FAX image model, 9 sparse models, 5 record models, and 6 word models which includes sparse bigrams (Mahoney, 2005).

5.6.2 PAQ8-ZPAQ: Architecture and Model

From PAQ8 onward, the algorithms utilise neural networks for weight mixing. PAQ8 is still the model with best compression ration in PAQ family using a weighted combination of multiple models' prediction. Most important addition is the incorporation of non-contiguous context matched, which improves noise robustness compared to traditional PPM model. Much like RNN and LSTM, PAQ8 can capture long-term dependencies since all of its models predict the next bit on the bit-level context. This allows PAQ8 to generalise well compared to PPM, which makes byte-level predictions. PAQ8 also performs reasonably against specific data types like spreadsheets or images (Knoll and de Freitas, 2012).

However, PAQ8 architecture also varies based on the version. This also extends to algorithms which may change to suit the data compressed. Image data, for instance, requires fewer prediction models. The most famous version of PAQ8 is PAQ8l which is a stable released version in March 2007 by Matt Mahoney. PAQ8l provides the best compression ratio and robust architecture. The version submitted to Hutter prize also includes dictionary preprocessing and word-level modelling which is not present in PAQ8l. Figure 14 provides a high-level overview of the PAQ8l architecture. In total, PAQ8l utilises 552 prediction models, combined through the model mixer into a single prediction, An adaptive probability map (APM), which typically reduce prediction error by 1%, then preprocesses the prediction before feeding it the arithmetic coder. APM is also known by another name as secondary symbol estimation (Knoll and de Freitas, 2012).

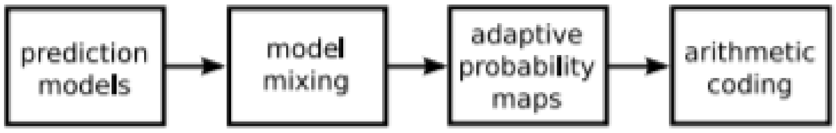


Figure 14: PAQ8 architecture (Knoll and de Freitas, 2012)

We provide an illustration of the PAQ8l model mixer architecture in Figure 15. The model highly resembles a single hidden layer neural network model. The subtle difference separating the model from a standard neural network is that the first and second layers' weights are trained independently for each node and online as the algorithm runs. While backpropagation and truncated backpropagation trains the nodes in the multi-layer network together, separate training minimises the predictive cross-entropy error. Unless the data is stationary, however, the parameters will not converge to fixed values. This makes PAQ8l a half-ensemble, half neural network model, which is designed to handle both stationary and non-stationary data.

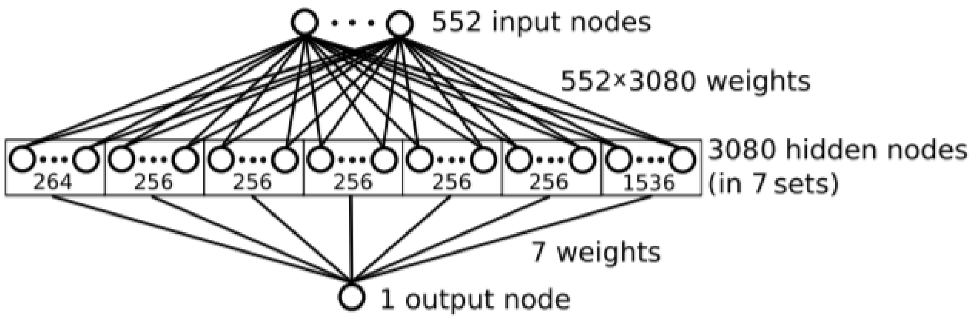


Figure 15: PAQ8 model mixer architecture (Knoll and de Freitas, 2012)

Another significant difference between neural network and PAQ81 lies in the hidden nodes which are partitioned into seven different sets. Each set size is shown in the rectangles in Figure 15, where the leftmost rectangle is denoted as set 1 and the rightmost as set 7. From each set, a single node is selected for each bit in the data file. For each bit in the data, the only updated edges are the 7 weights connect each group to the output node in Figure 15. This implies that, for each bit, the model only updates $552 \times 7 = 3,864$ compared to the $552 \times 3,080 = 1,700,160$ weights in the first layer. The reduction in weight update greatly improves the computation speed by multiple magnitudes compared to neural network models, which can be seen in section 6.3.

For both hidden and output layers, each of the PAQ81 model mixer's node is a Bernoulli logistic model:

$$p\left(y_t \mid \mathbf{x}_t, \mathbf{w}\right) = \text{Ber}\left(y_t \mid \text{sigm}\left(\mathbf{w}^T \mathbf{x}_t\right)\right)$$

where $\mathbf{x}_t \in [0,1]^{n_p}$ is the predictor vector at time t, $\mathbf{w} \in \mathbb{R}^{n_p}$ is the vector of weights, $y_t \in \{0,1\}$ is the next bit in the data to be compressed, and $\text{sigm}(\eta) = 1/\left(1+e^{-\eta}\right)$ is the sigmoid or logistic function. The number of predictors, n_p, is equal to 552 for the model's first layer and 7 for the model's second layer.

Let $\pi_t = \text{sigm}\left(\mathbf{w}^T\mathbf{x}_t\right)$. The $t-$th bit negative log-likelihood is given by

$$NLL(\mathbf{w}) = -\log\left[\pi_t^{\mathbb{I}(y_t=1)} \times (1-\pi_t)^{\mathbb{I}(y_t=0)}\right] = -\left[y_t \log \pi_t + (1-y_t)\log(1-\pi_t)\right]$$

where the indicator function is denoted by $\mathbb{I}(\cdot)$. The coding error/ cross-entropy error function term at time t is denoted as the last expression. The first order updates then update the logistic regression weights online:

$$\mathbf{w}_t = \mathbf{w}_{t-1} - \eta\nabla NLL\left(\mathbf{w}_{t-1}\right) = \mathbf{w}_{t-1} - \eta\left(\pi_t - y_t\right)\mathbf{x}_t$$

To ensure ongoing adaptation, the step size η is held constant.

PAQ9 updates upon PAQ8 by utilising an LZP preprocessor for faster processing of highly redundant files. PAQ9 codes literals as 9 bits and context length 12+ matches as 1 bit. The biggest difference from PAQ8 is that the context mixing architecture is a 2-input mixers chain instead of multiple input single mixer. The sparse order-1 contexts are mixed with gaps of 3 to 0, orders $2--6$, unigram, and then bigram respectively. Compared to PAQ8, PAQ9 trades compression performance for improved computation speed and reduced memory use (Mahoney, 2011).

ZPAQ model is the latest member of PAQ family which has a total of 255 components, about half of that of PAQ8. Like other PAQ models, each component predicts the output for the next bit. A 32-bit context as well as the prediction output of earlier components on the list are used as the next component's input. The final component then outputs a prediction to an arithmetic coder which encodes the next bit for the encoding process and decodes the next bit for the decoding process. ZPAQ's components are listed below (Mahoney, 2011):

1. Context Model CM: A user-specified size table maps the context to a prediction. The table entry contains a count where the prediction is adjusted so that the prediction error is proportionate to 1/count. This count ranges from 4 to 1020 and is incremented up to the limit specified by the user.

2. Constant `CONST`: A specified fixed and constant prediction.

3. Indirect context model `ICM`: A user-specified size hash table maps the context to an 8 bit state/ bit history. The `ICM` uses a high count, fixed limit to map history to prediction where the history contains the number of recent binary bits and the value of the latest bit (0 or 1).

4. `MATCH`: `MATCH` is composed of the user-specified size output buffer and pointer table. `MATCH` maps context to the pointer where the same context was most recently observed in the buffer. In proportion to the matching length, `MATCH` predicts the corresponding after the last observed match.

5. `AVG`: The user-specified weight mixing `AVG` combines the two predictions together. This mixer is in stretcher or logistic domain $\log(p/(1-p))$.

6. `MIX2`: Selected by a context, the user-specified size table of weights performs weighted averaging on the stretch predictions. Selected weight is updated after each prediction to improve input prediction, using prespecified adaptation rate.

7. `MIX`: `MIX` is similar to a `MIX2`. `MIX`, however, is performed over a user-specified earlier predictions array with a single weight per input per context.

8. Secondary symbol estimation `SSE`: From two adjacent 2-D table entries, a stretched input prediction together with a context interpolates a prediction output. As with a `CM`, `SSE` then updates the table to minimise the prediction error arising from the closer of the two entries. The probability dimension of the table is fixed at 64, but the table size, initial counts, and maximum counts are specified by the user, which leads to a different rate of adaptation.

9. Indirect secondary symbol estimation `ISSE`: Previous step sends `ISSE` a prediction and a context, mapped to a 8 bit state/ bit history like in `ICM`. The context of `MIX2` maps

to the bit history with input prediction and constant `CONST`. `ISSE` provides adjustment for the input prediction by utilising the current context's bit history.

6 Results

Except for available existing compressors, every model in this paper follows the arithmetic coding technique described in section 3. The main difference is that the predictor model varies, instantiating the different methods described in section 5. We will compare their compression effectiveness, performances over time during training, and finally their compression speeds.

6.1 Performance Comparison Against Benchmark

In this section, we compare different models, both existing compressors and our own experimental models. Our predictor models include a default frequency table (`FreqTable`), `PPM`, Decision Tree (`DCT`), Random Forest, `LSTM`, `LSTM`+1/2D-`CNN`, `LSTM`-stateful, `LSTM`-stateless, bi-directional `LSTM`, and `bi-LSTM`+Attention. The available existing compressors include `Gzip`, `Bzip2`, `7Zip`, `PAQ8`, `PAQ9`, `ZPAQ`, and `NNCP`. We also include `SBE`, which stands for *single block encoding*, a technique that turns each symbol sequence into strings of only position, such as `AATAT` to `A11010` and `T11`; this allows us to benchmark the compression ratio when there is no correlation between symbols. Each model may be accompanied by values of significant hyperparameters that significantly affect the performance of the models. Again, BpC represents Bits per Character, and by default we use 8-bit system, so uncompressed files have a BpC of 8.

In section 4.3.2, we saw that every character in DNA sequences of natural origin has roughly the same distribution. For both `E.Coli` and the artificial control sequence, the distribution of each symbol is as follows: `A`:24.6% `T`:24.6% `C`:25.4% `G`:25.4%. Because there are only 4 symbols, we can convert these symbols into 2-bit codewords: 00, 01, 10, and 11, which reduces the compressed size by a factor of 4, yielding a compression ratio of 2 BpC without any algorithm. As such, an intelligent algorithm should be able to compress better than 2 BpC using contextual information.

At the same time, an intelligent algorithm should not fail on the random control sequence.

Table 10: Compression size table of various algorithms for the artificial control sequence

Model	Specification	Size		BpC
		Original	Compressed	
Control	uncompressed	4638690	4638690	8.000
Gzip	9	4638690	1325921	2.287
Bzip2	9	4638690	1268262	2.187
7zip	tzip	4638690	1267529	2.186
PAQ8a	8	4638690	1161904	2.004
PAQ9a	9	4638690	1167729	2.014
ZPAQ	method 5	4638690	1163192	2.006
SBE	static	4638690	1165761	2.011
FreqTable	static	4638690	1165702	2.010
FreqTable	adaptive	4638690	1162577	2.005
PPM	3	4638690	1159829	2.000
LSTM	layer=2 hidden=32 batch=16 (20,20)	4638690	1161220	2.003

In Table 10, we first compare different compressors against the control sequence file.

Looking at BpC, we observe that standard compressors `Bzip2`, `Gzip`, and `7zip` are all unable to discern that there is no significance to the sequencing pattern (all significantly above 2 BpC). On the other hand, `PAQ` family, `PPM`, and `LSTM` all exhibit the ability to understand that only the distribution, not context, matters for optimal compression of this file.

For `LSTM`, `layer` is the total number of `LSTM` layers, `hidden` is the number of `LSTM` cells in each layer. `batch` is the batch size during training (the training size is always $100,000$). The final two numbers are `time steps` and `segment length`. For the above `LSTM`, the batch input dimension is $(16, 20, 20)$ for the first layer then $(16, 20, 32)$ for the following layers.

Because all non-standard algorithms pass the test on the artificial control sequence, we will now compare them on the `E.Coli` compression task. Below, we record significant results from our experiments in the Table 11 below, mostly with the best hyperparameters for each model. For the full result table, please refer to Appendix 7.2 where we compare a total of 63 models in Table 13 and Table 14. Note that the `LSTM` hyperparameters are specified as previously mentioned, except for bi-directional `LSTM`. In `bi-LSTM`, the number of `LSTM` cells is effectively doubled. For instance, for a `hidden`= 32 `batch`=16 $(20, 20)$ model, the first layer input dimension would be $(16, 20, 20)$, but the one of the next layer would be $(16, 20, 64)$. For attention-based models, `head` is the number of attention heads used.

Table 11 provides several insights into the DNA compression problem. First, the simplest/fastest model `FreqTable`'s compressed file (2.010 BpC) is only 6.25% larger than the best compressed file (1.892 BpC). However, standard algorithms `bzip2`, `gzip`, and `7zip`, show comparatively poor performances with over 2.1 BpC. For traditional machine learning algorithms, `PPM`, Naïve Bayes, and Random Forest perform better than the baseline of 2 BpC; only the decision tree model performs worse than the baseline. `PPM`, using a context of length 3, is the best of these at 1.964 BpC. While not shown, using context lengths that are not in multiples of 3 reduces the performance. The `PAQ` family, which is state-of-the-art on multiple benchmarks, has impressive performance with `PAQ8l` having as low as 1.900 BpC. `PAQ9` and `ZPAQ` trade some compression ratio for better speed.

Moving on to neural network models, we observe the significance of optimizing hyperparameters. The most basic 1-layer `LSTM` model already has a better performance of 1.956 BpC compared to the `PPM` model at 1.964 BpC. By adding a Convolutional Neural Network layer (`CNN`), we improve the performance to 1.947 for 1D and to 1.933 for 2D. The `NNCP` compressor is a stateful-`LSTM` model written in `C++` and optimised for speed and online updating with its own `LSTM` library. As such, it can train with a `batch` size as small as 8 without taking up over 24 hours. Optimising the timesteps and segment length to a multiple of 3 $(24, 24)$ and reducing `batch` to 8 gives us a state-of-the-art compression result at 1.885 BpC.

We also use the same specification as the model for `enwik9` compression, but since the model is too complex, our compression is sub-optimal at 1.948 BpC. We also implement our own stateful-`LSTM` model in `Python3`. We check that the result is comparable to `NNCP` by matching its parameters with `layer`=4, `hidden`=352, `batch`=16, and $(20, 20)$. While our compression takes significantly longer, we actually have a slightly better BpC at 1.893 compared to `NNCP`'s 1.895 under identical hyperparameters. Given timesteps 80 and segment length 6, our trade-off from stateful to stateless `LSTM` is significant from 1.920 to 1.903 BpC. However, this trade-off is justified as it allows us to use a bi-directional `LSTM` model, which

Table 11: Compression size table of various algorithms for `E.Coli` file

Model	Specification	Size		BpC
		Original	Compressed	
Ecoli	uncompressed	4638695	4638695	8.000
Gzip	9	4638695	1299059	2.240
Bzip2	9	4638695	1251004	2.158
7zip	tzip	4638695	1238459	2.136
PAQ8a	8	4638695	1102585	1.902
PAQ8l	8	4638695	1101724	1.900
PAQ9a	9	4638695	1115068	1.923
LPAQ9l	9	4638695	1109862	1.914
ZPAQ	method 5	4638695	1114292	1.922
SBE	static	4638695	1165778	2.011
FreqTable	static	4638695	1165644	2.010
FreqTable	adaptive	4638695	1162480	2.005
PPM	(3)	4638695	1138627	1.964
DCT	(128)	4638695	1169095	2.016
NaïveBayes	(64)	4638695	1145339	1.975
RandomForest	(128) estimators=20 + 10 per 100,000	4638695	1147840	1.980
LSTM	layer=1 hidden=256 batch=250 (20,20)	4638695	1133963	1.956
LSTM+1DCNN	layer=2 hidden=32 batch=250 (40,6)	4638695	1128797	1.947
LSTM+2DCNN	layer=2 hidden=32 batch=250 (40,6)	4638695	1120802	1.933
LSTM (NNCP-enwik9)	layer=7 hidden=512 batch=16 (20,20)	4638695	1129396	1.948
LSTM (NNCP)	layer=4 hidden=352 batch=8 (24,24)	4638695	1093223	1.885
LSTM (NNCP)	layer=4 hidden=352 batch=16 (20,20)	4638695	1098677	1.895
LSTM-Stateful	layer=4 hidden=352 batch=16 (20,20)	4638695	1097728	1.893
LSTM-Stateful	layer=4 hidden=32 batch=250 (80,6)	4638695	1103429	1.903
LSTM-Stateful	layer=4 hidden=32 batch=250 (40,6)	4638695	1112934	1.919
LSTM-Stateless	layer=4 hidden=32 batch=250 (80,6)	4638695	1113065	1.920
Bi-LSTM	layer=4 hidden=32 batch=250 (120,3)	4638695	1101875	1.900
Bi-LSTM+Attention	layer=2 hidden=32 head=3 (40,6)	4638695	1123789	1.938
TRFCP	layer=4 hidden=256 head=8 (32,32)	4638695	1109507	1.913

improves the performance to 1.900 and, if tuned properly, can likely bring the performance up further. Finally, we also try a simple attention model in combination with bi-LSTM, which gives a seemingly poor performance but our hyperparameters are also not optimal. For the pure attention-based transformer model, the performance at 1.913 BpC is worse than that of most of the LSTM models.

6.2 Categorical Accuracy Plot over Training Period

Next, we want to compare the performance over time during the compression of the E.Coli sequence. Again, there are only four symbols: A, T, C, and G. Therefore, the base categorical accuracy is 25%. Better categorical accuracy leads to better compression ratio since arithmetic coder codes a symbol with probability p in a number of bits arbitrarily close to $-\log p$. All algorithms we compare are LSTM-based and are trained every $100,000$ symbols on E.Coli sequence. Each 10% is the accuracy recorded at about every $460,000$ symbols or 5 training sets. For comparability, each LSTM has training batch equal to 250, timesteps equal to 40, segment length equal to 6, and 2–4 layers of LSTM. In total, we compare stateful LSTM 2–4 layers, 1, 2D-CNN+LSTM, 2-layer LSTM with attention, and bi-directional LSTM in Figure 16.

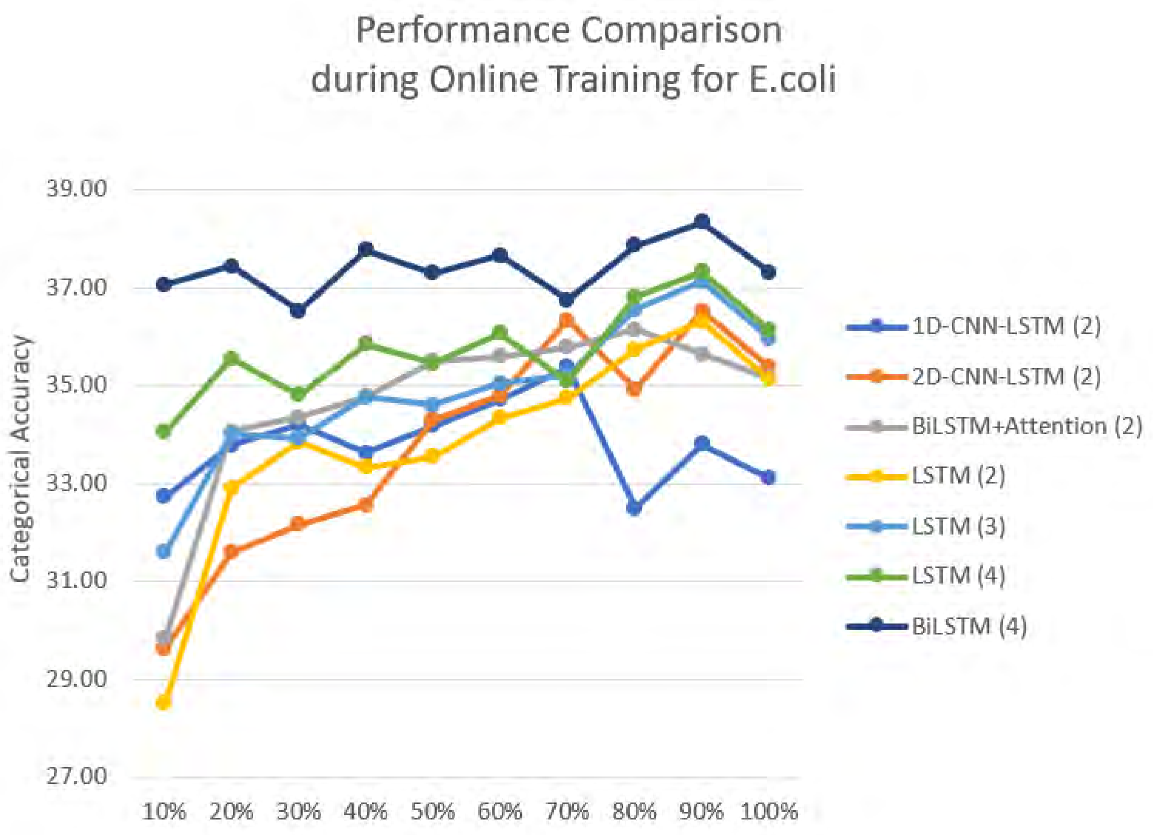


Figure 16: Categorical accuracy for various algorithms plotted over the training period during online training

From Figure 16, we can see that increasing the number of layers leads to better performance, especially during the first 10%–30%. In the first 10% alone, the performance increases from 28.51% to 31.60% to 34.05% when the number of layer increases from 2 to 3 to 4. For the whole file, this leads to a compression ratio increase from 1.928 to 1.925 to 1.919, respectively. The performance of 1, 2D- and BDA are about the same with worse performance than the 2-layer LSTM, since 1D-LSTM (1.947 BpC) falters off significantly at the end and 2D/CNN-LSTM (1.933 BpC) have poor performance in the beginning. Note that we only take the probability at every 10% point, so it might not be an accurate representation. This is why BDA seems to have better accuracy than the 2-layer stateful LSTM, but is outperformed compression-ratio wise. We can see that using bi-directional LSTM significantly improves performance consistence-wise and accuracy-wise. Although this leads to only a slightly better

BpC at 1.915, the categorical accuracy is higher than the 4-layer stateful-LSTM across the board.

6.3 Time vs. Performance on the DNA corpus

Finally, we want to see the trade-off between time and performance. We saw in section 6.1 and section 6.2 that LSTM-based models have the best compression ratio BpC-wise, especially when incorporating 3-grams and bi-directional traits. We want to provide some practical suggestions by illustrating the trade-off between compression ratio and time on various DNA sequences that are commonly stored. To do this, we use the 3-gram PPM, PAQ8l, and LSTM compression via C++ NNPC, to compress the 700 MB DNA corpus (Pratas and Pinho, 2018). While we would prefer to include our own bi-directional model, we accept that our implementation in Python is currently very inefficient and would take impractically long to decompress. We provide an extensive comparison in Table 12, as well as the overall result. We observe that gzip is outperformed even by the 2 BpC baseline where we simply replace A,T,C,G, with 00,01,10,11. However, this trade-off is justified as gzip can handle non A,T,C,G symbols. Overall, we see that there is a large jump in both compression rates and compression times from gzip to PPM and from PPM to PAQ8l. The jump from gzip to PPM trades 5.5 times increase in compression time for a 7.34% smaller compressed file (about 12.94 MB smaller than gzip). The jump from PPM to PAQ8l increases compression time by a factor of six for a 8.97% smaller compressed file (or about 14.65 MB). Using LSTM instead of PAQ8l, however, increases the compression time by over 10 times for only 0.28% better compression (0.43 MB). While the LSTM significantly outperforms PAQ8l in the large text compression benchmark, it only slightly outperforms PAQ8l because we compress each file separately, unlike enwik8 or enwik9 which are a single file. The compression speed is benchmarked on NVIDIA GeForce GTX 1660Ti, except for LSTM on large files which are run on NVIDIA Tesla K40c due to memory and other constraints. As such, the actual compression time of LSTM is closer to $15 - 20$ times that of PAQ8l. Assuming 1 terabyte of DNA files, we would trade off 1400 more hours to reduce the size from 238.12 GB to 216.75 GB. However, with multiple GPUs, e.g. 100 GPUs can reduce the time overhead to 14 hours, which might be worth the cost of storing an extra 12 GB in the cloud over multiple years. For memory, `PAQ8l` uses a total of 1.64 MB while `LSTM` uses 83.68 MB. The memory usage is constant throughout the process so care must be taken to ensure that the `GPUs` are capable of running the compressors.

Table 12: Compression ratio and speed benchmark on the DNA Corpus

Name	Size	Gzip		PPM		PAQ8l		LSTM	
		BpC	Time	BpC	Time	BpC	Time	BpC	Time
OrSa	43,262,523	2.094	51.56	1.942	309.45	1.708	2099.89	1.867	19,227.55
HoSa	189,752,667	2.014	246.75	1.881	1335.74	1.699	8187.01	1.704	95,834.34
GaGa	148,532,294	2.077	198.04	1.901	1081.25	1.848	5988.66	1.811	79,443.26
AnCa	142,189,675	2.065	187.09	1.923	991.64	1.655	6574.51	1.601	72,347.75
DaRe	62,565,020	2.009	83.42	1.891	510.72	1.603	2742.34	1.637	25,248.23
DrMe	32,181,429	2.108	42.91	1.937	222.67	1.879	1356.68	1.883	16,674.09
EnIn	26,403,087	2.113	32.24	1.912	203.30	1.733	1138.71	1.761	11,479.56
ScPo	10,652,155	2.137	12.53	1.956	84.02	1.915	801.79	1.902	5,230.04
WaMe	9,144,432	2.141	10.78	1.967	74.21	1.947	579.53	1.930	3,874.75
PlFa	8,986,712	2.000	11.90	1.852	71.68	1.737	592.54	1.714	4352.55
EsCo	4,641,652	2.131	5.59	1.951	31.45	1.899	197.62	1.886	1933.75
HePy	1,667,825	2.081	2.27	1.877	13.05	1.835	64.19	1.807	757.72
AeCa	1,591,049	2.127	1.93	1.934	10.52	1.903	74.99	1.863	723.18
HaHi	3,890,005	2.086	5.07	1.897	30.57	1.851	150.52	1.798	1662.39
YeMi	73,689	2.072	0.10	1.867	0.71	1.822	5.08	2.031	30.25
BuEb	18,940	2.260	0.02	2.056	0.15	1.985	1.14	2.533	10.10
AgPh	43,970	2.187	0.05	1.992	0.29	1.951	2.25	2.145	19.55
Overall	685,597,124	2.056	892.25	1.905	4,971.42	1.734	30,557.45	1.729	338,849.06

7 Conclusion and Future Work

7.1 Conclusion

This dissertation provides a comprehensive analysis and comparison of multiple lossless DNA compression algorithms. We also provides a concise mathematical formalisation for each compression scheme, giving some insight into its characteristics and inner workings. We conducted a basic analysis of some characteristics of DNA sequences, to understand how different algorithms leverage different DNA traits. This dissertation also contributes a survey on the current DNA compression literature.

Our exploratory analysis of DNA data features confirmed multiple hypotheses. For example, we established that DNA sequences exhibit a "codon-dictionaries nature", where 3-letter triplets in the sequence will form a semantic unit (an amino-acid codon), with a skewed distribution towards certain codons. This dissertation also ascertains that nucleotide sequences have an abnormally high number of length-$3n$ repeats, as well as a higher number of single symbol repeats than a random sequence with the same symbol distribution. Our extensive hyperparameter-tuning experiments illustrate that, for each compression method, using $3n$-gram contexts leads to better compression ratios when compressing DNA sequences. The data analysis also shows that DNA sequences exhibit palindromic and complementary palindromic traits, although not to the same extent as $3n$-gram characteristics. To leverage this particular characteristic, we propose a `bi-directional` context concept for DNA compression. Combining this concept with an `LSTM` results in our novel bi-directional `LSTM` or bi-`LSTM`. Our dissertation extensively tests bi-`LSTM` and finds its performance to be superior compared to a normal `LSTM` (when hyperparameters are otherwise kept similar). Porting the bi-`LSTM` code from `Python` to `C++`, and adjusting the code to be in line with machine language would likely provide state-of-the-art compression for DNA (in terms of compression ratio, though at the cost of a long training time).

Our goal was to explore and expand the field of permissive and general DNA compression algorithms, rather than building DNA-only compressors that fail on other data. As such, we deprioritised from reference-based sequence matching algorithms and pretrained neural network models. Our dissertation also inspects online learning-capable traditional machine learning algorithms and attention-based neural network models, which are unconventional created for compression. To best compare these algorithms, we created an overview comparison on the `E.Coli` DNA sequence, and an extensive comparison on the multi-species large DNA corpus.

Our overview study in Table 11 explores the limit of currently achievable compression ratios, where an `LSTM` model emerges as a clear winner when we reduce batch-size. However, reducing batch-size significantly increases computation time, which prevents us from testing our bi-`LSTM` model with the same hyperparameters, due to the slow speed of our `Python` implementation. Nevertheless, bi-`LSTM` is shown to have a better compression ratio than `LSTM` for the same hyperparameters, especially in our performance comparison during online training in Figure 16.

For a more holistic comparison, we investigate and compare the computation speed, compression ratio, and memory usage of DNA compression algorithms. Our benchmark on an artificially generated DNA file in Table 10 shows that standard compressors including `bzip2`, `gzip`, and `7zip` are not practically useful for DNA compression. Table 12 tests increasingly complex compressors on a myriad of DNA from those of virus to human and other mammals. This table provides a useful trade-off in speed and compression ratio, proving that no single compressor is state-of-the-art in every aspect. We can observe that `PAQ8l` provides near-optimal performance (1.734 BpC) for a relatively low-speed trade-off, but `PPM` still provides the best speed to compression ratio for a useful compressor (BpC$\leq$ 2). `PAQ8l` tends to perform better than `LSTM` in highly repetitive DNA sequences, such as human DNA (Homo Sapien or HoSa). Given this comparison table, specialised researchers can make an informed choice when picking appropriate compression methods for DNA sequences of specific sizes or of selected species.

7.2 Future Work

In this dissertation, we introduced the concept of bi-directional contexts to DNA compression, allowing compression algorithms to utilise the palindromic and complementary-palindromic nature of DNA sequences. We show that, given the same hyperparameters, the bi-`LSTM` will have better compression ratio than the standard stateful `LSTM`. Our compression algorithm is written in `Python`, which is far removed from optimised machine code. This leads to a great reduction in computation speed during the training process, about 5–10 times that of an optimised program binary. Possible future work can be done on implementing and optimising the bi-`LSTM` compression scheme in `C++`, which is closer tomachine language. The `C++` bi-`LSTM` should probably be tested on `enwik8` and `enwik9` to compete in Matt Mahoney's large text compression benchmark (Mahoney, 2011). Natural language might also exhibit some bi-directional traits, so it could be interesting to test if the bi-`LSTM` model could outperform the traditional stateful `LSTM` model, which is currently in 3rd place on the benchmark. Additional work could be done on optimising training process parallelisation, to improve the algorithm's speed when using multiple `GPUs`. Better parallelisation can significantly reduce compression time, which would help increase the practicality of the bi-`LSTM` compression scheme.

References

Current GenBank release notes. `https://www.ncbi.nlm.nih.gov/genbank/release/current/`. Accessed: 2020-06-29.

Nour S Bakr, Amr A Sharawi, et al. DNA lossless compression algorithms. *American Journal of Bioinformatics Research*, 3(3):72–81, 2013.

Timothy Bell, Ian H Witten, and John G Cleary. Modeling for text compression. *ACM Computing Surveys (CSUR)*, 21(4):557–591, 1989.

Fabrice Bellard. Lossless data compression with neural networks, 2019.

Gaëtan Benoit, Claire Lemaitre, Dominique Lavenier, Erwan Drezen, Thibault Dayris, Raluca Uricaru, and Guillaume Rizk. Reference-free compression of high throughput sequencing data with a probabilistic de Bruijn graph. *BMC bioinformatics*, 16(1):288, 2015.

James K Bonfield and Matthew V Mahoney. Compression of FASTQ and SAM format sequencing data. *PloS one*, 8(3):e59190, 2013.

Ying Chen, Weicai Ye, Yongdong Zhang, and Yuesheng Xu. High speed BLASTN: an accelerated MegaBLAST search tool. *Nucleic acids research*, 43(16):7762–7768, 2015.

Scott Christley, Yiming Lu, Chen Li, and Xiaohui Xie. Human genomes as email attachments. *Bioinformatics*, 25(2):274–275, 2009.

John Cleary and Ian Witten. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications*, 32(4):396–402, 1984.

Peter JA Cock, Christopher J Fields, Naohisa Goto, Michael L Heuer, and Peter M Rice. The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants. *Nucleic acids research*, 38(6):1767–1771, 2010.

Markus Hsi-Yang Fritz, Rasko Leinonen, Guy Cochrane, and Ewan Birney. Efficient storage of high throughput DNA sequencing data using reference-based compression. *Genome research*, 21(5):734–740, 2011.

Maj Gen and HI Suhaib Ahmed. Forensic DNA testing. Genetics Resource Centre. http://grcpk.com/wp-content/uploads/2014/10/16.-Forensic-DNA-Testing.pdf, October 2014. Accessed: 2020-05-25.

Faraz Hach, Ibrahim Numanagić, Can Alkan, and S Cenk Sahinalp. SCALCE: boosting sequence compression algorithms using locally consistent encoding. *Bioinformatics*, 28 (23):3051–3057, 10 2012. ISSN 1367-4803. doi: 10.1093/bioinformatics/bts593. URL https://doi.org/10.1093/bioinformatics/bts593.

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9 (8):1735–1780, 1997.

Richard CG Holland and Nick Lynch. Sequence squeeze: an open contest for sequence compression. *GigaScience*, 2(1):2047–217X, 2013.

Morteza Hosseini, Diogo Pratas, and Armando J Pinho. A survey on data compression methods for biological sequences. *Information*, 7(4):56, 2016.

Marcus Hutter. Human Knowledge Compression Contest. A data compression contest measured on a 100 MB extract of Wikipedia, 2006. URL http://prize.hutter1.net/.

Daniel C Jones, Walter L Ruzzo, Xinxia Peng, and Michael G Katze. Compression of next-generation sequencing reads aided by highly efficient de novo assembly. *Nucleic acids research*, 40(22):e171–e171, 2012.

Byron Knoll and Nando de Freitas. A machine learning perspective on predictive coding with PAQ8. In James A. Storer and Michael W. Marcellin, editors, *Proceedings of the Data Compression Conference*, pages 377–386. IEEE Computer Society, 2012. ISBN 978-1-4673-0715-4. doi: 10.1109/DCC.2012.44.

SR Kodituwakku and US Amarasinghe. Comparison of lossless data compression algorithms for text data. *Indian journal of computer science and engineering*, 1(4):416–425, 2010.

Shanika Kuruppu, Simon J Puglisi, and Justin Zobel. Reference sequence construction for relative compression of genomes. In *International Symposium on String Processing and Information Retrieval*, pages 420–425. Springer, 2011.

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

David JC MacKay. *Information theory, inference and learning algorithms*. Cambridge university press, 2003.

Matt Mahoney. Large text compression benchmark, 2011. URL http://mattmahoney.net/dc/text.html#:~:text=This%20competition%20ranks%20lossless%20data,About%20the%20test%20data.

Matthew Vincent Mahoney. Adaptive weighing of context models for lossless data compression. Technical Report CS-2005-16, Department of Computer Science, Florida Institute of Technology, Melbourne, FL, USA., 2005.

Giovanni Manzini and Marcella Rastero. A simple and fast DNA compressor. *Software: Practice and Experience*, 34(14):1397–1411, 2004.

Tomáš Mikolov, Stefan Kombrink, Lukáš Burget, Jan Černockỳ, and Sanjeev Khudanpur. Extensions of recurrent neural network language model. In *2011 IEEE international conference on acoustics, speech and signal processing (ICASSP)*, pages 5528–5531. IEEE, 2011.

Alistair Moffat, Radford M. Neal, and Ian H. Witten. Arithmetic coding revisited. *ACM Transactions on Information Systems*, 16(3):256–294, July 1998. ISSN 1046-8188.

Mahesh Pal. Random forest classifier for remote sensing classification. *International journal of remote sensing*, 26(1):217–222, 2005.

William R Pearson and David J Lipman. Improved tools for biological sequence comparison. *Proceedings of the National Academy of Sciences*, 85(8):2444–2448, 1988.

Diogo Pratas and Armando J Pinho. A DNA sequence corpus for compression benchmark. In *International Conference on Practical Applications of Computational Biology & Bioinformatics*, pages 208–215. Springer, 2018.

Irina Rish et al. An empirical study of the Naive Bayes classifier. In *IJCAI 2001 workshop on empirical methods in artificial intelligence*, volume 3, pages 41–46, 2001.

Jorma J. Rissanen. Generalized Kraft inequality and arithmetic coding. *IBM Journal of Research and Development*, 20(3):198–203, 1976. ISSN 0018-8646. doi: 10.1147/rd.203.0198.

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.

Tara N Sainath, Oriol Vinyals, Andrew Senior, and Haşim Sak. Convolutional, long short-term memory, fully connected deep neural networks. In *2015 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4580–4584. IEEE, 2015.

Christian Steinruecken. Lossless data compression: Introduction. Chapter 1 in *Lossless Data Compression*, Ph.D. dissertation, University of Cambridge, 2014a.

Christian Steinruecken. Arithmetic coding. Chapter 3 in *Lossless Data Compression*, Ph.D. dissertation, University of Cambridge, 2014b.

Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. LSTM neural networks for language modeling. In *Thirteenth annual conference of the international speech communication association*, 2012.

Rongjie Wang, Tianyi Zang, and Yadong Wang. Human mitochondrial genome compression using machine learning techniques. *Human Genomics*, 13(1):1–8, 2019.

Ian Hugh Witten, Radford Neal, and John Gerald Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, June 1987. ISSN 0001-0782.

Appendix

Table 13: Full table of various compression algorithms for E.Coli file (Part 1)

Model	Specification	Size		BpC
		Original	Compressed	
Ecoli	uncompressed	4638695	4638695	8.000
Gzip	9	4638695	1299059	2.240
Bzip2	9	4638695	1251004	2.158
7zip	tzip	4638695	1238459	2.136
PAQ8a	8	4638695	1102585	1.902
PAQ8l	8	4638695	1101724	1.900
PAQ9a	9	4638695	1115068	1.923
LPAQ9l	9	4638695	1109862	1.914
ZPAQ		4638695	1637985	2.825
ZPAQ	method 5	4638695	1114292	1.922
SBE	static	4638695	1165778	2.011
FreqTable	static	4638695	1165644	2.010
FreqTable	adaptive	4638695	1162480	2.005
PPM	(3)	4638695	1138627	1.964
DCT	(128)	4638695	1169095	2.016
NaïveBayes		4638695	1145339	1.975
NaïveBayes	(6)	4638695	1151124	1.985
NaïveBayes	(6) 256	4638695	1150929	1.985
NaïveBayes	(64)	4638695	1217974	2.101
NaïveBayes	(64) 1000	4638695	1327415	2.289
RandomForest	(6) estimators=50	4638695	1211435	2.089
RandomForest	(64) estimators=50	4638695	1221173	2.106
RandomForest	(64) estimators=20 + 10 per 100,000	4638695	1176852	2.030
RandomForest	(128) estimators=10 + 10 per 100,000	4638695	1152274	1.987
RandomForest	(128) estimators=20 + 10 per 100,000	4638695	1147840	1.980
LSTM	layer=1 hidden=64 batch=250 (20,20)	4638695	1179672	2.034
LSTM	layer=1 hidden=160 batch=250 (20,20)	4638695	1173204	2.023
LSTM	layer=1 hidden=256 batch=250 (20,20)	4638695	1133963	1.956
LSTM	layer=1 hidden=512 batch=250 (20,20)	4638695	1138251	1.963
LSTM+1DCNN	layer=2 hidden=32 batch=250 (40,6)	4638695	1128797	1.947
LSTM+1DCNN	layer=2 hidden=64 batch=250 (40,6)	4638695	1128797	1.950
LSTM+2DCNN	layer=2 hidden=32 batch=250 (40,6)	4638695	1120802	1.933

Table 14: Full table of various compression algorithms for `E.Coli` file (Part 2)

Model	Specification	Size		BpC
		Original	Compressed	
LSTM (NNCP+enwik9)	layer=7 hidden=512 batch=16 (20,20)	4638695	1129396	1.948
LSTM (NNCP)	layer=2 hidden=32 batch=16 (20,20)	4638695	1118429	1.929
LSTM (NNCP)	layer=3 hidden=32 batch=64 (20,20)	4638695	1140565	1.967
LSTM (NNCP)	layer=2 hidden=32 batch=16 (20,20)	4638695	1111045	1.916
LSTM (NNCP)	layer=4 hidden=90 batch=250 (60,6)	4638695	1194606	2.060
LSTM (NNCP)	layer=4 hidden=90 batch=250 (60,6)	4638695	1203370	2.075
LSTM (NNCP)	layer=4 hidden=352 batch=16 (20,20)	4638695	1098677	1.895
LSTM (NNCP)	layer=4 hidden=352 batch=16 (60,6)	4638695	1108255	1.911
LSTM (NNCP)	layer=4 hidden=352 batch=16 (120,3)	4638695	1109507	1.924
LSTM (NNCP)	layer=4 hidden=352 batch=16 (20,20)	4638695	1098677	1.895
LSTM (NNCP)	layer=4 hidden=352 batch=8 (20,20)	4638695	1093594	1.886
LSTM (NNCP)	layer=4 hidden=352 batch=8 (24,24)	4638695	1093223	1.885
LSTM (NNCP)	layer=4 hidden=352 batch=16 (24,24)	4638695	1099237	1.895
LSTM (NNCP)	layer=4 hidden=352 batch=16 (30,30)	4638695	1099517	1.896
LSTM (NNCP)	layer=5 hidden=352 batch=16 (30,30)	4638695	1099996	1.897
LSTM-Stateful	layer=4 hidden=352 batch=16 (20,20)	4638695	1097728	1.893
LSTM-Stateful	layer=4 hidden=32 batch=250 (80,6)	4638695	1103429	1.903
LSTM-Stateful	layer=2 hidden=32 batch=250 (40,6)	4638695	1117952	1.928
LSTM-Stateful	layer=3 hidden=32 batch=250 (40,6)	4638695	1115932	1.925
LSTM-Stateful	layer=4 hidden=32 batch=250 (40,6)	4638695	1112934	1.919
LSTM-Stateless	layer=4 hidden=32 batch=250 (40,6)	4638695	1119377	1.931
LSTM-Stateless	layer=4 hidden=32 batch=250 (60,12)	4638695	1118641	1.929
LSTM-Stateless	layer=4 hidden=32 batch=250 (60,6)	4638695	1119489	1.931
LSTM-Stateless	layer=4 hidden=32 batch=250 (60,6)	4638695	1113652	1.921
LSTM-Stateless	layer=5 hidden=32 batch=250 (60,6)	4638695	1116077	1.925
LSTM-Stateless	layer=4 hidden=32 batch=250 (80,6)	4638695	1113065	1.920
LSTM-Stateless	layer=5 hidden=32 batch=250 (60,6)	4638695	1118316	1.929
Bi-LSTM	layer=4 hidden=32 batch=250 (100,6)	4638695	1108978	1.913
Bi-LSTM	layer=4 hidden=32 batch=250 (120,3)	4638695	1106151	1.908
Bi-LSTM	layer=4 hidden=32 batch=250 (120,3)	4638695	1101875	1.900
Bi-LSTM	layer=4 hidden=32 batch=250 (80,6)	4638695	1109196	1.913
Bi-LSTM	layer=4 hidden=32 batch=250 (40,6)	4638695	1110173	1.915
Bi-LSTM	layer=4 hidden=352 batch=250 (80,6)	4638695	1114590	1.922
Bi-LSTM	layer=6 hidden=32 batch=250 (120,6)	4638695	1097047	1.892
Bi-LSTM+Attention	layer=2 hidden=32 head=3 (40,6)	4638695	1123789	1.938
TRFCP	layer=4 hidden=256 head=8 (32,32)	4638695	1109507	1.913